

Distributed Training of Deep Neural Networks

António Joaquim André Esteves

estesves@di.uminho.pt



Departamento de Informática e Centro Algoritmi

Escola de Engenharia

Universidade do Minho

Braga, July 2026

Contents

Contents	iii
1 Introduction	6
2 Distributed Training	8
2.1 Distributed systems	8
2.2 Basic concepts in distributed training	10
2.3 Communication backends	11
2.4 Collective communication operations	12
2.5 Point-to-point communication and synchronization operations	20
2.6 Types of parallelism in distributed training	20
2.7 How to select the parallelism type	21
2.8 Synchronous and asynchronous parameter updates	21
2.8.1 Synchronous parameter updates	21
2.8.2 Advantages of synchronous parameter updates	22
2.8.3 Disadvantages of synchronous parameter updates	22
2.8.4 Asynchronous parameter updates	23
2.8.5 Advantages of asynchronous parameter updates	23
2.8.6 Disadvantages of asynchronous parameter updates	23
2.8.7 The staleness problem in asynchronous updates	24
2.8.8 Choosing between synchronous and asynchronous updates	24
2.8.9 Practical implications	24
2.9 Strong and weak scaling	25
2.10 Hardware considerations	26
2.11 Conclusions	28
3 Data Parallelism	29
3.1 Distributed data parallelism	30
3.1.1 Distributed data parallel in PyTorch	36
3.1.2 Practice with distributed data parallel training	38
3.2 Parameter server	42
3.2.1 Running stochastic gradient descent with AllReduce	43
3.2.2 The parameter server architecture	44
3.2.3 Downpour stochastic gradient descent	45
3.2.4 Sandblaster L-BFGS	46
3.2.5 Implementing a parameter server using the PyTorch distributed RPC framework	47
3.3 AllReduce distributed training	56
3.3.1 Latency of ring AllReduce	59
3.4 Tree AllReduce	59
3.5 Two-tree AllReduce	60
3.6 Performance impact	62
3.7 Zero redundancy optimizer parallelism	64
3.7.1 Memory usage	64
3.7.2 ZeRO-1: partitioning the optimizer state	65

3.7.3	ZeRO-2: adding gradient partitioning	67
3.7.4	ZeRO-3: adding weight partitioning	68
3.8	Fully sharded data parallel	70
3.8.1	Practice with FSDP training in PyTorch	73
3.9	Conclusions	82
4	Model Parallelism	83
4.1	Pipeline parallelism	85
4.1.1	GPipe	85
4.1.2	PipeDream	89
4.1.3	Combining data and pipeline parallelisms	93
4.1.4	Pipeline parallelism in available frameworks	94
4.2	Tensor parallelism	104
4.2.1	Practice with tensor parallelism in PyTorch	106
4.3	Combining multiple types of parallelism	117
4.3.1	Data and pipeline parallelisms	117
4.3.2	Data, pipeline and model parallelisms	117
4.3.3	Data, pipeline and tensor parallelisms, and ZeRO optimization	119
4.3.4	Megatron-LM	119
4.3.5	FlexFlow	119
4.4	Conclusions	121
5	Mixture-of-Experts	123
5.1	Introduction to mixture of experts	124
5.1.1	Dense mixture of experts	125
5.1.2	Sparse mixture of experts	125
5.2	Taxonomy of mixture of experts	127
5.3	Algorithm design of mixture of experts	127
5.3.1	Gating function	127
5.3.2	Experts	133
5.3.3	Mixture of weight-efficient experts	137
5.3.4	Training and inference scheme for MoEs	139
5.3.5	Mixture of experts derivatives	141
5.4	Design of mixture of experts	142
5.4.1	Computation	142
5.4.2	Communication	143
5.4.3	Storage	143
5.5	Routing techniques	144
5.5.1	Top-k gating	144
5.5.2	Noisy top-k gating	146
5.5.3	Hash-based gating	147
5.5.4	Switch transformer gating	149
5.5.5	Soft MoE	150
5.6	Load balancing	152
5.7	Training and fine-tuning MoEs	155
5.8	Performance evaluation	160
5.8.1	Evaluation metrics and benchmarks	160
5.8.2	Performance challenges	161
5.9	Deployment of MoEs	162
5.9.1	Model capacity	162
5.9.2	Caching	164
5.9.3	Block sparsity	164
5.9.4	Mitigating inefficiencies of inference	165
5.10	Memory optimization	170

5.10.1	Redundancy optimization	171
5.10.2	Expert weight offloading	171
5.10.3	Activation recomputation	173
5.10.4	Low-precision training	174
5.11	Communication optimization	174
5.11.1	Collective communication operations redesign and scheduling	174
5.11.2	Concurrent execution	175
5.11.3	Computation-communication overlap	176
5.11.4	Concurrent collective communication	177
5.11.5	Communication compression	179
5.12	Open source MoEs	180
5.12.1	MegaBlocks	180
5.12.2	Mixtral of experts	184
5.12.3	Grok	186
5.12.4	DBRX	189
5.12.5	OpenMoE	192
5.12.6	DeepSeek-v2 and DeepSeek-v3	194
5.13	Other relevant MoEs	200
5.14	Conclusions	204
6	Implement Mixture-of-Experts in PyTorch	206
6.1	Decoder-only transformer	207
6.1.1	From text to tokens	207
6.1.2	Masked multi-headed self-attention	210
6.1.3	Feed-forward network	218
6.1.4	Decoder-only transformer block	218
6.1.5	Decoder-only transformer architecture	220
6.2	Build a mixture-of-experts model	223
6.2.1	Expert layers	224
6.2.2	Routing tokens to experts	225
6.2.3	Expert capacity	227
6.2.4	Load balancing loss and router z-loss	230
6.2.5	Decoder-only MoE implementation	233
6.3	Pretraining the MoE model	236
6.3.1	Best practices for training MoEs	236
6.3.2	Pretraining experiments	238
6.4	Conclusions	241
7	Other Types of Parallelism	242
7.1	Parallelism in mixture of experts	243
7.1.1	Fundamental parallelism	244
7.1.2	Hybrid parallelism	244
7.2	Expert parallelism	246
7.2.1	Definition and taxonomy	246
7.2.2	Advantages of expert parallelism	248
7.2.3	Challenges of expert parallelism	248
7.3	Parallelism in switch transformers	248
7.4	Parallelism in Tutel	249
7.4.1	Static pipelining	250
7.4.2	Adaptive parallelism switching	252
7.4.3	Adaptive pipelining for all-to-all communication	253
7.4.4	Dictionary of optimal parallelism and pipelining	255
7.5	DeepSpeed-MoE	255
7.5.1	Pyramid residual MoE architecture	255

7.5.2	DeepSpeed-MoE inference	257
7.5.3	Design of DeepSpeed-MoE inference system	257
7.5.4	Combining tensor-slicing, expert-slicing, data parallelism, and expert parallelism	257
7.5.5	Optimized communication subsystem	259
7.5.6	Optimized transformer and MoE kernels	260
7.6	Pathways	261
7.7	Parallelism in Megatron-Core	263
7.7.1	Expert parallelism	263
7.7.2	The challenges of combining expert parallelism with other types of parallelism	264
7.7.3	Parallel folding	265
7.7.4	Benefits of parallel folding	266
7.8	Expert parallelism in NVIDIA Megatron Bridge	267
7.9	Elastic expert parallelism in vLLM	270
7.9.1	Expert parallelism and data parallelism attention	272
7.9.2	What needs to change when the available computing resources change?	272
7.9.3	Scale-up flow	272
7.9.4	Scale-down flow	274
7.9.5	Coordinating reconfiguration steps across DP ranks	274
7.9.6	Fault tolerance	274
7.9.7	An example	274
7.10	Sequence parallelism	275
7.10.1	Megatron sequence parallelism	275
7.10.2	DeepSpeed-Ulysses	278
7.10.3	Ring attention	281
7.10.4	Striped attention	283
7.11	Implementation example	286
7.12	Conclusions	295

8 Conclusions 297

A Train a Simple Transformer Model with Fully Sharded Data Parallel 314

List of Figures

2.1	Evolution of size of state-of-the-art NLP models over time.	9
2.2	Example that illustrates the distributed computing terminology.	10
2.3	In a Broadcast operation, all ranks receive data from the root rank.	12
2.4	In the AllReduce operation, each rank receives the reduction of input values across ranks.	14
2.5	In the Reduce operation, one rank receives the reduction of input values across ranks.	14
2.6	In the AllGather operation, each rank receives the aggregation of data from all ranks.	15
2.7	In the Gather operation, the root rank receives data from all ranks.	16
2.8	In the scatter operation, the root rank distributes data to all ranks.	17
2.9	In the ReduceScatter operation, input values are reduced across ranks.	18
2.10	The AlltoAll operation exchanges data between all ranks.	19
2.11	Data, pipeline, and tensor parallelism.	21
2.12	Synchronous SGD workflow.	22
2.13	Asynchronous SGD workflow.	23
2.14	Real and ideal speedups of strong and weak scalings.	25
2.15	The fundamental questions answered by strong and weak scaling.	26
2.16	A generic architecture of a distributed computing system.	27
3.1	Data parallel distributed training with stochastic gradient descent.	30
3.2	The building blocks of forward and backpropagation.	31
3.3	The building block that represents the optimizer.	32
3.4	Example of a pebble graph for a model with four layers.	32
3.5	Forward and backpropagation steps of sequential training.	32
3.6	Data parallel training with two nodes.	33
3.7	Ring AllReduce animation that explains how the reduction works.	34
3.8	Apply AllReduce to combine gradients from all nodes.	35
3.9	Parallel training without overlap between computation and communication.	35
3.10	Parallel training with overlap between computation and communication.	36
3.11	Algorithm for PyTorch distributed data parallel training.	38
3.12	Algorithm for distributed stochastic gradient descent with AllReduce	44
3.13	A centralized parameter server architecture.	44
3.14	Algorithm for asynchronous distributed SGD with parameter server.	45
3.15	Downpour stochastic gradient descent and Sandblaster L-BFGS.	46
3.16	AllReduce operation.	56
3.17	An example of distributed data parallel training.	57
3.18	Reduction performed by a single master node	58
3.19	Visualization of the ScatterReduce phase with four worker nodes.	59
3.20	Visualization of the AllGather phase with four worker nodes.	60
3.21	The tree AllReduce algorithm visualization.	61
3.22	The two-tree AllReduce algorithm visualization.	62
3.23	NCCL latency for different AllReduce algorithms.	63
3.24	NCCL bandwidth for different AllReduce algorithms.	63
3.25	Model performance training comparison on ResNet50.	64
3.26	Memory usage for the three ZeRO stages.	65
3.27	All the steps of forward and backward passes in ZeRO-1.	66
3.28	The communication pattern in ZeRO-1.	66

3.29	All the steps of forward and backward passes in ZeRO-2.	67
3.30	The communication pattern in ZeRO-2.	67
3.31	The forward pass with ZeRO-3.	68
3.32	The backward pass with ZeRO-3.	68
3.33	The communication pattern with ZeRO-3.	69
3.34	Memory usage on an 8B model.	69
3.35	Traditional distributed data parallel training with two processes.	70
3.36	Fully-sharded data parallel training with two processes.	71
3.37	AllReduce DDP operation decomposed into FSDP ReduceScatter and AllGather operations. . .	71
3.38	Using flattened tensor weights in FSDP1.	72
3.39	Using per-parameter tensors in FSDP2.	73
3.40	Transformer architecture after being wrapped by FSDP2.	75
3.41	Implicit prefetching of AllGather operations in FSDP2.	76
4.1	A neural network with layers partitioned across four devices.	84
4.2	A neural network with four layers partitioned across two devices.	84
4.3	Low device utilization in model parallelism.	85
4.4	Naive pipeline parallelism to split a model into four GPUs.	86
4.5	GPipe pipeline parallelism considering four microbatches.	86
4.6	Improving the communication and computation overlap.	87
4.7	Pipeline bubbles in GPipe.	87
4.8	Influence of the number of microbatches over the bubble fraction.	88
4.9	GPipe gradient checkpointing.	89
4.10	PipeDream schedule.	89
4.11	Pipedream steady state	90
4.12	Comparing PipeDream and GPipe schedules.	90
4.13	Results of PipeDream.	91
4.14	Pipeline scheduling in PipeDream-flush.	92
4.15	Pipeline scheduling for PipeDream-2BW.	92
4.16	Data parallelism and pipeline parallelism.	93
4.17	Blocks of a transformer with tensor parallelism.	105
4.18	Communication in a tensor parallel transformer layer.	106
4.19	Comparing tensor and pipeline parallelisms.	106
4.20	Row-wise tensor parallelism applied to the first layer.	107
4.21	Row-wise tensor parallelism applied to the second layer.	108
4.22	Column-wise tensor parallelism applied to the first layer.	109
4.23	Column-wise tensor parallelism applied to the second layer.	110
4.24	Relationship diagram of the NVIDIA communication concepts.	110
4.25	Communication setup for distributed training with two GPUs.	111
4.26	Execution of the the code included in Listing 4.17.	113
4.27	3D parallelism in DeepSpeed.	117
4.28	Hybrid two-way data parallelism and two-stage pipeline parallelism.	118
4.29	Mapping workers to GPUs on a system with eight nodes, each with four GPUs.	118
4.30	1F1B pipeline schedule and the interleaved 1F1B pipeline schedule.	119
4.31	Four parallelism configurations for a convolution.	120
4.32	Parallelism configuration for a matrix multiplication.	120
4.33	FlexFlow overview.	121
5.1	A chronological overview of mixture-of-experts models.	125
5.2	An MoE layer in transformer-based models.	126
5.3	Taxonomy of Mixture of Experts.	128
5.4	Various gating functions employed in MoE models.	129
5.5	Imbalanced vs. balanced expert utilization.	131
5.6	Token routing dynamics in switch transformer.	133

5.7	Mixture of attention heads.	134
5.8	The taxonomy of MoPEs based placement.	137
5.9	Representation of training and inference in MoE.	140
5.10	Diverse parallel strategies for MoE models.	142
5.11	The gating process for a single token.	145
5.12	Compare the perplexity of dense, learned MoE, and hash-based MoE models.	148
5.13	Comparison of routing strategies for two tokens.	149
5.14	Comparison of models in terms of size and required computation.	151
5.15	Compare hard routing to Soft MoE.	152
5.16	Adaptive expert placement.	154
5.17	Training time and performance of Switch Transformer and T5.	156
5.18	Training time and performance of Switch Transformer and two versions of T5.	156
5.19	Test loss vs. sparse model weights.	157
5.20	Validation loss for different routing methods.	157
5.21	Numerical precision formats and roundoff errors.	158
5.22	SuperGLUE for diverse minibatch sizes and for different learning rates.	159
5.23	Transformers' precision and accuracy as a function of pretraining time.	159
5.24	System challenges in distributed MoE training.	160
5.25	Visualization of the expert activation pattern on selected layers.	163
5.26	Tiled matrix multiplication.	164
5.27	Expert computation in an MoE layer.	165
5.28	Block-sparse matrix multiplication compared to cuBLAS batched matrix multiplication.	165
5.29	Comparing the inference latency of MoE and dense models running on a single node.	166
5.30	Comparing the memory footprint of MoE and dense models during inference.	166
5.31	MoE model latency composition.	167
5.32	Average number of inactive experts.	167
5.33	Comparison between static and dynamic gating.	168
5.34	Illustration of the expert buffering mechanism.	169
5.35	Tradeoff between memory usage and cache latency.	169
5.36	The effect of load balancing.	170
5.37	Memory optimization techniques taxonomy.	171
5.38	Expert weight offloading strategies.	172
5.39	Communication optimization techniques taxonomy.	175
5.40	A mixture-of-experts layer.	181
5.41	Block-sparse matrix format used in MegaBlocks.	182
5.42	MoEs trained on The Pile with different capacity factors.	183
5.43	MegaBlocks dMoEs, Tutel dMoEs and Megatron-LM transformers trained on The Pile dataset.	183
5.44	MegaBlocks dMoEs, Tutel MoEs and Megatron-LM transformers trained on The Pile dataset.	184
5.45	Performance of Mixtral and different Llama models.	184
5.46	Overview of grouped-query method.	185
5.47	Sliding window attention.	185
5.48	Mixture of experts layer.	186
5.49	Proportion of tokens assigned to each expert.	187
5.50	Text samples where each token is colored with the first expert choice.	187
5.51	Comparison of Mixtral and LLaMa-2 models through diverse metrics.	188
5.52	Retrieval performance of Grok-1.5.	189
5.53	Comparison of DBRX with related open source models.	190
5.54	Comparison of the DBRX and LLaMa-2 performance.	190
5.55	Inference throughput for various models.	191
5.56	Visualization of the routing decision on OpenMoE.	193
5.57	Visualization of the routing decisions.	193
5.58	Evaluate OpenMoE on MTBench.	194
5.59	Comparison of DeepSeek-v2 with related models.	195

5.60	Illustration of different forms of attention.	195
5.61	Illustration of the architecture of DeepSeek-v2.	196
5.62	Benchmark performance of DeepSeek-V3 and its counterparts.	199
5.63	Illustration of DeepSeek-v3 multi-token prediction implementation.	199
5.64	Benchmark performance of DeepSeek-R1.	201
5.65	Scaling of transformer encoder with MoE layers in GShard.	201
5.66	An overview of the pretraining tasks in M6.	202
5.67	Overview of the V-MoE architecture.	203
5.68	Many-to-many dataset and multilingual model.	203
5.69	GLaM model architecture.	204
6.1	The original encoder-decoder transformer architecture.	208
6.2	The decoder-only transformer architecture.	209
6.3	Converting raw text into a sequence of tokens.	209
6.4	Token vocabulary for an LLM.	210
6.5	Input tensor of token embeddings.	211
6.6	Illustration of the dependency between input tokens and attention's output.	212
6.7	An example of the attention mechanism.	213
6.8	Creating key, query and value tensors.	214
6.9	Computing an attention score for a token pair.	214
6.10	Computing attention scores and output in self-attention.	215
6.11	Computing masked attention scores.	215
6.12	Multi-head attention.	215
6.13	Combining the output of multiple attention heads.	216
6.14	Feed-forward network.	218
6.15	Decoder-only transformer block.	219
6.16	Residual connection in a generic neural network layer.	219
6.17	Predicting next tokens with an LLM.	222
6.18	Autoregressive output generation with next token prediction.	223
6.19	Causal mask for next token prediction.	223
6.20	Adding multiple experts to a feed-forward network.	225
6.21	Routing mechanism for a single token.	226
6.22	Token routing dynamics in switch transformer.	228
6.23	MoE-based decoder-only transformer architecture.	233
6.24	Illustration of computing the output of an expert layer.	235
6.25	An example of divergence during MoE pretraining.	236
6.26	Training loss during MoE pre-training.	240
6.27	Validation loss during MoE pre-training.	240
6.28	Learning rate variation during MoE pre-training.	240
7.1	MoE hybrid parallelism techniques taxonomy in MoE trainin.	245
7.2	Runtime preferences of two different parallelism methods.	250
7.3	Weight migration due to switching parallelism.	250
7.4	Example of an MoE layer across three GPUs.	251
7.5	Distribution of strategies for various MoE configurations.	251
7.6	An example of data parallel execution flow in Tutel.	253
7.7	Specifying the parallelism method using <code>adaptive:r</code>	253
7.8	An example of EP+DP+MP execution flow in Tutel.	253
7.9	Token partition for All-to-All-Expert multi-stream overlapping.	254
7.10	Standard MoE and PR-MoE.	256
7.11	DS-MoE multi-dimensional parallelism.	258
7.12	DeepSpeed-MoE hierarchical All-to-All design.	259
7.13	DeepSpeed-MoE parallelism coordinated communication.	260
7.14	Comparison of dispatch overheads and communication patterns.	261

7.15	Pathways system overview.	261
7.16	Sequential vs. parallel dispatch for a three-node program.	262
7.17	Expert parallelism distributes experts across GPUs.	264
7.18	Traditional constraints vs. MoE parallel folding decoupling.	265
7.19	Parallel folding.	267
7.20	Expert parallelism distributes MoE experts across multiple GPUs.	267
7.21	Elastic expert parallelism scale-up.	271
7.22	Transformer architecture.	276
7.23	Transformer layer with tensor parallelism.	276
7.24	Transformer layer with tensor and sequence parallelism.	277
7.25	MLP layer with tensor and sequence parallelism.	277
7.26	Multi-head attention transformer.	279
7.27	DeepSpeed-Ulysses sequence parallelism design.	279
7.28	Ring attention sequence parallelism.	282
7.29	Reducing transformers memory cost with ring attention.	282
7.30	Partitioning of Q , K , and V sequences into blocks for both ring and striped attention.	284
7.31	Behavior of ring attention when applied to a casual self-attention problem.	284
7.32	Behavior of striped attention when applied to a casual self-attention problem.	285
7.33	Workload distribution of ring attention and striped attention.	286

List of Tables

2.1	Example that illustrates the distributed computing terminology.	10
2.2	Synchronous and asynchronous parameter updates.	24
3.1	Results for training the simple transformer with FSDP2.	82
5.1	Overview of diverse auxiliary loss functions used in MoE models.	129
5.2	Comparative configurations of MoE with FFN experts.	135
5.3	Open-source MoE models.	136
5.4	Comparison of open-source MoE frameworks.	143
5.5	Comparative analysis of computation-communication overlap technologies.	178
5.6	Comparison of Mixtral with LLaMa on multilingual benchmarks.	186
5.7	Comparison of Grok model with related models on four benchmarks.	188
5.8	Training efficiency of DBRX and other models.	191
5.9	Quality of DBRX Instruct and leading open models.	192
5.10	Quality of DBRX Instruct and leading closed models.	192
5.11	Data mixture used during pre-training of three versions of OpenMoE.	193
5.12	Top tokens selected by each expert in OpenMoE.	194
5.13	Training costs of DeepSeek-v3.	200
6.1	Results obtained during MoE pre-training.	238
7.1	All-to-All overhead and speedup when using communication and computation overlap.	251
7.2	Analysis on communication complexity of MoE parallelism.	252
7.3	Parallelism requirements of attention layers and MoE layers.	264
7.4	Megatron-Core parallelism dimensions.	266
7.5	Hyperparameters for training CIFAR10 MoE model that were kept fixed.	293
7.6	Additional hyperparameters for training the CIFAR10 MoE model.	294
7.7	Additional hyperparameters for training the CIFAR10 PR-MoE model.	294
7.8	Metrics relative to the CIFAR10 CNN models.	295

Listings

2.1	Initialization through environment variables.	11
2.2	Initialization through a TCP server.	12
2.3	Initialization through a shared file.	12
2.4	Python code to experiment with the broadcast collective operation.	13
2.5	Python code to try the AllReduce collective operation.	14
2.6	Python code for experimenting with the reduce collective operation.	15
2.7	Python code to demonstrate the AllGather collective operation.	15
2.8	Python code for experimenting with the gather collective operation.	17
2.9	Python code for experimenting with the scatter collective operation.	17
2.10	Python code for experimenting with the ReduceScatter collective operation.	18
2.11	Python code for experimenting with the AlltoAll collective operation.	20
3.1	Synchronize the gradients in DDP.	35
3.2	A simple neural network with three layers.	38
3.3	Initialize the group of processes to use in distributed training.	39
3.4	Prepare a distributed data loader.	39
3.5	Instantiate the model and wrap it with DistributedDataParallel.	40
3.6	A minimalist training loop.	40
3.7	Pseudo-code for start training with DDP.	41
3.8	Create a CIFAR10 dataset for training.	41
3.9	Saving and loading model checkpoints on a DDP context.	42
3.10	The neural network that we will train on the MNIST dataset in a distributed way.	48
3.11	Helper functions to call a given method locally or on a remote node.	49
3.12	Constructor method of the parameter server class.	49
3.13	The forward method of the parameter server class.	49
3.14	Methods to retrieve distributed gradients and to wrap the local weights with RRefs.	50
3.15	Methods to initialize the parameter server.	50
3.16	The constructor method of the worker class.	51
3.17	Method to retrieve a list of RRefs for the remote weights that reside on the parameter server.	51
3.18	The forward method of the worker class.	51
3.19	Training loop: instantiating the worker and the distributed optimizer.	52
3.20	Training loop: run the forward-backward-optimizer steps on each minibatch.	52
3.21	Calculate accuracy on the test dataset after the model was trained.	53
3.22	Function to initialize RPC in each worker and run the training loop.	53
3.23	Specify the arguments necessary to instantiate the parameter server and workers.	54
3.24	Create a process corresponding to either a parameter server or a worker.	55
3.25	Transformer model initialization and wrap the model with fully_shard.	75
3.26	Inspect how the sharded weights are distributed by ranks.	76
3.27	Explicit prefetching during the forward and backward passes.	76
3.28	Casting the unshared and sharded weights to different precisions.	77
3.29	Training loop where the optimizer works with DTensors.	78
3.30	Load state dictionaries.	78
3.31	Save state dictionaries.	79
3.32	Load a model state dictionary from file using the DCP API.	79

3.33	Save the model state dictionary to file using the DCP API.	79
3.34	The main function to train the transformer model with FSDP2 (part 1).	80
3.35	The main function to train the transformer model with FSDP2 (part 2).	81
4.1	Partition a model in two pipeline stages with FairScale.	94
4.2	Partition a model in two pipeline stages with DeepSpeed.	94
4.3	A memory efficient description of a model in DeepSpeed.	95
4.4	Associate a schedule to a model stage and run a forward step over this schedule.	97
4.5	A partial transformer model.	97
4.6	Manually splitting a transformer model in two stages.	98
4.7	A second version of a transformer model.	98
4.8	Summary of the transformer model architecture.	99
4.9	Specify the point where a model must be split.	99
4.10	The transformer model restructured by the <code>pipeline</code> class.	99
4.11	Adding data parallelism to a pipeline stage of a model.	100
4.12	A simple transformer decoder model.	101
4.13	Initializing the distributed training process.	102
4.14	Split the simple transformer decoder model manually.	102
4.15	Function for splitting the simple transformer model automatically.	103
4.16	The main function to train the simple transformer with pipeline parallelism.	103
4.17	Experiment the scatter and gather collective operations.	112
4.18	Row-wise tensor parallelism applied to Linear layers.	114
4.19	Main function to train the model with tensor parallelism.	115
4.20	Using <code>torch.distributed.tensor.parallel</code> to apply tensor parallelism.	116
5.1	Implementation of the noisy top-k gating.	147
5.2	Pseudo-code for a two-layer feed-forward dMoE.	181
6.1	Tokenize a textual sequence.	209
6.2	Implementation of masked multi-headed self-attention.	217
6.3	Implementation of the feed-forward network.	218
6.4	Implementation of the decoder-only transformer block.	220
6.5	Implementation of the decoder-only transformer.	221
6.6	Implementation of an expert layer.	224
6.7	Interleaving expert layers with dense transformer blocks.	225
6.8	Implementation of the <code>softmax</code> router.	227
6.9	Implementation of a fully-functional router.	229
6.10	Compute the load balancing loss term.	231
6.11	Compute the the ST-MoE router z-loss.	232
6.12	Implementation of an expert layer.	234
6.13	Implementation of an MoE block.	235
6.14	Switch transformer weight initialization strategy.	237
7.1	Configure the MoE model.	268
7.2	Enable expert parallelism in the MoE model.	268
7.3	Enable expert tensor parallelism in the MoE model.	268
7.4	Apply DeepEP token dispatcher optimization.	268
7.5	Apply HybridEP token dispatcher optimization.	269
7.6	Control token dropping to improve load balancing.	269
7.7	Configure an optimized MoE model in Megatron Bridge.	270
7.8	Create two weight groups in DeepSpeed MoE for the optimizer.	287
7.9	Initialize DeepSpeed distributed training.	288
7.10	Convolutional neural network with a MoE layer for CIFAR10 classification.	288
7.11	Initialize the DeepSpeed distributed backend.	289
7.12	Create train and test CIFAR10 datasets.	289
7.13	Wrap the model with DeepSpeed.	290
7.14	Train the MoE model.	291

7.15	Evaluate the model on the test set.	292
A.1	Python implementation of the transformer feed-forward network.	314
A.2	Python implementation of the transformer attention layer.	315
A.3	Python implementation of the transformer block.	315
A.4	Python implementation of a simple transformer model.	316
A.5	Utilities to inspect the model wrapped by FSDP2 and the mixed precision.	316
A.6	Class for saving and loading model and optimizer state dictionaries to/from file (part 1). . . .	317
A.7	Class for saving and loading model and optimizer state dictionaries to/from file (part 2). . . .	318
A.8	Class for saving and loading model and optimizer state dictionaries to/from file (part 3). . . .	319
A.9	Class for saving and loading model and optimizer state dictionaries to/from file (part 4). . . .	320
A.10	Class for saving and loading model and optimizer state dictionaries to/from file (part 5). . . .	321
A.11	Set prefetch AllGather operations over model layers.	321
A.12	The main function to train the transformer model with FSDP2 (part 1).	322
A.13	The main function to train the transformer model with FSDP2 (part 2).	323

List of Acronyms

AMP	Automatic Mixed Precision
Async-SGD	Asynchronous Stochastic Gradient Descent
BCSR	Blocked Compressed Sparse Row
BFGS	Broyden-Fletcher-Goldfarb-Shanno algorithm
BMM	Batched Matrix Multiplication
BPE	Byte-Pair Encoding
BTM	Branch-Train-Merge
BTX	Branch-Train-MiX
CIFAR	Canadian Institute For Advanced Research
CNN	Convolutional Neural Network
CP	Context Parallelism
cuBLAS	CUDA Basic Linear Algebra Subroutines
CUDA	Compute Unified Device Architecture
DAG	Directed Acyclic Graph
DCN	Dynamic Capacity Network
DCP	Distributed Checkpoint
DDP	Distributed Data Parallel
DeepEP	DeepSeek Expert Parallelism
DP	Data Parallelism
EDP	Expert Data Parallelism
EP	Expert Parallelism
EPLB	Expert Parallelism Load Balancer
ETP	Expert Tensor Parallelism
FFN	Feed-Forward Network
FLOP	Floating Point Operations
FLOPS	Floating Point Operations per Second
FoE	Fusion of Experts
FSDP	Fully Sharded Data Parallel
GPT	Generative Pre-trained Transformer
HBM	High Bandwidth Memory

HFU	Hardware FLOPS Utilization
HybridEP	Hybrid Expert Parallelism
GeGLU	GELU-based Gated Linear Unit
GELU	Gaussian Error Linear Unit
GLaM	Generalist Language Model
GQA	Grouped-Query Attention
IPC	Inter-Process Communication
KV	Key-Value
L-BFGS	Limited-memory Broyden-Fletcher-Goldfarb-Shanno algorithm
LM	Language Modeling
LLM	Large Language Model
LoRA	Low-Rank Adaptation
MFU	Model FLOPS Utilization
MLA	Multi-head Latent Attention
MLP	Multi Layer Perceptron
MNIST	Modified National Institute of Standards and Technology
MoD	Mixture of Depths
MoE	Mixture of Experts
MoPE	Mixture of Weight-efficient Experts
MoT	Mixture of Tokens
MP	Model Parallelism
MPI	Message Passing Interface
MT	Machine Translation
MTP	Multi-Token Prediction
NCCL	NVIDIA Collective Communication Library
NLP	Natural Language Processing
P2P	peer-to-peer
PEFT	Parameter Efficient Fine-Tuning
PP	Pipeline Parallelism
PR-MoE	Pyramid-Residual Mixture of Experts
RDMA	Remote Direct Memory Access
ReLU	Rectified Linear Unit
RLHF	Reinforcement Learning from Human Feedback
RoPE	Rotary Positional Embeddings
RPC	Remote Procedure Call
RRef	Remote Reference

SCCL	Synthesized Collective Communication Library
SFT	Supervised Fine-Tuning
SGD	Stochastic Gradient Descent
SM	Streaming Multiprocessor
SMoP	Sparse Mixture of Prompts
SP	Sequence Parallelism
SPMD	Single Program Multiple Data
SWA	Sliding Window Attention
SwiGLU	(Swish-Gated Linear Unit
T5	Text-to-Text Transfer Transformer
TP	Tensor Parallelism
TPU	Tensor Processing Unit
V-MoE	Vision-MoE
XLA	Accelerated Linear Algebra
ZeRO	Zero Redundancy Optimization

Chapter 1

Introduction

The rapid advancement of machine learning, and in particular the rise of deep neural networks and large language models, has fundamentally transformed the landscape of artificial intelligence. Over the past decade, there has been a dramatic increase in the scale of models, datasets, and compute requirements, driven by the pursuit of higher accuracy, generalization, and emergent capabilities. Modern models often contain billions or even trillions of parameters, making them impossible to train efficiently on a single device. As a result, distributed training and parallel computing have become essential components of state-of-the-art machine learning systems.

This document provides a comprehensive and unified study of the principles, methods, and systems that enable scalable training and deployment of large-scale models, with a particular focus on parallelism techniques and Mixture of Experts (MoE) architectures. It combines theoretical insights with practical implementations, offering both a conceptual framework and a hands-on perspective on building efficient distributed systems.

The **second chapter** establishes the foundations of distributed machine learning. It introduces the key concepts that underpin distributed systems, including nodes, ranks, world size, and process groups, as well as the mechanisms that enable communication between distributed processes. Special attention is given to communication primitives, such as point-to-point messaging and collective operations, such as AllReduce or All-to-All, which play a central role in coordinating computation across multiple devices. Additionally, the document examines synchronous and asynchronous training paradigms, discussing their respective trade-offs in terms of convergence stability, system efficiency, and scalability. Fundamental scaling concepts, such as strong and weak scaling, are also presented to provide a deeper understanding of how performance evolves with increased computational resources.

Building on these foundations, the **third chapter** explores core distributed training strategies, focusing on data parallelism and parameter server architectures. Data parallelism, which replicates the model across devices while partitioning the dataset, is presented as the most widely used approach in practice. The discussion includes gradient synchronization techniques, communication backends, and optimization strategies that improve efficiency, such as overlapping computation with communication. In contrast, the parameter server paradigm introduces a centralized or semi-centralized approach to model updates, enabling asynchronous execution and improved fault tolerance in certain scenarios. The document examines both approaches in depth, analyzing their strengths, limitations, and suitability for different workloads. Furthermore, it presents advanced memory optimization techniques, such as Zero Redundancy Optimizer (ZeRO) and Fully Sharded Data Parallel (FSDP), which allow models to scale beyond the memory constraints of individual devices by partitioning parameters, gradients, and optimizer states across the system.

The **fourth chapter** then focuses on model and computational parallelism, including naive model parallelism, pipeline parallelism, and tensor parallelism. These techniques enable the partitioning of models across devices, either by dividing layers, splitting computations within layers, or organizing execution into pipelines. The document highlights the trade-offs associated with each method, such as communication overhead, load imbalance, and hardware utilization. To address these limitations, it introduces hybrid parallelism strategies, which combine multiple approaches to achieve better scalability and efficiency. The integration of these

techniques is shown to be critical for training large-scale models, where no single parallelism method is sufficient on its own.

A central theme of this work is the Mixture of Experts (MoE) paradigm, which represents a major step forward in scaling neural networks, and are covered on **chapter five**. Unlike traditional dense models, MoE architectures rely on conditional computation, where only a subset of specialized sub-networks, called experts, is activated for each input. This enables models to dramatically increase their parameter count without a proportional increase in computational cost. The document provides a detailed analysis of MoE models, including their architectural components, routing mechanisms, and training strategies. It also presents a taxonomy of MoE approaches, covering both dense and sparse formulations, and examines key challenges such as load balancing, routing instability, communication overhead, and training efficiency. By addressing these challenges, MoE systems can achieve unprecedented levels of scalability and performance.

To bridge theory and practice, **chapter six** includes a comprehensive implementation-focused exploration of MoE models. It revisits the transformer architecture, particularly the decoder-only design used in large language models, and demonstrates how MoE layers can be integrated into this framework. Key aspects such as routing algorithms, expert capacity management, and load balancing mechanisms are discussed in detail, along with practical considerations for stable and efficient training. Through concrete implementation examples, the document highlights the engineering challenges involved in deploying MoE systems and provides guidance on best practices, including mixed precision training, initialization strategies, and numerical stability techniques.

Chapter seven, the last chapter of the document, is dedicated to advanced parallelism strategies specifically tailored for MoE architectures, where scalability becomes a multi-dimensional optimization problem. It examines how different parallelism techniques, including data, tensor, pipeline, sequence, and expert parallelism, can be combined into sophisticated hybrid configurations. The discussion traces the evolution from homogeneous strategies, which apply a uniform configuration across all layers, to heterogeneous approaches, which exploit the structural differences between dense and sparse components. Finally, it introduces dynamic and adaptive parallelism methods, capable of adjusting configurations at runtime to respond to changing workloads and system conditions. The increasing role of automatic parallelization frameworks is also highlighted, demonstrating how optimal configurations can be discovered through systematic search and optimization.

In addition, the chapter examines several state-of-the-art systems and frameworks that implement these ideas in practice. These include DeepSpeed, Tutel, Megatron, and Pathways, each of which introduces unique innovations in areas such as communication optimization, scheduling, and resource management. Concepts such as asynchronous dataflow execution and parallel folding are discussed as solutions to the inherent mismatch between dense and sparse computations in MoE models. The document also covers industry-grade implementations, such as NVIDIA Megatron Bridge and vLLM, highlighting features like optimized token routing, load balancing mechanisms, and elastic expert parallelism, which enable dynamic scaling and fault tolerance in real-world deployments.

Another important aspect addressed in this chapter is the challenge of long-context modeling, which requires efficient handling of very long input sequences. The document presents advanced techniques in sequence parallelism, including methods such as Megatron sequence parallelism, Ring Attention, and Striped Attention. These approaches reorganize computation and communication patterns to mitigate the quadratic complexity of attention mechanisms, enabling efficient scaling to longer sequences while maintaining performance.

Finally, the chapter concludes by bringing together these concepts through a practical implementation example, demonstrating how the DeepSpeed-MoE framework can be used to build, train, and evaluate MoE models. The presented example illustrates how theoretical principles translate into real-world systems and highlight the importance of carefully balancing computation, memory, and communication.

In conclusion, this document provides a comprehensive and integrated perspective on distributed training and parallelism in modern deep learning. It shows that scaling large models is not solely a matter of increasing computational resources, but rather a complex design challenge that requires the coordinated optimization of algorithms, system architectures, and communication strategies. As machine learning continues to evolve, these techniques will remain central to enabling the next generation of intelligent systems.

Chapter 2

Distributed Training

Contents

2.1	Distributed systems	8
2.2	Basic concepts in distributed training	10
2.3	Communication backends	11
2.4	Collective communication operations	12
2.5	Point-to-point communication and synchronization operations . . .	20
2.6	Types of parallelism in distributed training	20
2.7	How to select the parallelism type	21
2.8	Synchronous and asynchronous parameter updates	21
2.9	Strong and weak scaling	25
2.10	Hardware considerations	26
2.11	Conclusions	28

This chapter provides an overview of distributed training in machine learning, beginning with the fundamentals of distributed systems and the motivations for their use, particularly the rapid growth in model size, dataset scale, and computational power requirements. It introduces key concepts and terminology such as nodes, ranks, world size, and process groups, and explains how distributed processes communicate using different backends and initialization methods. The chapter then explores communication mechanisms in detail, including both point-to-point and collective operations, which are essential for coordinating computation across multiple devices. It also presents different types of parallelism, data, pipeline, and tensor parallelism, and discusses how to choose the most appropriate strategy depending on model size, hardware, and training objectives. Further, it is examined synchronous and asynchronous parameter update schemes, highlighting their trade-offs in terms of performance, convergence, and system efficiency. It concludes by covering scaling concepts, strong and weak scaling, and key hardware considerations, emphasizing how memory, interconnect bandwidth, and system topology influence the design and efficiency of distributed training systems.

2.1 Distributed Systems

A distributed system consists of multiple software components which run on multiple machines. For example, the traditional database runs on a single machine. As the amount of data gets incredibly large, a single machine can no longer deliver desirable performance to the business, especially in situations such as Black Friday where network traffic can be unexpectedly high. To handle such pressure, modern high-performance database is designed to run on multiple machines, and they work together to provide high throughput and low latency to the user.

One important evaluation metric for distributed system is scalability. For example, when we run an application on 4 machines, we naturally expect that the application can run 4 times faster. However, due to communication overhead and difference in hardware performance, it is difficult to achieve linear speedup. Thus, it is important to consider how to make the application faster when we implement it. Algorithms of good design and system optimization can help to deliver good performance. Sometimes, it is even possible to achieve linear and super-linear speedup.

Why do we need distributed training in machine learning?

Back in 2012, AlexNet [110] won the champion of the ImageNet competition, and it was trained on two GTX 580 3GB GPUs. Today, most models that appear in the top AI conferences are trained on multiple GPUs. Distributed training is definitely a common practice when researchers and engineers develop AI models. There are several reasons behind this trend.

1. Model size increases rapidly. ResNet50 had 20 million parameters in 2015 [78], BERT-Large had 345 million parameters in 2018 [52], GPT-2 had 1.5 billion parameters in 2018 [162], and GPT-3 had 175 billion parameters in 2020 [17]. It is obvious that the model size grows exponentially with time, as illustrates Figure 2.1. The current largest model has exceeded more than 1000 billion parameters. Super large models generally deliver more superior performance compared to their smaller counterparts.

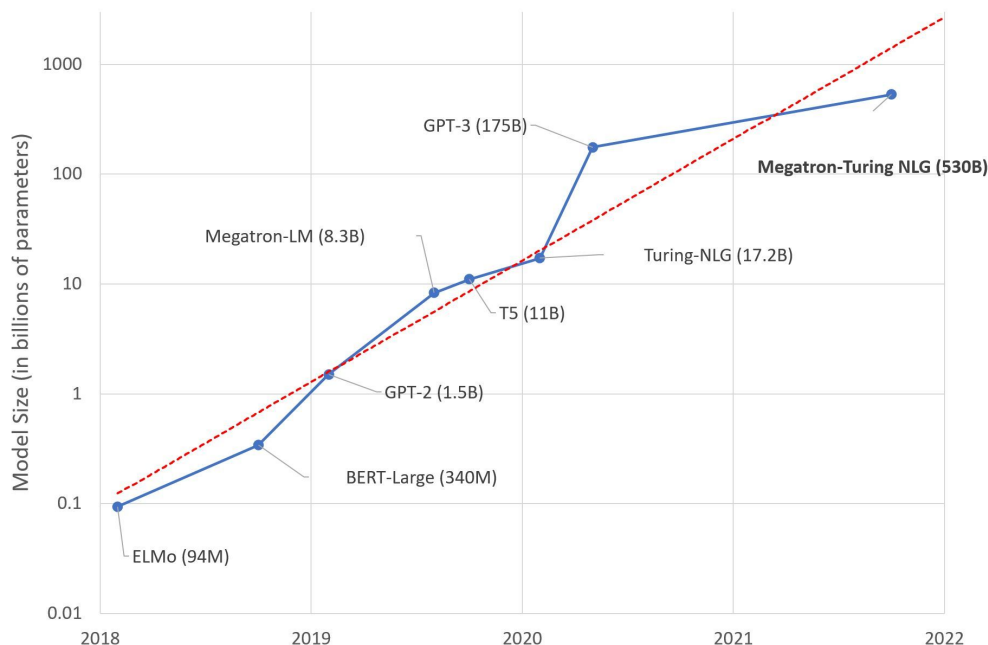


Figure 2.1: Evolution of size of state-of-the-art NLP models over time [5].

2. Dataset size increases rapidly. For most machine learning developers, MNIST and CIFAR10 datasets are often the first few datasets on which they train their models. However, these datasets are very small compared to well-known ImageNet datasets. Google even has its own (unpublished) JFT-300M dataset which has around 300 million images, and this is close to 300 times larger than the ImageNet-1k dataset.
3. Computing power gets stronger. With the advancement in the semiconductor industry, graphics cards become more and more powerful. Due to its larger number of cores, GPU is the most common compute platform for deep learning. From K10 GPU in 2012 to A100 GPU in 2020, the computing power has increased several hundred times. This allows us to performance compute-intensive tasks faster and deep learning is exactly such a task.

Nowadays, the model can be too large to fit into a single GPU, and the dataset can be large enough to train for a hundred days on a single GPU. Only by training our models on multiple GPUs with different parallelization techniques, we are able to speed up the training process and obtain results in a reasonable amount of time.

2.2 Basic Concepts in Distributed Training

Distributed training requires multiple machines and/or multiple GPUs. During training, there will be communication among these devices. To understand distributed training better, there are several important terms to be made clear. **textbfHost** is the main device in the communication network. It is often required as an argument when initializing the distributed environment. **Node** is a physical or virtual machine that is being used in model training. The only pre-requisite for communication between nodes used in model training is that they should be connected via high-speed LAN in order to reduce latency. A **port** mainly refers to master port on the host for communication. **World size** is the number of processes invoked during the training job and in the common setup each process is assigned to a different GPU. **Global rank** is a unique identifier assigned to a device in the network. The **local rank** is the unique identifier assigned to each process within a node. Finally, a **process group** is a communication network which include a subset of the devices. There is always a default process group which contains all the devices. A subset of devices can form a process group so that they only communicate among the devices within the group.

To illustrate these concepts, let us assume we have two nodes and each node has four GPUs (Figure 2.2). When we initialize distributed environment over these two machines, we essentially launch 8 processes, 4 processes on each machine, and each process is bound to a distinct GPU. Before initializing the distributed environment, we need to specify the host (master address) and port (master port). In this example, we can let host to be node with rank 0 and port be a number such as 29500. All the 8 processes will then look for the address and port and connect to one another. The default process group will then be created. The default process group has a world size of 8 and details about the group are listed in Table 2.1.

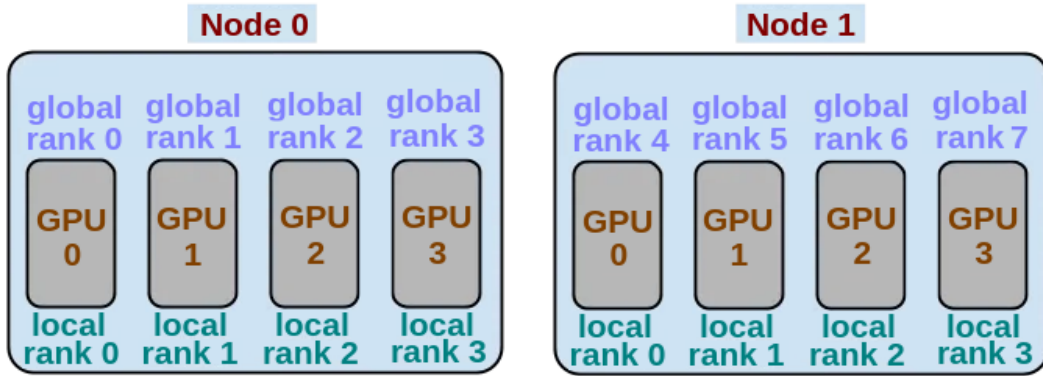


Figure 2.2: Example that illustrates the distributed computing terminology.

node rank	GPU ID	local rank	global rank	process ID
0	0	0	0	0
0	1	1	1	1
0	2	2	2	2
0	3	3	3	3
1	0	0	4	4
1	1	1	5	5
1	2	2	6	6
1	3	3	7	7

Table 2.1: Example that illustrates the distributed computing terminology.

In the process group, the processes can communicate (i) point-to-point, where one process send data to another process, and (ii) collective, where a group of process perform operations such as scatter, gather, all-reduce, or broadcast together.

2.3 Communication Backends

For these distributed processes to collaborate, they need a way to talk to each other. The PyTorch `torch.distributed` package manages this collaboration through communication backends and initialization methods using `torch.distributed.init_process_group()`. The backend parameter in `init_process_group()` specifies the library that we will use to exchange data between process.

NVIDIA Collective Communication Library (NCCL) is highly optimized for inter-GPU communication, both within a single node and across multiple nodes [96]. It provides efficient implementation of collective operations defined in Section 2.4.

Gloo is a platform agnostic backend that works for both CPU and GPU-based communication¹. For GPUs, it typically involves copying data from GPU to CPU memory, performing network communication, and then copying back to GPU. This makes it slower than NCCL for GPU tasks. This backend is used as a fallback if NCCL setup is problematic or specific operations are not optimized in NCCL.

Message Passing Interface (MPI) is a widely adopted standard and portable API in High-Performance Computing (HPC) to build scalable applications on multi-node compute clusters by communicating data via messages between distributed processes [151]. MPI provides functions for point-to-point communication, like sending and receiving messages, and collective communication such as broadcasting, scattering and gathering data across processes.

Once the backend is chosen, processes need a way to find each other and agree on the role assigned to each on, such as which node is rank 0, which node is rank 1, etc. This is a synchronization step specified by the `init_method`. There are three possibilities for specify the nodes' role.

1. Initialization through environment variables, where `init_method="env://"`.

In this case, processes read connection details from environment variables, such as `MASTER_ADDR`, which defines the IP address or host name of the master node (or rank 0 node), and `MASTER_PORT`, which specifies a free network port on the master node for initial coordination. Listing 2.1 provides an example where we create four processes and initialize the communication through environment variables. Each process creates a random tensor and all tensors are reduced to a unique tensor through an AllReduce operation.

```
1 import torch [sc name="example01"]
2 import torch.distributed as ptd [sc name="example01"]
3 import torch.multiprocessing as mp
4
5 def worker(rank):
6     ptd.init_process_group(backend="nccl", rank=rank, world_size=4)
7     torch.cuda.set_device(rank)
8     tensor = torch.randn(32 if rank == 0 else 64).cuda()
9     dist.all_reduce(tensor)
10    torch.cuda.synchronize(device=rank)
11
12 if __name__ == "__main__":
13     os.environ["MASTER_ADDR"] = "localhost"
14     os.environ["MASTER_PORT"] = "29500"
15     os.environ["TORCH_CPP_LOG_LEVEL"] = "INFO"
16     os.environ["TORCH_DISTRIBUTED_DEBUG"] = "DETAIL"
17     mp.spawn(worker, nprocs=4, args=())
```

Listing 2.1: Python code for creating four processes and using initialization through environment variables.

2. Initialization through a TCP server specified with `init_method = "tcp://<master_ip>:<port>"`.

The process designated as rank 0 starts a TCP server at the selected IP port and the other worker processes connect to this server to exchange information and create the group. The address `<master_ip>` must be reachable by all workers. In a single-node setup, using `<master_ip>=localhost` or

¹<https://github.com/pytorch/gloo>

<master_ip>=127.0.0.1 is common. Listing 2.2 provides an example where we create a group of four processes and initialize the communication through a TCP server.

```
1 import torch.distributed as ptd
2 # Use address of one of the machines
3 ptd.init_process_group(
4     backend      = "nccl",
5     init_method  = 'tcp://4.1.1.123:45678',
6     rank        = args.rank,
7     world_size   = 4,
8 )
```

Listing 2.2: Python code for creating four processes and using initialization through a TCP server.

3. Initialization through a shared file specified with `init_method = "file:///path/to/shared/file")`.

In this case, processes coordination is accomplished via a file on a shared file system, such as NFS, accessible by all. Rank 0 process write its connection info to a temporary file and the other processes read it. It is useful in environments with a reliable shared file system where direct network discovery might be complex. Listing 2.3 provides an example where we create a group of four processes and initialize the communication through a shared file.

```
1 import torch.distributed as ptd
2 # rank should always be specified
3 ptd.init_process_group(
4     backend      = "nccl",
5     init_method  = 'file:///mnt/nfs/sharedfile',
6     world_size   = 4,
7     rank        = args.rank,
8 )
```

Listing 2.3: Python code for creating four processes and using initialization through a shared file.

2.4 Collective Communication Operations

In this section, we describe the collective operations supported by NVIDIA nccl library [96]. The collective operations have to be called in each rank, or GPU, using the same count and the same data type, to form a complete collective operation. Failure to do so will result in an undefined behavior.

Broadcast

The Broadcast operation copies an N -element buffer from the root rank to all the ranks. The root argument of the operation is one of the ranks, not a device number, and is therefore impacted by a different rank to device mapping.

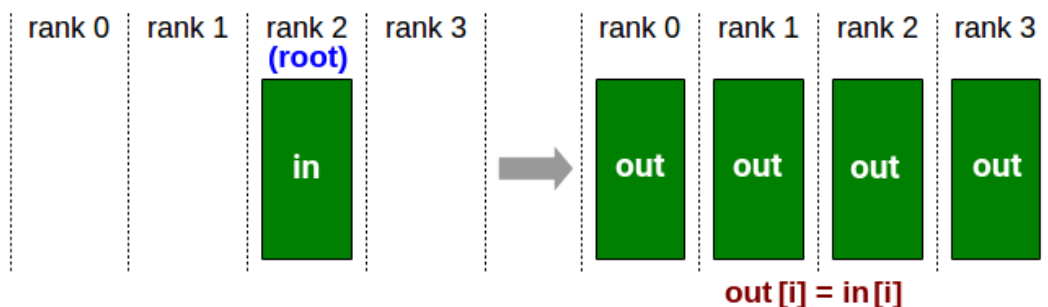


Figure 2.3: In a Broadcast operation, all ranks receive data from the root rank.

We can write a small example in PyTorch to demonstrate how the broadcast operation works. First, it is necessary to initialize a process group with `dist.init_process_group`, which sets up the communication

backend, determines how many workers exist, assigns a rank to each worker, and establishes a connection between workers. We can get the rank of a worker using `dist.get_rank`.

We create a tensor with nonzero values on `rank=0` and tensors full of zeros on the other workers. We then distribute the `rank=0` tensor to all other nodes, often called "ranks" in distributed communication, with `dist.broadcast(tensorIn, src=nodeSrc)` (Listing 2.4).

```

1 # FILE: collective_ops.py
2 import torch
3 import torch.distributed as ptd
4
5 def init_process():
6     grp = ptd.init_process_group(backend='nccl')
7     torch.cuda.set_device(ptd.get_rank())
8     return grp
9
10 def example_broadcast():
11     if ptd.get_rank() == 0:
12         tensor = torch.tensor([1, 2, 3, 4, 5], dtype=torch.float32).cuda()
13     else:
14         tensor = torch.zeros(5, dtype=torch.float32).cuda()
15     print(f"Tensor before broadcast on rank {ptd.get_rank()}: {tensor}")
16     ptd.broadcast(tensor, src=0)
17     print(f"Tensor after broadcast on rank {ptd.get_rank()}: {tensor}")
18
19 if __name__ == '__main__':
20     grp = init_process()
21     example_broadcast()
22     ptd.destroy_process_group(grp)

```

Listing 2.4: Python code to experiment with the broadcast collective operation.

We run the above script with `torchrun -nproc_per_node=3 collective_ops.py` and it produces the following output.

```

Tensor before broadcast on rank 1: tensor([0., 0., 0., 0., 0.], device='cuda:1')
Tensor before broadcast on rank 0: tensor([1., 2., 3., 4., 5.], device='cuda:0')
Tensor before broadcast on rank 2: tensor([0., 0., 0., 0., 0.], device='cuda:2')

```

```

Tensor after broadcast on rank 2: tensor([1., 2., 3., 4., 5.], device='cuda:2')
Tensor after broadcast on rank 0: tensor([1., 2., 3., 4., 5.], device='cuda:0')
Tensor after broadcast on rank 1: tensor([1., 2., 3., 4., 5.], device='cuda:1')

```

Note that the rank messages are printed out of order, since we have no control over which print statement is executed first.

AllReduce

The AllReduce operation performs a reduction over data, for example a sum, the minimum, or maximum, across devices and stores the result in the receive buffer of every rank (process). For example, in a sum AllReduce operation between k ranks, each process will provide an array of N values, and receives identically as result an array of N values, where $out[i] = in_0[i] + in_1[i] + \dots + in_{k-1}[i]$.

Listing 2.5 demonstrates the AllReduce operation. Communication setup is accomplished with the `init_process` included in Listing 2.4. First, each worker creates a 1D tensor all filled with the `rank+1` value. We then retrieve this tensor from all workers, combine them through the sum operation, and redistribute the resulting tensor `[6., 6., 6., 6., 6.]` by all nodes.

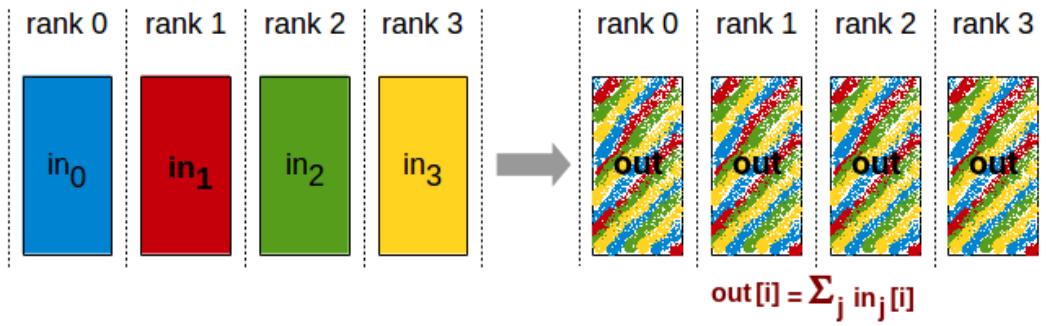


Figure 2.4: In the AllReduce operation, each rank receives the reduction of input values across ranks.

```

1 def example_AllReduce():
2     tensor = torch.tensor([ptd.get_rank() + 1] * 5, dtype=torch.float32).cuda()
3     print(f"Tensor before AllReduce on rank {ptd.get_rank()}: {tensor}")
4     ptd.all_reduce(tensor, op=ptd.ReduceOp.SUM)
5     print(f"Tensor after AllReduce on rank {ptd.get_rank()}: {tensor}")
6
7 if __name__ == '__main__':
8     grp = init_process()
9     example_AllReduce()
10    ptd.destroy_process_group(grp)

```

Listing 2.5: Python code to try the AllReduce collective operation.

The execution of the code included in Listing 2.5 with 3 workers produces the following output.

```

Tensor before AllReduce on rank 0: tensor([1., 1., 1., 1., 1.], device='cuda:0')
Tensor before AllReduce on rank 2: tensor([3., 3., 3., 3., 3.], device='cuda:2')
Tensor before AllReduce on rank 1: tensor([2., 2., 2., 2., 2.], device='cuda:1')

Tensor after AllReduce on rank 0: tensor([6., 6., 6., 6., 6.], device='cuda:0')
Tensor after AllReduce on rank 1: tensor([6., 6., 6., 6., 6.], device='cuda:1')
Tensor after AllReduce on rank 2: tensor([6., 6., 6., 6., 6.], device='cuda:2')

```

Reduce

The Reduce operation performs the same operation as AllReduce, but stores the result only in the receive buffer of a specified root rank. The *root* argument of the operation is one of the ranks, not a device number, and is therefore impacted by a different rank to device mapping. A *Reduce*, followed by a *Broadcast*, is equivalent to an AllReduce operation.

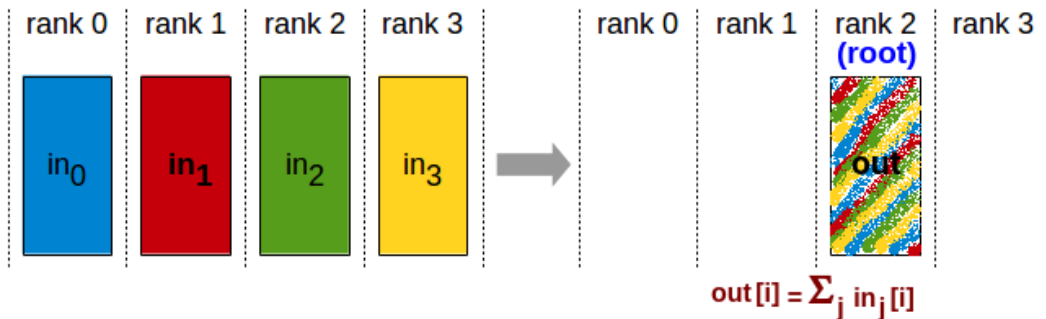


Figure 2.5: In the Reduce operation, one rank receives the reduction of input values across ranks.

Listing 2.6 illustrates reduce operation. First, each worker creates a 1D tensor all filled with the `rank+1` value. We then retrieve this tensor from all workers, combine them through the sum operation, and send the resulting tensor `[6., 6., 6., 6., 6.]` to rank 0 node.


```

1 def example_reduce():
2     rank = ptd.get_rank()
3     tensor = torch.tensor([rank + 1] * 5, dtype=torch.float32).cuda()
4     print(f"Tensor before reduce on rank {rank}: {tensor}")
5     ptd.reduce(tensor, dst=0, op=ptd.ReduceOp.SUM)
6     print(f"Tensor after reduce on rank {rank}: {tensor}")
7
8 if __name__ == '__main__':
9     grp = init_process()
10    example_reduce()
11    ptd.destroy_process_group(grp)

```

Listing 2.6: Python code for experimenting with the reduce collective operation.

The execution of the code included in Listing 2.6 with 3 workers produces the following output.

```

Tensor before reduce on rank 2: tensor([3., 3., 3., 3., 3.], device='cuda:2')
Tensor before reduce on rank 1: tensor([2., 2., 2., 2., 2.], device='cuda:1')
Tensor before reduce on rank 0: tensor([1., 1., 1., 1., 1.], device='cuda:0')

Tensor after reduce on rank 1: tensor([2., 2., 2., 2., 2.], device='cuda:1')
Tensor after reduce on rank 2: tensor([3., 3., 3., 3., 3.], device='cuda:2')
Tensor after reduce on rank 0: tensor([6., 6., 6., 6., 6.], device='cuda:0')

```

AllGather

The AllGather operation gathers N values from k ranks into an output buffer of size $k * N$, and distributes that result to all ranks. The output is ordered by the rank index. The AllGather operation is therefore impacted by a different rank to device mapping. Executing a ReduceScatter, followed by AllGather, is equivalent to an AllReduce operation.

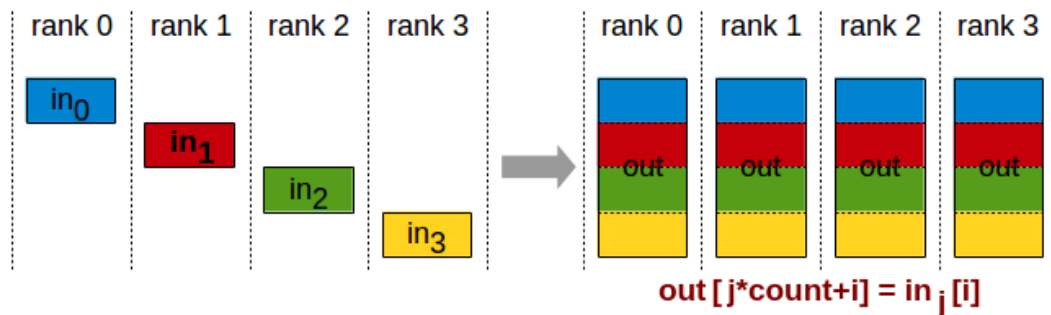


Figure 2.6: In the AllGather operation, each rank receives the aggregation of data from all ranks in the order of the ranks.

```

1 def example_all_gather():
2     rank = ptd.get_rank()
3     tensor = torch.tensor([rank + 1] * 5, dtype=torch.float32).cuda()
4     gather_list = [
5         torch.zeros(5, dtype=torch.float32).cuda()
6         for _ in range(ptd.get_world_size())
7     ]
8     print(f"Tensor before AllGather on rank {rank}: {tensor}")
9     ptd.all_gather(gather_list, tensor)
10    print(f"List of tensors after AllGather on rank {rank}: {gather_list}")
11
12 if __name__ == '__main__':
13     grp = init_process()
14     example_all_gather()
15     ptd.destroy_process_group(grp)

```

Listing 2.7: Python code to demonstrate the AllGather collective operation.

Listing 2.7 demonstrates how the AllGather operation works. Each node creates a 1D tensor all filled with the rank+1 value. Additionally, each node instantiates an object to store the tensors that will be gathered, the list

`gather_list`, with a number of elements equal to the size of our group of nodes. Each element of the list is initialized with an all zero tensor. The `AllGather` operation retrieves the defined tensor from all workers and places all of them in the list `gather_list` of every node.

The execution of the code included in Listing 2.7 with 3 workers produces the following output.

```
Tensor before AllGather on rank 0: tensor([1., 1., 1., 1., 1.], device='cuda:0')
Tensor before AllGather on rank 1: tensor([2., 2., 2., 2., 2.], device='cuda:1')
Tensor before AllGather on rank 2: tensor([3., 3., 3., 3., 3.], device='cuda:2')
List of tensors after AllGather on rank 1: [
  tensor([1., 1., 1., 1., 1.], device='cuda:1'),
  tensor([2., 2., 2., 2., 2.], device='cuda:1'),
  tensor([3., 3., 3., 3., 3.], device='cuda:1') ]
List of tensors after AllGather on rank 2: [
  tensor([1., 1., 1., 1., 1.], device='cuda:2'),
  tensor([2., 2., 2., 2., 2.], device='cuda:2'),
  tensor([3., 3., 3., 3., 3.], device='cuda:2') ]
List of tensors after AllGather on rank 0: [
  tensor([1., 1., 1., 1., 1.], device='cuda:0'),
  tensor([2., 2., 2., 2., 2.], device='cuda:0'),
  tensor([3., 3., 3., 3., 3.], device='cuda:0') ]
```

Gather

The `Gather` operation gathers N values from k ranks into an output buffer on the root rank of size $k * N$. The `root` argument of the operation is one of the ranks, not a device number, and is therefore impacted by a different rank to device mapping.

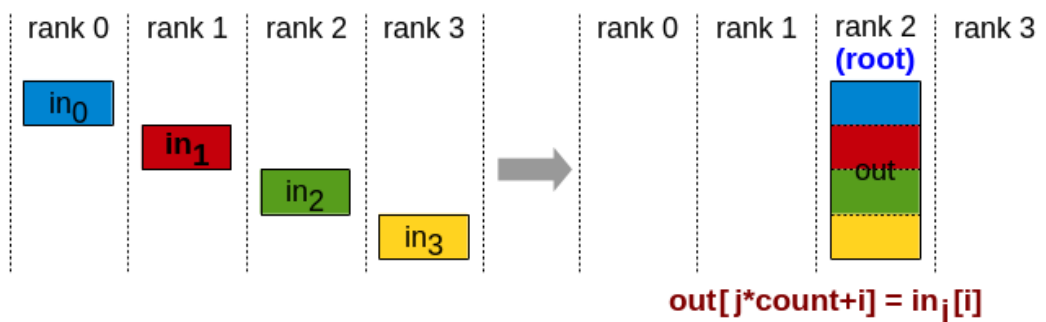


Figure 2.7: In the `Gather` operation, the root rank receives data from all ranks.

Listing 2.8 illustrates the `gather` operation. In this case, the rank 0 node instantiates an object to store the tensors that will be gathered, the list `gather_list`, with a number of elements equal to the size of our group of nodes. Each element of the list is initialized with an all zero tensor. Additionally, each node creates a 1D tensor all filled with the `rank+1` value. The rank 0 node, specified by the `dst` argument of the `gather` function, retrieves this tensor from all workers and places them in `gather_list`.

The execution of the code included in Listing 2.8 with 3 workers produces the following output.

```
Tensor before gather on rank 1: tensor([2., 2., 2., 2., 2.], device='cuda:1')
Tensor before gather on rank 2: tensor([3., 3., 3., 3., 3.], device='cuda:2')
Tensor before gather on rank 0: tensor([1., 1., 1., 1., 1.], device='cuda:0')
List of tensors gathered by rank 0:
[
  tensor([1., 1., 1., 1., 1.], device='cuda:0'),
  tensor([2., 2., 2., 2., 2.], device='cuda:0'),
  tensor([3., 3., 3., 3., 3.], device='cuda:0')
]
```

```

1 def example_gather():
2     rank = ptd.get_rank()
3     tensor = torch.tensor([rank + 1] * 5, dtype=torch.float32).cuda()
4     if rank == 0:
5         gather_list = [
6             torch.zeros(5, dtype=torch.float32).cuda()
7             for _ in range(ptd.get_world_size())
8         ]
9     else:
10        gather_list = None
11    print(f"Tensor before gather on rank {rank}: {tensor}")
12    ptd.gather(tensor, gather_list, dst=0)
13    if rank == 0:
14        print(f"List of tensors gathered by rank 0: {gather_list}")
15
16 if __name__ == '__main__':
17     grp = init_process()
18     example_gather()
19     ptd.destroy_process_group(grp)

```

Listing 2.8: Python code for experimenting with the gather collective operation.

Scatter

The Scatter operation distributes a total of $N * k$ values from the root rank to k ranks, each rank receiving N distinct values. The root argument of the operation is one of the ranks, not a device number, and is therefore impacted by a different rank to device mapping.

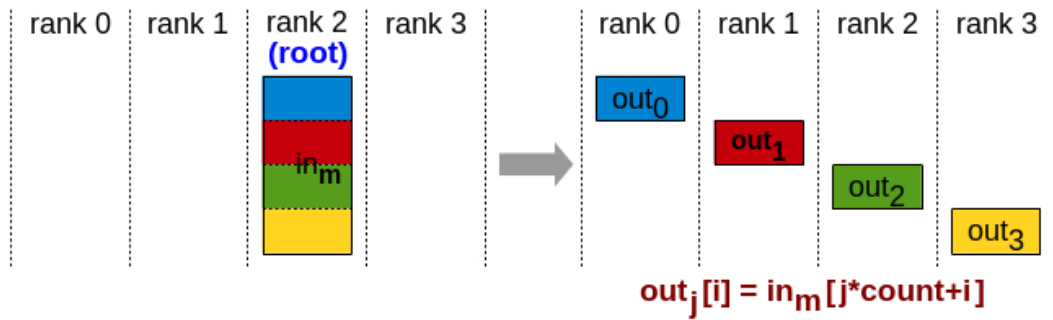


Figure 2.8: In the scatter operation, the root rank distributes data to all ranks.

```

1 def example_scatter():
2     rank = ptd.get_rank()
3     if rank == 0:
4         scatter_list = [
5             torch.tensor([i + 1] * 5, dtype=torch.float32).cuda()
6             for i in range(ptd.get_world_size())
7         ]
8         print(f"Rank 0 list of tensors to scatter: {scatter_list}")
9     else:
10        scatter_list = None
11    tensor = torch.zeros(5, dtype=torch.float32).cuda()
12    print(f"Tensor before scatter on rank {rank}: {tensor}")
13    ptd.scatter(tensor, scatter_list, src=0)
14    print(f"Tensor after scatter on rank {rank}: {tensor}")
15
16 if __name__ == '__main__':
17     grp = init_process()
18     example_scatter()
19     ptd.destroy_process_group(grp)

```

Listing 2.9: Python code for experimenting with the scatter collective operation.

Listing 2.9 illustrates the scatter operation. In this case, the rank 0 node instantiates a list `scatter_list` with a number of elements equal to the size of our group of nodes. The element in position `i` of the list is initialized

with a 1D tensor all filled with the value $i+1$. Additionally, each node creates an all zero 1D tensor. Then, rank 0 node scatters the list of tensors by all nodes, and each node rank i receives the tensor in position i of the scattered list.

The execution of the code included in Listing 2.9 with 3 workers produces the following output.

```
Rank 0 list of tensors to scatter: [
    tensor([1., 1., 1., 1., 1.], device='cuda:0'),
    tensor([2., 2., 2., 2., 2.], device='cuda:0'),
    tensor([3., 3., 3., 3., 3.], device='cuda:0') ]
Tensor before scatter on rank 0: tensor([0., 0., 0., 0., 0.], device='cuda:0')
Tensor before scatter on rank 2: tensor([0., 0., 0., 0., 0.], device='cuda:2')
Tensor before scatter on rank 1: tensor([0., 0., 0., 0., 0.], device='cuda:1')

Tensor after scatter on rank 2: tensor([3., 3., 3., 3., 3.], device='cuda:2')
Tensor after scatter on rank 1: tensor([2., 2., 2., 2., 2.], device='cuda:1')
Tensor after scatter on rank 0: tensor([1., 1., 1., 1., 1.], device='cuda:0')
```

ReduceScatter

The ReduceScatter operation performs the same operation as Reduce, except that the result is scattered in equal-sized blocks between ranks, each rank getting a chunk of data based on its rank index. The ReduceScatter operation is impacted by a different rank to device mapping since the ranks determine the data layout.

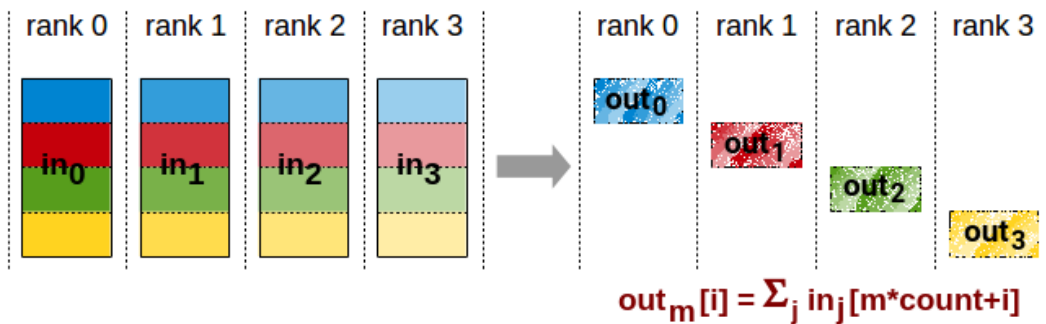


Figure 2.9: In the ReduceScatter operation, input values are reduced across ranks, with each rank receiving a parcel of the result.

```
1 def example_reduce_scatter():
2     rank = ptd.get_rank()
3     world_size = ptd.get_world_size()
4     input_tensor = [
5         torch.tensor(
6             [(rank + 1) * i for i in range(1, 3)], dtype=torch.float32).cuda() ** (j + 1)
7         for j in range(world_size)
8     ]
9     output_tensor = torch.zeros(2, dtype=torch.float32).cuda()
10    print(f"List of tensors before ReduceScatter on rank {rank}: {input_tensor}")
11    ptd.reduce_scatter(output_tensor, input_tensor, op=ptd.ReduceOp.SUM)
12    print(f"Output tensor after ReduceScatter on rank {rank}: {output_tensor}")
13
14 if __name__ == '__main__':
15     grp = init_process()
16     example_reduce_scatter()
17     ptd.destroy_process_group(grp)
```

Listing 2.10: Python code for experimenting with the ReduceScatter collective operation.

Listing 2.10 demonstrates how the ReduceScatter operation works. Considering an example with 3 nodes, each rank node creates a list of three tensors, `input_tensor`, with the values defined by Equation 2.1. Each node also defines an `output_tensor`, filled with zeros. After the ReduceScatter operation, the output tensor in rank node will have sum of the input tensors in position rank from all nodes.

$$input_tensor = \begin{bmatrix} [((rank + 1) * 1)^1, ((rank + 1) * 2)^1], \\ [((rank + 1) * 1)^2, ((rank + 1) * 2)^2], \\ [((rank + 1) * 1)^3, ((rank + 1) * 2)^3] \end{bmatrix} \quad (2.1)$$

The execution of the code included in Listing 2.10 with 3 workers produces the following output. The output tensor results from the sums defined in Equation 2.2.

$$\begin{aligned} [1, 2] + [2, 4] + [3, 6] &= [6, 12] \\ [1, 4] + [4, 16] + [9, 36] &= [14, 56] \\ [1, 8] + [8, 64] + [27, 216] &= [36, 288] \end{aligned} \quad (2.2)$$

List of tensors before ReduceScatter on rank 0: [

```
tensor([ 1., 2.], device='cuda:0'),
tensor([ 1., 4.], device='cuda:0'),
tensor([ 1., 8.], device='cuda:0')]
```

List of tensors before ReduceScatter on rank 1: [

```
tensor([ 2., 4.], device='cuda:1'),
tensor([ 4., 16.], device='cuda:1'),
tensor([ 8., 64.], device='cuda:1')]
```

List of tensors before ReduceScatter on rank 2: [

```
tensor([ 3., 6.], device='cuda:2'),
tensor([ 9., 36.], device='cuda:2'),
tensor([27., 216.], device='cuda:2')]
```

Output tensor after ReduceScatter on rank 2: tensor([36., 288.], device='cuda:2')

Output tensor after ReduceScatter on rank 1: tensor([14., 56.], device='cuda:1')

Output tensor after ReduceScatter on rank 0: tensor([6., 12.], device='cuda:0')

AlltoAll

In an AlltoAll operation between k ranks, each rank provides an input buffer of size $k * N$ values, where the j -th chunk of N values is sent to destination rank j . Each rank receives an output buffer of size $k * N$ values, where the m -th chunk of N values comes from source rank m .

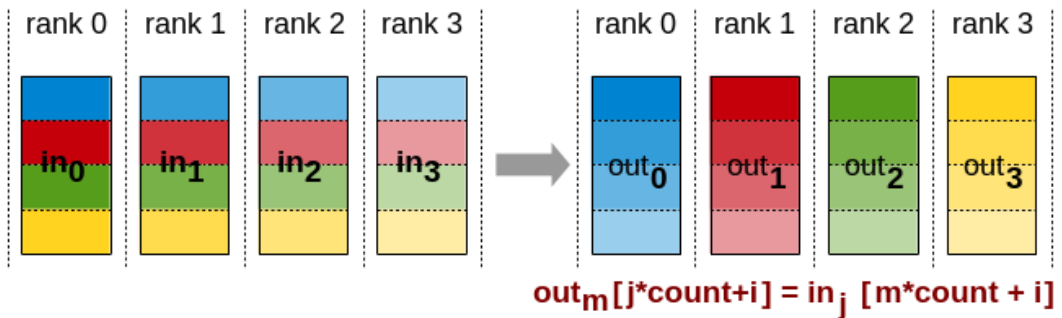


Figure 2.10: The AlltoAll operation exchanges data between all ranks, where each rank sends different data to every other rank and receives different data from every other rank.

Listing 2.11 demonstrates how the AlltoAll operation works. Considering an example with 3 nodes, each rank node creates an input tensor filled with values given by $[0, 1, 2] + rank * 3$. After the AlltoAll operation, the output tensor in rank node will have the values in position rank from the input tensor of all nodes.

```

1 def example_all2all():
2     rank = ptd.get_rank()
3     world_size = ptd.get_world_size()
4     input_tensor = torch.arange(world_size) + rank * world_size
5     input_tensor = input_tensor.cuda()
6     print(f"Input tensor before All2All on rank {rank}: {input_tensor}")
7     output_tensor = torch.empty([world_size], dtype=torch.int64).cuda()
8     ptd.all_to_all_single(output_tensor, input_tensor)
9     print(f"Output tensor after All2All on rank {rank}: {output_tensor}")
10
11 if __name__ == '__main__':
12     grp = init_process()
13     example_all2all()
14     ptd.destroy_process_group(grp)

```

Listing 2.11: Python code for experimenting with the AlltoAll collective operation.

The execution of the code included in Listing 2.11 with 3 workers produces the following output.

```

Input tensor before All2All on rank 0: tensor([0, 1, 2], device='cuda:0')
Input tensor before All2All on rank 1: tensor([3, 4, 5], device='cuda:1')
Input tensor before All2All on rank 2: tensor([6, 7, 8], device='cuda:2')

Output tensor after All2All on rank 0: tensor([0, 3, 6], device='cuda:0')
Output tensor after All2All on rank 1: tensor([1, 4, 7], device='cuda:1')
Output tensor after All2All on rank 1: tensor([2, 5, 8], device='cuda:2')

```

2.5 Point-to-Point Communication and Synchronization Operations

In addition to the collective operations, we can consider point-to-point communication and synchronization operations, such as (i) **push**, when one node N_1 sends some data to node N_2 , (ii) **pull**, when node N_2 requests some data from node N_1 ; Pull differs from push only in terms of who initiates the communication, (iii) **wait**, when a node pauses its computation and waits for data to be received from another node, and (iv) **barrier**, when many machines wait until all those machines reach the same point in their execution, in order to continue execution from that point. In PyTorch, a barrier can be created with the function `torch.distributed.barrier`.

2.6 Types of Parallelism in Distributed Training

Training modern machine learning models has become an increasingly demanding computational challenge. As models scale to hundreds of billions or even trillions of weights and datasets grow to unprecedented sizes, single-device training approaches are no longer feasible. Based on community estimates, GPT-4's training reportedly utilized a cluster of 20k-25k NVIDIA A100 GPUs over 3 to 4 months. Therefore, if the same training were attempted on a single A100 GPU, assuming linear scaling, it would take between 5 thousand and 8.3 thousand years to complete.

There are three main types of distributed parallel training: data parallel, pipeline parallel, and tensor parallel. The latter two methods are often considered two alternatives of model parallelism. Data parallelism replicates the entire model on each GPU, and each GPU processes a different slice of the training data. In pipeline parallelism, the model is divided into stages, each allocated to a different GPU. Mini-batches of data are further divided into micro-batches, which are fed into the pipeline. This allows different stages to process different micro-batches concurrently, improving GPU utilization compared to naive model parallelism. Tensor parallelism splits individual operations or layers, for example, a large matrix multiplication from a transformer attention module, across multiple GPUs. In this example, the weight matrix can be split and partial computations are done on different GPUs. Tensor parallelism is adequate for dealing with extremely large individual layers of a model. In the context of sparse Mixture of Experts models, arises another type of parallelism, expert parallelism, where the expert of a layer are split among GPUs.

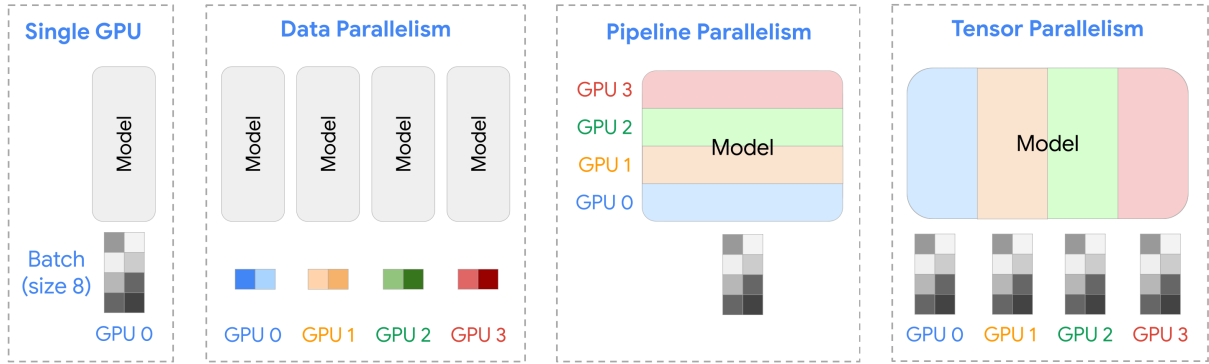


Figure 2.11: Data, pipeline, and tensor parallelism [126].

2.7 How to Select the Parallelism Type

We should opt for data parallelism as the default option, when our concern is convergence speed, our batch size is on the smaller side (smaller than 64) and our model fits into a single GPU. It works fine on both single-machine and multi-machine configurations, it is easy to implement, and is typically implemented in Deep Learning frameworks such as PyTorch and TensorFlow.

The option must be model parallelism if our model does not fit into a single device, when our model has a lot of parallel branches/paths, or if we want achieve high device utilization and memory efficiency. It is not recommended to use across multiple machines due to high communication costs. In some cases we have to implement it by ourselves.

We should opt for pipeline parallelism if our model does not fit into a single device, but using model parallelism is too difficult. It works really well with sequential models like CNNs and transformers, both across devices and across machines due to low communication cost between them, and it is not very hard to implement. It might be necessary to try several alternatives to split the model in order to balance the workload across devices.

The option can be hybrid parallelism if we are training a really large model or a lot of large models. If we are splitting the model across workers, there is usually no reason not to consider data parallelism on top of model parallelism for quicker convergence.

In conclusion, we compute gradients for subsets of samples in parallel (data parallel), we compute gradients for subsets of parameters in parallel (model parallel), and we split gradients into a composition of sequential steps (pipeline parallel).

2.8 Synchronous and Asynchronous Parameter Updates

When training models in a distributed setting with multiple workers processing different subsets of data, a fundamental question arises: how should the updates calculated by individual workers be combined to update the shared model parameters? The two primary strategies are synchronous and asynchronous updates, each presenting distinct trade-offs between computational efficiency, communication overhead, and convergence behavior [112].

2.8.1 Synchronous Parameter Updates

In a synchronous update scheme, typically implemented as synchronous Stochastic Gradient Descent (Sync-SGD), all workers operate in a lock-step fashion. The process generally follows these steps (Figure 2.12).

1. **Broadcast:** A central entity, often a parameter server or a designated worker, broadcasts the current model parameters (w_t) to all N workers.
2. **Compute the gradients:** Each worker i computes the gradient $\nabla f_i(w_t)$ using its local data batch and the exact same parameter version w_t .

3. **Communicate and aggregate:** Workers send their computed gradients back to the central entity. Crucially, the system waits until all workers have reported their gradients.
4. **Update:** The central entity aggregates the gradients, commonly by averaging $\frac{1}{N} \sum_{i=1}^N \nabla f_i(w_t)$, and applies the update to the model parameters $w_{t+1} = w_t - \eta \left(\frac{1}{N} \sum_{i=1}^N \nabla f_i(w_t) \right)$, where η is the learning rate.
5. **Repeat:** Steps 1 to 4 are repeated using the updated parameters w_{t+1} .

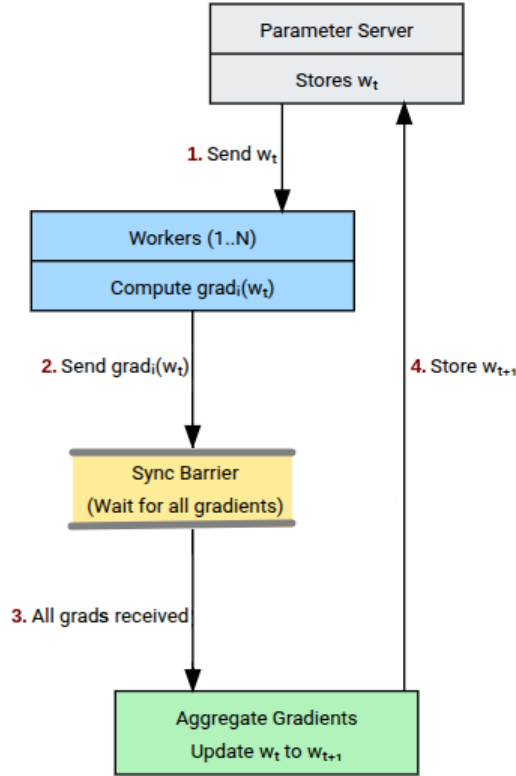


Figure 2.12: Workflow of synchronous SGD. Workers compute gradients based on the same model version (w_t) and wait at a synchronization barrier before the aggregated gradient updates the central model [112].

2.8.2 Advantages of Synchronous Parameter Updates

The first advantage of updating the model's parameters synchronously is **algorithmic simplicity**. The behavior of synchronous SGD closely mimics standard mini-batch SGD performed on a single machine, just with a much larger effective batch size, correspondent to the sum of mini-batches across all workers. Theoretical convergence properties often directly result from sequential SGD **convergence analysis**, making it easier to reason about convergence rates and hyperparameter tuning. All gradient calculations within a single update step use an identical model state, ensuring **consistent updates** based on the most current aggregated information. Given the same initial state, data partitioning, and random seeds, synchronous training runs are deterministic and **reproducible**.

2.8.3 Disadvantages of Synchronous Parameter Updates

The pace of the entire system is dictated by the slowest worker in each iteration (**Straggler problem**). If one worker is slow due to hardware variations, network issues, or encountering a particularly complex data batch, all other workers must stay idle, waiting for it to finish. This significantly reduces overall throughput and hardware utilization.

The barrier that synchronizes worker's communication constitutes a **synchronization Overhead**. Coordinating many workers simultaneously can lead to communication bottlenecks, especially as the number of workers increases.

2.8.4 Asynchronous Parameter Updates

Asynchronous update schemes, such as Asynchronous Stochastic Gradient Descent (Async-SGD), prioritize worker utilization and system throughput by removing the synchronization barrier. Workers operate more independently through the following steps.

1. **Pull parameters:** Worker i requests the latest available parameters, say w_{τ_i} , from the parameter server. τ_i represents the version or timestamp of the parameters received by worker i .
2. **Compute gradient:** The worker computes the gradient $\nabla f_i(w_{\tau_i})$ using its local data and the parameters w_{τ_i} it received.
3. **Push update:** The worker sends its computed update $(-\eta \nabla f_i(w_{\tau_i}))$ back to the parameter server.
4. **The sever applies update:** The parameter server receives the update from worker i and applies it immediately to its current parameter version, $w_{new} = w_{current} - \eta \nabla f_i(w_{\tau_i})$. Note that $w_{current}$ may already incorporate updates from other workers that arrived after worker i pulled w_{τ_i} .
5. **Repeat:** The worker immediately starts the next iteration over steps 1 to 4 by pulling the newest parameters, which might already reflect its own previous update and updates from others.

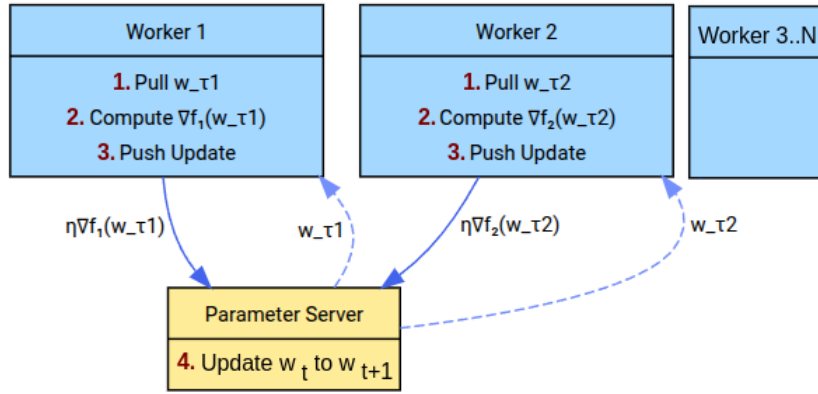


Figure 2.13: Workflow of asynchronous SGD. Workers pull parameters, compute gradients, and push updates independently without waiting. Updates are applied immediately by the Parameter Server, potentially based on stale parameters w_{τ} [112].

2.8.5 Advantages of Asynchronous Parameter Updates

Since workers do not wait for each other, asynchronous updates allow for an **higher throughput**. Faster workers can perform more updates in the same amount of wall-clock time, leading to potentially faster initial progress. Asynchronous updates eliminates idle time caused by waiting at synchronization barriers, especially beneficial in heterogeneous environments, which corresponds to an **improved hardware utilization**. Slow workers do not block faster ones, mitigating the Straggler problem inherent in synchronous methods, which we call **Straggler tolerance**.

2.8.6 Disadvantages of Asynchronous Parameter Updates

Gradient staleness is the defining challenge. By the time the update based on w_{τ_i} from worker i is applied at the server (at time t), the server's parameters may have already been updated several times by other workers. The gradient $\nabla f_i(w_{\tau_i})$ is thus "stale" relative to the parameters w_t it is being applied to. The staleness, often measured as $t - \tau_i$, increases when more workers are considered and when there is a higher communication latency.

Stale gradients induce **convergence issues** in the optimization process. Updates might be based on outdated information, potentially leading to suboptimal convergence paths, oscillations, or even divergence if not managed carefully, for example, by reducing the learning rate. While convergence can often be proven under certain assumptions, like bounded staleness, the statistical efficiency, defined as the convergence per unit of computation/gradient evaluations, might be lower than in synchronous SGD.

The non-deterministic nature of update timing makes **debugging and reproducing** specific training runs extremely difficult.

Managing concurrent access and updates to the parameter server requires a more **complex implementation** to ensure correctness and avoid race conditions, although mature deep learning frameworks often handle these issues.

2.8.7 The Staleness Problem in Asynchronous Updates

The core issue in asynchronous SGD is that a gradient $\nabla f_i(w_{\tau_i})$ computed by worker i based on parameters w_{τ_i} (parameters from time τ_i) is used to update the parameters w_t currently held by the parameter server at a later time t . The difference $w_t - w_{\tau_i}$ represents the changes made by other workers during the interval $(\tau_i, t]$.

If this difference is large, meaning a high staleness, the gradient $\nabla f_i(w_{\tau_i})$ might be a poor approximation of the gradient at the current point w_t , which is $\nabla f_i(w_t)$. This discrepancy can hinder convergence. Theoretical analyses often show that asynchronous SGD can converge if the learning rate is sufficiently small and the staleness is bounded, but the required learning rate might be smaller than in synchronous SGD, potentially slowing convergence in terms of iteration count even if wall-clock time per iteration is faster.

2.8.8 Choosing Between Synchronous and Asynchronous Updates

The decision involves balancing system efficiency (throughput, utilization) with statistical efficiency (convergence quality and speed per iteration).

Feature	Synchronous SGD updates	Asynchronous SGD updates
Coordination	lock-step; barrier synchronization per iteration	independent operation; no waiting time
Gradient computation	gradients computed on the same model parameters (w_t)	gradients computed on potentially stale parameters (w_{τ})
Throughput	limited by slowest worker (Straggler effect)	higher potential; not blocked by slow workers
System utilization	workers may be idle while waiting	workers are generally kept busy
Communication	bursty; requires barrier synchronization	more continuous; no barrier is required
Convergence	simpler analysis; often more stable	more complex analysis; can be less stable or slower overall
Staleness	no gradient staleness within an iteration	gradient staleness is inherent
Reproducibility	easier to reproduce (given fixed seeds/data)	harder due to non-deterministic timing
Implementation	simpler barrier logic	requires careful handling of concurrent updates

Table 2.2: Advantages and disadvantages of synchronous and asynchronous parameter updates [112].

2.8.9 Practical Implications

High latency networks exacerbate the staleness problem in asynchronous SGD and increase waiting times in synchronous SGD. Low latency and high bandwidth networks favor synchronous SGD and make asynchronous SGD staleness less problematic. With a very large number of workers, synchronous SGD barriers become expensive, and asynchronous SGD staleness increases. In homogeneous clusters, the Straggler problem in synchronous SGD is less severe. Heterogeneous clusters often benefit asynchronous SGD's tolerance to varying worker speeds. Some models and tasks might be more sensitive to stale gradients than others.

While asynchronous SGD initially promised significant speedups, practical experience and research have shown that the detrimental effects of staleness can negate the throughput advantage, especially concerning final

model accuracy. Synchronous SGD, despite the Straggler issue, often provides more stable and predictable convergence, making it a common default choice, particularly when using efficient communication protocols like AllReduce. Hybrid approaches like Stale Synchronous Parallel, which allow a bounded amount of staleness, attempt to find a middle ground. Ultimately, the optimal choice depends heavily on the specific hardware environment, network conditions, and the characteristics of the learning task at hand.

2.9 Strong and Weak Scaling

Strong scaling and weak scaling are two fundamental concepts in parallel computing used to measure how effectively a parallel program utilizes increasing computational resources. Speedup is a way to measure the impact of the parallelism.

Strong scaling refers to the efficiency of processing a workload of fixed size with increasing numbers of workers. The goal is to solve a problem faster by reducing the execution time, and the key idea is that the total amount of work remains constant, but it is distributed across more workers, meaning the workload per worker decreases. The ideal result is obtaining an execution time that is inversely proportional to the number of workers. For example, doubling the workers should ideally halve the execution time, which correspond to a linear speedup (Figure 2.14 (i)). The governing law of strong parallelism is often analyzed using the Amdahl's Law, which indicates that the speedup is ultimately limited by the fraction of the workload that cannot be parallelized. Amdahl's law can be formulated as $\text{speedup} = \frac{1}{s + p/N}$, where s is the proportion of execution time spent on the sequential part, p is the proportion of execution time spent on the part that can be parallelized, and N is the number of workers.

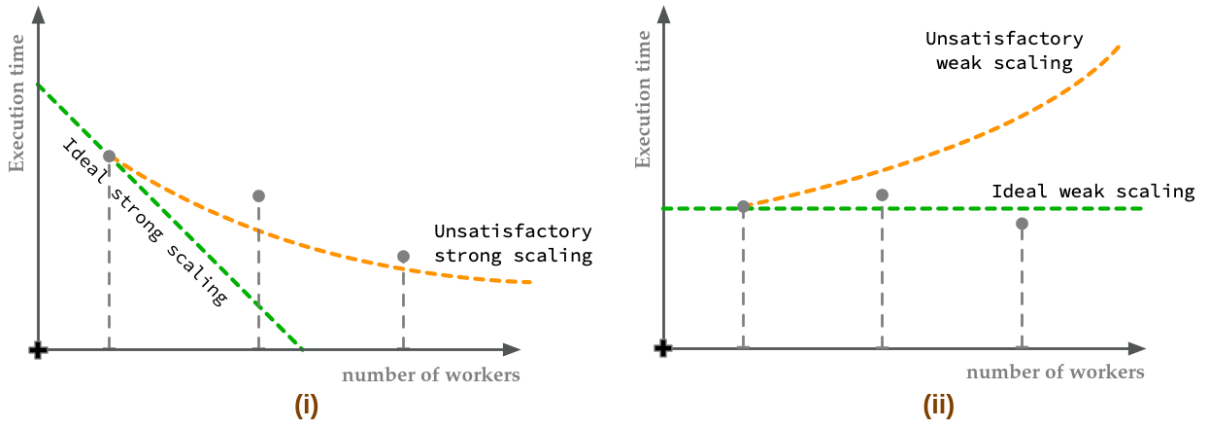


Figure 2.14: Real and ideal speedups of strong and weak scalings. (i) Ideally in strong scaling, the execution time would decrease linearly on a log-log plot as the number of workers increases. If the parallelization was perfect, the slope would be -1 . In this example, the task does not have good strong scaling performance, but we have not tried it with enough workers to find the point where adding new workers results in no additional speedup. (ii) Ideally in weak scaling, the execution time would remain constant if we simultaneously and proportionally scale the problem size and the number of workers. In this example, weak scaling does not increase the execution time, so this task has good weak scaling performance. This means that a larger problem can be solved in about the same amount of time as a smaller problem as long as the number of workers increases proportionally.

Weak scaling measures the change in execution time when both the problem size and the number of workers are increased proportionally, which means that the size of the work assigned to each worker is kept fixed. The objective is to solve a larger problem in the same amount of time, or maintain efficiency. The main idea is that the amount of work assigned to each worker remains constant, but the total work done by the system increases with the number of workers. The ideal result is to maintain the execution time constant, regardless of the increase in the number of workers and the total problem size (Figure 2.14 (ii)). The governing law of weak parallelism is analyzed using Gustafson's Law, which relates the speedup to the sequential fraction of the workload on a single worker, assuming the work per worker is constant. The Gustafson's law is based on the approximations that the parallel part scales linearly with the amount of resources, and that the sequential part does not increase with respect to the size of the problem. It provides the formula for scaled speedup, $\text{scaled speedup} = s + p \times N$, where s , p and N have the same meaning as in Amdahl's law. The fundamental questions answered by strong and weak scalings are illustrated in Figure 2.15.

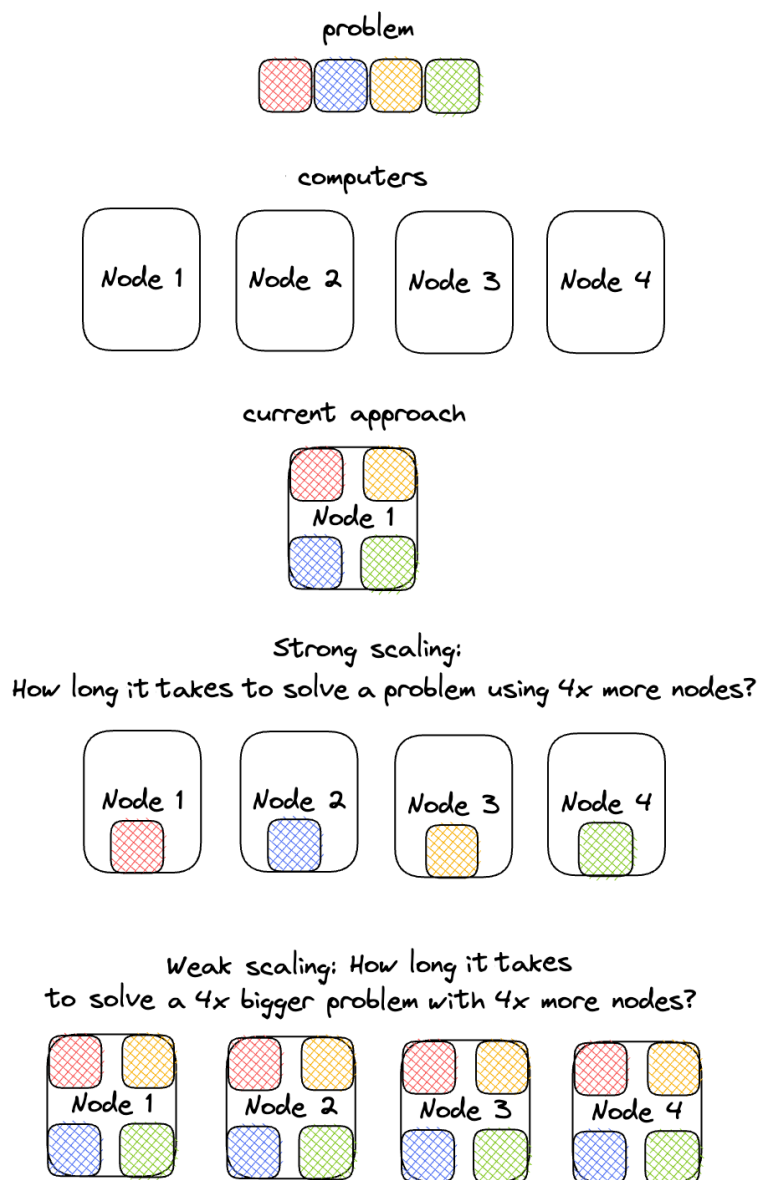


Figure 2.15: The fundamental questions answered by strong and weak scaling [14].

2.10 Hardware Considerations

The main hardware characteristics that govern distributed training are the following.

Memory capacity: The total memory footprint of training, including the model state, gradients, optimizer states and activations, frequently exceeds the High Bandwidth Memory (HBM) available on a single accelerator. Models larger than the memory capacity of one GPU mandate model parallelism.

Interconnect bandwidth: The speed at which accelerators can communicate with each other directly dictates the feasibility of parallel architectures. High-bandwidth paths permit communication to occur efficiently alongside computation. Intra-node connections like NVIDIA NVLink provide massive data transfer rates for high-frequency synchronization like tensor parallelism. Inter-node slower network fabrics, like InfiniBand or RoCE, can bottleneck distributed training and favor coarser communication methods like pipeline parallelism.

Computation vs. communication ratio: Modern accelerators, like NVIDIA H100 and H200, execute computational kernels much faster than they transfer data. At sufficiently large scales, or with small batch sizes, this asymmetry causes systems to become communication-bound rather than compute-bound.

Hardware topology: 2D and 3D parallel strategies are structured hierarchically to map onto cluster networks.

For example, a cluster might run tensor parallelism over fast intra-node links, while using pipeline parallelism and data parallelism across slower cross-node networks.

Parallelism Strategies and Hardware Mapping

Data parallelism divides the training batch across devices and synchronizes gradients. It requires high inter-node network bandwidth, such as InfiniBand or RoCE, and large batch sizes to hide synchronization costs. Fully sharded data parallelism significantly reduces the memory footprint per GPU but increases network communication frequency.

Tensor parallelism shards individual layers and matrix multiplications across multiple accelerators. It requires ultra-high-speed intra-node interconnects, such as NVLink. Slow interconnects introduce severe latency during collective operations like AllReduce.

Pipeline parallelism splits the model vertically through sequential layer groups. It has low demand on inter-chip bandwidth but requires that accelerators to wait for preceding layers to compute. To minimize the resulting pipeline bubbles, hardware with fast point-to-point transfers and fine-tuned scheduling is recommended.

Expert parallelism routes different tokens in a sequence to specialized, sparse sub-networks, namely Mixture of Experts. It demands massive All-to-All communication. High-bandwidth fabrics, like NVSwitch, are critical for routing tokens without bottlenecking the network.

It is important to keep in mind the hardware systems where we train the models. Normal GPU clusters used for training consist of multiple compute nodes that are connected using either fast Ethernet or a specialized communication backend like InfiniBand. Each compute node will contain multiple GPUs. The GPUs communicate with the CPU and CPU RAM via PCIe. The GPUs within a single compute node are commonly connected via a fast interconnect like NVIDIA's NVLink (Figure 2.16).

It is important to keep in mind this hierarchy when evaluating different training distribution schemes since GPUs within the same compute node can communicate much faster than GPUs located on different nodes. As a concrete example, BLOOM was trained on 48 compute nodes, with 8 GPUs each. Rough estimates for the bandwidth at each level was (i) in NVLink (GPU to GPU), it was approximately 450GB/s up and down, accounting for a total 900GB/s of bandwidth, (ii) in PCIe (GPU memory to CPU RAM), it was approximately 25GB/s up and down, for a 16× PCIe 4.0 connection, and (iii) in Ethernet, for node to node communication within the same data center, it was approximately 10GB/s. An alternative to Ethernet is InfiniBand, which is faster at around 50GB/s. For NVLink and PCIe, the bandwidth scales linearly with the number of lanes connected.

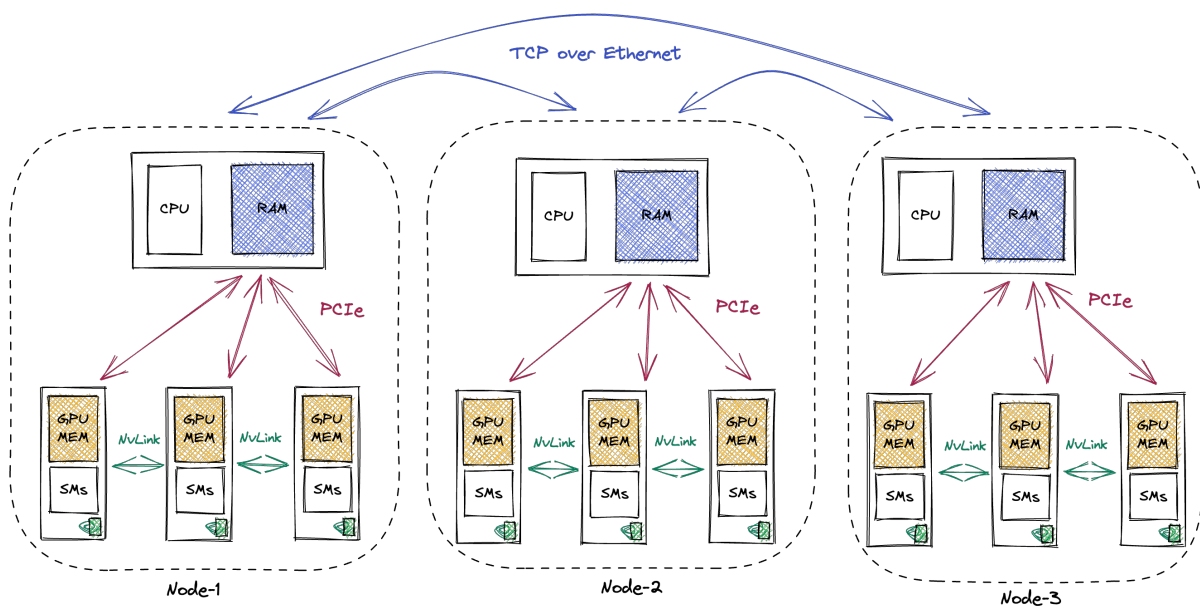


Figure 2.16: A generic architecture of a distributed computing system [14].

2.11 Conclusions

In this chapter, we explored the fundamental principles and practical aspects of distributed training for machine learning. As models and datasets continue to grow in size and complexity, distributed computing has become an essential tool for achieving feasible training times and scalable performance. We began by introducing the concept of distributed systems and motivating their use in machine learning through the rapid expansion of model sizes, data volumes, and computational capabilities. This naturally led to the need for parallel and distributed training strategies that leverage multiple GPUs and compute nodes.

Key concepts such as nodes, ranks, world size, and process groups were presented to establish a clear understanding of how distributed environments are structured and managed. We also examined communication backends and initialization methods, which enable efficient coordination and data exchange between distributed processes. A significant portion of the chapter focused on communication mechanisms, including collective operations such as broadcast, AllReduce, gather, and scatter, as well as point-to-point communication and synchronization primitives. These operations form the backbone of distributed computation, allowing processes to exchange and aggregate information efficiently.

We then discussed different parallelism strategies, data parallelism, pipeline parallelism, and tensor parallelism, and provided guidance on selecting the most appropriate approach based on model characteristics, hardware constraints, and performance goals. The importance of hybrid strategies for large-scale systems was also highlighted. The chapter further examined the trade-offs between synchronous and asynchronous parameter updates, emphasizing the balance between system throughput, convergence stability, and implementation complexity. While asynchronous methods can improve hardware utilization, synchronous training often provides more stable and predictable convergence in practice.

Finally, we covered scaling concepts, including strong and weak scaling, and analyzed how hardware factors such as memory capacity, interconnect bandwidth, and system topology influence the efficiency of distributed training systems. In conclusion, effective distributed training requires careful consideration of both algorithmic design and system architecture. By understanding the interplay between communication, computation, and hardware, practitioners can design scalable training pipelines that fully exploit modern computing resources.

Chapter 3

Data Parallelism

Contents

3.1	Distributed data parallelism	30
3.2	Parameter server	42
3.3	AllReduce distributed training	56
3.4	Tree AllReduce	59
3.5	Two-tree AllReduce	60
3.6	Performance impact	62
3.7	Zero redundancy optimizer parallelism	64
3.8	Fully sharded data parallel	70
3.9	Conclusions	82

This chapter presents the fundamental approaches for scaling machine learning training through distributed systems, with a focus on data parallelism and parameter server architectures. It begins by introducing distributed data parallel training, explaining how models are replicated across multiple devices while datasets are partitioned, and how gradients are synchronized using collective communication operations such as AllReduce. The chapter also introduces the parameter server paradigm as an alternative approach for distributed training across multiple machines. In this model, a central server maintains the most up-to-date model weights, while worker nodes compute gradients and communicate with the server, often asynchronously. The discussion highlights different execution strategies, including synchronous AllReduce-based methods and asynchronous training, as well as their trade-offs in terms of scalability, fault tolerance, and convergence behavior. Building on these foundations, the chapter explores practical implementations of distributed training frameworks, particularly PyTorch's Distributed Data Parallel and RPC-based parameter server setups. It examines key optimization techniques such as gradient bucketing, overlapping computation and communication, and efficient synchronization mechanisms. In addition, it covers large-scale distributed optimization methods, including asynchronous stochastic gradient descent, Downpour SGD, and Sandblaster L-BFGS, which leverage the parameter server architecture. It also analyzes the role of communication algorithms, notably ring, tree, and two-tree AllReduce, and evaluates their performance in terms of bandwidth and latency. It then presents advanced memory optimization strategies, including the Zero Redundancy Optimizer and Fully Sharded Data Parallel, which reduce memory usage by partitioning model states, gradients, and weights across devices. Therefore, the chapter provides both theoretical insights and practical guidance for designing efficient, scalable distributed training systems. It highlights the key trade-offs between synchronous and asynchronous execution, centralized versus decentralized coordination, and the balance between computation, communication, and memory efficiency in large-scale machine learning.

3.1 Distributed Data Parallelism

In modern deep learning, because the dataset is too big to be fit into the computing device memory, we could only do stochastic gradient descent over mini-batches (Figure 3.1). This is the idea behind distributed data parallel training, where the entire model is deployed to multiple nodes and the data is sharded, that is, it is horizontally split. Thus, each instance of the model works on a part of the data. For example, if we have 100k data points in the training dataset, every time we try to update the model we only use for example a micro-batch with 256 data points to calculate an estimate of the gradients in each node, otherwise we ran out of memory. The shortcoming of stochastic gradient descent is that the estimate of the gradients might not accurately represent the true gradients that are obtained with the full dataset. Therefore, model training may take longer to converge. A natural way to have a more accurate estimate of the gradients is to use a larger mini-batch size or even use the full dataset. To allow this, the gradients of micro-batches were calculated on each node, the final estimate of the gradients is the weighted average of the gradients calculated from all the micro-batches.

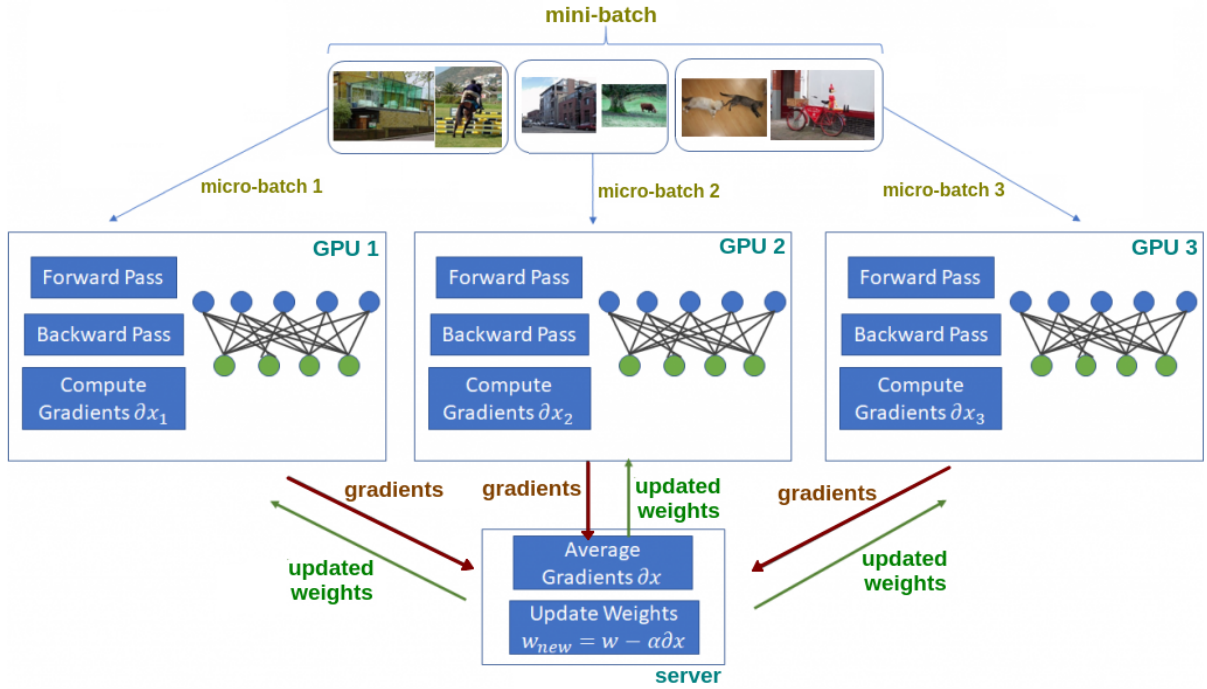


Figure 3.1: Data parallel distributed training with stochastic gradient descent.

Mathematically, data parallelism is valid because Equation 3.1 holds.

$$\begin{aligned}
 \frac{\partial Loss}{\partial w} &= \frac{\partial [\frac{1}{n} \sum_{i=1}^n f(x_i, y_i)]}{\partial w} \\
 &= \frac{1}{n} \sum_{i=1}^n \frac{\partial f(x_i, y_i)}{\partial w} \\
 &= \frac{m_1}{n} \frac{\partial [\frac{1}{m_1} \sum_{i=1}^{m_1} f(x_i, y_i)]}{\partial w} + \frac{m_2}{n} \frac{\partial [\frac{1}{m_2} \sum_{i=m_1+1}^{m_1+m_2} f(x_i, y_i)]}{\partial w} + \dots + \frac{m_k}{n} \frac{\partial [\frac{1}{m_k} \sum_{i=m_1+m_2+\dots+m_{k-1}+1}^{m_1+m_2+\dots+m_k} f(x_i, y_i)]}{\partial w} \\
 &= \frac{m_1}{n} \frac{\partial l_1}{\partial w} + \frac{m_2}{n} \frac{\partial l_2}{\partial w} + \dots + \frac{m_k}{n} \frac{\partial l_k}{\partial w}
 \end{aligned} \tag{3.1}$$

Where w are the model weights, $\frac{\partial Loss}{\partial w}$ is the true loss gradient relative to the weights calculated on a mini-batch of size n , $\frac{\partial l_k}{\partial w}$ is the gradient calculated on a micro-batch in node k , x_i and y_i are the features and labels of data point i , $f(x_i, y_i)$ is the loss for data point i calculated from the forward propagation, n is the total number of data points in the mini-batch, k is the number of nodes used in distributed training, m_k is the number of data points

assigned to node k , and $m_1 + m_2 + \dots + m_k = n$. When $m_1 = m_2 = \dots = m_k = \frac{n}{k}$, we could further express the loss gradient as Equation 3.2.

$$\frac{\partial Loss}{\partial w} = \frac{1}{k} \left[\frac{\partial l_1}{\partial w} + \frac{\partial l_2}{\partial w} + \dots + \frac{\partial l_k}{\partial w} \right] \quad (3.2)$$

In data parallel training, each node trains a copy of the model using its own portion of the data. Thus, each node runs forward propagation with the same model weights. Next, we summarize a general flow adopted by data parallelism to update the model weights.

1. Each node gets a micro-batch of different data from a common mini-batch.
2. Each node computes the gradients of the weights using its own micro-batch of data.
3. Each node sends its gradients back to the main node.
4. Once the main node got the gradients from all k nodes, which means there is an implicit synchronization here, calculates the average of the gradients received from all k nodes.
5. The model weights are updated using the averaged gradients, as in standard stochastic gradient descent.
6. The updated weights are then broadcast to all the nodes so that we can continue training on the next mini-batch of data.
7. Steps 1 to 6 are repeated for a specified number of mini-batches or until convergence is reached.

It is worth noting that there are different ways to implement data parallelism in distributed training, and the specific details of how weights are updated may vary depending on the framework and algorithm used. However, the idea of using gradient averaging to synchronize the model weights across node remains a common approach.

Data parallelism allows for training a model faster by replicating this model among multiple compute nodes, and dividing the dataset among them. Data parallelism works particularly well for models that are very parameter efficient, like CNNs. The term efficiency refers to a high ratio between floating-point operations and forward passes or number of weights. To simplify things, in this section we will consider only sequential models, where the output is given by $Dout = Layer_N(Layer_{N-1}(\dots(Layer_1(Din))))$.

Figure 3.2 illustrates the functional building blocks that we will use for understanding how parallel training works. Each block takes the inputs on the left and transforms them into the outputs on the right. The purpose of the cache block is to store its input data until it is needed again [13].

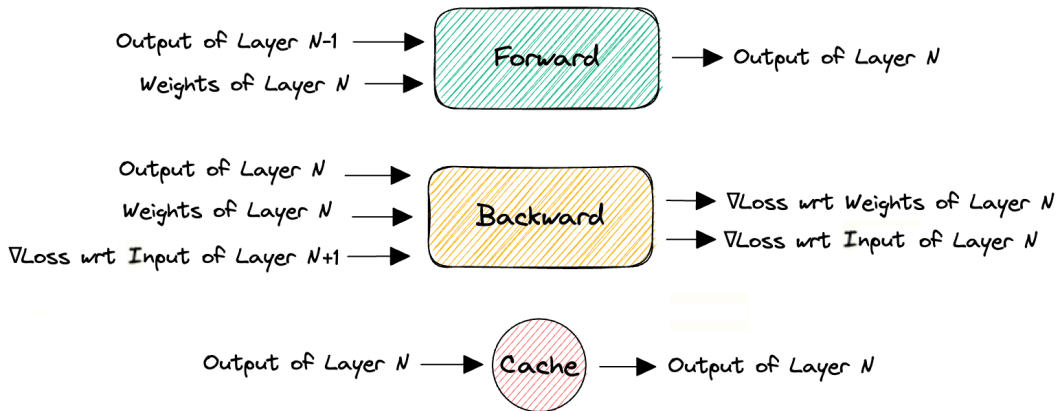


Figure 3.2: The building blocks used to represent the forward and backpropagation during model training [13].

After having run backpropagation to compute the gradients, we use the optimizer to update the weights, and we represent this optimization task with the block included in Figure 3.3.

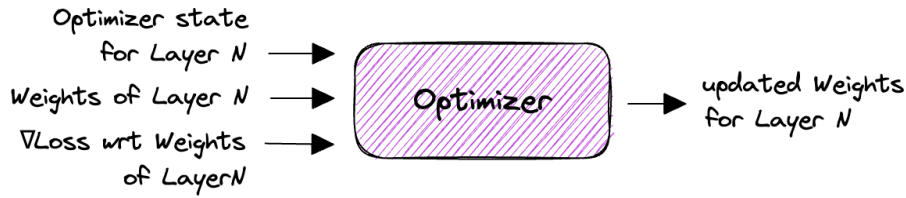


Figure 3.3: The building block that represents the weights' optimization [13].

In general, the state associated to model training consists of the following types of information: (1) layer weights, (2) cached layer outputs, also called activations, (3) gradients with respect to the weights, often just called gradients, (4) gradients with respect to the layer inputs, often called errors, and (5) the optimizer state.

When we put together the building blocks for the forward, backpropagation and cache of each layer, we end up with a visual representation for the model computation graph that is called **pebble graph**. The pebble graph depicted in Figure 3.4 illustrates how the cached activations are built up during the forward pass (in green) and discarded once the corresponding backward pass is run (in red). We see that the gradients with respect to each layer's weights w_i are available after having run the backwards pass for a given layer (in purple). The graph also shows the gradients with respect to the layer inputs (in brown). The figure is also annotated with the "time step" in which every part of the data is produced and consumed.

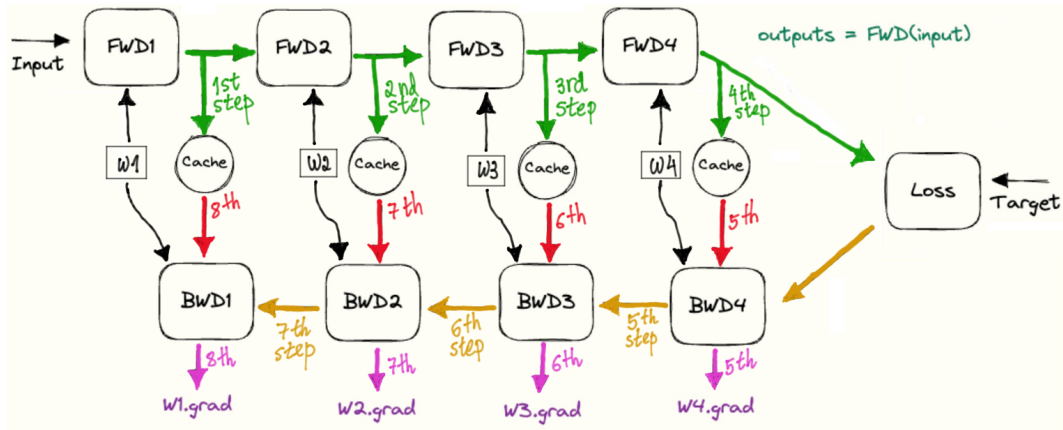


Figure 3.4: Example of a pebble graph for a model with four layers [13].

We depicted backpropagation in Figure 3.4 as it occurs in sequential training, where we have a single compute node, which has the complete model loaded into memory. During each iteration of training, we load the next minibatch and perform a forward pass through the model while caching each layer's outputs. Then, we calculate the loss and run the backward pass, which calculates the gradients. This process is illustrated in a simplified form in Figure 3.5, using MNIST images as input data and a minibatch size of 6.

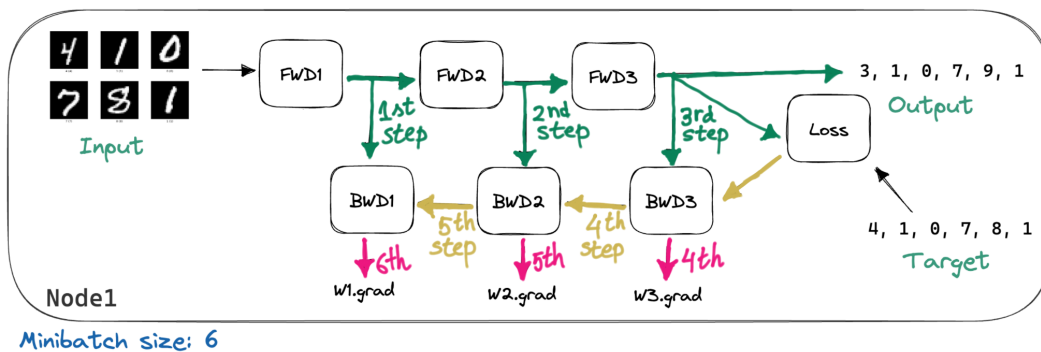


Figure 3.5: Forward and backpropagation steps of sequential training with MNIST images [13].

Data parallelism works by duplicating the model across N machines, splitting each minibatch into N chunks, and letting each machine process a distinct chunk. By splitting the data across multiple nodes, two nodes in Figure 3.6, there is less work to do for each node, and, if we neglect communication overhead, training should be $2\times$ faster. The samples in each minibatch can be independently processed, except in batch normalization layers, hence communication is not required during the forward pass, which calculates the output for each sample, nor during the backward pass, which calculates the gradient of a single sample's loss with respect to the weights.

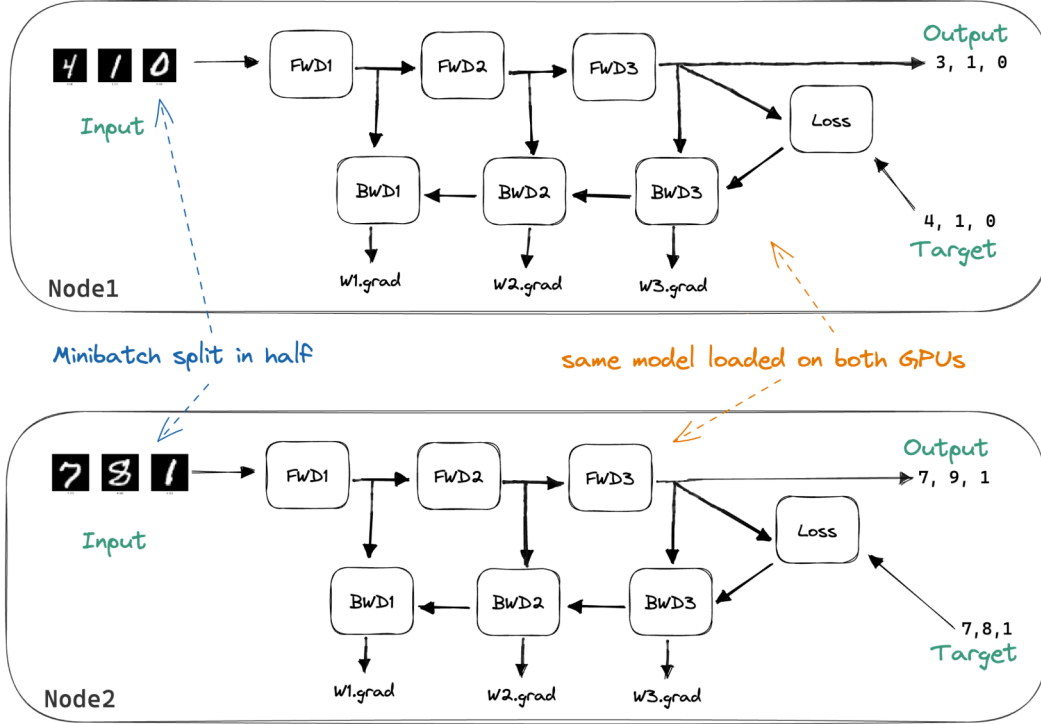


Figure 3.6: Pebble graph relative to data parallel training with two nodes [13].

To achieve sequential consistency, it is necessary to synchronize the gradients before updating the model weights. We consider that a distributed algorithm is sequentially consistent if the resulting gradients are the same as if we had calculated them using sequential training on a single machine. The most commonly used loss functions include a mean error calculated across all samples of the minibatch, as occurs in Equation 3.3.

$$\text{loss}(\text{minibatch}) = \frac{1}{N} \sum_{i=0}^{\text{minibatchSize}} \text{error}(\text{output}_i, \text{target}_i) \quad (3.3)$$

The gradient of a sum is the sum of the gradient of each term. Thus, we can calculate the gradients of the samples independently on each machine and sum them up before performing the weights' update. If we are using stochastic gradient descent (SGD), there is no difference between synchronizing the weights or the gradients, because $\frac{1}{N} \sum_{i=1}^N (W + \lambda \nabla W_i) = W + \frac{\lambda}{N} \sum_{i=1}^N \nabla W_i$. However, this equivalence does not hold when a stateful optimizer, like Adam, is being used because updating the state is a non-linear function of the gradients. If we use Adam and synchronize the weights instead of the gradients, the optimizer states on each node diverge and we lose the sequential consistency. After the synchronization, we want the gradient on each node to be the same and its value is given by Equation 3.4.

$$\nabla^{\text{synchronized}} W = \frac{1}{\text{NumberNodes}} \sum_{i=0}^{\text{NumberNodes}} \nabla W_i \quad (3.4)$$

Once the synchronization is complete, we can perform the weight update and update the optimizer state. Summing up the distributed gradients and making the sum available on every node is achieved using the `MPI.AllReduce` operation.

Conceptually, data-parallel training is sequentially consistent. However, this does not mean that we get equal outputs between sequential training and data parallel training in practice. To sum up the gradients, we need to use the MPI AllReduce operation, which collects the individual results from each node, computes the reduction from all of them which, in our case, is the sum of the gradients obtained for each chunk into which the minibatch was divided, and communicates the result back to all nodes. AllReduce chooses the ordering for summing up the gradients for us, for example, $(\nabla_1 + \nabla_2) + \nabla_3$ or $\nabla_1 + (\nabla_2 + \nabla_3)$. This would not be a problem if the summation was commutative and associative. Unfortunately, floating-point math is not associative, hence the result will not be exactly the same as in sequential training. In practice, the difference between the expected gradients from sequential training versus the gradients from data-parallel training is small enough that we can just ignore this issue. However, it is good to keep in mind that the gradients are not exactly the same, and debugging is more difficult than in sequential training.

Let us now briefly discuss AllReduce as used in data-parallel training. We will cover AllReduce in more detail in Section 3.3. The MPI specification guarantees that the result of AllReduce is exactly the same on every node. This is not straightforward, due to the non-associativity of floating-point operations. Hence we have to make sure that local reductions happen in the same order. For example, the naive algorithm illustrated in Figure 3.7 would not fulfill the MPI specification, since the local sums happen in different orders.

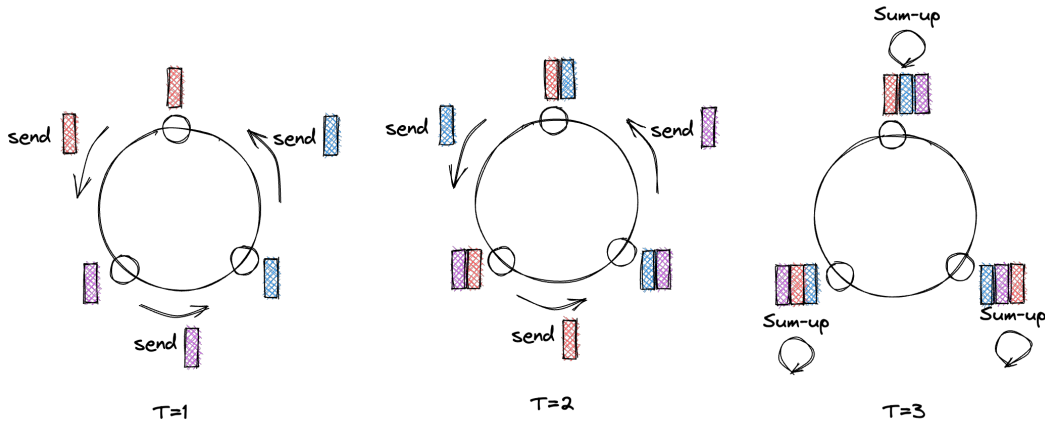


Figure 3.7: Ring AllReduce animation that explains how the reduction operation is implemented in a three-nodes setup [13].

Although we are not going to discuss AllReduce in detail, let us briefly address the bandwidth and latency of Ring AllReduce, one of the most common algorithms for implementing AllReduce. In Ring AllReduce, in each communication between nodes, a node sends $\frac{\text{NumberParameters}}{\text{NumberNodes}}$ values to its two neighbors, where the division by `NumberNodes` is a consequence of each node being responsible for aggregating only a parcel of the total amount of parameter gradients. Thus, considering all AllReduce transactions, each node needs to transfer $2(\text{NumberNodes} - 1) \frac{\text{NumberParameters}}{\text{NumberNodes}}$ floats between itself and its two neighbors in the ring. Discarding the $\frac{1}{\text{NumberNodes}}\%$ inaccuracy, the size of data transferred does not depend on the number of nodes. Further, Ring AllReduce is bandwidth optimal, meaning there is no way to achieve the same task and to transfer less data. This allows us to estimate that the time spent on communication is greater than or equal to $\frac{\text{model size}}{\text{bandwidth}}$, considering a full-duplex connection that ensures that, during Ring AllReduce, each node is sending and receiving data simultaneously.

Again for Ring AllReduce, we need to perform $(\text{NumberNodes} - 1)$ `MPI.ReduceScatter` operations, and $(\text{NumberNodes} - 1)$ `MPI.AllGather` operations. This means that while adding more nodes does not require bigger data transfer, it does require linearly more communication rounds. How much that impacts the overall runtime will depend on the latency between the nodes.

We can visualize how AllReduce is used in Figure 3.8. Here, once the gradients are calculated, AllReduce is performed using all nodes in our `MPI.Communicator`.

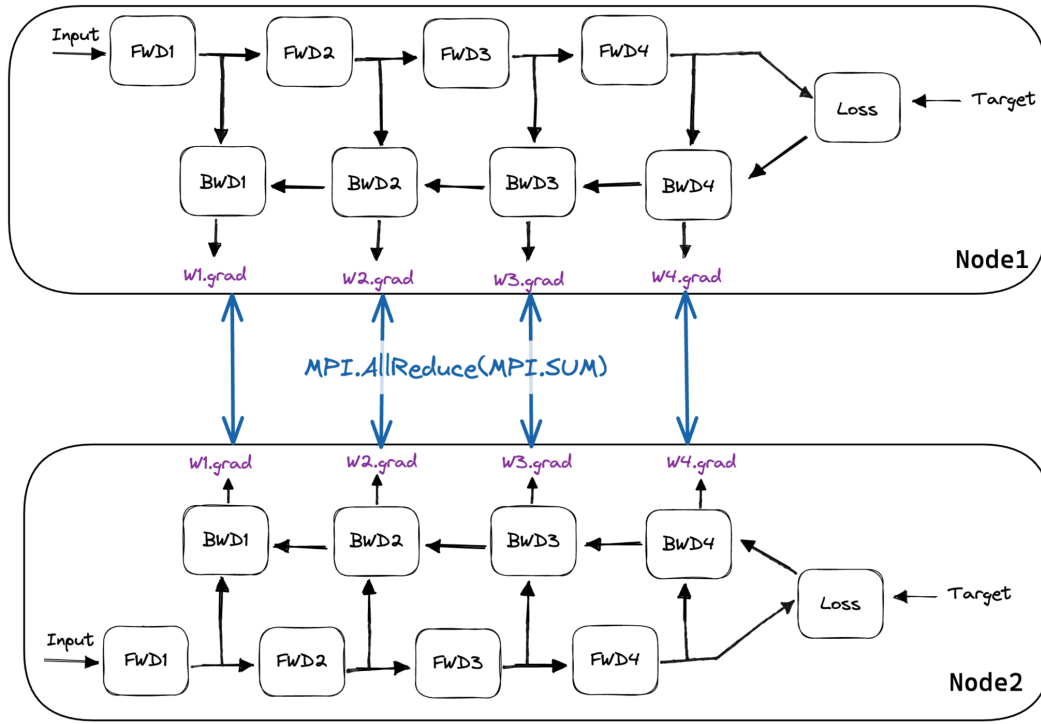


Figure 3.8: Apply AllReduce to combine gradients from all nodes, two nodes in this case [13].

The straightforward way of implementing data parallel distributed training is to run a full forward and backward pass, and before calling `optimizer.step()`, syncing the gradients. In PyTorch, syncing the gradients can be as simple as the two lined of code included in Listing 3.1.

```
1 for param in model.parameters():
2     comm.Allreduce(MPI.IN_PLACE, param.grad, op=MPI.SUM)
```

Listing 3.1: Synchronizing the gradients of all model's weights calculated by multiple nodes in data parallel training.

This implementation is suboptimal since it divides our training into two stages. During the first stage, forward and backward passes, the network does nothing and is waiting for the processors to finish computing. During the second stage, AllReduce, the network is communicating as fast as possible, while the processors are idle. We can visualize this type of training in Figure 3.9.

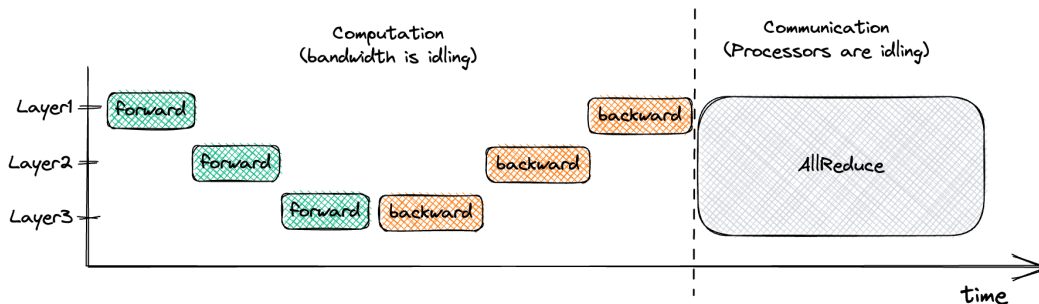


Figure 3.9: Running parallel training without considering overlap between computation and communication [13].

However, Figure 3.9 shows that, for example, the gradients of the third layer are available once we have performed the backward pass through this third layer. If we start a non-blocking AllReduce for third layer's parameter gradients as soon as they are ready, our network will be busy doing useful work while our processors are independently calculating the gradients for the second layer. This strategy, called interleaving of communication and computation, allows us to optimize parallel training. We can visualize the interleaved implementation of data parallel training in Figure 3.10.

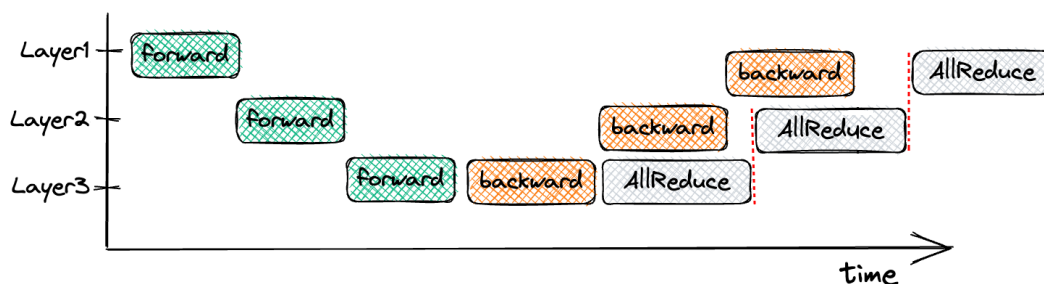


Figure 3.10: Running parallel training using overlap between computation and communication [13].

In conclusion, data parallel training works particularly well when (i) our network is parameter efficient, since each minibatch requires sending all model weights over the network and (ii) the minibatch size is large. The minibatch size is an upper limit to the degree of data parallelism. Small minibatch sizes mean small inputs, which decreases the operational intensity of matrix multiplication, and that will lead to inefficient computation.

3.1.1 Distributed Data Parallel in PyTorch

PyTorch's Distributed Data Parallel (DDP) enables efficient training by running multiple replicas of the same model across different processes, typically with one process per GPU. At the start of training, each process initializes an identical copy of the model with the same weights. This ensures that distributed training remains equivalent to training on a single machine.

Once initialized, the dataset is partitioned across processes so that each process works on a different subset of data. During each iteration, every process independently executes a forward pass on its local mini-batch, followed by a backward pass to compute gradients. Because each process sees different data, the resulting gradients differ across replicas.

The key operation in DDP occurs during the backward pass. As gradients are computed, DDP automatically intercepts them, through the specified hooks, and coordinates their synchronization across processes. Rather than waiting for the entire backward pass to complete, gradients are grouped into buckets and communicated as soon as they are ready. This allows communication to overlap with computation, improving efficiency.

Gradient synchronization is performed using the **AllReduce** operation, which aggregates gradients, typically by averaging, across all processes and redistributes the resulting gradients. After this step, all model's replicas share the same gradients. Each process then independently updates its model using the optimizer, ensuring that all replicas remain synchronized.

The training process in DDP can be summarized as follows.

1. Initialization: Multiple processes are launched, each loading the same model architecture and identical weights.
2. Data distribution: The dataset is split across processes, typically using a `DistributedSampler`.
3. Forward pass: Each process performs local computation independently, without communication.
4. Backward pass: Gradients are computed locally via backpropagation.
5. Gradient synchronization: Gradients are grouped into buckets and synchronized using `AllReduce`.
6. Optimizer step: Each process updates its model using the synchronized gradients.
7. Repeat: The process repeats steps 1 to 6 for the subsequent training iterations.

The performance of DDP relies on three key optimization strategies: gradient bucketing, overlapping computation with communication, and skipping synchronization.

Gradient Bucketing

Neural networks produce many small gradient tensors, and communicating them individually incurs significant overhead. DDP addresses this by grouping gradients into larger buffers called *buckets*. Once a bucket is filled, it is communicated using a single AllReduce operation. This reduces communication calls and improves bandwidth utilization.

Overlapping Computation and Communication

In a naive approach, communication begins only after the model's backward pass is complete, causing idle time. DDP avoids this by starting communication as soon as each bucket is ready, while the backward pass continues for remaining layers. This overlap hides communication latency and improves resource utilization.

Skipping Synchronization

Since synchronization can be expensive, DDP allows it to be skipped in certain scenarios, such as gradient accumulation. Using mechanisms like `no_sync()`, gradients can be accumulated locally over multiple iterations and synchronized only when necessary. This reduces communication frequency but must be used carefully to preserve convergence.

Hook Mechanism

The efficiency of DDP is enabled by its **hook mechanism**, which integrates with PyTorch's autograd system. During the backward pass, PyTorch traverses the computational graph in reverse order to compute gradients. DDP attaches an autograd hook to each parameter, triggering it as soon as the corresponding gradient is computed. This allows DDP to react immediately without modifying user code. Hooks support the above mentioned optimization techniques.

- Gradient bucketing: gradients are collected into buckets and an hook helps detecting when a given bucket is complete;
- Overlapping: hooks also help launching communication as soon as gradients are available;
- Skipping synchronization: hooks are also usefull to alter the gradient's synchronization behavior by accumulating gradients locally when needed.

This event-driven mechanism allows DDP to efficiently coordinate computation and communication.

Figure 3.11 contains the pseudo-code for the PyTorch's Distributed Data Parallel algorithm. The `constructor` method executes two main steps, broadcasting model states to all nodes and seting up autograd hooks. Calling `add_post_hook` is used to inform the autograd engine that it must execute the specified logic (`autograd_hook`) immediately after the specified parameter's gradient (`acc`) as been computed, in the backward pass. The `forward` method is a simple wrapper of the model's `forward` method, and is used for traversing the autograd graph and mark unused weights as ready. The `autograd_hook` takes the internal parameter index as input, which helps to find (i) the parameter tensor associated with this index and (ii) the bucket it belongsings to. It writes the local gradient to the correct offset in the bucket and then launches an asynchronous AllReduce operation. There is an additional finalizing step, omitted in the pseudo-code, that waits for AllReduce operations and writes the aggregated gradient value into the local gradients at the end of the backward pass.

Despite its advantages, DDP has some limitations, namely (i) the communication overhead increases with model size and number of GPUs, (ii) performance depends on the configuration parameters, such as the bucket size, and (iii) efficient data loading and process synchronization are required to achieve good performance. In conclusion, DDP accelerates training by optimizing multiple model replicas in parallel and synchronizing gradients after each backward pass. Through optimized communication strategies such as gradient bucketing, overlap of computation with communication, and selective synchronization, it achieves high efficiency and scalability while maintaining correctness.

```

Input: Process rank r, bucket capacity c, local model net
method constructor(net):
    if r=0 then
        Broadcast net states to other processes
        Initialize buckets, allocate weights to buckets
        in the reverse order they appear in net.parameters()
    for p in net.parameters() do
        acc = gradient accumulator of parameter p
        add_post_hook(acc, autograd_hook)
method forward(Din):
    Dout = net(Din)
    Traverse autograd graph starting at Dout and mark unused weights as ready
    return Dout
method autograd_hook(param_index):
    Get bucket bi and bucket offset using param_index
    Get parameter p that corresponds to param_index
    view ← bi.narrow(offset, p.size())
    view.copy_(p.grad)
    if all grads in bi are ready then
        Mark bi as ready
    Launch AllReduce on ready buckets in order
    if all buckets are ready then
        Block waiting for all AllReduce operations

```

Figure 3.11: Algorithm for PyTorch distributed data parallel training class [119].

3.1.2 Practice with Distributed Data Parallel Training

To demonstrate how distributed data parallel training works in practice we will consider the simple model defined in Listing 3.2.

```

1 class SimpleModel(nn.Module):
2     # A simple model to explore DDP training
3     def __init__(self):
4         super(SimpleModel, self).__init__()
5         self.net1 = nn.Linear(10, 10)
6         self.relu = nn.ReLU()
7         self.net2 = nn.Linear(10, 5)
8
9     def forward(self, x):
10        return self.net2(self.relu(self.net1(x)))

```

Listing 3.2: A simple neural network with three layers.

Integrating DDP into a standard PyTorch training requires a few modifications: (1) setup the distributed environment, (2) initialize the group of processes to use, (3) prepare a distributed data loader, (4) wrap the model with `DistributedDataParallel`, and (5) adjust the training loop.

1. Environment Setup

First, we have to launch multiple Python processes, one for each node/GPU. Standard tools like `torchrun` or the older `torch.distributed.launch` handle this. These tools set the necessary environment variables, such as `MASTER_ADDR`, `MASTER_PORT`, `RANK`, and `WORLD_SIZE`, which are needed for calling `init_process_group`. We also have to determine the `local_rank`, which typically corresponds to the GPU index the current process should use.


```

1 import torch
2 import torch.distributed as dist
3 import os
4
5 # Assuming that RANK, WORLD_SIZE, LOCAL_RANK environment variables were set by launcher
6 rank = int(os.environ['RANK'])
7 world_size = int(os.environ['WORLD_SIZE'])
8 local_rank = int(os.environ['LOCAL_RANK'])
9
10 # Initialize the group of processes
11 dist.init_process_group(
12     backend = 'nccl', # 'nccl' for GPU, 'gloo' for CPU
13     rank = rank,
14     world_size = world_size,
15 )
16 # Set the device for the current process
17 torch.cuda.set_device(local_rank)
18 device = torch.device(f"cuda:{local_rank}")

```

Listing 3.3: Initialize the group of processes to use in distributed training.

2. Initialize the Group of Processes

Next, we initialize the distributed backend using the code included in Listing 3.3, which selects the communication backend and the number of process to consider in distributed training. It is highly recommended to use the nccl backend for training in NVIDIA GPUs due to its superior performance.

3. Prepare a Distributed Data Loader

We also have to modify the data loading to include DistributedSampler. This sampler ensures each process gets a different chunk of a minibatch without overlap between chunks (Listing 3.4).

```

1 from torch.utils.data import DataLoader, Dataset
2 from torch.utils.data.distributed import DistributedSampler
3
4 # Assume that 'train_dataset' is our torch.utils.data.Dataset instance
5 train_sampler = DistributedSampler(
6     train_dataset,
7     num_replicas = world_size,
8     rank = rank,
9 )
10
11 # Use 'shuffle=False' because DistributedSampler handles shuffling
12 # Using 'pin_memory=True' can speed up host-device communication
13 train_loader = DataLoader(
14     train_dataset,
15     batch_size = chunk_size,
16     sampler = train_sampler,
17     num_workers = num_workers_per_process,
18     pin_memory = True,
19     shuffle = False,
20 )

```

Listing 3.4: Prepare a distributed data loader.

Note that the argument `batch_size` of the `DataLoader` refers to the chunk size assigned to each process. The total effective batch size across all processes is `chunk_size * world_size`. We should set the argument `shuffle` to `False` in the `DataLoader` because the `DistributedSampler` takes care of shuffling the data appropriately across epochs.

4. Wrap the Model

Now, we instantiate the model and move it to the computing device associated to the current process before wrapping the model with `DistributedDataParallel` (Listing 3.5).

```

1 import torch.nn as nn
2 from torch.nn.parallel import DistributedDataParallel as DDP
3
4 # Instantiate the model
5 model = SimpleModel().to(device) # Move the model to the correct device first
6
7 # Wrap the model with DistributedDataParallel
8 # 'device_ids' should contain the single GPU ID for this process
9 # 'output_device' should be the same as 'device_ids[0]'
10 model = DDP(model, device_ids=[local_rank], output_device=local_rank)

```

Listing 3.5: Instantiate the model and wrap it with DistributedDataParallel.

`device_ids` tells DDP which GPU(s) this process manages, usually just one and identified by the `local_rank`, and `output_device` specifies where the output from the model should be placed, typically is the same device.

5. Training Loop Adjustment

The core training loop remains the same. The main difference is that `loss.backward()` now implicitly triggers the gradient synchronization across all processes (Listing 3.6). It is standard practice the training loop to include logging, saving model checkpoints, and a validation step, which run primarily on a single process, usually the one with rank zero, to avoid redundant operations and a cluttered output. We should call `train_sampler.set_epoch(epoch)` at the beginning of each epoch to ensure a different data shuffling per epoch when using `DistributedSampler`. Finally, we call `dist.destroy_process_group()` when training finishes to free the computing resources.

```

1 optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
2
3 for epoch in range(num_epochs):
4     # Set epoch for sampler to ensure proper shuffling across epochs
5     train_sampler.set_epoch(epoch)
6     model.train()
7
8     for batch_id, (data, target) in enumerate(train_loader):
9         data, target = data.to(device), target.to(device) # Move data to the process GPU
10        optimizer.zero_grad()
11        output = model(data)
12        loss = loss_fn(output, target)
13        loss.backward() # Triggers gradient synchronization
14        optimizer.step() # Updates local replica with averaged gradients
15
16        if rank == 0 and batch_id % log_interval == 0: # Log only on rank 0 device
17            print(f"Epoch: {epoch} | Batch: {batch_id} | Loss: {loss.item()}")
18
19 # Kill the created processes
20 dist.destroy_process_group()

```

Listing 3.6: A minimalist training loop.

Listing 3.7 contains (i) an example of the code necessary for launching the distributed data parallel training in multiple GPUs and (ii) the structure of the function that each parallel process executes (`ddp_train`). The implementation of all the tasks described in this function, except creating the training dataset, can be found in the previous five listings. Just to complete the implementation, Listing 3.8 includes a function for creating a CIFAR10 dataset for training.

```

1 def ddp_train(
2     # Function executed by all processes
3     rank: int,
4     world_size: int,
5     final_model_path: str,
6 ):
7     os.environ["MASTER_ADDR"] = "localhost"
8     os.environ["MASTER_PORT"] = "54321" # select any idle port on your machine
9     init_process_group(backend="nccl", rank=rank, world_size=world_size) # "gloo" | "nccl"
10
11     Setup training hyperparameters
12     Create a training Dataset
13     Create a training DistributedSampler
14     Create a training DataLoader -> train_loader
15     Instantiate the model -> model
16     Wrap the model with DistributedDataParallel -> ddp_model
17     Select the optimizer -> optimizer
18
19     for epoch in range(number_epochs):
20         train_sampler.set_epoch(epoch)
21         ddp_model.train()
22         for (data, target) in train_loader:
23             Train the ddp_model for an iteration
24     Kill the created processes
25
26 if __name__ == "__main__":
27     world_size = torch.cuda.device_count()
28     final_model_path = Path("./models/CIFAR10_ddp_epoch15.pt")
29     mp.spawn(
30         ddp_train,
31         args = (world_size, final_model_path),
32         nprocs = world_size, # number of processes - # gpus
33     )

```

Listing 3.7: Pseudo-code for start training with DDP.

```

1 def cifar_dataset(data_root: Path) -> Dataset:
2     transform = transforms.Compose(
3         [
4             transforms.ToTensor(),
5             transforms.Normalize(
6                 mean = (0.49139968, 0.48215827, 0.44653124),
7                 std = (0.24703233, 0.24348505, 0.26158768),
8             ),
9         ]
10    )
11    trainset = CIFAR10(root=data_root, train=True, transform=transform, download=True)
12    return trainset

```

Listing 3.8: Create a CIFAR10 dataset for training.

We present next a few consideration about distributed data parallel training.

a. Saving and Loading Model Checkpoints

When saving a `DistributedDataParallel` model, it is often necessary only one process, for example the one with rank zero, to save the training state dictionary. `DistributedDataParallel` wraps the original model, so the saved state dictionary will have keys prefixed with "module.". Thus, we need to account for this prefix when loading the training state dictionary into a non-distributed model or, otherwise, access the underlying model using `model.module.state_dict()` (Listing 3.9).

```

1 # Saving the model checkpoint only on rank 0
2 if rank == 0:
3     torch.save(model.module.state_dict(), "simple_model.pth")
4
5 # Loading the model checkpoint on all ranks
6 map_location = {'cuda:%d' % 0: 'cuda:%d' % local_rank} # Map to current device
7 checkpoint = torch.load("simple_model.pth", map_location=map_location)
8
9 # Create a model instance
10 model_instance = SimpleModel().to(device)
11
12 # Load the state dictionary into the instantiated model
13 model_instance.load_state_dict(checkpoint)
14
15 # Wrap the model with DistributedDataParallel
16 ddp_model = DDP(model_instance, device_ids=[local_rank], output_device=local_rank)

```

Listing 3.9: Saving and loading model checkpoints on a DDP context.

b. Batch Normalization

DistributedDataParallel handles the synchronization of batch normalization statistics correctly across processes. Unlike DataParallel, we usually do not need `torch.nn.SyncBatchNorm`, although it can be used if necessary.

c. Mixed Precision

DistributedDataParallel supports automatic mixed precision (AMP) through `torch.cuda.amp`. Thus, we wrap the model with DistributedDataParallel after instantiating a GradScaler and follow the standard AMP procedure in the training loop. According to PyTorch documentation, when the forward pass for a given operation has `float16` inputs, the backward pass for this operation will produce `float16` gradients. Gradient values with small magnitudes may not be representable in `float16`. These values will be approximated by zero, so the update for the corresponding weights will produce no effect. To prevent underflow, **gradient scaling** multiplies the losses by a scale factor and runs the backward pass using the scaled losses. As a consequence, the gradients computed during the backward pass will also be scaled by the same factor. In other words, gradient values will have a larger magnitude, so they are approximated by zero.

d. Identify the Parameters without Calculated Gradients

If some of the model's weights are not updated through the gradients during the backward pass, for example due to conditional logic in the forward method, DistributedDataParallel's backward pass synchronization might hang, waiting for gradients that are not calculated. Setting `find_unused_parameters=True` in the DistributedDataParallel constructor resolves this issue, but it adds some overhead. It is generally preferable to ensure that all weights require gradients participate in the loss computation.

Overall, DistributedDataParallel provides a high-performance mechanism for scaling training across multiple GPUs and nodes. By understanding its multiprocessing architecture, reliance on collective communication for gradient averaging, and the necessary adjustments to data loading and model wrapping, we can effectively train larger and more complex models faster than with sequential training.

3.2 Parameter Server

The previous section discussed how to scale model training on multiple processes, often running on single machine CPUs or GPUs. When we need to scale up to huge datasets and models, eventually one machine would be insufficient. Thus, this section covers methods for using multiple machines to run model training, namely the parameter server proposal. Parameter server was first proposed by Google in the DistBelief paper [47].

Among the parameter server implementations are (i) PyTorch RPC-based distributed training (RPC), where RPC facilitates communication between the worker nodes and the parameter server, enabling efficient synchronization of the model parameters during training, and (ii) TensorFlow ParameterServerStrategy

that distributes the training steps for a cluster that scales up to thousands of workers, accompanied by parameter servers. This configuration is known as asynchronous training.

We should consider using parameter server (i) in a federated learning scenario or (ii) when we want to train the model with multiple workers asynchronously using spot instances. In this case, if some workers fail, it does not hinder the cluster from continue working. This enables the cluster to train using instances that may occasionally become unavailable, such as preemptible or spot instances.

Communicating over the network can have high latency, so an important aspect of distributed training is overlapping computation and communication. For the best performance, we want the worker nodes to be occupied with useful work while communication is going on, rather than having to stop and wait for the communication to finish.

3.2.1 Running Stochastic Gradient Descent with AllReduce

AllReduce allows for running optimization algorithms, such as stochastic gradient descent (SGD), in a distributed way [46]. The key idea is to parallelize each minibatch of size BS in multiple chunks of size uBS . We start with a copy of the weights w_t on each worker. If the SGD update rule is given by Equation 3.5, if L_i is the loss function associated to data point i , there are N worker nodes and we consider $BS = N \cdot uBS$, then we can re-write the update rule as indicated in Equation 3.6.

$$w_{t+1} = w_t - \eta_t \cdot \frac{1}{BS} \sum_{b=1}^{BS} \nabla L_{i_{b,t}}(w_t) \quad (3.5)$$

$$w_{t+1} = w_t - \eta_t \cdot \frac{1}{N} \sum_{n=1}^N \frac{1}{uBS} \sum_{b=1}^{uBS} \nabla L_{i_{n,b,t}}(w_t) \quad (3.6)$$

In this way, the computation of the inner sum in Equation 3.6 for $n = 1$ is assigned to worker 1, the computation of the inner sum for $n = 2$ is assigned to worker 2, until the computation of the inner sum for $n = N$ being assigned to worker N . After all the gradients are computed, we can perform the outer sum with an AllReduce operation, and we get the aggregated gradients formulated in Equation 3.7. After the AllReduce operation, all the worker nodes will have these aggregated gradients. After that, each worker can individually update the weight values from w_t to w_{t+1} and the results should be identical in all nodes. This corresponds to the algorithm included in Figure 3.12.

$$\text{aggregated gradients} = \sum_{n=1}^N \frac{1}{uBS} \sum_{b=1}^{uBS} \nabla L_{i_{n,b,t}}(w_t) \quad (3.7)$$

This workflow can also be adopted for running other optimization algorithm, such as Adam and SGD with Momentum, distributedly and using AllReduce.

Among the advantages of distributed SGD with all-reduce are (i) it is easy to reason about, since it is statistically equivalent to minibatch SGD, (ii) it is easy to implement, since all the worker machines have the same role and it runs on top of standard distributed computing primitives. Some disadvantages include (i) while the communication relative to the AllReduce operations is occurring, the workers are mostly idle, since it does not overlap computation and communication, (ii) the effective minibatch size is growing with the number of nodes, and there are cases where we do not want to run training with a large minibatch size for statistical reasons and, as so, distributed SGD can prevent us from scaling training to a large number of nodes.

There are two general options to get the training samples in distributed learning: (i) reserve one or more non-worker nodes for storing the training data, for example, through a distributed in-memory filesystem, and have the worker nodes load training samples from those servers, (ii) partition the training data among the workers and store the parcels locally on the worker's memory.

```

Requires: number of nodes  $N$ , per-node minibatch size  $uBS$ 
Requires: learning rate schedule  $\eta_t$ , initial parameters  $w_0$ , number of iterations  $T$ 
Requires: loss function values  $L_1, L_2, \dots, L_{uBS}$ 
for  $n = 1$  to  $N$  run in parallel on node  $n$ 
    Load initial  $w_0$ 
    for  $t = 1$  to  $T$  do
        Select a minibatch of data  $[i_{n,1,t}, i_{n,2,t}, \dots, i_{n,uBS,t}]$  of size  $uBS$ 
        Compute gradients  $g_{n,t} \leftarrow \frac{1}{uBS} \sum_{b=1}^{uBS} \nabla L_{i_{n,b,t}}(w_{t-1})$ 
        Run AllReduce across all workers to compute aggregated gradients  $G_t = \sum_{n=1}^N g_{n,t}$ 
        Update the model weights  $w_t \leftarrow w_{t-1} - \frac{\eta_t}{N} \cdot G_t$ 
    end for
end parallel for
return  $w_T$  obtained from any selected node

```

Figure 3.12: Algorithm for distributed stochastic gradient descent with AllReduce [46].

3.2.2 The Parameter Server Architecture

Often the convergence of an optimization algorithm is measured by some function over the model's weights at time t , for example, the loss function or the norm of the gradient, which decreases with t and gives us feedback about the algorithm progress. Thus, to evaluate the algorithm convergence, we must have access to the weight values at time t when the algorithm is running. In SGD, the weight values after running the algorithm for t timesteps is given by $w_{t+1} = w_t - \eta_t \nabla L_i(w_t)$.

In sequential training, the weights at time t are just the values of some array stored in memory at this time. But in a distributed setting, there is no shared memory and communication must be done explicitly. Each machine will usually have one or more copies of the weights at any given time, some of which may have been updated less recently than others, especially if we want to do something more complicated than vanilla AllReduce. This raises a relevant question when using a distributed algorithm: what are the weight values at a certain time?

In SGD with AllReduce, since the weight values are the same on all worker nodes, which is guaranteed by the AllReduce operation, the identical values owned by every node are the effective weight values at any given time. The parameter server model gives a different answer to the raised question, since it assigns to a single machine, the parameter server, the explicit responsibility of maintaining the current weight values. The most up-to-date weights are those stored in the parameter server's memory. The parameter server updates its weights by using gradients computed on the worker nodes, and pushed to the parameter server. Periodically, the parameter server broadcasts its updated parameters to all the workers, so that they can use the updated weights to compute the gradients. Figure 3.13 depicts the diagram of a centralized parameter server architecture.

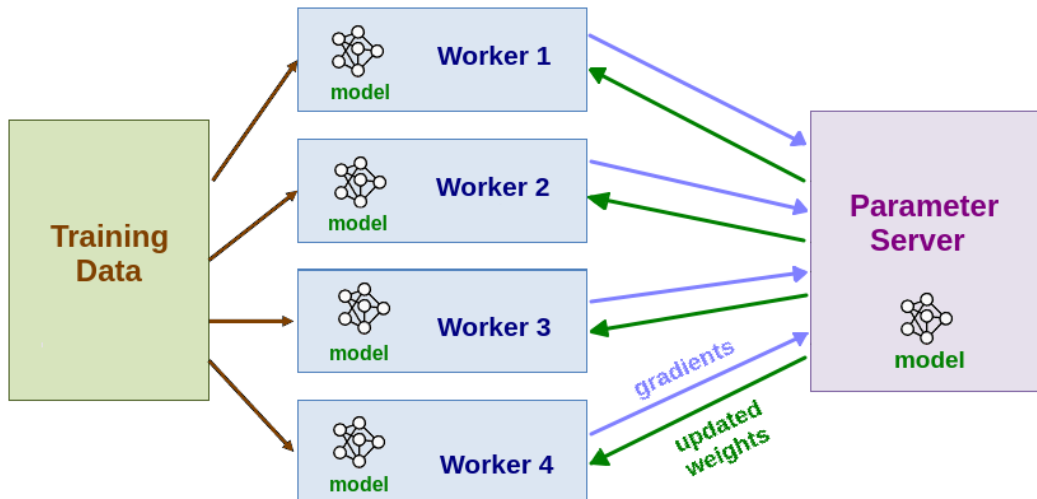


Figure 3.13: A centralized parameter server architecture.

There are many ways to run model training over a parameter server architecture, but one of the most common is asynchronous distributed training, and its algorithm is included in Figure 3.14.

```

Requires: loss function values  $L_1, L_2, \dots, L_{uBS}$ 
Requires: number of worker nodes  $N$ , per-node minibatch size  $uBS$ 
Requires: learning rate  $\eta$ , initial weights  $w_0$ , number of iterations per worker  $T$ 

for  $n = 1$  to  $N$  run in parallel on node  $n$ 
  Load  $w_{n,0}$  from the parameter server
  for  $t = 1$  to  $T$  do
    Select a minibatch of data  $[i_{n,1,t}, i_{n,2,t}, \dots, i_{n,uBS,t}]$  of size  $uBS$ 
    Compute the gradients  $g_{n,t} \leftarrow \frac{1}{uBS} \sum_{b=1}^{uBS} \nabla L_{i_{n,b,t}}(w_{n,t-1})$ 
    Push the gradients  $g_{n,t}$  to the parameter server
    Receive the new model weights  $w_{n,t}$  from the parameter server
  end for
end parallel for

run in parallel on parameter server
  Initialize the model weights  $w \leftarrow w_0$ 
  loop
    Receive a gradient  $g$  from a worker node
    Update model weights  $w \leftarrow w - \eta g$ 
    Send updated weights  $w$  back to the worker
  end loop
end run on parameter server
return  $w_T$  obtained from any node

```

Figure 3.14: Algorithm for asynchronous distributed stochastic gradient descent with parameter server [46].

If the weights are too numerous for a single parameter server to handle, we can use multiple parameter servers. In this case, we partition the weights equally among the multiple parameter server, and each parameter server is only responsible for maintaining the weights in its partition. When a worker wants to send its gradients, it will partition the gradient vector and send each chunk to the corresponding parameter server. Later, the worker will receive the corresponding chunk of the updated model's weights from that parameter server.

In the next two sub-sections, we present two large-scale distributed optimization algorithms: Downpour SGD and Sandblaster L-BFGS. Both methods leverage the concept of a centralized parameter server presented before. Both methods were designed to tolerate different processing speeds on the different worker nodes, and withstand even the total failure of worker nodes due to them being randomly shut down or restarted [47].

3.2.3 Downpour Stochastic Gradient Descent

Stochastic gradient descent is likely the most commonly used optimization algorithm for training neural networks [15] [35] [128]. But the original formulation of SGD is inherently sequential, making it impractical to apply to use cases where the model needs to be trained on very large datasets. To apply SGD to large datasets, in [47] the authors introduced Downpour SGD, an online variant of asynchronous stochastic gradient descent that uses multiple replicas of a model. The training data is divided into a number of subsets and run a copy of the model on each of these subsets. The models communicate updates through centralized parameter server, which has the most updated state of all the model weights, which are sharded across several machines. Thus, if we have 10 parameter server shards, each shard is responsible for storing and updating 1/10-th of the model weights (Figure 3.15 (i)). This approach is asynchronous in two ways: the model replicas run independently of each other, and the parameter server weight's shards also run independently of one another.

In the simplest implementation, before processing each mini-batch, each model replica requests an updated copy of its model weights from the parameter server. Because the models are also partitioned across multiple

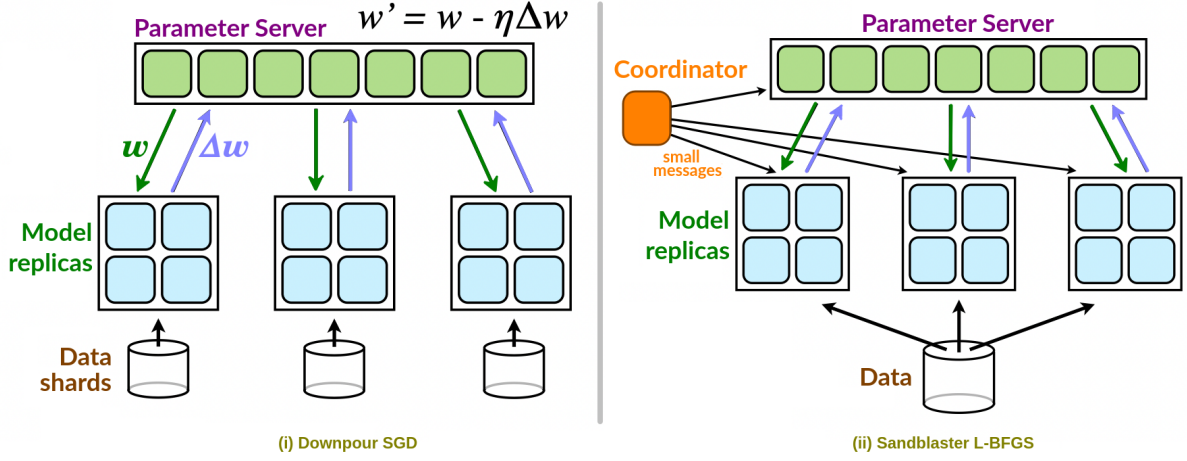


Figure 3.15: (i) Downpour SGD, where model replicas asynchronously fetch weights w and push their gradients δw to the parameter server; (ii) Sandblaster L-BFGS, where a single coordinator sends small messages to the model replicas and the parameter server to orchestrate optimization [47].

machines, each model replica communicates only with the subset of parameter server shards that hold the model weights that belong to the same partition. After receiving an updated copy of its weights, the model replica processes a minibatch of data, calculates the weights' gradients, and sends the gradients to the parameter server, which then applies the gradients to update the weights. The communication overhead of Downpour SGD can be reduced by limiting each model replica to request updated weights only every n_{fetch} iterations and send the updated gradients only every n_{push} iterations, where n_{fetch} might not be equal to n_{push} . In fact, the process of fetching the weights, pushing the gradients, and processing the training data must be performed in three weakly synchronized threads.

Downpour SGD is more robust to machines failures than standard synchronous SGD. In synchronous SGD, if one machine fails, the entire training process gets stalled, whereas in asynchronous SGD, if the machine where a model replica resides fails, the other model replicas continue processing their training data and updating the model parameters via the parameter servers. The diverse forms of asynchronous processing in Downpour SGD introduce an additional stochasticity in the optimization process. Mostly, a model replica often computes the gradients based on weight values that are slightly outdated, while other model replica might have updated the weights on the parameter server at the same time. There are other sources of stochasticity beyond this, since the parameter server shards act independently, there is no guarantee that at any given moment the weights on a given shard of the parameter server have undergone the same number of updates, or that the updates were applied in the same order. Moreover, because the model replicas are allowed to fetch parameters and push gradients in separate threads, there may be additional subtle inconsistencies in the weights' state.

One technique that greatly increases the robustness of Downpour SGD is using the Adagrad [58] adaptive learning rate scheduler. Rather than using a single fixed learning rate on the parameter server, η in Figure 3.15, Adagrad uses a distinct and adaptive learning rate with each weight. Let $\eta_{i,K}$ be the learning rate of the i -th weight at iteration K and $\nabla w_{i,K}$ its gradient, then we set $\eta_{i,K} = \gamma / \sqrt{\sum_{j=1}^K \nabla w_{i,j}^2}$. Because these learning rates are computed only from the summed squared gradients of each weight, Adagrad is easily implemented locally within each parameter server shard. The value of γ , the constant scaling factor for all learning rates, is generally larger than the best fixed learning rate used without Adagrad. Adagrad extends the maximum number of model replicas that can productively work simultaneously, and combined with learning rate warmup applied to a single model replica before initiating the other replicas, it has eliminated the most stability concerns in training deep neural networks with Downpour SGD.

3.2.4 Sandblaster L-BFGS

Batch methods work well in training small neural networks [111]. The authors of [47] developed the Sandblaster batch method for training large models on large datasets, and also proposed an implementation using the Limited-memory BFGS (L-BFGS) algorithm [128]. The key idea in Sandblaster is distributed

parameter storage and manipulation. The core of the L-BFGS optimization algorithm resides in a coordinator process (Figure 3.15 (ii)), which does not have direct access to the model weights. Instead, the coordinator issues commands drawn from a small set of operations, such as the dot product, scaling, coefficient-wise addition, or multiplication, that can be executed by each parameter server shard independently, with the results being stored locally on the same shard. Additional information, like the history cache used by L-BFGS, is also stored on the parameter server shard on which it was computed. This allows running models with billions of weights without incurring in the overhead of sending all the parameters and gradients to a single server.

In most parallel implementations of L-BFGS, data is distributed by many machines and each machine is responsible for computing the gradients on a subset of the data. The gradients are sent back to a central server, or aggregated via a tree [1]. In some cases, the algorithm has to wait for the slowest machine, and therefore do not scale well to large shared clusters. To account for this problem, in [47] they applied the following load balancing strategy: The coordinator assigns each of the N model replicas a small portion of the work, much smaller than $1/N$ -th of the total size of a batch, and assigns new parts of the work to replicas whenever they become idle. This strategy allows for faster model replicas to do more work than slower replicas. To further manage slow replicas at the end of a batch, the coordinator schedules multiple copies of the outstanding portions and uses the result from whichever model replica finishes first. This scheme is similar to using backup tasks in MapReduce [48]. Prefetching of data, along with supporting data affinity by assigning sequential portions of data to the same worker, solves the data access issues. In contrast to Downpour SGD, which requires relatively high frequency and high bandwidth weight synchronizations with the parameter server, Sandblaster workers only fetch parameters at the beginning of each batch, when they have been updated by the coordinator, and only send the gradients every few completed portions, to protect against replica failures and restarts.

3.2.5 Implementing a Parameter Server Using the PyTorch Distributed RPC Framework

This section details the implementation of a parameter server using PyTorch's Distributed RPC framework. With Distributed RPC framework, we will create a training setup where multiple workers use RPC to communicate with the same parameter server and use `RRef`¹ to access states on the remote parameter server instance. Each worker will run its dedicated backward pass in a distributed way through stitching of the Autograd graph across multiple nodes using distributed Autograd.

PyTorch RPC main concepts

The distributed RPC framework makes it easy to run functions remotely, supports referencing remote objects without copying the real data around, and provides Autograd and optimizer APIs to transparently run the backward pass and update parameters across multiple machines. The features of the framework can be categorized into four sets.

Remote Procedure Call (RPC) supports running a function on the specified destination worker with the given arguments and getting the return value back or creating a reference to the return value. There are three main RPC APIs: `rpc_sync` for synchronous communication, `rpc_async` for asynchronous communication, and `remote` for asynchronous communication and the caller gets a reference to the remote return value. We should use the synchronous API if our code cannot proceed without the return value. Otherwise, use the asynchronous API to get a future, and wait on the future when the return value is needed on the caller. The `remote` API is useful when we want to create something remotely but do not need to get something back.

Remote Reference (RRef) functions as a distributed shared pointer to a local or remote object. Each `RRef` only has one owner and the object only lives on that owner. Non-owner workers holding `RRefs` can get copies of the object from the owner by explicitly requesting it. This is useful when a worker needs to access some data object, but itself is neither the creator or the owner of the object.

Distributed Autograd groups together the local Autograd engines of all workers involved in the forward pass of distributed training, and automatically reach out to them during the backward pass to compute gradients. This is especially helpful if the forward pass involves multiple machines, such as in parameter server training.

¹<https://pytorch.org/docs/stable/rpc.html#torch.distributed.rpc.RRef>

In this way, we no longer need to worry about how to send gradients via RPC and about the order of launching the local Autograd engines.

The **Distributed Optimizer**'s constructor takes an optimizer, such as SGD or Adagrad, and a list of weight RRefs, creates an **Optimizer** instance on each distinct RRef owner, and updates the weights accordingly when running `step`. When we run distributed forward and backward passes, the weights and gradients will be scattered across multiple workers, and hence it requires an optimizer on each of the involved workers. Distributed Optimizer wraps all those local optimizers into one, and provides a concise constructor and `step` method.

```
1 import argparse
2 import os
3 import time
4 from threading import Lock
5 import torch
6 import torch.distributed.autograd as dist_autograd
7 import torch.distributed.rpc as rpc
8 import torch.multiprocessing as mp
9 import torch.nn as nn
10 import torch.nn.functional as F
11 from torch import optim
12 from torch.distributed.optim import DistributedOptimizer
13 from torchvision import datasets, transforms
14
15 # Neural network to classify MNIST images
16 class Net(nn.Module):
17     def __init__(self, num_gpus=0):
18         super(Net, self).__init__()
19         print(f"Using {num_gpus} GPUs for training")
20         self.num_gpus = num_gpus
21         device = torch.device(
22             "cuda:0" if torch.cuda.is_available() and self.num_gpus > 0 else "cpu")
23         print(f"Place the first 2 convolutions on {str(device)}")
24         # Place the convolution layers on the first GPU, or CPU if there is no GPU
25         self.conv1 = nn.Conv2d(1, 32, 3, 1).to(device)
26         self.conv2 = nn.Conv2d(32, 64, 3, 1).to(device)
27         # Place the remaining layers on the 2nd GPU, if there is one
28         if "cuda" in str(device) and num_gpus > 1:
29             device = torch.device("cuda:1")
30
31         print(f"Place the remaining layers on {str(device)}")
32         self.dropout1 = nn.Dropout2d(0.25).to(device)
33         self.dropout2 = nn.Dropout2d(0.5).to(device)
34         self.fc1 = nn.Linear(9216, 128).to(device)
35         self.fc2 = nn.Linear(128, 10).to(device)
36
37     def forward(self, x):
38         x = self.conv1(x)
39         x = F.relu(x)
40         x = self.conv2(x)
41         x = F.max_pool2d(x, 2)
42         x = self.dropout1(x)
43         x = torch.flatten(x, 1)
44         # Move tensor 'x' to the next device if necessary
45         next_device = next(self.fc1.parameters()).device
46         x = x.to(next_device)
47
48         x = self.fc1(x)
49         x = F.relu(x)
50         x = self.dropout2(x)
51         x = self.fc2(x)
52         output = F.log_softmax(x, dim=1)
53         return output
```

Listing 3.10: The neural network that we will train on the MNIST dataset in a distributed way.

To implement the parameter server, we start by importing the necessary modules (Listing 3.10) and by defining a simple convolutional neural network that we will train on the MNIST dataset.

Listing 3.11 implements some helper functions that will be useful for the rest of our parameter server. This code uses `rpc_sync`² and `RRef` for defining a function that invokes a given method on an object living on a remote node. The handle to the remote object is given by the `rref` argument of the functions, and we run it on its owning node, `rref.owner()`. On the caller node, we run this command synchronously through the use of `rpc_sync`, meaning that we block the caller until a response is received.

```

1 """
2 Helper functions
3
4 On the local node, call a method with first 'arg' as the value held by the RRef.
5 Other 'arg's are passed in as arguments to the function called. This is useful
6 for calling instance methods. 'method' can be any matching function, including
7 class methods.
8 """
9 def call_method(method, rref, *args, **kwargs):
10     return method(rref.local_value(), *args, **kwargs)
11 """
12 Given an RRef, returns the result of calling the provided method on the value
13 held by the RRef. This call is done on the remote node that owns the RRef and
14 passes along the given argument.
15 Example: If the value held by the RRef is of type 'Foo', then
16 'remote_method(Foo.bar, rref, arg1, arg2)' is equivalent to calling
17 '<foo_instance>.bar(arg1, arg2)' on the remote node and getting the result back.
18 """
19 def remote_method(method, rref, *args, **kwargs):
20     args = [method, rref] + list(args)
21     return rpc.rpc_sync(rref.owner(), call_method, args=args, kwargs=kwargs)

```

Listing 3.11: Helper functions to call a given method locally or on a remote node.

Listing 3.12 defines the constructor of the parameter server class. It inherits `nn.Module` and instantiates the neural network defined in Listing 3.10. It also registers the identification of device that will be used for placing the network.

```

1 # Parameter Server class
2 class ParameterServer(nn.Module):
3     def __init__(self, num_gpus=0):
4         super().__init__()
5         model = Net(num_gpus=num_gpus)
6         self.model = model
7         self.input_device = torch.device(
8             "cuda:0" if torch.cuda.is_available() and num_gpus > 0 else "cpu"
9         )

```

Listing 3.12: Constructor method of the parameter server class.

Listing 3.13 defines the `forward` method of the parameter server class. Regardless of the device where the model is trained, we move the output to CPU, as the Distributed RPC framework only supports sending CPU tensors over RPC. We have intentionally disabled sending CUDA tensors over RPC to avoid having different devices on the caller/callee.

```

1 class ParameterServer(nn.Module):
2     # ...
3     def forward(self, inp):
4         inp = inp.to(self.input_device)
5         out = self.model(inp)
6         # The output tensor is moved to CPU
7         out = out.to("cpu")
8         return out

```

Listing 3.13: The `forward` method of the parameter server class.

²https://pytorch.org/docs/stable/rpc.html#torch.distributed.rpc.rpc_sync

Next, we create two methods that are useful for training and verification purposes (Listing 3.14). The `get_dist_gradients` method gets a Distributed Autograd context identifier as argument and calls `dist_autograd.get_gradients` in order to retrieve the gradients computed by the distributed Autograd. It also iterates through the resulting dictionary with gradients and converts each tensor to a CPU tensor, as PyTorch might not support sending tensors over RPC yet. The `get_param_rrefs` method iterates through our model weights and wrap them with local RRefs. This method will be called over RPC by the worker nodes and returns a list of the weights to be optimized. This is required as input to the distributed optimizer, which requires the weights to optimize as a list of RRefs.

```

1 # Use distributed Autograd to retrieve gradients accumulated for this model.
2 # Primarily used for verification.
3 def get_dist_gradients(self, contextID):
4     grads = dist_autograd.get_gradients(contextID)
5     # This output is forwarded over RPC, and PyTorch might not yet accept GPU tensors.
6     # Thus, tensors are moved from GPU memory to CPU.
7     cpu_grads = {}
8     for k, v in grads.items():
9         k_cpu, v_cpu = k.to("cpu"), v.to("cpu")
10        cpu_grads[k_cpu] = v_cpu
11    return cpu_grads
12
13 # Wrap the local weights with RRefs. This is necessary for building the
14 # DistributedOptimizer that will optimize the weights remotely.
15 def get_param_rrefs(self):
16     param_rrefs = [rpc.RRef(param) for param in self.model.parameters()]
17     return param_rrefs

```

Listing 3.14: Methods to retrieve distributed gradients and to wrap the local weights with RRefs.

Finally, we create two methods that help us initialize the parameter server (Listing 3.15). Note that there will only be one instance of the parameter server across all processes, all workers will interact with the same parameter server and update the same weights that it stores. As seen in `run_parameter_server`, the server itself does not take any independent actions. It waits for requests from workers, which we will define below, and responds to them by running the requested function.

```

1 # The global parameter server instance.
2 param_server = None
3 # A lock to ensure that we only have one parameter server.
4 global_lock = Lock()
5
6 def get_parameter_server(num_gpus=0):
7     # Returns a singleton parameter server to all trainer processes.
8     global param_server
9     # Ensure that we get only one handle to the parameter server.
10    with global_lock:
11        if not param_server:
12            # Instantiate the ParameterServer
13            param_server = ParameterServer(num_gpus=num_gpus)
14        return param_server
15
16 def run_parameter_server(rank, world_size):
17     """
18     The parameter server just acts as a host for our model and responds to
19     the requests received from the workers.
20     'rpc.shutdown()' waits for all workers to complete, which in this case
21     means that the parameter server will wait for all workers to complete,
22     and then exits.
23     """
24     print("Parameter server master process initializing RPC")
25     rpc.init_rpc(name="parameter_server", rank=rank, world_size=world_size)
26     print("RPC initialized! Running the parameter server...")
27     rpc.shutdown()
28     print("RPC shutdown on the parameter server.")

```

Listing 3.15: Methods to initialize the parameter server.

`rpc.shutdown()` will not immediately shut down the parameter server. Instead, it waits for all workers to also call `rpc.shutdown()`. This gives us the guarantee that the parameter server will not go offline before all workers have completed their training task.

Listing 3.16 implements the constructor method of the `WorkerNet` class. This class also inherits `nn.Module`, and the constructor method uses `rpc.remote` to obtain a remote reference (an `RRef`) to the parameter server. We are not copying the parameter server to the local process, instead, we can think of `self.param_server_rref` as a distributed shared pointer to the parameter server that lives on a separate process.

```
1 # The worker class.
2 # nn.Module corresponds to the network that will be trained by this worker.
3 class WorkerNet(nn.Module):
4     def __init__(self, num_gpus=0):
5         super().__init__()
6         self.num_gpus = num_gpus
7         self.param_server_rref = rpc.remote(
8             "parameter_server",
9             get_parameter_server,
10            args = (num_gpus,)
11        )
```

Listing 3.16: The constructor method of the worker class.

Listing 3.17 defines the method `get_global_param_rrefs`. This method is necessary because the `DistributedOptimizer` requires as input a list of `RRefs` corresponding to the remote weights to be optimized, so here we obtain these `RRefs`. Since the only remote process that a given `WorkerNet` interacts with is the `ParameterServer`, we simply invoke a `remote_method` on the `ParameterServer`. This method uses the `get_param_rrefs` method of the `ParameterServer` class. The method returns a list of `RRefs` to the weights that need to be optimized. In this case, our `WorkerNet` does not define its own weights; if it did, we would need to wrap each weight with an `RRef` as well and include it into our input to the `Distributed Optimizer`.

```
1 class WorkerNet(nn.Module):
2     # ...
3     def get_global_param_rrefs(self):
4         remote_params = remote_method(
5             ParameterServer.get_param_rrefs,
6             self.param_server_rref)
7         return remote_params
```

Listing 3.17: Method to retrieve a list of `RRefs` for the remote weights that reside on the parameter server.

The `forward` method of the worker class uses a synchronous RPC call to run the forward pass of model training, which corresponds to the `forward` method of the `ParameterServer` class. In this RPC call, we pass the `self.param_server_rref` as argument, which is a remote handle to the `ParameterServer`. This call sends an RPC to the node on which our `ParameterServer` is running, invokes the `forward` pass, and returns the `Tensor` corresponding to the model's output (Listing 3.18).

```
1 class WorkerNet(nn.Module):
2     # ...
3     def forward(self, x):
4         model_output = remote_method(
5             ParameterServer.forward,
6             self.param_server_rref,
7             x
8         )
9         return model_output
```

Listing 3.18: The `forward` method of the worker class.

The next step is to implement the loop for training our model, where we pass some input data through the network and compute the loss. The training loop is similar to the sequential version, with some modifications due to the nature of our network being distributed across machines.

Listing 3.19 instantiates the `WorkerNet` and the `DistributedOptimizer`. As mentioned above, we must apply into the optimizer all of the global weights, global across all nodes participating in distributed training, that need to be optimized. Additionally, we also provide as input the local optimizer to be used, which in this case is SGD. We can configure the underlying optimizer algorithm in the same way as we do with a local optimizer. All arguments of `optim.SGD` will be provided by the `DistributedOptimizer`. As an example, we provide the `DistributedOptimizer` with a custom learning rate of 0.03 that will be used as the learning rate by all local SGD optimizers.

```

1 def run_training_loop(rank, num_gpus, train_loader, test_loader):
2     # Runs the typical neural network forward + backward + optimizer steps,
3     # Instantiate the worker
4     net = WorkerNet(num_gpus=num_gpus)
5     # Instantiate the distributed optimizer
6     param_rrefs = net.get_global_param_rrefs()
7     opt = DistributedOptimizer(optim.SGD, param_rrefs, lr=0.03)
8     # ...

```

Listing 3.19: Training loop: instantiating the worker and the distributed optimizer.

The training loop included in Listing 3.20 iterates over the provided training `DataLoader`. Before the forward, backward, and optimizer steps, we first create a distributed Autograd context to run them. This is necessary to record RPCs invoked in the model's forward pass, so that an appropriate graph can be constructed, which includes all participating distributed workers in the backward pass. The distributed Autograd context returns a `contextID` that serves as an identifier for accumulating and optimizing gradients corresponding to a particular iteration.

As opposed to calling `loss.backward()`, which would run the backward pass on a local worker, we call `dist_autograd.backward()` and pass as input the `contextID` as well as the loss. In addition, we pass the `contextID` as input of the optimizer call, which is required to be able to look up the corresponding gradients computed by this particular backward pass across all nodes.

```

1 def run_training_loop(rank, num_gpus, train_loader, test_loader):
2     # ...
3     for i, (data, target) in enumerate(train_loader):
4         with dist_autograd.context() as contextID:
5             model_output = net(data)
6             target = target.to(model_output.device)
7             loss = F.nll_loss(model_output, target)
8             if i % 5 == 0:
9                 print(f"Rank {rank} :: training batch {i} :: loss {loss.item()}")
10            dist_autograd.backward(contextID, [loss])
11            # Ensure that the distributed Autograd ran successfully and
12            # the gradients were returned.
13            assert remote_method(
14                ParameterServer.get_dist_gradients,
15                net.param_server_rref,
16                contextID,
17            ) != {}
18            opt.step(contextID)
19
20        print("Training complete!")
21        print("Getting accuracy....")
22        get_accuracy(test_loader, net)

```

Listing 3.20: Training loop: run the forward-backward-optimizer steps on each minibatch obtained from the training data loader.

The function `get_accuracy`, included in Listing 3.21, computes the test accuracy of the model after it has been trained, much like in sequential training.

```

1 def get_accuracy(test_loader, model):
2     model.eval() # Put the model in evaluation mode
3     correct_sum = 0
4     # Use a GPU in evaluation if available
5     device = torch.device("cuda:0" if model.num_gpus > 0
6         and torch.cuda.is_available() else "cpu")
7     with torch.no_grad():
8         for i, (data, target) in enumerate(test_loader):
9             out = model(data, -1)
10            pred = out.argmax(dim=1, keepdim=True)
11            pred, target = pred.to(device), target.to(device)
12            correct = pred.eq(target.view_as(pred)).sum().item()
13            correct_sum += correct
14
15     print(f'Accuracy {correct_sum / len(test_loader.dataset)}')
```

Listing 3.21: Calculate accuracy on the test dataset after the model was trained.

Just as we defined a main loop in `run_parameter_server` for the parameter server, which is responsible for initializing RPC, we define a similar loop for the workers in Listing 3.22. The difference is that the workers must run the training loop defined above. Analogously to `run_parameter_server`, by default `rpc.shutdown()` waits for all processes, both workers and parameter server, to call `rpc.shutdown()` before each worker exits. This ensures that the nodes are terminated gracefully and no node goes offline while another is expecting it to be online.

```

1 # Main loop of the worker.
2 def run_worker(rank, world_size, num_gpus, train_loader, test_loader):
3     print(f"Worker rank {rank} initializing RPC")
4     rpc.init_rpc(
5         name=f"trainer_{rank}",
6         rank=rank,
7         world_size=world_size)
8
9     print(f"Worker {rank} finished initializing RPC")
10
11     run_training_loop(rank, num_gpus, train_loader, test_loader)
12     rpc.shutdown()
```

Listing 3.22: Function to initialize RPC in each worker and run the training loop.

We completed the implementation of worker and parameter server, and all that is left is the code to launch the workers and the parameter server. First, we must provide the arguments necessary to instantiate the parameter server and workers (Listing 3.23). `world_size` corresponds to the total number of nodes that will participate in training, and accounts all workers and the parameter server. It is also necessary to provide a unique rank for each individual process, from 0 (where we will run the single parameter server) to `world_size - 1`. `master_addr` and `master_port` are arguments that can be used to identify where the rank 0 process is running, and will be used by individual nodes to discover each other. To test the implementation locally, we can simply use `localhost` as `master_addr` and a common `master_port` for all instances spawned. Our implementation only supports between 0 and 2 GPUs, but can be extended to allow more GPUs.

```

1 if __name__ == '__main__':
2
3     parser = argparse.ArgumentParser(
4         description="Parameter Server RPC based training")
5
6     parser.add_argument(
7         "--world_size",
8         type=int,
9         default=4,
10        help="Total number of participating processes. Should be the sum of
11        master node and all worker nodes.")
12
13    parser.add_argument(
14        "--rank",
15        type=int,
16        default=None,
17        help="Global rank of this process. Use 0 for master.")
18
19    parser.add_argument(
20        "--num_gpus",
21        type=int,
22        default=0,
23        help="Number of GPUs to use for training, Currently we support between 0
24        and 2 GPUs. Note that this argument will be passed to the parameter server.")
25
26    parser.add_argument(
27        "--master_addr",
28        type=str,
29        default="localhost",
30        help="Address of master, will default to localhost if not provided.
31        Master must be able to accept network traffic on the address:port.")
32
33    parser.add_argument(
34        "--master_port",
35        type=str,
36        default="29500",
37        help="Port that the master listens to, will default to 29500 if not
38        provided. Master must be able to accept network traffic on the host and port.")
39
40    args = parser.parse_args()
41    assert args.rank is not None, "must provide rank argument."
42    assert args.num_gpus <= 3, f"Only 0-2 GPUs are currently supported, got {args.num_gpus}."
43    os.environ['MASTER_ADDR'] = args.master_addr
44    os.environ['MASTER_PORT'] = args.master_port

```

Listing 3.23: Specify the arguments necessary to instantiate the parameter server and workers.

Finally, we create a process corresponding to either a parameter server or a worker, depending on the command line arguments (Listing 3.24). It is created a `ParameterServer` if we provide as rank the value 0, and it is created a `WorkerNet` otherwise. We are using `torch.multiprocessing` to launch a subprocess corresponding to the function that we want to execute, and waiting on this process's completion from the main thread with `p.join()`. If we initialize a worker, we also provide as arguments the training and test `DataLoaders` for the MNIST dataset.


```

1 processes = []
2 world_size = args.world_size
3
4 if args.rank == 0:
5     # start a parameter server on this process
6     p = mp.Process(target=run_parameter_server, args=(0, world_size))
7     p.start()
8     processes.append(p)
9 else:
10    # Get the training data
11    train_loader = torch.utils.data.DataLoader(
12        datasets.MNIST(
13            '../data',
14            train = True,
15            download = True,
16            transform = transforms.Compose(
17                [
18                    transforms.ToTensor(),
19                    transforms.Normalize((0.1307,), (0.3081,))
20                ]
21            )
22        ),
23        batch_size = 32,
24        shuffle = True,
25    )
26    test_loader = torch.utils.data.DataLoader(
27        datasets.MNIST(
28            '../data',
29            train = False,
30            transform = transforms.Compose(
31                [
32                    transforms.ToTensor(),
33                    transforms.Normalize((0.1307,), (0.3081,))
34                ]
35            )
36        ),
37        batch_size = 32,
38        shuffle = True,
39    )
40    # Start a training worker on this process
41    p = mp.Process(
42        target = run_worker,
43        args = (
44            args.rank,
45            world_size,
46            args.num_gpus,
47            train_loader,
48            test_loader
49        )
50    )
51    p.start()
52    processes.append(p)
53
54 for p in processes:
55     p.join()

```

Listing 3.24: Create a process corresponding to either a parameter server or a worker.

To run the code locally, assuming that it is saved in the `rpc_parameter_server.py` file, we use the following command for the server and each worker that we want to spawn, in separate operating system terminals:

```
python rpc_parameter_server.py -world_size=WORLD_SIZE -rank=RANK
```

For example, for a master node with a world size of 2, the command would be:

```
python rpc_parameter_server.py -world_size=2 -rank=0
```

The worker can then be launched with the following command, issued on a separate terminal:

```
python rpc_parameter_server.py -world_size=2 -rank=1
```

This creates a training session with one server and a single worker.

We can also include in the previous commands the arguments `-master_addr=<ADDRESS>` and `-master_port=<PORT>` to indicate the address and port that the master worker is listening to, for example, to evaluate a scenario where workers and master node run on different machines.

3.3 AllReduce Distributed Training

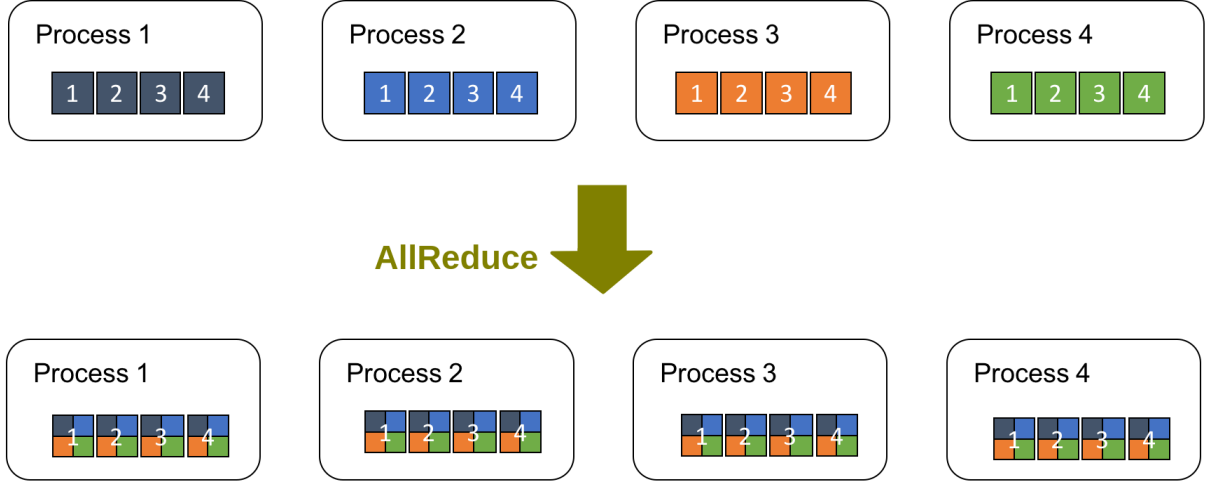


Figure 3.16: AllReduce operation.

As the number of weights grow, parameter server becomes bottleneck, making high performance accelerators under-utilized. Baidu brought ring-based AllReduce algorithm into distributed training. This collective communication pattern came from MPI protocol [69]. The idea is to perform updates without using centralized server, by communicating directly among accelerators.

An AllReduce operation can be decomposed and implemented with ReduceScatter and AllGather MPI operations. Collective communication operations have been analyzed for long time. For example, the cost of an operation can be modeled by $A \times \alpha + B \times N\beta + C \times N\gamma$, where α is the time for transferring a message, β is the time to process one byte of data, γ is the cost to perform the reduction operation per byte, and N is the size of the message. The factor $(A \times \alpha)$ is the latency, $(B \times N\beta)$ is the bandwidth, and $(C \times N\gamma)$ is the computation factor.

For a ring AllReduce operation, factors are as follows: the latency factor is $2(p-1)\alpha$, the bandwidth factor is $2\frac{p-1}{p}N\beta$, the computation factor is $\frac{p-1}{p}N\gamma$, where p is the number of processes. If we have p processes, each data should be partitioned into p chunks, the size of each of which is $\frac{N}{p}$, and $p-1$ data exchange should happen. There are two messages, send and receive, involved for each data exchange, so latency factor is $2(p-1)$. Data is transferred using a logical ring communication pattern, namely $p_0 \rightarrow p_1 \rightarrow \dots \rightarrow p_{p-1} \rightarrow p_p$. The ReduceScatter operation performs such logical ring communication pattern in $p-1$ iterations, resulting in a bandwidth of $\frac{p-1}{p} \cdot N$ for the ReduceScatter operation. Following AllGather operation also transfers data across the logical ring, the amount of data to be transferred is the same as ReduceScatter. Hence, the overall bandwidth factor becomes $2\frac{p-1}{p}N\beta$.

Data parallelism addresses the computational bottleneck of training by distributing batches of training data across multiple GPUs, with each device maintaining a complete copy of the model. This approach enables parallel processing of different data samples while keeping the model architecture unmodified, significantly reducing training time through increased throughput. Figure 3.17 illustrates the distributed data parallel training approach.

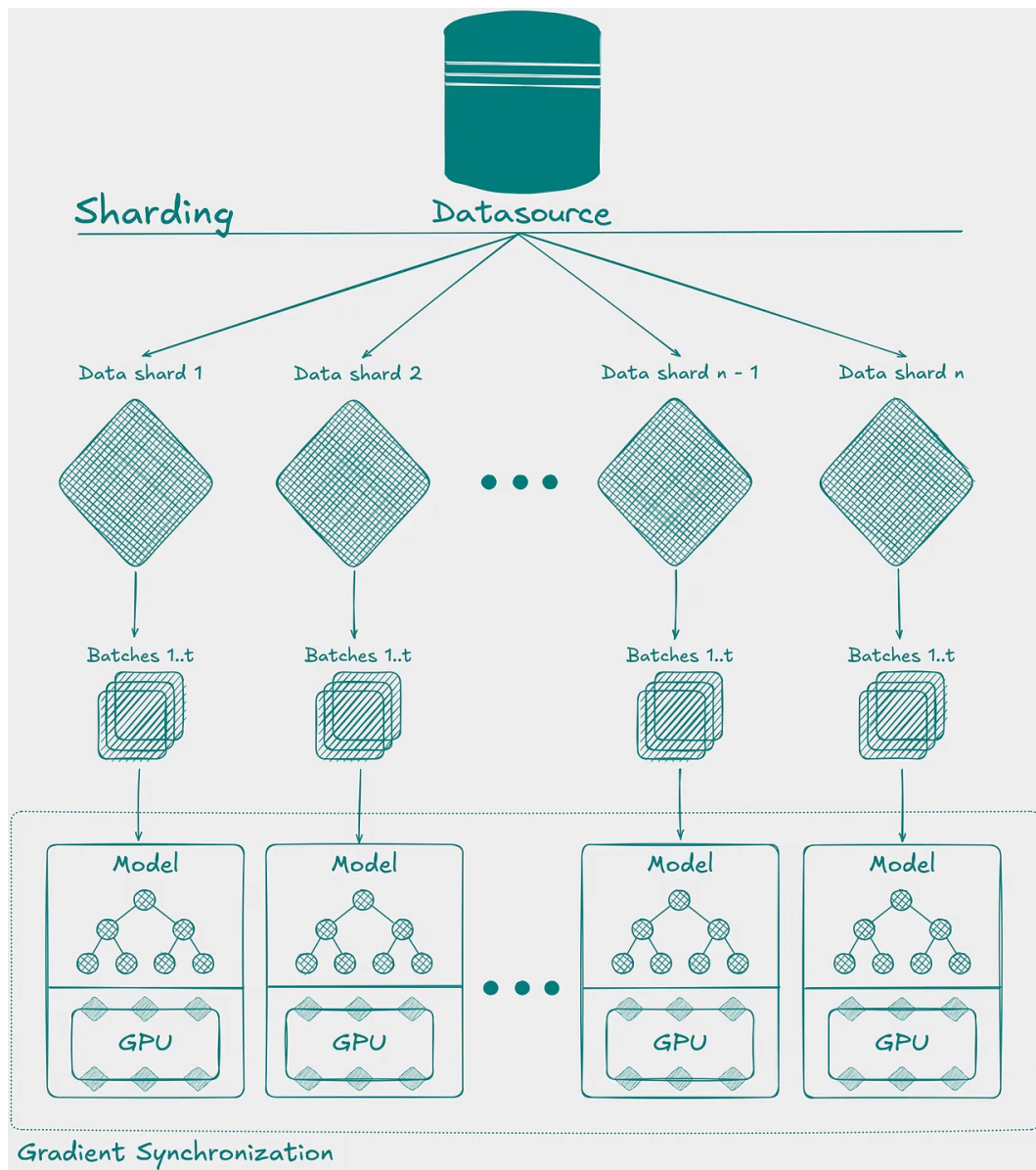


Figure 3.17: An example of distributed data parallel training [188].

Gradients are synchronized across devices after each minibatch backward pass, ensuring all devices have the same aggregate gradients before the weights are updated. During this synchronization, each device's local gradients are combined, and the result is distributed back to all nodes, allowing them to update their model weights consistently.

It is important to note that this strategy is viable only when the entire model can fit in the GPU memory, which can become a critical constraint as model sizes grow. It is ideal for scenarios with large datasets and relatively smaller models, typically under a few billion weights, making it a good choice for many computer vision tasks.

Additionally, when using distributed data parallel training, one must consider that the effective batch size increases with the number of GPUs. This may require adjusting the learning rate and other hyperparameters to maintain model convergence. Different algorithms and communication patterns can be used in gradient synchronization of data parallel training. One popular approach is the ring AllReduce algorithm.

Ring AllReduce

The ring AllReduce is a communication pattern that tries to efficiently solve the problem of combining data across multiple devices or machines. In naive approaches, where all worker nodes send their data to a single master node for reduction, the communication overhead grows linearly with the number of nodes, creating a significant bottleneck that limits scaling. For example, with N worker nodes, each sending D bytes of data,

the master node must receive and process $N \cdot D$ bytes of data while also becoming a network bottleneck as it handles N separate incoming communications (Figure 3.18).

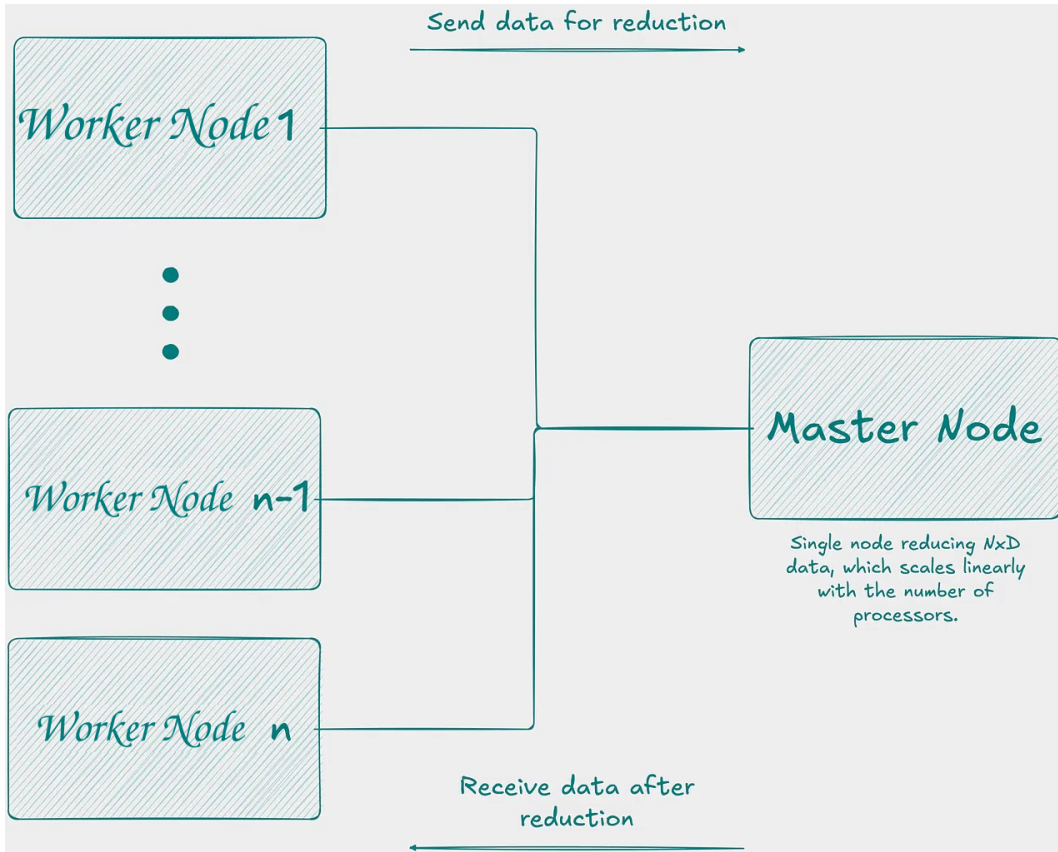


Figure 3.18: Reduction performed by a single master node [188].

The ring AllReduce algorithm arranges worker nodes in a logical ring, where each only communicates with its immediate neighbors. The communication occurs in two phases: ScatterReduce and AllGather. During **ScatterReduce**, each node divides its local data into equal chunks, one per worker node in the ring, and progressively shares and reduces these chunks as they travel around the ring (Figure 3.19). Considering $N=4$ nodes, there will be $N - 1=3$ ScatterReduce steps. In the first step, node 0 sends chunk 0 to node 1, in the second step sends chunk 3 to node 1, and in the third step sends chunk 2 to node 1. In steps 1 to 3, node 1 sends chunks 1, 0, and 3 to node 3. In steps 1 to 3, node 2 sends chunks 2, 1, and 0 to node 3. Finally, in steps 1 to 3, node 3 sends chunks 3, 2, and 1 to node 0. After receiving a chunk, each node reduces this chunk with its local chunk in the same position. Thus, when the ScatterReduce phase is finished, node 0 has chunk 1 completely reduced, node 1 has chunk 2 completely reduced, node 2 has chunk 3 completely reduced, and node 3 has chunk 0 completely reduced.

In the subsequent **AllGather** phase, the worker nodes circulate the reduced chunks around the ring, with each worker collecting a copy of all chunks. After $N - 1$ steps, where N is the number of workers, every worker obtains a complete copy of the fully reduced chunks (Figure 3.20).

The total data transferred in ring AllReduce can be expressed as $\frac{2(N-1)K}{N} \approx 2K$. This expression reflects the algorithm's two-phase structure: $N - 1$ steps for ScatterReduce and $N - 1$ steps for AllGather, giving us the $2(N - 1)$ factor. In each step, nodes communicate chunks of size K/N , as the original data of size K is divided into N equal chunks. The expression simplifies to $2K$, showing that the transferred data remains constant regardless of the number of nodes. Given optimal GPU topology arrangement, the algorithm achieves bandwidth optimality when latency costs are negligible compared to bandwidth constraints.

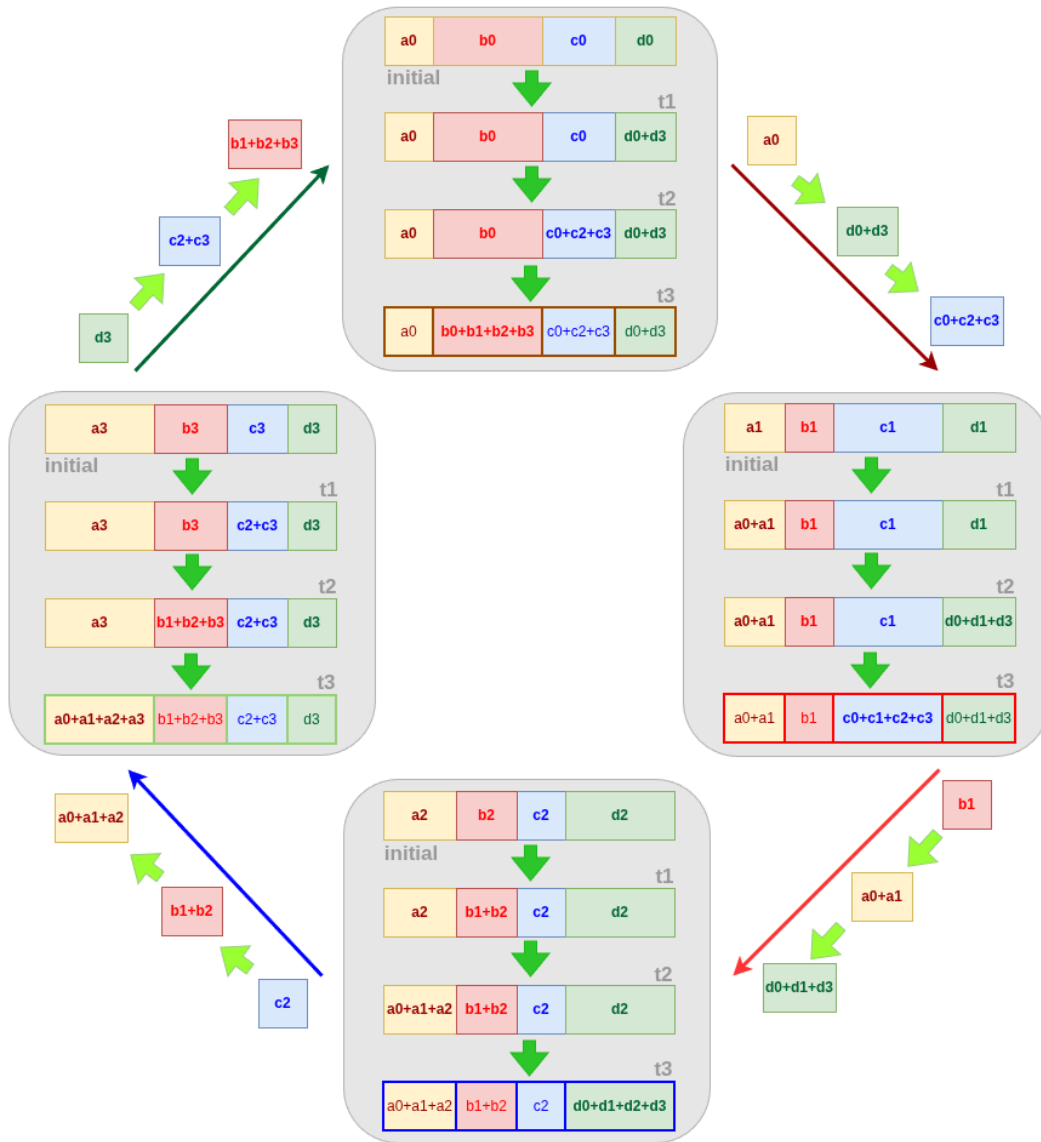


Figure 3.19: Visualization of the ScatterReduce phase with four worker nodes.

3.3.1 Latency of Ring AllReduce

For large distributed systems, latency is not negligible. While ring AllReduce achieves optimal bandwidth utilization under ideal conditions, its latency profile creates scaling limitations. For each synchronization round, the algorithm must perform $2(N - 1)$ sequential communication steps, where N represents the number of participating nodes. This creates a linear relationship between latency and node count that becomes challenging in large-scale training clusters. Given how computationally expensive network operations are, the cumulative effect of these communication steps can substantially impact overall training performance as more nodes join the cluster.

3.4 Tree AllReduce

The tree AllReduce is a distributed algorithm that uses a binary tree topology to combine data across multiple nodes efficiently (latency-wise). In Figure 3.21, nodes at the top send data down to their parent nodes, which combine the received values. The process takes $\log_2 N$ steps, making its latency dramatically lower than the ring AllReduce N steps, with 1024 nodes, tree AllReduce completes in just 10 communication steps versus ring reduce's 1024 steps. Here is how tree AllReduce works.

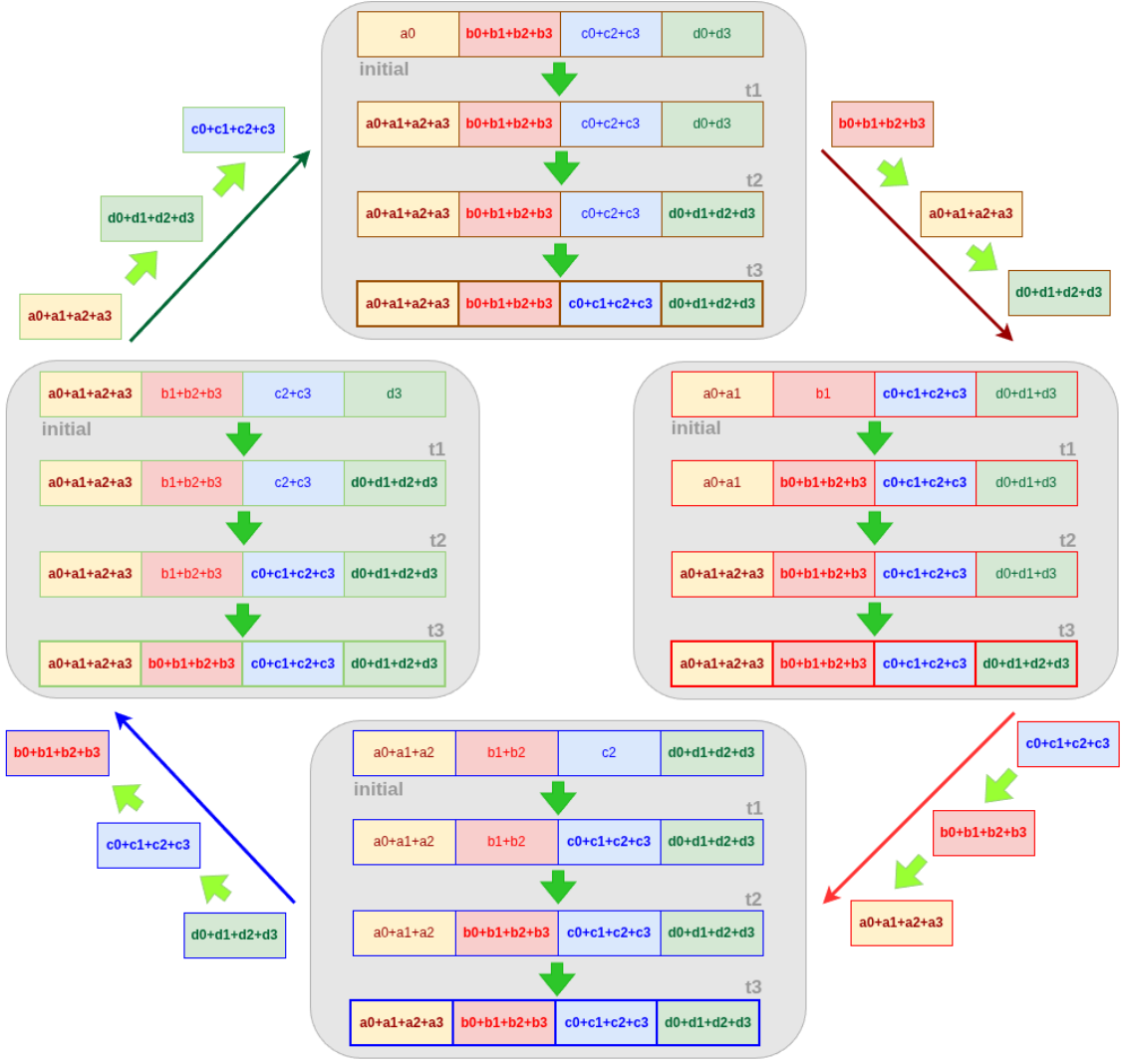


Figure 3.20: Visualization of the AllGather phase with four worker nodes.

Reduce phase: (1) Leaf nodes send their values to their parent nodes, (2) Parent nodes combine received values and send to their parents, (3) This continues until root node performs final reduction.

Broadcast phase: (1) Root node sends result to its two children, (2) Each parent broadcasts received value to its two children, (3) Process continues until reaching the leaf nodes.

The total latency is $2 \cdot \log_2 N \cdot \text{steps} - \log_2 N$ for reduce and $\log_2 N$ for broadcast.

The key limitation is bandwidth utilization. Leaf nodes, which constitute half of all nodes, use their network links in a half-duplex manner. Although they send data during the reduce phase and receive data during the broadcast phase, they never fully leverage their full-duplex capacity at any single point in time. While pipelining can help internal nodes overlap sending and receiving operations, it does not address the fact that leaf nodes alternate between sending and receiving, rather than performing both simultaneously, thus leaving potential bandwidth underutilized.

3.5 Two-Tree AllReduce

Two-tree AllReduce [172] is an algorithm that tackles the bandwidth underutilization in simple tree reductions by starting with a complete binary tree and then constructs a complementary tree. In one tree, a node acts as a leaf and, in the other, it becomes an internal node. By shifting or mirroring the numbering of the original complete tree to form the second tree, the internal nodes of one tree align with the leaves of the other, creating an interwoven communication pattern (Figure 3.22).

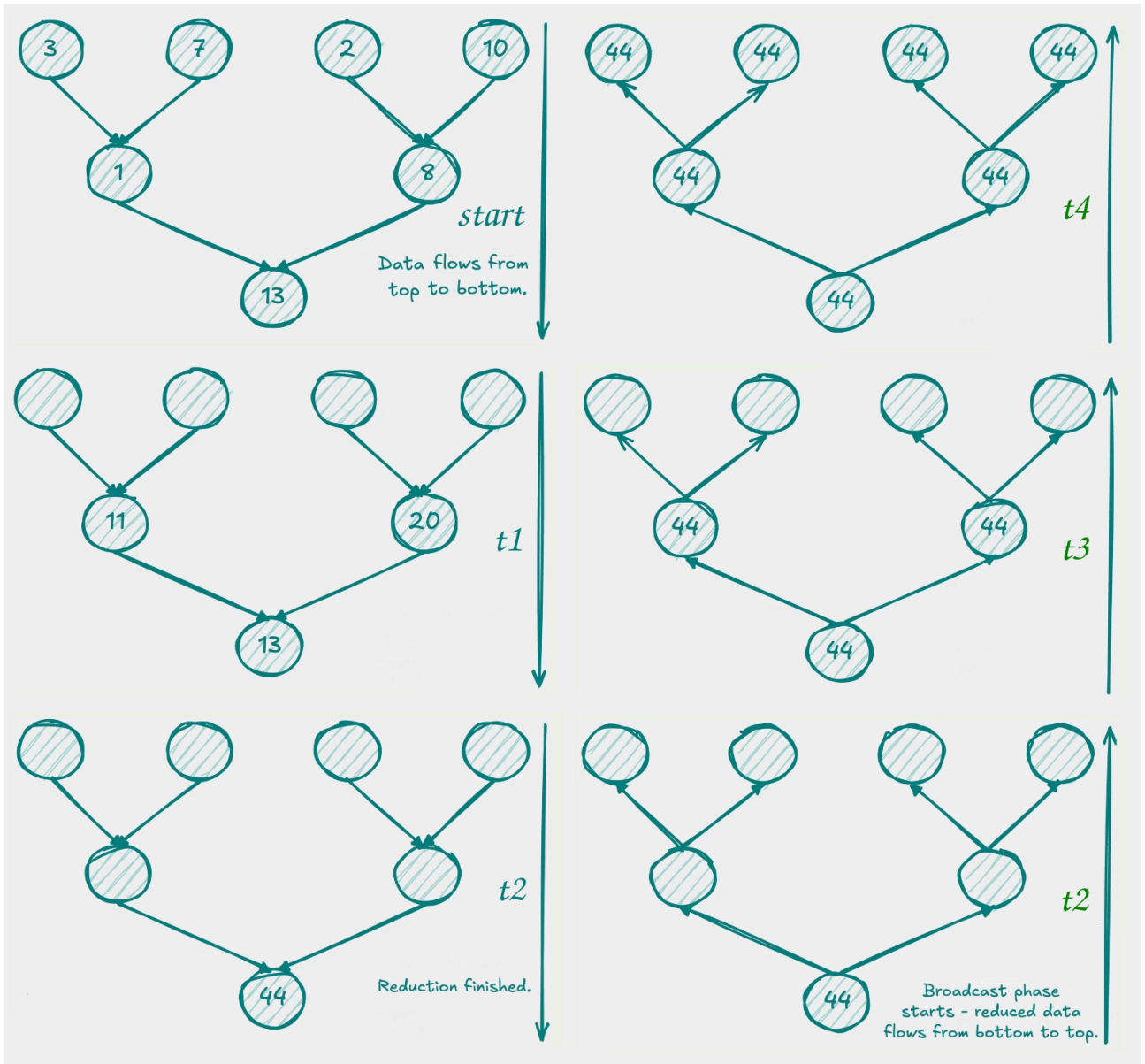


Figure 3.21: The tree AllReduce algorithm visualization [188].

After constructing these two complementary trees, the data is split into two halves. Except for the initial steps, when the data is being first sent, and the final steps, when the last parcels of data are sent, each intermediate communication phase allows every node to both send and receive simultaneously, nearly doubling the effective bandwidth usage compared to a single-tree approach. While not every stage can perfectly utilize all bandwidth due to these initial and final steps, the vast majority of steps achieve full-duplex efficiency. At the same time, the algorithm still operates in a logarithmic number of steps, like a standard tree AllReduce, maintaining low latency overall.

The two-tree AllReduce combines the $O(\log N)$ latency advantages of tree-based methods with high-bandwidth efficiency, making it an attractive algorithm for large-scale distributed training environments.

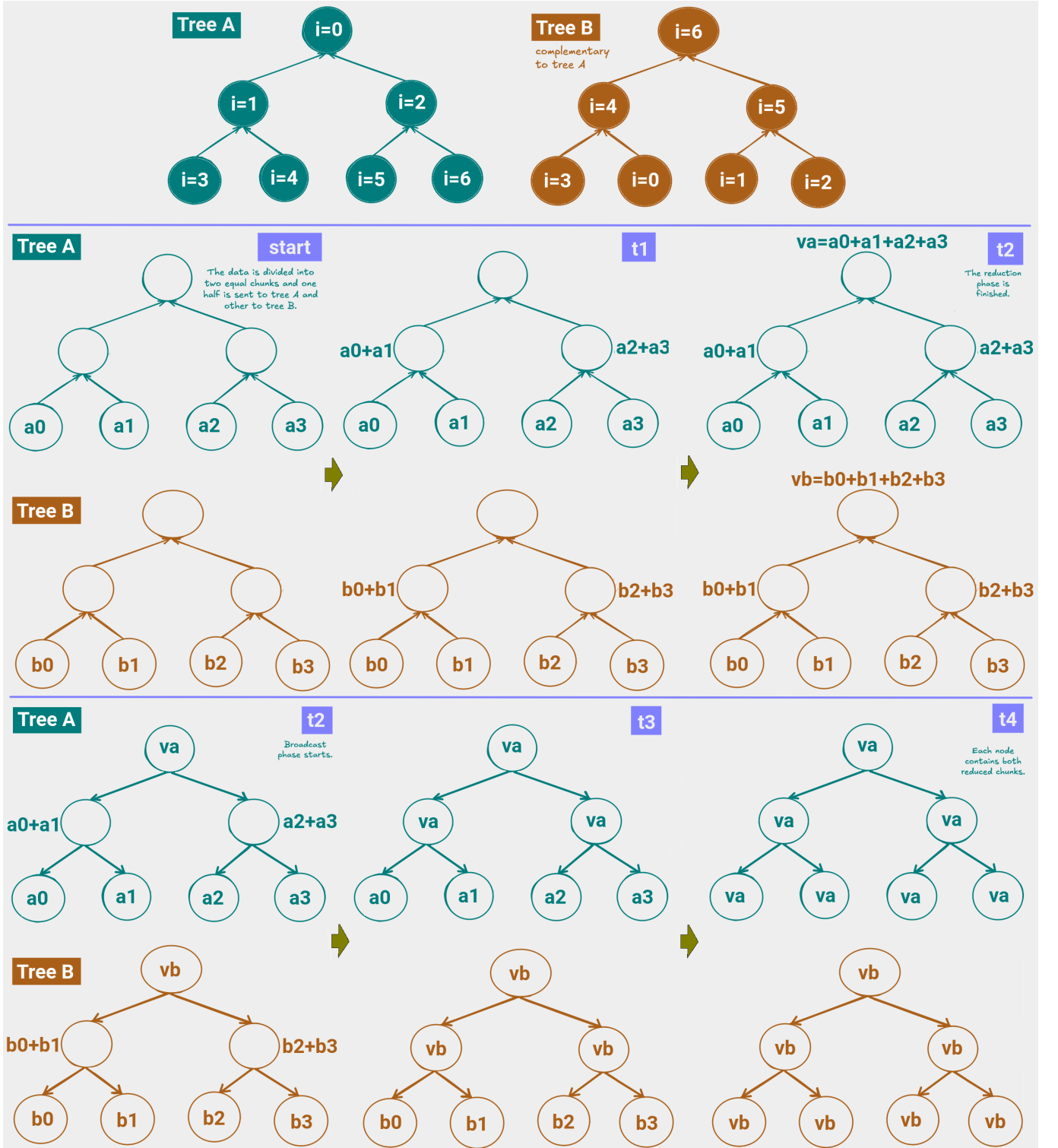


Figure 3.22: The two-tree AllReduce algorithm visualization [188].

3.6 Performance Impact

In [96], NVIDIA documents the results of evaluating the impact on communication performance of different AllReduce algorithms, including ring AllReduce, hierarchical ring AllReduce, and two-tree AllReduce. Their findings are summarized in Figures 3.23, 3.24 and 3.25. Tree AllReduce allows for much lower latency than ring AllReduce. For larger number of GPUs, tree AllReduce also achieves much higher bandwidth utilization than ring AllReduce. When the number of GPUs gets larger, the training speed for trees, measured by the number of images consumed during a certain interval, also gets higher by a greater margin than for rings and hierarchical rings.

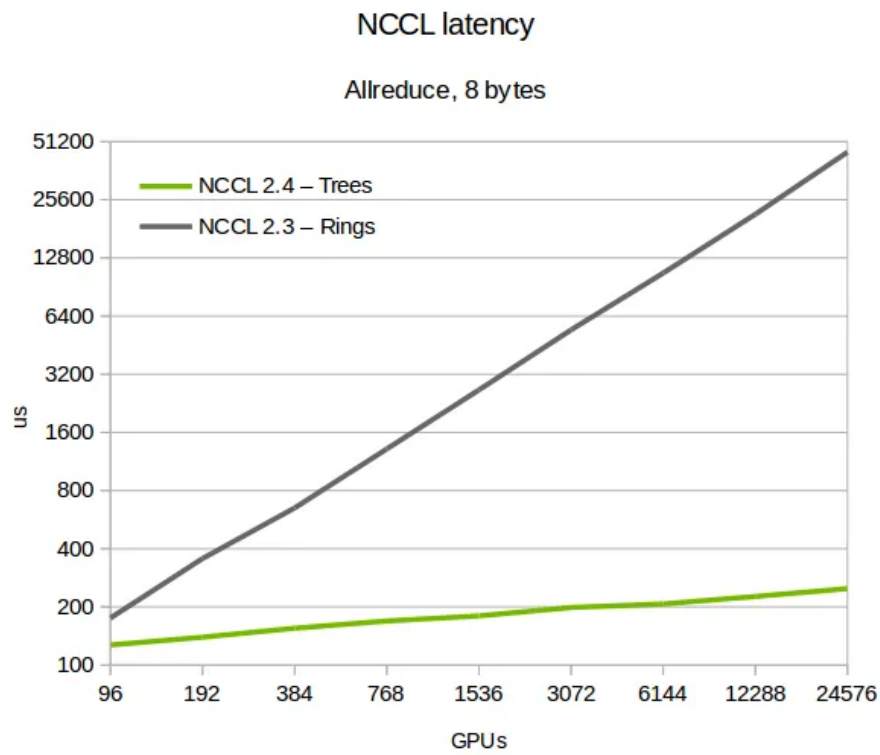


Figure 3.23: NVIDIA Collective Communications Library (NCCL) latency for different AllReduce algorithms [96].

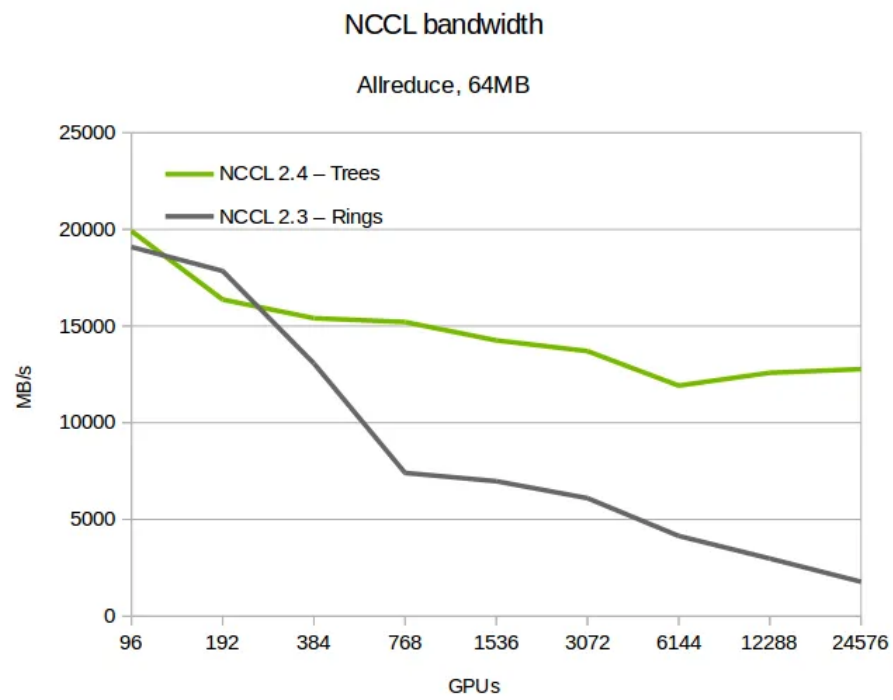


Figure 3.24: NCCL bandwidth for different AllReduce algorithms [96].

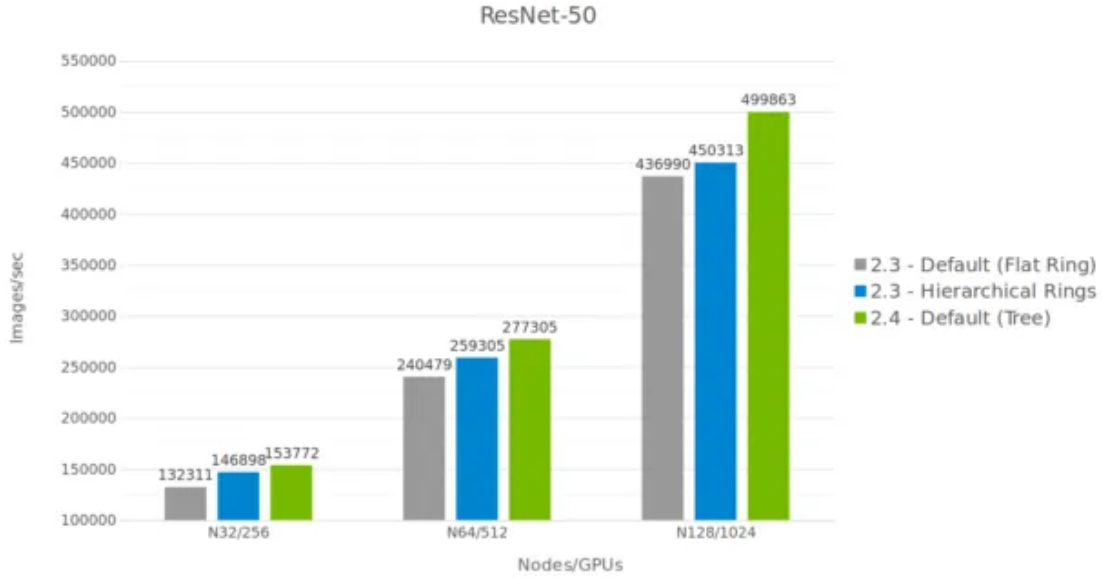


Figure 3.25: Model performance training comparison on ResNet50. The performance gap between the compared approaches widens as the number of GPUs scales up [96].

3.7 Zero Redundancy Optimizer Parallelism

Data parallelism is efficient, but also redundant. Every device, often a GPU, stores the same weights, gradients, and optimizer state, leading to significant memory duplication. For small models, this is fine. For billion-parameter models, it is unsustainable. That is where Zero Redundancy Optimization (ZeRO) comes in [164]. ZeRO eliminates this duplication by sharding model state across GPUs: in stage 1 it is sharded the optimizer state, in stage 2 it is also sharded the gradients, and in stage 3 it is also sharded the model weights. This means each GPU holds only part of the total training state, reducing memory use drastically. Combined with offloading techniques, to CPU or disk, ZeRO enables models that once exceeded memory limits to train efficiently.

The ZeRO approach is organized into three possible optimization stages.

- ZeRO-1: shard the optimizer state across nodes,
- ZeRO-2: shard the optimizer state and gradients across nodes,
- ZeRO-3: shard the optimizer state, gradients and model's weights across nodes.

The activations are not in the list of elements that we shard. Since each model replica receives a different microbatch, the activations on each rank also differ, so they are not duplicated and thus cannot be sharded. We now investigate how much we save with the partitioning strategy adopted by each ZeRO stage.

3.7.1 Memory Usage

Consider that the model's weight count is ψ . In mixed precision training, with the Adam optimizer, for each weight we need to store the following information in memory.

- Model's weights in half precision (BF16/ FP16): 2ψ ,
- Model's gradients in half precision (BF16/ FP16): 2ψ ,
- Model's weights and optimizer state (first and second moments) in FP32: $4\psi + (4\psi + 4\psi)$,
- Model's gradients in FP32 (optional): 4ψ , only included if we want to accumulate gradients in FP32.

If we do not accumulate gradients in FP32, this gives us a total memory consumption of $2\psi + 2\psi + 12\psi$, and if we do it gives us $2\psi + 6\psi + 12\psi$. Let us focus for now on the case without FP32 gradient accumulation for

simplicity.

The idea of ZeRO is to shard these objects across the nodes, with each node only storing a slice of the objects. These slices are then reconstructed when and if needed, thereby dividing memory usage by the data parallelism degree N_d .

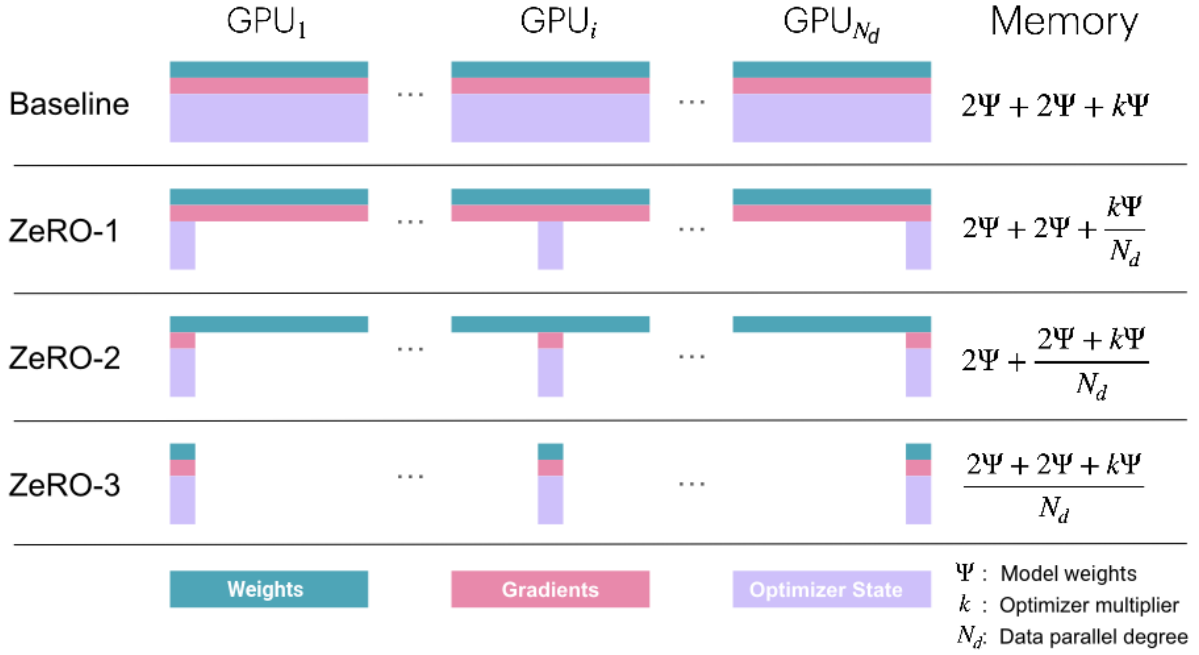


Figure 3.26: Memory usage for the three ZeRO stages [194].

Figure 3.26 summarizes the memory required by the three ZeRO stages. The required memory is $2\psi + 2\psi + k\psi$ in the data parallel baseline, $2\psi + 2\psi + \frac{k\psi}{N_d}$ in ZeRO-1, $2\psi + \frac{2\psi + k\psi}{N_d}$ in ZeRO-2, and $\frac{2\psi + 2\psi + k\psi}{N_d}$ in ZeRO-3. Here, ψ denotes the number of weights, k denotes the memory multiplier of optimizer state, as we have seen it is $k = 12$ for Adam, and N_d denotes the data parallelism degree. If we are using FP32 gradient accumulation with ZeRO-2 or ZeRO-3, we would need to add an additional $\frac{4\psi}{N_d}$ to the gradient term. Let us explain the memory requirements by exploring how each ZeRO stage works. We start with ZeRO-1.

3.7.2 ZeRO-1: Partitioning the Optimizer State

In vanilla data parallelism, all ranks gather the same gradients after the backward pass and simultaneously perform identical optimizer steps. This represents a lot of duplication, and as so we must try to avoid duplication and reduce the memory usage.

In ZeRO-1, the optimizer state is partitioned into N_d equal parts, where N_d is the parallelism degree. This means that each model replica allocated to a data parallel rank, only keeps track of $\frac{1}{N_d}$ of the optimizer state, and during the optimization step, only $\frac{1}{N_d}$ of the FP32 weights are updated. However, during the forward pass, each replica needs all the weights. Thus, we need to add an additional AllGather operation after the optimizer step so that each model replica has the full set of updated weights.

This explains the memory usage $2\psi + 2\psi + \frac{k\psi}{N_d}$ that we saw in Figure 3.26. Here is a summary of the sequence of operations for a single training step with ZeRO stage 1.

1. Perform a forward pass with the same full set of BF16 weights on each replica, but different micro-batches across replicas.
2. Perform a backward pass with the same full set of gradients on each replica, but different micro-batches across replicas.
3. Perform a ReduceScatter operation on the gradients to get a reduced $\frac{1}{N_d}$ parcel of the gradients on each rank.

4. Each replica performs an optimizer step on its local parcel of the optimizer state, corresponding to $\frac{1}{N_d}$ of the optimizer state, to get $\frac{1}{N_d}$ of the FP32 updated weights, which can then be converted to $\frac{1}{N_d}$ of the full set of BF16 weights.
5. Perform an AllGather operation on the BF16 weights to send the missing slices to each replica.

ReduceScatter is two times faster than AllReduce. Figure 3.27 clarifies the ReduceScatter operation, detailing all the steps of the forward and backward passes. Compared to vanilla data parallelism, ZeRO-1 changes the AllReduce gradient communication to a ReduceScatter operation and adds an AllGather operation over all weights after the optimizer step, as illustrated in Figure 3.28.

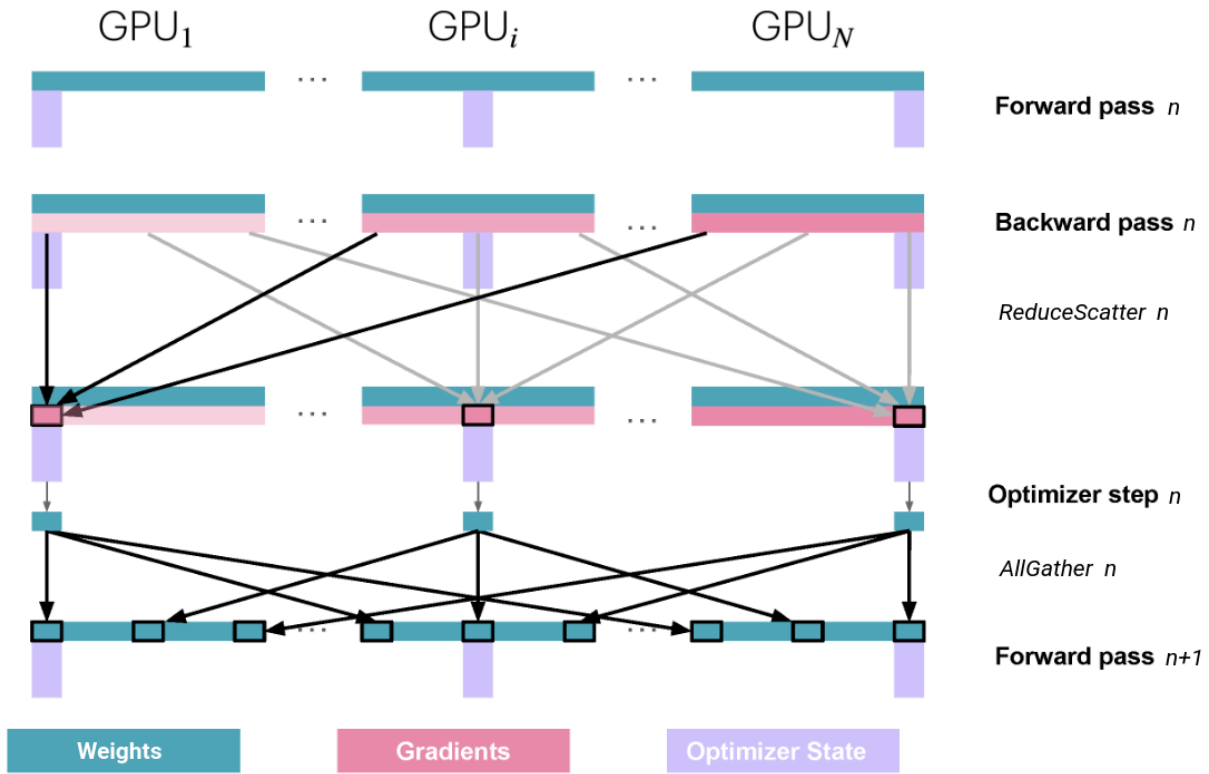


Figure 3.27: All the steps of forward and backward passes in ZeRO-1 [194].

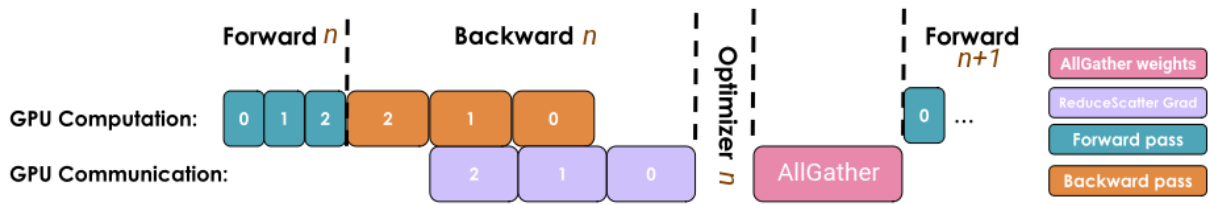


Figure 3.28: The communication pattern in ZeRO-1 [194].

We have seen in data parallel that we can overlap the AllReduce gradient communication with the backward pass computation. In ZeRO-1, we can also investigate how to efficiently overlap the newly added AllGather of the BF16 weights. There are two main strategies for this: (i) **during the optimizer step**, since we can initiate the AllGather immediately after the optimizer updates the first slice of weights, it allows the communication to potentially overlap with the updating of the other weights, (ii) **during the forward pass**, since we can overlap the AllGather of each layer's weights with the forward pass.

Unfortunately, these techniques are not straightforward to implement and require sophisticated use of hooks and bucketing. In practice, we can just use PyTorch's native ZeRO-3 implementation (FSDP2).

In ZeRO-1, the optimizer state has been partitioned, which means that each replica only updates $\frac{1}{N_d}$ of the state. In this way, there is no real need to have all the gradients on all ranks, as only a subset of these gradients

is needed for the optimization step. This take us into stage 2 of ZeRO.

3.7.3 ZeRO-2: Adding Gradient Partitioning

Since each replica only needs the gradient shard corresponding to its optimizer state shard, it makes sense to shard gradients as well, similarly to the optimizer state. Then, during the backward pass, instead of performing an AllReduce over the gradients, we only perform a ReduceScatter operation, the same that is done in ZeRO-1. The difference is that here, we only store in memory the $\frac{1}{N_d}$ parcel of the gradients that is needed by this rank, thus saving more memory compared to ZeRO-1. In the case of FP32 gradient accumulation, we only need to keep $\frac{1}{N_d}$ of the FP32 gradients used to accumulate the BF16 gradients coming from the ReduceScatter operation. And in the optimizer step, these $\frac{1}{N_d}$ FP32 gradients are used to update the local shard of the optimizer state. The forward and backward passes of ZeRO-2 are depicted in Figure 3.29.

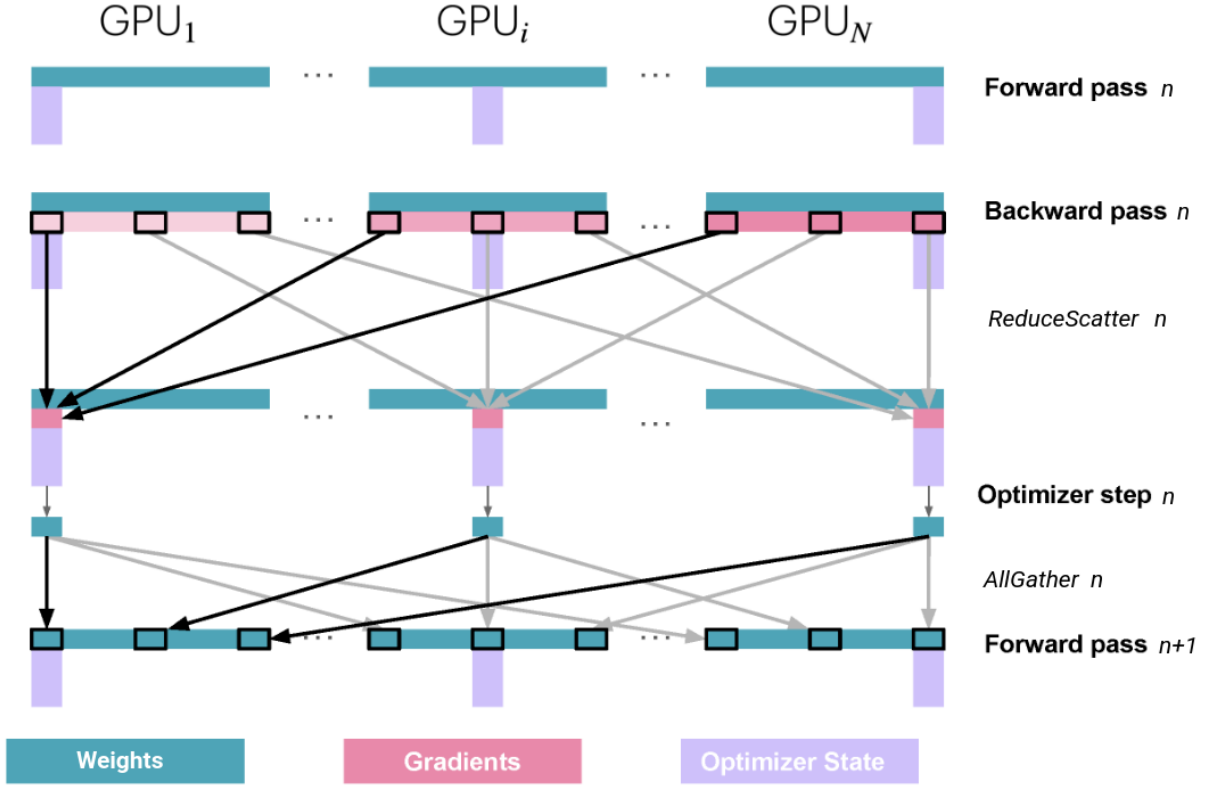


Figure 3.29: All the steps of forward and backward passes in ZeRO-2 [194].

It is easy to see that sharding the gradients leads to a memory requirement of $2\psi + \frac{2\psi + k\psi}{N_d}$, and as N_d is increased, we can use $8\times$ less memory than the baseline. In terms of communication, ZeRO-2 follows the same pattern as ZeRO-1, with the only difference being that we communicate and release memory on the fly: they both require a ReduceScatter operation for the gradients and an AllGather operation over all weights (Figure 3.30). ZeRO-2 is thus also equivalent to vanilla data parallel training with regard to communication.

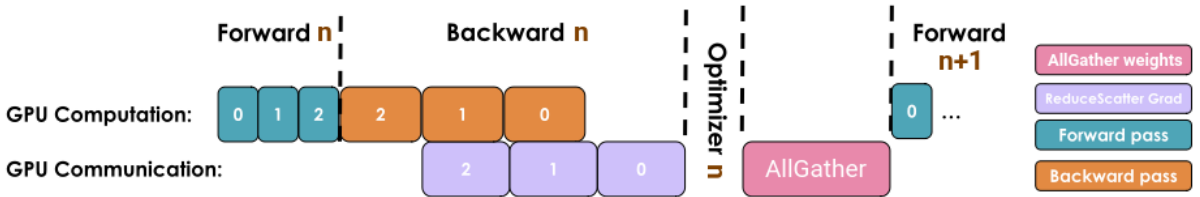


Figure 3.30: The communication pattern in ZeRO-2 [194].

We conclude that there is no real overhead to using ZeRO-2 over ZeRO-1 besides an higher implementation complexity, and indeed ZeRO-2 is usually a better option.

3.7.4 ZeRO-3: Adding Weight Partitioning

For stage 3, we extend the approach of sharding the optimizer state and gradients over model replicas to also shard the model's weights. PyTorch's native implementation of this stage is called Fully Sharded Data Parallelism (FSDP).

So, how do we do a forward and backward pass in practice if the weights of the model are distributed? We gather them on demand when they are necessary. In the forward pass, this is done as depicted in Figure 3.31.

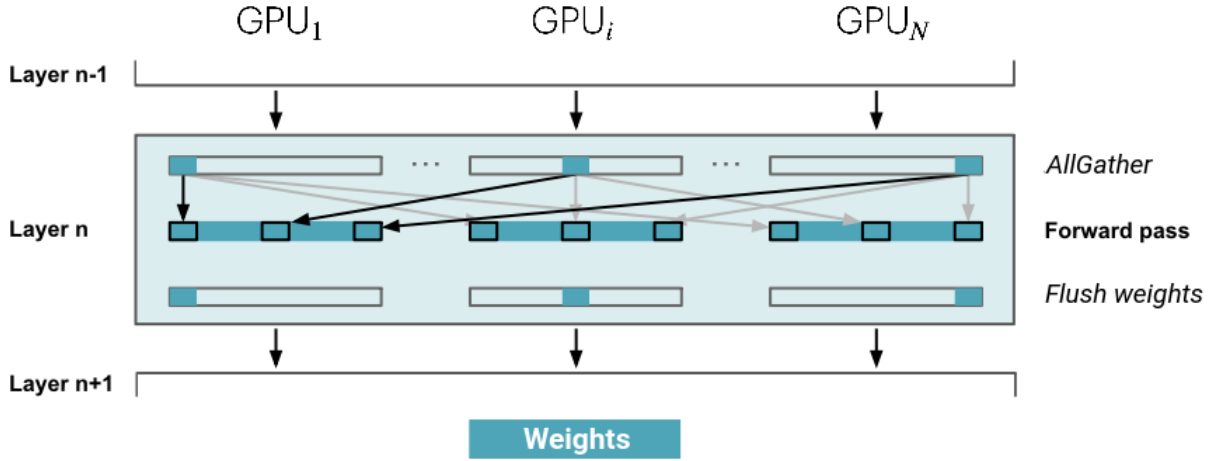


Figure 3.31: The forward pass with ZeRO-3 [194].

As we perform the forward pass and sequentially go through the layers, we retrieve the necessary weights on demand and immediately flush them from memory when we do not need them anymore. The backward pass works the same way, just inverted in terms of sequence. Figure 3.32 illustrates how we produce the gradient shards in the backward pass.

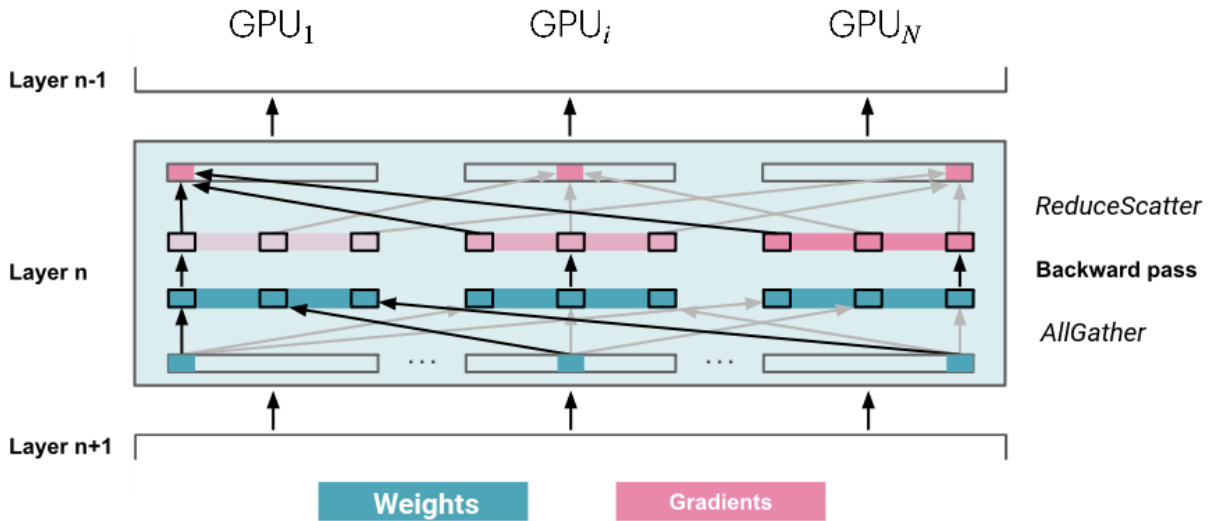


Figure 3.32: The backward pass with ZeRO-3 [194].

The other issue is that we need to run the AllGather operations continuously throughout the forward and backward pass in a training step, which amounts to $2 \cdot \text{num_layers} - 1$ additional AllGathers in a training step compared to ZeRO-2. Each operation has a small base latency overhead, as we can see in Figure 3.33, due to AllGather being slower than the forward pass plus freeing weights.

During the forward pass we run AllGather operations for the weights when we need them, so there is a ψ communication cost. Since we discard the weights immediately after we use them in the forward pass, we need one more AllGather during the backward pass as well, incurring in another ψ communication cost. Finally, we

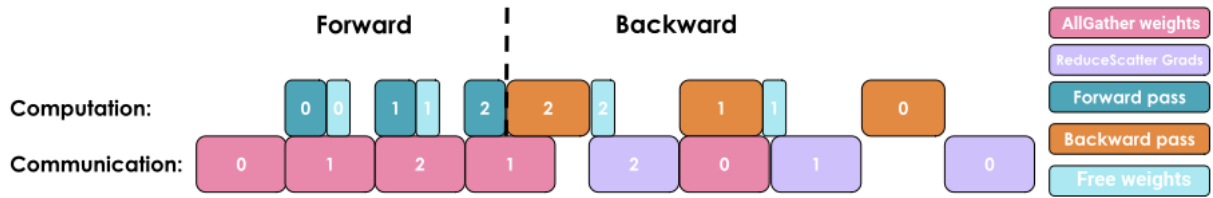


Figure 3.33: The communication pattern with ZeRO-3 [194].

need the same ReduceScatter operation as in ZeRO-2 for the gradients, which also has a ψ communication cost. So, we arrive at a total communication cost of 3ψ , compared to 2ψ for ZeRO-2.

This may sound like a lot of communication overhead, but it is actually not a big deal, as we can overlap the communication of the weights for the next layer with the forward pass of the current layer, which is called prefetching. With prefetching, we can perform the AllGather over the weights for layer $n + 1$ while we do the forward pass for layer n , and similarly, we run the AllGather over the weights for layer $n - 1$ while doing the backward pass for layer n . Of course, this overlap only works as long as we do not scale data parallelism too much. As a rule of thumb, data parallelism degree should not exceed 512.

In terms of memory, we can see that our requirement has reached the value of $\frac{2\psi + 2\psi + k\psi}{N_d}$, which means we can theoretically drive memory usage down indefinitely if we can increase the data parallelism degree, at least for the model-related parameters. Notice that it does not help with the intermediate activations. For that, we can use activation checkpointing and gradient accumulation.

We can now summarize what we have discussed about data parallelism and ZeRO. We have seen that we can increase the throughput of training significantly with data parallelism, simply scaling training by adding more model replicas. With ZeRO, we can train even models that would ordinarily not fit into a single GPU by sharding the weights, gradients, and optimizer state across nodes, while incurring into a small communication cost.

However, there are some limits here: data parallelism only works if a layer of the model fits in a GPU, and ZeRO can only partition the weights, gradients, and optimizer state, not the activation memory. For an LLM, the activation memory scales with sequence length and batch size. We could just limit those, but in practice we do not want to be limited by hardware to train only with a short sequence length (Figure 3.34). To overcome this issue, we have to explore an orthogonal axis of parallelism, tensor parallelism. Unlike ZeRO-3, which relies on heavy weight communication, tensor parallelism proposes to shard weights, gradients, optimizer state, and activations across nodes without requiring any communication of model weights between GPUs.

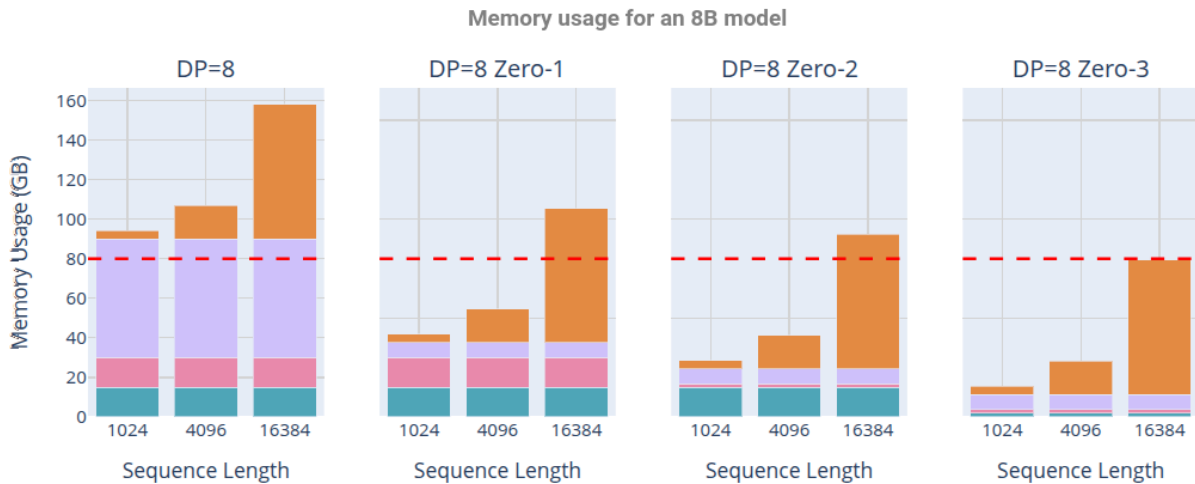


Figure 3.34: Memory usage on an 8B model [194].

3.8 Fully Sharded Data Parallel

While ZeRO [164] revolutionized memory optimization, Fully Sharded Data Parallel (FSDP) further generalized it. FSDP is a memory-efficient technique designed to train massive models by distributing memory usage across GPUs. First introduced in [164], FSDP extends traditional data parallelism by splitting model weights, gradients and optimizer state by the workers, which are often GPUs. Unlike conventional distributed data parallel (DDP) approaches, where each GPU stores a full copy of the model, FSDP ensures each GPU holds only a fraction of the model, a **shard**, significantly reducing per-device memory requirements while preserving training efficiency. To further maximize memory efficiency, FSDP can offload the weights, gradients and optimizer state to CPUs when the instance is not active in the computation. The FSDP approach is standard in frameworks like PyTorch and DeepSpeed for large-scale training.

As in ZeRO stage 3, FSDP employs the following three partitioning/sharding steps.

1. Optimizer state partitioning (P_{os}): Optimizer state, defined by the momenta or variance, is divided across N_d GPUs, each one storing and updating only $\frac{1}{N_d}$ of the total state. This partitioning significantly reduces memory usage by lowering the per-GPU optimizer state requirements. After each training step, an AllGather operation [43] synchronizes the updated weights across all data-parallel processes, ensuring consistency.
2. Gradient partitioning (P_g): Gradients are sharded across GPUs during backward pass. The gradients for each weight partition are reduced via ReduceScatter and sent to the GPU responsible for that partition. Once the gradients are no longer needed, they are released, minimizing memory overhead for each shard.
3. Weight partitioning (P_p): Model weights are split across GPUs, where each GPU stores $\frac{1}{N_d}$ of the total weights. Weights required by other GPUs are broadcasted during the forward and backward passes. This adds an approximately $1.5\times$ communication overhead but delivers substantial memory savings.

An intuitive way to understand FSDP is by comparing it to traditional distributed data parallelism (DDP), which is illustrated in Figure 3.35. In DDP, each GPU stores a complete copy of the model and processes a portion of the data. Gradients are averaged across all GPUs using an AllReduce operation before the weight's update step, ensuring that every GPU updates an identical model copy.

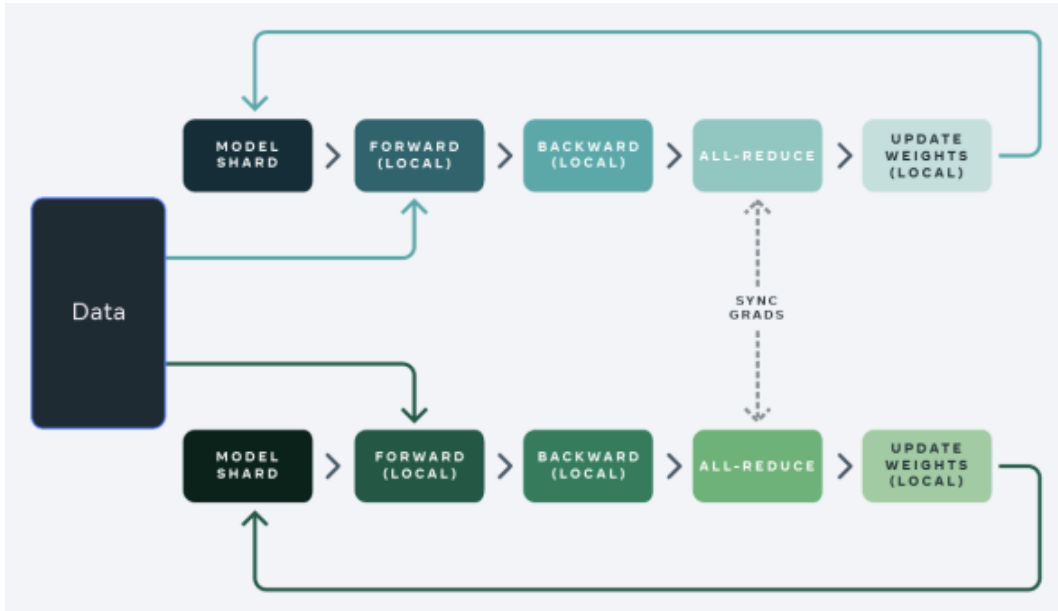


Figure 3.35: Traditional distributed data parallel training with two processes [153].

In contrast, FSDP distributes not just the data but also the model itself, with each GPU responsible for only a shard of the weights, a shard of the optimizer state, and a shard of the gradients. To perform the forward pass, a GPU temporarily gathers the full model weights from other GPUs via an AllGather operation (Figure 3.36). Before the backward pass, the necessary weights are gathered again through a second AllGather operation.

After the backward pass, a ReduceScatter operation averages and distributes the gradients, ensuring each GPU updates its local portion of the model with an aggregated version of the necessary gradients. The ReduceScatter and AllGather operations mirror the DDP AllReduce, allowing FSDP to shard data, the entire model, and optimizer state, saving memory (Figure 3.37).

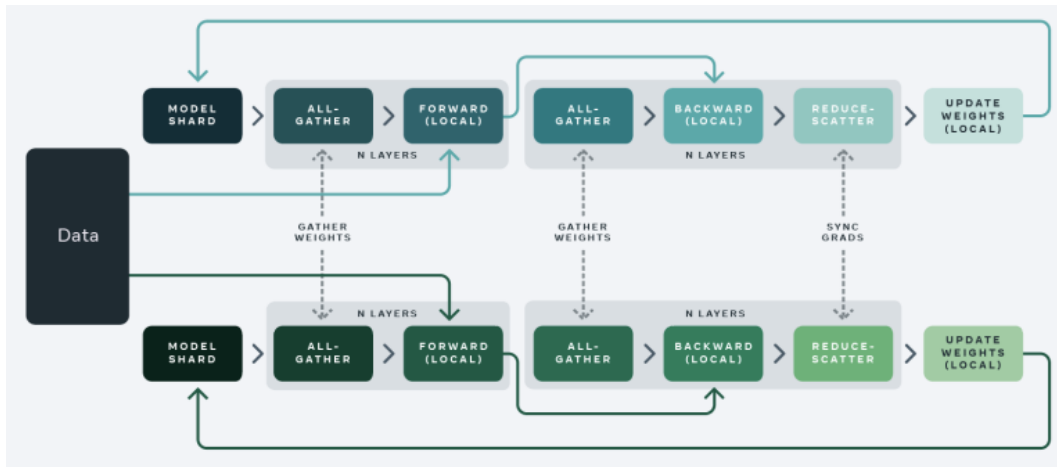


Figure 3.36: Fully-sharded data parallel training with two processes [153].

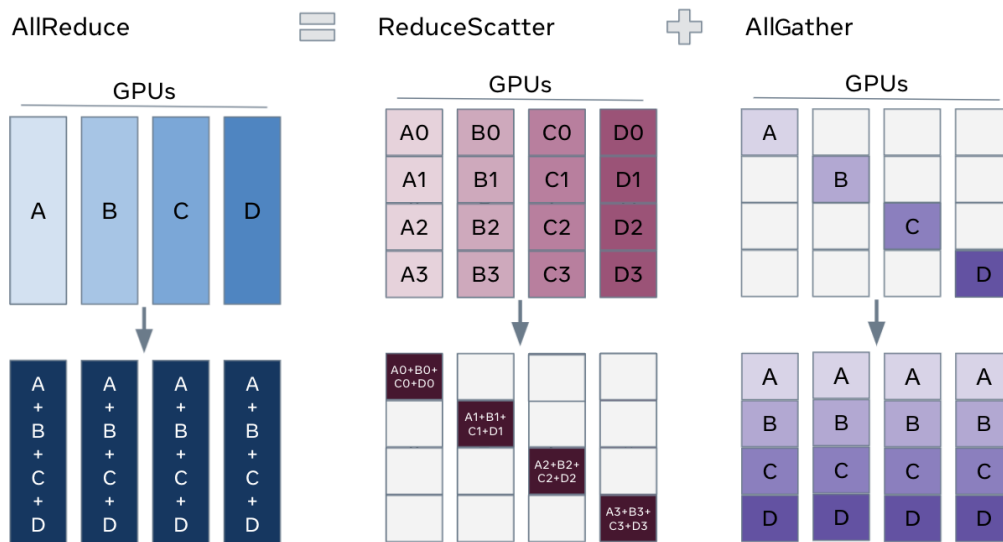


Figure 3.37: AllReduce DDP operation decomposed into FSDP ReduceScatter and AllGather operations [153].

Unlike DDP, where each GPU maintains and updates a full model replica, FSDP only synchronizes the model shards. The full model aggregation occurs temporarily through AllGather operations, solely when required for forward and backward computations, being discarded afterward.

FSDP1 vs. FSDP2

As we discussed before, Fully Sharded Data Parallel distributes model weights, gradients, and optimizer state across multiple GPUs, allowing for training models that exceed the memory capacity of a single device. The memory corresponding to the weights of the layer being processed will be freed when the forward pass through it is finished. AllGather operations are used to gather necessary data from different GPUs during model training. Including CPU offload in FSDP allows for keeping weights on the CPU and move them to the GPU when the forward and backward passes are executed. Thus, the optimizer updates will be done on CPU, so the optimizer state stays there too.

A common goal to FSDP1 and FSDP2 is to make AllGather operations efficient, considering the constraint imposed by the NCCL library which requires that each GPU contributes with a tensor of the same size in the

collective operations. FSDP1 flattens and concatenates all tensors before splitting them, potentially leading to type and metadata conflicts (Figure 3.38). In the example portrayed on the figure and to keep the tensor shards of the same size, after the weight tensors were flattened, tensor $t1$ (of size (2, 3)) and the first part of $t2$ (of size (3, 3)) are combined into a single tensor shard, and the second part of tensor $t2$ is combined with $t3$ (of size (2, 2)) into a single tensor shard. Thus, tensor $t2$ is split between GPUs, which is not ideal as we will see.

The FSDP1 approach used to shard tensors has its advantages, such as the flattened and concatenated tensor shards occupy a contiguous region in memory and we can use views to easily restore the original tensors ($t1$, $t2$, $t3$). A disadvantage in having a tensor weight ($t2$) split by two GPUs, is that this complicates the forward and backward passes over the layer (L2) whose weights are split between GPUs. Recall that after performing the forward pass over a layer (L1), the weights ($t1$ and part of $t2$) are freed from memory, but the partial $t2$ is required by the other GPU. FSDP1 forces tensors, for example $t1$, $t2$, and $t3$, to all have the same data type and other metadata. But this can lead to non-deterministic memory spikes, making the memory management challenging.

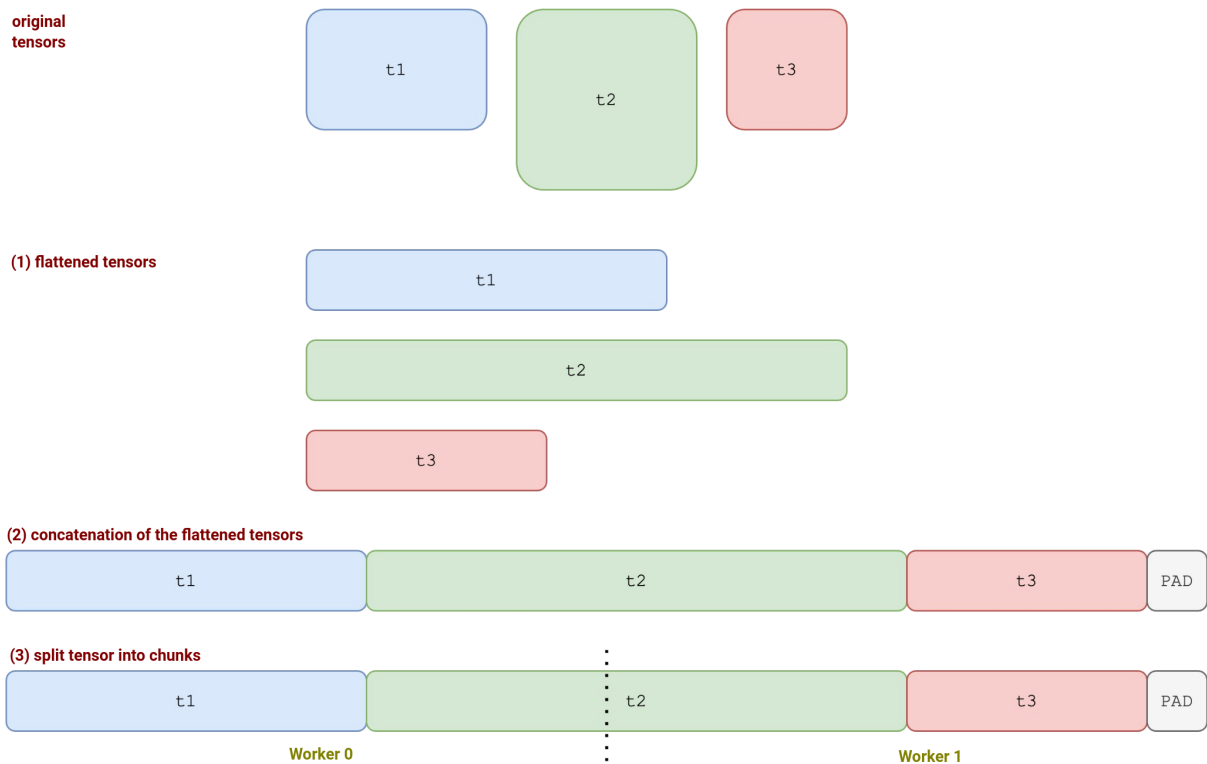


Figure 3.38: Using flattened tensor weights in FSDP1.

FSDP2 uses a per-parameter tensor sharding strategy. To understand per-parameter sharding, let us consider the same weight tensors: $t1$ of size (2, 3), $t2$ of size (3, 3), and $t3$ of size (2, 2). We can slice each parameter in half over dimension 0 (row dimension in Figure 3.39), and pad the slices of a different size to have the same size (which occurs in tensor $t2$). The resulting shards are: (i) chunk 1 is composed of the first row of $t1$, the first two rows of $t2$ and the first row of $t3$, and (ii) chunk 2 is composed of the second row of $t1$, the third row of $t2$ plus a padding row, and the second row of $t3$. In this way, every weight tensor has a representation on every GPU. In FSDP2, the tensor for each shard is a distributed tensor, `D-tensor`, which allows for sharding individual tensors, preserving their original data types and metadata. But this strategy also has a disadvantage of requiring extra memory copies during the AllGather operations, as illustrated in the bottom part of Figure 3.39.

Comparing with FSDP1, FSDP2 has following advantages: (i) by representing sharded weights as `DTensor` sliced across dimension i , allows for easy manipulation of individual weights, communication-free sharded state dictionaries, and a simpler device initialization flow, (ii) improving the memory management to achieve lower and deterministic GPU memory usage, by avoiding `recordStream`, and doing so without any CPU synchronization, (iii) differing a tensor subclass extension point to customize the AllGather, for example for `float8` AllGather over `float8` fully connected layers, and NF4 in QLoRA, and (iv) mixing frozen and

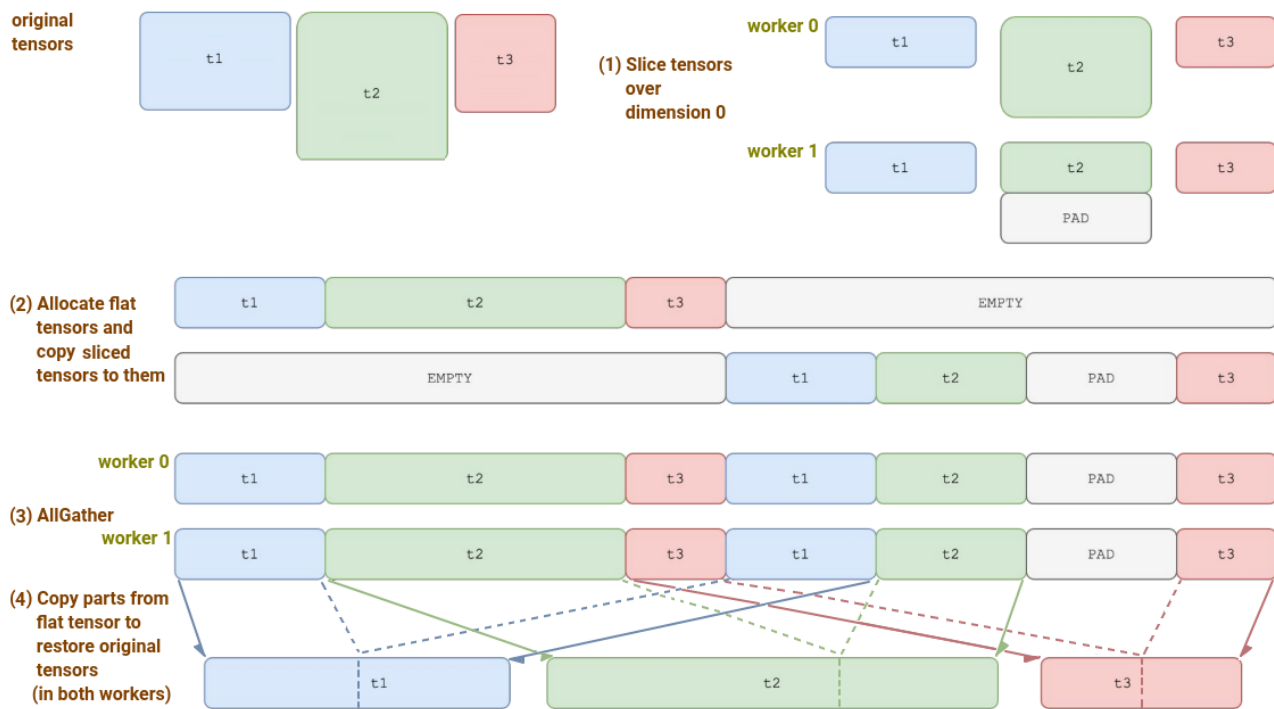


Figure 3.39: Using per-parameter tensors in FSDP2.

unfrozen weights in the same communication group without using extra memory. However, FSDP2 requires extra copies during the AllGather operations, when compared to FSDP1.

3.8.1 Practice with FSDP Training in PyTorch

Before experimenting with PyTorch FSDP, let us summarize its main features.

1. Model Wrapping

Models need to be wrapped using a FSDP policy to make sure FSDP uses the memory. For example, let us take a model with 100 layers. If the model is not wrapped, then the entire 100 layers will be collected by a single **FSDP unit** and when gathering the weight shards, it will be necessary to gather all the weights for the 100 layers and therefore we do not save any GPU memory. However, if we wrap the model, as soon as a specific condition is met, where the condition can be assigning 20 layers to each rank, the current FSDP unit is finalized and a new FSDP unit is started. It is important to know which layers to wrapped. In large language models (LLMs), often the layers that need to be wrapped are attention and feed-forward layers.

2. Mixed Precision

FSDP gives us the flexibility to choose the gradient and buffer precisions. It is recommended to use BF16 for all since FP16 has a rescaling cost associated, and the rescaler needs to find out how much to rescale because if rescaling is not accurately done, it leads to poor performance. PyTorch supports mixed precision, for example, gradients can use BF16 while buffers can use FP16. Batch normalization is automatically kept in FP32 for better precision. Local gradients during backward pass are also always converted to FP32. Models are always saved in FP32 format for maximum portability, even if trained in BF16.

3. Backward Prefetch

The backward prefetch strategy is responsible for controlling when the next FSDP unit's weights will be requested, or equivalently when the next AllGather operation will be conducted. By setting it to **BACKWARD_PRE**, requesting the next FSDP's unit weights can begin and arrive sooner, that is, before the computation of the current unit starts. This overlaps the AllGather communication and gradient computation which increases the training speed in exchange for slightly higher memory consumption. This will reduce communication overhead.

4. CPU Offloading

CPU offloading comes into play in case the model is very large and would not fit into the GPU memory, even with FSDP. Currently, PyTorch supports weight and gradient offloading to CPU. It takes weights and gradients to the CPU as soon as the GPU fails to fit them. Optimizers require that weights and gradients to be on the same device. So, currently, gradient offloading is done implicitly. Using this feature may slow down the training considerably, due to the frequent copying of tensors from host to GPU, but it could help improve memory efficiency and train larger-scale models.

5. Activation Checkpointing

Activation checkpointing is a simple and effective mechanism of selectively swapping between memory space occupied by layer's activations and recomputing the activations. A common practice in LLMs is to checkpoint each transformer block and reduce the amount of recomputation, while increasing the necessary memory. This is quite effective for 13B sized models, which increases the model throughput by 10%. For 7B models, it is not necessary activation checkpointing.

6. Sharding Strategy

The sharding strategy defines how we want to shard the model. `FULL_SHARD` is the default choice, where all model weights, optimizer state, and gradients are sharded. `SHARD_GRAD_OP` is also supported where model weights are not sharded, which leads to less communication overhead. `NO_SHARD` is also supported and it is an alternative similar to distributed data parallel, where each GPU will have a complete replica of the model in it. In the `HYBRID_SHARD` strategy, it is used the `FULL_SHARD` mode within nodes and `NO_SHARD` mode between nodes.

7. Checkpointing

Checkpointing allows for continue training from the last checkpoint as well as use the model once the training is completed. FSDP supports saving and loading checkpoints on both `FULL_STATE_DICT` and `SHARDED_STATE_DICT` settings, using `FileSystemReader` and `FileSystemWriter` of the PyTorch distributed shard checkpoints. In `FULL_STATE_DICT` setting, all the weights, optimizer state, and gradients are taken to rank 0 and saved using the CPU memory. This means there is an upper limit of model sizes that can be saved based on CPU memory. To resume training, the objects to be sharded need to be scattered to the other ranks as well. In `SHARDED_STATE_DICT` alternative, each rank saves its own weights, optimizer state, and gradients.

Fully Sharded Data Parallel 2 (FSDP2)

In `DistributedDataParallel` (DDP) training, each rank owns a model replica and processes a batch of data, finally it uses `AllReduce` to synchronize the gradients across ranks. Comparing with DDP, FSDP reduces GPU memory footprint by sharding model weights, gradients, and optimizer state. It makes it feasible to train models that cannot fit on a single GPU. The main steps of PyTorch FSDP2 are as follows.

- Except during forward and backward passes, weights are fully sharded.
- Before forward and backward passes, sharded weights are `AllGathered` into unsharded weights.
- During backward pass, local unsharded gradients are `ReduceScattered` into sharded gradients.
- Optimizer updates sharded weights with sharded gradients, resulting in sharded optimizer state.

FSDP can be considered a decomposition of the DDP's `AllReduce` operation into a `ReduceScatter` and an `AllGather` operations.

1. Model initialization

During model initialization, `fully_shard` converts the model weights from plain `torch.Tensor` to a `DTensor` in-place. The parameters are moved to the training device. Different from DDP, we have to apply `fully_shard` on submodules as well as the main model (Listing 3.25). In the transformer example that we adopt here, we apply `fully_shard` to each layer first, then to the main model. During forward computation

of i -th layer, the other layers are sharded to reduce memory usage. In the `fully_shard(model)` unit, FSDP2 excludes weights from `model.layers` and adds the remaining weights into a common unit for improving the performance of AllGather and ReduceScatter operations.

```
1 from torch.distributed.fsdp import fully_shard, FSDPModule
2 model = Transformer()
3 for layer in model.layers:
4     fully_shard(layer)
5 fully_shard(model)
6
7 print(model)
```

Listing 3.25: Transformer model initialization and wrap the model with `fully_shard`.

When we print the transformer model in Listing 3.25, we obtain the FSDP2 architecture included in Figure 3.40. We see that the ten transformer blocks are allocated to `FSDPTransformer` units and the remaining layers are allocated to a `FSDPTransformer` unit. `FSDPTransformer` combines the `Transformer` and the `FSDPModule` classes. The same type of junction occurs in `FSDPTransformerBlock`.

```
FSDPTransformer(
  (tok_embeddings): Embedding(1024, 16)
  (pos_embeddings): Embedding(64, 16)
  (dropout): Dropout(p=0, inplace=False)
  (layers): ModuleList(
    (0-9): 10 x FSDPTransformerBlock(
      (attention_norm): LayerNorm((16,), eps=1e-05, elementwise_affine=True)
      (attention): Attention(
        (resid_dropout): Dropout(p=0, inplace=False)
        (wq): Linear(in_features=16, out_features=16, bias=False)
        (wk): Linear(in_features=16, out_features=16, bias=False)
        (wv): Linear(in_features=16, out_features=16, bias=False)
        (wo): Linear(in_features=16, out_features=16, bias=False)
      )
      (ffn_norm): LayerNorm((16,), eps=1e-05, elementwise_affine=True)
      (feed_forward): FeedForward(
        (w1): Linear(in_features=16, out_features=64, bias=True)
        (gelu): GELU(approximate="none")
        (w2): Linear(in_features=64, out_features=16, bias=True)
        (resid_dropout): Dropout(p=0, inplace=False)
      )
    )
  )
  (norm): LayerNorm((16,), eps=1e-05, elementwise_affine=True)
  (output): Linear(in_features=16, out_features=1024, bias=False)
)
```

Figure 3.40: Transformer architecture after being wrapped by FSDP2 `fully_shard`.

`fully_shard` shards weights across ranks, and converts `model.parameters()` from a plain `torch.Tensor` to `DTensor` to represent sharded weights. By default, FSDP2 slices tensors over dimension 0, and as so the `DTensor` is sharded over the ranks given by `Shard(dim=0)`. For example, if we have N ranks and a weight tensor has N rows before sharding, after sharding each rank will have one row of the weight. We can inspect a sharded weight `param` using `param.to_local()`, as illustrated in Listing 3.26. The optimizer is instantiated after wrapping the model with `fully_shard`. Both model and optimizer state dictionaries are represented as `DTensors`. `DTensor` facilitates optimization, gradient clipping, and model checkpointing

```

1 from torch.distributed.tensor import DTensor
2 for param in model.parameters():
3     assert isinstance(param, DTensor)
4     assert param.placements == (Shard(0),)
5     # inspect sharded weights with param.to_local()
6     print(f'{param.to_local()}')
7 optim = torch.optim.Adam(model.parameters(), lr=1e-2)

```

Listing 3.26: Inspect how the sharded weights are distributed by ranks.

2. Forward and backward with prefetching

`fully_shard` registers forward and backward hooks to run AllGather over the weights before the computation, and reshards the weights after computation. To overlap AllGathers with computation, FSDP2 offers (i) implicit prefetching, which works automatically during the training loop and (ii) explicit prefetching to control AllGather scheduling manually. In **implicit prefetching**, the CPU thread issues AllGather i before the computation over layer i starts. When the AllGathers run faster, they are queued in the GPU stream while layer i computation happens in the default stream (Figure 3.41). In CPU-bound problems, such as a Transformer trained with a large batch size, AllGather $i+1$ can overlap with computation for layer i . Implicit prefetching works similarly in the backward, except that AllGathers are issued in the reverse order of post-forward.

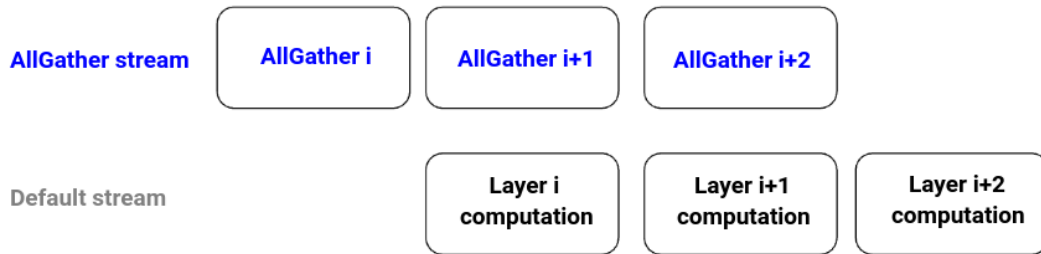


Figure 3.41: Implicit prefetching of AllGather operations in FSDP2.

```

1 num_to_forward_prefetch = 2
2 for i, layer in enumerate(model.layers):
3     if i >= len(model.layers) - num_to_forward_prefetch:
4         break
5     layers_to_prefetch = [
6         model.layers[i + j] for j in range(1, num_to_forward_prefetch + 1)
7     ]
8     layer.set_modules_to_forward_prefetch(layers_to_prefetch)
9
10 num_to_backward_prefetch = 2
11 for i, layer in enumerate(model.layers):
12     if i < num_to_backward_prefetch:
13         continue
14     layers_to_prefetch = [
15         model.layers[i - j] for j in range(1, num_to_backward_prefetch + 1)
16     ]
17     layer.set_modules_to_backward_prefetch(layers_to_prefetch)
18
19 # Training loop
20 for _ in range(num_iterations):
21     # Triggers first AllGather earlier, allowing to overlap
22     # AllGather with computation before the forward pass 'model(x)'
23     model.unshard()
24     x = torch.randint(0, vocab_size, (batch_size, seq_len), device=device)
25     loss = model(x).sum()
26     loss.backward()
27     optim.step()
28     optim.zero_grad()

```

Listing 3.27: Explicit prefetching, to schedule AllGather operations over two layers in advance, during the forward and backward passes.

In **explicit prefetching**, we can specify the forward ordering using `set_modules_to_forward_prefetch` and backward ordering with `set_modules_to_backward_prefetch`. According to Listing 3.27, the CPU thread issues AllGather $i + 1$ and $i + 2$ while layer i computation occurs.

Explicit prefetching works well (i) in CPU-bound computation, where using implicit prefetching turns the CPU thread too slow to issue AllGather for layer $i+1$ when the computation over layer i occurs. We have to explicitly issue AllGather $i+1$ before running the forward on layer i , (ii) we want to prefetch two or more layers, since implicit prefetching only AllGathers the next layer at a time to keep memory usage at a minimum. With explicit prefetching, we can AllGather multiple layers at a time for better performance with increased memory, (iii) when we want to issue the first AllGather earlier, as implicit prefetching happens at the time of calling `model(x)`, the first AllGather gets exposed. Thus, we can call `model.unshard()` explicitly to issue the first AllGather earlier.

3. Enabling mixed precision

FSDP2 offers a flexible mixed precision policy to speed up training. The common use case is (i) cast float32 weights to bfloat16 during the forward and backward computations, (ii) upcast gradients to float32 to preserve accuracy during ReduceScatter operations (Listing 3.28).

Comparing with automatic mixed precision provided by `torch.amp`, FSDP2 mixed precision has the following advantages: (i) increased performance and flexible parameter casting, since all weights in a `FSDPModule` are cast together at the module boundary and we can set a different mixed precision policy for each layer. For example, the first two layers can have the weights in float32 while the remaining layers can have the weights in bfloat16, (ii) reducing gradients to float32 can be critical for numeric stability, since the gradients might vary a lot from rank to rank.

```
1 model = Transformer(model_args)
2
3 # Unsharded weights are represented in bfloat16
4 # Sharded weights are represented in float32
5 fsdp_kwargs = {
6     "mp_policy": MixedPrecisionPolicy(
7         param_dtype = torch.bfloat16,
8         reduce_dtype = torch.float32,
9     )
10 }
11 for layer in model.layers:
12     fully_shard(layer, **fsdp_kwargs)
13 fully_shard(model, **fsdp_kwargs)
14
15 # Confirm that sharded weights are float32
16 for param in model.parameters():
17     assert param.dtype == torch.float32
18 # Confirm that unsharded weights are bfloat16
19 model.unshard()
20 for param in model.parameters(recurse=False):
21     assert param.dtype == torch.bfloat16
22 model.reshard()
23
24 # The optimizer state is represented in float32
25 optim = torch.optim.Adam(model.parameters(), lr=1e-2)
```

Listing 3.28: Casting the unshared and sharded weights to different precisions.

4. Gradient clipping and optimizer with DTensors

The optimizer is initialized after wrapping the model with `fully_shard`, and stores the model weights as DTensors. To clip gradients, we can employ `torch.nn.utils.clip_grad_norm_` since it works with weights stored in DTensors (Listing 3.29). Operations over a DTensor are correctly dispatched to operate over the partial tensors distributed across ranks and preserve the single device semantic.


```

1 # The optimizer uses DTensors to store the model's weights
2 optim = torch.optim.Adam(model.parameters(), lr=1e-2)
3 # Training loop
4 for _ in range(num_iterations):
5     x = torch.randint(0, vocab_size, (batch_size, seq_len), device=device)
6     loss = model(x).sum()
7     loss.backward()
8     torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=max_norm)
9     optim.step()
10    optim.zero_grad()

```

Listing 3.29: Training loop where the optimizer works with weights stored as DTensors.

5. State dictionaries with DTensors

Here, we show how to convert a full state dictionary into a DTensor state dictionary when the model is loaded from file, and how to convert a DTensor state dictionary to full state dictionary when the model is saved to file.

5a. Load state dictionaries

We allocate the model to a meta device and call `fully_shard` to convert the model weights from plain `torch.Tensor` to DTensors. After reading the full state dictionary from file with `torch.load`, we call `distribute_tensor` to convert plain `torch.Tensors` to DTensors, using the same rank placement and device mesh that was read from file with `model.state_dict()`. Finally, we call `model.load_state_dict` to send the DTensor state dictionaries to the model (Listing 3.30).

```

1 from torch.distributed.tensor import distribute_tensor
2
3 # Use 'mmap=True' to reduce CPU memory usage
4 full_sd = torch.load(
5     "checkpoints/model_state_dict.pth",
6     mmap = True,
7     weights_only = True,
8     map_location = 'cpu',
9 )
10 meta_sharded_sd = model.state_dict()
11 sharded_sd = {}
12
13 for param_name, full_tensor in full_sd.items():
14     sharded_meta_param = meta_sharded_sd.get(param_name)
15     sharded_tensor = distribute_tensor(
16         full_tensor,
17         sharded_meta_param.device_mesh,
18         sharded_meta_param.placements,
19     )
20     sharded_sd[param_name] = nn.Parameter(sharded_tensor)
21
22 # Use 'assign=True' since we can not call 'copy_' on a meta tensor
23 model.load_state_dict(sharded_sd, assign=True)

```

Listing 3.30: Load state dictionaries from file, convert them to DTensors, and send the DTensors to the model.

5b. Save state dictionaries

`model.state_dict()` returns a DTensor state dictionary. We can convert a DTensor into a plain `torch.Tensor` by calling `full_tensor()`. Internally, it issues an AllGather across ranks to get the DTensor shards and create a plain `torch.Tensor` with the unsharded weights. Rank 0 uses `full_param.cpu()` to offload the tensor to CPU, one weight at a time, to avoid increasing the GPU memory usage with unsharded weights (Listing 3.31). The optimizer state dictionaries work similarly to weight dictionaries.


```

1 sharded_sd      = model.state_dict()
2 cpu_state_dict = {}
3
4 for param_name, sharded_param in sharded_sd.items():
5     full_param = sharded_param.full_tensor()
6     if torch.distributed.get_rank() == 0:
7         cpu_state_dict[param_name] = full_param.cpu()
8     else:
9         del full_param
10
11 torch.save(cpu_state_dict, "checkpoints/model_state_dict.pth")

```

Listing 3.31: Convert the model's weights, sharded as DTensors, to plain tensors, offload the unsharded weights to CPU, and save the unsharded weight to file.

6. State dictionaries with DCP API

Distributed Checkpoint (DCP) supports loading and saving models from multiple ranks in parallel. It handles load-time resharding, which enables saving in one cluster topology and loading into another. DCP is different than `torch.save` and `torch.load` in a few ways: (i) it produces multiple files per checkpoint, at least one per rank, (ii) it operates in place, meaning that the model should allocate its data first and DCP uses that storage.

6a. Load state dictionaries

We can load a full state dictionary into a FSDP2 model with `set_model_state_dict`. With `broadcast_from_rank0=True`, we can load the full state dictionary only on rank 0 to avoid increasing the CPU memory usage. DCP shards the tensors and broadcasts them to other ranks (Listing 3.32).

```

1 from torch.distributed.checkpoint.state_dict import set_model_state_dict
2 set_model_state_dict(
3     model = model,
4     model_state_dict = full_sd,
5     options = StateDictOptions(
6         full_state_dict = True,
7         broadcast_from_rank0 = True,
8     ),
9 )

```

Listing 3.32: Load a model state dictionary from file using the DCP API.

6b. Save state dictionaries

To save state dictionaries to file, we can use `get_model_state_dict` with `full_state_dict=True` and `cpu_offload=True`. This function runs AllGathers to get the sharded DTensors and offloads them to CPU (Listing 3.33). DCP works similarly to the DTensor API.

```

1 from torch.distributed.checkpoint.state_dict import get_model_state_dict
2 model_state_dict = get_model_state_dict(
3     model = model,
4     options = StateDictOptions(
5         full_state_dict = True,
6         cpu_offload = True,
7     )
8 )
9 torch.save(model_state_dict, "model_state_dict.pth")

```

Listing 3.33: Save the model state dictionary to file using the DCP API.

Listings 3.34 and `lst:torch-fsdp2-10b` contain the main function for training the Transformer model with FSDP2. It essentially is built with the several block of code presented above. The complete code can be consulted in Appendix A.

```

1 def main(args):
2     _min_gpu_count = 2
3     if not verify_min_gpu_count(min_gpus=_min_gpu_count):
4         print(f"Unable to locate sufficient {_min_gpu_count} GPUs to run this example.")
5         exit()
6
7     rank = int(os.environ["LOCAL_RANK"])
8     if torch.accelerator.is_available():
9         device_type = torch.accelerator.current_accelerator()
10        device = torch.device(f"{device_type}:{rank}")
11        torch.accelerator.device_index(rank)
12        print(f"Running rank {rank} on device {device}")
13    else:
14        device = torch.device("cpu")
15        print(f"Running on device {device}")
16
17    # Initialize the default process group and communication backend.
18    backend = torch.distributed.get_default_backend_for_device(device)
19    torch.distributed.init_process_group(backend=backend, device_id=device)
20
21    # Configure a Transformer model with 10 layers, 4 attention heads,
22    # a maximum sequence length of 64, no dropout, and a vocabulary size of 1024.
23    torch.manual_seed(0)
24    vocab_size = 1024
25    batch_size = 32
26    seq_len = 64
27    model_args = ModelArgs(
28        n_layers = 10,
29        n_heads = 4,
30        vocab_size = vocab_size,
31        max_seq_len = seq_len,
32        dropout_p = 0,
33    )
34
35    # Instantiate the model on the meta device to avoid unnecessary memory usage, since
36    # FSDP2 will take care of moving parameters to the correct device and shard them.
37    with torch.device("meta"):
38        model = Transformer(model_args)
39
40    fsdp_kwargs = {}
41    if args.mixed_precision:
42        fsdp_kwargs["mp_policy"] = MixedPrecisionPolicy(
43            param_dtype = torch.bfloat16,
44            reduce_dtype = torch.float32,
45        )
46
47    # Wrap each layer with FSDP2 and then the entire model.
48    # This allows for more fine-grained control over sharding and prefetching.
49    for layer in model.layers:
50        fully_shard(layer, **fsdp_kwargs)
51    fully_shard(model, **fsdp_kwargs)
52
53    # Print the model to verify that it is correctly wrapped with FSDP2.
54    inspect_model(model)
55
56    # Optionally set up explicit prefetching for the forward and backward passes.
57    # With prefetching, FSDP modules should explicitly prefetch AllGathers in forward.
58    # The prefetching runs after this module's AllGather copy-out.
59
60    if args.explicit_prefetching:
61        set_modules_to_forward_prefetch(model, num_to_forward_prefetch=2)
62        set_modules_to_backward_prefetch(model, num_to_backward_prefetch=2)

```

Listing 3.34: The main function to train the transformer model with FSDP2 (part 1).

```

1  # If a checkpoint exists, load the model and optimizer state from the checkpoint.
2  checkpointer = Checkpointer("checkpoints", dcp_api=args.dcp_api)
3  if checkpointer.last_training_time is None:
4      model.to_empty(device=device)
5      model.reset_parameters()
6  else:
7      checkpointer.load_model(model)
8
9  # If mixed precision is enabled, inspect the model to verify that
10 # weights are in the correct data type.
11 if args.mixed_precision:
12     inspect_mixed_precision(model)
13
14 # Construct the optimizer based on the model parameters. If a checkpoint was loaded,
15 # load the optimizer state from the checkpoint as well.
16 optim = torch.optim.Adam(model.parameters(), lr=1e-2)
17 if checkpointer.last_training_time is not None:
18     checkpointer.load_optim(model, optim)
19
20 # Run a simple training loop for 10k iterations, where in each iteration we:
21 # 1. Optionally unshard the model if explicit prefetching is enabled
22 # 2. Generate random input data
23 # 3. Compute the loss and perform the backward pass
24 # 4. Clip the gradients and update the model weights using the optimizer
25 # 5. Reset the optimizer gradients
26
27 for _ in range(10000):
28     if args.explicit_prefetching:
29         model.unshard()
30     x = torch.randint(0, vocab_size, (batch_size, seq_len), device=device)
31     loss = model(x).sum()
32     loss.backward()
33     torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
34     optim.step()
35     optim.zero_grad()
36
37 # Save the model and optimizer state to a checkpoint,
38 # and then clean up the process group.
39 checkpointer.save(model, optim)
40 torch.distributed.destroy_process_group()

```

Listing 3.35: The main function to train the transformer model with FSDP2 (part 2).

7. Results

Table 3.1 presents the results obtained with training the simple transformer, for 10k iterations on 2, 3 or 4 GPUs NVIDIA Titan Xp. The commands used to launch training were:

```

torchrn -nproc_per_node 2|3|4 transformer.py
torchrn -nproc_per_node 2|3|4 transformer.py -explicit-prefetch
torchrn -nproc_per_node 2|3|4 transformer.py -explicit-prefetch -mixed-precision

```

The main goal was to validate the FSDP2 recipe for training a model on multiple GPUs and it was achieved. However, the results show that neither increasing the number of GPUs, nor using AllGather prefetch, nor training with mixed precision accelerated training. This result was not expected and two possible explanations are the model being small and the communication among GPUs being slow.

GPUs	Memory/GPU (MB)	Execution time
2	335	697 s (11m:37s)
3	333	694 s (11m:34s)
4	333	696 s (11m:36s)
with explicit prefetch		
2	335	695 s (11m:35s)
3	333	699 s (11m:39s)
4	333	703 s (11m:43s)
with explicit prefetch and mixed precision		
2	346	776 s (12m:56s)
3	346	794 s (13m:14s)
4	346	797 s (13m:17s)

Table 3.1: Results for training the simple transformer, for 10k iterations, on 2, 3 or 4 GPUs NVIDIA Titan Xp.

3.9 Conclusions

In this chapter, we explored distributed data parallelism, where multiple replicas of a model process different portions of the data in parallel. By leveraging collective communication operations such as AllReduce, this approach enables synchronized gradient updates and preserves the semantic equivalence of sequential training. We also discussed practical implementations, particularly PyTorch’s Distributed Data Parallel, highlighting optimizations such as gradient bucketing, communication-computation overlap, and efficient process coordination.

We also introduced the parameter server paradigm as an alternative framework for distributed learning. Unlike the decentralized nature of AllReduce-based methods, the parameter server adopts a centralized or sharded architecture in which model weights are maintained by dedicated servers and updated through communication with worker nodes. This design facilitates asynchronous training, improves robustness to node failures, and supports large-scale deployments, although it introduces challenges related to stale gradients and communication overhead.

The chapter further compared synchronous and asynchronous training strategies, emphasizing their trade-offs. While synchronous methods offer stability and consistent convergence behavior, asynchronous approaches, commonly used in parameter server systems, can improve hardware utilization and fault tolerance at the cost of increased optimization complexity.

We also analyzed the critical role of communication in distributed systems, examining algorithms such as ring, tree, and two-tree AllReduce. These methods determine the efficiency of gradient aggregation and have a significant impact on scalability, particularly in large clusters where latency and bandwidth constraints become dominant.

To address memory limitations in large-scale training, we studied advanced techniques such as the Zero Redundancy Optimizer and Fully Sharded Data Parallel. These approaches reduce memory duplication by partitioning optimizer states, gradients, and model weights across devices, enabling the training of models that would otherwise exceed hardware limits. Their effectiveness relies on carefully balancing memory savings against increased communication requirements.

The chapter highlights that successful distributed training is not determined by a single technique, but rather by the careful integration of multiple design choices, including parallelization strategy, communication patterns, synchronization methods, and memory management. Understanding the interplay between these factors allows practitioners to build scalable, efficient, and robust training systems that fully exploit modern distributed computing infrastructures.

Chapter 4

Model Parallelism

Contents

4.1	Pipeline parallelism	85
4.2	Tensor parallelism	104
4.3	Combining multiple types of parallelism	117
4.4	Conclusions	121

This chapter presents the main techniques used to train large-scale neural networks across multiple devices through different forms of parallelism. It begins by introducing model parallelism, where a model is partitioned across GPUs, highlighting its advantages for handling memory constraints as well as its limitations, such as low device utilization and communication overhead. To address these inefficiencies, the chapter explores pipeline parallelism, describing how minibatches can be divided into microbatches to better utilize hardware resources. Key approaches such as GPipe and PipeDream are discussed, including their scheduling strategies, memory trade-offs, and techniques to reduce pipeline idle time (bubbles). Practical aspects, including GPU communication, gradient accumulation, and framework implementations, such as PyTorch, DeepSpeed, and Megatron, are also covered. The chapter then introduces tensor parallelism, which distributes computations within individual layers across devices, explaining how matrix operations can be partitioned efficiently to minimize synchronization costs. Finally, the chapter examines how these different parallelism strategies, model, pipeline, tensor, and data parallelism, can be combined into hybrid approaches to scale training to extremely large models. It also discusses practical considerations, communication mechanisms, and advanced systems that automate or optimize parallelization strategies.

In naive model parallelism, we split a neural network across multiple computing devices, assigning layers or sequences of layers to each one. This approach is useful when the model is too large to fit in the memory of a single device, or when certain parts of the model require more computation than others. During the forward pass, activations are transferred between devices, and the backward pass similarly gradients are transferred across devices (Figure 4.1). This approach allows for training models that exceed a single device’s memory capacity but introduces communication overhead when moving from one layer to the next one.

Consider an example of a four-layer sequential model, where $\text{output} = L_4(L_3(L_2(L_1(\text{input}))))$, and that we split the computation among two GPUs as follows: GPU0 computes $\text{out}_2 = L_2(L_1(D_{\text{in}}))$ and GPU1 computes $D_{\text{out}} = L_4(L_3(\text{out}_2))$. A complete forward pass requires computing out_2 on GPU0, transferring the out_2 tensor to GPU1, and computing the output of the model in GPU1. Then, it starts the backward pass. For the backward pass, we send the gradients with respect to out_2 from GPU1 to GPU0. GPU0 then completes the backward pass based on the gradients it received. Because the sending operations do not modify the information, naive model parallelism generates, unlike data parallelism, the same output as sequential training. This makes debugging easier.

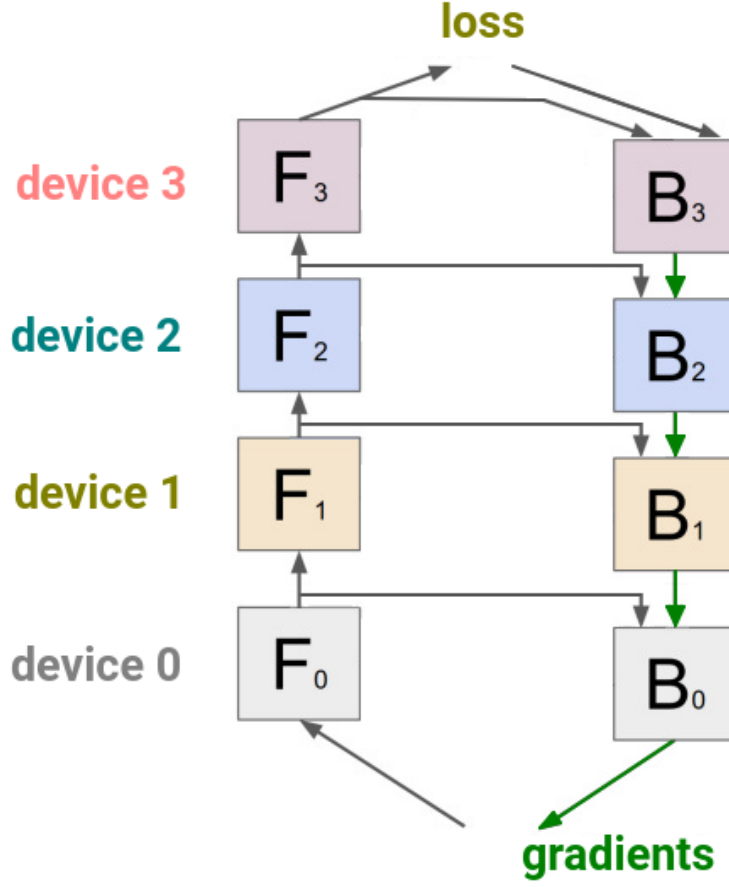


Figure 4.1: An example of a neural network with layers partitioned across four devices. F_k is the forward computation function of the k -th block. B_k is the back-propagation function, which depends on both B_{k+1} from the upper layer and F_k [88].

The pebble graph included in Figure 4.2 illustrates how naive model parallelism works. GPU0 performs its forward pass and caches the activations (green). Then, we use MPI to send the output of L2 to GPU1. GPU1 finishes the forward pass, calculates the loss using the target values, and starts the backward pass. Once GPU1 has finished, the gradient with respect to out2 is sent to GPU0, which then completes the backward pass. Here, we only run node-to-node communication, `MPI.Send` and `MPI.Recv`, and it is not necessary collective communications as in data parallelism.

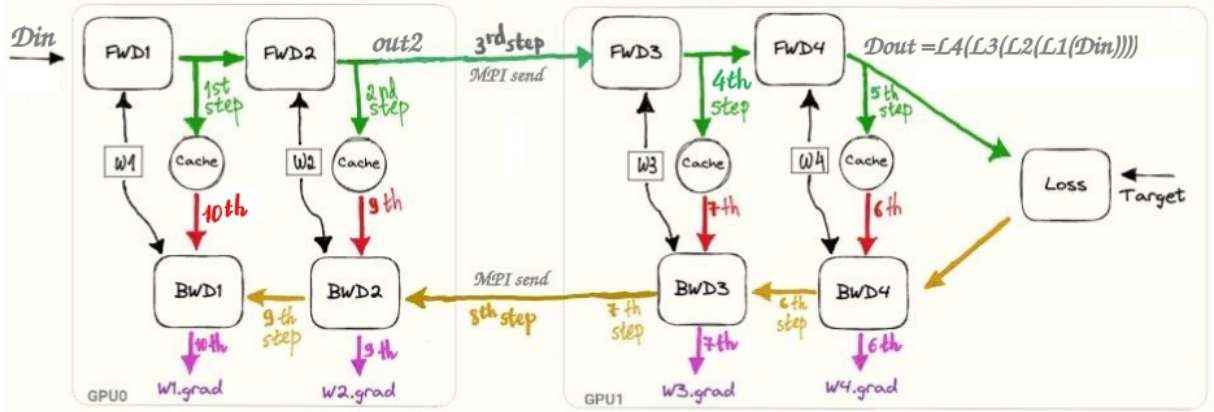


Figure 4.2: Example of a neural network with four layers partitioned across two devices [14].

This example reveals some inefficiencies of naive model parallelism: (i) there is a **low GPU utilization**, since at any given time, only one GPU is occupied, while the other GPUs are idle. If we add more GPUs, each one would be busy only $1/\text{\#GPUs}$ of the time, neglecting the communication overhead. The low utilization suggests that there may be a way to speed up training by assigning useful work to idle GPUs, (ii) there is **no overlap between communication and computation**, since when we are sending over the network the

intermediate outputs, in the forward pass, and the intermediate gradients, in the backward pass, the GPUs are idle. We already discussed computation and communication overlap in data parallelism, (iii) it occurs a **high and unbalanced memory usage**. In our example, GPU0 keeps the activations in cache almost until the end of the minibatch. If the batch size is large, this can create memory problems. Later we will see how to combine data and pipeline parallelism to address this problem.

Let us now clarify the low device utilization issue. Due to the sequential nature of forward and backward propagation, only one device does computation at a time, while the others remain idle, resulting in pipeline bubbles where most time is spent on communication and waiting. This effect is illustrated in Figure 4.3. For a model split across N devices, each device is active only about $2/N$ of the time, excluding communication overhead.

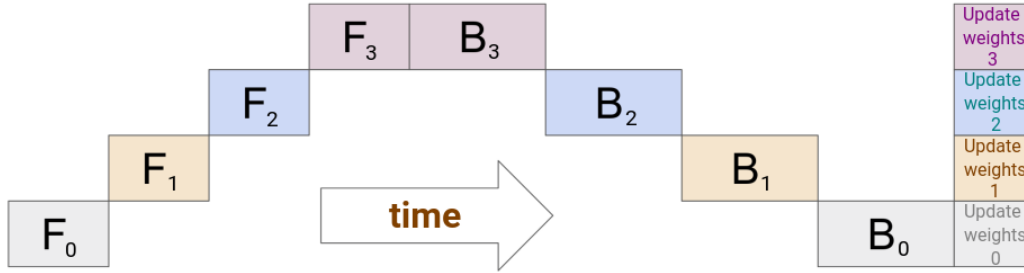


Figure 4.3: Low device utilization in model parallelism due to the sequential dependency of the network [88].

The main problem of naive model parallelism, is that all but one GPU are idle at any given moment. So if we use four GPUs, this is almost equivalent to using a single GPU with four times more memory, and ignoring the rest of the hardware. Furthermore, there is the overhead of copying the data between devices. So, four 6 GB GPUs will be able to train the same model as a single 24 GB GPU using naive model parallelism, except the latter will complete the training faster, since it does not have the data copying overhead. However, if we have 40 GB GPUs and need to train a 45 GB model, then we can do it with four 40 GB cards, but just barely, because the gradients and the optimizer state take up a lot of space in this large model.

4.1 Pipeline Parallelism

Pipeline parallelism (PP) combines model parallelism with data parallelism to reduce the inefficient utilization of GPUs associated with naive parallelism. The main idea is to split each minibatch into multiple microbatches, creating a pipeline, and allocate the different microbatches of the pipeline to different GPUs that process the microbatches simultaneously. Note that every microbatch needs two passes, one forward and one backward. The communication among successive GPUs during the forward pass only involves the activations, and the gradients in the backward pass. How these passes are scheduled and how the gradients are aggregated vary in different approaches. The number of GPUs that participate on the minibatch stages is known as **pipeline depth**. The main challenge is managing the **pipeline bubble**, that is, the idle time between forward and backward passes, and keeping the work balanced across stages.

4.1.1 GPipe

1. GPipe Algorithm

GPipe aims to increase the efficiency of GPU utilization by splitting each minibatch into smaller, equal-sized microbatches [88]. In this way, we can run the forward and backward passes independently for each microbatch, as long as there is no batch normalization layers. It is possible to use batch normalization and GPipe by computing the normalizing statistics over the microbatch instead the minibatch, which often works but distributed training will not be equal to sequential training anymore. If we sum up the gradients over all microbatches, we get the gradient for the minibatch. This is valid since the gradient of a sum of parcels is the sum of the gradients of every parcel. This process is denominated gradient accumulation. Since a given layer is assigned only to a single GPU, the sum of the gradients over all microbatches of a minibatch can be

performed locally on that GPU, without any communication with other GPUs. The local gradient accumulation is mathematically equivalent to sequential training. Due to the non-associativity of floating-point operations in real hardware, the outputs of distributed and sequential training will not be completely equal. In most cases, this is not a problem.

Let us consider a model partitioned across four GPUs. The general problem of partitioning an arbitrary model among GPUs, such that computation is balanced and communication is minimized, is fairly difficult and requires performance profiling. In transformers it is easy to solve since the model is implemented essentially with N transformer, all with the same functionality and dimensions. In naive pipeline parallelism, the schedule of the model layers is as depicted in Figure 4.4.

Timestep	0	1	2	3	4	5	6	7
GPU3				F3,1	B3,1			
GPU2			F2,1			B2,1		
GPU1		F1,1					B1,1	
GPU0	F0,1							B0,1

Figure 4.4: Naive pipeline parallelism applied for splitting a model into four GPUs [14].

In naive pipeline parallelism, there is only one GPU occupied at any given time. Furthermore, each time step of the pipelines could be quite long, since each GPU has to run a forward or backward pass for the whole minibatch. With GPipe, we split each minibatch into four microbatches, as illustrated in Figure 4.5.

Timestep	0	1	2	3	4	5	6	7	8	9	10	11	12	13
GPU3				F3,1	F3,2	F3,3	F3,4	B3,4	B3,3	B3,2	B3,1			
GPU2			F2,1	F2,2	F2,3	F2,4		B2,4	B2,3	B2,2	B2,1			
GPU1		F1,1	F1,2	F1,3	F1,4				B1,4	B1,3	B1,2	B1,1		
GPU0	F0,1	F0,2	F0,3	F0,4							B0,4	B0,3	B0,2	B0,1

Figure 4.5: GPipe pipeline parallelism where we split each minibatch into four microbatches [14].

In this figure, $F_{g,u}$ corresponds to the forward pass over microbatch u using the model partition allocated to the GPU g . The most relevant result of GPipe scheduling, is the time step being shorter than the correspondent time step in the naive model parallel schedule, since with GPipe a GPU only processes a quarter of the minibatch at a time. However, splitting the minibatch into smaller microbatches will add computation overhead, partly because we have to launch more kernels globally. If the model partitions are small and the microbatches are small too, there may not be enough opportunity for within-GPU parallelism to achieve a high GPC core utilization. Despite everything, GPipe and its approach that creates microbatches are a big improvement over naive pipeline parallelism since now more than one GPU is occupied at the same time.

Let us look next at the remaining inefficiencies of GPipe, and how we can address them, namely the overlap of computation and communication, the pipeline bubbles, and the memory usage.

2. Overlap Computation with Communication

Unfortunately, there is not a lot of opportunity to overlap computation and communication if the forward and backward passes take the same amount of time on each GPU. This can be seen in the scheduling depicted in Figure 4.5, since each GPU can not start processing a given microbatch u before the previous GPU has finished

processing the same microbatch u . If all stages take the same amount of time, we get distinct communication and computation durations.

An alternative to improve communication and computation overlap could be to split each microbatch in two halves, and then overlap communication of the first half with computation of the second half. Whether or not this works in practice depends on the model partitioning and the network performance. Figure 4.6 shows how computation would be overlapped with communication in this scenario. The arrows included in the figure represent the dependencies that occur on the first half, of the first microbatch, and the forward pass.

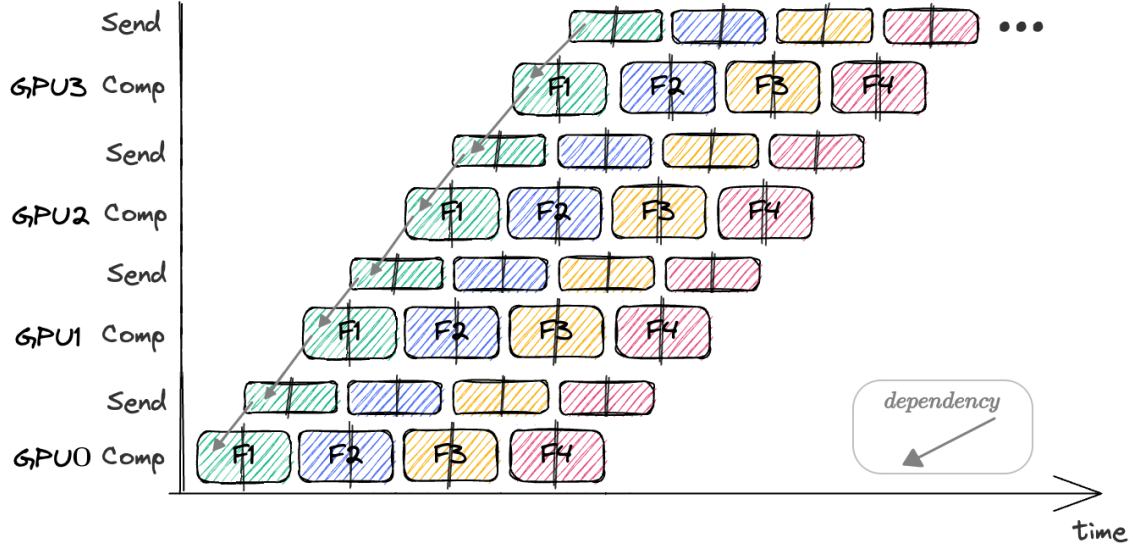


Figure 4.6: Improving the communication and computation overlap [14].

3. GPipe Bubbles

Let us now approach the main inefficiency of GPipe, the size of the pipeline bubble. **Bubbles** are spots in the pipeline where no useful work is being done. They are caused by dependencies between the operations. For example, in Figure 4.7 GPU3 can not execute F1 until GPU2 has executed F1 and sent the result.

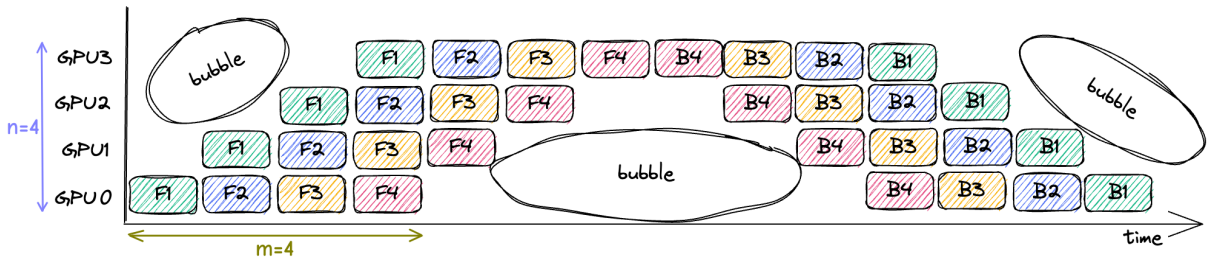


Figure 4.7: Pipeline bubbles in GPipe [14].

The fraction of time wasted on the bubble depends on the pipeline-depth n and the number of minibatches m , as indicated by Equation 4.1. The $2 \cdot m \cdot n$ term is the overall amount of useful work and results from each of the n nodes performing m forward and m backward passes. The term $2 \cdot n \cdot (m + n - 1)$ is the overall execution time of a batch. During each forward and backward passes, each node performs m parcels of work and waits $n - 1$ time steps for new work to arrive.

$$\text{bubble_fraction} = 1 - \frac{2 \cdot n \cdot m}{2 \cdot n \cdot (m + n - 1)} = 1 - \frac{m}{m + n - 1} \quad (4.1)$$

Therefore, increasing the size of the minibatch, which increases the number of minibatches m , is necessary for making the bubble fraction small. Figure 4.8 shows an example of calculating the bubble fraction for (i) a minibatch consisting of a single microbatch ($m = 1$) and (ii) minibatch decomposed in four minibatches

($m = 4$). The figure proves that a larger m means less wasted time. However, a large minibatch requires careful learning rate scaling and will increase the memory necessary for caching the activations.

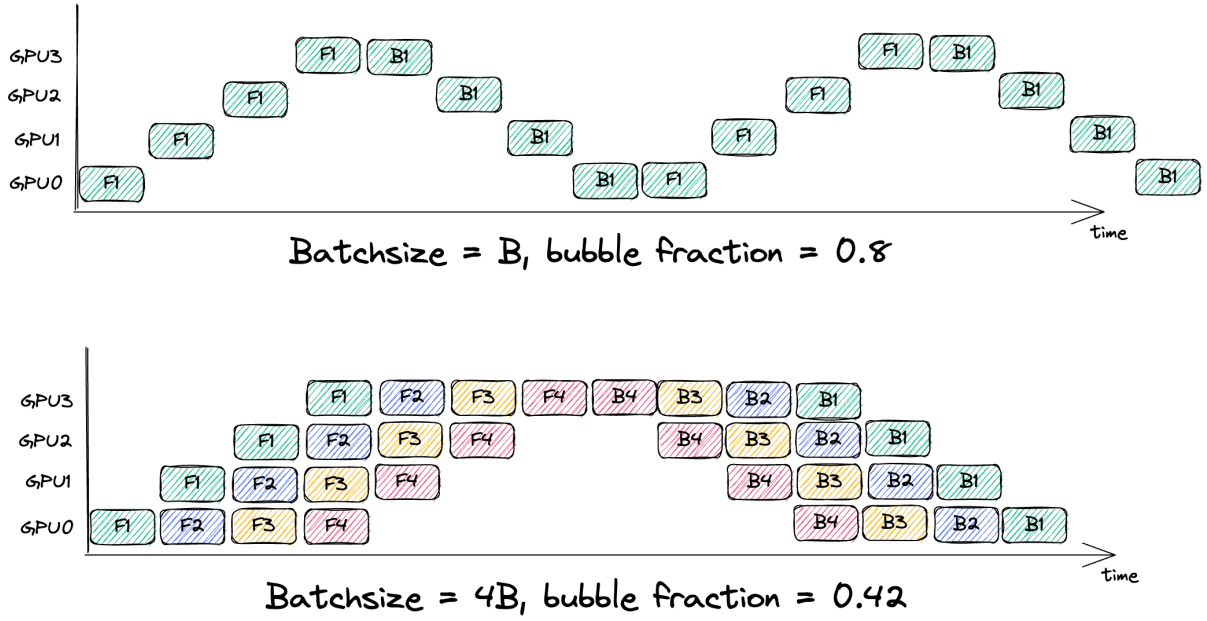


Figure 4.8: Influence of the number of microbatches over the bubble fraction [14].

4. Memory Usage

Increasing the batch size increases the memory required for caching the activations linearly. In GPipe, we need to cache the activations for each microbatch from the time it was forwarded until the corresponding backward pass occurs. Taking GPU0 as an example, Figure 4.5 shows that the activations for microbatch 1 have to remain in memory from timestep 0 until timestep 13.

In [88], the authors utilize gradient checkpointing to bring down the memory demand. In **gradient checkpointing**, instead of caching all the activations necessary to compute the gradients, we recompute the activations on the fly during the backward pass. This lowers the required memory but increases the computational cost.

Let us assume all layers have roughly the same size. Recalling that the activations are the output of a layer and simultaneously the input of the next layer, the memory required for caching the activations in each GPU is given by Equation 4.2. For each layer, we have to cache its inputs. Assuming that the layer width is a constant, a single cached input has a size proportional to $O(\text{batchsize})$.

$$O \left(\text{batchsize} \cdot \underbrace{\frac{\#layers}{\#GPUs}}_{\#layers \text{ per GPU}} \right) \quad (4.2)$$

Instead, we could adopt the gradient checkpointing technique and only cache the input of each model partition, that is, the input received by each GPU from the previous GPU or from the data loader. This lowers the maximum memory required on each GPU to the value given by Equation 4.3, where the term $O(\text{batchsize})$ is the space necessary for caching the mentioned input activations exchanged between successive GPUs.

$$O \left(\text{batchsize} + \underbrace{\frac{\#layers}{\#GPUs}}_{\#layers \text{ per GPU}} \underbrace{\frac{\text{batchsize}}{\#microbatches}}_{\text{microbatch size}} \right) \quad (4.3)$$

When the backward pass for a given microbatch runs, we have to recompute the activations that are necessary for calculating the gradients for that microbatch. This requires $O(\frac{batchsize}{\#microbatches})$ space for each of the $O(\frac{\#layers}{\#GPUs})$ layers on each GPU. This space corresponds to the second term of Equation 4.3. Figure 4.9 explains visually the memory required by GPipe when using gradient checkpointing. It shows two GPUs during the backward pass (purple). GPU2 has recomputed the activations for microbatch 3, while GPU3 has recomputed activations for microbatch 2. At the GPU boundary, the activations for the whole batch stay cached from the forward until the backward pass.

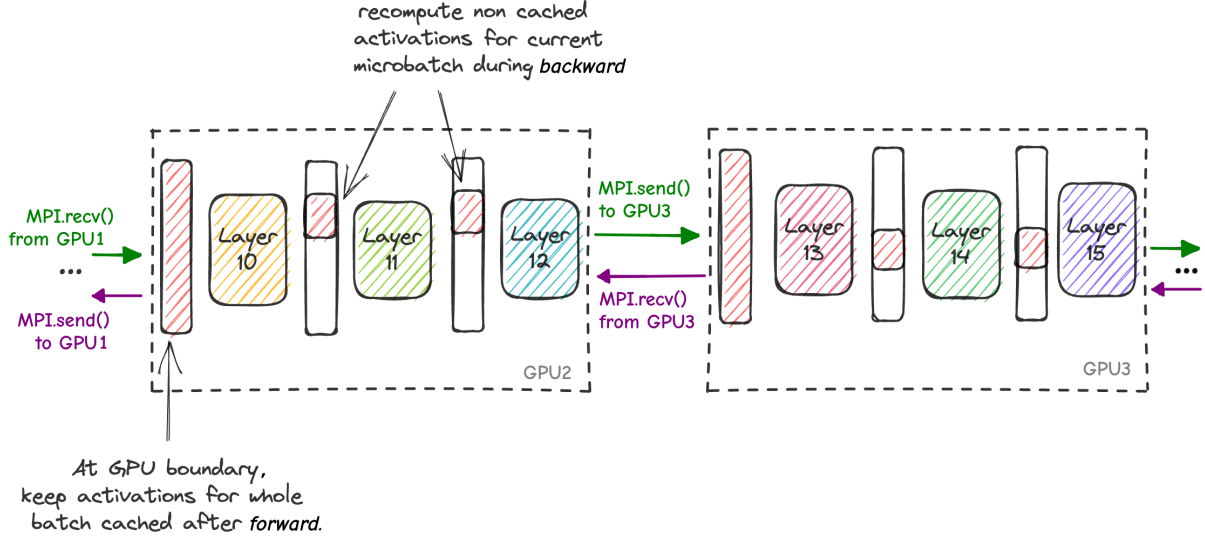


Figure 4.9: GPipe gradient checkpointing during the backward pass, illustrated for two GPUs only [14].

4.1.2 PipeDream

PipeDream is a different approach for pipeline parallel training [75]. It offers another option for decreasing the memory required by microbatch training, which is orthogonal to gradient checkpointing.

1. The PipeDream Algorithm

PipeDream starts the backward pass for a microbatch as soon as the last pipeline stage has completed the corresponding forward pass. We can discard the cached activation for the m -th microbatch as soon as we perform the corresponding backward pass. With PipeDream, this backward pass happens earlier than in GPipe, which reduces the required memory.

Figure 4.10 plots a PipeDream schedule, with four GPUs ($n = 4$) and eight microbatches ($m = 8$). Indeed, this schedule is called PipeDream Flush 1F1B. Blue boxes represent the forward passes, numbered with the associated microbatch identifier, while the green boxes represent the backward passes.

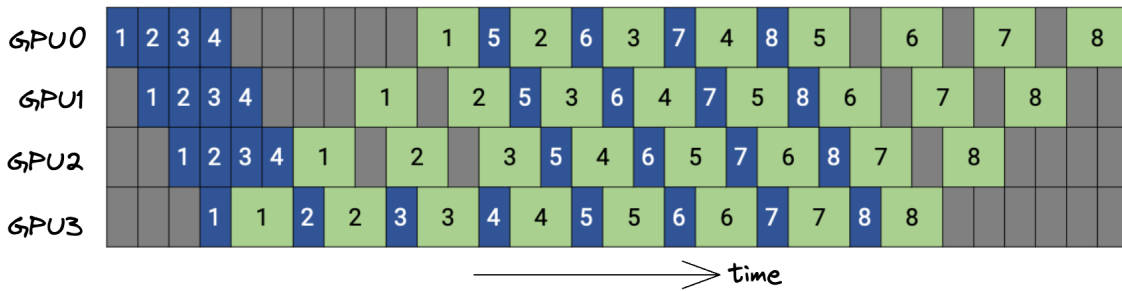


Figure 4.10: PipeDream schedule [145].

For both GPipe and PipeDream, the required memory for caching activations is expressed by Equation 4.4, when we do not consider gradient checkpointing.

$$O\left(\max_microbatches_in_flight \cdot microbatch_size \cdot \frac{\#layers}{\#GPUs}\right) \quad (4.4)$$

With the PipeDream schedule drawn in Figure 4.10, we have at most as many microbatches in flight as the pipeline depth. A microbatch is **in-flight** when it requires more than one forward pass, and it has not completed all the backward passes yet. The **pipeline depth** is the number of GPUs that process a microbatch until all gradients for that microbatch have been calculated. This becomes obvious by looking at GPU0 in Figure 4.10. During the steady state, GPU0 launches the forward pass for a new microbatch only after completing a backward pass of an earlier microbatch. The steady state is the time when the maximum memory is needed, and occurs after the warm-up phase, as indicated in Figure 4.11.

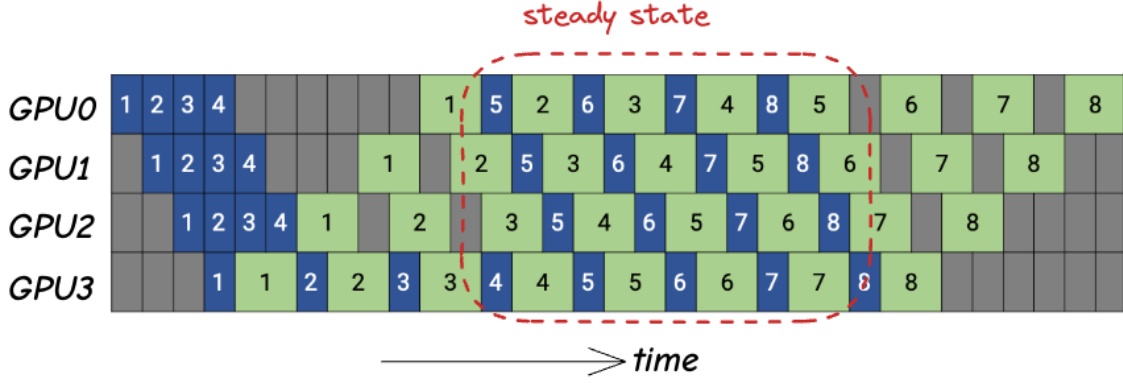


Figure 4.11: PipeDream steady state [75].

Considering the example included in Figure 4.12, we see that PipeDream has a maximum of four microbatches in flight, while with GPipe there are eight microbatches, doubling the memory required to cache the activations. As we have eight microbatches per batch in this example, GPipe will first compute the forward pass for all microbatches before starting the first backward pass.

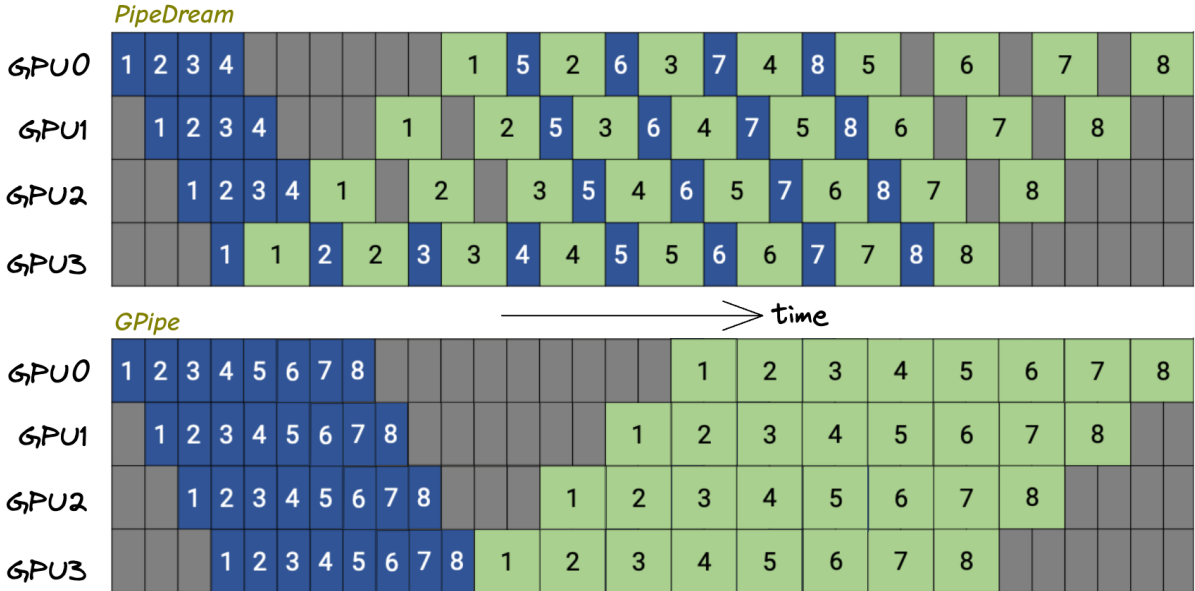


Figure 4.12: Comparing PipeDream and GPipe schedules for 4 GPUs and 8 microbatches.

In terms of bubble fraction, Figure 4.12 clearly shows that there is no difference between PipeDream and GPipe. The bubble is a result of the inherent dependencies between the operations on the microbatches, which PipeDream does not change. Visually, looking at the PipeDream schedule included in Figure 4.12, if we shift the blue boxes of the forward pass left and the green boxes of the backward pass right, we get the GPipe schedule. This explains why the bubble fraction is the same.

There are other variations of the PipeDream schedule, and the presented one is called 1F1B because during the steady state each node alternates between running a forward pass and a backward pass. Despite that, the schedule presented in Figure 4.10 is still sequentially consistent.

Since PipeDream does not have an end-of-batch global gradient synchronization across all the GPUs, a naive implementation of 1F1B schedule can easily lead to the forward and backward passes of one microbatch using different versions of the model's weights, thus lowering the learning efficiency. PipeDream proposed the following techniques to tackle this issue: (i) **weight stashing**, where each GPU keeps track of several model versions and makes sure that the same version of weights is used in the forward and backward passes across one minibatch, (ii) **vertical synchronization**, where the version of model weights flows between GPUs processing the same stage together with activations and gradients. Then the computation adopts the corresponding stashed version propagated from the previous worker. This process maintains the model version consistency across GPUs. Note that it is asynchronous, and as so it is different from GPipe [208].

At the beginning of a training session, PipeDream first profiles the memory required by computations and the duration that each model layer resides in memory, and then it optimizes a solution for partitioning layers into pipeline stages, which constitutes a dynamic programming problem. Figure 4.13 presents the PipeDream accuracy vs. the training time for different configurations of a VGG16 network, with 4 and 16 nodes. We observe that a larger number of stage nodes allows the same precision to be achieved in a much shorter time and the asynchronous parallel configuration exhibits the worst performance. The figure also plots the training time speedup for different parallelism configurations relative to a single node configuration. PipeDream using 8 stages achieved the best performance.

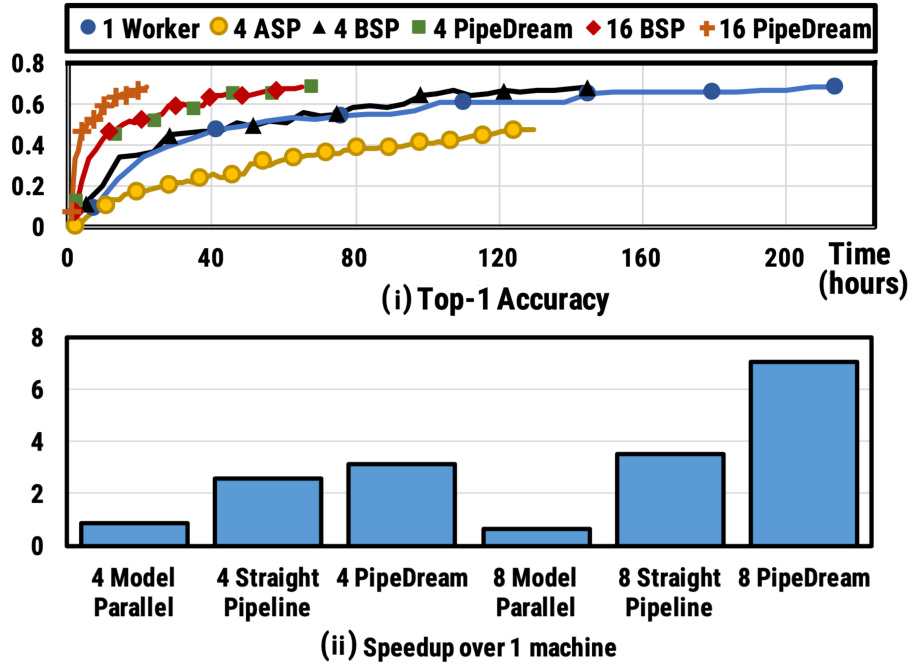


Figure 4.13: Results of PipeDream: (i) Accuracy vs. time for different configurations of VGG16 with 4 and 16 workers. The integer numbers in the plot caption indicate the number of stage nodes used, ASP means asynchronous parallel, and BSP stands for bulk synchronous parallel; (ii) Training time speedup for different parallelism configurations. Straight pipeline refers to pipeline parallelism without data parallelism [75].

2. PipeDream Variants

Two variations of PipeDream were proposed in [144] to reduce the memory required by the stashed model versions. **PipeDream-flush** adds a globally synchronized pipeline flush periodically, just like GPipe (Figure 4.14). In this way, it greatly reduces the required memory by only maintaining a single version of the model weights in memory and sacrificing the throughput slightly.

PipeDream-2BW maintains two versions of the model weights in memory, and the suffix "2BW" means double-buffered weights. It requires less memory than PipeDream and GPipe, and also avoids GPipe's expensive pipeline flushes. **Flushing a pipeline** means not scheduling any new operations until all currently

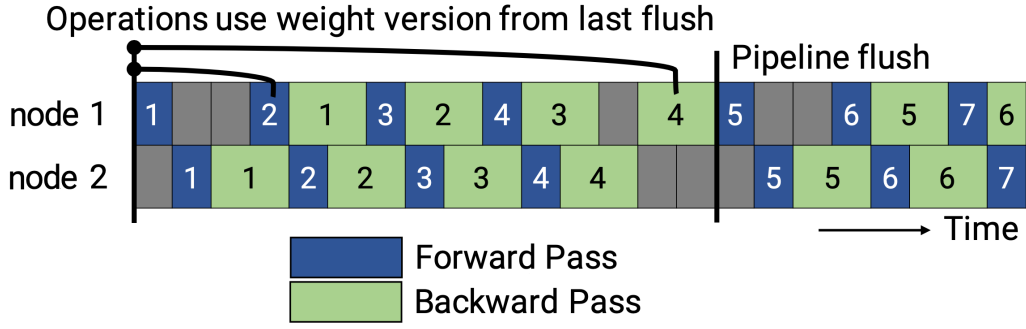


Figure 4.14: Pipeline scheduling in PipeDream-flush [144].

scheduled operations are done processing. Once the pipeline is flushed, we know that the gradients, accumulated over the microbatches, are sequentially consistent. Then we can perform the optimizer step.

PipeDream-2BW generates a new model version every k microbatches and k must be larger than the pipeline depth m : $k > m$. A new and updated model version can not fully replace the old version immediately. In particular, in-flight microbatches cannot use the newest weight version for their backward passes, for example, microbatch 7 on node 3 at time $t = 21$, since the forward pass for this microbatch was already initiated (at $t = 19$) using an older weight version. Thus, newly generated weight versions need to be buffered for future use. However, the total number of weight versions that need to be maintained is at most 2, since the weight version used to generate a new weight version can immediately be discarded, as no future microbatches that pass through that stage use the old weight version any longer. For example, in Figure 4.15, each worker can discard $W_i^{(0)}$ once they are done processing the backward pass for microbatch 8, since all subsequent microbatches use a later weight version for both their forward and backward passes.

Figure 4.15 shows the pipeline scheduling for the PipeDream-2BW’s double-buffered weight update scheme. It is assumed that the backward passes take twice as long as forward passes. As indicated, PipeDream-2BW only stashes two weight versions at every node, reducing the total memory footprint while no longer requiring expensive pipeline stalls. $W_i^{(v)}$ corresponds to the weights on node i belonging to version v , which contains weight gradients generated from microbatch v . New weight versions are generated in the checkered green boxes.

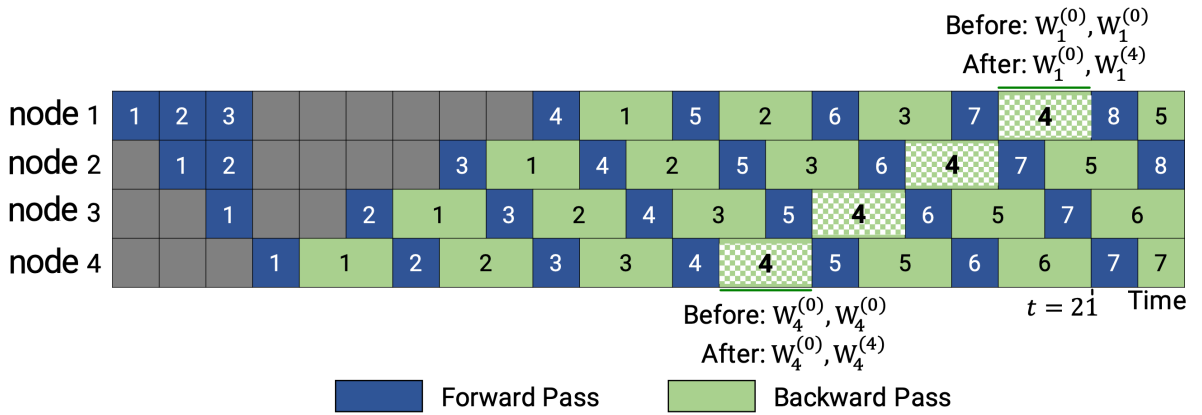


Figure 4.15: Pipeline scheduling for PipeDream-2BW [144].

3. Communication Volume

Let us briefly discuss the volume of network communication that is necessary to implement pipeline parallelism. This analysis is the same for GPipe and PipeDream. For simplicity, let us assume a model with only dense layers, where all have an equal dimension of N . During the forward pass, each GPU will send and receive data of size $batchsize \cdot N$. The same holds for the backwards pass, bringing our total communication volume to $(\#GPUs - 1) \cdot 2 \cdot batchsize \cdot N$ floats. The "-1" term comes from the initial GPU not having to receive and the last GPU not having to send anything.

Compare this to data parallelism, where each GPU has to AllReduce the gradients for all its layers. In our dense model example, using Ring AllReduce, each GPU needs to transfer roughly $2 \cdot \frac{\#layers \cdot N^2}{\#GPUs}$ floats. Depending on the configuration of our model and training setup, data parallelism may be more communication intensive. However, as we saw we can overlap the data parallel communication quite well, which is not possible with pipeline parallelism.

4.1.3 Combining Data and Pipeline Parallelisms

So far, we discussed three ways of implementing pipeline parallelism: naive model parallelism, GPipe, and PipeDream. Next, we show how pipeline parallelism can be combined with data parallelism, allowing us to use even bigger minibatch sizes without running out of memory.

Data and pipeline parallelism are orthogonal and can both be used at the same time, as long as the minibatch size is big enough to allow a reasonable microbatch size. In pipeline parallelism, each GPU needs to communicate with the next pipeline stage during the forward pass, as well as with the previous pipeline stage during the backward pass. In data parallelism, each GPU needs to communicate with all other GPUs to which the same model layers have been allocated. We need to AllReduce the gradients among all layer replicas after the pipeline has been flushed. We can overlap the AllReduce communication with the backward pass computation of the final microbatch to reduce training time, just as in regular data parallel training.

In practice, the orthogonal communication partners in pipeline and data parallelism are implemented using MPI Communicators. These form subgroups of all GPUs and allow performing collective communication only within a subgroup. Any given GPU- i will be part of two communicators, one containing all GPUs that hold the same layer slice as GPU- i (data parallelism), and one containing the GPUs that hold the other layer slices of GPU- i model replica (pipeline parallelism). See Figure 4.16 for an example.

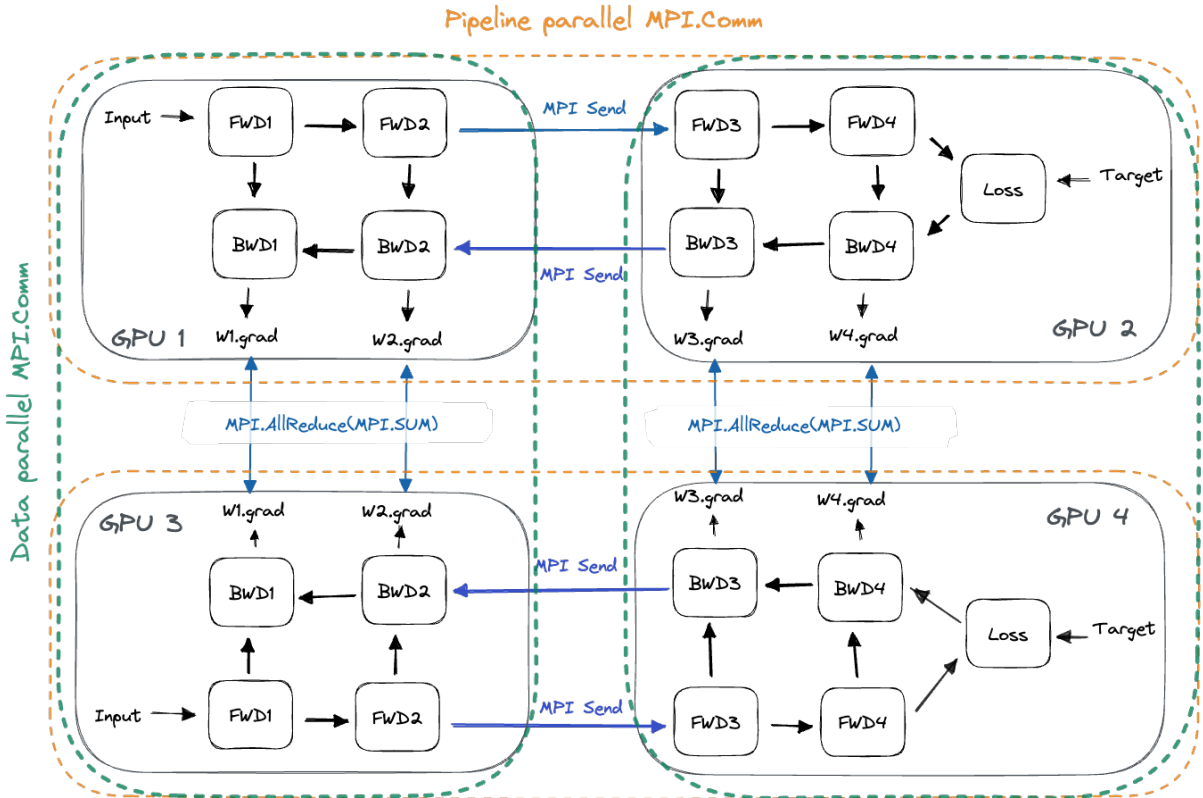


Figure 4.16: Combine data parallelism and pipeline parallelism [14].

4.1.4 Pipeline Parallelism in Available Frameworks

1. FairScale

FairScale is a PyTorch extension library for high performance and large scale training. It provides `fairscale.nn.Pipe`, an implementation of GPipe that was adopted from `torchgpip`. For example, to run a model composed of a Linear layer, a ReLU activation, and second Linear layer, on two GPUs, we need to include the layers in a `torch.nn.Sequential` container and then wrap it with `fairscale.nn.Pipe` (Listing 4.1). According to this code, the first two layers will run on GPU 0 and the last layer on GPU 1.

```
1 import fairscale
2 import torch
3 import torch.nn as nn
4
5 model = nn.Sequential(
6     torch.nn.Linear(10, 10),
7     torch.nn.ReLU(),
8     torch.nn.Linear(10, 5)
9 )
10 model = fairscale.nn.Pipe(model, balance=[2, 1])
```

Listing 4.1: Partition a model in two pipeline stages with FairScale.

2. DeepSpeed

DeepSpeed provides hybrid data and pipeline parallelism and can be further combined with model parallelism such as Megatron-LM. PyTorch's `torch.nn.Sequential` is a convenient container for expressing pipeline parallel models and can be parallelized by DeepSpeed with no modification, as indicated in Listing 4.2. `PipelineModule` uses its `layers` argument as the sequence of layers that comprise the model. After initialization, `net` is divided into two pipeline stages and its layers moved to the corresponding GPUs. If more than two GPUs are present, DeepSpeed will also use hybrid data parallelism [196].

```
1 from deepspeed.pipe import PipelineModule
2
3 net = nn.Sequential(
4     nn.Linear(in_features, hidden_dim),
5     nn.ReLU(inplace=True),
6     nn.Linear(hidden_dim, out_features)
7 )
8 net = PipelineModule(layers=net, num_stages=2)
```

Listing 4.2: Partition a model in two pipeline stages with DeepSpeed.

The performance of pipeline parallel training strongly relies on load balancing the work assigned to the different stages. DeepSpeed provides several mechanisms for partitioning the model across GPUs. These strategies can be set with the `partition_method` argument of `PipelineModule`. The supported partitioning methods are: (i) "parameters", which balances the number of trainable parameters on each pipeline stage. It is useful in memory-constrained environments and when the size of a layer is proportional to the computation time, (ii) "type:[regex]", which balances the layers whose class names match the regular expression [regex]. For example, `partition_method="type:transformer"` would balance the number of transformer layers per stage, (iii) "uniform" balances the number of layers per stage.

Creating a model in a `Sequential` container and pass it to a `PipelineModule` is adequate for specifying that the layers can work in pipeline. However, this approach does not scale to massive models because each worker node replicates the whole model in CPU memory. DeepSpeed provides a `LayerSpec` class that delays the construction of modules until the model layers have been partitioned across nodes. Therefore, each node will only allocate the layers that have been assigned to it, and the memory required in the CPU will be the model size and not #nodes times the model size.

As an example, consider the partial AlexNet model included in Listing 4.3, modified to use `LayerSpecs`. The model description is almost the same as in a sequential version, the differences are (i) the model inherits the class `PipelineModule` and (ii) we wrap each layer of the architecture with `LayerSpec`, such as, `nn.ReLU(inplace=True)` is replaced by `LayerSpec(nn.ReLU, inplace=True)`.


```

1 from deepspeed.pipe import PipelineModule, LayerSpec
2 class AlexNetPipe(PipelineModule):
3     def __init__(self, num_classes=10, **kwargs):
4         self.num_classes = num_classes
5         specs = [
6             LayerSpec(nn.Conv2d, 3, 64, kernel_size=11, stride=4, padding=2),
7             LayerSpec(nn.ReLU, inplace=True),
8             ...
9             LayerSpec(nn.ReLU, inplace=True),
10            LayerSpec(nn.Linear, 4096, self.num_classes),
11        ]
12    super().__init__(layers=specs, loss_fn=nn.CrossEntropyLoss(), **kwargs)

```

Listing 4.3: A memory efficient description of a model in DeepSpeed.

Some models can not be fully pipelined because some layers are reused by multiple modules of the architecture. For example, a transformer commonly uses an embedding layer early in the pipeline to map text tokens to hidden state embeddings, and then reuses the embedding layer to map the hidden states back to text at the end of the pipeline. If the model was restricted to be fully pipelined, this embedding layer reutilization would block us from exploring pipeline parallelism.

DeepSpeed provides a `TiedLayerSpec` that is an extension of `LayerSpec`. `TiedLayerSpec` requires a `key` as an additional argument. Each reutilization of a layer is wrapped with `TiedLayerSpec`, and the `key` identifies where the layer is reused. A tied layer is replicated on every pipeline stage that reuses it. Training proceeds as usual, but an additional `AllReduce` of the tied gradients is added after all backward passes complete. This `AllReduce` ensures that the weights of the tied layer remain synchronized across pipeline stages.

3. NVIDIA Megatron

NVIDIA provides significant support for distributed training on GPUs. Here we mention two of them: Megatron-LM and Megatron Core. Megatron-LM is a framework that includes Megatron Core plus configured scripts for distributed training. It is adequate for training large transformer models distributedly and for quick experimentation. Megatron Core is a composable library with GPU-optimized building blocks for custom training frameworks. It provides transformer building blocks, advanced parallelism strategies (tensor, pipeline, data, expert, and context parallelism), mixed precision support, and model architectures. It is adequate for building custom training pipelines.

The `pipeline_parallel` package of Megatron Core provides two distinct pipeline parallelism schedules, one with and the other without overlap of communication with computation [145]. The package also contains methods for the point-to-point communication that is necessary between pipeline stages.

NVIDIA recommends pipeline parallelism for training very deep models, with more than 50 layers, to combine it with tensor parallelism in large models, and to facilitate the distribution of the required memory across GPUs. For example, to train LLaMA on custom data and explore pipeline parallelism, by splitting the model in 8 pipeline stages, and allocating 4 layers to each virtual pipeline stage, we can run the following command. The `pretrain_gpt.py` script is provided in the NVIDIA Megatron-LM github ¹.

```

torchrun -nproc_per_node=8 pretrain_gpt.py
    -pipeline-model-parallel-size 8
    -num-layers-per-virtual-pipeline-stage 4
    -num-layers 32
    -hidden-size 4096
    -num-attention-heads 32
    -seq-length 2048
    -data-path /path/to/our/data
    ...

```

¹<https://github.com/NVIDIA/Megatron-LM/tree/main>

4. Amazon NeuronxDistributed

In AWS NeuronxDistributed, the pipeline parallelism is focused on transformer models, where each computation core is assigned with a subset of transformer layers. NeuronxDistributed uses Pytorch FX to trace the model and do partition on the FX intermediate representation. Pytorch FX is a toolkit that allows us to transform `nn.Module` instances, and consists of three main components: a symbolic tracer, an intermediate representation, and Python code generation. The users of NeuronxDistributed only have to specify where to cut the model to create the pipeline stages, and the framework will cut the pipeline stages and assign the corresponding modules to each computation core automatically. Automatic partition will be supported in the future.

5. Pytorch

Pipeline parallelism (PP) can be an effective technique for: large scale training, bandwidth limited clusters, and large model inference. These scenarios have an aspect in common, the computation per device cannot hide the communication on conventional data parallelism, for example, the cost of the AllGather operations in fully sharded data parallel (FSDP).

On the other hand, pipeline parallelism is more difficult to implement than data parallelism because we have to partition the execution of a model in addition to the model weights. The partitioning of execution often requires intrusive changes to our model. Another aspect of complexity comes from scheduling microbatches in a distributed environment, where data dependencies have to be considered.

The `torch.distributed.pipelining` package makes it easier to implement pipeline parallelism. It consists of two parts: a splitting frontend and a distributed runtime. The splitting frontend takes our model and splits it into partitions, considering the data dependencies. The distributed runtime executes the pipeline stages on different devices in parallel, handling aspects such as minibatch split in microbatches, scheduling the execution of stages' forward and backward passes, and communication between partitions.

Overall, the pipelining package provides the following features: (i) splitting the model architecture based on simple directives, (ii) pipeline scheduling, including GPipe, 1F1B, interleaved 1F1B and looped BFS schedules, and providing the infrastructure for writing customized schedules, (iii) complete support of inter-host pipeline parallelism, (iv) can be combined with other PyTorch parallel techniques, such as data and tensor parallelism.

5a. Build PipelineStages

Before we can use a `PipelineSchedule`, we have to create a `PipelineStage` object for wrapping the model partition allocated to each stage. A `PipelineStage` is responsible for allocating communication buffers and creating send/receive operations to communicate with its neighbor stages. It handles the intermediate buffers for the outputs, of the forward pass, that have not been consumed yet, and it provides the support for running the backward pass of the associated model's partition.

A `PipelineStage` needs to know the input and output shapes of the associated model's partition, in order to correctly allocate the communication buffers. A `PipeliningShapeError` event will be raised if the runtime shapes do not match the specified shapes. When combining PP with other forms of parallelism or training with mixed precision, these additional techniques must be taken into account when informing `PipelineStage` about the shape and data type of the stage module's output.

We can construct a `PipelineStage` instance directly, by passing to the constructor an argument that is an `nn.Module` containing the parcel of the model that must run on this stage. This step may require changing the original model description.

Alternatively, the splitting frontend of the package can employ graph partitioning techniques to split our model into a series of `nn.Module` automatically. This alternative requires that the model is traceable with `torch.export`, for producing a graph representing the tensor computations associated to the provided module. At the moment, combining the resulting `nn.Module` with other parallelism techniques is experimental.

5b. Execute stages with PipelineSchedule

The next step is to attach each PipelineStage to a pipeline schedule, and run the schedule with adequate input data. Listing 4.4 contains an example using GPipe schedule.

```
1 from torch.distributed.pipelining import ScheduleGPipe
2
3 # Create a schedule
4 schedule = ScheduleGPipe(stage, n_microbatches)
5 # A minibatch of input data
6 x = torch.randn(batch_size, in_dim, device=device)
7
8 # Run the pipeline with input minibatch 'x'
9 # 'x' will be divided into microbatches automatically
10 if rank == 0:
11     schedule.step(x)
12 else:
13     output = schedule.step()
```

Listing 4.4: Associate a schedule to a model stage and run a forward step over this schedule.

`torchrun` is a python script to the module `torch.distributed.run`. Thus, calling `torchrun example.py` is equivalent to run `python -m torch.distributed.run example.py`. It can be used for single-node distributed training, to spawn processes per device. It can be used either for CPU training or GPU training. `torchrun` can also be used in multi-node distributed training, by spawning up multiple processes on each node. In both cases of distributed training, single- or multi-node, `torchrun` launches the specified number of processes per node, using the option `-nproc-per-node`.

The code included in Listing 4.4 has to be launched for each worker node of the pipeline, thus we can use `torchrun` to launch multiple processes, as occurs in the next command.

```
torchrun -nproc_per_node=2 example.py
```

5c. Splitting a model manually

To directly instantiate a PipelineStage, we are responsible for providing an `nn.Module` instance that contains the relevant `nn.Parameters` and `nn.Buffers`, and defines a `forward` method that executes the operations relevant for this stage. Listing 4.5 contains a partial description of a Transformer model, an example of a model's architecture adequate for easy partitioning.

```
1 class Transformer(nn.Module):
2     def __init__(self, model_args: ModelArgs):
3         super().__init__()
4         self.tok_embeddings = nn.Embedding(...)
5
6         # Using a ModuleDict lets us delete layers without affecting names,
7         # ensuring checkpoints will correctly save and load
8         self.layers = torch.nn.ModuleDict()
9         for layer_id in range(model_args.n_layers):
10             self.layers[str(layer_id)] = TransformerBlock(...)
11
12         self.output = nn.Linear(...)
13
14     def forward(self, tokens: torch.Tensor):
15         # Handling layers being 'None' at runtime enables easy pipeline splitting
16         h = self.tok_embeddings(tokens) if self.tok_embeddings else tokens
17
18         for layer in self.layers.values():
19             h = layer(h, self.freqs_cis)
20
21         h = self.norm(h) if self.norm else h
22         output = self.output(h).float() if self.output else h
23         return output
```

Listing 4.5: A partial transformer model.

A model defined in this manner can be easily organized in stages by first initializing the whole model, using a meta-device to avoid out-of-memory errors, deleting the undesired layers for a given stage, and then creating a `PipelineStage` that wraps the model (Listing 4.6). When combining PP with data or model parallelism techniques, we may have to provide `output_args` when the output shape or data type of the model chunk is affected by the other forms of parallelism.

```

1 with torch.device("meta"):
2     assert num_stages == 2, "This is a simple 2-stage model splitting"
3
4     # We instantiate the entire model, then delete the parts we do not want
5     # in each stage. This can be done using a helper function that
6     # automatically divides the layers across stages.
7     model = Transformer()
8
9     if stage_index == 0:
10        # Prepare the first stage
11        del model.layers["1"]
12        model.norm = None
13        model.output = None
14
15    elif stage_index == 1:
16        # Prepare the second stage
17        model.tok_embeddings = None
18        del model.layers["0"]
19
20    from torch.distributed.pipelining import PipelineStage
21    stage = PipelineStage(
22        model,
23        stage_index,
24        num_stages,
25        device,
26    )

```

Listing 4.6: Manually splitting a transformer model in two stages. If `stage_index` is 0, it is created stage 0, otherwise, if `stage_index` is 1, it is created stage 1.

5d. Splitting a model automatically

There are cases where we have a full model description and do not want to modify it into a sequence of model partitions. In these cases, we can resort to the pipeline API. Listing 4.7 provides another version of partial transformer model.

```

1 class Model(torch.nn.Module):
2     def __init__(self) -> None:
3         super().__init__()
4         self.emb = torch.nn.Embedding(10, 3)
5         self.layers = torch.nn.ModuleList(
6             Layer() for _ in range(2)
7         )
8         self.lm = LMHead()
9
10    def forward(self, x: torch.Tensor) -> torch.Tensor:
11        x = self.emb(x)
12        for layer in self.layers:
13            x = layer(x)
14        x = self.lm(x)
15        return x

```

Listing 4.7: A second version of a transformer model.

If we print the model architecture, we get the summary included in Listing 4.8. We identify multiple hierarchies in the architecture, which make it harder to split the model manually.

```

1 Model(
2     (emb): Embedding(10, 3)
3     (layers): ModuleList(
4         (0-1): 2 x Layer(
5             (lin): Linear(in_features=3, out_features=3, bias=True)
6         )
7     )
8     (lm): LMHead(
9         (proj): Linear(in_features=3, out_features=3, bias=True)
10    )
11 )

```

Listing 4.8: Summary of the transformer model architecture defined in Listing 4.7.

Let us see how the `pipeline` class works through a simple example included in Listing 4.9. The `pipeline` splits our model according to the provided `split_spec` argument, where `SplitPoint.BEGINNING` means that we want to add a split point before the execution of certain submodule in the `forward` method of the model, and `SplitPoint.END` signals `pipeline` that we want a split point after the indicated submodule. In this particular case, we indicate that we want to add a split point before the execution of `layers.1`, where `layers` is a module and 1 is submodule of `layers`.

```

1 from torch.distributed.pipeline import pipeline, SplitPoint
2 # An example micro-batch input
3 x = torch.LongTensor([1, 2, 4, 5])
4 pipe = pipeline(
5     module      = model,
6     mb_args     = (x,),
7     split_spec = {
8         "layers.1": SplitPoint.BEGINNING,
9     }
10 )

```

Listing 4.9: Specify the point where a model must be split.

If we print the `pipe` object created in Listing 4.9, we obtain the summary included in Listing 4.10. The model partitions are represented by `GraphModules`, `submod_0` and `submod_1`, each one defined with the operations, weights and hierarchies of the original description. In addition, a top-level `forward` method is created to capture the data flowing between the created partitions. This data flow will be reformulated by the `pipeline` runtime later, to run distributedly.

```

1 GraphModule(
2     (submod_0): GraphModule(
3         (emb): InterpreterModule()
4         (layers): Module(
5             (0): InterpreterModule(
6                 (lin): InterpreterModule()
7             )
8         )
9     )
10    (submod_1): GraphModule(
11        (layers): Module(
12            (1): InterpreterModule(
13                (lin): InterpreterModule()
14            )
15        )
16        (lm): InterpreterModule(
17            (proj): InterpreterModule()
18        )
19    )
20 )
21 def forward(self, x):
22     submod_0 = self.submod_0(x); x = None
23     submod_1 = self.submod_1(submod_0); submod_0 = None
24     return (submod_1,)

```

Listing 4.10: The transformer model architecture after being restructured by the `pipeline` class.

The pipe object provides a method for retrieving the model partitions, as indicated next.

```
stage_mod: nn.Module = pipe.get_stage_module(stage_idx)
```

The returned `stage_mod` is a `nn.Module`, that allows us to instantiate an optimizer, save and load model checkpoints, and apply other forms of parallelism. The pipe object also allows us to create a distributed stage's runtime on a device given a group of processes, as indicated in the following line of code.

```
stage = pipe.build_stage(stage_idx, device, group)
```

Alternatively, we can build the stage runtime after applying some modification to `stage_mod`, by using the functional version of the `build_stage` method. Listing 4.11 provides an example where we add data parallelism to a pipeline stage of the model, and we create the pipeline stage runtime only after the two model modifications were applied. The pipeline frontend uses a tracer, `torch.export`, to capture a model into a single graph.

```
1 from torch.distributed.pipelining import build_stage
2 from torch.nn.parallel import DistributedDataParallel
3
4 dp_mod = DistributedDataParallel(stage_mod)
5 info = pipe.info()
6 stage = build_stage(dp_mod, stage_idx, info, device, group)
```

Listing 4.11: Adding data parallelism to a pipeline stage of a model, and creating the pipeline stage runtime only after both types of parallelism were applied.

5e. Details about model splitting

To split a model, the `pipeline` class first turns the model into a directed acyclic graph by tracing the model execution flow using `torch.export`. Then, it groups together the operations and parameters needed by a stage into reconstructed submodules, such as `submod_0` and `submod_1`. `pipeline` does not only cut the structure of model in several parts, as occurs when we use the `Module.children` method, but also splits the forward method of the model. This is necessary because `Module.children()` lacks information about the following key aspects necessary for pipelining: (i) the execution order of child modules in the forward pass, (ii) the activation flows between child modules, and (iii) information about the functional operators that may exist between child modules, such as a ReLU activations or additions, which are not captured by `Module.children`. The `pipeline` class ensure that the forward behavior is completely considered. It also captures the activation flow between the partitions, helping the distributed runtime to make correct send/receive calls without the user having to modify the training loop.

Another flexibility of `pipeline` is that the split points can be at arbitrary level within the model structure. The original hierarchy in the model's structure will be reconstructed, without user intervention, in the created model stages. As a result, fully-qualified names (FQNs) referencing submodules or parameters are still valid after the model has been partitioned. This makes it possible to apply techniques as FSDP, tensor parallelism or model checkpointing to the partitioned modules without great code modifications.

5f. Implementing a custom schedule

The user can implement a custom pipeline schedule by extending one of the following two classes: `PipelineScheduleSingle` or `PipelineScheduleMulti`. `PipelineScheduleSingle` is for schedules that assign only one stage per rank and `PipelineScheduleMulti` is for schedules that assign multiple stages per rank. For example, `ScheduleGPipe` and `Schedule1F1B` are subclasses of `PipelineScheduleSingle`. Whereas, `ScheduleInterleaved1F1B`, `ScheduleLoopedBFS`, `ScheduleInterleavedZeroBubble`, and `ScheduleZBVZeroBubble` are subclasses of `PipelineScheduleMulti`.

5g. Apply pipeline parallelism to a transformer model

This section uses a GPT-based transformer to demonstrate how to train a model with the pipeline parallelism provided by the `torch.distributed.pipelining` package.

i. Setup

With `torch.distributed.pipelining` we split the execution of a model and schedule the computation on micro-batches. For this purpose, we adopted a simplified transformer decoder model. The model architecture has multiple transformer decoder layers, which is adequate to demonstrate how to split a model into different chunks. First, we define the model in Listing 4.12.

```
1 import torch
2 import torch.nn as nn
3 from dataclasses import dataclass
4
5 @dataclass
6 class ModelArgs:
7     dim: int = 512
8     n_layers: int = 8
9     n_heads: int = 8
10    vocab_size: int = 10000
11
12 class Transformer(nn.Module):
13     def __init__(self, model_args: ModelArgs):
14         super().__init__()
15
16         self.tok_embeddings = nn.Embedding(model_args.vocab_size, model_args.dim)
17
18         # Using a ModuleDict lets us delete layers without affecting names,
19         # ensuring checkpoints can be saved and loaded correctly
20         self.layers = torch.nn.ModuleDict()
21         for layer_id in range(model_args.n_layers):
22             self.layers[str(layer_id)] = nn.TransformerDecoderLayer(
23                 model_args.dim, model_args.n_heads
24             )
25
26         self.norm = nn.LayerNorm(model_args.dim)
27         self.output = nn.Linear(model_args.dim, model_args.vocab_size)
28
29     def forward(self, tokens: torch.Tensor):
30         # Handling layers being 'None' at runtime enables splitting the pipeline easily
31         h = self.tok_embeddings(tokens) if self.tok_embeddings else tokens
32
33         for layer in self.layers.values():
34             h = layer(h, h)
35
36         h = self.norm(h) if self.norm else h
37         output = self.output(h).clone() if self.output else h
38         return output
```

Listing 4.12: A simple transformer decoder model.

Then, we initialize the distributed training process. In this case, we are defining some global variables to use later in the code: the current process `rank`, the number of processes to use in training (`world_size`), and the computing device (Listing 4.13). We also define a few global variables that are specific to pipeline parallelism: `pp_group`, which is the process group that will be used for send/receive communications, `stage_index`, which in our case is a single rank per stage and as so the index is equivalent to the rank, and `num_stages`, which is equivalent to `world_size`.

The variable `num_stages` specifies the number of stages that will be considered in the pipeline parallelism schedule. For example, setting `num_stages=4` means that a microbatch will go through 4 forward and 4 backward steps before it is completed. The `stage_index` is necessary for the framework to know how to communicate between stages. For example, in the stage with `stage_index=0` is where the data is obtained from the data loader and where it is not necessary to receive data from any previous stage to perform the computations.

```

1 import os
2 import torch.distributed as dist
3 from torch.distributed.pipelining import (
4     pipeline, SplitPoint, PipelineStage, ScheduleGPipe
5 )
6
7 global rank, device, pp_group, stage_index, num_stages
8
9 def init_distributed():
10     global rank, device, pp_group, stage_index, num_stages
11     rank = int(os.environ["LOCAL_RANK"])
12     world_size = int(os.environ["WORLD_SIZE"])
13     device = torch.device(f"cuda:{rank}") if torch.cuda.is_available() \
14         else torch.device("cpu")
15     dist.init_process_group()
16
17     # This group can be a sub-group in the N-D parallel case
18     pp_group = dist.new_group()
19     stage_index = rank
20     num_stages = world_size

```

Listing 4.13: Initializing the distributed training process.

ii. Partition the transformer model

As we discussed before, there are two different ways of partitioning a model. The first one is the manual mode in which we can manually create two instances of the model by deleting portions of the model's structure. In this example, we consider only two stages, or two ranks, and the model is split in two parts (Listing 4.14).

```

1 def manual_model_split(model) -> PipelineStage:
2     if stage_index == 0:
3         # Prepare the first model's stage with layers 0:3
4         for i in range(4, 8):
5             del model.layers[str(i)]
6         model.norm = None
7         model.output = None
8
9     elif stage_index == 1:
10        # Prepare the second model's stage with layers 4:7
11        for i in range(4):
12            del model.layers[str(i)]
13        model.tok_embeddings = None
14
15    stage = PipelineStage(
16        model,
17        stage_index,
18        num_stages,
19        device,
20    )
21    return stage

```

Listing 4.14: Split the simple transformer decoder model in two chunks manually.

The first stage does not have layer normalization nor the output layer, and it only includes the first four transformer blocks. The second stage does not have the input embedding layers, but includes the final four transformer blocks, and the output layers. The function then returns the PipelineStage of the current rank.

The second splitting method uses the PyTorch tracer to automatically split the model according to the `split_spec` argument. In this case, we inform `torch.distributed.pipelining` where to cut the model. In Listing 4.15, we inform the pipeline API to cut the model before the 4-th transformer decoder block, mirroring the manual split implemented in Listing 4.14. Similarly, we can retrieve a PipelineStage by calling the `build_stage` method after specifying where to cut the model.


```

1 def tracer_model_split(model, example_input_microbatch) -> PipelineStage:
2     pipe = pipeline(
3         module      = model,
4         mb_args     = (example_input_microbatch,),
5         split_spec = {
6             "layers.4": SplitPoint.BEGINNING,
7         }
8     )
9     stage = pipe.build_stage(stage_index, device, pp_group)
10    return stage

```

Listing 4.15: Function for splitting the simple transformer decoder model automatically, using the `pipeline` class.

iii. Define the main training function

In the `main` function, we will create a specific pipeline schedule that will be applied to the created stages. In the code included in Listing 4.16, the model is split manually, but line 16 can be uncommented to try the automatic split based on the graph created by the PyTorch tracer. We have to provide the schedule with the number of microbatches and the loss function that will be used to evaluate the outputs predicted by the model.

```

1 if __name__ == "__main__":
2     init_distributed()
3     num_microbatches = 4
4     model_args = ModelArgs()
5     model = Transformer(model_args)
6
7     # 1 - split the model manually
8     stage = manual_model_split(model)
9
10    # 2 - split the model with the help of the tracer
11    # stage = tracer_model_split(model, example_input_microbatch)
12
13    model.to(device)
14
15    def tokenwise_loss_fn(outputs, targets):
16        loss_fn = nn.CrossEntropyLoss()
17        outputs = outputs.reshape(-1, model_args.vocab_size)
18        targets = targets.reshape(-1)
19        return loss_fn(outputs, targets)
20
21    schedule = ScheduleGPipe(
22        stage,
23        n_microbatches = num_microbatches,
24        loss_fn         = tokenwise_loss_fn
25    )
26
27    for id in range(10000):
28        # Dummy data
29        x = torch.ones(32, 500, dtype=torch.long)
30        y = torch.randint(0, model_args.vocab_size, (32, 500), dtype=torch.long)
31        example_input_microbatch = x.chunk(num_microbatches)[0]
32        x = x.to(device)
33        y = y.to(device)
34
35        if rank == 0:
36            schedule.step(x)
37
38        elif rank == 1:
39            losses = []
40            output = schedule.step(target=y, losses=losses)
41            if id % 100 == 0:
42                print(f"[step {id}] microbatches losses {[f'{l.item():.4f}' for l in losses]}")
43
44    dist.destroy_process_group()

```

Listing 4.16: The main function to train the simple transformer model with pipeline parallelism.

The `step` method processes the entire minibatch and automatically splits it into microbatches based on the value of the `n_microbatches` passed as argument to the schedule. The microbatches are then operated on according to the schedule class we selected. Listing 4.16 uses the `GPipe` schedule, which executes all the forwards steps of the two stages, including all the microbatches of every stage, followed by all the backward steps. The output returned from rank 1 will be the same as if the model was on a single GPU processing the entire batch. We can also use a losses container to store the corresponding losses for each microbatch.

iv. Launch the distributed processes

Finally, we can use the following command to run our Python script with `torchrun`, in a single host, and using two processes. The script is written in a way that rank 0 performs the required logic for pipeline stage 0, and rank 1 performs the logic for pipeline stage 1.

```
torchrun -nnodes 1 -nproc_per_node 2 pipelining_tutorial.py
```

Training the model for 10k iterations on two GPUs, one NVIDIA Titan Xp and one NVIDIA RTX 2080 Ti, took 98 minutes and 26 seconds.

In conclusion, we applied pipeline parallelism to train a simple transformer model distributedly using the `torch.distributed.pipelining` classes. We explored setting up the environment, defining a transformer model, and partitioning it for distributed training. We discussed two methods of model partitioning, manual and tracer-based, and demonstrated how to schedule computations on micro-batches across different stages. Finally, we covered the execution of the pipeline schedule and the launch of distributed processes using `torchrun`.

4.2 Tensor Parallelism

Both model and pipeline parallelisms split a model vertically. But we can also horizontally partition the computation over one tensor across multiple devices, and this is considered tensor parallelism (TP). A paradigmatic example of tensor parallelism consists of horizontally splitting the multiplication of large matrices and distributing the partial multiplications to different GPUs. Each GPU computes part of the operation, and the partial results are gathered at the end.

Tensor parallelism (TP) was first proposed in [184] for large-scale model training. It is a technique that distributes the operations of a single layer across multiple devices, namely GPUs. Consider a multi-layer perceptron (MLP) block of a transformer model, which is built with a series of pairs of a fully-connected (Linear) layer and a non-linear GeLU activation. Thus, an input tensor X is multiplied by a weight matrix A to produce an output Y , which is then passed through the non-linear GeLU activation. Rather than allocating the entire matrix A to a single GPU, tensor parallelism partitions A into smaller blocks ($A_1, A_2, \dots, A_n$), each one processed a different GPU. The way in which matrix A is split is important. A naive approach would split the weight matrix A across its rows and partition the input X across its columns and, therefore, $Y = \text{GeLU}(XA)$ will be calculated as indicated in Equation 4.5 [189].

$$\begin{aligned} A &= \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \\ X &= [X_1 \ X_2] \\ Y &= \text{GeLU}(X_1 \cdot A_1 + X_2 \cdot A_2) \end{aligned} \tag{4.5}$$

The drawback of this partitioning scheme is that it imposes a synchronization point before the activation function, since the partial results $X_1 \cdot A_1$ and $X_2 \cdot A_2$ must be combined before we can apply the GeLU for obtaining the correct result. This is due to $\text{GeLU}(X_1 \cdot A_1 + X_2 \cdot A_2) \neq \text{GeLU}(X_1 \cdot A_1) + \text{GeLU}(X_2 \cdot A_2)$. Synchronization points penalize the overall computation time and can increase the training time significantly. Therefore, we must eliminate synchronizations whenever possible. An alternative to the naive partitioning is to split A along its columns and $Y = \text{GeLU}(XA)$ will be calculated as indicated in Equation 4.6.

$$\begin{aligned}
A &= [A_1 \ A_2] \\
[Y_1 \ Y_2] &= [\text{GeLU}(X \cdot A_1) \ \text{GeLU}(X \cdot A_2)]
\end{aligned} \tag{4.6}$$

This solution allows for each GPU to apply the activation function to its partial output independently, thereby dismissing the synchronization point and improving the overall training efficiency (Figure 4.17 (i)).

In Megatron-LM, tensor parallelism is also applied to the self-attention layers by splitting the key (K), the query (Q), and the value (V) matrices column-wise. In this way, each GPU processes a subset of the attention heads and performs the corresponding matrix multiplications locally (Figure 4.17 (ii)). This eliminates communication among GPUs during the attention calculation, since each GPU only needs the parcel of input that is relevant to its heads.

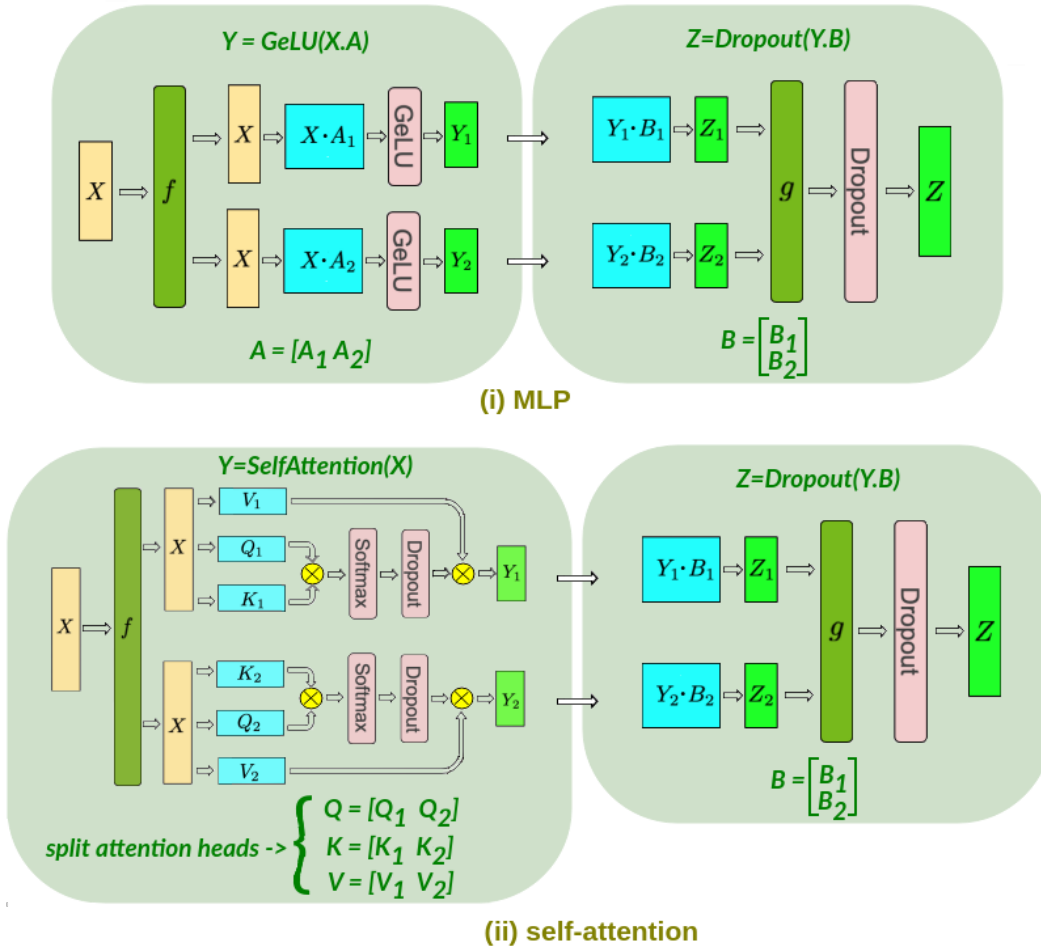


Figure 4.17: Blocks of a transformer with tensor parallelism. f and g are conjugate, since f is an identity operator in the forward pass and AllReduce in the backward pass, while g is an AllReduce in the forward pass and identity in the backward pass [184].

The second Linear layer of the MLP and the output Linear layer after self-attention are split row-wise, as illustrated in both right panels of Figure 4.17. This design leverages the fact that each GPU already has the complete output for its assigned attention heads, by concatenating those outputs locally, each GPU can process its own subset of rows without having to communicate with other GPUs. Thus, row-splitting B in both right panels avoids a synchronization point among GPUs.

The attention layer performs matrix-matrix multiplications between input X and the query weights Q , X and the key weights K , and X and the value weights V , using column-wise partitioning to execute the multiplications in parallel. The results of the first two multiplications are combined, using another matrix-matrix multiplication, to produce the score that is passed through the softmax . Finally, the output of softmax is multiplied by $X \cdot V$ (Equation 4.7) [208].

$$\text{Attention}(X, Q, K, V) = \text{softmax}\left(\frac{(X \cdot Q) \cdot (X \cdot K)^T}{\sqrt{d_k}}\right) \cdot (X \cdot V) \quad (4.7)$$

The necessary communication is handled using two conjugate operators: f , which corresponds to an AllReduce operation in the backward pass and an identity function in the forward pass, and g , which is an AllReduce operation in the forward pass and an identity function in the backward pass. This solution ensures that only two AllReduce operations occur in the forward pass and two in the backward pass, thereby keeping the synchronization cost low (Figure 4.18).

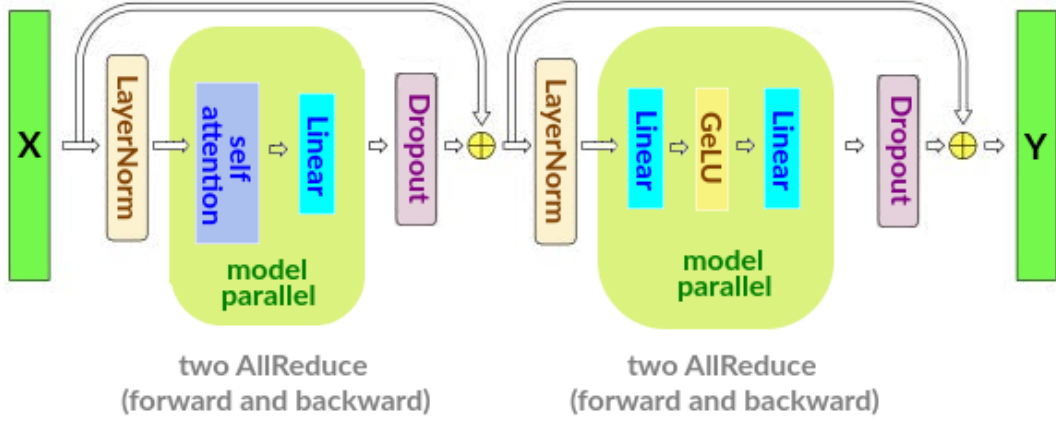


Figure 4.18: Communication in a tensor parallel transformer layer involves four total operations across its forward and backward passes [184].

Megatron-LM's tensor parallelism focus on maximizing the GPU independence to minimize communication, which is limited to the mandatory AllReduce operations. In this way, the synchronization points are minimized and GPUs are mostly occupied.

4.2.1 Practice with Tensor Parallelism in PyTorch

When a model does not fit in a single GPU, we have to shard it by multiple GPUs. There are two sharding methods: pipeline parallelism and tensor parallelism. Assuming that we have a model with 2 Linear layers, with 4×4 and 4×2 weight tensors, and 2 GPUs, the result of applying pipeline and tensor parallelism is depicted in Figure 4.19.

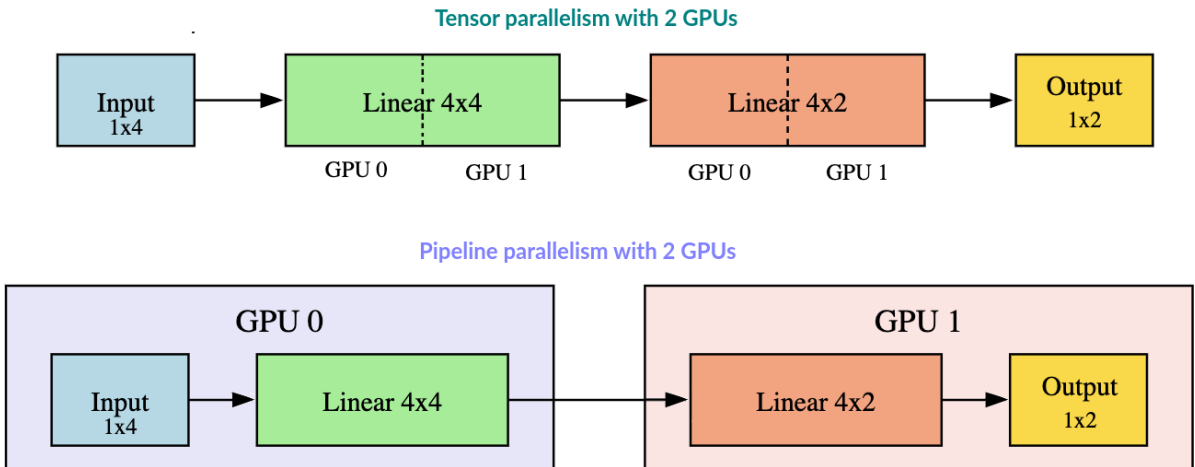


Figure 4.19: Comparing tensor and pipeline parallelisms considering two GPUs and a model with two Linear layers.

Tensor parallelism shards the Linear layers into multiple GPUs, and all GPUs get a parcel of every Linear layers. While pipeline parallelism splits the linear layers into multiple GPUs, and each GPU is assigned a

disjunct subset of the layers. Each method has its own pros and cons, but in this blog, we will focus on Tensor Parallelism using PyTorch. Tensor parallelism can be applied row-wise and column-wise. In row-wise tensor parallelism, we split a layer along the rows, while in column-wise tensor parallelism, we split a layer along the columns.

1. Row-Wise Tensor Parallelism

By using the same Linear layers defined above, when we apply row-wise tensor parallelism, the 4×4 weight tensor of the first layer is split along the rows into two tensors and each GPU is assigned a 2×4 tensor. GPU 0 gets a 2×4 tensor with the first two rows of the original tensor and GPU 1 gets a 2×4 tensor with the last two rows.

For the second Linear layer, we split the 4×2 weight tensor along the rows into two 2×2 tensors. Thus, GPU 0 gets a 2×2 tensor with the first two rows of the original tensor and GPU 1 gets a 2×2 tensor with the last two rows.

Since the input is a 1×4 tensor, it is split along the columns into two 1×2 tensors that are scattered to the GPUs. GPU 0 gets a 1×2 tensor containing the first two columns of the input tensor and GPU 1 also gets a 1×2 tensor containing the last two columns of the input tensor. Each GPU performs a matrix-matrix multiplication between the 1×2 input and the 2×4 weights for the first layer, i.e., $(1 \times 2) @ (2 \times 4) = (1 \times 4)$. After that, we aggregate the partial results from both GPUs, using the sum operator (Figure 4.20).

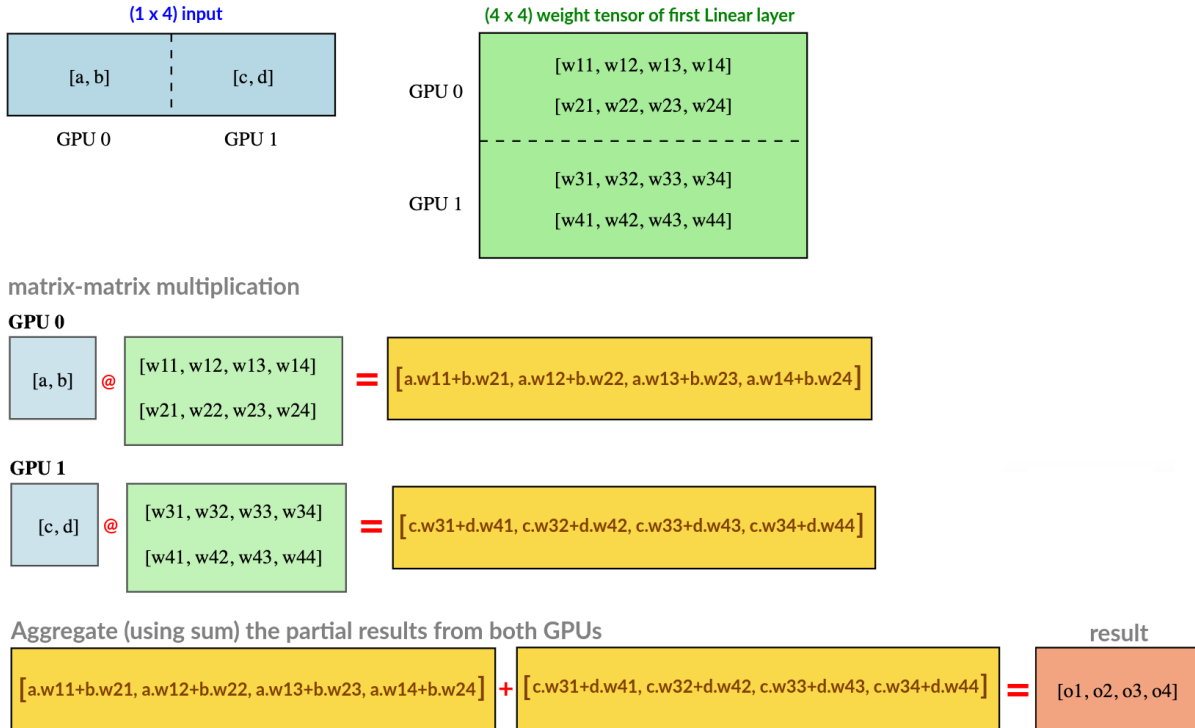


Figure 4.20: Illustration of row-wise tensor parallelism applied to the first Linear layer.

The output $[o_1, o_2, o_3, o_4]$ from the first Linear layer becomes the input of the second Linear layer. This tensor is split along the columns into two 1×2 tensors, which are scattered to the GPUs. GPU 0 gets a 1×2 tensor containing $[o_1, o_2]$ and GPU 1 gets $[o_3, o_4]$.

Each GPU performs a matrix-matrix multiplication between the 1×2 input and the 2×2 weights for the second layer, i.e., $(1 \times 2) @ (2 \times 2) = (1 \times 2)$. After that, we aggregate the partial results from both GPUs, using the sum operator (Figure 4.21).

2. Column-Wise Tensor Parallelism

Still considering the same Linear layers as in row-wise parallelism, if we apply row-wise tensor parallelism, the 4×4 weight tensor of the first layer is split along the columns into two tensors and each GPU is assigned a

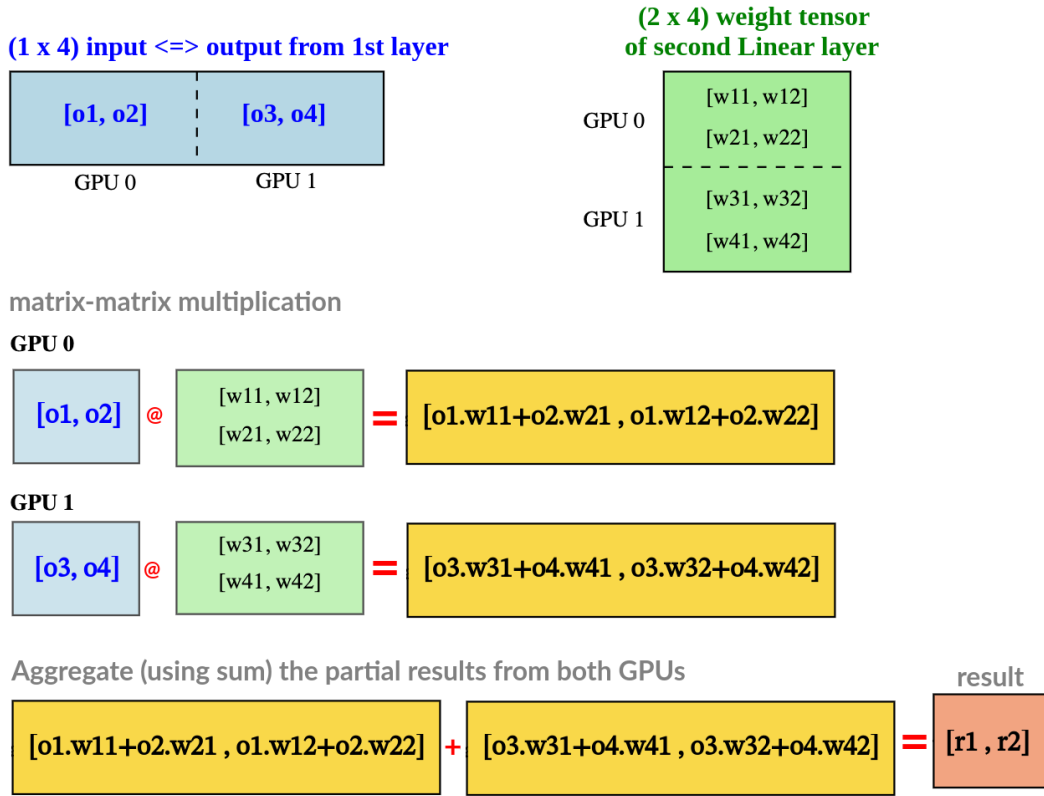


Figure 4.21: Illustration of row-wise tensor parallelism applied to the second Linear layer.

4×2 tensor. GPU 0 gets a 4×2 tensor with the first two columns of the original tensor and GPU 1 gets a 4×2 tensor with the last two columns.

The 1×4 input tensor is scattered as is to both GPUs. Then, each GPU performs a matrix-matrix multiplication between the 1×4 input and the 4×2 weights for the first layer, i.e., $(1 \times 4) @ (4 \times 2) = (1 \times 2)$. After that, we aggregate the partial results from both GPUs, using a concatenation to obtain a 1×4 tensor (Figure 4.22).

The output $[o_1, o_2, o_3, o_4]$ from the first Linear layer becomes the input of the second Linear layer. This tensor is scattered as is to both GPUs. For the second Linear layer, we split the 4×2 weight tensor along the columns into two 4×1 tensors. Thus, GPU 0 gets a 4×1 tensor with the first column of the original tensor and GPU 1 gets a 4×1 tensor with the second column.

The 1×4 input tensor is scattered as is to both GPUs. Then, each GPU performs a matrix-matrix multiplication between the 1×4 input and the 4×1 weights for the first layer, i.e., $(1 \times 4) @ (4 \times 1) = (1 \times 1)$. After that, we aggregate the partial results from both GPUs, using a concatenation to obtain a 1×2 tensor (Figure 4.23).

Since we shard the weights into N devices (two in our current example), we reduce the required memory by $1/N$. Thus, the more devices we use, the bigger model we can train.

3. Communication among NVIDIA GPUs

The core idea in inter-process communication lies shared memory, which simply means that multiple execution contexts, threads or processes, can access the same memory space without needing to copy data between contexts. On CPUs, this typically supported by operating system shared memory, such as, `mmap` or POSIX shared memory, enabling processes to exchange data efficiently. GPUs also have their version of on-chip shared memory for threads within a block communicate. The most relevant principle here is avoiding redundant data copies by letting different agents access the same underlying storage.

Building on that idea, CUDA Inter-Process Communication (IPC) extends the concept of shared memory for GPUs. Normally, GPU memory is private to the process that allocates it, but CUDA IPC allows one process to allocate memory on a GPU and then expose a handle that other processes can use to access that same memory region. This avoids expensive host-device transfers and makes it possible for multi-process applications, such

as in a training setup where we use a process per GPU, to cooperate efficiently. In that sense, CUDA IPC is communication layer for sharing GPU memory across processes.

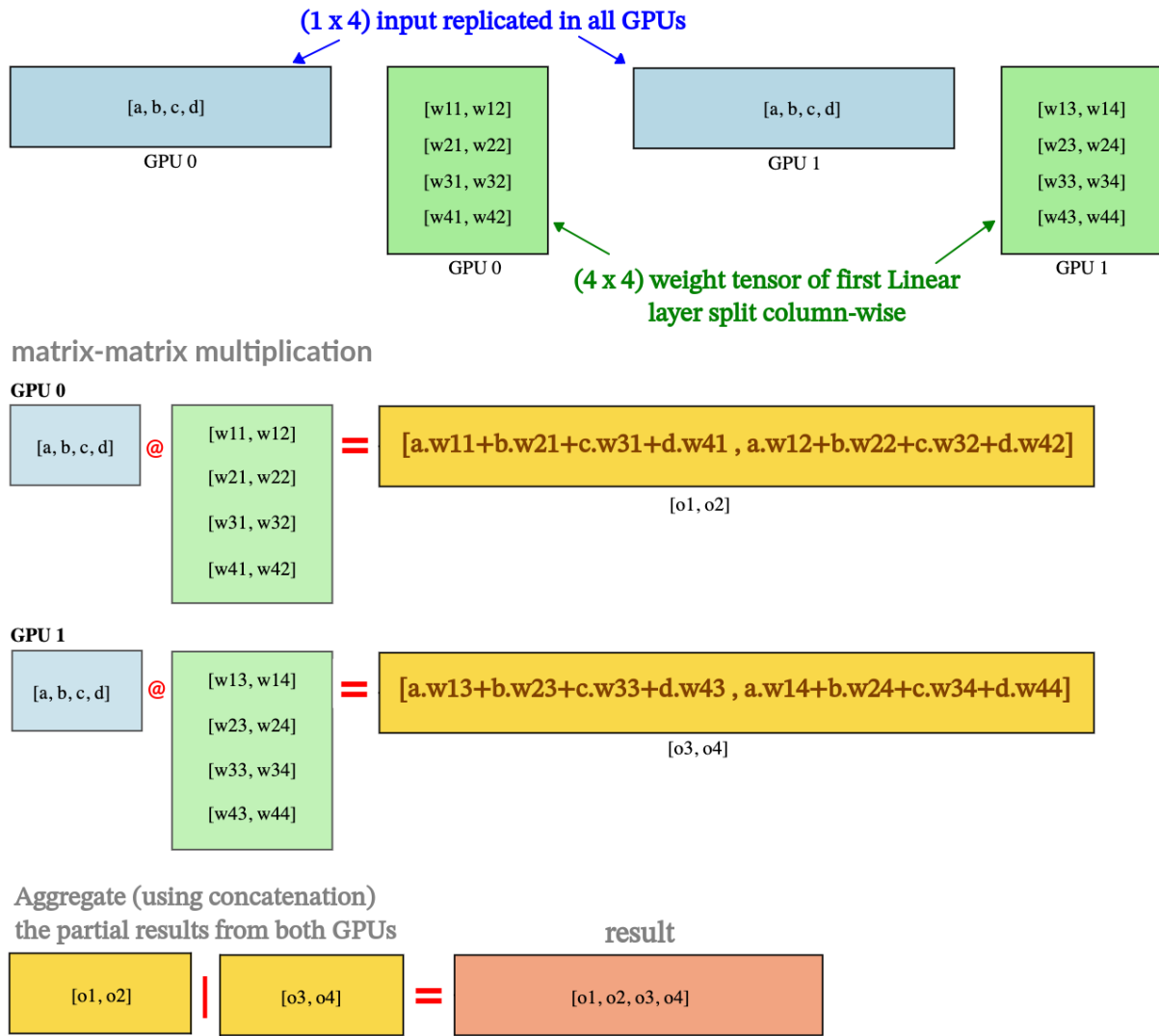
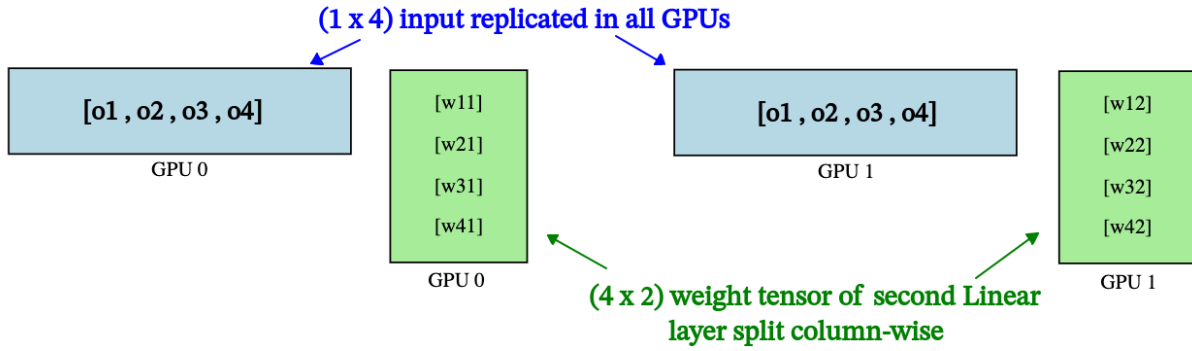


Figure 4.22: Illustration of column-wise tensor parallelism applied to the first Linear layer.

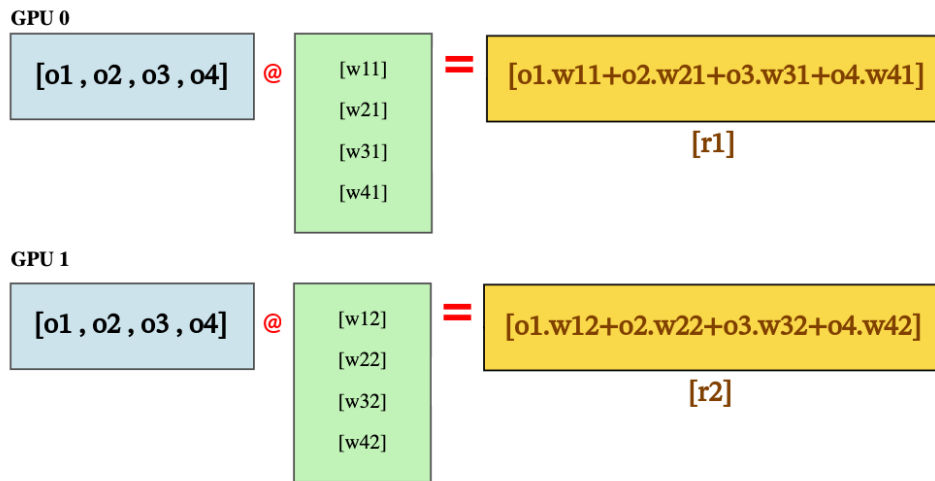
Alongside this, but orthogonal in purpose, is peer-to-peer (P2P) communication. While CUDA IPC focuses on sharing memory across processes, P2P focuses on communication across devices. When enabled, P2P allows one GPU to directly read from or write to another GPU's memory over interconnects like PCIe or NVLink. Without P2P, data transfers between GPUs must be staged through host memory, which is much slower. With P2P, transfers happen directly between GPUs, significantly improving bandwidth and latency.

At the top of this stack sits the NVIDIA Collective Communications Library (NCCL), which provides high-level communication primitives such as AllReduce, broadcast, and AllGather. These are exactly the patterns needed in distributed training, for example, to synchronize the gradients across GPUs. NCCL itself does not reinvent the low-level mechanisms. Instead, it orchestrates and optimizes them. Internally, it uses CUDA IPC to share buffers across processes when needed, uses P2P to move data directly between GPUs on the same node, and resorts to networking technologies when scaling across multiple machines. It also leverages shared memory for coordination and control paths. Crucially, NCCL automatically selects the fastest available transport depending on the hardware topology, so developers do not have to manually manage these details.

Therefore, we can view these technologies as layers of abstraction: shared memory provides the conceptual foundation to avoid unnecessary copies, CUDA IPC applies that idea to GPU memory across processes, P2P enables efficient direct communication between GPUs, and NCCL stays above all of them, combining and optimizing these mechanisms to implement scalable, high-performance collective communication (Figure 4.24).



matrix-matrix multiplication



Aggregate (using concatenation)

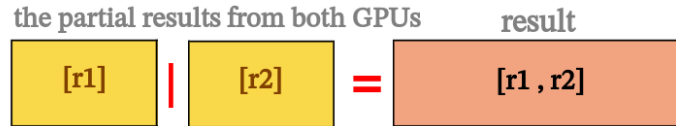


Figure 4.23: Illustration of column-wise tensor parallelism applied to the second Linear layer.

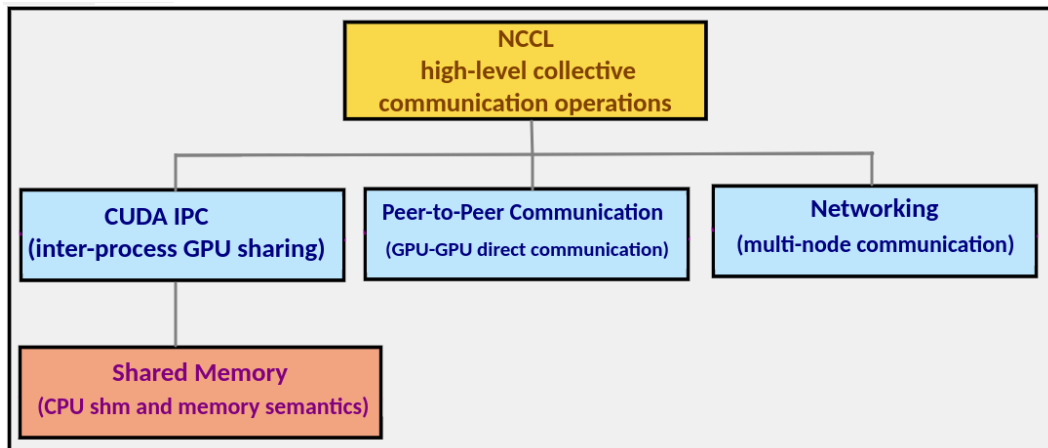


Figure 4.24: Relationship diagram of the NVIDIA communication concepts.

4. Manual Implementation of Row-Wise Tensor Parallelism

When training our model with PyTorch distributed and two GPUs, PyTorch spawns two processes, process 0 runs on GPU 0 and process 1 runs on GPU 1. These processes exchange gradients and weights using the NVIDIA NCCL backend. For managing communication correctly, there are three important variables: RANK is the GPU rank, WORLD_SIZE is the number of processes (or GPUs) that will be involved in communication, and LOCAL_WORLD_SIZE is the number of GPUs per node, relevant when we use multiple nodes. If we are using a single node, WORLD_SIZE and LOCAL_WORLD_SIZE will be the same value.

In our simple scenario, GPU 0 is RANK 0 and GPU 1 is RANK 1, and WORLD_SIZE is 2. Both RANK and WORLD_SIZE are accessible through environment variables automatically set by PyTorch distributed. Assuming that RANK 0 opens port 29950 and RANK 1 opens port 29951, the communication setup is as illustrated in Figure 4.25. Sockets, or open ports, often check heartbeats and communicate simple strings among multi-processes, while NCCL is designed specifically for NVIDIA peer-to-peer GPU communication.

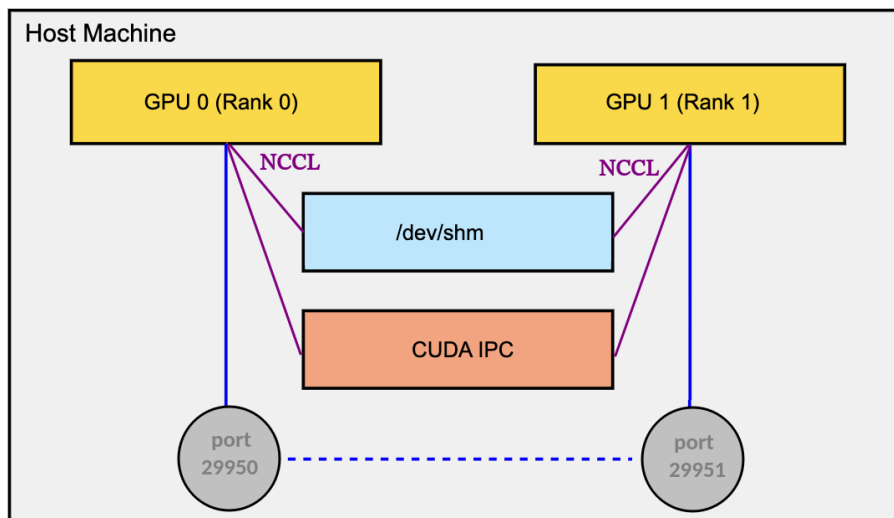


Figure 4.25: Communication setup for distributed training with two GPUs and a single node.

Simple scatter-gather example

Before we apply tensor parallelism to our simple model, let us explore the scatter and gather collective operations. For that purpose, we will use the Python code included in Listing 4.17.

If we save the code included in Listing 4.17 on the `TP_scatter_gather.py`, to run it with two GPUs we just have to execute the next command.

```
torchrn -nproc-per-node=2 TP_scatter_gather.py
```

The output of the execution is as follows.

```
Local rank 0 tensor received from scatter: tensor([1., 1.], device='cuda:0')
Local rank 1 tensor received from scatter: tensor([5., 5.], device='cuda:1')
Gathered list of tensors:
  [tensor([2., 2.], device='cuda:0'), tensor([6., 6.], device='cuda:0')]
```

As discussed in Section 2.4, `dist.scatter` shards a list with N tensors into N GPUs, where N is 2 in our example and the list is `[[1, 1], [5, 5]]`, and each GPU will receive a different tensor from the list, GPU 0 receives the tensor `[1, 1]` and GPU 1 receives the tensor `[5, 5]`.

An tensor for storing the output of the scatter operation must be initialized in each GPU, which in our example was done with `output_tensor = torch.zeros(tensor_size, device=device)`. This tensor will be filled with data received during the execution of `dist.scatter`.

```

1 import torch
2 import torch.nn as nn
3 import torch.distributed as dist
4 import os
5
6 def main():
7     world_size = torch.cuda.device_count()
8     local_rank = int(os.environ["LOCAL_RANK"])
9     device = f'cuda:{local_rank}'
10    dist.init_process_group(backend='nccl')
11
12    tensor_size = 2
13
14    output_tensor = torch.zeros(tensor_size, device=device)
15
16    if dist.get_rank() == 0:
17        t_ones = torch.ones(tensor_size, device=device) # [1,1]
18        t_fives = torch.ones(tensor_size, device=device) * 5 # [5,5]
19        scatter_list = [t_ones, t_fives]
20    else:
21        scatter_list = None
22
23    # scatter_list - list of tensors to be scattered
24    # output_tensor - where the received tensor will be stored in each node
25    dist.scatter(output_tensor, scatter_list, src=0)
26
27    print(f'Local rank {local_rank} tensor received from scatter: {output_tensor}')
28
29    g_in_tensor = output_tensor + 1
30
31    if dist.get_rank() == 0:
32        t_ones1 = torch.ones(tensor_size, device=device)
33        t_ones2 = torch.ones(tensor_size, device=device)
34        gather_list = [t_ones1, t_ones2]
35    else:
36        gather_list = None
37
38    # g_in_tensor - tensor to gather from all nodes
39    # gather_list - rank 0 list where the gathered tensors will be stored
40    dist.gather(g_in_tensor, gather_list, dst=0)
41
42    if dist.get_rank() == 0:
43        print(f'Gathered list of tensors: {gather_list}')
44
45 if __name__ == "__main__":
46    main()

```

Listing 4.17: Experiment the scatter and gather collective operations.

After the scatter operation, each GPU adds one to the the received tensor data, resulting in a tensor `g_in_tensor=[2,2]` in GPU 0 and `g_in_tensor=[6,6]` in GPU 1. If we are in RANK 0 GPU, we create two tensors, one to receive the data that will be gathered from GPU 0 and the other for the data that will be received from GPU 1. These tensors are included in `gather_list`.

Next, we execute a gather operation, which will collect `g_in_tensor` from all ranks and put them in the rank 0 `gather_list`. Finally, GPU 0 prints `gather_list`, containing `[2, 2]` received from GPU 0 and `[6, 6]` received from GPU 1. The flow of execution, including the scatter and gather operations, for the execution of this example is shown in Figure 4.26.

We now focus on adding tensor parallelism to Linear layers and train a simple model. In Listing 4.18, it is defined a Linear layer with row-wise tensor parallelism, following the principles described before. The weight tensor of the Linear layer that will be allocated to each GPU has a weight tensor of size $in_features/\#gpus \times out_features$. In the `forward` method, the input tensor is split column-wise and the parcels are scattered to the GPUs, and each GPU receives a parcel of size $batch_size \times in_features/\#gpus$. This input parcel is passed

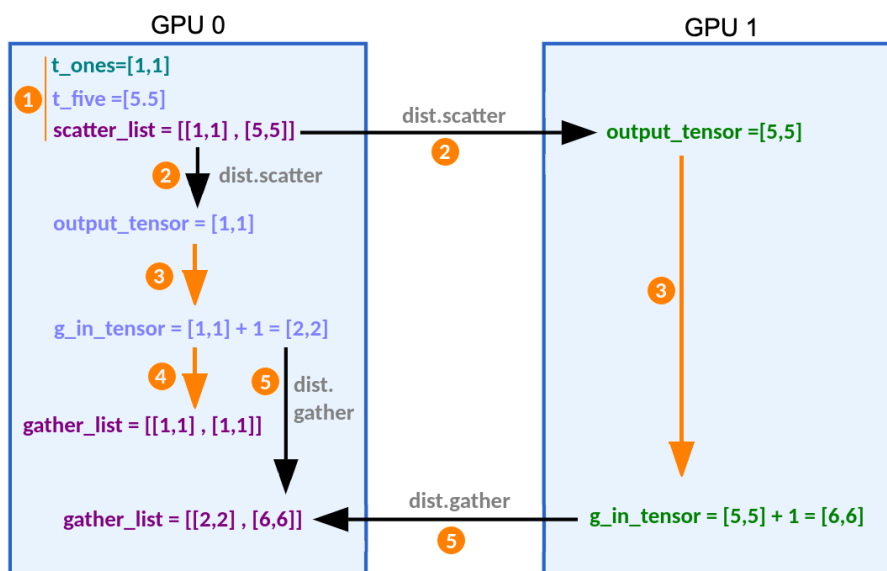


Figure 4.26: The main events associated with the execution of the code included in Listing 4.17.

through the Linear layer parcel. In the end, we run an AllReduce operation to gather the output from all Linear layer parcels, sharded among the GPUs. The outputs from all GPUs are reduced using the sum. After that, all GPUs will have the same output for the Linear layer.

Listing 4.18 also defines a simple model with a Linear layer, a ReLU activation function, a second Linear, and a second ReLU. The first Linear layer has a 100×50 weight tensor and the second one has a 50×20 weight tensor. Listing 4.19 contains the main function used to train this model. It instantiates the model and an optimizer. The training loop simply generates a 32×100 random tensor, pass it through the model, perform a backward pass, and update the model weights.

If we save the code included in Listings 4.18 and 4.19 in the file `TP_linear2.py`, we can train the model using the next command. Execution time for 10k iterations, with two GPUs, was 38 s.

```
torchrn -nproc-per-node=2 TP_linear2.py
```

The model output has a size of 32×20 . In the first Linear layer, we multiply the 32×100 input by the 100×50 tensor weight and obtain an output with shape 32×50 . In the second Linear layer, we multiply the 32×50 output from first layer by the 50×20 weight tensor, and we obtain a final output with shape 32×20 .

5. Tensor Parallelism using `torch.distributed.tensor.parallel`

In PyTorch, tensor parallelism (TP) is built on top of the DistributedTensor (DTensor) and provides different parallelism styles: Colwise, Rowwise, and Sequence Parallelism. The entry point to parallelize a module using tensor parallelism is `parallelize_module()`, which applies tensor parallelism based on a user-specified plan.

```
torch.distributed.tensor.parallel.parallelize_module(
    module,
    device_mesh,
    parallelize_plan,
    *,
    src_data_rank
)
```

The `parallelize_plan` contains a `ParallelStyle` that indicates how we want the module to be parallelized. `device_mesh` is an object that describes the mesh topology of devices for distributing training with DTensor. `parallelize_plan` can be either (i) a `ParallelStyle` object containing information on how

we prepare input/output for tensor parallelism or (ii) it can be a dictionary of module fully qualified names. `src_data_rank` is the rank of the source data when scattering/broadcasting shards of data/weights to other ranks.

```

1 import os
2 import time
3 import torch
4 import torch.nn as nn
5 import torch.nn.functional as F
6 import torch.distributed as dist
7
8 # Linear layer with row-wise tensor parallelism.
9 class Linear(nn.Module):
10     def __init__(self, in_features, out_features):
11         super().__init__()
12         self.in_features = in_features
13         self.out_features = out_features
14
15         self.rank = dist.get_rank()
16         self.world_size = dist.get_world_size()
17         self.device = f'cuda:{self.rank}'
18
19         # row-wise tensor parallelism applied to Linear layer
20         self.local_in_features = in_features // self.world_size
21         self.local_out_features = out_features
22
23         self.linear = nn.Linear(self.local_in_features, self.local_out_features)
24
25     def forward(self, x, batch_size):
26
27         # input shape = [batch_size, local_in_features]
28         local_input = torch.zeros(batch_size, self.local_in_features, device=self.device)
29
30         # Scatter a parcel of input with size [batch_size, local_in_features/world_size]
31         # to every GPU (input is split column-wise)
32         dist.scatter(
33             local_input,
34             list(x.chunk(self.world_size, dim=1)) if self.rank == 0 else None,
35             src=0
36         )
37
38         # Pass input through the Linear layer
39         output1 = self.linear(local_input)
40
41         # AllReduce partial Linear layers' outputs from all GPUs using sum.
42         # All GPUs will have the same output for the Linear layer.
43         dist.all_reduce(output1, op=dist.ReduceOp.SUM)
44
45         return output1
46
47 # Simple model with Linear, ReLU, Linear, ReLU layers.
48 class Model(nn.Module):
49     def __init__(self, inf, hidf, outf, device) -> None:
50         super().__init__()
51         self.linear1 = Linear(inf, hidf).to(device)
52         self.linear2 = Linear(hidf, outf).to(device)
53
54     def forward(self, x, bs):
55         x = F.relu(self.linear1(x, bs))
56         return F.relu(self.linear2(x, bs))

```

Listing 4.18: Model with two pairs Linear-ReLU, where it is applied row-wise tensor parallelism to the Linear layers.

```

1 def main():
2     world_size = torch.cuda.device_count()
3     local_rank = int(os.environ["LOCAL_RANK"])
4     device = f'cuda:{local_rank}'
5     dist.init_process_group(backend='nccl')
6
7     batch_size = 32
8
9     model = Model(inf=100, hidf=50, outf=20, device=device)
10
11     # Create an optimizer for the TP model
12     lr = 0.1
13     optimizer = torch.optim.AdamW(model.parameters(), lr=lr, foreach=True)
14
15     if dist.get_rank() == 0:
16         ts = time.time()
17
18     for id in range(10000):
19
20         if dist.get_rank() == 0:
21             input_tensor = torch.randn(batch_size, 100, device=device)
22         else:
23             input_tensor = None
24
25         output = model(input_tensor, batch_size)
26         output.sum().backward()
27         optimizer.step()
28
29         if dist.get_rank() == 0 and id % 100 == 0:
30             print(f"Iteration {id} completed.")
31
32     if dist.get_rank() == 0:
33         te = time.time()
34         print(f"Execution time for 10k iterations: {te-ts} s")
35         print(f'{output}\n {output.shape}')
36
37 if __name__ == "__main__":
38     main()

```

Listing 4.19: Main function to train the model that uses row-wise tensor parallelism.

`DeviceMesh` is a PyTorch higher level abstraction that manages a group of processes, `ProcessGroup`. It allows users to effortlessly create inter-node and intra-node process groups without worrying about how to set up ranks correctly for different sub process groups. Users can also easily manage the underlying `process_groups/devices` for multi-dimensional parallelism via `DeviceMesh`. `DeviceMesh` is useful when working with multi-dimensional parallelism, such as 3D parallelism, where parallelism composability is required. For example, when our parallel solution requires both communication across hosts and within each host. Without `DeviceMesh`, users would need to manually set up NCCL communicators, cuda devices on each process before applying any parallelism, which could be quite complicated.

Tensor parallelism supports the following parallel styles.

- `ColwiseParallel` partitions a module column-wise, and the allowed module types are `Linear` and `Embedding`,
- `RowwiseParallel` partitions a module row-wise, and the supported types of modules are also `Linear` and `Embedding`,
- `SequenceParallel` replicates the module parameters and runs the sharded computation with input sharded on the sequence dimension; currently, it supports `LayerNorm`, `Dropout`, and `RMSNorm` layers,
- `PrepareModuleInput` configures the module's inputs to convert the input tensors to `DTensors` at runtime according to the specified input layouts, and performs layout redistribution according to the desired input layouts,

- `PrepareModuleOutput` configures the module's outputs to convert the output tensors to DTensors at runtime according to the specified output layouts, and performs layout redistribution according to the desired output layouts,
- `PrepareModuleInputOutput` configures the module's inputs and outputs to convert the input and output tensors, of the module, to DTensors at runtime according to input and output layouts, and performs layout redistribution according to the desired input and output layouts.

To demonstrate how to use `torch.distributed.tensor.parallel` for applying tensor parallelism, we consider the same model with two pairs of a `Linear` layer and a `ReLU` activation, as in the manual parallelization. Therefore, the training loop is the same as presented in Listing 4.19 and the model architecture is reproduced in Listing 4.20. This listing also contains the specific aspects related to the PyTorch API, namely (i) defining the mesh topology of devices for distributing training, and (ii) creating a custom parallelization plan for the model, specifying that we want to apply column-wise parallelism to the first `Linear` layer and row-wise parallelism to the second `Linear` layer. The execution time for training the model for 10k iterations was 108 s.

```

1 class SimpleModel(nn.Module):
2     # Simple model with two Linear layers and two ReLU activations.
3     def __init__(self):
4         super(SimpleModel, self).__init__()
5         self.linear1 = nn.Linear(100, 50)
6         self.relu1 = nn.ReLU()
7         self.linear2 = nn.Linear(50, 20)
8         self.relu2 = nn.ReLU()
9
10    def forward(self, x):
11        return self.relu2(self.linear2(self.relu1(self.linear1(x))))
12
13    # Define the mesh topology of devices for distributing training (assigns IDs to GPUs).
14    device_mesh = init_device_mesh(
15        device_type="cuda",
16        mesh_shape=(world_size,)
17    )
18
19    # Create the model and move it to GPU.
20    model = SimpleModel().to("cuda")
21
22    # Custom parallelization plan for the model: column-wise parallel for
23    # the first linear layer and row-wise parallel for the second linear layer.
24    tp_model = parallelize_module(
25        module = model,
26        device_mesh = device_mesh,
27        parallelize_plan = {
28            "linear1": ColwiseParallel(),
29            "linear2": RowwiseParallel(),
30        },
31    )

```

Listing 4.20: Model with two pairs Linear-ReLU, where it is used `torch.distributed.tensor.parallel` to apply tensor parallelism to the `Linear` layers.

Instead of opting for row-wise parallelism in both `Linear` layers, as we did in the manual parallelization, the PyTorch recommendation was followed, and column-wise parallelism was applied to the first `Linear` layer and row-wise parallelism to the second. The goal was to eliminate the need for collective operations in the middle of the two layers. Although the manual implementation is a bit more laborious, the truth is that even including a scatter operation at the beginning of the computation of each layer and an `AllReduce` operation at the end, the training time is only 35% of the training time with the PyTorch API, i.e, 38s vs 108s.

4.3 Combining Multiple Types of Parallelism

4.3.1 Data and Pipeline Parallelisms

Pipeline parallelism improves both the memory and compute efficiency of during training of large models by partitioning the layers of a model into stages that can be processed in parallel. DeepSpeed supports hybrid data and pipeline parallelisms, which be further combined with model parallelism. An illustration of this tri-dimensional parallelism is shown in Figure 4.27 [196]. Results show that 3D parallelism allows for training models with more than a trillion parameters.

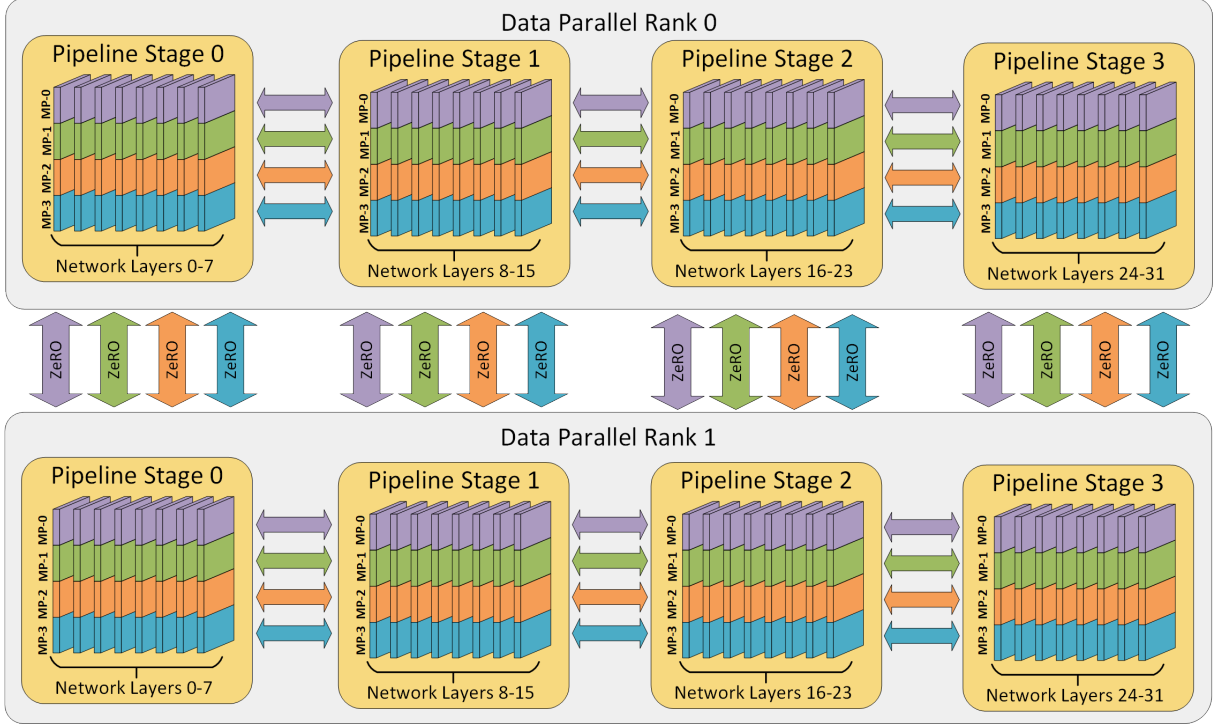


Figure 4.27: 3D parallelism in DeepSpeed [196].

DeepSpeed uses gradient accumulation to extract pipeline parallelism. Each batch of training data is split into micro-batches that can be processed in parallel by the different pipeline stages. When a stage completes its forward pass over a micro-batch, the output activations are sent to the next stage of the pipeline. Similarly, when a stage completes its backward pass over a micro-batch, the gradients with respect to the input activations are sent to the previous stage of the pipeline. Each backward pass accumulates gradients locally. Next, all data parallel groups perform a reduction of the gradients in parallel. Lastly, the optimizer updates the model weights using the reduced gradients.

Figure 4.28 illustrates how DeepSpeed trains over a minibatch with eight micro-batches using hybrid two-way data parallelism and two-stage pipeline parallelism. GPUs 0 and 2 are arranged in a pipeline and will alternate forward (F_i) and backward (B_i) passes. They will then run an AllReduce (AR) operation over the gradients with their data parallel paired GPU, GPU 1 and 3, respectively. GPUs 1 and 3 are also arranged in a two-stage pipeline. Finally, the two pipelines stages, $(0 \parallel 1) \mid (2 \parallel 3)$, update the model weights.

4.3.2 Data, Pipeline and Model Parallelisms

Data, model and pipeline parallelisms each perform a specific role in improving memory and compute efficiency. To achieve high memory efficiency, the layers of the model are split into multiple pipeline stages (0:3 in Figure 4.27), and the layers of each stage are further divided using model parallel parcels ($MP_0:MP_3$ in Figure 4.27). This 2D combination simultaneously reduces the memory consumed by the model, optimizer state, and activations. However, we cannot partition the model indefinitely without incurring in a communication overhead that limits the computation efficiency [197].

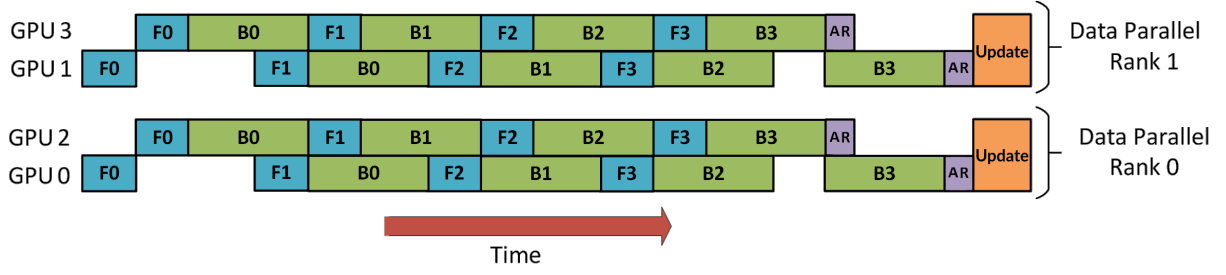


Figure 4.28: Training over a minibatch with eight micro-batches using hybrid two-way data parallelism and two-stage pipeline parallelism [196].

To allow the number of nodes to scale beyond model and pipeline parallelisms without sacrificing computation efficiency, we can resort to data parallelism (DP), for example, ZeRO data parallelism. ZeRO-DP not only reduces the required memory, by also sharding the optimizer state, but also allows for scaling to arbitrarily large number of GPUs with minimal communication overhead by exploiting topology aware mapping of processes to nodes.

In Figure 4.29, each dimension of the tri-dimensional parallelism is carefully mapped to the nodes to minimize the required memory by exploiting two architectural properties: minimization of the intra- and inter-node communication bandwidth and introducing parallelism in communication to emulate a larger overall bandwidth.

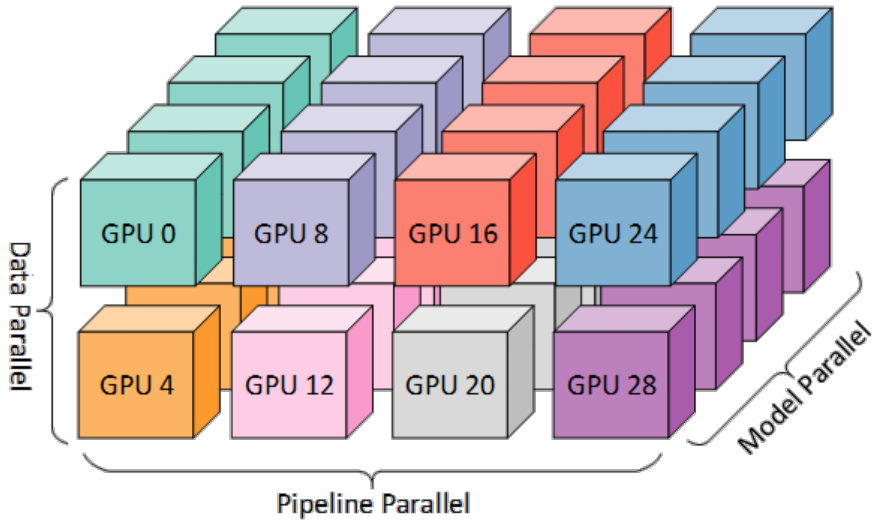


Figure 4.29: Mapping workers to GPUs on a system with eight nodes, each with four GPUs. GPU of the same color are on the same node [197].

Model parallelism has the largest communication cost among data, model and pipeline parallelisms, and as so we must place the model parallel parcels in the same node to utilize the larger intra-node bandwidth. Here we can use Megatron-LM for implementing model parallelism by slicing the tensors. Data parallel parcels are placed within a node when model parallelism does not fit all the processes in a node. Otherwise, they are placed across nodes. Pipeline parallelism has the lowest communication cost, and as so we can schedule pipeline stages across nodes without incurring in high communication bandwidth increase.

The size of the gradients communicated by each data parallel parcel decreases linearly by using both pipeline and model parallelisms, and thus the total communication cost is decreased in relation to pure data parallelism. Furthermore, each data parallel parcel performs its communication independently and in parallel among a subset of localized processes. As a result, the effective bandwidth associated with data parallel communication is amplified by using a combination of reduced communication cost and increased locality and parallelism.

4.3.3 Data, Pipeline and Tensor Parallelisms, and ZeRO Optimization

One of the main features of DeepSpeed is ZeRO, a memory optimization technique often combined with data parallelism. The ZeRO technique was discussed in Section 3.7. Normally, ZeRO is a standalone feature that does not require pipeline or tensor parallelism, but it can be combined with pipeline and tensor parallelisms. When ZeRO data parallelism is combined with pipeline parallelism, and optionally with tensor parallelism, it typically enables only optimizer sharding (ZeRO stage 1).

While it is theoretically possible to combine ZeRO stage 2, where we do optimizer state and gradient sharding, with pipeline parallelism, it will have a negative performance impact. This combination would require an additional ReduceScatter collective operation per microbatch to aggregate the gradients before sharding them, which adds a potentially significant communication overhead. As usual in pipeline parallelism, small micro-batches are used and the focus is on trying to balance arithmetic intensity (microbatch size) with minimizing the pipeline bubble (number of microbatches). Therefore, the ReduceScatter communication cost would be highly penalizing.

In addition, there are already fewer layers than normal due to also using pipeline parallelism, and as so the memory savings would not be high. Pipeline parallelism already reduces the gradient size by $1/\text{degree_parallelism}$ and, thus, the gradient sharding savings on top of that are less significant than on pure data parallelism. ZeRO stage 3 is not a good choice either, for the same reason, it requires more inter-node communication. When we use ZeRO, the other benefit comes from ZeRO-offload, where the optimizer state can be offloaded to CPU.

4.3.4 Megatron-LM

The authors of [145] combined pipeline, tensor and data parallelism with a new pipeline scheduling strategy and named their approach PTD-P. Instead of only positioning a continuous set of layers (a model chunk) on a device, each worker node can be assigned with multiple chunks of smaller continuous subsets of layers, for example, device 1 has the layers 1, 2, 9, 10, and device 2 has the layers 3, 4, 11, 12. Thus, each device has two model chunks. The number of microbatches in one batch should be divisible by the number of workers (m). If there are v model chunks per worker node, the pipeline bubble time can be reduced by a multiplier of v compared to a GPipe scheduling (Figure 4.30).

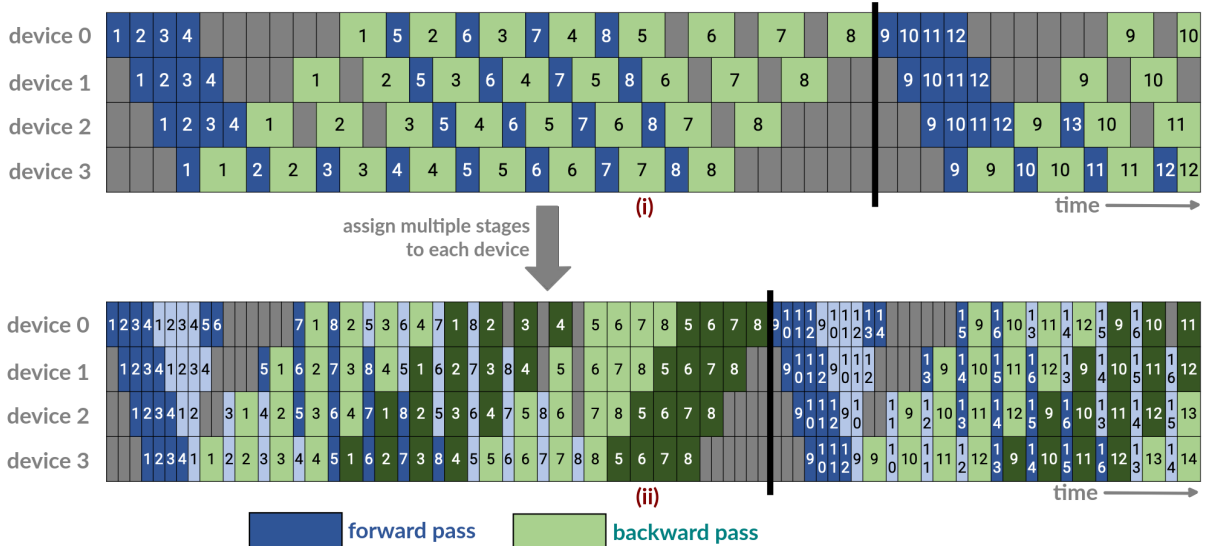


Figure 4.30: (i) The default 1F1B pipeline schedule, as in PipeDream-flush, and (ii) the interleaved 1F1B pipeline schedule. The first model chunks assigned to a node are in dark blue/green and the second chunks are in light blue/green [145].

4.3.5 FlexFlow

Data parallelism works well for compute-heavy layers but struggles when parameter synchronization becomes expensive. Model parallelism avoids that synchronization but introduces a communication overhead between

layers and limits parallelism within each operation. In practice, real systems often rely on a combination of these approaches.

To move beyond these limitations, the authors of [97] introduced a conceptual framework for parallelization, called the SOAP search space. Instead of restricting parallelism to either the data or the model dimension, SOAP considers four distinct but composable axes: split along the sample dimension, split across operations, divide tensor data, and split the model weights. This generalization is important because it reveals that most existing approaches actually explore only small subspaces of a much larger design space. Prior systems typically operate along one or two of these dimensions, whereas FlexFlow is designed to search across all of them simultaneously, enabling hybrid strategies that combine multiple forms of parallelism in a fine-grained way.

Figure 4.31 shows four configurations for a convolution, ranging from pure data parallelism to richer hybrid strategies that also partition weights and tensors. Figure 4.32 illustrates, through a matrix multiplication example, how a configuration decomposes an operation into multiple independent tasks mapped to different devices. What emerges from these examples is the finding that parallelism is not a binary choice but rather a combinatorial problem where each operation can be decomposed in multiple ways, and a full training strategy results from choosing a configuration for every operation in the network.

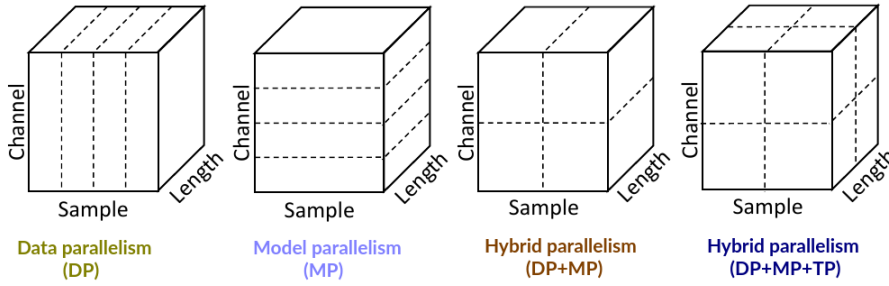


Figure 4.31: Four parallelism configurations for a convolution. Dashed lines show partitioning the tensor [97].

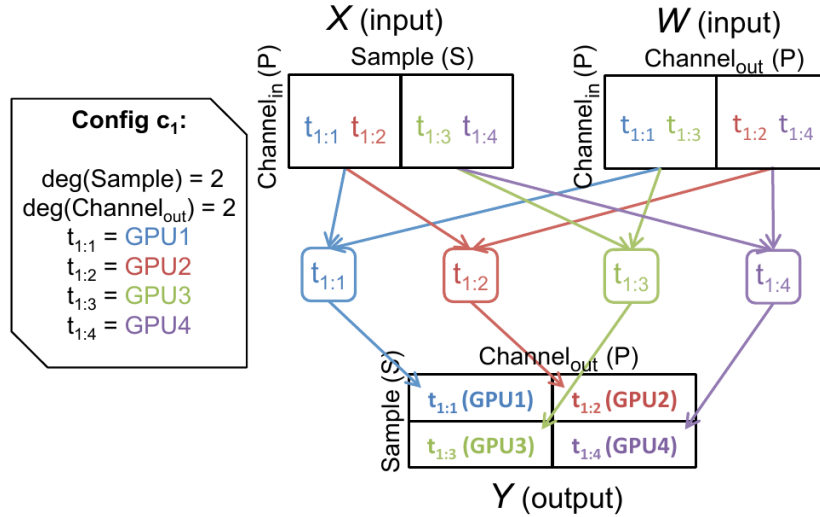


Figure 4.32: An example of parallelism configuration for a matrix multiplication operation [97].

The flexibility of FlexFlow comes at a cost: the number of possible strategies grows exponentially with the size of the model. Thus, the core challenge is not just defining the richer design space, but finding good strategies within it efficiently. This is where the FlexFlow system comes in. As outlined in Figure 4.33, FlexFlow takes as input a model represented as an operator graph and a description of the available computation resources, and automatically searches for an efficient parallelization strategy. It consists of three main components: an execution optimizer that explores candidate strategies, an execution simulator that predicts their performance, and a runtime system that executes the best parallelization found.

The execution simulator is the main technical contribution. Instead of running each candidate strategy on real hardware, which would be prohibitively slow, FlexFlow predicts the execution time by simulating how

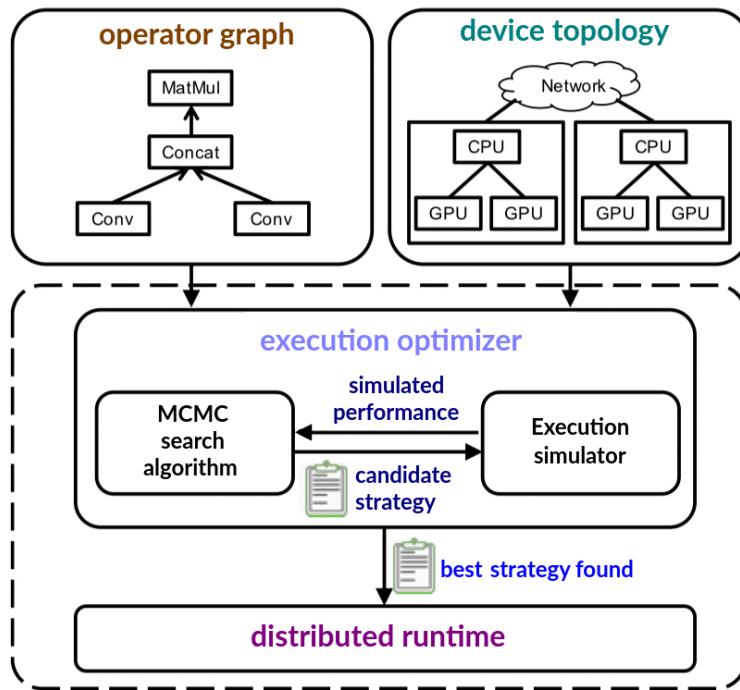


Figure 4.33: FlexFlow overview [97].

tasks would execute and communicate. The authors rely on the observation that the runtime of neural network operations is largely determined by tensor sizes and hardware characteristics, and is relatively insensitive to the actual data values. An important refinement is the delta simulation algorithm. Because the search process explores strategies that differ only slightly from one another, it is wasteful to recompute the entire simulation each time. Instead, FlexFlow incrementally updates only the parts of the task graph affected by a change. This optimization dramatically speeds up evaluation, making the simulator up to three orders of magnitude faster than actual execution, while still producing sufficiently accurate predictions to guide the search.

On top of this simulator, FlexFlow builds a stochastic search procedure based on Markov Chain Monte Carlo. Starting from an initial strategy, the system repeatedly proposes small modifications, such as changing how a single operation is partitioned, and uses the simulated runtime to decide whether to accept the new configuration or not. This process balances exploitation of better strategies with exploration of the search space, enabling the system to escape local optima and discover more efficient configurations.

Experiments demonstrated that FlexFlow consistently finds better strategies than both traditional methods and prior automated systems. Across a range of benchmarks, it achieved up to 3.8 speedups, while also reducing communication overhead and improving scalability.

4.4 Conclusions

In this chapter, we explored model parallelism, which enables training models that exceed the memory capacity of a single device by partitioning layers across multiple GPUs. Although conceptually simple and memory-efficient, naive model parallelism suffers from poor hardware utilization, communication overhead, and imbalanced workloads, which limit its practical efficiency. To overcome these limitations, we introduced pipeline parallelism, which combines model parallelism with data parallelism and improves device utilization by splitting minibatches into microbatches and executing them in a pipelined fashion across multiple devices. Pipeline parallelism requires GPU-to-GPU communication between the model partitions during the forward pass, to send activations, and the backward pass, to send gradients.

Approaches such as GPipe and PipeDream demonstrate how careful scheduling can significantly increase throughput while maintaining correctness. GPipe emphasizes simplicity and gradient accumulation, whereas PipeDream reduces memory footprint and improves execution overlap through asynchronous execution

strategies and weight management techniques. Despite these advancements, pipeline parallelism introduces new challenges, including pipeline bubbles, memory requirements for storing activations, and the need for balanced partitioning.

Next, we examined tensor parallelism, which distributes computations within individual layers across devices. Unlike pipeline parallelism, which split models vertically, tensor parallelism partitions operations horizontally, allowing for fine-grained parallelism and improved scalability. By carefully designing partitioning schemes that minimize synchronization points, tensor parallelism can achieve high efficiency, particularly in large transformer-based architectures.

A key takeaway from this chapter is that no single parallelism strategy is sufficient for training modern large-scale models. Instead, the most effective systems rely on hybrid approaches, combining data, pipeline, and tensor parallelism to balance memory usage, communication cost, and computational efficiency. In this context, the combination strategies were examined in detail.

The integration of data and pipeline parallelisms enables both improved resource utilization and scalability, by replicating pipeline stages across data-parallel groups and overlapping gradient synchronization with computation. Extending this idea, the joint use of data, pipeline, and model parallelisms further reduces memory consumption by partitioning both layers and their internal parameters, while careful placement of each parallel dimension minimizes communication overhead and exploits hardware locality.

The subsection on data, pipeline, and tensor parallelisms with ZeRO optimization highlighted the importance of memory-efficient training, showing how optimizer and gradient sharding can complement other parallelism techniques. However, it also emphasized that aggressive sharding strategies must be balanced against communication costs, especially in pipeline settings where microbatching amplifies communication frequency.

More advanced systems were also discussed. Megatron-LM demonstrated how combining multiple forms of parallelism with optimized scheduling, such as interleaving model chunks within each device, can significantly reduce pipeline idle time and improve overall throughput. On the other hand, FlexFlow introduced a generalized and automated perspective, framing parallelization as a search problem over multiple dimensions. By exploring combinations of parallelism strategies using simulation-driven optimization, it shows that efficient configurations can be discovered automatically, often surpassing manually designed solutions.

Thus, parallelism techniques are fundamental for deep learning, enabling the training of increasingly large and complex models. As hardware architectures evolve and model sizes continue to grow, the design of efficient and well-balanced parallel training strategies, particularly those combining multiple forms of parallelism, will remain a central challenge and a key area of innovation.

Chapter 5

Mixture-of-Experts

Contents

5.1	Introduction to mixture of experts	124
5.2	Taxonomy of mixture of experts	127
5.3	Algorithm design of mixture of experts	127
5.4	Design of mixture of experts	142
5.5	Routing techniques	144
5.6	Load balancing	152
5.7	Training and fine-tuning MoEs	155
5.8	Performance evaluation	160
5.9	Deployment of MoEs	162
5.10	Memory optimization	170
5.11	Communication optimization	174
5.12	Open source MoEs	180
5.13	Other relevant MoEs	200
5.14	Conclusions	204

This chapter presents the Mixture of Experts (MoE) paradigm as a key approach for scaling neural networks, particularly large language models, while maintaining computational efficiency. It begins by introducing the concept of conditional computation, where multiple specialized sub-networks, called experts, are selectively activated through a gating mechanism, allowing models to increase capacity without a proportional increase in computation. The chapter then provides a taxonomy of MoE models, organizing the literature along algorithmic, system, and application dimensions. From an algorithmic perspective, it examines the design of MoE layers within transformer architectures, including dense and sparse formulations, gating mechanisms, and expert design choices, as well as extensions such as weight-efficient experts and hybrid architectures. Subsequent sections focus on training and inference strategies, detailing approaches such as dense-to-sparse transitions, expert merging, and model distillation. The chapter also explores routing techniques in depth, comparing methods like Top-k gating, Switch Transformers, hash-based routing, and Soft MoE, and discusses their trade-offs in terms of efficiency, stability, and specialization. Given the inherent challenges of sparse and dynamic execution, the chapter further examines system-level design, including expert parallelism, communication patterns, and memory management. Particular attention is given to load balancing, capacity management, and adaptive expert placement, which are critical for efficient and stable large-scale training. Finally, it is addressed training dynamics, performance evaluation, and practical challenges, highlighting issues such as instability, overfitting during fine-tuning, communication bottlenecks, and hardware utilization. As a whole, the chapter provides a comprehensive overview of both the theoretical foundations and practical considerations required to design and deploy scalable MoE systems.

5.1 Introduction to Mixture of Experts

Although Mixture of Experts (MoE) are very popular in recent AI research, the concept of conditional computation has its roots in work from the early 1990s. The idea of an MoE has its origins in [92], where the authors proposed a supervised learning framework for an ensemble of several networks. Each of these networks processes a different subset of the training data, and the choice of which network to use is handled by a gating mechanism. Therefore, each network in this system is an expert that specializes in processing some segment of the input data. The authors of [103] proposed a hierarchical MoE that can be trained in a supervised fashion using an expectation-maximization algorithm.

The work [11] explores four possible techniques for estimating the gradient of stochastic neurons and hard-threshold activation functions by using REINFORCE, a straight-through estimator. Conditional computation is one example of a hard-threshold activation function, where the activations of certain neurons are completely eliminated from the computation. The neural network structure explored was comprised of many computational units connected by a distributed network of gates that can be used to eliminate parcels of the computation. By turning off portions of the network, we can greatly reduce the computational cost of a large neural network.

Dynamic Capacity Networks [3] dynamically allocate compute capacity to different parts of the input data by defining low-capacity and high-capacity sub-networks. For most of the data, low-capacity sub-networks are applied. However, a gradient-based attention mechanism can be used to select portions of the input, based upon the network’s sensitivity to this region of the data, to which high-capacity sub-networks will be applied.

Transformer-based large language models (LLMs) became a cornerstone of recent advances in artificial general intelligence (AGI), driving progress across a wide range of domains, including natural language processing, computer vision, and multimodal learning. Their remarkable capabilities stem from large model scales, extensive training data, and significant computational investment. However, as these models continue to grow, efficiently scaling their capacity while controlling computational cost has emerged as a critical challenge. Mixture of Experts has gained renewed attention as a promising solution to this problem. Originally proposed several decades ago, MoE introduces a modular architecture in which multiple specialized sub-networks, or experts, are selectively activated for each input through a gating mechanism. In particular, sparsely-gated MoE enables only a small subset of experts to participate in each forward pass, allowing models to significantly increase weight capacity without a proportional rise in computation. This paradigm aligns well with established scaling laws and has been successfully adopted in recent large-scale LLMs such as Mixtral-8x7B and other industry-grade systems. Figure 5.1 summarizes the release of MoE models between 2017 and 2024, identifying the application area through four colors, and whether they are open source (above the timeline) or proprietary (below the timeline). It can be seen that 51 out of 61 were designed for the area of natural language processing (NLP).

Before we present expert parallelism, we will review the MoE literature. First, we introduce a taxonomy that organizes the field into three key dimensions: algorithm, system, and application. From an algorithmic perspective, we examine the integration of MoE layers into transformer architectures, focusing on gating mechanisms, expert design, and emerging variants such as soft MoE, weight-efficient experts, and hybrid dense-sparse training approaches. From the system perspective, we review advancements in computation, communication, and storage designed to accommodate the sparse and dynamic nature of MoE workloads. Through this structured analysis, we aim to provide a clear understanding of recent progress in MoE-based LLMs [19].

MoE models train faster compared to dense models, as well as have much faster inference compared to a dense model with the same number of weights. However, MoE-style LLMs also consume more VRAM, as all experts must be loaded into memory. Additionally, MoE models are prone to overfitting and notoriously difficult to finetune, which makes them more complicated to use in practical applications compared to dense models.

In transformer-based large language models (LLMs), a mixture-of-experts (MoE) layer consists of a set of N expert networks $\{f_1, \dots, f_N\}$ and a gating network $\mathcal{G}$. The gating network, typically implemented as a linear layer followed by a `softmax` function, determines how input tokens are routed to the experts. In practice, MoE layers commonly replace the feed-forward network (FFN) sublayer within each transformer block, as FFNs

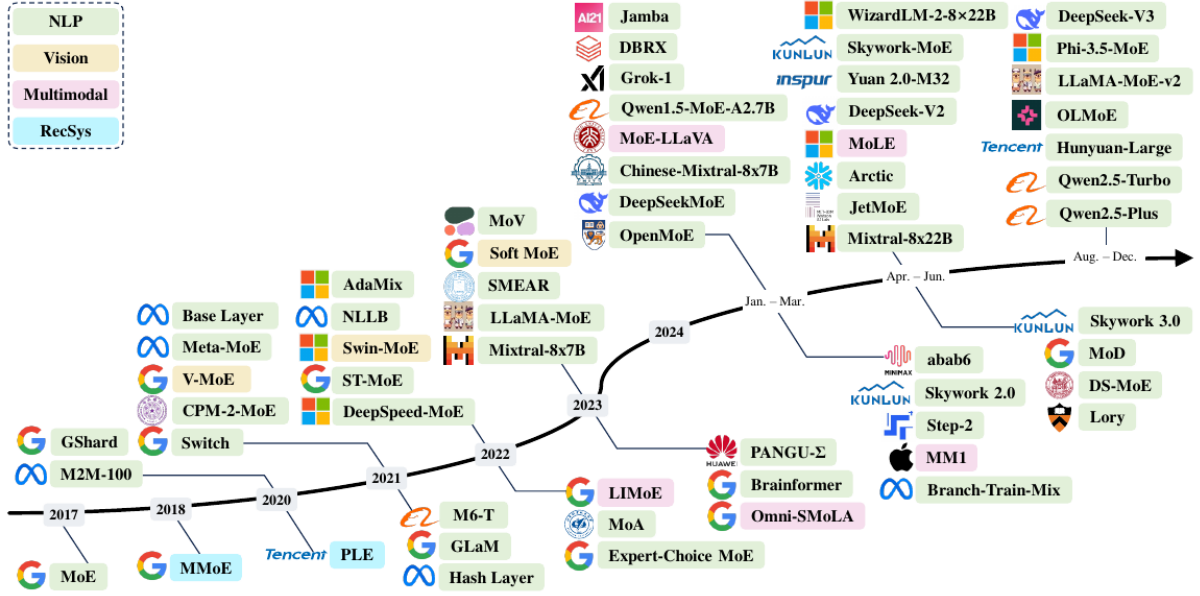


Figure 5.1: A chronological overview of several representative mixture-of-experts (MoE) models in recent years. The timeline is primarily structured according to the release dates of the models. MoE models located above the arrow are open-source, while those below the arrow are proprietary and closed-source. MoE models from various domains are marked with distinct colors: natural language processing (NLP) in green, computer vision in orange, multimodal in pink, and recommender systems (RecSys) in cyan [19].

account for the majority of weights and computational cost in large-scale models.

Formally, each expert f_i , parameterized by $\mathbf{W}_i$, receives the same input $\mathbf{x}$ and produces an output $f_i(\mathbf{x}; \mathbf{W}_i)$. The gating network $\mathcal{G}$, parameterized by Θ , outputs a distribution over experts denoted by $\mathcal{G}(\mathbf{x}; \Theta)$. Based on the design of the gating mechanism, MoE models can be broadly categorized into dense MoE and sparse MoE.

5.1.1 Dense Mixture of Experts

In a dense MoE, all experts are activated for every input, and their outputs are combined using weighted averaging. This approach was widely adopted in early MoE work and has recently been revisited in modern variants.

The output of a dense MoE layer is given by Equation 5.1, where the gating weights $\mathcal{G}(\mathbf{x}; \Theta)_i$ are computed with Equation 5.2, and $g(\mathbf{x}; \Theta)$ denotes the gating logits obtained before applying the softmax.

$$\mathcal{F}_{\text{dense}}^{\text{MoE}}(\mathbf{x}; \Theta, \{\mathbf{W}_i\}_{i=1}^N) = \sum_{i=1}^N \mathcal{G}(\mathbf{x}; \Theta)_i f_i(\mathbf{x}; \mathbf{W}_i) \quad (5.1)$$

$$\mathcal{G}(\mathbf{x}; \Theta)_i = \text{softmax}(g(\mathbf{x}; \Theta))_i = \frac{\exp(g(\mathbf{x}; \Theta)_i)}{\sum_{j=1}^N \exp(g(\mathbf{x}; \Theta)_j)} \quad (5.2)$$

While dense MoE can achieve strong performance due to full expert utilization, it is computationally expensive, as all experts participate in every forward pass. The architecture of the dense MoE layer is depicted in Figure 5.2 (a).

5.1.2 Sparse Mixture of Experts

Sparse MoE addresses the inefficiency of dense MoE by activating only a subset of experts for each input. Specifically, the gating function selects the top- k experts based on their scores, significantly reducing

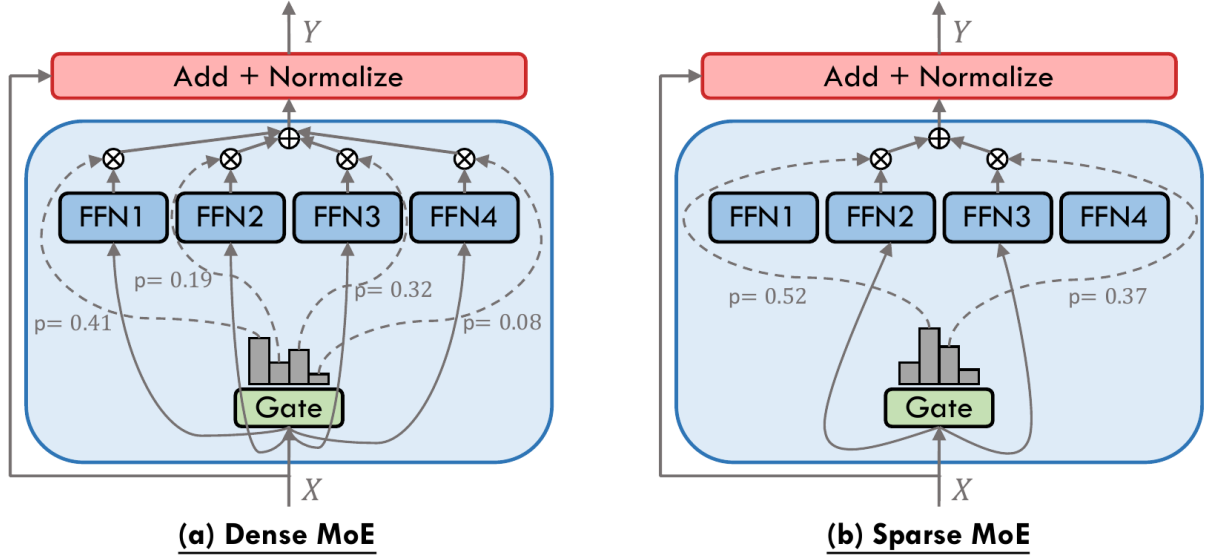


Figure 5.2: Illustration of an MoE layer in transformer-based models. For each input X , the Linear-softmax gating mechanism will (a) select all experts in a dense MoE or (b) select the top- k experts in a sparse MoE to perform conditional computation. The expert layer returns the output of the selected expert multiplied by the gate value, which is the softmax of the gating function output [19].

computation while maintaining high model capacity. The sparse gating mechanism modifies the gating function as indicated in Equation 5.3, where the TopK operator is defined in Equation 5.4.

$$\mathcal{G}(\mathbf{x}; \Theta)_i = \text{softmax}(\text{TopK}(g(\mathbf{x}; \Theta) + \mathcal{R}_{\text{noise}}, k))_i \quad (5.3)$$

$$\text{TopK}(g(\mathbf{x}; \Theta), k)_i = \begin{cases} g(\mathbf{x}; \Theta)_i, & \text{if } g(\mathbf{x}; \Theta)_i \text{ is among the top-}k \text{ values of } g(\mathbf{x}; \Theta) \\ -\infty, & \text{otherwise} \end{cases} \quad (5.4)$$

After applying softmax, entries set to $-\infty$ effectively become zero, ensuring that only the top- k experts are active. In practice, small values of k , such as $k = 1$ or $k = 2$, are commonly used. The addition of a noise term $\mathcal{R}_{\text{noise}}$ promotes exploration among experts and improves training stability. The architecture of the sparse MoE layer is depicted in Figure 5.2 (b).

Although sparse MoE significantly improves computational efficiency, it can introduce a load balancing problem, where some experts are over-utilized while others remain underused. To mitigate this, auxiliary loss functions are employed to encourage balanced expert usage. Given a batch $\mathcal{B} = \{x_1, x_2, \dots, x_T\}$, the load-balancing loss is defined in Equation 5.5.

$$\mathcal{L}_{\text{load-balancing}} = N \sum_{i=1}^N \mathcal{D}_i \mathcal{P}_i \quad (5.5)$$

where

$$\mathcal{D}_i = \frac{1}{T} \sum_{x \in \mathcal{B}} \mathbb{1}\{\arg \max \mathcal{G}(\mathbf{x}; \Theta) = i\} \quad (5.6)$$

$$\mathcal{P}_i = \frac{1}{T} \sum_{x \in \mathcal{B}} \mathcal{G}(\mathbf{x}; \Theta)_i \quad (5.7)$$

Here, $\mathcal{D}_i$ represents the fraction of tokens assigned to expert i and $\mathcal{P}_i$ represents the average gating probability for that expert. Minimizing this loss encourages uniform expert utilization. The optimal condition is achieved when $\mathcal{D}_i$ and $\mathcal{P}_i$ are given by Equation 5.8, ensuring a balanced workload across all experts.

$$\mathcal{D}_i = \frac{1}{N}, \quad \mathcal{P}_i = \frac{1}{N}, \quad \forall i \quad (5.8)$$

Overall, a sparse MoE provides an effective trade-off between model capacity and computational efficiency, making it the dominant approach in modern LLMs. Unless otherwise specified, when we use the term MoE here, we want to refer the sparse variant.

5.2 Taxonomy of Mixture of Experts

Mixture of Experts (MoE) has emerged as a key architectural paradigm for scaling model capacity without incurring proportional increases in computational cost. By leveraging a set of specialized expert networks and a gating mechanism that dynamically selects a subset of experts for each input, MoE enables conditional computation, where resources are allocated adaptively based on input complexity. This property has made MoE particularly attractive for large language models (LLMs), where scaling efficiency is critical. As a result, MoE-based designs are now widely adopted in modern LLMs, enabling substantial increases in weight count alongside improved performance.

Recent advances demonstrate the effectiveness of this paradigm across a range of models. For instance, Mixtral-8x7B builds upon the Mistral 7B architecture by incorporating eight parallel feed-forward networks (experts) per layer. Despite activating only a fraction of its total weights (approximately 13B active weights), it achieves performance competitive with significantly larger dense models such as LLaMA-2-70B and GPT-3.5. Similarly, DeepSeekMoE extends the DeepSeek LLM with an MoE design, achieving comparable performance to LLaMA 2 7B while requiring substantially less computation. Smaller-scale models further highlight the efficiency of MoE: Qwen1.5-MoE, with only 2.7B active weights, matches or surpasses the performance of standard 7B dense models.

Given the rapid development and increasing complexity of MoE-based LLMs, a systematic framework is necessary to organize existing research and guide future work. For example, we can adopt a taxonomy that categorizes MoE advancements along three primary dimensions: (i) algorithm design, which focuses on architectural innovations such as gating mechanisms, expert structures, routing strategies, and auxiliary objectives; (ii) system design, which addresses the practical challenges of deploying MoE models, including computation efficiency, communication overhead, and memory management in distributed environments; (iii) application, which examines the use of MoE across various domains, including natural language processing, computer vision, recommender systems, and multimodal tasks.

Figure 5.3 presents an overview of this taxonomy along with representative studies in each category. In particular, auxiliary loss functions play a crucial role in stabilizing training and improving expert utilization. Table 5.1 summarizes commonly used auxiliary loss functions, such as load-balancing loss, importance loss, routing regularization, and mutual information-based losses, along with their typical coefficient settings across different implementations. In the following sections, we provide a detailed analysis of this taxonomy, highlighting the key design choices and the system-level considerations.

5.3 Algorithm Design of Mixture of Experts

5.3.1 Gating Function

The gating function, also referred to as the routing function or router, is a fundamental component of mixture-of-experts (MoE) architectures. It determines which experts are activated for a given input and how their outputs are combined. Based on their processing strategies, gating mechanisms can be categorized into

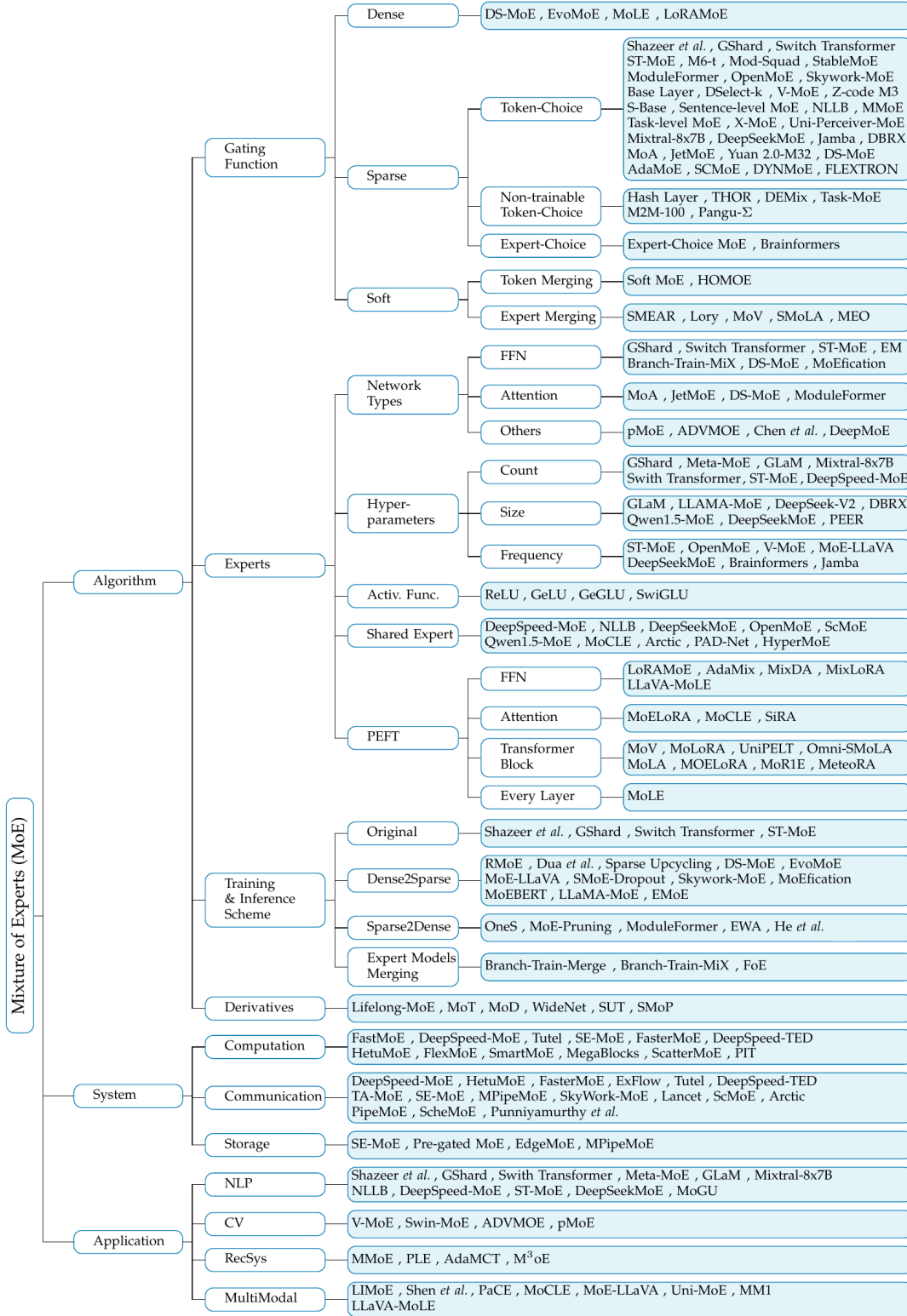


Figure 5.3: Taxonomy of Mixture of Experts [19].

Reference	Auxiliary loss	Coefficient
Shazeer et al. [177], <u>V-MoE</u> [170]	$L_{importance} + L_{load}$	$w_{importance} = 0.1, w_{load} = 0.1$
GShard [114], Switch-T [62], GLaM [56], Mixtral-8x7B [99], DBRX [45], Jamba [122], DeepSeekMoE [44], DeepSeek-V2 [49], Skywork-MoE [207]	L_{aux}	$w_{aux} = 0.01$
ST-MoE [246], OpenMoE [217], MoA [231], JetMoE [180]	$L_{aux} + L_z$	$w_{aux} = 0.01, w_z = 0.001$
Mod-Squad [31], <u>Moduleformer</u> [181], <u>DS-MoE</u> [154]	L_{MI}	$w_{MI} = 0.001$

Table 5.1: Overview of diverse auxiliary loss functions and their typical coefficient configurations. The work that first proposed each auxiliary loss is highlighted in bold text, followed by references that adopts the same approach. Studies that modified the original formulation are indicated with underlined text [19].

three types: sparse, dense, and soft gating. Sparse gating activates a subset of experts, dense gating activates all experts, and soft gating introduces fully differentiable routing mechanisms, often via token or expert merging.

Sparse gating is the dominant paradigm in modern MoE models, enabling conditional computation by activating only a subset of experts per input token. As illustrated in Figure 5.4, a variety of sparse routing strategies have been explored.

In **token-choice gating**, each input token selects its top- k experts based on gating scores. This strategy, introduced by Shazeer et al. [177], allows multiple experts to contribute to a single token representation, improving performance through an ensemble-like behavior. The top- k mechanism has been widely adopted, although simpler variants such as top-1 routing (Figure 5.4 (a)) have also demonstrated competitive performance.

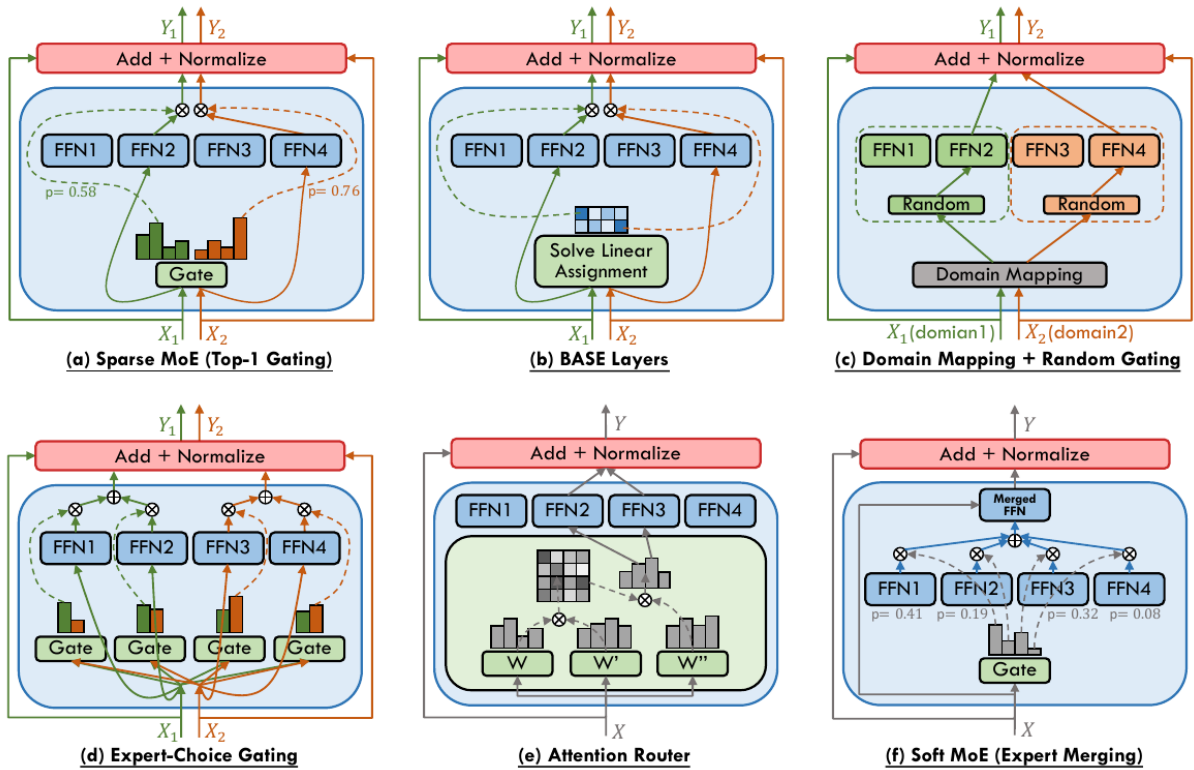


Figure 5.4: The illustration of various gating functions employed in MoE models, including (a) sparse MoE with top-1 gating [62], (b) BASE layers [115], (c) the combination of grouped domain mapping and random gating [169], (d) expert-choice gating [243], (e) attention router [212], and (f) soft MoE with expert merging [142].

Addressing the training instability

In [61], the authors found that the gating mechanism cannot be computed in low precision because MoE models

are sensitive to roundoff errors. Due to the exponential in the gating function, defined in Equation 5.2, roundoff errors can occur in the gating layer and lead to training instabilities. For this reason, the gating logits $g(\mathbf{x}; \Theta) = \Theta \cdot \mathbf{x}$ that are too large could negatively impact training stability. However, simply clipping the gating logits by applying a hard threshold to remove large values, can prejudice the model performance. Instead, the router z-loss defined in Equation 5.9 was proposed in [56] as an auxiliary loss to encourage smaller values, and fewer roundoff errors, within the gating mechanism.

$$\mathcal{L}_z = \frac{1}{T} \sum_{x \in \mathcal{B}} \left(\log \sum_{i=1}^N e^{\Theta \cdot x_i} \right)^2 \quad (5.9)$$

Where T is the number of tokens in the minibatch $\mathcal{B}$, N is the number of experts, and $\Theta \cdot x_i$ are the gating logits.

This loss only penalizes large logits within the gating mechanism of the MoE. To incorporate the gating z-loss into the training process of the MoE model, we can simply combine it with the normal training loss, for example the cross-entropy loss, and the auxiliary load balancing loss (Equation 5.10), where we weight these losses according to their relative importance using scalars c_{LB} and c_z . The gating z-loss significantly improves training stability without causing any noticeable deterioration in model quality.

$$\mathcal{L}_{total} = \mathcal{L}_{CE} + c_{LB} \cdot \mathcal{L}_{load-balancing} + c_z \cdot \mathcal{L}_z \quad (5.10)$$

Solving the fine-tuning problems

Although sparse models perform well with large datasets, we have seen that fine-tuning these models can be difficult. In [246], authors confirm that these fine-tuning problems are due to overfitting. They found that a sparse model converges more quickly than a dense model in both cases. However, the validation quality of the sparse model is much worse than that of the dense model on the downstream task with less data. In other words, the sparse model overfits due to a lack of sufficient data.

To mitigate overfitting, the authors' first attempt was to only fine-tune certain portions of the MoE model's weights. Fine-tuning just the MoE layers yielded a drastic deterioration in performance, while selectively fine-tuning other groups of weights performs comparably to fine-tuning all weights. MoE models benefit from high learning rates and smaller batch sizes, both of which improve generalization by introducing more noise to the training process. Additionally, the capacity factor does not drastically impact fine-tuning results. In the end, the authors of [246] point out that there is no definitive solution to solve the difficulties of fine-tuning MoE models.

Load balancing

Mixture of Experts models utilize a gating network for dynamic routing. This architectural choice introduces a significant training challenge: the potential for severe load imbalance. If the model is not properly encouraged, the gating network might learn to route a disproportionate number of tokens to a small, favored subset of experts. This behavior undermines the entire principle of MoE, as it leaves most of the model's capacity underutilized and leads to training instabilities. To address the imbalance issue, MoE models incorporate an **auxiliary loss function** specifically designed to encourage a balanced distribution of tokens across all available experts. This loss is added to the primary task loss, for example the cross-entropy, during training, guiding the router toward more balanced decisions.

Let us consider a gating network that, early in training, discovers that one or two experts are slightly better at processing certain common tokens. During backpropagation, the gating network's weights will be updated to favor these experts even more. This creates a feedback loop where the favored experts receive more training data, become more specialized and effective, and are therefore chosen even more frequently. This phenomenon, often called **expert collapse**, has two major negative consequences: (i) under-trained experts since the neglected experts receive few or no tokens. In this way, their weights are rarely updated, and they hardly learn any useful specialization. Therefore, they will not contribute to the model's performance; (ii) inefficient capacity utilization; if only a fraction of the model's weights are active, the effective model capacity is very small. Thus,

we have large memory cost associated to a large model without obtaining the computational and performance benefits.

Figure 5.5 illustrates the difference between an imbalanced state, where a few experts dominate, and the desired balanced state.

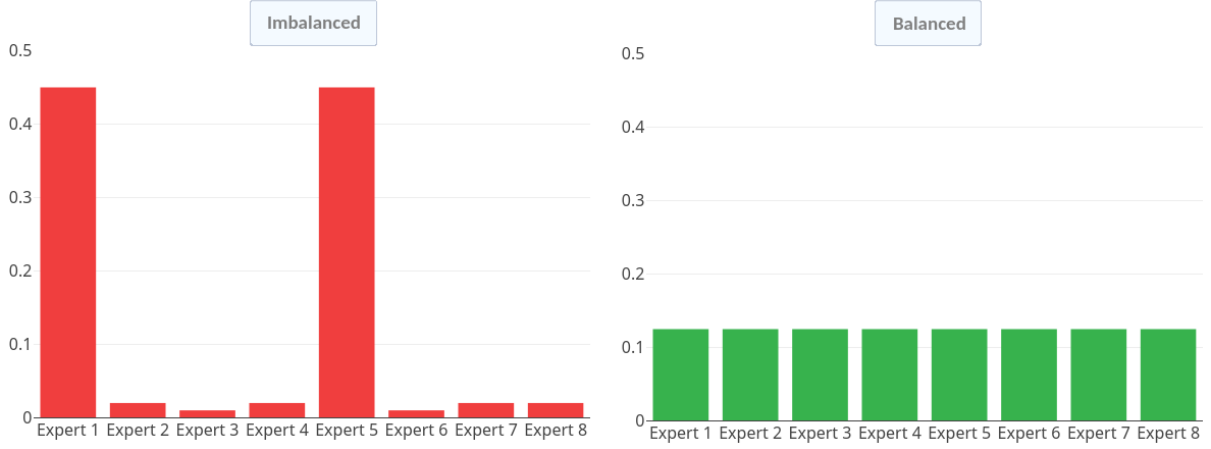


Figure 5.5: An imbalanced expert utilization concentrates computation on a few experts, leading to collapse. The auxiliary loss encourages the gating network to achieve a balanced expert utilization, ensuring all experts are selected [113].

The goal a load balancing loss term is to penalize the gating network for the imbalance. The most common approach, proposed in [177], is to calculate a value based on the distribution of tokens and router probabilities across a minibatch. Let us consider two quantities for a batch of tokens, where N is the total number of experts, (i) the fraction of tokens in a minibatch B that assigned to each expert, $\mathcal{D}_i$, which was defined in Equation 5.6; If we have B tokens in the minibatch and use top-1 gating, this is simply the number of tokens routed to expert i , divided by B ; (ii) the average router probability per expert, $\mathcal{P}_i$, which was defined in Equation 5.7; It represents the importance the gating network gives to an expert, regardless of whether it is chosen or not.

The auxiliary loss, $\mathcal{L}_{aux}$, is then calculated as the dot product of these two vectors, scaled by the number of experts N (Equation 5.11).

$$\mathcal{L}_{aux} = N \cdot \sum_{i=1}^N \mathcal{D}_i \cdot \mathcal{P}_i \quad (5.11)$$

To minimize this loss, the model must avoid that any single expert i has both a high token fraction $\mathcal{D}_i$ and a high average probability $\mathcal{P}_i$. The loss is lowest when the products $\mathcal{D}_i \cdot \mathcal{P}_i$ are small and evenly distributed, which happens when the token assignment to experts is balanced. The auxiliary loss is then combined with the main task loss, for example $\mathcal{L}_{CE}$, to form the final loss function that is used in backpropagation (Equation 5.12).

$$\mathcal{L}_{total} = \mathcal{L}_{CE} + \alpha \cdot \mathcal{L}_{aux} \quad (5.12)$$

The constant α is a small scalar value that controls the strength of the balancing incentive. If α is too small, the balancing strength will be too weak to prevent expert collapse. If α is too large, the model may prioritize perfect load balancing at the expense of its performance on the main task, leading to poor overall accuracy. A common starting value for α is around 0.01. This simple but effective mechanism is a standard component in nearly all MoE training pipelines, acting as the essential regulator that allows these large, sparse models to be trained effectively.

Table 5.1 summarizes commonly used auxiliary losses and their coefficient configurations across different studies. One formulation measures the importance of each expert within a batch $\mathcal{B}$ using Equation 5.13.

$$\text{Importance}(\mathcal{B}) = \sum_{x \in \mathcal{B}} \mathcal{G}(\mathbf{x}; \Theta) \quad (5.13)$$

Based on this value, the importance regularization term is defined in Equation 5.14, where CV denotes the coefficient of variation.

$$\mathcal{L}_{\text{importance}} = \text{CV}(\text{Importance}(\mathcal{B}))^2 \quad (5.14)$$

In prior work [246], it was found that the routing mechanism cannot be computed in low precision because MoE models are sensitive to roundoff errors. Due to the exponential in the routing function (shown below), such roundoff errors can occur in the router and lead to training instabilities.

In addition to auxiliary losses, **expert capacity** constraints are commonly imposed to limit the number of tokens processed by each expert. As discussed in GShard [114], this introduces challenges such as token overflow and routing inefficiencies. Various strategies, including prioritized routing, randomization, and batch-level heuristics, have been proposed to address these issues. Recent analyses, such as OpenMoE [217], highlight additional phenomena, such as routing bias toward early tokens ("Drop-towards-the-End") and early specialization of experts during training.

The Switch Transformer uses a global expert capacity variable to define the maximum number of tokens that can be routed to each expert in any MoE layer. The expert capacity is calculated with Equation 5.15.

$$\text{expert capacity} = \frac{\text{tokens per minibatch}}{\text{number of experts}} \cdot \text{capacity factor} \quad (5.15)$$

In the gating (or switch) layer, each token is routed to the expert that is assigned the highest probability by the gating mechanism. If too many tokens, namely a number of tokens exceeding the expert capacity, are sent to a single expert, computation for these tokens will be skipped. The skipped tokens are passed directly to the next layer via the residual connection. Setting the capacity factor greater than one allows the MoE to handle cases where tokens are not perfectly balanced across experts (Figure 5.6). The expert capacity must be tuned to avoid having too many dropped tokens, while not wasting computation or memory. The load balancing loss can also help with this by evenly routing tokens to experts. Switch Transformer performs better at lower capacity factors compared to a vanilla MoE, which allows us to improve training efficiency via better load balancing between experts.

To improve the routing quality and load balancing, several extensions have been proposed.

- Balanced assignment approaches, such as BASE layers (Figure 5.4 (b)), which reformulate routing as an optimization problem.
- Continuous relaxations, like DSelect-k, improve differentiability.
- Adaptive routing methods dynamically adjust the number of selected experts per token.
- Alternative routing units, such as sentence-level gating, aggregate token information before routing.
- Advanced routing networks, including attention-based routers (Figure 5.4 (e)), improve expert selection efficiency.

Additionally, non-trainable routing approaches have been explored, including hashing-based routing and domain-based mappings. Hybrid strategies, such as Random Routed Experts (Figure 5.4 (c)), combine structured and stochastic routing.

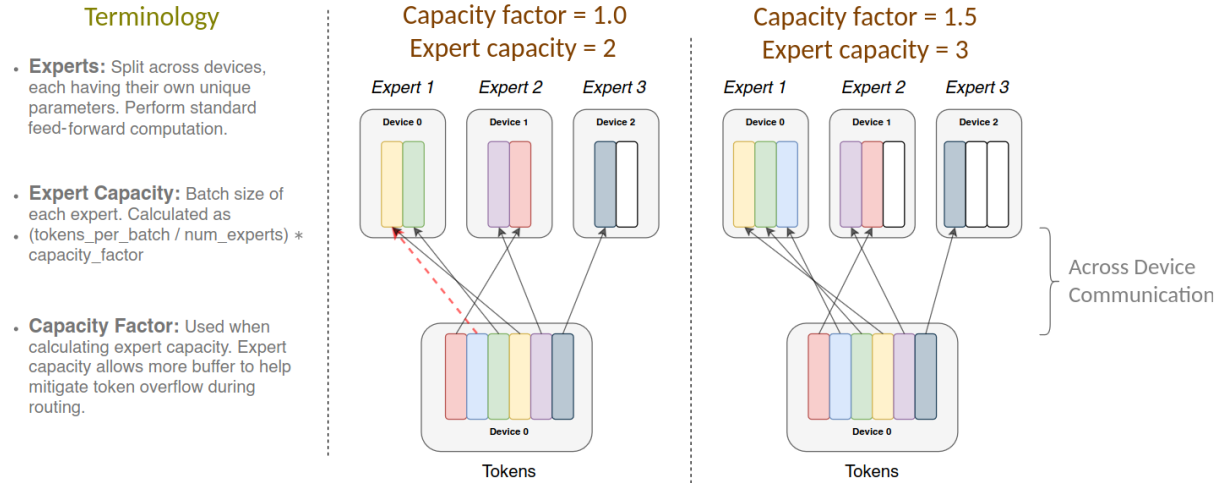


Figure 5.6: Illustration of token routing dynamics in switch transformer. Each expert processes a fixed batch-size of tokens modulated by the capacity factor. Each token is routed to the expert with the highest router probability, but each expert has a fixed batch size of $(\text{total tokens} / \text{number experts}) * \text{capacity factor}$. If the tokens are unevenly dispatched then certain experts will overflow, indicated by dotted red lines, resulting in these tokens not being processed by this layer. A larger capacity factor alleviates this overflow issue, but also increases computation and communication costs [61].

An alternative paradigm, known as **expert-choice gating** (Figure 5.4 (d)), reverses the routing process: each expert selects the top- k tokens it processes. This inherently balances expert workloads without auxiliary losses but may lead to uneven token coverage.

Dense gating, formulated in Equation 5.1, activates all experts for each input. While computationally expensive, it can yield strong performance due to complete expert utilization. Dense MoE remains relevant in specific contexts, such as weight-efficient fine-tuning, where expert modules are lightweight and the overhead of activating all experts is manageable.

Soft MoE addresses the discrete optimization challenges of sparse routing by employing fully differentiable mechanisms. As shown in Figure 5.4 (f), soft approaches rely on weighted combinations rather than hard expert selection.

Token merging. Soft MoE methods, for example [157], merge input tokens via weighted averaging before expert processing. This approach improves training stability and naturally maintains load balance, though it is less suitable for autoregressive settings.

Expert merging. Alternatively, expert merging approaches combine expert weights through weighted averaging, resulting in a single effective expert representation. Methods such as SMEAR (Figure 5.4 (f)) avoid discrete routing decisions and enable standard gradient-based optimization.

Subsequent work extends this idea to autoregressive models and fine-tuning settings, showing improved performance over both sparse and dense baselines. Additional variants, like MoA in Figure 5.7 (a), further explore combining expert structures with attention mechanisms.

Thus, gating function design plays a central role in determining MoE performance, efficiency, and scalability. While sparse routing remains the dominant choice in LLMs, alternative approaches, dense and soft, offer complementary advantages under different computational and application constraints, as summarized in Figure 5.4, Figure 5.7, and Table 5.1.

5.3.2 Experts

This section examines the design of expert networks within the MoE framework, complementing the gating mechanisms discussed in previous section.

Since the integration of MoE into transformer architectures [114] [62] [246], MoE layers (experts) have been replacing the feed-forward network (FFN) modules of transformers. In most implementations, each expert

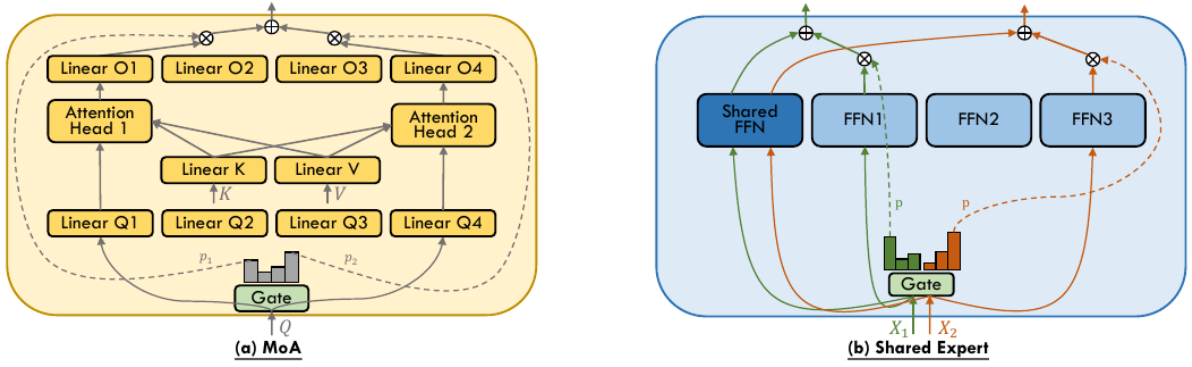


Figure 5.7: Illustration of (a) mixture of attention heads [231] and (b) shared expert [163] architectures [19].

mirrors the structure of a standard FFN, a design paradigm that remains dominant. Subsequent refinements to this structure are discussed further ahead in this section.

The preference for **feed-forward network** experts stems from the observation that FFN layers exhibit greater sparsity and specialization compared to self-attention layers. Empirical analyses [154] show that FFN experts activate more selectively (around 20%) compared to attention layers (around 80%), indicating stronger specialization. Earlier studies [233] [121] also demonstrate that only a subset of FFN neurons are active per input. Furthermore, the Emergent Modularity phenomenon [236] reveals that neurons naturally organize into task-specific groups, supporting the suitability of MoE as a modular extension of pretrained transformers.

Beyond FFNs, recent work has explored the combination of MoE with the **attention** mechanism. For example, the Mixture of Attention Heads (MoA) architecture [231], illustrated in Figure 5.7 (a), introduces experts into the attention layer, specifically for query and output projections. To reduce the computational overhead, key and value projections are shared across experts, while only query and output projections differ. Subsequent models such as DS-MoE [154], JetMoE [180], and ModuleFormer [181] further refine this integration of MoE with attention layers.

Alternative expert architectures have also been explored, including convolutional neural networks (CNNs) [34] [232] [32] [205] [72]. More recently, weight-efficient fine-tuning (PEFT) techniques, such as LoRA-based experts, have been incorporated into MoE frameworks, offering improved adaptability and efficiency (Section 3).

The performance and efficiency of MoE models depend heavily on several key **hyperparameters**: (i) the number of experts per layer, (ii) the size of each expert, and (iii) the frequency of MoE layers. Table 5.2 provides a comparative overview of these design choices, while Table 5.3 summarizes configurations and benchmark results for recent open-source models.

Early MoE models employed thousands of **experts per layer** [177] [114] [62], achieving strong performance but suffering from reduced robustness under domain shift [8]. Later work, for example GLaM [56], identified that using around 64 experts with top-2 gating offers a strong balance between efficiency and performance. Reflecting this finding, most modern MoE models (Table 5.3) include up to 64 experts per layer [99] [45] [246] [207] [161].

Scaling **expert capacity** can be achieved by increasing the intermediate dimension of each expert. For example, GLaM [56] prioritizes larger expert dimensions while keeping the number of experts fixed. In contrast, DeepSeekMoE [49] [44] introduced fine-grained expert segmentation, splitting FFN dimensions into smaller experts while increasing both expert count and active experts. This improves specialization and flexibility. Similar strategies are adopted in Qwen1.5-MoE [161] and DBRX [45]. Additionally, approaches like PEER [81] demonstrate that very fine-grained experts, even single-neuron experts, can achieve strong performance vs. efficiency trade-offs.

MoE layers are typically inserted in place of FFNs at regular intervals. Common configurations include replacing every second layer (1/2) [114] [56], every fourth layer (1/4) [246], or all layers (1/1) [62] [44]. Recent trends, such as Mixtral [99], favor placing MoE in every layer with fewer experts per layer. Variants,

Reference	Models	Expert Count (Activ./Total)	d_{model}	d_{ffn}	d_{expert}	#L	#H	d_{head}	Placement Frequency	Activation Function	Share Expert Count
GShard [114] (2020)	600B	2/2048	1024	8192	d_{ffn}	36	16	128	1/2	ReLU	0
	200B	2/2048	1024	8192	d_{ffn}	12	16	128	1/2	ReLU	0
Switch [62] (2021)	150B	2/512	1024	8192	d_{ffn}	36	16	128	1/2	ReLU	0
	37B	2/128	1024	8192	d_{ffn}	36	16	128	1/2	ReLU	0
	7B	1/128	768	2048	d_{ffn}	12	12	64	1/2	GEGLU	0
	26B	1/128	1024	2816	d_{ffn}	24	16	64	1/2	GEGLU	0
GLaM [56] (2021)	395B	1/64	4096	10240	d_{ffn}	24	64	64	1/2	GEGLU	0
	1571B	1/2048	2080	6144	d_{ffn}	15	32	64	1	ReLU	0
	0.1B/1.9B	2/64	768	3072	d_{ffn}	12	12	64	1/2	GEGLU	0
	1.7B/27B	2/64	2048	8192	d_{ffn}	24	16	128	1/2	GEGLU	0
DeepSpeed-MoE [163] (2022)	8B/143B	2/64	4096	16384	d_{ffn}	32	32	128	1/2	GEGLU	0
	64B/1.2T	2/64	8192	32768	d_{ffn}	64	128	128	1/2	GEGLU	0
	350M/13B	2/128	1024	$4d_{model}$	d_{ffn}	24	16	64	1/2	GeLU	0
	1.3B/52B	2/128	2048	$4d_{model}$	d_{ffn}	24	16	128	1/2	GeLU	0
	PR-350M/4B	2/32-2/64	1024	$4d_{model}$	d_{ffn}	24	16	64	1/2, 10L-32E, 2L-64E	GeLU	1
	PR-1.3B/31B	2/64-2/128	2048	$4d_{model}$	d_{ffn}	24	16	128	1/2, 10L-64E, 2L-128E	GeLU	1
ST-MoE [246] (2022)	0.8B/4.1B	2/32	1024	2816	d_{ffn}	27	16	64	1/4, add extra FFN	GEGLU	0
	32B/269B	2/64	5120	20480	d_{ffn}	27	64	128	1/4, add extra FFN	GEGLU	0
Mixtral [99] (2023)	13B/47B	2/8	4096	14336	d_{ffn}	32	32	128	1	SwiGLU	0
LLAMA-MoE [160] (2023)	39B/141B	2/8	6144	16384	d_{ffn}	56	48	128	1	SwiGLU	0
	3.0B/6.7B	2/16	4096	11008	688	32	32	128	1	SwiGLU	0
	3.5B/6.7B	4/16	4096	11008	688	32	32	128	1	SwiGLU	0
DeepSeekMoE [44] (2024)	3.5B/6.7B	2/8	4096	11008	1376	32	32	128	1	SwiGLU	0
	0.24B/1.89B	8/64	1280	-	$1/4 d_{ffn}$	9	10	128	1	SwiGLU	1
	2.8B/16.4B	8/66	2048	10944	1408	28	16	128	1, except 1st layer	SwiGLU	2
	22B/145B	16/132	4096	-	$1/8 d_{ffn}$	62	32	128	1, except 1st layer	SwiGLU	4
	8339M/650M	2/16	768	3072	d_{ffn}	12	12	64	1/4	SwiGLU	1
OpenMoE [217] (2024)	2.6B/8.7B	2/32	2048	8192	d_{ffn}	24	24	128	1/6	SwiGLU	1
Qwen1.5-MoE [161] (2024)	6.8B/34B	2/32	3072	12288	d_{ffn}	32	24	128	1/4	SwiGLU	1
	2.7B/14.3B	8/64	2048	5632	1408	24	16	128	1	SwiGLU	4
DBRX [45] (2024)	36B/132B	4/16	6144	10752	d_{ffn}	40	48	128	1	SwiGLU	0
Jamba [122] (2024)	12B/52B	2/16	4096	14336	d_{ffn}	32	32	128	1/2, 1:7 Attn. :Mamba	SwiGLU	0
Skywork-MoE [207] (2024)	22B/146B	2/16	4608	12288	d_{ffn}	52	36	128	1	SwiGLU	0
Yuan 2.0-M32 [212] (2024)	3.7B/40B	2/32	2048	8192	d_{ffn}	24	16	256	1	SwiGLU	0
OLMoE [141] (2024)	1.3B/6.9B	8/64	2048	1024	d_{ffn}	16	16	128	1	SwiGLU	0
DeepSeek-V3 [51] (2024)	37B/671B	9/257	7168	18432	2048	61	128	128	1, except 1st 3 layers	SwiGLU	1

Table 5.2: Comparative configurations of MoE with FFN experts in selected models. Model differentiation in each reference is achieved by using the model size, indicated either by total or activated/total weight count. Both activated and total expert counts encompass the count of shared experts when utilized. d_{model} is the hidden size, d_{ffn} is the intermediate size of FFNs, d_{expert} is the intermediate size of FFN experts, #L is the number of layers, #H and d_{head} are the number of attention heads and attention head dimensions [19].

Name	Time	Affiliation	Params.		Benchmarks				Link
			Activ.	Total	MLLM	GSM8K	MATH	HumanEval	
Mistral-8x7B-v0.1	2023.12	Mistral	13B	47B	70.6	58.4, 74.4 (8-shot)	28.4	40.2	https://huggingface.co/mistralai/Mistral-8x7B-v0.1
DeepSeekMoE-16B-Base	2024.1	DeepSeek	3B	16B	45.0	18.8 (8-shot)	4.3	26.8	https://huggingface.co/deepseek-ai/deepseek-moe-16b-base
Grok-1	2024.3	xAI	86B	314B	73.0	62.9	23.9	63.2	https://github.com/xai-org/grok-1
Qwen1.5-MoE-A2.7B	2024.3	Alibaba	3B	14B	62.5	61.5 (8-shot)	-	34.2	https://huggingface.co/Qwen/Qwen1.5-MoE-A2.7B
DBRX Instruct	2024.3	Databricks	36B	132B	73.7	72.8	-	70.1	https://huggingface.co/databricks/dbrx-instruct
Jamba-v0.1	2024.3	AI21 Labs	12B	52B	67.4	59.9 (3-shot)	-	29.3	https://huggingface.co/ai21labs/Jamba-v0.1
Mistral-8x22B-v0.1	2024.4	Mistral	39B	141B	77.8	78.6, 88.4 (8-shot)	41.8	45.1	https://huggingface.co/mistralai/Mistral-8x22B-v0.1
Arctic Instruct	2024.4	Snowflake	17B	480B	67.3	74.2	-	-	https://huggingface.co/Snowflake/snowflake-arctic-instruct
DeepSeek-V2	2024.5	DeepSeek	21B	236B	78.5	79.2 (8-shot)	43.6	48.8	https://huggingface.co/deepseek-ai/DeepSeek-V2
DeepSeek-V2-Chat (RL)	2024.5	DeepSeek	21B	236B	77.8	92.2 (8-shot)	53.9	81.1	https://huggingface.co/deepseek-ai/DeepSeek-V2-Chat
Yuan 2.0-M32	2024.5	IEIT	4B	40B	72.2	92.7 (8-shot)	55.9 (8-shot)	74.4	https://huggingface.co/IEIT/Yuan/Yuan2-M32
Skywork-MoE-Base	2024.6	Kunlun	22B	146B	77.4	76.1	31.9	43.9	https://huggingface.co/Skywork/Skywork-MoE-Base
OLMoE-0924-Base	2024.9	Ai2	1.3B	6.9B	54.1	3.0 (8-shot)	-	22.4	https://huggingface.co/allenai/OLMoE-1B-7B-0924
DeepSeek-V3-Base	2024.12	DeepSeek	37B	671B	87.1	89.3 (8-shot)	61.6	65.2	https://huggingface.co/deepseek-ai/DeepSeek-V3-Base

Table 5.3: A collection of recent open-source models detailing activated and total weight counts, alongside performance benchmarks such as MMLU [82] (5-shot), GSM8K [37] (5-shot), MATH [83] (4-shot), and HumanEval [26] (0-shot), unless specified otherwise [19].

such as V-MoE [170] and MoE-LLaVA [123], highlight trade-offs between computational cost and performance depending on the expert placement. Figure 5.7 and Tables 5.2 and 5.3 illustrate these design variations.

MoE models inherit the activation function choice from dense transformers. Over time, **activation functions** have evolved from ReLU [70] to more advanced variants such as GeLU [84], GeGLU, and SwiGLU [175], which are now widely adopted. In addition, modern MoE architectures commonly incorporate other improvements from LLM design, including RMSNorm, Grouped Query Attention (GQA), and Rotary Position Embeddings (RoPE), aligning MoE models with state-of-the-art dense architectures.

Shared expert designs combine dense and sparse computation to improve efficiency and stability. The motivation behind using shared experts is minimizing the amount of redundant information between experts. By having a set of shared experts, we can allow the network to store shared information within these experts, rather than having to replicate the same information across several different experts. For example, DeepSpeed-MoE [163] introduced the Residual-MoE architecture, where each token is processed by both a fixed (shared) expert and a routing-selected expert, maintaining an efficiency comparable to top-1 routing. This concept, illustrated in Figure 5.7 (b), has evolved into a widely adopted design where one or more shared experts complement sparse MoE routing. Models such as DeepSeekMoE [44], OpenMoE [217], and Qwen1.5-MoE [161] incorporate multiple shared experts, particularly when using fine-grained expert segmentation.

To compute the output of an MoE layer with shared experts, we simply add the shared experts’ output to the routed output (Equation 5.16).

$$y = \sum_{j \in \tau_s} E_j(\mathbf{x}) + \sum_{i \in \tau_K} p_i(\mathbf{x}) \cdot E_i(\mathbf{x}) \quad (5.16)$$

Where y is the output of the MoE layer for a token embedding $\mathbf{x}$, τ_s is the set of shared experts, $E_j(\mathbf{x})$ is the output of shared expert j , τ_K are top- K experts, and $p_i(\mathbf{x})$ is the importance of expert i .

Further extensions to shared experts strategy have been proposed.

- PAD-Net [79] converts experts selected dynamically into static shared ones;
- HyperMoE [237] employs shared hypernetworks across experts and layers;
- ScMoE [18] separates shared and routed computation to improve overlap between communication and computation;
- Hybrid architectures, as Snowflake Arctic [186], combine dense and MoE layers, as illustrated in Figure 5.8 (d).

Figure 5.8 provides a broader view of how **weight-efficient experts** (MoPEs) can be integrated at different levels of transformer architecture, including attention modules, FFNs, and entire layers.

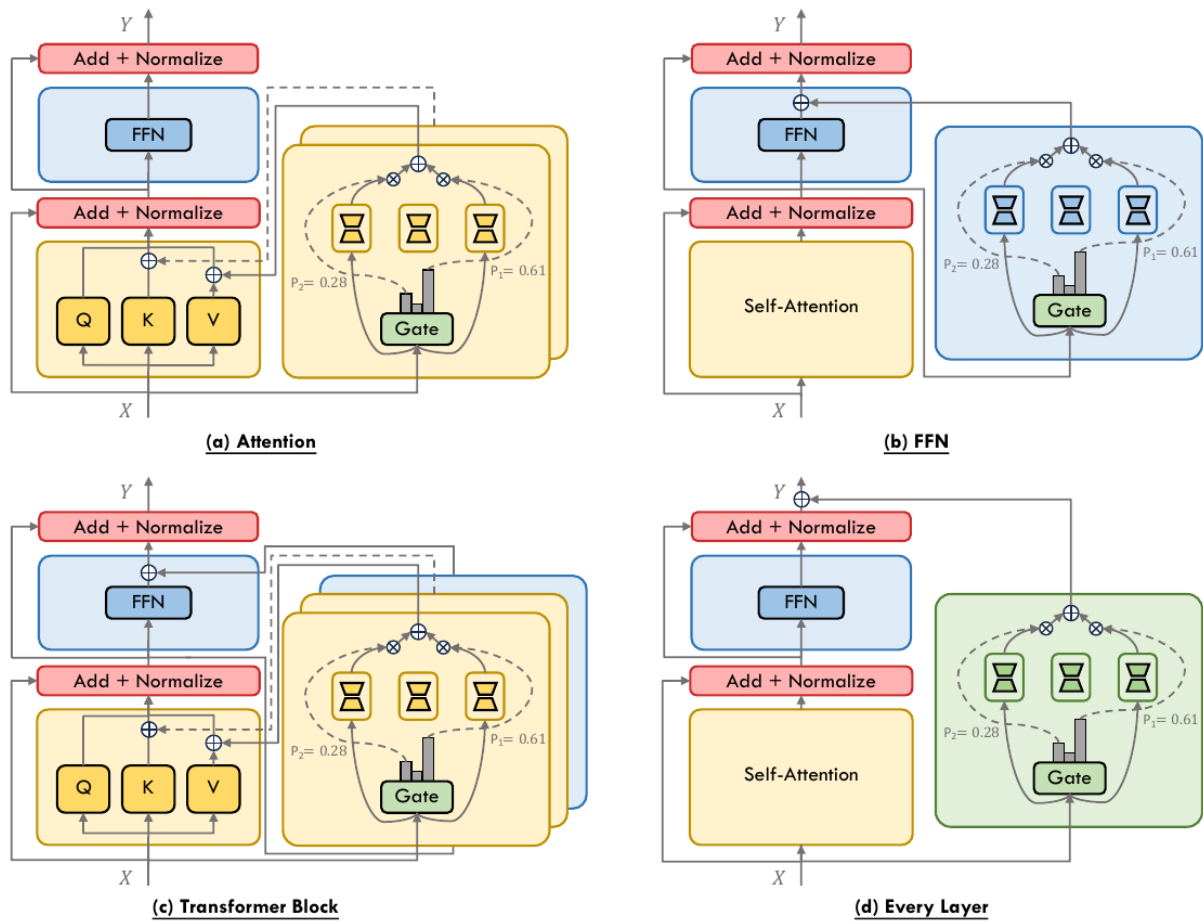


Figure 5.8: The illustration of the taxonomy of MoPEs based placement within the Transformer model architecture. (a) exemplifies the integration of MoPE with the Key and Value modules of the attention mechanism, with applicability extending to Query and Output projection modules. (b) represents the application of MoPE to the FFN. (c) refers to the MoPE integration at the level of the Transformer block, wherein two distinct groups of experts are applied to attention and FFN, where separate sets of experts are allocated to both attention and FFN, each regulated by its own gating mechanism. (d) illustrates a layer-wise integration of MoPE, in which each Transformer layer is regarded as a unified entity with a gating orchestrating the interplay among experts [19].

In conclusion, expert design plays a crucial role in shaping MoE model capacity, efficiency, and specialization. As illustrated across Figure 5.7, Figure 5.8, Table 5.2, and Table 5.3, recent work explores increasingly flexible and fine-grained expert configurations, reflecting a shift toward more modular and adaptive architectures.

5.3.3 Mixture of Weight-Efficient Experts

Large language models (LLMs) pretrained on large-scale datasets have demonstrated strong generalization capabilities across diverse tasks. However, adapting these models to specific downstream applications typically requires fine-tuning. Conventional full fine-tuning, which updates all model weights, becomes increasingly impractical as model sizes grow. To address this limitation, weight-efficient fine-tuning (PEFT) methods have been proposed, which update only a small subset of weights while keeping the base model frozen.

A prominent example is LoRA [86], which introduces low-rank adaptation matrices that capture task-specific updates. While PEFT methods are computationally efficient and effective, they often struggle with multi-task generalization and may suffer from catastrophic forgetting due to limited adaptability.

To overcome these challenges, mixture of weight-efficient experts (MoPE) combines the strengths of MoE and PEFT by constructing experts using weight-efficient modules and coordinating them through a gating mechanism. This integration enhances multi-task performance while maintaining high computational efficiency. Figure 5.8 presents a taxonomy of MoPE designs across different integration levels.

1. Apply MoPE to Feed-Forward Networks

Following the standard MoE design, many MoPE approaches apply weight-efficient experts to the FFN module of transformer blocks. During training, only the expert weights and gating network are updated, leaving the base model unchanged. As illustrated in Figure 5.8 (b), the forward computation can be expressed by Equation 5.17, where $\mathbf{x}'$ is the output of the attention module and is given by Expression 5.18, $\Delta\mathbf{W}_i^{\text{ffn}}$ denotes the weight-efficient experts, and G^{ffn} is the gating function.

$$\text{FFN}^{\text{MoE}}(\mathbf{x}') = \text{FFN}(\mathbf{x}') + \sum_{i=1}^n \mathbf{x}' \Delta\mathbf{W}_i^{\text{ffn}} \cdot G^{\text{ffn}}(\mathbf{x}')_i \quad (5.17)$$

$$\mathbf{x}' = \text{LayerNorm}(\text{SA}(\mathbf{x}) + \mathbf{x}) \quad (5.18)$$

Early work, such as LoRAMoE [54], integrates LoRA-based experts into FFNs, dividing them into groups that separately model task-specific knowledge and alignment with instruction data. Similarly, AdaMix [206] introduces Adapter-based experts with stochastic routing while maintaining low inference cost.

Other approaches focus on multi-stage or domain-aware adaptation. For example, MixDA [53] separates domain adaptation and task adaptation into different training stages. LLaVA-MoLE [27] extends this framework to multimodal settings using top-1 routing. In contrast, MixLoRA [116] aligns closely with standard MoE by combining LoRA experts with shared FFN weights, improving efficiency and reducing latency.

2. Apply MoPE to Attention Layers

MoPE can also be applied to attention mechanisms by inserting weight-efficient experts into the Query, Key, Value, or Output projections. As illustrated in Figure 5.8 (a), integrating MoPE into Key and Value modules yields an attention output given by Equation 5.19, where $\Delta\mathbf{W}^k$ and $\Delta\mathbf{W}^v$ are the changes of weight-efficient experts applied to the key and value projections.

$$\text{SA}^{\text{MoE}}(\mathbf{x}) = \text{Softmax} \left(\frac{\mathbf{Q} \left(\mathbf{K}^T + \sum_{i=1}^n \mathbf{x} \Delta\mathbf{W}_i^k \cdot G^k(\mathbf{x})_i \right)}{\sqrt{d_{\text{head}}}} \right) \left(\mathbf{V} + \sum_{i=1}^n \mathbf{x} \Delta\mathbf{W}_i^v \cdot G^v(\mathbf{x})_i \right) \quad (5.19)$$

Recent methods extend MoE into attention while maintaining efficiency. For example, MoELoRA [135] applies LoRA experts to Query and Value matrices with top- k routing and contrastive regularization. MoCLE [71] addresses task conflicts by clustering tasks and assigning experts accordingly, while incorporating both task-specific and universal experts. While SiRA [244] improves stability through capacity constraints, auxiliary losses, and expert dropout mechanisms.

3. Apply MoPE to Transformer Blocks

At a higher level, MoPE can be applied to entire transformer blocks by introducing separate expert groups for attention and feed-forward network (FFN) components, each governed by its own gating function. As illustrated in Figure 5.8 (c), the forward process over the attention layer and FFN is given by Equation 5.20 and 5.21, respectively.

$$\mathbf{x}' = \text{LayerNorm}(\text{SA}^{\text{MoE}}(\mathbf{x}) + \mathbf{x}) \quad (5.20)$$

$$\mathbf{y} = \text{LayerNorm}(\mathbf{x}' + \text{FFN}^{\text{MoE}}(\mathbf{x}')) \quad (5.21)$$

Representative approaches of this strategy include MoV [224], which uses weight-efficient vectors to modulate attention and FFN components, and MoLORA [224], which combines LoRA experts across both modules. UniPELT [137] further generalizes this idea by integrating multiple PEFT techniques, adapters, prefix-tuning, and LoRA, as experts with dynamic gating. Extensions like Omni-SMoLA [210] introduce modality-specific experts for multimodal tasks, while MoLA [67] explores layer-wise expert allocation to reduce redundancy.

Specialized applications include MOELoRA [55] for medical domains, which incorporates task-informed gating, and Intuition-MoE [229], which leverages semantic clustering to guide routing. These works highlight the flexibility of MoPE in handling diverse tasks and domains.

4. Whole-Layer Integration

Recent work extends MoPE to the entire transformer layer, treating it as a unified unit rather than separating attention and FFN. As shown in Figure 5.8 (d), after incorporating MoPE, the forward process over the attention layer and FFN is expressed by Equation 5.22 and 5.23, respectively.

$$\mathbf{x}' = \text{LayerNorm}(\text{SA}(\mathbf{x}) + \mathbf{x}) \quad (5.22)$$

$$y = \text{LayerNorm}(\mathbf{x}' + \text{FFN}(\mathbf{x}')) + \sum_{i=1}^n \mathbf{x} \Delta \mathbf{W}_i^{\text{layer}} \cdot G^{\text{layer}}(\mathbf{x})_i \quad (5.23)$$

This formulation enables joint modeling of attention and FFN dynamics within a single MoE framework. A representative example is MoLE [213], which treats pretrained LoRA modules as experts and learns layer-wise gating functions to dynamically combine them. This approach allows flexible composition of pretrained components, significantly enhancing adaptability across tasks.

Hence, MoPE integrates the scalability of MoE with the efficiency of PEFT, offering a powerful framework for multi-task adaptation. As illustrated in Figure 5.8, MoPE can be applied at multiple levels, from FFN and attention modules to entire transformer layers, highlighting its versatility and growing importance in modern LLM design.

5.3.4 Training and Inference Scheme for MoEs

Alongside architectural innovations, substantial research has focused on training and inference strategies for mixture-of-experts models, aiming to improve both computational efficiency and model quality. Early MoE approaches follow a straightforward paradigm [170] [177] [114] [62] [246]: the MoE model is constructed and trained from scratch, and inference directly mirrors the training configuration without structural changes (Figure 5.9 (a)). More recent work introduces flexible schemes that leverage transitions between dense and sparse representations to combine their complementary advantages. As illustrated in Figure 5.9, these approaches can be categorized into three main paradigms.

- Dense-to-sparse, where training begins with a dense model and transitions to a sparse MoE configuration (Figure 5.9 (c));
- Sparse-to-dense, where a sparse model is converted into a dense representation for efficient deployment (Figure 5.9 (d));
- Expert model merging, where multiple pretrained models are integrated into a unified MoE framework (Figure 5.9 (b)).

Dense-to-sparse approaches aim to reduce training cost and improve stability by leveraging pretrained dense models. Early methods, such as RMoE [211], first train a dense model and then fine-tune it into an MoE architecture, highlighting the need to align expert outputs with the original FFN representations. Similarly, [57] introduces sparsity by adding a gating module after dense pretraining and refining FFNs under sparse activation.

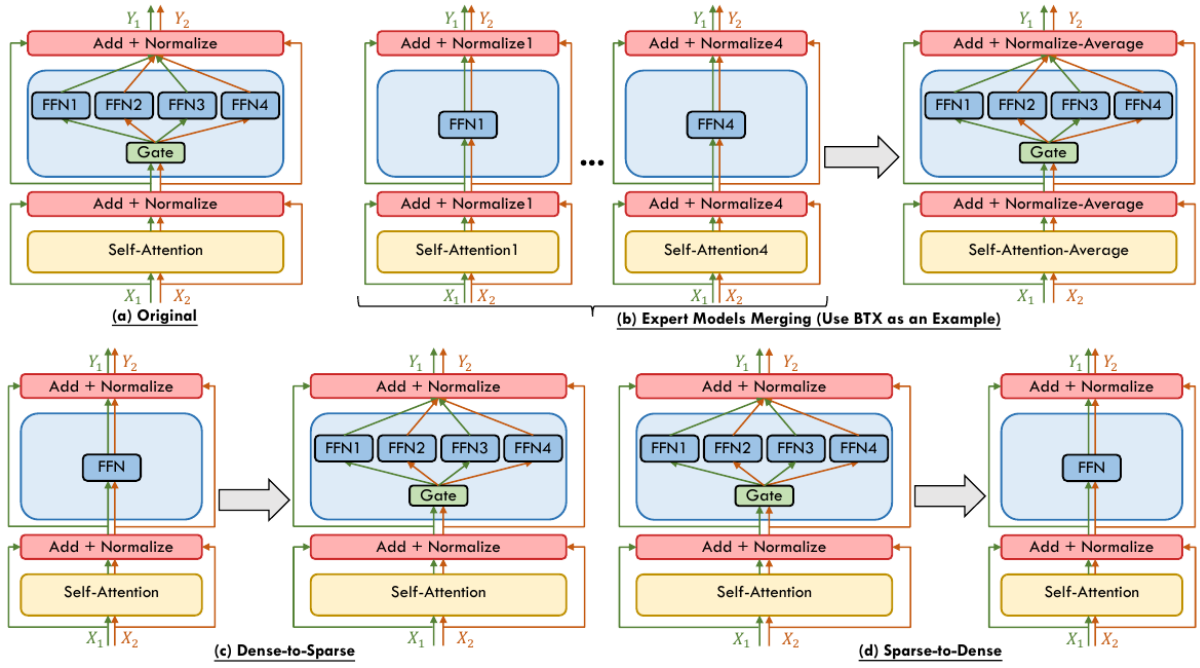


Figure 5.9: Schematic representation of training and inference schemes related to MoE. It provides an abstracted view of model transition, without focusing specific model states during training or inference. Subfigure (a) depicts the original scheme without architectural transformation. Subfigure (b) depicts the merging of distinct expert models, exemplified by BTX [190]. Subfigure (c) depicts the transition from a dense model to a sparse model. Subfigure (d) depicts the inverse process, where a sparse model is converted to a dense model [19].

A representative framework of this strategy is EvoMoE [148], which progressively transitions from dense to sparse routing via the dense-to-sparse gate. Initially, all experts are activated; over time, routing becomes increasingly selective. A similar strategy is employed in MoE-LLaVA [123], which starts from a dense model and fine-tunes only the MoE layers.

A widely adopted approach is sparse upcycling [107], which initializes an MoE model from a pretrained dense checkpoint. In this process, pretrained FFN layers are replicated as experts, while the gating network is randomly initialized. To solve the low performance when we start fine-tuning, gating outputs are normalized such that expert contributions sum to one. While this normalization is beneficial for upcycling vision models, it is not effective at improving the performance of upcycled language models. Initially, in sparse MoE approaches the softmax function was applied to the selected top- k gating values, which is formulated as $\text{softmax}(\text{TopK}(g(x; \Theta), k))$. However, subsequent MoEs opted for applying the softmax function across all potential gating values before selecting the top- k subset, formulated as $\text{TopK}(\text{softmax}(g(x; \Theta)), k)$.

Building on this, Skywork-MoE [207] shows that while upcycling improves efficiency, training from scratch often yields better performance. It also introduces expert similarity as a diagnostic measure, observing that lower similarity between experts correlates with better performance due to increased specialization.

Hybrid approaches further combine dense and sparse computation. For example, DS-MoE [154] employs dense training, where all experts are activated, while using sparse activation during inference. This design reflects the observation that attention layers exhibit less sparsity than FFNs. Other works focus on constructing experts from dense FFNs through strategies such as random partitioning [160], clustering [233], or importance-based splitting [247]. Methods such as SMOE-Dropout [28] gradually increase the number of active experts during training to prevent overfitting.

Sparse-to-dense methods aim to improve MoE performance by converting MoE models into dense structures. One common approach is knowledge distillation, where a dense student model learns from a sparse MoE teacher. Switch Transformer [62] demonstrates that such distillation can preserve a significant portion of performance gains while drastically reducing weight count, especially when combining teacher predictions with ground-truth labels. Similarly, OneS [218] introduces a knowledge integration framework that aggregates expert knowledge through methods such as averaging or singular value decomposition, followed by distillation.

This approach retains a substantial portion of MoE performance while producing a compact dense model.

Another direction focuses on expert pruning. Since only a subset of experts contributes significantly during inference, removing underutilized experts can simplify the model. For instance, [133] progressively eliminates redundant experts, transforming the model into a single-expert structure tailored to the target task. Other methods, such as ModuleFormer [181], apply task-aware pruning, while vision-specific approaches [90] average expert weights after training to revert MoE layers back to dense FFNs. More generally, compression frameworks [80] combine pruning, quantization, and structural simplification to optimize MoE deployment.

Another emerging paradigm constructs MoE models by **merging expert models** that were independently trained. The Branch-Train-Merge (BTM) framework [118] trains multiple domain-specific expert language models separately and combines them at inference through ensembling or weight averaging. Extending this idea, Branch-Train-MiX (BTX) [190] integrates these pretrained models into a unified MoE architecture, where FFN layers are treated as experts and combined via a gating mechanism. Other components, such as attention layers, are merged through weight averaging, followed by joint fine-tuning.

More broadly, Fusion of Experts (FoE) [201] addresses the integration of multiple expert models as a supervised learning problem, enabling the combination of complementary knowledge across heterogeneous data distributions. Overall, training and inference schemes play a crucial role in unlocking the full potential of MoE models. As summarized in Figure 5.9, the transition between dense and sparse representations, and the ability to merge independently trained experts, provides a flexible framework for balancing performance, efficiency, and deployability.

5.3.5 Mixture of Experts Derivatives

Building upon the core algorithmic principles of mixture-of-experts, a wide range of extensions and variants, referred to here as MoE derivatives, have been proposed. These approaches adapt the fundamental ideas of expert specialization and conditional computation to new architectures, training paradigms, and application domains.

One line of work focuses on architectural extensions of the standard transformer. For example, WideNet [219] increases model capacity by replacing feed-forward network (FFN) layers with MoE modules while maintaining weight sharing across layers, except for normalization components. Similarly, the Sparse Universal Transformer (SUT) [192] introduces sparse MoE into the Universal Transformer framework, combining weight sharing with a stick-breaking-based dynamic halting mechanism to reduce computation without sacrificing generalization.

Another direction addresses limitations of conventional MoE routing, particularly the reliance on discrete token-expert assignments. Such discrete matching [177] [114] [56] [62] [246] [243] [8] can lead to training instability and imbalanced expert utilization. To mitigate this, the authors of [7] proposed the mixture of tokens (MoT), which blends tokens from different examples before routing them to experts. This approach increases the diversity of token-expert interactions and improves learning stability.

The MoE paradigm has also inspired innovations in weight-efficient fine-tuning (PEFT). For instance, sparse mixture-of-prompts (SMoP) [33] applies the gating mechanism to prompt tuning, learning multiple specialized soft prompts instead of a single long prompt. This reduces inefficiencies and enhances task adaptability. In addition, MoE has been incorporated into lifelong learning (LLL) settings. Lifelong-MoE [29] dynamically expands the model by introducing new experts as new tasks are encountered, while employing regularized pretraining to mitigate catastrophic forgetting. This design allows continuous adaptation without degrading performance on previously learned tasks.

More recently, the concept of conditional computation has been generalized beyond expert selection to other dimensions of model execution. For example, mixture-of-depths (MoD) [166] introduces a gating mechanism that decides whether each token should be processed by a given transformer layer. This enables adaptive allocation of computation across sequence positions, reducing overall FLOPs compared to both standard transformers and MoE-based architectures.

The evolution of MoE derivatives reflects two key trends: (i) generalization of conditional computation,

extending gating mechanisms beyond expert selection to dimensions such as tokens, layers, or prompts; (ii) integration with diverse paradigms, including weight-efficient fine-tuning, lifelong learning, and adaptive computation. These developments demonstrate the flexibility and adaptability of the MoE framework, highlighting its potential as a unifying principle for scalable and efficient model design across a wide range of domains.

5.4 Design of Mixture of Experts

While mixture-of-experts architectures significantly improve the scalability of large language models, they introduce new challenges for training and inference systems due to their sparse and dynamic execution patterns. In particular, the conditional activation of experts leads to irregular computation, communication, and memory access patterns that differ substantially from dense models.

To address these challenges, **expert parallelism**, first introduced in GShard [114], has become a foundational strategy for scaling MoE models. In this paradigm, experts are distributed across different devices, while non-expert components are replicated. As illustrated in Figure 5.10 (a), the execution pipeline involves: gate routing, input encoding, All-to-All dispatch, expert computation, All-to-All combine, and output decoding. Input encoding consolidates tokens assigned to the same expert into contiguous memory regions, enabling efficient execution of general matrix multiplications. The subsequent All-to-All communication distributes tokens to the appropriate devices for expert computation, after which results are recombined.

Modern systems further combine expert parallelism with other parallelization strategies, such as data, tensor, pipeline, and sequence parallelism, to improve scalability. Figure 5.10 illustrates several hybrid configurations, including (b) data + expert + tensor parallelism, (c) data + expert + pipeline parallelism, and (d) expert + tensor parallelism. These hybrid approaches introduce complex trade-offs between computation, communication, and memory usage, requiring careful system-level design. Table 5.4 summarizes representative open-source MoE frameworks and their optimization strategies.

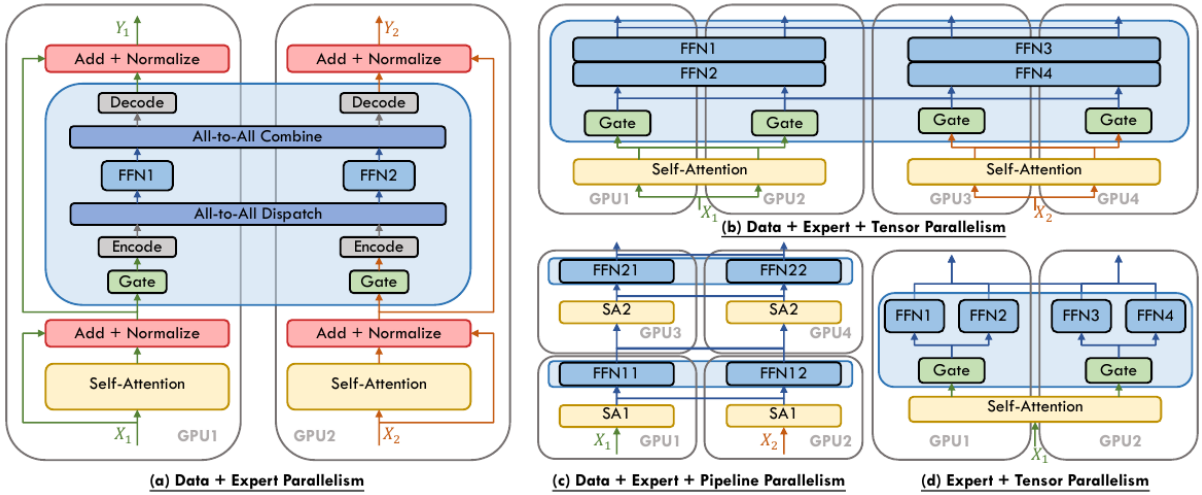


Figure 5.10: Illustration of diverse parallel strategies in MoEs. For clarity and conciseness, the figure omits some all-to-all, AllReduce, and point-to-point communication within parallelism, and Normalization, Encode, Decode, Gate in subfigures (b), (c), and (d) [19].

5.4.1 Computation

Although MoE is designed to scale weights efficiently, it introduces several computational challenges. A key issue is load imbalance across experts, where uneven token distribution leads to synchronization bottlenecks. While algorithmic solutions, that improve gating and capacity constraints, mitigate this issue, system-level approaches also play an important role. Frameworks such as SE-MoE [178], Tutel [140], FlexMoE [149], and SmartMoE [225] implement dynamic expert placement strategies to balance workloads across devices. Additionally, FasterMoE [77] introduces expert replication (shadowing) to alleviate extreme imbalance.

Reference	Affiliation	Optimizations			Link
		Computation	Communication	Storage	
OpenMoE [217]	Colossal-AI	V	V		https://github.com/hpcaitech/ColossalAI
ScatterMoE [191]	Mila Quebec	V			https://github.com/shawntan/scattermoe
Megablocks [65]	Stanford University	V			https://github.com/stanford-futuredata/megablocks
Tutel [140]	Microsoft	V	V		https://github.com/microsoft/tutel
SE-MoE [178]	Baidu	V	V	V	https://github.com/PaddlePaddle/Paddle
HetuMoE [150]	Peking University	V	V		https://github.com/PKU-DAIR/Hetu
Deepspeed-MoE [163]	Microsoft	V	V		https://github.com/microsoft/DeepSpeed
FastMoE [76]	Tsinghua University	V	V		https://github.com/laekov/fastmoe
Fairseq [8], [152]	Meta				https://github.com/facebookresearch/fairseq/tree/moe
Mesh-TensorFlow [176]	Google				https://github.com/tensorflow/mesh

Table 5.4: Comparative overview of the open-source MoE system frameworks, arranged chronologically by reference publication date from newest to oldest [19].

Another challenge arises from the overhead of MoE-specific operations, including gate routing, input encoding, and output decoding. These operations involve frequent memory movement and irregular computation, making them less efficient on hardware optimized for dense workloads. To address this, systems such as DeepSpeed-MoE [163], FastMoE [76], HetuMoE [150], and Tutel [140] provide custom GPU kernels tailored for MoE operations.

Further optimizations focus on efficient handling of dynamic sparsity. For example, MegaBlocks [65] reformulates MoE as block-sparse matrix operations, enabling efficient GPU execution without dropping tokens. Similarly, PIT [240] introduces a compiler-based approach that converts sparse computations into dense tiling patterns using permutation-invariant transformations. However, these approaches may introduce additional data reformatting overhead. To address this, ScatterMoE [191] eliminates costly data copying by directly operating on scattered data using grouped matrix operations, thereby reducing memory footprint and improving extensibility beyond FFN-based experts.

5.4.2 Communication

Communication is often the primary bottleneck in MoE systems due to frequent All-to-All data transfers. Each MoE layer typically requires multiple communication rounds during forward and backward passes, significantly increasing latency. This challenge is amplified in distributed environments where communication spans both intra-node (for example, using NVLink) and inter-node (for example, using Ethernet or InfiniBand) connections. Load imbalance further exacerbates delays by causing synchronization stalls. To mitigate these issues, several optimizations have been proposed.

- Hierarchical all-to-all communication, adopted by DeepSpeed-MoE [163], HetuMoE [150], and ScheMoE [183], prioritizes high-bandwidth intra-node transfers while minimizing inter-node communication.
- Topology-aware routing, used by FasterMoE [77], SE-MoE [178], and TA-MoE [24], reduces cross-node communication by aligning routing decisions with the hardware topology.
- Expert placement optimization opts for strategically distributing experts in such a way that minimizes the communication overhead [163] [207] [185].

Another important technique is communication-computation overlap via pipelining. Systems such as Tutel [140], PipeMoE [30], and MPipeMoE [235] overlap All-to-All communication with expert computation. However, the overlap is often limited by inherent dependencies. To further extend the overlap, Lancet [100] incorporates non-MoE computations into the pipeline, while ScMoE [18] restructures computation to decouple dependencies, enabling substantial overlap. Similar ideas are adopted in hybrid architectures like Snowflake Arctic [186].

5.4.3 Storage

The large number of weights in MoE models places significant pressure on memory capacity, particularly in resource-constrained environments. Although expert parallelism distributes weights across devices, a single

device may still need to manage multiple experts. This issue is especially critical during inference on edge devices with limited memory.

To address this problem, several approaches leverage hierarchical storage: (i) systems such as SE-MoE [178], Pre-gated MoE [91], and EdgeMoE [222] store only active experts in GPU high-bandwidth memory (HBM), while offloading inactive experts to CPU memory or storage, (ii) these approaches incorporate prefetching and expert selection forecasting to overlap data transfer with computation and reduce latency. Additionally, memory optimization techniques focus on reducing intermediate storage overhead. For example, MPipeMoE [235] shares activation buffers across different tensor partitions and uses recomputation and CPU offloading to reduce memory usage during backpropagation.

5.5 Routing Techniques

The performance of a Mixture of Experts model depends heavily on its gating/routing mechanism. This component is responsible for directing each input token to a small subset of experts. The quality of this routing decision has a direct impact on model performance, training stability, and computational efficiency. While a standard Top-k router is a common starting point, it can lead to problems like load imbalance, where some experts are consistently over-selected while others remain under-utilized.

In this section we cover a set of routing technique thought to address the limitations of basic gating: noisy Top-k gating, the Switch Transformer strategy, hash-based routing, and Soft MoE. We will examine the trade-offs between different routing algorithms, considering their computational cost, effect on load balancing, and impact on expert specialization.

The common mechanism used for routing tokens in a sparse mixture of experts model is Top-k Gating. It is a simple and effective approach adopted by many successful MoE architectures. The gating network is a simple Linear layer that produces a logit for each expert, representing a preference for that expert to process the current input token. The TopK function then selects the k experts with the highest logit scores.

5.5.1 Top-k Gating

For a given input token representation x , the process begins with the gating network, parameterized by a weight matrix W_g , which computes the logits $h(x)$ using $h(x) = W_g \cdot x$. Here, $h(x)$ is a vector where each element $h(x)_i$ corresponds to the logit for expert i . The TopK operation selects the indices of the k highest values in $h(x)$. Let us call this set of indices S .

The final gating scores $G(x)_i$ can be calculated by applying a softmax function only to the selected logits (Equation 5.24). This ensures the weights for the chosen experts sum to one.

$$G(x)_i = \begin{cases} \frac{e^{h(x)_i}}{\sum_{j \in S} e^{h(x)_j}} & \text{if } i \in S \\ 0 & \text{if } i \notin S \end{cases} \quad (5.24)$$

The output y of the MoE layer is then the weighted sum of the outputs $E_i(x)$ from the selected experts, as indicated in Equation 5.25.

$$y = \sum_{i \in S} G(x)_i \cdot E_i(x) \quad (5.25)$$

The value of k is a relevant design choice. Using $k = 2$ allows each token to be processed by two experts, providing a richer representation than routing to a single expert. This can improve model quality at the cost of increased computation. In contrast, setting $k = 1$, as is the case in the Switch Transformers, minimizes computational and communication overhead.

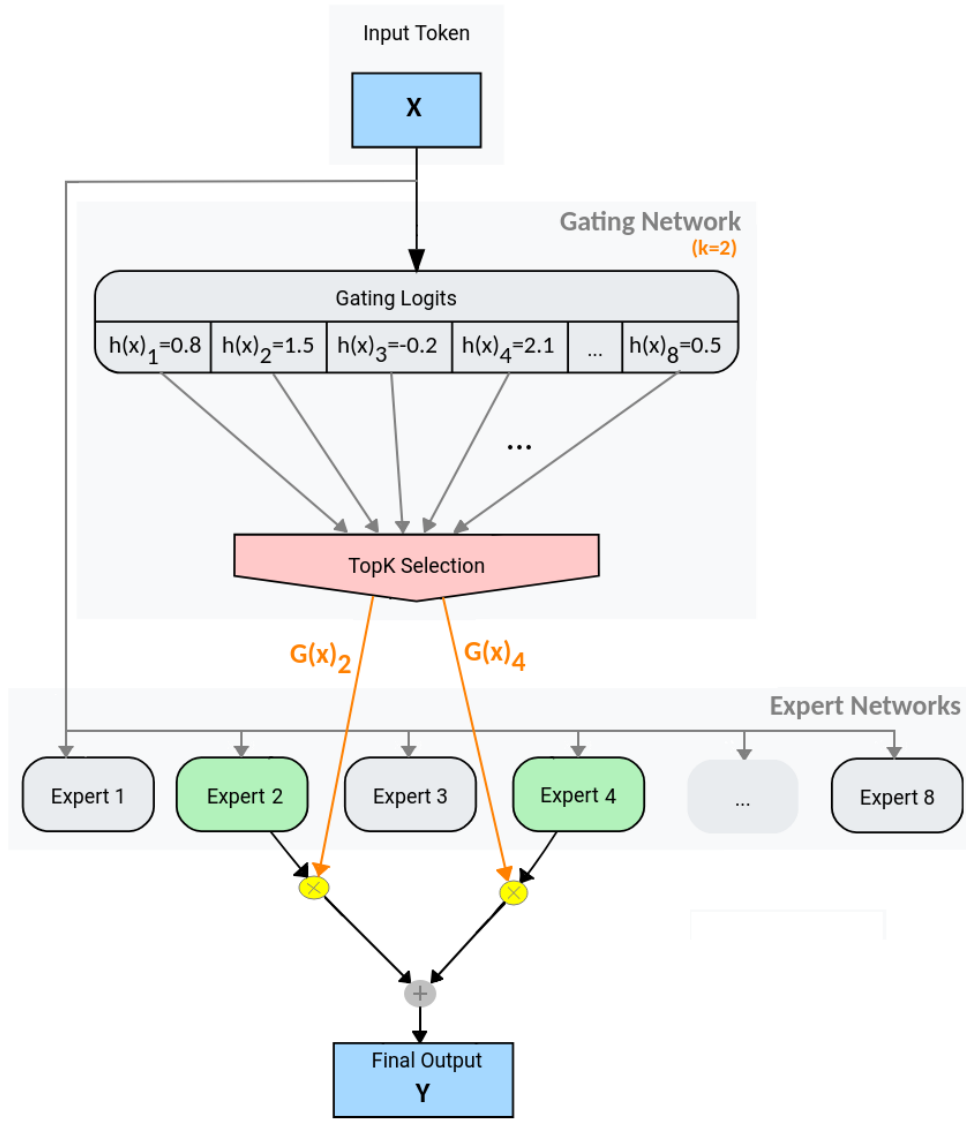


Figure 5.11: The gating process for a single token. The gating network computes logits, and the TopK function selects the two experts with the highest scores, experts 2 and 4, to process the token [113].

While functionally simple, Top-k gating has a significant operational weakness: it can lead to severe load imbalance. During training, the gating network may learn to favor a small subset of experts, while others remain underutilized or are rarely selected. This phenomenon arises because the gating network is a learned function, and without constraints, it will optimize for predictive accuracy, which may involve repeatedly using the same few experts that prove most effective in the early stages of training. This imbalance introduces two major problems: (i) training inefficiency, where the rarely selected experts receive few training signals, causing their weights to be poorly optimized. The model effectively fails to use its full capacity, as a large fraction of its weights contribute very little to its performance, (ii) computational waste, since experts are often distributed across multiple GPUs, if one expert is assigned an excessive number of tokens, the associated device gets overloaded, while devices hosting rarely used experts are kept mostly idle.

To manage the flow of tokens and prevent any single expert from being overwhelmed, MoE implementations introduce a capacity factor. This factor helps defining the capacity C of the buffer that hosts the tokens each expert can process in a minibatch. The capacity C of an expert is defined by Equation 5.26.

$$C = \text{capacity factor} \cdot \frac{\text{tokens in minibatch}}{\text{number of experts}} \quad (5.26)$$

A capacity factor greater than 1.0 provides a buffer for minor imbalances. For example, a value of 1.25 allows each expert to process up to 25% more tokens than the average. However, if a popular expert receives more

tokens than its capacity C , the tokens in excess are skipped. These skipped tokens do not get processed by any expert. Instead, they bypass the MoE layer entirely and are passed through the residual connection. This is a form of information loss and can degrade model performance, as the model is unable to apply specialized computation to those tokens. Finding the adequate capacity factor is a trade-off, a higher value reduces the number of dropped tokens but increases memory allocation and can lead to more wasted computation if the load is balanced. The primary mechanism for combating load imbalance in a standard Top-k gating is the auxiliary load balancing loss (Equation 5.5).

5.5.2 Noisy Top-k Gating

The standard top-k gating makes routing decisions deterministically, meaning that it selects always the same experts to process a given token. This can lead to repetitive routing patterns during training. Certain experts may consistently receive the highest scores from the gating network, attracting the majority of tokens, while others are starved of data. This imbalance hinders the model’s ability to develop a diverse set of specialized experts and can even cause some experts to collapse, contributing nothing to the final output.

To mitigate this issue, noisy top-k gating adds a small amount of random noise to the gating network’s output logits before selecting the top-k experts. In this way, it constitutes a form of regularization that encourages the model to explore different routing paths and prevents it from relying too often on a small subset of experts. The random value added to the logits can be sampled from a Gaussian distribution.

The mathematical formulation derives from the standard gating mechanism. Let $h(x)$ represent the vector of logits produced by the gating network for an input token x , where $h(x) = x \cdot W_g$ and W_g are the gating network’s weights. In noisy gating, we calculate a modified set of logits $h_{noisy}(x)$, as defined in Equation 5.27.

$$h_{noisy}(x) = h(x) + Noise \quad (5.27)$$

The noise term is usually drawn from a standard Gaussian distribution with zero mean and a tunable standard deviation. The implementation proposed in [177] scales the noise using a learnable weight matrix W_{noise} (Equation 5.28).

$$Noise = \text{StandardNormal}() \cdot \text{softplus}(x \cdot W_{noise}) \quad (5.28)$$

The `StandardNormal` generates random values from $\mathcal{N}(0, 1)$, and the `softplus` function ensures the scaling factor is always positive. The top-k selection is then performed on these noisy logits. This process is only active during training. During inference, the noise is disabled to ensure deterministic and stable outputs.

For example, considering that we have $N = 4$ experts and Top-k gating produces as logits $[5.1, 2.3, 3.7, 3.2]$, the selected experts will be the first and third ones. When using the noisy Top-k gating, the logits after adding noise could be $[5.2, 2.1, 3.3, 3.6]$, and the selected experts would be the first and fourth ones.

The primary benefit of adding noise is load balancing the tokens assigned to the different experts. This prevents any single expert from becoming a bottleneck and ensures all experts receive a sufficient variety of training examples to learn meaningful specializations. This mechanism complements the auxiliary load balancing loss. While the loss function penalizes imbalance, noisy gating proactively discourages it from forming in the first place. The result is often a more stable training process with smoother loss curves and a lower likelihood of expert collapse.

The implementation of the noisy top-k gating is straightforward, as illustrated in Listing 5.1. It uses two `Linear` layers, one for obtaining the regular logits and the other for learning the noise standard deviation. In the rest, it follows Equations 5.27 and 5.28. The use of a learnable W_{noise} matrix allows the model to adapt the noise level on a per-token basis, which is generally more effective than using a fixed noise level for the entire training.

```

1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4
5 class NoisyTopkGating(nn.Module):
6     def __init__(self, d_model, num_experts, top_k):
7         super().__init__()
8         self.top_k = top_k
9         self.gate_linear = nn.Linear(d_model, num_experts) # logits layer
10        self.noise_linear = nn.Linear(d_model, num_experts) # learnable noise stddev layer
11
12    def forward(self, x):
13        # x shape: (batch_size * sequence_length, d_model)
14        logits = self.gate_linear(x)
15
16        # Add noise only during training
17        if self.training:
18            noise_std = self.noise_linear(x)
19            noise_std = F.softplus(noise_std) # Make the noise stddev positive
20            noisy_logits = logits + (torch.randn_like(logits) * noise_std)
21        else:
22            noisy_logits = logits
23
24        # Select the top-k experts
25        top_k_logits, indices = torch.topk(noisy_logits, self.top_k, dim=-1)
26
27        # Create a sparse routing mask
28        zeros = torch.full_like(noisy_logits, float('-inf'))
29        # Insert the selected top-k logits in the respective positions of tensor 'zeros'
30        sparse_logits = zeros.scatter(-1, indices, top_k_logits)
31        router_output = F.softmax(sparse_logits, dim=-1)
32
33    return router_output, indices

```

Listing 5.1: PyTorch implementation of the noisy top-k gating.

Noisy top-k gating is a simple yet effective technique for improving the stability and performance of MoE models. It encourages exploration in the routing space, leading to better load distribution and more specialized expert performance, without incurring in a significant computational overhead.

5.5.3 Hash-based Gating

While learned gating networks offer dynamic routing, they bring additional complexity, including trainable weights, potential instability, and the need for auxiliary loss functions. An alternative approach sidesteps these issues by removing the learning process from the router. Hash-based routing is a deterministic method that assigns tokens to experts using a fixed, non-learned function, trading the potential for intelligent specialization for simplicity and stability.

Hash-based routing is simple. Instead of passing a token's representation through a Linear layer to compute routing logits, we apply an hash function to some property of the token. The resulting hash value is then mapped to an expert index using the modulo operator. The process for a single token x is the following.

- Select a feature: Choose a feature of the token to hash. This is often the token's unique ID from the vocabulary, but could also be a combination of its ID and position in the sequence.
- Apply the hash function: Compute the hash value for the selected feature. Let us denote the hash function as H .
- Determine expert index: We use the modulo operator to obtain the remainder of the division of the hash value by the number of experts, N , to obtain the index E_i of the assigned expert (Equation 5.29).

$$E_i = H(\text{token feature}) \% N \quad (5.29)$$

The following sequence summarizes the previous steps.

Input token $x \rightarrow$ [Hash function $H(x)$] $\rightarrow$ hash value $\rightarrow$ [hash value % number experts] $\rightarrow$ Expert index

The hash function and the modulo operation are easy to compute and require no trainable weights. The same input token feature will always map to the same expert, making the routing decision static and predictable throughout training and inference.

Opting for a non-learned router might seem counterintuitive, as it discards the model’s ability to learn intelligent routing patterns. However, this approach offers significant engineering advantages. The most obvious benefit is the complete elimination of the gating network. This means: (i) the router has no weights, the model becomes slightly smaller and consumes less memory, as there are no weights or biases associated with the router, (ii) no router computation, the forward pass avoids the matrix multiplication required by a standard gating network, replacing it with a much faster hash and modulo operation.

A hash function carefully chosen will distribute the tokens uniformly across the available experts. This statistical uniformity means that, over a large batch of tokens, each expert is expected to receive a similar number of tokens. This property solves the load imbalance problem associated with learnable routers. Consequently, the auxiliary load-balancing loss is no longer necessary. The total loss function simplifies to just the primary task loss, for example the cross-entropy loss, $\mathcal{L}_{total} = \mathcal{L}_{CE}$. By removing the auxiliary loss and its associated weight, we also eliminate the training instability associated with its calculation, and so the z-loss is no longer necessary.

The primary drawback of hash-based routing is its impact on model performance. Learned gating allows the model to develop semantic specialization. For example, one expert might become specialized at handling punctuation and grammar, while another focuses on scientific terminology. The router learns to send the right token to the right specialist. Since token assignment is pseudo-random, each expert is forced to be a generalist. It must learn to process a random subset of the entire data distribution, preventing the emergence of fine-grained specialization. This typically results in lower model quality, measured by an higher perplexity or a lower accuracy, compared to a sparse MoE model with a learned router.

Figure 5.12 shows that hash-based MoE provides a better performance (lower perplexity) than a dense model, but it generally underperforms a standard MoE with learned routing.

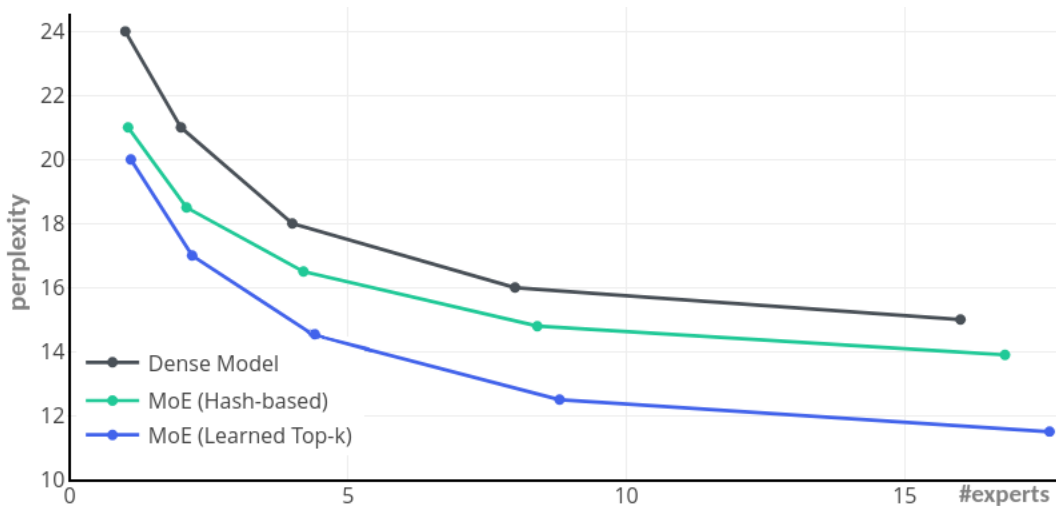


Figure 5.12: Comparing the perplexity of a dense model, a learned MoE, and an hash-based MoE. Hash-based routing offers an intermediate performance, improving on dense models but does not match the performance of learned, specialized routing [113].

In practice, hash-based routing is often used as the experimental baseline. It helps identifying the performance gains that come from learned, sparse activation versus those that come simply from having more weights. If a complex, learned MoE model cannot outperform a hash-based equivalent, it often indicates problems with the training setup or the routing mechanism itself.

5.5.4 Switch Transformer Gating

Routing tokens to multiple experts, $k > 1$, is a common method to combine specialized knowledge, yet it introduces significant computational and communication overhead. This approach requires each token to be processed by multiple experts, and that their outputs are weighted and combined. The Switch Transformer architecture [62] presents a radical simplification to this process, each token is routed to only one expert.

This top-1 routing, $k = 1$, dramatically simplifies the MoE layer. Instead of computing a weighted sum of outputs from several experts, the layer's output for a given token is simply the output of a single expert, scaled by the router's gating score.

The core of the Switch Transformer is its routing mechanism. For an input token representation x , the gating network computes a distribution over all N experts. However, instead of selecting the top- k experts, it only selects the expert with the highest score (Equation 5.30).

$$y = G(x) \cdot E_{\arg \max_i G(x)_i}(x) \quad (5.30)$$

Where $G(x)$ is the vector of gating scores, and E_i is the i -th expert network. This design choice has the following advantages.

- Reduced computation: The forward pass for one token only involves activating one expert, not k experts.
- Simplified communication: There is no need to aggregate outputs from multiple experts, which reduces communication bandwidth in distributed settings.
- Architectural simplicity: The implementation is more straightforward compared to managing multiple assignments per token.

Figure 5.13 illustrates the difference between a standard MoE with $k = 2$ and a Switch layer with $k = 1$. In the standard model, each token is processed by two experts. In the Switch model, each token is dispatched to only one.

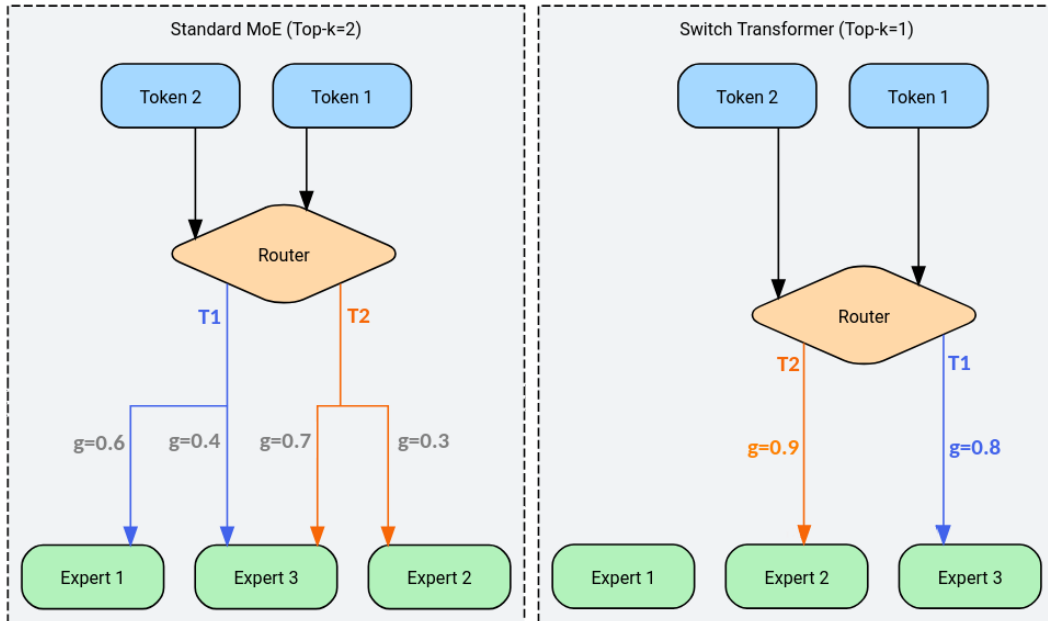


Figure 5.13: Comparison of routing strategies for two tokens. The standard MoE sends each token to the top two experts, whereas the Switch Transformer sends each token to only the top expert [113].

A top-1 routing strategy might appear to worsen the problem of load imbalance. If the router has a strong preference, it could consistently select the same expert, leaving others unused. To counteract this, Switch Transformers use a refined version of the auxiliary load-balancing loss.

The goal remains the same, to encourage the router to distribute tokens uniformly across all available experts. The loss is calculated over a minibatch of T tokens and considering N experts. It is the dot product of two vectors: (i) the fraction of tokens dispatched to each expert i (f_i), a vector that measures the portion of the minibatch each expert receives, and (ii) the fraction of the router probability allocated to each expert i (P_i), a vector that measures the average probability of the router assigning all tokens in the minibatch to each expert i . The auxiliary loss, $\mathcal{L}_{aux}$, is defined in Equation 5.31.

$$\mathcal{L}_{aux} = \alpha \cdot N \sum_{i=1}^N f_i \cdot P_i \quad (5.31)$$

Where $f_i = \frac{1}{T} \sum_{x \in \text{minibatch}} \mathbb{I}(\arg \max G(x) = i)$ is the fraction of tokens in the minibatch assigned to expert i , $P_i = \frac{1}{T} \sum_{x \in \text{minibatch}} p_i(x)$ is the average router probability for expert i over the minibatch, $p_i(x)$ is the `softmax`'s output relative to expert i when the token x is being assigned, α is a tunable hyperparameter that controls the weight of the auxiliary loss. In [62], the authors found that values around 0.01 are a good choice.

Multiplying f_i and P_i together encourages the router's probability distribution P to align with the actual token dispatch distribution f . This encourages the router to spread its probability mass more evenly, which in turn leads to a more balanced load.

The simplification of using $k = 1$ routing introduces an implementation detail in expert capacity. In a distributed setup, each expert, residing on a specific device, is allocated a static buffer to handle a certain number of tokens per batch. The size of this buffer is determined by the capacity factor C (Equation 5.32).

$$\text{Expert Capacity} = \left(\frac{\text{tokens per minibatch}}{\text{number experts}} \right) \cdot C \quad (5.32)$$

An perfectly balanced router would send exactly $\frac{T}{N}$ tokens to each expert. The capacity factor can be used to introduce some statistical variance in the allocation of tokens to experts. For example, a capacity factor of 1.25 means each expert can handle 25% more tokens than the average. When an expert capacity is exceeded, the Switch architecture introduced another simplification: the token is dropped. It is not processed by any expert. Instead, its representation from the residual connection is passed directly to the next layer. This is equivalent to the token passing through an identity function within the MoE layer. While dropping tokens seems detrimental, experiments show that with a well-tuned capacity factor and auxiliary loss, the percentage of dropped tokens is less than 1% and does not significantly harm overall model performance.

The design of Switch Transformers yields a compelling trade-off between model size and computational cost. By using a large number of experts, the total weight count in the model can be massive, but because only one expert is activated per token, the training and inference FLOPs remain comparable to a much smaller dense model (Figure 5.14).

The authors of Switch Transformer also found that training stability is an issue. Using lower-precision formats like Bfloat16 was important, especially for the router's `softmax` computation, to prevent instabilities caused by large logit values. In conclusion, the Switch Transformer presents an efficient and scalable architecture. The advantages are a high computational efficiency, simplified routing logic, and reduced communication needs. The disadvantages are the potential for dropping tokens when the capacity is exceeded, and an increased tendency for training instability, requiring careful tuning of the auxiliary loss and the capacity factor.

5.5.5 Soft MoE

Routing mechanisms as the top-k gating and switch transformer gating do a hard assignment of tokens to experts. A token is routed to a discrete, small set of experts, while all other experts are ignored for that specific token's computation. This hard selection is the source of the computational savings in sparse models, but it also introduces challenges like non-differentiability and the need for auxiliary load-balancing losses. Soft MoE

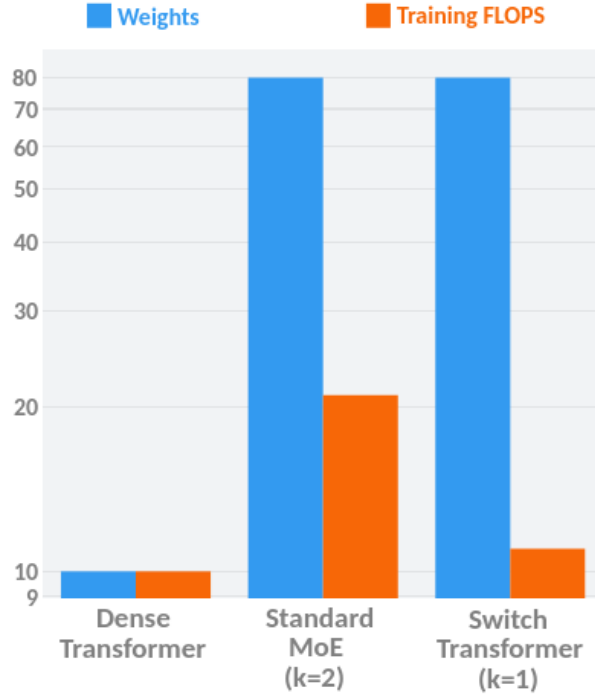


Figure 5.14: Comparison of models in terms of size and required computation. Both MoE models have $8\times$ the weights of the dense model. The standard MoE ($k = 2$) more than doubles the FLOP (21 vs. 10), while the Switch Transformer ($k = 1$) adds only a small computational overhead (11 vs. 10) [113].

opted for a different approach that replaces this discrete selection by a soft and weighted combination of all experts. Instead of choosing which experts to use, the gating network determines a weight for every expert, and the final output is a weighted sum of the outputs from all experts. This makes the entire MoE layer fully differentiable and elegantly sidesteps the training instabilities associated with hard gating.

In a Soft MoE layer, the gating network operates similarly to a standard router by producing a logit for each expert. However, instead of using these logits to select the top- k experts, it applies a `softmax` function to all of them. This converts the logits into a set of positive weights that sum to one, effectively forming a probability distribution over the experts.

The final output for an input token x is not the output of a few selected experts, but a linear combination of the outputs from all N experts. The contribution of each expert $E_i(x)$ is scaled by its corresponding weight w_i , produced by the `softmax`. Given an input x , the gating network G computes a vector of logits $h(x)$. The weights w are then calculated with Equation 5.33.

$$w = \text{softmax}(h(x)) \quad (5.33)$$

The output y of the Soft MoE layer is the weighted sum defined in Equation 5.34.

$$y = \sum_{i=1}^N w_i \cdot E_i(x) \quad (5.34)$$

The diagrams included in Figure 5.15 illustrate the difference between the data flow in a hard-gating MoE and a Soft MoE.

The primary advantage of Soft MoE is its elegant solution to the training challenges of sparse models. Other advantages are (i) being full differentiable, since the entire layer, including the gating module, is a continuous function. This allows gradients to flow smoothly through the entire network during backpropagation, and (ii)

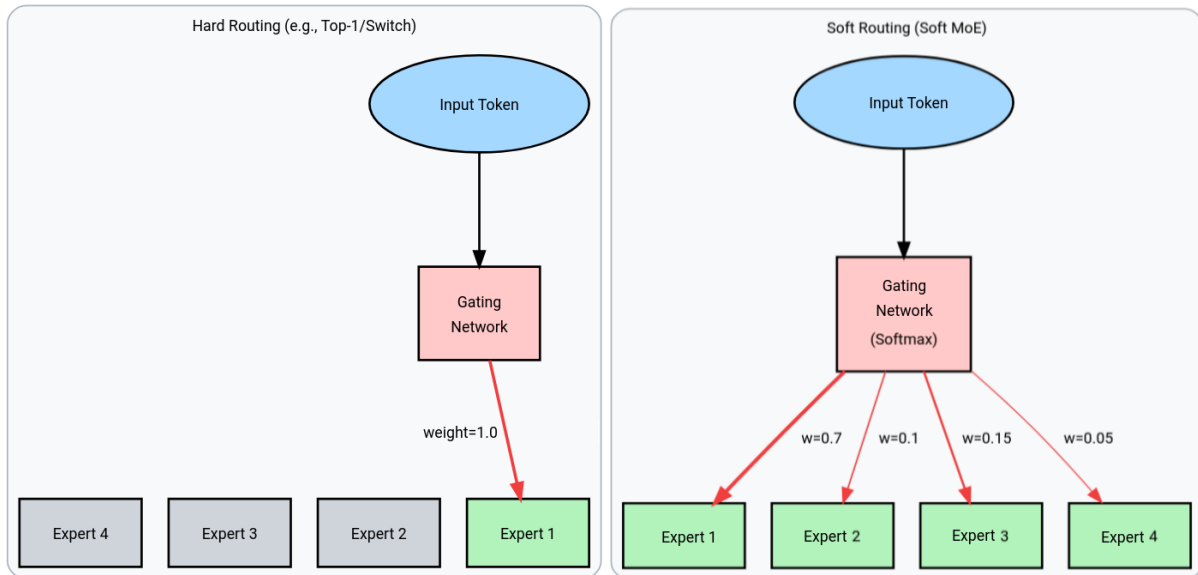


Figure 5.15: In hard routing (left), the gating network selects a discrete expert (Expert 1), and all computation flows through it; the other experts remain inactive for this token. In Soft MoE (right), the gating network computes a weight for every expert; the final output is a weighted combination of all expert outputs [113].

implements load balancing implicitly because every expert receives a non-zero weight and thus every expert participates in the forward and backward passes for every token. This naturally prevents the expert collapse phenomenon, where some experts receive no training signals, and it is not necessary an explicit auxiliary loss to force load balancing.

However, this elegance brings two significant and often prohibitive costs: (i) a large computation cost, which is the opposite of the fundamental benefit of a sparse MoE, reducing the computation cost. In a sparse MoE, only a fraction of the model’s weights are active for any given input, making the computation cost (FLOP) not proportional to model size. Soft MoE completely eliminates this advantage. Since every token is processed by every expert, the computation cost is equivalent to running an ensemble of N models and then averaging their outputs. This makes it as expensive as a dense model with a weight count equal to the sum of all experts; (ii) a much larger memory space; in a sparse model, only the active experts’ weights need to be stored in high-speed memory, such as GPU HBM. In Soft MoE, all expert weights must be loaded simultaneously in memory, which is impractical for models with a large number of experts.

Given the high computation cost and the large memory space required, a pure Soft MoE is rarely used in large-scale language models where computational efficiency is a primary objective. Its formulation serves more as a theoretical benchmark and an analytic tool. However, the idea of soft, fully differentiable routing influenced the design of more practical, hybrid systems. For example, we can use top- k gating to select a small subset of experts and then apply a process each token with a soft, weighted combination within the selected subset. This brings some of the training stability of soft gating while keeping most of the computational benefits of sparse MoE.

Soft MoE, with its mathematical simplicity in training, provides a counterpoint to the computational sparsity necessary for scalability. In other words, Soft MoE represents one extreme of the MoE architecture spectrum, where training stability is maximized at the expense of inference efficiency. Meanwhile, Switch Transformer occupies the other extreme, prioritizing computational efficiency over training stability.

5.6 Load Balancing

Load imbalance in MoE systems manifests in multiple dimensions. During training, some experts may become overloaded while others remain underutilized, leading to inefficiencies and routing collapse where the model repeatedly selects a few experts. This phenomenon occurs because in a normal MoE training, the gating network converges to mostly activate the same few experts, which self-reinforces as favored experts are trained quicker

and hence selected more. The consequences are multifaceted: computational resources are wasted on idle experts, communication bottlenecks emerge around popular experts, and model capacity is underutilized [228].

The traditional approach to mitigate this imbalance, as we saw before, is adding an auxiliary loss term to the training loss [177]. This auxiliary term encourage the gating network to give all experts equal importance, ensuring that all experts receive a roughly equal number of training samples. However, this approach brings some complications. The auxiliary loss term introduces interference gradients during training, which conflict with the primary loss objective, making it difficult to balance load and achieve high levels of accuracy. In this section we dive deeper in load balancing and present two strategies: expert capability management and adaptive expert placement.

Expert capability management

The main challenge in a MoE lies in managing computational resources when there is a great and unpredicted diversity in expert popularity during training. Early MoE implementations often suffered from either catastrophic memory overflow when popular experts received too many tokens, or significant computational waste due to conservative over-provisioning. Traditional capacity management approaches relied on static allocation strategies that failed to adapt to the dynamic nature of expert selection patterns, leading to either token dropping, which degraded the model performance, or excessive memory consumption, which limited model scalability.

Expert capacity management represents one of the most critical aspects of MoE design, since it impacts both training stability and computational efficiency. The concept was first systematically introduced in GShard citelepikhin2020gshard and further refined in Switch Transformer [62], establishing the theoretical and practical foundations for managing token overflow in sparse expert gating. The Expert Capacity EC is defined in Equation 5.35.

$$EC = \frac{B \cdot CF \cdot K}{N} \quad (5.35)$$

where B is the minibatch size, CF is the capacity factor that determines the balance between computational efficiency and token utilization [62], K is the number of selected experts, and N denotes the number of experts. When $CF < 1$, some tokens will always be dropped, whereas when $CF > 1$ an additional capacity is considered to reduce the token dropping occurrence, albeit at the cost of increased memory consumption and potential computational overhead.

The capacity management goes beyond simple token allocation and encompasses sophisticated token drop handling mechanisms. Expert capacity introduces a threshold of how many tokens can be processed by any expert, and when the number of tokens routed to an expert exceeds its capacity, the excess tokens may be dropped. The Switch Transformer [62] introduced an approach where tokens in excess are sent to the next layer via residual connections without being processed, effectively bypassing the MoE layer. This mechanism provides a fail-safe against extreme load imbalances while maintaining the system stability and ensuring that no computational paths are permanently blocked.

Recent advances have introduced more sophisticated capacity management strategies. Vision-MoE (V-MoE) [170] demonstrates that in computer vision applications, where images typically generate many patches, a small pre-defined expert capacity is essential to reduce hardware constraints, though low capacity tends to lead to patches being dropped, creating a critical trade-off between memory efficiency and information preservation. The mathematical formulation of token dropping has been formalized in recent work [23], where the dropping probability $P(drop_i)$ for a given expert i is defined in Equation 5.36.

$$P(drop_i) = P\left(\sum_{j=1}^T [gating(j) = i] > EC_i\right) \quad (5.36)$$

where $gating(\cdot)$ corresponds to the gating network, EC_i is the capacity of i -th expert, and T denotes the number of tokens. This formulation led to the development of adaptive capacity mechanisms that dynamically adjust the expert capacity based on real-time load patterns, representing a significant advancement over static capacity allocation strategies adopted by earlier systems.

Adaptive expert placement

While capacity management balances token allocation, the adaptive expert placement technique focus on solving the runtime imbalance by replicating and replacing experts. As shown in Figure 5.16, the core idea of expert replacement is to identify unpopular and popular experts during training, then applying scheduling methods to adjust the workload in the different devices.

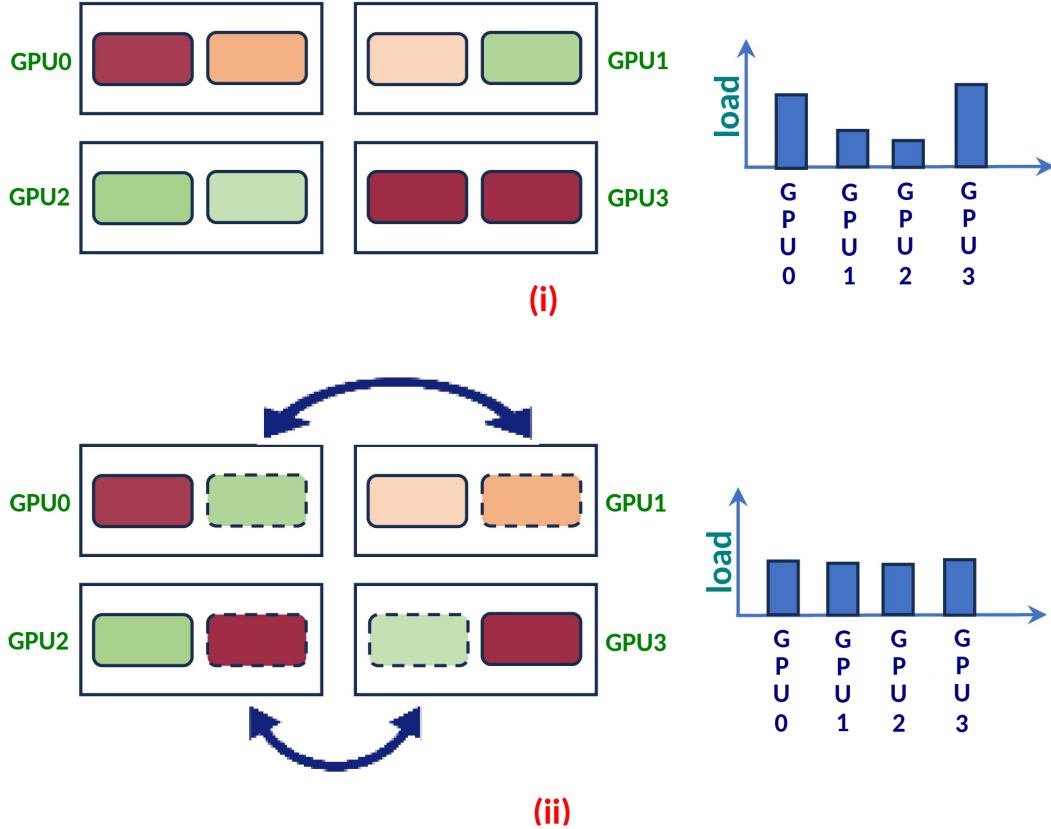


Figure 5.16: Adaptive expert placement. (i) Load imbalance across devices and (ii) load balancing with expert replacement [228].

Traditional load balancing approaches operate reactively, responding to imbalance after it has occurred rather than predicting and preventing it. To improve this situation, FasterMoE [77] created a predictive performance model and a roofline-like analysis to guide dynamic shadowing of popular experts and enforcing a topology-aware expert selection that takes in consideration the network latency. SmartMoE [225] introduces an efficient searching algorithm within an enlarged hybrid parallelism space, achieving higher training throughput over FasterMoE by decomposing optimization opportunities into static offline pools and dynamic online selection.

As the expert replication strategies of FasterMoE follows a binary approach, either replicate experts across all workers or not at all, lead to excessive communication overhead and memory cost. To improve this situation, the Prophet [203] proposal was to replicate popular experts among a subset of workers rather than all workers, reducing communication overhead while maintaining load balance benefits. Building upon this idea, Pro-Prophet [204] decomposes load balancing into a planner that profiles inter-iteration token distributions and a scheduler that maximizes communication-computation overlap, achieving substantial load-balancing enhancement. FlexMoE [149] addresses the large online adjustment space in expert replacement through selective expert replication across GPUs according to the popularity distribution, providing dynamic device placement that adapts to changing expert popularity patterns. HECATE [159] introduces hierarchical expert tree structures that enable multi-level load balancing and efficient expert discovery.

A limitation of early MoEs was the difficulty of considering data locality when making expert placement decisions. But distributing experts to balance computational load without considering data locality often result in additional data movements that cancel the performance benefits of load balancing. This leads to suboptimal overall model performance, particularly in distributed training scenarios where the communication costs are preponderant. Authors of CCFuser [202] found that distributing popular experts across devices is adverse of data locality. To address this founding, they placed the popular experts on devices where the data resides and distributed across devices only the unpopular experts.

Most MoE models are designed primarily for stable training in fault-free environments and lack robust mechanisms to handle node failure or network partitions. The absence of fault-tolerant expert scheduling meant that system failures could lead to significant performance degradation or complete training interruption. Moreover, existing approaches do not consider how to maintain load balancing properties when the available computational resources change dynamically due to failures or resource allocation changes. In [215], the authors apply optimal replica placement to popular experts under node-failure constraints, adaptively reallocating expert replicas to maintain balanced loads and full node utilization after failures.

The evolution of MoE load balancing techniques reveals several fundamental insights that shape the design of efficient sparse model training. Effective load balancing requires a holistic approach that combines algorithm design, model architecture, and hardware characteristics. Moreover, the temporal dynamics of expert selection patterns require adaptive rather than static optimization strategies. Expert popularity distributions evolve significantly during training, requiring systems that can continuously monitor and respond to dynamic load patterns. On the other hand, the trade-off between load balancing and data locality represents a fundamental challenge that requires joint optimization rather than sequential approaches. Hence, systems that prioritize perfect load balancing without considering data locality often achieve poor performance.

5.7 Training and Fine-tuning MoEs

Performance and training speed

Sparse mixtures of experts train faster than dense models. In [61], the authors compare a dense transformer (T5-Base) with various switch transformers, a mixture of experts model in which the tokens produced by the feed-forward layers of the transformer blocks are either routed to their top expert or dropped. Trained using the same resources, 32 TPU-v3, the mixtures of experts achieved the same performance as the dense transformer in 7 times less training time (Figure 5.17).

Moreover, compared to a larger dense model, T5-Large, a switch transformer with 64 experts trained 2.5 times faster, despite using only two sevenths of the compute cost (Figure 5.18).

Changing the gating network enables additional performance gains. In [243], the authors found that expert-choice gating models trains twice as fast compared to top-k expert gating models, including switch transformers.

Scaling

Sparse models are relevant because they allow the number of weights to increase much more than the number of floating-point operations they require during training and inference. Dense models follow predictable scaling laws: if there are no limitations resulting from a lack of quality data for model training, as the number of weights increases, the cross-entropy loss decreases almost exponentially. As illustrated in Figure 5.19, sparse models also exhibit a consistent performance increase when they are scaled.

The authors of [36] found that this relationship between model size and performance also holds for other gating strategies, such as the Sinkhorn algorithm, reinforcement learning and token hashing. However, the performance increase when scaling the model is less than that observed in dense models. Instead, the authors found that the performance improvement obtained by adding more experts decreases as the models become larger. Once a dense model obtains around 900 billion weights, the authors predict that there will no further improvement by using a sparse mixture of experts. However, it is worth noting that when scaling the sparse models, the recommendation proposed in the Chinchilla model [85] was not taken into account, which suggests

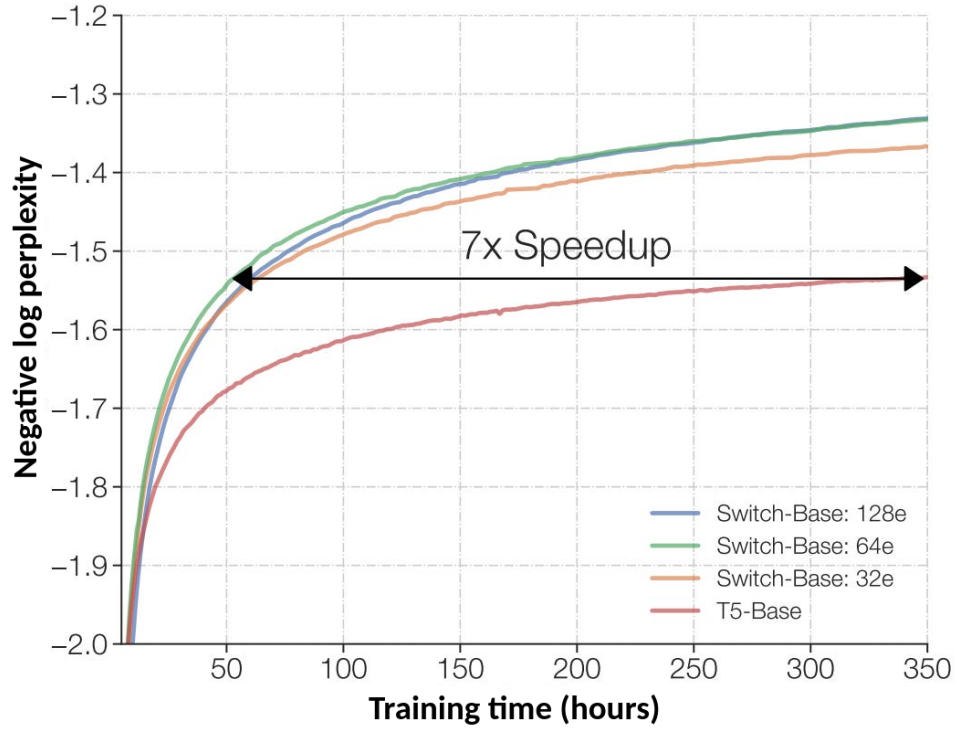


Figure 5.17: Training time and performance (negative log perplexity) of a dense transformer (T5-Base) and Switch Transformer MoE with 32, 64 and 128 experts per layer [61].

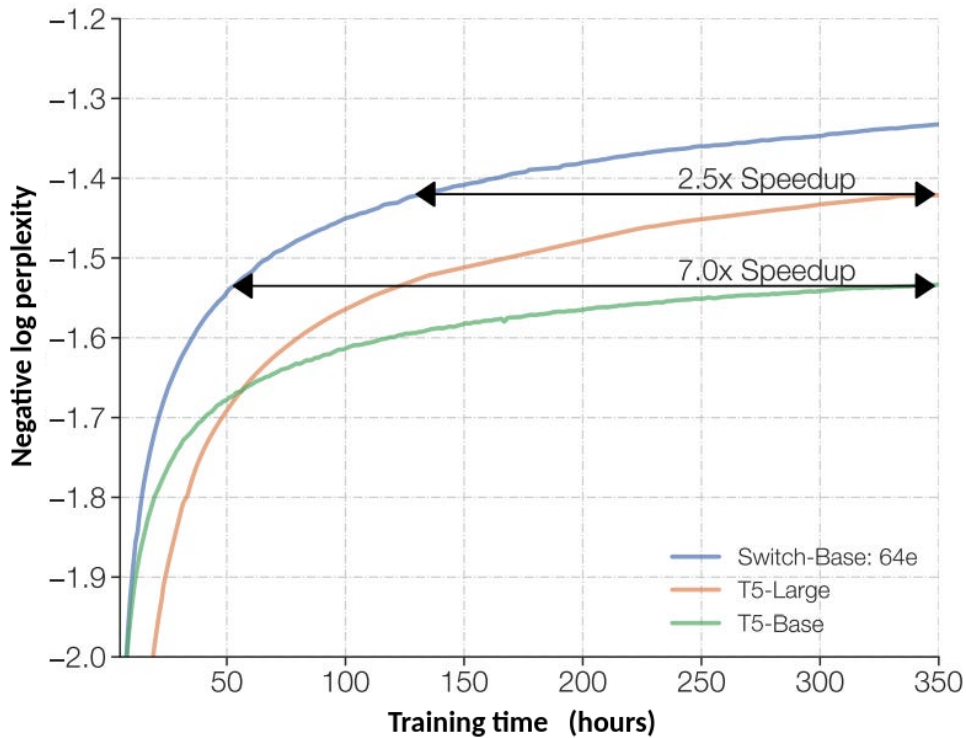


Figure 5.18: Training time and performance (negative log perplexity) of two dense transformers (T5-Base and T5-Large) and a Switch Transformer, a mixture of experts model with 64 experts per layer. T5-Large was trained with 3.5 times more compute than T5-Base or Switch Transformer [61].

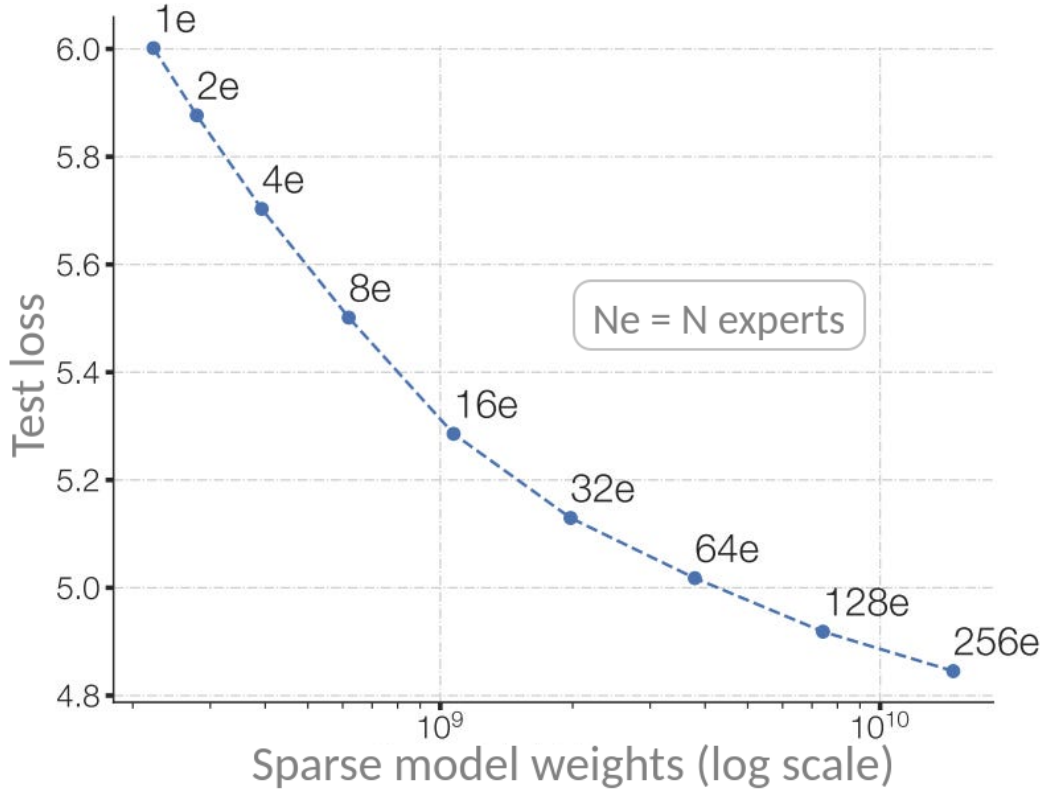


Figure 5.19: The cross-entropy test loss of sparse mixture of experts decreases exponentially when the number of experts increases from 1 to 256. The X-axis is in logarithmic scale [61].

that an increase in the number of model weights should be accompanied by a proportional increase in the number of tokens used during training. Instead, all models were trained on 130 billion tokens. Figure 5.20 shows that the cross-entropy loss of sparse mixture of experts decreases when the number of experts increases, for different routing methods: Sinkhorn (S-BASE), reinforcement learning (RL-R) and token hashing (Hash). Naturally, a lower loss correlates positively with a more performant model.

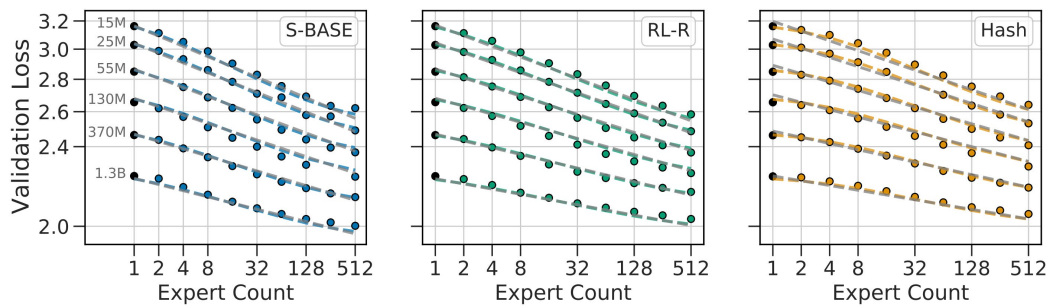


Figure 5.20: Validation loss for different routing methods: Sinkhorn (S-BASE), reinforcement learning (RL-R) and token hashing (Hash). All models were trained using the same compute cost [36].

In dense models, increasing the number of weights makes a model more sample efficient, meaning it trains faster. This also occurs in mixture of experts, when the number of experts and also weights increase, the model requires fewer samples to achieve the same level of performance.

Instabilities

Compared to dense transformers, sparse mixtures of experts are more prone to suffering from training instabilities. A training instability occurs when, instead of decreasing, the loss diverges to infinity. Mixtures of experts are particularly problematic because they contain more exponential function calls, which makes them more susceptible to roundoff errors that arise when we use mixed precision during training.

To reduce the computation and communication costs, large neural networks store weights as float32 but operate on them using Bfloat16. float32 uses 1 bit to represent the sign, 8 bits to represent the exponent and 23 bits to represent the mantissa. Bfloat16 has the same range as float32, since it uses the same 8 bits to represent the exponent, but only uses 7 bits to represent the mantissa. Consequently, the roundoff errors for Bfloat16 are several orders of magnitude higher. Figure 5.21 shows how these errors get worse for large, positive numbers, but the errors are much higher in Bfloat16 than in float32. A similar pattern occurs for negative numbers.

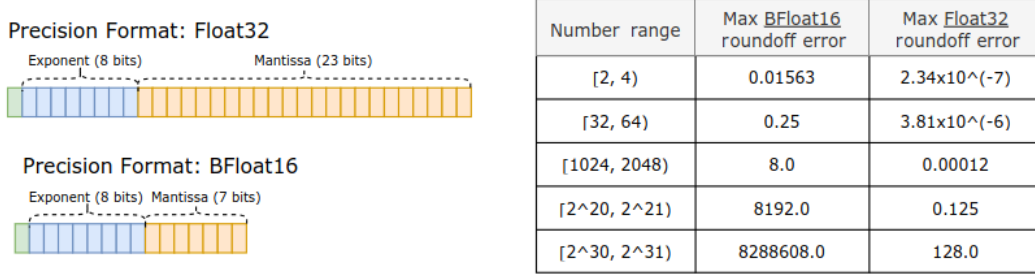


Figure 5.21: Numerical precision formats and roundoff errors. Larger numbers have larger roundoff errors. Bfloat16 has up to 65,536 times worse roundoff errors than float32. The router z-loss encourages the absolute magnitude of numbers to be small, which does not hinder model performance and reduces roundoff errors. The router z-loss is most effective into functions where larger errors can drastically change the relative output (e.g. exponential and sinusoidal functions) [246].

Mixtures of experts contain more exponential functions than dense models, because of the softmax function in their gating networks. Compared to the other operations in a neural network, for example addition or matrix multiplication, the exponential function is numerically unstable: a small change in the input, which might be caused by a roundoff error, can result in a large change in the output. This problem is particularly acute for high magnitude inputs, as the roundoff errors are larger.

Work [246] demonstrated that penalizing high magnitude inputs of the gating network’s softmax function improves training stability without reducing the model’s performance. If T is the number of tokens in a minibatch, N is the number of experts, W are the gating network’s weights, X are the minibatch’s tokens representation, and β is a hyperparameter that reflects the importance of Z-loss, the auxiliary Z-loss is given by Equation 5.37.

$$\mathcal{L}_z = \frac{\beta}{T} \sum_{i=1}^T \left(\log \sum_{j=1}^N e^{(W \cdot X)_{ij}} \right) \quad (5.37)$$

This loss function is minimized when $W \cdot X$, the inputs to the gating network’s softmax function, are small. Computation of the gating probabilities depends only on the relative magnitude of the gating logits, so penalizing high inputs does not necessarily solve them instability problem. The gating probabilities determine which experts process a token and how the expert outputs are scaled. Consequently, numerical instabilities due to roundoff errors have a compound effect.

The authors of [246] reported that training instability can be further reduced by (i) jittering the inputs to the gating network’s softmax by multiplying them by a random number drawn uniformly from $[0.99, 1.01]$ interval, (ii) replacing multiplicative activations, such as GELU gated linear units (GEGLU), with rectified linear units (ReLU), (iii) removing multiplicative normalizations, such as root mean squared error normalization, and (iv) applying dropout throughout the model. Unfortunately, these methods also degrade the model’s accuracy.

Fine-tuning and instruction tuning

Sparse models perform well when training on large and diverse datasets. However, during fine-tuning, they are susceptible to overfitting and consequently may perform worse than their dense counterparts. Mitigating this problem requires setting noisier hyperparameters. The authors of [246] demonstrate that, in contrast to dense models, sparse models perform better during fine-tuning when the learning rate is higher and the batch size is

smaller (Figure 5.22). Dropping out entire experts does not help, but increasing the dropout probability within each expert has a moderate, but positive effect.

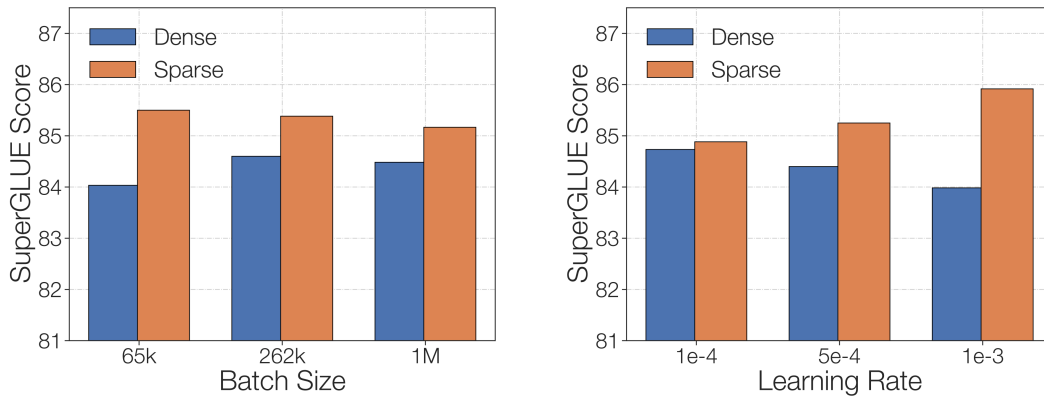


Figure 5.22: SuperGLUE score (higher is better) for diverse minibatch sizes and for different learning rates. Sparse models benefit from smaller minibatch sizes and higher learning rates during fine-tuning. Dense models are the opposite. Both types of transformer models were trained with the same compute cost. First, they were pretrained for 500k steps in minibatches of one million tokens of the CommonCrawl C4 dataset and then they were fine-tuned on the SuperGlue benchmark dataset [246].

Sparse mixtures of experts benefit more from instruction tuning than dense models. Work [179] established that, in the absence of instruction tuning, mixtures of experts do worse than dense models when evaluated or further fine-tuned on downstream tasks. However, once both types of model are instruction-tuned, mixtures of experts outperform dense models.

Upcycling

A useful trick that was introduced in [107], is to upcycle an existing dense model through the initialization of a mixture of experts using the weights of the dense model checkpoint. In a transformer, this is done by training the gating network from scratch, but initializing the experts with the weights of the feed-forward layers they replace. Holding compute constant and continuing training, an upcycled mixture of experts trains faster than the dense model that was at its origin (Figure 5.23).

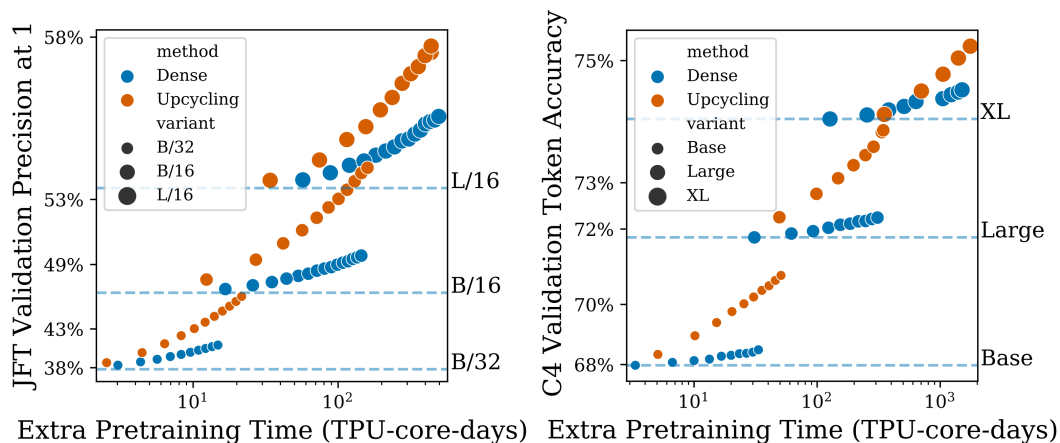


Figure 5.23: Precision and accuracy, as a function of pretraining time, for different transformer models. Upcycled models train faster than continuing to train the dense models on which they are based. The panel on the left shows performance on an image classification dataset and the panel on the right shows token prediction performance on CommonCrawl. B/32 is a vision transformer with a patch size of 32, B/16 and L/16 are vision transformers with a patch size of 16. Base, Large and XL are variants of the T5 language model [107].

So, if there already exists a dense model checkpoint, upcycling it to a mixture of experts might improve performance. And, if there is not a dense model, given that training a mixture of experts is more complicated, one possibility is to train a dense model and then upcycle it to a sparse MoE.

Distillation

After training, a sparse mixture of experts model can be distilled into a smaller, dense model. If all experts are equally utilized, a sparse model and a dense model should require the same amount of GPU memory. The sparse model will have greater throughput and hence requires large batch sizes. However, if the expert utilization is imbalanced, then training a mixture of experts and then distilling it into a dense model is still worth doing, as some of the performance gain is retained. For example, [61] compares a sparse mixture of experts model to a dense T5-Base model that is 100 times smaller. The mixture of experts naturally performs better. However, distilling the weights of the mixture of experts model into the dense model preserves 29% of the performance gain.

5.8 Performance Evaluation

While MoE architectures offer compelling advantages for scaling LLMs, training MoEs presents a number of challenges that differ from traditional dense model training. These challenges arise from the inherent characteristics of sparse activation, dynamic gating, and expert specialization, creating complex interdependencies that cannot be addressed through simple hardware scaling or conventional optimization techniques. Figure 5.24 illustrates the main bottlenecks in distributed MoE training. To facilitate the comprehension, we first present the evaluation metrics and a baseline for assessing the performance of MoE training systems, and then we will analyze the main system challenges and their implications for system design [228].

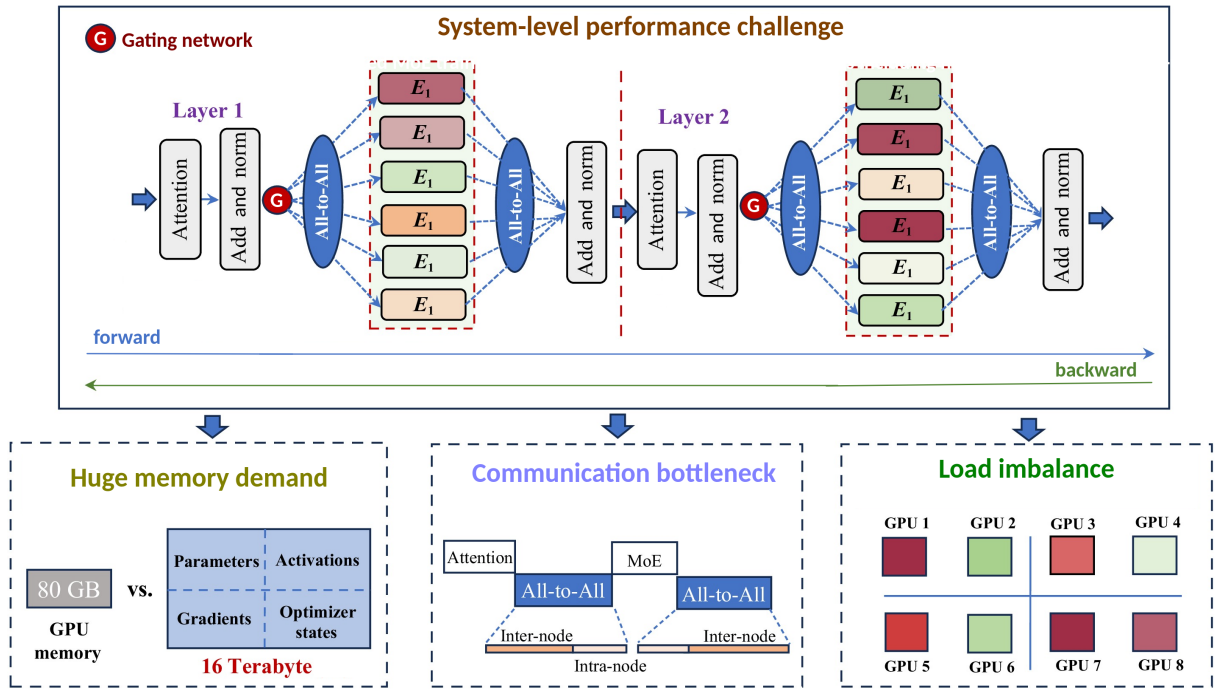


Figure 5.24: System challenges in the distributed MoE training context, illustrating three key bottlenecks including huge memory demand, All-to-All communication overhead, and load imbalance across experts, along in a typical distributed training setup [228].

5.8.1 Evaluation Metrics and Benchmarks

Evaluating MoE training systems requires a methodology that captures both standard distributed training metrics and the performance dimensions introduced by sparse architectures. However, the absence of standardized benchmarks complicates cross-system comparisons, as different works employ varying model configurations, hardware platforms, and measurement methods. In this section, we summarize the key metrics and benchmarks necessary for MoE system evaluation.

Similar to dense LLM training, MoEs are mostly evaluated by (i) training throughput, measured as the number of tokens or samples output per second, and (ii) computational performance, quantified as the number of

floating-point operations executed per second (FLOPS), which is proportional to the training time and cost. However, the sparse activation and dynamic gating characteristics of a MoE brought additional evaluation dimensions that are critical in understanding system efficiency: a system achieving a high raw throughput may still have a low performance if inter-device communication consumes a significant fraction of the training time per iteration, or if load imbalance leaves most GPUs under-utilized. Therefore, hardware efficiency metrics are essential for quantifying accelerator utilization under achieved throughput. Model FLOPS Utilization (MFU) measures the ratio of achieved FLOPS to hardware theoretical peak, while Hardware FLOPS Utilization (HFU) represents hardware efficiency when activation recomputation is applied. These two metrics indicate whether the observed bottlenecks result from a lower accelerator utilization.

Beyond computational efficiency metrics, two bandwidth metrics critically impact MoE training efficiency. Network bandwidth utilization measures the obtained communication throughput as a percentage of theoretical peak, while memory bandwidth utilization typically quantifies how effectively the system saturates the bandwidth of the accelerators' High-Bandwidth-Memory (HBM). These metrics help identifying system bottlenecks and guide us towards the adequate optimizations: a low network utilization indicates that we must optimize the communication, while a low memory bandwidth utilization suggests that we must improve the memory access patterns.

MLPerf-Training [138] provides standard evaluation protocols, but its LLM benchmarks are mostly oriented for dense architectures. In the absence of standardized benchmarks, several evaluation proposals have emerged. Megatron-LM [184] is a reference for production-scale hybrid parallelism with MoE extensions. DeepSpeed-MoE [163] supports five different types of parallelism and exploits both GPU and CPU memory. It enables users to combine different types parallelisms. FasterMoE [77] and Tutel [140] are references for communication optimization techniques in MoE training. Comparative evaluations typically measure performance improvements relative to these baselines under controlled conditions, identical model configurations, expert counts, hardware platforms, and datasets, to isolate the impact of specific optimization techniques.

5.8.2 Performance Challenges

Complex memory management

MoE models require substantially more memory than dense models of equivalent computational complexity. While a dense transformer model with similar floating-point operations might require tens of gigabytes of memory, MoE models with hundreds of experts can demand hundreds of gigabytes to several terabytes of memory for storing the weights only. For instance, one trillion-weight MoE model trained under Bfloat16-float32 mixed-precision requires approximately 16 terabytes of memory to accommodate the model weights, gradients, activations, and optimizer states. This massive scale goes beyond the memory capacity of some GPU clusters, demanding sophisticated memory management strategies that span multiple memory hierarchies.

Beyond raw capacity requirements, the MoE training workload introduces fundamental management complexity problems. First, memory fragmentation occurs due to the sparse distribution of expert weights across different devices and memory segments, leading to inefficient memory utilization and an increased memory management overhead. Second, additional routing metadata and gating networks' states must be maintained for each model layer, including the historic of expert selection, the load balancing statistics, and the gating probabilities. Third, dynamic activation patterns complicate memory locality optimization, as the set of active experts varies unpredictably across samples and along the training iterations.

The technical complexity is a consequence of the inherent dynamism of MoE training. Since expert token allocation varies continuously during training, traditional static memory allocation strategies are inadequate. On the other hand, dynamic memory allocation schemes can improve memory utilization by 15%-20% compared to static approaches but incur in additional runtime overhead due to frequent allocation and deallocation operations [184] [108]. Furthermore, the memory hierarchy considerations become more complex due to the need to efficiently manage expert weights distributed across different memory tiers depending on activation patterns and memory constraints [104].

Communication bottlenecks

Expert parallelism across multiple accelerators introduces severe communication bottlenecks mainly due to All-to-All communication operations, which constitute the most critical constraint of MoE training. Unlike the predictable communication patterns in dense model training, MoE models exhibit dynamic, data-dependent communication patterns that create significant optimization challenges. Quantitative analyses reveal that in distributed MoE training, All-to-All communication operations represent 35%-50% of the iteration time in typical multi-node configurations [99] [77]. Moreover, All-to-All communication exhibits an extremely high latency variability due to the unpredictable nature of token routing decisions, compromising the training predictability and convergence.

Many communication bottlenecks in MoE training are a consequence of the data-dependent communication patterns. Since token routing in common gating strategies are decided dynamically at runtime, the communication has particular characteristics: (i) communication operations are difficult to predict and optimize in advance, preventing effective pre-computation and scheduling; (ii) the dynamic nature of routing contributes to an higher imbalanced data exchange between devices, creating congested and underutilized connections; (iii) the need for global synchronization across all participating devices increases the synchronization barriers and reduces the pipeline efficiency. To make matters worse, the severity of communication bottlenecks is highly dependent on the hardware topology and interconnect characteristics. Within nodes connected by high-bandwidth, such as low-latency NVLink connections, All-to-All communication efficiency can reach 70%-80% of the peak bandwidth. However, on inter-node communications over InfiniBand or Ethernet connections, the efficiency drops dramatically to 20%-35% [140] [182]. This efficiency disparity necessitates topology-aware optimization strategies that adapt communication patterns to specific network architectures and bandwidth hierarchies.

Load imbalance

Load imbalance represents the most challenging performance issue in MoE training, and is the result of the fundamental architectural principles of expert specialization and dynamic token routing. This challenge is inherently difficult to resolve because perfect load balancing fundamentally conflicts with the core premise of MoE architectures, enabling experts to specialize on different types of input patterns.

Load imbalance manifests in two critical problems that significantly impact training efficiency: (i) resource under-utilization occurs when certain experts or accelerators remain in idle or low-utilization states for extended periods, leading to poor hardware efficiency and wasted computational resources, (ii) performance bottlenecks emerge when popular experts introduce throughput limitations, since the overall training speed is constrained by the slowest processing expert.

The complexity of load balancing in MoE training and inference is further increased due to its dynamic nature. The degree of load imbalance evolves continuously throughout the training process as experts progressively specialize on different types of data, making their utilization patterns increasingly heterogeneous. This dynamic evolution renders static load balancing strategies inadequate and necessitates adaptive mechanisms capable of real-time adjustment based on current routing patterns, expert utilization statistics, and training phase characteristics [242] [203].

5.9 Deployment of MoEs

A mixture of experts typically contains a huge number of weights. Consequently, although the computation cost is low, the required memory is extremely large and the model must be distributed across multiple GPUs. To do this, without incurring in accuracy or latency penalties, requires paying close attention to expert provisioning and device allocation.

5.9.1 Model Capacity

Despite the load balancing objective that is often present during training, through a loss term like z-loss, at inference time, expert utilization is extremely unbalanced (Figure 5.25). Authors of [87] evaluated the expert

activation patterns for state-of-the-art mixture of expert architectures and observe that, for language modeling and machine translation tasks, a small number of experts has always allocated a large share of tokens, while the other experts are completely idle. Figure 5.25 (i) shows the experts' activation for a mixture of experts layer in a language model, evaluated on three datasets. Figure 5.25 (ii) shows the experts' activation for two expert layers during a machine translation task, evaluated on translations from English to French (FR), to Japanese (JP) and to Asturian (AS). Here, each pixel represents a batch and a vertical line represents the entire dataset. Since there are large white regions in panel (ii), which correspond to the absence of expert activity, the imbalance in machine translation is particularly high. This is due to a high temporal correlation in machine translation, if one sequence makes use of a particular expert, the probability that the next sequence will use that expert is higher.

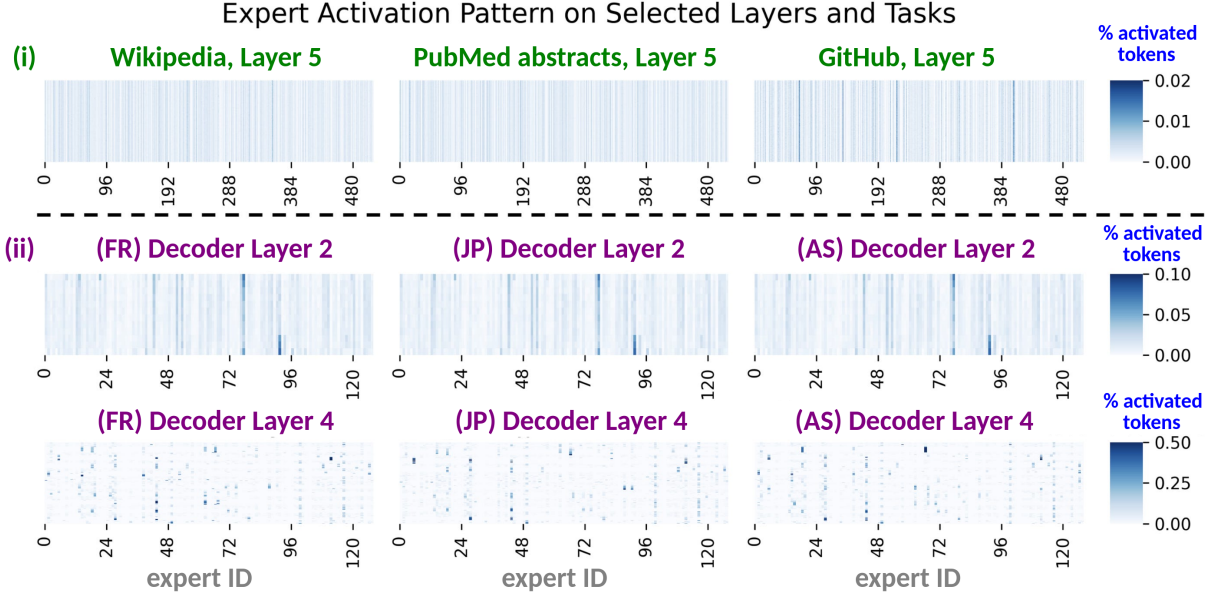


Figure 5.25: Visualization of the expert activation pattern on selected layer of (i) language modeling and (ii) machine translation. Activation is normalized. The expert activation pattern exhibits strong imbalance on all the tasks, and the imbalance is consistent. Specifically, on machine translation decoder the sparseness is enormous, and the expert also demonstrates strong temporal correlation [87].

This imbalance makes it difficult to predict the correct capacity for each expert at inference time. If expert utilization is roughly equal, then provisioning experts is straightforward. Let N be the number of experts in a model, T be the number of tokens in each minibatch and C be the capacity factor. Then, in a naive implementation we statically provision each GPU with a fixed capacity of $\frac{C \cdot T}{N}$ tokens per expert. To make efficient use of GPU parallelism, the expert feed-forward networks on each device are stacked as tensors. If an expert is overloaded, the additional tokens are dropped or routed to an expert with spare capacity. If an expert has low utilization, padding tokens are allocated to it instead. This means that the token representations can also be stacked as a tensor, improving efficiency.

The dimension of the expert tensor is $\text{num_experts_on_device} \cdot \frac{C \cdot T}{N} \cdot \text{expert_output_dimension}$ and the dimensions of the token tensor is $\text{num_experts_on_device} \cdot \frac{C \cdot T}{N} \cdot \text{token_representation_dimension}$. These two tensors are multiplied to obtain the experts' output.

When expert allocation is unbalanced, the overloaded experts will drop or reroute tokens, reducing performance. ON the other hand, experts with low utilization will waste compute resources processing the padding tokens. If the imbalance is slight, token dropping can be reduced by increasing the capacity factor. However, a very large imbalance results in an enormous waste of resources.

If expert utilization can be accurately predicted ahead of time, then individual capacity factors can be set for each model. Work [87] demonstrates how this can be done by examining the historic of data loaded into each GPU. The optimal allocation of experts to GPUs is isomorphic to the multi-way partitioning problem, which is NP-hard. This can be approximated with a greedy algorithm, where at each step, the expert with the highest historical workload is allocated to the GPU with the smallest load. Unfortunately, this approximation is less

effective if the token sequences are temporally correlated, as in machine translation. To fix this issue, an expert's predicted workload needs to be increased by an amount that depends on the historical workload of each of its correlated experts, multiplied by the strength of the correlation and also multiplied by a scaling factor.

5.9.2 Caching

A further optimization that can be made is to offload the least used experts onto the CPU. This can be implemented using a least recently used cache - and this would account for temporal correlations. If there is a cache miss, there is a latency penalty, as the required expert must be copied to a GPU for inference. Work [87] found that for a machine translation model with 128 experts, reducing the cache size from 128 to 64 decreases memory usage by 15% but increases latency by 0.15 seconds (i.e. 2.5 times).

5.9.3 Block Sparsity

GPUs are designed to perform the same computation multiple times in parallel. Therefore, they are more efficient at performing dense matrix multiplications than at executing sparse operations. To exploit the GPU efficiency, deep learning libraries, including those specialized in training mixtures of experts, make heavy use of batched matrix multiplications (BMM). A BMM requires that the batched matrices to have the same shape, and some libraries even require that the shapes are statically specified at compile time. This is the reason why, at inference time, expert capacity must be accurately predicted and why the inputs to the experts must be padded or truncated.

Matrix multiplication consists of the repeated application of the dot product between the rows of the first matrix and the columns of the second (Figure 5.26). To multiply large matrices together, the GPU splits the input and output matrices into tiles. Each output tile of matrix C , with size $(Mtile \times Ntile)$ is computed by stepping through the inner dimension of the input matrices A and B , tile by tile, multiplying a tile of A with size $(Mtile \times Ktile)$ by a tile of B with size $(Ktile \times Ntile)$, and aggregating all the resulting tiles.

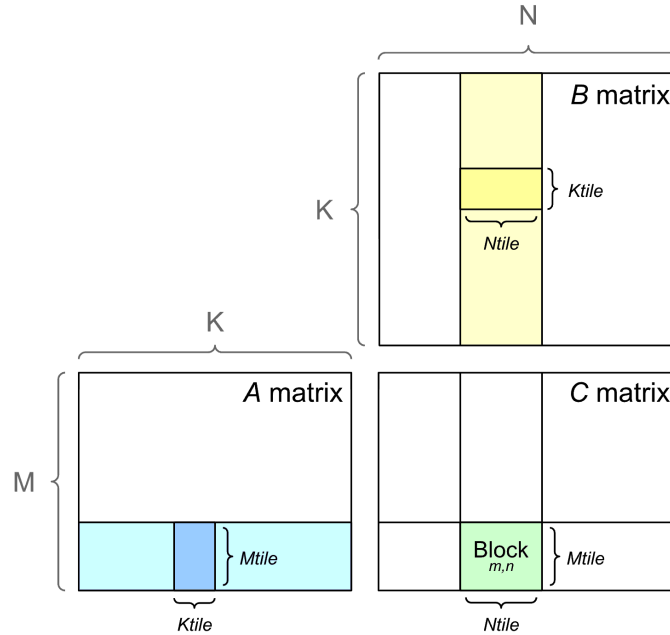


Figure 5.26: Tiled matrix multiplication between A and B [39].

In [65], the authors observed that the forward and backward passes in a mixture of experts feed-forward layer can be expressed as block sparse matrix multiplications. In a block sparse matrix, the nonzero elements are clustered into square sub-matrices. This is useful, because block sparse matrix multiplications can also be tiled, enabling them to exploit GPU parallelism. As the empty blocks in the output matrix do not need to be computed, in principle block sparse matrix multiplication can achieve a similar throughput as batched dense matrix multiplication.

Figure 5.27 illustrates how we can evolve from an implementation of a MoE layer using a batched matrix multiplication (i), where the computations of all experts are done in parallel, to a block diagonal matrix multiplication(ii), and even to a block-sparse matrix multiplication (iii). Batched matrix multiplication requires that all experts are assigned the same number of tokens and that all experts have the same shape, which in many cases is difficult to guarantee. The expert computation can also be implemented with a block diagonal matrix multiplication, which still requires that all blocks have the same size. To avoid these constraints, we can implement expert computation using a block-sparse matrix multiplication. In this case, we run a block diagonal matrix multiplication but the blocks do not have the same size and we obtain a block diagonal matrix with variable sized blocks made up of many smaller blocks. We can compute this diagonal matrix efficiently using block-sparse matrix multiplication.

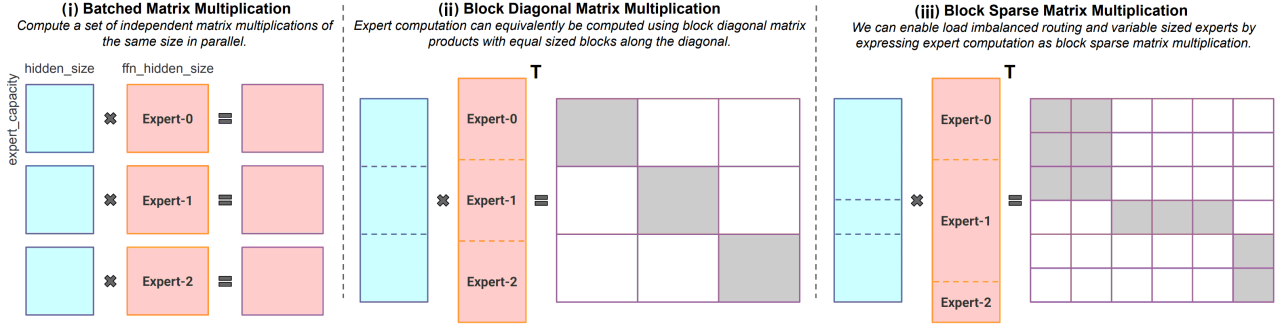


Figure 5.27: Expert computation in an MoE layer with three experts. (i) State-of-the-art MoE implementations use batched matrix multiplication to compute all experts within a layer in parallel. This imposes the constraints that all experts are assigned the same number of tokens and that all experts have the same shape. (ii) Expert computation can be analogously implemented with a block diagonal matrix multiplication with identically sized blocks. (iii) In order to relax these constraints, we can construct a block diagonal matrix with variable sized blocks made up of many smaller blocks. We can compute this matrix efficiently using block-sparse matrix multiplication [65].

When implemented as a block sparse matrix multiplication, a mixture of experts feed-forward layer can flexibly and efficiently handle unbalanced token assignments without needing to drop tokens or waste compute processing padding tokens. Work [65] provided the CUDA kernels necessary to do this. Their implementation achieves 98.6% of the throughput obtained by cuBLAS for batched dense matrix multiplications, evaluated on mixture of experts feed-forward layers (Figure 5.28). Using block sparse matrix multiplications decreased the training time by 1.38 times for the smallest model and 4.35 times for the largest one.

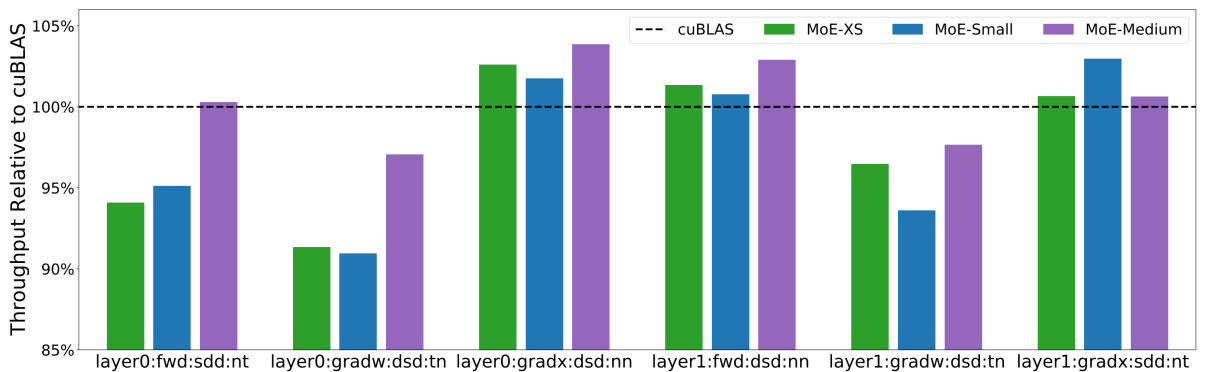


Figure 5.28: Block-sparse matrix multiplication compared to cuBLAS batched matrix multiplication. cuBLAS implementation is compared to MoE-XS, MoE-Small and MoE-Medium models. The block-sparse matrix multiplication kernels achieved 98.6% of the cuBLAS's average throughput with a standard deviation of 4% and a maximum and minimum relative throughput of 104% and 91%, respectively [65].

5.9.4 Mitigating Inefficiencies of Inference

Has we have seen in previous section, a naive inference over a MoE model can result is low GPU utilization and exhibit a latency much higher than expected for the available computation resources [87]. In Figure 5.29, we compare the inference latency of MoE and dense models running on a single node. While we expected

that MoE models could run inference with a latency similar to their flop-equivalent dense counterparts, we observe that they are 15 times slower on the language modeling (LM) task, 22 times slower on the machine translation (MT) task using a transformer encoder, and 3 times slower on the machine translation when we use a transformer decoder architecture.

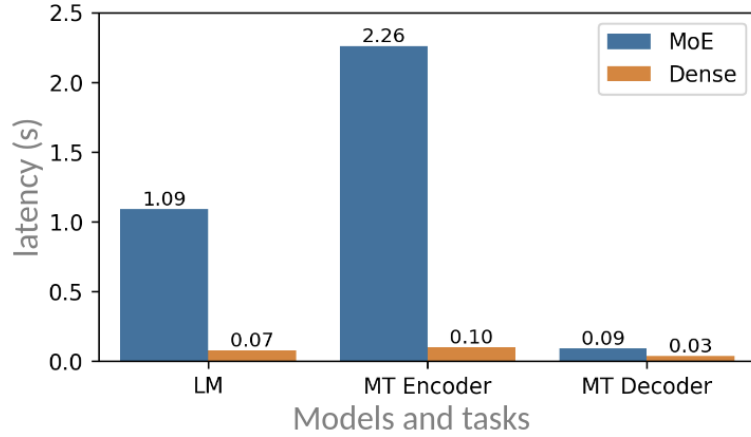


Figure 5.29: Comparing the inference latency of MoE and dense models running on a single node. While theoretically MoE models should be able to infer on a similar latency as their flop-equivalent dense counterparts, we find that in practice they are 15 times slower on language modeling (LM), 22 times slower on machine translation (MT) encoder and 3 times slower on machine translation decoder [87].

Before proposing a solution to this problem, the authors of [87] first characterized MoE inference as a workload. To start, they examine the space consumption. The expert weights are the biggest chunk, but there is also a large memory to store activations compared to the dense model, as illustrated in Figure 5.30.

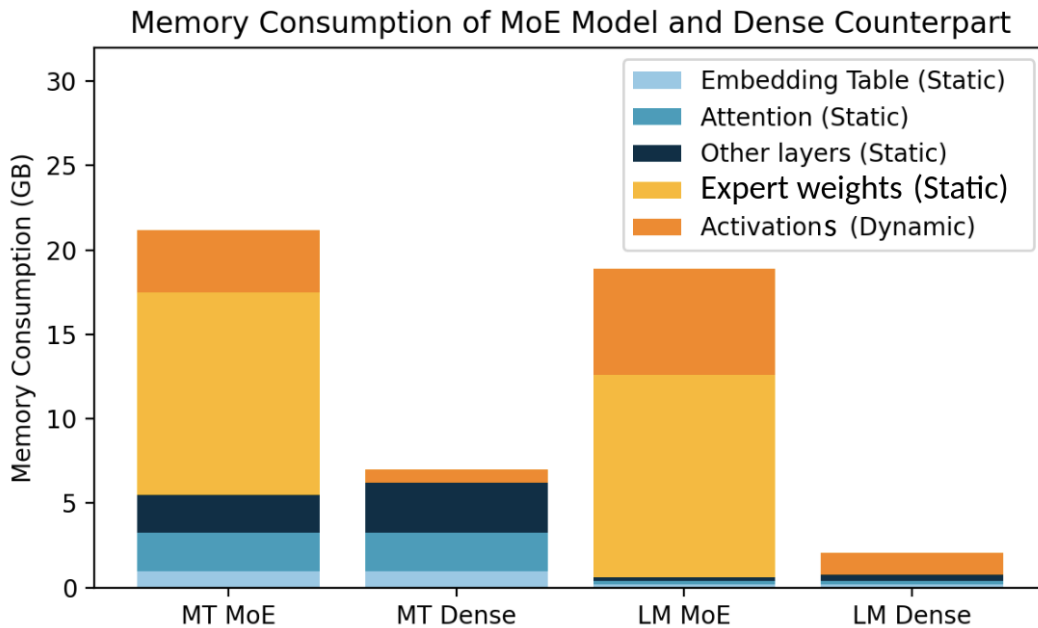


Figure 5.30: Comparing the memory footprint of MoE models to dense models during inference. The MoE models require significantly more memory usage when deployed on GPUs. Besides the large memory consumption due to the expanded model capacity, introduced by expert weights, it also requires more memory for activations. The results are relative to a batch size of 48 in machine translation (MT), and a batch size of 8 in language modeling (LM) [87].

In terms of latency, the contribute of the expert routing, expert execution and All-to-All communication was above 75% in all evaluated configurations. This contribute is around 95% when using a single node and the language modeling task. Thus, the authors concluded that MoE inference is running inefficiently (Figure 5.31).

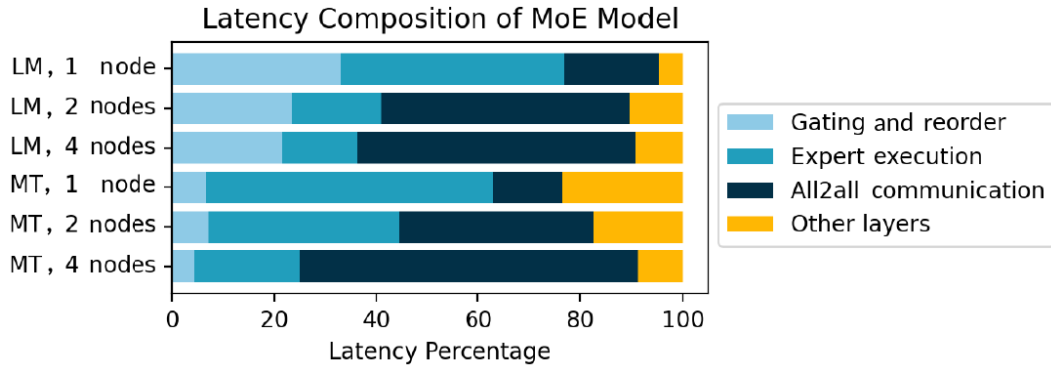


Figure 5.31: MoE model latency composition. Besides All-to-All communication, other components of the model, such as the gating function and expert execution, are also inefficient. Communication overhead increases significantly when more than one node is involved. Results are relative to a batch size of 8 on language modeling (LM) and a batch size of 48 on machine translation (MT) [87].

Figure 5.32 plots the average number of inactive experts, on language modeling (LM) and machine translation (MT), as a function of the layer identifier. Most, if not all experts are active in the LM MoE and especially in the MT encoder MoE. However, activation on the MT decoder MoE is very sparse, even when a batch size of 96 and a dynamic gating strategy are used. For example, in the translation from English to Asturian and for the later layers, 102 experts out of 128 are inactive.

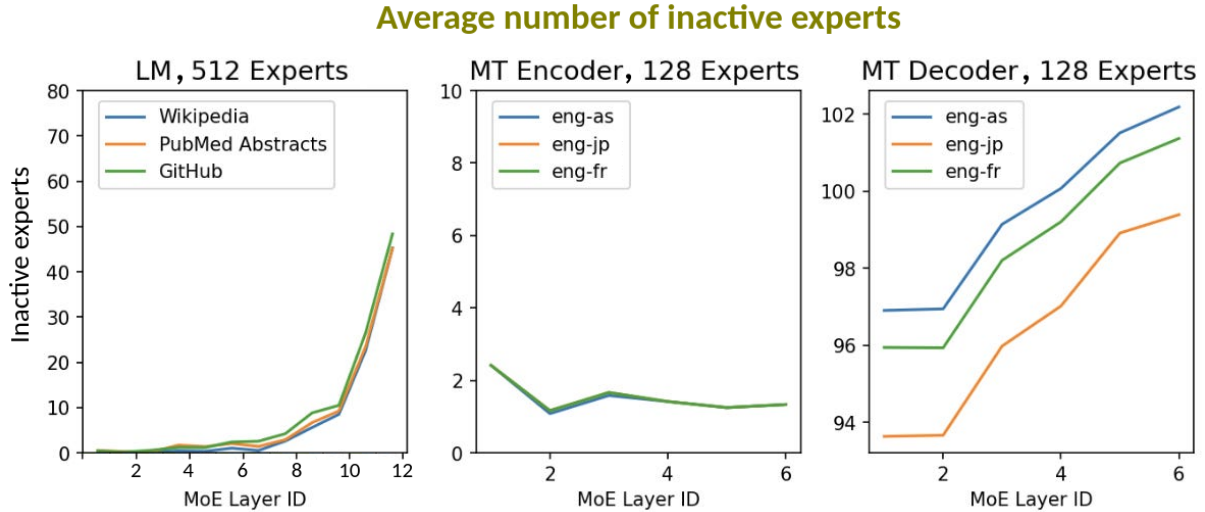


Figure 5.32: Average number of inactive experts on language modeling and machine translation. Most, if not all experts are activated throughout the LM and MT encoder. However, activation on MT decoder is extremely sparse, even if we utilize a batch size of 96 and a dynamic gating strategy [87].

In the rest of this section, we will discuss solutions for the three major challenges of MoEs: inefficient token routing, high memory consumption, and workload imbalance across devices. The reported solutions are dynamic gating, expert buffering, and load balancing.

Dynamic Gating Optimization

Traditional MoE systems rely on static gating, where each expert has a fixed capacity for processing tokens. This often leads to inefficiencies: some experts become overloaded, causing token dropping, while others remain underutilized, wasting resources. To solve this issue, the authors of [87] proposed dynamic gating, where expert capacity adapts to the actual number of assigned tokens. Instead of fixed dispatch masks, tokens are sorted and distributed dynamically based on demand. This approach eliminates token dropping and improves the robustness, reduces memory and communication waste by avoiding empty placeholders, and uses a two-step communication strategy, that includes exchanging the size and transferring the tokens, to manage variable workloads.

As illustrated in Figure 5.33 (a), static gating distributes tokens using fixed-size dispatch masks, which can result in token dropping and underutilization. As shown in Figure 5.33 (b), in dynamic gating the tokens are

instead sorted and distributed dynamically based on actual demand, allowing each expert to receive a variable number of tokens. This mechanism eliminates token dropping, reduces unnecessary memory allocations, and improves communication efficiency. Dynamic gating significantly improves performance, namely the throughput increases by up to 11 times in multi-node setups, memory usage is reduced, up to 79.6% reduction in some cases, and enables much larger batch sizes under the same hardware constraints.

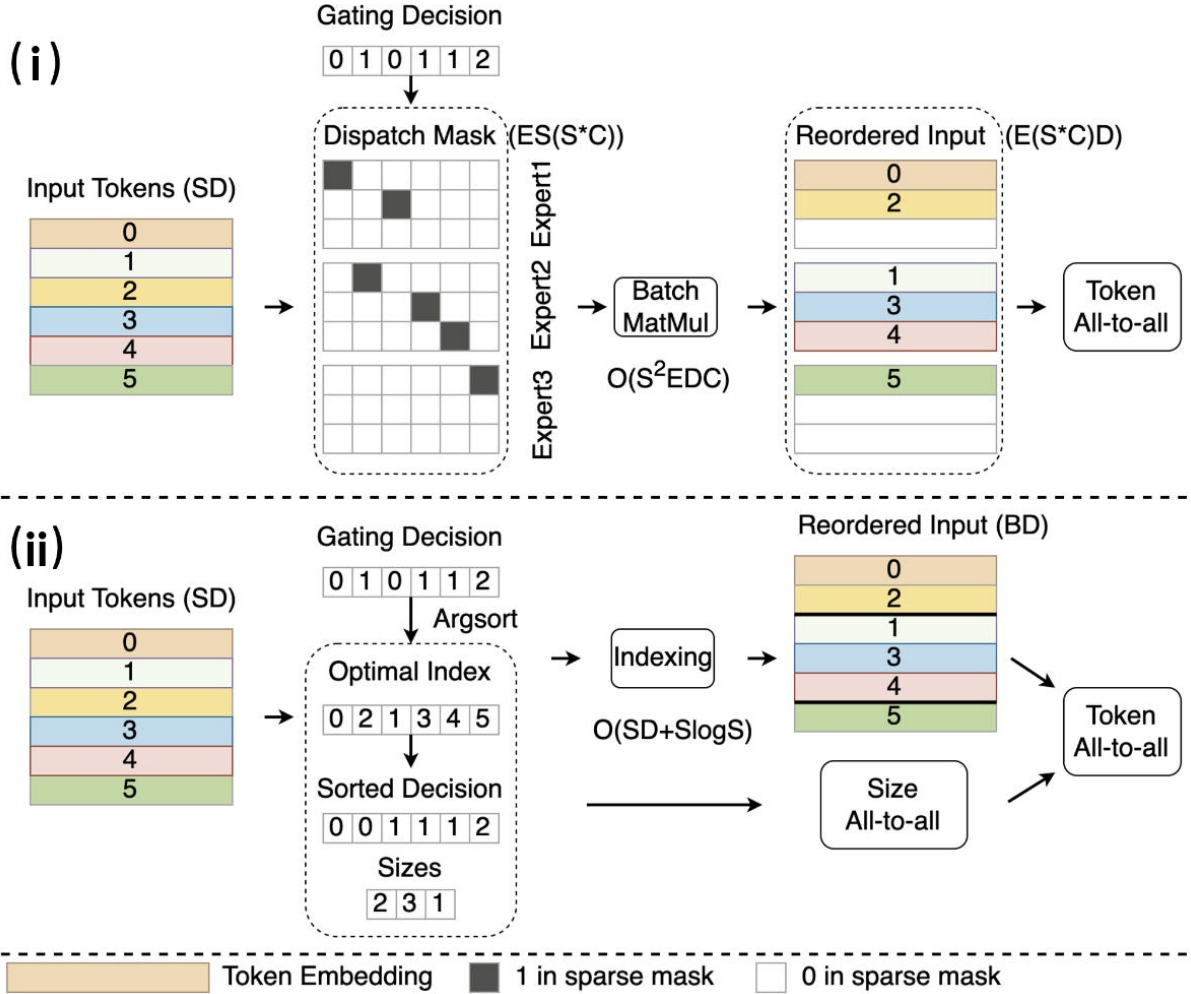


Figure 5.33: Comparison between the static gating in [8] and [114], and dynamic gating in [87]. For simplicity, we consider three experts ($E = 3$), six tokens ($S = 6$), a capacity factor of 0.5 ($C = 0.5$), and a top-1 gating. Tensor shapes are annotated in parentheses. (i) Static gating. Under a static gating policy, each expert always processes a predefined amount of tokens, which may lead to token overflow or empty tokens. (ii) Dynamic gating. Under a dynamic gating policy, each expert only processes the tokens that are assigned to it. The token distribution mechanism is simplified with less complexity, and the communication and computation are reduced [87].

Expert Buffering

Even with dynamic gating, MoE models still require large memory due to many expert weights. However, not all experts are active at all times. The proposed expert buffering technique leverages this sparsity through the following measures: (i) frequently used (popular) experts are stored in GPU memory, (ii) rarely used experts are offloaded to CPU memory, and (ii) when needed, experts are transferred to GPU dynamically (Figure 5.34).

To manage GPU memory, a cache system is used, where only the active experts are cached. When the cache is full, inactive experts are evicted first. A Last-In-First-Out (LIFO) policy is used for efficient replacement of experts residing in the cache. The buffering technique reduces the static memory usage significantly, up to 2.25 GB saved, maintains a competitive throughput, especially in multi-node environments, and introduces a tradeoff between latency and memory due to CPU-GPU transfers.

Figure 5.35 shows the decoder model latency vs. the memory consumption under a series of cache configurations. The size of the cache is annotated above the blue line. The authors varied the size of the cache in each GPU and measured the decoder latency and peak memory consumption. They concluded that the

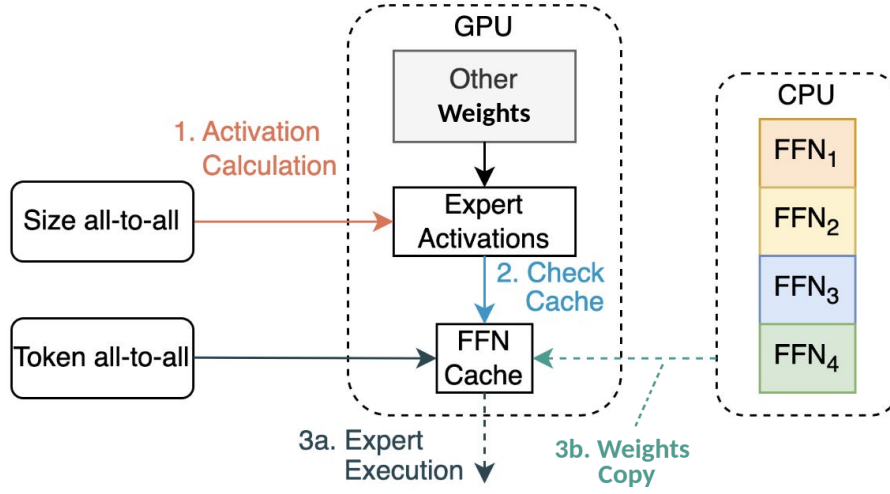


Figure 5.34: Illustration of the expert buffering mechanism. Expert weights are moved to CPU memory for reducing the used GPU memory. On GPU memory, we allocate space only for a few expert entries to buffer active or popular experts. (1) During inference, an All-to-All message containing the size is sent in stage 1, as shown in Figure 5.33 (ii), to indicate which experts located in the current device are active. (2) Then the expert cache will check whether the active experts currently reside in the buffer. (3a) If found in cache, weights in the expert buffer will be used to process the tokens. (3b) If not found in cache, then the expert weights will be requested from the CPU memory. The number of cache entries on GPU memory is a tunable parameter to adjust for desirable GPU memory usage and latency [87].

primary contributor to the increased latency is the constrained CPU-GPU bandwidth, which saturated around 12GB/s during the experiments. Thus, the layers with a high cache miss rate degrade the overall performance.

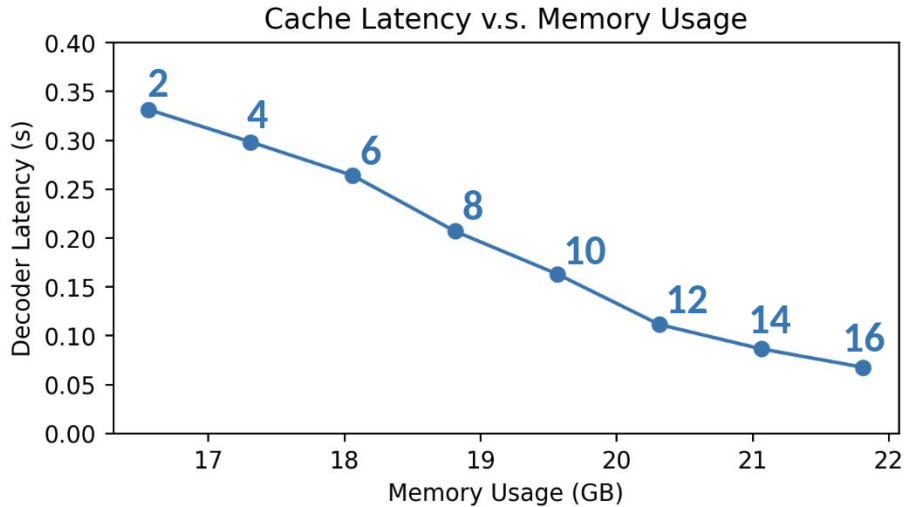


Figure 5.35: Tradeoff between the memory usage and the cache latency under different cache configuration on machine translation decoder. Corresponding cache size per GPU is above the blue line [87].

Load Balancing Across Devices

Another issue in MoE systems is uneven workload distribution. Some devices host frequently used experts, while others remain underutilized, causing bottlenecks and inefficiencies. To address this problem, the authors of [87] proposed load balancing strategies: (i) greedy balancing, where experts are sorted by historical workload, each expert is assigned to the device with the lowest current load, and in this way it ensures a more even distribution of work. Greedy balancing reduces the peak device load significantly and improves throughput by up to 19.5%; (ii) anti-correlation balancing, which was designed for cases where expert activations are correlated. It assigns experts based on both workload and correlation patterns, and avoids placing similar experts, those that are frequently co-activated, on the same device. Anti-correlation balancing further balances workload in complex cases, for example machine translation decoders, brings slight performance improvements, and reduces the cache miss rates.

The effectiveness of the greedy balancing approach is shown in Figure 5.36, which compares workload distribution before and after balancing. Results show that greedy is able to balance the load for the language modeling (LM) use case by improving both metrics, maximum load and average maximum per device, significantly, reducing from more than 0.6 to less than 0.4. In machine translation (MT) use case, greedy was able to balance the expert load assignment of the MT encoder.

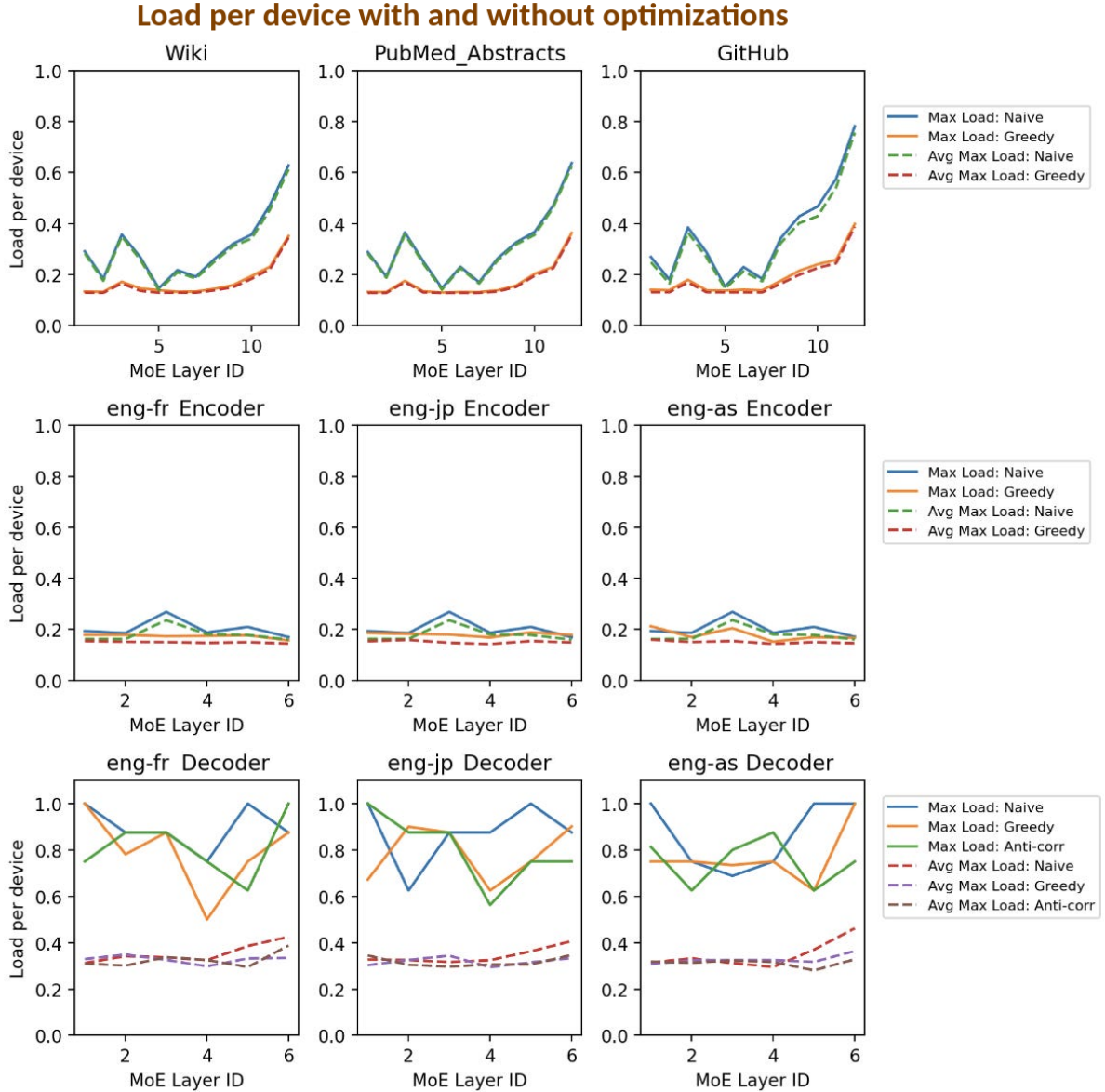


Figure 5.36: The effect of the load balancing mechanism: Based on historical activation data, the algorithm is able to significantly reduce the load imbalance problem on LM tasks and improve the robustness by limiting the maximum workload on a single device [87].

5.10 Memory Optimization

In this section, we discuss the critical challenge of memory consumption in large-scale mixture-of-experts training, emphasizing that, unlike dense models, MoE models face fundamentally different constraints due to their enormous weight count [228]. Excessive memory usage not only limits the minibatch size but also reduces accelerator efficiency, ultimately hindering optimal training performance [104]. Although parallelism helps distribute computation, it often increases memory overhead through redundant weight storage and communication buffers, making targeted memory optimization essential. We organize the memory optimization techniques, specifically designed for MoE training, into four major categories that address different aspects of the memory performance challenge: redundancy optimization techniques, activation recomputation strategies, expert weight offloading approaches, and low-precision training methods. Figure 5.37 illustrates the taxonomy of these memory optimization approaches and their relationships.

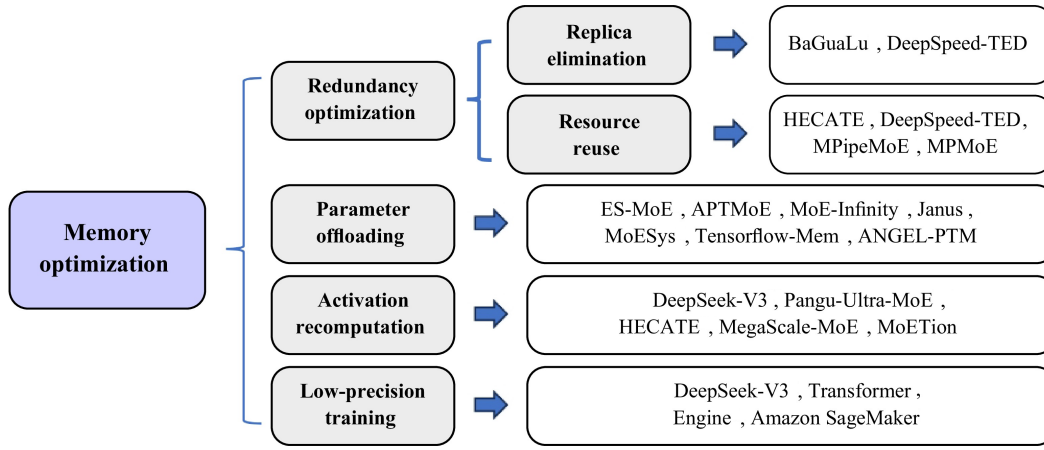


Figure 5.37: Memory optimization techniques taxonomy in the context of MoE training [228].

5.10.1 Redundancy Optimization

In distributed MoE training, redundancy arises primarily from data parallelism, where each device maintains a full copy of the weights, gradients, and optimizer states. This becomes especially problematic in MoE systems because optimizers, such as Adam, maintain additional statistics, namely momentum and variance, significantly amplifying memory requirements [106].

A key solution to this issue is Zero Redundancy Optimizer (ZeRO) [164], which we discussed in Section 3.7 and is implemented in the DeepSpeed framework [168]. As we have seen, ZeRO introduces a three-stage partitioning strategy: optimizer states (ZeRO-1), gradients (ZeRO-2), and model weights (ZeRO-3) are progressively sharded across devices. This approach reduces memory consumption to roughly $1/M$ of the original footprint, where M is the number of devices. ZeRO-1 in particular has been widely adopted in MoE systems such as DeepSpeed-TED [185] and the Llama-3 training framework [200]. Comparable approaches include PyTorch’s Fully Sharded Data Parallel (FSDP) [238], which enables fine-grained weight sharding, and BaGuaLu [136], a topology-aware optimizer that reduces optimizer memory overhead to less than 5%, enabling training at extreme scales, for example, 174 trillion-weight MoE models.

Beyond eliminating redundancy, temporal memory reuse strategies exploits the fact that tensors across layers do not overlap in their lifetimes during forward and backward passes. By reusing memory buffers across non-overlapping operations, systems can significantly reduce peak memory usage. For example, DeepSpeed-TED [185] uses a tile-based strategy to recycle temporary buffers, reducing memory fragmentation by around 40%. Similarly, HECATE [159] enables expert-level memory sharing across layers by dynamically materializing weights only when needed.

In pipeline parallelism contexts, MPipeMoE [235] introduced a multi-faceted reuse scheme, including shared activation memory, shared gradient buffers across stages, and cyclic reuse of expert weights. Building on this, MPMoE [234] jointly optimizes pipeline scheduling and memory configuration using performance models. In general, these redundancy and memory reuse techniques substantially lower memory requirements and fragmentation, enabling the training of extremely large MoE models. Although they introduce additional communication overhead and scheduling complexity, they fundamentally improve the balance between memory efficiency and computational scalability, making large-scale MoE training feasible.

5.10.2 Expert Weight Offloading

Expert weight offloading is a crucial memory optimization strategy for large-scale MoE models, where we move inactive weights from GPU memory to slower storage levels to alleviate memory constraints (Figure 5.38). In such systems, weights that are not immediately needed are temporarily stored in host DRAM or NVMe SSDs and later transferred back into high-bandwidth GPU memory (HBM) during forward and backward passes. This general approach has proven effective in dense large language models, where systems like ZeRO-Infinity [165] leverage prefetching and overlap techniques to achieve more than a sixfold increase in model scale by efficiently utilizing NVMe bandwidth [95] [156].

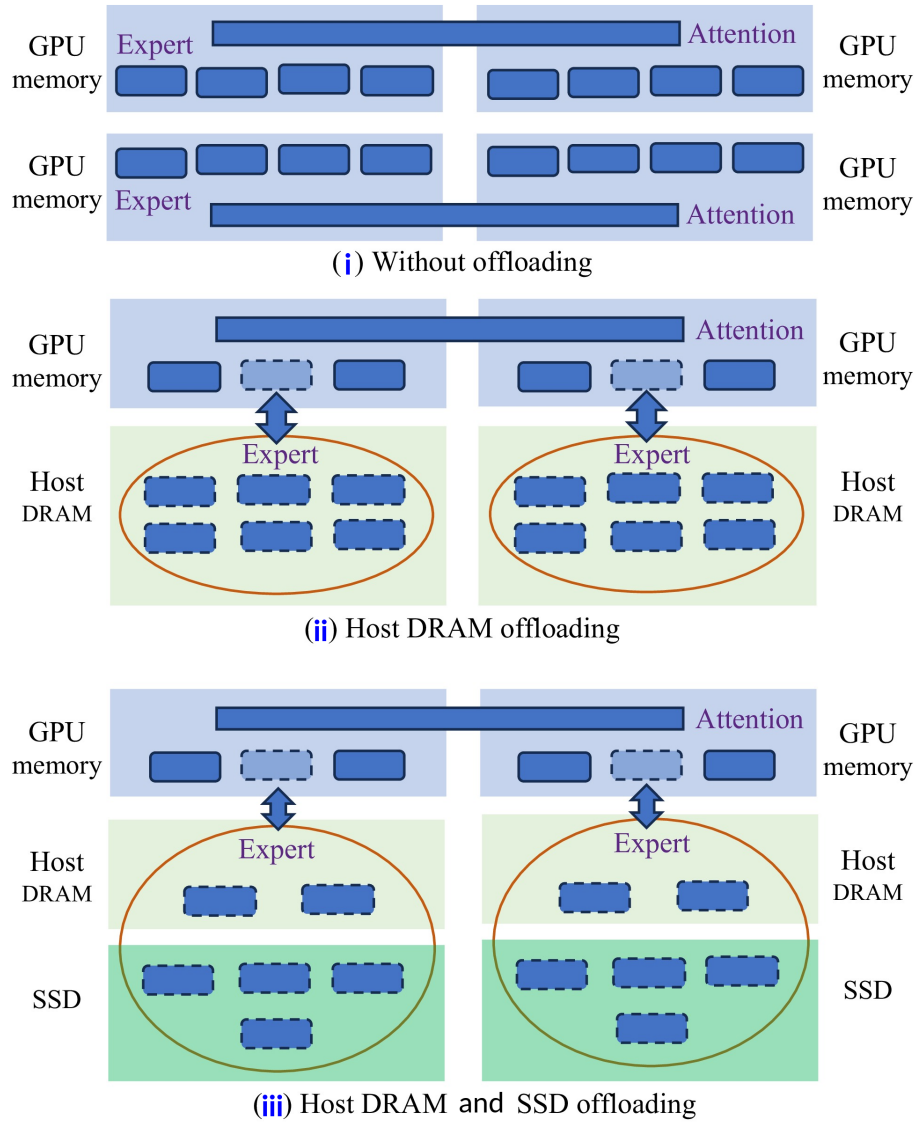


Figure 5.38: Expert weight offloading strategies. (i) Traditional GPU-only storage, (ii) host DRAM offloading for inactive experts, and (iii) hierarchical offloading using both host DRAM and SSD storage [228].

However, MoE architectures amplify the challenges of offloading. Because MoE models contain a huge weight count, primarily concentrated in experts, traditional layer- and model-level offloading techniques become inefficient. The limited CPU-GPU communication bandwidth, which is 32 GB/s in PCIe 4.0, creates severe bottlenecks, undermining the computational advantages of sparsity. As a result, naive offloading strategies introduce excessive latency and fail to scale MoE training effectively.

To overcome these limitations, the literature proposes a shift toward expert-granular offloading, aligning memory management with the natural structure of MoE computation. ES-MoE [104] exemplifies this paradigm by offloading weights and optimizer states at the level of individual experts and executing them sequentially rather than in large batches. This fine-grained approach significantly improves efficiency, enabling support for up to 67 times more experts and achieving a 17.5-fold increase in training throughput. The key advantage lies in selectively keeping only the currently needed experts in fast memory, thereby minimizing unnecessary data transfers.

A further refinement is introduced through activation-aware offloading, which addresses the unpredictability of expert selection. In standard MoE gating, it is difficult to anticipate which experts will be activated, leading to reactive data transfers that incur in latency penalties. MoE-Infinity [220] tackles this issue by leveraging temporal locality that exists in activation patterns. Its Expert Activation Matrix Collection (EAMC) framework analyzes recurring activation sequences across inputs, enabling predictive prefetching of experts. By identifying patterns in which certain experts are frequently co-activated for specific inputs, the system transforms reactive

memory handling into a proactive strategy. Similarly, Janus [132] introduces a data-centric training framework in which experts are dynamically transferred across workers, coordinating CPU-GPU communication with computation to reduce idle time and improve resource utilization.

Systems such as MoESys [223] combine multiple storage levels and communication channels: dense components are distributed using ZeRO-3-like strategies over high-bandwidth NVLink connections, while sparse expert weights are offloaded to DRAM and SSDs. This hybrid approach demonstrates that efficient MoE training depends not only on algorithmic innovation but also on careful integration with hardware topology and communication infrastructure. In conclusion, these advances demonstrate that effective offloading in MoE systems requires fine-grained, predictive, and hardware-aware strategies. By moving beyond coarse-grained techniques and leveraging expert-level control, activation prediction, and hierarchical storage, recent approaches significantly reduce memory pressure while preserving, or even enhancing, training performance.

5.10.3 Activation Recomputation

Activation recomputation is a fundamental memory optimization strategy in large-scale training, and particularly in MoE model training. It has been shown that intermediate activations often dominate overall memory usage, especially in deep architectures. Activation recomputation, also known as gradient checkpointing, addresses this issue by discarding intermediate activations during the forward pass and recomputing them later during backpropagation, thereby trading an additional computation for a reduced memory footprint [108] [94]. This technique is especially valuable in MoE systems, where memory pressure can be exacerbated by load imbalance during the early training stages, sometimes leading to out-of-memory situations [200].

Despite its effectiveness, traditional activation recomputation suffers from a critical limitation: it operates directly within the training’s critical execution path. This means that while it reduces memory usage, it often introduces non-negligible computational overhead, slowing down overall training and discouraging adoption in performance-sensitive environments. Recent advances have therefore focused on improving efficiency through selectivity and precision awareness, ensuring that recomputation is applied only where it yields global benefits.

For instance, MegaScale-MoE [101] introduces selective recomputation, where only activations associated with memory-intensive operations are recomputed, while those that are computationally expensive to regenerate are maintained. This balances memory savings against recomputation cost, effectively eliminating the traditional throughput penalty. Similarly, DeepSeek-V3 [51] combines recomputation with FP8 mixed-precision storage, showing that reducing numerical precision, when carefully managed, can further decrease memory usage without compromising training stability.

More fine-grained approaches push this idea further by tailoring recomputation to the level of individual operations. PanGu-Ultra-MoE [193] adopts operator-level recomputation instead of layer-wise strategies, introducing techniques such as Key-Value (KV)-only recomputation in attention modules. In this scheme, only the key and value tensors are recomputed, while other activations are selectively stored or recomputed depending on their computational and memory costs. In parallel, HECATE [159] rethinks activation management through sparse materialization, computing and storing only the activations corresponding to the experts actually selected during gating. Full activation patterns are then reconstructed when needed, reducing unnecessary memory usage.

The inherent sparsity of MoE models, where each token is processed by only a subset of experts, creates both opportunities and challenges for activation recomputation. On one hand, inactive experts do not require recomputation, enabling more efficient memory-computation trade-offs than in dense models. On the other hand, dynamic gating complicates the identification of which activations are worth preserving. To address this dilemma, MoETion [66] introduces a "sparse checkpoint" mechanism, which selectively saves only a small subset of activations based on their importance for gradient computation and their likelihood of reuse. By leveraging routing information, this approach minimizes recomputation overhead while maintaining training efficiency.

Thus, the evolution of activation recomputation in MoE systems reflects a shift from uniform, layer-based

checkpointing toward selective, sparse, and precision-aware strategies. By exploiting the structural properties of MoE architectures, particularly sparsity and dynamic gating, actual techniques achieve significantly improved memory efficiency without incurring in the severe performance penalty that once limited the practicality of activation recomputation.

5.10.4 Low-Precision Training

Low-precision training is a complementary strategy for reducing memory consumption in large-scale MoE models. The support of 8-bit floating point (FP8) computation in modern accelerators, such as NVIDIA H100 GPUs, made it possible to significantly lower memory requirements while maintaining competitive model performance. By executing general matrix multiplications in FP8 during the forward passes, as well as in the computation of weights and activation gradients, memory usage can be reduced by roughly half compared to BF16 precision, enabling more efficient scaling of massive models.

A growing ecosystem of frameworks supports this approach, including Microsoft’s FP8-LM [139], NVIDIA’s Transformer Engine [40], and Amazon SageMaker [195], all of which provide flexible mechanisms for switching between precision levels depending on training requirements. The effectiveness of these approaches is illustrated by DeepSeek V3 [51], which successfully applies FP8 mixed-precision training to a 671 billion-weight MoE model, achieving strong performance while substantially lowering memory demands.

However, low-precision training introduces unique challenges in the MoE context. Compared to dense models, MoE architectures are more sensitive to numerical precision because of their dynamic gating mechanisms, where small differences in values can influence which experts are selected. Aggressive quantization may therefore amplify instability, leading to issues such as expert collapse, where only a subset of experts are consistently chosen, degrading model quality and reducing the benefits of sparsity.

Despite these limitations, low-precision training remains a highly promising direction. Its ability to drastically reduce memory consumption makes it a crucial component of modern MoE optimization strategies, particularly when combined with other techniques such as redundancy elimination, offloading, and activation recomputation. Thus, achieving efficient and stable MoE training at scale requires careful balancing between precision reduction and numerical robustness, making this an active and important area of ongoing research [51] [40] [139].

5.11 Communication Optimization

This section highlights how the intrinsic properties of MoE models, namely sparse activation and dynamic routing, create communication patterns that differ fundamentally from those of dense models. In traditional distributed training, communication is relatively regular and predictable, but in MoE systems it becomes data-dependent and highly dynamic, leading to inefficiencies if handled with conventional approaches. As a result, optimizing communication is essential to fully unlock the efficiency gains promised by MoE architectures. MoE-oriented communication optimization technologies will be divided into three key categories: All-to-All collective optimization, computation-communication overlap techniques, and communication compression methods (Figure 5.39).

5.11.1 Collective Communication Operations Redesign and Scheduling

A central challenge arises from the use of All-to-All collective communication, which is required to route tokens to the appropriate experts. In its naive form, often referred to as linear All-to-All, every device communicates simultaneously with all others, without considering differences in bandwidth between intra-node and inter-node links. This leads to poor utilization of high-bandwidth intra-node connections and to inefficient global data exchange [234]. Moreover, in hybrid parallel training configurations, All-to-All operations frequently overlap with other collective operations, such as AllReduce, during backpropagation, resulting in bandwidth contention and degraded performance [132]. These limitations make it clear that communication must be made resource-aware and topology-sensitive.

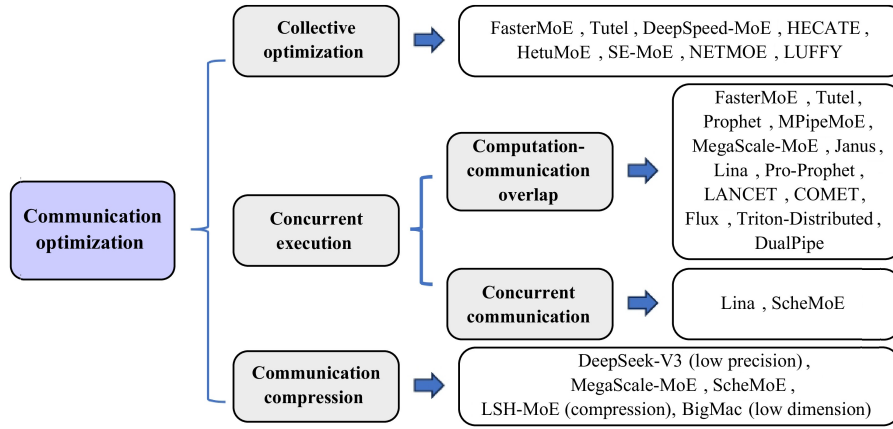


Figure 5.39: Communication optimization techniques taxonomy in the MoE model training context [228].

To address this problem, researchers have proposed hierarchical All-to-All communication schemes that explicitly account for hardware topology. For example, HetuMoE [150] introduces a one-dimensional (1D) topology-aware strategy, where data destined for GPUs within the same node is first aggregated, then it is transferred across nodes in bulk, and finally it is distributed locally. This approach reduces inter-node communication frequency while maximizing the use of fast intra-node links. Building on this idea, systems such as DeepSpeed-MoE [163] and SE-MoE [178] implemented two-dimensional (2D) All-to-All communication, decomposing communication into separate intra-node and inter-node phases. This decomposition allows large data chunks to be prepared efficiently within nodes before being transmitted across the network, improving overall throughput.

Further refinements extend these hierarchical principles. For example, FasterMoE [77] introduces group-based All-to-All, ensuring that intra-node communication completes quickly so that expert computation can begin earlier, thereby reducing idle time. Tutel [140] enhances the 2D approach through optimized data layouts and compiler-level improvements, minimizing synchronization overhead between communication stages. More advanced systems, such as HECATE [159], incorporate dynamic scheduling mechanisms that adapt communication patterns in real time based on network conditions and workload distribution, demonstrating the increasing importance of runtime adaptability in MoE systems.

Beyond collective operations redesign, it is also important to adopt locality-aware scheduling for reducing the communication volume altogether, rather than merely optimizing how it is performed. The key idea is to exploit the observation that certain experts are accessed more frequently or jointly by specific inputs. NETMOE [74] leverages this principle by strategically placing experts close to the data that they use most often, thereby minimizing cross-node communication. Similarly, LUFFY [25] extends this concept through dynamic expert migration, relocating experts during training based on evolving workload patterns. This adaptability is particularly valuable in heterogeneous cluster environments, where communication costs vary significantly across different links.

The reviewed literature shows that communication optimization in MoE systems requires a multi-layered approach: redesigning collective operations to respect hardware topology, scheduling them intelligently to avoid contention, and reducing communication demand through locality-aware strategies. These innovations collectively reveal that efficient MoE training is not only a matter of computational sparsity, but also of communication efficiency tightly coupled with system architecture and runtime behavior.

5.11.2 Concurrent Execution

We now focus on concurrent execution as a key strategy for improving communication efficiency during MoE training. A naive implementation typically executes communication and computation sequentially, as well as different communication collectives in isolation. Such serialized workflow leads to substantial inefficiencies, particularly in MoE models where communication costs, especially from All-to-All operations, are already significant due to dynamic gating.

To overcome this limitation, recent work emphasizes overlapping communication with computation and interleaving multiple communication operations. These approaches aim to better utilize available hardware resources by ensuring that idle time caused by one operation can be filled by another. For instance, in hybrid parallel MoE training, combining expert parallelism and expert sharding, collective operations such as AllGather or ReduceScatter can be executed concurrently with All-to-All communication [155]. This concurrency exploits the fact that different operations may use distinct hardware resources, for example, network bandwidth versus compute units, enabling resource complementarity and improving overall system throughput.

Another important observation is that communication operations themselves can interfere with each other. In particular, All-to-All communication may be delayed by concurrent AllReduce operations during the backward pass, leading to bandwidth contention and reduced efficiency. Addressing this issue requires careful scheduling to interleave communication tasks in a way that minimizes conflicts and maximizes parallelism. By reordering, overlapping, or pipelining communication stages, systems can reduce waiting times and avoid underutilization of high-bandwidth links.

The concurrent execution strategies demonstrate that communication optimization in MoE training is not limited to redesigning individual collectives, but also involves coordinating when and how different operations are executed. Through computation-communication overlap and intelligent interleaving, we can significantly improve training efficiency, transforming communication from a bottleneck into a more effectively managed component of large-scale MoE training [155].

5.11.3 Computation-Communication Overlap

The idea of computation-communication overlap is a central technique for improving the efficiency of MoE training systems, leveraging the inherent parallelism introduced by sparse expert routing. Unlike dense models, where computation typically follows strict sequential dependencies, MoE architectures allow simultaneous execution of multiple experts, making it possible to overlap communication, for example, token exchange across devices, with ongoing computation.

At the most basic level, task-level overlap treats communication and computation as separate coarse-grained tasks that can be scheduled concurrently. Systems such as FasterMoE [77] and Tutel [140] implement this overlap by assigning computation and communication to different CUDA streams, enabling asynchronous execution whenever data dependencies permit. Tutel further improves flexibility by introducing unified data layout abstractions that allow different parallelism strategies to coexist without costly data transformations. Similarly, MPipeMoE [234], inspired by micro-batching techniques from GPipe [89], enhances overlap by splitting minibatches into smaller units and interleaving communication and computation across adjacent microbatches (Section 4.1.1). These approaches demonstrate that even simple scheduling improvements can significantly increase hardware utilization.

More advanced predictive scheduling methods extend task-level overlap by exploiting temporal patterns in MoE routing. For example, Prophet [203] analyzes recurring token routing behavior to predict expert loads and proactively schedule weight transfers, while also overlapping expert-related AllReduce operations with non-expert communication. Pro-Prophet [204] further formalizes this idea by dividing optimization into an offline phase, which profiles activation distributions, and an online scheduler that maximizes overlap dynamically during execution. Janus [132] complements these ideas with a data-centric design in which experts are dynamically migrated between workers, coordinating computation with CPU-GPU communication to reduce idle time.

The scope of overlap optimization is further expanded by systems such as LANCET [100], which operate at the full computation graph level. By analyzing intermediate representations, LANCET identifies global opportunities for overlapping communication and computation that are invisible to layer- or task-level optimizers. DeepSeek-V3 [51] introduces DualPipe, a bidirectional pipeline parallelism strategy that achieves near-perfect overlap through careful decomposition of tasks into fine-grained chunks, such as attention, All-to-All dispatch, computation over feed-forward networks, and combining stages. Importantly, DualPipe also dynamically allocates GPU resources, such as streaming multiprocessors, between communication and

computation, demonstrating that effective overlap requires not only scheduling but also resource partitioning.

Beyond these coarse-grained strategies, operator-level overlap pushes optimization further by decomposing individual operators into smaller computational units. Instead of scheduling entire tasks, these approaches fuse computation and communication within the same kernel. However, naively running communication and computation kernels simultaneously may degrade performance, making kernel fusion and fine-grained decomposition essential. FLUX [22] exemplifies this approach by breaking operations into extremely fine tiles and merging communication and computation into unified kernels, eliminating synchronization overhead. Similarly, COMET [230] addresses mismatches in granularity between communication and computation by decomposing dependencies down to the tensor-element level, enabling tighter integration and improved accelerator utilization.

Compiler-based approaches, such as Triton-distributed [241], provide an even higher level of abstraction. By embedding communication primitives, such as OpenSHMEM [38], directly into the compiler, these systems allow developers to express combined communication-computation logic in a unified programming model, while the compiler automatically generates optimized fused kernels. This represents a shift toward automated, compiler-driven optimization, reducing the burden on system designers.

Finally, the most advanced systems adopt hybrid approaches that combine task-level and operator-level optimizations. For instance, Lina [117] integrates predictive scheduling of communication at the task level with fine-grained micro-operation decomposition at the operator level, enabling precise control over overlap. Similarly, MegaScale-MoE [101] employs a hierarchical overlap strategy, coordinating execution at both inter-operator (macro) and intra-operator (micro) levels, such as tile-based matrix-matrix multiplication decomposition. This multi-level design maximizes overlap opportunities across different granularities of execution.

A detailed comparison of these techniques is summarized in Table 5.5. The evolution we described reveals a clear trajectory, MoE systems have moved from simple asynchronous scheduling toward deeply integrated, multi-level overlap strategies. By combining task-level concurrency, predictive scheduling, operator fusion, and compiler support, it is possible to reach near-optimal hardware utilization. This progression underscores a key insight, achieving high performance in MoE training requires not just faster communication or computation individually, but careful coordination between them across all levels of system design.

5.11.4 Concurrent Collective Communication

We now deepen the communication optimization discussion by focusing on concurrent collective communication, addressing inefficiencies that arise when multiple communication stages are executed strictly in sequence. In MoE training, especially under hybrid parallelism, the communication pattern becomes increasingly complex, combining multiple collective operations such as All-to-All and AllReduce. While this complexity creates challenges, it also opens up additional opportunities for concurrency and scheduling optimization.

A key limitation of traditional All-to-All implementations is their serial execution model, where different phases, such as intra-node and inter-node communication, are carried out one after another, even though they are not inherently dependent. Recognizing this inefficiency, recent approaches propose decoupling these stages into independent tasks. For example, ScheMoE [183] separates intra-node and inter-node communication and executes them concurrently using distinct CUDA streams. Because intra-node communication relies on high-bandwidth local interconnects while inter-node transfers depend on slower network links, this decoupling allows intra-node latency to be effectively hidden behind inter-node communication, improving overall efficiency without disrupting computation.

However, introducing concurrency at the level of collective operations also exposes a new bottleneck: bandwidth contention between overlapping communication operations. In particular, during the backward pass, All-to-All and AllReduce operations frequently compete for the same network resources, leading to degraded performance despite the concurrency. This reveals that simply enabling parallel execution is not sufficient, careful coordination and prioritization are required to ensure efficient resource utilization.

Work	TLO	OLO	CO	EP	RA	Core technical innovation
FasterMoE [77]	V	X	X	X	V	Asynchronous sub-task execution with dedicated CUDA streams for communication and computation, enabling out-of-order execution when possible.
Tutel [140]	V	X	V	X	V	Unified data layout abstraction that maintains compatibility across DP/EP/hybrid configurations, enabling transparent parallelism switching.
MPipeMoE [235]	V	X	X	X	V	Online algorithm to adaptively configure pipeline granularity for optimal communication-computation overlap in MoE sub-stages, coupled with performance-model-guided runtime selection.
Prophet [203]	V	X	X	V	X	Exploits temporal locality in token routing patterns to predict expert loads and proactively schedule weight transfers across iterations, while restructuring gradient aggregation to overlap expert All-Reduce with non-expert layer communications.
Pro-Prophet [204]	V	X	X	V	V	Decomposes load balancing into offline planner (profiling inter-iteration distributions) and online scheduler (maximizing overlap within iterations), enabling data-dependent optimization.
Lina [117]	V	V	X	V	V	Discovers temporal locality in expert activations through pattern analysis, enabling predictive prefetching and priority-based collectives scheduling.
Janus [132]	V	X	X	V	V	Implements data-centric framework where experts migrate on-demand between workers, coordinating asynchronous CPU-GPU transfers with computation scheduling to minimize idle time.
LANCET [100]	V	X	V	V	X	Extends overlap optimization from individual MoE layers to entire computation graph through compiler IR analysis, discovering global overlap opportunities invisible to local optimizers.
FLUX [22]	X	V	V	X	V	Introduces aggressive over-decomposition paradigm, partitioning operations into much finer tiles than conventional approaches and fusing tiled communication with computation into unified kernels.
MegaScale-MoE [101]	V	V	X	X	V	Implements hierarchical overlap at inter-operator level through macro module execution, and at intra-operator level through tile-based GEMM decomposition enabling concurrent execution of communication and computation kernels.
DualPipe [50]	V	X	V	X	V	Near-perfect overlap through bidirectional pipeline with fine-grained chunk decomposition and manual SM allocation between comm./comp.
COMET [230]	X	V	V	X	X	Decomposes coarse computation and communication operation dependencies into fine-grained data dependencies, enabling kernel fusion at tensor element granularity rather than operation level.
Triton-distributed [241]	X	V	V	X	V	Integrates OpenSHMEM primitives directly, into Triton compiler enabling to express complex joint optimizations of computation, memory access, and communication through high-level Python.

Table 5.5: Comparative analysis of computation-communication overlap technologies in the MoE training context. TLE: task-level overlap, OLO: operator-level overlap, CO: compiler optimization, EP: expert prediction, RA: resource allocation, V: supported, X: not supported [228].

To address this issue, Lina [117] provides a systematic analysis of communication contention and introduces a fine-grained scheduling framework based on tensor partitioning. In this approach, large communication operations are decomposed into smaller, uniform micro-operations, enabling priority-based scheduling at a much finer granularity. This allows the system to allocate bandwidth selectively, for example, giving priority to latency-critical All-to-All operations while allowing AllReduce operations to opportunistically use remaining bandwidth. Such a strategy transforms communication from a set of coarse, competing operations into a flexibly scheduled pipeline, where resource allocation can be dynamically adapted to workload demands.

Thus, concurrent collective communication represents an important evolution in MoE systems design. It shows that performance gains are no longer achieved solely by optimizing individual communication primitives, but by orchestrating multiple collectives simultaneously and intelligently managing shared network resources. Together with earlier advances in hierarchical communication and overlap techniques, these methods demonstrate that maximizing MoE training efficiency requires a holistic view of communication, where concurrency, prioritization, and hardware-aware scheduling are tightly integrated.

5.11.5 Communication Compression

As a final type of communication optimization, we address communication compression as a complementary strategy for reducing the overhead of data exchange in MoE training. We know that All-to-All communication can involve millions of bytes per operation [183], making it a major bottleneck even when optimized through scheduling and overlap. From a data-centric perspective, reducing the volume or size of transmitted information becomes a natural solution. However, MoE training imposes stricter constraints than traditional distributed learning: compression must preserve not only numerical fidelity but also token semantics and routing decisions, limiting the applicability of many standard techniques.

One straightforward approach is volume compression, where data is compressed before transmission and decompressed afterward. ScheMoE [183] follows this paradigm by introducing abstractions that allow low-bit representations and compression algorithms to be seamlessly applied within MoE layers. More sophisticated methods exploit redundancy in routed data. For example, LSH-MoE [147] observes that tokens routed to the same experts often share structural similarities, and applies Principal Component Analysis (PCA) to cluster data and transmit only representative centroids instead of full embeddings. This reduces communication volume while retaining essential information.

Another important direction is low-dimensional communication, which reduces tensor size rather than compressing values directly. BigMac [102] implements this idea through dimension-compressed communication adaptation, where tensors are projected into lower-dimensional spaces before the first All-to-All operation and reconstructed afterward. By inserting learnable projection layers at expert boundaries, BigMac achieves substantial latency reductions, up to 3.09 times, while preserving model quality, demonstrating that dimensionality reduction can be an effective alternative to traditional compression.

A widely adopted and highly effective technique is low-precision communication, where data is transmitted using fewer bits. Quantization plays a central role here. QMoE [63] shows the extreme potential of this approach by compressing the 1.6 trillion-weight switch transformer [62] from 3.2 TB to 160 GB, less than one bit per weight, highlighting the enormous compression capacity of MoE models. In practice, more moderate strategies are often used to balance efficiency and stability. For instance, MegaScale-MoE [101] reduces inter-node communication precision from FP32 to BF16, halving bandwidth requirements without harming convergence, while DeepSeek-V3 [51] adopts FP8 precision for expert communication, achieving similar reductions.

Despite these advances, most existing systems still follow a compress-communicate-decompress pipeline, where decompression is required before computation can proceed. This introduces latency on the critical path and limits the overall benefits of compression. To tackle this limitation, recent research proposes compression-aware communication, a paradigm shift inspired by database systems such as TADOC [227] and CompressDB [226], where operations are performed directly on compressed data. Applying this idea to MoE systems would allow activations to remain compressed throughout both communication and computation, eliminating decompression overhead and redefining compressed data as an active computational representation.

rather than a passive encoding.

Current implementations, such as DeepSeek-V3 [51], partially move in this direction by combining low-precision communication with selective full-precision computation for sensitive operations. However, they still treat compressed formats primarily as transmission optimizations. Fully realizing compression-aware communication, where computation natively operates on compressed representations, remains an open and promising research direction.

In conclusion, we have shown that communication compression in MoE systems evolved along three axes: reducing data volume through compression and clustering, reducing dimensionality, and reducing precision. The future will involve integrating compression directly into the computational workflow.

5.12 Open Source MoEs

5.12.1 MegaBlocks

MegaBlocks is a system for efficient Mixture-of-Experts training on GPUs [65]. The motivation was the limitations of available frameworks, which restrict the dynamic gating in MoE layers to satisfy the constraints of used software and hardware. For example, TPUs and their XLA compiler require all tensor shapes to be known statically and often struggle with fine-grained operations like scatters and gather. GPUs are more flexible, but the sparse computation in MoEs does not map cleanly to the software primitives supported in major libraries. To tackle these limitations, the authors of MegaBlocks reformulated MoE computation in terms of block-sparse operations and developed block-sparse GPU kernels that efficiently handle the dynamism present in MoEs. This approach never drops tokens and maps efficiently to modern GPUs, enabling end-to-end training speedups of up to 40% and 2.4 times over state-of-the-art frameworks for MoE and deep neural networks training, respectively.

MoE layers are made up of many experts, which are them-selves small neural networks. Each token is dynamically routed to a subset of the experts for computation based on scores computed by a gating network. The experts are commonly defined to be small multi-layer perceptron (MLP). Figure 5.40 illustrate an MoE layer.

The first stage of an MoE layer is the router, which is responsible for assigning tokens to experts. The router also produces probabilities for each assignment that reflect the confidence of the mapping. These probabilities are encoded as a matrix of scores for each token-expert pair, which are used to linearly combine the top k expert outputs for each token.

The second stage is denominated as permutation. State-of-the-art MoE implementations aim to compute all expert layers in parallel for making effective use of the parallelism available on GPUs. Has we have discussed before, the key operation used to implement experts' logic is batched matrix multiplication (BMM). However, implementing MoE computation with BMM of the same shape, the experts must be constrained to have weight matrices of the same shape and the number of tokens assigned to each expert must be equal. In order to satisfy this constraint, some authors defined a fixed expert capacity, which is the number of tokens that each expert can be assigned. If the tokens assigned to an expert exceeds its capacity, the extra tokens are dropped. Thus, the main role of the permutation stage is to group tokens assigned to each expert and drop tokens that exceed the expert capacity.

Once the data has been permuted, the experts can be computed in parallel (third stage). When the experts are MLPs, the computation associated to all experts in a layer is done with a batched matrix multiplication. For convolutional experts, the layers can be computed with grouped convolutions.

After the experts are computed, the output vectors are unpermuted for restoring the ordering used in the layer's input (fourth stage). The last step in MoE computation consists of scaling the output vectors by the token scores with which they were assigned to their respective experts. When a token is assigned to multiple experts, the multiple weighted vectors are summed to produce the final layer output for each token.

The authors of MegaBlocks wrote block-sparse CUDA kernels that perform the operations necessary to the forward and backward passes of the MoE layers. Some of primitive operations they support are identified by a three-letter pattern $L_3L_2L_1$, where L_3 specifies the type of output, L_2 specifies the type of the left input

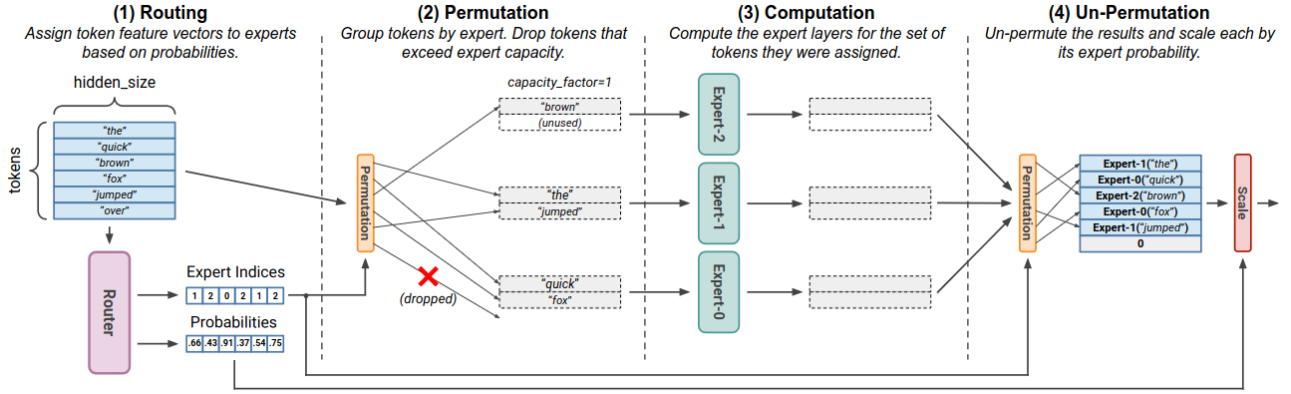


Figure 5.40: A mixture-of-experts layer. Shown for 3 experts, top- $k=1$, and a capacity factor of 1 with the prevalent, token dropping formulation. First, (1) tokens are mapped to experts by the router. Along with expert assignments, the router produces probabilities that reflect the confidence of the assignments. Second, (2) the feature vectors are permuted to group tokens by expert assignment. If the number of tokens assigned to an expert exceeds its capacity, extra tokens are dropped. Third, (3) the expert layers are computed for the set of tokens they were assigned as well as any padding needed for unused capacity. Lastly, (4) the results of the expert computation are unpermuted and weighted by the router probabilities. The outputs for dropped tokens are shown here set to zero [65].

operand and L_1 specifies the type of the right input operand. Each letter L_i can be "S" for sparse matrix or "D" for dense matrix. As an example, the product of two dense matrices with a sparse output is denominated as SDD, which is also referred to as sampled dense-dense matrix multiplication. They implemented SDD, DSD, and DDS operations targeting NVIDIA GPUs, and the kernels support all combinations of transposed and non-transposed inputs.

Considering a MoE feed-forward layer where each expert is a two-layer MLP, (i) the forward pass requires an SDD operation followed by a DSD (Listing 5.2), and (ii) the backward pass is implemented with an SDD^T and a $DS^T D$ for the second layer data gradient and weight gradient, respectively, followed by a $DS D^T$ and a $DD^T S$ for the first layer data gradient and weight gradient, respectively.

```

1 x.shape: (num_tokens, hidden_size)
2 def dmoe_forward(self, x):
3     # (1) Assign tokens to experts.
4     #
5     # indices.shape: (num_tokens)
6     # weights.shape: (num_tokens)
7     indices, weights = router(x)
8
9     # (2) Create the sparse matrix topology.
10    #
11    # This describes the matrix in Figure 27 (iii).
12    topology = make_topology(indices)
13
14    # (3) Permute the tokens to group by expert.
15    x = padded_gather(x, indices)
16
17    # (4): Compute the expert layers
18    #
19    # inner_dim = ffn_hidden_size * num_experts
20    # self.w1.shape: (hidden_size, inner_dim)
21    # self.w1.shape: (inner_dim, hidden_size)
22    x = sdd(x, self.w1, topology)
23    x = dsd(x, self.w2)
24
25    # (5) Unpermute and scale the tokens
26    x = padded_scatter(x, indices)
27    return x * weights

```

Listing 5.2: Pseudo-code for a two-layer feed-forward dMoE. The code follows Figure 5.40 with three changes. First, we construct the sparse matrix topology from Figure 5.27 (iii) from expert assignments (line 12). Second, we pad each expert batch to a multiple of the block size during permutation (line 15). Lastly, we compute the experts in parallel by iterating between SDD and DSD operations (lines 22-23) [65].

MegaBlocks uses blocked compressed sparse row (BCSR) as the primary sparse matrix format. BCSR makes it simple to iterate across the nonzeros in a row, which is necessary for operations like $DS D$ and DDS^T . Iterating over blocks also has minimal overhead with BCSR. One challenge with BCSR sparse matrices is efficiently computing SDD operations in parallel. On kernel launch, each thread block needs to identify the row and column of its output block so that it knows which rows and columns of the input matrices are needed to compute it. Because BCSR only encodes column indices for each block, identifying the row index of a nonzero block requires a search through the row offsets. Hence, to efficiently parallelize SDD , they additionally materialize the row indices for each nonzero block so that thread blocks can trivially look up the coordinates of sparse blocks in the output matrix.

Computing forward and backward passes for model training requires sparse matrix transposition. However, iterating over BCSR matrices in transposed order requires searching through each row to identify if the block in the target column is nonzero. To enable efficient iteration over BCSR matrices in transposed order, they construct the metadata for the transposed matrix but do not explicitly transpose the nonzero values. Instead, they construct an array of indices, one for each nonzero block, which are stored in transposed order and contain the offset of each nonzero block in memory. This additional metadata allows efficient iteration through the matrix in transposed order with a layer of indirection, as shown in Figure 5.41.

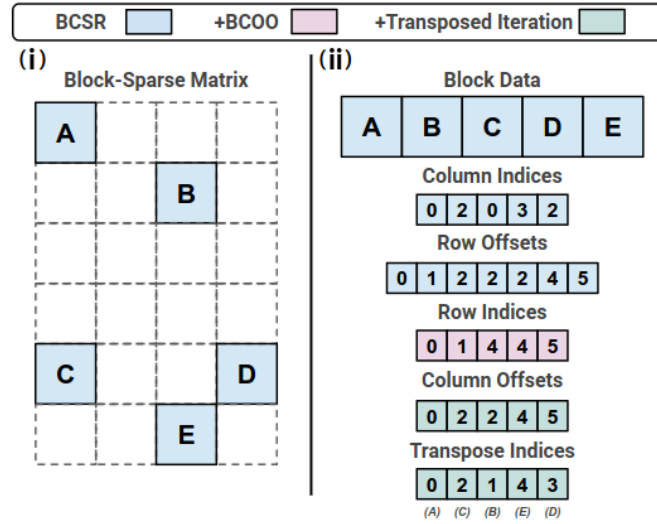


Figure 5.41: Block-sparse matrix format used in MegaBlocks. Panel (ii) shows the encoding for the sparse matrix in panel (i). Indices and offsets in our encoding are block-wise. We use blocked compressed sparse row (BCSR) as our primary sparse matrix format. We additionally store the row indices of each nonzero block and a secondary index of transpose indices [65].

Despite the use of load balancing losses, prior work has shown that token routing is still highly imbalanced. To quantify the effect of token dropping on model quality, the authors of [65] trained MoE models on The Pile dataset [68] with several capacity factors. They trained MoEs with capacity factor equal 1, 1.5, and 2 as well as the dynamic capacity factor, a technique proposed by Tutel [140]. The obtained results are shown in Figure 5.42. While dropping tokens reduces model quality, increasing the capacity factor brings additional computation and memory costs. Thus, these authors throw away the capacity factor hyperparameter. To replace the capacity factor, they just determined the block-sparse structure dynamically for each batch of tokens, doing exactly the amount of work required for each expert.

To assess the efficiency of MegaBlocks technique for avoiding token dropping, they compared it to the dMoE method proposed in Tutel [140], where the capacity factor is set dynamically to the minimum value that avoids token dropping. For each model, Figure 5.43 plots the end-to-end training time and final loss achieved on a validation set. Compared to the padding-based approach for avoiding token dropping, MegaBlocks technique for adaptive MoE computation with block sparsity enables end-to-end training speedups of 1.38, 2.0 and 4.35 times for MoE-XS, MoE-Small, and MoE-Medium, respectively. In addition to computational overhead, the padding-based approach implemented in Tutel significantly increases the amount of memory required to store activations in the MoE layers.

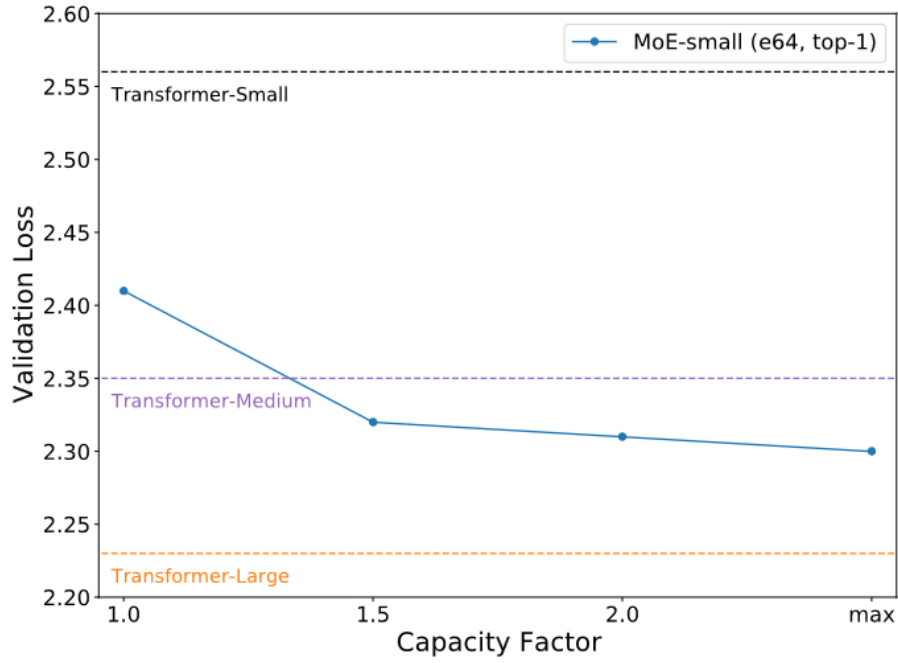


Figure 5.42: MoEs trained on The Pile dataset with different capacity factors [65]. The loss reached by the MoE models decreases significantly as expert capacity is increased, but at the cost of additional computation. The lowest loss is achieved by the "max" capacity factor model, which avoids dropping tokens through the dynamic capacity factor mechanism proposed in [140]

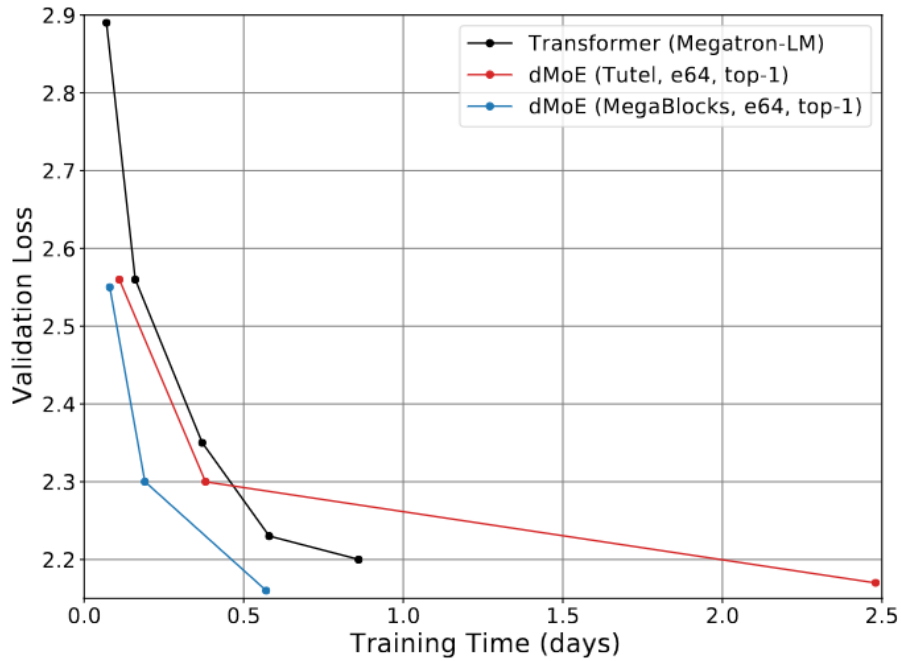


Figure 5.43: MegaBlocks dMoEs, Tutel dMoEs and Megatron-LM transformers trained on The Pile dataset. MegaBlocks uses block-sparse operation to handle the dynamic and load imbalanced computation in MoEs, which enables 1.38, 2.0 and 4.35 times end-to-end training speedups for MoE-XS, MoE-Small, and MoE-Medium respectively compared to the padding-based approach used by Tutel. The advantage of our approach increases with the size of the model, as the memory requirements of padding expert batches forces Tutel to use smaller micro batch sizes which decreases hardware efficiency. Compared to dense transformer language models, MegaBlocks achieves 1.8 to 2.44 times end-to-end training speedups for the same validation loss across these models [65].

Even with the most efficient capacity factor for each related MoE, MegaBlocks reduces the training time required to reach a given validation loss 1.38, 1.37 and 1.18 times, compared to MoE-XS, MoE-Small and MoE-Medium, respectively. In addition to these speedups, the MegaBlocks approach reduces the cost of using MoEs by decreasing the number of hyperparameters that need to be re-tuned for each model and task (Figure 5.44).

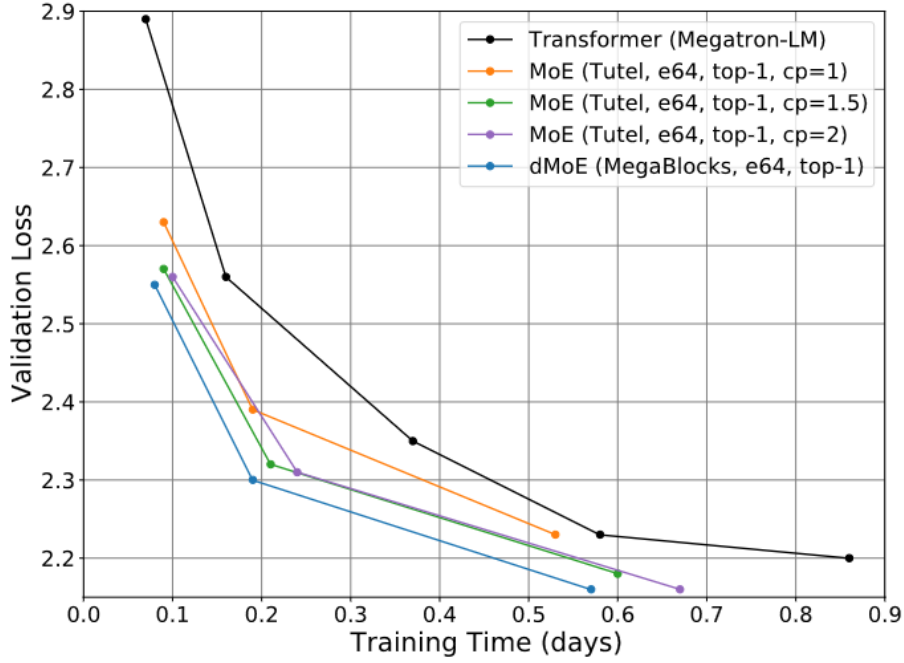


Figure 5.44: MegaBlocks dMoEs, Tutel MoEs and Megatron-LM transformers trained on The Pile dataset. Even with the most efficient capacity factor for each MoE, MegaBlocks reduces the training time required to reach a given validation loss by 1.38, 1.37 and 1.18 times for MoE-XS, MoE-Small and MoE-Medium respectively. In addition to these speedups, our approach reduces the cost of using MoEs by decreasing the number of hyperparameters that need to be re-tuned for each model and task [65].

5.12.2 Mixtral of Experts

Mixtral $8 \times 7B$, also known as Mixtral of Experts, is an MoE-based extension of the open-source Mistral-7B model [98] that is fluent in English, French, Italian, German, and Spanish.

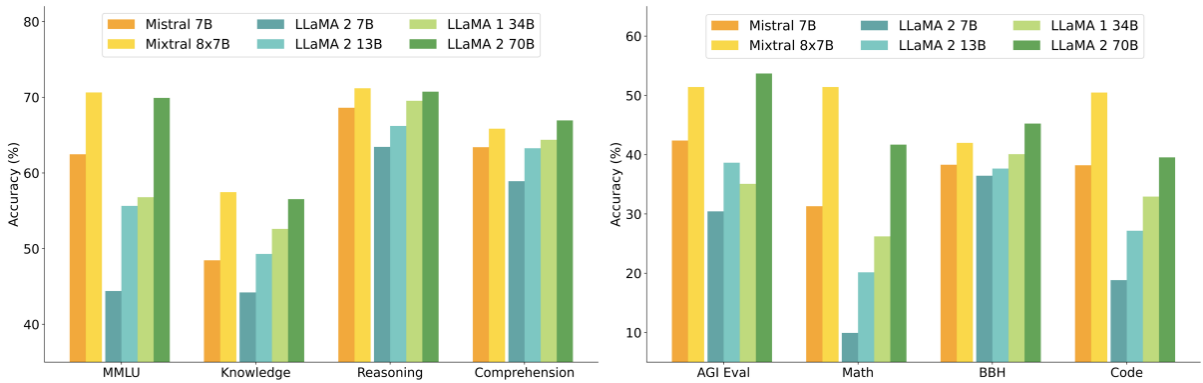


Figure 5.45: Performance of Mixtral and different LLaMa models on a wide range of benchmarks. Mixtral outperforms or matches LLaMa-2 70B on all benchmarks. In particular, it is vastly superior in mathematics and code generation [99].

Mixtral converts every layer of Mistral dense model to an expert layer with eight experts. Two of these experts are active for each token, yielding a model with 47 billion total weights and 13 billion active weights. The model has a context length of 32k, which is 4 times larger than its dense counterpart. As shown in Figure 5.45, Mixtral outperforms Mistral on several benchmarks and especially excels in code generation, mathematics, and multilingual benchmarks, even exceeding the performance of the larger LLaMA-2-70B model in some cases.

The base architecture of Mixtral $8 \times 7B$ is a decoder-only transformer that exactly matches the architectural configuration of Mistral-7B [98]. Compared to the standard decoder-only LLM architecture, there are a few changes made by Mistral-7B: (i) adopted grouped-query attention (GQA) [2], which shares key and value projections between groups of self-attention heads to improve efficiency (Figure 5.46), and (ii) uses sliding window attention (SWA) [10], which computes masked self-attention over a fixed window of size W , for each token, to allow the LLM to handle sequences of arbitrary length with a reduced inference cost (Figure 5.47).

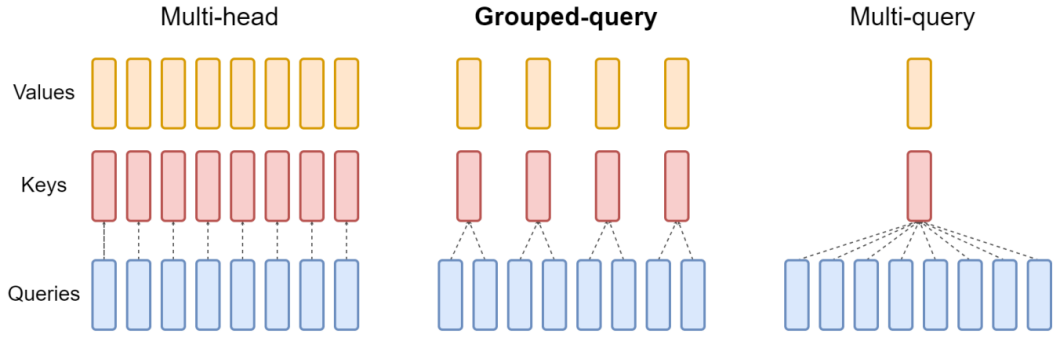


Figure 5.46: Overview of grouped-query method. Multi-head attention has H query, key, and value heads. Multi-query attention shares a single key and the value heads across all query heads. Grouped-query attention instead shares single key and value heads for each group of query heads, interpolating between multi-head and multi-query attention [2].

While regular self-attention can be seen as a global attention mechanism, where each element in a sequence can attend to every other sequence element, sliding window attention (SWA) can be viewed as local attention, since here we restrict the context size around the current query position. Sliding window attention can be combined with both multi-head attention and grouped-query attention. In causal self-attention, the current token can attend to all previous tokens and itself (Figure 5.47 (i)). With sliding window attention, the current token can only attend to itself and previous tokens within a certain window, for example, itself and $W - 1 = 4 - 1 = 3$ tokens in Figure 5.47 (ii). In Figure 5.47 (iii), it is illustrated the effective context for the 10-th element in the sequence on layer 4. Thus, previous tokens that are outside the sliding window still influence next word prediction. At each attention layer, information influences up to W positions ahead. Hence, after l attention layers, one position is influenced by information relative to the previous $l \times (W - 1)$ tokens [64].

The number of operations in regular attention is quadratic in relation to the sequence length, $O(S^2)$, and the memory increases linearly with the number of tokens. At inference time, this incurs in higher latency and smaller throughput due to reduced cache availability. To alleviate this issue, in sliding window attention the number of operations varies linearly with the sequence length, $O(W \cdot S)$, where W is the window of size.

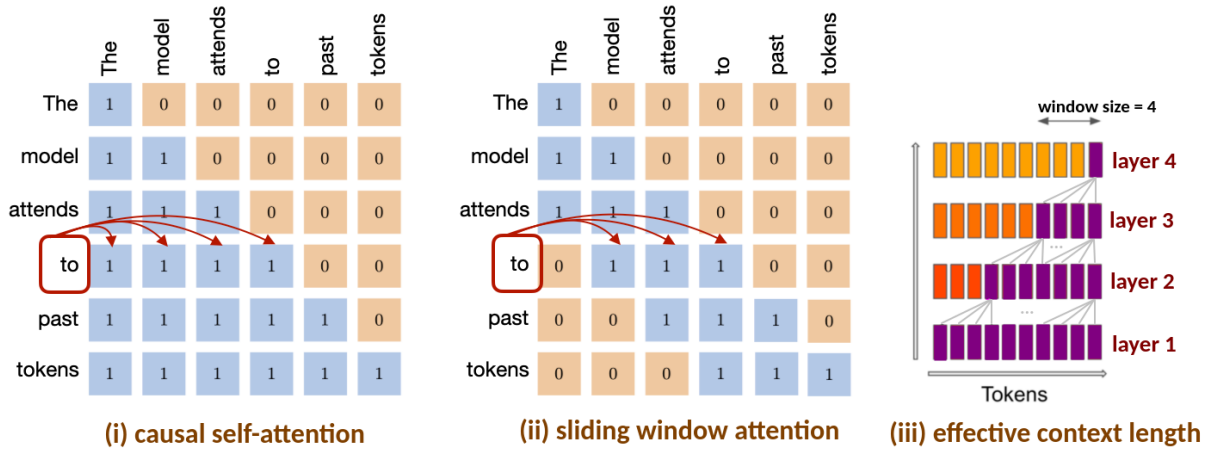


Figure 5.47: Illustration of (i) causal self-attention, (ii) sliding window attention, and (iii) effective context length [167] [98].

Because Mixtral uses SWA, it can employ tricks like (i) rolling buffer, also known as circular cache, to make the KV cache more memory efficient or (ii) chunked prefill, to increase inference speed. As mentioned previously, Mixtral converts every layer of the LLM to an expert layer (Figure 5.48). Within each expert layer, a simple routing mechanism is adopted that takes a softmax over the Top-K logits of a Linear layer for every token. Each input vector is assigned to 2 of the 8 experts by the router. The layer's output is the weighted sum of the outputs of the two selected experts. In Mixtral, an expert is a standard feed-forward block as in a vanilla transformer architecture.

Authors of [99] mention that Mixtral is also pretrained over a multilingual corpus, allowing the model to understand multiple languages. As shown in Table 5.6, Mixtral universally outperforms LLaMA models on

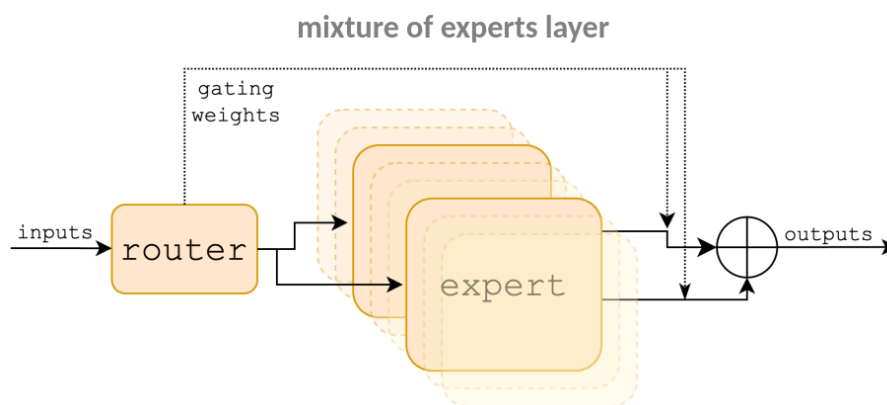


Figure 5.48: Mixture of experts layer [99].

multi-lingual benchmarks.

Model	Active Params	French			German			Spanish			Italian		
		Arc-c	HellaS	MMLU	Arc-c	HellaS	MMLU	Arc-c	HellaS	MMLU	Arc-c	HellaS	MMLU
LLaMA-1 33B	33B	39.3%	68.1%	49.9%	41.1%	63.3%	48.7%	45.7%	69.8%	52.3%	42.9%	65.4%	49.0%
LLaMA-2 70B	70B	49.9%	72.5%	64.3%	47.3%	68.7%	64.2%	50.5%	74.5%	66.0%	49.4%	70.9%	65.1%
Mixtral 8x7B	13B	58.2%	77.4%	70.9%	54.3%	73.0%	71.5%	55.4%	77.6%	72.5%	52.8%	75.1%	70.9%

Table 5.6: Comparison of Mixtral with LLaMa on multilingual benchmarks. On ARC Challenge, Hellaswag, and MMLU, Mixtral outperforms LLaMa-2 70B on four languages: French, German, Spanish, and Italian [99].

These authors also performed a detailed analysis of how experts are selected for tokens across several domains to see if any interpretable pattern can be deduced. Figure 5.49 plots the proportion of tokens assigned to each expert on different domains from The Pile dataset for layers 0, 15, and 31. Here, we consider experts that are either selected as a first or second choice by the router. The gray dashed vertical line corresponds to $\frac{1}{\text{number of experts}} = \frac{1}{8}$, the proportion expected with uniform assignment. Since the proportions oscillate around $\frac{1}{8}$, means that there are no obvious patterns in token assignment.

However, the MoE does exhibit some structured behavior. For example, the words "self" in Python code and "Question" in English, even though they are comprised of multiple tokens, often get routed through the same expert. Similarly, indentation tokens in code are usually sent to the same expert, and consecutive sequences of tokens that are close to each other, generally are sent to the same expert (Figure 5.50). These results indicate that (i) experts are not specialized by topic but (ii) the MoE's routing mechanism does obey some structured behavior with respect to the syntax or contents of the model's input.

After Mixtral, a larger version of the model was released, called Mixtral-8x22B. This model has 141 billion weights and 39 billion active weights, making it approximately three times larger than the original Mixtral model. Mixtral-8x22B is especially capable on coding and mathematics tasks, has an expanded context length of 64k, and is natively capable of function calling. The key benefits of Mixtral-8x22B compared to other open models are summarized in Figure 5.51.

5.12.3 Grok

Although there are no technical details about the model, one of the most notable recent examples of MoE-based LLMs is xAI's Grok. The initial Grok-1 model was released in early 2024. Researchers indicated that the model is a 314 billion weight MoE with 25% of weights active for each token, meaning that on average there are around 70-80 billion active weights. The architecture and base model weights of Grok-1 were open-sourced. However, this is a pretrained base model, and no details were provided on the model's post training process.

Shortly after the initial release of Grok-1, a follow-up version of the model was published with better reasoning and longer context understanding capabilities. For example, Grok-1.5 [216] performs much better on mathematics and coding-related tasks (Table 5.7). Grok-1.5 can process sequences up to 128k tokens with perfect retrieval on the Needle In A Haystack (NIAH) test (Figure 5.52). Authors also noted that the model

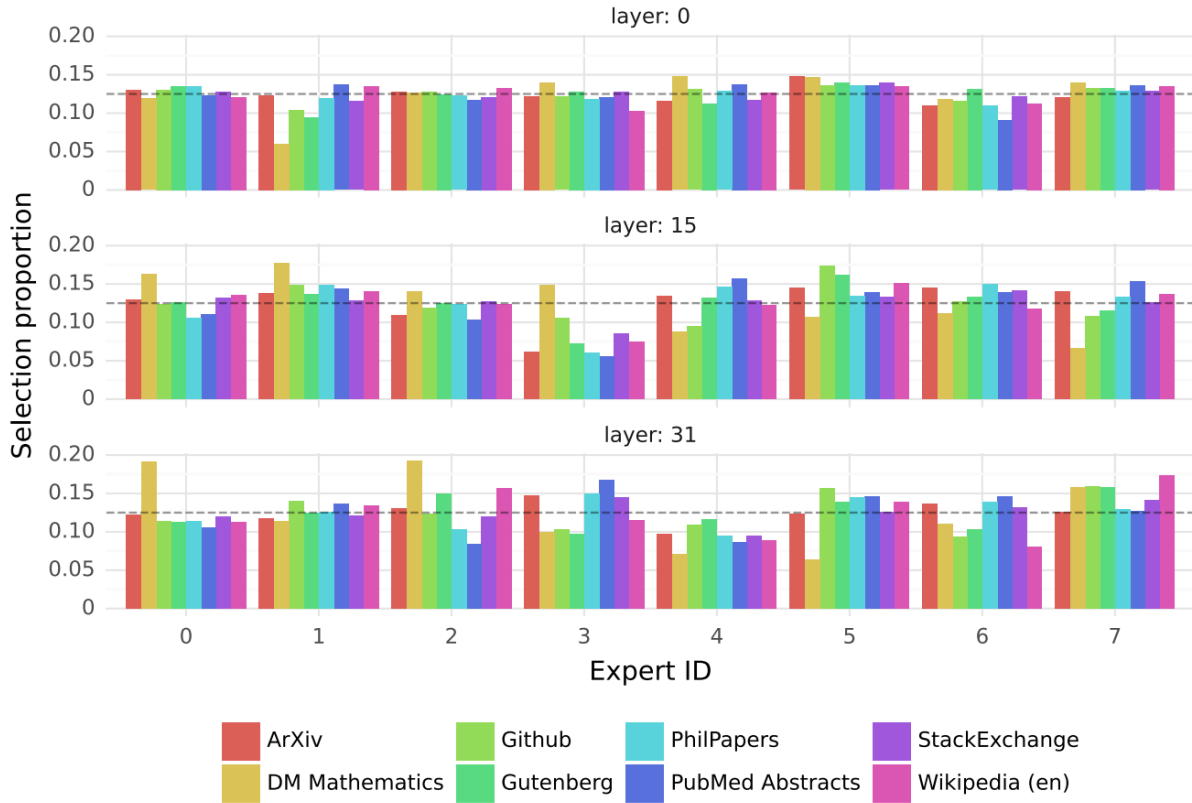
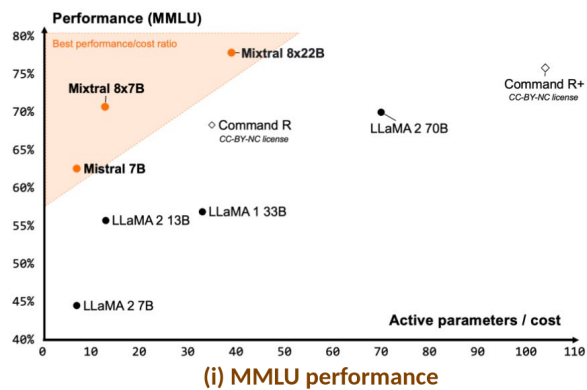


Figure 5.49: Proportion of tokens assigned to each expert on different domains from The Pile dataset for layers 0, 15, and 31. The gray dashed vertical line marks $\frac{1}{8}$, i.e. the proportion expected with uniform sampling. Here, we consider experts that are either selected as a first or second choice by the router [99].



Figure 5.50: Text samples where each token is colored with the first expert choice. The selection of experts appears to be more aligned with the syntax rather than the domain, especially at the initial and final layers [99].



Model	Active parameters	Common sense and reasoning					Knowledge	
		MMLU	HellaS	WinoG	Arc C (5)	Arc C (25)	TriQA	NaturalQS
LLaMA 2 70B	70B	69.9%	87.1%	83.2%	86.0%	85.1%	77.57%	35.5%
Command R	35B	68.2%	87.0%	81.5%	-	66.5%	-	-
Command R+	104B	75.7%	88.6%	85.4%	-	71.0%	-	-
Mistral 7B	7B	62.47%	83.1%	78.0%	77.2%	78.1%	68.8%	28.1%
Mistral 8x7B	12.9B	70.63%	86.6%	81.2%	85.8%	85.9%	78.4%	36.5%
Mistral 8x22B	39B	77.75%	88.5%	84.7%	91.3%	91.3%	82.2%	40.1%

(ii) reasoning capacity

Model	French			German			Spanish			Italian		
	Arc-C	HellaS	MMLU	Arc-C	HellaS	MMLU	Arc-C	HellaS	MMLU	Arc-C	HellaS	MMLU
Mistral 7B	44.2%	63.9%	50.6%	39.7%	58.4%	49.6%	43.9%	64.8%	51.4%	41.2%	60.8%	51.3%
Mistral 8x7B	54.3%	76.0%	66.1%	52.7%	71.0%	64.9%	53.7%	76.3%	67.5%	51.1%	72.9%	65.9%
Mistral 8x22B	59.1%	79.2%	75.1%	57.1%	75.9%	74.1%	58.4%	79.9%	75.7%	55.3%	77.1%	75.8%
LLaMA 2 70B	49.9%	72.5%	64.3%	47.3%	68.7%	64.2%	50.5%	74.5%	66.0%	49.4%	70.9%	65.1%

(iii) Multi-lingual

Model	Active parameters	Coding		Maths		
		HumanE	MBPP	GSM8K maj@1 (5-shot)	GSM8K maj@8 (8-shot)	Math maj@4
LLaMA 2 70B	70B	29.3%	49.8%	53.6%	69.6%	13.8%
Command R	35B	-	-	56.6%	-	-
Command R+	104B	-	-	70.7%	-	-
Mistral 7B	7B	26.22%	50.2%	36.5%	50.0%	12.7%
Mistral 8x7B	12.9B	40.2%	60.7%	58.4%	74.4%	28.4%
Mistral 8x22B	39B	45.1%	71.2%	78.6%	88.4%	41.8%

(iv) mathematics and coding capabilities

Figure 5.51: Comparison of Mistral and LLaMa-2 models through diverse metrics. (i) Measure of the massive multitask language understanding (MMLU) performance versus inference budget trade-off, that is, the number of active weights. Mistral 7B, Mistral 8x7B and Mistral 8x22B all belong to a family of highly efficient models compared to the other open models. (ii) Performance on widespread common sense, reasoning and knowledge benchmarks of the top-leading LLM open models: MMLU, HellaSwag (10-shot), Wino Grande (5-shot), Arc Challenge (5-shot), Arc Challenge (25-shot), TriviaQA (5-shot) and NaturalQS (5-shot). (iii) Comparison of Mistral models and LLaMA 2 70B on HellaSwag, Arc Challenge and MMLU in French, German, Spanish and Italian. (iv) Performance on popular coding and mathematics benchmarks of the leading open models: HumanEval pass@1, MBPP pass@1, GSM8K maj@1 (5 shot), GSM8K maj@8 (8-shot) and Math maj@4 [198].

Benchmark	Grok-1	Grok-1.5	Mistral Large	Claude 2	Claude 3 Sonnet	Gemini Pro 1.5	GPT-4	Claude 3 Opus
MMLU	73%	81.3%	81.2%	75%	79%	83.7%	86.4%	86.8%
MATH	5-shot	5-shot	5-shot	5-shot	5-shot	5-shot	5-shot	5-shot
GSM8K	23.9%	50.6%	—	—	40.5%	58.5%	52.9%	61%
HumanEval	4-shot	4-shot	81%	88%	92.3%	91.7%	92%	95%
	62.9%	90%	5-shot	0-shot CoT	0-shot CoT	11-shot	5-shot	0-shot CoT
	8-shot	8-shot	45.1%	70%	73%	71.9%	67%	84.9%
	0-shot	0-shot	0-shot	0-shot	0-shot	0-shot	0-shot	0-shot

Table 5.7: Comparison of Grok model with related models on four benchmarks [216].

maintains solid instruction-following capabilities when given a lot of context, which is a much better sign of long context capabilities compared to pure retrieval augmented generation (RAG).

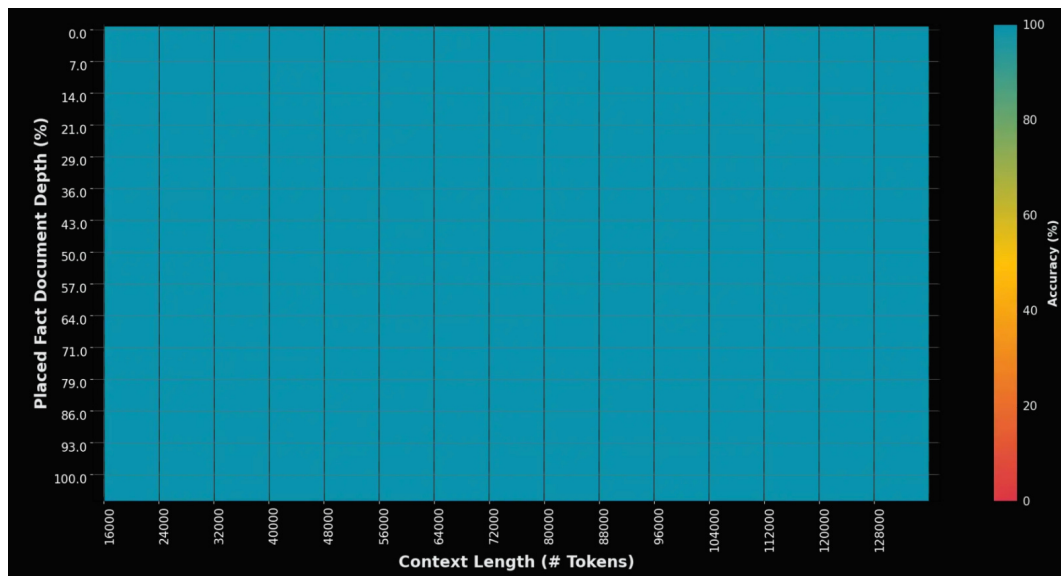


Figure 5.52: Retrieval performance of Grok-1.5 on the Needle In A Haystack (NIAH) test [216].

Given that Grok-1.5 and Grok-1 were released in such close succession, we can infer that the advancements made by Grok-1.5 are driven by post training, it is extremely unlikely that a different pretrained base model was created during this time. More recently, Grok-2 was released, which has improved reasoning, coding, and chat capabilities, as measured by Chatbot Arena. Grok-2 also has a variety of other small improvements, for example, tool usage, retrieval, or factuality, and a distilled version of Grok-2, called Grok-2-mini, was released with the main model. However, no public details were shared about the architecture of Grok-2, the model was likely trained from scratch and we are not sure if it is a MoE.

5.12.4 DBRX

DBRX is the latest model in the series of Open LLMs released by Mosaic. Two versions of the model were released, a base model (DBRX base) and a fine-tuned model (DBRX Instruct). DBRX is an MoE-based LLM with the following specifications: 132 billion total weights with 36 billion active weights, 16 experts in each MoE layer with 4 experts active for each token, pretrained on 12 trillion tokens of optimized text, and 4 times improvement in pretraining efficiency. Figure 5.53 compares DBRX with related open source models, and it outperforms other models on language understanding, programming, and mathematics.

Most notably, DBRX is a fine-grained MoE model. In other words, the model uses a larger number of experts in each MoE layer, but each individual expert is smaller. For reference, both Mixtral and Grok-1 contain eight experts, two of which are active for any given token, within each of their MoE layers. By using fine-grained experts, each MoE layer has more expert combinations, 65 times more in particular, to choose from, which was found to improve quality [45].

The pretraining dataset for DBRX is very large, but authors of [45] also invest significantly into improving the quality of the data. As a result, the statistical training efficiency of DBRX is higher than normal, training is faster because we achieve higher accuracy with fewer tokens. More specifically, the authors estimate that the new data is twice more efficient token-for-token, meaning that we can train over half tokens and achieve the same level of performance. This claim was verified by testing the impact of the new model's pretraining data in isolation, i.e., using a fixed model with different pretraining data.

The authors concluded that better pretraining data made a substantial impact on model quality. They trained a 7B model on 1T tokens using the DBRX pretraining data, and it reached 39.0% on the Databricks Gauntlet benchmark, a better result than the 30.9% obtained by the MPT-7B model.

Additionally, curriculum learning was used to train DBRX, where a mixture of pretraining data is dynamically

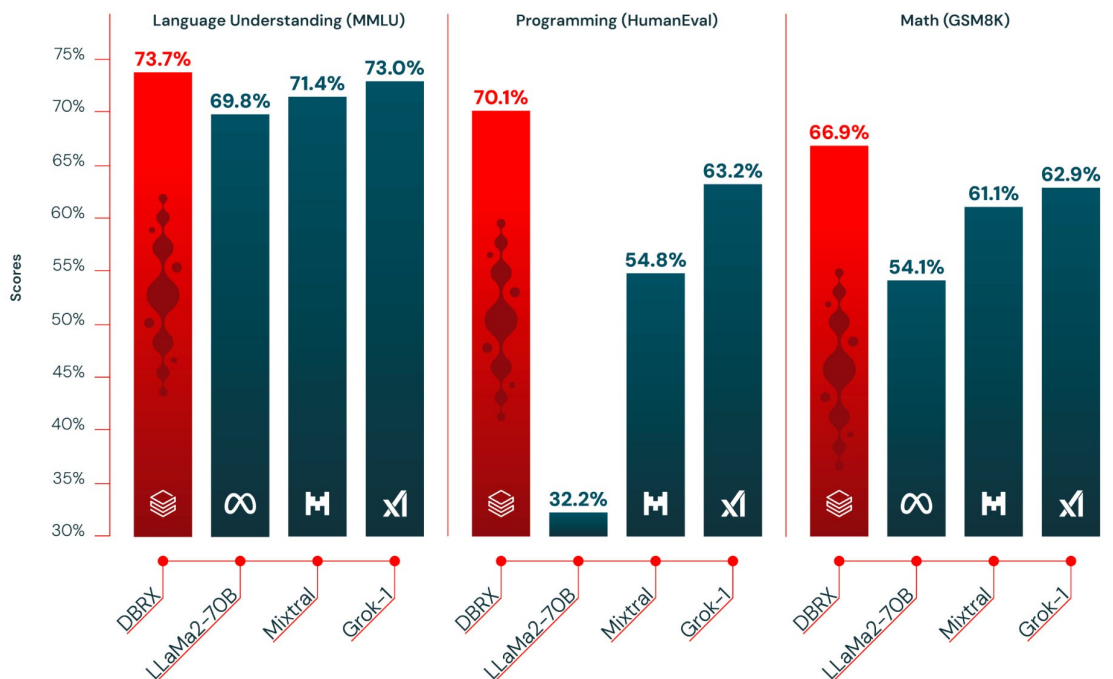


Figure 5.53: Comparison of DBRX with related open source models. DBRX outperforms other models on language understanding (MMLU), programming (HumanEval), and mathematics (GSM8K) [45].

changed throughout the pretraining process. The details of this curriculum learning strategy were later outlined in [12]. The curriculum learning strategy used by DBRX just upsamples smaller, domain-specific datasets towards the end of training because this data is higher quality relative to data obtained via web-crawling. This simple curriculum learning strategy is found to provide a significant boost in performance on difficult benchmarks (Figure 5.54).

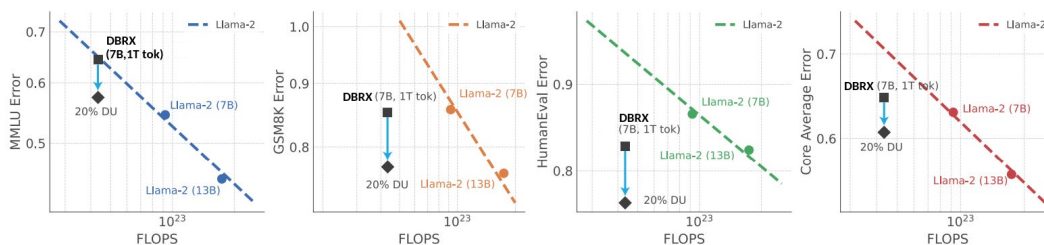


Figure 5.54: Comparison of the DBRX and LLaMa-2 performance. On key benchmarks, the DBRX 7B models trained with data mixture have errors at or below the error vs. FLOP scaling line of the LLaMa-2 family of models. The performance of the 7B, 13B, and 70B variants of the LLaMa-2 models was first evaluated on MMLU, GSM8K, HumanEval, and the Gauntlet v0.3 Core Average. Then, it is performed linear regression on the log of the error on these metrics vs. the log of the FLOPS used to train these models. This scaling relationship is plotted as a dashed line in the log-log plots. We can observe that the models in the LLaMa-2 family lie close to this scaling line. For MMLU, the DBRX models (square markers) lie on the LLaMa-2 scaling line. For the other metrics, the DBRX models are significantly below the scaling line [12].

DBRX has a context length of 32k and uses the GPT-4 tokenizer, available as `tiktoken`¹. According to the authors, the GPT-4 tokenizer was selected mostly due to performance. This tokenizer has a large vocabulary and is very token efficient, which naturally improves decoding and training speed by representing the same amount of text with fewer tokens.

The proposal of DBRX comes with large improvements in terms of pretraining efficiency. Beyond what we have learned about so far, there are several additional sources of efficiency gains mentioned in [45]: (i) the MoE architecture, which is found in smaller-scale experiments to require 1.7 times fewer FLOPS during training, (ii) other modifications to the decoder-only architecture, such as RoPE, GeLU activation, and GQA, and (iii) better optimization strategies.

¹<https://github.com/openai/tiktoken>

When considering all data, architecture, and optimization changes, the end-to-end training process for DBRX requires four times less compute when compared to the pretraining pipeline used for prior models. To determine this number, authors of [45] compare a smaller variants of DBRX to their prior MPT-7B model, finding that the smaller DBRX models achieve similar performance on the Databricks Gauntlet benchmark, while using 3.7 times fewer FLOPS during training (Table 5.53).

Model	Total weights	Active weights	Gauntlet score	Relative FLOPs
DBRX MoE-A	7.7B	2.2B	30.5%	1 ×
MPT-7B (1T tokens)	–	6.7B	30.9%	3.7 ×
DBRX Dense-A (1T tokens)	–	6.7B	39.0%	3.7 ×
DBRX Dense-A (500B tokens)	–	6.7B	32.1%	1.85 ×
DBRX MoE-B	23.5B	6.6B	45.5%	1 ×
LLaMA2-13B	–	13.0B	43.8%	1.7 ×

Table 5.8: Metrics for several test scenarios used to validate the training efficiency of the DBRX MoE architecture and end-to-end training pipeline [45].

DBRX also comes with improvements to inference efficiency, since it is up to two times faster than LLaMA-2-70B at 150 tokens per second per user in load tests. These measurements are made using an optimized serving infrastructure with TensorRT-LLM and 16 bit precision, which is very fast. The MoE architecture of DBRX also aids inference efficiency due to the relatively low number of active weights. For example, DBRX is 40% of the size of Grok-1 in both total and active weights (Figure 5.55).

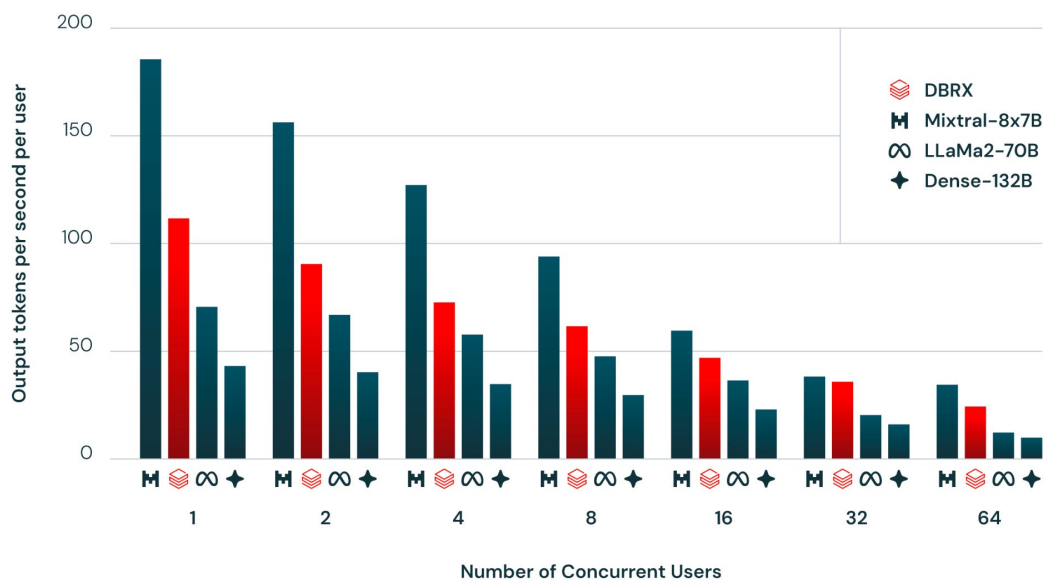


Figure 5.55: Inference throughput for various models on Databricks optimized serving infrastructure using NVIDIA TensorRT-LLM at 16-bit precision with the best optimization options. Models are run in tensor-parallel across the entire node. The input prompt contains approximately 2k prompt tokens and 256 output tokens are generated. One new user spawns every second [45].

Training MoEs is generally difficult due to instabilities that arise during training and communication bottlenecks. However, DBRX achieves impressive results in terms of stability, efficiency, and performance due to the optimized pretraining strategy outlined in [45]. The impressive pretraining pipeline used by DBRX is enabled by a large number of small, practical changes, not by a single change or advancement.

In comparison to other open LLMs, we see that DBRX-Instruct achieves better performance on composite benchmarks when compared to Mixtral (Table 5.9). Despite being a general-purpose LLM, DBRX has impressive programming skills, outperforming Grok-1 that has more than twice its size, and even specialized coding models like CodeLLaMA-70B. DBRX also performs well on reasoning and mathematics tasks.

Compared to closed models, DBRX surpasses the performance of GPT-3.5 and is competitive with Gemini-1.0 Pro. Gemini-1.0 Pro only outperforms DBRX on GSM8K, while Mixtral-Medium performs better on a select few

Model	DBRX Instruct	Mixtral Instruct	Mixtral Base	LLaMA2-70B Chat	LLaMA2-70B Base	Grok-1
Open LLM Leaderboard (Avg of next 6 rows)	74.5%	72.7%	68.4%	62.4%	67.9%	–
ARC-challenge 25-shot	68.9%	70.1%	66.4%	64.6%	67.3%	–
HellaSwag 10-shot	89.0%	87.6%	86.5%	85.9%	87.3%	–
MMLU 5-shot	73.7%	71.4%	71.9%	63.9%	69.8%	73.0%
Truthful QA 0-shot	66.9%	65.0%	46.8%	52.8%	44.9%	–
WinoGrande 5-shot	81.8%	81.1%	81.7%	80.5%	83.7%	–
GSM8k CoT 5-shot maj@1	66.9%	61.1%	57.6%	26.7%	54.1%	62.9% (8-shot)
Gauntlet v0.3 (Avg of 30+ tasks)	66.8%	60.7%	56.8%	52.8%	56.4%	–
HumanEval 0-Shot, pass@1 (Programming)	70.1%	54.8%	40.2%	32.2%	31.0%	63.2%

Table 5.9: Quality of DBRX Instruct and leading open models. Values in **magenta** are the highest scores [45].

tasks that are considered (Table 5.10). At a high level, DBRX seems to be good at programming, mathematics, general knowledge, commonsense reasoning, and information retrieval.

Model	DBRX Instruct	GPT-3.5	GPT-4	Claude 3 Haiku	Claude 3 Sonnet	Claude 3 Opus	Gemini 1.0 Pro	Gemini 1.5 Pro	Mistral Medium	Mistral Large
MT Bench (inflection corrected, n=5)	8.39 ± 0.08	–	–	8.41 ± 0.04	8.54 ± 0.09	9.03 ± 0.06	8.23 ± 0.08	–	8.05 ± 0.12	8.90 ± 0.06
MMLU 5-shot	73.7%	70.0%	86.4%	75.2%	79.0%	86.8%	71.8%	81.9%	75.3%	81.2%
HellaSwag 10-shot	89.0%	85.5%	95.3%	85.9%	89.0%	95.4%	84.7%	92.5%	88.0%	89.2%
HumanEval 0-Shot pass@1 temp=0, N=1 (programming)	70.1%	48.1%	67.0%	75.9%	73.0%	84.9%	67.7%	71.9%	38.4%	45.1%
GSM8k CoT maj@1 (5-shot)	72.8% (5-shot)	57.1% (5-shot)	92.0% (5-shot)	88.9%	92.3%	95.0%	86.5% (maj1@32)	91.7% (11-shot)	66.7% (5-shot)	81.0% (5-shot)
WinoGrande 5-shot	81.8%	81.6%	87.5%	–	–	–	–	–	88.0%	86.7%

Table 5.10: Quality of DBRX Instruct and leading closed models. Other than inflection corrected **MTBench**, which the authors measured on model endpoints, the other results were borrowed from the respective model’s publications [45].

5.12.5 OpenMoE

Despite the success of MoEs in the language modeling domain, the number of truly open-source MoEs, meaning that code, information, data, weights and more are all shared publicly, is relatively low. To solve this issue, OpenMoE [217] made a large-scale effort to train a suite of decoder-only MoE LLMs ranging from 650 million to 34 billion weights. These models adopt fine-grained experts with two granularities, 16 or 32 experts. The results from this effort are documented in [217] and all models are shared openly.

OpenMoE models follows the architectural configuration of ST-MoE [246], including the same routing mechanism and number of active experts, $k = 2$. The authors observed that using an MoE on every layer of the LLM introduces more computational overhead during routing and induces a worse cost-effectiveness trade-off than interleaved MoE usage. Therefore, authors choose to covert only one in every four or six transformer layers to an MoE layer, finding that larger strides yield a better trade-off in terms of cost and efficiency.

The pretraining dataset used for OpenMoE contains a large percentage of code. In fact, code comprised over 50% of the dataset during the early phases of pretraining, but this ratio was adjusted later in training due to being sub-optimal (Table 5.11). For alignment, OpenMoE undergoes supervised fine-tuning (SFT) after pretraining, using the data from WildChat, to induce better instruction-following capabilities.

One of the key contributions of OpenMoE is a detailed analysis of the routing decisions made within the models. First, we see that, similarly to results shown in prior work [99], experts do not tend to specialize in any particular domain (Figure 5.56). However, we do see some level of expert specialization across natural languages and specific tasks, as shown in Figure 5.57.

Model Period	Version I OpenMoE-Base before 780B tokens	Version II OpenMoE-8B/32E after 780B tokens	Version III OpenMoE-34B/32E from start to end
Dataset	Sampling Ratio		
RedPajama	50.0%	83.5%	67.5%
C4	7.50%	15.0%	15.0%
Wikipedia	2.25%	6.50%	4.50%
Stackexchange	1.00%	2.50%	1.00%
ArXiv	1.25%	4.50%	4.50%
Books	2.25%	6.50%	4.50%
GitHub	2.25%	5.00%	5.00%
Commoncrawl	33.5%	43.5%	33.0%
Wikipedia-en	0.00%	6.50%	2.50%
The Stack Dedup	50.0%	10.0%	30.0%

Table 5.11: Data mixture used during pre-training of three versions of OpenMoE.

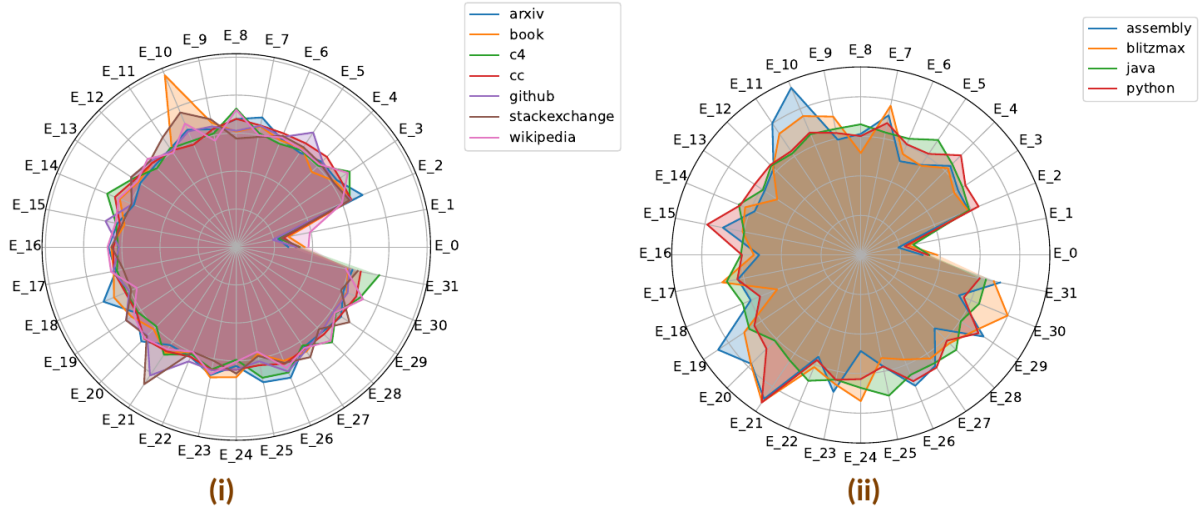


Figure 5.56: Visualization of the routing decision on the (i) RedPajama dataset and (ii) The Stack dataset. E_i denotes the ratio of tokens routed to the i -th expert [217].

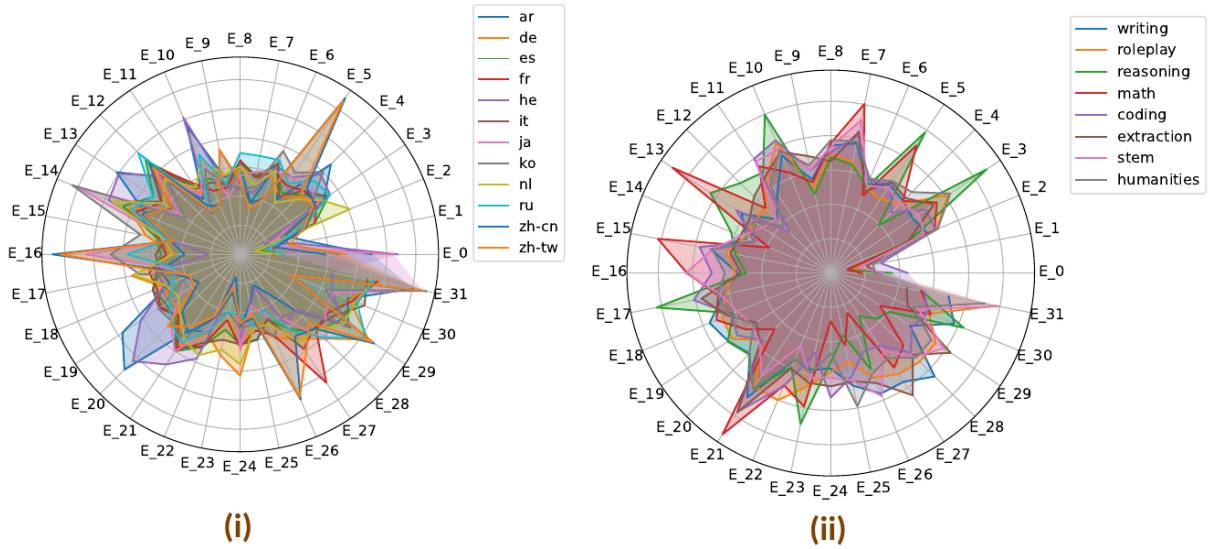


Figure 5.57: (i) Visualization of the routing decision on TED-Parallel-Corpus dataset including 12 languages: *ar* (Arabic), *de* (German), *es* (Spanish), *fr* (French), *he* (Hebrew), *it* (Italian), *ja* (Japanese), *ko* (Korean), *nl* (Dutch), *ru* (Russian), *zh-cn* (Chinese Simplified), *zh-tw* (Chinese, Traditional). (ii) Visualization of the routing decision on MT-Bench. The conversation history was adopted when evaluating OpenMoE MT-Bench as the visualization data source. E_i denotes the ratio of tokens routed to the i -th expert [217].

When we dig into this trend, however, we see that the dynamics of token routing are primarily dictated by the token ID. In other words, the same token will almost always be routed to the same expert, no matter the context in which that token exists. This pattern is referred to as "context-independent specialization" in [217].

This is a very interesting finding because the tokens with the same Token ID have very diverse contexts in different sentences. For instance, the token "an" can also be part of "an apple" or "another". However, all these tokens have very strong specialization on only a few fixed experts. Interestingly, experts have observable patterns in the tokens that they prefer (Table 5.12). For example, "have", "has" and "had" are all routed to the same expert, while one expert receives the "=", "and" and "\n" tokens, very common tokens within programming languages. We see in [217] that such routing patterns are solidified during the early stages of pretraining and rarely change later in training.

Expert ID	Top tokens
0	\n , ' , ' , s , - , \$, y , - , , 2
1	\n , 1 , , 2 , \ , S , . , - , C , {
21	, and , , . , \n , = , \t , the , , n
30	} , ed , d , have , ing , , , has , s , " , had
31	to , can , s , of , ing , will , not , e , ed , would

Table 5.12: Top tokens selected by each expert [217].

Beyond the routing patterns observed in [217], we also see that OpenMoE models exhibit some routing behaviors that could reduce their performance. For example, the models tend to drop tokens later in the sequence, which can damage performance on long sequence tasks, such as multi-turn chat. Figure 5.58 shows that OpenMoE models perform worse on multi-turn chat problems.

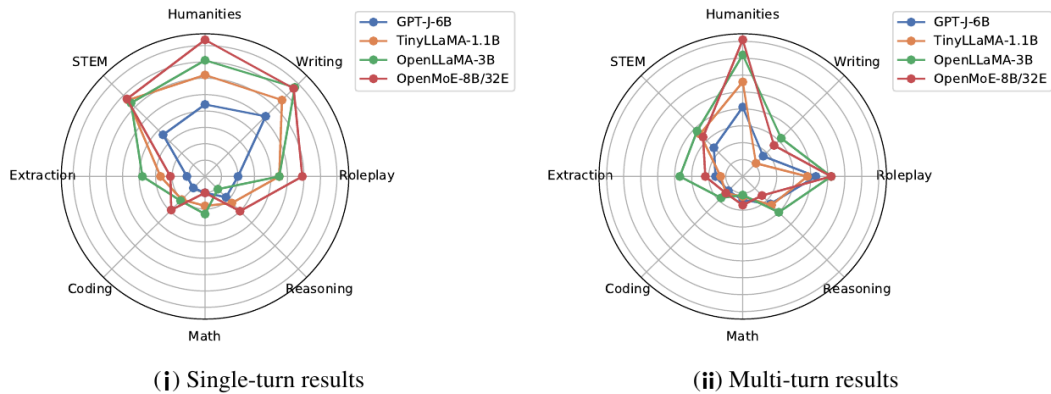


Figure 5.58: Evaluate OpenMoE on MTBench [217].

Because routing dynamics are fixed during the early phases of the pretraining process, these behaviors are hard to fix during post training. In fact, OpenMoE models are considered to struggle with the domain gap between data used during pretraining and SFT, where the token routing dynamics become sporadic due to the difference in data composition. To solve these issues, authors of [217] recommend mixing instruction-following data into the pretraining dataset.

Overall, OpenMoE models do not set new state-of-the-art performance among MoE LLMs. The larger contribution of OpenMoE models is their transparency. The details and artifacts publicly shared in [217] can accelerate open research efforts on MoEs by providing necessary resources to conduct further research on this topic.

5.12.6 DeepSeek-v2 and DeepSeek-v3

DeepSeek MoE models, including DeepSeek-v2 [49] and DeepSeek-v3 [51], have made waves within the LLM research community for a variety of reasons: their weights are shared publicly, they come with technical

reports that share many details, their performance is on par with closed models, and their training costs are pretty reasonable.

DeepSeek-v2 [49], a 236 billion weight MoE with 21 billion active weights, proposed the MoE architecture used by the later DeepSeek-V3 model. The DeepSeek MoE models are a bit different than prior models, as they slightly modify the underlying transformer block to boost performance and efficiency. As shown in Figure 5.59, the DeepSeek-v2 model, in addition to performing well, is quite impressive from a training and inference efficiency perspective, making it a strong starting point for the much larger DeepSeek-v3 model.

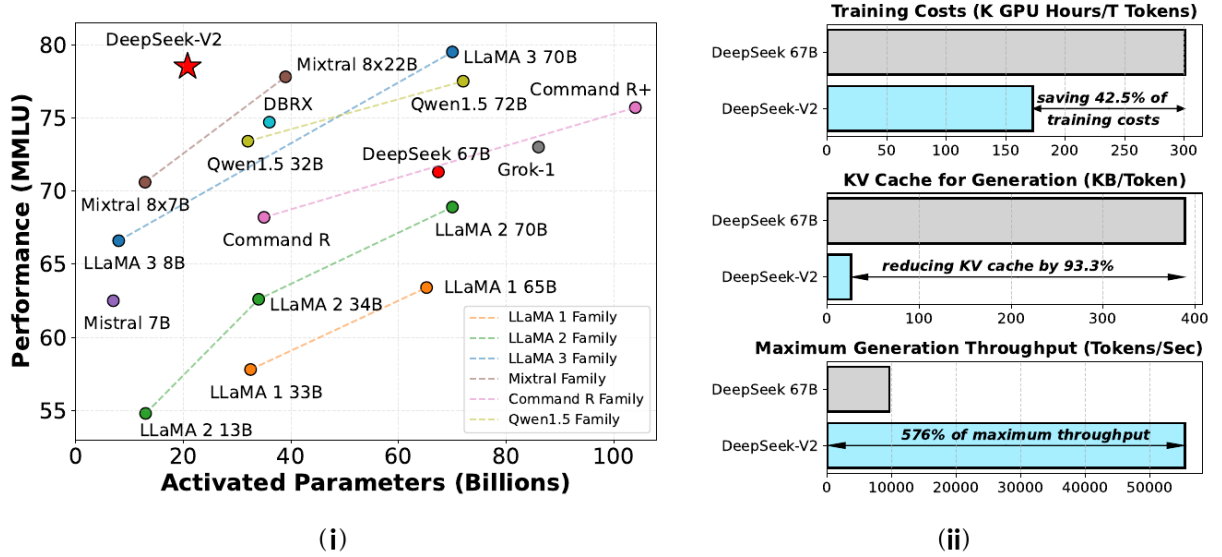


Figure 5.59: Comparison of DeepSeek-v2 with related models. (i) MMLU accuracy vs. activated weights, among different open-source models. (ii) Training costs and inference efficiency of DeepSeek 67B (dense) and DeepSeek-v2 [49].

Instead of standard, multi-headed attention, DeepSeek-v2 adopts multi-head latent attention (MLA), which is an efficient attention variant. Similarly to multi-query attention or grouped-query attention, MLA aims to minimize memory consumed by the model’s KV cache. Unlike other efficient attention variants, however, MLA does not have a significant performance trade-off (Figure 5.60). In particular, the gain in memory efficiency is achieved via a low-rank, joint K-V projection that allows us to represent all key and value vectors with a much smaller latent vector (Figure 5.60). We can upsample this vector, just linearly project it to form several, larger vectors, to restore the full key and value vectors, but we only have to store the latent vector in our KV cache, thus drastically reducing memory consumption. Adopting MLA decreases the size of DeepSeek-v2’s KV cache by over 93% compared to a 67 billion weight dense model.

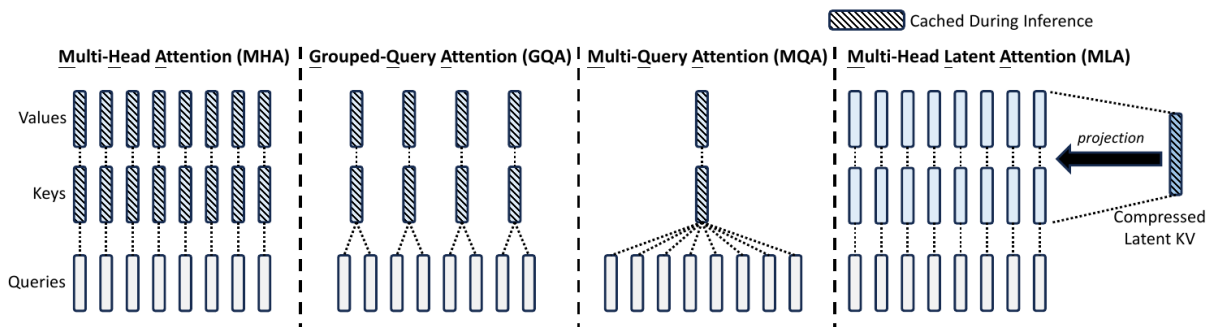


Figure 5.60: Simplified illustration of multi-head attention (MHA), grouped-query attention (GQA), multi-query attention (MQA), and multi-head latent attention (MLA). Through jointly compressing the keys and values into a latent vector, MLA significantly reduces the KV cache during inference [49].

Beyond using MLA, DeepSeek models adopt a unique MoE layer structure. Similar to DBRX, these models use fine-grained experts. However, a subset of these experts are shared. The motivation for adopting such a structure is to encourage specialization among a larger number of experts while minimizing redundant

information between experts. A schematic of the block structure used by DeepSeek models is provided in Figure 5.61.

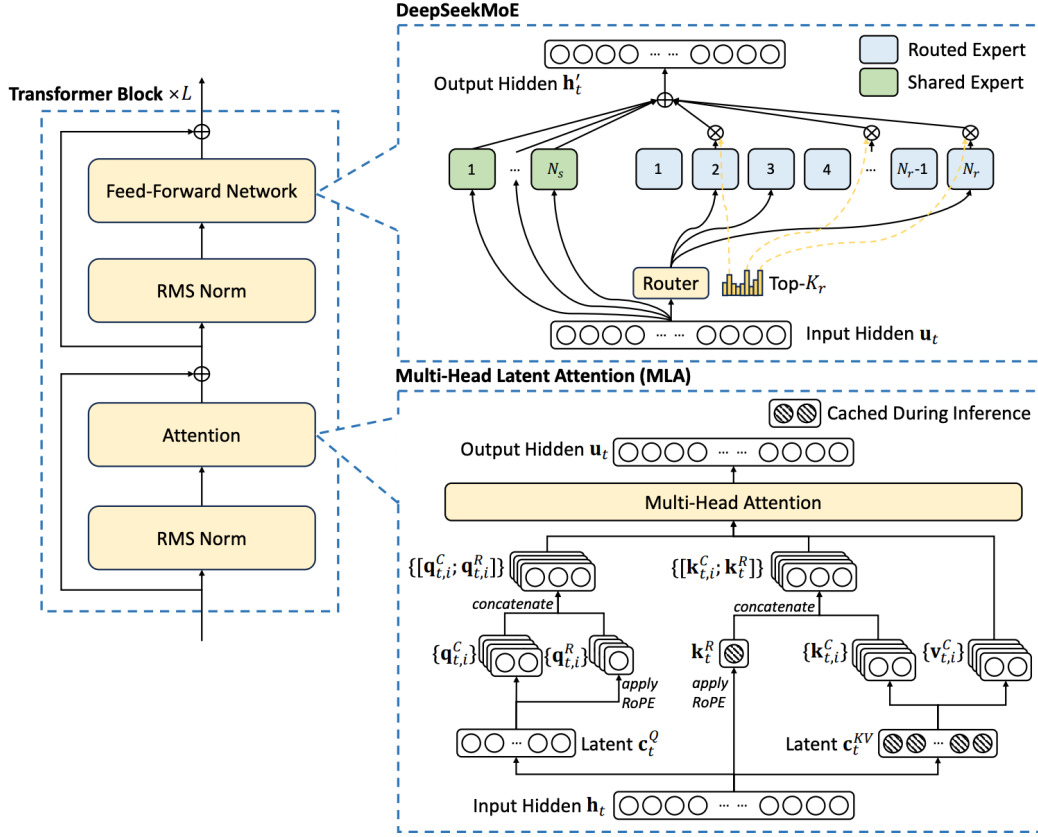


Figure 5.61: Illustration of the architecture of DeepSeek-v2. MLA ensures efficient inference by significantly reducing the KV cache for generation, and DeepSeekMoE enables training strong models at an economical cost through the sparse architecture [49].

DeepSeek-V2 is still a transformer architecture [199], where each transformer block consists of an attention module and a feed-forward network (FFN). However, for both the attention module and the FFN, it was designed and employed innovative architectures. For attention, the authors designed MLA, which utilizes low-rank key-value joint compression to eliminate the bottleneck of inference-time key-value cache, thus supporting efficient inference. In the feed-forward networks, DeepSeek-v2 follows the DeepSeekMoE architecture [44], which adopts fine-grained expert segmentation and shared expert isolation for higher potential in expert specialization. Since it was applied expert parallelism during training, the authors implemented supplementary mechanisms to control communication overheads and ensure load balance.

Low-Rank Key-Value Joint Compression

The core of MLA is the low-rank joint compression, for keys and values to reduce the size of the KV cache, defined in Equations 5.38 to 5.40.

$$\mathbf{c}_t^{KV} = W^{DKV} \cdot \mathbf{h}_t \quad (5.38)$$

$$\mathbf{k}_t^C = W^{UK} \cdot \mathbf{c}_t^{KV} \quad (5.39)$$

$$\mathbf{v}_t^C = W^{UV} \cdot \mathbf{c}_t^{KV} \quad (5.40)$$

where $\mathbf{c}_t^{KV} \in \mathbb{R}^{d_c}$ is the compressed latent vector for keys and values, d_c is $\ll d_h n_k$ and denotes the KV compression dimension, $W^{DKV} \in \mathbb{R}^{d_c \times d}$ is the down-projection matrix, and $W^{UK}, W^{UV} \in \mathbb{R}^{d_h n_h \times d_c}$ are the up-projection matrices for keys and values, respectively. During inference, MLA only needs to cache $\mathbf{c}_t^{KV}$, so its KV cache has only $d_c l$ elements, where l denotes the number of layers. In addition, during inference, since W^{UK} can be absorbed into W^Q , and W^{UV} can be absorbed into W^O , we even do not need to compute keys and values out for attention. Figure 5.60 illustrates how the KV joint compression in MLA reduces the KV cache.

In order to reduce the activation memory during training, these authors also performed low-rank compression for the queries, even if it cannot reduce the KV cache:

$$\mathbf{c}_t^Q = W^{DK} \cdot \mathbf{h}_t \quad (5.41)$$

$$\mathbf{q}_t^C = W^{UQ} \cdot \mathbf{c}_t^Q \quad (5.42)$$

where $\mathbf{c}_t^Q \in \mathbb{R}^{d'_c}$ is the compressed latent vector for queries, d'_c is $\ll d_h n_h$ and denotes the query compression dimension, and $W^{DK} \in \mathbb{R}^{d'_c \times d}$, $W^{UQ} \in \mathbb{R}^{d_h n_h \times d'_c}$ are the down-projection and up-projection matrices for queries, respectively.

Decoupled Rotary Position Embedding

Following DeepSeek 67B dense model, rotary position embedding (RoPE) [187] was also incorporated in DeepSeek-V2. However, RoPE is incompatible with low-rank KV compression. To be specific, RoPE is position-sensitive for both keys and queries. If we apply RoPE for the keys $\mathbf{k}_t^C$, W^{UK} used in Equation 5.39 will be coupled with a position-sensitive RoPE matrix. In this way, W^{UK} cannot be absorbed into W^Q any more during inference, since a RoPE matrix related to the currently generating token will lie between W^Q and W^{UK} , and matrix multiplication does not obey a commutative law. As a result, we must recompute the keys for all the prefix tokens during inference, which will significantly hinder the inference efficiency.

As a solution, they proposed the decoupled RoPE strategy that uses additional multi-head queries $\mathbf{q}_{t,i}^R \in \mathbb{R}^{d_h^R}$ and a shared key $\mathbf{k}_t^R \in \mathbb{R}^{d_h^R}$ to carry RoPE, where d_h^R denotes the per-head dimension of the decoupled queries and key. Equipped with the decoupled RoPE strategy, MLA performs the following computation:

$$[\mathbf{q}_{t,1}^R; \mathbf{q}_{t,2}^R; \dots; \mathbf{q}_{t,n_h}^R] = \mathbf{q}_t^R = \text{RoPE}(W^{QR} \mathbf{c}_t^Q) \quad (5.43)$$

$$\mathbf{k}_t^R = \text{RoPE}(W^{KR} \mathbf{h}_t) \quad (5.44)$$

$$\mathbf{q}_{t,i} = [\mathbf{q}_{t,i}^C; \mathbf{q}_{t,i}^R] \quad (5.45)$$

$$\mathbf{k}_{t,i} = [\mathbf{k}_{t,i}^C; \mathbf{k}_t^R] \quad (5.46)$$

$$\mathbf{o}_{t,i} = \sum_{j=1}^t \text{softmax} \left(\frac{\mathbf{q}_{t,i}^T \mathbf{k}_{j,i}}{\sqrt{d_h + d_h^R}} \right) \mathbf{v}_{j,i}^C \quad (5.47)$$

$$\mathbf{u}_t = W^O [\mathbf{o}_{t,1}; \mathbf{o}_{t,2}; \dots; \mathbf{o}_{t,n_h}] \quad (5.48)$$

where $W^{QR} \in \mathbb{R}^{d_h^R \times d'_c}$ and $W^{KR} \in \mathbb{R}^{d_h^R \times d}$ are matrices to produce the decoupled queries and key, respectively; $\text{RoPE}(\cdot)$ denotes the operation that applies RoPE matrices; and $[\cdot; \cdot]$ denotes the concatenation operation. During inference, the decoupled key should also be cached. Therefore, DeepSeek-V2 requires a total KV cache containing $(d_c + d_h^R) \cdot l$ elements.

Authors of [49] also adopt an interesting load balancing strategy for handling the fine-grained experts used by DeepSeek-v2. In addition to using the auxiliary load balancing loss proposed in [61], DeepSeek-v2 has

two auxiliary loss terms that aim to balance communication between devices during distributed training. First, the loss defined in Equation 5.49 enforces load balancing over device-level groups of experts, where (a) f'_i is defined in Equation 5.50 and corresponds to the average fraction of tokens sent to experts on device i , and (b) P'_i is defined in Equation 5.51 and corresponds to the average fraction of probability allocated to experts residing on device i .

$$\mathcal{L}_{device} = w_{device} \cdot \sum_{i=1}^D f'_i \cdot P'_i \quad (5.49)$$

$$f'_i = \frac{1}{|\mathcal{E}_i|} \sum_{j \in \mathcal{E}_i} f_j \quad (5.50)$$

$$P'_i = \sum_{j \in \mathcal{E}_i} P_j \quad (5.51)$$

Second, the terms f_j and P_j constitute the standard load balancing components of the loss, where (a) f_j is defined in Equation 5.52, corresponds to the fraction of tokens sent to expert j , and it is not differentiable, and (b) P_j is defined in Equation 5.53, corresponds to the fraction of probability allocated to expert j , and it is differentiable.

$$f_j = \frac{1}{T} \sum_{x \in \mathcal{B}} \mathbb{1}\{\arg \max p(x) = j\} \quad (5.52)$$

$$P_j = \frac{1}{T} \sum_{x \in \mathcal{B}} p_j(x) \quad (5.53)$$

Using fine-grained experts means that we must dispatch each token to a larger number of experts. In a distributed training setting, experts may be on different devices and multiple experts reside on each device. To ensure the communication and computation are balanced between devices, we need additional auxiliary losses that (i) group experts by the device on which they reside and (ii) encourage the MoE to perform balanced routing on a per-device basis. For example, the auxiliary loss shown above encourages balanced computation among devices. There is an extra loss proposed in [49] to encourage balanced communication among devices as well.

DeepSeek-v3 [51] is a much larger version of DeepSeek-v2, having 671 billion weights and 37B active weights. This larger model is pretrained on a massive corpus comprised of 14.8 trillion tokens. After pretraining, a multi-phase post training pipeline is applied: (i) the model first undergoes a two-stage context extension procedure in which it is fine-tuned, via supervised fine-tuning (SFT), to have a maximum context length of 32k, then further fine-tuned to have a context length of 128K, (ii) after context extension, the model undergoes further SFT and Reinforcement Learning from Human Feedback (RLHF) to align it to human preferences, (iii) capabilities from the recently-proposed R1 reasoning model are also distilled into DeepSeek-v3 during post training.

The final DeepSeek-v3 model outperforms closed-source models and achieves similar performance to even the best closed LLMs (Figure 5.62). DeepSeek-v3 also made several modifications to the training and load balancing strategies of the MoE, leading the model’s training process to be both efficient and stable.

The architecture of DeepSeek-v3 is inspired by its predecessor, for example, MLA, fine-grained experts, and shared experts are all used by DeepSeek-v3. However, unlike DeepSeek-v2, DeepSeek-v3 uses a multi-token

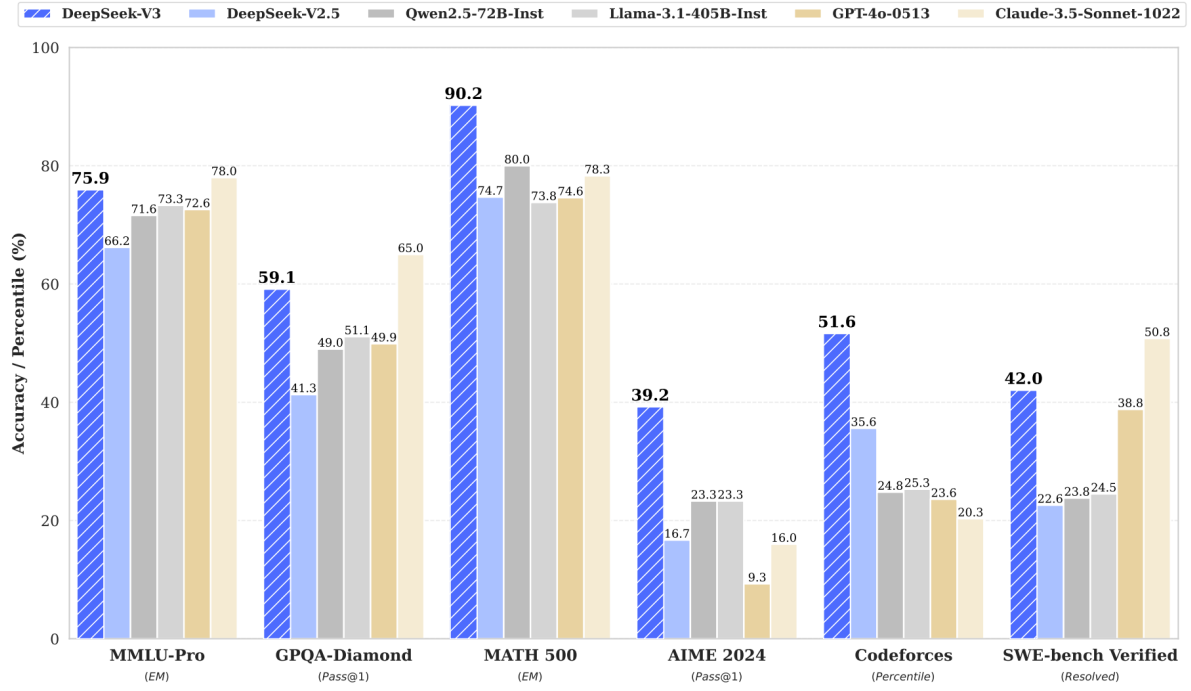


Figure 5.62: Benchmark performance of DeepSeek-V3 and its counterparts [51].

prediction (MTP) training objective. This objective is an extension of the supervised, cross entropy-based next token prediction objective that is used almost universally for training LLMs. Instead of predicting the next token for each token within a sequence, MTP predicts D future tokens. These predictions are made sequentially by a set of additional modules that are added to the model’s architecture (Figure 5.62).

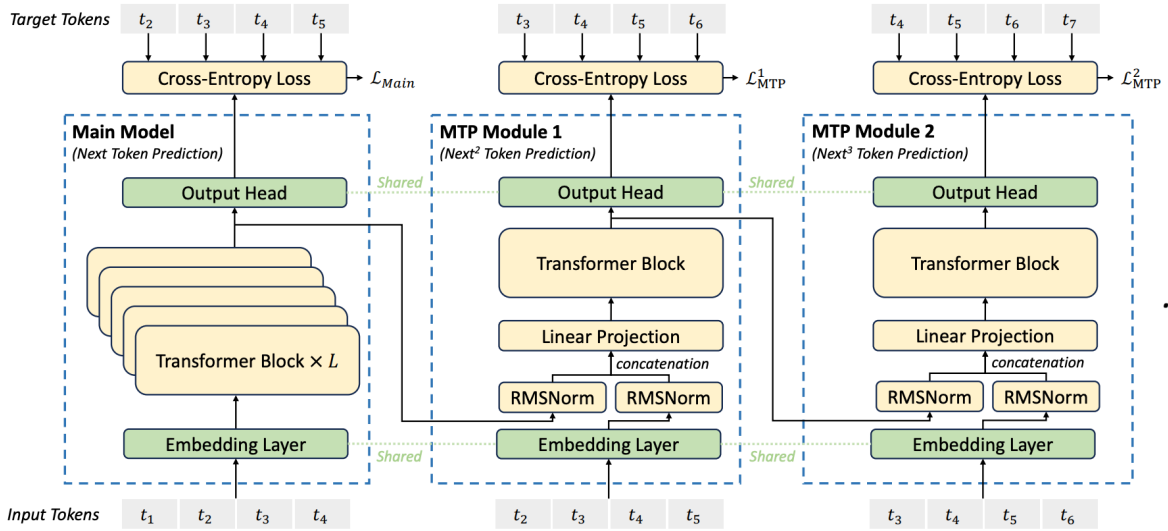


Figure 5.63: Illustration of DeepSeek-v3 multi-token prediction (MTP) implementation. The complete causal chain is kept for the prediction of each token at each depth [51].

Once several future tokens have been predicted, we can apply the cross-entropy loss normally. Applying this loss over several future tokens predicted via MTP provides a richer training signal to the model, which improves training efficiency and overall performance. Going further, these additional modules used for MTP can also be used to improve inference efficiency via speculative decoding. However, authors of [51] state that the MTP strategy is used purely to benefit model performance, the additional modules are discarded after training.

An auxiliary-loss-free load balancing strategy is also used by DeepSeek-v3 that simply adds a per-expert bias term to the selection of Top-K experts (Equation 5.54). At each iteration, the bias term b_i for each expert is either increased or decreased by a fixed factor γ based upon whether that expert was underloaded or overloaded,

respectively. Importantly, these biases are only used when selecting the top-K experts, they do not impact the computation of expert probability within the router. This approach is found to effectively balance expert utilization within the MoE and eliminate performance deterioration due to the use of load balancing losses. However, authors in [51] note that they still use an auxiliary load balancing loss, with a very low scaling factor, when training DeepSeek-v3.

$$g'_{i,t}(x) = \begin{cases} s_{i,t}(x) & , \text{ if } s_{i,t}(x) + b_i \in \text{top-}k(\{s_{j,t}(x) + b_j \mid 1 \leq j \leq N_r\}, k) \\ 0 & , \text{ otherwise} \end{cases} \quad (5.54)$$

Where $g_{i,t}(x)$ is the gating value for the i -th expert, b_i is a bias term for the i -th expert, and $s_{i,t}(x)$ is the affinity between token t and expert i .

Due to the efficiency and performance benefits of the strategies outlined above, DeepSeek-v3 is incredibly economical. Plus, the model is trained using a novel FP8 mixed precision training framework, marking the first validation of 8-bit training for large-scale LLMs. In total, training the final model was estimated to cost 5.6M USD (Table 5.13). In conclusion, the DeepSeek-v3 is (i) trained in a very economical fashion and with several novel advancements like FP8 training and MTP, (ii) incredibly impressive for an open model and highly competitive with even the best closed LLMs, (iii) based on an interesting MoE architecture with several novel modifications.

Training Costs	Pre-Training	Context Extension	Post-Training	Total
in H800 GPU hours	2664k	119k	5k	2788k
in USD	5.328M	0.238M	0.01M	5.576M

Table 5.13: Training costs of DeepSeek-v3, assuming the rental price of an H800 GPU is 2 USD per GPU hour [51].

DeepSeek-v3 also serves as a base model for DeepSeek-R1 [73], a recently-released open reasoning model. R1 is an open replication of the o1-style of models that have been recently explored by OpenAI. As explained in its detailed technical report, this model uses pure reinforcement learning to learn how to solve complex verifiable reasoning tasks by crafting extremely long chains of thought. As shown in Figure 5.64, the performance of R1 is superior to OpenAI-o1 and DeepSeek-v3.

5.13 Other Relevant MoEs

In addition to the original sparsely-gated MoE citeshazeer2017moe, Switch Transformer [61], and ST-MoE [246], that we have been discussing along this chapter, there are other relevant MoE proposals.

GShard [114] proposes a framework and a set of tools for easily implementing parallel computation patterns in sparsely-gated MoE layers. The framework is used to scale a transformer-based neural machine translation model to a size of over 600B weights. Using GShard, this model was trained on over 2048 TPUs for 4 days and surpasses the quality of prior models on translating over 100 different languages to English. Some useful takeaways for effectively training MoE models in GShard include: (i) every feed-forward module in the encoder-decoder transformer was replaced with an MoE layer, (ii) top-2 gating was applied in both encoder and decoder, (iii) the gating mechanism always selects the top expert, but a second expert is randomly chosen with a probability proportional to its weight.

A transformer encoder layer consists of two consecutive layers, namely a self-attention layer followed by a position-wise feed-forward layer. A decoder adds third cross-attention layer, which attends over encoder output. In [114], they sparsely scaled the transformer with conditional computation by replacing every feed-forward layer with a position-wise MoELayer with a variant of top-2 gating in both the encoder and the decoder (Figure 5.65). The number of transformer layers and the number of experts per MoE layer was varied in order to scale the model capacity.

Mesh-Tensorflow [176] greatly simplified the exploration of efficient, model-parallel distributed training strategies, by splitting the model across multiple devices. Previously, data parallel strategies that split the input

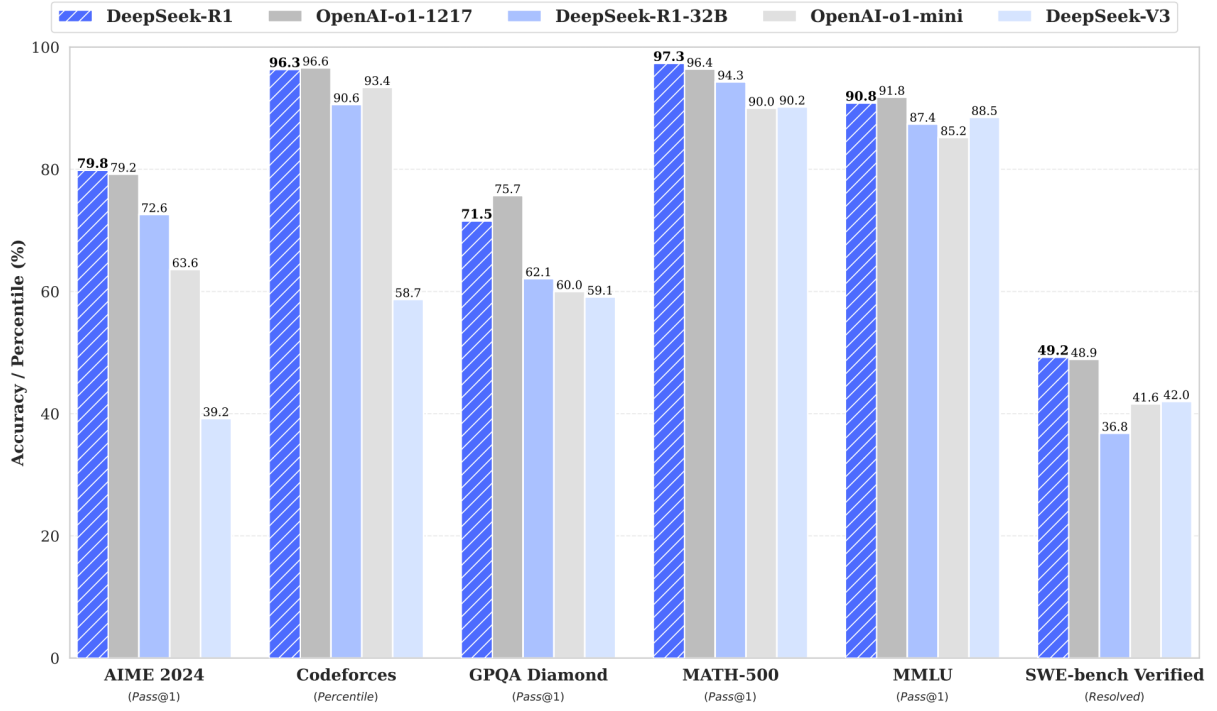


Figure 5.64: Benchmark performance of DeepSeek-R1 [73].

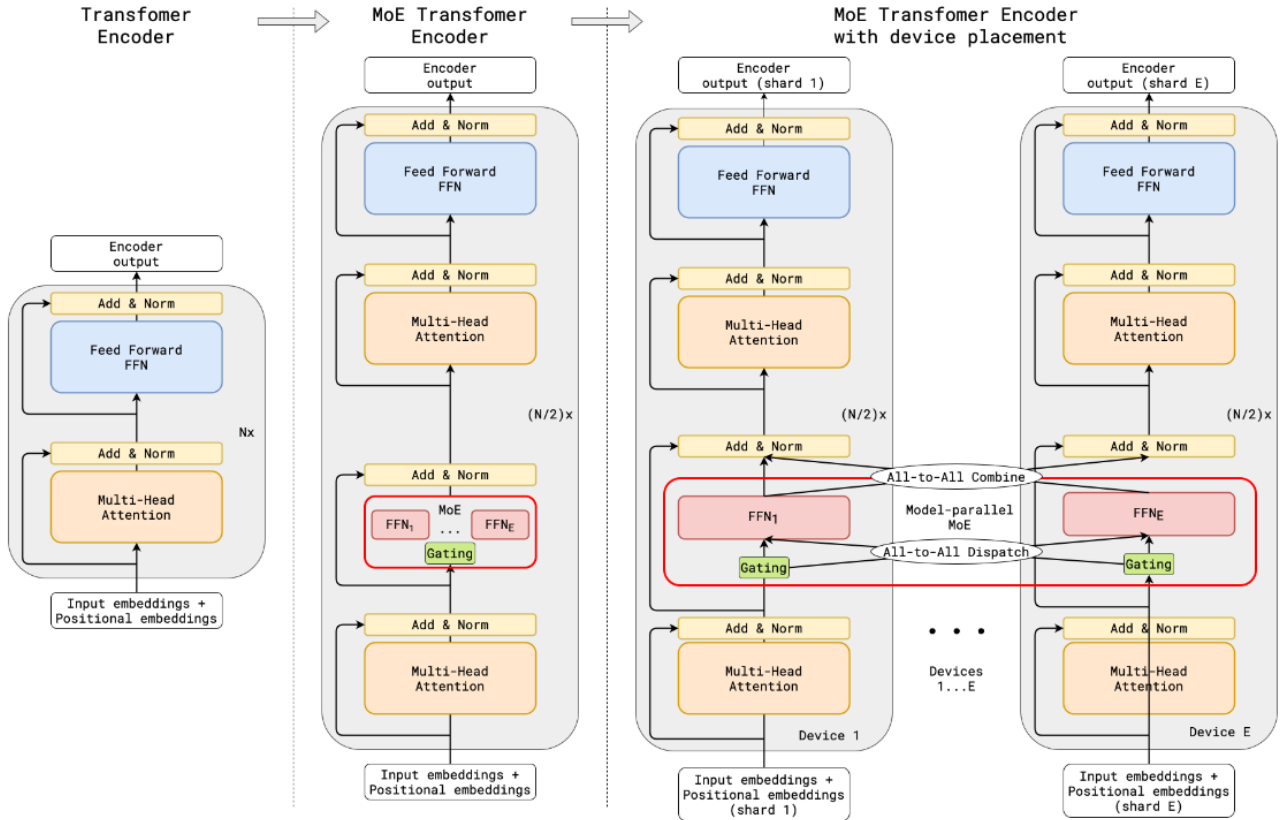


Figure 5.65: Illustration of scaling the transformer encoder with MoE layers. The MoE layer replaces the every other Transformer feed-forward layer. Decoder modification is similar. (a) The encoder of a standard transformer model is a stack of self-attention and feed forward layers interleaved with residual connections and layer normalization. (b) By replacing every other feed forward layer with a MoE layer, we get the model structure of the MoE transformer encoder. (c) When scaling to multiple devices, the MoE layer is sharded across devices, while all other layers are replicated [176].

minibatch of data across multiple devices, were the predominant approach for distributed training of neural networks but, as have seen in Chapter 3, data parallel training has several limitations. The most important is, training large models is harder because each device must keep a full copy of the model in memory and efficiency decreases when using a small minibatch size. Mesh-Tensorflow is a general language that can be used to specify distributed computations. After its release, MoE layers were re-introduced into the transformer architecture, since programming data-parallel and model-parallel training strategies became very simple. In [176], the authors used Mesh-Tensorflow to train a 5B weights transformer model that surpasses state-of-the-art results on machine translation tasks.

M6 [124] proposed a Chinese-only dataset, which contains 1.9TB of images and 292GB of text, for pretraining multi-modal language models. From this dataset, several language models were pretrained using a novel, cross-modal pretraining strategy. Figure 5.66 summarizes the pretraining tasks in M6. The masking strategies allow the learning of different tasks under the same framework. M6 is pretrained with image-based text denoising, image captioning, text denoising, and language modeling. One of the models that was trained in [124] is a 100B weights MoE model that replaces all feed-forward sub-layers in the transformer depicted above with MoE layers comprised of 1024 experts.

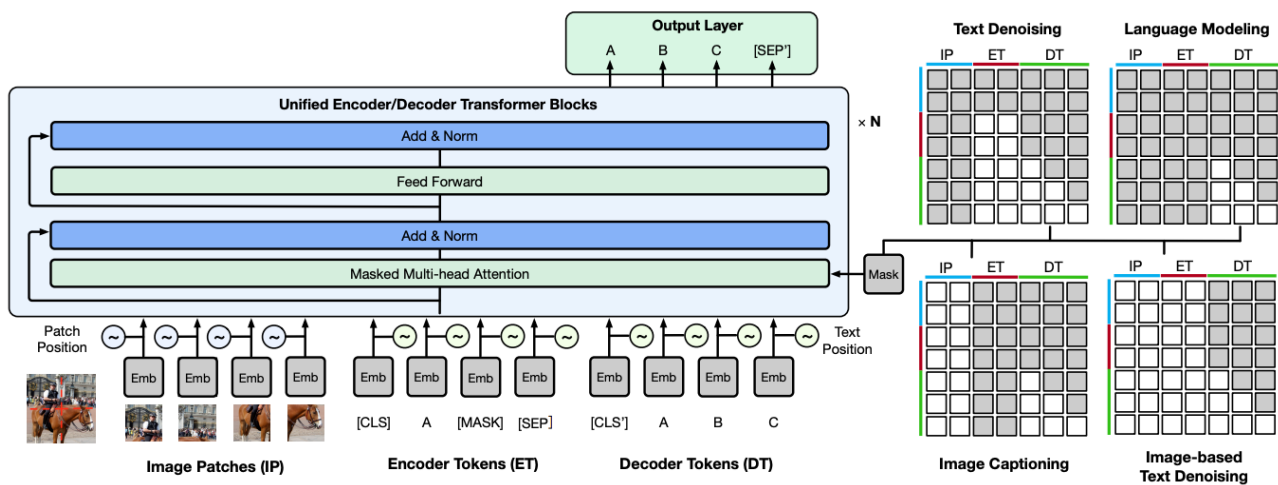


Figure 5.66: An overview of the pretraining tasks in M6. The design of masking strategies allows the learning of different tasks under the same framework. M6 is pretrained with image-based text denoising, image captioning, text denoising, and language modeling [124].

Although MoE models are clearly beneficial for language modeling, it is unclear whether similar benefits hold for computer vision. In [170], the authors explored a Vision MoE (V-MoE) model that combines the vision transformer architecture with sparse MoE layers (Figure 5.67). They replaced some of the feed-forward networks of the vision transformer with MoE layers. Such an approach yields a desirable tradeoff between performance and compute. Going further, they also proposed a new gating algorithm that can adaptively leverage more compute for certain inputs at inference time.

Scaling up dense LLMs often run into hardware limitations, but MoE models can be scaled almost arbitrarily. For this reason, MoE models are a promising architectural advancement, but their adoption is often limited by their larger size and unique architecture. Aiming to make conditional computation easier to use in practice, authors of [163] create the DeepSpeed-MoE framework for end-to-end training and inference of MoE models. This framework defines several MoE architectures, proposes model compression techniques that reduce MoE size up to 3.7 times, and implements an optimized inference engine that can improve latency by 7.3 times.

In [59], the authors created a Many-to-Many machine translation model that can translate directly between any pair of one hundred languages. While prior models are highly English-centric, these authors aimed to create a model that better reflects translation needs worldwide. First, they created an open source training dataset that covers thousands of language translation directions with supervised data, through large-scale mining. Second, they explored how to effectively increase the model capacity through a combination of dense scaling and language-specific sparse weights. Figure 5.68 illustrates their data mining strategy as well as the model architecture.

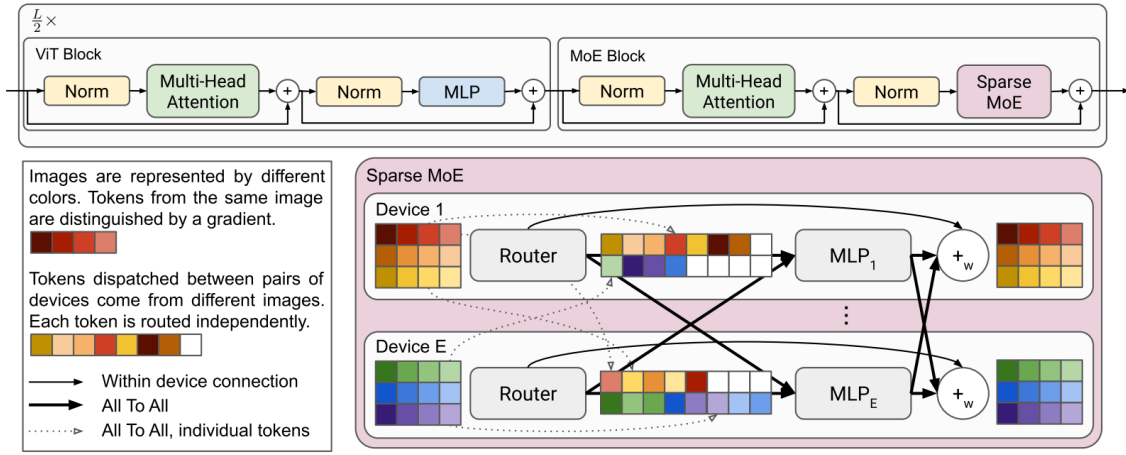


Figure 5.67: Overview of the V-MoE architecture. V-MoE is composed of L ViT blocks. In some blocks, the MLP is replaced with a sparsely activated mixture of MLPs. Each MLP expert is stored on a separate device, and processes a fixed number of tokens. The communication of these tokens between devices is shown in this example, which depicts the case when $k = 1$ expert is selected per token. Here each expert uses a capacity ratio $C = 4/3$: the sparse MoE layer receives 12 tokens per device, but each expert has capacity for 16 ($(16 \cdot 1)/12 = 4/3$). Non-expert components of V-MoE such as routers, attention layers and normal MLP blocks are replicated identically across devices [170].

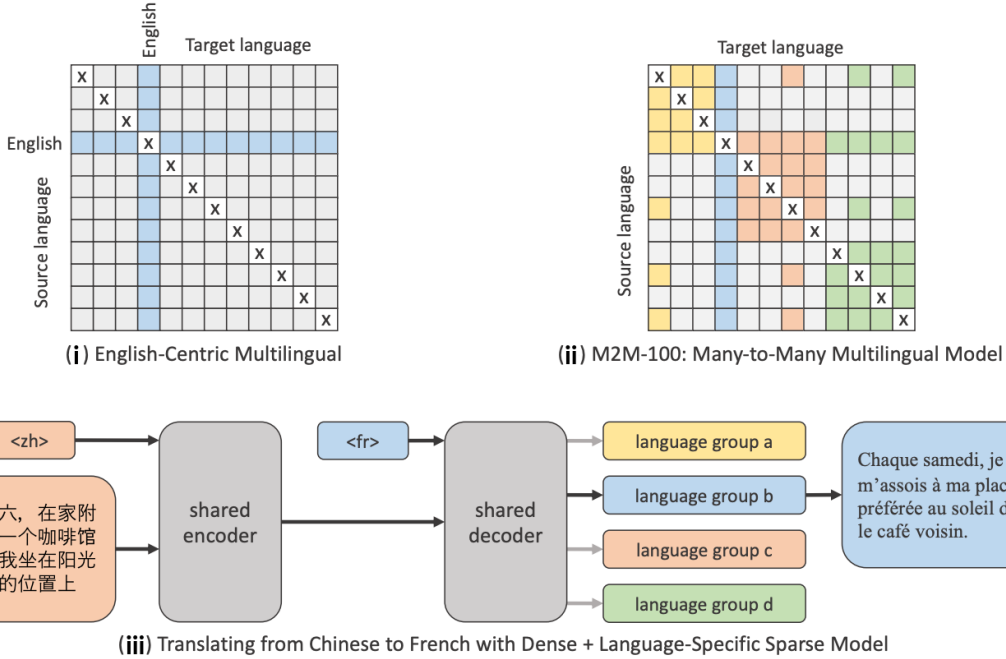


Figure 5.68: Summary of the many-to-many dataset and multilingual model. English-Centric data (i) only contains training data to and from English, where as the Many-to-Many multilingual setting (ii) contains data directly through various different directions. The proposed model, M2M-100, combines dense and sparse language-specific weights to translate directly between languages (iii) [59].

Generalist Language Model (GLaM) [56] extends the work of Switch Transformers, where encoder-decoder transformer architectures were used, to decoder-only language models. The goal of this work was to train a GPT-3 quality model but more efficiently, using an order of magnitude less compute resources. GLaM provides a suite of LLMs that use MoE layers to scale up model capacity while maintaining reasonable training costs. The largest model in this suite has 1.2 trillion weights, meaning it is seven time larger than GPT-3, performs better than GPT-3 across nearly 30 NLP tasks, is trained using only one third of the energy used for GPT-3, and requires half of the computation of GPT-3 at inference time.

GLaM leverages sparsely activated MoE models. Similar to the GShard MoE transformer, they replaced the feed-forward network of every transformer layer with an MoE layer, as shown in Figure 5.69. Each MoE layer employs a series of independent feed-forward networks as experts. A gating function then uses a softmax activation function to model a probability distribution over these experts. This distribution indicates how well

each expert is able to process the incoming input.

GLaM models use top-two gating, meaning that there are two active experts in each layer. The authors also explored larger capacity factors, finding that this metric is useful for controlling the amount of compute used during training or inference.

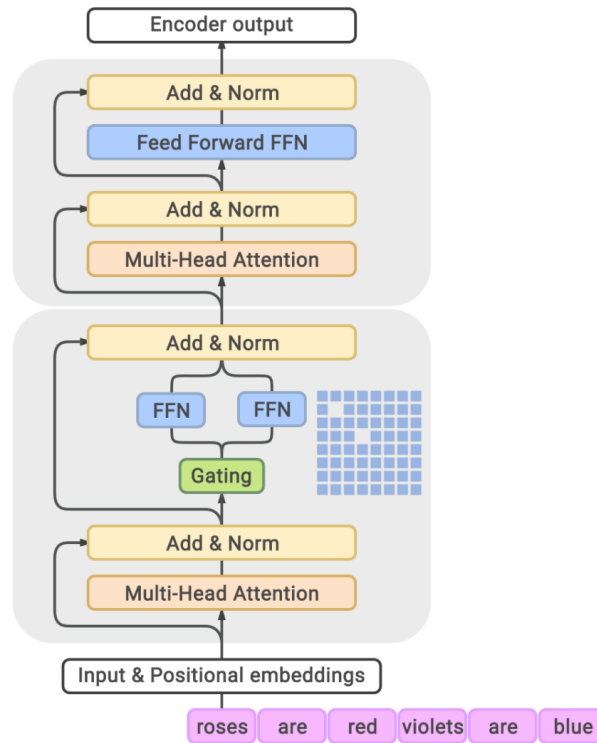


Figure 5.69: GLaM model architecture. Each MoE layer, the bottom block, is interleaved with a transformer layer, the upper block. For each input token, for example, 'roses', the gating module dynamically selects two most relevant experts out of 64, which is represented by the blue grid in the MoE layer. The weighted average of the outputs from these two experts will then be passed to the upper transformer layer. For the next token in the input sequence, two different experts will be selected [56].

5.14 Conclusions

In this chapter, we examined the Mixture of Experts paradigm as a powerful approach for scaling neural networks while maintaining computational efficiency. By leveraging conditional computation, MoE architectures activate only a subset of specialized experts for each input, enabling a substantial increase in model capacity without a proportional rise in computational cost. This key idea has positioned MoE as a central technique in the development of modern large-scale models, particularly in the context of transformer-based architectures.

We first established a structured understanding of the field through a taxonomy of MoE methods, highlighting the interplay between algorithmic design, system considerations, and application domains. From an algorithmic perspective, the integration of MoE layers into transformers revealed a rich design space involving dense and sparse formulations, gating mechanisms, and expert architectures. Sparse MoE has emerged as the dominant approach, striking a balance between efficiency and scalability, while variants such as soft routing and weight-efficient experts demonstrate the flexibility of the paradigm in addressing stability and fine-tuning challenges.

The chapter further explored training and inference strategies, revealing that MoE systems are not limited to a single static configuration. Approaches such as dense-to-sparse transitions, sparse-to-dense distillation, and expert merging frameworks enable flexible model development and deployment. These strategies emphasize that MoE models can be adapted to different stages of the lifecycle, from efficient pretraining to practical inference.

A critical component of MoE performance lies in the design of routing mechanisms, which determine how tokens are assigned to experts. Techniques such as Top-k gating, Switch Transformers, noisy routing, hash-based assignment, and Soft MoE illustrate the trade-offs between computational efficiency, training stability, and expert specialization. While sparse routing enables scalability, it introduces challenges such as load imbalance and routing instability, requiring additional mechanisms to ensure effective expert utilization.

From a systems perspective, the chapter highlighted the complexity introduced by sparse and dynamic execution patterns. Efficient large-scale deployments require careful handling of computation, communication, and memory. Techniques such as expert parallelism and hybrid parallel strategies enable scalable training, but also introduce significant challenges, particularly due to communication-intensive operations such as All-to-All exchanges.

To address these issues, we examined load balancing and resource management techniques, including auxiliary losses, expert capacity constraints, and adaptive expert placement. These mechanisms are essential for preventing expert collapse, improving hardware utilization, and maintaining stable training dynamics. However, they also reveal a fundamental tension between expert specialization and uniform resource usage, which remains a core challenge in MoE design.

Finally, the chapter discussed training dynamics and performance considerations, emphasizing that MoE models, while highly efficient at scale, introduce unique difficulties. These include numerical instabilities, sensitivity to routing decisions, and challenges in fine-tuning with limited data. Additionally, the evaluation of MoE systems requires not only traditional metrics such as throughput and accuracy but also system-level measures of hardware utilization and communication efficiency.

In summary, Mixture of Experts represents a versatile and scalable framework for deep learning, enabling unprecedented model sizes and capabilities. However, its effectiveness depends on a careful balance between algorithmic design and system-level optimization. As models continue to grow and computational resources evolve, future advances will likely focus on more robust routing mechanisms, improved load balancing strategies, and tighter integration between model architecture and hardware-aware system design.

Chapter 6

Implement Mixture-of-Experts in PyTorch

Contents

6.1	Decoder-only transformer	207
6.2	Build a mixture-of-experts model	223
6.3	Pretraining the MoE model	236
6.4	Conclusions	241

This chapter presents a practical and implementation-focused exploration of Mixture of Experts (MoE) models within the context of large language models. It begins by establishing the necessary foundations, revisiting the decoder-only transformer architecture, which serves as the backbone of most contemporary language models. Key components such as tokenization, embeddings, masked multi-head self-attention, and feed-forward networks are explained and implemented in PyTorch to build a complete baseline transformer model. Building on this foundation, the chapter then introduces the MoE extension of the transformer, highlighting how standard feed-forward networks are replaced by expert layers composed of multiple independent sub-networks. It details how these experts are implemented efficiently and interleaved within the transformer architecture, enabling increased model capacity without proportional computational cost. A central focus of the chapter is the design of routing mechanisms, which dynamically assign tokens to a subset of experts. The chapter explains top-k softmax routing, optional noisy routing for regularization, and the concept of active weights, which underpins the efficiency of sparse MoE models. Additionally, it addresses expert capacity constraints and the practical challenges of batching and dispatching tokens to experts. To ensure stable and effective training, the chapter further explores load balancing strategies, including auxiliary loss functions and the router z-loss, which mitigate issues such as routing collapse and numerical instability. It also presents alternative approaches and discusses their impact on training dynamics. The implementation is then brought together into a complete MoE-based decoder-only transformer, showing how all components, attention layers, expert layers, and routing, are integrated into a unified architecture. Finally, the chapter covers pretraining procedures, including dataset preparation, configuration choices, and best practices such as mixed precision training, routing precision considerations, and weight initialization techniques. It concludes with experimental results that illustrate the importance of stability techniques when training MoE models in practice.

Research on large language models (LLMs) has dominated the artificial intelligence domain over the last eight years. However, the architecture upon which most LLMs are based, the decoder-only transformer, has remained fixed despite the chaotic and rapid advancements in this field. More recently, we are starting to see a new architecture, called a Mixture-of-Experts (MoE), being adopted in many top language models. For example, it is thought that GPT-4 is MoE-based, as well as the recently-proposed, and very popular, DeepSeek-v3 and DeepSeek-R1 models. To push the boundaries of open-source model capabilities, DeepSeek-V3 was designed as a large Mixture-of-Experts (MoE) model with 671B weights, of which 37B are active for each token [51].

MoE-based LLMs use a modified version of the decoder-only transformer that has become popular due to their ability to make the training and usage of large models more efficient. MoE-based LLMs are very large in terms of their number of total weights. However, only a subset of these weights, selected dynamically during inference, are used when computing the model’s output. The sparsity of MoEs drastically reduces the cost of very large and powerful LLMs.

Given that many frontier LLMs are starting to use MoE-based architectures, developing an in-depth understanding of MoEs is important. In this chapter, we build and pretrain a mid-sized MoE model, called nanoMoE, from scratch in PyTorch. All of the code for nanoMoE is borrowed from the repository <https://github.com/wolfecameron/nanoMoE>, which is a fork of Andrej Karpathy’s nanoGPT library, expanded to support the MoE architecture. To understand how nanoMoE works, we will start by outlining necessary background information. Then, we will build each component of nanoMoE, that we will try to pretrain in the end of this process.

6.1 Decoder-Only Transformer

In order to understand MoE-based LLMs, we first need to understand the standard architecture upon which most LLMs are based, the decoder-only transformer architecture. This architecture is a modified version of the encoder-decoder transformer architecture proposed in [199] and was popularized by GPT (Figure 6.1). While explaining the architecture, we will rely on nanoGPT, a minimal and functional implementation of a decoder-only transformer, as a reference [209].

Original architecture

The transformer, originally proposed for solving machine translation tasks in [199], has both an encoder and a decoder module. We will not focus on the full encoder-decoder transformer here. The decoder-only transformer, which is more commonly-used in modern LLMs, simply removes the encoder from this architecture and uses only the decoder, as indicated by the name. In practice, this means that every layer of the decoder-only transformer architecture contains a masked self-attention layer and a feed-forward layer. To form the full decoder-only transformer architecture, we just stack L of these layers, which are identical in structure but have independent weights, on top of each other. A depiction of this structure is provided in Figure 6.2.

Next, we will discuss each component of the architecture to gain a better understanding. We will start with the input structure for the model, followed by the components of each layer, self-attention and feed-forward layers, and how they are combined to form the complete architecture.

6.1.1 From Text to Tokens

The input to an LLM is just a sequence of text, the prompt. However, the input that we see in Figure 6.2 is not a sequence of text. Rather, the model’s input is a list of token vectors. Thus, our first task is to understand how the input text is converted to the token vectors.

Tokenization

The first step of constructing the input for an LLM is breaking the raw textual input, a sequence of characters, into discrete tokens. This process is called tokenization and is handled by a tokenizer. There are many kinds of tokenizers, but Byte-Pair Encoding (BPE) tokenizers [173] are the most common. These tokenizers take a sequence of raw text as input and break this text into a sequence of discrete tokens as illustrated in Figure 6.3.

Packages for training and interacting with LLMs, such as HuggingFace or torchtune, provide interfaces for interacting with tokenizers. Additionally, OpenAI has released the tiktoken package for interacting with GPT tokenizers. The code snippet included in Listing 6.1 tokenizes a textual sequence according to the following steps: (i) load the LLaMa-3.1 tokenizer, (ii) define a text prompt, (iii) get tokens and token identifiers for the prompt using the loaded tokenizer, (iv) get token embeddings using a `torch.nn.Embedding` layer. The execution of this code produces the next output. We can observe the ID associated to each token and that each token embedding is a 768-dimensional vector.

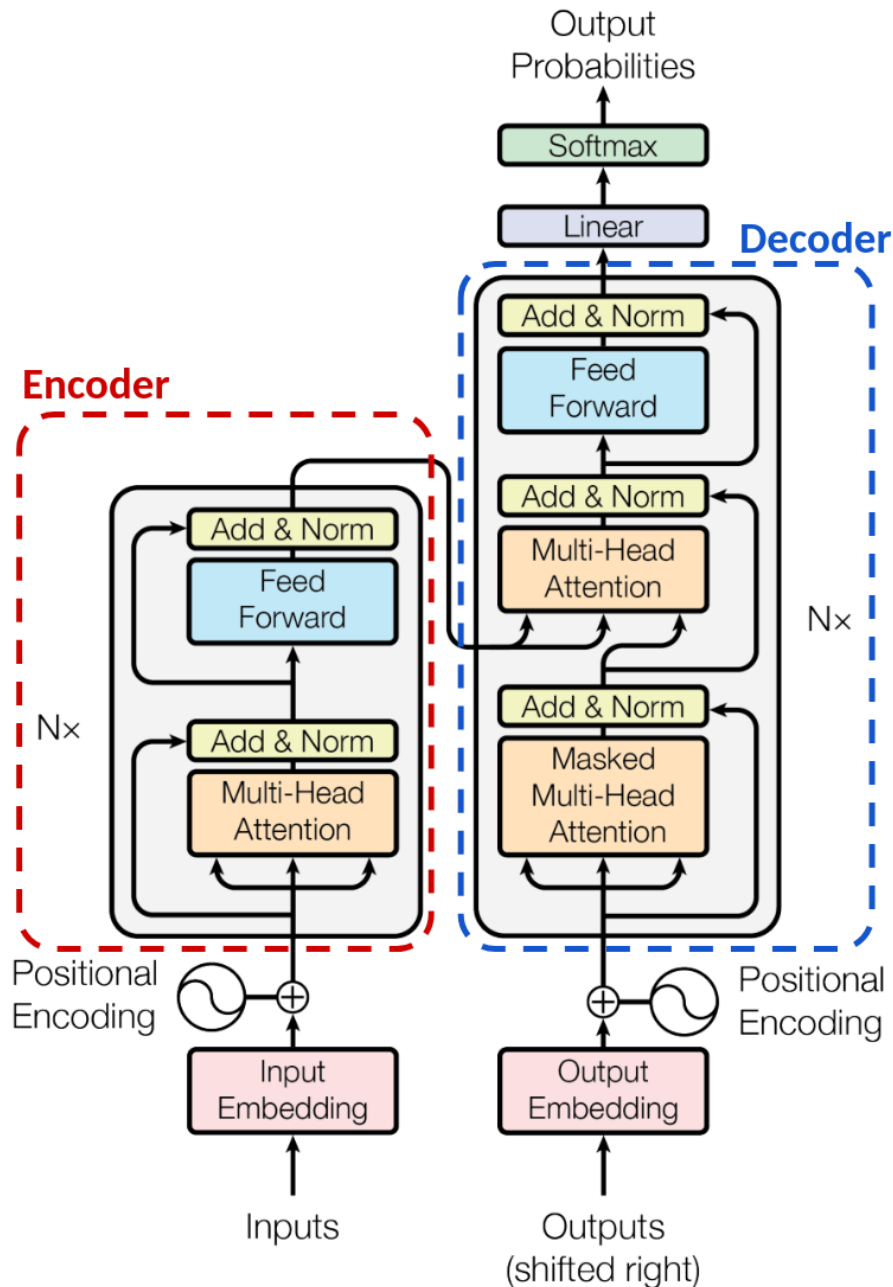


Figure 6.1: The original encoder-decoder transformer architecture [199].

Original text: This raw text will be tokenized
 Tokens: ['This', 'Ġraw', 'Ġtext', 'Ġwill', 'Ġbe', 'Ġtoken', 'ized']
 Token IDs: [2028, 7257, 1495, 690, 387, 4037, 1534]
 Token embeddings shape: torch.Size([7, 768])

Here, the $\dot{G}$ character indicates that a token immediately follows a white space. Such special characters are tokenizer-dependent. For example, many tokenizers instead use a # character to indicate the continuation of a word, which would yield ['token', '#ized'] for the final two tokens in the above sequence.

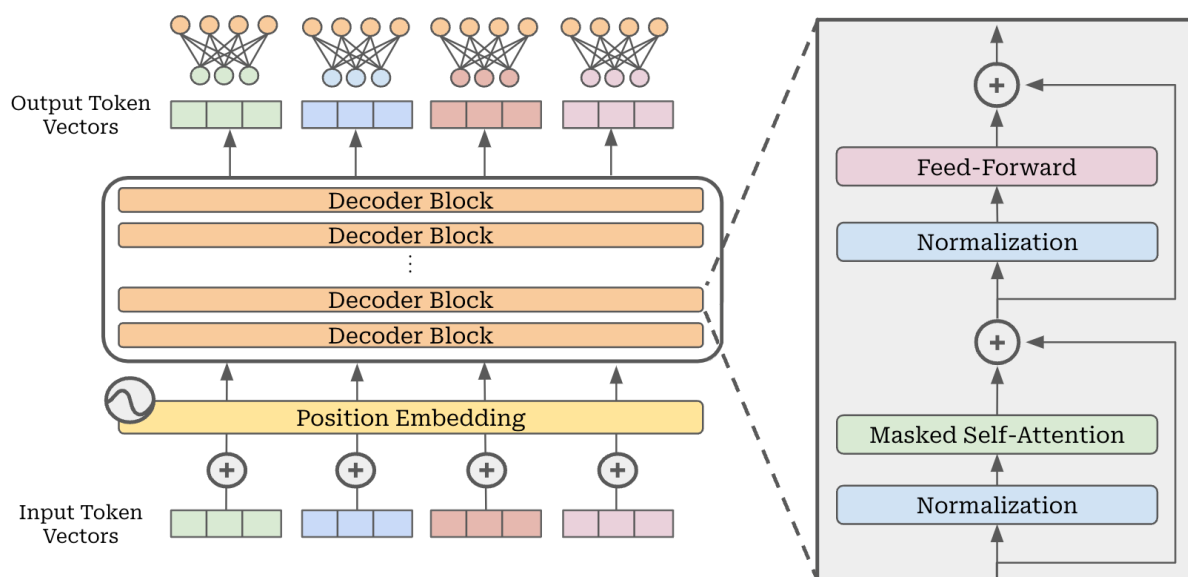


Figure 6.2: The decoder-only transformer architecture [209].

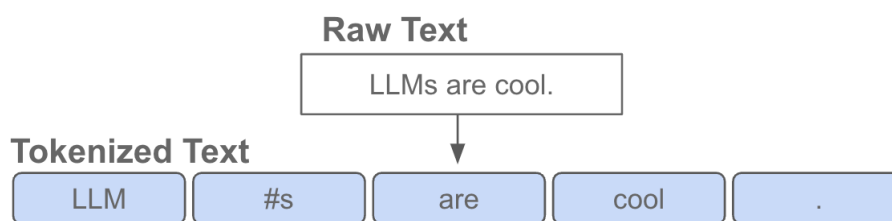


Figure 6.3: Converting raw text into a sequence of tokens [209].

```

1 import torch
2 from transformers import AutoTokenizer
3
4 # Load the LLaMa-3.1 tokenizer
5 tokenizer = AutoTokenizer.from_pretrained('meta-llama/Llama-3.1-8B')
6
7 # Raw text prompt
8 text = "This raw text will be tokenized"
9
10 # Create tokens for the prompt using a tokenizer
11 tokens = tokenizer.tokenize(text)
12 token_ids = tokenizer.convert_tokens_to_ids(tokens)
13 # token_ids = tokenizer.encode(text) # Create token IDs in one step
14
15 # View the results
16 print("Original text:", text)
17 print("Tokens:", tokens)
18 print("Token IDs:", token_ids)
19
20 # Instantiate a token embedding layer
21 VOCABULARY_SIZE: int = 128000
22 EMBEDDING_DIM: int = 768
23 token_embedding_layer = torch.nn.Embedding(
24     num_embeddings = VOCABULARY_SIZE,
25     embedding_dim = EMBEDDING_DIM,
26 )
27
28 # Get token embeddings (IDs must be passed as a tensor, not a list)
29 token_emb = token_embedding_layer(torch.tensor(token_ids))
30 print(f'Token embeddings shape: {token_emb.shape}')

```

Listing 6.1: Tokenize a textual sequence [209].

Vocabulary

Each LLM is trained with a specific tokenizer, though a single tokenizer may be used for several different LLMs. The set of tokens that can be produced by a given tokenizer is also fixed. Therefore, an LLM has a fixed set of tokens that it understands, which correspond to the tokens produced by the tokenizer, and is trained on. The fixed set of tokens is often referred to as the LLM’s vocabulary (Figure 6.4). The vocabulary size changes between models and depends on several factors, where multilingual models tend to have larger vocabularies, but vocabulary sizes from 64k to 256k tokens are common in recent LLMs.

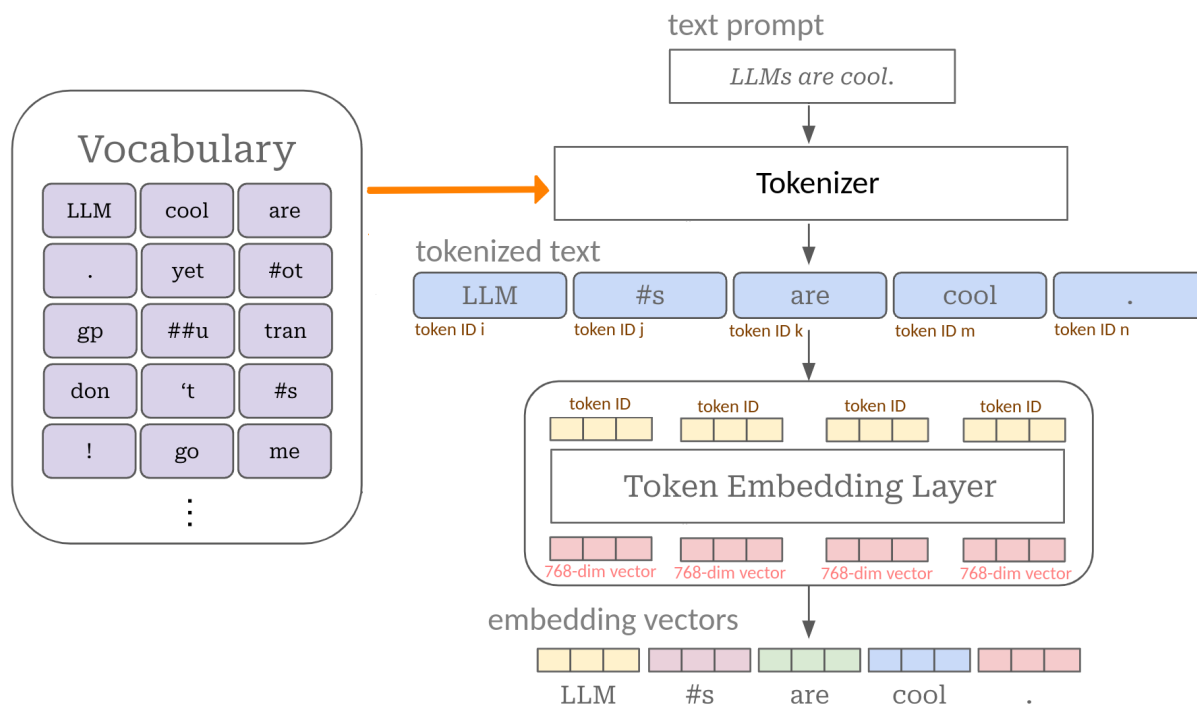


Figure 6.4: Token vocabulary for a large language model [209].

Token Identifiers and Token Embeddings

Each token in the LLM’s vocabulary is associated with a unique integer identifier. For example, the prior code yields this sequence of identifiers when tokenizing the text prompt: [2028, 7257, 1495, 690, 387, 4037, 1534]. Each of these identifiers is associated with a vector, known as a token embedding, produced by the Embedding layer. An Embedding layer is just a large learnable lookup table that maps token identifiers to token embedding vectors. To retrieve the embedding for a token, we just lookup for the corresponding entry in the embedding layer, given the token identifier (Figure 6.5).

We now have a list of token embeddings and we can stack these embeddings into a tensor to form the actual input that is ingested by the transformer (Figure 6.2). In PyTorch, the creation of this tensor is handled automatically by the tokenizer and embedding layer, as shown in Listing 6.1.

The token embedding tensor is of size $[T, d]$, where T is the number of tokens in our input and d is the dimension of token embeddings that is adopted by the LLM. We usually have a minibatch of B input sequences instead of a single input sequence, forming an input tensor of size $[B, T, d]$. The dimension d impacts the sizes of all layers or activations within the transformer, which makes d an important hyperparameter. Prior to passing this tensor to the transformer as input, we also add a positional embedding to each token in the input, which communicates the position of each token within the sequence to the transformer.

6.1.2 Masked Multi-Headed Self-Attention

Now, we can apply the token embedding tensor to the decoder-only transformer for processing. As previously outlined, the transformer contains repeated blocks with self-attention and a feed-forward network, each one followed by a normalization operation. Thus, we will analyze the self-attention first.

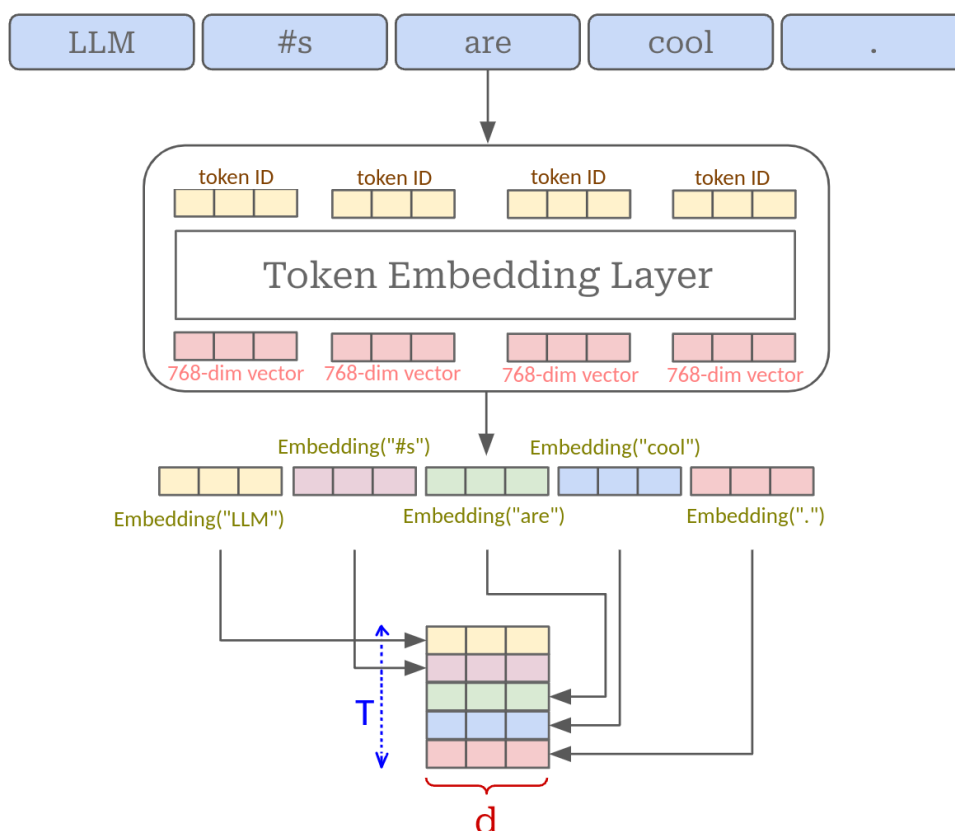


Figure 6.5: Input tensor of token embeddings [209].

Self-attention transforms the representation of each token in a sequence based upon its relationship to other tokens in the sequence (Figure 6.6). Intuitively, self-attention bases the representation of each token on the other tokens in the sequence, including itself, that are most relevant to that token. In other words, we learn which tokens to pay attention to when trying to understand the meaning of a token in our sequence. For example, we see in Figure 6.7 that the representation for the word "making" is heavily influenced by the words "more" and "difficult", which help to convey the overall meaning of the sentence.

An attention function maps a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key [199].

Scaled dot product attention

Given our input token tensor of size $[T, d]$ where, for simplicity, we assume that we are processing a single input sequence instead of a minibatch, we begin by projecting the input using three separate linear projections, forming three separate sets of transformed token tensors. These projections are referred to as the key, query and value projections (Figure 6.8).

The intuition behind the name of each projection is as follows: (i) a query is what you use to search for information and it represents the current token for which we want to find other relevant tokens in the sequence; (ii) the key represents each other token in the sequence and acts as an index to match the query with other relevant tokens in the sequence; (iii) the value is the actual information that is retrieved once a query matches a key. The value is used to compute each token's output in self-attention.

Computing the attention scores

After projecting the input, we compute an attention score $a[i, j]$ for each pair of tokens $[i, j]$ in our input sequence. Intuitively, this attention score, which lies in the $[0, 1]$ range, captures how much a given token should pay attention to another token in the sequence, higher attention scores indicate that a pair of tokens are very relevant to each other. As mentioned above, attention scores are generated using the key and query vectors.

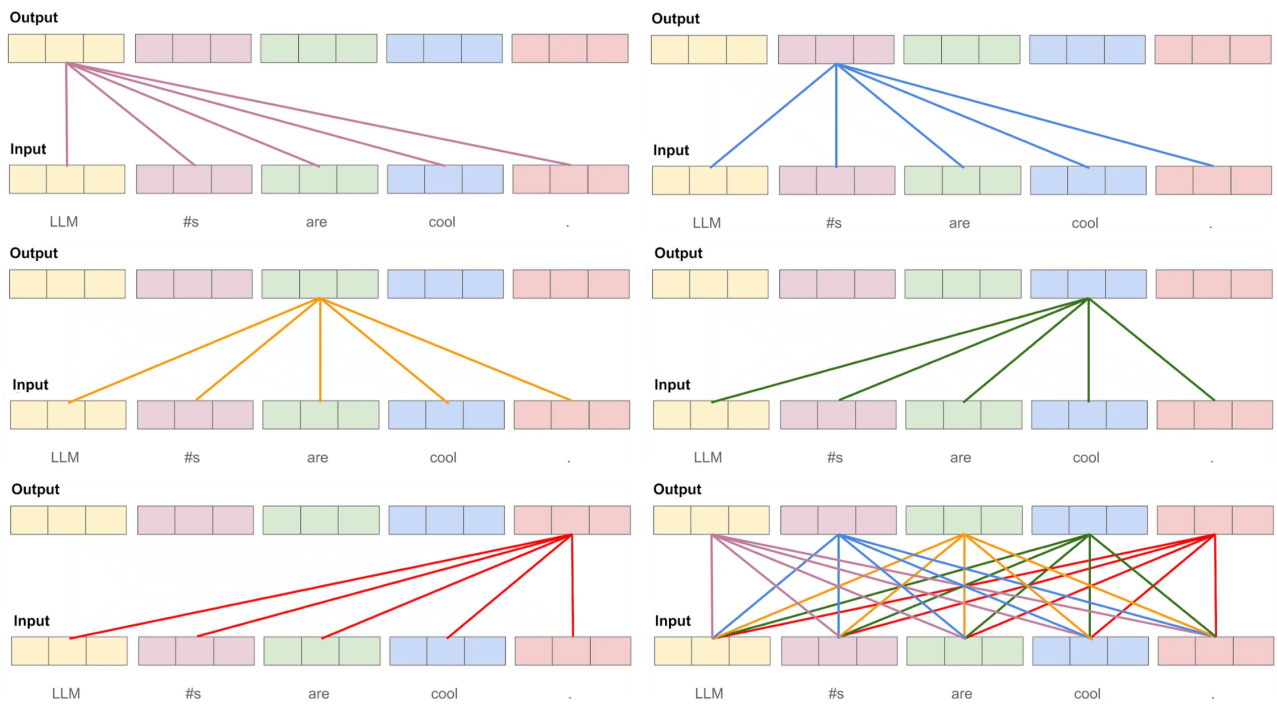


Figure 6.6: Illustration of the dependency between input tokens and attention's output [209].

We compute $a[i, j]$ by taking the dot product of the query vector for token i with the key vector for token j . Figure 6.9 depicts the computation of the attention score for a token pair.

We can efficiently compute all pairwise attention scores in a sequence by (i) stacking the query and key vectors into two matrices and (ii) multiplying the query matrix with the transposed key matrix. This operation outputs a matrix of size $[T, T]$, called the attention matrix, that contains all pairwise attention scores over the entire sequence. Then, we divide each value in the attention matrix by the square root of d , an approach that has been found to improve training stability [199], and apply a `softmax` operation to each row of the attention matrix (Figure 6.10). After the `softmax` has been applied, each row of the attention matrix is a valid probability distribution, where the row contains positive values that sum to one. The i -th row of the attention matrix stores the attention scores between the i -th token and all tokens in the input sequence.

Computing output

Once we have the attention scores, the output of self-attention for a token is a weighted combination of the value vectors, where the weights are the attention scores. To compute this output, we simply multiply the attention matrix by the value matrix as shown in Figure 6.10. Self-attention preserves the size of its input, since a transformed d -dimensional vector is produced for each input token.

Masked self-attention

Until now, we discussed unmasked self-attention. However, as mentioned previously, the decoder-only transformers use masked self-attention, which modifies the underlying attention pattern by masking out the tokens that come after the token under analysis in the sequence. Each token can only consider tokens that come before it, the posterior tokens are masked.

Let us consider a token sequence ["LLM", "#s", "are", "cool", "."] and compute masked attention scores for the token "are". In unmasked self-attention, we would compute an attention score between "are" and every token in the sequence. However, in masked self-attention, we only compute attention scores for "LLM", "#s", and "are". Masked self-attention prohibits us from looking forward in the sequence. In practice, this is achieved by simply setting all attention scores for these tokens to negative infinity, yielding a pairwise probability of zero for masked tokens after the application of `softmax` (Figure 6.11).

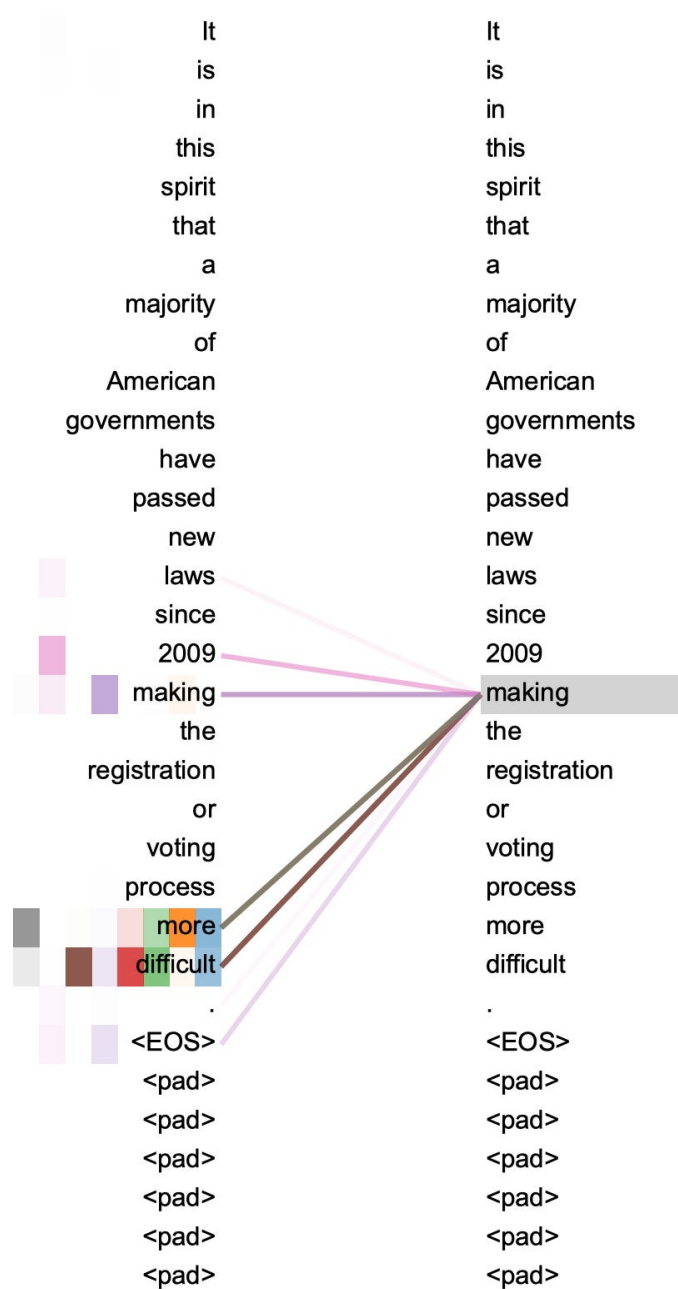


Figure 6.7: An example of the attention mechanism following long-distance dependencies in the encoder self-attention in layer 5 of 6. Many of the attention heads attend to a distant dependency of the verb **making**, completing the phrase **making...more difficult**. Attentions here shown only for the word **making**. Different colors represent different heads [199].

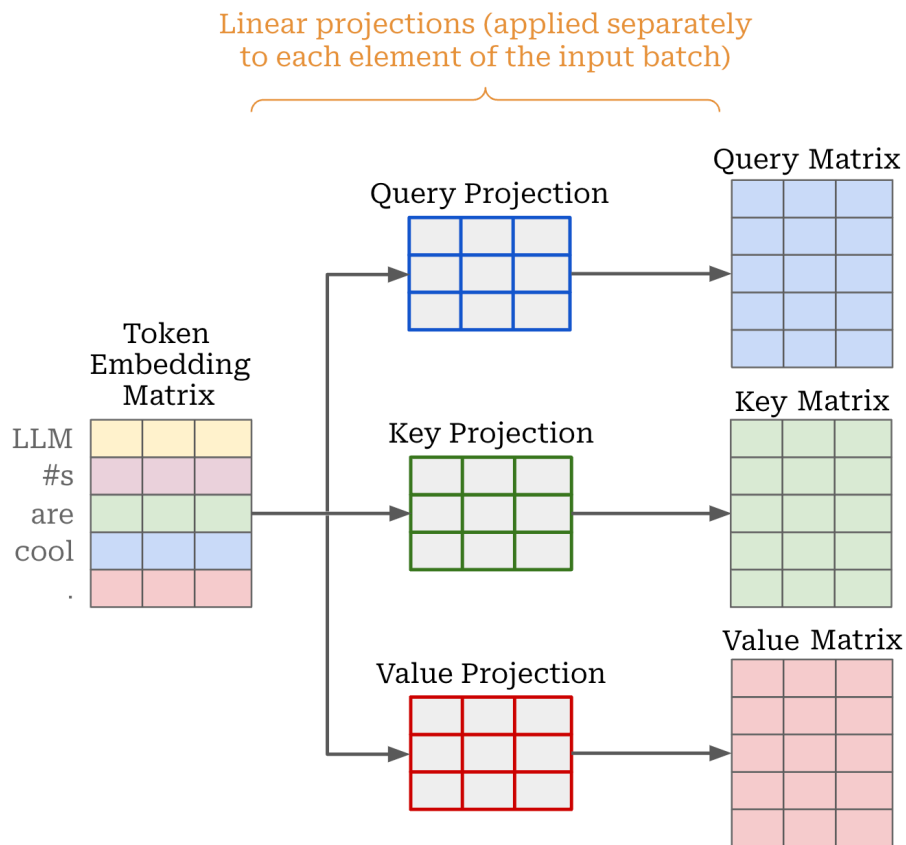


Figure 6.8: Creating key, query and value tensors [209].

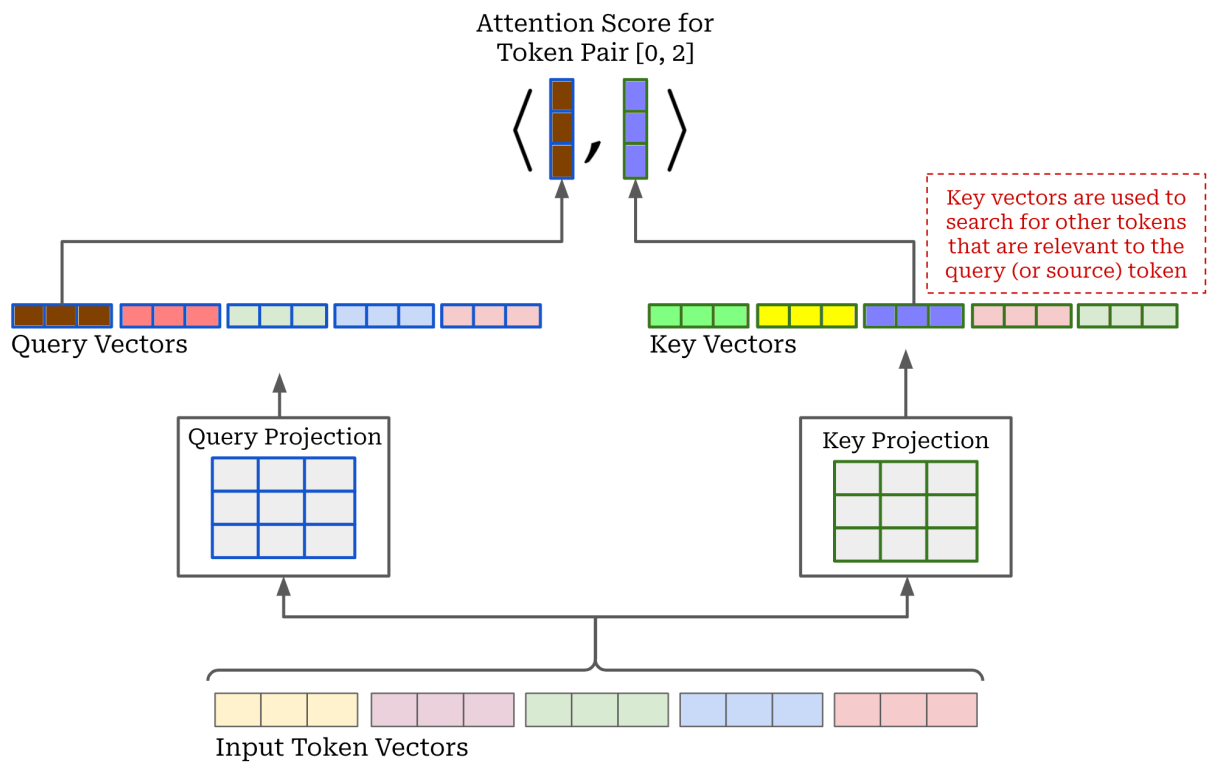


Figure 6.9: Computing an attention score for a token pair [209].

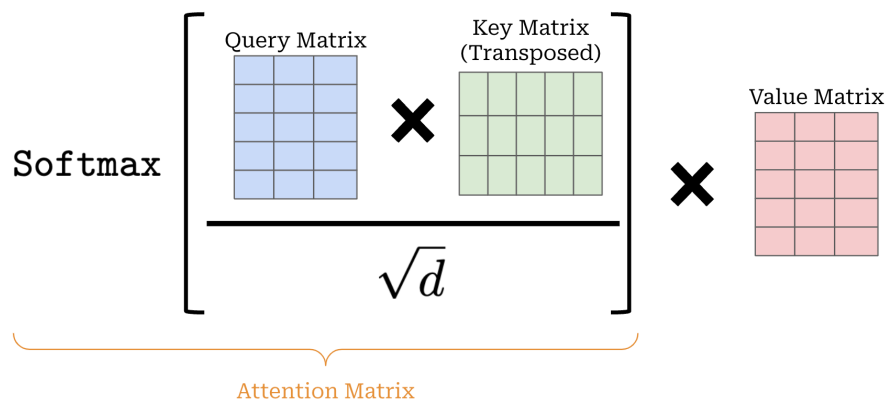


Figure 6.10: Computing attention scores and output in self-attention [209].

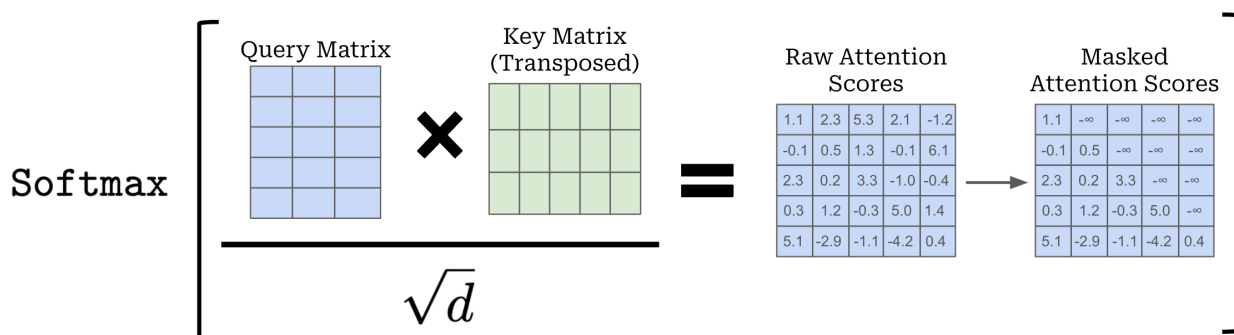


Figure 6.11: Computing masked attention scores [209].

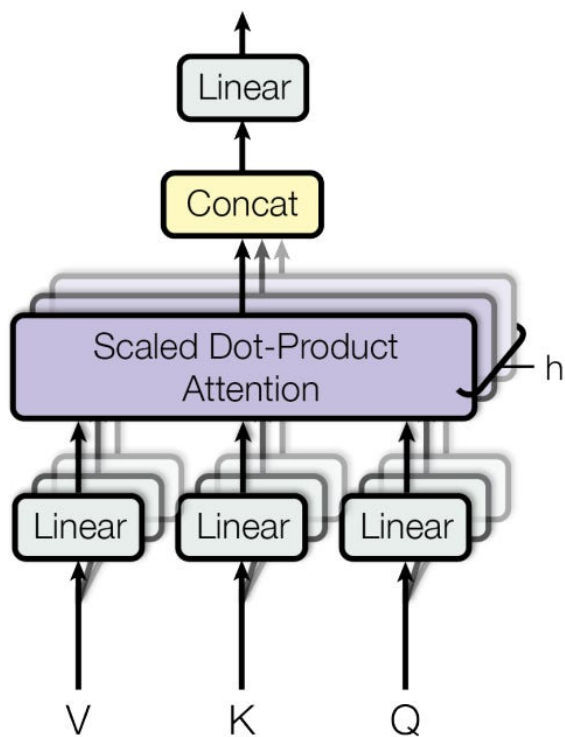


Figure 6.12: Multi-head attention [199].

Attention heads

The attention operation we have described so far uses `softmax` to normalize attention scores that are computed across the sequence. Although this approach forms a valid probability distribution, it also limits the ability of self-attention to focus on multiple positions within the sequence, the probability distribution can easily be dominated by one or a few words. To solve this issue, we typically compute attention across multiple heads in parallel (Figure 6.12).

Within each head, the masked attention operation is identical. However, (i) we use separate key, query, and value projections for each attention head, and (ii) reduce the dimensionality of the key, query, and value vectors, which can be done by modifying the linear projection, to reduce computational costs.

More specifically, we will change the dimensionality of vectors in each attention head from d to d/H , where H is the number of attention heads, to keep the computational cost of multi-headed self-attention essentially constant.

Now, we have several attention heads that compute self-attention in parallel, but we still need to produce a single output representation from the multiple heads of our self-attention module. We have several options for combining the output of each attention head, such as concatenation, averaging, or projecting. The original implementation of multi-headed self-attention concatenates the output of each head and then linearly projects the concatenated output (Figure 6.13). Since each attention head outputs token vectors of dimension d/H , the concatenated output of all attention heads has dimension d . Thus, the multi-headed self-attention operation still preserves the original size of the input.

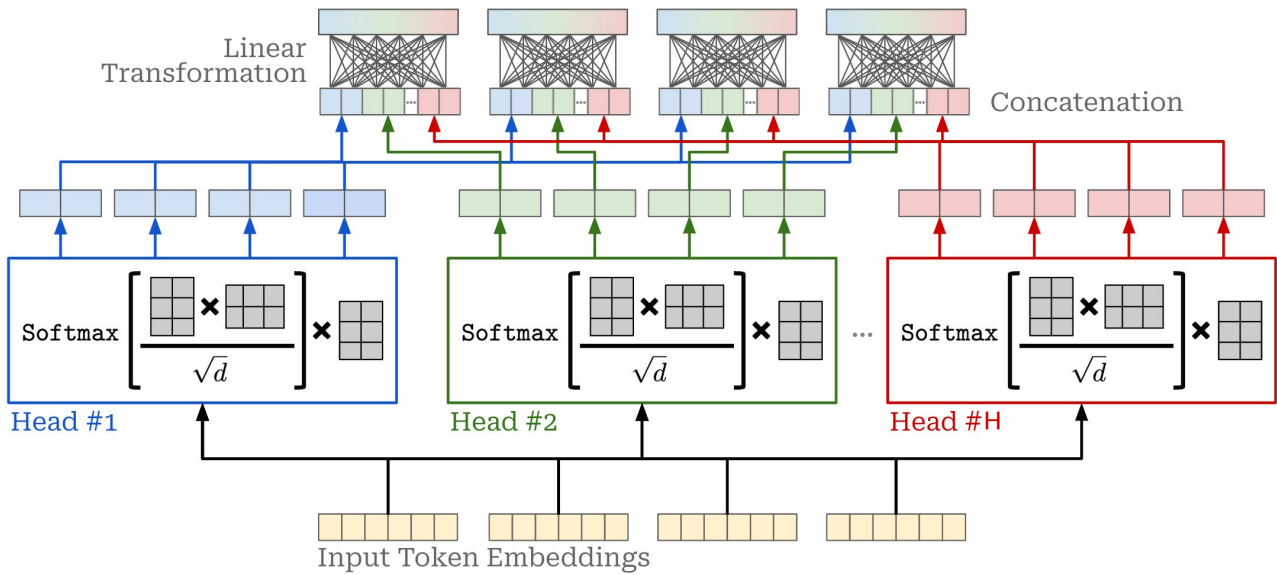


Figure 6.13: Combining the output of multiple attention heads [209].

Implementation

An implementation of masked multi-headed self-attention is included in Listing 6.2. Here, we go beyond a single input sequence of size $[T, d]$ and process a batch of inputs of size $[B, T, d]$. This code implements each of the components that we described.

- Lines 43-49: computes the key, query and value projections using a single linear projection for each attention head and splits/reshapes them as necessary.
- Lines 51-55: computes the attention scores, masks the attention scores and then applies a `softmax` transformation to the result.
- Lines 57-58: computes the output vectors by taking the product of the attention matrix and the value matrix.

- Lines 60-62: concatenates the outputs from each attention head and applies a linear projection to obtain the final output.

```

1 # Source: https://github.com/karpathy/nanoGPT/blob/master/model.py
2 import math
3 import torch
4 from torch import nn
5 import torch.nn.functional as F
6
7 class CausalSelfAttention(nn.Module):
8     def __init__(self, d, H, T, bias=False, dropout=0.2):
9         """
10         Arguments:
11             d: embedding vector dimension
12             H: number of attention heads
13             T: maximum length of input sequences (in tokens)
14             bias: whether or not to use bias in linear layers
15             dropout: probability of dropout
16         """
17         super().__init__()
18         assert d % H == 0
19
20         # Key, query, value projections for all heads, but in a batch
21         # output it has a 3x dimension because it includes key, query and value
22         self.c_attn = nn.Linear(d, 3*d, bias=bias)
23
24         # Projection of concatenated attention head outputs
25         self.c_proj = nn.Linear(d, d, bias=bias)
26
27         # Dropout layers
28         self.attn_dropout = nn.Dropout(dropout)
29         self.resid_dropout = nn.Dropout(dropout)
30         self.H = H
31         self.d = d
32
33         # Causal mask to ensure that attention is only applied to
34         # the left in the input sequence
35         self.register_buffer(
36             "mask", torch.tril(torch.ones(T, T)).view(1, 1, T, T)
37         )
38
39     def forward(self, x):
40         B, T, _ = x.size() # batch size, sequence length, embedding dimensionality
41
42         # Compute query, key, and value vectors for all heads in batch.
43         # Split the output into separate query, key, and value tensors.
44         q, k, v = self.c_attn(x).split(self.d, dim=2) # [B, T, d]
45         # Reshape tensor into sequences of smaller token vectors for each head
46         k = k.view(B, T, self.H, self.d // self.H).transpose(1, 2) # [B, H, T, d // H]
47         q = q.view(B, T, self.H, self.d // self.H).transpose(1, 2)
48         v = v.view(B, T, self.H, self.d // self.H).transpose(1, 2)
49
50         # Compute the attention matrix, perform masking, and apply dropout
51         att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1))) # [B, H, T, T]
52         att = att.masked_fill(self.mask[:, :, :, T:T] == 0, float('-inf'))
53         att = F.softmax(att, dim=-1)
54         att = self.attn_dropout(att)
55
56         # Compute output vectors for each token
57         y = att @ v # [B, H, T, d // H]
58
59         # Concatenate outputs from each attention head and project linearly
60         y = y.transpose(1, 2).contiguous().view(B, T, self.d)
61         y = self.resid_dropout(self.c_proj(y))
62         return y

```

Listing 6.2: Implementation of masked multi-headed self-attention [209].

Although we use some matrix manipulations and operations in PyTorch, the implementation exactly matches the described functionality for the masked multi-head self-attention.

6.1.3 Feed-Forward Network

In addition to masked self-attention, each block of the transformer contains a feed-forward network (Figure 6.14). In this block, we pass each token vector of the sequence through the same feed-forward neural network. Usually, this is a two-Linear layer network with a non-linear activation in between, for example, a ReLU, GeLU or SwiGLU activation [175], in the hidden layer. In most cases, the dimension of the hidden layer is larger than the original dimension of the token embeddings, for example, 4 times larger. Implementing a feed-forward neural network in PyTorch is accomplish with a Linear layer, a GeLU activation, a second Linear layer and Dropout (Listing 6.3).

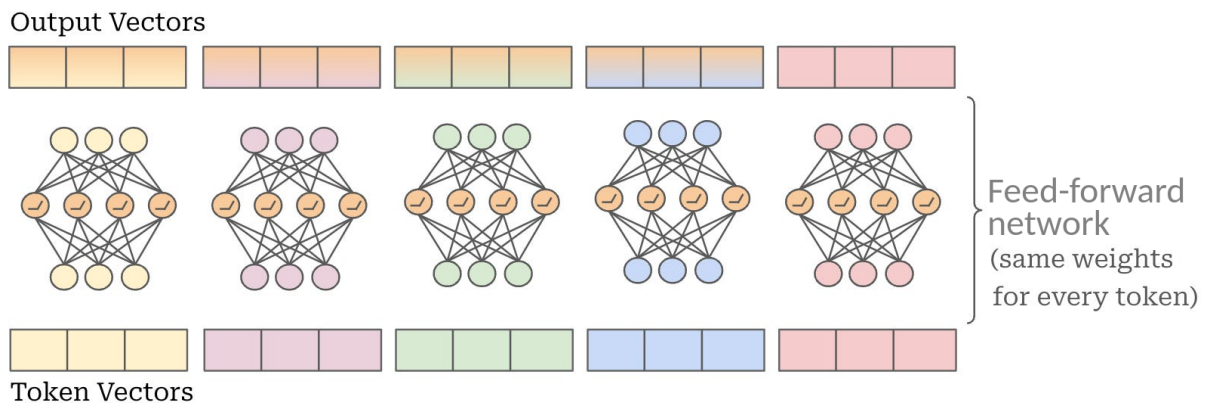


Figure 6.14: Feed-forward network [209].

```

1 from torch import nn
2
3 class MLP(nn.Module):
4
5     def __init__(self, d, bias=False, dropout=0.2):
6         """
7         Arguments:
8             d:          embedding vector dimension
9             bias:       whether to use bias in linear layers or not
10            dropout:    probability of dropout
11         """
12
13         super().__init__()
14         self.fc1 = nn.Linear(d, 4 * d, bias=bias)
15         self.gelu = nn.GELU()
16         self.fc2 = nn.Linear(4 * d, d, bias=bias)
17         self.dropout = nn.Dropout(dropout)
18
19     def forward(self, x):
20         x = self.fc1(x)
21         x = self.gelu(x)
22         x = self.fc2(x)
23         x = self.dropout(x)
24         return x

```

Listing 6.3: Implementation of the feed-forward network [209].

6.1.4 Decoder-Only Transformer Block

To build a decoder-only transformer block, we use both the masked multi-head self-attention and the feed-forward network that we have discussed, as well as two normalization operations and two residual connections. A depiction of the full decoder-only transformer block is shown Figure 6.15.

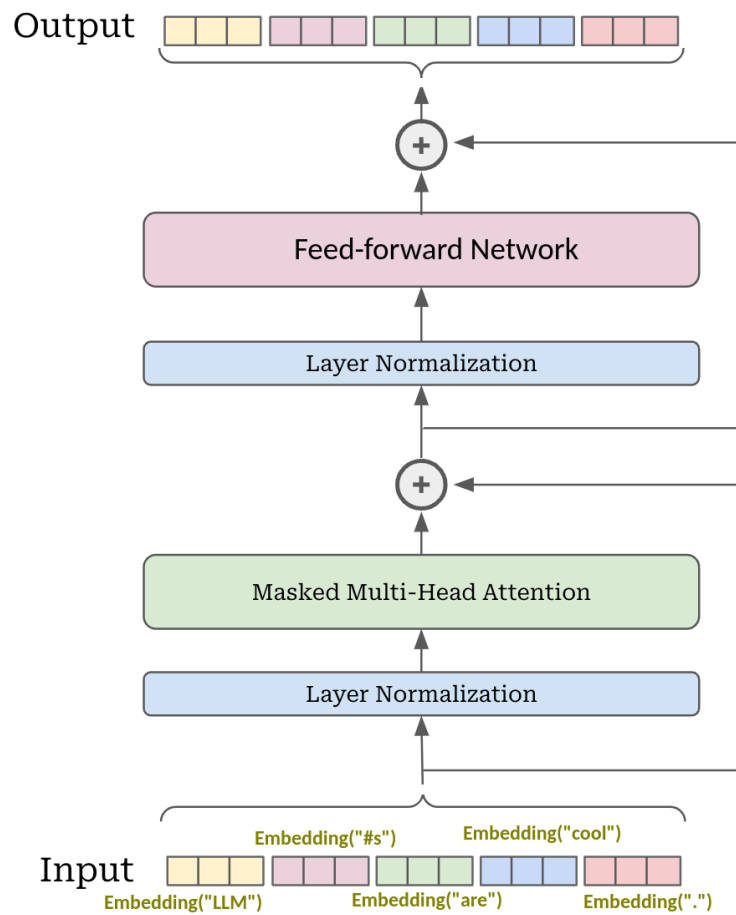


Figure 6.15: Decoder-only transformer block [209].

A residual connection [78] simply adds the input of a neural network layer to the output of this layer before passing this representation to the next layer, as opposed to solely passing the layer's output to the next layer without adding the input (Figure 6.16).

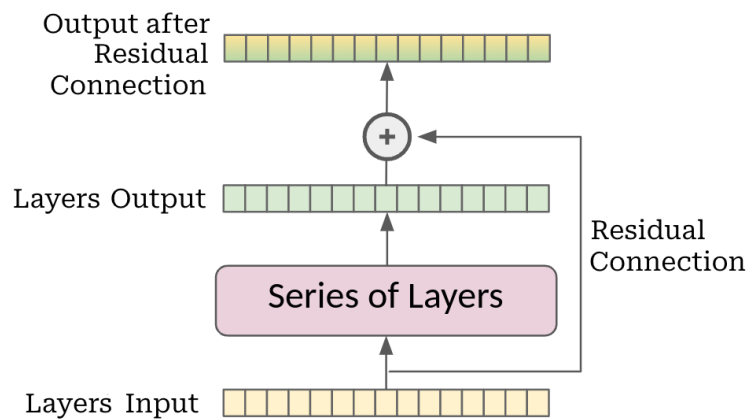


Figure 6.16: Residual connection in a generic neural network layer [209].

Residual connections are widely used within deep neural networks and can be applied to any series of layers. Adding residual connections helps to avoid issues with vanishing and exploding gradients, and generally improves the stability of training by providing a short cut that allows gradients to flow freely through the network during backward pass.

Normalizing the input, or output, of a neural network layer can also aid training stability. Although many types of normalization exist, the most commonly used normalization variant in transformers is layer normalization (Equation 6.1). Here, the normalization operation has two components, performing normalization and applying

a learnable affine transformation. In other words, we multiply the normalized values by the weight γ and add a bias β instead of directly using the normalized output. Both the weight and bias are learnable parameters that can be trained along with other network weights.

$$\text{LayerNorm}(x) = \frac{x - \text{mean}(x)}{\text{variance}(x) + \epsilon} \cdot \gamma + \beta \quad (6.1)$$

Where the mean and variance are calculated over all values within each input sequence, ϵ is small constant to avoid dividing by zero, γ and β are learnable parameters.

Transformer block implementation

A decoder-only transformer block implementation is provided in Listing 6.4. Here, we use our prior attention and feed-forward transformation implementations. By using the modules we have already defined, the decoder-only transformer block implementation actually becomes quite simple.

```

1 # Source: https://github.com/karpathy/nanoGPT/blob/master/model.py
2 from torch import nn
3
4 class Block(nn.Module):
5     def __init__(self, d, H, T, bias=False, dropout=0.2):
6         """
7         Arguments:
8             d:          embedding vector dimension
9             H:          number of attention heads
10            T:          maximum length of input sequences (in tokens)
11            bias:       whether or not to use bias in Linear layers
12            dropout:    probability of dropout
13         """
14         super().__init__()
15         self.ln_1 = nn.LayerNorm(d)
16         self.attn = CausalSelfAttention(d, H, T, bias, dropout)
17         self.ln_2 = nn.LayerNorm(d)
18         self.ffn = MLP(d, bias, dropout)
19
20     def forward(self, x):
21         x = x + self.attn(self.ln_1(x))
22         x = x + self.ffn(self.ln_2(x))
23         return x

```

Listing 6.4: Implementation of the decoder-only transformer block [209].

6.1.5 Decoder-Only Transformer Architecture

After understanding the type of input we have apply at the transformer input and implementing the transformer block, to implement the decoder-only transformer we just repeat the transformer block L times. For each block, the size of the model's input $[B, T, d]$ is maintained, so the output of the L -th transformer block is also a tensor of this size (Figure 6.17).

A full implementation of a decoder-only transformer architecture is included in Listing 6.5. The architecture contains several components, including two embedding layers for tokens and positions, the L transformer blocks, and a final classifier that includes a layer normalization and a Linear layer, for performing next token prediction given an output token embedding as input. The model operates by just passing its input, a set of input token IDs with size $[B, T]$, through each of these components to produce a set of output token IDs.

```

1 # Source: https://github.com/karpathy/nanoGPT/blob/master/model.py
2 import torch
3 from torch import nn
4 import torch.nn.functional as F
5
6 class Transformer(nn.Module):
7
8     def __init__(self, d, H, T, V, layers, bias=False, dropout=0.2):
9         """
10         Arguments:
11             d:      embedding vector dimension
12             H:      number of attention heads
13             T:      maximum length of input sequences (in tokens)
14             V:      size of the token vocabulary
15             layers: number of decoder-only blocks
16             bias:   whether or not to use bias in linear layers
17             dropout: probability of dropout
18         """
19         super().__init__()
20         self.transformer = nn.ModuleDict(dict(
21             wte = nn.Embedding(V, d), # token embeddings
22             wpe = nn.Embedding(T, d), # position embeddings
23             drop = nn.Dropout(dropout),
24             blocks = nn.ModuleList([Block(d,H,T,bias,dropout) for _ in range(layers)]),
25             ln_f = nn.LayerNorm(d),
26             head = nn.Linear(d, V, bias=bias),
27         ))
28
29     def forward(self, tokenIDs, targets=None):
30         # tokenIDs is a [B, T] matrix of token indices
31         # targets is a [B, T] matrix of target (next) token indices
32         device = tokenIDs.device
33         _, T = tokenIDs.size() # [B, T]
34         pos = torch.arange(0, T, dtype=torch.long, device=device)
35
36         # Generate token and position embeddings
37         tok_emb = self.transformer.wte(idx) # [B, T, d]
38         pos_emb = self.transformer.wpe(pos) # [T, d]
39         x = self.transformer.drop(tok_emb + pos_emb)
40
41         # Pass 'x' through all decoder-only blocks
42         for block in self.transformer.blocks:
43             x = block(x)
44         x = self.transformer.ln_f(x) # Final normalization
45
46         if targets is not None:
47             # Compute the loss if we provided targets as input
48             logits = self.transformer.head(x)
49             loss = F.cross_entropy(
50                 logits.view(-1, logits.size(-1)),
51                 targets.view(-1),
52                 ignore_index=-1,
53             )
54         else:
55             # Only look at last token when performing inference
56             logits = self.transformer.head(x[:, [-1], :])
57             loss = None
58
59         return logits, loss

```

Listing 6.5: Implementation of the decoder-only transformer [209].

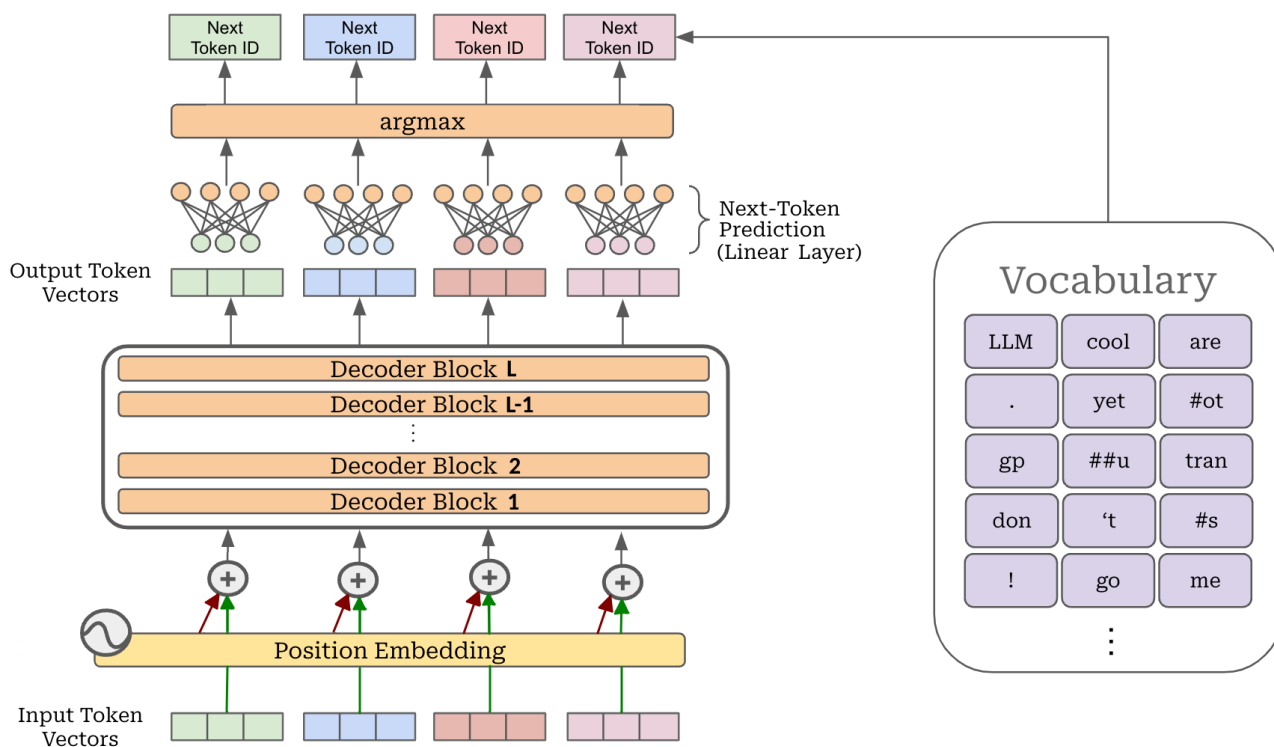


Figure 6.17: Predicting next tokens with a large language model [209].

Next-token prediction

LLMs are trained specifically to perform next-token prediction, meaning they are specialized in predicting the next token given a list of tokens as input. We have seen that the transformer's output is a list of token vectors for each input token. Thus, it can predict the next token for any of these inputs tokens by using the following steps.

- Pick the output embedding for a particular token.
- Pass the selected embedding through a Linear layer whose output has the dimension of the model's vocabulary.
- Apply an `argmax` function to the Linear layer's output to get the ID of the token that corresponds to the maximum value.

To generate a sequence of text, we repeat these three steps multiple times. We apply a textual prompt at the transformer input, pass everything through the decoder-only transformer, pick the last token vector in the transformer output sequence, predict the next token using the final Linear layer and the `argmax` function, append the obtained next token to the input sequence, and repeat these steps. This autoregressive decoding process is used by all LLMs for generating their output (Figure 6.18).

After completing the transformer implementation, we should clarify why most LLMs only use the decoder component of the original encoder-decoder transformer. The key distinction between encoder and decoder in a transformer is the type of attention they use. Encoder uses unmasked self-attention, meaning all tokens in the sequence, including those before and after a given token, are considered by the self-attention mechanism. In contrast, the decoder uses masked self-attention, which prevents a token from attending to those tokens that follow it in the sequence (Figure 6.19).

Due to the masked self-attention, decoders fit well on the next token prediction task. If the model can look forward in the sequence when predicting the representation of the next token, then it could simply cheat and directly copy the next token in the sequence to be used as prediction. Masked self-attention forces the model to learn generalizable patterns for predicting the next token from those that come before it, making the decoder adequate for LLMs.

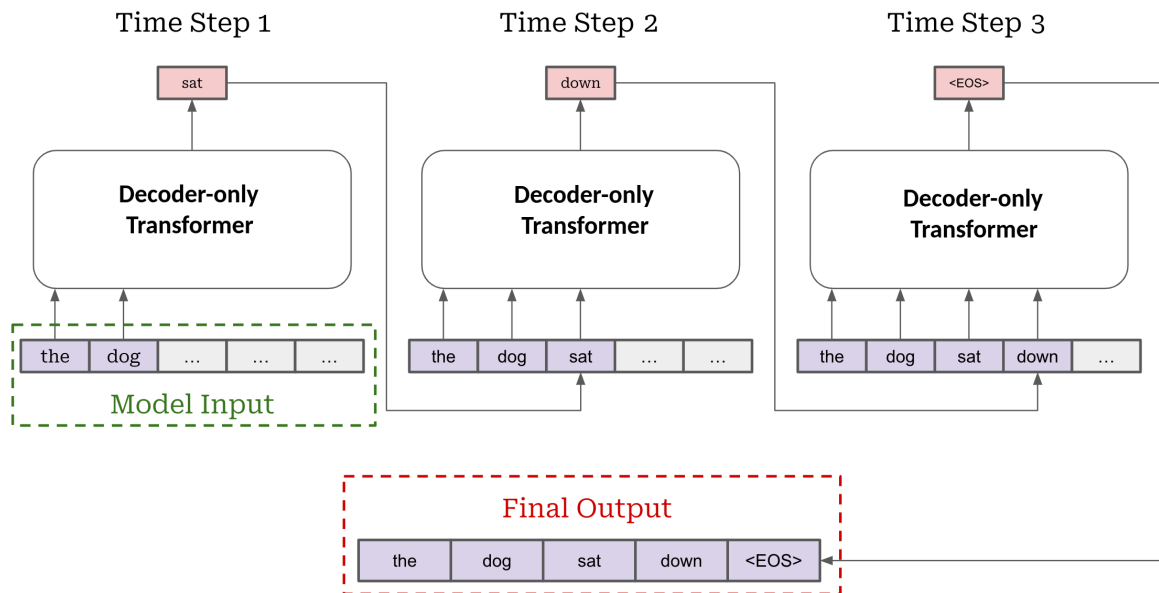


Figure 6.18: Autoregressive output generation where in each step the model predicts the next token in the sequence [209].

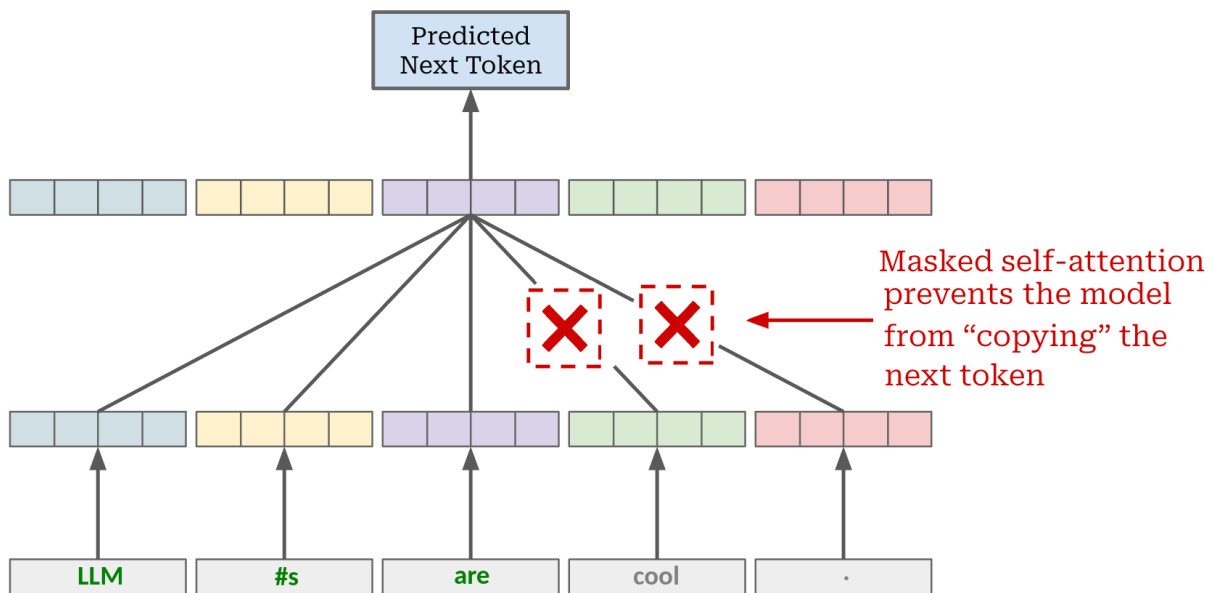


Figure 6.19: Causal mask for next token prediction [209].

6.2 Build a Mixture-of-Experts Model

Deep neural networks usually process all inputs using the same weights. Mixture of Experts (MoE) models modify this architectural behavior and instead apply different weights to each sample. The result is a sparsely-activated model, with a huge number of weights, but requiring a computational cost that is not proportional to the model size [61].

After gaining intuition on the decoder-only transformers, we will create a Mixture-of-Experts (MoE) model. MoE-based LLMs adopt the same decoder-only transformer architecture, but the architecture is modified in a few ways. Modifying the transformer architecture is not difficult, but there are a few aspects that must be implemented correctly for the model to work correctly. Additionally, training a MoE model properly requires some extra attention and understanding, since MoE models are more difficult to train than dense transformers.

6.2.1 Expert Layers

Compared to the dense decoder-only transformer, the main modification introduced by an MoE model is in the feed-forward network of the transformer block. Usually, this block includes one feed-forward network that is applied to all token vectors. Instead of having a single feed-forward network, an MoE uses multiple feed-forward networks, each one with independent weights. We denote each feed-forward network as an expert, and a module containing multiple experts is an expert layer. If we have N experts in a layer, we can refer to the i -th expert using the notation E_i (Equation 6.2).

$$\text{Experts: } \{E_i(\cdot)\}_{i=1}^N \quad (6.2)$$

PyTorch Implementation

Listing 6.6 contains the implementation of an expert layer in PyTorch. It adopts an architecture identical to the feed-forward neural network that was implemented in Listing 6.3, but here we use several feed-forward networks instead of one. The main complexity of this implementation is that we do not use standard `Linear` layers of PyTorch. Instead, we instantiate the weights and biases of all experts using `Parameter` objects, so that we can compute the output of all experts in batch by using a batched matrix multiplication (BMM). In this way, we avoid having to iterate over the experts to compute all outputs, which improves the computation efficiency.

```
1 """
2 Based upon ColossalAI OpenMoE:
3 https://github.com/hpcaitech/ColossalAI/blob/main/colossalai/moe/experts.py
4 """
5 import torch
6 from torch import nn
7
8 class MLPExperts(nn.Module):
9
10     def __init__(self, d, n_exp=8, bias=False, dropout=0.2):
11         """
12         Arguments:
13             d: embedding vector dimension
14             n_exp: the number of experts to create in the expert layer
15             bias: whether or not to use bias in linear layers
16             dropout: probability of dropout
17         """
18         super().__init__()
19         self.bias = bias
20         self.fc1_weights = nn.Parameter(torch.empty(n_exp, d, 4 * d))
21         self.fc1_bias = nn.Parameter(torch.empty(n_exp, 1, 4 * d)) if self.bias else None
22         self.fc2_weights = nn.Parameter(torch.empty(n_exp, 4 * d, d))
23         self.fc2_bias = nn.Parameter(torch.empty(n_exp, 1, d)) if self.bias else None
24         self.gelu = nn.GELU()
25         self.dropout = nn.Dropout(dropout)
26
27     def forward(self, x):
28         x = torch.bmm(x, self.fc1_weights)
29         if self.bias:
30             x += self.fc1_bias
31         x = self.gelu(x)
32         x = torch.bmm(x, self.fc2_weights)
33         if self.bias:
34             x += self.fc2_bias
35         x = self.dropout(x)
36         return x
```

Listing 6.6: Implementation of an expert layer [209].

Creating an MoE

To create the MoE-based decoder-only transformer, we convert the transformer's feed-forward networks to expert layers. Each expert within the expert layer has an architecture that is identical to the original, feed-forward network from that layer. We just include several independent feed-forward networks within the same expert layer (Figure 6.20).

Expert Layer

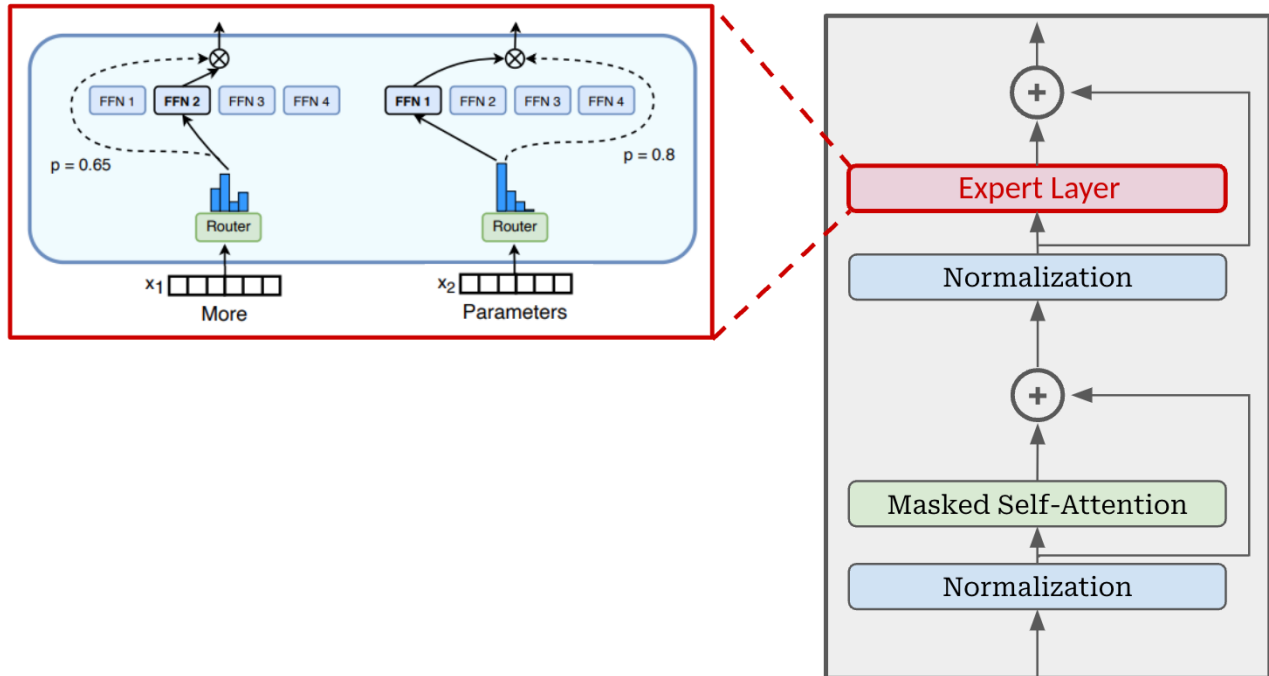


Figure 6.20: Including multiple experts in the same feed-forward network [199].

However, we do not want to replace every feed-forward network (FFN) in the dense transformer by an expert layer. Most MoE-based LLMs use a stride P , meaning that only one FFN in every P FFNs is converted into an expert layer and the other FFNs are left unaltered. For example, the ST-MoE model has 32 experts in each expert layer and uses a frequency of replacement equal to $1/4$, meaning that one in every four FFNs is replaced by an expert layer [246].

Listing 6.7 exemplifies how to selectively add a dense transformer block or an expert layer to the transformer architecture. The frequency with which an expert layer is interleaved between dense transformer blocks determines the total number of experts in the MoE, and functions as a mechanism for balancing performance and efficiency.

```

1 transformer_blocks = []
2 for i in range(num_blocks):
3     use_moe = (i % P) == 0
4
5     # When use_moe = False, it is inserted a dense transformer block.
6     # When use_moe = True, it is inserted an expert layer.
7     transformer_blocks.append(Block(use_moe=use_moe))

```

Listing 6.7: Interleaving expert layers with dense transformer blocks [209].

6.2.2 Routing Tokens to Experts

The primary benefit of MoE-based architectures is their efficiency, but using experts alone does not improve efficiency. In fact, adding more experts to each layer of the model significantly increases the total number of weights, and the amount of necessary computation. To improve efficiency, we need to sparsely select and use only a subset of experts from each expert layer. By sparsely utilizing experts, we can get the benefits of a much larger model without a significant increase in the computational costs of training and inference. Using an MoE

architecture makes it possible to attain better trade-offs between model quality and inference efficiency than dense models typically achieve.

Selecting experts

Let us consider a single token, represented by a d -dimensional token vector. Our goal is to select a subset of experts, with size k , to process this token. Thus, we need an algorithm to compute and optimize the routing of the token to the selected experts.

In the simplest routing strategy we apply a linear transformation to the token vector, obtaining a vector of size N , the number of experts. Then, we can apply a **softmax** function to form a probability distribution over the set of experts for our token (Figure 6.21). We can use this distribution to choose experts to which our token should be routed by selecting the top- K experts in the distribution. In addition to allowing the selection of experts, the top- k probabilities will also be used to obtain an output that combines the outputs produced by the k selected experts.

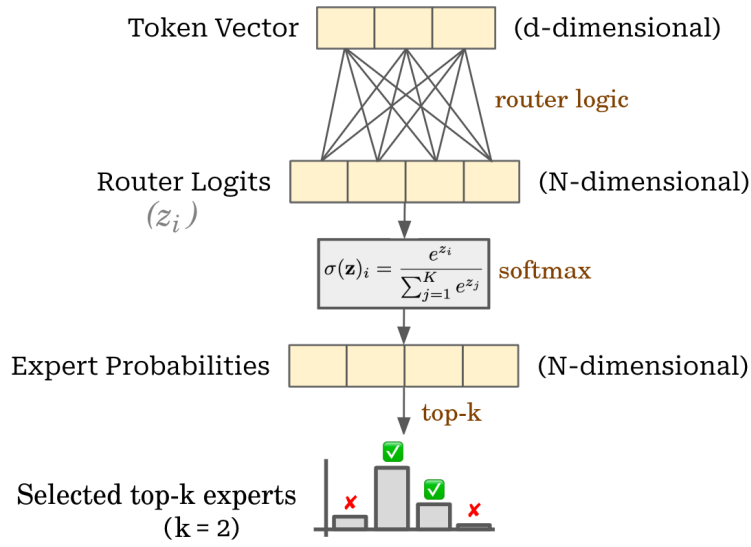


Figure 6.21: Routing mechanism for a single token [209].

Softmax router implementation

As described above, this routing strategy just uses a Linear layer. An implementation of this **softmax** router is included in Listing 6.8, where the output of the router is (i) a set of top- K expert indices for each token in the input and (ii) the top- K expert probabilities, i.e., the probability values for each of the top- K indices, associated with the selected experts. Despite its simplicity, this routing strategy is effective and most modern MoEs adopt a similar linear routing scheme plus a **softmax**.

Optionally, we can add noise into the router logic, an approach proposed in [51], one of the earliest works on applying MoEs to neural networks. By adding a small amount of learnable noise to the router logits, we can help to regularize the MoE's training process. Equations 6.3, 6.4, and 6.5 formalize the routing strategy when noise is added to the router logits.

$$H(x)_i = (x \cdot W_g)_i + \mathcal{N}(0, 1) \cdot \text{softplus}((x \cdot W_{\text{noise}})_i) \quad (6.3)$$

$$G(x) = \text{softmax}(\text{KeepTopK}(H(x), k)) \quad (6.4)$$

$$\text{KeepTopK}(v, k)_i = \begin{cases} v_i & \text{if } v_i \text{ is in the top-}k \text{ elements of } v \\ -\infty & \text{otherwise} \end{cases} \quad (6.5)$$

where $(x \cdot W_g)_i$ is the regular router logits, $\mathcal{N}(0, 1) \cdot \text{softplus}(H(x), k)$ represents the noise added to the router logits, W_{noise} are the learnable weights that control the level of added noise, and the top- k router's outputs beyond the top k are set to zero.

```

1 import torch
2 from torch import nn
3 from torch.nn import functional as F
4
5 class BasicSoftmaxRouter(nn.Module):
6     def __init__(self, d, n_exp = 8, top_k = 2, use_noisy_top_k = True):
7         """
8         Arguments:
9             d:                embedding vector dimension
10            n_exp:             the number of experts to create in the expert layer
11            top_k:             the number of active experts for each token
12            use_noisy_top_k:   whether to add noise when computing expert output
13        """
14        super().__init__()
15
16        # Router settings
17        self.top_k = top_k
18        assert self.top_k >= 1 and self.top_k <= n_exp
19        self.use_noisy_top_k = use_noisy_top_k
20
21        # Linear projection for (noisy) softmax routing.
22        # It is not used bias (see page 4, equation 4, https://arxiv.org/abs/1701.06538)
23        self.w_g = nn.Linear(d, n_exp, bias=False)
24        self.w_noise = nn.Linear(d, n_exp, bias=False) if self.use_noisy_top_k else None
25
26    def forward(self, x):
27        # Equation 4 in https://arxiv.org/abs/1701.06538
28        logits = self.w_g(x) # [B, T, d] -> [B, T, n_exp]
29        if self.use_noisy_top_k:
30            # Optionally, add noise to the router logits.
31            # Softplus is a smooth approximation to ReLU function that is always positive
32            # Softplus(x) = 1/beta * log(1+exp(beta.x))
33            noise = F.softplus(self.w_noise(x))
34            noise *= torch.randn_like(noise)
35            logits += noise
36        top_k_logits, top_k_indices = logits.topk(self.top_k, dim=-1) # [B, T, k]
37        return top_k_logits, top_k_indices

```

Listing 6.8: Implementation of the softmax router [209].

Active weights

Because we only select a subset of experts to process each token within an expert layer, there is a concept of active weights. Only a portion of the MoE model's weights are active, given by the experts selected from each expert layer, when processing a given token. The total computation performed by the MoE is proportional to the number of active weights rather than the total number of weights.

6.2.3 Expert Capacity

To improve hardware utilization, most implementations of sparse models have static batch sizes for each expert. The expert capacity refers to the number of tokens that can be routed to each expert. If this capacity is exceeded, the tokens in excess are passed unaltered to the next layer through a residual connection [246].

The computation performed in an expert layer is dynamic. We choose the tokens to be computed by each expert based on the output of the router, which changes depending upon the sequences of tokens provided as input to the MoE. The dynamic nature of the input for each expert can make the implementation of an expert layer complicated, since each expert's input will have a different and unpredictable size.

Expert capacity

Most practical implementations of MoEs avoid this problem by using fixed batch sizes for each expert, this is a useful trick for improving hardware utilization. Each expert uses the same static batch size, referred to as expert capacity. The expert capacity, defined in the Equation 6.6, dictates the maximum number of tokens in each batch that can be assigned to any expert.

$$\text{expert_capacity} = \frac{K \cdot B \cdot T}{N} \cdot \text{capacity_factor} \quad (6.6)$$

Where K is the number of active tokens at every moment, $B \cdot T$ is the number of tokens in a minibatch, and N is the number of experts.

Expert capacity is controlled via the capacity factor parameter. A capacity factor of 1 means that tokens are routed uniformly to all experts, while setting the capacity factor greater than 1 provides an extra capacity to handle imbalanced token routing between experts, but this comes at the cost of higher memory usage and lower efficiency.

If the number of tokens routed to an expert exceeds the expert capacity, then we drop these extra tokens by performing no computation and letting their representation flow directly to the next layer via the transformer's residual connection. Figure 6.22 illustrates token routing dynamics in switch transformer, where a capacity factor of 1.5 helps to reduce the number of dropped tokens. MoEs perform well with relatively low capacity factors, but we should make sure to avoid too many tokens being dropped. The capacity factor can also be different during training and evaluation. For example, ST-MoE [246] uses a capacity factor of 1.25 and 2.0 during training and evaluation, respectively.

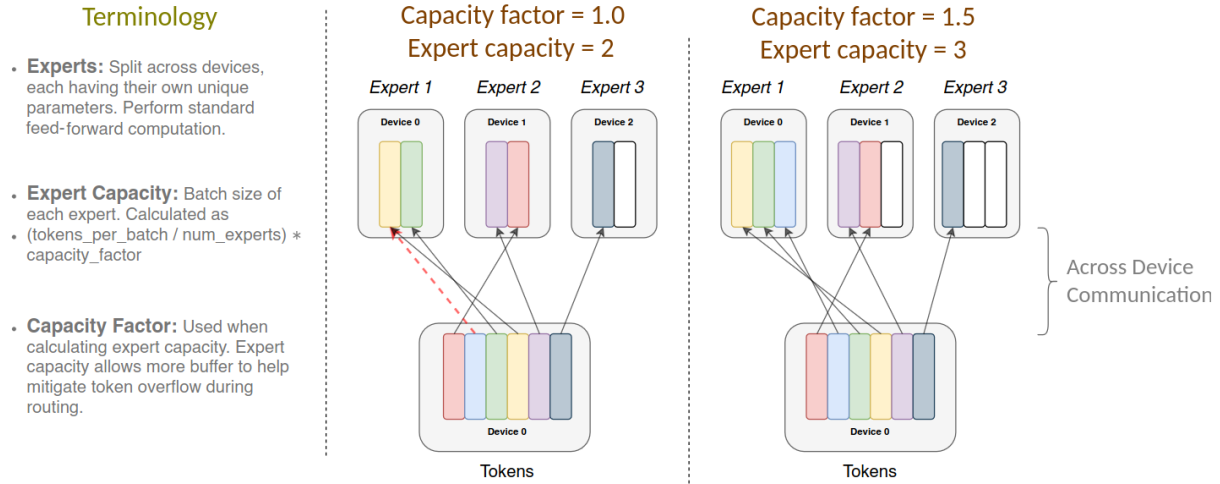


Figure 6.22: Illustration of token routing dynamics in switch transformer. Each expert processes a fixed batch-size of tokens modulated by the capacity factor. Each token is routed to the expert with the highest router probability, but each expert has a fixed batch size of $(\text{total tokens} / \text{number experts}) * \text{capacity factor}$. If the tokens are unevenly dispatched then certain experts will overflow, indicated by dotted red lines, resulting in these tokens not being processed by this layer. A larger capacity factor alleviates this overflow issue, but also increases computation and communication costs [61].

PyTorch implementation

We implement a fully-functional router in Listing 6.9. This router shares the same logic as the implementation included in Listing 6.8, that is, it uses a `Linear` layer and a `softmax` function. However, we will go beyond this implementation by creating fixed-size input tensors for each of the experts. The implementation can be summarized in the following steps.

- Lines 25-29 compute the output of the (noisy) linear router.
- Lines 31-34 calculate the top-K experts and their associated probabilities.
- Lines 36-38 compute the expert capacity.
- Lines 40-60 use PyTorch indexing and tensor manipulations to construct the batch of expert inputs.
- Lines 63-64 create the final batch of expert inputs.

```

1 class Router(nn.Module):
2     def __init__(self, d, n_exp=8, top_k=2, use_noisy_top_k=True, capacity_factor=1.25):
3         """
4         Arguments:
5             d:                embedding vector dimension
6             n_exp:            the number of experts to create in the expert layer
7             top_k:            the number of active experts for each token
8             use_noisy_top_k:  whether to add noise when computing expert output
9             capacity_factor:  used to compute expert capacity
10        """
11        super().__init__()
12        self.d = d
13        self.n_exp = n_exp
14        self.top_k = top_k
15        assert self.top_k >= 1 and self.top_k <= n_exp
16        self.use_noisy_top_k = use_noisy_top_k
17        self.capacity_factor = capacity_factor
18        self.w_g = nn.Linear(d, n_exp, bias=False)
19        self.w_noise = nn.Linear(d, n_exp, bias=False) if self.use_noisy_top_k else None
20
21    def forward(self, x):
22        B, T, _ = x.size()
23        num_tokens = B * T # Total number of tokens in the batch
24        # Equation 4 in https://arxiv.org/abs/1701.06538
25        logits = self.w_g(x) # [B, T, d] -> [B, T, n_exp]
26        if self.use_noisy_top_k: # Add noise to the router logits
27            noise = F.softplus(self.w_noise(x))
28            noise *= torch.randn_like(noise)
29            logits += noise
30        # Top-K experts selection, compute probabilities over active experts
31        top_k_logits, top_k_indices = logits.topk(self.top_k, dim=-1) # [B, T, K]
32        router_probs = torch.full_like(logits, float('-inf')) # [B, T, n_exp]
33        router_probs.scatter_(-1, top_k_indices, top_k_logits)
34        router_probs = F.softmax(router_probs, dim=-1)
35        # Compute the expert capacity
36        exp_capacity = math.floor(self.top_k*self.capacity_factor*num_tokens/self.n_exp)
37        exp_capacity += exp_capacity % 2 # Make sure expert capacity is an even integer
38        exp_capacity = int(exp_capacity)
39        # Make a multi-hot mask of chosen experts: 0=expert not chosen, 1=expert chosen.
40        exp_mask = F.one_hot(top_k_indices, num_classes=self.n_exp) # [B, T, K, n_exp]
41        exp_mask = exp_mask.view(num_tokens, self.top_k, self.n_exp) # [B*T, K, n_exp]
42        exp_mask = exp_mask.permute(1, 0, 2) # [K, B * T, n_exp]
43        # Compute index for each token in expert batch.
44        # 'cumsum' counts top-1 first, top-2 second, etc.,
45        # to prioritize top experts when dropping tokens.
46        exp_rank = exp_mask.reshape(self.top_k*num_tokens, self.n_exp) # [K*B*T, n_exp]
47        exp_rank = torch.cumsum(exp_rank, dim=0) - 1 # [K*B*T, n_exp]
48        exp_rank = exp_rank.reshape(self.top_k, num_tokens, self.n_exp) # [K, B*T, n_exp]
49        # Mask entries beyond expert capacity and compute used capacity
50        exp_mask *= torch.lt(exp_rank, exp_capacity) # [K, B*T, n_exp]
51        # Matrix storing token position in the batch of the corresponding expert
52        exp_rank = torch.sum(exp_mask * exp_rank, dim=-1) # [K, B * T]
53        # Mask probabilities to only include selected experts
54        router_probs = router_probs.view(num_tokens, self.n_exp)[None, :] # [1, B*T, n_exp]
55        exp_weights = exp_mask * router_probs # [K, B*T, n_exp]
56        # Position of each token within the capacity buffer of the selected expert
57        exp_rank_sc = F.one_hot(exp_rank, num_classes=exp_capacity) # [K,B*T,exp_capacity]
58        # Weight of selected expert for each token at position of capacity for that expert
59        exp_weights = torch.sum(exp_weights.unsqueeze(3)*exp_rank_sc.unsqueeze(2), dim=0)
60        exp_mask = exp_weights.bool() # binary mask of selected experts for each token
61        # Reshape tokens into batches for each expert, return both weights and batches
62        # [n_exp, exp_capacity, B * T] * [B * T, d] -> [n_exp, exp_capacity, n_embd]
63        x = x.view(num_tokens, self.d)
64        expert_batches = exp_mask.permute(1, 2, 0).type_as(x) @ x
65        return exp_weights, expert_batches

```

Listing 6.9: Implementation of a fully-functional router [209].

6.2.4 Load Balancing Loss and Router z-Loss

The gating network, or router, tends to converge to a solution where it always produces large weights for the same few experts. This imbalance is self-reinforcing, since the favored experts are trained more rapidly and thus are selected even more by the gating network [177]. The routing strategies we discussed do not explicitly encourage a balanced selection of experts in each layer. As a result, the model will converge to a state of repeatedly selecting the same few experts for every token instead of utilizing all experts. This phenomenon is commonly referred to as **routing collapse**.

Load balancing loss

To encourage a balanced selection of experts during training, we can simply add an additional term to the training loss that rewards the model for uniformly selecting all experts. More specifically, we define the auxiliary loss term defined in Equation 6.7, which utilizes (i) the load balancing, achieved when the $\mathcal{D}_i$ values are uniformly distributed, where $\mathcal{D}_i$ is defined in Equation 6.8 as the number of tokens assigned to each expert i ; and (ii) the expert importance, defined in Equation 6.9 as the probability of assigning tokens to each expert i . Such an approach is proposed in [173], where authors proposed a loss that considers two quantities: (i) the fraction of tokens dispatched to each expert and (ii) the fraction of router probability allocated to each expert.

If we stack all $\mathcal{D}_i$ values in an N -dimensional vector and all $\mathcal{P}_i$ values in another vector, we calculate the loss term by taking the dot product of these two vectors. This loss is minimized when expert probability and load balancing are uniformly distributed.

$$\mathcal{L}_{\text{load-balancing}} = N \sum_{i=1}^N \mathcal{D}_i \mathcal{P}_i \quad (6.7)$$

where

$$\mathcal{D}_i = \frac{1}{T} \sum_{x \in \mathcal{B}} \mathbb{1}\{\arg \max \mathcal{G}(\mathbf{x}; \Theta) = i\} \quad (6.8)$$

$$\mathcal{P}_i = \frac{1}{T} \sum_{x \in \mathcal{B}} \mathcal{G}(\mathbf{x}; \Theta)_i \quad (6.9)$$

Here, $\mathcal{D}_i$ represents the fraction of tokens assigned to expert i and $\mathcal{P}_i$ represents the average gating probability $\mathcal{G}$ for that expert. Minimizing this loss encourages uniform expert utilization.

An implementation of the load balancing loss in PyTorch is included in Listing 6.10. This implementation follows these steps.

- Lines 9-17 define all constants and input tensors necessary to load balancing loss computation.
- Lines 19-24 compute the fraction of tokens assigned to each expert.
- Lines 26-27 compute the fraction of probability $\mathcal{G}$ assigned to each expert.
- Lines 29-31 calculates the scaled dot product between the fraction of tokens and the fraction of probability for each expert.

```

1 """
2 Computes switch transformer auxiliary load balancing loss
3 (https://arxiv.org/abs/2101.03961), according to equations 7, 8, 9.
4 """
5 import torch
6 import torch.nn.functional as F
7
8 # Constants
9 B = 16      # minibatch size
10 T = 256     # sequence length
11 n_exp = 8   # number of experts
12 K = 2       # number of active expert
13
14 # Define the tensors needed to compute the load balancing loss term
15 indices      = torch.randint(1, n_exp + 1, (B, T, K))    # top-K indices ([B, T, K])
16 expert_probs = F.softmax(torch.rand(B, T, n_exp), dim=2) # expert probs ([B, T, n_exp])
17
18 # Equation 8: compute the fraction of tokens assigned to each expert.
19 # The total number of tokens is defined as total tokens in minibatch * K
20 with torch.no_grad():
21     one_hot_indices = F.one_hot(indices, num_classes=n_exp) # [B, T, K, n_exp]
22     one_hot_indices = torch.sum(one_hot_indices.float(), dim=2) # [B, T, n_exp]
23     tokens_per_expert = torch.mean(one_hot_indices.float(), dim=(0, 1))
24
25 # Equation 9: compute the fraction of router probability assigned to each expert
26 prob_per_expert = torch.mean(expert_probs.float(), dim=(0, 1))
27
28 # Equation 7: take the scaled dot product between probability fraction and
29 # token fraction vectors, and multiply the result by the number of experts.
30 load_balance_loss = n_exp * torch.sum(prob_per_expert * tokens_per_expert)

```

Listing 6.10: Compute the load balancing loss term [209].

Router z-loss

To complement the load balancing loss, authors of [175] proposed another auxiliary loss term, called the router z-loss. The router z-loss constrains the size of the router logits, not probabilities, before the softmax is applied. The router z-loss is defined in Equation 6.10.

$$\mathcal{L}_z = \frac{1}{T} \sum_{x \in \mathcal{B}} \left(\log \sum_{i=1}^N e^{\text{topK}(x \cdot W_g)_i} \right)^2 \quad (6.10)$$

where T is the number of tokens in the minibatch $\mathcal{B}$, N is the number of experts, $\text{topK}(x \cdot W_g)_i$ are the router logits before the softmax, and the term inside the summation captures the magnitude of the router logits for token x .

We do not want these logits to be too large due to the fact that the router calls the exponential function, which is used to compute the softmax. However, these logits can become very large during training, which can lead to round-off errors that destabilize the training process, even when using float32 precision. The router z-loss encourages the MoE to keep these logits small and, as a consequence, avoid the round-off errors. The router calculates the probability distribution over the experts in float32 precision. However, at large scales, this is still insufficient to allow for a reliable training [175].

An implementation of the router z-loss is provided in Listing 6.11, which considers three main steps.

- Lines 5-11 define the input tensor needed to compute the router z-loss, which contains the router logits.
- Line 18 takes a squared logsumexp over the router logits. This is a numerically stable shorthand for applying the exponential, the sum, and the logarithm operations in sequence.
- Line 21 sums the result of the above operation over all tokens and divide by the total number of tokens, to obtain an average value.

```

1 """
2 Computes ST-MoE router z loss [https://arxiv.org/abs/2202.08906]
3 according to equation 10.
4 """
5 # Constants
6 B = 16      # minibatch size
7 T = 256     # sequence length
8 n_exp = 8   # number of experts
9
10 # Define the input tensor for router z-loss
11 router_logits = torch.rand(B, T, n_exp) # [B, T, n_exp]
12
13 # Exponentiate logits, sum logits of each expert, take log, and square.
14 # Using 'torch.logsumexp' is equivalent to the following:
15 # z_loss = torch.exp(router_logits)
16 # z_loss = torch.sum(z_loss, dim=-1)
17 # z_loss = torch.log(z_loss) ** 2.0
18 router_z_loss = torch.logsumexp(router_logits, dim=-1) ** 2.0 # [B, T]
19
20 # Sum over all tokens and divide by the number of tokens
21 router_z_loss = torch.mean(router_z_loss)

```

Listing 6.11: Compute the the ST-MoE router z-loss [209].

Combining the auxiliary loss terms

To combine the auxiliary loss terms, we can just add them to the common language modeling loss during training. Each auxiliary loss term will have a scaling factor by which it is multiplied, then we sum all of the scaled losses together (Equation 6.11). The recommended scaling factors for load balancing loss and router z-loss are 0.001 and 0.01, respectively.

$$\mathcal{L} = \mathcal{L}_{LM}(x) + w_{\text{load-balancing}} \cdot \mathcal{L}_{\text{load-balancing}} + w_{\text{routerZ}} \cdot \mathcal{L}_z \quad (6.11)$$

where $\mathcal{L}_{LM}(x)$ is the common language modeling loss, the second term is the weighted load balancing loss, and the third term is the weighted router z-loss.

Although these auxiliary loss terms work quite well, recent research [51] has shown that, depending upon how the scaling factors are set, such auxiliary loss terms might sacrifice model performance for training stability in some cases. As such, the optimal process and strategies for training MoEs is an active research topic.

For example, the DeepSeek-v3 [51] base model used to create the DeepSeek-R1 reasoning model, uses an auxiliary loss-free load balancing strategy, which simply adds a dynamic bias to the router output when selecting top-k experts (Equation 6.12). This bias is increased for experts that are rarely selected and decreased for experts that are selected too often, thus increasing the chance that under-utilized experts will be selected. This dynamic bias is found to improve load balancing without sacrificing model performance. However, load balancing losses are still used in [51], but with a smaller scaling factor.

$$g'_{i,t} = \begin{cases} s_{i,t} & , \text{ if } s_{i,t} + b_i \in \text{TopK}(\{s_{j,t} + b_j \mid 1 \leq j \leq N_r\}, k_r) \\ 0 & , \text{ otherwise} \end{cases} \quad (6.12)$$

where $g_{i,t} = \frac{g'_{i,t}}{\sum_{j=1}^{N_r} g'_{j,t}}$ is the gating value for the i -th expert, b_i is the bias term for the i -th expert, $s_{i,t}$ is the affinity between token t and expert i , N_r is the numbers of routed experts, and k_r is the number of active routed experts.

We keep monitoring the expert load on the whole minibatch of each training step. At the end of each step, we decrease the bias term by γ if the corresponding expert is overloaded, and increase it by γ if the corresponding expert is underloaded, where γ is the bias update speed hyperparameter [51].

6.2.5 Decoder-Only MoE Implementation

After introducing and implementing all the main components of a mixture-of-experts, we will combine them to build a MoE-based decoder-only transformer. The MoE blocks within this model will contain (i) a masked multi-head self-attention layer and (ii) an expert layer or a normal feed-forward network, where an expert layer arises every P blocks on the model architecture (Figure 6.23).

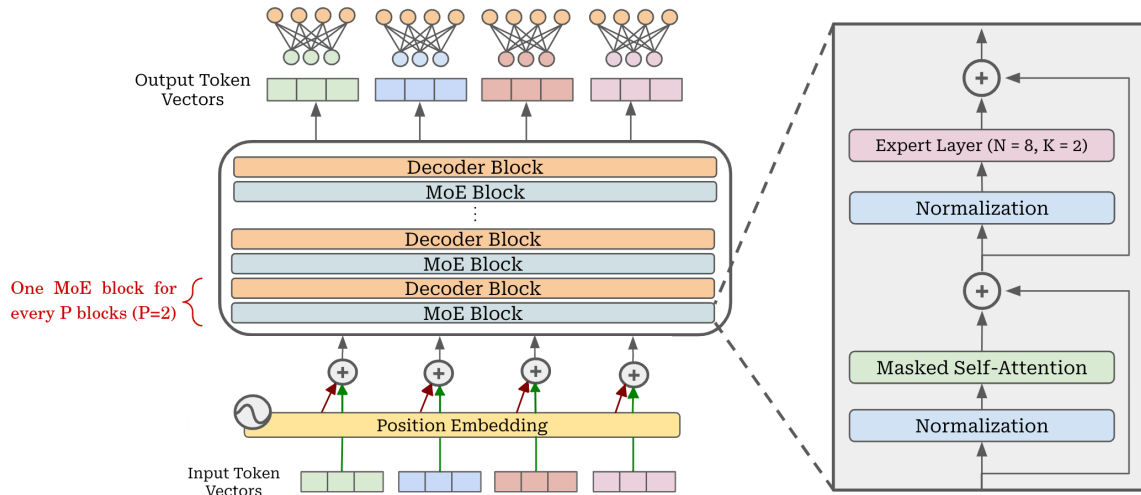


Figure 6.23: MoE-based decoder-only transformer architecture [209].

This block structure is similar to that of a dense decoder-only transformer, but we replace the feed-forward network with an expert layer, forming an MoE block. A MoE block is interleaved with dense blocks, in a proportion of one MoE Block for $P - 1$ dense blocks. First, we cover a few remaining details regarding how the final output of the expert layer is obtained. Then, we will present a full implementation of the MoE-based decoder-only transformer.

Computing expert layer output

Once we have used the router to determine the set of active experts for a given set of token, we can compute the final output for this expert layer as follows: (1) send the set of tokens to the assigned experts, (2) compute the output of the active experts for these tokens, (3) obtain the weighted average of the experts' outputs for each token, where the weights are simply the probabilities assigned to each active expert by the router.

This process is illustrated in Figure 6.24 for a single token. Recent research on MoEs has also introduced the idea of shared experts, which are always active for all tokens. When we include shared experts in the MoE design, the routing logic is slightly modified, but the same core ideas we discussed still apply.

Listing 6.12 presents the PyTorch implementation of an expert layer. On line 34, we get a minibatch of data for each expert and the associated expert probabilities for each token, using the router. We then pass these minibatches through the expert feed-forward networks (line 38), to get the output of each expert. Finally, we multiply each expert output by the associated probability in lines 42-44, producing the final output of the expert layer.

Now, we can modify the decoder-only transformer block to optionally use an expert layer in place of the usual feed-forward network. This is accomplished in Listing 6.13, where we replaced the MLP module with the MoELayer, building an MoEBlock.

The final implementation of the MoE model exactly matches the decoder-only transformer implementation presented in Listing 6.5. The only change is that we replace every P -th Block with an MoEBlock. We do not present the implementation here, since it is very identical to Listing 6.5.

```

1 # Based on ColossalAI OpenMoE
2 class MOELayer(nn.Module):
3     def __init__(self, d, n_exp=8, top_k=2, use_noisy_top_k=True, capacity_factor=1.25,
4         bias=False, dropout=0.2):
5         """
6         Arguments:
7             d:                embedding vector dimension
8             n_exp:            the number of experts to create in the expert layer
9             top_k:            the number of active experts for each token
10            use_noisy_top_k:   whether to add noise when computing expert output or not
11            capacity_factor:   used to compute expert capacity
12            bias:              whether to use bias in linear layers or not
13            dropout:           probability of dropout
14        """
15        super().__init__()
16        self.router = Router( # Noisy top k router
17            d = d,
18            n_exp = n_exp,
19            top_k = top_k,
20            use_noisy_top_k = use_noisy_top_k,
21            capacity_factor = capacity_factor,
22        )
23        self.experts = MLPExperts( # All MLP experts of the MoE layer
24            d = d,
25            n_exp = n_exp,
26            bias = bias,
27            dropout = dropout,
28        )
29
30    def forward(self, x: torch.Tensor):
31        B, T, d = x.size() # Input shape
32        num_tokens = (B * T)
33
34        # Pass each token through the router
35        exp_weight, exp_mask, exp_batches = self.router(x)
36
37        # Compute the experts' outputs
38        exp_out = self.experts(exp_batches) # [n_exp, exp_capacity, d]
39
40        # Aggregate experts' outputs using the weights produced by the router.
41        # Equation 2, page 4, ST-MoE paper [https://arxiv.org/abs/2202.08906].
42        exp_weight = exp_weight.view(num_tokens, -1) # [B * T, n_exp * exp_capacity]
43        exp_out = exp_out.view(-1, d) # [n_exp * exp_capacity, d]
44        output = exp_weight @ exp_out # [B * T, d]
45
46        # Resize the output
47        return output.view(B, T, d)

```

Listing 6.12: Implementation of an expert layer [209].

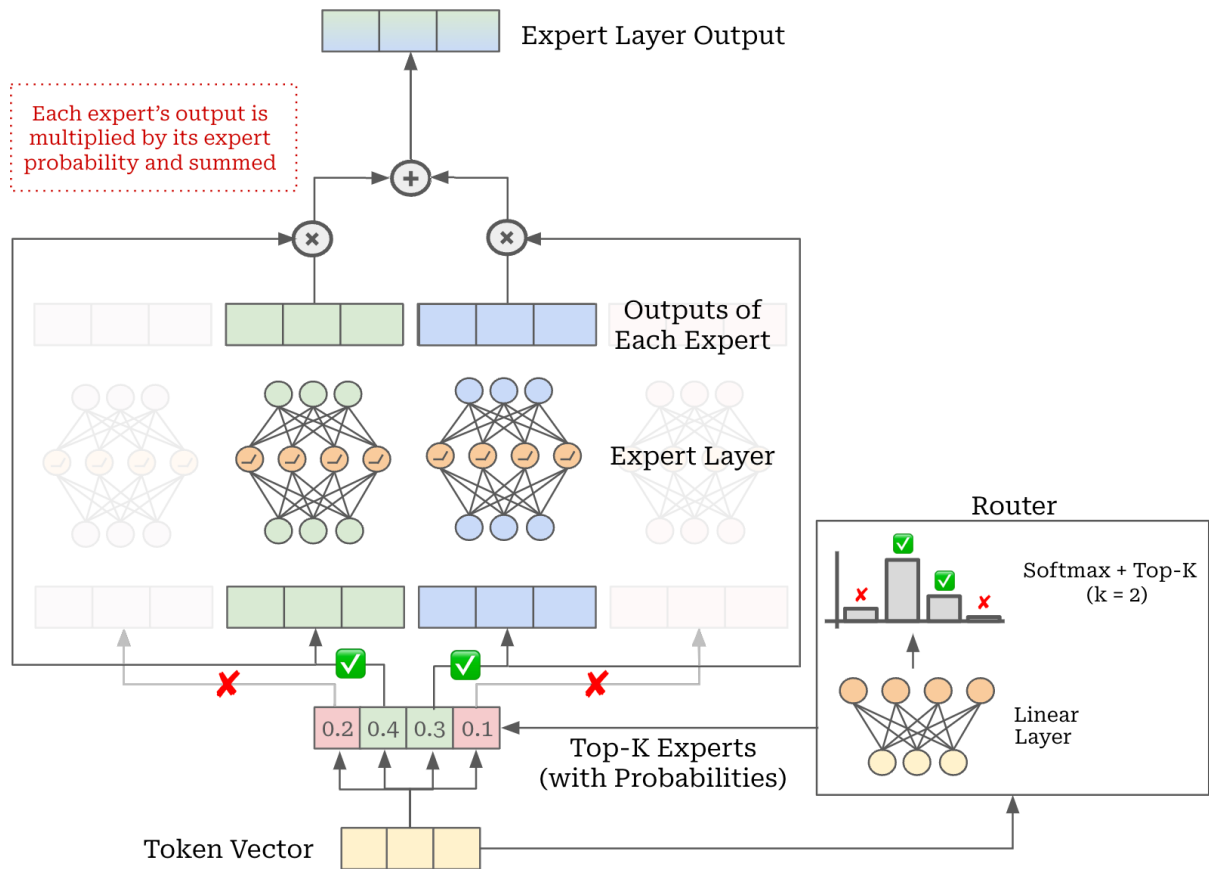


Figure 6.24: Illustration of computing the output of an expert layer for one token [209].

```

1 class MoEBlock(nn.Module):
2     def __init__(self, d, H, T, n_exp, top_k, use_noisy_top_k = True,
3         capacity_factor = 1.25, bias = False, dropout = 0.2):
4         """
5         Arguments:
6             d:                embedding vector dimension
7             H:                number of attention heads
8             T:                maximum length of input sequences (in tokens)
9             n_exp:            the number of experts to create in the expert layer
10            top_k:             the number of active experts for each token
11            use_noisy_top_k:   whether to add noise when computing expert output or not
12            capacity_factor:   used to compute expert capacity
13            bias:              whether to use bias in linear layers or not
14            dropout:           probability of dropout
15        """
16        super().__init__()
17        self.ln_1 = nn.LayerNorm(d)
18        self.attn = CausalSelfAttention(d, H, T, bias, dropout)
19        self.ln_2 = nn.LayerNorm(d)
20        self.mlp = MOELayer(
21            d,
22            n_exp,
23            top_k,
24            use_noisy_top_k,
25            capacity_factor,
26            bias,
27            dropout,
28        )
29
30    def forward(self, x):
31        x = x + self.attn(self.ln_1(x))
32        x = x + self.mlp(self.ln_2(x))
33        return x

```

Listing 6.13: Implementation of an MoE block [209].

6.3 Pretraining the MoE Model

In this section, we discuss how to pretrain the MoE model that we implemented. The nanoMoE implementation reuses the code that we have seen so far in this chapter. The key components of this implementation are the following.

- Model implementation is on the `model.py` file, containing the MoE version of the decoder-only transformer model.
- Training code is included in `train.py`, which included all the training code.
- Dataset, we will pretrain our MoE model on a 25B token subset of the OpenWebText dataset.
- The configuration used to pretrain the MoE model, which we will detail in the next section, is included in `train_moe_v1.yaml` to `train_moe_v5.yaml`.

6.3.1 Best Practices for Training MoEs

Despite several notable successes of MoE, widespread adoption has been hindered by complexity, communication costs and training instability [61]. Although MoEs were proposed a long time ago, their popularity has increased drastically for LLM research only recently. For years, the main impediment to the adoption of MoEs was their difficulty of use. Relative to dense models, MoEs are more complex and generally prone to training instability. An MoE-based transformer only makes slight modifications to the decoder-only transformer architecture. Thus, let us understand why training an MoE is more complicated.

There are two main issues that occur when training an MoE: (i) routing collapse, where the model converges a state that always utilizes the same few experts and (ii) numerical instability, where the MoE may experience round-off errors, especially in the router that employs exponential operations to implement the `softmax`.

These issues lead to training instability, meaning that the loss may simply diverge during the training process. Figure 6.25 presents an example of the divergence that occurs during MoE training. When this happens, we need to stop the training process and restart from a saved checkpoint. We want a stable training process that avoids these instabilities, so, let us cover the best practices for improving MoE training stability.

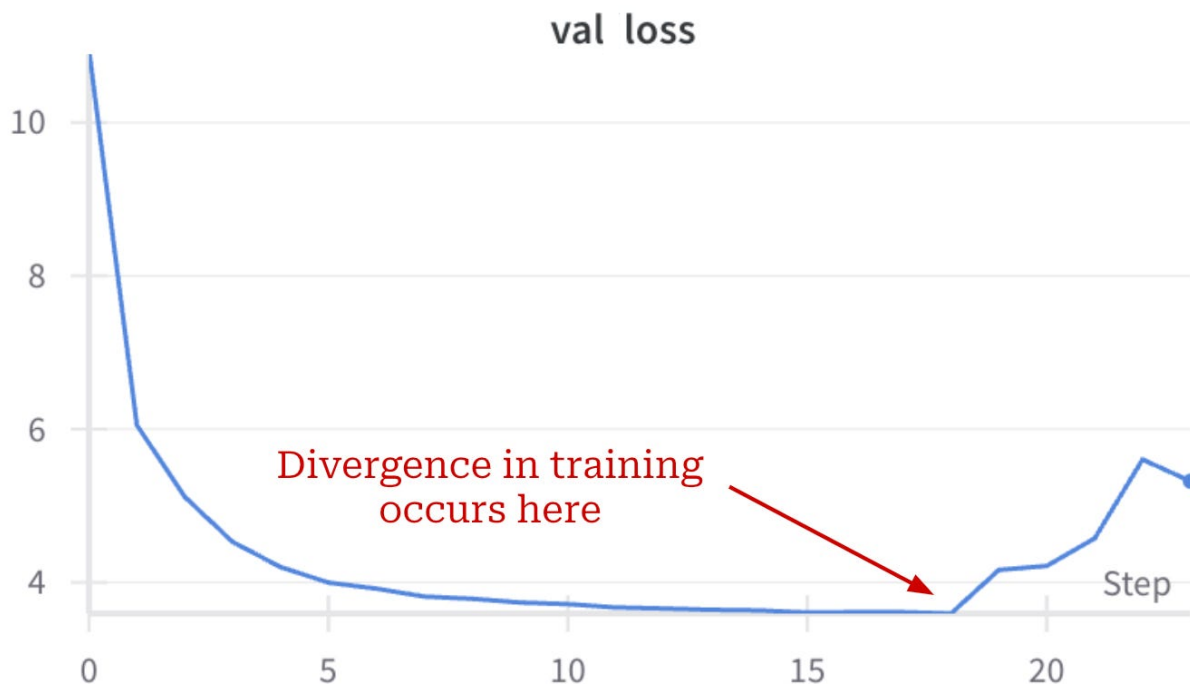


Figure 6.25: An example of divergence during MoE pretraining [209].

Auxiliary loss terms

As discussed previously, we do not have to choose between auxiliary loss terms when training an MoE. Instead, we can just combine multiple auxiliary loss terms into a single loss function. In our case, we use both the load balancing loss and the router z-loss during training. Using the correct auxiliary loss terms improves training stability by enabling uniform usage of experts and avoiding routing collapse during training.

Training precision

When training an LLM, it usually makes sense to use mixed precision training, which converts some components of the model to run in a lower float16 or Bfloat16 precision format instead of float32 precision. This functionality is supported automatically in PyTorch via the automatic mixed precision (AMP) package and can significantly reduce training costs without deteriorating model performance. In other words, this is a pretraining speedup that we can easily enable with minimal code changes.

Compared with the Bfloat16 baseline, the relative loss error of our FP8-training model remains consistently below 0.25%, an error that is within the acceptable range of training randomness [51].

Mixed precision has been used for some time, but researchers have more recently explored methods for reducing LLM training precision even further, lower than 16-bits. For example, DeepSeek-v3 [51] is trained using 8-bit precision. However, maintaining the same level of model quality becomes more difficult as training precision is reduced. Implementing large-scale LLM training with FP8 precision requires novel and complex quantization techniques. Otherwise, training an LLM at such low precision may negatively impact the model's performance.

As we mentioned before, the routing mechanism within an MoE is prone to numerical instability. Computing the router's output in lower precision makes this problem even worse. This issue is explicitly mentioned in [61], where authors found that low precision training leads to large round-off errors in the router. To solve this issue, we must run the router in float32 precision even when training with AMP, which can be achieved by simply disabling AMP in the MoE's routing mechanism. The following fragment of code illustrates how to run the router computation with AMP disabled.

```
with torch.amp.autocast(device_type='cuda', enabled=False):  
    # AMP is disabled for code in this block  
    <run router code here>
```

Weight initialization

A common technique used to improve training stability of large neural networks is applying the correct weight initialization strategy, such as Glorot or He initialization. These techniques, along with strategies like batch normalization, unlocked the ability to train very deep neural networks, which was quite difficult before. For LLMs, we usually adopt the same weight initialization strategies. However, the authors of [61] recommend adopting a slightly modified weight initialization scheme that is specifically designed for MoEs.

This weight initialization strategy samples weights from a truncated normal distribution with zero mean ($\mu = 0$) and a standard deviation given by $\sigma = \sqrt{s/n}$, where s is a scale hyperparameter and n is the size of the input to the layer being initialized (fan-in). Sampled values that are outside the $[a, b]$ interval are redrawn until they are within these bounds. Authors of [61] also recommend using a reduced scale hyperparameter $s = 0.1$ to improve quality and reduce the likelihood of destabilized training. A PyTorch implementation of this modified weight initialization strategy is included in Listing 6.14.

```
1 # Linear layers have flipped dimensions in PyTorch: [out_dim, in_dim]  
2 w_fan_in = module.weight.shape[-1]  
3 w_std    = (scale / w_fan_in) ** 0.5  
4 torch.nn.init.trunc_normal_(  
5     module.weight,  
6     mean = 0.0,  
7     std  = w_std,  
8     a    = -2*w_std, # lower bound of truncation  
9     b    = 2*w_std,  # upper bound of truncation  
10 )
```

Listing 6.14: Switch transformer weight initialization strategy [209].

MoE finetuning

We will not do model fine-tuning here, but we should be aware that MoEs can be more difficult to fine-tune compared to dense models. In particular, MoEs are prone to overfitting due to the fact that they have a huge number of weights. These large models are great for pretraining over massive datasets, but they can overfit when fine-tuned over a small amount of data. To prevent overfitting when fine-tuning an MoE, we can apply a higher dropout probability.

6.3.2 Pretraining Experiments

In this section we will cover MoE pre-training. In the experiments presented here, we used two GPU setups: (i) a Titan Xp and an RTX 2080 Ti, and (ii) two RTX 3060. These are commodity GPUs with only 12 GB and 11 GB. The pretraining settings have been scaled down accordingly, allowing everything to fit in GPU memory and run completely in less than a week.

General pretraining settings

The configurations used during pretraining are described next (Table 6.1).

- Model architecture: six transformer blocks, six attention heads per self-attention layer, the embedding vector dimension is $d = 368$, the number of experts is $N = 8$, the number of active experts is $k = 2$, and the stride used to replace a dense transformer block for an MoE block is $P = 2$.
- Expert capacity: the capacity factor was set to 1.25 for training and 2.0 for evaluation.
- Auxiliary loss terms: we used the load balancing auxiliary loss (except in the base configuration), with a scaling factor of 0.01, and the router z-loss (in the last three configurations), with a scaling factor of 0.001.
- Precision: we used automatic mixed precision (Bfloat16) for training, but the router logic was run in Bfloat16 (in the first three configurations) or float32 precision (in the last two configurations).
- Learning rate: we adopted a standard LLM learning rate strategy, with a linear warmup from 6×10^{-5} to 6×10^{-4} during the initial 2k training iterations, followed by a cosine decay to 6×10^{-5} until 50k iterations, and then maintain the minimum learning rate of 6×10^{-5} during the remaining iterations.
- Weight initialization: we used standard normally distributed values to initialize the weights (in the first four configurations) or the switch transformer-style initialization presented in Listing 6.14 (in the last configuration) to improve the MoE training stability.
- The number of iterations was set to 100,000.

Configuration	Train loss	Val loss	Train time per step (ms)	MFU (%)	GPUs
base	3.43	3.44	397	1.28	Titan Xp+RTX2080 Ti
aux loss	3.46	3.47	394	1.29	Titan Xp+RTX2080 Ti
aux loss + router z-loss	3.45	3.47	156	3.24	2 RTX 3060
aux loss + router z-loss + FP32	3.45	3.47	156	3.25	2 RTX 3060
aux loss + router z-loss + FP32 + switch transformer initialization	3.45	3.47	398	1.28	Titan Xp+RTX2080 Ti

Table 6.1: Results obtained during MoE pre-training: training loss, validation loss, training time per minibatch and model FLOPs utilization (MFU).

Pretraining dataset

We used the OpenWebText dataset for pretraining the MoE. The pretraining process is scaled down to 25 billion tokens. The reduced-size dataset allows for pretraining to complete in (i) around 5 days on the Titan Xp plus RTX 2080 Ti GPUs or (ii) around 2 days on two RTX 3060 GPUs. However, we can easily scale

this up to a full pretraining run by obtaining a better GPU setup, for example, eight A100 GPUs, and setting `max_iters = 600,000` instead of 100,000.

Results

To evaluate the impact of different techniques on MoE's training stability, we performed five different experiments. First, we pretrained a baseline MoE model without any auxiliary losses or best practices. Then, we enabled several improvements, one-by-one, to observe their impact on pretraining stability: auxiliary load balancing loss, router z-loss, full precision (FP32) in the router, and the improved weight initialization scheme.

To run pretraining, we just have to use the following command, which assumes that we want to run pretraining on a single node with two GPUs and the configuration file is `train_moe_v1.yaml`.

```
torchrun --standalone --nproc_per_node=2 train.py <config/train_moe_v1.yaml>
```

The main results of these five experiments are shown in Table 6.1. The first conclusion is that all five configurations run with success without instability. The second conclusion is that all configurations achieved similar results in terms of training and validation loss. Contrary to expectations and by a narrow margin, the base configuration achieved the smallest loss in training (3.43) and in validation (3.44). The training time per epoch was around 397 ms on the Titan Xp plus RTX 2080 Ti GPUs and 156 ms on two RTX 3060 GPUs. This means that the 3060 GPUs reduced the training time to 39.6% compared to the older models. Conversely, model FLOPs utilization increased from 1.28% to 3.24% when using the 3060 GPUs.

The training loss observed during MoE pre-training is plotted in Figure 6.26, where we can see that there are very small differences among the five configurations. The validation loss obtained during MoE pre-training is plotted in Figure 6.27 and, again, the five configurations exhibit very similar loss trends. Finally, the learning rate variation is depicted in Figure 6.28. The learning rate follows the described scheme: a linear warmup from 6×10^{-5} to 6×10^{-4} during the first 2k iterations, followed by a cosine decay to 6×10^{-5} until the iteration 50k, and then maintain the learning rate at 6×10^{-5} during the remaining 50k iterations.

We were able to do inference with the trained models using the following command.

```
python sample.py --start="who was Pablo Picasso?" --num_samples=5 --max_new_tokens=100
```

However, despite the generated sentences being mostly coherent, the content is erratic and little more than random, as illustrated by the following answers to the prompt "who was Pablo Picasso?".

Answer 1: The Beatles, they're telling us, so we all want to know if we're better than him or not. Pablo's a genius, but he's a pretty decent human being. And he's a really good artist, too. I know he's a genius, man, but he's an genius. And I don't want to be the last man that got a call up.

Answer 2: Strap in your faces and say, How convenient. You know more about the work than I ever did, and what you know about somebody who was one of the most influential artists of the entire history of film. And Pablo Picasso became a true friend of Anthony Rapp, the man who would become the president of the National Film Registry, a firm dedicated to independent film making. Pasquan Santiago, the man who would become the president of the National ...

Answer 3: By George's 1516 period, Pablo was the first artist to be made of the color-blind. His work was infused with the kind of mania that made painting the world a more rewarding experience. His 1418 period of time did not come without a thought of his art. His art was inspired by art as a way to look at the world with the purpose of showing the things that matter. The artist's art could not exist without art inspiration. Only by giving a free ...

Answer 4: One time I had the idea of going out on a business trip and the other someone said they were going out on a date. But I wasn't there. Even though Pablo Picasso is I don't know what he was doing - he told me his line about whether or not the artist is ever going to be a reporter. A lot of the people working on the day, they don't know, but they don't know.



Figure 6.26: Training loss during MoE pre-training.

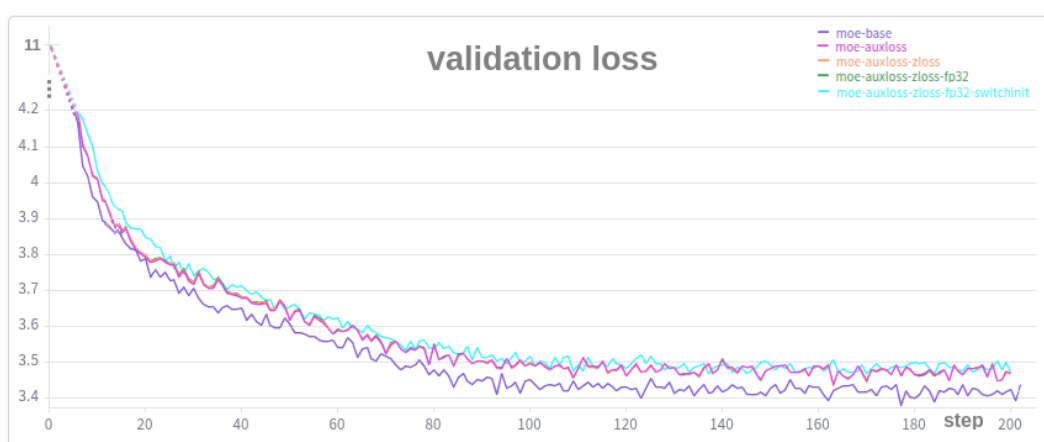


Figure 6.27: Validation loss during MoE pre-training.

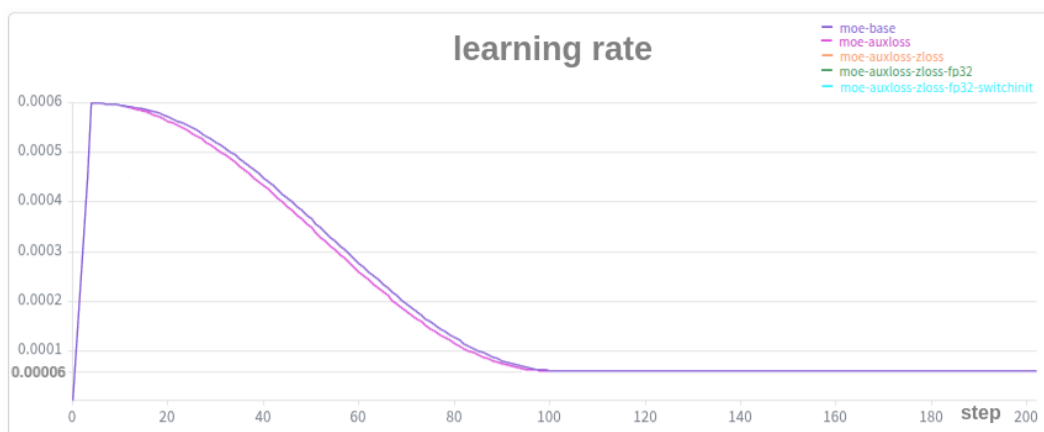


Figure 6.28: Learning rate variation during MoE pre-training.

Answer 5: On set in the 90s, Pablo Picasso was involved in a lot of the biggest-but-a-thumb-hewn movements, and he never really made it out of that era. Kasich, who's been involved on multiple sets in the past, is very much a part of the moment. At the start of the 90s, Pablo was involved in some fantastic shows around the country, including the 90s.

We should note that our model size is 54.7M, while the smallest GPT-2 has a size of 124M and our model was trained for 100k iterations on a reduced version of the OpenWebText dataset. It is only a proof of concept of how to train a MoE model, not a really useful large language model.

6.4 Conclusions

In this chapter, we presented a comprehensive, hands-on exploration of Mixture of Experts (MoE) models, focusing on their implementation and training within transformer-based architectures. We began by revisiting the decoder-only transformer, detailing its core components, including tokenization, embeddings, masked multi-head self-attention, and feed-forward networks, and demonstrating how these elements can be implemented from scratch in PyTorch. This foundation was essential for understanding how MoE models extend standard dense architectures.

Building upon this baseline, we introduced the MoE paradigm, where the feed-forward network in selected transformer layers is replaced with an expert layer composed of multiple independent subnetworks. This modification enables a significant increase in model capacity while maintaining manageable computational cost, thanks to the sparse activation of experts. The implementation showed how expert layers can be efficiently constructed and interleaved with dense transformer blocks, forming hybrid architectures that balance performance and efficiency.

A central aspect of MoE models is the routing mechanism, which determines how tokens are dynamically assigned to experts. We explored practical routing strategies based on top-k softmax selection, including optional noisy routing for improved regularization. The concept of active weights was emphasized as the key to computational efficiency, as only a subset of experts contributes to each forward pass. At the same time, we highlighted the complexity of implementing routing in practice, particularly when managing variable-sized expert inputs and ensuring efficient batching.

To address these challenges, we examined mechanisms such as expert capacity, which enforces fixed batch sizes for each expert and improves hardware utilization. However, this introduces trade-offs, including the need to drop excess tokens when capacity is exceeded. This illustrates a broader theme in MoE systems: the balance between theoretical flexibility and practical implementation constraints.

The chapter also emphasized the importance of training stability, a key difficulty in MoE models. Issues such as routing collapse and numerical instability were addressed through auxiliary loss functions, including load balancing loss and router z-loss. These techniques encourage uniform expert utilization and prevent extreme router outputs, while additional practices, such as running the router in full precision, using appropriate weight initialization, and applying mixed precision training, further enhance stability.

Finally, we integrated all components into a complete MoE-based decoder-only transformer and explored its pretraining process. Through practical experiments, we demonstrated how different stabilization techniques contribute to successful training, showing that naive implementations often fail to converge, while carefully designed systems can train reliably even on limited hardware.

In summary, the chapter highlighted that while MoE models offer substantial scalability advantages, their successful deployment requires careful attention to both architectural design and implementation details. Efficient routing, balanced expert utilization, and stable training procedures are all critical for unlocking the full potential of MoE systems. As such, building MoE models is not merely an extension of standard transformers, but a more complex engineering task that combines algorithmic insight with system-level considerations.

Chapter 7

Other Types of Parallelism

Contents

7.1	Parallelism in mixture of experts	243
7.2	Expert parallelism	246
7.3	Parallelism in switch transformers	248
7.4	Parallelism in Tutel	249
7.5	DeepSpeed-MoE	255
7.6	Pathways	261
7.7	Parallelism in Megatron-Core	263
7.8	Expert parallelism in NVIDIA Megatron Bridge	267
7.9	Elastic expert parallelism in vLLM	270
7.10	Sequence parallelism	275
7.11	Implementation example	286
7.12	Conclusions	295

This chapter explores advanced parallelism strategies for scaling Mixture of Experts (MoE) models, focusing on how modern large-scale systems efficiently distribute computation across multiple devices. It begins by introducing the need for multi-dimensional parallelism, highlighting the limitations of single-device training and motivating the use of distributed approaches that combine multiple parallelization techniques. The chapter first reviews the fundamental forms of parallelism, including data, tensor, pipeline, and sequence parallelism, as well as MoE-specific approaches such as expert parallelism and expert tensor parallelism. Each method is examined in terms of its role in addressing memory constraints, improving computational efficiency, and managing communication overhead, particularly in the context of sparse MoE architectures. Building on these foundations, it is then presented a comprehensive discussion of hybrid parallelism strategies, which combine multiple techniques to achieve scalable and efficient training. It traces the evolution from homogeneous approaches, where the same configuration is applied across all layers, to heterogeneous designs that exploit the differences between dense and sparse components, and finally to dynamic methods that adapt parallelism strategies at runtime. The growing importance of automatic parallelization systems is also highlighted, showing how optimal configurations can be discovered through search and optimization. A central part of the chapter is devoted to expert parallelism, detailing its formulation, implementation, and trade-offs. This includes the mechanics of token routing, All-to-All communication, and load balancing, as well as its integration with other parallel dimensions. It is also analyzed how expert parallelism is applied in real systems, including Switch Transformers and frameworks such as Tutel and DeepSpeed-MoE, where advanced optimizations for communication, scheduling, and inference efficiency are introduced. Further sections examine state-of-the-art large-scale systems, such as Pathways and Megatron-Core, which rethink traditional distributed training paradigms. In particular, concepts like asynchronous dataflow execution and parallel folding are presented as solutions to the inherent mismatch between dense and sparse computations in MoE models. The chapter also covers industry-grade implementations, including NVIDIA Megatron Bridge and vLLM, highlighting practical features such as optimized token dispatching, load balancing, and elastic expert parallelism for dynamic scaling and fault tolerance. Finally, the chapter addresses sequence parallelism techniques for handling long-context workloads, including Megatron sequence parallelism, Ring Attention, and Striped Attention. These methods demonstrate how communication patterns and computation can be reorganized to reduce memory usage and improve scalability for very long sequences. The chapter concludes with a practical implementation example, demonstrating how to build and train a Mixture of Experts model using the DeepSpeed-MoE framework. This example illustrates how expert parallelism and memory optimization techniques, such as ZeRO, can be applied in practice to efficiently train and evaluate MoE models on real tasks.

7.1 Parallelism in Mixture of Experts

The rapid growth in model size enabled by MoE architectures makes single-device or single-node training fundamentally inadequate for modern large language models. Scaling MoE systems requires distributed training techniques that operate across several parallel dimensions. These include data parallelism for distributing batches across devices, tensor parallelism for splitting computations within layers, and expert parallelism for partitioning experts across accelerators [163, 184, 108]. To fully utilize modern multi-node and multi-accelerator infrastructures, hybrid combinations of these strategies have become a cornerstone of state-of-the-art MoE systems [228].

This section provides an overview of multi-dimensional parallelism methods used for large-scale MoE training. We first describe fundamental parallel paradigms and discuss their roles in MoE architectures. Next, we examine the progression of hybrid parallelism approaches: starting from uniform (homogeneous) designs, moving to heterogeneous methods that differentiate between layer types, and finally dynamic techniques that adapt to runtime conditions. This progression highlights a conceptual shift from static parallel configurations toward adaptive and workload-aware optimization.

7.1.1 Fundamental Parallelism

Data parallelism

Data parallelism (DP) is one of the most commonly used strategies in deep learning. It distributes mini-batches across multiple devices while keeping a full model copy on each device [119]. Each device independently computes forward and backward passes and synchronizes gradients using AllReduce operations [174, 164]. This approach improves throughput by processing data in parallel, but it replicates model weights across devices, leading to high memory usage. Consequently, DP becomes less efficient for extremely large models.

Tensor parallelism

Tensor parallelism (TP) divides large operations within a layer into smaller parts that can be computed across multiple devices [184, 108]. Typically, tensors are split along rows or columns, and collective communication, for example through AllReduce operations, is required to combine results during forward and backward passes [146]. While TP helps address memory constraints, it introduces frequent synchronization overhead, making it better suited for intra-node deployment where communication latency is lower.

Pipeline parallelism

Pipeline parallelism (PP) partitions a model into sequential stages, assigning each stage to a different device [89, 143, 60]. Data flows between stages through point-to-point communication, which is generally lighter than collective operations. PP is beneficial in bandwidth-limited environments; however, it can suffer from pipeline bubbles, causing idle time and reduced efficiency, particularly at the beginning and end of the pipeline [158].

Sequence parallelism and context parallelism

Sequence parallelism (SP) and context parallelism (CP) target inefficiencies when processing long input sequences. Sequence parallelism splits inputs along the sequence dimension, reducing activation memory in layers such as LayerNorm and Dropout [108, 130]. Context parallelism, on the other hand, partitions attention key-value states, decreasing the memory footprint of attention computations [130]. CP can also overlap communication with computation, improving overall performance [200].

Expert parallelism

In MoE models, feed-forward layers dominate memory usage due to large expert matrices. Expert parallelism (EP) distributes different experts across devices [62, 242]. Since tokens are dynamically routed to experts, EP requires All-to-All communication to redistribute tokens among devices [140, 163]. Each device only stores a subset of experts, reducing memory overhead. In practice, EP is often combined with DP, with attention layers handled via DP and expert layers via EP.

Expert tensor parallelism

Expert tensor parallelism (ETP) extends EP, by further splitting individual experts, when their size exceeds device capacity [146, 171]. Unlike standard TP, which applies uniformly across all layers, ETP specifically targets expert layers. This is useful when certain experts receive a disproportionate share of tokens [36]. However, it increases communication overhead due to additional synchronization, requiring careful balancing between efficiency and cost.

7.1.2 Hybrid Parallelism

Although each parallel strategy offers specific advantages, combining them naively can lead to inefficiencies. The main difficulty lies in mismatches between parallelization methods and the irregular computation patterns of MoE models. For example, pipeline parallelism often results in excessive idle time due to uneven workloads. Similarly, Fully Sharded Data Parallel (FSDP) increases the cost of weight gathering for experts compared to dense models.

MoE systems present unique challenges: (i) different computation patterns between attention and expert layers, (ii) dynamic workloads due to evolving routing behavior, and (iii) extremely large weight counts combined with

sparse activation [114, 163]. Addressing these challenges has enabled the training of massive models, such as GShard, with 600 billion weights and trillion-scale MoE systems.

The hybrid parallelism for MoE models can be classified into three categories, each addressing different challenges: (i) homogeneous strategies that optimize parallelism approaches by considering both workload and topology characteristics; (ii) heterogeneous approaches that exploit fine-grained intra-layer characteristics; and (iii) dynamic parallelism techniques that adapt to runtime conditions (Figure 7.1).

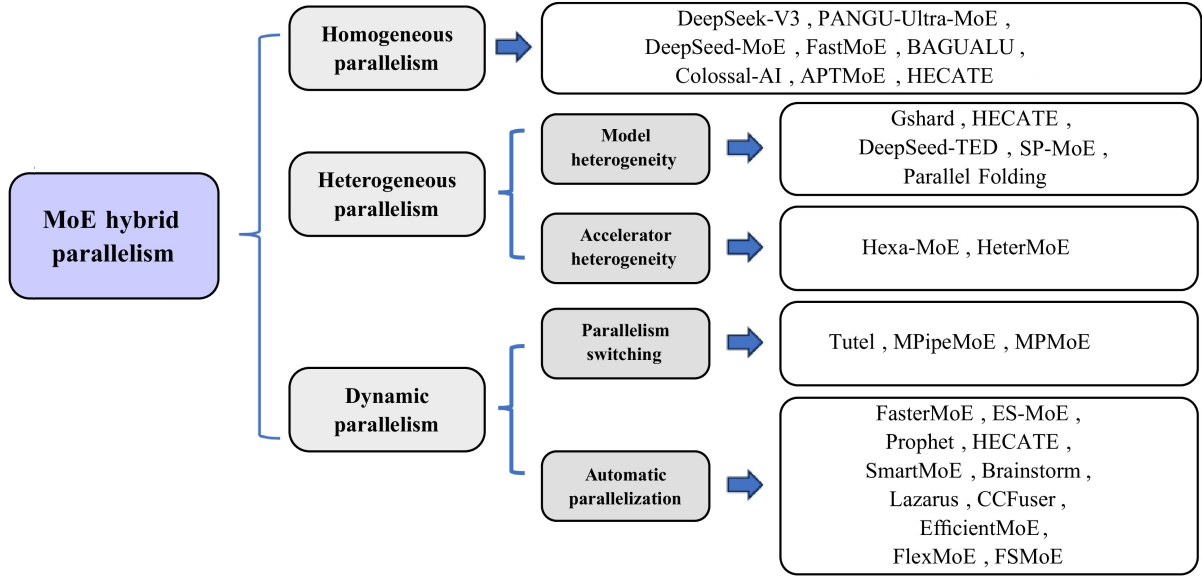


Figure 7.1: MoE hybrid parallelism techniques taxonomy in MoE training [228].

Homogeneous Hybrid Parallelism

Early systems applied the same parallelism configuration across all layers. For example, FastMoE uses a consistent DP+EP scheme throughout the model [76], while DeepSpeed-MoE adds TP to create a three-dimensional setup [163]. Later works introduced topology-aware optimizations, such as FasterMoE’s communication modeling and Colossal-AI’s node-local expert placement [77, 120]. However, these methods fail to exploit differences between attention and expert computations, leading to inefficiencies.

Heterogeneous Hybrid Parallelism

To address this limitation, heterogeneous approaches apply different strategies to different components. GShard uses DP for attention layers and EP for MoE layers [114]. Other systems, like DeepSpeed-TED and EZ-MoE, extend this idea through hierarchical and optimized parallel designs [185, 120]. Megatron-Core introduces parallel folding to decouple parallelism across layer types, improving scalability [41, 127]. Advanced approaches such as FSDP further improve memory efficiency through sparse weight materialization [159].

Heterogeneity is also explored at the hardware level. Systems like HeterMoE assign different computations to devices based on their capabilities [214], while HEXA-MoE introduces specialized operators to improve efficiency on diverse hardware [134]. Additionally, some works adapt parallelism strategies across different training phases to match varying computational demands [155].

Dynamic Hybrid Parallelism

Because MoE workloads change during training, static parallelism can become suboptimal. Dynamic approaches adapt strategies at runtime based on system conditions and workload changes. Tutel enables flexible switching between parallel modes using a unified data layout [140]. MPipeMoE and related systems adjust pipeline depth and resource allocation dynamically to handle fluctuating memory and compute demands [235]. Predictive techniques further anticipate workload changes to prevent bottlenecks [234].

Automatic Parallelization

Due to the complexity of selecting optimal configurations, automated tools have been developed to determine efficient parallel strategies. Systems such as Alpa, Aceso, nnScaler, and Mist formulate parallelization as an optimization problem [239, 129, 125, 245]. In MoE models, expert placement becomes an additional dimension, leading to adaptive approaches like SmartMoE and Prophet [225, 203].

7.2 Expert Parallelism

Expert parallelism (EP) refers to a model-parallelism strategy in which the different expert sub-networks are distributed across devices, often GPUs, rather than being replicated or tensor-sharded in the usual way. As we have seen in Chapter 5, an MoE model comprises a gating network that selects a subset of expert networks, for example, feed-forward layers, to process each input token. By activating only a few experts per input, one gains a weight-efficient model that uses a huge total weight count while keeping the computation cost at a lower value [21].

EP leverages this sparsity during distributed training and inference, where each device holds one or more experts rather than all experts, and tokens are routed to the devices which hold the selected experts. This contrasts with tensor parallelism, where we split weights of a layer across devices, or pipeline parallelism, which splits layers across stages. EP is a type of model parallelism that distributes experts of an MoE across GPUs, but unlike other model-parallel techniques, EP is applied to only the expert layers thus does not impact the parallel mapping of the remaining layers.

In this section, we will lay out the definition and motivation for expert parallelism, how it fits into the taxonomy of parallelism approaches, such as data, tensor, or pipeline parallelism, and the high-level benefits and trade-offs.

7.2.1 Definition and Taxonomy

More formally, consider a MoE layer in a neural network. Suppose there are E experts in this layer, labeled $f_1, f_2, \dots, f_E$. An input token vector x is routed via a gating function $g(x)$ that selects a subset of experts, often the top- k scoring experts, to process x . In this scenario, the MoE layer output for this token is a weighted sum of the experts' outputs, as indicated in Equation 7.1.

$$y = \sum_{i=1}^E w_i(x) f_i(x) \quad (7.1)$$

where $w_i(x)$ are non-zero only for the selected subset of experts.

In EP, each expert f_i is assigned to a particular device d_j , or group of devices. Thus, device d_j might host a subset of experts $\mathcal{E}_j \subseteq \{\mathcal{E}_1, \mathcal{E}_2, \dots, \mathcal{E}_E\}$. Then, when tokens are routed, one must send the token data to the device(s) hosting the selected experts, compute $f_i(x)$, and then collect the outputs. This allows each device to perform full expert forward and backward passes for the experts it hosts.

EP can be thought of as a specialization of model parallelism particular to MoE architectures. Many large MoE systems use a hybrid parallelism combining, such as DP + TP + EP, to scale the model over three dimensions: data, tensor weights, and expert networks.

There are a few reasons for expert parallelism being relevant.

- **weight scale:** MoE models often have a huge weight count, often tens of billions, while they only activate a small fraction of weights per input. EP allows distributing those weights across devices, so no single device must hold all of them. As models scale to larger sizes and fail to fit on a single GPU, we require more advanced forms of parallelism. EP is a form of model parallelism where we place different experts on different GPUs for better performance.

- **FLOP/activation efficiency:** By routing each input token to a small subset of experts, the model needs fewer FLOPs than activating all experts. EP then ensures that the computational workload is distributed across devices and each device executes reasonably large matrix multiplications rather than many tiny multiplications.
- **Memory efficiency:** Because not all experts are active for each input, and each device holds only part of the experts, the peak memory footprint per device is reduced compared to naive replication of all experts.
- **Scalability:** EP enables scaling the number of experts E and the model capacity without linearly increasing the per-token compute budget, assuming that only a small number of experts is active per token. This helps achieve almost constant cost per token with increasing weight count.

EP provides an elegant mechanism to map sparsely activated experts onto distributed hardware while preserving high model capacity. It relies on efficient all-to-all communication and balanced routing to ensure scalability. Modern frameworks optimize these aspects through load-balancing losses, fused collective operations, and hybrid parallel designs.

EP works by partitioning experts across multiple devices and dynamically routing token representations to the appropriate devices that host the selected experts. Each expert is a self-contained network, typically a feed-forward network within a transformer block. During the forward pass, each input token is assigned to one or more experts according to the routing probabilities produced by the gating network, which can be softmax-normalized linear projection of the token embeddings.

Formally, let E denote the total number of experts and D the number of available devices. We define an expert-to-device mapping function $\phi : \{\mathcal{E}_1, \mathcal{E}_2, \dots, \mathcal{E}_E\} \rightarrow \{\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_D\}$, such that expert $\mathcal{E}_i$ resides on the device given by $\phi(i)$. Each token x_j is routed to a subset of experts $S_j = \{\mathcal{E}_{j1}, \dots, \mathcal{E}_{jk}\}$ with the corresponding gating scores $w_{j1}, \dots, w_{jk}$. The forward pass output is computed with Equation 7.2.

$$y_j = \sum_{\mathcal{E}_i \in S_j} w_{ji} f_{\mathcal{E}_i}(x_j) \quad (7.2)$$

where $f_{\mathcal{E}_i}$ is the expert network functionality, typically implemented with a two-layer MLP and a non-linear activation function.

This dynamic routing implies an all-to-all communication pattern between devices: tokens assigned to experts on remote devices must be sent to those devices, processed, and returned. The design challenge in EP is therefore to minimize communication overhead and maintain load balancing so that each device processes a similar number of tokens. The common implementation of all-to-all routing in MOEs follows the next steps, which are used in systems like GShard [114], switch transformers [62], and DeepSpeed-MoE [163].

- **Token grouping:** For each token, the router determines the top-k experts. Tokens are grouped by destination expert.
- **All-to-all dispatch:** Each device sends token embeddings to the devices hosting the relevant experts. This step dominates communication cost.
- **Local expert computation:** Each device executes its local experts on the received tokens, producing expert outputs.
- **AllGather:** Expert outputs are sent back to the originating devices.
- **Combination:** The multiple outputs for each token are combined through a weighted sum, where the weights are the gating scores w_{ji} , to produce the final MoE layer output.

In the frameworks that support MoE modeling, both dispatch and gather operations are implemented using highly optimized collective primitives such as NCCL all-to-all or fused CUDA kernels. The cost is largely proportional to $O(B \cdot H/D)$, where B is the minibatch size, H is the expert layer hidden dimension, D is the number of devices, and assuming uniform routing.

The performance of EP is determined by the ratio of communication time t_{comm} to computation time t_{comp} . Let T denote the total number of tokens per batch, k the number of selected experts per token, E the number of experts, and D the number of devices. A simplified cost model can be defined through Equations 7.3, 7.4, and 7.5.

$$t_{total} \approx t_{comp} + t_{comm} \quad (7.3)$$

$$t_{comm} \propto \frac{B \cdot k \cdot H}{D \cdot BW} \quad (7.4)$$

$$t_{comp} \propto \frac{B \cdot k \cdot H^2}{D \cdot TFLOPs} \quad (7.5)$$

where BW is the interconnect bandwidth and $TFLOPs$ is the compute throughput per device. Efficient systems therefore aim to (i) minimize k , (ii) overlap communication with compute, and (iii) route tokens so that most stay local to a node. State-of-the-art frameworks, such as Megablocks [65] and DeepSpeed-MoE [163], adopt block-sparse layouts and fused all-to-all operations to reach near-linear scaling up to hundreds of GPUs.

Since expert parallelism only distributes MoE layers, production-grade systems often combine it with other parallelism types: (i) data and expert parallelisms, where each data-parallel replica holds a different subset of experts, reducing memory cost per GPU while maintaining large global batch size; (ii) tensor and expert parallelisms, where inside each expert, weight matrices are tensor-parallelized to fit very large hidden dimensions [56]; (iii) pipeline and expert parallelisms, where different layers are placed in pipeline stages while MoE experts are spread within each stage [9].

These hybrid solutions achieve efficient scaling across thousands of accelerators. For instance, Google’s Pathways system [9] demonstrated scaling of MoE models to trillions of weights using combinations of data, tensor, and expert parallelisms.

7.2.2 Advantages of Expert Parallelism

Expert parallelism offers several key benefits.

Efficient resource usage: By storing only a subset of experts per device, memory requirements are significantly reduced, enabling much larger models.

Scalability: Model size can grow by adding more experts and devices, supporting trillion-scale architectures such as switch transformer and GShard.

Faster training: Sparse activation allows increased model capacity without proportional computation growth, leading to reduced training time, where 4 times speedups were reported.

7.2.3 Challenges of Expert Parallelism

Communication overhead: Token routing requires All-to-All communication, where tokens are sent to and from expert devices. This can dominate runtime if not carefully optimized.

Load balancing: Uneven token distribution can overload certain experts. Designing gating mechanisms that balance performance and fairness is critical.

Infrastructure demands: Efficient EP relies on high-performance interconnects. Without optimized communication, such as NVLink and InfiniBand, scalability is limited due to the network bottleneck.

7.3 Parallelism in Switch Transformers

Expert and Data Parallelism

Next, we describe the partitioning strategy for expert and data parallelism in switch transformers [62]. In the explanation, we will use the following notation: B is the number of tokens in the minibatch, N is the number of

cores/devices, n is the data parallelism degree, m is the model parallelism degree, E is the number of experts in switch layers, and C is the expert capacity, that we can view as the batch size of each expert.

Switch transformers will allocate all of their cores to the data partitioning dimension n , which also corresponds to the number of experts in the model. For each token, a router local to each core computes its assignment to the experts. The output is a binary matrix of size $[n, B/n, E, C]$, which is partitioned across the first dimension and determines expert assignment. This binary matrix is then used to implement a gather operation through a matrix multiplication with the input tensor of size $[n, B/n, d_{model}]$. This operation can be expressed in Einstein notation as indicated by Equation 7.6.

$$\text{einsum}([n, B/n, d_{model}], [n, B/n, E, C], \text{dimension} = [B/n]) \quad (7.6)$$

resulting in the final tensor of shape $[n, E, C, d_{model}]$, which is sharded across the first dimension. Because each core has its own expert, we do an all-to-all communication of size $[E, C, d_{model}]$ to now shard the E dimension instead of the n -dimension. There are additional communication costs of `bfloat16` tensors of size $E \times C \times d_{model}$ in the forward pass to analogously receive the tokens from each expert located on different cores.

Expert, Model and Data Parallelism

Expert, model and data parallelism are combined to balance the FLOPS per token and the weight count. When the number of experts is scaled, the number of weights is increased, but do not change the FLOPs per token. In order to increase FLOPs, we must also increase the feed-forward network d_{ff} dimension, which also increases the model's weights, but at a slower rate. This presents a trade-off: as we increase d_{ff} , we will run out of memory per core, which then requires that we increase m . But since we have a fixed number of cores N , and $N = n \times m$, we must decrease n , which forces us to use of a smaller minibatch size in order to hold tokens per core constant.

When combining both model and expert parallelisms, we will have an all-to-all communication cost for routing the tokens to the correct experts along with the internal AllReduce communication cost associated to the model parallelism. Balancing the FLOPS, communication costs and memory per core becomes quite complex when combining all three methods where the best mapping is empirically determined.

7.4 Parallelism in Tutel

Tutel [140] is a comprehensive MoE framework that enables the construction of full MoE layers with built-in adaptive optimizations. These optimizations are seamlessly integrated and remain transparent to developers, meaning that Tutel does not alter the standard interfaces of existing deep learning frameworks and can be easily incorporated into them.

The dynamic behavior of MoE layers introduces additional complexity, especially when attempting to accelerate a single expert using multiple GPUs to improve throughput. Prior studies have shown that increasing the number of experts yields diminishing returns beyond a certain threshold, typically around 256 experts [163, 36, 61]. As a result, large-scale systems often deploy relatively few experts compared to the number of available GPUs, assigning multiple GPUs to each expert to boost performance.

We consider three common parallelization techniques previously applied to MoE systems [61]: expert parallelism (EP), which distributes experts across devices; data parallelism (DP), which partitions input data; and model parallelism (MP), which divides computations within a single expert. These techniques can also be combined to form hybrid configurations.

In this section, we will use the following **notation**: W is the world size used for All-to-All exchange, D is the feed-forward network (FFN) channel count for each sample, H is the FFN hidden size for each sample, E_g is the number of local experts per GPU, E is the total number of experts, C_g is the token capacity per GPU, C is the total token capacity across GPUs, P is the total weights of all experts, and f is the the capacity factor.

Experimental observations indicate that relying on a fixed parallelization strategy is often inefficient in dynamic workloads. For instance, Figure 7.2 compares EP+DP and EP+MP under varying conditions. The results show that performance differences between these two approaches can range from 7.39% to 27.76%, depending on workload characteristics. This highlights that no single configuration is universally optimal.

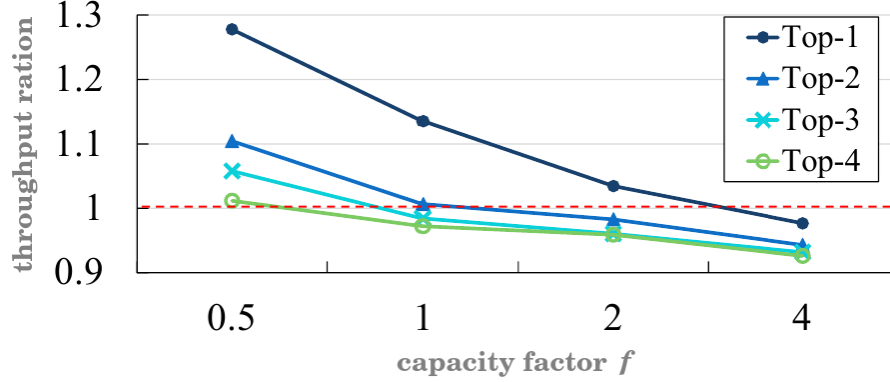


Figure 7.2: Runtime preferences of two different parallelism methods. The Y-axis measures the throughput ratio of EP+MP to EP+DP. It compares their throughput under varying capacity factor f , varying the amount of workload, and different top- k configurations, where greater than 1.0 implies that EP+MP outperforms EP+DP, and vice versa. Model configuration: the feed-forward network hidden size is 16k, the feed-forward network channel size is 2048, and the minibatch size is 4 [140].

Switching between parallelization schemes during execution, however, introduces significant overhead. Existing implementations typically bind training to a specific strategy, as different approaches impose distinct requirements on data partitioning, weight distribution, optimizer states, and execution APIs. Moreover, transitioning between strategies requires costly weight redistribution, as illustrated in Figure 7.3. These challenges largely explain why runtime switching is rarely adopted.

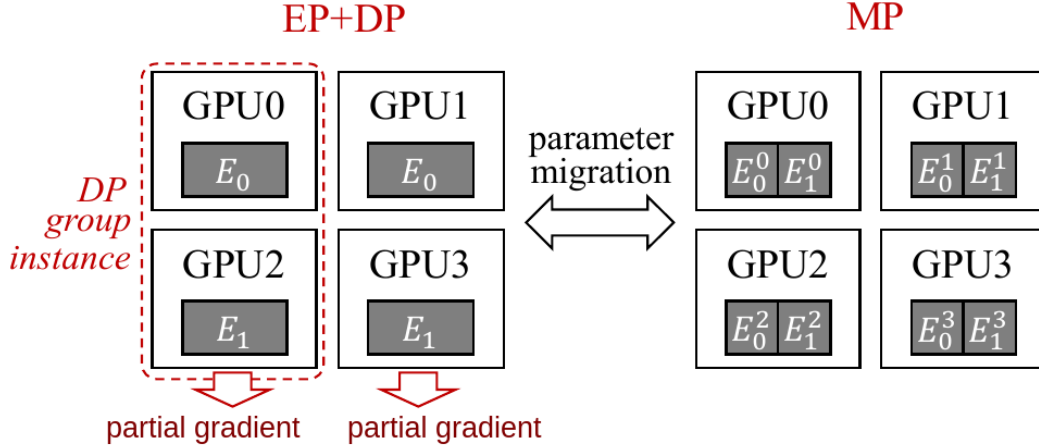


Figure 7.3: Weight migration due to switching parallelism between conventional EP+DP and MP. E_i^p refers to the p -th slice (in model parallelism) of the i -th expert. When superscript p is not included means that there is no slicing. EP+DP replicates each expert on two GPUs each, and MP slices each expert across four GPUs, respectively [140].

7.4.1 Static Pipelining

MoE layers shown in Figure 7.4 often fail to fully utilize GPU resources because All-to-All communication and feed-forward network (FFN) computation are executed sequentially to dispatch tokens and combine outputs. Since All-to-All primarily involves data transfer rather than heavy computation, overlapping it with FFN execution can significantly improve utilization. As demonstrated in Table 7.1, this overlap can provide speedups of up to 1.86 $\times$.

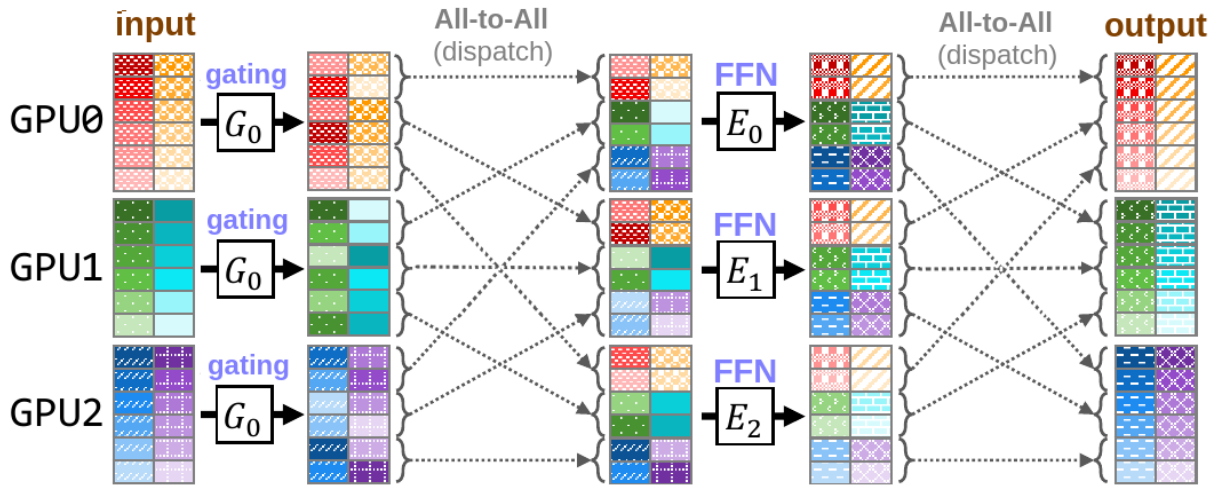


Figure 7.4: Example of an MoE layer across three GPUs, expert E_i on GPU i . G_0 represents the gating function that is shared across all GPUs. Different colors or patterns indicate different samples (columns of inputs) and different color gradients indicate different tokens within a sample (rows of inputs). This example considers two samples per batch, six tokens per sample, and evenly dispatched with top-1 routing with a capacity factor of 1.0 [140].

Number of GPUs	16	64	256
MoE overhead (ms)	560.9	698.9	866.4
Computation overhead (ms)	371.8	375.1	386.3
All-to-All overhead (ms)	189.1	323.8	491.3
All-to-All overhead ratio	33.7%	46.3%	56.7%
Potential overhead saving	33.7%	46.3%	43.3%
Potential speedup	1.51 ×	1.86 ×	1.76 ×

Table 7.1: Ratio of All-to-All overhead and potential speedup by fully overlapping All-to-All and computation in a typical MoE setting. Model configuration: the feed-forward network hidden size is 4k, the feed-forward network channel size is 4k, two experts per GPU, and 64k tokens per iteration [140].

Nonetheless, static pipelining approaches struggle to cope with the variability of MoE workloads. As depicted in Figure 7.5, optimal performance depends on both the choice of All-to-All algorithm, Linear or 2DH, and the degree of pipelining. Linear communication performs better at smaller scales (blue bars), whereas hierarchical (2DH) approaches are more effective at larger scales (red bars). This variability implies that fixed configurations are insufficient, motivating dynamic optimization during runtime.

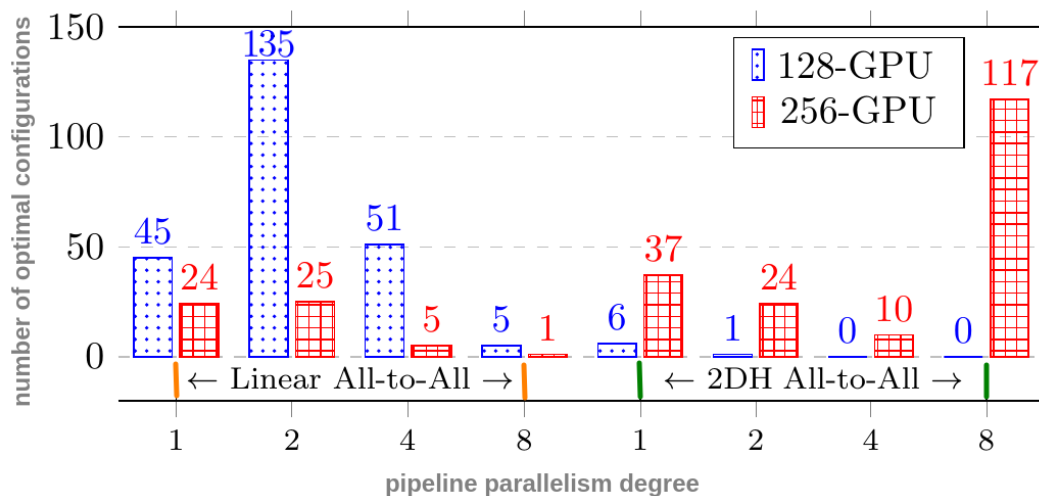


Figure 7.5: The distribution of optimal pipeline strategies for various MoE workload configurations. Each column indicates the number of configurations that perform best with the strategy described on X-axis [140].

Furthermore, the interaction between computation and communication complicates optimization. Performance differences between communication strategies may vary significantly when executed concurrently with computation kernels, making isolated evaluation unreliable. Effective optimization must therefore consider both aspects jointly.

7.4.2 Adaptive Parallelism Switching

Minimal subset for switching

Although there are seven possible configurations for combining expert, data, and model parallelisms, implementing all of them is unnecessary. Instead, Tutel reduces the problem by analyzing communication complexity, as computation cost remains consistent across strategies. As summarized in Table 7.2, methods that are suboptimal or equivalent to others can be eliminated.

This analysis shows that only two configurations need to be supported: DP and EP+DP+MP. These are sufficient to achieve optimal performance across different scenarios. In this way, the rest of the section focuses only on DP and EP+DP+MP.

Parallelism method	Communication complexity	Limitation	Comment
① DP	$O(P)$	-	Possibly optimal
② MP	$O(C_g \cdot W)$	-	No better than ⑥
③ EP	$O(C_g)$	$E/W \geq 1$	No better than ⑥
④ DP+MP	$O(C_g \cdot r + P/r)$	$1 \leq r \leq W$	No better than ⑦ for any r
⑤ EP+DP	$O(C_g + P/E)$	-	A special case of $r = 1$ in ⑦
⑥ EP+MP	$O(C_g \cdot \max\{1, W/E\})$	-	A special case of $r = W/E$ in ⑦
⑦ EP+DP+MP	$O(C_g \cdot W/E)$, if $r \geq W/E$ $O(C_g \cdot r + P/E/r)$, if $1 \leq r < W/E$	-	Possibly optimal

Table 7.2: Analysis on communication complexity of MoE parallelism in Tutel [140].

Execution flow of zero-cost switchable parallelism

As discussed in Section 7.4.1, a switchable parallelism mechanism must preserve a consistent data layout and execution pattern across different configurations of MoE training. In the following, we describe the designs for both DP and EP+DP+MP. The term *zero-cost* indicates that transitioning between parallelism schemes introduces no additional overhead beyond $O(1)$ associated with lightweight weight or token relocation.

Switchable data parallelism

The switchable DP scheme largely follows standard data-parallel training, where each device processes its local subset of tokens. However, model weights are partitioned according to the ZeRO stage-3 strategy [164] (see Figure 7.6). In this design, each GPU stores only a distinct shard of the weights. During the forward pass, a global AllGather operation reconstructs the full weight set, while the backward pass uses ReduceScatter to distribute the gradients.

This differs from conventional DP, which typically performs a single AllReduce during the backward pass. Nonetheless, both approaches have equivalent communication complexity, since AllReduce can be decomposed into a ReduceScatter followed by an AllGather. In Figure 7.7, the configuration $r = 0$ corresponds to this switchable data parallel mode.

Switchable EP+DP+MP

The hybrid EP+DP+MP configuration is designed to be fully compatible with switchable data parallelism. Both approaches share the same input format and weight partitioning scheme (Figure 7.8). Internally, however, the computation is organized to match the communication and computational complexity expected for the combined EP+DP+MP setting, as described in ⑦ of Table 7.2.

A tuning parameter r is introduced to indicate how GPUs are grouped. Specifically, all devices are divided into multiple groups of size $\lceil (W/E)/r \rceil$. DP is applied within each group, while MP is used across groups. At

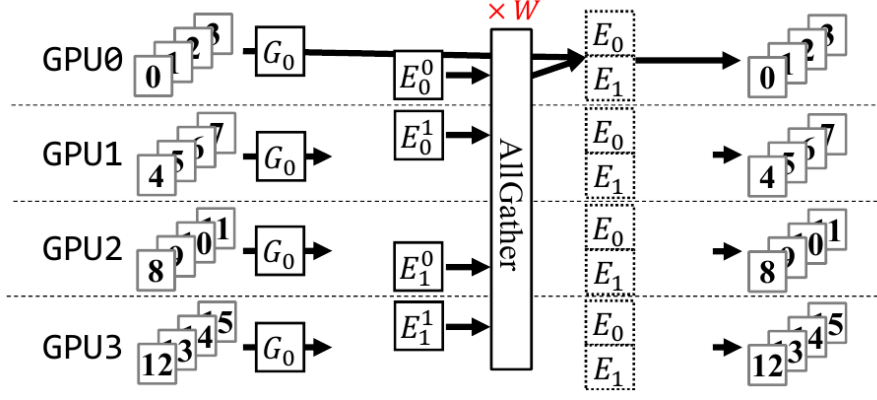


Figure 7.6: An example of data parallel execution flow in Tutel. AllGather is performed across all W GPUs [140].

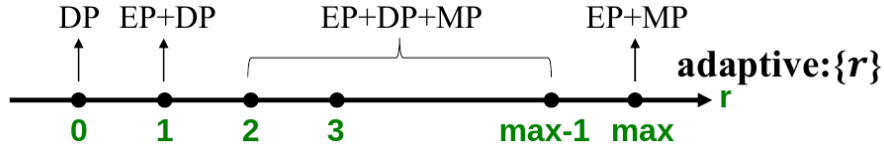


Figure 7.7: Specifying a parallelism method using `adaptive:r`, with `max` standing for the value $\lceil W/E \rceil$, and all r values larger than this upper-bound are treated as $\lceil W/E \rceil$ [140].

the start of execution, tokens are replicated r times in a manner similar to MP. At the end, a local reduction aggregates the results. AllGather communication is restricted to devices in the same group.

As r increases, the group size decreases. When r reaches W/E , each group contains a single device, effectively eliminating intra-group communication. This explains why for $r \geq W/E$, the additional $O(P/E/r)$ communication cost disappears. In Figure 7.7, values of r ranging from 1 to W/E correspond to EP+DP+MP configurations. The border cases $r = 1$ and $r = \text{max} = \lceil W/E \rceil$ correspond exactly to EP+DP and EP+MP, respectively.

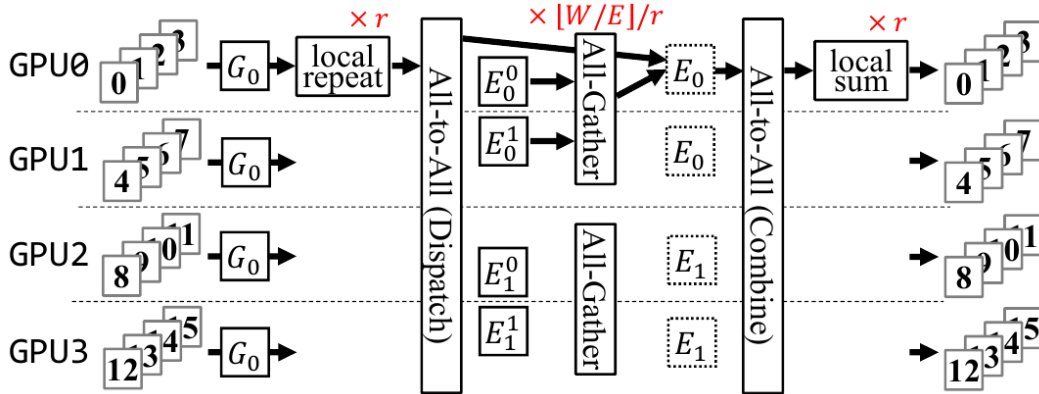


Figure 7.8: An example of EP+DP+MP execution flow in Tutel. Local repeat generates r copies of gating function results, local sum reduces r outputs from MoE combine, and all-gather is performed across $\lceil (W/E)/r \rceil$ GPUs [140].

7.4.3 Adaptive Pipelining for All-to-All Communication

This section introduces the concept of adaptive pipelining in Tutel. Since the latency of All-to-All communication plays a crucial role in determining the optimal pipelining configuration, the proposed approach jointly optimizes both the pipelining depth and the choice of All-to-All algorithm, Linear or 2DH. While the present discussion focuses on token partitioning for pipelining, Section 7.4.4 explains how the system identifies the optimal combination of pipelining weights and communication strategies.

Token partitioning for multi-stream pipelining

To effectively overlap computation and communication, tokens must be divided into smaller segments, enabling fine-grained parallel execution across multiple GPU streams. Each stream can then independently handle computation or communication tasks. Traditional partitioning methods, such as batch splitting or pipeline parallelism [89], typically divide all operations in a layer. However, applying such strategies directly to MoE models is problematic, as it can worsen load imbalance during token dispatch and interfere with mechanisms such as batch prioritized routing [170].

To address these issues, Tutel adopts a more selective partitioning approach: instead of dividing the entire MoE layer, only the two All-to-All communication steps and the intermediate expert computation are partitioned. This design avoids disrupting routing behavior while still enabling effective overlap. Figure 7.9 illustrates this strategy in a two-GPU setting.

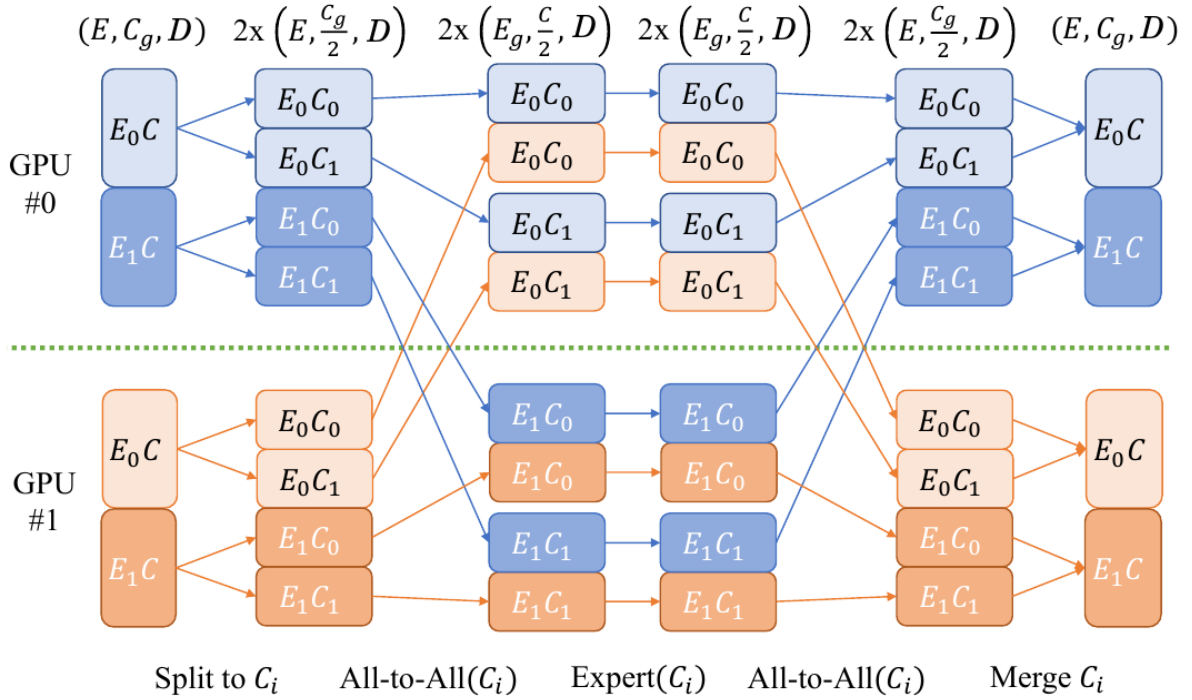


Figure 7.9: Overview of token partition on two-expert-two-GPU for overlapping All-to-All communication with expert computation. E_i means data is sent to i -th GPU and processed by i -th expert, and C_i means data belongs to i -th partition of capacity dimension. All-to-All and expert operations of different capacity partitions can be overlapped [140].

During the forward pass, each GPU receives input tensors of shape (E, C_g, D) , which are divided along the capacity dimension into two smaller partitions, each with shape $(E, C_g/2, D)$, denoted as C_0 and C_1 . These partitions are processed asynchronously. Specifically, each partition C_i is first sent to execute an All-to-All operation on the communication stream in sequence order. The customized All-to-All operator accepts these partitioned inputs and performs data shuffling on-the-fly, producing outputs of shape $(E_g, C/2, D)$.

Once an All-to-All operation completes for a partition, its corresponding output is immediately scheduled for expert computation on a separate computation stream. After the expert computation finishes, the resulting data is sent to a second All-to-All operation, again on the communication stream. A synchronization barrier is placed after completing the second All-to-All step. Following this barrier, the partitioned outputs are merged to reconstruct the final tensor of shape (E, C_g, D) .

The backward pass mirrors this workflow. The inputs are replaced by gradients of the outputs, the expert computation is replaced with the corresponding backward operations, and the final result yields gradients with respect to the original inputs.

It is important to note that all partitioning, reshaping, and merging operations are implemented using specialized operators that operate in-place. As a result, this design avoids additional memory copy overhead compared to

non-overlapped execution.

7.4.4 Dictionary of Optimal Parallelism and Pipelining

Tutel maintains a lookup structure to store the optimal configurations for parallelism and pipelining across different ranges of expert capacities. Concretely, this structure is implemented as a hash map of the form $\lfloor c/R \rfloor \rightarrow \{r^*, d^*, a^*\}$, where c denotes the expert capacity observed in a given training iteration, and R defines a discretization window, whose default value is 128, that groups nearby capacity values into the same index. The tuple $\{r^*, d^*, a^*\}$ represents the best-performing configuration, including the adaptive parameter r , the chosen pipelining depth, and the selected All-to-All communication strategy, respectively.

To construct this mapping in advance, the system evaluates candidate configurations for each possible key value $\lfloor c/R \rfloor$. The number of required trials to determine the optimal configuration is relatively small and can be estimated using Equation 7.7.

$$\text{trials per key} = (\log_{3/2} \lceil W/E \rceil + 2) \cdot 4 \cdot 2 \quad (7.7)$$

where $(\log_{3/2} \lceil W/E \rceil + 2)$ is the number of needed trials to search for r^* via Ternary Search because r in range $[1, \lceil W/E \rceil - 1]$ determines a convex optimal distribution, plus two extra trials for $r = 0$ and $r = \lceil W/E \rceil$. The factor "4" is the number of needed trials for d^* , since we limit the search space of the pipelining degree to $\{1, 2, 4, 8\}$. To our practices, larger degrees than 8 hardly improve the overlapping between computation and communication, while significantly inflating All-to-All overhead. The factor "2" refers to the number of All-to-All algorithms, Linear and 2DH.

7.5 DeepSpeed-MoE

7.5.1 Pyramid Residual MoE Architecture

DeepSpeed-MoE introduced an enhanced MoE design that improves the effectiveness of expert utilization across model layers [163]. The authors observed that later layers benefit more from increased expert capacity. Based on this insight, the proposed Pyramid-MoE structure allocates a larger number of experts to deeper layers compared to earlier ones. As illustrated in Figure 7.10 (i), the final layers may contain twice as many experts as initial layers.

In addition, a Residual-MoE architecture is proposed, where each token is processed through both a shared dense MLP and a selected expert. In this design, the dense MLP provides a consistent computation path, while the routed expert introduces specialization. In Figure 7.10 (ii), the fixed MLP components are represented by the orange blocks.

By integrating Pyramid-MoE and Residual-MoE, the resulting architecture is referred to as Pyramid-Residual-MoE (PR-MoE), in which all conventional MoE layers are replaced by PR-MoE layers. Figure 7.10 illustrates the differences between (i) standard MoE and (ii) PR-MoE designs.

This section first describes how MoE models can be trained efficiently using expert parallelism. We then outline the limitations of applying these techniques directly to PR-MoE models, followed by proposed extensions to support efficient training of PR-MoE architectures.

Efficient training of MoE models

Achieving high efficiency in MoE training requires maintaining sufficiently large token batches for each expert to ensure effective utilization of computational resources. However, in MoE layers, input tokens are divided among multiple experts, which reduces the number of tokens processed by each expert compared to dense layers.

A straightforward solution is to combine data parallelism with expert parallelism [105], typically aligning the degree of expert parallelism with the number of experts. This approach increases the overall number of

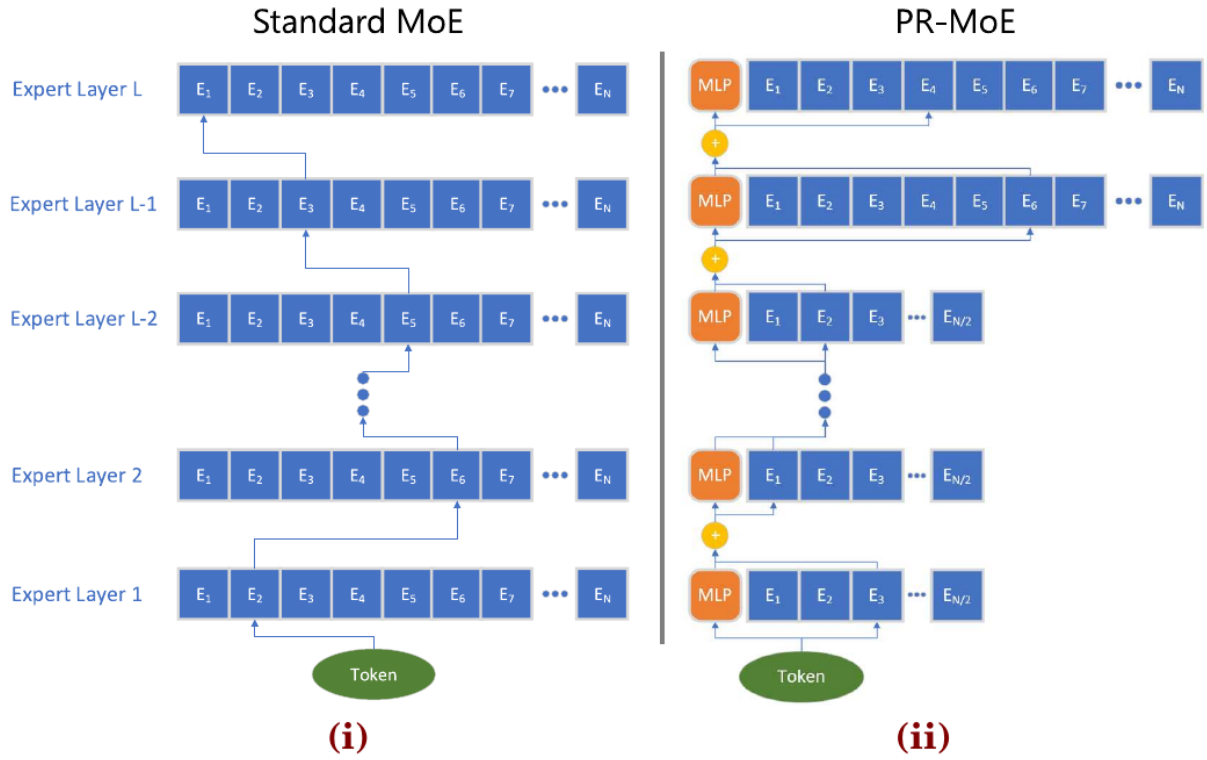


Figure 7.10: Illustration of (i) standard MoE and (ii) PR-MoE [163].

tokens processed per MoE replica, ensuring that each expert receives an adequate workload and maintaining comparable efficiency to dense layers.

Challenges in training PR-MoE

Training PR-MoE models introduces additional complexity due to the variable number of experts across layers. As previously noted, setting expert parallelism equal to the number of experts is often optimal for standard MoE models. However, this strategy cannot be directly applied to PR-MoE, where different layers have different expert counts.

If the parallelism degree is chosen based on the smallest number of experts, layers with more experts will require mapping multiple experts onto a single GPU. This leads to reduced batch sizes per expert and higher memory consumption. Conversely, selecting the parallelism degree based on the largest expert count results in uneven workload distribution, as some GPUs handle more experts than others, causing load imbalance and reduced efficiency.

DeepSpeed-MoE with multi-expert and multi-data parallelism

To overcome these limitations, an extended parallelization strategy is implemented within DeepSpeed-MoE. This approach introduces flexible multi-expert and multi-data parallelism, allowing different layers to use distinct parallel configurations.

For example, consider a PR-MoE model deployed on 128 GPUs with layers containing 32, 64, or 128 experts. In this case, non-MoE components can be trained using 128-way data parallelism, while MoE layers use variable expert parallelism degrees, {32, 64, 128}, combined with the corresponding data parallelism degrees, {4, 2, 1}. This ensures that each GPU is responsible for exactly one expert per layer, regardless of the total number of experts.

As a result, this design eliminates the reduction in tokens per expert, avoids load imbalance, and prevents additional memory overhead. With this flexible framework, DeepSpeed-MoE [105] can efficiently train PR-MoE models and adapt to future MoE variants with heterogeneous expert configurations.

7.5.2 DeepSpeed-MoE Inference

Efficient inference is essential for making MoE models practical in real-world applications, where both latency and computation cost are critical factors. Unlike training, inference typically operates with small batch sizes, meaning that latency is dominated by the time required to fetch model weights from memory rather than raw computation. Consequently, inference performance in MoE models is largely determined by two key aspects: the total weight count of the model and the effective memory bandwidth available.

Here, we will focus on system-level optimizations aimed at maximizing memory bandwidth utilization. Specifically, DeepSpeed-MoE (DS-MoE) leverages a distributed multi-GPU setup to aggregate memory bandwidth across many devices, thereby accelerating inference. This approach enables significant performance gains, achieving up to $7.3\times$ lower latency and reduced computation cost compared to baseline MoE systems, as well as up to $4.5\times$ faster inference and $9\times$ lower computation cost relative to dense models with comparable quality. DS-MoE is part of the broader DeepSpeed inference framework described in [6].

7.5.3 Design of DeepSpeed-MoE Inference System

MoE inference presents a unique trade-off between best-case and worst-case execution scenarios: (i) In the most favorable case, each token routed through an MoE model, using for example top-1 gating, activates only a single expert per layer. As a result, the effective computation path corresponds to the size of the underlying dense model, which is significantly smaller than the total weight count. For instance, in a 1.3B+MoE-1284 model, only about 1.3 billion weights are accessed per token, despite the full model containing 52 billion weights; (ii) In contrast, the worst-case scenario arises when different tokens within a batch activate different experts. In such situations, the combined set of accessed weights may approach the full model size, making it difficult to achieve low latency and high throughput.

The goal of the DeepSpeed-MoE inference system is to move performance closer to the best-case scenario. This is accomplished through three complementary strategies.

- **Model partitioning and parallelism:** The model is carefully distributed across multiple GPUs using appropriate parallelism techniques. Tokens that share similar execution paths are grouped and processed together, reducing redundant weight loading and maximizing effective memory bandwidth.
- **Communication optimization:** Efficient scheduling and coordination of communication operations ensure that token routing and data movement across devices are performed with minimal overhead.
- **Kernel-level optimization:** Specialized optimizations for transformer and MoE operations improve execution efficiency on each device, further reducing inference latency.

The following sections provide a detailed analysis of these three optimization strategies.

7.5.4 Flexible Combination of Tensor-Slicing, Expert-Slicing, Data Parallelism, and Expert Parallelism

To enable low-latency and high-throughput inference for large-scale MoE models, our system is designed with three key objectives: reducing the critical path of data processing on each device, maximizing the utilization of aggregate memory bandwidth, and leveraging the combined memory capacity of multiple GPUs. To achieve this, we employ a combination of parallelization techniques: (i) expert parallelism [105] together with expert-level weight slicing for MoE components, and (ii) data parallelism and tensor-slicing for non-expert components. Figure 7.11 illustrates how these strategies are applied within a single MoE Transformer layer, which consists of both expert modules, like feed-forward networks, and non-expert components, like attention layers.

In the following, we describe how different parts of the model are partitioned and processed.

Expert parallelism and expert-slicing for expert weights

As highlighted by the MoE inference trade-off, although each token activates only a single expert per layer, a batch of tokens may require access to a large portion, or even the entirety, of the expert weights. This creates

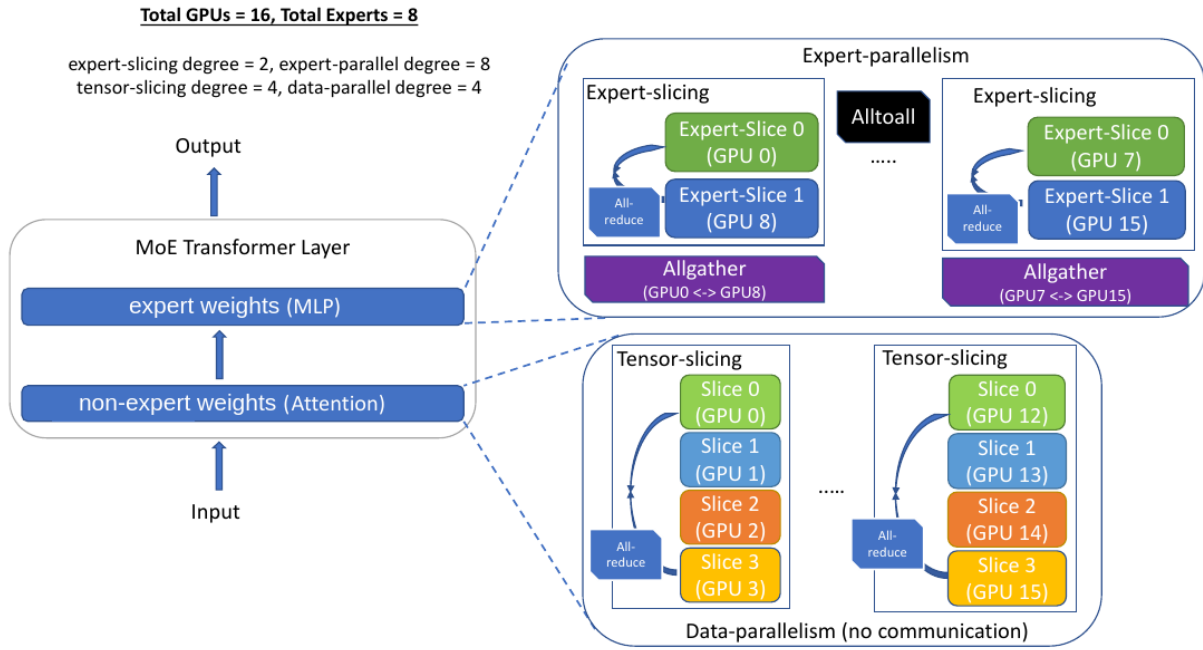


Figure 7.11: DS-MoE design that embraces the complexity of multi-dimensional parallelism for different partitions, expert and non-expert, of the model [163].

challenges for achieving both low latency and high throughput.

To mitigate this, experts are distributed across multiple devices, and tokens that are routed to the same expert are grouped together and processed along a shared execution path. Using expert parallelism, different groups of tokens can be processed concurrently on different GPUs, improving throughput while limiting the weight footprint per device.

For example, in a 1.3B+MoE-128 model, setting the expert parallelism degree to 128 ensures that each GPU handles a single expert and its corresponding token group. In this case, the effective model size per device is reduced to approximately 1.3 billion weights, significantly smaller than a dense model of equivalent quality, which has 6.7B weights. In an ideal scenario without communication overhead, this reduction suggests that MoE inference could achieve up to 5× speedup.

To further enhance scalability, we introduce expert-slicing, which extends tensor-slicing techniques to expert weights. This approach partitions individual experts across multiple GPUs, either horizontally or vertically, providing an additional level of parallelism. Expert-slicing is particularly beneficial in latency-sensitive scenarios where the system scales beyond the number of available experts.

Data parallelism and tensor-slicing for non-expert weights

While expert parallelism effectively reduces the weight footprint associated with expert modules, it does not address the non-expert components of the model. These components, such as attention layers, can still dominate memory usage and limit performance due to constraints of individual GPU memory and bandwidth.

To overcome these limitations, tensor-slicing is applied within a node, distributing non-expert weights across multiple GPUs. This allows the system to leverage both the combined memory capacity and the aggregate memory bandwidth within a node, enabling support for very large non-expert weight counts.

Although tensor-slicing could, in principle, be extended for multiple nodes, doing so typically incurs significant communication overhead and reduces computational efficiency due to smaller workload granularity. Instead, inter-node scaling is achieved using data parallelism, where replicas of non-expert weights process different input batches independently. This strategy avoids inter-node communication costs and maintains efficient utilization of computational resources.

Synergy of multidimensional parallelism

By integrating expert parallelism and expert-slicing with tensor-slicing and data parallelism, the DS-MoE inference framework can scale to extremely large models, including those with trillions of expert weights and hundreds of billions of non-expert weights. This multidimensional parallelization minimizes the processing burden on individual devices while leveraging the aggregated bandwidth and memory resources of large GPU clusters.

As a result, the system is able to achieve both low latency and high throughput at scale. However, fully realizing these benefits also depends on efficient communication primitives and optimized device-level kernels, which are discussed in the subsequent sections.

7.5.5 Optimized Communication Subsystem

Expert parallelism relies heavily on All-to-All communication across devices that host the different experts. In DeepSpeed-MoE, this communication is initially implemented using NCCL through the `torch.distributed` interface. However, at large scale, this approach introduces significant overhead. To improve performance, the authors designed a customized communication layer that integrates Microsoft SCCL [20], achieving better efficiency compared to the standard NCCL implementation.

Even with these optimizations, scaling expert parallelism to a large number of devices remains challenging, as communication latency tends to increase linearly with the number of participating GPUs. To mitigate this issue, they also introduced two communication optimizations that leverage low-level point-to-point NCCL primitives and specialized CUDA kernels for data layout transformations.

Hierarchical All-to-All

Hierarchical communication strategies are commonly used in collective operations such as AllReduce and broadcast to reduce communication depth. Inspired by this idea, they implemented a hierarchical All-to-All mechanism using a two-stage process. First, a data layout transformation is applied, followed by an intra-node All-to-All communication. A second transformation is then performed before executing an inter-node All-to-All.

This hierarchical structure reduces the effective communication complexity from $O(p)$ to $O(G + p/G)$, where G denotes the number of GPUs per node and p represents the total number of GPUs. Figure 7.12 illustrates this design. Although this approach increases the total volume of communication by approximately a factor of two, it improves scalability in practice, particularly for small batch sizes where latency dominates bandwidth considerations.

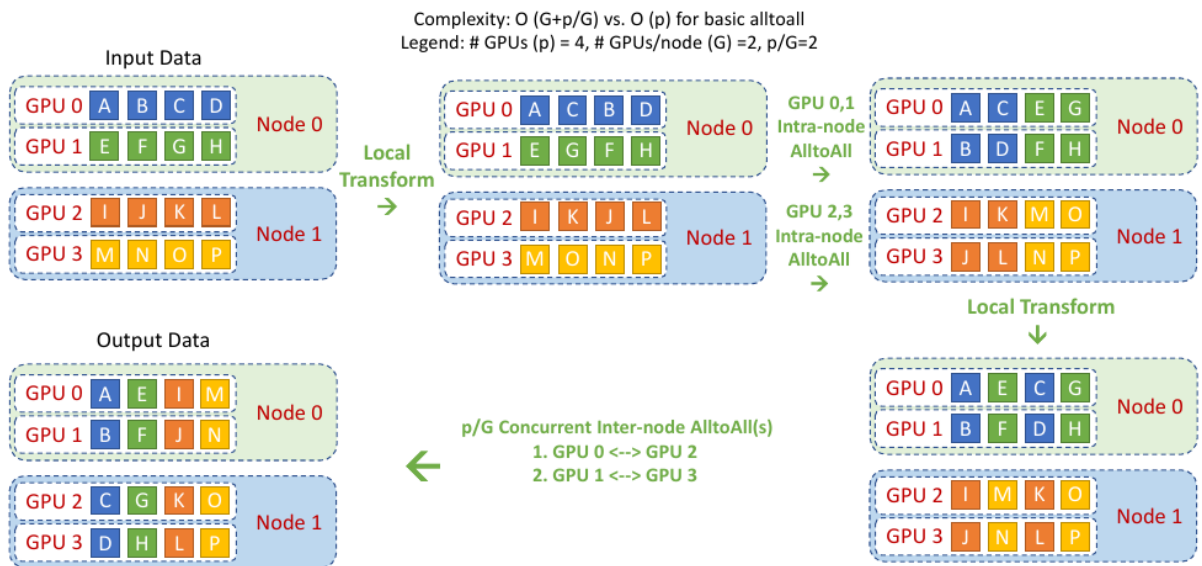


Figure 7.12: Illustration of the DeepSpeed-MoE hierarchical All-to-All design [163].

Parallelism-coordinated communication optimization

Effectively coordinating communication across multiple parallelism dimensions, such as expert parallelism, tensor-slicing, and data parallelism, is non-trivial. Tensor-slicing distributes computations across devices and requires AllReduce operations, whereas expert parallelism assigns different experts to devices and requires All-to-All communication. Treating these communication patterns independently results in inefficiencies.

A more efficient approach leverages the interaction between these parallel strategies. During tensor-slicing, the AllReduce operation replicates data across participating devices. When tensor-parallel computations are followed by expert-parallel operations, this redundancy can be exploited to restrict All-to-All communication to smaller subsets of devices. Specifically, All-to-All exchanges can be performed only among devices sharing the same tensor-parallel rank, instead of across all expert-parallel devices, as illustrated in Figure 7.13. This reduces the communication latency from $O(p)$ to $O(p/L)$, where L is the tensor-slicing degree.

A similar optimization applies when the execution order is reversed. If expert-parallel computation is followed by tensor-slicing operations, the All-to-All communication can still be restricted to subsets of devices. This is then followed by an AllGather operation across tensor-parallel groups to replicate the required data. In this case, the overall latency is reduced from $O(p)$ to $O(p/L) + O(L)$.

These optimizations significantly improve scalability. For example, in a system with 128 GPUs using 8-way tensor-slicing and 128-way expert parallelism, the communication cost of the All-to-All operation can be reduced from approximately $(128C_1 + C_2)$ to $(16C_1 + C_2)$, where C_1 and C_2 are constants related to latency, message size, and bandwidth characteristics.

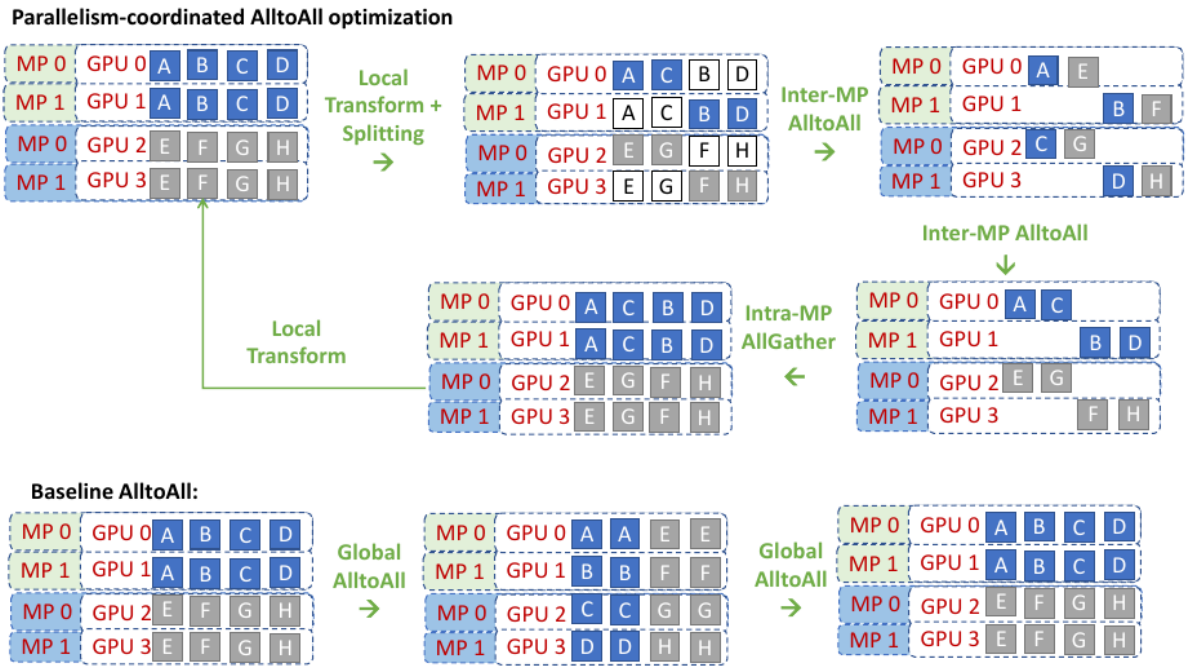


Figure 7.13: Illustration of the DeepSpeed-MoE parallelism coordinated communication [163].

7.5.6 Optimized Transformer and MoE Kernels

The DS-MoE inference system uses highly optimized GPU kernels for both transformer and MoE components to maximize performance, especially memory bandwidth utilization. For MoE-specific operations, namely gating and token routing, traditional implementations rely on sparse tensor computations and einsum operations, which are inefficient due to high computational complexity and unnecessary operations involving zeros.

To address this, DS-MoE introduces two main optimizations: (i) dense representations, which instead of sparse tensors, a compact token-to-expert mapping table is used, reducing memory usage and computation overhead,

(ii) kernel fusion, where multiple operations, like top-k selection, prefix sum, or token routing, are combined into a single GPU kernel, lowering kernel launch overhead.

Additionally, expensive sparse einsum operations are replaced by efficient data layout transformations, reducing computational complexity from cubic to quadratic, that is, from $S \times E \times M \times c_e$ to $S \times M \times c_e$. These optimizations significantly improve efficiency, resulting in over 6 times reduction in MoE kernel latency.

7.6 Pathways

Pathways introduced a novel approach to distributed machine learning, by rethinking how large-scale training workloads are executed across accelerator hardware [9]. Conventional distributed training frameworks typically rely on the single program multiple data (SPMD) paradigm, in which all devices execute identical computations in a synchronized manner, coordinated through collective communication primitives. While this approach is efficient for dense and homogeneous workloads, it becomes limiting when dealing with emerging training strategies such as model pipelining and sparse models. These shortcomings are highlighted in Figure 7.14, which compares the low-latency yet inflexible behavior of SPMD systems with earlier centralized execution models, such as TensorFlow v1, which incur higher coordination overhead.

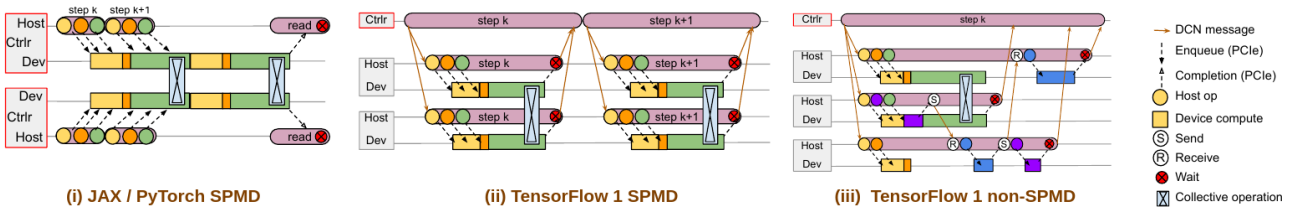


Figure 7.14: Comparison of dispatch overheads and communication patterns between multi-controller and single-controller systems. (i) Jax and PyTorch SPMD independently enqueues accelerator computations asynchronously over fast PCIe; (ii) TensorFlow v1 SPMD requires control messages over slower datacenter network; (iii) TensorFlow v1 non-SPMD programs require cross-host coordination or data transfer through explicit send (S) and receive (R) operations [9].

To address these issues, Pathways adopts an asynchronous dataflow-based execution model. Rather than enforcing globally synchronized training steps, it models computation as a directed graph of interdependent operations. In this representation, each node corresponds to a compiled unit of computation, for example, a model shard, while edges capture the flow of data between them. A key feature of this design is the use of futures for data exchange, allowing operations to begin as soon as their required inputs are available, instead of waiting for system-wide synchronization points. This concept is fundamental to the architecture and is illustrated in Figure 7.15 (i), where the distributed program appears as a dataflow DAG composed of computational nodes linked by data dependencies.

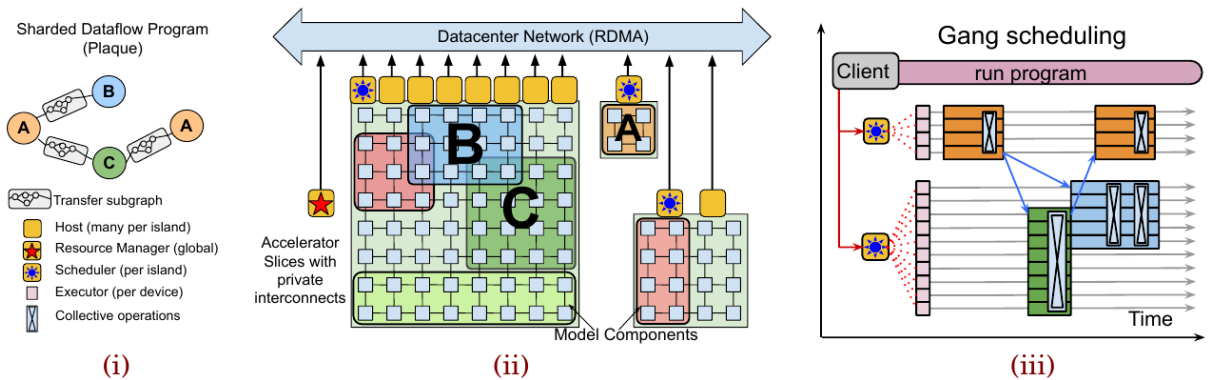


Figure 7.15: Pathways system overview. (i) Distributed computation expressed as a DAG where each node represents an individual compiled function, and edges between nodes represent data flows between functions. (ii) Resource Manager allocates subsets of an island's accelerators ("virtual slices") for each compiled function. (iii) Centralized schedulers for each island gang-schedule computations that are then dispatched by per-shard executors. Red arrows indicate control messages, blue arrows show data-path transfers [9].

A central feature of this design is that, although the dataflow graph is logically partitioned, it remains compact in its representation. Even when a computation is distributed across thousands of shards, Pathways does not explicitly construct every individual dependency between shards. Instead, each computation is abstracted as a single node, leaving the runtime responsible for managing how data is partitioned and routed across devices. This approach prevents the graph from growing excessively large, making it possible to scale efficiently to clusters with thousands of accelerators. The authors highlight that keeping the graph representation compact is essential for controlling both scheduling complexity and communication costs at scale.

From an architectural standpoint, Pathways integrates this flexible execution model with a form of centralized coordination. As illustrated in Figure 7.15 (ii) and (iii), the system is composed of a global resource manager, schedulers assigned to individual islands, and executors operating at the device level. The resource manager maps computations onto "virtual devices," effectively decoupling logical execution from the underlying physical hardware. At the same time, schedulers coordinate execution across accelerators using gang scheduling, which enforces a consistent ordering of operations and ensures that distributed computations proceed efficiently without deadlocks. This centralized yet scalable structure enables Pathways to handle heterogeneous workloads and shared cluster usage, capabilities that are difficult to achieve with traditional multi-controller approaches.

One of the primary challenges faced by centralized dataflow systems is dispatch latency, which arises from the overhead of coordinating and launching operations across many devices. This issue is illustrated in Figure 7.16 (i), where sequential dispatch results in pipeline stalls when host-side coordination takes longer than the computation being performed. Pathways addresses this limitation by introducing a model of parallel asynchronous dispatch, depicted in Figure 7.16 (ii). The key observation is that compiled machine learning functions often have statically known shapes and resource requirements, allowing the system to prepare and schedule multiple operations ahead of time, even before all dependencies are fully satisfied. As a result, host-side tasks such as buffer allocation and scheduling can be overlapped across multiple operations, effectively removing the bottlenecks associated with sequential dispatch.

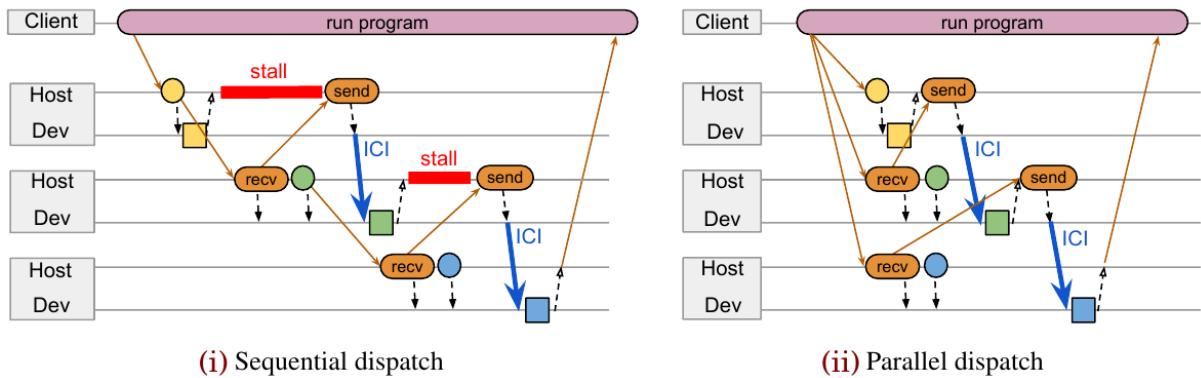


Figure 7.16: Sequential vs. parallel dispatch for a three-node program. When a computation's on-device execution time is shorter than the time spent in scheduling, resource allocation, and coordination, the asynchronous pipeline stalls because of host-side work in sequential dispatch. Parallel asynchronous dispatch overcomes this bottleneck by running the host-side work in parallel, exploiting the statically known resource usage of regular compiled functions. Schedulers are omitted for brevity [9].

Another key component of the design is the introduction of virtual devices and flexible placement strategies. Rather than binding computations to specific physical hardware, Pathways allows users to define sets of virtual devices, which are then mapped onto available accelerators by the system. This level of abstraction simplifies the implementation of distributed training techniques, particularly those that require partitioning models across multiple devices or even across clusters. Additionally, it improves overall system utilization by enabling multiple workloads to share hardware resources in a dynamic and efficient manner.

By combining asynchronous execution, centralized coordination, and adaptive resource management, Pathways is able to naturally support a broad spectrum of modern training paradigms. For instance, pipeline parallelism can be represented as a chain of dependent operations within the dataflow graph, while sparse models can dynamically direct computation to selected subsets of devices. Furthermore, the system can execute multiple programs concurrently on shared infrastructure. In contrast to traditional SPMD-based systems, which often

require complex engineering workarounds to accommodate such patterns, Pathways incorporates them directly into its execution model.

Despite this flexibility, Pathways maintains performance levels comparable to highly optimized SPMD approaches. The authors report near 100% utilization of accelerators for standard workloads, along with strong throughput for more challenging configurations such as pipeline-parallel training. These results indicate that an asynchronous dataflow approach, when paired with effective scheduling and optimized dispatch mechanisms, can achieve efficiency on par with conventional methods while supporting a much wider range of workloads.

Pathways represents a transition away from rigid, synchronization-driven distributed training architectures toward a more general and expressive framework based on asynchronous dataflow principles. The work demonstrates how such a shift influences system design at multiple levels, from execution and scheduling to resource management. By decoupling computation from strict global synchronization and introducing scalable, flexible orchestration, Pathways offers a solid foundation for training future large-scale, sparse, and heterogeneous machine learning models.

7.7 Parallelism in Megatron-Core

7.7.1 Expert Parallelism

In distributed training, different parallelism strategies partition distinct aspects of a model: layers in pipeline parallelism, weight matrices in tensor parallelism, sequences in context parallelism, and data samples in data parallelism. Mixture-of-Experts models introduce an additional opportunity, as their experts are independent subnetworks. This property enables a fifth form of parallelism, known as expert parallelism (EP), where experts themselves are distributed across GPUs. With an expert parallelism degree equal P and E experts, each GPU stores E/P experts. The forward pass in MoE training with expert parallelism is described by the following steps.

1. Routing: A routing mechanism selects the top- k experts for each input token.
2. Dispatch: Tokens are redistributed via all-to-all communication to the GPUs hosting the selected experts.
3. Computation: Each GPU processes only the tokens assigned to its local experts.
4. Aggregation: Results are sent back to the originating GPUs, again using all-to-all communication.

Expert parallelism introduces a distinctive set of trade-offs.

- Communication: Relies heavily on all-to-all collectives, with communication volume scaling with the number of tokens rather than the number of experts.
- Computation: Each GPU evaluates a smaller subset of experts, but each expert operates over the full hidden dimension.
- Memory: Increasing the EP degree reduces the number of experts per GPU, thereby lowering memory requirements.

This approach offers two important advantages (Figure 7.17). First, aggregating tokens from multiple GPUs for a single expert increases computational intensity, improving the efficiency of matrix-matrix multiplications. Second, the total communication volume remains effectively constant as the number of experts grows; scaling is achieved by increasing the number of GPUs rather than the cost of communication.

However, expert parallelism alone is insufficient. Its applicability is limited to MoE layers, since attention layers do not contain experts and therefore still depend on other parallelization strategies such as tensor parallelism, context parallelism, or similar approaches. In addition, both dense layers, like attention, and sparse layers, as MoE experts, require pipeline parallelism to further divide model parameters when training at large scale. This combination of different parallelism strategies introduces a structural imbalance between dense and sparse layers, leading to the so-called dense-sparse mismatch described in the following section.

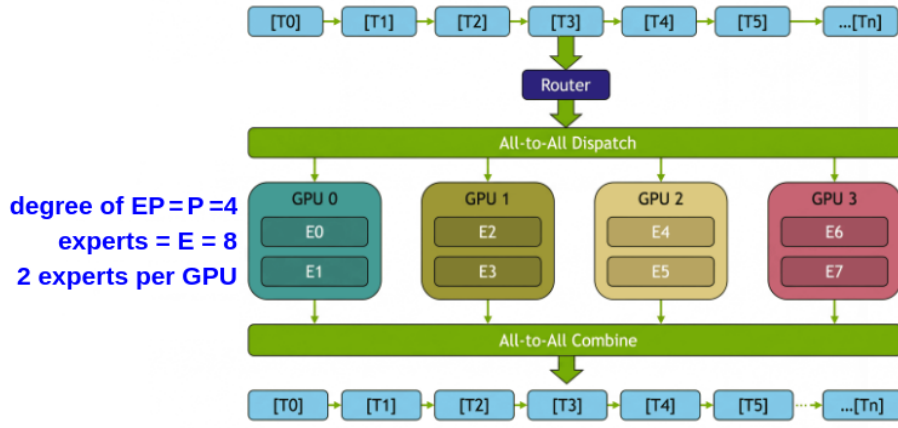


Figure 7.17: Expert parallelism distributes experts across GPUs. The all-to-all communication dispatches tokens to their assigned experts and combines results.

7.7.2 The Challenges of Combining Expert Parallelism with Other Types of Parallelism

A single transformer block contains two fundamentally different computation patterns, summarized in Table 7.3.

Aspect	Attention (Dense)	MoE (Sparse)
Computation	Every token attends to all others	Each token routes to k out of E experts
TP	Large QKV matrices benefit from high TP	Small per-expert dimensions make high TP counterproductive
CP	Long sequences benefit from high CP	No sequence dependency; CP is irrelevant
EP	Not applicable (no experts)	Essential for distributing many experts

Table 7.3: Contrasting parallelism requirements of attention and MoE layers within a single transformer block.

The dense-sparse mismatch. Conventional frameworks typically enforce a single parallelism configuration across all layer types. However, the optimal settings for dense and sparse components are fundamentally at odds. High tensor parallelism (TP) improves performance for attention layers but leads to fragmentation of expert partitions in MoE layers. Similarly, context parallelism (CP) is advantageous for long-sequence attention, yet offers no benefit for MoE computation. In contrast, expert parallelism (EP) is critical for scaling MoE layers but has no impact on attention.

As with the weight-compute imbalance, this mismatch originates from the sparsity inherent in MoE models. However, in this case, it manifests at the level of parallelism design: it is a structural incompatibility between the requirements of dense and sparse layers, rather than a problem that can be resolved through parameter tuning. Existing MoE frameworks typically incorporate EP as a sub-dimension of data parallelism (DP), as indicated in Equation 7.8.

$$\text{World Size} = TP \times CP \times PP \times DP, \text{ where } EP \subseteq DP \quad (7.8)$$

This limitation arises from a core assumption in existing frameworks: the use of a uniform parallelism strategy across all layers. In such designs, attention layers operate with a configuration defined by (TP, CP, PP, DP), while MoE layers derive expert parallelism (EP) as a subdivision of the data parallel (DP) group. Although convenient, this approach introduces several fundamental challenges.

Challenge 1: multiplicative GPU requirements. Traditional systems require at least $TP \times CP \times PP \times DP$ GPUs. Because EP is embedded within DP, specifying $EP = 8$ implicitly enforces $DP \geq 8$. When combined with $CP = 8$ for long-context workloads, the minimum configuration becomes $1 \times 8 \times 1 \times 8 = 64$ GPUs. This is unnecessarily restrictive, as both attention and MoE layers could, in principle, operate on the same set of 8

GPUs. As a result, the hardware requirements for MoE training are artificially inflated, increasing the barrier to entry.

Challenge 2: constrained parallelism choices. Since both attention and MoE layers must share the same tensor parallel (TP) configuration, practitioners are forced into suboptimal trade-offs. A high TP setting (e.g., $TP = 8$) is well-suited for distributing large attention matrices but causes MoE experts—typically much smaller—to be overly fragmented, reducing computational efficiency. On the other hand, a low TP configuration (e.g., $TP = 1$) preserves the integrity of expert computations but underutilizes parallelism in attention layers. Neither option allows both components to achieve their ideal performance simultaneously.

Challenge 3: inefficient cross-node communication. Because EP is constrained within DP, increasing EP often forces all-to-all communication to span multiple nodes, where the available bandwidth can be 5 to 10 times lower than within-node interconnects such as NVLink. At the same time, context parallelism (CP) for attention layers may also require cross-node data exchange. Without the ability to independently map EP and CP to high-bandwidth communication domains, communication overhead quickly becomes a dominant cost during training.

Importantly, these issues are not simply the result of poor parameter tuning. Instead, they stem from the underlying assumption that all layers must adhere to a single, shared parallelism configuration. This raises a fundamental question: how can these constraints be relaxed while still enabling each layer type to operate under its most suitable parallelization strategy?

7.7.3 Parallel Folding

Parallel folding is introduced in Megatron-Core as a solution to the dense-sparse mismatch problem. Instead of requiring attention and MoE layers to operate under a shared parallelism configuration, it separates their mappings, allowing each to adopt the most suitable parallelization strategy.

Decoupled parallelism mappings

The central idea is to remove the constraint that attention and MoE layers must share the same parallel layout. Each type of layer is instead assigned an independent configuration tailored to its computational characteristics. In this scheme, attention layers are organized using a combination of $TP \times CP \times DP \times PP$, which is well-suited for dense, sequence-oriented computation. In contrast, MoE layers operate with a distinct configuration defined by $ETP \times EP \times EDP \times PP$, where expert tensor parallelism (ETP) and expert data parallelism (EDP) are specialized dimensions introduced for handling expert-based computation. The only shared constraint between these two layouts is pipeline parallelism (PP), which must remain aligned to ensure correct forward and backward propagation across layers (Figure 7.18).

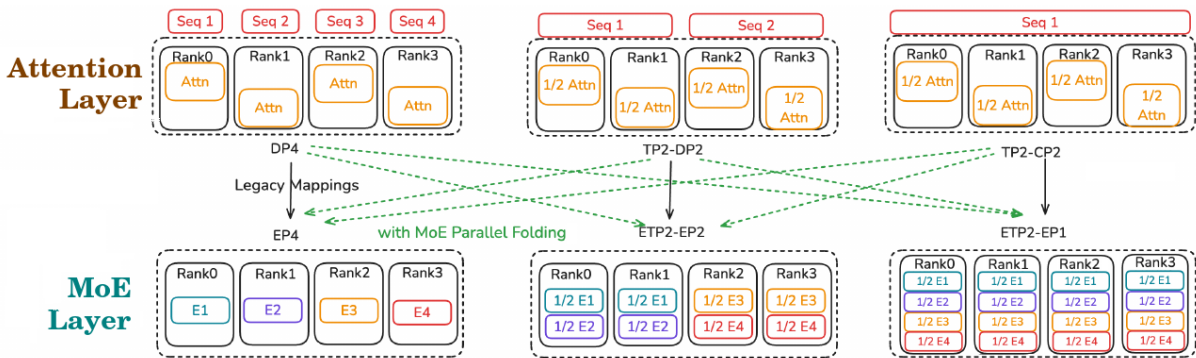


Figure 7.18: Parallelism mappings: traditional constraints vs. MoE parallel folding decoupling.

Multi-dimensional parallelism stack

Building on this decoupling, Megatron-Core coordinates a richer parallelism design that spans five distinct dimensions: tensor, context, data, pipeline, and expert parallelisms (Table 7.4). This unified framework enables flexible composition of parallel strategies, allowing dense and sparse components of the model to scale efficiently without interfering with each other's optimal configurations.

Dimension	Applies to	Purpose
TP (Tensor)	Attention	Shard large QKV/projection matrices
CP (Context)	Attention	Distribute long sequences
DP (Data)	Attention	Process different batches
PP (Pipeline)	Both	Split model by layers (must be consistent)
EP (Expert)	MoE	Distribute experts across GPUs
ETP (Expert Tensor)	MoE	Shard within experts (rarely used)
EDP (Expert Data)	MoE	Replicate experts for throughput

Table 7.4: Megatron-Core parallelism dimensions.

Key configuration guidelines can be summarized as follows: (i) attention layers benefit from high tensor parallelism to efficiently process large matrix operations and from high context parallelism to handle long sequences; (ii) MoE layers are best suited to configurations with high expert parallelism, typically with $ETP = 1$, as they involve many smaller expert networks; and (iii) pipeline parallelism must remain aligned across both layer types to preserve correct data flow throughout forward and backward passes.

In addition to the parallelism strategies outlined above (Figure 7.19), Megatron-Core extends the parallel folding framework by incorporating memory optimization techniques such as distributed optimizers and Fully Sharded Data Parallel (FSDP) with expert parallelism support. When combining a **distributed optimizer with expert parallelism**, each rank stores only the weights and gradients corresponding to its local experts. Optimizer states are partitioned across replicas of the same expert using expert data parallelism (EDP), eliminating redundant storage for non-local experts and restricting gradient synchronization to smaller, more efficient groups.

Further memory savings can be achieved by integrating **FSDP with expert parallelism**. Megatron-Core implements a customized FSDP scheme that shards model parameters, gradients, and optimizer states across both data and expert parallel groups using a dual DeviceMesh abstraction. This design significantly reduces memory overhead while overlapping communication operations, such as AllGather and ReduceScatter, with computation to maintain performance. The approach is compatible with tensor, expert, and context parallelism, and supports mixed-precision training, including Bfloat16, FP8, and FP4 formats.

7.7.4 Benefits of Parallel Folding

As shown in Figure 7.18, parallel folding in MoE removes the traditional constraint $EP \leq DP$ by allowing expert parallelism to *fold* across arbitrary subsets of the attention parallelism dimensions. This results in several key advantages:

1. **Relaxed EP scaling beyond DP:** The restriction $EP \leq DP$ no longer applies, since EP can span across the combined $TP \times CP$ (and DP) groups. For example, consider an attention configuration with $TP = 4$, $CP = 2$, $DP = 8$, and $PP = 4$, totaling 256 GPUs. In a traditional setup, EP is bounded by DP, limiting $EP \leq 8$. With parallel folding, however, MoE layers can use $ETP = 1$, $EP = 64$, and $EDP = 1$ (while keeping $PP = 4$ unchanged). In this case, EP effectively spans the $TP \times CP \times DP$ space, enabling an eightfold increase in expert parallelism without affecting the optimal attention configuration.
2. **Lower minimum hardware requirements:** Conventional setups with $CP = 8$ and $EP = 8$ require at least 64 GPUs due to their multiplicative combination. With parallel folding, CP and EP can be mapped onto the same set of devices, reducing the minimum requirement to just 8 GPUs.
3. **Independent performance tuning:** Decoupling allows each layer type to operate under its preferred configuration. Attention layers can leverage high TP values for efficient handling of large matrix operations, while MoE layers can adopt $ETP = 1$ to preserve full expert width and maximize matrix-matrix multiplication efficiency.
4. **Efficient communication locality:** Both attention, via CP, and MoE, via EP, rely on all-to-all communication. Parallel folding enables these communication patterns to remain within high-bandwidth, NVLink-connected GPU groups, minimizing the need for slower cross-node data transfers.

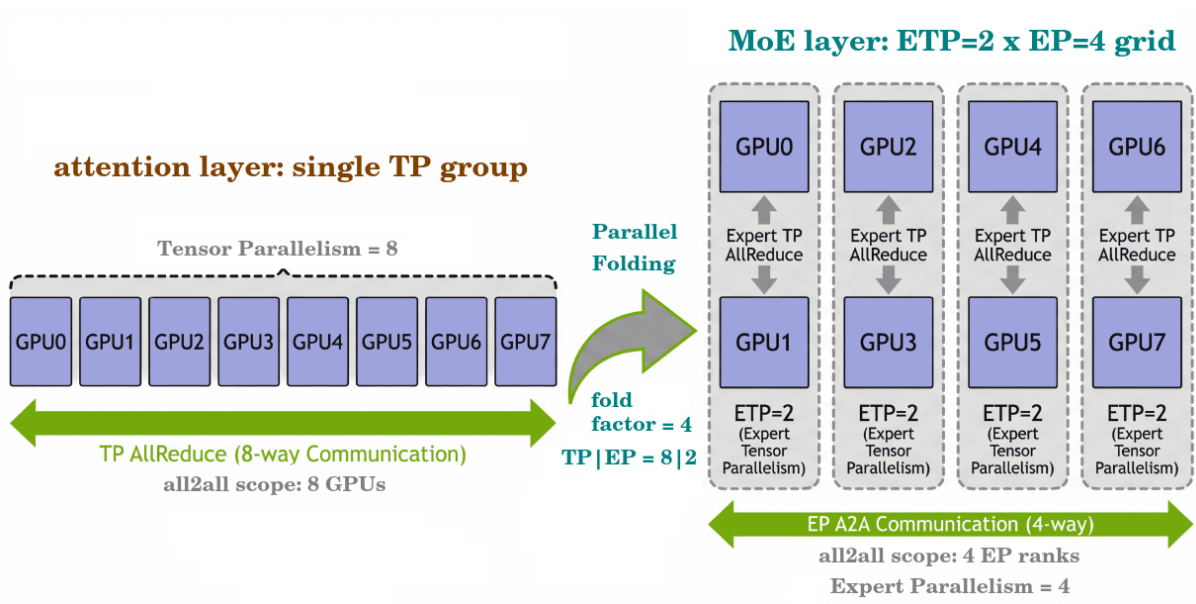


Figure 7.19: Parallel folding: decoupled attention and MoE parallelism mappings.

7.8 Expert Parallelism in NVIDIA Megatron Bridge

As we have been discussing, expert parallelism distributes experts of a Mixture of Experts model across GPUs. Unlike other model-parallel techniques, EP is applied to only the expert layers and does not impact the parallel mapping of the rest of the layers [42]. In an MoE, each expert focuses on a specific subtask or domain, while a gating network dynamically activates the most appropriate expert based on the current input (Figure 7.20).

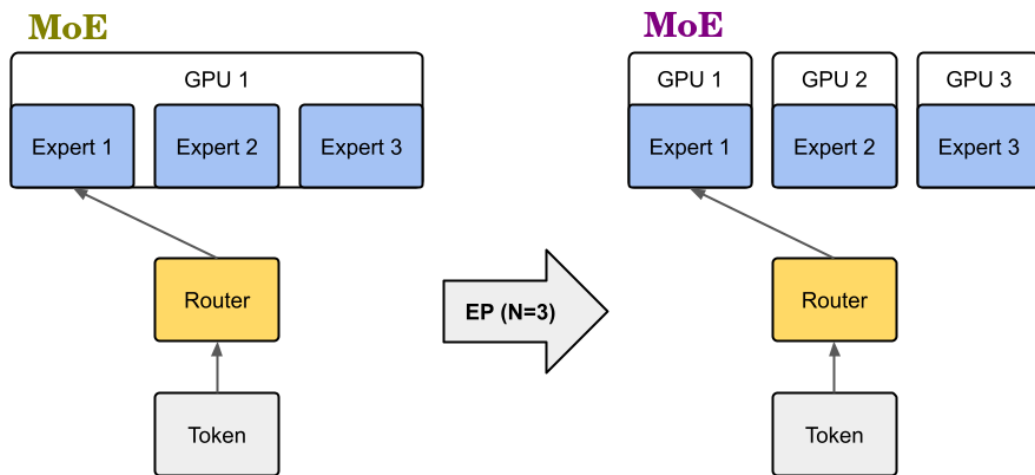


Figure 7.20: Expert parallelism distributes MoE experts across multiple GPUs while keeping other layers replicated [42].

MoE configuration

To enable MoE in Megatron Bridge, we have to configure the essential hyperparameters of the MoE model, the number of experts, the number of experts activated per token, and the hidden size of expert feed-forward network, as illustrated in Listing 7.1.


```

1 from megatron.bridge.models import GPTModelProvider
2
3 # Configure basic MoE model
4 model_config = GPTModelProvider(
5     num_moe_experts=8,           # Number of experts in the MoE module
6     moe_router_topk=2,          # Number of experts activated per token
7     moe_ffn_hidden_size=8192,   # Hidden size of expert feed-forward networks (FFN)
8     # ... other model hyperparameters ...
9 )

```

Listing 7.1: Configure the MoE model [42].

Enable expert parallelism

To enable expert parallelism, we must set `expert_model_parallel_size` in our model configuration. For example, if the model has eight experts (`num_moe_experts=8`), then setting `expert_model_parallel_size=4` results in each GPU processing two experts. The number of experts should be divisible by the expert parallel size (Listing 7.2).

```

1 # Configure MoE model with expert parallelism
2 model_config = GPTModelProvider(
3     num_moe_experts=8,
4     expert_model_parallel_size=4, # Distribute 8 experts for 4 GPUs (2 experts per GPU)
5     # ... other model hyperparameters ...
6 )

```

Listing 7.2: Enable expert parallelism in the MoE model [42].

Enable expert tensor parallelism

To enable expert tensor parallelism (ETP), we must set `expert_tensor_parallel_size` in our model (Listing 7.3).

```

1 model_config = GPTModelProvider(
2     num_moe_experts=8,
3     expert_model_parallel_size=4,
4     expert_tensor_parallel_size=2, # Apply tensor parallelism within each expert
5     # ... other model hyperparameters ...
6 )

```

Listing 7.3: Enable expert tensor parallelism in the MoE model [42].

DeepEP and HybridEP token dispatch optimizations

DeepEP and HybridEP are high-performance MoE token dispatchers that improve throughput and efficiency on specific GPU architectures, in particular (i) DeepEP is optimized for Ampere, Hopper, B200, and B300 GPUs, and (ii) HybridEP, is optimized for GB200, GB300 with NVL72, and Ampere, Hopper, B200, B300 GPUs. These dispatchers replace the standard token routing mechanism with an optimized flexible dispatcher that provides better performance for MoE workloads (Listings 7.4 and 7.5).

```

1 from megatron.bridge.models import GPTModelProvider
2 from megatron.bridge.training.flex_dispatcher_backend import apply_flex_dispatcher_backend
3
4 model_config = GPTModelProvider(
5     num_moe_experts=8,
6     expert_model_parallel_size=4,
7     # ... other model hyperparameters ...
8 )
9
10 # Apply DeepEP optimization
11 apply_flex_dispatcher_backend(model_config, moe_flex_dispatcher_backend="deepep")

```

Listing 7.4: Apply DeepEP token dispatcher optimization [42].

```

1 from megatron.bridge.models import GPTModelProvider
2 from megatron.bridge.training.flex_dispatcher_backend import apply_flex_dispatcher_backend
3
4 model_config = GPTModelProvider(
5     num_moe_experts=8,
6     expert_model_parallel_size=4,
7     # ... other model hyperparameters ...
8 )
9
10 # Apply HybridEP optimization
11 apply_flex_dispatcher_backend(model_config, moe_flex_dispatcher_backend="hybridep")

```

Listing 7.5: Apply HybridEP token dispatcher optimization [42].

DeepSeek expert parallelism (**DeepEP**) is an all-to-all communication primitive that replaces the regular NCCL operation. DeepEP rests on three pillars: (i) dedicated streaming multiprocessor (SM) allocation, where a fixed subset of SMs, for example 20 SMs, is reserved to aggressively push network traffic to the limit and decouple routing from the main computation, (ii) permutation fusion, which combines token permutation, that is, gathering tokens for each expert, and token unpermutation directly to maximize memory bandwidth utilization, (iii) support for toggling between normal, high-throughput prefill, and low latency, decode workloads with cuda graph compatibility, modes.

Hybrid expert parallelism (**HybridEP**) combines intra-node NVLink topologies with cross-node communication schemes, making it highly efficient for multi-node systems based on Grace and Blackwell GPUs. It is an hierarchical dispatch strategy that utilizes NVLink-C2C and NVLink domains locally, offloading network traffic to reduce global streaming multiprocessors usage compared to pure DeepEP alternative. It also imposes a reduced expert parallelism overhead by leveraging NVSHMEM-based primitives, which brings measurable throughput improvements, up to 14% on DeepSeek-V3 and 9.9% on MXFP8 precision configurations compared to DeepEP.

Token dropping for load balancing

Token dropping allows for balancing work across experts through the capacity factor mechanism. This feature allows the model to drop tokens when experts are overloaded, preventing stragglers and improving overall throughput. To apply the token dropping mechanism, we must use the `apply_moe_token_drop` function, as indicated in Listing 7.6.

```

1 from megatron.bridge.models import GPTModelProvider
2 from megatron.bridge.training.utils.moe_token_drop import apply_moe_token_drop
3
4 model_config = GPTModelProvider(
5     num_moe_experts=8,
6     moe_router_topk=2,
7     moe_token_dispatcher_type="alltoall",          # Required for token dropping
8     moe_router_load_balancing_type="aux_loss",    # Required load balancing type
9     # ... other model hyperparameters ...
10 )
11
12 # Apply token dropping with the help of the capacity factor
13 apply_moe_token_drop(
14     model_config,
15     moe_expert_capacity_factor=1.0,                # Capacity multiplier per expert
16     moe_pad_expert_input_to_capacity=True,        # Pad inputs to capacity length
17 )

```

Listing 7.6: Control token dropping to improve load balancing [42].

The arguments of `apply_moe_token_drop` are (i) `moe_expert_capacity_factor`, which controls the maximum number of tokens each expert can process. A factor of 1.0 means that tokens are uniformly assigned to all experts. Lower values, such as 0.8, means that there will be more dropped tokens but improves load balancing; (ii) `moe_pad_expert_input_to_capacity`, when enabled, each expert input is padded to the capacity length for homogeneous minibatch sizes.

The requirements to control token dropping are (i) token dispatcher must be `alltoall` or `alltoall_seq`, and (ii) load balancing type must be `aux_loss`, `seq_aux_loss`, or `None`. Token dropping can improve training throughput by 10-30% in imbalanced MoE models, but may affect convergence if it is too aggressive. Thus, we must start with a capacity factor of 1.0 and gradually reduce it if necessary.

Listing 7.7 aggregates all these facilities allowed by Megatron Bridge to optimize an MoE model.

```

1 from megatron.bridge.models import GPTModelProvider
2 from megatron.bridge.training.config import ConfigContainer
3 from megatron.bridge.training.flex_dispatcher_backend import apply_flex_dispatcher_backend
4 from megatron.bridge.training.utils.moe_token_drop import apply_moe_token_drop
5
6 # Configure the MoE model with expert parallelism
7 model_config = GPTModelProvider(
8     # Transformer hyperparameters
9     num_layers = 32,
10    hidden_size = 4096,
11    num_attention_heads = 32,
12
13    # MoE hyperparameters
14    num_moe_experts=8, # 8 experts on total
15    moe_router_topk=2, # 2 activate experts per token
16    moe_ffn_hidden_size=8192, # Expert FFN hidden dimension
17    moe_token_dispatcher_type="alltoall", # Token dispatcher type
18    moe_router_load_balancing_type="aux_loss", # Load balancing via auxiliary loss
19
20    # Expert parallelism
21    expert_model_parallel_size = 4, # Distribute experts across 4 GPUs
22    expert_tensor_parallel_size = 2, # Apply tensor par. within each expert
23
24    # ... other model hyperparameters ...
25 )
26
27 # Apply DeepEP token dispatch optimization
28 apply_flex_dispatcher_backend(model_config, moe_flex_dispatcher_backend="deepep")
29
30 # Control token dropping to improve load balancing
31 apply_moe_token_drop(
32     model_config,
33     moe_expert_capacity_factor=1.0,
34     moe_pad_expert_input_to_capacity=True,
35 )
36
37 config = ConfigContainer(
38     model=model_config,
39     # ... other configuration hyperparameters ...
40 )

```

Listing 7.7: Configure an optimized MoE model in Megatron Bridge [42].

7.9 Elastic Expert Parallelism in vLLM

Expert parallelism is a fundamental approach for efficiently serving Mixture-of-Experts models, particularly when aiming for high throughput. Large-scale deployments, often referred to as WideEP, distribute experts across many workers, significantly increasing KV cache capacity. This expanded capacity supports either very high request concurrency or extremely long context windows. Such properties are especially important in reinforcement learning scenarios, which often require both long sequences and high throughput, as well as in agent-based applications where multi-turn interactions naturally lead to extended context lengths.

In vLLM and similar inference systems, EP has traditionally been configured statically. Once a deployment is launched, its capacity remains fixed. This rigidity creates practical limitations: when demand increases beyond system capacity, the deployment cannot dynamically scale up, and when demand drops, resources cannot be

reduced to save costs. The only solution in such cases is to restart the service with a new configuration, which introduces latency and may disrupt active requests or lead to dropped traffic.

Elastic expert parallelism (Elastic EP) addresses these limitations [4]. It allows vLLM deployments to adjust the number of active workers during runtime, enabling the system to scale resources up or down in response to changing workloads with minimal service interruption.

Scaling is achieved by modifying the number of data-parallel (DP) workers. In vLLM, this affects both the size of the expert-parallel group and the way experts are assigned across devices. The adjustment can be triggered easily through a single API call such as the following one.

```
curl -X POST http://localhost:8000/scale_elastic_ep
-H "Content-Type: application/json"
-d '{"new_data_parallel_size": 8'
```

This command dynamically updates the deployment to use eight data-parallel workers, without requiring a system restart. Elastic expert parallelism scale-up, where a new GPU joins an active deployment, expanding the EP group without restarting existing workers, is illustrated in Figure 7.21.

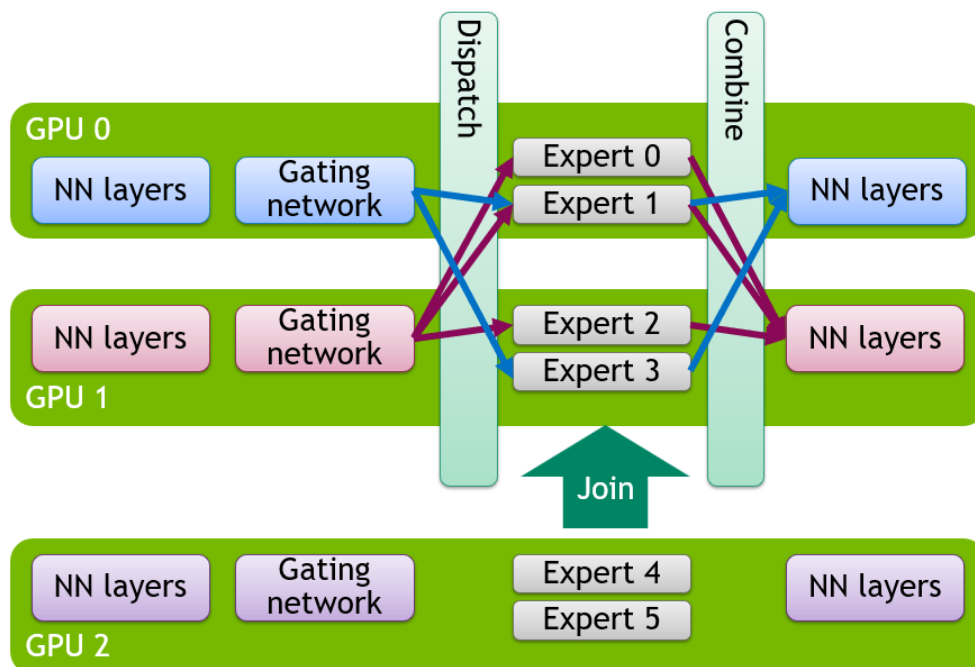


Figure 7.21: Elastic expert parallelism scale-up: a new GPU joins an active deployment, expanding the EP group without restarting existing workers [4].

This section outlines the design of Elastic EP in vLLM, including the mechanisms for scaling both up and down, coordination with ongoing inference requests, and interactions with load balancing and communication layers, such as expert parallelism load balancer (EPLB). Additionally, it highlights the relevance of this capability for improving fault tolerance in serving systems. In particular, the NIXL EP backend is discussed as a promising option, as its communication design supports efficient reconfiguration and enables features such as failure detection, reporting, and recovery.

Elastic EP enables dynamic scaling of MoE deployments in vLLM by adjusting the number of data-parallel workers at runtime, eliminating the need for restarts. Scaling operations are triggered via a simple API call (POST /scale_elastic_ep), allowing live reconfiguration and redistribution of experts. This capability forms an important foundation for building fault-tolerant serving systems in vLLM. The NIXL EP backend further enhances this design by reducing reinitialization overhead and providing mechanisms for detecting and recovering from failures.

7.9.1 Expert Parallelism and Data Parallelism Attention

In Mixture-of-Experts models, attention layers remain fully dense, where some of the feed-forward layers are replaced by sparse experts, where a subset of these experts will be dynamically selected to process each token. Before introducing elastic scaling, it is useful to first summarize the two underlying parallelization strategies that support it.

Data-parallel attention operates at the request level: each computing core processes a distinct subset of incoming requests and maintains its own scheduler and KV cache. This approach is particularly advantageous in architectures such as multi-head latent attention (MLA), where tensor parallelism (TP) would otherwise replicate the KV cache across multiple GPUs, leading to inefficient memory usage and reduced effective batch sizes.

Expert parallelism, in contrast, is applied specifically to the expert layers. Rather than dividing individual experts across devices, each expert is assigned to specific GPUs, and tokens are routed only to the devices hosting the corresponding experts.

Within vLLM, attention computations are performed independently on each data-parallel worker, while expert layers are executed collectively through a shared expert-parallel group spanning those workers. The size of this EP group is determined by the product of the data-parallel and tensor-parallel dimensions. With Elastic EP, adjusting the number of DP workers at runtime directly changes the EP group size, allowing experts to be redistributed dynamically across devices.

7.9.2 What Needs to Change When the Available Computing Resources Change?

Adjusting the degree of data parallelism at runtime involves more than simply starting or stopping worker processes. Changing the size of the expert-parallel group impacts several critical components of the system's execution state, defined by the following information.

- Communication groups: Distributed groups such as EP, DP, and the global world group rely on fixed rank memberships. Any change in parallelism requires rebuilding these groups with updated rank assignments.
- Expert placement: When the EP configuration is modified, the mapping of experts to specific devices must be recomputed to reflect the new topology.
- Model weights: Newly added devices need to receive the appropriate model weights, while existing devices may require updates due to expert redistribution.
- Runtime optimizations: Techniques such as `cuda graph capture` and `torch.compile` depend on static execution assumptions. These optimizations must be regenerated when the system configuration changes.

To safely handle these updates, scaling is implemented as a coordinated state machine. The process is divided into well-defined stages, each with explicit synchronization points. These synchronization barriers are carefully designed to interact correctly with ongoing forward passes, ensuring consistent and stable execution during reconfiguration.

7.9.3 Scale-Up Flow

Increasing the number of data-parallel workers from N to M , where $M > N$, is inherently more involved than scaling down, since additional ranks must be introduced into an already running system.

1. Trigger and request handling

The process is initiated through the `/scale_elastic_ep` endpoint. If the flag `VLLM_ELASTIC_EP_DRAIN_REQUESTS=1` is enabled, the system first waits for all ongoing requests to complete, with a maximum wait time defined by the `drain_timeout` parameter, which is 120 seconds by default. If this option is disabled, the scaling operation begins immediately without waiting for in-flight requests to finish.

2. Initialization of new engine cores

The creation of additional computing workers is handled by the Ray-based data-parallel backend. During scaling, Ray provisions the required number of new DP workers on available GPUs to match the target configuration. These newly created ranks are initialized using the current expert layout but start with placeholder weights. They remain in a waiting state until subsequent phases complete weight synchronization and topology update.

The readiness of these workers is coordinated through a two-step signaling mechanism. The first signal enables existing ranks to construct standby communication groups, while the second signal allows the system to proceed with weight distribution and integration.

3. Standby communication groups

Rather than immediately replacing the active communication setup, vLLM first constructs standby communication groups that reflect the updated set of ranks. These groups are instantiated using `StatelessGroupCoordinator`, which operates independently of PyTorch's global `WORLD` communicator. This design allows the new configuration to be prepared in advance, while the existing communication groups continue to handle ongoing inference. As a result, forward execution can proceed without interruption during the transition.

When using the `nixl_ep` backend, this process can be made more efficient through incremental updates. Instead of rebuilding all expert-parallel connections, vLLM can dynamically adjust membership by invoking the `connect_ranks()` and `disconnect_ranks()` functions. This enables seamless integration of new ranks while preserving existing connections.

4. Expert mapping and weight transfer

After the standby communication groups are established, they are used to distribute the current expert assignment and to copy non-expert model weights from existing ranks to the newly added ones. This transfer is balanced across the existing ranks to distribute the workload efficiently. Elastic EP leverages the same GPU-to-GPU communication mechanisms used by EPLB for moving expert weights, but extends this process to include attention layers, normalization layers, embeddings, and other non-expert modules. These transfers take advantage of high-speed interconnects such as NVLink within a node and RDMA across nodes.

At this stage, expert weights are intentionally not migrated. Their redistribution is deferred to a later step handled by EPLB once the updated topology is activated. During this transition phase, regular EPLB operations are temporarily suspended to prevent interference with the reconfiguration process.

5. The switch

The switch represents the transition point where all ranks move from the previous configuration to the updated one. During this phase, vLLM performs the following coordinated actions.

- Clears existing cuda graph captures and resets any `torch.compile` state.
- Activates the standby communication groups as the new EP, DP, and global communication groups.
- Removes the previously active communication groups.
- Updates MoE layers to reflect the new expert-parallel size.
- Reinitializes execution paths to regenerate cuda graphs and compiled kernels aligned with the new configuration.

In addition, runtime coordination variables, such as execution flags, wave counters, and step trackers, are synchronized across the updated DP group to ensure all ranks resume from the same state.

At this point, newly added ranks become fully active participants in the DP group and can execute forward passes, including attention computations. However, they do not yet host expert weights. The reassignment of experts is handled in the subsequent stage.

6. Expert parallelism load balancer reshuffle

Once the new configuration is active, EPLB redistributes expert assignments across all M ranks. This phase updates the expert mapping and transfers the necessary expert weights to their new locations. After this redistribution completes, EPLB resumes its normal execution.

7.9.4 Scale-Down Flow

Reducing the number of data-parallel workers from M to N follows a process similar to scaling up, but with a key distinction: the redistribution of expert ownership must occur first. Before any ranks are removed, all M engine cores participate in an EPLB reshuffle. This step ensures that all expert weights currently hosted on ranks scheduled for removal are migrated to the remaining N active ranks. As a result, once the scale-down proceeds, no departing rank retains ownership of expert weights.

7.9.5 Coordinating Reconfiguration Steps Across DP Ranks

A challenge in dynamic reconfiguration arises from the asynchronous nature of data-parallel computing cores. Different ranks may receive the scaling signal at slightly different times. Consequently, while some ranks may be ready to proceed to the next reconfiguration stage, others may still be completing an additional forward pass. If the early ranks were to advance prematurely, the system could become inconsistent, with some ranks reconfiguring while others continue executing, potentially causing a deadlock.

To prevent this, Elastic EP employs a two-phase synchronization mechanism. The first phase uses a barrier with a timeout: if not all ranks reach the barrier within the allowed time, it is assumed that some ranks are still processing an extra iteration. In this case, the ranks that arrived early return to the execution loop and perform another step. On the next attempt, once all ranks reach the synchronization point simultaneously, a second barrier, without a timeout fallback, ensures that all ranks transition to the next stage in a coordinated manner.

7.9.6 Fault Tolerance

Elastic EP plays a fundamental role in enabling fault-tolerant behavior in vLLM, as it provides the mechanism required to dynamically reconfigure the system after failures occur. When a worker (rank) fails, Elastic EP allows the system to adapt without requiring a full restart. Specifically, it supports removing the failed rank, redistributing its associated experts, and later restoring capacity by adding replacement resources.

The recovery process can be summarized in three main steps:

- Failure detection: The system identifies failures through health monitoring mechanisms or signals provided by the underlying communication backend.
- Scale-down: The failed rank is removed from the deployment, and its expert assignments are redistributed among the remaining active ranks.
- Scale-up: Once new resources become available, additional ranks are introduced to restore the original capacity.

The NIXL EP backend further supports this process by providing built-in capabilities for detecting failures at the expert-parallel level, reporting issues, and re-establishing communication with newly added ranks. This makes it particularly well-suited for scenarios requiring robust and adaptive recovery mechanisms.

7.9.7 An Example

Launch with elastic EP enabled

Our example uses DeepSeek-V2-Lite-Chat as a small MoE model. The current implementation targets Ray data-parallel deployments with `tensor_parallel_size=1`, one API server, and without dual batch overlap (DBO). DBO is a microbatching and inference acceleration strategy used in vLLM.

```
vllm serve deepseek-ai/DeepSeek-V2-Lite-Chat
  -trust-remote-code
  -tensor-parallel-size 1
  -data-parallel-size 2
  -data-parallel-backend ray
  -api-server-count 1
  -enable-expert-parallel
  -enable-elastic-ep
  -enable-eplb
  -eplb-config.num_redundant_experts 0
  -all2all-backend allgather_reducescatter
  -gpu-memory-utilization 0.8
```

Scale up and down at runtime

With the Ray data-parallel backend, adding capacity can be as simple as joining another node to the Ray cluster. Once Ray sees the new GPUs, Elastic EP can scale the deployment onto them at runtime.

For example, on a new worker node we run the following commands.

```
ray start -address="$HEAD_NODE_IP:6379"

curl -X POST http://localhost:8000/scale_elastic_ep
  -H "Content-Type: application/json"
  -d '{"new_data_parallel_size": 16'
```

To scale down at runtime, we run the next command.

```
curl -X POST http://localhost:8000/scale_elastic_ep
  -H "Content-Type: application/json"
  -d '{"new_data_parallel_size": 8'
```

7.10 Sequence Parallelism

Very long sequence training is an important direction in large language models training. In actual inference processes, especially in agent workflows, the model's generalization ability for long sequences and complex scenarios represents the model's reliability in practical applications. Long sequence scenarios also pose higher demands for large model training. Due to the $O(N^2)$ complexity characteristic of attention computation, when the actual input sequence grows, memory usage exhibits exponential explosion. This poses enormous challenges to the usability of memory-constrained accelerators in long sequence training scenarios.

Sequence parallelism (SP) can be used to reduce the dependency on large memory for long sequence training under multi-GPU or multi-node conditions. Sequence parallelism is a parallel strategy that partitions along the sequence dimension. Mature sequence parallelism methods include Megatron's sequence parallelism, DeepSpeed-Ulysses sequence parallelism, ring attention, and striped attention.

7.10.1 Megatron Sequence Parallelism

In [109], authors use a single stack transformer encoder or decoder with L layers, as shown in Figure 7.22. At the start of the network, the input tokens are fed into a word embedding table with size $v \times h$ and the token embeddings are combined with learned positional embeddings with size $s \times h$, where s is the sequence length, h is the hidden dimension, and v is the vocabulary size. The output of the embedding layer, which is the input to the transformer block, is a 3-D tensor of size $s \times b \times h$, where b is the minibatch size. Each transformer layer consists of a self-attention block with a attention heads followed by a multi-layer perceptron (MLP) with

two layers which increase the hidden size to $4h$ and then reduce it back to h . Input to and output from each transformer layer have the same size, $s \times b \times h$. The output from the last transformer layer is projected back into the vocabulary dimension to calculate the cross-entropy loss. In the explanation that follows, in addition to the hyperparameters already mentioned, we will use p to represent the pipeline-parallel degree and t the tensor-parallel degree.

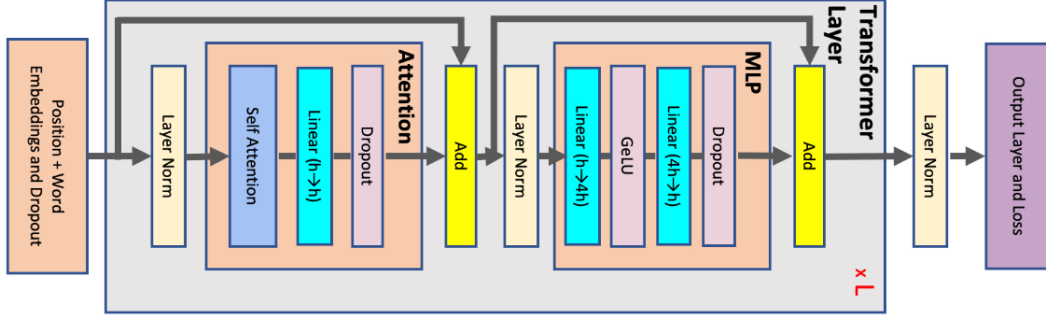


Figure 7.22: Transformer architecture. Each gray block represents a single transformer layer that is replicated L times [109].

Summing the memory required for attention, MLP, and the layer-norms, the memory required to store the activations for a single layer of a transformer network is given by Equation 7.9.

$$\text{Activations memory per layer} = s \cdot b \cdot h \cdot \left(34 + 5 \cdot \frac{a \cdot s}{h} \right) \quad (7.9)$$

Tensor parallelism, as shown in Figure 7.23, parallelizes the parts of the transformer layer that take the most time during training and as a result, it is computationally efficient. However, it leaves the layer-norms as well as the dropouts after attention and MLP blocks intact and as a result, they are replicated across the tensor parallel group. These elements do not require a lot of compute but demand a considerable amount of activation memory. When using tensor parallelism, the memory required for storing the activations of a single layer of the transformer is given by Equation 7.10. Quantitatively, the term $10 \cdot s \cdot b \cdot h$ in this equation is due to these replicated operations and as a result they are not divided by the tensor parallel degree t .

$$\text{Activations memory per layer} = s \cdot b \cdot h \cdot \left(10 + \frac{24}{t} + 5 \frac{a \cdot s}{h \cdot t} \right) \quad (7.10)$$

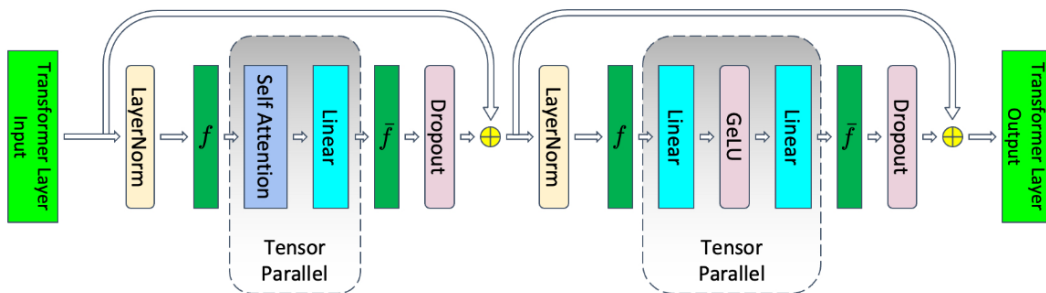


Figure 7.23: Transformer layer with tensor parallelism. f and $\bar{f}$ are conjugate. f is no operation in the forward pass and AllReduce in the backward pass. $\bar{f}$ is AllReduce in the forward pass and no operation in the backward pass [109].

We notice that in the non-tensor parallel regions of a transformer layer, the operations are independent along the sequence dimension. This characteristic allows us to partition these regions along the sequence dimension s . Partitioning along the sequence dimension reduces the memory required for the activations. This extra level of parallelism introduces new communication collectives before f and after $\bar{f}$ which will act as converters between sequence and tensor parallel regions. For example, in the forward pass, we need an extra AllGather before the operator f in Figure 7.23. These extra communications introduce overhead and will slow down the

training. To avoid these extra communications, we combine these operations with the f and $\bar{f}$ operators and introduce new operations g and $\bar{g}$ as shown in Figure 7.24. As it can be seen, g and $\bar{g}$ are the converters between sequence and tensor parallel regions. We derive these operations in the remainder of this section.

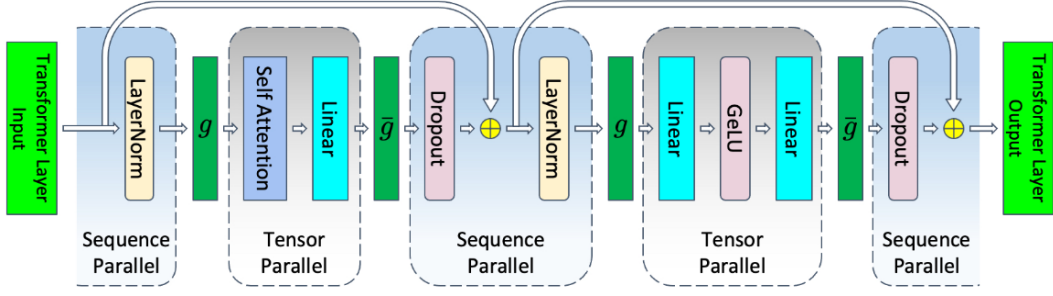


Figure 7.24: Transformer layer with tensor and sequence parallelism. g and $\bar{g}$ are conjugate. g is AllGather in the forward pass and ReduceScatter in the backward pass. $\bar{g}$ is ReduceScatter in forward pass and AllGather in backward pass [109].

We detail g and $\bar{g}$'s derivation using the MLP block. In the non-parallel form, as shown in Figure 7.22, the layer-norm followed by the MLP block can be formulated as indicated in Equation 7.11.

$$\begin{aligned} Y &= \text{LayerNorm}(X) \\ Z &= \text{GeLU}(Y \cdot A) \\ W &= Z \cdot B \\ V &= \text{Dropout}(W) \end{aligned} \quad (7.11)$$

where X is input to the layer-norm with size $s \times b \times h$ and A and B are the weight matrices of both Linear layers of the MLP, and have a size of $h \times 4h$ and $4h \times h$, respectively. The combined form of the tensor and sequence parallelisms present in the above operations is shown in Figure 7.25. The subscripts represent splitting among accelerators and superscripts depict the dimension along which the splitting is done. For example, X_1^s is the first accelerator's part of X that is split along the s dimension, sequence dimension, while Z_2^h is the second accelerator's part of Z that is split along the h dimension, hidden dimension.

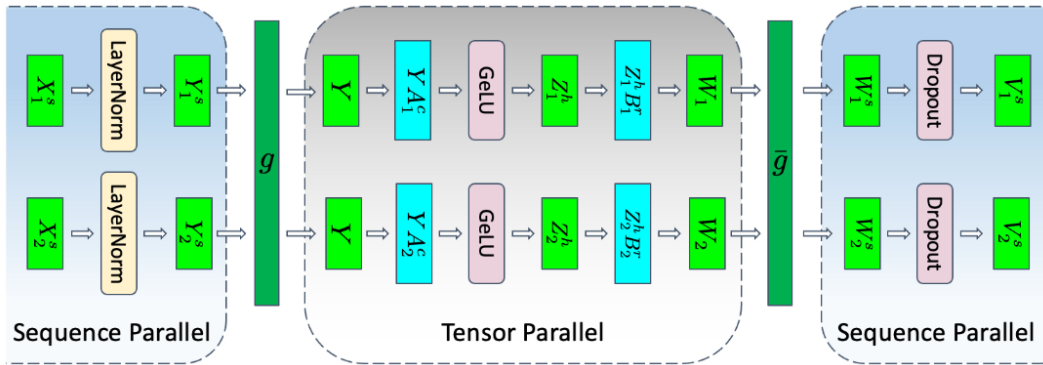


Figure 7.25: MLP layer with tensor and sequence parallelism. g and $\bar{g}$ are conjugate. g is AllGather in forward pass and ReduceScatter in backward pass. $\bar{g}$ is ReduceScatter in forward pass and AllGather in backward pass [109].

The input to the layer-norm is parallelized along the sequence dimension $X = [X_1^s, X_2^s]$. As a result, the output of the layer-norm will also be parallel along the sequence dimension $Y = [Y_1^s, Y_2^s]$. The Linear layer followed by the GeLU non-linearity require the entire input Y and therefore we need to perform an AllGather. This implies that g is an AllGather operation along the sequence dimension in the forward pass. By splitting A along its columns (A_1^c and A_2^c) and B along its rows (B_1^r and B_2^r), we avoid communications and arrive at W_1 and W_2 . These two tensors are not parallel anymore and need to be summed as $W = W_1 + W_2$ before they are fed into the dropout layer. However, dropout needs its input to be parallel in the sequence dimension s . Instead of summing and then parallelizing in the sequence dimension, we combine these two operations into a

ReduceScatter operation. As a result, $\bar{g}$ can be a single ReduceScatter operation in the forward pass. Putting it all together, we arrive at Equation 7.12.

$$\begin{aligned}
[Y_1^s, Y_2^s] &= \text{LayerNorm}([X_1^s, X_2^s]) \\
Y &= g(Y_1^s, Y_2^s) \\
[Z_1^h, Z_2^h] &= [\text{GeLU}(Y \cdot A_1^c), \text{GeLU}(Y \cdot A_2^c)] \\
W_1 &= Z_1^h \cdot B_1^r \text{ and } W_2 = Z_2^h \cdot B_2^r \\
[W_1^s, W_2^s] &= \bar{g}(W_1, W_2) \\
[V_1^s, V_2^s] &= [\text{Dropout}(W_1^s), \text{Dropout}(W_2^s)]
\end{aligned} \tag{7.12}$$

If we follow a similar break-down for the backward pass, we find that g and $\bar{g}$ are conjugate of each other. g is an AllGather in the forward pass and a ReduceScatter in the backward pass, and $\bar{g}$ is a ReduceScatter in the forward pass and an AllGather in the backward pass.

A similar breakdown can be applied to the layer normalization followed by the attention layer part of the transformer block included in Figure 7.24.

Tensor parallelism requires four AllReduces in a single forward and backward pass, whereas combined tensor and sequence parallelisms require four AllGathers and four ReduceScatters in a single forward and backward pass. At the first look, it seems that tensor with sequence parallelism requires more communications compared to tensor parallelism. However, we note that a ring AllReduce is composed of two steps: a ReduceScatter followed by an AllGather. As a result, the communication bandwidth used for tensor parallelism and tensor together with sequence parallelism are the same. Therefore, sequence parallelism does not introduce any communication overhead.

From Equation 7.12, sequence parallelism along with tensor parallelism divide all the activations required for the backward pass along the parallel dimension except for the tensor Y that is required for the first linear operation. To alleviate this issue, we do not store the full tensor Y for the backward pass. Instead, we store only the Y_i^s part on the i -th tensor parallel rank and perform an extra AllGather in the backward pass. To eliminate the latency introduced by this extra AllGather, we overlap this communication with the computation required to calculate gradients with respect to Y , and as a result, we reduce the overhead.

Using sequence parallelism along with tensor parallelism, the memory required to store the activations per transformer layer reduces from Equation 7.10 to Equation 7.13.

$$\text{Activations memory per layer} = s \cdot b \cdot h \cdot \left(\frac{10}{t} + \frac{24}{t} + 5 \cdot \frac{a \cdot s}{h \cdot t} \right) = \frac{s \cdot b \cdot h}{t} \cdot \left(34 + 5 \cdot \frac{a \cdot s}{h} \right) \tag{7.13}$$

This expression corresponds to Equation 7.9 divided by the tensor parallel size. This means that using tensor and sequence parallelism, we can distribute activations among the tensor parallel group and reduce the required memory by the tensor parallel degree t .

7.10.2 DeepSpeed-Ulysses

Figure 7.26 contains a sketch of the building blocks of a typical multi-head attention transformer architecture [199]. It consists of input sequences which are projected into queries (Q), keys (K) and values (V) embeddings. Q , K , and V are typically 3D tensors of size $N \times b \times d$, where N is sequence length, b is the minibatch size and d is the hidden dimension. The Q , K , V tensors are fed into the attention block, a central component of the transformer model. The output of attention layer is the input for the multi-layer perceptron (MLP), also known as position-wise feed-forward network, of the transformer architecture.

An attention layer followed by the MLP block are replicated multiple times to form an encoder, a decoder or an encoder-decoder transformer model.

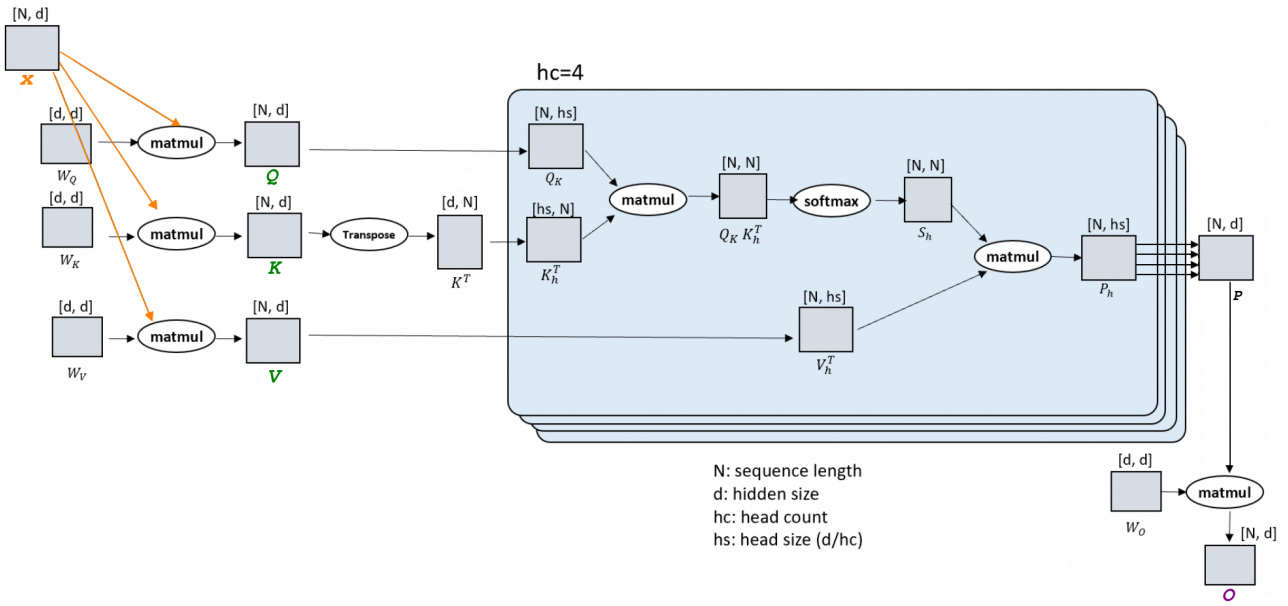


Figure 7.26: Multi-head attention transformer [93].

Figure 7.27 shows the core design of DeepSpeed-Ulysses. As occurs in the well known transformer architecture, the design consists of N input sequences partitioned across the P available devices. Each local $\frac{N}{P}$ partition is projected into queries (Q), keys (K) and values (V) embeddings. Next, Q , K , and V embeddings are gathered into global Q , K , and V through highly optimized All-to-All collective operations between participating compute devices. Sequel to All-to-All collective is the attention computation per head, formulated in Equation 7.14.

$$\text{output context} = \text{softmax}\left(\frac{Q \cdot K^T}{\sqrt{d}}\right) \cdot V \quad (7.14)$$

After the attention computation, another All-to-All collective operation transforms the output context tensor, produced by the attention computation, to sequence parallel ($\frac{N}{P}$) for being used by the subsequent operations, such as MLP matrix multiplication and layer normalization, in the remaining layers of the transformer block.

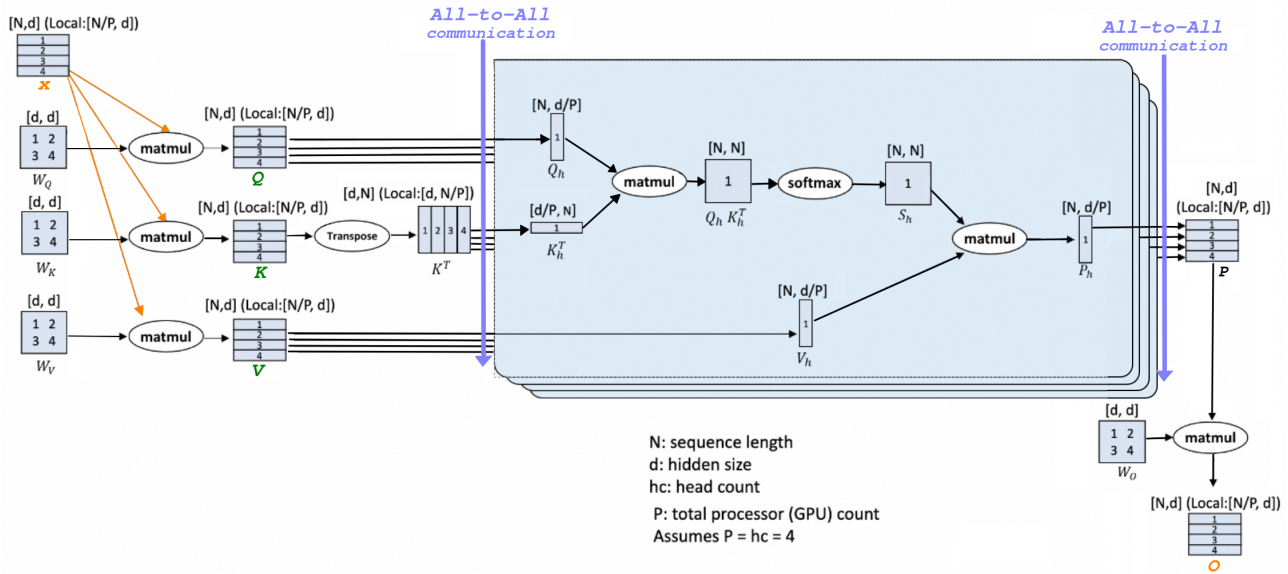


Figure 7.27: DeepSpeed-Ulysses sequence parallelism design [93].

Communication Analysis

What distinguishes DeepSpeed-Ulysses from the other existing long-sequence approaches is the much smaller aggregate communication volume and an overall better scalability with the increasing degree of sequence parallelism, compared to existing solutions, as demonstrated by the communication volume analysis that follows.

On modern clusters with intra-node NVSwitch interconnect and an inter-node fat tree IB topology, the communication volume transmitted per link for an All-to-All that aggregates a message of size M over P GPUs is $\frac{M}{P}$. For a transformer model with an hidden size h , a sequence length N , and a parallelism degree P , DeepSpeed sequence parallelism performs All-to-All for the Q , K , and V projections with an aggregate message of size $3 \cdot N \cdot h$ before the attention computation, and another All-to-All to aggregate the output context projection with a size $N \cdot h$ for each transformer layer. Therefore, DeepSpeed sequence parallelism incurs an aggregate communication volume per link of $\frac{4 \cdot N \cdot h}{P}$, or with the complexity of $O(N/P)$. Note that this communication volume is constant when both N and P are increased proportionally.

In contrast, the existing approaches like Megatron-LM incur a communication volume that increases linearly with N , regardless of P , resulting in the communication complexity of $O(N)$. For instance, Megatron-LM performs two AllGathers with a message volume of $N \cdot h$ and two ReduceScatters with a volume of $N \cdot h$ for each transformer block. However, the cost of each AllGather and ReduceScatter, with size M , remains M when $P \gg 1$, instead of $\frac{M}{P}$. Therefore, Megatron-LM sequence parallelism incurs a communication volume per link of $4 \cdot N \cdot h$ which is P times larger than that for DeepSpeed sequence parallelism. This allows DeepSpeed sequence parallelism to enable training with extremely long sequences while achieving significantly higher training efficiency compared to the existing approaches. Our evaluation results match this analysis.

Memory Efficiency

While DeepSpeed sequence parallelism reduces the activation memory when training with longer sequences, it does not impact the memory consumed by the model states. Therefore, to support large sequence length training with a large language model, DeepSpeed sequence parallelism is integrated with ZeRO-3. As we discussed in Section 3.7, ZeRO is a memory optimization technique for training large models [164] [165]. Unlike the classic data parallel training of neural networks, where model states are replicated across data parallel ranks, ZeRO-3 optimizes memory usage by partitioning model states across data parallel ranks. However, with sequence parallelism, training data can be considered in both batch (sample) and sequence dimensions and the associated parallel groups combined to form a larger group for ZeRO parallelism. Therefore, ZeRO-3 partitioning was extended to combine data parallel and sequence parallel ranks. In other words, in DeepSpeed sequence parallelism, ZeRO partitions the model states across both sequence and data parallel groups, and collects rank partitions (AllGather) when they are needed. Similarly, gradients are reduced across both data and sequence parallel ranks for weight updates. ZeRO support allows for huge memory savings in both sequence and data dimensions and enables scaling not just to large sequence lengths but also to large models.

General and Attention Agnostic Solution

DeepSpeed implementation of distributed attention is general enough to support any attention: self-attention, cross-attention, causal attention in both their dense and sparse versions, and their various optimized kernels that support long-sequence at local attention level, such as different versions of FlashAttention. The general property of DeepSpeed-Ulysses stems from the modular nature of its core design: an attention-centric sequence parallelism design. Prior to attention computation is sequence parallelism on $\frac{N}{P}$ partitions, where attention computation is head parallelism with full attention per head but just with fewer heads. Thus, attention computation can be replaced with any type of attention mechanism, for example, dense attention and various forms of sparse attention.

The main limitation of Ulysses is being constrained by the number of attention heads. Especially in grouped query attention, where the number of KV heads is much smaller than the number of Q heads, and Ulysses may not be able to split across more GPUs.

7.10.3 Ring Attention

Ring Attention [131] can be understood as a rethinking of sequence parallelism for long-context transformers. The core motivation stems from a fundamental limitation: although transformers are highly effective, their memory and computation costs scale poorly with sequence length due to the quadratic nature of self-attention and the need to store intermediate activations. This makes very long sequences, on the order of millions of tokens, impractical on a single device.

To alleviate this, prior work has explored sequence parallelism, where the input sequence is partitioned across multiple devices. While this reduces per-device memory, it introduces significant communication overhead. Because self-attention requires interactions between all tokens, devices must frequently exchange information, often through All-to-All communication or AllGather operations. These communications are costly and cannot be fully overlapped with computation, making them a key bottleneck that limits scalability. Some optimized implementations reduce the communication volume, but still require reconstructing large parts of the sequence or suffer from constraints such as dependence on the number of attention heads.

Ring Attention builds on the idea of distributing the sequence across devices but fundamentally changes how communication is performed. Instead of global communication patterns, it organizes devices into a logical ring. Each device holds a block of tokens, a query block, and attention is computed in a block-wise manner. The key-value (KV) blocks are not gathered centrally, instead, they are passed incrementally from one device to the next along the ring. This design is closely illustrated in Figure 7.28. The top part of the figure shows a ring of devices where each host keeps its local query block, while key-value blocks circulate from one device to the next. As a device performs attention computation on its query block and the current KV block, it simultaneously sends that KV block to the next device and receives a new one from the previous device. The bottom part of the figure reframes this process as a block-by-block computation on a standard transformer layer: the query block remains fixed on a device while the KV blocks are iterated over horizontally, and each pairwise interaction contributes incrementally to the final attention output before passing through the feed-forward network.

A crucial property that enables this approach is that block-wise attention is invariant to the order in which key-value blocks are processed, as long as their contributions are correctly accumulated. This allows KV blocks to be streamed in any order, making it possible to compute attention incrementally without ever assembling the full sequence on any device.

The most important benefit of this design is that communication is fully overlapped with computation. While each device computes attention for its query block, it is simultaneously sending and receiving KV blocks. If the computation time per block is greater than or equal to the communication time, then communication introduces effectively no additional overhead. This is a major departure from traditional sequence parallelism, where communication delays computation and becomes a limiting factor.

Another key advantage is memory efficiency. In standard approaches, memory usage often still scales with sequence length because large activation tensors must be stored or reconstructed. In contrast, Ring Attention requires each device to store only a small number of blocks (query, KV , and outputs), making its memory consumption dependent on block size rather than total sequence length. This removes the per-device memory bottleneck and allows arbitrarily long sequences to be processed.

As a consequence, Ring Attention achieves linear scaling of context length with the number of devices. If a single device can process a sequence of length s , then with n devices, Ring Attention enables processing sequences of length approximately $n \times s$. This level of scalability is not achievable with standard sequence parallelism due to its communication overhead.

Figure 7.29 provides the pseudocode of the Ring Attention algorithm. Ring Attention is compatible with existing code for memory efficient transformers, since it just needs to call whatever available memory efficient computation locally on each host, and overlap the communication of key-value blocks between hosts with block-wise computation. The implementation uses the collective operation `jax.lax.ppermute` to send and receive key value blocks between nearby hosts.

Conceptually, Ring Attention transforms sequence parallelism from a communication-heavy paradigm into a streaming computation model. Instead of repeatedly synchronizing large amounts of data, devices participate

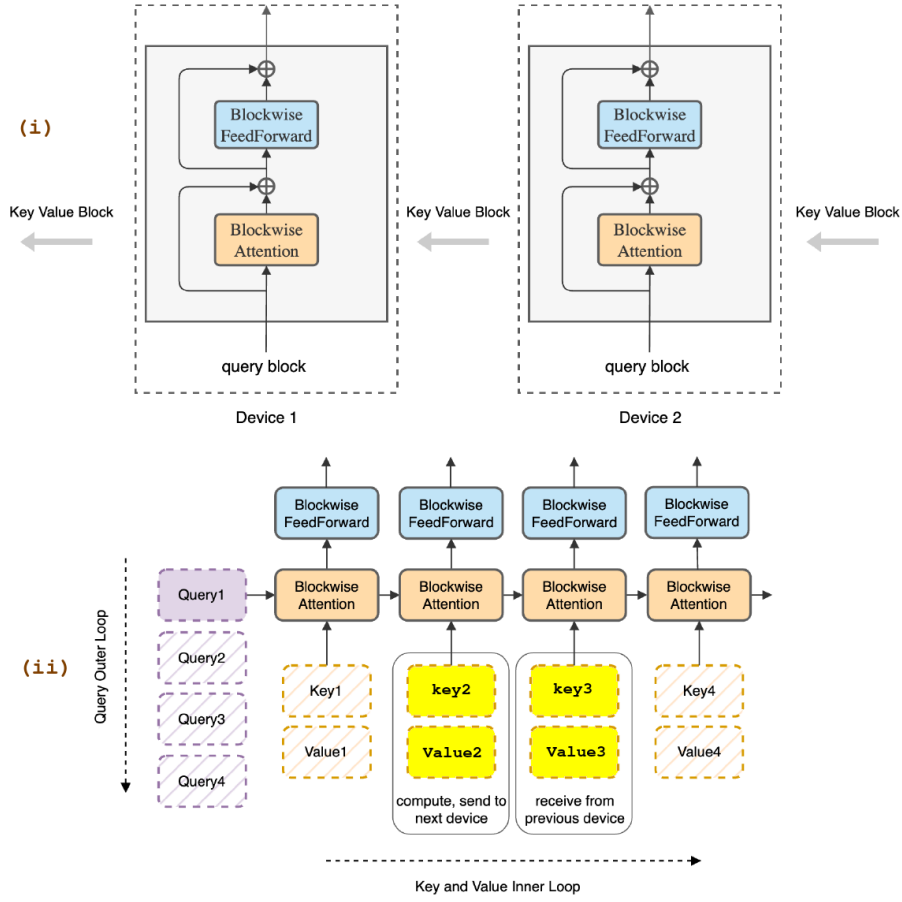


Figure 7.28: Ring attention sequence parallelism. (i) We use the same model architecture as the original transformer but reorganize the compute. In the diagram, we explain this by showing that in a ring of hosts, each host holds one query block, and key-value blocks traverse through a ring of hosts for attention and feed-forward computations in a block-by-block fashion. As we compute attention, each host sends its key-value blocks to the next host while it receives key-value blocks from the preceding host. The communication is overlapped with the computation of block-wise attention and feed-forward. (ii) We compute the original transformer block-by-block. Each host is responsible for one iteration of the query's outer loop, while the key-value blocks rotate among the hosts. As visualized, a device starts with the first query block on the left; then we iterate over the key-value blocks sequence positioned horizontally. The query block, combined with the key-value blocks, are used to compute self-attention (orange box), whose output is pass to feed-forward network (blue box) [131].

```

Required: Input sequence  $x$ . Number of hosts  $N_h$ 
Initialize
Split input sequence into  $N_h$  blocks that each host has one input block
Compute query, key, and value for its input block on each host
for each transformer layer do
  for  $count = 1$  to  $N_h - 1$  do
    for each host concurrently do
      Compute memory efficient attention incrementally using local query, key, value blocks
      Send key and value blocks to next host and receive key and value blocks from previous host
    end for
  end for
  for each host concurrently do
    Compute memory efficient feed-forward using local attention output
  end for
end for

```

Figure 7.29: Reducing transformers memory cost with ring attention [131].

in a continuous flow where key-value blocks circulate and are processed incrementally. This streaming mechanism, combined with full overlap of communication and computation, eliminates the main inefficiency of previous approaches. Importantly, this method preserves exact self-attention and does not rely on approximations, unlike many alternative approaches for long-context modeling. This makes it suitable for applications requiring precise long-range dependencies, such as long-document understanding, multimodal processing, and reinforcement learning with extended trajectories.

In conclusion, Ring Attention can be seen as an evolution of sequence parallelism. By replacing expensive global communication with ring-based streaming and by overlapping communication with computation, it removes the primary bottleneck that limits traditional methods. The result is a system that achieves memory efficiency, eliminates communication overhead, and scales context length linearly with the number of devices, enabling near-infinite context modeling in practice.

7.10.4 Striped Attention

Striped Attention [16] is presented as an extension and optimization of Ring Attention, designed specifically to improve performance in causal transformer models. The paper begins by situating itself in the broader effort to enable Transformers to process extremely long sequences. Ring Attention had already shown that it is possible to scale context length linearly with the number of devices by distributing attention computation across a ring of accelerators while overlapping communication and computation. However, this new work identifies a key inefficiency in Ring Attention when applied to causal self-attention, and proposes a simple but effective modification to address it.

The fundamental issue arises from the nature of causal attention. In causal transformers, each token only attends to previous tokens, which creates a triangular attention matrix where roughly half of the interactions are masked out. In theory, this structure should reduce computation by about a factor of two. Indeed, optimized single-device implementations exploit this by skipping the masked regions. However, Ring Attention does not benefit fully from this reduction. The reason lies in how the workload is distributed across devices.

Figure 7.30 illustrates the distinction between the original Ring Attention partitioning and the proposed Striped Attention scheme. In Ring Attention, each device is assigned a contiguous block of tokens. In contrast, Striped Attention redistributes tokens so that each device receives evenly spaced elements from across the entire sequence. Specifically, tokens are partitioned based on their position modulo the number of devices, so each device holds a "stripe" of the sequence rather than a contiguous chunk. This simple permutation has a profound effect on workload balance during attention computation.

To understand why, it is useful to recall how Ring Attention operates. As shown in Figure 7.31, each device keeps its query block fixed while key-value blocks are circulated around the ring over multiple iterations. On each iteration, every device computes attention between its query block and the current key-value block. For causal attention, a mask determines which interactions are valid. However, because each device has a contiguous block of tokens, the mask structure varies across devices: some may have entirely valid (unmasked) computations, while others may have entirely invalid (masked) ones. This leads to severe workload imbalance. Since the overall runtime is determined by the slowest device, which is the one performing full computation, the system cannot take advantage of the masked regions to speed up execution. As a result, even though half the attention matrix is theoretically unnecessary, Ring Attention effectively performs as if all computations were required.

Striped Attention resolves this inefficiency by changing only the data layout, not the overall algorithmic structure. After permuting the sequence as shown in Figure 7.30, the algorithm proceeds almost identically to Ring Attention. Devices are still arranged in a ring, and key-value blocks still circulate from one device to the next. However, because each device now holds tokens distributed throughout the sequence, the causal mask encountered during computation is no longer skewed.

This improvement is visualized in Figure 7.32, which shows how Striped Attention behaves under the same conditions as Figure 7.31. In this case, every device processes a mix of masked and unmasked interactions in each iteration. This ensures that all devices have roughly equal workloads, and importantly, that each device

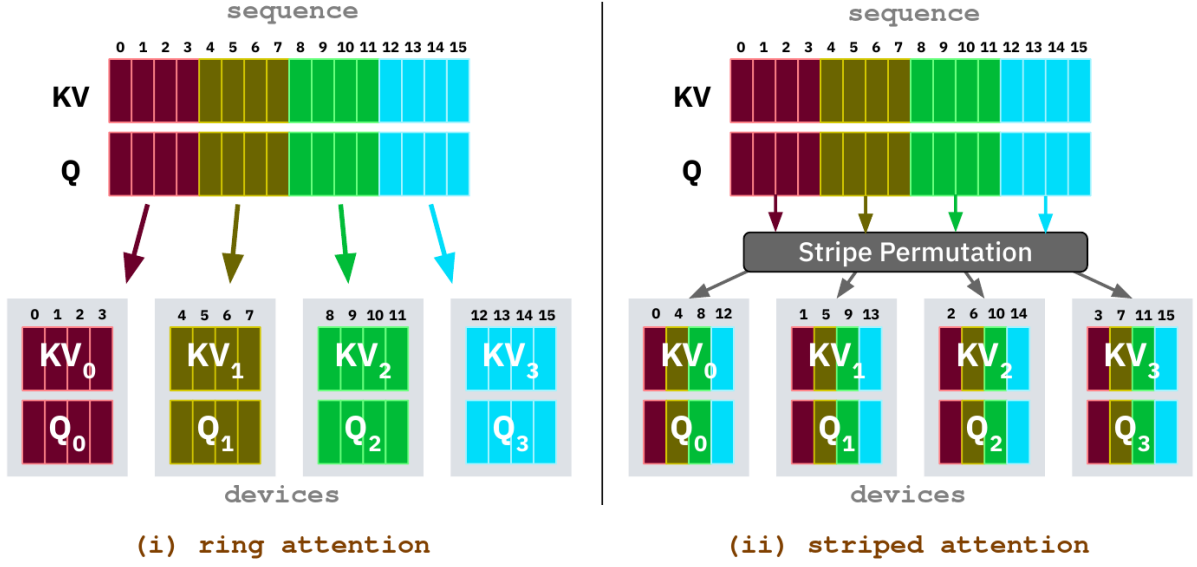


Figure 7.30: Initial partitioning of the Q , K , and V sequences into blocks for both ring attention and striped attention. Because they travel together in the ring attention algorithm, the K and V sequences are depicted as a single sequence. Note that for both ring attention and striped attention, the tokens in the input sequence are partitioned among devices before running the first layer of the model, and remain partitioned in the same layout throughout the forward and backward passes. As a result, Q , K , V are automatically partitioned in the desired layout at the beginning of each attention layer when using both ring attention and striped attention, with no extra per-layer communication required to prepare them in this state [16].

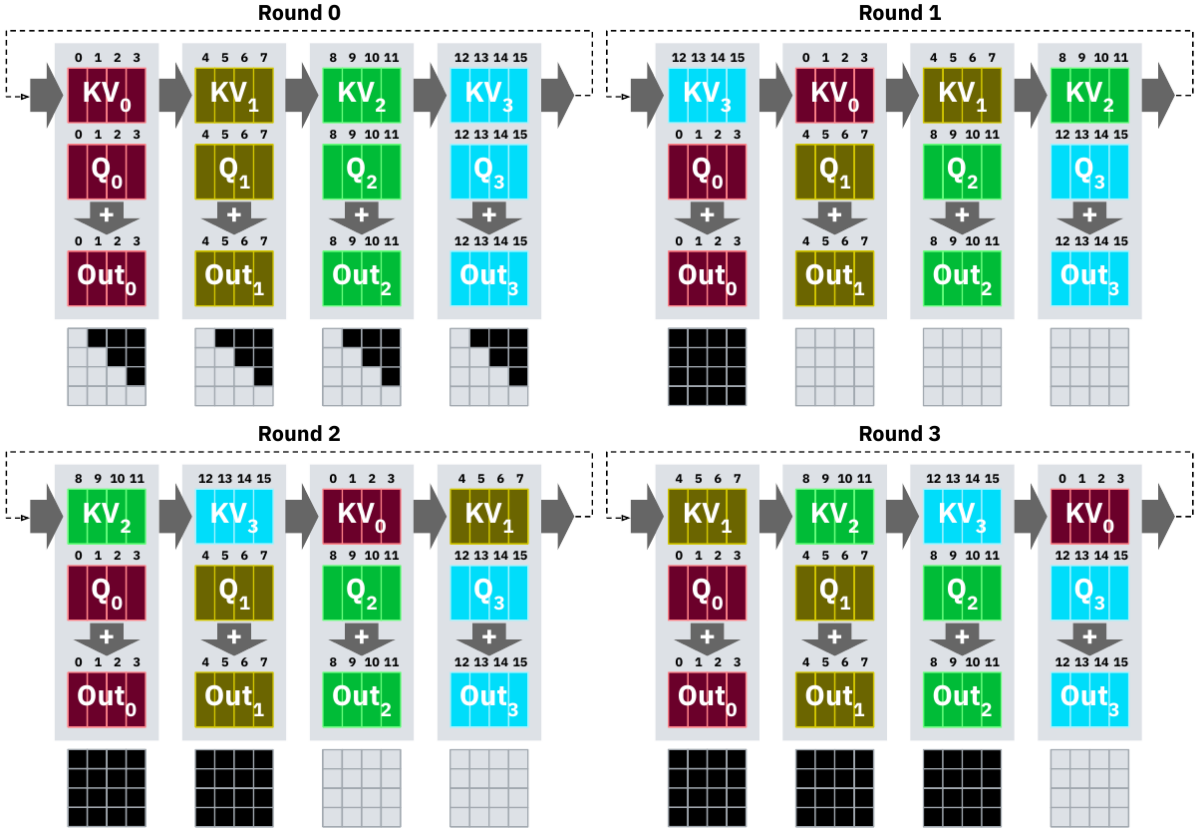


Figure 7.31: Behavior of ring attention as applied to a small causal self-attention problem with $n_{seq} = 16$, distributed across $N = 4$ devices. The Q blocks remain stationary, while K and V blocks pass from neighbor to neighbor in a circular fashion on each round. The square tile shown under each device on each round indicates the causal mask for the query/key interactions computed by that device on that round; elements masked out with values of $-\infty$ are indicated in black. We can see that on all but the first round, workload imbalances prevent us from making effective use of the structure of the causal mask to reduce run time [16].

can skip a comparable fraction of computations due to the causal mask. In contrast to Ring Attention, where only some devices benefit from masking, Striped Attention distributes this benefit evenly.

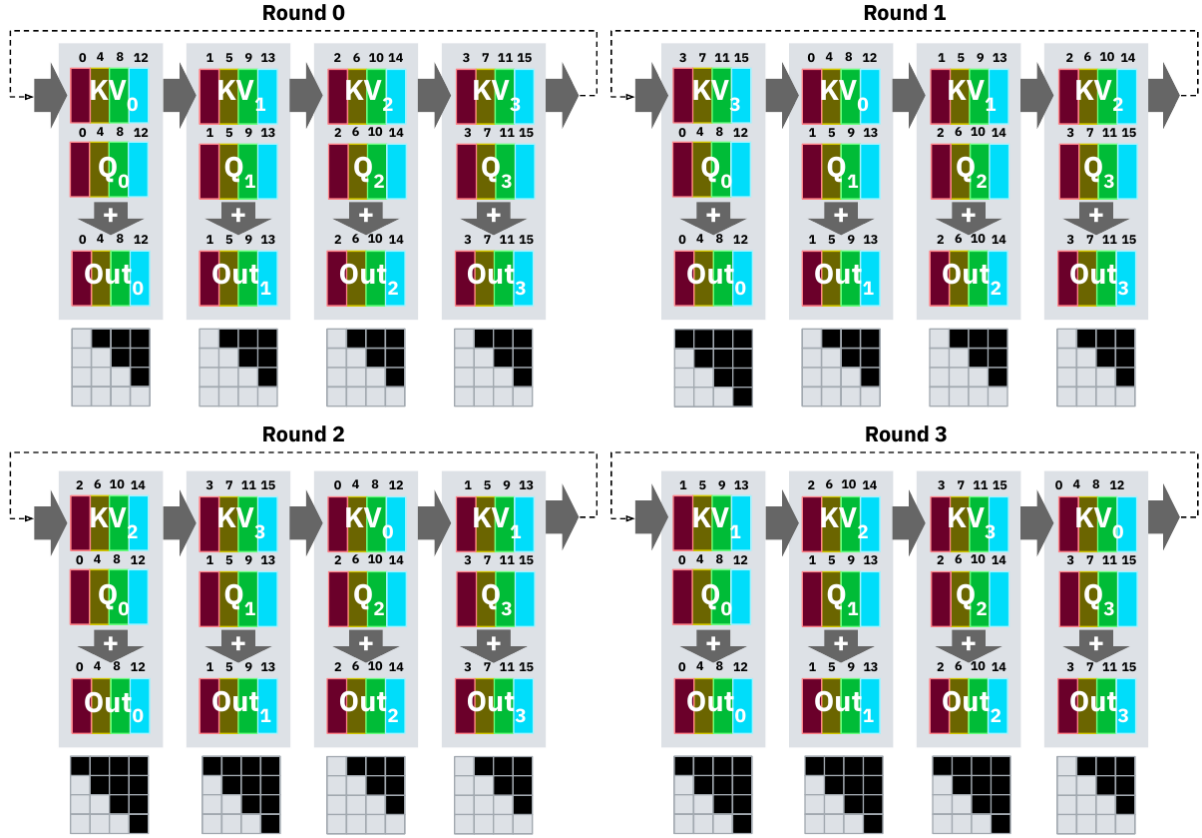


Figure 7.32: Behavior of striped attention as applied to the same causal self-attention problem as above. Unlike in ring attention, every block contains tokens distributed throughout every part of the original input sequence. As in Figure 7.31, the square matrices under each device depict the causal mask encountered by each device on each round. We note that the causal masks provide each device with a roughly equal portion of skipable work on each iteration, resolving the workload imbalance in ring attention [16].

The effect of this redesign becomes even clearer in Figure 7.33, which compares workload distributions between the two methods. In Ring Attention, certain devices may be assigned entirely masked workloads on some iterations, while others handle fully unmasked computations. This imbalance limits performance because the fast devices must wait for the slow ones. In Striped Attention, however, each device is assigned a balanced portion of the computation, with a similar mix of masked and unmasked operations. As a result, all devices progress at roughly the same rate, maximizing parallel efficiency and enabling consistent performance gains from skipping masked computations.

From a theoretical perspective, this modification allows Striped Attention to approach the ideal two times speedup associated with causal masking. While exact gains depend on hardware and implementation details, the paper shows that in practice, Striped Attention achieves substantial improvements. Experiments on GPUs and TPUs demonstrate speedups of up to 1.45 times and 1.65 times, respectively, at large sequence lengths. These gains become more pronounced as sequence length increases, since larger blocks allow more opportunities to skip masked computations.

Importantly, Striped Attention preserves all the advantages of Ring Attention. It remains an exact attention method, meaning it does not approximate or degrade the attention computation. It also retains the ability to scale context length linearly with the number of devices and continues to overlap communication with computation effectively. The only change is the permutation of the input sequence prior to partitioning, which is carefully designed to preserve correctness. Because attention is permutation-equivariant, this transformation does not alter the final output, as long as positional information and targets are permuted consistently.

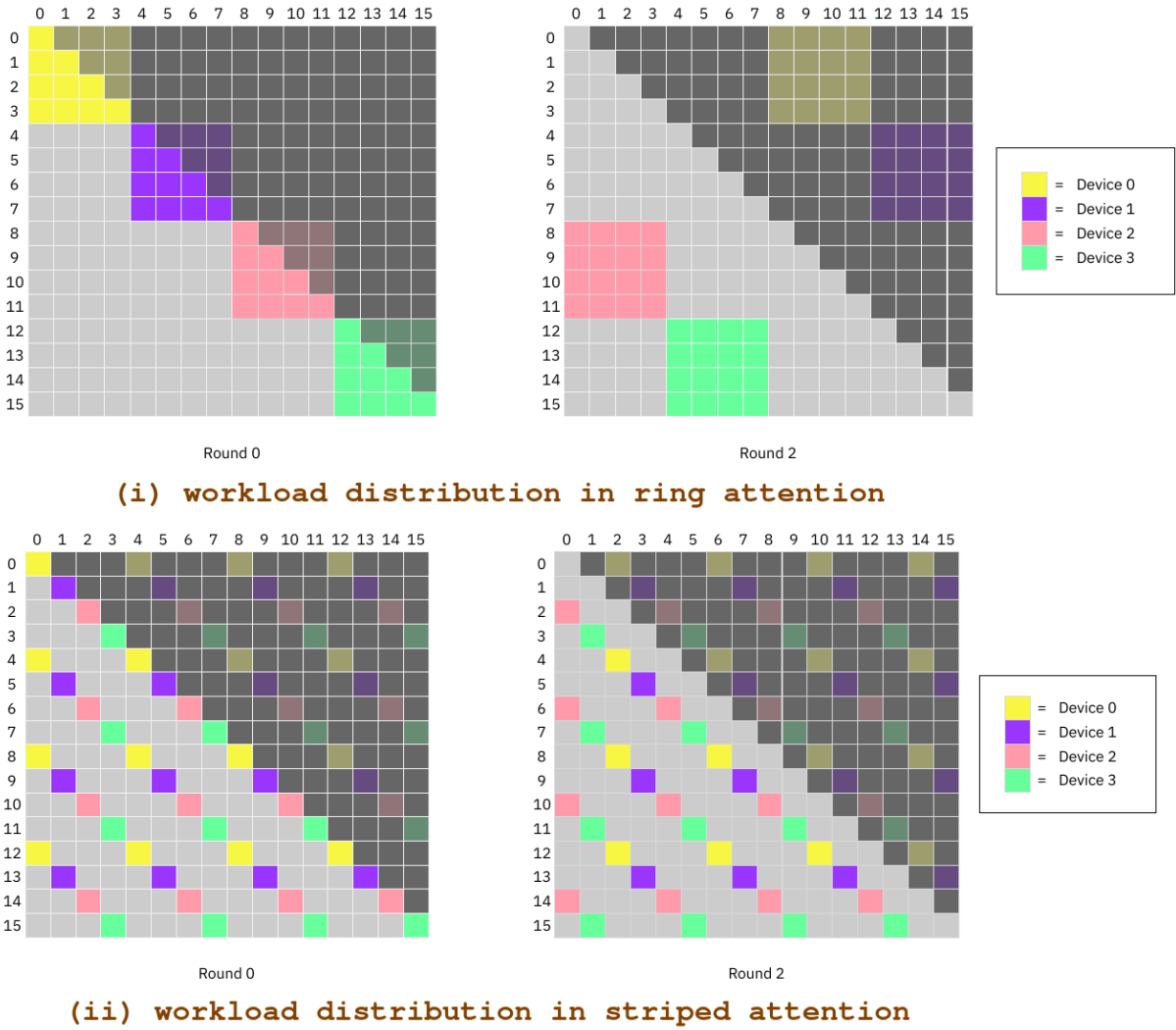


Figure 7.33: Workload distribution on rounds 0 and 2 of ring attention and striped attention. The square matrices represent the set of all possible pairwise query/key interactions; row indices correspond to queries, and column indices correspond to keys. All cells above the diagonal are masked out by the causal mask and can be skipped. The colors indicate which devices are responsible for which parts of the computation. On round 2, we can see that some devices in ring attention are responsible for workloads which are entirely masked out, whereas striped attention maintains a balanced workload across all devices [16].

The authors identified a subtle but important limitation of Ring Attention when applied to causal transformers: workload imbalance prevents it from exploiting the structure of causal masking. By introducing a striped data layout, the authors ensure that all devices perform balanced work and benefit equally from masked computation. The result powerful improvement that enhances throughput without changing the underlying algorithm or sacrificing accuracy, making it a valuable advancement for training long-context causal transformer models.

7.11 Implementation Example

In this section we will implement and train a simple convolutional neural network to classify CIFAR10 images, using a Mixture of Experts layer, with the help of DeepSpeed-MoE framework. We will consider expert parallelism and ZeRO memory optimizations. DeepSpeed provides two forms of MoE support: DeepSpeed MoE and Automatic Expert Parallelism (AutoEP). DeepSpeed MoE is available as the `deepspeed.moe.layer.MoE` class, which has the following arguments.

- `hidden_size` is the MoE's layer input and output dimension.
- `expert` is a PyTorch module that defines the MoE expert, for example, it can be an MLP or simply a Linear layer.
- `num_experts` defines the total number of experts per MoE layer.

- `ep_size` specifies the expert parallelism degree.
- `k` indicates the top-k gating value, where only `k=1` and `k=2` are supported.
- `capacity_factor` is the token capacity of the expert at training time.
- `eval_capacity_factor` is the capacity of the expert at evaluation time.
- `min_capacity` defines the minimum capacity per expert, regardless of the capacity factor.
- `use_residual` specifies if the MoE layer is a residual MoE (True) or a conventional MoE (False) layer.
- `noisy_gate_policy` allows us to specify the noisy gate policy, where the valid options are `Jitter`, `RSample`, and `None`.
- `drop_tokens` can be used to indicate whether to drop tokens (True) or not (False). If we set this argument to False, it is equivalent to define an infinite capacity.
- `use_rts` allows us to indicate whether to use random token selection (True) or not (False). The random token selection technique improves model training convergence.
- `use_tutel` can be used to specify whether to use Tutel optimizations (True) or not (False).
- `enable_expert_tensor_parallelism` to enable tensor parallelism in the expert layer (True) or not (False).
- `top2_2nd_expert_sampling` to indicate whether we want to perform sampling from the second expert (True) or not (False).

The argument of the forward method are (i) `hidden_states`, the input to the MoE layer and (ii) `used_token`, and optional tensor for masking only the used tokens. The method returns a tuple containing the layer output, the gate loss, and the expert count.

To build a Pyramid-Residual MoE (PR-MoE) layer, which in reality includes more than one basic layer, we have (i) to provide a list, such as `[4, 8]`, for the `num_experts` argument and (ii) to set the `use_residual` argument to True, to indicate that the MoE layer is a residual MoE layer. This is exemplified in the next fragment of Python code.

```
self.experts = deepspeed.moe.layer.MoE(
    hidden_size = 64,
    expert       = ExpertModule(),
    num_experts  = [4,8],
    ep_size      = 4,
    use_residual = True
)
```

With the this code, we instruct DeepSpeed runtime to train an MoE model with 4 experts in the initial MoE layer and 8 experts in the second MoE layer, on 4 GPUs, and $4/4 = 1$ expert/GPU on the first layer and $8/4 = 2$ experts/GPU on the second layer.

To use MoE layers in DeepSpeed, we must create two weight groups that are passed to an optimizer. The function that creates these weight groups is `split_params_into_different_moe_groups_for_optimizer`, and it can be used as illustrated in Listing 7.8.

```
1 from deepspeed.moe.utils import split_params_into_different_moe_groups_for_optimizer
2 def create_moe_param_groups(model):
3     weights = {'params': [p for p in model.parameters()], 'name': 'parameters'}
4     weight_groups = split_params_into_different_moe_groups_for_optimizer(weights)
5     return weight_groups
```

Listing 7.8: Create two weight groups in DeepSpeed MoE for the optimizer.

The created weight groups can then be fed to the ZeRO stage-2 optimizer as indicated in Listing 7.9. The `deepspeed.initialize` method wraps our model for DeepSpeed distributed training, and also returns (i) an optimizer prepared to optimize the created weight groups differently and (ii) a distributed ready data loader.

```

1 model = CNN()
2 weight_groups = create_moe_param_groups(model)
3 model_engine, optimizer, train_dataloader, __ = deepspeed.initialize(
4     args = args,
5     model = model,
6     model_parameters = weight_groups,
7     training_data = train_dataset)

```

Listing 7.9: Initialize DeepSpeed distributed training.

We can pass additional options to DeepSpeed through the args dictionary passed to `deepspeed.initialize` method, such as (i) "stage": 2, to indicate ZeRO stage-2 optimization, (ii) "cpu_offload": true, to offload model states to CPU and save memory, (iii) "overlap_comm": true, to overlap communication with computation, and (iv) "fp16": {"enabled": true, "fp16_master_weights_and_grads": true}, for mixed precision training.

```

1 class CNN(nn.Module):
2     # Simple convolutional neural network for CIFAR10 classification.
3     def __init__(self, args):
4         super(CNN, self).__init__()
5         self.conv1 = nn.Conv2d(3, 6, 5)
6         self.pool = nn.MaxPool2d(2, 2)
7         self.conv2 = nn.Conv2d(6, 16, 5)
8         self.fc1 = nn.Linear(16 * 5 * 5, 120)
9         self.fc2 = nn.Linear(120, 84)
10        self.moe = args.moe
11        if self.moe:
12            # An MoE expert is a Linear layer with 84 input and 84 output features
13            fc3 = nn.Linear(84, 84)
14            self.moe_layer_list = []
15            # Create 'n_e' MoE experts, each equal to 'fc3'
16            for n_e in args.num_experts:
17                self.moe_layer_list.append(
18                    deepspeed.moe.layer.MoE(
19                        hidden_size = 84,
20                        expert = fc3,
21                        num_experts = n_e,
22                        ep_size = args.ep_world_size,
23                        use_residual = args.mlp_type == "residual",
24                        k = args.top_k,
25                        min_capacity = args.min_capacity,
26                        noisy_gate_policy = args.noisy_gate_policy,
27                    )
28                )
29            self.moe_layer_list = nn.ModuleList(self.moe_layer_list)
30            self.fc4 = nn.Linear(84, 10)
31        else:
32            self.fc3 = nn.Linear(84, 10)
33    def forward(self, x):
34        x = self.pool(F.relu(self.conv1(x)))
35        x = self.pool(F.relu(self.conv2(x)))
36        x = x.view(-1, 16 * 5 * 5)
37        x = F.relu(self.fc1(x))
38        x = F.relu(self.fc2(x))
39        if self.moe:
40            for layer in self.moe_layer_list:
41                x, _, _ = layer(x)
42            x = self.fc4(x)
43        else:
44            x = self.fc3(x)
45    return x

```

Listing 7.10: Simple convolutional neural network with a MoE layer for CIFAR10 classification.

We present in Listing 7.10 a simple convolutional neural network (CNN) with a MoE layer for CIFAR10 image classification. The neural network includes a Conv2d-MaxPool-Conv2d section, and a Linear-ReLU pair, a second Linear-ReLU pair, and a final Linear layer. The MoE layer includes `n_e` experts, each is a `nn.Linear(84, 84)` sub-layer, the parallelism degree is specified by the `args.ep_world_size` value, the MoE layer will be of the residual type if `args.mlp_type` is "residual", the number of active experts per token (`k`) is defined by the `args.top_k` value, the minimum capacity per expert is specified by the `args.min_capacity` value, and the noisy gate policy is defined by `args.noisy_gate_policy`.

Listing 7.11 shows the fragment of code responsible for initializing the DeepSpeed distributed backend.

```
1 # Initialize the DeepSpeed distributed backend
2 deepspeed.init_distributed()
3 _local_rank = int(os.environ.get("LOCAL_RANK"))
4 get_accelerator().set_device(_local_rank)
```

Listing 7.11: Initialize the DeepSpeed distributed backend.

The code included in Listing 7.12 allows for downloading, if necessary, the torchvision CIFAR10 training and test sets. The download is performed by rank 0 only. The training set has 50k images and the test set has 10k images, of size $3 \times 32 \times 32$. The code also defines a transformation to convert the original PIL images to tensors and to normalize their values to the $[-1 : 1]$ range.

```
1 transform = transforms.Compose(
2     [
3         transforms.ToTensor(),
4         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
5     ]
6 )
7
8 if torch.distributed.get_rank() != 0:
9     # It might be necessary to download cifar10 dataset.
10    # Rank 0 will perform the downloading.
11    torch.distributed.barrier()
12
13 # Download CIFAR10 training set in rank 0
14 trainset = torchvision.datasets.CIFAR10(
15     root = "./data",
16     train = True,
17     download = True,
18     transform = transform,
19 )
20
21 testset = torchvision.datasets.CIFAR10(
22     root = "./data",
23     train = False,
24     download = True,
25     transform = transform
26 )
27
28 if torch.distributed.get_rank() == 0:
29     # CIFAR10 training data is downloaded.
30     # Signal the other ranks that they can proceed.
31     torch.distributed.barrier()
```

Listing 7.12: Create train and test CIFAR10 datasets.

The code contained in Listing 7.13 instantiates the convolution neural network MoE model, creates a separate weight group for each MoE expert, reads the training configuration, initializes DeepSpeed to obtain a distributed model, a distributed data loader and a DeepSpeed optimizer, specify the data type to use during mixed precision training, and selects the cross-entropy loss function.

```

1  # Instantiate the convolution neural network MoE model
2  model = CNN(args)
3
4  # Get the list of weights that require gradients
5  weights = filter(lambda p: p.requires_grad, model.parameters())
6
7  # Create a separate weight group for each MoE expert
8  if args.moe_param_group:
9      weights = create_moe_param_groups(model)
10
11 # Initialize DeepSpeed to use the following features:
12 # 1) a distributed model
13 # 2) a distributed data loader
14 # 3) a DeepSpeed optimizer
15
16 ds_config = get_ds_config(args)
17
18 model_engine, optimizer, trainloader, __ = deepspeed.initialize(
19     args = args,
20     model = model,
21     model_parameters = weights,
22     training_data = trainset,
23     config = ds_config,
24 )
25
26 # Get the local device name and the local rank
27 local_device = get_accelerator().device_name(model_engine.local_rank)
28 local_rank = model_engine.local_rank
29
30 # For data types different than 'float32', specify the data type conversion
31 target_dtype = None
32 if model_engine.bfloat16_enabled():
33     target_dtype = torch.bfloat16
34 elif model_engine.fp16_enabled():
35     target_dtype = torch.half
36
37 # Select the cross-entropy loss function
38 criterion = nn.CrossEntropyLoss()

```

Listing 7.13: Wrap the model with DeepSpeed.

Listing 7.14 contains the code for the training loop. It has an outer loop over the number of epochs and an inner loop over the data loader minibatches. It is a very standard training loop with the forward pass, calculating the loss with the predicted and true image labels, calculating the loss gradients, and updating the model weights.

```

1 # Iterate over the dataset multiple times
2 for epoch in range(args.epochs):
3     running_loss = 0.0
4     for i, data in enumerate(trainloader):
5         # The loaded minibatch of images and labels is a list with two tensors
6         inputs, labels = data[0].to(local_device), data[1].to(local_device)
7
8         # Convert images to 'target_dtype' if necessary
9         if target_dtype != None:
10             inputs = inputs.to(target_dtype)
11
12         # Run the forward pass
13         outputs = model_engine(inputs)
14
15         # Compute the loss
16         loss = criterion(outputs, labels)
17
18         # Calculate the loss gradients
19         model_engine.backward(loss)
20
21         # Update the model weights
22         model_engine.step()
23
24         # Print the training statistics
25         running_loss += loss.item()
26         if local_rank == 0 and i % args.log_interval == (args.log_interval - 1):
27             # Print statistics every 'log_interval' minibatches
28             print(f"[{epoch+1:d}, {i+1:5d}] loss: {running_loss/args.log_interval:.3f}")
29             running_loss = 0.0

```

Listing 7.14: Train the MoE model.

Finally, Listing 7.15 contains the code for evaluating the trained model on the test set. It calculates the overall accuracy and the accuracy per class.

Table 7.5 shows the hyperparameters that were kept fixed during the diverse training experiments. It defines a minibatch size of 16, selects the Adam and specifies its parameters, selects the WarmupLR learning rate scheduler and defines its parameters, specifies that we want to clip the gradients larger than 1.0, and defines several parameters for the ZeRO optimization, such as the optimization stage (0 by default), the bucket size used in AllGather operations (5×10^7), the bucket size used in Reduce operation (5×10^7), indicates that we want to overlap communication with computation, and that we do not want offload model states to CPU memory.

Table 7.6 contains an example of additional hyperparameters used to train a CIFAR10 **MoE model**. In this case, it specifies that we want to use only one node with 4 GPUs, sets the number of training epochs to 30, specifies that the mixed precision training data type is Bfloat16, sets the ZeRO optimization stage to 0, specifies that our model has a standard MoE layer with 8 experts, sets the expert parallelism degree to 4, sets the gating strategy to top-2, and indicates that we want to create separate a weight group per expert.

Table 7.7 contains an example of additional hyperparameters used to train a CIFAR10 **PR-MoE model**. In this example, it specifies that we want to use one node with 4 GPUs, sets the number of training epochs to 30, specifies that the mixed precision training data type is Bfloat16, sets the ZeRO optimization stage to 0, specifies that our PR-MoE layer has residual sub-layers, with 4 and 8 experts, sets the expert parallelism degree to 4, sets the gating strategy to top-1, sets the noisy gate policy to "RSample", and indicates that we want to create separate a weight group per expert.


```

1 # Define the test DataLoader
2 testloader = torch.utils.data.DataLoader(
3     testset ,
4     batch_size = test_batch_size ,
5     shuffle     = False ,
6     num_workers = 2,
7 )
8
9 # To calculate the total accuracy
10 correct , total = 0, 0
11
12 # To calculate the accuracy per class
13 class_correct = list(0.0 for i in range(10))
14 class_total   = list(0.0 for i in range(10))
15
16 # Test loop .....
17 model_engine.eval()
18 with torch.no_grad():
19     for data in testloader:
20         images , labels = data
21         if target_dtype != None:
22             images = images.to(target_dtype)
23         outputs = model_engine(images.to(local_device))
24         _, predicted = torch.max(outputs.data , 1)
25         # Count the total predictions and the correct predictions
26         total += labels.size(0)
27         correct += (predicted == labels.to(local_device)).sum().item()
28
29         # Calculate the accuracy for each class
30         batch_correct = (predicted == labels.to(local_device)).squeeze()
31         for i in range(test_batch_size):
32             label = labels[i]
33             class_correct[label] += batch_correct[i].item()
34             class_total[label] += 1
35
36 if model_engine.local_rank == 0:
37     print(f"Accuracy of the network on {total} test images: {100*correct/total :.0f} %")
38
39     # Print the accuracy of all classes
40     for i in range(10):
41         print(
42             f"Accuracy of {classes[i]: >5s}: {100*class_correct[i]/class_total[i]:2.0f}%" )

```

Listing 7.15: Evaluate the model on the test set.

Table 7.8 contains the training time, the used GPU memory and the test accuracy for the CIFAR10 CNN, the CNN with a MoE layer, and the CNN with a PR-MoE layer. The reported results were obtained on a node with four GPUs NVIDIA RTX 3060 with 12GB VRAM each. Before discussing the results, we should note that our dense model is quite small compared to common large language models, where MOEs are usually included. Our goal with the current experiments was essentially to apply expert parallelism. Thus, it is no surprise that the obtained accuracy, around 55-58% for 10 classes, is not impressive. The first relevant conclusion is the addition of the expert layer to more than double the training time. Considering ZeRO stage-2 optimizations, instead of stage-0, also increased the training time in 21%. Training a model with a PR-MoE layer, with [4, 8] experts, took 68% more time than training a model with a MoE layer containing 8 experts. The second relevant conclusion is the impact that using two active experts, instead of only one, has on the model accuracy. The accuracy rises from 22% to 55%. We do not have great conclusions to highlight about the GPU memory cost difference among different model configurations, since the model is small. Thus, the PR-MoE version with two active experts was the slowest to train but achieved the highest accuracy. Using full-precision FP32 did not increase the training time when compared to Bfloat16, meaning that (i) we are not using mixed precision effectively or (ii) the communication time among GPUs dominates over computation time.

train_batch_size	16
optimizer: type params: lr betas eps weight_decay	Adam 0.0001 [0.8, 0.999] 10^{-8} 3×10^{-7}
scheduler: type params: warmup_min_lr warmup_max_lr warmup_num_steps	WarmupLR 0 0.001 1000
gradient_clipping	1.0
prescale_gradients	False
bf16: enabled	<code>args.dtype == bf16</code>
fp16: enabled fp16_master_weights_and_grads loss_scale loss_scale_window hysteresis min_loss_scale initial_scale_power	<code>args.dtype == fp16</code> False 0 500 2 1 15
wall_clock_breakdown	False
zero_optimization: stage allgather_partitions reduce_scatter allgather_bucket_size reduce_bucket_size overlap_comm contiguous_gradients cpu_offload	<code>args.stage</code> True True 50000000 50000000 True True False

Table 7.5: Hyperparameters for training CIFAR10 MoE models that were kept fixed.

Hyperparameter	Value	Description
num_nodes	1	Number of nodes
num_gpus	4	Number of GPUs per node
epochs	30	number of training epochs
log-interval	64	interval between successive logs
dtype	"bf16"	Datatype used for training ("bf16", "fp16", "fp32")
stage	0	ZeRO optimization stage (0, 1, 2, 3)
moe	True	use DeepSpeed mixture of experts
ep-world-size	4	expert parallelism degree
num-experts	[8,]	number of experts layers
mlp-type	"standard"	type of MoE layer ("standard", "residual")
top-k	2	MoE gating top-k (1, 2)
min-capacity	0	MoE minimum capacity of an expert
noisy-gate-policy	None	MoE noisy gating (only supported with top-1). Valid values are None, RSample, and Jitter
moe-param-group	True	Create separate MoE weight groups, required when using ZeRO with MoE

Table 7.6: Additional hyperparameters for training a CIFAR10 MoE model.

Hyperparameter	Value	Description
num_nodes	1	Number of nodes
num_gpus	4	Number of GPUs per node
epochs	30	number of training epochs
log-interval	64	interval between successive logs
dtype	"bf16"	Datatype used for training ("bf16", "fp16", "fp32")
stage	0	ZeRO optimization stage (0, 1, 2, 3)
moe	True	use DeepSpeed mixture of experts
ep-world-size	4	expert parallelism degree
num-experts	[4, 8]	number of experts layers
mlp-type	"residual"	type of MoE layer ("standard", "residual")
top-k	1	MoE gating top-k (1, 2)
min-capacity	0	MoE minimum capacity of an expert
noisy-gate-policy	"RSample"	MoE noisy gating (only supported with top-1). Valid values are None, RSample, and Jitter
moe-param-group	True	Create separate MoE weight groups, required when using ZeRO with MoE

Table 7.7: Additional hyperparameters for training a CIFAR10 PR-MoE model.

GPUs	Model	Training time (s)	Used memory per GPU (MB)	Test accuracy
1	Dense CNN	10m 15s	1090	49%
2	Dense CNN	12m 05s	293	50%
4	Dense CNN	10m 30s	293	50%
4	CNN with a MoE layer (8 experts, top-2, stage=0, dtype="bf16")	26m 05s	709	53%
4	CNN with a MoE layer (16 experts, top-2, stage=0, dtype="bf16")	27m 50s	711	55%
4	CNN with a MoE layer (16 experts, top-1, stage=0, dtype="bf16")	21m 25s	709	22%
4	CNN with a MoE layer (16 experts, top-2, stage=0, dtype="fp32")	24m 35s	703	56%
4	CNN with a MoE layer (16 experts, top-2, stage=2, dtype="bf16")	33m 40s	903	56%
4	CNN with a PR-MoE layer ([4,8] experts, top-2, stage=0, dtype="bf16")	43m 45s	711	58%

Table 7.8: Training time, used GPU memory and test accuracy for the CIFAR10 CNN, CNN with a MoE layer, and CNN with a PR-MoE layer.

7.12 Conclusions

In this chapter, we explored the role of advanced parallelism techniques in enabling the large-scale training and deployment of Mixture of Experts (MoE) models. The chapter demonstrated that efficient scaling in MoE systems requires not only the use of individual parallelism strategies, but also their careful combination and adaptation to the unique characteristics of sparse architectures. We began by examining the fundamental forms of parallelism, including data, tensor, pipeline, and sequence parallelism, along with MoE-specific approaches such as expert parallelism and expert tensor parallelism. Each of these techniques addresses different aspects of scalability, from distributing model parameters and computations to reducing memory usage and improving hardware utilization. However, the chapter highlighted that these methods introduce inherent trade-offs, particularly in terms of communication overhead and synchronization requirements.

Building on these foundations, we analyzed the evolution of hybrid parallelism strategies, which combine multiple techniques to achieve better performance and scalability. The progression from homogeneous designs to heterogeneous approaches and ultimately to dynamic parallelism illustrates an important shift toward more flexible and workload-aware systems. In particular, heterogeneous methods exploit the structural differences between dense attention layers and sparse expert layers, while dynamic strategies adapt configurations at runtime to respond to changing workloads and system conditions. The inclusion of automatic parallelization frameworks further emphasizes the growing complexity of these systems, where optimal configurations can no longer be determined manually.

A central focus of the chapter was expert parallelism, which enables the distribution of experts across devices and forms the backbone of large-scale MoE systems. While this approach provides significant benefits in terms of memory efficiency and scalability, it also introduces challenges such as All-to-All communication costs, load balancing, and routing-induced irregularity. These challenges require specialized optimizations, including efficient communication primitives, token grouping strategies, and auxiliary mechanisms to ensure balanced workloads across experts.

It was also examined how expert parallelism is integrated into real-world systems, such as Switch Transformers, Tutel, and DeepSpeed-MoE. These systems highlight different optimization strategies, including flexible parallel configurations, communication-computation overlap, and expert-aware scheduling. In particular, DeepSpeed-MoE demonstrates how multi-dimensional parallelism and system-level optimizations can be combined to achieve both high throughput and low latency, especially during inference.

Beyond individual frameworks, we discussed more advanced system designs such as Pathways and Megatron-Core, which rethink traditional distributed training architectures. Pathways introduces an asynchronous dataflow execution model, enabling flexible scheduling and improved hardware utilization, while Megatron-Core addresses the dense–sparse mismatch through parallel folding, allowing different layers to adopt independent parallelization strategies. These approaches represent a significant step forward in handling the structural heterogeneity of MoE models.

The chapter further addressed practical system implementations, including NVIDIA Megatron Bridge and vLLM, which incorporate features such as optimized token dispatching, load balancing mechanisms, and elastic expert parallelism. In particular, elastic scaling demonstrates how distributed MoE systems can dynamically adjust their resource usage at runtime, improving both efficiency and resilience in real-world deployments.

Another important topic covered was sequence parallelism for long-context training, including approaches such as Megatron sequence parallelism, DeepSpeed-Ulysses, Ring Attention, and Striped Attention. These methods show how communication patterns and computation can be reorganized to mitigate the quadratic complexity of attention and enable efficient processing of extremely long sequences, further extending the scalability of modern models.

Finally, it is presented a practical implementation example using the DeepSpeed-MoE framework, illustrating how expert parallelism and memory optimization techniques such as ZeRO can be applied in practice. This example reinforces the idea that successful deployment of MoE models requires not only theoretical understanding, but also careful engineering of both model architecture and distributed training infrastructure.

In conclusion, the chapter highlights that scalability in MoE systems is fundamentally a multi-dimensional optimization problem, involving trade-offs between computation, memory, communication, and system complexity. Future progress will likely depend on tighter integration between model design and system architecture, as well as on increasingly automated and adaptive parallelization techniques, capable of efficiently exploiting heterogeneous hardware and dynamic workloads.

Chapter 8

Conclusions

Chapters two through seven collectively highlight that the evolution of modern deep learning is fundamentally driven by the need to scale models far beyond the limits of single-device computation. Distributed training emerges as the essential foundation that enables this scaling, requiring not only multiple GPUs or compute nodes, but also a deep understanding of how computation, communication, and hardware interact. Efficient training is no longer determined solely by algorithmic improvements, but by the ability to orchestrate large-scale systems in which communication patterns, synchronization mechanisms, and hardware characteristics play a decisive role.

Within this context, parallelism appears as the central paradigm for achieving scalability. However, the chapters make clear that parallelism is not a single technique, but a rich and multi-dimensional design space. Data parallelism, pipeline parallelism, tensor parallelism, and expert parallelism each address different limitations related to computation, memory, and model size. A key unifying insight is that no individual strategy is sufficient on its own. Instead, the most effective solutions rely on hybrid approaches that combine several forms of parallelism in a coordinated manner, allowing systems to balance workload distribution, memory usage, and communication costs.

As models grow in scale, communication becomes a dominant bottleneck. Operations such as gradient aggregation or token routing across devices often determine performance, more than raw computation. The chapters consistently emphasize that optimizing communication, through efficient collective algorithms, overlapping communication with computation, and minimizing synchronization overhead, is critical for achieving scalability. This insight shifts the focus from purely computational efficiency to a broader systems perspective, where algorithm design must account for the realities of distributed environments.

Another recurring theme is the importance of memory efficiency. Current neural network architectures frequently exceed the memory capacity of individual devices, driving the development of techniques such as weight sharding, optimizer state partitioning, and splitting the model layers. While these methods enable the training of significantly larger models, they also introduce additional communication overhead and system complexity. As a result, scaling becomes a continuous trade-off between memory savings, communication cost, and computational efficiency, reinforcing the idea that all resources must be jointly optimized.

The Mixture of Experts paradigm represents a major step forward in scaling model capacity by introducing conditional computation. By activating only a subset of model weights for each input, MoE architectures achieve a substantial increase in size without a proportional increase in computation. However, this efficiency comes at the cost of increased system complexity. Challenges such as load balancing, routing stability, and communication-intensive operations, particularly All-to-All exchanges, make MoE systems significantly harder to design and train. Consequently, their success depends on carefully balancing algorithmic innovation with system-level optimization.

From an implementation perspective, the six chapters demonstrate that translating these ideas into practice is far from trivial. Building scalable systems requires more than theoretical understanding, it demands careful engineering of routing mechanisms, batching strategies, and training stabilization techniques. Issues such

as numerical instability, uneven workload distribution, and inefficient hardware utilization must be addressed through a combination of algorithmic adjustments and practical optimizations. This highlights that scalability is as much an engineering challenge as it is a theoretical one.

Finally, the work points toward a future in which distributed training systems become increasingly adaptive and automated. The growing complexity of hybrid parallelism and heterogeneous architectures makes manual optimization impractical, leading to the emergence of dynamic and automated parallelization frameworks. These systems aim to determine optimal configurations at runtime, adapting to model structure, hardware characteristics, and workload variability. This evolution reflects a broader trend toward tighter integration between model design and system infrastructure.

In summary, the global conclusion is that scaling modern deep learning systems is inherently a multi-dimensional optimization problem. It requires the careful co-design of model architecture, parallelization strategies, and distributed systems, while continuously balancing trade-offs between computation, communication, memory, and complexity. The most effective approaches are those that embrace this complexity through hybrid, adaptive, and system-aware designs, enabling the training and deployment of increasingly large and capable models.

Bibliography

- [1] Alekh Agarwal, Olivier Chapelle, Miroslav Dudik, and John Langford. A reliable effective terascale linear learning system. <https://arxiv.org/abs/1110.4198>, 2013.
- [2] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. <https://arxiv.org/abs/2305.13245>, 2023.
- [3] Amjad Almahairi, Nicolas Ballas, Tim Cooijmans, Yin Zheng, Hugo Larochelle, and Aaron Courville. Dynamic capacity networks. <https://arxiv.org/abs/1511.07838>, 2016.
- [4] Itay Alroy, Yongji Wu, Rui Qiao, Tyler Smith, Moein Khazraee, Omri Kahalon, Tzu-Ling Kan, and Ron Tourgeman. Elastic expert parallelism in vllm. <https://vllm-project.github.io/2026/05/14/elastic-expert-parallelism.html>, 2026.
- [5] Ali Alvi and Paresh Kharya. Using DeepSpeed and megatron to train Megatron-Turing NLG 530B, the world’s largest and most powerful generative language model. <https://arxiv.org/abs/2201.11990>, 2021.
- [6] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, and Yuxiong He. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale. <https://arxiv.org/abs/2207.00032>, 2022.
- [7] Szymon Antoniak, Michal Krutul, Maciej Pióro, Jakub Krajewski, Jan Ludziejewski, Kamil Ciebia, Krystian Król, Tomasz Odrzygóźdź, Marek Cygan, and Sebastian Jaszczur. Mixture of tokens: Continuous moe through cross-example aggregation. <https://arxiv.org/abs/2310.15961>, 2023.
- [8] Mikel Artetxe, Shruti Bhosale, Naman Goyal, Todor Mihaylov, Myle Ott, Sam Shleifer, Xi Victoria Lin, Jingfei Du, Srinivasan Iyer, Ramakanth Pasunuru, Giri Anantharaman, Xian Li, Shuohui Chen, Halil Akin, Mandeep Baines, Louis Martin, Xing Zhou, Punit Singh Koura, Brian O’Horo, Jeff Wang, Luke Zettlemoyer, Mona Diab, Zornitsa Kozareva, and Veselin Stoyanov. Efficient large scale language modeling with mixtures of experts. <https://arxiv.org/abs/2112.10684>, 2021.
- [9] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Dan Hurt, Michael Isard, and et al. Pathways: Asynchronous distributed dataflow for ml, 2022.
- [10] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. <https://arxiv.org/abs/2004.05150>, 2020.
- [11] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. <https://arxiv.org/abs/1308.3432>, 2013.
- [12] Cody Blakeney, Mansheej Paul, Brett W. Larsen, Sean Owen, and Jonathan Frankle. Does your data spark joy? performance gains from domain upsampling at the end of training. <https://arxiv.org/abs/2406.03476>, 2024.
- [13] Simon Boehm. Data-parallel distributed training of deep learning models. <https://siboehm.com/articles/22/data-parallel-training>, 2022.

- [14] Simon Boehm. Pipeline-parallelism: Distributed training via model partitioning. <https://siboehm.com/articles/22/pipeline-parallel-training>, 2022.
- [15] Leon Bottou. Stochastic gradient learning in neural networks. In *Proceedings of Neuro-Nimes 91*, 1991.
- [16] William Brandon, Aniruddha Nrusimha, Kevin Qian, Zachary Ankner, Tian Jin, Zhiye Song, and Jonathan Ragan-Kelley. Striped attention: Faster ring attention for causal transformers. <https://arxiv.org/abs/2311.09431>, 2023.
- [17] Tom Brown, Benjamin Mann, Nicholas Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Nathaniel Shinn, Pranav Mazin, Girish Sastry, Amanda Askell, and et al. Language models are few-shot learners. <https://arxiv.org/abs/2005.14165>, 2020.
- [18] Weilin Cai, Juyong Jiang, Le Qin, Junwei Cui, Sunghun Kim, and Jiayi Huang. Shortcut-connected expert parallelism for accelerating mixture-of-experts. <https://arxiv.org/abs/2404.05019>, 2024.
- [19] Weilin Cai, Juyong Jiang, Fan Wang, Jing Tang, Sunghun Kim, and Jiayi Huang. A survey on mixture of experts in large language models. *IEEE Transactions on Knowledge and Data Engineering*, page 1–20, 2025.
- [20] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 62–75. ACM, 2021.
- [21] Aman Chadha and Vinija Jain. Mixture of experts. *Distilled AI*, 2020. <https://aman.ai/primers/ai/mixture-of-experts/>.
- [22] Li-Wen Chang, Wenlei Bao, Qi Hou, Chengquan Jiang, Ningxin Zheng, Yinmin Zhong, Xuanrun Zhang, and et al. Flux: Fast software-based communication overlap on gpus through kernel fusion. <https://arxiv.org/abs/2406.06858>, 2024.
- [23] Sherman Chann. Knowing enough about moe to explain dropped tokens in gpt-4. <https://152334h.github.io/blog/knowing-enough-about-moe/>, 2023.
- [24] Chang Chen, Min Li, Zhihua Wu, Dianhai Yu, and Chao Yang. Ta-moe: Topology-aware large scale mixture-of-expert training. <https://arxiv.org/abs/2302.09915>, 2023.
- [25] Fahao Chen, Peng Li, Zicong Hong, Zhou Su, and Song Guo. Communication-efficient sparsely-activated model training via sequence migration and token condensation. <https://arxiv.org/abs/2411.15419>, 2024.
- [26] Mark Chen, Jerry Tworek, Heewoo Jun, and Qiming Yuan. Evaluating large language models trained on code. <https://arxiv.org/abs/2107.03374>, 2021.
- [27] Shaoxiang Chen, Zequn Jie, and Lin Ma. Llava-mole: Sparse mixture of lora experts for mitigating data conflicts in instruction finetuning mllms. <https://arxiv.org/abs/2401.16160>, 2024.
- [28] Tianlong Chen, Zhenyu Zhang, Ajay Jaiswal, Shiwei Liu, and Zhangyang Wang. Sparse moe as the new dropout: Scaling dense and self-slimmable transformers. <https://arxiv.org/abs/2303.01610>, 2023.
- [29] Wuyang Chen, Yanqi Zhou, Nan Du, Yanping Huang, James Laudon, Zhifeng Chen, and Claire Cui. Lifelong language pretraining with distribution-specialized experts. <https://arxiv.org/abs/2305.12281>, 2023.
- [30] Xin Chen, Hengheng Zhang, Xiaotao Gu, Kaifeng Bi, Lingxi Xie, and Qi Tian. Pipeline moe: A flexible moe implementation with pipeline parallelism. <https://arxiv.org/abs/2304.11414>, 2023.
- [31] Zhen Chen and Yilun Shen. Mod-squad. In *CVPR*, 2023.
- [32] Zixiang Chen, Yihe Deng, Yue Wu, Quanquan Gu, and Yuanzhi Li. Towards understanding mixture of experts in deep learning. <https://arxiv.org/abs/2208.02813>, 2022.

- [33] Jung-Yoon Choi. Sparse mixture-of-prompts for parameter-efficient learning. In *EMNLP*, 2023.
- [34] Mohammed Nowaz Rabbani Chowdhury, Shuai Zhang, Meng Wang, Sijia Liu, and Pin-Yu Chen. Patch-level routing in mixture-of-experts is provably sample-efficient for convolutional neural networks. In *ICML*, 2023.
- [35] Dan Ciresan, Ueli Meier, Luca Gambardella, and Jurgen Schmidhuber. Deep big simple neural nets excel on handwritten digit recognition. *CoRR*, abs/1003.0358, 2010.
- [36] Aidan Clark, Diego de Las Casas, Aurelia Guy, Arthur Mensch, Michela Paganini, Jordan Hoffmann, Bogdan Damoc, Blake Hechtman, Trevor Cai, Sebastian Borgeaud, and et al. Unified scaling laws for routed language models. *CoRR*, 2022.
- [37] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, and Mark Chen. Training verifiers to solve math word problems. <https://arxiv.org/abs/2110.14168>, 2021.
- [38] Open SHMEM Specification Committee. Openshmem application programming interface version 1.6. http://www.openshmem.org/site/sites/default/site_files/OpenSHMEM-1.6.pdf, 2024.
- [39] NVIDIA Corporation. Matrix multiplication background user’s guide. <https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html>, 2023.
- [40] NVIDIA Corporation. Transformerengine: A library for accelerating transformer models on nvidia gpu. <https://github.com/NVIDIA/TransformerEngine>, 2023.
- [41] NVIDIA Corporation. Nvidia megatron core. <https://developer.nvidia.com/megatron-core>, 2025.
- [42] NVIDIA Corporation. Nemo megatron bridge: Parallelisms guide (v0.42). <https://docs.nvidia.com/nemo/megatron-bridge/latest/parallelisms.html>, 2026.
- [43] NVIDIA Corporation. Nvidia collective communication library documentation (v2.30). <https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/index.html>, 2026.
- [44] Danqi Dai. Deepseekmoe. <https://arxiv.org/abs/2401.06066>, 2024.
- [45] Databricks. Introducing dbrx: A new state-of-the-art open llm. <https://www.databricks.com/blog/introducing-dbrx-new-state-art-open-llm>, 2024.
- [46] Christopher de Sa. Advanced optimization techniques for machine learning - lecture 20, cornell university. <https://www.cs.cornell.edu/courses/cs4787/2025fa/lectures/lecture20.html>, 2025.
- [47] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.
- [48] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [49] DeepSeek AI. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. <https://arxiv.org/abs/2405.04434>, 2024.
- [50] DeepSeek AI. Dualpipe: A bidirectional pipeline parallelism algorithm for computation-communication overlap in deepseek v3/r1 training. <https://github.com/deepseek-ai/DualPipe>, 2025.
- [51] DeepSeek AI, Aixin Liu, and et al. Deepseek-v3 technical report. <https://arxiv.org/abs/2412.19437>, 2024.
- [52] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. <https://arxiv.org/abs/1810.04805>, 2019.

- [53] Shizhe Diao, Tianyang Xu, Ruijia Xu, Jiawei Wang, and Tong Zhang. Mixture-of-domain-adapters: Decoupling and injecting domain knowledge to pre-trained language models memories. <https://arxiv.org/abs/2306.05406>, 2023.
- [54] Shihan Dou, Enyu Zhou, Yan Liu, Songyang Gao, Jun Zhao, Wei Shen, Yuhao Zhou, Zhiheng Xi, Xiao Wang, Xiaoran Fan, Shiliang Pu, Jiang Zhu, Rui Zheng, Tao Gui, Qi Zhang, and Xuanjing Huang. Loramoe: Alleviate world knowledge forgetting in large language models via moe-style plugin. <https://arxiv.org/abs/2312.09979>, 2023.
- [55] Shihan Dou, Enyu Zhou, Yan Liu, Songyang Gao, Jun Zhao, Wei Shen, Yuhao Zhou, Zhiheng Xi, Xiao Wang, Xiaoran Fan, Shiliang Pu, Jiang Zhu, Rui Zheng, Tao Gui, Qi Zhang, and Xuanjing Huang. Loramoe: Alleviate world knowledge forgetting in large language models via moe-style plugin. <https://arxiv.org/abs/2312.09979>, 2023.
- [56] Nan Du, Yanping Huang, Andrew M. Dai, Sid Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, and et al. Glam: Efficient scaling of language models with mixture-of-experts. In *International Conference on Machine Learning (ICML)*, 2022.
- [57] Dheeru Dua, Shruti Bhosale, Vedanuj Goswami, James Cross, Mike Lewis, and Angela Fan. Tricks for training sparse translation models. <https://arxiv.org/abs/2110.08246>, 2021.
- [58] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.
- [59] Angela Fan, Shruti Bhosale, Holger Schwenk, Zhiyi Ma, Ahmed El-Kishky, Siddharth Goyal, Mandeep Baines, Onur Celebi, Guillaume Wenzek, and et al. Beyond english-centric multilingual machine translation. <https://arxiv.org/abs/2010.11125>, 2020.
- [60] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. Dapple: A pipelined data parallel approach for training large models. <https://arxiv.org/abs/2007.01045>, 2020.
- [61] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *CoRR*, 2021.
- [62] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23, 2022.
- [63] Elias Frantar and Dan Alistarh. Qmoe: Extreme compression for mixture-of-experts models. <https://arxiv.org/abs/2310.16795>, 2023.
- [64] Zichuan Fu, Wentao Song, Yejing Wang, Xian Wu, Yefeng Zheng, Yingying Zhang, Derong Xu, Xuetao Wei, Tong Xu, and Xiangyu Zhao. Sliding window attention training for efficient large language models. <https://arxiv.org/abs/2502.18845>, 2025.
- [65] Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. Megablocks: Efficient sparse training with mixture-of-experts. <https://arxiv.org/abs/2211.15841>, 2023.
- [66] Swapnil Gandhi and Christos Kozyrakis. Sparse checkpointing for fast and reliable moe training. <https://arxiv.org/abs/2412.15411>, 2026.
- [67] Chongyang Gao, Kezhen Chen, Jinmeng Rao, Baochen Sun, Ruibo Liu, Daiyi Peng, Yawen Zhang, Xiaoyuan Guo, Jie Yang, and VS Subrahmanian. Higher layers need more lora experts. <https://arxiv.org/abs/2402.08562>, 2024.
- [68] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, and et al. The Pile: An 800GB dataset of diverse text for language modeling. <https://arxiv.org/abs/2101.00027>, 2020.
- [69] Andrew Gibiansky. Bringing hpc techniques to deep learning. <https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/>, 2017.

- [70] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *AISTATS*, 2011.
- [71] Yunhao Gou, Zhili Liu, Kai Chen, Lanqing Hong, Hang Xu, Aoxue Li, Dit-Yan Yeung, James T. Kwok, and Yu Zhang. Mixture of cluster-conditional lora experts for vision-language instruction tuning. <https://arxiv.org/abs/2312.12379>, 2023.
- [72] Sam Gross, Marc’Aurelio Ranzato, and Arthur Szlam. Hard mixtures of experts for large scale weakly supervised vision. <https://arxiv.org/abs/1704.06363>, 2017.
- [73] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, and et al. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, 2025.
- [74] Zihang Guo and et al. Netmoe: Accelerating moe training through dynamic sample placement. In *SC (International Conference for High Performance Computing, Networking, Storage and Analysis)*, 2023.
- [75] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. <https://arxiv.org/abs/1806.03377>, 2018.
- [76] Jiaao He, Jiezhong Qiu, Aohan Zeng, Zhilin Yang, Jidong Zhai, and Jie Tang. Fastmoe: A fast mixture-of-expert training system. <https://arxiv.org/abs/2103.13262>, 2021.
- [77] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. Fastermoe: modeling and optimizing training of large-scale dynamic pre-trained models. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 120–134. Association for Computing Machinery, 2022.
- [78] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. <https://arxiv.org/abs/1512.03385>, 2015.
- [79] Shwai He, Liang Ding, Daize Dong, Boan Liu, Fuqiang Yu, and Dacheng Tao. Pad-net: An efficient framework for dynamic networks. <https://arxiv.org/abs/2211.05528>, 2023.
- [80] Shwai He, Daize Dong, Liang Ding, and Ang Li. Towards efficient mixture of experts: A holistic study of compression techniques. <https://arxiv.org/abs/2406.02500>, 2024.
- [81] Xu Owen He. Mixture of a million experts. <https://arxiv.org/abs/2407.04153>, 2024.
- [82] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *CoRR*, 2020.
- [83] Dan Hendrycks, Collin Burns, and Saurav Kadavath. Measuring mathematical problem solving. <https://arxiv.org/abs/2103.03874>, 2021.
- [84] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). <https://arxiv.org/abs/1606.08415>, 2016.
- [85] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, and et al. Training compute-optimal large language models. <https://arxiv.org/abs/2203.15556>, 2022.
- [86] Edward J. Hu, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *ICLR*, 2021.
- [87] Haiyang Huang, Newsha Ardalani, Anna Sun, Liu Ke, Hsien-Hsin Lee, Anjali Sridhar, Shruti Bhosale, Carole-Jean Wu, and Benjamin Lee. Towards moe deployment: Mitigating inefficiencies in mixture-of-expert (moe) inference. <https://arxiv.org/abs/2303.06182>, 2023.
- [88] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, Hyoungho Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. Gpipe: Easy scaling with micro-batch

- pipeline parallelism. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.
- [89] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyukJoong Lee, Jiquan Ngiam, Quoc Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. <https://arxiv.org/abs/1811.06965>, 2019.
 - [90] Yongqi Huang, Peng Ye, Xiaoshui Huang, Sheng Li, Tao Chen, Tong He, and Wanli Ouyang. Experts weights averaging: A new general training scheme for vision transformers. <https://arxiv.org/abs/2308.06093>, 2023.
 - [91] Ranggi Hwang, Jianyu Wei, Shijie Cao, Changho Hwang, Xiaohu Tang, Ting Cao, Mao Yang, and Minsoo Rhu. Pre-gated moe: An algorithm-system co-design for fast and scalable mixture-of-expert inference. <https://arxiv.org/abs/2308.12066>, 2023.
 - [92] Robert Jacobs, Michael Jordan, Steven Nowlan, and Geoffrey Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3:79–87, 1991.
 - [93] Sam Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models. <https://arxiv.org/abs/2309.14509>, 2023.
 - [94] Deepak Jain and et al. Mario: Near zero-cost activation checkpointing in pipeline parallelism. In *MLSys Workshop / SysML*, 2021.
 - [95] Deepak Jain, Hai Li, and et al. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of ASPLOS*, 2020.
 - [96] Sylvain Jeaugey. Massively scale your deep learning training with nccl 2.4. <https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4/>, 2019.
 - [97] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. <https://arxiv.org/abs/1807.05358>, 2018.
 - [98] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, and et al. Mistral 7b. <https://arxiv.org/abs/2310.06825>, 2023.
 - [99] Albert Qiang Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, and Devendra Singh Chaplot. Mixtral of experts. *arXiv:2401.04088*, 2024.
 - [100] Chenyu Jiang, Ye Tian, Zhen Jia, Shuai Zheng, Chuan Wu, and Yida Wang. Lancet: Accelerating mixture-of-experts training via whole graph computation-communication overlapping. <https://arxiv.org/abs/2404.19429>, 2024.
 - [101] Chao Jin, Ziheng Jiang, Zhihao Bai, Zheng Zhong, Juncai Liu, Xiang Li, Ningxin Zheng, Xi Wang, Cong Xie, Qi Huang, Wen Heng, Yiyuan Ma, Wenlei Bao, Size Zheng, Yanghua Peng, Haibin Lin, Xuanzhe Liu, Xin Jin, and Xin Liu. Megascale-moe: Large-scale communication-efficient training of mixture-of-experts models in production. <https://arxiv.org/abs/2505.11432>, 2025.
 - [102] Zewen Jin, Shengnan Wang, Jiaan Zhu, Hongrui Zhan, Youhui Bai, Lin Zhang, Zhenyu Ming, and Cheng Li. Bigmac: A communication-efficient mixture-of-experts model structure for fast training and inference. <https://arxiv.org/abs/2502.16927>, 2025.
 - [103] Michael Jordan and Robert Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural Computation*, 6:181–214, 1993.
 - [104] Yechan Kim, Hwijoon Lim, and Dongsu Han. Scaling beyond the gpu memory limit for large mixture-of-experts model training. In *Proceedings of the 41st International Conference on Machine Learning*. JMLR.org, 2024.

- [105] Young Jin Kim, Ammar Ahmad Awan, Alexandre Muzio, Andres Felipe Cruz Salinas, Liyang Lu, Amr Hendy, Samyam Rajbhandari, Yuxiong He, and Hany Hassan Awadalla. Scalable and efficient moe training for multitask multilingual models. <https://arxiv.org/abs/2109.10465>, 2021.
- [106] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. <https://arxiv.org/abs/1412.6980>, 2014.
- [107] Aran Komatsuzaki, Joan Puigcerver, and Joshua Lee-Thorp. Sparse upcycling: Training mixture-of-experts from dense checkpoints. In *ICLR*, 2022.
- [108] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. <https://arxiv.org/abs/2205.05198>, 2022.
- [109] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. <https://arxiv.org/abs/2205.05198>, 2022.
- [110] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. <https://arxiv.org/abs/1404.5997>, 2014.
- [111] Quoc Le, Jiquan Ngiam, Adam Coates, Abhik Lahiri, Bobby Prochnow, and Andrew Ng. On optimization methods for deep learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML, page 265–272. Omnipress, 2011.
- [112] ApX Machine Learning. Advanced optimization techniques for machine learning. <https://apxml.com/courses/optimization-techniques-ml>, 2026.
- [113] ApX Machine Learning. Mixture of experts: Core concepts and hands-on implementation. <https://apxml.com/courses/mixture-of-experts-advanced-implementation>, 2026.
- [114] Dmitry Lepikhin, Hyoungho Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. <https://arxiv.org/abs/2006.16668>, 2020.
- [115] Mike Lewis, Angela Fan, and Sergey Edunov. Base layers: Simplifying training of large, sparse models. In *ICML*, 2021.
- [116] Dengchun Li, Yingzi Ma, Naizheng Wang, Zhengmao Ye, Zhiyuan Cheng, Yinghao Tang, Yan Zhang, Lei Duan, Jie Zuo, Cal Yang, and Mingjie Tang. Mixlor: Enhancing large language models fine-tuning with lora-based mixture of experts. <https://arxiv.org/abs/2404.15159>, 2024.
- [117] Jiamin Li, Yimin Jiang, Yibo Zhu, Cong Wang, and Hong Xu. Accelerating distributed moe training and inference with lina. <https://arxiv.org/abs/2210.17223>, 2024.
- [118] Margaret Li, Suchin Gururangan, Tim Dettmers, Mike Lewis, Tim Althoff, Noah A. Smith, and Luke Zettlemoyer. Branch-train-merge: Embarrassingly parallel training of expert language models. <https://arxiv.org/abs/2208.03306>, 2022.
- [119] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: experiences on accelerating data parallel training. *Proceedings of the VLDB Endowment*, 13(12):3005–3018, 2020.
- [120] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. Colossal-ai: A unified deep learning system for large-scale parallel training. <https://arxiv.org/abs/2110.14883>, 2023.
- [121] Zonglin Li, Chong You, Srinadh Bhojanapalli, Daliang Li, Ankit Singh Rawat, Sashank J. Reddi, Ke Ye, Felix Chern, Felix Yu, Ruiqi Guo, and Sanjiv Kumar. The lazy neuron phenomenon: On emergence of activation sparsity in transformers. <https://arxiv.org/abs/2210.06313>, 2022.

- [122] Oron Lieber, Baruch Lenz, Hila Bata, and Gabi Cohen. Jamba: A hybrid transformer-mamba language model. *arXiv:2403.19887*, 2024.
- [123] Bin Lin, Zhenyu Tang, Yang Ye, Jinfa Huang, Junwu Zhang, Yatian Pang, Peng Jin, Munan Ning, Jiebo Luo, and Li Yuan. Moe-llava: Mixture of experts for large vision-language models. <https://arxiv.org/abs/2401.15947>, 2024.
- [124] Junyang Lin, Rui Men, An Yang, Chang Zhou, Ming Ding, Yichang Zhang, Peng Wang, Ang Wang, Le Jiang, and et al. M6: A chinese multimodal pretrainer. <https://arxiv.org/abs/2103.00823>, 2021.
- [125] Zhiqi Lin, Youshan Miao, Quanlu Zhang, Fan Yang, Yi Zhu, Cheng Li, Saeed Maleki, Xu Cao, and et al. nnscler: constraint-guided parallelization plan generation for deep learning training. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2024.
- [126] Phillip Lippe. Uva deep learning tutorials. https://github.com/phlippe/uvadlc_notebooks, 2025.
- [127] Dennis Liu, Zijie Yan, Xin Yao, Tong Liu, Vijay Korthikanti, Evan Wu, Shiqing Fan, Gao Deng, Hongxiao Bai, and et al. Moe parallel folding: Heterogeneous parallelism mappings for efficient large-scale moe model training with megatron core. <https://arxiv.org/abs/2504.14960>, 2026.
- [128] Dong Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical Programming*, 45:503–528, 1989.
- [129] Guodong Liu, Youshan Miao, Zhiqi Lin, Xiaoxiang Shi, Saeed Maleki, Fan Yang, Yungang Bao, and Sa Wang. Aceso: Efficient parallel dnn training through iterative bottleneck alleviation. In *Proceedings of the Nineteenth European Conference on Computer Systems*, page 163–181. Association for Computing Machinery, 2024.
- [130] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. <https://arxiv.org/abs/2310.01889>, 2023.
- [131] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. <https://arxiv.org/abs/2310.01889>, 2023.
- [132] Juncai Liu, Jessie Hui Wang, and Yimin Jiang. Janus: A unified distributed training framework for sparse mixture-of-experts models. In *Proceedings of the ACM SIGCOMM 2023 Conference*, page 486–498. Association for Computing Machinery, 2023.
- [133] Xudong Lu, Qi Liu, Yuhui Xu, Aojun Zhou, Siyuan Huang, Bo Zhang, Junchi Yan, and Hongsheng Li. Not all experts are equal: Efficient expert pruning and skipping for mixture-of-experts large language models. <https://arxiv.org/abs/2402.14800>, 2024.
- [134] Shuqing Luo, Jie Peng, Pingzhi Li, Hanrui Wang, and Tianlong Chen. Hexa-moe: Efficient and heterogeneous-aware training for mixture-of-experts. <https://arxiv.org/abs/2411.01288>, 2025.
- [135] Tongxu Luo, Jiahe Lei, Fangyu Lei, Weihao Liu, Shizhu He, Jun Zhao, and Kang Liu. Moelora: Contrastive learning guided mixture of experts on parameter-efficient fine-tuning for large language models. <https://arxiv.org/abs/2402.12851>, 2024.
- [136] Zixuan Ma, Jiaao He, Jiezhong Qiu, Huanqi Cao, Yuanwei Wang, Zhenbo Sun, Liyan Zheng, Haojie Wang, Shizhi Tang, and et al. Bagualu: targeting brain scale pretrained models with over 37 million cores. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 192–204. Association for Computing Machinery, 2022.
- [137] Yuning Mao, Lambert Mathias, Rui Hou, Amjad Almahairi, Hao Ma, Jiawei Han, Wen-tau Yih, and Madian Khabsa. Unipelt: A unified framework for parameter-efficient language model tuning. In *ACL*, 2022.

- [138] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, and et al. Mlperf training benchmark. <https://arxiv.org/abs/1910.01500>, 2020.
- [139] Paulius Micikevicius, Sharan Narang, Jonah Alben, and et al. Fp8-lm: Training large language models with fp8 precision. <https://arxiv.org/abs/2310.18313>, 2023.
- [140] Microsoft. Tutel: Adaptive mixture-of-experts at scale. <https://github.com/microsoft/tutel>, 2023.
- [141] Niklas Muennighoff, Luca Soldaini, Dirk Groeneveld, Kyle Lo, Jacob Morrison, Sewon Min, Weijia Shi, Pete Walsh, Oyvind Tafjord, Nathan Lambert, Yuling Gu, Shane Arora, Akshita Bhagia, Dustin Schwenk, David Wadden, Alexander Wettig, Binyuan Hui, Tim Dettmers, Douwe Kiela, Ali Farhadi, Noah A. Smith, Pang Wei Koh, Amanpreet Singh, and Hannaneh Hajishirzi. Olmoe: Open mixture-of-experts language models. <https://arxiv.org/abs/2409.02060>, 2024.
- [142] Mohammed Muqeeth, Haokun Liu, and Colin Raffel. Soft merging of experts with adaptive routing. <https://arxiv.org/abs/2306.03745>, 2023.
- [143] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, page 1–15. Association for Computing Machinery, 2019.
- [144] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel DNN training. *CoRR*, 2020.
- [145] Deepak Narayanan, Mohammad Shoeybi, Jared Casper and Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters using Megatron-LM. *CoRR*, 2021.
- [146] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, and et al. Efficient large-scale language model training on gpu clusters using megatron-lm. <https://arxiv.org/abs/2104.04473>, 2021.
- [147] Xiaonan Nie, Qibin Liu, Fangcheng Fu, Shenhan Zhu, Xupeng Miao, Xiaoyang Li, Yang Zhang, Shouda Liu, and Bin Cui. Lsh-moe: Communication-efficient moe training via locality-sensitive hashing. <https://arxiv.org/abs/2411.08446>, 2024.
- [148] Xiaonan Nie, Xupeng Miao, Shijie Cao, Lingxiao Ma, Qibin Liu, Jilong Xue, Youshan Miao, Yi Liu, Zhi Yang, and Bin Cui. Evomoe: An evolutionary mixture-of-experts training framework via dense-to-sparse gate. <https://arxiv.org/abs/2112.14397>, 2021.
- [149] Xiaonan Nie, Xupeng Miao, Zilong Wang, Zichao Yang, Jilong Xue, Lingxiao Ma, Gang Cao, and Bin Cui. Flexmoe: Scaling large-scale sparse pre-trained model training via dynamic device placement. *Proceedings of the ACM on Management of Data*, 1(1):1–19, 2023.
- [150] Xiaonan Nie, Pinxue Zhao, Xupeng Miao, Tong Zhao, and Bin Cui. Hetumoe: An efficient trillion-scale mixture-of-expert distributed training system. <https://arxiv.org/abs/2203.14685>, 2022.
- [151] University of Tennessee. Mpi: A message-passing interface standard, version 4.1. <https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf>, 2023.
- [152] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. <https://arxiv.org/abs/1904.01038>, 2019.
- [153] Myle Ott, Sam Shleifer, Min Xu, Priya Goyal, Quentin Duval, and Vittorio Caggiano. Fully sharded data parallel: faster ai training with fewer gpus. <https://engineering.fb.com/2021/07/15/open-source/fsdp/>, 2021.

- [154] Bowen Pan, Yikang Shen, Haokun Liu, Mayank Mishra, Gaoyuan Zhang, Aude Oliva, Colin Raffel, and Rameswar Panda. Dense training, sparse inference: Rethinking training of mixture-of-experts language models. <https://arxiv.org/abs/2404.05567>, 2024.
- [155] Xinglin Pan, Wenxiang Lin, Lin Zhang, Shaohuai Shi, Zhenheng Tang, Rui Wang, Bo Li, and Xiaowen Chu. Fsmoe: A flexible and scalable training system for sparse mixture-of-experts models. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, page 524–539. ACM, 2025.
- [156] Xuehai Peng, Feng Wu, and et al. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of ASPLOS*, 2020.
- [157] Joan Puigcerver, Carlos Riquelme, Basil Mustafa, and Neil Houlsby. From sparse to soft mixtures of experts. In *ICLR*, 2024.
- [158] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble pipeline parallelism. <https://arxiv.org/abs/2401.10241>, 2023.
- [159] Yuhao Qing, Guichao Zhu, Fanxin Li, Lintian Lei, Zekai Sun, Xiuxian Guan, Shixiong Zhao, Xusheng Chen, Dong Huang, Sen Wang, and Heming Cui. Hecate: Unlocking efficient sparse model training via fully sharded sparse data parallelism. <https://arxiv.org/abs/2502.02581>, 2025.
- [160] Xiaoye Qu, Daize Dong, Xuyang Hu, Tong Zhu, Weigao Sun, and Yu Cheng. Llama-moe v2: Exploring sparsity of llama from perspective of mixture-of-experts with post-training. <https://arxiv.org/abs/2411.15708>, 2024.
- [161] Qwen Team. Qwen1.5-moe. <https://arxiv.org/abs/2309.16609>, 2024.
- [162] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf, 2019.
- [163] Samyam Rajbhandari, Conglong Li, Zhewei Yao, and et al. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. In *ICML*, 2022.
- [164] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. <https://arxiv.org/abs/1910.02054>, 2020.
- [165] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. <https://arxiv.org/abs/2104.07857>, 2021.
- [166] David Raposo, Sam Ritter, Blake Richards, Timothy Lillicrap, Peter Conway Humphreys, and Adam Santoro. Mixture-of-depths: Dynamically allocating compute in transformer-based language models. <https://arxiv.org/abs/2404.02258>, 2024.
- [167] Sebastian Raschka. The big llm architecture comparison. <https://magazine.sebastianraschka.com/p/the-big-llm-architecture-comparison>, 2026.
- [168] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2020.
- [169] Xiaozhe Ren, Pingyi Zhou, Xinfan Meng, Xinjing Huang, Yadao Wang, Weichao Wang, Pengfei Li, Xiaoda Zhang, Alexander Podolskiy, Grigory Arshinov, Andrey Bout, Irina Piontkovskaya, Jiansheng Wei, Xin Jiang, Teng Su, Qun Liu, and Jun Yao. Pangu- σ : Towards trillion parameter language model with sparse heterogeneous computing. <https://arxiv.org/abs/2303.10845>, 2023.
- [170] Carlos Riquelme, Joan Puigcerver, Basil Mustafa, Maxim Neumann, Rodrigo Jenatton, Andre Susano Pinto, Daniel Keysers, and Neil Houlsby. Scaling vision with sparse mixture of experts. In *NeurIPS*, 2021.

- [171] Stephen Roller, Sainbayar Sukhbaatar, Arthur Szlam, and Jason Weston. Hash layers for large sparse models. <https://arxiv.org/abs/2106.04426>, 2021.
- [172] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Computing*, 35(12):581–594, 2009.
- [173] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. <https://arxiv.org/abs/1508.07909>, 2016.
- [174] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. <https://arxiv.org/abs/1802.05799>, 2018.
- [175] Noam Shazeer. Glu variants improve transformer. <https://arxiv.org/abs/2002.05202>, 2020.
- [176] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, and Ashish Vaswani. Mesh-tensorflow: Deep learning for supercomputers. In *NeurIPS*, 2018.
- [177] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarczyk, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *CoRR*, 2017.
- [178] Liang Shen, Zhihua Wu, Weibao Gong, Hongxiang Hao, Yangfan Bai, Huachao Wu, Xinxuan Wu, Haoyi Xiong, Dianhai Yu, and Yanjun Ma. Se-moe: A scalable and efficient mixture-of-experts distributed training and inference system. <https://arxiv.org/abs/2205.10034>, 2022.
- [179] Sheng Shen, Le Hou, Yanqi Zhou, Nan Du, S. Longpre, Jason Wei, Hyung Won Chung, Barret Zoph, William Fedus, Xinyun Chen, and et al. Mixture-of-experts meets instruction tuning: A winning combination for large language models. In *International Conference on Learning Representations*, 2023.
- [180] Yikang Shen, Zhen Guo, Tianle Cai, and Zengyi Qin. Jetmoe: Reaching llama2 performance with 0.1m dollars. <https://arxiv.org/abs/2404.07413>, 2024.
- [181] Yikang Shen, Zheyu Zhang, Tianyou Cao, Shawn Tan, Zhenfang Chen, and Chuang Gan. Moduleformer: Modularity emerges from mixture-of-experts. <https://arxiv.org/abs/2306.04640>, 2023.
- [182] Shaohuai Shi, Xinglin Pan, Xiaowen Chu, and Bo Li. Pipemoe: Accelerating mixture-of-experts through adaptive pipelining. In *IEEE Conference on Computer Communications*, pages 1–10, 2023.
- [183] Shaohuai Shi, Xinglin Pan, Qiang Wang, Chengjian Liu, Xiaozhe Ren, Zhongzhe Hu, Yu Yang, Bo Li, and Xiaowen Chu. Schemoe: An extensible mixture-of-experts distributed training system with tasks scheduling. In *Proceedings of the Nineteenth European Conference on Computer Systems*, page 236–249. Association for Computing Machinery, 2024.
- [184] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *CoRR*, 2019.
- [185] Siddharth Singh, Olatunji Ruwase, Ammar Ahmad Awan, Samyam Rajbhandari, Yuxiong He, and Abhinav Bhatele. A hybrid tensor-expert-data parallelism approach to optimize mixture-of-experts training. In *Proceedings of the 37th International Conference on Supercomputing*, page 203–214. ACM, 2023.
- [186] Snowflake. Snowflake arctic, 2024.
- [187] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. <https://arxiv.org/abs/2104.09864>, 2023.
- [188] Martynas Subonis. Distributed data parallel training. <https://martynassubonis.substack.com/p/distributed-data-parallel-training>, 2024.
- [189] Martynas Subonis. Tensor and fully sharded data parallelism. <https://martynassubonis.substack.com/p/tensor-and-fully-sharded-data-parallelism>, 2025.

- [190] Sainbayar Sukhbaatar, Hao Xu, and et al. Branch-train-mix: Scaling base models with mixture-of-experts specialization, 2024.
- [191] Shawn Tan, Yikang Shen, Rameswar Panda, and Aaron Courville. Scattered mixture-of-experts implementation. <https://arxiv.org/abs/2403.08245>, 2024.
- [192] Shiyu Tan. Sparse universal transformer. In *EMNLP*, 2023.
- [193] Yehui Tang, Yichun Yin, Yaoyuan Wang, Hang Zhou, Yu Pan, Wei Guo, Ziyang Zhang, Miao Rang, Fangcheng Liu, and et al. Pangu ultra moe: How to train your big moe on ascend npus. <https://arxiv.org/abs/2505.04519>, 2025.
- [194] Nouamane Tazi, Ferdinand Mom, Haojun Zhao, Phuc Nguyen, Mohamed Mekkouri, Leandro Werra, and Thomas Wolf. The ultra-scale playbook: Training llms on gpu clusters. <https://huggingface.co/spaces/nanotron/ultrascale-playbook>, 2025.
- [195] AWS Machine Learning Blog Team. Accelerate mixtral 8x7b pre-training with expert parallelism on amazon sagemaker. <https://aws.amazon.com/blogs/machine-learning/accelerate-mixtral-8x7b-pre-training-with-expert-parallelism-on-amazon-sagemaker/>, 2024.
- [196] DeepSpeed Team. Pipeline parallelism. <https://www.deepspeed.ai/tutorials/pipeline/>, 2026.
- [197] DeepSpeed Team, Rangan Majumder, and Junhua Wang. Deepspeed: Extreme-scale model training for everyone. <https://www.microsoft.com/en-us/research/blog/deepspeed-extreme-scale-model-training-for-everyone/>, 2020.
- [198] Mistral AI Team. Cheaper, better, faster, stronger. <https://mistral.ai/news/mixtral-8x22b/>, 2024.
- [199] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. <https://arxiv.org/pdf/1706.03762.pdf>, 2023.
- [200] Aditya Vavre, Ethan He, Dennis Liu, Zijie Yan, June Yang, Nima Tajbakhsh, and Ashwath Aithal. Llama 3 meets moe: Efficient upcycling. <https://arxiv.org/abs/2412.09952>, 2024.
- [201] Hongyi Wang, Felipe Maia Polo, Yuekai Sun, Souvik Kundu, Eric Xing, and Mikhail Yurochkin. Fusing models with complementary expertise. <https://arxiv.org/abs/2310.01542>, 2024.
- [202] Hulin Wang, Yaqi Xia, Donglin Yang, Xiaobo Zhou, and Dazhao Cheng. Harnessing inter-gpu shared memory for seamless moe communication-computation fusion. *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2025.
- [203] Wei Wang, Zhiquan Lai, Shengwei Li, Weijie Liu, Keshi Ge, Yujie Liu, Ao Shen, and Dongsheng Li. Prophet: Fine-grained Load Balancing for Parallel Training of Large-scale MoE Models. In *2023 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 82–94. IEEE Computer Society, 2023.
- [204] Wei Wang, Zhiquan Lai, Shengwei Li, Weijie Liu, Ke shi Ge, Ao Shen, Huayou Su, and Dongsheng Li. Pro-prophet: A systematic load balancing method for efficient parallel training of large-scale moe models. <https://arxiv.org/abs/2411.10003>, 2024.
- [205] Xin Wang, Fisher Yu, Lisa Dunlap, Yi-An Ma, Ruth Wang, Azalia Mirhoseini, Trevor Darrell, and Joseph E. Gonzalez. Deep mixture of experts via shallow embedding. <https://arxiv.org/abs/1806.01531>, 2019.
- [206] Y. Wang, S. Agarwal, S. Mukherjee, Xiaodong Liu, and Jianfeng Gao. Adamix: Mixture-of-adaptations for parameter-efficient model tuning. In *EMNLP*, 2022.
- [207] Tianyi Wei, Ben Zhu, and Lei Zhao. Skywork-moe. <https://arxiv.org/abs/2406.06563>, 2024.

- [208] Lilian Weng. How to train really large models on many gpus? <https://lilianweng.github.io/posts/2021-09-25-train-large/>, 2021.
- [209] Camaron Wolfe. nanomoe: Mixture-of-experts llms from scratch in pytorch. <https://cameronrwolfe.substack.com/p/nano-moe>, 2025.
- [210] Jialin Wu, Xia Hu, Yaqing Wang, Bo Pang, and Radu Soricut. Omni-smola: Boosting generalist multimodal models with soft mixture of low-rank experts. <https://arxiv.org/abs/2312.00968>, 2023.
- [211] Lemeng Wu, Mengchen Liu, Yinpeng Chen, Dongdong Chen, Xiyang Dai, and Lu Yuan. Residual mixture of experts. <https://arxiv.org/abs/2204.09636>, 2022.
- [212] Shaohua Wu, Jiangang Luo, Xi Chen, Lingjun Li, Xudong Zhao, Tong Yu, Chao Wang, Yue Wang, Fei Wang, Weixu Qiao, Houbo He, Zeru Zhang, Zeyu Sun, Junxiong Mao, and Chong Shen. Yuan 2.0-m32: Mixture of experts with attention router. <https://arxiv.org/abs/2405.17976>, 2024.
- [213] Xun Wu, Shaohan Huang, and Furu Wei. Mixture of lora experts. <https://arxiv.org/abs/2404.13628>, 2024.
- [214] Yongji Wu, Xueshen Liu, Shuowei Jin, Ceyu Xu, Feng Qian, Z. Morley Mao, Matthew Lentz, Danyang Zhuo, and Ion Stoica. Hetermoe: Efficient training of mixture-of-experts models on heterogeneous gpus. <https://arxiv.org/abs/2504.03871>, 2025.
- [215] Yongji Wu, Wenjie Qu, Xueshen Liu, Tianyang Tao, Yifan Qiao, Zhuang Wang, Wei Bai, Yuan Tian, Jiaheng Zhang, Z. Morley Mao, Matthew Lentz, Danyang Zhuo, and Ion Stoica. Lazarus: Resilient and elastic training of mixture-of-experts models. <https://arxiv.org/abs/2407.04656>.
- [216] xAI. Announcing grok-1.5. <https://x.ai/blog/grok-1.5>, 2024.
- [217] Fei Xue, Zhe Zheng, Yang Fu, Jun Ni, Wei Zhou, and Yang You. Openmoe: An early effort on open mixture-of-experts language models. [arXiv:2402.01739](https://arxiv.org/abs/2402.01739), 2024.
- [218] Fuzhao Xue, Xiaoxin He, Xiaozhe Ren, Yuxuan Lou, and Yang You. One student knows all experts know: From sparse to dense. <https://arxiv.org/abs/2201.10890>, 2022.
- [219] Fuzhao Xue, Ziji Shi, Futao Wei, Yuxuan Lou, Yong Liu, and Yang You. Go wider instead of deeper. <https://arxiv.org/abs/2107.11817>, 2021.
- [220] Leyang Xue, Yao Fu, Zhan Lu, Luo Mai, and Mahesh K. Marina. Moe-infinity: Activation-aware expert offloading for efficient moe serving. <https://arxiv.org/abs/2401.14361>, 2024.
- [221] Zijie Yan, Hongxiao Bai, Xin Yao, Dennis Liu, Tong Liu, Hongbin Liu, Pingtian Li, Evan Wu, Shiqing Fan, and et al. Scalable training of mixture-of-experts models with megatron core. <https://arxiv.org/abs/2603.07685>, 2026.
- [222] Rongjie Yi, Liwei Guo, Shiyun Wei, Ao Zhou, Shangguang Wang, and Mengwei Xu. Edgemoe: Empowering sparse large language models on mobile devices. <https://arxiv.org/abs/2308.14352>, 2023.
- [223] Dianhai Yu, Liang Shen, Hongxiang Hao, Weibao Gong, Huachao Wu, Jiang Bian, Lirong Dai, and Haoyi Xiong. Moesys: A distributed and efficient mixture-of-experts training and inference system for internet services. <https://arxiv.org/abs/2205.10034>, 2024.
- [224] Ted Zadouri, Ahmet Ustun, Arash Ahmadian, Beyza Ermis, Acyr Locatelli, and Sara Hooker. Pushing mixture of experts to the limit: Extremely parameter efficient moe for instruction tuning. <https://arxiv.org/abs/2309.05444>, 2023.
- [225] Mingshu Zhai, Jiaao He, Zixuan Ma, Zan Zong, Runqing Zhang, and Jidong Zhai. Smartmoe: Efficiently training sparsely-activated models through combining offline and online parallelization. <https://www.usenix.org/system/files/atc23-zhai.pdf>, 2023.

- [226] Feng Zhang, Weitao Wan, Chenyang Zhang, Jidong Zhai, Yunpeng Chai, Haixiang Li, and Xiaoyong Du. Compressdb: Enabling efficient compressed data direct processing for databases. In *Proceedings of SIGMOD*, page 1655–1669. Association for Computing Machinery, 2019.
- [227] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. Tadic: Text analytics directly on compression. *The VLDB Journal*, 30(2):163–188, 2020.
- [228] Qi Zhang, Jidong Zhai, and Weimin Zheng. A survey on accelerated technologies for mixture-of-experts model training systems. *Tsinghua Science and Technology*, 31(3):1411–1439, 2026.
- [229] Rongyu Zhang, Yijiang Liu, Huanrui Yang, Shenli Zheng, Dan Wang, Yuan Du, Li Du, and Shanghang Zhang. T-rex: Mixture-of-rank-one-experts with semantic-aware intuition for multi-task large language model finetuning. <https://arxiv.org/abs/2404.08985>, 2024.
- [230] Shulai Zhang, Ningxin Zheng, Haibin Lin, Ziheng Jiang, Wenlei Bao, Chengquan Jiang, Qi Hou, Weihao Cui, and et al. Comet: Fine-grained computation-communication overlapping for mixture-of-experts. <https://arxiv.org/abs/2502.19811>, 2025.
- [231] Xiaofeng Zhang, Yikang Shen, Zeyu Huang, Jie Zhou, Wenge Rong, and Zhang Xiong. Mixture of attention heads: Selecting attention heads per token. In *EMNLP*, 2022.
- [232] Yihua Zhang, Ruisi Cai, Tianlong Chen, Guanhua Zhang, Huan Zhang, Pin-Yu Chen, Shiyu Chang, Zhangyang Wang, and Sijia Liu. Robust mixture-of-expert training for convolutional neural networks. <https://arxiv.org/abs/2308.10110>, 2023.
- [233] Zhaopeng Zhang, Zhirui Zhang, and Peng Li. Moefication: Transformer feed-forward layers are mixtures of experts. In *ACL Findings*, 2022.
- [234] Zheng Zhang, Yaqi Xia, Hulin Wang, Donglin Yang, Chuang Hu, Xiaobo Zhou, and Dazhao Cheng. Mpmoe: Memory efficient moe for pre-trained models with adaptive pipeline parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 35(6):998–1011, 2024.
- [235] Zheng Zhang, Donglin Yang, Yaqi Xia, Liang Ding, Dacheng Tao, Xiaobo Zhou, and Dazhao Cheng. Mpipemoe: Memory efficient moe for pre-trained models with adaptive pipeline parallelism. <https://arxiv.org/abs/2506.22175>, 2025.
- [236] Zhengyan Zhang, Zhiyuan Zeng, Yankai Lin, Chaojun Xiao, Xiaozhi Wang, Xu Han, Zhiyuan Liu, Ruobing Xie, Maosong Sun, and Jie Zhou. Emergent modularity in pre-trained transformers. *Findings of ACL*, 2023.
- [237] Hao Zhao, Zihan Qiu, Huijia Wu, Zili Wang, Zhaofeng He, and Jie Fu. Hypermoes: Towards better mixture of experts via transferring among experts. <https://arxiv.org/abs/2402.12656>, 2024.
- [238] Yanli Zhao, Xianyi Chen, and et al. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *Proceedings of the VLDB Endowment*, 2023.
- [239] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. <https://arxiv.org/abs/2201.12023>, 2022.
- [240] Ningxin Zheng, Huiqiang Jiang, Quanlu Zhang, Zhenhua Han, Lingxiao Ma, Yuqing Yang, Fan Yang, Chengruidong Zhang, Lili Qiu, Mao Yang, and Lidong Zhou. Pit: Optimization of dynamic sparse deep learning models via permutation invariant transformation. In *Proceedings of the 29th Symposium on Operating Systems Principles*, page 331–347. Association for Computing Machinery, 2023.
- [241] Size Zheng, Wenlei Bao, Qi Hou, Xuegui Zheng, Jin Fang, Chenhui Huang, Tianqi Li, and et al. Triton-distributed: Programming overlapping kernels on distributed ai systems with the triton compiler. <https://arxiv.org/abs/2504.19442>, 2025.

- [242] Yanqi Zhou, Tao Lei, Hanxiao Liu, Nan Du, Yanping Huang, Vincent Zhao, Andrew Dai, Zhifeng Chen, Quoc Le, and James Laudon. Mixture-of-experts with expert choice routing. <https://arxiv.org/abs/2202.09368>, 2022.
- [243] Yanqi Zhou, Tao Lei, Hanxiao Liu, Nan Du, Yanping Huang, Vincent Zhao, Andrew M Dai, Zhifeng Chen, Quoc V Le, and James Laudon. Mixture-of-experts with expert choice routing. In *NeurIPS*, 2022.
- [244] Yun Zhu, Nevan Wichers, Chu-Cheng Lin, Xinyi Wang, Tianlong Chen, Lei Shu, Han Lu, Canoe Liu, Liangchen Luo, Jindong Chen, and Lei Meng. Sira: Sparse mixture of low rank adaptation. <https://arxiv.org/abs/2311.09179>, 2023.
- [245] Zhanda Zhu, Christina Giannoula, Muralidhar Andoorveedu, Qidong Su, Karttikeya Mangalam, Bojian Zheng, and Gennady Pekhimenko. Mist: Efficient distributed training of large language models via memory-parallelism co-optimization. In *Proceedings of the Twentieth European Conference on Computer Systems*, page 1298–1316. ACM, 2025.
- [246] Barret Zoph, Irwan Bello, Saurabh Kumar, Nan Du, Yanping Huang, Jeff Dean, Noam Shazeer, and William Fedus. St-moe: Designing stable and transferable sparse expert models. [arXiv:2202.08906](https://arxiv.org/abs/2202.08906), 2022.
- [247] Simiao Zuo, Qingru Zhang, Chen Liang, Pengcheng He, Tuo Zhao, and Weizhu Chen. Moebert: from bert to mixture-of-experts via importance-guided adaptation. <https://arxiv.org/abs/2204.07675>, 2022.

Appendix A

Train a Simple Transformer Model with Fully Sharded Data Parallel

This appendix contains the Python code for training a simple Transformer with PyTorch Fully Sharded Data Parallel.

```
1 from dataclasses import dataclass
2 import torch
3 import torch.nn as nn
4 import torch.nn.functional as F
5
6 @dataclass
7 class ModelArgs:
8     n_layers: int = 2
9     vocab_size: int = 8
10    max_seq_len: int = 16
11    dim: int = 16
12    n_heads: int = 4
13    dropout_p: float = 0.1
14
15 class FeedForward(nn.Module):
16     def __init__(self, dim, hidden_dim, dropout_p):
17         super().__init__()
18         self.w1 = nn.Linear(dim, hidden_dim)
19         self.gelu = nn.GELU()
20         self.w2 = nn.Linear(hidden_dim, dim)
21         self.resid_dropout = nn.Dropout(dropout_p)
22
23     def forward(self, x):
24         return self.resid_dropout(self.w2(self.gelu(self.w1(x))))
25
26     def reset_parameters(self):
27         self.w1.reset_parameters()
28         self.w2.reset_parameters()
```

Listing A.1: Python implementation of the transformer feed-forward network.

```

1 class Attention(nn.Module):
2     def __init__(self, args: ModelArgs):
3         super().__init__()
4         assert args.dim % args.n_heads == 0
5         self.head_dim = args.dim // args.n_heads
6         self.n_heads = args.n_heads
7         self.dropout_p = args.dropout_p
8         self.resid_dropout = nn.Dropout(args.dropout_p)
9
10        self.wq = nn.Linear(args.dim, args.dim, bias=False)
11        self.wk = nn.Linear(args.dim, args.dim, bias=False)
12        self.wv = nn.Linear(args.dim, args.dim, bias=False)
13        self.wo = nn.Linear(args.dim, args.dim, bias=False)
14
15    def forward(self, x):
16        bsz, seq_len, _ = x.size()
17        queries, keys, values = self.wq(x), self.wk(x), self.wv(x)
18        queries = queries.view(bsz, seq_len, self.n_heads, self.head_dim)
19        keys = keys.view(bsz, seq_len, self.n_heads, self.head_dim)
20        values = values.view(bsz, seq_len, self.n_heads, self.head_dim)
21
22        queries = queries.transpose(1, 2) # (bsz, n_heads, seq_len, head_dim)
23        keys = keys.transpose(1, 2) # (bsz, n_heads, seq_len, head_dim)
24        values = values.transpose(1, 2) # (bsz, n_heads, seq_len, head_dim)
25
26        output = F.scaled_dot_product_attention(
27            queries,
28            keys,
29            values,
30            None,
31            self.dropout_p if self.training else 0,
32        )
33        output = output.transpose(1, 2).contiguous().view(bsz, seq_len, -1)
34        return self.resid_dropout(self.wo(output))
35
36    def reset_parameters(self):
37        self.wq.reset_parameters()
38        self.wk.reset_parameters()
39        self.wv.reset_parameters()
40        self.wo.reset_parameters()

```

Listing A.2: Python implementation of the transformer attention layer.

```

1 class TransformerBlock(nn.Module):
2     def __init__(self, args: ModelArgs):
3         super().__init__()
4         self.attention_norm = nn.LayerNorm(args.dim)
5         self.attention = Attention(args)
6         self.ffn_norm = nn.LayerNorm(args.dim)
7         self.feed_forward = FeedForward(
8             args.dim, hidden_dim=4 * args.dim, dropout_p=args.dropout_p
9         )
10
11    def forward(self, x):
12        h = x + self.attention(self.attention_norm(x))
13        out = h + self.feed_forward(self.ffn_norm(h))
14        return out
15
16    def reset_parameters(self):
17        self.attention_norm.reset_parameters()
18        self.attention.reset_parameters()
19        self.ffn_norm.reset_parameters()
20        self.feed_forward.reset_parameters()

```

Listing A.3: Python implementation of the transformer block.


```

1 # A simple transformer model inspired by the nanoGPT model:
2 # https://github.com/karpathy/nanoGPT.
3 class Transformer(nn.Module):
4     def __init__(self, args: ModelArgs):
5         super().__init__()
6         assert args.vocab_size is not None
7         assert args.max_seq_len is not None
8         self.model_args = args
9         self.max_seq_len = args.max_seq_len
10        self.tok_embeddings = nn.Embedding(args.vocab_size, args.dim)
11        self.pos_embeddings = nn.Embedding(args.max_seq_len, args.dim)
12        self.dropout = nn.Dropout(args.dropout_p)
13        self.layers = nn.ModuleList()
14        for _ in range(args.n_layers):
15            self.layers.append(TransformerBlock(args))
16        self.norm = nn.LayerNorm(args.dim)
17        self.output = nn.Linear(args.dim, args.vocab_size, bias=False)
18
19    def forward(self, tokens):
20        _bsz, seq_len = tokens.size()
21        assert seq_len <= self.max_seq_len
22        h = self.tok_embeddings(tokens)
23        pos = torch.arange(0, seq_len, device=tokens.device)
24        p = self.pos_embeddings(pos) # positional embeddings of shape (seq_len, dim)
25        h = h + p
26        h = self.dropout(h)
27        for layer in self.layers:
28            h = layer(h)
29        h = self.norm(h)
30        output = self.output(h).float()
31        return output
32
33    def reset_parameters(self):
34        self.tok_embeddings.reset_parameters()
35        self.pos_embeddings.reset_parameters()
36        self.norm.reset_parameters()
37        self.output.reset_parameters()

```

Listing A.4: Python implementation of a simple transformer model.

```

1 import torch
2 from model import Transformer
3 from torch.distributed.fsdp import FSDPModule
4 from torch.distributed.tensor import Shard
5
6 # Verify if the model is correctly wrapped with FSDP2
7 def inspect_model(model: FSDPModule):
8     assert isinstance(model, Transformer)
9     assert isinstance(model, FSDPModule)
10
11     if torch.distributed.get_rank() == 0:
12         print(model)
13
14     for param in model.parameters():
15         assert param.placements == (Shard(0),)
16         assert param.dtype == torch.float32
17         # print(param.get_local_tensor())
18
19 # Verify if mixed precision was correctly applied
20 def inspect_mixed_precision(model: FSDPModule):
21     model.unshard()
22     for param in model.parameters(recurse=False):
23         assert param.dtype == torch.bfloat16
24     model.reshard()

```

Listing A.5: Utilities to inspect the model wrapped by FSDP2 and the mixed precision.

```

1 import os
2 import time
3
4 import torch
5 import torch.nn as nn
6 from torch.distributed.checkpoint.state_dict import (
7     _init_optim_state ,
8     get_model_state_dict ,
9     get_optimizer_state_dict ,
10    set_model_state_dict ,
11    set_optimizer_state_dict ,
12    StateDictOptions ,
13 )
14 from torch.distributed.fsdp import FSDPModule
15 from torch.distributed.tensor import distribute_tensor , DTensor
16
17
18 MODEL_CHECKPOINT = "model_state_dict.pt"
19 OPTIM_CHECKPOINT = "optim_state_dict.pt"
20 PARAMS           = "params"
21
22
23 def get_latest_checkpoint_folder(path):
24     max_num = None
25     if not os.path.exists(path):
26         return max_num
27     for name in os.listdir(path):
28         folder_path = os.path.join(path, name)
29         if os.path.isdir(folder_path):
30             try:
31                 num = int(name)
32                 if max_num is None or num > max_num:
33                     max_num = num
34             except ValueError:
35                 pass # Skip non-numeric folder names
36     return max_num
37
38
39 class Checkpointer:
40     def __init__(self, folder: str, dcp_api: bool):
41         self.folder = folder
42         self.dcp_api = dcp_api
43         self.last_training_time = get_latest_checkpoint_folder(
44             f"{folder}/{ 'dcp_api' if dcp_api else 'dtensor_api' }"
45         )
46
47     def is_empty(self):
48         return self.last_training_time is None

```

Listing A.6: Class for saving and loading model and optimizer state dictionaries to/from file (part 1).

```

1  def load_model(self, model: FSDPModule):
2      last_model_checkpoint = (
3          f"{self.folder}/{ 'dcp_api' if self.dcp_api else 'dtensor_api' }"
4          f"/{self.last_training_time}/{MODEL_CHECKPOINT}"
5      )
6      full_sd = torch.load(
7          last_model_checkpoint,
8          mmap = True,
9          weights_only = True,
10         map_location = "cpu"
11     )
12     if self.dcp_api:
13         set_model_state_dict(
14             model = model,
15             model_state_dict = full_sd,
16             options = StateDictOptions(
17                 full_state_dict = True,
18                 broadcast_from_rank0 = True,
19             ),
20         )
21     return
22
23     meta_sharded_sd = model.state_dict()
24     sharded_sd = {}
25     for param_name, full_tensor in full_sd.items():
26         sharded_meta_param = meta_sharded_sd.get(param_name)
27         sharded_tensor = distribute_tensor(
28             full_tensor,
29             sharded_meta_param.device_mesh,
30             sharded_meta_param.placements,
31         )
32         sharded_sd[param_name] = nn.Parameter(sharded_tensor)
33     # choose 'assign=True' since we cannot call 'copy_' on meta tensor
34     model.load_state_dict(sharded_sd, strict=False, assign=True)

```

Listing A.7: Class for saving and loading model and optimizer state dictionaries to/from file (part 2).

```

1  def load_optim(self, model: FSDPModule, opt: torch.optim.Optimizer):
2      last_optim_checkpoint = (
3          f"{self.folder}/{ 'dcp_api' if self.dcp_api else 'dtensor_api' }"
4          f"/{ self.last_training_time }/{ OPTIM_CHECKPOINT }"
5      )
6      full_sd = torch.load(
7          last_optim_checkpoint, mmap=True, weights_only=True, map_location="cpu"
8      )
9      if self.dcp_api:
10         set_optimizer_state_dict(
11             model = model,
12             optimizers = opt,
13             optim_state_dict = full_sd,
14             options = StateDictOptions(
15                 full_state_dict = True,
16                 broadcast_from_rank0 = True,
17             ),
18         )
19         return
20     _init_optim_state(opt)
21     param_groups = opt.state_dict()["param_groups"]
22     state = opt.state_dict()["state"]
23
24     full_param_groups = full_sd["param_groups"]
25     full_state = full_sd["state"]
26
27     for param_group, full_param_group in zip(param_groups, full_param_groups):
28         for key, value in full_param_group.items():
29             if key == PARAMS:
30                 continue
31             param_group[key] = value
32
33         for pid, full_pid in zip(param_group[PARAMS], full_param_group[PARAMS]):
34             if pid not in state:
35                 continue
36             param_state = state[pid]
37             full_param_state = full_state[full_pid]
38             for attr, full_tensor in full_param_state.items():
39                 sharded_tensor = param_state[attr]
40                 if isinstance(sharded_tensor, DTensor):
41                     # exp_avg is DTensor
42                     param_state[attr] = distribute_tensor(
43                         full_tensor,
44                         sharded_tensor.device_mesh,
45                         sharded_tensor.placements,
46                     )
47                 else:
48                     # step is plain tensor
49                     param_state[attr] = full_tensor
50
51     opt.load_state_dict(
52         {
53             "param_groups": param_groups,
54             "state": state,
55         }
56     )

```

Listing A.8: Class for saving and loading model and optimizer state dictionaries to/from file (part 3).

```

1  def _get_full_model_state_dict(self, model: FSDPModule):
2      if self.dcp_api:
3          return get_model_state_dict(
4              model = model,
5              options = StateDictOptions(
6                  full_state_dict = True,
7                  cpu_offload = True,
8              ),
9          )
10
11     sharded_sd = model.state_dict()
12     cpu_state_dict = {}
13     for param_name, sharded_param in sharded_sd.items():
14         full_param = sharded_param.full_tensor()
15         if torch.distributed.get_rank() == 0:
16             cpu_state_dict[param_name] = full_param.cpu()
17         else:
18             del full_param
19     return cpu_state_dict
20
21 def _get_full_optimizer_state_dict(
22     self,
23     model: FSDPModule,
24     opt: torch.optim.Optimizer,
25 ):
26     if self.dcp_api:
27         return get_optimizer_state_dict(
28             model = model,
29             optimizers = opt,
30             options = StateDictOptions(
31                 full_state_dict=True,
32                 cpu_offload=True,
33             ),
34         )
35     is_rank_zero = torch.distributed.get_rank() == 0
36     sharded_sd = opt.state_dict()
37     sharded_state = sharded_sd["state"]
38     full_state = {}
39     for group_id, sharded_group in sharded_state.items():
40         group_state = {}
41         for attr, sharded_tensor in sharded_group.items():
42             if isinstance(sharded_tensor, DTensor):
43                 # "exp_avg" in AdamW is 'DTensor'
44                 full_tensor = sharded_tensor.full_tensor()
45             else:
46                 # "step" in AdamW is plain tensor
47                 full_tensor = sharded_tensor
48             if is_rank_zero:
49                 group_state[attr] = full_tensor.cpu()
50             else:
51                 del full_tensor
52         if is_rank_zero:
53             full_state[group_id] = group_state
54         else:
55             del group_state
56     if is_rank_zero:
57         return {
58             "param_groups": sharded_sd["param_groups"],
59             "state": full_state,
60         }
61     else:
62         return {}

```

Listing A.9: Class for saving and loading model and optimizer state dictionaries to/from file (part 4).

```

1  def save(self, model: FSDPModule, optim: torch.optim.Optimizer):
2      model_state_dict = self._get_full_model_state_dict(model)
3      optim_state_dict = self._get_full_optimizer_state_dict(model, optim)
4      if torch.distributed.get_rank() == 0:
5          new_training_time = int(time.time() * 1000)
6          new_checkpoint_folder = f"{self.folder}/{ 'dcp_api' if self.dcp_api
7                                  else 'dtensor_api'}/{ new_training_time}"
8          new_model_checkpoint = f"{new_checkpoint_folder}/{MODEL_CHECKPOINT}"
9          new_optim_checkpoint = f"{new_checkpoint_folder}/{OPTIM_CHECKPOINT}"
10         os.makedirs(new_checkpoint_folder, exist_ok=True)
11         torch.save(model_state_dict, new_model_checkpoint)
12         torch.save(optim_state_dict, new_optim_checkpoint)

```

Listing A.10: Class for saving and loading model and optimizer state dictionaries to/from file (part 5).

```

1  import argparse
2  import os
3  import torch
4  from checkpoint import Checkpointer
5  from model import ModelArgs, Transformer
6  from torch.distributed.fsdcp import fully_shard, MixedPrecisionPolicy
7  from utils import inspect_mixed_precision, inspect_model
8
9  def verify_min_gpu_count(min_gpus: int = 2) -> bool:
10     """
11     Verify that we have at least 2 GPUs to run this distributed example.
12     """
13     has_gpu = torch.accelerator.is_available()
14     gpu_count = torch.accelerator.device_count()
15     return has_gpu and gpu_count >= min_gpus
16
17 # .....
18 def set_modules_to_forward_prefetch(model, num_to_forward_prefetch):
19     """
20     Set prefetch AllGather operations over model layers in forward pass.
21     """
22     for i, layer in enumerate(model.layers):
23         if i >= len(model.layers) - num_to_forward_prefetch:
24             break
25         layers_to_prefetch = [
26             model.layers[i + j] for j in range(1, num_to_forward_prefetch + 1)
27         ]
28         layer.set_modules_to_forward_prefetch(layers_to_prefetch)
29
30 # .....
31 def set_modules_to_backward_prefetch(model, num_to_backward_prefetch):
32     """
33     Set prefetch AllGather operations over model layers in backward pass.
34     """
35     for i, layer in enumerate(model.layers):
36         if i < num_to_backward_prefetch:
37             continue
38         layers_to_prefetch = [
39             model.layers[i - j] for j in range(1, num_to_backward_prefetch + 1)
40         ]
41         layer.set_modules_to_backward_prefetch(layers_to_prefetch)

```

Listing A.11: Set prefetch AllGather operations over model layers.

```

1 def main(args):
2     _min_gpu_count = 2
3     if not verify_min_gpu_count(min_gpus=_min_gpu_count):
4         print(f"Unable to locate sufficient {_min_gpu_count} GPUs to run this example.")
5         exit()
6
7     rank = int(os.environ["LOCAL_RANK"])
8     if torch.accelerator.is_available():
9         device_type = torch.accelerator.current_accelerator()
10        device = torch.device(f"{device_type}:{rank}")
11        torch.accelerator.device_index(rank)
12        print(f"Running rank {rank} on device {device}")
13    else:
14        device = torch.device("cpu")
15        print(f"Running on device {device}")
16
17    # Initialize the default process group and communication backend.
18    backend = torch.distributed.get_default_backend_for_device(device)
19    torch.distributed.init_process_group(backend=backend, device_id=device)
20
21    # Configure a Transformer model with 10 layers, 4 attention heads,
22    # a maximum sequence length of 64, no dropout, and a vocabulary size of 1024.
23    torch.manual_seed(0)
24    vocab_size = 1024
25    batch_size = 32
26    seq_len = 64
27    model_args = ModelArgs(
28        n_layers = 10,
29        n_heads = 4,
30        vocab_size = vocab_size,
31        max_seq_len = seq_len,
32        dropout_p = 0,
33    )
34
35    # Instantiate the model on the meta device to avoid unnecessary memory usage, since
36    # FSDP2 will take care of moving parameters to the correct device and shard them.
37    with torch.device("meta"):
38        model = Transformer(model_args)
39
40    fsdp_kwargs = {}
41    if args.mixed_precision:
42        fsdp_kwargs["mp_policy"] = MixedPrecisionPolicy(
43            param_dtype = torch.bfloat16,
44            reduce_dtype = torch.float32,
45        )
46
47    # Wrap each layer with FSDP2 and then the entire model.
48    # This allows for more fine-grained control over sharding and prefetching.
49    for layer in model.layers:
50        fully_shard(layer, **fsdp_kwargs)
51    fully_shard(model, **fsdp_kwargs)
52
53    # Print the model to verify that it is correctly wrapped with FSDP2.
54    inspect_model(model)
55
56    # Optionally set up explicit prefetching for the forward and backward passes.
57    # With prefetching, FSDP modules should explicitly prefetch AllGathers in forward.
58    # The prefetching runs after this module's AllGather copy-out.
59
60    if args.explicit_prefetching:
61        set_modules_to_forward_prefetch(model, num_to_forward_prefetch=2)
62        set_modules_to_backward_prefetch(model, num_to_backward_prefetch=2)

```

Listing A.12: The main function to train the transformer model with FSDP2 (part 1).

```

1  # If a checkpoint exists, load the model and optimizer state from the checkpoint.
2  checkpointer = Checkpointer("checkpoints", dcp_api=args.dcp_api)
3  if checkpointer.last_training_time is None:
4      model.to_empty(device=device)
5      model.reset_parameters()
6  else:
7      checkpointer.load_model(model)
8
9  # If mixed precision is enabled, inspect the model to verify that
10 # weights are in the correct data type.
11 if args.mixed_precision:
12     inspect_mixed_precision(model)
13
14 # Construct the optimizer based on the model parameters. If a checkpoint was loaded,
15 # load the optimizer state from the checkpoint as well.
16 optim = torch.optim.Adam(model.parameters(), lr=1e-2)
17 if checkpointer.last_training_time is not None:
18     checkpointer.load_optim(model, optim)
19
20 # Run a simple training loop for 10k iterations, where in each iteration we:
21 # 1. Optionally unshard the model if explicit prefetching is enabled
22 # 2. Generate random input data
23 # 3. Compute the loss and perform the backward pass
24 # 4. Clip the gradients and update the model weights using the optimizer
25 # 5. Reset the optimizer gradients
26
27 for _ in range(10000):
28     if args.explicit_prefetching:
29         model.unshard()
30     x = torch.randint(0, vocab_size, (batch_size, seq_len), device=device)
31     loss = model(x).sum()
32     loss.backward()
33     torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
34     optim.step()
35     optim.zero_grad()
36
37 # Save the model and optimizer state to a checkpoint,
38 # and then clean up the process group.
39 checkpointer.save(model, optim)
40 torch.distributed.destroy_process_group()

```

Listing A.13: The main function to train the transformer model with FSDP2 (part 2).