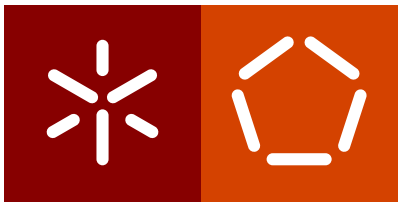
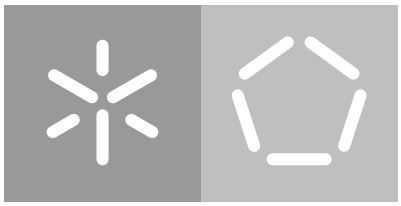


**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

João Carlos Faria Padrão

# Fault Tolerant Decentralized Deep Neural Networks

October 2020



**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

João Carlos Faria Padrão

# Fault Tolerant Decentralized Deep Neural Networks

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
**Carlos Baquero**
**Vitor Enes**

October 2020

# ACKNOWLEDGEMENTS

I would like to thank my family, specially my father, my mother and my sister for supporting my during the last years of my academic path. I would also like to give a special thanks to my friends Carlos, Hugo, Luís, Pedro, Renato and Sequeira for all the knowledge they shared with me and for listening to my thoughts.

And last, I would like to give a very special thanks to my supervisors, Vitor Enes and professor Carlos Baquero for always being present and helpful. Without them this thesis would exist, so I thank you from the bottom of my heart.

## STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration. I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho

## ABSTRACT

Machine Learning is trending in computer science, especially Deep Learning. Training algorithms that follow this approach to Machine Learning routinely deal with vast amounts of data. Processing these enormous quantities of data requires complex computation tasks that can take a long time to produce results. Distributing computation efforts across multiple machines makes sense in this context, as it allows conclusive results to be available in a shorter time frame.

Distributing the training of a Deep Neural Network is not a trivial procedure. Various architectures have been proposed, following two different paradigms. The most common one follows a centralized approach, where a centralized entity, broadly named parameter server, synchronizes and coordinates the updates generated by a number of workers. The alternative discards the centralized unit, assuming a decentralized architecture. The synchronization between the multiple workers is assured by communication techniques that average gradients between a node and its peers.

High-end clusters are the ideal environment to deploy Deep Learning systems. Low latency between nodes assures low idle times for workers, increasing the overall system performance. These setups, however, are expensive and are only available to a limited number of entities. On the other end, there is a continuous growth of edge devices with potentially vast amounts of available computational resources.

In this dissertation, we aim to implement a fault tolerant decentralized Deep Neural Network training framework, capable of handling the high latency and unreliability characteristic of edge networks. To manage communication between nodes, we employ decentralized algorithms capable of estimating parameters globally.

**Keywords**: Distributed Systems, Machine Learning, Artificial Intelligence, Fault Tolerance.

# RESUMO

Machine Learning, mais especificamente Deep Learning, é um campo emergente nas ciências da computação. Algoritmos de treino aplicados em Deep Learning lidam muito frequentemente com vastas quantidades de dados. Processar estas enormes quantidades de dados requer operações computacionais complexas que demoram demasiado tempo para produzir resultados. Distribuir o esforço computacional por múltiplas máquinas faz todo o sentido neste contexto e permite um aumento significativo de desempenho.

Distribuir o método de treino de uma rede neuronal não é um processo trivial. Várias arquiteturas têm sido propostas, seguindo dois diferentes paradigmas. O mais comum segue uma abordagem centralizada, onde uma entidade central, normalmente denominada de *parameter server*, sincroniza e coordena todas as atualizações produzidas pelos *workers*. A alternativa passa por descartar a entidade centralizada, assumindo uma arquitetura descentralizada. A sincronização entre *workers* é assegurada através de estratégias de comunicação descentralizadas.

Clusters de alta performance são o ambiente ideal para a implementação de sistemas de Deep Learning. A baixa latência entre nodos assegura baixos períodos de inatividade nos *workers*, aumentando assim o rendimento do sistema. Estas instalações, contudo, são muito custosas, estando apenas disponíveis para um pequeno número de entidades. Por outro lado, o número de equipamentos nas extremidades da rede, com baixo aproveitamento de poder computacional, continua a crescer, o que torna o seu uso desejável.

Nesta dissertação, visamos implementar um ambiente de treino de redes neuronais descentralizado e tolerante a faltas, apto a lidar com alta latência na comunicações e baixa estabilidade nos nodos, caraterística de redes na extremidade. Para coordenar a comunicação entre os nodos, empregamos algoritmos de agregação, capazes de criar uma visão geral de parametros numa topologia.

**Palavras Chave**: Sistemas Distribuídos, Machine Learning, Inteligência Artificial, Tolerância a Faltas.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# 1

# INTRODUCTION

## 1.1 CONTEXT

Machine Learning has been powering an increasingly vast amount of services, ranging from recommendation systems [7] to pattern recognition [19] and others. One of the enablers of this growth is the ever-increasing amount of data available to analyzed and processed by services that satisfy such purposes. These large quantities of data allow the employment of larger and more complex models that tend to generate better accuracies, resulting in more relevant results.

Neural networks are a specific set of algorithms that have revolutionized machine learning. Traditional Neural Network algorithms execute a sequence of steps that gradually minimize a loss function while consuming a dataset. This process, known as training, is obviously slow when dealing with reasonably sized datasets or models. To mitigate the inevitable increases in execution times, the community resorted to concurrent/distributed computing, in order to split the computational effort across multiple machines.

The most common approach to distributed Neural Network training is the parameter server concept. This architecture relies on a centralized computing unit, the parameter server, that coordinates the communication between all the nodes that compose the cluster. The alternative is to eliminate the parameter server and allow the workers on the cluster to freely communicate between them, in a decentralized manner.

## 1.2 MOTIVATION

Leading Neural Network frameworks are implemented on high-end servers, equipped with state of the art GPUs. On the other end of the spectrum, there is a continuous expansion of edge devices with potentially unused computation power. Additionally, the continuous expansion and improvement of networks and infrastructures, facilitates the development of distributed applications that run on the edge. The possibility of using these devices in this context has already been considered by Hardy et al. [11], but the method described relies

on a centralized server to coordinate the training. This raises a question: *Can decentralized distributed Machine Learning be feasible on edge devices?*.

Decentralized architectures allow end users to have total control over their contributions to the network. Therefore, there is a higher degree of privacy, since inputs are processed locally, without the need to be transferred to a centralized server.

The prospect of vast amounts of worker nodes reinforces the urge for a decentralized solution, to avoid the predictable bottleneck observed when a server is responsible for processing an enormous amount of requests. This congestion can be witnessed either on the communication network, or in the lack of processing capability to respond to all the requests efficiently.

## 1.3 Main Contributions

In this thesis we aim to implement an efficient decentralized distributed Machine Learning framework, designed to run on an unreliable environment. This framework should capable of reacting to failures in the network, while providing acceptable results in a reasonable time.

## 1.4 Dissertation Outline

This dissertation is organized as follows. Chapter 2 presents a summary of machine learning. Chapter 3 presents an overview of the challenges and approaches to distributed Deep Learning. Chapters 4 and 5 point out the most widely used techniques used in centralized and decentralized Deep learning, respectively. In chapter 6 we introduce our novel approach to distributed Deep Learning and on chapter 7 we evaluate the algorithm presented in the previous chapter. Finally in chapter 8 we conclude this thesis and share some ideas for future research.

# 2

# MACHINE LEARNING

We start this thesis by giving an overview of machine learning along with its different branches and challenges. The aim of this chapter is not dive deep into the technicalities of machine learning, but to give the reader the necessary context to understand the ideas and decisions taken during the development of this dissertation.

## 2.1 DEFENITION

Machine learning (ML), is a field of Artificial Intelligence that allows a computer system to achieve a goal without the need to specify the steps necessary to reach it. Instead, the system resorts to learning and inference techniques. Due to its wide spectrum, machine learning has branched into numerous subfields that can be classified by several parameters. The most direct distinction is between supervised and unsupervised learning. Supervised learning occurs when the learning task is supplied with a set of labeled data denoted as training data. This data is assumed to be correctly labeled and indicates whether the system is producing the expected results. The knowledge extracted from the training data is then used to make a prediction based on unseen data. Unsupervised learning occurs when the learning task is supplied with a set of unlabeled data. Here the task is to infer certain patterns or rules that allow the system to make predictions based on unseen data. Imagine we want to build a system that determines whether a picture contains a car. In a supervised learning setting, the system would be trained with a set of labeled images that identifies each image has to having a car or not. In an unsupervised setting, however, the training data would not be labeled, and it would be up to the algorithm to identify patterns that define a car. Supervised learning is the most common form of machine learning, so unsupervised learning will not be considered in this thesis.

Another important classifier of a machine learning task is the distinction between batch and online learning. A batch learning task trains over large amounts of data before being ready to make predictions. On the other hand, in online learning there is no established training period. As a result training and prediction operations can execute at the same time. This type of algorithms are often used when data is available incrementally, as opposed to

batch training where train datasets are available to be processed. Batch training is the more common than online learning, so it will be the focus of this thesis.

## 2.2 DEEP LEARNING

The emergence of machine learning is in part due to the evolution of deep learning, the most popular machine learning field. Deep learning algorithms use deep learning neural networks (DNNs), a type of artificial neural networks (ANNs). Artificial neural networks draw inspiration from biological neural networks that typically constitute animal brains. ANNs are comprised of neurons, typically organized in layers connected by a set of edges [2]. The organization of these layers and edges can vary depending on the type of network. ANNs whose neurons are connected to neurons of the next layer are called feedforward networks. Figure 1 represents a simple feedforward network. ANNs that dont restrict the orientation of connections between nodes are called recurrent networks.

Deep Neural Networks are a specific class of ANNs with multiple layers between the input and the output layers. The input layer being the first layer of the network and the output layer the last. The layers between the output and input layers are refered as hidden layers.

The neurons of the network are organized in layers. Each layer contains a nonempty set of neurons. Let us denote the set containing all the edges as $E$, the set containing all the layers, such that $V = \bigcup_{t=0}^{T} V_t$ as $V$ and the number of layers as $T$. In a feedforward network, every edge in $E$ that has its origin in a layer $V_{t-1}$ can only be connected to a node belonging to the layer $V_t$. In a recurring network, every edge in $E$ that has its origin in a layer $V$ can be connected to a node belonging to any layer in $V$. The first layer, $V_0$, is the input layer. It contains $m + 1$ neurons, where $m$ is the amplitude of the input. The additional neuron is the bias neuron which always outputs 1. Layers $V_1$ to $V_{T-1}$ are the hidden layers. The final layer, $V_T$ is the output layer.

Each neuron accommodates a simple activation function, responsible for determining the node's output. This function normalizes the weighted sum of the incoming connections, producing a measure of how positive the weighted sum is. Biases can also be used to calculate the output value of a neuron, together with the return of the activation function. A bias is an extra neuron per layer that does not have any incoming connections and outputs the value 1. This additional factor is used to improve the calibration of the output.

Consider a Neural Network and a labeled training dataset $\{\langle x_1, l_1\rangle, \ldots, \langle x_n, l_n\rangle\}$. When the network is presented with the input $x_i$, it produces an output $y_i$, that, in most cases, can be very different from the expected value $l_i$. To improve the percentage of correct predictions from a Neural Network, it is necessary to minimize the cost function, $\Delta$, represented by the network, typically defined as the sum of square differences:

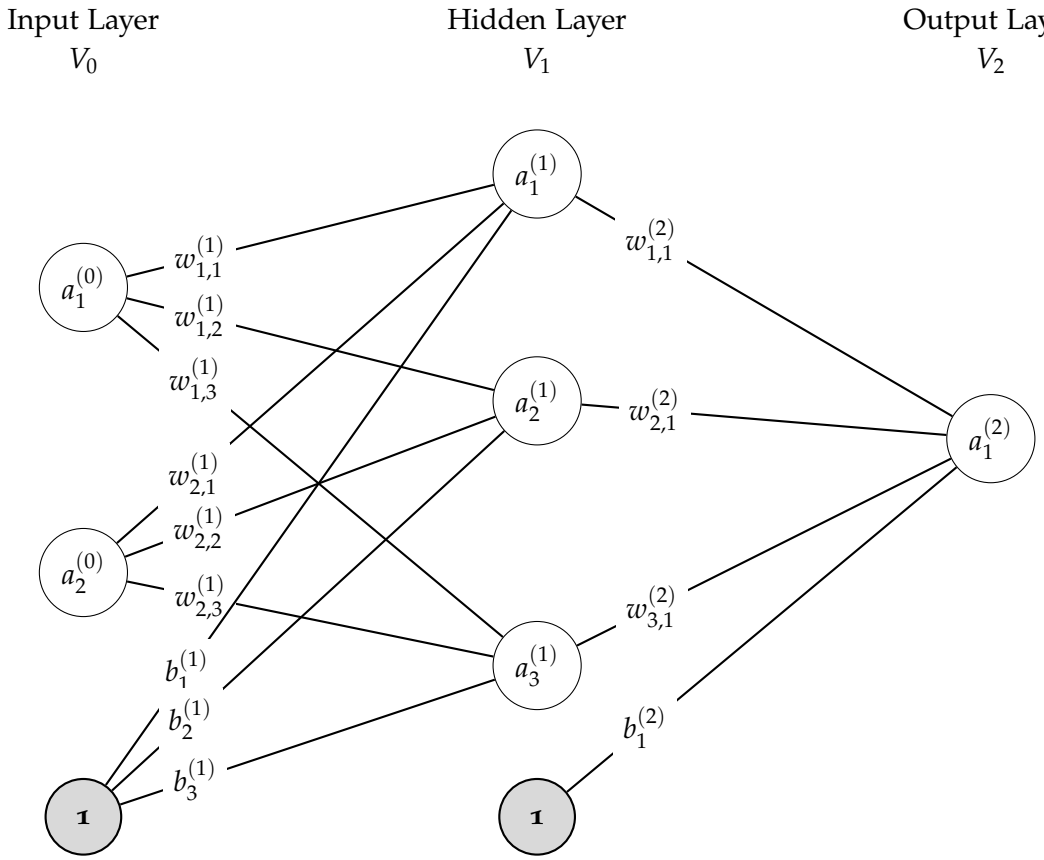



Figure 1: Simple feedforward neural network with 3 layers

$$\Delta = \frac{1}{n}\sum_{i=1}^{n}(l_i - y_i)^2$$

The predominant minimization heuristic used to train Neural Networks is Stochastic Gradient Decent (SGD), based on the Gradient Decent method. Gradient Decent is an iterative optimization procedure that, at each iteration, improves the solution by taking a step along the negative gradient of the function to be minimized, at the current point [22]. Since we do not have access to the full domain, $D$, of the problem, this procedure is not feasible in this case. SGD bypasses that limitation by allowing the step to be taken along a random gradient, based on a sample of the domain $D$. This gradient is calculated using the backpropagation algorithm. With the gradient calculated, the network can be optimized.

## 2.3 SUMMARY

In this Chapter, we presented a broad definition of Machine Learning and one of its subfields, Deep Learning. We also covered the basic structure of a neural network and how these

structures are trained. In the next Chapter, we will discuss the need to distribute Deep Learning and various techniques used to achieve that.

# 3

# DISTRIBUTED DEEP LEARNING

As we saw in the previous Chapter, most of the computational effort required to train a deep neural network is the result of basic linear algebra transformations. To accelerate the training process, it is necessary to distribute this operation efficiently. As other large-scale computational systems, there are two different alternatives to distribute approach this challenge: vertical scaling or horizontal scaling.

## 3.1 VERTICAL SCALING

Vertical scaling involves adding more resources to a single machine. The emergence of deep learning has lead vendors such as Nvidia, to develop GPUs with versatile architectures that better accommodate the need for highly parallel tasks. TPUs [13] also yield high performance while executing highly parallel tasks. These processing units specializes in transformations of multi-dimensional array structures, and are usually used in combination with TensorFlow [1].

## 3.2 HORIZONTAL SCALING

Horizontal scaling involves partitioning tasks across multiple machines. To achieve desirable performance, ML algorithms need to be adapted to a distributed setting. This process is not always straightforward and often presents problems that affect most distributed applications. Overcoming these challenges, however, is desirable for several reasons. The first reason is the increase in fault tolerance because, in the event of a failure, the system can continue to operate. Another reason relates to the high I/O demand of deep neural networks. Partitioning data across several machines increases the total I/O bandwidth of the system. In this thesis we will focus on horizontal scaling.

## 3.3 DISTRIBUTED DEEP LEARNING ARCHITECTURES

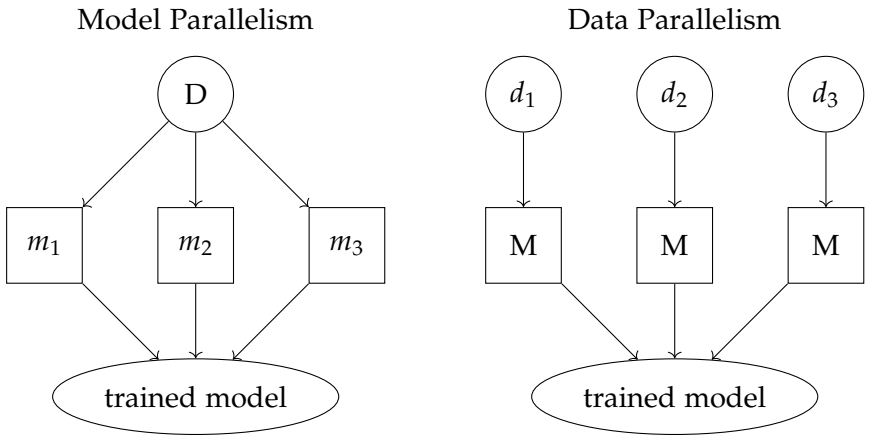
When we are distributing computation across several machines, it is important to consider all of the alternative ways of accomplishing it. Depending on the available hardware or on the model itself, one might find some techniques more suited to the problem than others. In distributed deep learning, the first decision falls on whether to implement model or data parallelism.

### 3.3.1 *Model Parallelism*

Model parallelism, represented in Figure 2a, dictates that the model must be split across all machines and that all workers process the same data. To get a holistic view of the model it is necessary to aggregate all the portions split across the workers. This approach tends to yield greater performance with models with local connectivity structures [9].

### 3.3.2 *Data Parallelism*

Data parallelism, represented in Figure 2b, dictates that the data must be split across all machines and that all workers have a copy of the same model. Each worker processes different data batches using the same model. This approach supports all deep learning algorithms.




(a) Model Parallelism (b) Data Parallelism

Figure 2: Two types of parallelism in Distributed Deep Learning. Data is represented by the letter D, or $d_n$ if split, and the model is represented by the letter M, or $m_n$ if split.

### 3.3.3 *Summary*

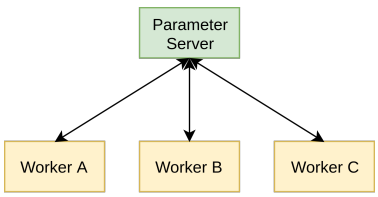
As with most distributed systems, the arrangement of the workers is particularly important to ensure good performance and coordination. Topologies fall under two major categories: centralized and decentralized. In the next two Chapters we will study these classes and present several examples that implement these concepts.

# 4

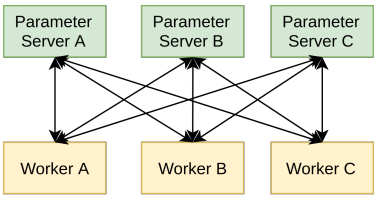
# CENTRALIZED DISTRIBUTED DEEP LEARNING

## 4.1 PARAMETER SERVER

Most relevant distributed Machine Learning frameworks, such as TensorFlow[1] [1], MXNet[2] [5] or CNTK[3], support the centralized parameter server architecture [17]. This architecture, illustrated in Figure 3, is devised to allow workers to calibrate their own parameters according to a portion of the dataset, and then synchronize their variables with the central parameter server (PS), establishing a starting point for the next round. The PS architecture provides a global view of the system, allowing the parameters to be stored on a persistent data store. In the event of a worker failure, the model parameters stored on the PS are used to restore the malfunctioning worker the current state.




(a) Parameter Sever architecture with a single Parameter Server




(b) Parameter Sever architecture with multiple Parameter Servers

Figure 3: Parameter Server architecture

With a small number of workers, it is possible to achieve a near-linear boost in performance. However, a further increase in the number of workers can expose the bandwidth limitations of the communication layer due to a large amount of data being sent through the same channels. One solution is to increase the batch size, leading to fewer synchronization steps. A larger global batch, though, can decrease the efficiency of the model [14]. Additionally, GPUs have limited memory, which diminishes the viability of large batches of data. To further hinder the training performance, current developments in accelerators and networks suggest an ever evident disparity between computation and communication speeds. New

1 https://www.tensorflow.org/
2 https://mxnet.apache.orrg/
3 https://www.microsoft.com/en-us/cognitive-toolkit/

hardware and algorithms continue to reduce the computation time, while network speeds continue getting faster but at a much slower rate. Another solution is to implement sharded parameter servers [6, 9] that divide the ownership of the model parameters. This design leads to a waste of potentially expensive computational resources.

Synchronous versions of the parameter server concept guarantee the maximum possible convergence, at the potential cost of performance. The presence of slow workers can hurt the system performance since all workers need to communicate their gradients to end the current round. Several techniques mitigate the negative effect of slow workers. Stale Synchronous Parallel (SSP) [12] allows workers to run at different paces within a certain interval. Faster workers that move too far ahead of the slower workers are paused. This technique maintains a strong model convergence when the number of slow workers is low. Barrierless Asynchronous Parallel (BAP) [10] removes the synchronization from the system to minimize the effect of slow workers on the system, thus workers communicate with the PS in parallel without waiting. This technique obtains the maximum speedup possible. The presence of slow workers restrains the model convergence though, as slow workers send gradients based on stale model parameters.

## 4.2 Federated Learning

Federated Learning [3] is a hybrid approach to distributed neural network training, where each node downloads the model and computes the gradients localy, with its own data, and then sends the resuls to a cloud based server. As a result, only the training coordination is centralized whereas the training and the data are decentralized. This approach is applied on the domain of mobile phones, using the data stored on this devices. Due to its rather specific domain, we will not consider this design on this thesis.

## 4.3 Summary

In this Chapter, we covered the main approach to centralized Deep Learning, the parameter server architecture, as well as Federated Learning. We discussed its variations along with its advantages and disadvantages. In the next Chapter, we will analyze the other approach to distributed Deep Learning, Decentralized Deep Learning.

# 5

# DECENTRALIZED DISTRIBUTED DEEP LEARNING

Decentralized approaches to distributed Deep Learning remove the parameter server from the system, redirecting the training and coordination to all the worker nodes. Communication between them ensures the organization and correctness of the training process. This technique removes the central server as the single point of failure and potential bottleneck. Most decentralized learning algorithms rely on the allreduce operation which reduces results across all workers in a decentralized manner.

## 5.1 ALLREDUCE

Many distributed applications benefit from reducing a set of values and distributing the results across all workers, as illustrated in Figure 4. In distributed Deep Learning this concept is particularly useful to aggregate gradients, reduce them and disseminate the result to all workers. The reduction step, denoted as $\oplus$, is performed by an optimization algorithm, typically Stochastic Gradient Descent, or an optimized version of this algorithm. The dissemination of the results varies, depending on the specification of the allreduce algorithm and the topology formed by the workers, as this versatile operation can be applied in various topologies such as rings or trees or.

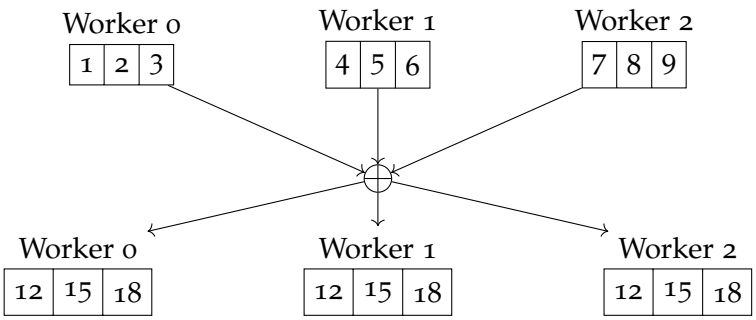



Figure 4: Example of an allreduce operation with sum function as reduction operation

## 5.2 RING ALLREDUCE

Decentralized distributed Machine Learning frameworks, like Horovod[1] [21], have proven to be a feasible and valid alternative to purely centralized systems. Horovod implements the ring allreduce algorithm [20], illustrated in Figure 5, a realization of the allreduce concept, enabling worker nodes to synchronously average gradients between them, without the need of a parameter server. Ring allreduce organizes workers in a virtual ring topology. This concept is replicated in TensorFlow with MultiWorkerMirroredStrategy. Fault tolerance was not the main concern when developing this algorithms. If a failure occurs, the system will revert to a previous checkpoint and resume the training.

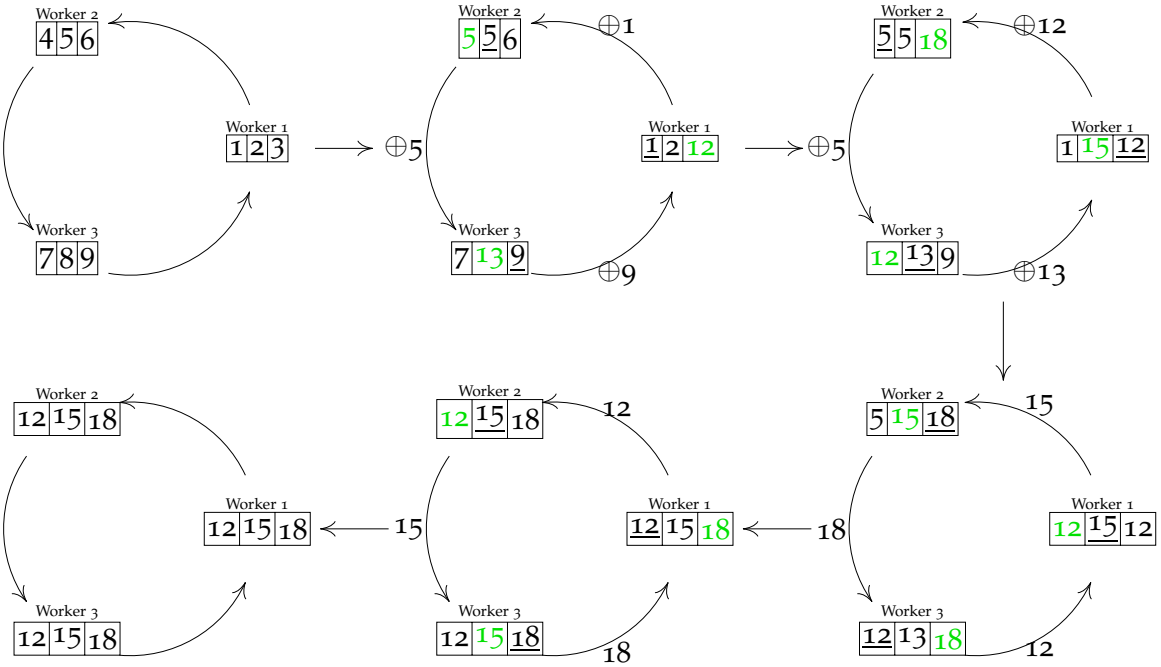



Figure 5: Horovod ring allreduce example

## 5.3 TREE-BASED ALLREDUCE

Tree-based topologies prove to be advantageous when performing allreduce operations, as they are highly scalable, simple and efficient [20]. Implementing allreduce on a tree-based topology is also intuitive. In each round, the nodes in the tree aggregate their gradients with the ones received from their children. When the aggregated gradients reach the root of

1 https://eng.uber.com/horovod/

the tree, the final round gradients are calculated and then broadcasted to all the remaining nodes. Figure 6 illustrates this process.

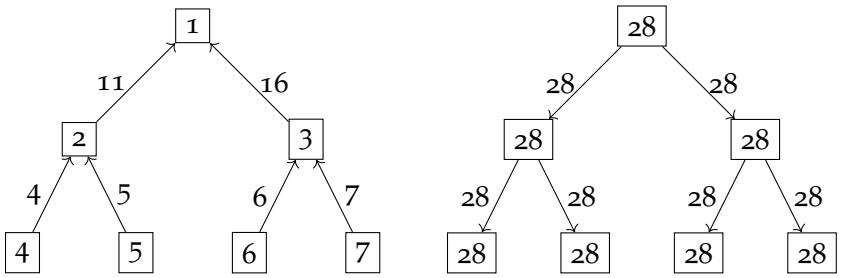



Figure 6: Tree-based allreduce

Variants of this design have already been implemented and tested in distributed Deep Learning. MXNet provides topology-aware allreduce for distributed training [5]. This approach makes use of binary trees to perform reduce and broadcast operations. Fault tolerance was not one of the aims when designing this algorithm though, so no failure detection or recovery mechanisms were specified.

Margolin and Barak [18] proposes tree-based fault-tolerant collective operations (FTCO), extending existing tree-based algorithms. The FTCO algorithm detects node failures and excludes them from the topology. This approach allows the application to keep running in the event of a node failure with a small latency penalty. FTCO does not tolerate link failures though. In the event of a failure on a link connecting two nodes, there is no guarantee that the application will keep running.

Chen et al. [4] proposes RABIT, a reliable allreduce, and broadcast interface library, specifically designed to distribute Deep Neural Networks training. This library can handle node failures by pausing every node until the malfunctioning node is restarted. After the restart, the failed node receives the latest model parameters from its peers. Once this step is completed, the training can resume. As with FTCO, RABIT does not provide link failure tolerance.

## 5.4 GOSSIPGRAD

Another approach to decentralized machine learning is GossipGraD [8], a gossip comunication protocol designed for scaling machine learning. This protocol ensures the indirect dessemination of gradients through all nodes in $\log_2(nodes)$ steps, where each step represents a computed batch. GossipGraD also implements a peer rotation mechanism in order to reduce communication inbalance. This approach does not mention faul tolerance mechanism though. In the event of a node or link failure, the algorithm does not specify a recovery mechanism, so it is not suitable for unstable networks.

## 5.5 SUMMARY

In this Chapter, we covered the main approaches to decentralized Deep Learning. We presented multiple topologies that implement this concept while discussing their advantages and disadvantages. In the next Chapter we will introduce our novel fault tolerant algorithm.

# 6

# FAULT TOLERANT TREE

As we demonstrated in the previous Chapter in section 5.3, tree topologies provide good properties to implement allreduce algorithms. Furthermore, these topologies have been adapted to tolerante node failures. Given this characteristic, we decided that it would be worth to invest our effort in finding a way to adapt the tree-based allreduce algorithm to tolerate link failures. For simplicity, we assume the network to be a fully connected graph. We also assume the nodes to be immune to failures, as this circumstance has already been studied [4, 18].

## 6.1 TOPOLOGY

To increase resilience to link failures, workers establish connections with their brother and uncle nodes, as illustrated in Figure 7, decreasing the possibility of isolated nodes in case of a link failure. These backup links, under normal circumstances, are only used to transmit small messages containing metadata. In the event of a failure on a main link, the backup links are used to spread the missing messages. This topology is based on the Epidemic Broadcast Trees [16], as there are two types of node connections: primary and secondary. The roles that these links fulfill are also similar, as primary links are used to transmit data and secondary links are used to transmit metadata.

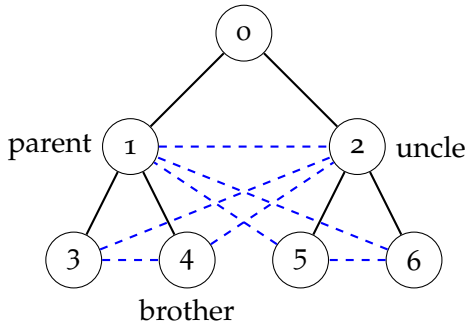



Figure 7: Fault Tolerant Tree topology. The black edges represent the main links and the blue dashed edges represent the backup links. Node 3 connections are also labeled.

## 6.2 REDUCE

We commence by describing the algorithm on the first of two steps (reduce and broadcast), of an allreduce operation. The algorithm sends data messages through the main links and metadata messages through the backup links. The backup links, the uncle and brother of a node, serve as faulty link detectors through the use of timeouts. In the event of a link failure, the timeouts will expire and these backup links will retransmit the necessary data messages. The types of message required to implement this behavior are:

- **data_reduce**$_{j,i}$(*data*) - Data message
- **meta_reduce_sent**$_{j,i}$(*target*) - Metadata message indicating that node $j$ sent a data message to *target*
- **meta_reduce_received**$_{j,i}$(*sender*) - Metadata message indicating that node $i$ received a data message from *sender*
- **meta_reduce_request**$_{j,i}$() - Metadata message requesting data from node $i$
- **data_reduce_request**$_{j,i}$(*target*, *data*) - Data message

The main portion of the algorithm that every node runs is shown in algorithm 1. Each node starts with the gradients that are calculated at each iteration. The node then waits from messages from its neighbours. If there are no communication failures, the node waits for messages and responds accordingly. Upon receiving a *data_reduce* message (line 10), the node aggregates the gradients (line 12), sends a confirmation of reception to the sender backup nodes (lines 13 and 14), sends the aggregated result to its parent (line 7) and finally sends a *meta_reduce_sent* message to each of its backup nodes (lines 8 and 9). The reduce step, from the perspective of the node 3, is illustrated in Figure 8.

**Algorithm 1:** Reduce step

**Inputs:** g; // gradients

**Procedure** *data_timeout(j)* **do**
    sleep($t$);
    **if** $\neg$ *received meta_reduce_received*$_{k,i}$*(j)* **then**
        **send**$_{i,j}$ meta_reduce_request();

**Upon event** *received from all children* **do**
    **send**$_{i,parent}$ data_reduce(g);
    **send**$_{i,uncle}$ meta_reduce_sent();
    **send**$_{i,brother}$ meta_reduce_sent();

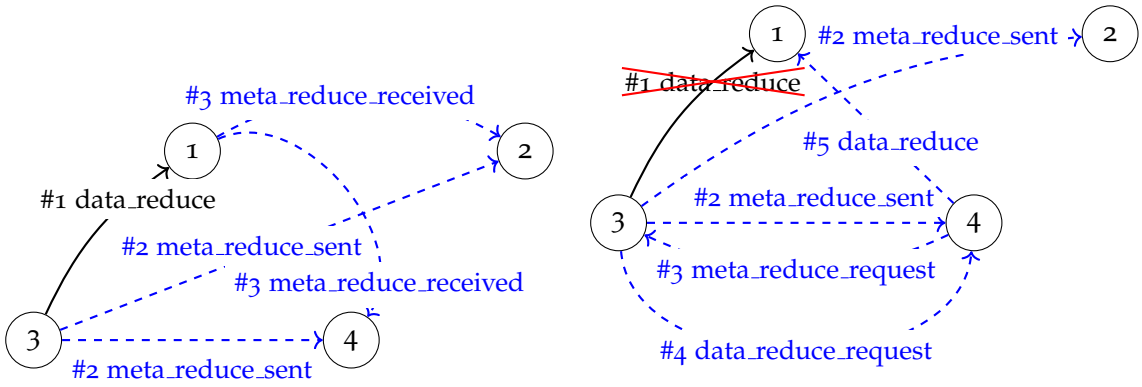
**Upon event** *data_reduce*$_{j,i}$*(g′)* **do**
    **if** $\neg$ *received g′* **then**
        g $\leftarrow$ g $\oplus$ g′;
        **send**$_{i,brother}$ meta_reduce_received(j);
        **send**$_{i,child \neq j}$ meta_reduce_received(j);

**Upon event** *meta_reduce_received*$_{j,i}$*(k)* **do**
    **Set** node k received gradients from node j;

**Upon event** *meta_reduce_sent*$_{j,i}$*(k)* **do**
    **if** $\neg$ *received meta_reduce_received*$_{k,i}$*(j)* **then**
        timeout_data(j);

**Upon event** *meta_reduce_request*$_{j,i}$*()* **do**
    **send**$_{i,j}$ data_reduce_request(*parent*, g);

**Upon event** *data_reduce_request*$_{j,i}$*(k, g′)* **do**
    **send**$_{i,k}$ data_reduce(g′);




(a) Reduce step without failures

(b) Reduce step with failure of the link between node 3 and node 1. For simplicity, it is only represented the timeout from node 4, although node 2 will also trigger a timeout

Figure 8: Reduce step from node 3 perspective

## 6.3 BROADCAST

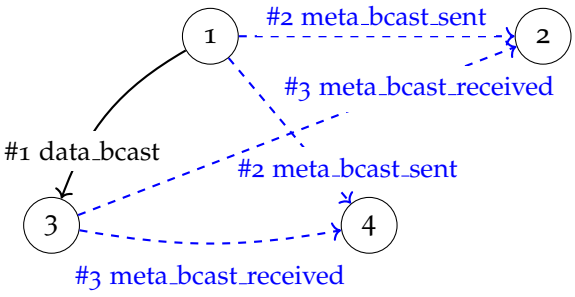
The second step of allreduce is broadcasting the final value to all nodes. The broadcast, illustrated in Figure 9, starts when the root aggregates all results and broadcasts them to its children. When a node receives a broadcast message, it sends a metadata message to its brother and uncle indicating that it received the latest update. For each child, the node sends it a data message, and a metadata message to its brother and the children's brother, indicating that it sent a data message. A failure is detected when a node does not receive confirmation that a neighbour received the latest update within a timeout. In this case, the node sends the latest update to the neighbour. The types message required to implement this behavior are:

- **data_bcast**$_{j,i}$(g) - Message with data to be broadcasted
- **meta_bcast_received**$_{j,i}$() - Metadata message indicating that node $j$ received the broadcast message
- **meta_bcast_sent**$_{j,i}$($k$) - Metadata message indicating that node $i$ broadcasted a data message to node $k$

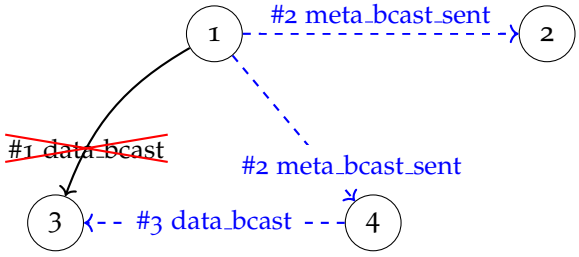
Algorithm 2 details the functioning of a broadcast step. Upon receiving a *data_bcast* message, the node sends to its backup links a *meta_bcast_received* message (lines 8 and 8), indicating that it received the latest gradients. Then it propagates the update for each child (line 10), along with one *meta_bcast_sent* for each of the child's backup nodes (lines 11 and 12).

**Algorithm 2:** Broadcast step

```
1  Procedure bcast_timeout(g, k) do
2      sleep(t);
3      if ¬ received meta_bcast_received_{k,i}() then
4          send_{i,k} data_bcast(g);

5  Upon event data_bcast_{j,i}(g) do
6      if ¬ received g then
7          send_{i,brother} meta_bcast_received();
8          send_{i,uncle} meta_bcast_received();
9          forall children ch do
10             send_{i,ch} data_bcast(g);
11             send_{i,child≠ch} meta_bcast_sent(ch);
12             send_{i,brother} meta_bcast_sent(ch);

13 Upon event meta_bcast_received_{j,i}() do
14     Set node j received latest update;

15 Upon event meta_bcast_sent_{j,i}(k) do
16     if ¬ received meta_bcast_received_{k,i}() then
17         timeout_bcast(g, k)
```




(a) Broadcast step without failures




(b) Broadcast step with failure of the link between node 3 and node 1. For simplicity, it is only represented the timeout from node 4, although node 2 will also trigger a timeout

Figure 9: Broadcast step from node 3 perspective

## 6.4 PROBABILISTIC ANALISYS

The presented algorithm provides a mechanism to continue computation in the advent of a link failure. If multiple failures occur in the network, a node in the graph can become disconnected, which leads to a stop in the computation. An example of such event is illustrated on Figure 10.

To determine the impact of this effect on the algorithm, we measure the probability of a node becoming disconnected network. The test was performed by removing random links from the network until a node beacame disconnected. The results are presented on Figure 11.

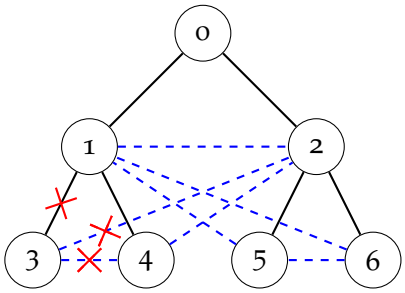
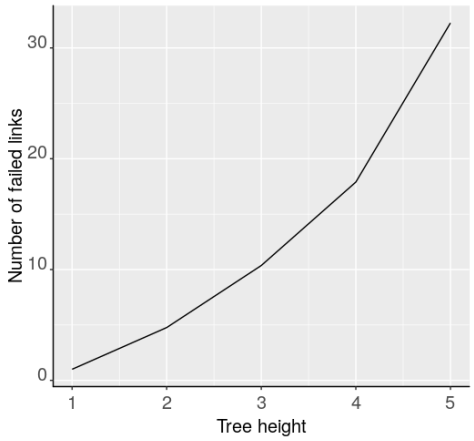
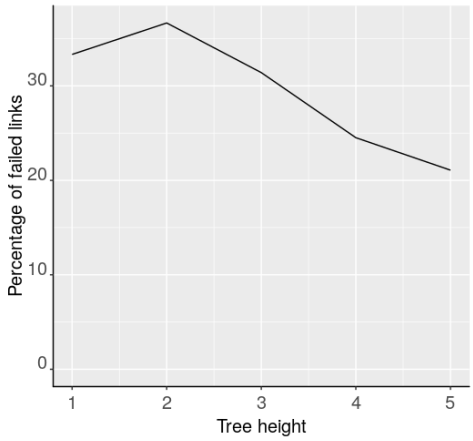



Figure 10: Example of a tree with a disconnected node (node 3)




(a) Average number of failed links until a node is isolated




(b) Average percentage of failed links until a node is isolated

Figure 11

Figure 11a shows that the average number of tolerated failed links increases with the tree height. This behavior is expected, as the total number of links also increases. Figure 11b shows that the percentage toleranted failed links decreases with the tree height. Since the failure of only two links (links that are connected to the tree root) can halt the all reduce operation, it is expected that the percentage of tolerated failed links decreases.

## 6.5 SUMMARY

In this Chapter, we presented the fault tolerant tree topology. We defined its behavior under numerous failure circumstances and analyzed its resilience to link failures. In the next Chapter, we will test this approach and compare it to other architectures presented in Chapters 4 and 5.

# 7

# EVALUATION

In this Chapter we evaluate the algorithm proposed on the previous Chapter and compare its performance to other distributed deep learning algorithms.

## 7.1 EXPERIMENTAL STEUP

We evaluated all alternatives on a single server (four Intel E5-4620 8-Core CPUs at 2.2 GHz and 126 GB of RAM). This closed environment allows for more precise monitoring and control over the communication between workers. Each node is restricted to one CPU core to emulate a single device. To simulate a disconnected link, we defined firewall rules.

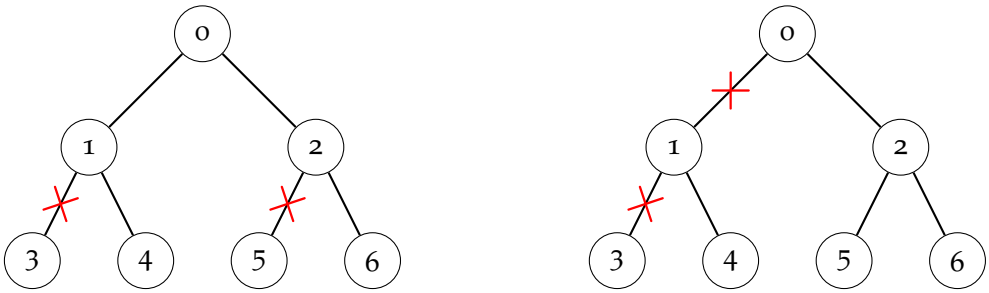
We test all alternatives with the MNIST dataset [15], composed of 60,000 training images and 10,000 test images. These images represent handwritten digits (10 classes), and the goal is to construct an accurate image classifier. For all distributed algorithms we use the same neural network, implemented using the tensorflow framework, with 4 layers with a total of 407050 parameters. We batch the data using batch of 100 images per iteration.

## 7.2 FAULT TOLERANCE COST

In this section, we present the cost tolerating a link failure. In a tree, concurrent failures can be classified as parallel or serial. A parallel failure, Figure 12a, occurs when the failed links are at the same height. Serial failures, Figure 12b, occur when the two links are at consecutive heights. As we can see in tables 1 and 2, the cost of one failure and two parallel failures is approximately the same. On the other hand, the cost of two serial failures is approximately double, as the recovery times cannot overlap.

| No. of nodes | No Faults | 1 Fault | 2 Parallel Faults | 2 Serial Faults |
|---|---|---|---|---|
| 7 | 0.033 | 1.449 | 1.495 | 2.822 |
| 15 | 0.062 | 1.510 | 1.513 | 2.890 |
| 31 | 0.091 | 1.497 | 1.494 | 2.880 |

Table 1: Allreduce step duration (in seconds) with a 500 millisecond timeout




(a) Parallel failure (b) Serial failure

Figure 12: Types of link failures on a tree

| No. of nodes | No Faults | 1 Fault | 2 Parallel Faults | 2 Serial Faults |
|---|---|---|---|---|
| 7 | 0.034 | 2.441 | 2.473 | 4.839 |
| 15 | 0.065 | 2.541 | 2.498 | 4.860 |
| 31 | 0.093 | 2.509 | 2.541 | 4.911 |

Table 2: Allreduce step duration (in seconds) with a 1 second timeout

## 7.3 Results without failures

In this section we present the results gathered from testing the fault tolerant tree topology in a environment without failures. We also compare it with other topologies presented previously. All abreviantions used in this Chapter are described in table 3

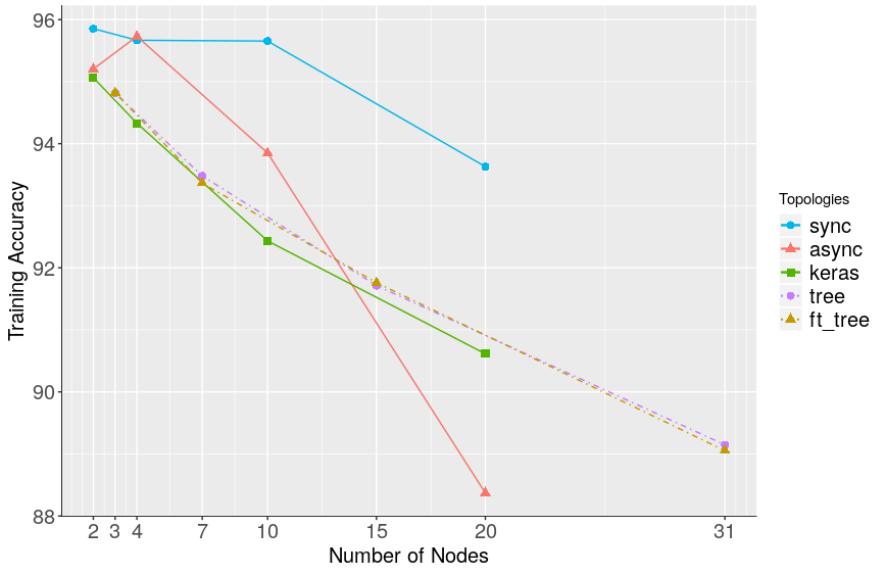
| Abreviation | Description |
|---|---|
| sync | Synchronous parameter server architecture |
| async | Asynchronous parameter server architecture |
| keras | TensorFlow native distribution technique using the MultiWorkerMirroredStrategy, invoked by the Keras API |
| tree | Standard tree-based allreduce |
| ft_tree | Fault tolerant tree allreduce |

Table 3: Allreduce step duration (in seconds) with a 1 second timeout

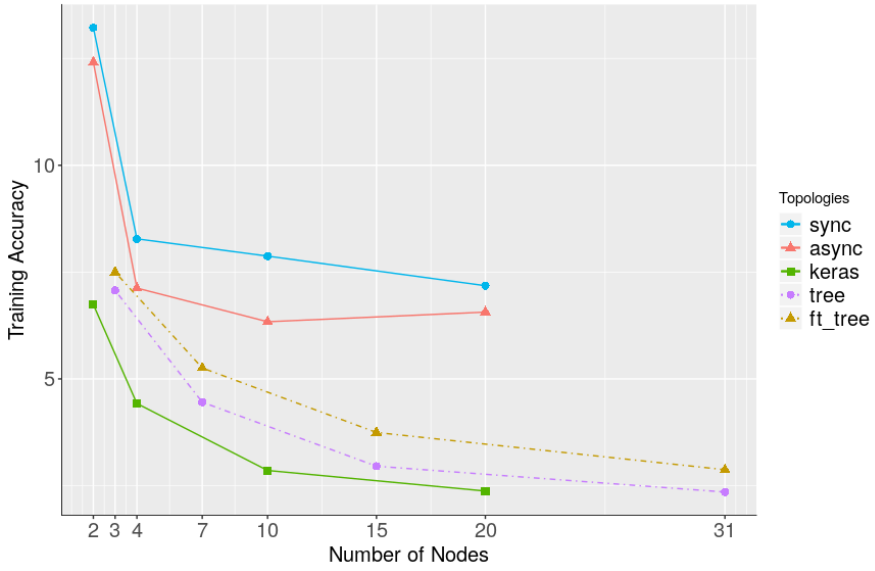
Figure 13a shows the resulting training accuracy and Figure 13b shows the training time.

As we can see in Figure 13a, the best accuracy is achieved using a synchronous parameter server approach, as this architecture guarantees the maximum convergence, at the expense of time. This tradeoff is visible in Figure 13b. On the other hand, the asynchronous parameter server approach trades accuracy for performance. Both centralized approaches show slower training times when compared to decentralized ones.

The decentralized topologies show faster training time, while keeping high accuracies. We can see that tree-based allreduce has the edge over ring-based allreduce accuracy wise.




(a) Train accuracy



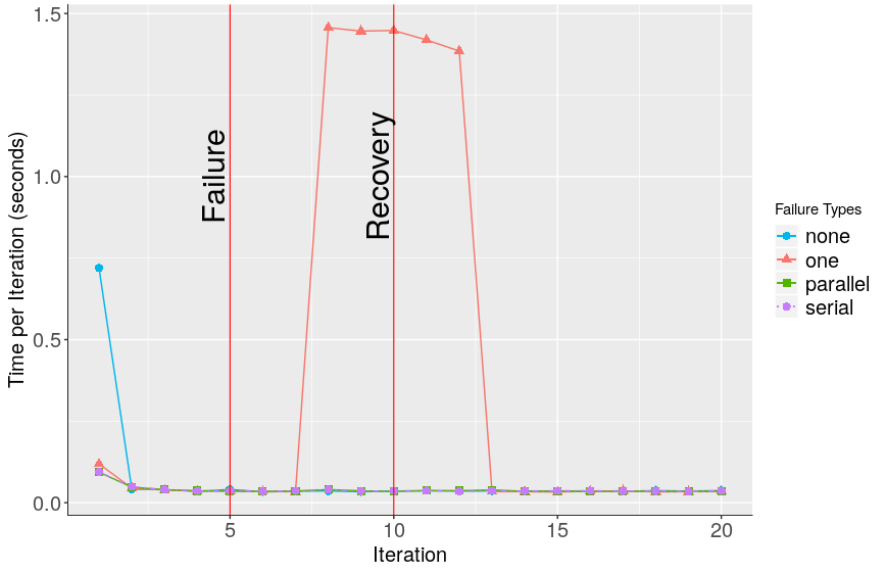

(b) Train time

Figure 13: Training accuracy and time using different topologies.

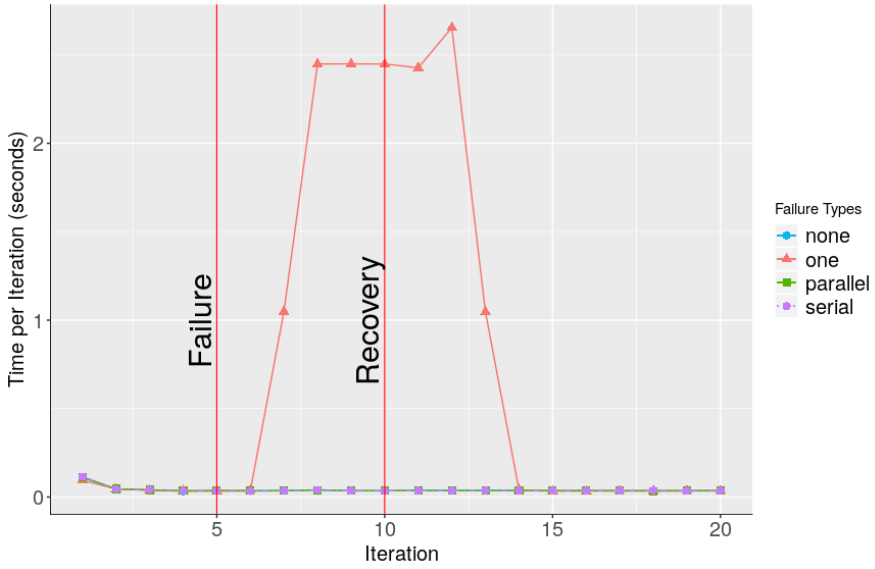
This difference, although, isn't large. Time wise, we can see that the ring based approach has the edge over the tree-based allreduce. It is also important to note that the fault tolerant mechanisms employed on the tree-based allreduce add a time penalty.

## 7.4 Results with failures

In this section, we present the results gathered from testing the fault tolerant tree topology in an environment with link failures. For demonstration purposes, we assume that the links fail between the fifth and the tenth training iteration. Figures 14, 15, 16 and 17 show the results.

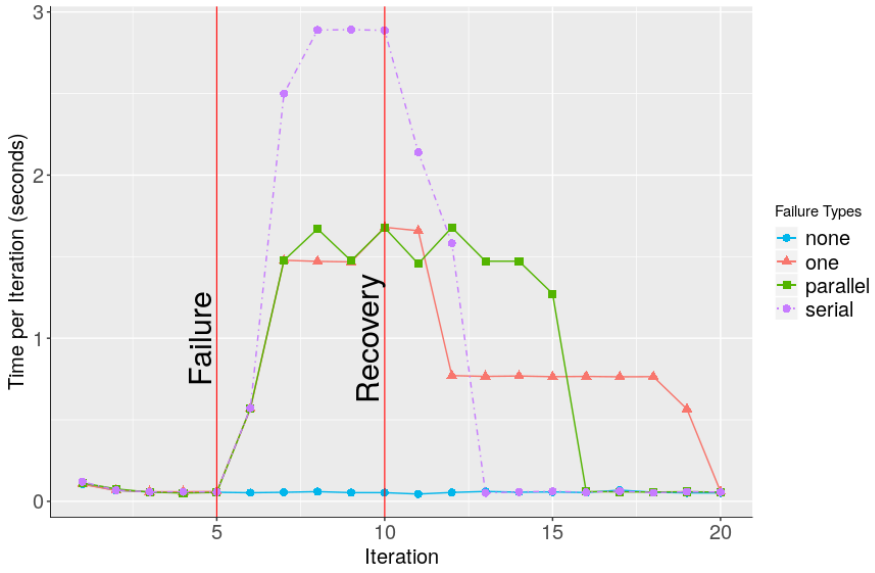



(a) Time per iteration with a 500 millisecond timeout

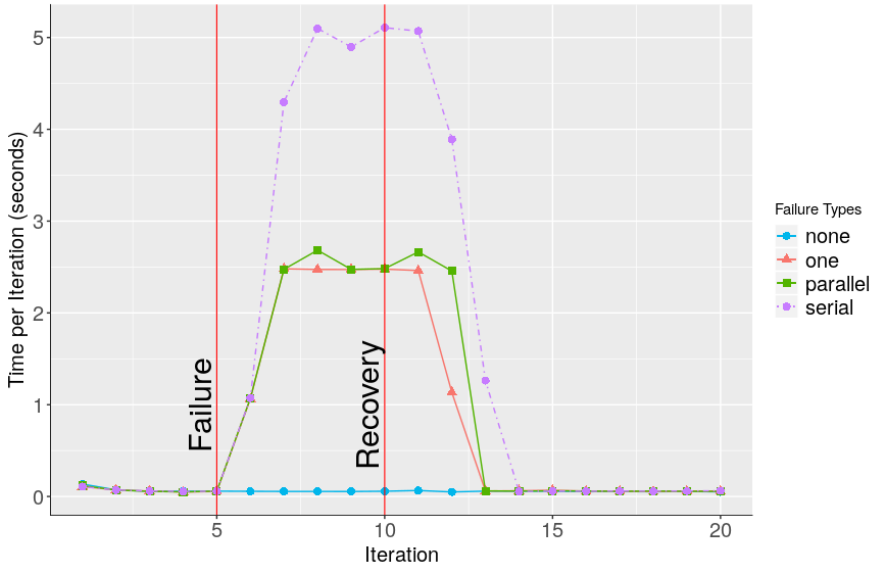



(b) Time per iteration with a 1 second timeout

Figure 14: Time per iteration with a fault tolerant tree with 3 nodes

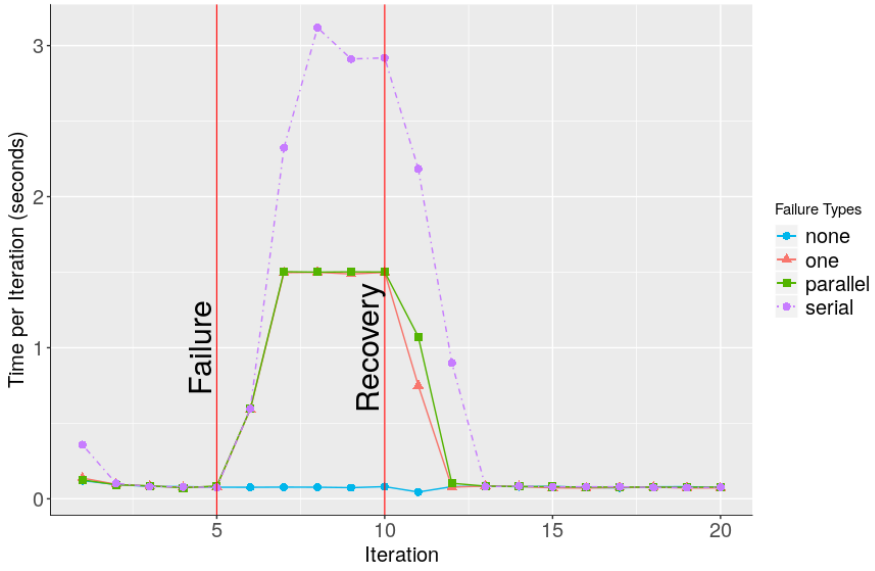



(a) Time per iteration with a 500 millisecond timeout

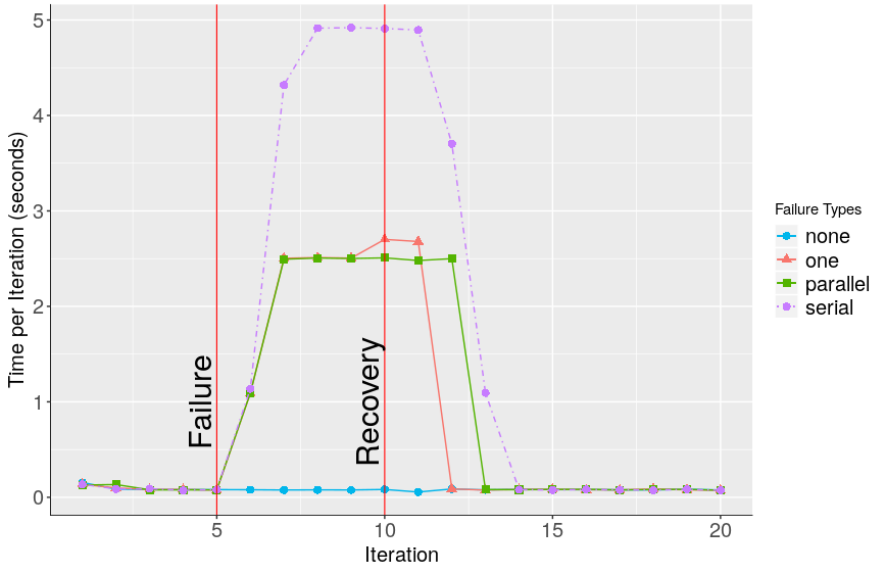



(b) Time per iteration with a 1 second timeout

Figure 15: Time per iteration with a fault tolerant tree with 7 nodes

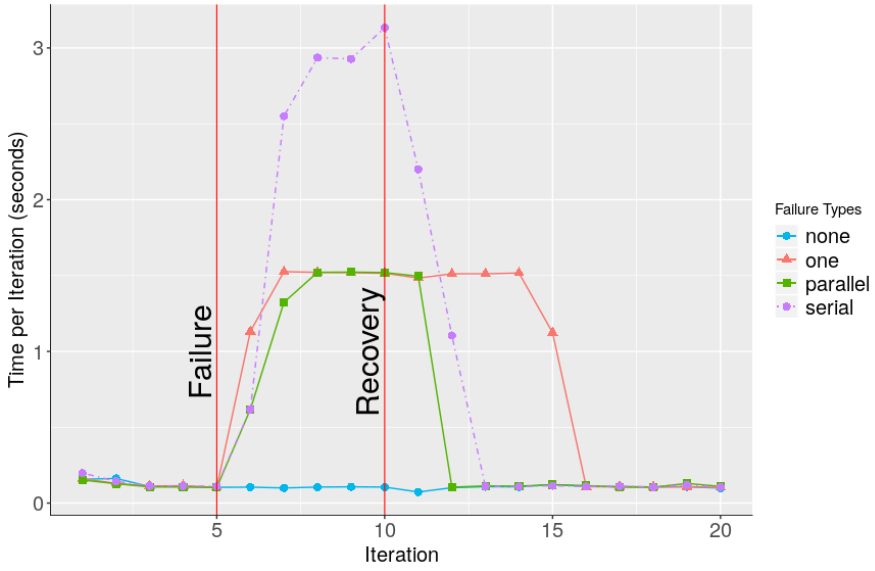



(a) Time per iteration with a 500 millisecond timeout

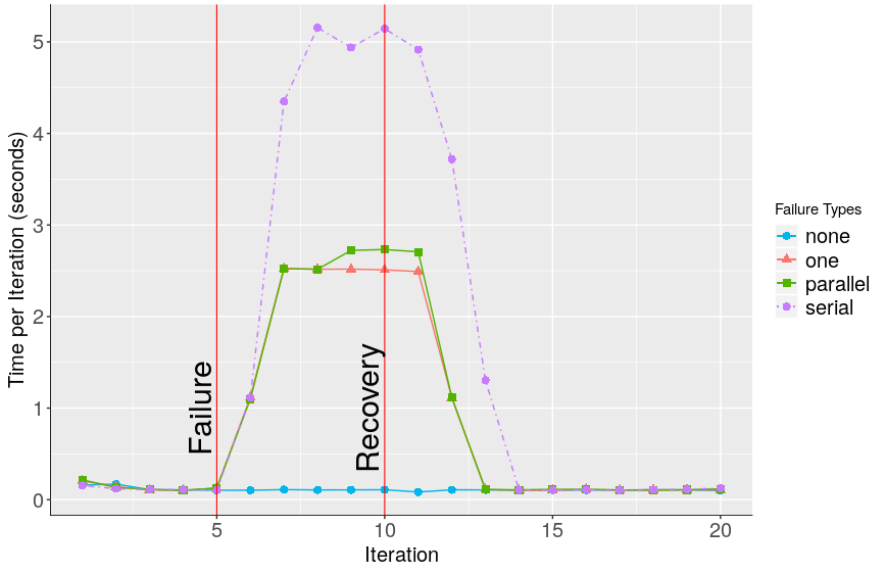



(b) Time per iteration with a 1 second timeout

Figure 16: Time per iteration with a fault tolerant tree with 15 nodes




(a) Time per iteration with a 500 millisecond timeout




(b) Time per iteration with a 1 second timeout

Figure 17: Time per iteration with a fault tolerant tree with 31 nodes

## 7.5 SUMMARY

In this Chapter, we analyzed the performance of the fault tolerant tree algorithm and compared it to its competitors presented in Chapters 4 and 5. We also presented the cost that a link failure introduces in the system.

# 8

# CONCLUSIONS AND FUTURE WORK

This work presented a novel distributed fault tolerant neural network training technique. The prototype was based on the TensorFlow framework and compared with multiple well-studied distribution techniques. We see potential on our approach, as it presents good performance in the tested environments.

Future works should focus on implementing this approach using the TensorFlow strategy API, as it will potentially improve the performance of the fault tolerant tree topology and facilitate its distribution on the TensorFlow ecosystem. Future work should also aim to improve the performance of the algorithm by implementing adaptative timeouts instead of static timeouts. This improvement could greatly reduce the training time when a failure occurs in the system.

# BIBLIOGRAPHY


[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

[2] Imad A Basheer and Maha Hajmeer. Artificial neural networks: fundamentals, computing, design, and application. *Journal of microbiological methods*, 43(1):3–31, 2000.

[3] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečnỳ, Stefano Mazzocchi, H Brendan McMahan, et al. Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046*, 2019.

[4] Tianqi Chen, Ignacio Cano, and Tianyi Zhou. Rabit: A reliable allreduce and broadcast interface. *Transfer*, 3(2).

[5] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[6] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 571–582, 2014.

[7] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*, RecSys '16, pages 191–198, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4035-9. doi: 10.1145/2959100.2959190. URL http://doi.acm.org/10.1145/2959100.2959190.

[8] Jeff Daily, Abhinav Vishnu, Charles Siegel, Thomas Warfel, and Vinay Amatya. Gossipgrad: Scalable deep learning using gossip communication based asynchronous gradient descent. *arXiv preprint arXiv:1803.05880*, 2018.

[9] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[10] Minyang Han and Khuzaima Daudjee. Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 8(9):950–961, 2015.

[11] Corentin Hardy, Erwan Le Merrer, and Bruno Sericola. Distributed deep learning on edge-devices: feasibility via adaptive compression. In *Network Computing and Applications (NCA), 2017 IEEE 16th International Symposium on*, pages 1–8. IEEE, 2017.

[12] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, pages 1223–1231, 2013.

[13] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 1–12. IEEE, 2017.

[14] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[15] Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits, 1998. *URL http://yann. lecun. com/exdb/mnist*, 10:34, 1998.

[16] Joao Leitao, Jose Pereira, and Luis Rodrigues. Epidemic broadcast trees. In *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 301–310. IEEE, 2007.

[17] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI*, volume 14, pages 583–598, 2014.

[18] Alexander Margolin and Amnon Barak. Tree-based fault-tolerant collective operations for mpi. In *Workshop on Exascale MPI (ExaMPI)*, 2018.

[19] Nasser M Nasrabadi. Pattern recognition and machine learning. *Journal of electronic imaging*, 16(4):049901, 2007.

[20] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

[21] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[22] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.