Computer Assisted Diagnosis (CAD)
with Enhanced Cognitive Architectures (BorisCAD)

Rafael Samorinha

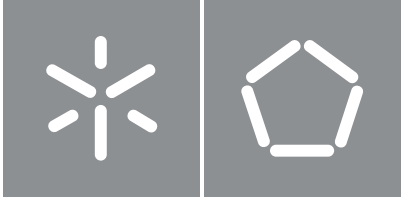
UMinho | 2023

Universidade do Minho
Escola de Engenharia

Rafael André Escalhão Samorinha

Computer Assisted Diagnosis (CAD) with Enhanced Cognitive Architectures (BorisCAD)

julho de 2023



Universidade do Minho
Escola de Engenharia

Rafael André Escalhão Samorinha

Computer Assisted Diagnosis (CAD) with Enhanced Cognitive Architectures (BorisCAD)

Dissertação de Mestrado
Engenharia Eletrónica Industrial e Computadores
Sistemas Embebidos e Computadores

Trabalho efetuado sob a orientação do(a)
Professor Doutor Carlos Manuel Gregório Santos Lima e **Professor Doutor Adriano José da Conceição Tavares**

julho de 2023

Agradecimentos

Desde já, gostaria de expressar o meu agradecimento a todos aqueles que contribuíram para finalizar esta etapa. A concretização deste trabalho não teria sido possível sem a ajuda e o apoio de várias pessoas que fizeram parte do meu percurso académico, permitindo-me crescer como pessoa e profissional.

À minha família, agradeço todo o apoio, força e motivação ao longo destes anos, bem como por nunca terem desistido de mim, incentivando-me a prosseguir e lutar por aquilo que sempre desejei fazer.

Aos meus orientadores, Adriano Tavares e Carlos Lima, agradeço pela disponibilidade e ajuda prestadas, bem como por nunca terem reduzido os níveis de exigência.

Aos restantes inquilinos da Casinha do Povo, quero agradecer pela companhia durante estes looooooongos anos, sempre a incentivar-me para fazer mais e melhor, muitas das vezes por caminhos dúbios. Como já vários sábios disseram, sem vocês acabava um ano mais cedo.

À Marcela... por tudo aquilo que fizeste por por mim neste último ano. Por todos os dias em que foste chatinha comigo para eu fazer coisas, e por nunca desistires de mim. Eu tenho toda a certeza que sem ti não estaria tão confiante no meu potencial como estou hoje, nem teria alcançado tudo o que consegui. Obrigado por estares sempre ao meu lado e nunca duvidares.

"So gimme some fin... noggin... dude!" - Crush

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

Computer-assisted medical image diagnosis plays a vital role in modern healthcare by enabling accurate and timely detection of various medical conditions. However, conventional image classification techniques often fail to integrate crucial elements such as perception, reasoning, and episodic experiences, essential for achieving optimal performance. This research endeavours to bridge this gap by designing, implementing, and evaluating a cognitive architecture incorporating these elements. The findings of this research will contribute to the development of a deployable cognitive architecture that can provide accurate and reliable diagnoses for a wide range of medical conditions, benefiting both healthcare professionals and patients alike. Furthermore, the potential impact of this research extends beyond medical image analysis, with implications in autonomous systems, robotics, and intelligent decision support systems. By harnessing the potential of cognitive architectures, computer-assisted systems can be revolutionised, leading to improved diagnostic accuracy and fostering innovation in diverse industries reliant on advanced cognitive capabilities.

In order to achieve this objective, sub-symbolic methods for perception and symbolic methods for reasoning will integrate the cognitive architecture. The bottom level, responsible for sub-symbolic processing, will incorporate advanced segmentation and classification using convolutional neural networks to handle perception tasks. In contrast, the top level will utilise a decision forest, an ensemble of decision trees, to perform sophisticated reasoning tasks using symbolic data. Additionally, this study will focus on integrating episodic experiences within the architecture by incorporating working and long-term memory mechanisms, enhancing its predictive capabilities.

The evaluation of the cognitive architecture demonstrated its effectiveness within the context of the tested datasets and image sizes. However, it is essential to acknowledge that developing a deployable version for medical image diagnosis requires further testing and validation. Expanding the evaluation to include a broader range of pathologies and imaging modalities is crucial to ensure the architecture's robustness and adaptability in diverse clinical scenarios. By incorporating a more diverse set of pathologies and imaging modalities into the evaluation process, the cognitive architecture can undergo rigorous testing to assess its performance across various medical conditions. This expanded evaluation will help identify potential limitations and areas for improvement, ensuring that the architecture can deliver accurate and reliable diagnoses across a broader spectrum of medical conditions.

Keywords: convolutional neural networks; decision tree; memory; cognitive architecture; image classification; learning; adaptation; medical diagnosis; artificial intelligence; Boris

Resumo

O diagnóstico assistido por computador a partir de imagens médicas desempenha um papel vital na saúde moderna, permitindo a detecção precisa e atempada de várias condições médicas. No entanto, as técnicas convencionais para classificação de imagens não integram elementos cruciais da análise humana, tais como a percepção, raciocínio e experiências episódicas. O trabalho apresentado procura preencher esta lacuna através do design, implementação e avaliação de uma arquitetura cognitiva que incorpora estes elementos. Os resultados obtidos contribuem para o desenvolvimento de novos métodos capazes de fornecer diagnósticos precisos e confiáveis para uma ampla gama de condições médicas, beneficiando tanto os profissionais de saúdo como os pacientes. Além disto, o potencial deste trabalho estende-se além da análise de imagens médicas, com possíveis implicações em sistemas autónomos, robótica e sistemas inteligentes de suporte à decisão. Ao aproveitar o potencial das arquiteturas cognitivas, estes sistemas podem ser revolucionados, levando a uma melhoria na precisão e a estimulação para a inovação em diversas indústrias que dependem de capacidades cognitivas avançadas.

Para alcançar o objetivo principal de diagnóstico, serão integrados na arquitetura cognitiva métodos sub-simbólicos para percepção e métodos simbólicos para raciocínio. O nível mais abaixo, responsável pelo processamento sub-simbólico, incorporará redes neuronais convolucionais para segmentação e classificação, de forma a emular percepção no sistema. Por outro lado, o nível superior irá usar uma floresta de decisão, um conjunto de árvores, para realizar tarefas de raciocínio usando dados simbólicos. Além disto, este estudo irá focar-se na integração de experiências episódicas, incorporando mecanismos de memória de trabalho e a longo prazo, de forma a incluir dados passados na decisão atual.

A avaliação desta arquitetura demonstrou a sua eficácia no contexto do conjunto de dados e tamanhos de imagem usados. No entanto, é essencial reconhecer que o desenvolvimento de uma versão aplicável ao diagnóstico de imagens médicas requer testes e validações adicionais. Desta forma, a expansão do sistema para incluir uma ampla gama de patologias e modalidades de imagem é crucial para garantir a robustez e adaptabilidade da arquitetura em diversos cenários clínicos. Ao incorporar um conjunto mais diversificado de patologias e modalidades de imagem no processo de avaliação, a arquitetura pode ser submetida a testes rigorosos de forma a avaliar o seu desempenho no mundo real. Esta ampla avaliação ajudará a identificar possíveis limitações, garantindo que o sistema fornece diagnósticos precisos e confiáveis num espectro vasto de condições médicas.

Palavras-chave: redes neuronais convolucionais; árvore de decisão; memória; arquitetura cognitiva; classificação de imagens; aprendizagem; adaptação; diagnóstico médico; inteligência artificial; Boris

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Acronyms

ACS Action-Centred Subsystem. 10, 11

ACT-R Adaptive Control of Thought-Rational. 8, 9, 12

AI Artificial Intelligence. 5, 6, 12

CAD Computer-Aided Diagnosis. 6, 16, 47

CBR Case-Based Reasoning. 6, 13, 15

CNN Convolutional Neural Network. 18, 21–24

CNNs Convolutional Neural Networks. 18, 21–24, 75

CS Computation System. 22

CV Computer Vision. 6

DT Decision Tree. 18

DTs Decision Trees. 18, 21–23, 75

fMRI Functional Magnetic Resonance Imaging. 9

IoU Intersection over Union. 126, 128

KNN K-Nearest Neighbours. 18

LIDA Learning Intelligent Distribution Agent. 12

LTM Long-Term Memory. xiv, 22, 115–118, 143, 145

MCC Matthews Correlation Coefficient. 60, 112, 129–133, 135, 136

MCS Meta-Cognitive System. 11

MLP Multilayer perceptron. 23

MS Motivational System. 11

NACS Non-Action-Centred Subsystem. 10, 11

RBR Rule-Based Reasoning. 6, 13, 15

ReLU Rectified Linear Unit. 27–30, 77, 78, 87

SGD Stochastic Gradient Descent. 54, 56

SVM Support Vector Machine. 18

UML Unified Modeling Language. 28, 32, 33, 44, 46, 53, 61, 62, 86

WM Working Memory. xiv, 14, 19, 22, 65, 115, 118, 143

Chapter 1

Introduction

Medical imaging has revolutionised healthcare by enabling professionals to diagnose and treat patients. However, accurately identifying specific pathologies in X-rays can be challenging due to the similarities in images and their characteristics. This dissertation proposes a cognitive architecture approach to develop a system that accurately identifies a specific pathology, such as lung cancer, in medical documents (Chest X-rays).

Accurate identification of pathology in medical documents is of paramount importance in the healthcare domain. Misdiagnosis or delayed diagnosis can severely affect patients, resulting in sub-optimal outcomes and increased healthcare costs. Therefore, there is a pressing need for more effective and efficient methods to aid healthcare professionals in accurately interpreting medical images and improving diagnostic accuracy.

Previous research has revealed limitations in relying solely on symbolic or sub-symbolic models for human-like decision-making in complex domains like medical image diagnosis [7]. Symbolic models rely on predefined rules and logic, which may not capture the full range of patterns and relationships in medical images. On the other hand, sub-symbolic models utilise statistical or machine-learning techniques to identify patterns in data. However, they may need a more detailed understanding of the underlying decision-making processes. Thus, a cognitive architecture incorporating both symbolic and sub-symbolic methods provides a more comprehensive modelling approach with broader capabilities. Cognitive architecture models, inspired by human cognition, simulate how people think, learn, and make decisions, enhancing the system's ability to identify pathology in medical documents accurately.

The specific pathology the system can identify will depend on the dataset the architecture uses for training. Therefore, this dissertation focuses on developing and evaluating a system using a cognitive architecture approach to identify medical document pathology. The proposed approach aims to leverage

the strengths of both symbolic and sub-symbolic models, combining the interpretability of symbolic models and the pattern recognition capabilities of sub-symbolic models. By integrating these methods, the cognitive architecture approach offers a promising solution to improve accuracy and decision-making in medical image analysis.

In addition to developing the cognitive architecture approach, this dissertation will evaluate its performance using a comprehensive methodology. The system will be trained and tested on a diverse dataset of medical documents, including Chest X-rays. The evaluation will assess the system's accuracy, efficiency, and potential benefits in medical document analysis. The results of this study have significant implications for improving the accuracy and efficiency of medical diagnosis and treatment, ultimately benefiting patients and healthcare professionals.

1.1 General Motivations

Machine learning and artificial intelligence have made remarkable advancements in image identification and computer vision. However, these methods may require more advanced perception and decision-making capacity for more complex tasks, such as medical image classification. Inaccurate identification or diagnosis of specific pathology or abnormalities can have severe consequences, including incorrect treatment and poor patient outcomes.

The medical field is a domain where accurate classification is of utmost importance. While medical professionals rely on their expertise and experience to diagnose pathology, this process can be time-consuming and susceptible to error. Therefore, developing a system that accurately identifies pathology in medical images holds great potential. It could provide medical staff with a valuable second opinion, reducing the risk of misdiagnosis and significantly improving patient outcomes. This dissertation focuses on developing a cognitive architecture approach tailored to identify medical image pathology. Furthermore, the cognitive architecture approach employed in this research can have implications beyond the medical field, as it could be applied to other domains and applications requiring advanced perception and decision-making capabilities.

The potential benefits of employing a cognitive architecture approach for medical image identification are numerous. By leveraging the strengths of symbolic and sub-symbolic models, the cognitive architecture approach aims to enhance accuracy and efficiency in medical diagnosis. It can alleviate the workload of medical staff, improve diagnostic outcomes, and save lives. Moreover, developing a cognitive architecture approach for this application could drive advancements in other fields where perception and decision-

making are critical, such as autonomous vehicles or robotics.

1.2 Contributions of this Thesis

The research presented in this study contributes significantly in two key areas. Firstly, it proposes an innovative cognitive architecture approach that effectively identifies specific pathology in medical documents. By seamlessly integrating symbolic and sub-symbolic components, this approach harnesses the strengths of both models, providing a comprehensive and detailed range of capabilities for accurate diagnosis. This novel approach represents a notable advancement in medical image identification.

Secondly, the study demonstrates the practicality and effectiveness of the proposed approach through extensive experimentation and evaluation using diverse medical datasets. The experimental results consistently highlight the superiority of the cognitive architecture approach in terms of accuracy and efficiency, surpassing existing methods. These findings validate the feasibility and success of the proposed approach and underline its potential to enhance medical image identification significantly.

As a result, this research makes noteworthy contributions to medical image identification and cognitive architecture modelling. It introduces an innovative approach that combines symbolic and sub-symbolic components, thereby improving the accuracy of medical diagnosis. Moreover, this study establishes a solid foundation for future research and development in this domain, paving the way for further advancements and applications of cognitive architecture in various domains. Overall, these contributions position the research as a valuable and impactful contribution to the scientific community.

1.3 Structure of the Dissertation

This dissertation is organised into five main chapters, each contributing to the development of a cognitive architecture approach for identifying pathology in medical images.

Chapter 2 provides a comprehensive overview of the state-of-the-art in cognitive architectures and medical image classification. It covers existing research and methods used in cognitive architectures, including symbolic and sub-symbolic models, as well as medical image classification. Additionally, this chapter explores the potential benefits of utilising a cognitive architecture approach in medical document analysis, highlighting its implications for improving the accuracy and efficiency of medical diagnosis and treatment.

Chapter 3 focuses on the design of the proposed cognitive architecture approach for identifying pathology in medical documents. This chapter describes the components of the architecture, outlines

the processing steps involved in identifying pathology, and explains the decision-making process.

Chapter 4 describes the implementation of the proposed cognitive architecture approach. It discusses how the architecture was implemented and addresses the challenges encountered in integrating the previously designed components to work together effectively.

Chapter 5 presents the tests and results of the cognitive architecture approach for identifying pathology in medical images. This chapter outlines the experimental design, describes the data used to evaluate the system, and presents the performance metrics used to assess the accuracy and efficiency of the system. Additionally, it discusses the limitations of the proposed cognitive architecture approach and explores potential future directions for improvement.

Finally, Chapter 6 provides the final conclusions of this dissertation. It summarises the main findings and contributions of the research, emphasising the effectiveness of the cognitive architecture approach for identifying pathology in medical images. This chapter serves as a culmination of the dissertation, offering a comprehensive understanding of the cognitive architecture approach and its potential impact on medical image classification and diagnosis.

Chapter 2

Literature Review / State of the Art

This chapter presents a comprehensive background to establish the foundation for developing the proposed system. The chapter begins with a general overview, followed by an in-depth exploration of the characteristics of the cognitive architectures that will be utilised as the baseline in the system's implementation.

Initially, the role of artificial intelligence (AI) in the medical field is discussed, focusing on its utilisation to enhance pathology diagnosis. Existing successful implementations of AI systems in this domain are highlighted, illustrating their potential benefits. Given the dissertation's specific focus on employing cognitive architectures for medical diagnosis, relevant literature is reviewed to showcase how these approaches have proven valuable in training neural networks to mimic human brain functioning [8].

Subsequently, an introduction is provided on the cognitive architectures employed in the system's implementation. This entails elucidating their foundational concepts and exploring the most recent advancements in these architectures. The work of Goria *et al.* [9] serves as a comprehensive reference, offering an extensive overview of these architectures and their diverse applications.

2.1 Artificial Intelligence In The Medical Field

Due to the increasing need for faster and more precise diagnosis when dealing with serious diseases, there is a growing interest in understanding the precision and speed of the human brain when identifying pathology in medical exams [10].

One of the most promising applications of intelligent techniques for diagnostic sciences is biomedical image classification [11]. Various Artificial Intelligence (AI) techniques for automating this diagnosis have

already been explored, and it was concluded that the biomedical image classification using Computer Vision (CV) techniques is rapidly improving the field [11, 12].

As referred to in [11], new imaging modalities and methods of interpreting tasks are being developed, such as model-based intelligent analysis and decision-making tools like cognitive architectures, which will be the main focus of this dissertation.

Many examples of AI and CV applications in diagnostic sciences have already been developed, including the detection of breast cancer using mammography. Screening programs are being introduced into deep neural networks to detect occurrences of node-negative tumours or larger tumours in more advanced stages, providing faster diagnosis and better treatment [13].

Similar methods in Computer-Aided Diagnosis (CAD) have also been developed, using radiology and radiography to identify several pathologies [14]. Case-Based Reasoning (CBR) and Rule-Based Reasoning (RBR) [15] can also be applied, such as in the early diagnosis of gastrointestinal cancer [16]. In this case, no pre-processing or neural networks are implemented, only specific cases and rules that output the probability of a patient having this type of cancer after trimming all the available data.

These applications raise ethical questions, as using medical documents to teach AI raises legal and ethical concerns [17]. Researchers have designed a system to select patients and exams that meet specific criteria for network learning [1]. Figure 1, presented in [1], shows how screening for new content for learning is selected and added to the dataset for further evaluation.

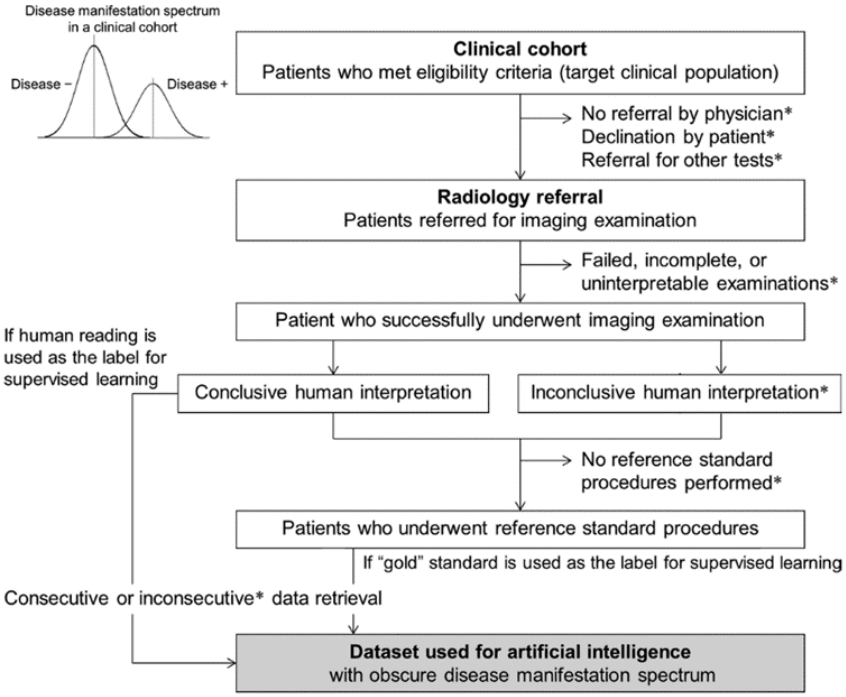


Figure 1: Patient exclusion by means of clinical referrals. [1]

2.2 Computer Vision

As seen in the previous topic, most applications and systems use computer vision in the process of identifying pathology. There are many ways to use computer vision techniques, but the most efficient one in this system is via image understanding architectures [18]. With it, image processing and interpretation become possible when applying different models, inference engines and learning methods, making a system vision capable [19].

For this project specifics, pattern recognition [20] and general feature extraction methods will be used as a part of pre-processing in creating the learning dataset that will be further applied to the neural network or the cognition process. An example of applying a similar method is shown in [21], where a convolutional neural network is used in a study on how its depth influences the accuracy of a large-scale image recognition system. Although this study is more focused on deep neural network characteristics, some methods for computer vision techniques are also discussed and might become relevant when organising a big chunk of image data.

2.3 Cognitive Architectures

A Cognitive Architecture is all the theory related to the model or structure of the human mind in the field of artificial intelligence and overall cognitive science. The main objective is to provide a template of the mind to work in artificial systems, making different elements cooperate to achieve intelligent behaviour in a complex environment [22]. In sum, it is a broadly-scoped, domain-generic computational cognitive model, capturing the essential structured and processes of the mind, to be used for a broad, multiple-level, multiple-domain analysis of behaviour. [23]

During the years, there have been multiple attempts to achieve human-like decision-making within machines. Referring to some of the cognitive architectures that will be explored in this sub-section, and have received multiple updates [24] since then, even having some examples of already implemented and tested architectures to be used in artificial vision systems [25]. Some other examples of using cognitive architectures to mimic human interaction can be seen in autonomous robotic systems, making them able to communicate, have human-like coordination and the capability to adapt to novel situations, learning through experience [26].

A more detailed study of these architectures is explained further in this dissertation. It is only relevant to show some previously researched knowledge about the theme rather than to explain these models

exhaustively.

2.3.1 ACT-R (Adaptive Control of Thought-Rational)

The Adaptive Control of Thought-Rational (ACT-R) cognitive architecture aims to uncover the fundamental cognitive and perceptual operations that underlie human cognition [27]. It follows a modular framework that captures the intricate workings of the human mind, with tasks comprising discrete operations.

At the heart of ACT-R lies the distinction between declarative and procedural knowledge (Figure 2). Declarative knowledge encompasses factual information organised into chunks, accessed through buffers facilitating communication among modules within ACT-R. These buffers serve as the interfaces through which information flows between the perceptual, cognitive, and motor processes.

Procedural knowledge, on the other hand, involves the acquisition and execution of skills and behaviours. It encompasses knowledge about performing tasks through well-defined steps or algorithms. In ACT-R, procedural knowledge is implemented using production rules, which encode sequences of actions and conditions.

Declarative and procedural knowledge interact to drive cognitive processing. Declarative knowledge provides rules and information that guide procedural knowledge, while procedural knowledge enables the efficient execution of cognitive tasks based on acquired skills. Together, they form the foundation of cognitive architectures like ACT-R, allowing for the modelling and simulation of human-like cognitive processes and behaviours.

ACT-R comprises two main module types. The perceptual-motor modules enable the architecture to interact with the external world. For example, the visual module facilitates visual perception and processing, while the manual module facilitates motor control and object manipulation. These modules enable ACT-R to acquire environmental information and generate appropriate motor responses.

The memory modules within ACT-R are responsible for storing and manipulating knowledge. Declarative memory stores factual information, enabling the retrieval of facts and concepts. On the other hand, procedural memory contains rules and procedures that guide behaviour and cognitive processes. It allows ACT-R to execute cognitive tasks by activating production rules, which specify the conditions for rule firing and the corresponding actions to be performed.

Recent research has focused on integrating visual attention mechanisms into the ACT-R architecture. Visual attention is crucial in selectively directing cognitive resources toward relevant stimuli. By incorporating these mechanisms, ACT-R can model various cognitive phenomena that depend on the

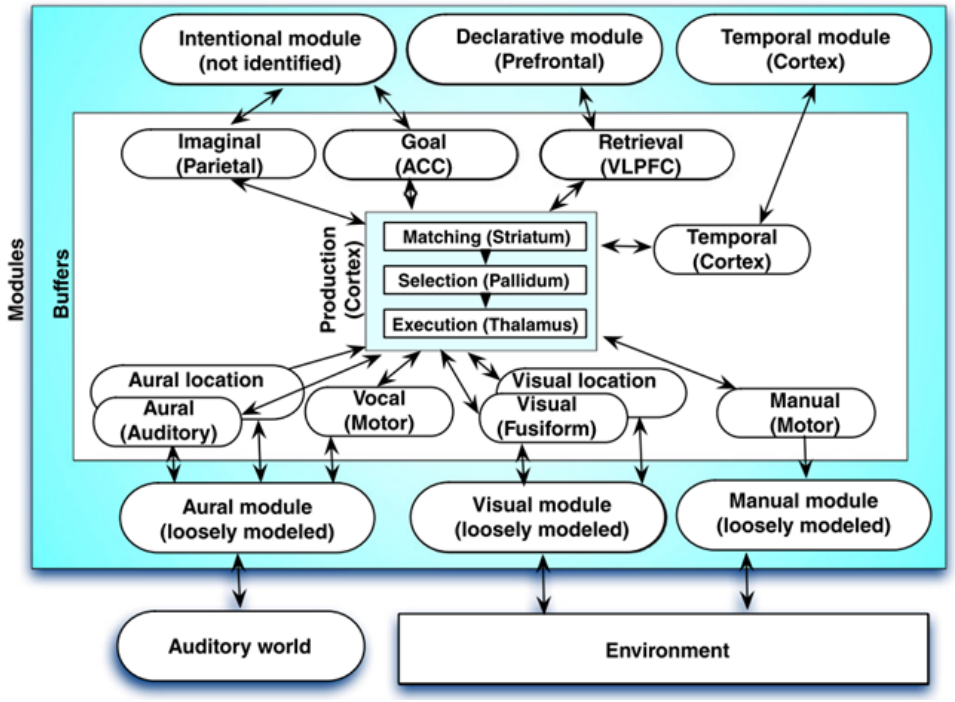


Figure 2: ACT-R cognitive architecture and how its components work together. [2]

speed and selectivity of attentional processes. This integration enhances the understanding of how high-level cognition and cognitive processes interact with visual attention and how attentional focus influences information processing within the architecture [2].

To validate the effectiveness of the ACT-R model, researchers have explored its relationship with fMRI readings. These studies involve mapping the ACT-R modules to specific brain regions using model-based fMRI analysis. By examining brain activity patterns during cognitive tasks, researchers can identify the neural substrates associated with different aspects of ACT-R processing. This approach provides insights into the correspondence between ACT-R's cognitive processes and the underlying neural mechanisms, shedding light on the neural basis of human cognition and memory retrieval processes [28].

2.3.2 CLARION

The second cognitive architecture in the study is the CLARION, which implies that human cognition is a dual-process/dual-representation [3], that focuses on the distinction between explicit and implicit cognitive processes. Also, this model is integrative, involving cognition, motivation and meta-cognition in a single state [29]. It is an integrative cognitive architecture, consisting of a number of distinct but symbiotic subsystems (with critical mutual dependencies and complex interactions) connected as a dual-representational structure in each subsystem.

The top level of CLARION (as in Figure 3) contains all the explicit knowledge, which is easily accessible but requires more attention resources. In contrast, the bottom level contains all the implicit knowledge, which is harder to access but mostly automatic. Essentially, it is a dual-process theory of mind [3]. Since the data in these two levels is different, many researchers show that it is justified to integrate the processing results to capture the interaction between the implicit and explicit processing in humans. [7].

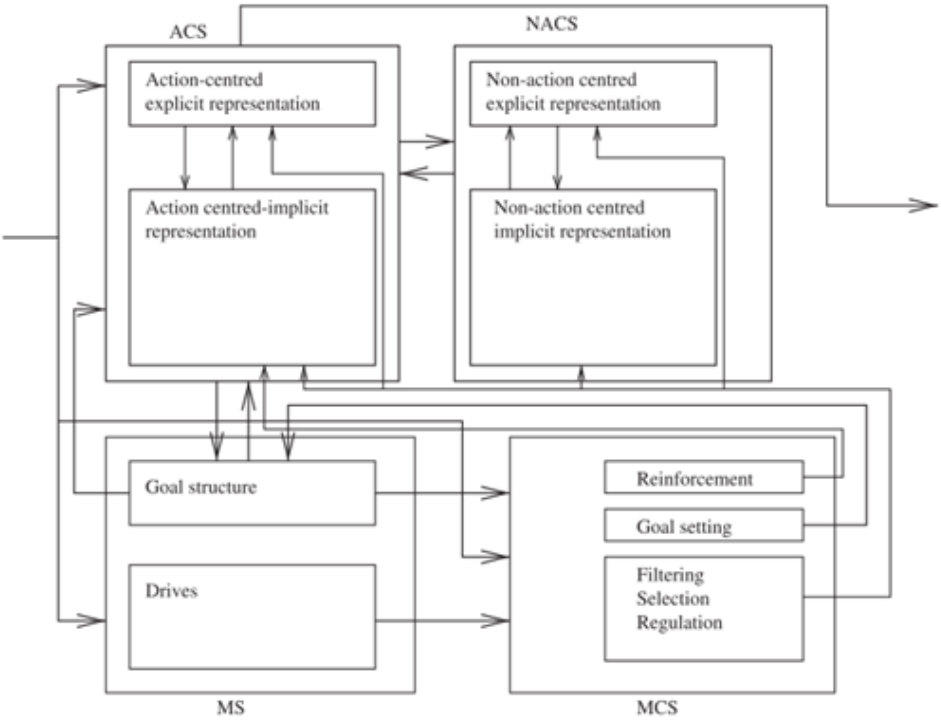


Figure 3: A high-level representation of CLARION.[3]

CLARION can then be further divided in the Action-Centred Subsystem (ACS), that contains procedural knowledge concerning actions and procedures and can be referred as the long-term procedural memory; and in the Non-Action-Centred Subsystem (NACS), which contains declarative knowledge as it serves as the long-term declarative memory, both semantic and episodic.

The Action-Centred Subsystem can then be further split into two parts: the bottom level and the top level. At the bottom level, the context of Implicit Decision Networks is implemented. At this level, knowledge is less accessible and reactive, and the distributed representation is used to capture the inaccessible nature of implicit (tacit) knowledge. The inputs to this part of the module can be sensory, working memory items, and the current goal. These inputs are represented as dimension-value pairs, and the actions are represented as nodes on the output layer, each consisting of one or more dimensions. Chunking is also introduced as a collection of dimension/value pairs that represent either conditions or actions of rules of the top level. This top-level deals with explicit rules that are more accessible and consciously

applied. It contains "condition →action" pairs with condition chunk and action chunk. These rules come from different sources, including extracted and refined rules (RER rules), independently learned rules (IRL rules), and fixed rules (FR rules). Each rule comprises one condition chunk and one action chunk with multiple dimensions connected to bottom-level (micro) features.

The Non-Action-Centred Subsystem also incorporates explicit and implicit knowledge in task performance. It emphasized the coexistence and redundancy of explicit and implicit knowledge and their integration in iterative processing. The NACS includes semantic and episodic memory, which are formed through acquiring and assimilating general knowledge from external sources or experiences. Since both systems are connected, it also performs memory retrievals and inferences under the control of the ACS. In contrast, the previously mentioned includes procedural memory, which involves long-term and short-term memory of learned procedures.

In sum, the action-centred modules include implicit bottom-level action decision networks, explicit top-level action rule groups, working memory and gold structure. Non-action-centred modules include explicit general knowledge stores (for instance, an explicit semantic memory) and implicit associative memory networks, such as implicit semantic memory. These two sets of modules constitute the two subsystems referenced above that operate under the influence of motivational and meta-cognitive processes.

A more recent version of the CLARION model as the one presented in [30] and as seen in Figure 3 also shows two additional subsystems: the Motivational System (MS) that is concerned with drives and their interactions [31]. The other subsystem is referred to as Meta-Cognitive System (MCS) and is closely tied to the MS since it monitors, controls and regulates the cognitive processes to improve cognitive performance. These two subsystems set essential parameters of the ACS and NACS, interrupting and changing the ongoing processes in these, making them responsible for achieving the best possible learning process [32].

Summing the essential characteristics, CLARION displays the dichotomy of implicit and explicit processes, focusing mainly on the cognition-motivation-environment interaction. Furthermore, it bases itself on the constant interaction of multiple subsystems and modules involving implicit cognition, explicit cognition, motivation and meta-cognition (described further ahead).

Given the structural complexity of this architecture, questions arises on whether it has too many mechanisms. CLARION is generally grounded in existing psychological theories, constitutes a comprehensive psychological theory by itself, is reasonably compact, and matches a wide range of psychological data. [33, 34]

CLARION can then learn independently, regardless of whether there is *a priori* or externally provided

domain knowledge. At the same time, it does not exclude innate biases, innate behavioural propensities or prior knowledge. [23].

2.3.3 SOAR

SOAR stands for State, Operator, And Result [4], is a cognitive architecture that will be the main focus of this dissertation. Like the other models presented before, SOAR aims to create a computational structure underlying general intelligence. It has many shared elements with both ACT-R (referenced before) and Learning Intelligent Distribution Agent (LIDA), not researched since it became obsolete with the development of more advanced models. Recently, SOAR has stepped away from ACT-R since it has a bigger emphasis on general AI, whereas ACT-R is focused on cognitive modelling. [35].

The SOAR's original name, for State, Operator And Result, refers to the problem and hypothesis which underlies SOAR creation. The original theory behind it is a "Problem Space Hypothesis" [36] (first out of five original hypotheses), which contends that all goal-oriented behaviour can be cast as a search through a space of possible states while attempting to achieve the designed goal. A single operator is selected and applied to the current state for each process step. This selection leads to internal changes like retrieving knowledge from one of the memories or even an external action with the world, generating a result.

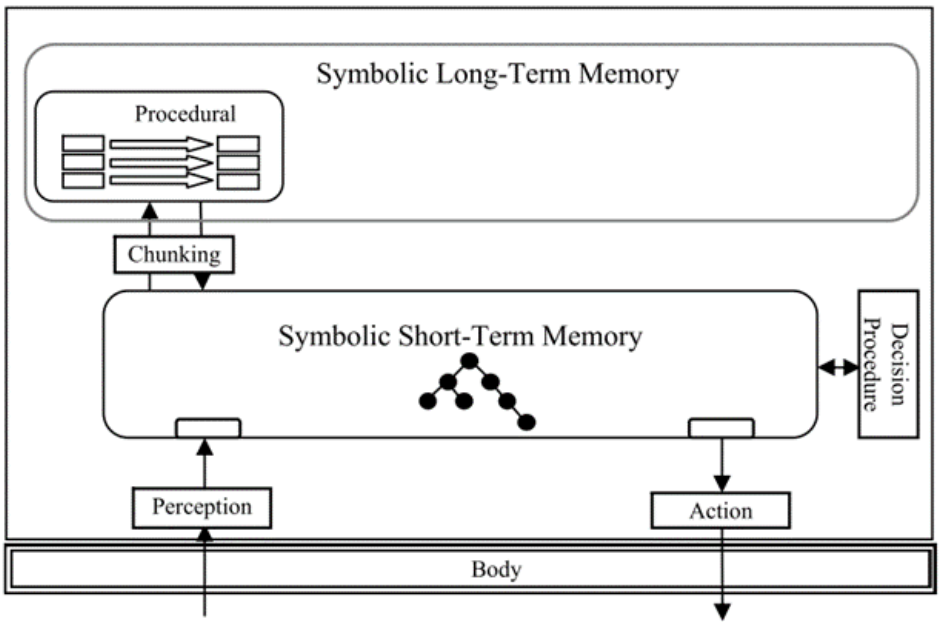


Figure 4: Structure of traditional SOAR.[4]

The traditional SOAR model, as shown in Figure 4, consists of a single long-term memory, encoded as production rules and a single short-term memory, representing a symbolic graph structure so that objects can be represented with properties and relations. These memories hold the agent's assessment

of the current situation, derived from perception and retrieval of knowledge from the other memory. At the lowest level, SOAR's processing consists of matching and firing rules (RBR and CBR). This method provides a flexible, context-dependent representation of knowledge, with their conditions matching the current situation and their actions retrieving information relevant to the current situation.

With this approach, it becomes visible that there will be limited knowledge available to choose between rules, especially in the conditions of the rules, which will need more data to be picked out of the others. When introducing operators, SOAR makes it possible for the rules to act as associative memory, retrieving essential and relevant information to the current situation, firing rules in parallel, and adapting in different ways to the same situation [4, 35].

The most recent version of SOAR, although it retains the strength of the original SOAR, provides a flexible model of control and meta-reasoning, expanding the types of knowledge SOAR could represent, reason, and learn. This evolution makes it more inspired by human capabilities, always adding functionalities [37].

Figure 5 [5] shows the improved and most current version of SOAR. The major additions to this new edition include the working memory activation, which provides extra meta-information about the recency and how useful a working memory element is for the situation. A new reinforcement learning engine tunes the numeric preferences of operator selection rules, and the appraisal detector generates emotions, feeling and an internal reward signal for reinforcement learning.

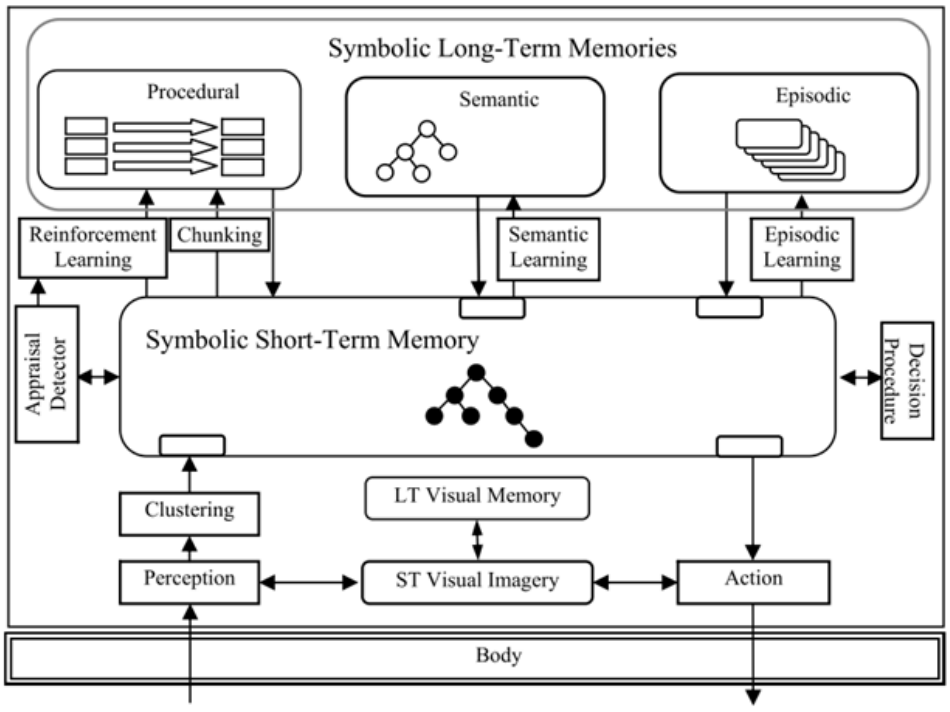


Figure 5: SOAR 9.[5]

This model also includes two new types of memory [38]: semantic memory, which comprises symbolic structures representing facts and episodic memory, temporally ordered snapshots of the working memory, quickly accessed without having to go through a slow shorting mechanism.

SOAR's processing cycle is still driven by procedural knowledge encoded as production rules. However, the new components added will influence decision-making indirectly by retrieving or creating structure in the symbolic working memory, causing different rules to match and fire. This model can now also be used for visual-feature and visual-spatial reasoning. It includes a short-term memory where images are constructed and manipulated, added to a long-term memory that contains images that can be retrieved and manipulated, a process that may create symbolic structures from these visual images. Visual imagery is controlled by the symbolic system, which issues commands to construct, manipulate and examine visual images [5, 39].

A practical example of the usage of this cognitive architecture is in a humanoid service robot capable of executing simple action skills: navigating, grasping and recognising objects or people, while using the SOAR cognitive architecture as the reasoner by selecting which action the robot should complete, addressing it towards the goal [40, 41].

As shown above, these architectures aim to model a human-like behaviour in terms of a cognitive cycle, performed sequentially or in parallel, culminating in an action or decision. With this, a comparison between CLARION and SOAR is of all interest. Following the study by Lucentini *et al.* [42], it is possible to conclude that these architectures generalise and apply the concept of sub-symbolic (also called non-symbolic or "numeric"), where a "bottom-up" approach to learning is displayed.

As of SOAR, all the information obtained is stored in a Working Memory (WM), from where it is retrieved and used as an input/output mechanism. These Working Memory Elements are usually generated externally via a perception module and pre-processed before storage, making it a highly symbolic architecture. For CLARION, a diverse representation is more prominent, dividing each module into the two parts described before. In this case, the data is represented in dimension-value pairs, with the information type and its corresponding value. For the goal structure, SOAR has no built-in motivational process, unlike CLARION, which has an exclusive module for handling goals and motivations, capable of influencing the decision process for a specific situation.

With this, Lucentini *et al.* [42] conclude that SOAR is a predominantly symbolic architecture and CLARION, on the contrary, has a mixed approach that combines the benefits of both paradigms, symbolic and non-symbolic.

2.4 Meta-Learning

Meta-Learning is a term widely used in the artificial intelligence field. It is viewed as an understanding and adaptation of cognition on a higher level than merely acquiring knowledge [43]. That is, a person aware and capable of meta-learning can assess if the learning approach is correct and adjust it according to the requirements of a specific task.

The primary goal of a meta-learning system is understanding the interaction between the learning mechanism and the concrete contexts in which that mechanism is applicable [44]. To be precise, meta-learning monitors the automatic learning process in the context of the learning problems it encounters and tries to adapt its behaviour to perform better [45].

The meta-knowledge needed for this process can then be extracted from a previous learning episode on a single dataset or even from different domains. Having previous information about a learning subject can make the system capable of few-shot learning, providing the network or the cognitive model fast adaptation over a specific situation based on previously learned knowledge [46].

Few-shot learning, as the name implies, refers to the practice of feeding a learning model with a minimal amount of training data. This method is broadly applied in computer vision, where small training sample results are accurate predictions or actions, proving to be the go-to solution whenever a tiny amount of training data is available [47, 48].

In the field of cognitive science and cognitive architectures, most meta-learning comes from applying rules previously defined as the reasoning method, where the model decides a specific action depending on which rules are fired during the execution.

2.5 Rule-Based and Case-based Reasoning

Rule-Based Reasoning and Case-Based Reasoning have always been considered two critical and complementary reasoning methodologies in artificial intelligence, especially in systems with meta-learning capabilities. Over the years, there have been attempts to integrate these two types of reasoning to achieve the perfect generalised reasoning to be applied to any situation, capable of dealing with real-life situations and providing a comprehensive representation of a final action [15].

With a reasoner of these types, the system can perform cycles of match-resolve-act, where it matches an input to a pre-existing production rule (usually defined inside a memory of the model). It then checks for a conflict where multiple rules are set simultaneously and needs to be filtered by which one will have

more impact in the final action. Finally, the actions defined by the rules are executed, originating changes in the memory's content, whether to end the cycle or to adapt to the next iteration.

Aspects like this will also require extra caution when analysing whether pre-implemented rules will make sense in the current situation [49]. In some cases, wrongly set rules or cases for the reasoning engine may result in non-capable learning mechanisms where the model will not have the ability to evolve to new situations and will not be able to execute new actions. That is why their methodologies must be adapted to the situation's specifics, allied to knowledge in few-shot or even one-show learning.

2.6 Similar Work

Besides all the examples of similar work so far, some mentions are not directly connected to any of the areas described but also incorporate the end goal this dissertation aims to achieve: the CAD for pathology identification. In the context of medical image diagnosis, various studies and applications have explored the use of deep learning classifiers for X-ray imaging. However, these approaches typically do not incorporate a comprehensive cognitive architecture as a reasoning mechanism.

In the study by Majkowska *et al.*, deep learning models were employed to interpret chest radiography to detect various abnormalities such as pneumothorax, opacity, nodules or masses, and fractures on frontal chest radiographs [50]. The researchers aimed to assess the performance of these models by comparing their interpretations to reference standards adjudicated by radiologists. The evaluation of the deep learning models was adjusted to account for population characteristics, providing a comprehensive assessment of their diagnostic capabilities. The findings of this study highlight the potential of deep learning approaches in aiding radiologists with the interpretation of chest radiographs, contributing to more accurate and efficient diagnoses of pulmonary conditions.

The study conducted by Jaiswal *et al.* focused on using deep learning techniques to identify pneumonia in chest X-ray images [51]. Their research aimed to develop an automated system that could accurately detect pneumonia cases, achieving high accuracy and comparable performance to human experts. By training a deep neural network model on a large dataset of chest X-ray images, they demonstrated the potential of deep learning for enhancing pneumonia diagnosis and improving patient outcomes.

Alternatively, there have been comprehensive studies focusing on applying deep learning algorithms to detect critical findings in CT scans [52, 53]. Chilamkurthy *et al.* conducted a retrospective study to assess the performance of deep learning algorithms in detecting critical findings in head CT scans. Their findings, published in The Lancet, demonstrated the potential of deep learning models to aid in detecting

abnormalities, contributing to improved diagnosis and patient care [52].

Moreover, using machine learning techniques for X-ray imaging has also been explored. Notably, industry leaders like Google have been involved in research and development efforts in this area [54]. Their work has focused on leveraging deep learning and artificial intelligence to analyze chest CT scans, automating the analysis process and potentially assisting radiologists in identifying abnormalities and providing accurate diagnoses [53].

These studies highlight the significant advancements in medical imaging, with deep learning algorithms and machine learning techniques showing promise in enhancing the accuracy and efficiency of image interpretation. The integration of these technologies has the potential to revolutionise radiology practices, leading to more effective diagnoses, improved patient outcomes, and streamlined healthcare workflows.

Chapter 3

Architectural Planning

Image classification is a fundamental problem in computer vision, with a wide range of applications in fields such as medical imaging [55, 56, 57], surveillance [58], and biometrics [59], among others. Image classification aims to automatically assign a label to an input image based on its visual content. Traditional methods for image classification rely on hand-crafted featured and predefined classifiers, which are limited in their ability to capture the complexity and diversity of natural images. These methods often involve extracting specific features from the image, such as edges, corners or textures [60], and using these features to train a classifier, namely, a Support Vector Machine (SVM)[61, 62] or a K-Nearest Neighbours (KNN)[63]classifier. Modern methods for image classification continue to face limitations, albeit at a lower scale compared to traditional methods.

However, with the advent of deep learning, machine learning approaches have shown great promise for image classification. These approaches involve using neural networks to learn the representations and classifiers from the data. Convolutional Neural Network (CNN) have emerged as a powerful tool for image classification [64, 65, 66], due to their ability to learn hierarchical representations of images and their high accuracy in many tasks.

The decision to incorporate both symbolic and sub-symbolic structures in BorisCAD draws inspiration from the behaviour of the CLARION cognitive architecture. CLARION, a cognitive architecture that combines cognitive and neural processes, has demonstrated the benefits of integrating symbolic and sub-symbolic representations in cognitive tasks.

Similarly, in BorisCAD, combining symbolic Decision Trees (DTs) and sub-symbolic CNNs allows for a more comprehensive and practical approach to image classification. This integration leverages the strengths of both methods, enabling BorisCAD to capture the complexity and diversity of natural images while maintaining interpretability through the DT component.

3.1 The Blueprint: Designing a Brain-like Architecture

From Chapter 2 seems clear that the CLARION cognitive architecture, a theoretical framework that simulates the cognitive processes of the human mind, was the most versatile choice for adapting to the needs of this work. CLARION's flexibility and adaptability make it well-suited to address the specific requirements of the image classification task. Its ability to seamlessly integrate both symbolic and sub-symbolic processing allows for a comprehensive approach that combines interpretability with neural networks' powerful representation learning capabilities. As a result, CLARION provides a solid foundation for developing the proposed BorisCAD architecture, offering a promising solution to enhance the efficiency and effectiveness of image classification tasks.

The CLARION cognitive architecture is organised into two main subsystems: the explicit and implicit. The explicit subsystem is primarily symbolic, using logical and linguistic representations for reasoning about the world with conscious thought and decision-making. It contains a Working Memory (WM) module, which holds the current state of the environment and provides a basis for decision-making. This WM is a dynamic structure that allows for the integration of new information and the manipulation of existing information, allowing it to interact with other modules to determine the appropriate action based on the current state of the environment.

The implicit subsystem, on the other hand, is sub-symbolic, using distributed representations to process information. This subsystem contains a procedural module responsible for learning and executing complex behaviours, learning through trial and error, and adjusting its behaviour based on the feedback it receives from the environment.

Symbolic knowledge often refers to knowledge represented by symbols or rules associated with explicit, conscious, and rational reasoning processes. On the other hand, sub-symbolic knowledge represents knowledge shown as activation patterns in a neural network or other machine learning models associated with implicit, unconscious, and intuitive reasoning processes. Thus, the distinction between symbolic and sub-symbolic knowledge is crucial in a cognitive architecture like CLARION. It allows for the integration of high-level reasoning and low-level perception and action, helping to identify the processes best suited for each type of knowledge, making it an essential topic in this work.

The CLARION architecture also includes a motivational subsystem, which plays a crucial role in determining the goals and priorities of the system. It uses affective and motivational states to influence the decision-making process by affecting the goals and priorities of the system.

By comparing CLARION with other well-known cognitive architectures, it becomes evident why

CLARION is a better choice for the specific application at hand. One significant factor is CLARION's distinction between explicit and implicit processes, knowledge, and learning. This differentiation allows CLARION to provide a more comprehensive and adaptable framework than the ACT-R architecture.

Although a well-established cognitive architecture, ACT-R does not offer the same level of versatility as CLARION for the intended application. ACT-R lacks the explicit-implicit distinction crucial for effectively handling complex reasoning processes. In contrast, CLARION's explicit and implicit processes provide a more nuanced understanding of cognitive operations, facilitating better modelling of human-like cognitive behaviour.

Another factor that sets CLARION apart from other architectures, such as SOAR, is the requirement for a large amount of initial knowledge (a priori) to perform similarly to CLARION. SOAR does not differentiate between symbolic and sub-symbolic knowledge, which limits its versatility in handling more intricate reasoning tasks. In contrast, CLARION's ability to incorporate symbolic and sub-symbolic knowledge allows for a more flexible and nuanced representation of complex cognitive processes.

With an understanding of the importance of symbolic and sub-symbolic knowledge in cognitive architecture, it is possible to turn the attention to the development of the BorisCAD architecture. It will be built upon the strengths of the CLARION architecture and adapted to meet the specific needs of the problem at hand. This architecture aims to integrate both symbolic and sub-symbolic knowledge to allow for efficient high-level reasoning and low-level perception and action. To achieve this goal, BorisCAD will incorporate a variety of cognitive subsystems that work together to achieve the desired outcome.

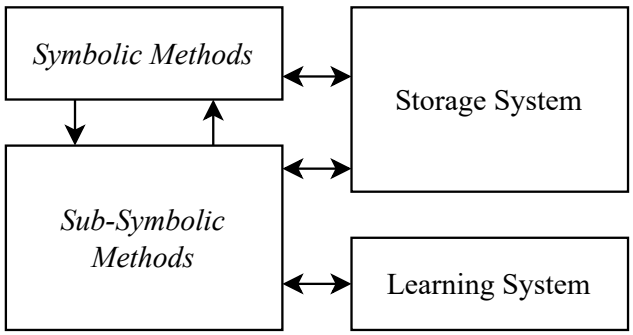


Figure 6: Modular disposition of the architecture.

Figure 6 illustrates the main components of the architecture, which are composed of four key modules. The first two modules are the symbolic and sub-symbolic, which are interconnected to integrate both types of knowledge. The sub-symbolic module is responsible for handling sensory information and lower-level processing. This will then generate explicit symbols for the symbolic module to handle, applying higher-level reasoning. The storage module plays a critical role in the architecture by providing a temporary

space for relevant prior information to be stored, guiding the decision-making process and maintaining context during task execution. This information can include sensory input, recent actions, and task goals. Finally, the learning module regulates the system's training by adjusting several hyperparameters that the sub-symbolic methods make use.

This representation shows how the modules will coexist in this architecture but still needs to display the type of components each module will comprise clearly. Figure 7 accurately depicts what BorisCAD will include in its structure.

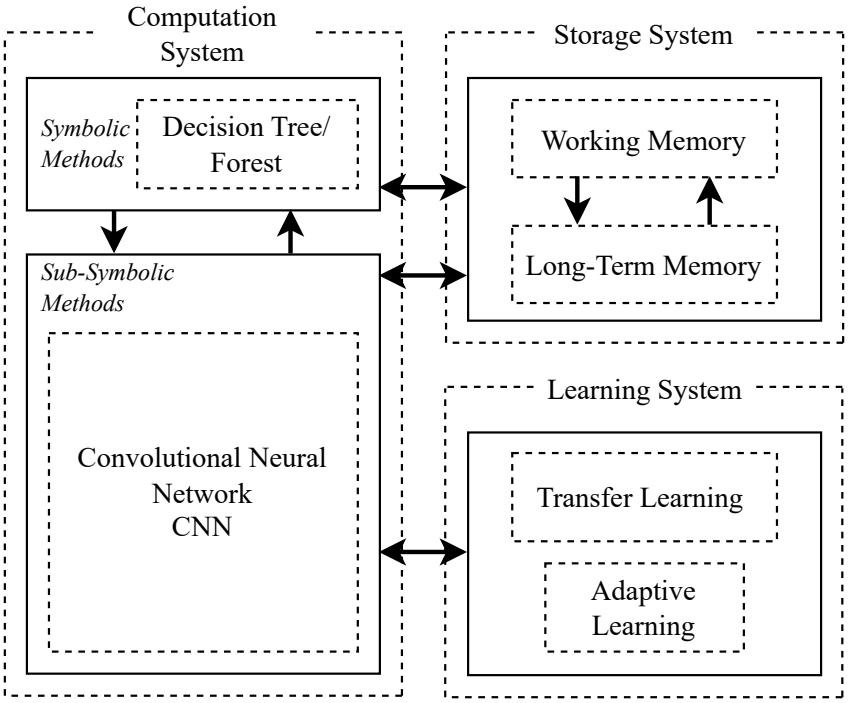


Figure 7: Structure of the architecture.

Convolutional Neural Networks (CNNs) were chosen as the sub-symbolic method within the cognitive architecture, primarily for their capability to extract meaningful features and gain a comprehensive understanding of image contents. CNNs excel in learning hierarchical representations from images, capturing intricate patterns and achieving high accuracy in classification tasks. They employ deep layers of convolutional and pooling operations to progressively recognise complex visual patterns, enabling them to encode image information into feature representations for classification effectively.

However, despite their high performance, CNNs may sometimes generate inconclusive classification, where the highest probability output aligns differently from the actual image class. By incorporating DTs into the architecture, the system takes advantage of the probabilities generated by the CNN during the forward pass by examining the connections between these probabilities and the initial training labels. This allows the DTs to identify complex patterns and correlations that may not be apparent solely based on the

highest probability value or a predetermined threshold. The DTs provide an extra layer of classification, which is particularly useful in situations where a high probability value does not accurately represent the proper class of the image.

The integration of DTs enables the architecture to account for situations where the CNN tends to overfit certain classes [67], such as "no-finding" in the case of medical diagnosis. By considering the probabilities and their relationships, the DTs can detect patterns that indicate the correct classification, even if the highest probability does not align. This enhances the accuracy and reliability of the classification process, reducing the risk of critical diagnostic errors.

Similarly to CLARION's architecture, a Working Memory (WM) was integrated into the architecture to provide a temporary storage space for relevant information. Additionally, a Long-Term Memory (LTM) stores information for extended periods.

The Learning System module, as a pivotal component in this architecture, plays a crucial role in regulating the training of the sub-symbolic components. It incorporates transfer learning methods [64, 68, 69, 70], allowing the system to leverage previously learned models for retraining on different datasets. It will also include similar methods to adapt currently trained models to new scenarios with different image sizes and a different number of classification labels.

Moreover, the Learning System module incorporates advanced techniques such as adaptive learning rate optimisation. By utilising methods like Adadelta [71] and others [72, 73], the system dynamically adjusts the learning rate during training. This enables the system to fine-tune the learning rate based on the training progress, ensuring that the model converges smoothly and avoids getting stuck in local optima. By adjusting the learning rate, the system can control the pace of learning, allowing it to navigate complex optimisation landscapes more effectively. This optimisation technique enhances the efficiency of the training process and improves the generalisation capabilities of the sub-symbolic components, ultimately leading to better accuracy and robustness.

With this, it is possible to say that BorisCAD can also be split into several levels in the Computation System (CS): it will be referred to as the top level, the group of symbolic methods, and as the bottom level, the modules that represent sub-symbolic items. In this case, the top level will consist of one or multiple Decision Trees, and the bottom level will consist of neural networks, namely, Convolutional Neural Networks.

3.2 Computation System: The Cognitive Engine

BorisCAD's Computation System incorporates a hybrid system design with sub-symbolic processing at the bottom level and symbolic reasoning at the top level. The sub-symbolic level represents the perception and recognition processes of the brain, while the symbolic level corresponds to procedural rules. This duality reflects the information processing mechanism of the human brain, where the sub-symbolic processes extract information from sensory data, and the symbolic processes use that information to make decisions and act.

At the bottom level, BorisCAD employs neural networks, specifically Convolutional Neural Networks, to extract and learn features from image data. These sub-symbolic processes learn complex representations of images, enabling the system to recognise objects and patterns in visual data. At the top level, the system uses symbolic reasoning to make decisions based on the features learned by the sub-symbolic processes. Decision Trees are employed for their interpretability and flexibility, allowing the system to reason about complex relationships between features and make accurate predictions.

3.2.1 Sub-Symbolic Processing: The Bottom Level

The representation of image data is a two-dimensional grid of pixels, whether monochromatic or colour. Each pixel corresponds to one or multiple numerical values, respectively. However, the spatial relationship between pixels is often ignored, and images are flattened into one-dimensional vectors to feed through a fully connected MLP. This approach was revised because it disregards the relationship between nearby pixels that can be leveraged to build efficient models for learning from image data.

Convolutional Neural Networks [74], a family of neural networks designed for learning from image data by exploiting the spatial relationship between pixels, were introduced to address this issue. CNN-based architectures have become ubiquitous in computer vision and have significantly improved performance on datasets such as Imagnet [75].

CNNs owe their design to inspirations from biology, group theory, and experimentation. They are computationally efficient due to requiring fewer parameters than fully connected architectures and are easily parallelised across GPU cores [76]. Consequently, CNNs have emerged as credible competitors even on tasks with one-dimensional sequence structures, such as audio [77], text [78], and time series analysis [74], where recurrent neural networks are conventionally used. Furthermore, clever adaptations of CNNs have enabled them to be applied to graph-structured data [79] and recommender systems.

CNNs exploit spatial invariance to detect objects in images. The idea is that the method of recognising

objects should not be overly concerned with the object's precise location in the image. The network should focus on local regions in the earliest layers without regard for the contents of the image in distant regions. In contrast, deeper layers should capture longer-range features of the image in a way similar to higher-level vision in nature. To accomplish this, the earliest layers of the network should respond similarly to the same patch, regardless of where it appears in the image. This concept is called translation invariance (or translation equivariance). To achieve this, CNNs use a convolutional layer, a fourth-order weight tensor that convolves an input image with learnt filters to produce a feature map.

Adding channels to CNNs can help reduce the complexity lost due to the restrictions imposed by translation invariance and locality, where channels allow different types of information to be processed separately. These networks manipulate the dimensionality of images, move the information from location-based to channel-based representations, and efficiently handle large numbers of categories, processing and extracting valuable data from images.

These networks operate on images by cross-correlating the input with a kernel and adding a scalar bias to produce an output. The kernel and scalar bias are the two parameters of a convolutional layer, randomly initialised during training, similar to a fully connected layer. Each feature map in a CNN represents learned representations (features) in the spatial dimensions of the subsequent layer. Convolutional filters used in these operations are flexible and capable of detecting edges and lines and performing operations like blurring or sharpening images.

Convolutional Neural Networks use receptive fields to capture spatial dependencies and detect these image patterns. The receptive field refers to the input's effective region that influences a neuron's computation in a given layer. As information flows through successive convolutional layers, the receptive field expands due to convolution and pooling operations. Each neuron in deeper layers considers a larger input context, capturing more global information. This hierarchical integration of information enables CNNs to extract higher-level features for tasks like object recognition and semantic segmentation.

Segmentation and classification networks are two types of neural networks often used in computer vision tasks. While they have different goals, they can work together to improve image classification accuracy.

Segmentation networks are designed to classify every pixel in an image, grouping them into object, background, and foreground categories. They are often used for tasks such as object detection and scene understanding. These networks use an encoder-decoder architecture to extract features from the input image and produce a segmentation map. On the other hand, classification networks are designed to classify an entire image into predefined categories, such as different types of animal or object. These

networks use a simpler architecture than segmentation networks, often consisting of just a few convolutional and fully connected layers. Although these two types of networks have different goals, they can work together to improve image classification accuracy. By using a segmentation network to identify and classify different regions of an image, a classification network can classify the image based on these regions. This process can reduce the impact of noise and other factors that affect image classification accuracy.

In this case, the segmentation step extracts the relevant foreground information from the images, explicitly isolating the lungs from the surrounding background. While there is a scarcity of available datasets specifically focused on semantic segmentation of the lungs (such datasets are more prevalent for brain MRI), this limitation is inconsequential to the task.

In this context, the segmentation process enables the classification model to operate exclusively on the lung regions, removing unnecessary or irrelevant background information. By segmenting the lungs, the subsequent classification step can effectively focus on analysing and categorising the extracted lung images without interfering with unrelated image components containing the essential diagnostic features to make accurate predictions and provide valuable insights.

a) Segmentation Network

U-Net is a convolutional neural network architecture designed for semantic image segmentation tasks. It was introduced by researchers at the University of Freiburg in 2015 [80] and has since become a popular choice in medical image analysis and other image segmentation applications.

The architecture of U-Net consists of a contracting path and an expansive path, which together form a U-shape, hence the name "U-Net". The contracting path is similar to a typical convolutional neural network and consists of a series of convolutional and pooling layers that reduce the spatial resolution of the image while increasing the number of feature channels. The expansive path is a mirror image of the contracting path and consists of a series of convolutional and up-sampling layers that increase the spatial resolution of the image while decreasing the number of feature channels.

One of the critical innovations of U-Net is the use of skip connections between the expanding and contracting paths. These skip connections allow the network to retain fine-grained spatial information while capturing high-level features at different scales. The skip connections are used to concatenate feature maps from the contracting path to the corresponding feature maps in the expansive path, allowing the network to combine low-level and high-level features to make accurate segmentation predictions. U-Net is particularly effective in medical image segmentation tasks, such as segmenting tumours or organs

from medical images, adressing the issue at hand: extracting the foreground of chest x-ray images.

In recent years, researchers have demonstrated that introducing an attention mechanism into the U-Net can enhance local feature expression and improve segmentation performance, thus improving medical image segmentation. In a study presented in [81], attention gates were added to the skip connection of the U-Net architecture to highlight salient feature information while disambiguating irrelevant and noisy feature responses. The resulting model, AGResU-Net, achieved competitive performance in three authoritative MRI brain tumour benchmarks. Another study presented in [82] proposed a novel attention gate model that automatically learns to focus on target structures of varying shapes and sizes in medical imaging. The proposed Attention U-Net architecture was evaluated on two large abdominal CT datasets, consistently improving the prediction performance of U-Net across different datasets and training sizes while preserving computational efficiency. In [83], an attention-gated network was applied to real-time automated scan plane detection for fetal ultrasound screening, incorporating self-gated soft-attention mechanisms to improve object location and reduce false positives. The proposed attention mechanism is generic and can be easily incorporated into any existing classification architectures, with generated attention maps allowing for a better understanding of the model's reasoning process.

The Segmentation Network at the bottom-level of the BorisCAD architecture is a variant of the U-Net architecture that incorporates attention gates to improve the local feature expression and segmentation performance, similar to the AGResU-Net and Attention U-Net models discussed in the literature. The block diagram of this network will resemble the U-Net architecture, but with additional attention gate blocks incorporated into the skip connections, as shown in Figure 8.

This U-Net with attention gates comprises down-sampling and up-sampling layers. One of this network's critical features is skip connections, equipped with attention gates to highlight salient information while disambiguating irrelevant and noisy feature responses. These skip connections are concatenated between each up-sampling layer and its corresponding down-sampling layer, allowing the network to retain fine-grained spatial information while capturing high-level features at different scales.

The network takes an input image and passes it through the contracting path, which consists of the down-sampling layers. The down-sampling layers reduce the spatial resolution of the input image while increasing the number of feature channels by applying a maximum pooling after each step. Then, the network passes the feature maps through the expansive path, which consists of the up-sampling layers. The up-sampling layers increase the spatial resolution of the image while decreasing the number of feature channels. At each up-sampling layer, the feature maps are concatenated with the corresponding feature

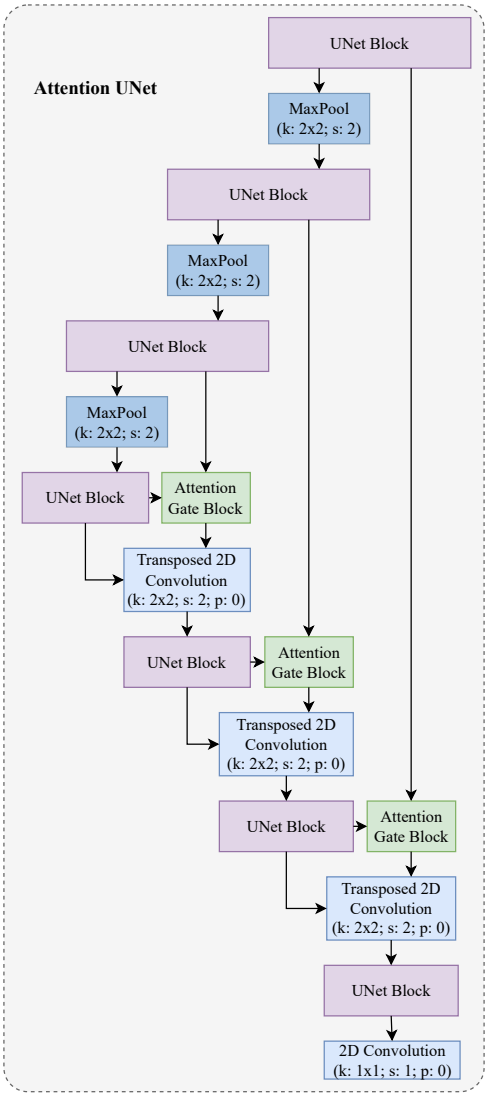
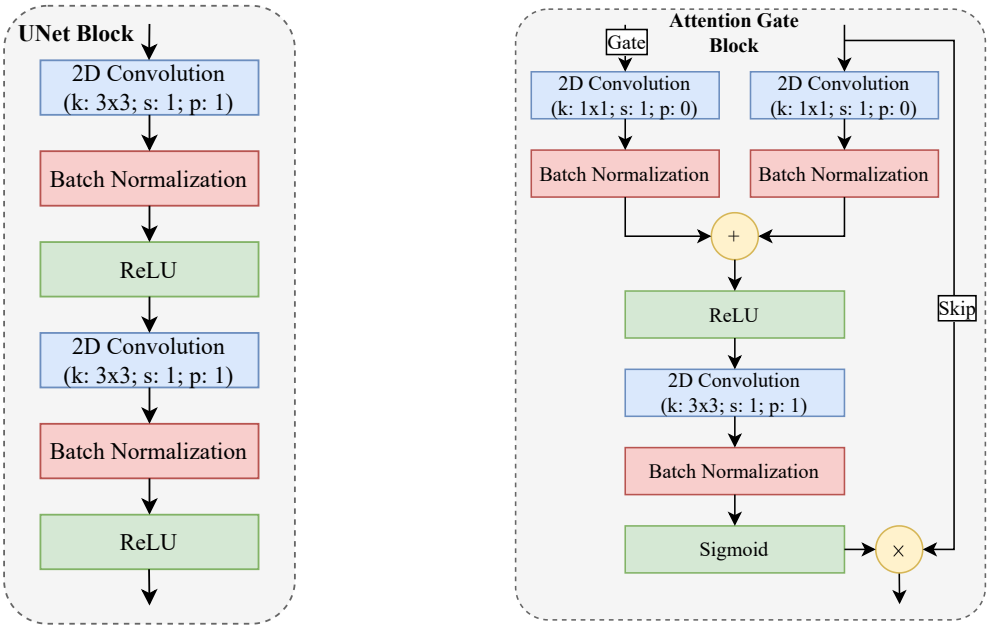


Figure 8: AttentionU-Net Block Diagram.

maps from the down-sampling path, passing through an attention-gate operation to refine the features, generating the final binary mask of the input image.

Figure 9 illustrates the two types of blocks used in the design. The U-Net block, shown in Figure 9a, consists of a 2D convolutional layer that applies 3x3 filters to the input feature map with zero padding to maintain the original size and spatial resolution. The batch normalisation is then applied, followed by the Rectified Linear Unit (ReLU) activation function. This process is repeated twice for each block to capture more complex features and abstract patterns in the input data. In the up-sampling process, an inverse convolution is computed before this block to increase the spatial resolution of the feature maps.

Figure 9b depicts the Attention Gate Block, which selectively focuses on the most informative parts of an image during the segmentation process. This block takes in two inputs: the gate map from the previous block and the precomputed skip connection. Both inputs undergo convolution, normalization,



(a) U-Net Block. (b) Attention Gate Block.

Figure 9: AttentionU-Net: Attention Gate Block and U-Net Block

and addition operations, followed by a ReLU activation. Next, a convolutional layer is applied, followed by batch normalization and a *sigmoid* activation function. The *sigmoid* activation enables element-wise multiplication between the original skip connection, containing information from the contracting path, and the weight map. This process generates the final output of the attention gate block, emphasizing the relevant features for the segmentation task.

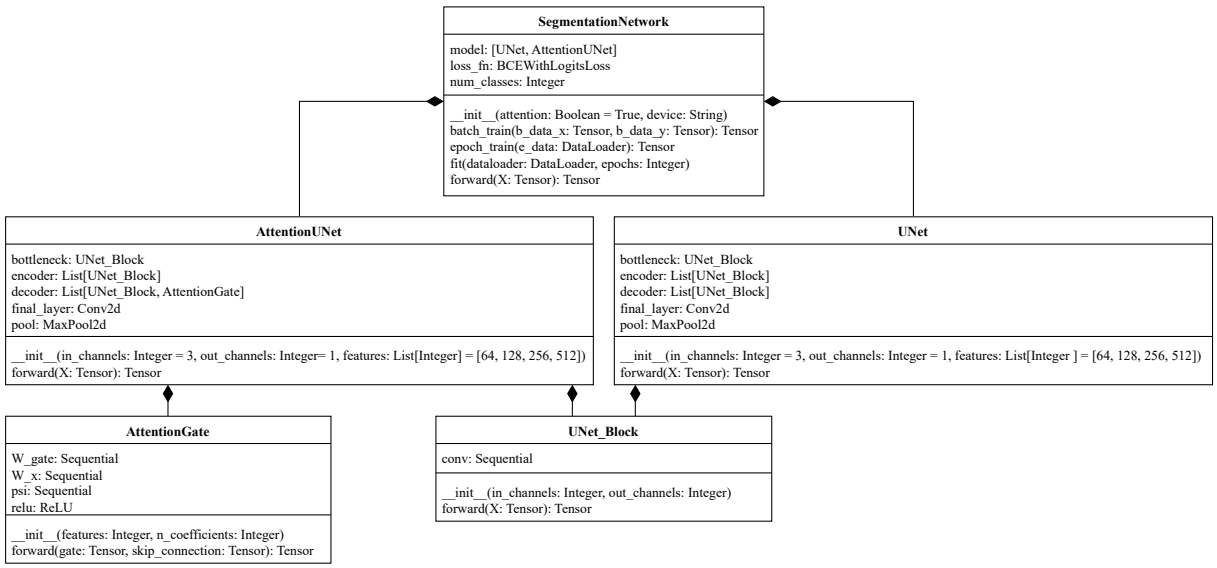


Figure 10: Segmentation Network UML.

The UML diagram of the Segmentation Network, as depicted in Figure 10, provides an overview of the potential connections between the standard *U-Net* network and the enhanced *AttentionU-Net*, which

were described earlier. This design takes into account the user's discretion in selecting either of these networks based on their specific requirements. The methods employed to train the neural network will be elaborated upon in the comprehensive analysis of the complete classification network. Although they share a similar process, the metrics utilised for evaluation are distinct and will be discussed separately.

b) Classification Network

ResNet (short for "Residual Network") is a deep neural network architecture designed to address the problem of vanishing gradients while training very deep neural networks. It was introduced by He *et al.* [84] and has since become a widely used architecture in various computer vision tasks, such as image classification, object detection, and segmentation.

The critical innovation of ResNet is the use of residual connections, which enable the network to learn residual functions rather than directly learning the underlying mapping. These residual connections add shortcut connections that bypass one or more layers and allow the network to propagate gradients more effectively, which in turn helps to alleviate the problem of vanishing gradients and enable the training of very deep networks.

The ResNet architecture consists of a series of residual blocks, each containing one or more convolutional layers and shortcut connections. The basic residual block has two convolutional layers, each followed by a ReLU activation function and a shortcut connection that adds the input to the output of the second convolutional layer. Variations of the residual block can also be added, such as the bottleneck block, which uses 1x1 convolutions to reduce the number of channels before the addition to the 3x3 convolutions.

Figure 11 shows the complete structure of a ResNet network. For starters, there are three main groups of layers to be analysed. The network input consists of a sequence of operations: a single convolution, a batch normalisation, and a ReLU activation. After this, a Max-Pooling operation is applied to down-sample the output feature maps, reducing their spatial dimensionality and increasing their level of abstraction. The output of a network of this type consists of an average pooling operation, which computes the average value of each feature map across its spatial dimension, reducing the number of parameters in the network, and improving its robustness. This result is then flattened to be fed into a fully-connected layer, which maps the feature vector to the output classes or categories, with a dimension equal to the total amount of labels to be extracted.

As described previously, the ResNet structure will consist of the input and output sequences shown above and a series of blocks, with their designs depicted in 12. The identity block shown in figure 12b

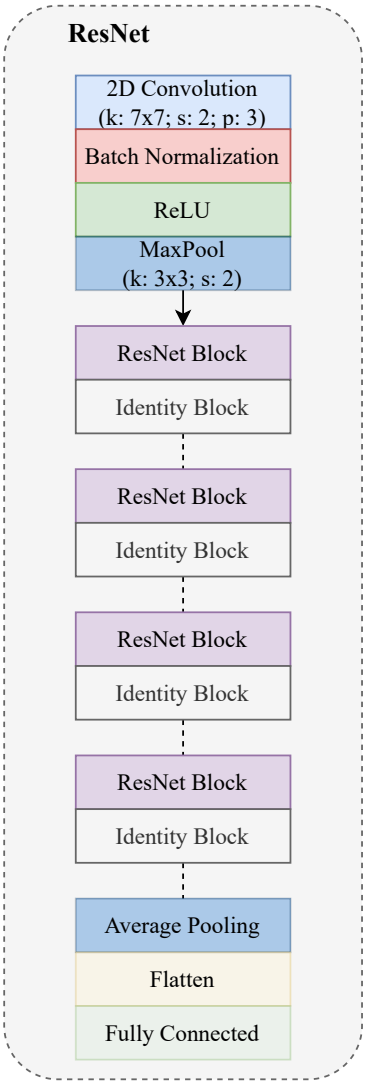
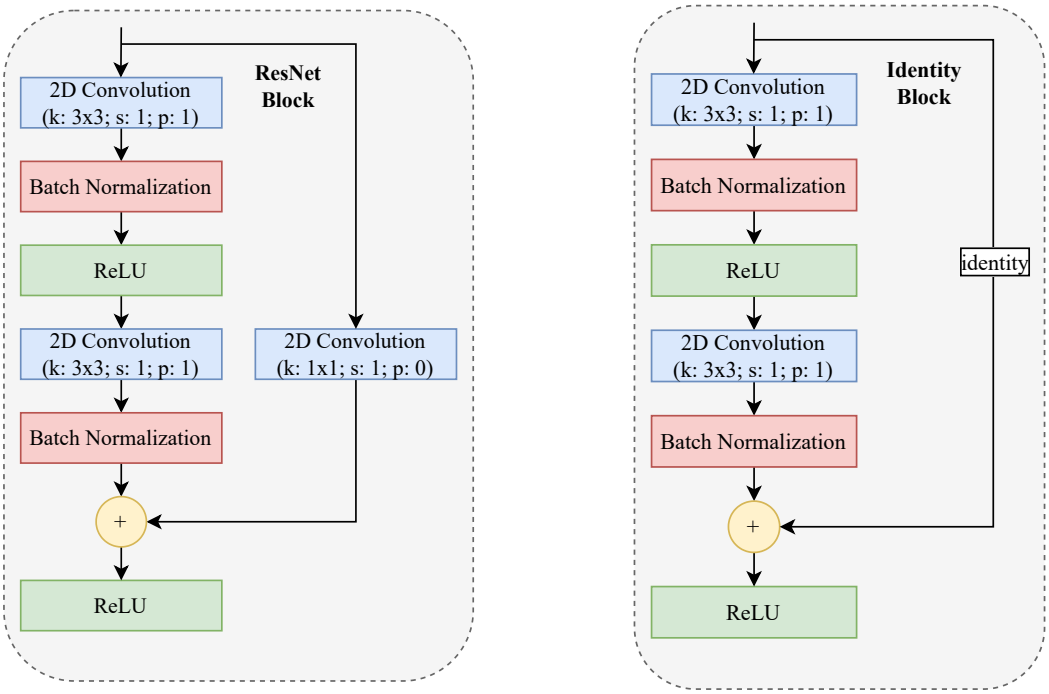


Figure 11: ResNet Block Diagram.

consists of two convolutional layers that aim to present the input's dimensionality. This block's skip connection directly sums the module's input to the output of the second normalisation, where the result passed through a ReLU activation to generate the final feature map. This shortcut connection is a simple identity mapping that bypasses the convolutional layers and allows the gradient to flow directly through the block without affecting the input feature map. The ResNet or bottleneck block shown in figure 12a does the same operation. However, it includes a 1x1 convolutional layer that changes the number of channels on the input feature map prior to the add operation. This convolution increases or decreases the number of channels in the feature map to adapt the network to different target domains.

The illustrious table 1 presents a comprehensive overview of the various sizes available for the ResNet architecture, namely ResNet18, ResNet34, ResNet50, ResNet101, and ResNet152. On closer inspection, a subtle modification can be discerned in the ResNet50 and its superior counterparts. Specifically, the



(a) ResNet Block. (b) Identity Block.

Figure 12: ResNet: ResNet Block and Identity Block.

skip connection that once skipped two 3x3 convolutional layers now traverses three layers, where only one layer features 3x3 kernel filters, flanked by two 1x1 convolutions. Regrettably, the first two sizes of this network fail to adequately capture the intricate dimensions required for medical image classifications. As such, only the latter three sizes shall be examined in the remainder of this study.

Table 1: The five types of ResNet (adapted from[6])

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	$7 \times 7, 64$, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	1×1	average pool, X-d fc				

On the ResNet network, the two types of blocks shown in Figure 12 are used. For example, if there is a sequence of three filters modules in the *conv_2* group of filters (as shown in table 1), the first block will always be a bottleneck block, while the rest will be identity blocks. This operation is performed at every stage and for every size. Further details on this process will be discussed during the implementation

analysis.

The UML diagram for the Classification Network, presented in Figure 13, displays the classification component at the system's lower level. Like the Segmentation Network, this component features distinct methods for training the neural network model.

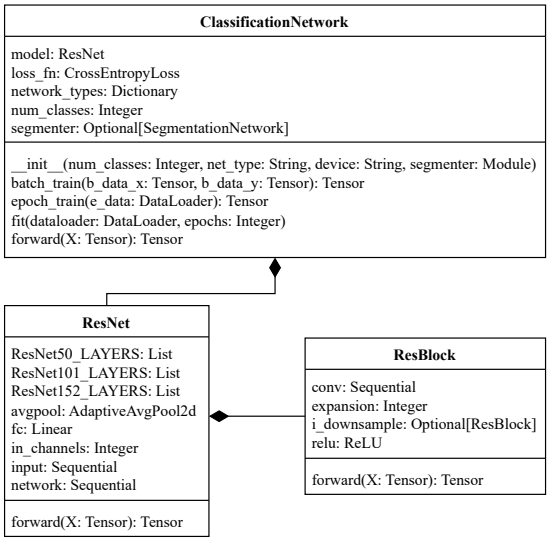


Figure 13: Classification Network UML.

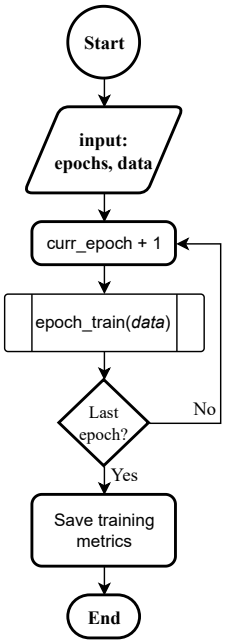
Commencing with the "fit" method, it assumes responsibility for dividing neural network training into multiple epochs. This method invokes the "epoch_train" process, denoting a finer level of granularity. Figure 14a visually represents the flowchart outlining its sequential steps.

Conversely, the "epoch_train" method, illustrated in Figure 14b, entails partitioning the accessible data into batches. The batch size is predetermined, and this method facilitates the model's training using all available data for each epoch.

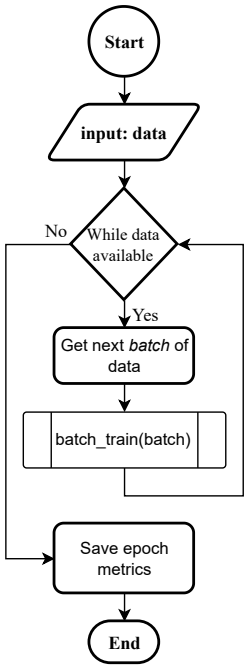
Furthermore, the batch training procedure, depicted in Figure 15, shows the sequential execution of a complete training cycle on a batch of information. The data is fed forward through the model, leading to the calculation of a loss. Subsequently, this loss is utilised to perform back-propagation, updating the model's kernels and weights. In addition, metrics are computed for analytical purposes, and optimises are adjusted accordingly (further elaborated in the subsequent section 3.4: Learning System).

Notably, the classification and segmentation models exhibit a sole disparity in their training processes. Specifically, the classifier harnesses information from the segmentation, provided a segmentation model is available. Naturally, this scenario does not apply when training the segmentation model itself.

In summary, the segmentation technique serves to discern and isolate important information embedded within an image, thereby facilitating a more nuanced and precise comprehension of its contents. The fundamental elements and their interrelationships within the scene can be identified by



(a) "fit" method flowchart.



(b) "epoch_train" method flowchart.

Figure 14: Classification Network fit and epoch training flowcharts.

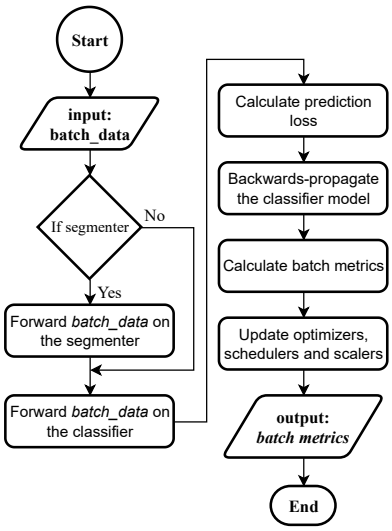


Figure 15: Batch training method's flowchart for the Classification Network.

segmenting an image into different regions or objects. Subsequently, the classification network can utilise this information to effectuate more reasonable and informed determinations regarding the image's content. Segmentation thus represents a pre-processing step that extracts discerning features from the image and provides a more elaborate and refined representation of its content, subsequently facilitating enhanced accuracy and adaptability. The Computation System's Bottom-Level architecture was consequently formulated to incorporate both types of networks described above, in order to achieve the desired accuracy and adaptability. Figure 16 displays the UML for the system's Bottom-Level,

showcasing the connections between both networks and itself.

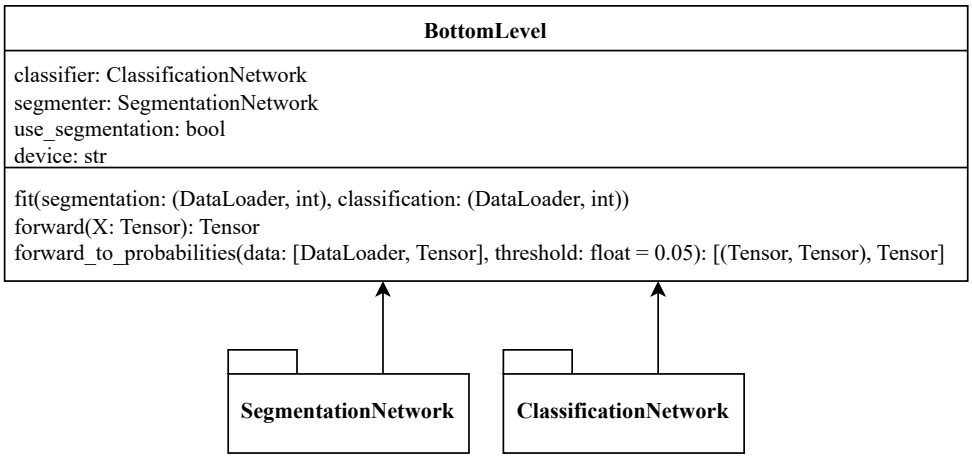


Figure 16: Bottom level UML design.

Figure 17 below accurately depicts the data type and shapes at the connections between the modules. The image is forwarded in the segmentation network in the first stage. This stage will generate two new images: the mask image and the resulting multiplication between the input and this binary information, giving the background-less version of the original image. The isolated version is then forwarded to the classification network to generate the corresponding probabilities.

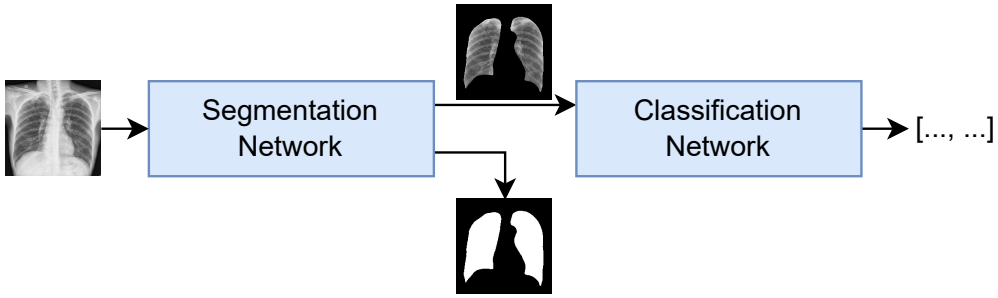


Figure 17: Bottom-Level Network Outputs.

During this process, some other details will be gathered, such as activations between layers of the neural networks to be stored in the memories for future retrieval. This is further detailed ahead in 3.3: the Storage System Module.

3.2.2 Symbolic Reasoning: The Top Level

Extracting knowledge from domain experts to build expert systems has been challenging since the early days of knowledge-based systems. The interview process, where a knowledge engineer interacts with a domain expert to extract and refine a set of rules, has been a common approach. However, as

Waterman *et al.* [85] pointed out, "the pieces of basic knowledge are assumed and combined so quickly that it is difficult for him to describe the process", making it challenging and time-consuming for all parties involved. As a result, knowledge acquisition has been identified as the "bottleneck" problem in building knowledge-based systems [86]. To overcome this bottleneck, some researchers, such as Michie *et al.* [87], have advocated using inductive methods to extract general rules from concrete examples. In this approach, the expert provides a framework of essential concepts in the task domain, augmented by tutorial examples. A suitable induction engine carries out the hard work of extracting knowledge.

Decision Tree induction[88, 89] provides a methodology for building systems that make decisions based on structured knowledge. Decision Trees are a rule-action system because they represent knowledge as a set of rules, each corresponding to a condition and an action. The condition is a set of input attributes, and the action is the decision that should be taken when those attributes are present.

A decision tree is a tree-like model that uses a branching structure to represent decisions and their consequences. It is a form of supervised learning that takes a set of attributes as input and outputs a decision based on those attributes. Each internal node of the tree corresponds to a test on an attribute, and the branches that emanate from the node represent the possible outcomes of the test. The tree leaves correspond to decisions or actions based on the path from the tree's root to a leaf node.

The decision process in a decision tree and the inference process in a neural network differ in their decision-making approaches. In a decision tree, the input attributes traverse the tree's internal nodes until a leaf node is reached corresponding to the decision or action to be taken. In contrast, a neural network uses weights and biases to determine the input's contribution to the network's output, which results from the inference process.

Using a decision tree to make decisions based on the probabilities generated by a neural network is a suitable approach because it promotes transparency in the decision-making process. The probabilities obtained from the neural network can be treated as input attributes for the decision tree, where each probability can be considered a feature in the decision-making context. This allows decision-makers to understand how the probabilities contribute to the final decision. Specifically, the decision tree model presents each branch as a particular combination of probability values, making it clear which probability values are most influential in determining the final decision.

In this way, the decision tree can provide a more intuitive and interpretable framework for making decisions, especially when explaining the reasoning behind the decision to others [90]. By providing a clear path of decisions and actions, the decision tree allows more effective communication of the decision-making process, improving transparency and accountability.

Constructing a decision tree resumes the iterative process of splitting the data based on different features and feature thresholds until a stopping criterion is met, using a heuristic called recursive partitioning (commonly referred to as Divide and Conquer). A split is a subset of input data that contains multiple elements, each comprising multiple features and a label that defines them. Several algorithms will be used for this process to test the accuracy, versatility, and complexity of the tree.

At the end, each internal node of the decision tree has an attribute-based test and an outgoing branch for each possible outcome. Each leaf node will have an associated class, and in order to classify a record, one must be at the root node and successively visit internal nodes until a leaf node is reached. The outcome of the attribute-based test at an internal node determines the branch traversed and, thus, the next node visited. The class that classifies the record is simply the class of the final leaf node.

Once the decision tree has been built, it may be necessary to prune it to improve its accuracy [91]. This involves using a separate validation set to evaluate the performance of the tree branches on unseen data, as opposed to the training set. If a branch is ineffective, it can be pruned and removed from the tree. The decision tree is pruned by recursively replacing each subtree with its mean target value (in the case of regression) or the most common class (in the case of classification) until the accuracy of the pruned tree in the validation set is not significantly lower than that of the original tree. This approach helps create a more accurate decision tree by identifying and eliminating weak branches.[92]

The pseudocode algorithm 1 displays the recursive building process of a decision tree, where each group of data is split and analysed for each node until a stopping criterion is found. The recursive usage of this function during the algorithm generates the binary tree pattern.

Without proper constraints, decision trees can quickly overfit to the training data and fail to generalise to new data. One way to prevent this is to set stop criteria during the tree building process, as shown in algorithm 1. In this regard, various stopping criteria can be used to ensure that the tree does not grow too large and reduce overfitting. In this context, some of the commonly used stopping criteria in decision tree building are explored.

- Maximum tree depth: maximum length of a path from root to leaf. Trees generated are not side balanced, whereas one side of the tree will have more inspections than the other side of the tree;
- Minimum sample split: the minimum that is imposed to stop further splitting nodes;
- Minimum samples per leaf: reduced number of samples per leaf might mean overfitting. This can be converted to minimum sample split if verified after the splitting at a node;
- Minimum impurity decrease: when the splitting of a node does not show a cost-effective improvement over the original node;

- No label variety: when all the labels of a split have the same value.

Algorithm 1 Decision Tree building algorithm's pseudocode.

Require:

$D(X, y)$: feature matrix of size ($n_{samples}$, $n_{features}$), target vector of size ($n_{samples}$,)
max_{depth}: maximum depth of the tree (decrements)
min_{split}: minimum number of samples required to split a node
criterion: split decision criterion ('entropy' or 'gini')

```
procedure BuildTree
    Create a node N
    if D contains only one class then
        set N as a leaf node with class label equal to y
    else if max_depth is 0 or D has fewer than min_split samples then
        set N as a leaf node with class label equal to the majority class in D
    else
        for f in X do
            for unique value t of X[:, f] do
                Left, Right ←Split D[:, f] ≤t
                Compute the impurity of the split using the chosen criterion
                if Impurity of the split is better than the best split so far then
                    Update the best split to be the current split, with threshold t and feature f
                end if
            end for
        end for
        if If no valid split was found then
            Set N as a leaf node with class label equal to the majority class in D
        else
            Set N as an internal node with:
            - Left child ←BuildTree(Left, max_depth - 1, min_split, criterion)
            - Right child ←BuildTree(Right, max_depth - 1, min_split, criterion)
        end if
    end if
return node N
end procedure
```

Although decision trees are widely used for machine learning tasks, they can sometimes have lower accuracy than other methods and may need to be more balanced in certain instances. To overcome these issues, a popular technique is to use a set of decision trees, known as a random forest classifier [93, 94, 95].

Random forest classifiers are built using an ensemble of decision trees. The key idea behind random forests is to introduce randomness into the construction of individual trees to ensure that they are

independent and diverse from one another. This randomness is introduced by creating random subsets of the training data to construct each decision tree. Specifically, for each tree in the ensemble, a subset of the training data is randomly sampled with replacement from the original training set. This process is known as bootstrap aggregating or bagging.

After constructing all the individual decision trees, the final classification decision is made by aggregating the decisions of all the trees by taking a majority vote on the predictions made by each tree. The randomness introduced by the bootstrap aggregating and the feature selection process makes the decision trees independent of one another, leading to an ensemble of diverse and independent trees that are less likely to overfit the training data.

The number of trees in the random forest classifier also plays a vital role in determining its final performance. Increasing the number of trees in the ensemble generally leads to better test set accuracy. However, after a certain number of trees, the performance plateaus, and increasing the number of trees may yield slight improvement. The reason for this is that the addition of more trees to a random forest classifier increases computational complexity without necessarily leading to better performance.

Algorithm 2 shows the process described before. In here, D is the training set, T is the number of trees to build, F is the number of features to consider when finding the best split, and N is the number of samples to use when building each tree. RF is the resulting random forest model consisting of T decision trees.

Algorithm 2 Random Forest algorithm's pseudocode.

Require:

$D(X, y)$: feature matrix of size ($n_{samples}$, $n_{features}$), target vector of size ($n_{samples}$,)
T: Number of trees
F: Number of features
N: Number of samples

```
1:  procedure RandomForest
2:      RF ←[]
3:      for t = 1 to T do
4:          D_t ←Randomly sample N examples from D with replacement
5:          F_t ←Randomly select F features from D_t
6:          N_t ←BuildTree((D_t, F_t))
7:          Add N_t to RF
8:      end for
9:  return RF
10: end procedure
```

In each iteration of the loop, a new decision tree is built by randomly sampling N examples from the

training set D with replacement (the replacements mean that the values are removed from the original set in order to avoid repetition when building the following trees), and randomly selecting F features to consider at each split. The BuildTree function shown in the pseudocode 1 is then called with the subset of examples and features, and the resulting root node N_t of the decision tree is added to the random forest model RF. Finally, the function returns the completed random forest model.

This new approach can cause a significant decrease in the readability of the resulting trees, thus reducing general interpretability. The following algorithm shows a different approach for the classification problem at hand. Unlike the traditional random forest approach, the entire dataset will be used to construct each forest decision tree. Rather than creating subsets of the data, the labels are modified to only contain a single entry of data relevant to the current tree instead of a group of labels.

For instance, consider the training data with two unique labels: "cat" and "dog". These labels come in the format of [L1, L2], where each L value is a binary value (0 or 1). Instead of building a single tree or multiple trees on random subsets of data capable of identifying if an instance is a "cat" or a "dog", each decision tree will consider the same set of features to identify a single class. Thus, the final classification for the input will base itself on the output of this set's decision trees. This approach enables constructing a set of decision trees, each focused on a particular class, allowing for more efficient and accurate data classification.

As one can see in algorithm 3, there is no longer a need to sample a subset of data for each tree. Instead, each tree is constructed using the entire training set $D(X, y_l)$, where l represents each class to be classified. The resulting training set used for each tree contains a single label defining each instance, in contrast to the original set, which includes an array of labels, one for each class.

Algorithm 3 Modified Decision Forest algorithm's pseudocode.

Require:

$D(X, y)$: feature matrix of size ($n_{samples}$, $n_{features}$), target vector of size ($n_{samples}$,)

```
1:  procedure ModifiedDecisionForest
2:      uniqueLabels ←Unique values in D(y)
3:      DF ←[]
4:      for l in uniqueLabels do
5:          D_t ←D(X, y_l)
6:          N_t ←BuildTree(D_t)
7:          Add N_t to DF
8:      end for
9:  return DF
10: end procedure
```

The visual representation of decision trees, random forests, and modified decision forests can provide a clear understanding of the differences between these methods. Figure 18 shows a simple decision tree's structure, where each leaf node represents a possible outcome given the previously learned rules. The following example in figure 19 visually represents what a random forest would be. As one can see, an ensemble of decision trees is shown, where each tree is built on a random subset of data and features, and the final prediction is determined through a voting process based on the predictions of all the trees.

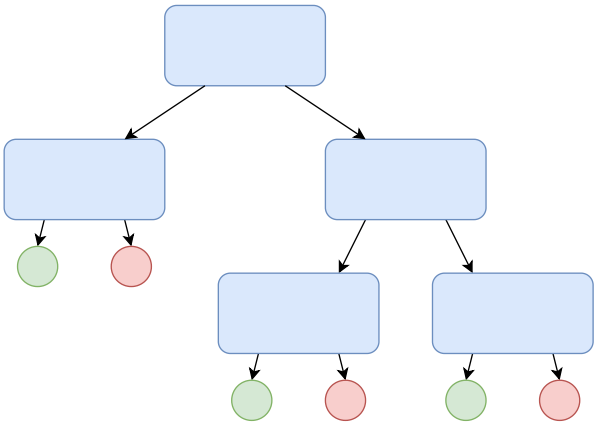


Figure 18: Decision Tree Structure Diagram.

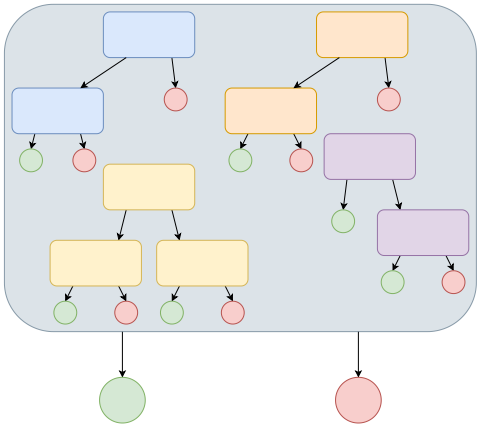


Figure 19: Random Forest Structure Diagram.

On the other hand, modified decision forests depicted in figure 20 are built by constructing decision trees on the entire dataset, with each tree focused on a particular class. The final prediction is based on the output of all decision trees, where each tree only considers a single data entry relevant to the current tree's class.

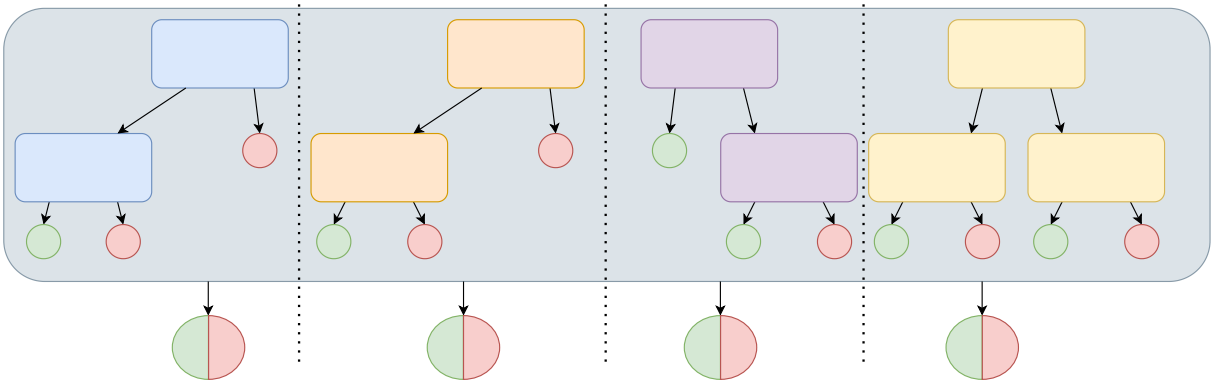


Figure 20: Decision Forest Structure Diagram.

As illustrated in the decision tree building algorithm outlined in Algorithm 1, a criterion is employed to determine the optimal split at each node. This criterion serves as a metric to identify the most suitable division that aligns with the characteristics of the current set and yields the most favourable outcomes. The subsequent algorithms provide descriptions of various criteria employed in this particular task.

ID3

The ID3 [88] algorithm is a popular decision tree algorithm used for solving classification problems. It is primarily employed to build decision trees from labeled training data, where the goal is to classify instances into predefined classes based on their feature values. By recursively selecting the most informative attributes to split the data, the ID3 algorithm aims to construct an efficient decision tree that can accurately classify unseen instances.

The ID3 algorithm consists of four steps. In the first step, the entropy of the original dataset is calculated. Entropy is a measure of the amount of uncertainty or randomness in the data. The formula to calculate the entropy is:

$$E(S) = -\sum_{i=1}^{n} p_i log_2(p_i) \tag{3.1}$$

where p_i is the proportion of instances in S that belong to class i.

In the second step, the information gained for each attribute is calculated. For each attribute A in the dataset, the information gain (IG) from using A to split the data is calculated. The formula for information gain is the following:

$$IG(S, A) = E(S) - \sum_{i=1}^{n} \frac{|S_v|}{|S|} E(S_v) \tag{3.2}$$

where S_v is the subset of S that has the value v for the attribute A.

The attribute A with the highest information gain is chosen as the splitting attribute. In the third step, the data is split based on the selected attribute. For example, if the splitting attribute is "age" and has three possible values young, middle-aged, old, then the data is partitioned into three subsets, one for each value.

In the fourth step, the ID3 algorithm is recursively applied to each subset until a stopping criterion is met. For example, the tree may stop growing when all instances in a subset belong to the same class, or when the tree reaches maximum depth.

C4.5

Quinlan [96] then developed the C4.5 algorithm, an extension of the ID3 algorithm, widely used for decision tree construction in classification tasks. Similar to ID3, C4.5 aims to build decision trees from labeled training data. However, C4.5 introduces additional features, such as handling missing attribute values and handling continuous attributes by discrediting them. It also uses a statistical measure called information gain ratio to determine the most informative attribute for splitting the data. This algorithm seeks to create decision trees that are more robust and accurate by considering various data complexities and incorporating statistical principles.

The Information Gain function often favours features with a larger number of categories, as they typically exhibit lower entropy. However, this tendency can lead to overfitting of the training data. To address this concern, Gain Ratio is introduced as a means of mitigating the issue. Gain Ratio penalises features with more categories by incorporating a concept known as Split Information or Intrinsic Information. The algorithm leverages this changes but still consist of four steps similar to those of the ID3 algorithm. In the first step, the entropy of the original dataset is calculated.

In the second step, the gain ratio for each attribute is calculated. The gain ratio is defined as the normalised information gain. The formula for the gain ratio is:

$$IGR(S, A) = \frac{IG(S, A)}{SplitInfo(S, A)} \tag{3.3}$$

where $SplitInfo(S, A)$ is a measure of the amount of information required to split S based on A, with its equation being:

$$SplitInfo(S, A) = \sum_{i=1}^{n} \frac{|S_v|}{|S|} log_2(\frac{|S_v|}{|S|}) \tag{3.4}$$

The attribute A with the highest gain ratio is chosen as the splitting attribute. In the third step, the data is split based on the selected attribute. The fourth step is the same as that of the ID3 algorithm, i.e., the C4.5 algorithm is recursively applied to each subset until a stopping criterion is met.

There are differences between the two algorithms in the metric used to split the data. Information gain will not favour any attributes by the number of distinct values, while the information gain ratio will favour attributes with fewer distinct values. When applied to attributes that can take on many distinct values, the information gain technique might learn the training set too well. On the other hand, the user will only be able to find attributes that require many distinct values. C4.5 can handle both continuous and discrete attributes, and attributes with differing costs, improving from it's predecessor, the ID3 algorithm. It also prunes trees after creation by going back through the tree once it has been created and attempting to remove branches that do not help by replacing them with leaf nodes. [97, 98]

CART

The CART (Classification and Regression Trees) [99] algorithm is a versatile decision tree algorithm commonly used for both classification and regression tasks. Unlike ID3 and C4.5, which focus primarily on classification, CART can handle both categorical and continuous features. CART constructs binary decision trees, where each internal node represents a feature test, and each leaf node corresponds to a predicted class or value. The algorithm iteratively searches for the best splits based on criteria such as Gini impurity or mean squared error to maximise the homogeneity of the resulting subsets. CART's flexibility

makes it suitable for a wide range of applications, providing accurate predictions and interpretable models for both classification and regression problems. It consists of three steps. The attribute and split point that minimises the impurity measure are chosen in the first step. For classification problems, the impurity measure used is typically the Gini impurity, which is defined as:

$$Gini(S) = 1 - \sum_{i=1}^{n} (p_i)^2 \tag{3.5}$$

where p_i is the proportion of instances in S that belong to class i.

In the second step, the data is split based on the selected attribute and split point. Instances with values less than or equal to the split point go to one subset, and instances with values greater than the split point go to the other subset. The split point is decided by calculating the Weighted Gini impurity for the two new nodes that will be generated and finding the minimum possible value for the data in question. In the third step, the CART algorithm is recursively applied to each subset of the data until a stopping criterion is met.

The Gini index and entropy are commonly used measures in data analysis and decision-making processes. Both metrics offer insights into the characteristics of a set or distribution. The Gini index quantifies the probability of misclassification, while entropy captures the level of uncertainty or randomness [98]. To better understand the similarities and differences between these measures, refer to the table 2 below:

Table 2: Gini and Entropy comparisson

Measure	Range	Interpretation	Sensitivity	Computational Complexity
Gini	[0, 1]	Probability of missclassification	Sensitive to class distribution	O(n)
Entropy	[0, log(n)]	Level of uncertainty or randomness	Sensitive to number of classes	O(n.log(n))

Given the nature of the binary data present in the labels, the primary metric employed to build decision trees is the information gained from the ID3 algorithm. This choice is driven by this metric's simplicity and suitability for binary data. Information gain measures the reduction in uncertainty achieved by selecting a specific attribute as the splitting criterion for a decision tree node. In the case of binary data, where the labels have only two possible values, the calculation of information gain becomes straightforward. By evaluating the information gained for various attributes, the ID3 algorithm identifies the attribute that maximises the reduction in uncertainty, enabling the construction of a practical decision tree for classification tasks. Additionally, the calculation of information gain ratio from the C4.5 algorithm will also be implemented and briefly evaluated.

It is now possible to fully define the Top-Level of the Cognitive System as a collection of classes

presented in the UML diagram in Figure 21. From here, this module will comprise three data classes, with the aim to better construct and define the nodes at the Decision Tree level. The Leaf Node and Decision Node will be core nodes of the Decision Tree structure while the Split Node will be used to evaluate the best possible split at a given point in the construction of the decision tree.

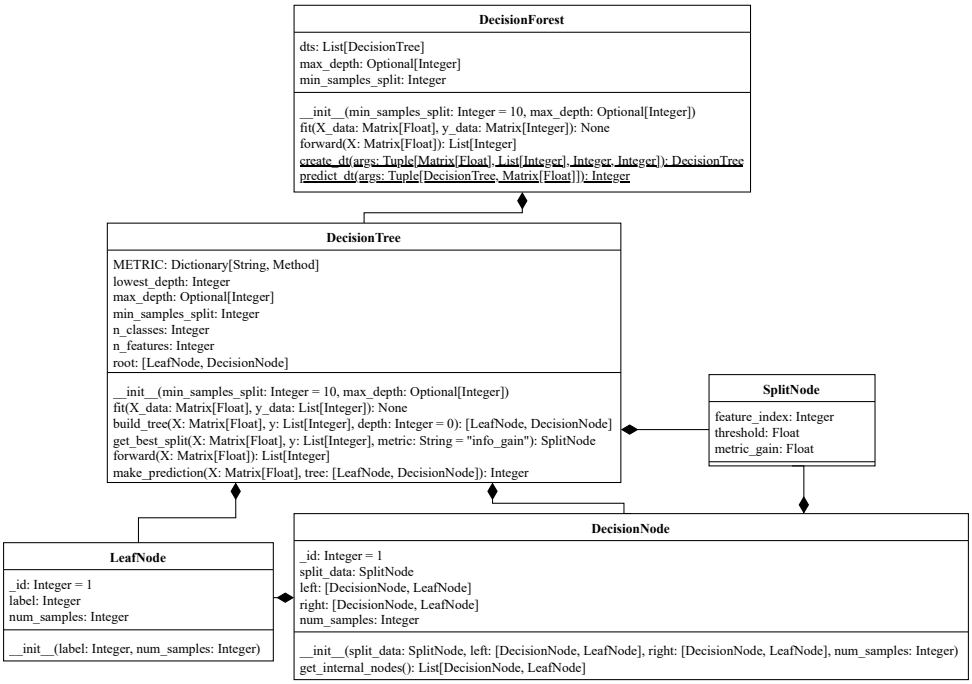


Figure 21: Designed Top Level's UML diagram.

The construction of the decision tree follows the algorithm presented in 1, incorporating predefined stopping conditions specified during the creation of the Decision Tree instance. The recursive "build_tree" function is invoked until these stopping conditions are satisfied, resulting in the generation of a complete decision tree for the given dataset, which is passed to the "fit" function. Similarly, the "forward" method utilises the prediction function to obtain labels for each data instance.

Notably, several parameters in the functions of the Decision Tree class are of type "Matrix". In this context, and since it refers to the data used to train or predict over, this Matrix will be a List of Lists with dimensions (n_instances, n_features) representing probability data for each instance that follows from the bottom level. The term "Matrix" is then used for clarity and display purposes.

These Decision Trees, as described earlier in algorithm 3, are independent and correspond to each class in the dataset. The "fit" function of the Decision Forest class employs parallel processing to optimise the construction of these decision trees. This approach maximises the utilisation of system resources and facilitates the simultaneous creation of multiple decision trees. Figure 22 visually illustrates the creation of these decision trees.

Similarly, the prediction process also makes use of the same parallel approach, where a process for each decision tree prediction is created and then compiled at the end.

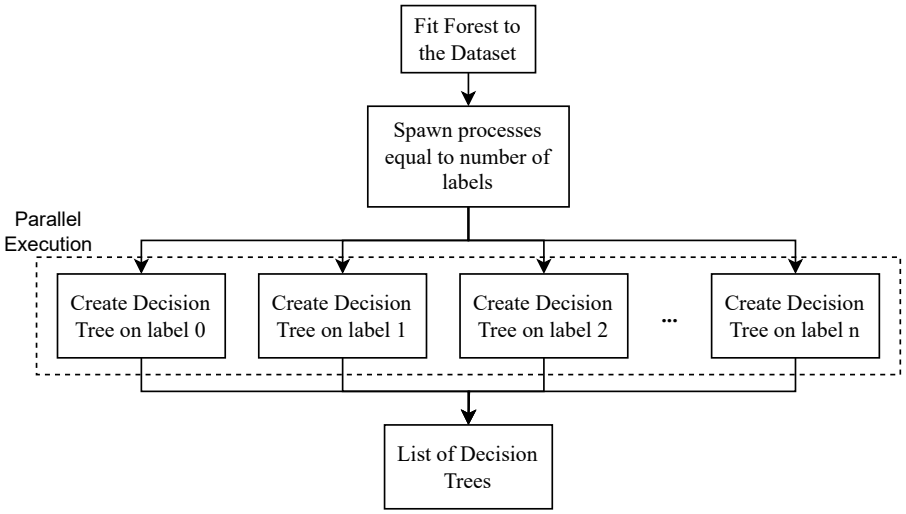


Figure 22: Decision Tree parallel creation.

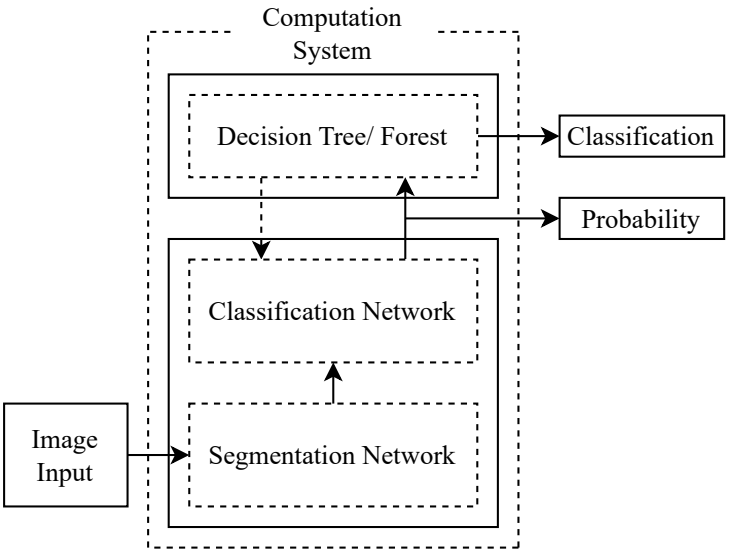


Figure 23: Computation System interconnections.

At this stage, the complete Computation System can be accurately defined as a combination of symbolic and sub-symbolic methods. Figure 23 depicts the sequence of transactions between the Classification Network and the induction method (either a single Decision Tree, Random Forest, or Decision Forest). As one can see, the connection between the bottom and top levels is unidirectional since the first gives information for the following to use. This process is known as "bottom-up" and will be the starting point of knowledge for the BorisCAD architecture.

In the context of the described Computation System, "bottom-up learning" refers to the process in

which knowledge is acquired and built upon from lower levels to higher levels. It involves extracting information and patterns from the input data or lower-level components and using them to inform the subsequent stages or higher-level components of the system.

This intermediate information transferred between both levels will also have an essential role in the decision-making process since it will be used from memory to influence the final decision of the architecture. This is further studied in the section 3.3 ahead.

At this juncture, the UML diagram representing the Computation System can be preseted, which serves as an integration point for both the Top and Bottom levels within the BorisCAD architecture. This diagram, displayed in Figure 24, illustrates the structural and behavioural relationships between the components involved. Within the Computation System, the class takes on the crucial responsibility of facilitating communication and coordination between the two levels. By ensuring a sequential application of the training methodology, it guarantees the seamless flow of information and processes between the Top and Bottom levels. Specifically, the forward function assumes a pivotal role in managing data traversal through the respective modules and facilitating memory updates as necessary.

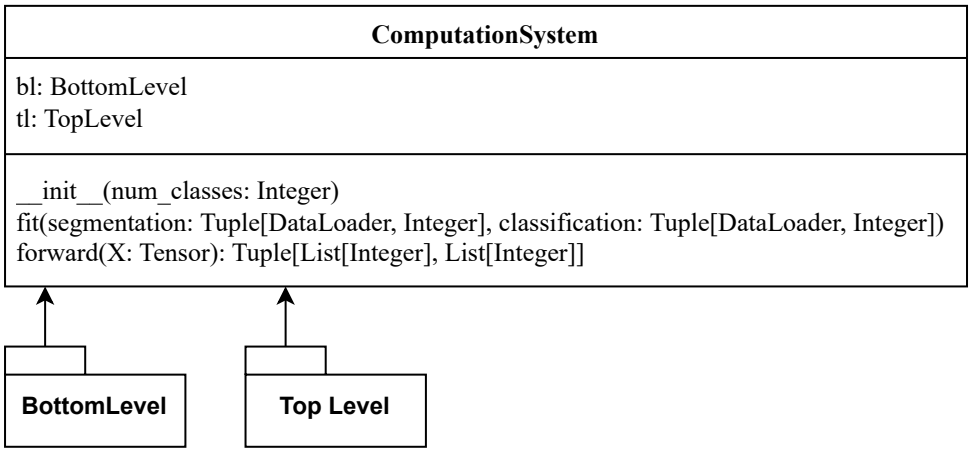


Figure 24: Computation System UML diagram.

3.3 Storage System: Working and Long-Term Memory

Our cognitive abilities include the remarkable feature of the human brain's ability to store and retrieve information. *Memory* is a complex process that involves various neural structures and mechanisms [100]. Long-term memory stores and retrieves information over an extended period and is essential for everyday activities such as remembering names, dates, and events. This type of memory is believed to be stored in different brain regions, including the hippocampus, amygdala, and prefrontal cortex [101]. In contrast,

short-term and working memory are more temporary forms of memory, necessary in many day-to-day activities.

Short-term memory allows us to hold a small amount of information in our minds for brief periods, essential for reading, listening, conversing, problem-solving, and decision-making. It is associated with the prefrontal cortex and other brain regions responsible for attention and executive control. However, it has a limited capacity and duration, with contents that are rapidly forgotten once attention is diverted.

Working memory is a more dynamic form that temporarily stores and manipulates information to complete tasks or solve problems. It is closely related to short-term memory but is more active and goal-directed. It is associated with several neural circuits and brain regions, including the dorsolateral prefrontal cortex, the parietal cortex, and the basal ganglia [102]. Working memory is crucial in cognitive tasks such as language comprehension, mathematical reasoning, and spatial processing.

In biological systems, the formation and retrieval of memories are intricate and multifaceted processes, characterised by complex interactions between various brain regions and molecular mechanisms, such as gene expression, protein synthesis, and synaptic plasticity [103]. However, when considering the memory model in BorisCAD cognitive architecture, certain biological complexities may be simplified or abstracted to enable a streamlined proof of concept. The BorisCAD architecture references only long-term and working memory, and, by only focusing on these two types of memory, it can efficiently store and manipulate the information required for Computer-Aided Diagnosis (CAD) tasks—long-term memory stores essential design elements, such as image features and their contents and past inference rules. Working memory manipulates these elements in real-time during the decision-making process.

Given this, the long-term and working memory components are designed to function as episodic memories, similar to the memory systems observed in cognitive architectures such as CLARION and SOAR. These episodic memories allow BorisCAD to store and retrieve detailed information about past events, experiences, and inference rules, enabling the architecture to make informed decisions based on similarities between episodes.

The architecture also enhances memory design by implementing a graph structure, which optimises information storage and manipulation [104]. This structure enables efficient representation and organisation of complex relationships between different elements. BorisCAD uses the graph structure to represent design elements such as parts, sub-assemblies, and assemblies as nodes, with their relationships represented as edges.

Each memory node represents an image perceived by the system's bottom level. The node comprises several elements that define the contents of a new piece of information at multiple abstraction layers.

These elements include high and low-level feature maps, which provide a visual or sensory representation of the image. Classification probabilities are also stored as numerical values, indicating the likelihood of an item belonging to a particular category or class. The induction class represents an almost final representation of a category or concept. At the same time, production rules refer to instructions the induction framework would take to achieve the final induction class. In this case, the production rules describe the branches the decision tree or forest takes to reach the final leaf node. Figure 25 better depicts which information will be stored, as well as the exact location from where it is fetched.

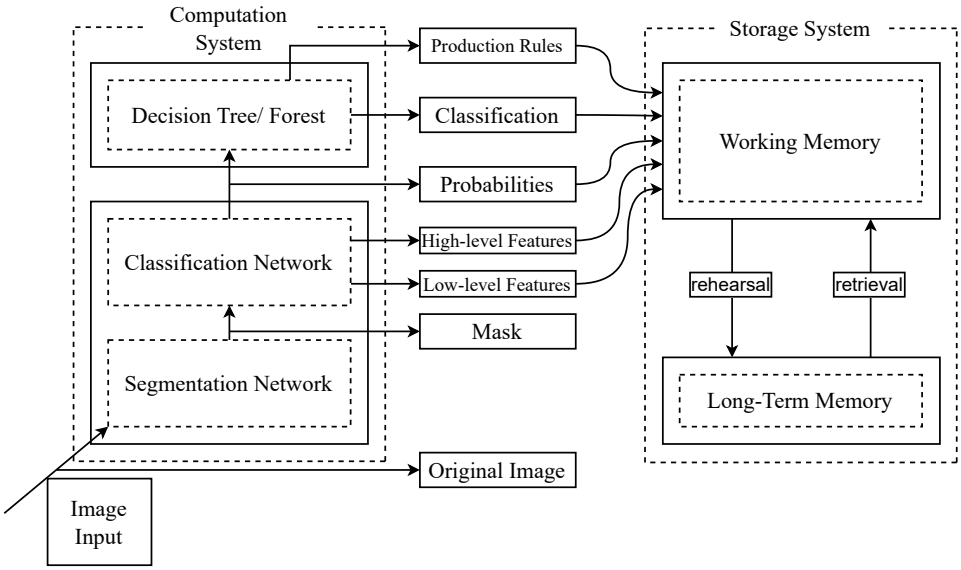


Figure 25: Storage System's Information Gather.

As described before, the Classification Network of the BorisCAD architecture will comprise ResNet blocks (figure 11). The segmented image is passed through the network until the output of the last block in the first group is obtained. This output is then considered as the low-level feature map. Similarly, the feature maps obtained after the last group of ResNet blocks can be considered high-level features. This way, a hierarchy of features can be obtained from the input image, with low-level features capturing basic patterns and high-level features capturing more complex patterns.

Furthermore, production rules can be gathered from the inference through the decision tree. Production rules are used in rule-based systems to represent knowledge as if-then statements. In a decision tree, each node represents a condition on a specific feature. Thus, a path from the root to a leaf can be defined as a set of production rules that collectively specify the conditions that must be satisfied to reach a final decision.

Moreover, the transfer of information between working memory and long-term memory actively involves encoding and retrieval. During the encoding phase, working memory processes the information

before transferring it to long-term memory for storage. This vital transfer occurs through rehearsal, which includes repeating the information within working memory. By utilising rehearsal, whether through maintenance rehearsal, which reiterates the information in its original form, or elaborative rehearsal, which establishes connections to existing knowledge, the likelihood of successfully transferring the information to long-term memory increases [105]. Conversely, retrieval, also known as recognition, actively retrieves stored information from long-term memory back to working memory. Recognition involves identifying previously learned information as familiar without necessarily recalling specific details.

A deliberate design choice aimed at streamlining the interaction and updates between the contents of working memory and long-term memory was made by upholding a shared reference to a node upon its inclusion. This approach guarantees that the same item is not redundantly added to memory at various stages, mainly when minor changes occur. The system sidesteps duplication and optimises memory management by consistently retaining both memories' nodes.

With this, it was found that the best method for retrieving information from long-term memory would be applying a mix of *Context-Based Retrieval* and *Priority Retrieval*. The current image will compare using the feature maps retrieved from the bottom level with other nodes in the long-term memory. The working memory will be populated depending on the similarity with other items in the long-term memory. With this, the items retrieved will also be updated with the notion of "hot", where a more accessed item will have more influence in the reasoning process of the memory module.

This retrieval will also be based on said "hot" index. A *threshold* is defined in order to avoid low-accessed items to be transferred to the working memory. This allows the memory to filter information based on how relevant an item is, and implementing a method similar to the concept of **remembering**.

Figure 26 depicts the overarching flow of the node creation and insertion process. A corresponding memory unit is generated upon receiving a new node, such as an object or image. Subsequently, this memory unit is initially inserted into the long-term memory, facilitating the achievement of recognition, as previously elucidated. This process manifests as a similarity assessment between the new and existing nodes within the memory, thereby engendering a dynamic graph structure composed of interconnected nodes.

In order to enhance the decision-making process, it becomes imperative to populate the working memory with akin items. These items are influential factors characterised by their similarity to the current node. Ultimately, the most recent node is appended to the conclusion of the memory, ensuring its inclusion in the ongoing cognitive processes. The number of similar nodes to be added to the working memory is

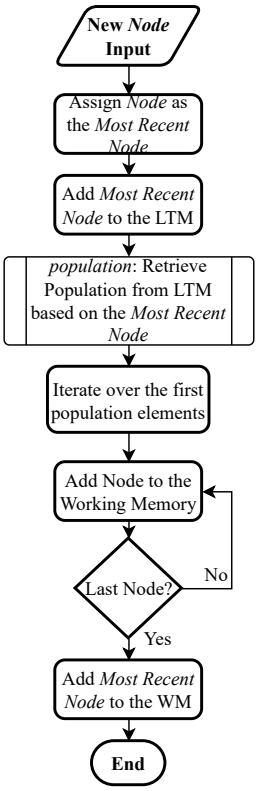


Figure 26: Storage System's new node creation flowchart.

contingent upon the capacity of the working memory, as it directly influences the amount of information that can be accommodated within the system.

The addition of a new node to the long-term memory is illustrated in Figure 27. The insertion process adheres to a systematic procedure within the framework of a dictionary structure. Upon receiving a new node, a comparison is conducted between the new node and the existing items in memory to identify similarities. If a similarity is detected, the procedure establishes a novel edge between the new node and the current nodes in memory, contingent upon meeting a predefined threshold. Subsequently, the node is inserted into the memory using its unique identifier.

The decision to employ a dictionary structure for long-term memory is underpinned by the imperative of scalability. This choice ensures the effective organisation and retrieval of information, facilitating the seamless management of burgeoning data volumes while maintaining operational efficiency.

Inserting nodes into the working memory adheres to a predefined flowchart, as depicted in Figure 28. The working memory, characterised by its finite capacity, constrains the number of nodes it can accommodate simultaneously. Upon receiving a new node for insertion, the memory checks for node presence, and if the node is already in memory, it is removed from its current position and appended to the memory's end. This strategy ensures that the most similar nodes remain at the end of the memory, even when they are already at a different position in memory.

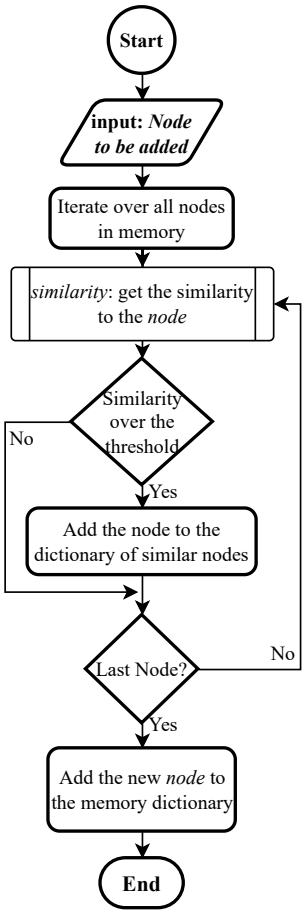


Figure 27: Long-term memory new node flowchart.

Besides, there is a memory capacity assessment; if the memory is full, necessitating space allocation for the new node, the oldest node in the memory is deleted, making room for the new node, placed at the end of the memory. In cases where the memory has available space, the new node is directly inserted at the end of the memory without any removal. This ensures that the working memory maintains the chronological sequence of node entries.

When creating a new memory, the working memory undergoes a subsequent process of population, as previously explained. Figure 29 illustrates the retrieval of existing memories from the long-term memory for insertion into the working memory. The system identifies the connected nodes of the current node as relevant contributors to the decision-making process. Moreover, only the nodes surpassing the remembering threshold are selected to populate the working memory.

Additionally, it is essential to highlight that the system actively updates the hotness of the nodes, even those that are not directly utilised but maintain connections with other nodes. This proactive update ensures that the information and relevance of the connected nodes are kept up-to-date and accounted for in subsequent cognitive processes.

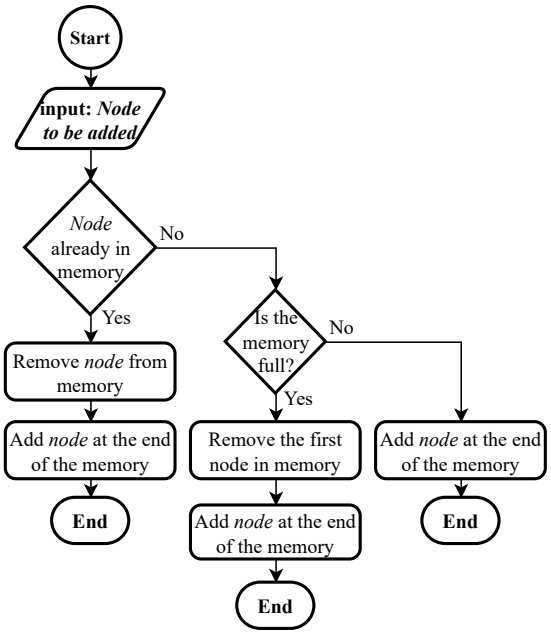


Figure 28: Working memory new node flowchart.

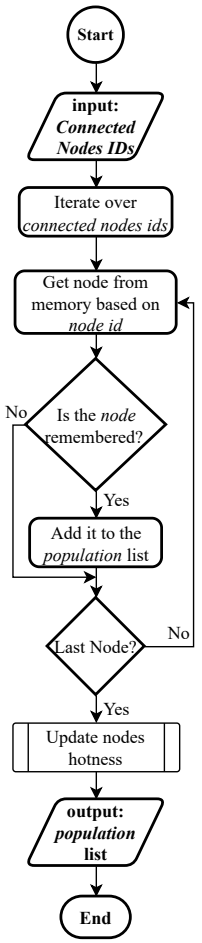


Figure 29: Node retrieval from long-term memory flowchart.

In order to ensure the desired behaviour, the procedure ”UpdateHotness” presented in pseudocode

algorithm 4 was explicitly devised. This procedure incorporates a series of steps to modify the hotness values of nodes within the system appropriately.

Algorithm 4 Long-term memory hotness update pseudocode.

Require:

$connected\_nodes$: list of nodes connected to the current node in use

```
1:  procedure UpdateHotness
2:      for node in memory do
3:          if node not in connected_nodes and not the current node then
4:              node hotness - 1
5:          end if
6:          if node not the current node then
7:              node hotness + 1
8:          end if
9:      end for
10: end procedure
```

Firstly, the algorithm verifies whether the currently iterated node is neither one of the connected nodes to the current node nor the current node itself. This verification serves as a criterion for identifying nodes that require a decrease in relevance. Upon satisfying this condition, the algorithm decrements the hotness value of the node by 1, indicating a reduction in its significance within the system. Furthermore, if the node is not the current node, it means that it is a member of the group of connected nodes, implying that the node's hotness index should be incremented.

By implementing these increments and decrements, the algorithm emphasizes nodes directly connected to the current node while simultaneously downplaying the relevance of indirectly connected nodes.

Finally, the subsystem can be visually represented using UML design, as depicted in Figure 30. The UML diagram showcases the relationships between the different types of memory, namely WorkingMemory and LongTermMemory, and the main class, StorageSystem, which serves as the interface for accessing and managing the memory.

The UML diagram illustrates the connections between these components, providing a clear overview of the system's structure. Additionally, the diagram highlights utilising an auxiliary class called Data within the MemoryNode class. This auxiliary class stores relevant information associated with each memory node. The diagram also demonstrates the connections between the MemoryNode class and the other three components: WorkingMemory, LongTermMemory, and StorageSystem.

As evident from the current discussion, the specific method pertaining to the influence of the working

memory on the decision-making process has not been expounded upon thus far. However, it is imperative to note that a comprehensive elucidation of this method will be provided subsequently, once the entire process is thoroughly described.

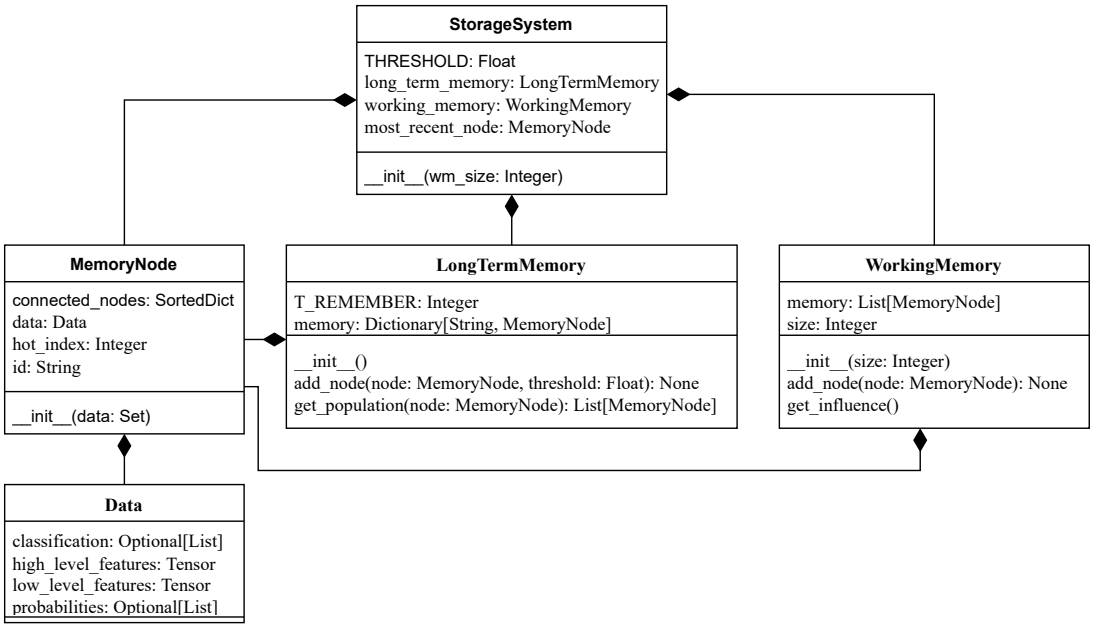


Figure 30: Storage System classes UML.

3.4 Learning System: Optimisation and Adaptation

The Learning System of the BorisCAD architecture is directly connected to the Bottom-Level Computation System, where neural networks reside, and optimisation plays a crucial role in training. The goal of optimisation is to find the parameters that minimise the cost function of a neural network. However, finding the global minimum of the cost function is challenging, especially for high-dimensional problems. Therefore, various optimisation algorithms have been developed to find the optimal parameters of a neural network efficiently.

Two popular optimisation algorithms used in deep learning are Stochastic Gradient Descent (SGD) and Adam. SGD is a simple but effective optimisation algorithm that updates the neural network's parameters in the direction of the negative gradient of the cost function. On the other hand, Adam is an adaptive optimisation algorithm that uses both the first and second moments of the gradient to adjust the learning rate. These algorithms have proven effective in a wide range of deep-learning applications. However, selecting the suitable optimisation algorithm for a particular task can be challenging, as it depends on several factors, such as the dataset's size, the model's complexity, and the available computational resources.

In addition to choosing an appropriate optimisation algorithm, another critical aspect of training a neural network is tuning the learning rate [106]. The learning rate is a hyperparameter that controls the step size at which the model's weights are updated during training. It is one of the most essential parameters to tune when training a neural network because it can significantly affect the model's performance. A high learning rate can cause the model to overshoot the minimum of the loss function, leading to oscillations or even divergence. A low learning rate can cause the model to converge slowly or get stuck in local minima.

The learning rate is usually indicated by the symbol α and is typically set to a small value between 0.0 and 1.0. Several techniques for adjusting the learning rate during training include fixed learning rates, learning rate schedules, and adaptive learning rate methods such as Adagrad, Adadelta, RMSprop, Adam, and others [107].

Fixed learning rates keep the learning rate constant throughout training, whereas learning rate schedules gradually reduce the learning rate over time. Adaptive learning rate methods adjust the learning rate based on the gradient of the loss function, which can help speed up convergence and prevent oscillations.

The *StepLR* technique involves decreasing the learning rate by a factor every step_size epochs. Specifically, the learning rate is multiplied by gamma every step_size epochs. The mathematical formula for the *StepLR* update rule can be expressed as:

$$\alpha = \alpha * gamma^{(epoch//step\_size)} \tag{3.6}$$

where α is the current learning rate, $gamma$ is the factor that decreases the learning rate, $epoch$ is the current epoch, and $step\_size$ is the number of epochs after which the learning rate is decreased.

The *ReduceLROnPlateau* method is another popular learning rate scheduling technique used in neural network training. Its main objective is to adjust the learning rate when it plateaus during training to help the network converge to a better optimum.

The method monitors a metric, such as the validation loss, after every epoch. If the metric does not improve for a specified number of epochs, the learning rate is reduced by a specific factor. This process is repeated until the learning rate is below a specified threshold. The user can set the factor by which the learning rate is reduced and the number of epochs after which the metric is checked. The algorithm for reducing the learning rate using *ReduceLROnPlateau* can be written as shown in 5.

These methods substantially increase the training efficiency of the neural networks and will be used both on the Classification and Segmentation modules described before. However, it requires careful tuning

Algorithm 5 ReduceLROnPlateau algorithm's pseudocode.

1: **if** the metric does not improve after 'patience' epochs **then**
2: $\alpha = \alpha * factor$
3: **end if**

of the parameters to ensure that the learning rate is reduced at the appropriate time and adapted to the model architecture and given problem.

Aside from hyperparameter tuning, further optimisation can be included. SGD stands for Stochastic Gradient Descent [108], one of the most widely used optimisation algorithms for training deep neural networks. It is a variant of gradient descent, which is a method of optimising the weights of a neural network in order to minimise its loss function.

In SGD, instead of computing the gradient of the entire dataset, the gradient of a randomly selected subset or "batch" of the dataset is computed, which is why it is called "stochastic". Using batches, this method can work with large datasets and update weights in each iteration, leading to faster convergence and better generalisation. The update rule is as follows:

$$w_{i+1} = w_i - \alpha \nabla L(w_i, x_{i:i+n}, y_{i:i+n}) \tag{3.7}$$

where w_i is the current weights, α is the learning rate, n is the batch size, $x_{i:i+n}$ is the input data subset, $y_{i:i+n}$ is the corresponding output data subset, and $\nabla L(w_i, x_{i:i+n}, y_{i:i+n})$ is the gradient of the loss function concerning the weights for the subset.

SGD has several variants, such as Nesterov accelerated gradient (NAG), which incorporate momentum terms to prevent oscillations in the optimisation process and accelerate convergence [109].

Adam is an acronym for "Adaptive Moment Estimation". It is a popular optimisation algorithm for training neural networks that uses adaptive learning rates and momentum to achieve better convergence during training. It is an extension of the Stochastic Gradient Descent (SGD) algorithm, and like the previous, it updates the network parameters based on the gradient of the loss function concerning the parameters.

Adam calculates the first and second moments of the gradients and uses them to update the learning rate for each parameter. The first moment is the mean of the gradients, and the second moment is the variance of the gradients. Adam also includes a bias-correction term to account for the first and second moments being initially biased towards zero.

The learning rate for each parameter is adapted based on the ratio of the first and second moments. This adaptive learning rate helps Adam to converge faster and more reliably than SGD on many deep learning problems. Adam also includes a momentum term that helps smooth the update process and

avoid getting stuck in local minima. The update rule for Adam can be expressed as:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{3.8}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \hat{v}t = \frac{v_t}{1 - \beta_2^t} \tag{3.9}$$

$$\theta_t = \theta t - 1 - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{3.10}$$

where θ are the model parameters, g_t is the gradient at time step t, m_t and v_t are the first and second moments of the gradient, $\hat{m}_t$ and $\hat{v}_t$ are the bias-corrected estimates of the first and second moments, β_1 and β_2 are the exponential decay rates for the first and second moments and are typically set to 0.9 and 0.999, respectively. α is the learning rate, and ϵ is a small constant to prevent division by zero, usually on the order of 10^{-8}.

Metrics play a crucial role in system performance evaluation, providing quantitative measures to assess effectiveness, efficiency, and overall quality. These standardised benchmarks enable objective comparisons between system implementations or variations. In classification, confusion matrices are commonly employed to compute these performance metrics.

A confusion matrix serves as a concise summary of a classification model's performance. It presents the counts of true positives, true negatives, false positives, and false negatives for each class. By examining the confusion matrix, evaluators can gain valuable insights into the model's performance and identify classes that pose challenges regarding prediction accuracy. Various performance metrics, such as accuracy, precision, recall, and F1-score, can be derived from the confusion matrix.

In binary classification, the positive class refers to the specific class of interest, while the negative class encompasses all other classes. However, defining positive and negative classes in multi-class classification becomes more intricate. One approach is to employ the one-vs-all classification strategy, where each class is treated as the positive class in separate evaluations against all other classes combined.

For example, consider a multi-class classification problem where images are being classified into three classes: A, B, and C. On a test set of 100 images, the classifier makes the following predictions:

- 30 images are predicted to be in class A, of which 25 are actually in class A, 3 are in class B, and 2 are in class C.
- 50 images are predicted to be in class B, of which 45 are actually in class B, 4 are in class A, and 1 is in class C.

- 20 images are predicted to be in class C, of which 18 are actually in class C, 1 is in class A, and 1 is in class B.

The information provided can be utilised to construct the following confusion matrix, as depicted in Table 3. This matrix captures multiple data points, which will subsequently be employed in the computation of classification metrics.

Table 3: Confusion Matrix example

	A	**B**	**C**
A	25	4	1
B	3	45	1
C	2	1	18

- **True positives (TP)**: The number of instances that were correctly classified as positive for each class. In this example, the TP for class A would be 25, the TP for class B would be 45, and the TP for class C would be 18.
- **True negatives (TN)**: The number of instances that were correctly classified as negative for each class. In this example, the TN for class A would be 70 (the sum of all the instances that were not A), the TN for class B would be 55 (the sum of all the instances that were not B), and the TN for class C would be 81 (the sum of all the instances that were not C).
- **False positives (FP)**: The number of instances that were incorrectly classified as positive for each class. In this example, the FP for class A would be 5 (the sum of all the instances that were not A but were classified as A), the FP for class B would be 8 (the sum of all the instances that were not B but were classified as B), and the FP for class C would be 3 (the sum of all the instances that were not C but were classified as C).
- **False negatives (FN)**: The number of instances that were incorrectly classified as negative for each class. In this example, the FN for class A would be 5 (the sum of all the instances that were A but were not classified as A), the FN for class B would be 5 (the sum of all the instances that were B but were not classified as B), and the FN for class C would be 2 (the sum of all the instances that were C but were not classified as C).

Note that in a multi-class problem, the sum of TP, TN, FP, and FN for all classes may not add up to the total number of instances, because some instances may be counted more than once. From this matrix, the following evaluation metrics can be calculated.

Accuracy

Accuracy quantifies the ratio of correct predictions to the total number of predictions, serving as a prevalent metric for classification model evaluation. However, it can be misleading in scenarios involving imbalanced classes or multiple classes, as a model focusing solely on the majority class may yield high accuracy while under performing for other classes.

$$\frac{(TP + TN)}{(TP + TN + FP + FN)} \tag{3.11}$$

Precision

Precision is a vital performance metric that assesses the proportion of true positives concerning all predicted positive cases. It quantifies the model's capability to identify instances of positive cases accurately. Precision is computed by dividing the number of true positives by the sum of true and false positives. A high precision value indicates that the model exhibits minimal false positive errors.

$$\frac{TP}{(TP + FP)} \tag{3.12}$$

Recall

Recall, referred to as sensitivity or true positive rate, signifies the ratio of true positives to all positive cases. It quantifies the model's efficacy in correctly identifying and capturing all instances of positive cases. The calculation involves dividing the number of true positives by the sum of true positives and false negatives. A high recall value indicates that the model exhibits minimal false harmful errors.

$$\frac{TP}{(TP + FN)} \tag{3.13}$$

F1-score

The F1-score is valuable when precision and recall are equally important, providing a balanced evaluation by considering both metrics. It is calculated as the harmonic mean of precision and recall, ensuring equal weighting of the two measures. This characteristic is beneficial in scenarios where false positives and negatives hold significant implications. For instance, in medical testing, false positives (which may result in unnecessary treatments or procedures) and false negatives (which can lead to missed diagnoses and delayed treatments) can have severe consequences. The F1-score ranges from 0

to 1, with higher values indicating superior performance in terms of precision and recall.

$$2 * \frac{Precision * Recall}{Precision + Recall} \tag{3.14}$$

Matthews Correlation Coefficient

The Matthews Correlation Coefficient (MCC) is extensively utilised as a metric to evaluate the effectiveness of binary classification models. It takes into account the number of true positives, true negatives, false positives, and false negatives, and provides a measure of the correlation between the predicted and actual binary labels. The MCC score ranges from -1 to 1, where a score of 1 indicates a perfect correlation, 0 denotes no correlation, and -1 represents a perfect inverse correlation. MCC is especially beneficial in situations where there is class imbalance or varying costs associated with false positives and false negatives.

$$\frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{3.15}$$

These performance metrics play a crucial role in evaluating and optimising the bottom-level classification process. By leveraging these metrics, the effectiveness of the learning algorithms can be measured and further improved. However, for the segmentation model, a distinct set of metrics is required, focusing on capturing the accuracy and overlap between the predicted mask and the ground truth.

Dice Score

The Dice score, also known as the F1 Dice score or Sørensen–Dice coefficient, is a widely used metric in image segmentation tasks. It quantifies the similarity between the predicted segmentation mask and the ground truth mask. Ranging from 0 to 1, a Dice score of 1 indicates perfect overlap between the predicted and ground truth masks, while a score of 0 signifies no overlap.

The Dice score is calculated using the provided formula, where "predicted" and "ground truth" refers to binary matrices representing the predicted and ground truth masks, respectively. The intersection (∩) denotes element-wise multiplication, and the union (+) denotes element-wise addition. The Dice score can be computed separately for each class or all classes combined.

$$\text{Dice Score} = 2 * \frac{\text{prediction} \cap \text{ground truth}}{\text{prediction} + \text{ground truth}} \tag{3.16}$$

IoU

Intersection over Union (IoU) is a commonly used metric in image segmentation tasks, particularly in object detection scenarios. It quantifies the overlap between the predicted bounding box (or mask) and the ground truth bounding box (or mask) of an object. The IoU calculation involves determining the ratio of the intersection to the union of the bounding boxes (or masks), where a value of 1 indicates a perfect overlap and 0 indicates no overlap.

$$IoU = \frac{Area\ of\ Overlap}{Area\ of\ Union} \tag{3.17}$$

In conclusion, the learning system incorporates optimisation algorithms to enhance performance and plays a crucial role in implementing metrics for the analysis of both classification and segmentation tasks. These metrics serve as quantitative measures to evaluate the system's effectiveness, efficiency, and quality. In classification, accuracy, precision, recall, and F1-score are utilised to assess the model's predictive capabilities. On the other hand, in segmentation, metrics like Dice score and Intersection over Union (IoU) are employed to measure the similarity between predicted and ground truth masks or bounding boxes. These metrics provide valuable insights into the system's performance, enabling stakeholders to make informed decisions and refine the learning process. By integrating optimisation algorithms and metrics, the learning system aims to continually improve its predictive and segmentation accuracy.

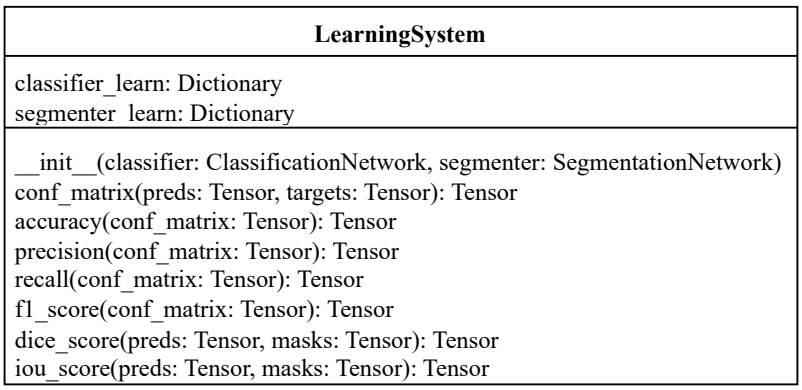


Figure 31: Learning System class UML.

The UML diagram in figure 31 illustrates the Learning System class. This class encompasses two essential dictionary variables designed to store the optimisation algorithms associated with the classifier and segmenter components. Additionally, the Learning System class features implemented metrics as individual methods. These methods serve as a means to evaluate and measure the performance of the classification and segmentation processes.

The complete design of the BorisCAD architecture, encompassing all its subsystems, namely the Computation System, Storage System, and Learning System, can now be comprehended. To provide an overview of the architecture's structure, the UML diagram depicted in Figure 32 is presented. This UML diagram encapsulates various auxiliary functions, aiding in the management of trained architectures, such as saving and loading. Additionally, it includes essential auxiliary functions, such as the "set_num_classes" method. This method allows for the modification of the number of classifiable classes in a pre-trained model, facilitating the application of transfer learning techniques. Moreover, this interface serves as the central hub for processing tasks related to the decision-making process, which will be elaborated upon in Section 3.5 of this dissertation.

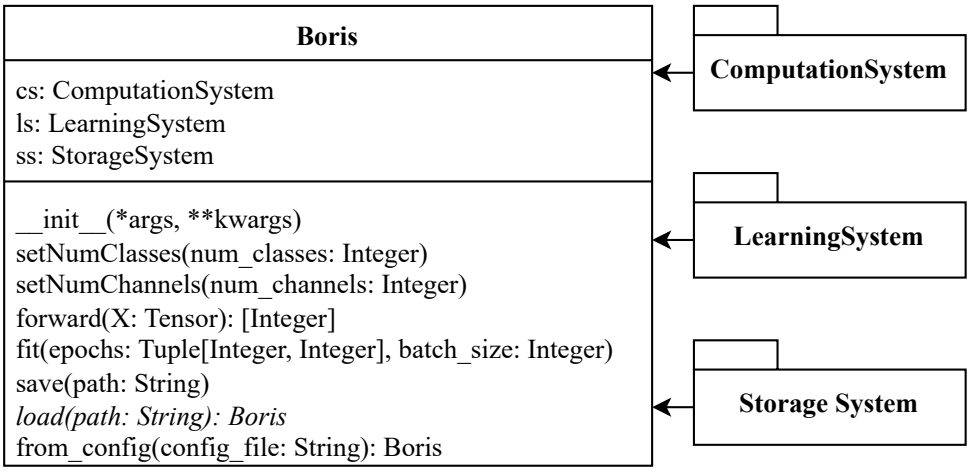


Figure 32: Boris class UML.

The decision to adopt the approach of creating an instance from a configuration stems from the necessity to thoroughly evaluate various parameters within the system as part of a proof-of-concept. By utilising a straightforward and adaptable configuration file that can be loaded during run time, the complexity associated with assigning values to variables during the instance creation process is significantly reduced. This approach allows for a more streamlined and manageable means of initialising the architecture instance. An illustrative example of such a configuration file is presented below, demonstrating the definition the parameters divided into subsections, similar to what the final archive will have. Here is an overview of the configuration structure:

- **Boris**:
  - **segmentation**: Specifies the directory path for the segmentation dataset.
  - **classification**: Specifies the directory path for the classification dataset.
  - **size**: Specifies the size of the input images, represented as [width, height].

- **num_classes**: Specifies the number of classes for the classification task.
- **Computation**:
  - **Bottom**:
    - **attention**: Determines whether attention mechanisms are enabled during segmentation.
    - **net_type**: Specifies the type of network architecture used in the bottom-level computation.
    - **use_segmentation**: Specifies whether the classification network incorporates segmentation as pre-processor for its data.
  - **Top**:
    - **min_samples_split**: Specifies the minimum number of samples required to split a node in the decision tree in the top-level.
    - **max_depth**: Specifies the maximum depth of the decision tree.
- **Storage**:
  - **wm_size**: Specifies the size of the working memory.

3.5 Holistic Approach to Decision-Making

A cognitive architecture necessitates a holistic approach to decision-making to capture the intricacies and complexities involved in processing and interpreting information. Medical image analysis, such as X-ray diagnosis, requires a comprehensive understanding beyond individual components or isolated metrics. By taking into account various levels of the architecture, such as the Bottom-level, Top-level, and Working Memory, a holistic approach enables the integration of multiple sources of information, each contributing unique insights. The Bottom-level provides detailed metrics specific to each class, allowing the architecture to weigh their influence accurately. The Top-level, considering the structure of the decision forest, captures the generalisation and entropy characteristics, influencing the overall decision process. The Working Memory plays a crucial role in leveraging past knowledge and integrating it with the current input, enriching the decision-making process with contextual information. By adopting a holistic approach, the cognitive architecture can account for the complexities inherent in medical image analysis, ensuring a more robust and reliable decision-making process that leverages the collective knowledge and information available throughout the architecture.

The diagram provided in Figure 33 offers a comprehensive depiction of the cognitive architecture,

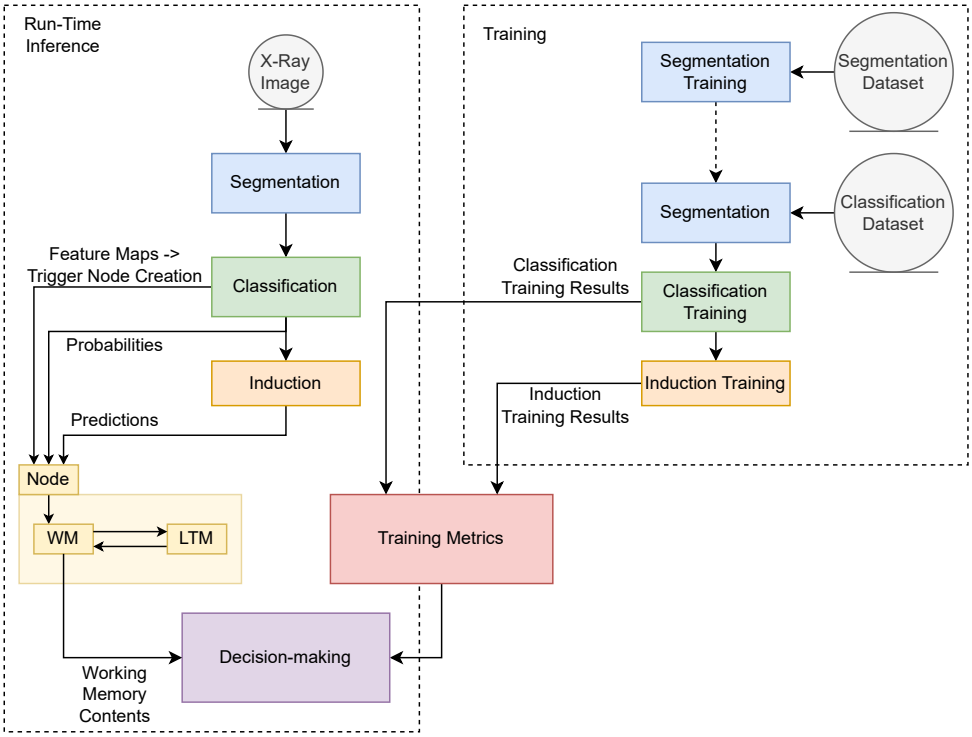


Figure 33: Decision-Making Diagram.

encompassing training and real-time inference processes. During the training phase in BorisCAD, a meticulous "bottom-up" approach is adopted where the segmentation network is first trained on an independent dataset, enabling the system to differentiate the foreground from the background in x-ray images accurately. This initial training step proves instrumental as the segmented output is a preprocessing step for the subsequent classification model, trained on a different dataset. Significantly, even during the training phase, this segmentation network is utilised as a preprocessing step for the classification model, ensuring consistent and cohesive training.

Furthermore, the same dataset is employed to generate probabilities for the Induction training. Leveraging the power of a Decision Forest consisting of an ensemble of Decision Trees, the architecture extracts valuable insights from the data, culminating in the formulation of meaningful probabilities for each class. The metrics garnered during the induction and classification training process are recorded as essential references for the decision-making process.

During the real-time inference of newly received x-ray images, the Storage System, which remained inactive throughout the training process, becomes an integral component of the architecture. Each incoming image undergoes a segmentation process, followed by classification using the trained model. This two-step process facilitates the generation of probabilities but also triggers the extraction of feature maps. As a result, a new memory node is dynamically created, seamlessly integrating with both the Working Memory and Long-Term Memory.

The decision-making mechanism encompasses a comprehensive evaluation of all components within the architecture. The Computation System influences the final decision by combining the probabilities and predictions with the pertinent metrics obtained during training. Moreover, employing a similarity-based reasoning approach similar to the CLARION cognitive architecture [110] via the Working Memory, the architecture judiciously considers the historical information stored in the memory nodes. By embracing this holistic approach, the cognitive architecture maximises its potential to provide accurate and robust predictions by considering procedural, semantic, and episodic components within the decision-making process.

The influence of the probabilities from the classification model on the decision-making process can be determined by multiplying the probabilities by the corresponding metric value. Denoting the probabilities as P_c for class c and the metric value as M_c for class c. The influence (I_c) of the probabilities from the classification can be calculated as:

$$I_c = P_c * M_c \tag{3.18}$$

From here, a list of weighted probabilities is created, depending on how the class defining each probability is accurately evaluated within the model. Similarly, the influence of predictions from the induction method can be computed by multiplying the predictions by the corresponding metric value. Defining the predictions as $Pred_c$ for class c and the metric value as M_c for class c. The influence (I_c) of the predictions from the induction can be calculated as:

$$I_c = Pred_c * M_c \tag{3.19}$$

For the Working Memory influence, a similar process can be achieved, this time by using the similarity index calculated during the node insertion in the Storage System. Denoting the final classification decision from previous nodes as $D_i = [D_1, D_2, ..., D_f]$ with f equal to the total number of existing classes and D_i representing the decision values for node i in memory. The similarity score between the current node and memory node i is S_i.

To calculate the influenced decision by the similarity weights and normalise these values, each decision value is multiplied by the corresponding similarity scores, summed in the class dimension and then divided by $(n-1)$, where n is the total number of nodes in memory. The influenced decision (ID) can be calculated as:

$$ID = \frac{\sum_{i=1}^{n} D_i * S_i}{n-1} \tag{3.20}$$

To incorporate the influence of the various methods, an equation is employed to determine the final

decision. The equation incorporates the weighted contributions of classification probabilities, induction predictions, and the influence of working memory nodes that exhibit similarity to the current node. The equation for the final decision is as follows:

$$FinalDecision = \alpha * I_{class}[P] + \beta * I_{ind}[Pred] + \omega * ID \tag{3.21}$$

In this equation, each component is multiplied by a corresponding weighting factor to emphasise its relative importance. The term $I_{class}[P]$ represents the weighted list of classification probabilities, reflecting the confidence associated with each class. Similarly, $I_{ind}[Pred]$ denotes the weighted list of induction predictions, which accounts for the predictions made by the induction model. Lastly, ID represents the weighted final decision of working memory nodes that exhibit similarity to the current node. The alpha factor determines the weight of the classification probabilities, the beta controls the influence of the induction predictions, and the omega governs the impact of the working memory nodes. By adjusting the weighting factors, the cognitive architecture can adapt to different scenarios and prioritise certain factors over others, enabling a flexible and robust decision-making process.

During the training stage of the BorisCAD procedure, a crucial step involves evaluating the trained models on a separate testing subset to obtain the necessary metrics for classification and induction. The models, trained on a separate training subset, are applied to the testing subset containing unseen data. This allows the computation of various metrics to assess the models' performance and generalisation ability. By analysing these metrics, informed decisions can be made regarding potential areas that require attention or adjustments in the decision criteria based on consistent misclassification or poor validation results.

To ensure unbiased evaluation, the train-test split process follows data shuffling to eliminate any bias introduced by the original order of the dataset. This randomisation allows the machine learning models to learn patterns and relationships in the training set without being influenced by specific ordering. The shuffled data is divided into two subsets: the training and testing sets. The allocation of the shuffled data is typically predefined, with a percentage assigned for training and the remaining portion used for testing. Most of the data is used for training the models, while the remaining samples in the testing set are reserved for evaluating the models' performance on unseen data.

3.6 Integrated Technologies

In the current era of rapid technological advancements, the seamless integration of various technologies has become essential across different domains, including academia, industry, and research. This integration is vital in developing innovative and efficient infrastructures that effectively address complex challenges and achieve remarkable outcomes.

Python, a high-level interpreted programming language, is a powerful tool for creating integrated technological solutions. Renowned for its adaptability, user-friendliness, and extensive libraries, Python is widely acclaimed in the industry. Its simplicity, readability, and versatility make it a preferred choice for programmers, allowing them to express complex concepts using fewer lines of code than other programming languages. Given its strengths, Python has been adopted as the primary programming language for implementing the cognitive architecture presented throughout this chapter.

The broad range of libraries offered by Python, including *NumPy*, *Pandas*, and *Matplotlib*, empowers efficient processing, manipulation, and analysis of large datasets. These libraries provide robust data structures, statistical functions, and visualisation tools that facilitate data-driven decision-making and enable the extraction of valuable insights.

PyTorch, an increasingly popular open-source machine learning framework, has gained substantial traction due to its dynamic computational graph and extensive support for deep learning applications. Leveraging PyTorch's vast library of pre-built neural network modules, it provides a flexible platform for rapidly developing and deploying advanced machine learning models.

Moreover, Python's integration capabilities extend to PyTorch's support for distributed computing, enabling the scalability of machine learning models across multiple Graphics Processing Units (GPUs) or even clusters of machines. This scalability empowers the training of larger and more intricate models, thereby enhancing the performance and accuracy of integrated technological solutions. Consequently, all previously explored machine learning mechanisms will be implemented utilising the existing frameworks of PyTorch.

Additionally, the *NetworkX* library is valuable for visualising and analysing complex network structures within BorisCAD, the Storage System's memories. As a Python library, NetworkX provides comprehensive tools for studying intricate networks. Its intuitive interface and extensive functionality make it suitable for visualising and analysing interconnected nodes and edges within these networks.

3.7 Data Catalogue

The "Data Catalogue" subsection introduces the primary use case of analysing X-ray images within BorisCAD, a cognitive architecture context. Its fundamental purpose is to identify and detect pathologies present in these images. By leveraging integrated technologies, BorisCAD aims to enhance diagnostic capabilities and provide valuable support to healthcare professionals. This subsection focuses on the data catalogue employed by BorisCAD, which plays a critical role in enabling accurate pathology identification.

The data catalogue encompasses a diverse range of X-ray images from two large datasets, facilitating the training and evaluation of machine-learning models. Through the exploration of this data catalogue, the subsection highlights the importance of comprehensive and representative datasets in developing and implementing integrated technological solutions for X-ray image analysis.

During the training of the segmentation network, a single dataset is employed, sourced from the studies conducted by Candemir *et al.,* [111] and Jaeger *et al.,* [112]. This dataset encompasses original lung X-ray images along with their corresponding masks. This dataset aims to facilitate the training of the segmentation network to accurately identify and delineate specific regions or objects of interest within the lung X-ray images. By utilising the paired images and masks, the segmentation network can learn to effectively distinguish the lung structures from the surrounding background and other adjacent tissues, thereby enabling precise and reliable segmentation results.

3.7.1 CheXpert Dataset

The CheXpert dataset [113] is a widely used and highly regarded medical imaging dataset specifically designed for training and evaluating machine learning models in chest X-ray interpretation. It consists of an extensive collection of chest X-ray images and corresponding radiologist interpretations. It is crucial as medical imaging is inherently subjective, and radiologists often have varying levels of certainty in their diagnoses. The dataset provides detailed annotations that reflect the degree of uncertainty associated with each radiologist's interpretation, enabling the exploration of uncertainty modelling and analysis within machine learning algorithms.

The CheXpert dataset encompasses a wide range of pathologies and abnormalities commonly encountered in chest X-ray imaging, including pneumonia, lung nodules, pleural effusion, and cardiomegaly. Moreover, the CheXpert dataset addresses the challenge of imbalanced data, a common issue in medical imaging datasets. It annotates many images with positive and negative pathology labels, ensuring a balanced representation of different pathologies and enabling robust training and

evaluation of machine learning models.

Although this dataset is extensive and provides a valuable resource for chest X-ray interpretation, it is essential to note that the images within the CheXpert dataset come from various view positions. This variation necessitates a pre-processing step to isolate the frontal view images, which serve as the standardised position for analysis. This pre-processing ensures consistency and enables accurate comparison and diagnosis when developing machine learning algorithms based on this dataset.

In addition to the abundant collection of X-ray images, the CheXpert dataset is accompanied by a meticulously documented file that encompasses vital information for each image, including details regarding the specific view position from which the X-ray was captured. This comprehensive file plays a pivotal role in facilitating efficient data management and analysis, eliminating the need for manual review or reliance on image processing techniques to determine the view position.

The bar graph in figure 34 visually represents the label distribution in the CheXpert dataset, displaying the total occurrences of each label. It provides an informative overview of the frequencies of different pathologies and abnormalities within the dataset, allowing easy identification of the most common and rare labels. This graph offers valuable insights into the prevalence of specific conditions, contributing to a comprehensive understanding of the label distribution in the CheXpert dataset.

The accompanying pie chart in figure 35 presents a clear comparison of the percentage composition of each label in the CheXpert dataset. Each label is depicted as a distinct segment, with the size of each segment indicating the proportionate occurrence of that label. This visual representation allows for a quick assessment of the label distribution, highlighting the relative importance of different pathologies and abnormalities. The pie chart provides a concise overview, enabling a straightforward interpretation and comparison of the various labels in the CheXpert dataset.

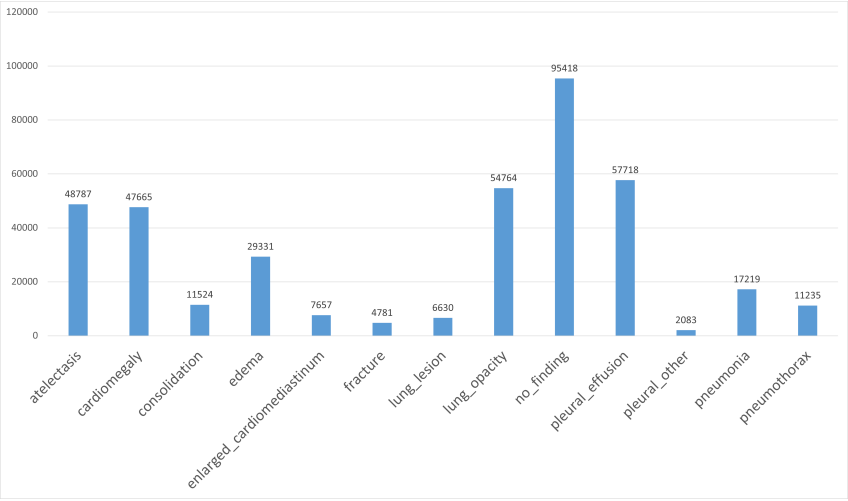


Figure 34: CheXpert total occurrences for each label.

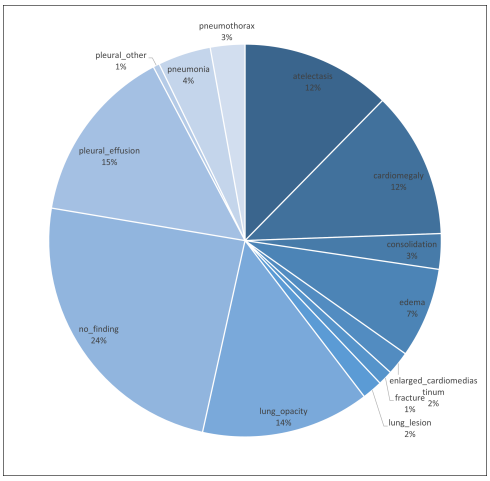


Figure 35: CheXpert label comparison pie-chart.

3.7.2 ChestX-ray8 Dataset

Including the NIH Chest X-ray Dataset [114] significantly enriches the domain of chest X-ray analysis, augmenting the existing knowledge base with a vast corpus of meticulously annotated images encompassing a broad spectrum of pathologies and abnormalities. This extensive dataset serves as a catalyst for in-depth explorations into diverse medical conditions, thereby fostering the advancement and evaluation of sophisticated machine-learning algorithms tailored explicitly for chest X-ray interpretation.

Moreover, the NIH dataset distinguishes itself from the CheXpert dataset under its inherent imbalanced label distribution, which has the potential to provide a distinct perspective on system performance. The presence of rare conditions within this dataset poses formidable challenges in algorithmic development and evaluation, necessitating a judicious and methodical approach to ensure impartiality and the robustness of the performance metrics employed. Attending to the intricacies of label imbalance engenders a deeper comprehension of the nuances involved in automated diagnostic methodologies, propelling the frontiers of knowledge and empowering advances in cutting-edge diagnostic techniques and clinical decision support systems.

The inherent label imbalance within the NIH Chest X-ray Dataset is readily discernible through the visualisation of the bar plot presented in Figure 36. Similarly, the accompanying pie chart depicted in Figure 37 succinctly captures the contrasting proportions of each label, providing a concise and visually impactful representation of the dataset's label distribution.

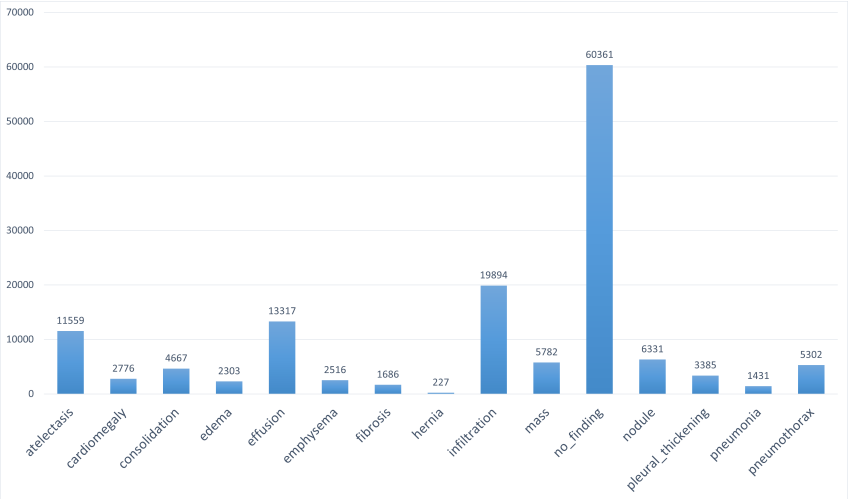


Figure 36: ChestX-ray8 total occurrences for each label.

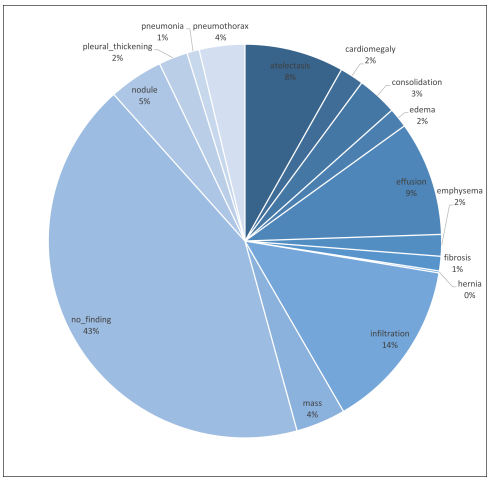


Figure 37: ChestX-ray8 label comparison pie-chart.

3.7.3 Pneumonia Dataset

Additionally, a smaller dataset was included in the data catalogue, which consisted of pneumonia-related X-ray images by Kermany *et al.,* [115]. While this dataset was not as extensive as the other datasets, it provides a valuable addition to the research. The inclusion of this dataset allowed for the exploration of more intricate differences between each label, providing further insights into how the developed cognitive architecture would behave in such scenarios.

The pneumonia dataset showcased a variety of pneumonia-related conditions, including bacterial and viral infections. This diversity allowed for a comprehensive analysis of the cognitive architecture's performance in identifying and classifying different types of pneumonia, naturally less perceivable than different pathologies. The dataset's labels included categories such as "Normal", "Bacterial Pneumonia" and "Viral Pneumonia".

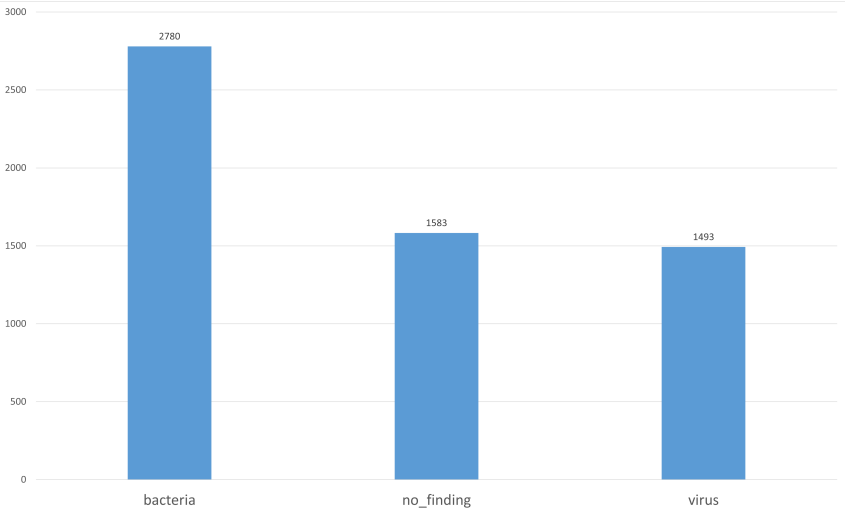


Figure 38: Pneumonia Dataset total occurrences for each label.

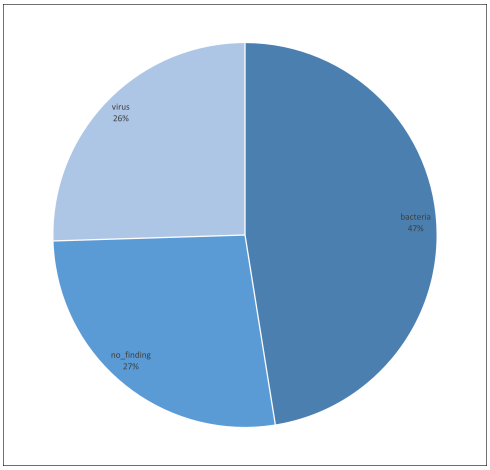


Figure 39: Pneumonia Dataset label comparison pie-chart.

To gain a better understanding of the dataset, an analysis of the total occurrences for each label was performed, as shown in Figure 38. This analysis revealed the distribution of different pneumonia-related conditions within the dataset. Furthermore, a label comparison pie chart, presented in Figure 39, illustrated the proportional representation of each label in the dataset.

3.7.4 Addressing class imbalance

Class imbalance is a prevalent challenge encountered in numerous machine learning tasks, characterised by a significant skew in the distribution of instances across different classes within a dataset. This imbalance can have deleterious effects on classifier performance, as it often leads to a prioritisation of the majority class at the expense of accurately predicting the minority class. Consequently, practical techniques for mitigating class imbalance have become an area of great interest

within the machine-learning community.

One method that has garnered considerable attention in recent years is the weighted random sample approach. The weighted random sample method addresses class imbalance by assigning distinct weights to instances during the training process. Notably, instances from the minority class are assigned higher weights than those from the majority class. By doing so, instead of randomly selecting instances from the entire dataset, instances are selected with probabilities inversely proportional to their assigned weights. This ensures a higher likelihood of selecting minority class instances, thereby reducing the bias towards the majority class and promoting a more balanced representation of the classes.

Algorithm 6 Weighted Random Sampler algorithm's pseudocode.

Require:

$DatasetD$ with instances and their corresponding class labels

procedure WeightedRandomSampler
    Calculate class frequencies for each class in D.
    Calculate weights W for each instance based on class frequencies.
    Normalise weights to sum up to 1.
    Initialise an empty sampled dataset S.
    for each class c **do**
        Obtain the instances belonging to class c.
        Calculate the number of instances I to sample from c based on W
        Sample class c using I.
        Add the sampled instances to S.
    end for
return Return the sampled dataset S.
end procedure

The pseudo-code algorithm in 6 elaborates on Weighted Random Sampler workings. The algorithm follows a simple procedure, starting with calculating class frequencies and weights. These weights are normalised to ensure they sum up to 1. The algorithm then iterates over each class, determining the desired number of instances to sample based on the desired balanced ratio. Finally, instances are sampled from each class according to their weights, and the resulting dataset exhibits a balanced class distribution.

Chapter 4

Practical Application and Execution of the Proposed Solution

This chapter focuses on the practical implementation and execution of the proposed solution for medical image classification, building upon the earlier presented comprehensive architecture design. The objective is to provide a detailed account of the steps to transform the theoretical framework into a functioning system capable of accurately classifying medical images.

The implementation process involves translating the architectural blueprint into practical code using the previously-mentioned Python libraries and frameworks. The chapter walks through the step-by-step development of the system, highlighting the key components, methodologies, and algorithms employed at each stage.

To ensure reproducibility and transparency, the provided code snippets are clear and concise, commented in full. The configurations and parameters used for training and testing the BorisCAD are also clearly provided.

Furthermore, the training process is explored in detail, covering the selection of loss functions and optimisation algorithms and the fine-tuning of hyperparameters and model selection to maximise accuracy and minimise overfitting.

4.1 Computational System

After designing BorisCAD's Computation System with its hybrid system design encompassing sub-symbolic processing and symbolic reasoning, it is time to delve into the implementation details. This phase brings the design to life, showcasing the actual code implementation for both levels: the bottom

level representing sub-symbolic processing and the top level encompassing symbolic reasoning.

At the bottom level, BorisCAD harnesses the power of neural networks, specifically Convolutional Neural Networks. These specialised networks extract and learn intricate features from image data. By implementing the sub-symbolic processes, the system can recognise objects and patterns within visual data by constructing complex representations of images.

Transitioning to the top level, the system employs symbolic reasoning to make decisions based on the features learned by the sub-symbolic processes. The implementation utilises Decision Trees for their interpretability and flexibility. DTs enable the system to reason effectively about complex relationships among the extracted features, facilitating accurate predictions and informed decision-making.

4.1.1 Bottom-Level Subsystem

At the bottom level of BorisCAD's Computation System, the implementation incorporates two existing neural networks: segmentation and classification networks. These networks take advantage of a popular deep learning framework, PyTorch, providing a seamless environment for designing and training robust models.

The framework utilised in this implementation introduces the concept of Tensors, which serve as a fundamental data structure for neural network computations. Tensors efficiently represent and manipulate multi-dimensional arrays, enabling complex operations and transformations within the network architecture.

Moreover, the framework simplifies the implementation of neural networks by offering intuitive methods and functionalities. Designing intricate neural network architectures becomes more straightforward and efficient with a comprehensive library of pre-defined layers and modules, including operations for convolution, activation functions, pooling, and more.

By leveraging the framework's tensor operations and user-friendly interface, BorisCAD's Computation System maximises the potential of deep learning techniques. The segmentation and classification networks seamlessly integrate the methods offered by the framework, enabling the system to extract meaningful features from image data and make accurate predictions based on the learned representations.

a) Segmentation Network

The implementation process begins with the construction of the Segmentation Network, as seen in Listing 1, a pivotal component of the module. The initial phase revolves around the instantiation of the network's essential attributes, primarily dedicated to training the UNet or AttentionUNet models. User input through a Boolean variable, denoted as *attention*, determines the selection between these models. Additionally, the initialisation of optimisers and schedulers is deferred to the Learning System, thus currently employing type hints and placeholders.

A crucial consideration in implementing the Segmentation Network is the utilisation of the `BCEWithLogitsLoss` function for semantic segmentation. This choice stems from its effectiveness in handling binary pixel-wise classification tasks, where each pixel is assigned to either foreground or background classes. By employing `BCEWithLogitsLoss`, the network can effectively address the challenges associated with the class imbalance and overlapping regions, ensuring accurate and robust segmentation results.

Listing 1 Segmentation Network __init__function.

```
11 class SegmentationNetwork(nn.Module):
12     def __init__(
13         self,
14         attention: bool = True,
15         device: torch.device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu"),
16     ) -> None:
17         """Segmentation Network constructor
18         Args:
19             attention (bool, optional): if attention or not. Defaults to True.
20             device (_type_, optional): Defaults to torch.device("cuda:0" if torch.cuda.is_available() else
    "cpu").
21         """
22         super(SegmentationNetwork, self).__init__()
23         self.device = device
24         # conditional usage of the attention network
25         self.model = AttentionUNet() if attention else UNet()
26         self.loss_fn = nn.BCEWithLogitsLoss()  # loss function to evaluate the model
27         # optimizer and scheduler are started in the Learning System
28         self.optim: torch.optim.Adam = None
29         self.scheduler: torch.optim.lr_scheduler.ReduceLROnPlateau = None
30         # scaler used when CUDA is available
31         self.scaler = torch.cuda.amp.GradScaler() if torch.cuda.is_available() else None
32         self.prev_training: torch.Tensor = None  # tensor to hold previous training metrics
33         self.to(self.device)  # sends the module to CUDA or keeps it in the CPU
```

The provided code segment demonstrates a pattern commonly observed in the bottom-level modules, highlighting the similarities between classes and methods within this section of the architecture. Notably, the inclusion of a device parameter allows for the specification of the device on which the networks will be executed. By default, this parameter assumes a value of either `"cpu"` or `"cuda:0"`, depending on the availability and compatibility of a suitable GPU for hardware acceleration

within the system. The incorporation of a device parameter enables the code to be more flexible and adaptable to different hardware configurations. It caters to scenarios where GPU acceleration is preferred when available, and seamlessly falls back to CPU execution if a compatible GPU is not present. This design choice enhances the code's generalisability and allows it to effectively utilise the available hardware resources. Furthermore, the final line of the code segment, labeled as *Line 32*, demonstrates the utilisation of the specified device for executing the module. This step ensures that the module is appropriately assigned to the desired device, thus aligning with the chosen hardware configuration for efficient execution.

Another characteristic that will be prevalent in the subsequent code snippets is the usage of the `super()` initialisation method in the context of inheriting from the `nn.Module` class. Through this inheritance, a subclass extends the base class and gains access to its properties and methods, relevant in facilitating tasks like forward and back propagation of the networks. The `nn.Module` class, in particular, has its own initialisation method, `__init__()`, responsible for initialising critical components such as parameters, buffers and sub-modules, while also performing necessary setup operations. To ensure the proper execution of these operations, it is needed for the subclass's `__init__()` method to invoke `super().__init__()`.

Listing 2 UNet block __init__function.

```
5  class UNet_Block(nn.Module):
6      def __init__(self, in_channels: int, out_channels: int) -> None:
7          """Convolution Module, two consecutive convolution with Normalization and ReLU activation
8          Args:
9              in_channels (int): number of input channels (either last layer output or number of channels in
       the image (RGB-3, Gray-1))
10             out_channels (int): number of output channels
11         """
12         super(UNet_Block, self).__init__()
13         # Define the convolutional layers
14         self.conv = nn.Sequential(
15             nn.Conv2d(in_channels=in_channels, out_channels=out_channels, kernel_size=3, padding=1,
                   bias=False),
16             nn.BatchNorm2d(num_features=out_channels),
17             nn.ReLU(inplace=True),
18             nn.Conv2d(in_channels=out_channels, out_channels=out_channels, kernel_size=3, padding=1,
                   bias=False),
19             nn.BatchNorm2d(num_features=out_channels),
20             nn.ReLU(inplace=True),
21         )
```

Delving deeper into the specific networks designed for the segmentation task, the code snippet that represents the implementation of the block discussed earlier (referenced as "Listing 2") is examined. This block corresponds to the architecture illustrated in Figure 9a. Within the `UNet_Block` class, the block's layers are defined using the `nn.Sequential` module. The layers consist of a 2D convolutional layer, batch normalisation, ReLU activation, another 2D convolutional layer, batch normalisation again,

and another ReLU activation. These layers collectively form the desired architecture of the block.

Similarly, the Listing 3 presents the implementation of the Attention Gate block designed in Figure 9b. The `AttentionGate` class is designed to incorporate attention mechanisms into a neural network. It takes two input parameters: `features`, representing the number of input channels or features, and `n_coefficients`, denoting the number of attention coefficients to be calculated. Within the class's `__init__()` method, the attention gate components are defined. These components include the `W_gate`, `W_x`, `psi`, and `relu` layers. The `W_gate` and `W_x` layers consist of a 1x1 convolution followed by batch normalisation, while the `psi` layer performs another 1x1 convolution followed by batch normalisation and a *sigmoid* activation. The `relu` layer utilises the ReLU activation function.

Listing 3 Attention Gate block __init__function.

```
27 class AttentionGate(nn.Module):
28     def __init__(self, features: int, n_coefficients: int) -> None:
29         """Attention Gate
30         Args:
31             features (int): number of features to
32             n_coefficients (int): number of transitory coefficients
33         """
34         super(AttentionGate, self).__init__()
35         # Define the W_gate module with a convolutional layer and batch normalization
36         self.W_gate = nn.Sequential(nn.Conv2d(in_channels=features, out_channels=n_coefficients,
               kernel_size=1), nn.BatchNorm2d(n_coefficients))
37         # Define the W_x module with a convolutional layer and batch normalization
38         self.W_x = nn.Sequential(nn.Conv2d(in_channels=features, out_channels=n_coefficients,
               kernel_size=1), nn.BatchNorm2d(n_coefficients))
39         # Define the psi module with a convolutional layer, batch normalization, and sigmoid activation
40         self.psi = nn.Sequential(nn.Conv2d(in_channels=n_coefficients, out_channels=1, kernel_size=1),
               nn.BatchNorm2d(1), nn.Sigmoid())
41         self.relu = nn.ReLU(inplace=True)  # ReLU activation
```

The forward function in the `AttentionGate` class seen in listing 4 implements the attention mechanism. It takes a gate tensor and a skip connection tensor as input. The gate tensor is processed through the `W_gate` layer, while the skip connection tensor is processed through the `W_x layer`. The attention coefficients are calculated by adding these processed tensors and applying the ReLU activation. The attention coefficients are then passed through the `psi` layer, which performs a convolution, batch normalisation, and *sigmoid* activation. Finally, the skip connection tensor is multiplied element-wise by the attention coefficients to obtain the modified tensor, which is returned as the output. This allows the network to dynamically weigh the importance of different elements in the skip connection based on the attention mechanism.

The implementation of the `AttentionUNet` class, as demonstrated in listing 5, introduces a customised variation of the UNet architecture. This modified version integrates attention gates, which are illustrated in Figure 8. It consists of encoder and decoder blocks responsible for the downsampling and upsampling operations, respectively.

Listing 4 Attention Gate forward method.

```
def forward(self, gate: torch.Tensor, skip_connection: torch.Tensor) -> torch.Tensor:
    """Forward function for the attention gate
    Args:
        gate (torch.Tensor): gate tensor
        skip_connection (torch.Tensor): tensor of the skip connection
    Returns:
        torch.Tensor: weighted attention gate tensor
    """
    g1 = self.W_gate(gate)  # Pass the gate tensor through the W_gate module
    x1 = self.W_x(skip_connection)  # Pass the skip connection tensor through the W_x module
    psi = self.relu(g1 + x1)  # Element-wise addition of g1 and x1 followed by ReLU activation
    psi = self.psi(psi)  # Pass the combined tensor through the psi module for attention weighting
    return skip_connection * psi  # Multiply the skip connection tensor with the attention weights
```

The input image passes through a series of `UNet_Block` layers in the encoder ladder. The `in_channels` argument determines the number of input channels for the first `UNet_Block`, while the `features` list defines the number of output channels. The output of each `UNet_Block` is fed as input to the next block in the encoder ladder, progressively reducing the spatial dimensions of the feature maps and increasing the number of channels.

Listing 5 Attention U-Net __init__function.

```
class AttentionUNet(nn.Module):
    # Encoder and decoder blocks are pretty much the same, but the decoder has a Transposed Convolution
        that receives the skip connection from the previous encoder
    def __init__(self, in_channels: int = 1, out_channels: int = 1, features: list[int] = [64, 128, 256,
        512]) -> None:
        """UNet Structure definition but with Attention gates
        Args:
            in_channels (_type_): number of input channels (channels of the image)
            out_channels (_type_): number of output channels (usually one image, so 1 output channel)
            features (list, optional): number of features for each layer in the network. Defaults to [64,
    128, 256, 512].
        """
        super(AttentionUNet, self).__init__()
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)  # Max pooling layer for downsampling
        self._num_channels = in_channels
        # Encoder and Decoder modules
        self.encoder = nn.ModuleList()
        self.decoder = nn.ModuleList()
        # Encoder ladder
        for feature in features:
            self.encoder.append(UNet_Block(in_channels=in_channels, out_channels=feature))
            in_channels = feature
        # Decoder ladder with the attention gate blocks
        for feature in reversed(features):
            self.decoder.append(nn.ConvTranspose2d(in_channels=feature * 2, out_channels=feature,
                kernel_size=2, stride=2))
            self.decoder.append(AttentionGate(feature, feature // 2))
            self.decoder.append(UNet_Block(feature * 2, feature))
        # Connection between encoder and decoder, the bottom of the U in the network
        self.bottleneck = UNet_Block(features[-1], features[-1] * 2)
        # Output Layer
        self.final_layer = nn.Conv2d(in_channels=features[0], out_channels=out_channels, kernel_size=1)
```

In the decoder ladder, transposed convolutions (`nn.ConvTranspose2d()`) are used for upsampling the feature maps, increasing their spatial dimensions while reducing the number of

channels. The decoder also incorporates attention gate blocks (`AttentionGate`), which receive the output of the corresponding transposed convolution and the skip connection from the encoder ladder. The attention gate blocks dynamically compute attention coefficients based on these inputs, selectively emphasising or suppressing features. The skip connection allows information from the encoder to be integrated into the decoder.

The `forward()` method of the `AttentionUNet` class shown in Listing 6 better depicts the processing within the network. During the encoding phase, the input passes through the encoder blocks, and the output of each block is stored in the `skip_connections` list. The `bottleneck` layer performs the final encoding step without a skip connection.

Listing 6 Attention U-Net forward method.

```
158 def forward(self, X: torch.Tensor) -> torch.Tensor:
159     """Forward method for the AttentionUnet
160     Args:
161         X (torch.Tensor): input tensor
162     Returns:
163         torch.Tensor: segmented output tensor
164     """
165     # Connections between an encoder block and the assigned decoder block
166     skip_connections = []  # List to store skip connections from the encoder
167     # Encoding
168     for block in self.encoder:  # Iterate through the encoder blocks
169         X = block(X)  # Pass input through the current encoder block
170         skip_connections.append(X)  # Store the skip connection
171         X = self.pool(X)  # Max Pooling after each block to downsample
172     X = self.bottleneck(X)  # Final encoding layer, with no skip connection assotiated
173     skip_connections = skip_connections[::-1]  # Reverse the skip connections
174     # Decoding
175     for idx in range(0, len(self.decoder), 3):  # Iterate through the decoder blocks
176         X = self.decoder[idx](X)  # Transposed Convolution (Upsampling)
177         skip_connection = skip_connections[idx // 3]  # Retrieve the corresponding skip connection
178         X = self.decoder[idx + 1](X, skip_connection)  # Attention module combining skip connection and
                current output
179         concat_skip = torch.cat((X, skip_connection), dim=1)  # Concatenate skip connection and output
180         X = self.decoder[idx + 2](concat_skip)  # Double convolution to refine the concatenated features
181     return self.final_layer(X)  # Output layer
```

In the decoding phase, the `skip_connections` are reversed to match the decoding order. The decoder blocks are applied in reverse order, upsampling the feature maps while dynamically computing attention coefficients. The output of the attention gate is concatenated with the corresponding skip connection and passed through the UNet block for further refinement.

The property `num_channels` shown in Listing 7 allows accessing and modifying the number of channels in the UNet model. When accessed, it returns the current number of channels. When modified using the setter method, it updates the number of channels to the specified value. Additionally, it updates the first encoder block of the UNet model with the new number of channels. The `num_channels` property is useful for handling changes in the number of input channels dynamically, enabling flexibility in adapting the UNet model to different input configurations.

Listing 7 Attention U-Net number of channels property.

```
139 @property
140 def num_channels(self) -> int:
141     """Get the number of channels.
142     Returns:
143         int: The number of channels.
144     """
145     return self._num_channels
146 
147 @num_channels.setter
148 def num_channels(self, num_channels: int, feature: int = 64) -> None:
149     """Set the number of channels.
150     Args:
151         num_channels (int): The number of channels.
152         feature (int, optional): The feature size. Defaults to 64.
153     """
154     self._num_channels = num_channels  # Set the number of channels
155     # Update the first encoder block with the new number of channels
156     self.encoder[0] = UNet_Block(in_channels=num_channels, feature=feature)
```

In order to regulate the training process, a mechanism was implemented within the Segmentation Network class, as described earlier in the design section. The general `fit()` method, depicted in Figure 14a, demonstrates the flow of operations involved in training and is presented in Listing 8, where the underlying logic behind the training process is illustrated.

This initial stage of training takes as input a `DataLoader` object that contains both the images and corresponding masks, as well as the specified number of epochs for training. To ensure the proper functioning of the architecture's Learning System, the existence of an optimiser and a scheduler is verified. If these mechanisms are not assigned, an error will be raised, thus ensuring their availability throughout the training process.

The training process is then carried out for each epoch, iterating over the dataset. At the end of each epoch, the training results, such as loss values or evaluation metrics, are collected and stored in a list. This allows for further analysis, storage, or utilisation of these results beyond the immediate training context.

As one can see, a context is used in *Line 05*. During the verification stage, some desynchronisation in the execution of the scheduler step was verified, thus throwing a warning. This warning does not affect the execution, but for display purposes, it is ignored. The context implementation is depicted in Listing 9, where the decorator `@contextlib.contextmanager` is used to allow the function to be used as a context manager.

The **`with`** statement is used to create a context where warnings are caught and filtered. It ensures that any warnings within the indented code block are handled appropriately. The line `warnings.filterwarnings("ignore", category=warning)` configures the warnings module to ignore a specific category of warning. The category parameter is passed as an argument to

Listing 8 Segmentation Network fit method.

```
def fit(self, dataloader: DataLoader, epochs: int = 8) -> torch.Tensor:
    """Fit method for the Segmentation Network Model.
    Args:
        dataloader (DataLoader): Dataloader containing image and mask data for training.
        epochs (int, optional): Number of epochs to train the model. Defaults to 8.
    Returns:
        torch.Tensor: Training metrics for the current training.
    """
    # Verifies if the model is correctly initialized, as in, if a Learning System is assigned to the Network
    assert self.optim is not None and self.scheduler is not None, "No Learning System assigned to the model"
    results = []  # List to store the results of each epoch
    since = time.time()  # Time at the start of training
    for epoch in range(epochs):
        print(f"Epoch {epoch+1}/{epochs}")
        epoch_results = self.epoch_train(e_data=dataloader)  # Trains the network on the current epoch
        with ignore_warning(UserWarning):  # A UserWarning is sometimes triggered because of desynchronization, it is not relavant so this context is used to avoid it
            self.scheduler.step(sum(epoch_results[:, 0]) / len(dataloader))  # Adjusts the learning rate based on the average loss
        # Print training metrics for the current epoch
        print(f"Loss: {sum(epoch_results[:, 0])/len(dataloader)}; Dice score: {sum(epoch_results[:, 1])/len(dataloader)}; Accuracy: {sum(epoch_results[:, 2])/len(dataloader)*100:.2f}%; IoU: {sum(epoch_results[:, 3])/len(dataloader)}\n")
        results.append(epoch_results)  # Stores the epoch results
    time_elapsed = time.time() - since  # Total training time
    print(f"Training complete in {time_elapsed // 60:.0f}m {time_elapsed % 60:.0f}s")
    self.prev_training = torch.cat(results, dim=0)  # Concatenates the epoch results into a single tensor
    return self.prev_training  # Returns the training metrics for the current training
```

the `ignore_warning` function, specifying the type of warning to be ignored. The `yield` statement is used as a placeholder for the block of code that will be executed when the context manager is entered. In this case, it allows the user to execute their desired code within the context where the warning is ignored. Once the indented block of code is executed, the context manager automatically exits, and any temporarily ignored warnings are no longer suppressed.

For each epoch of training, the process done is presented in Listing 10 based on what was described in Figure 14b. The provided code snippet presents a method named `epoch_train()` within the `SegmentationNetwork` class. This method is responsible for training the model on a single epoch of data. It takes a `DataLoader` object named `e_data` as input, which holds the training images and masks. During each iteration, the method trains on a batch of images and labels, obtained from the `e_data` *DataLoader*. The training metrics for each batch are stored in a tensor called `epoch_metrics`, which is concatenated with the metrics from previous batches. At the end, the method returns the aggregated `epoch_metrics` tensor, representing the metrics obtained during the entire epoch of training.

The specific implementation for batch training is depicted in Listing 11, aligning with the previously discussed design illustrated in Figure 15. To ensure the proper handling of data, specifically the images

Listing 9 Ignore Warning context.

```
import contextlib


@contextlib.contextmanager
def ignore_warning(warning: Type[Warning]):
    """Context manager to ignore a specific warning type.
    Args:
        warning (Type[Warning]): The type of warning to ignore.
    """
    import warnings

    with warnings.catch_warnings():  # Catches all warnings that occur within the context
        warnings.filterwarnings("ignore", category=warning)  # Ignores the specified warning type
        yield  # Yields control back to the caller while ignoring the warning
```

Listing 10 Segmentation Network epoch training method.

```
def epoch_train(self, e_data: DataLoader) -> torch.Tensor:
    """Trains in a single epoch of data
    Args:
        e_data (DataLoader): holds the training DataLoader, it contains the training images and masks
    Returns:
        torch.Tensor: returns the epoch_metrics for training
    """
    epoch_metrics = torch.zeros(0, 4, device=self.device)  # Initializes an empty tensor to store the metrics
    for images, labels in tqdm(e_data, unit=" batch"):  # Iterates over the batches of training images and labels
        batch_metrics = self.batch_train(b_data_x=images, b_data_y=labels)  # Trains on a single batch of training images and labels
        epoch_metrics = torch.cat([epoch_metrics, batch_metrics], dim=0)  # Concatenates the batch metrics to the epoch metrics tensor
    return epoch_metrics
```

and masks, on the appropriate device (CPU or GPU), *Lines 81-82* are utilised. These lines confirm that the data is correctly placed on the designated device, considering that this system supports both CPU and GPU utilisation.

Listing 11 Segmentation Network batch training method.

```
def batch_train(self, b_data_x: torch.Tensor, b_data_y: torch.Tensor) -> torch.Tensor:
    """Trains on a single batch of data
    Args:
        b_data_x (torch.Tensor): Image data; b_data_y (torch.Tensor): Mask data
    Returns:
        torch.Tensor: Batch training metrics
    """
    # Moves the input image data to the device (GPU if available)
    b_data_x = b_data_x.to(self.device, non_blocking=True)
    b_data_y = b_data_y.to(self.device, non_blocking=True)
```

The code snippet in Listing 12 performs a forward pass through the model to obtain predictions (`output`) given an input tensor (`b_data_x`). It then computes the loss between the predictions and the target mask (`b_data_y`) using the specified loss function. The `autocast` context manager in PyTorch enables automatic mixed precision training by automatically casting computations to a lower-precision data

type. It helps reduce memory usage and improve computational efficiency without sacrificing accuracy. The output is then thresholded using a *sigmoid* function with a threshold of 0.5 to obtain binary predictions (`predictions`).

Next, the code stacks the loss and other calculated metrics into a single tensor (`batch_metrics`). The metrics are calculated by calling a private method `_calculate_metrics()` with the predictions and target mask as inputs. The loss tensor is detached from the computation graph using `detach()` to prevent gradients from flowing back during metric calculation. Finally, the batch metrics tensor is unsqueesed to add a batch dimension and returned.

Listing 12 Segmentation Network batch training forward with context.

```
83      with torch.autocast(device_type=self.device.type, dtype=torch.bfloat16, enabled=True):  # autocasts to
            the specific device type
84          output: torch.Tensor = self.model(b_data_x)  # Forward pass: computes the model's output
85          loss: torch.Tensor = self.loss_fn(output, b_data_y)  # Computes the loss between the output and
                target mask
86      predictions = (torch.sigmoid(output) > 0.5).float()  # Applies a threshold to the output to obtain
            binary predictions
87      # Stacks the metrics into a single tensor and unsqueezes it to have a batch dimension
88      # Detaches the loss tensor from the computation graph and calculates metrics
89      batch_metrics = torch.stack([loss.detach(), *self._calculate_metrics(predictions, b_data_y)],
            dim=0).unsqueeze(0)
```

The `_calculate_metrics` function seen in Listing 13 takes predicted masks (`preds`) and ground truth masks (`masks`) as input and calculates several metrics. These metrics include the dice score, mask accuracy, and intersection over union (IoU) score. The function returns a list of torch Tensors, where each element represents a specific metric. The metrics are calculated using the corresponding functions `dice_score()`, `mask_accuracy()`, and `iou_score()` from the `LearningSystem` class and will be discussed further in this chapter.

Listing 13 Segmentation Network batch training metrics.

```
100 def _calculate_metrics(self, preds: torch.Tensor, masks: torch.Tensor) -> list[torch.Tensor]:
101     """Calculates various metrics based on the predicted masks and ground truth masks.
102     Args:
103         preds (torch.Tensor): Predicted masks.
104         masks (torch.Tensor): Ground truth masks.
105     Returns:
106         list[torch.Tensor]: List of metrics including dice score, mask accuracy, and intersection over
        union (IoU) score.
107     """
108     return [
109         LearningSystem.dice_score(preds, masks),  # Calculate the dice score metric
110         LearningSystem.mask_accuracy(preds, masks),  # Calculate the mask accuracy metric
111         LearningSystem.iou_score(preds, masks),  # Calculate the intersection over union (IoU) score
                metric
112     ]
```

Between *Line 90-98* from Listing 14, the provided code snippet involves a conditional block that handles the backpropagation and optimisation step during the training process. If the `self.scaler`

object exists, it implies that automatic mixed-precision training is enabled. In this case, the loss value (loss) is scaled using the `self.scaler.scale` method, followed by the backward pass (`backward()`) to compute gradients. Then, the optimiser (`self.optim`) performs a step based on the scaled gradients using the `self.scaler.step` method. Finally, the `self.scaler.update` method updates the scaling factor for future iterations. On the other hand, if the `self.scaler` object does not exist, indicating that mixed-precision training is not enabled, the backward pass is performed directly on the loss value (`loss.backward()`). Then, the optimiser (`self.optim`) performs a step based on the gradients. In both cases, the `batch_metrics` tensor is returned as the output of this code snippet, which contains the computed metrics for the batch, concluding the training for the specific batch of data.

Listing 14 Segmentation Network batch training backward pass and performance updates.

```
90      self.optim.zero_grad(set_to_none=True)  # Clears the gradients of the optimizer
91      if self.scaler:
92          self.scaler.scale(loss).backward()  # Backward pass: computes the gradients using automatic mixed
                precision (if enabled)
93          self.scaler.step(self.optim)  # Updates the model's parameters using the optimizer (scaled
                gradients)
94          self.scaler.update()  # Updates the scale for automatic mixed precision
95      else:
96          loss.backward()  # Backward pass: computes the gradients
97          self.optim.step()  # Updates the model's parameters using the optimizer (un-scaled gradients)
98      return batch_metrics  # Returns the batch training metrics
```

The forward pass of the segmentation model, described in Listing 15, takes an input tensor X and moves it to the specified device (GPU if available). It creates a clone of the input tensor `X_skip` for later multiplication. The input tensor `X` is then passed through the model, computing the model's output. To ensure that the output values are between 0 and 1, element-wise clamping and rounding are applied. Finally, the output tensor is obtained by performing element-wise multiplication between the processed output tensor and the cloned input tensor.

Listing 15 Segmentation Network forward method.

```
114 def forward(self, X: torch.Tensor) -> torch.Tensor:
115     """Forward pass of the model
116     Args:
117         X (torch.Tensor): Input tensor
118     Returns:
119         torch.Tensor: Output tensor
120     """
121     X = X.to(self.device)  # Moves the input tensor to the device (GPU if available)
122     X_skip = X.clone()  # Creates a clone of the input tensor for later multiplication
123     X = self.model(X)  # Forward pass: computes the model's output
124     X = torch.clamp(X, min=0, max=1).round()  # Applies element-wise clamping and rounding to ensure
            values are between 0 and 1
125     return X * X_skip  # Element-wise multiplication of the output tensor and the cloned input tensor
```

b) Classification Network

Following the Segmentation Network in this architecture's pipeline is the Classification Network. Code Listing 16 implements the initialisation function as described in the UML diagram in Figure 13. The `ClassificationNetwork` class includes a dictionary called `network_types` that stores lambda functions as values. These lambda functions generate distinct ResNet network architectures (e.g., ResNet50, ResNet101, ResNet152) based on the provided number of classes. During initialisation, the class selects the appropriate lambda function from the `network_types` dictionary based on the specified network type, enabling the creation of the corresponding ResNet network architecture. Furthermore, the class configures essential components like the loss function, optimiser, scheduler, and gradient scaler. Additionally, it supports the utilisation of a pre-trained segmentation model.

Listing 16 Classification Network __init__function.

```
11  class ClassificationNetwork(nn.Module):
12      network_types = {
13          "resnet50": lambda num_classes: ResNet(ResNet.ResNet50_LAYERS, num_classes=num_classes),
14          "resnet101": lambda num_classes: ResNet(ResNet.ResNet101_LAYERS, num_classes=num_classes),
15          "resnet152": lambda num_classes: ResNet(ResNet.ResNet152_LAYERS, num_classes=num_classes),
16      }
17
18      def __init__(
19          self,
20          num_classes: int,
21          net_type: str,
22          segmenter: nn.Module = None,
23          device: torch.device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu"),
24      ) -> None:
25          """Classification Network model.
26          Args:
27              num_classes (int): Number of output classes.
28              net_type (str, optional): Network type. Defaults to "resnet50".
29              device (str, optional): Device to be used (cuda or cpu). Defaults to cuda:0 if available,
    otherwise cpu.
30              segmenter (nn.Module, optional): Previously trained segmentation model. Defaults to None.
31          """
32          super(ClassificationNetwork, self).__init__()
33          self.device = device
34          # Checks if the network string is a valid implemented ones
35          if net_type not in ClassificationNetwork.network_types.keys():
36              raise RuntimeError("Invalid network type")
37          self.model = ClassificationNetwork.network_types[net_type](num_classes)  # Instantiates the chosen
                network type
38          self.loss_fn = nn.BCEWithLogitsLoss()  # Binary cross-entropy loss function
39          # optimizer and scheduler are started in the Learning System
40          self.optim: torch.optim.SGD = None  # Placeholder for the optimizer
41          self.scheduler: torch.optim.lr_scheduler.StepLR = None  # Placeholder for the learning rate
                scheduler
42          # scaler used when CUDA is available
43          self.scaler = torch.cuda.amp.GradScaler() if torch.cuda.is_available() else None  # GradScaler for
                mixed precision training
44          self.prev_training: torch.Tensor = None  # tensor to hold previous training metrics
45          # The classification network makes use of a previously trained segmentation model
46          self.segmenter = segmenter  # Segmentation model used for pre-processing
47          self.to(self.device)  # Sends the module to CUDA or keeps it on the CPU
```

Delving into the ResNet network-specific implementation, it is possible to define a general block

within this network architecture called `ResBlock`. This block is responsible for processing the input and producing the desired output. The `ResBlock` `class` is also a subclass of `nn.Module` from the PyTorch framework. Within its `__init__()` method as seen in Listing 17, parameters such as the number of input channels (`in_channels`), the number of output channels (`out_channels`), an optional downsampling block (`i_downsample`), and the stride size (`stride`) are taken. This blocks structure is implemented according to the designed block structure presented in Figure 12.

Listing 17 ResBlock __init__function.

```
6  class ResBlock(nn.Module):
7      # For a ResNet architecture, the expansion (relation between the number of input
8      # channels in a block vs the number of output channels of that block is '4')
9      expansion = 4
10
11     def __init__(self, in_channels: int, out_channels: int, i_downsample: ResBlock = None, stride: int =
           1) -> None:
12         """ResNet block definition
13         Args:
14             in_channels (int): input channels
15             out_channels (int): output channels
16             i_downsample (_type_, optional): donwsampling block. Defaults to None.
17             stride (int, optional): stride size. Defaults to 1.
18         """
19         super(ResBlock, self).__init__()
20         # Define the convolutional layers
21         self.conv = nn.Sequential(
22             nn.Conv2d(in_channels=in_channels, out_channels=out_channels, kernel_size=1, stride=1,
                   padding=0, bias=False),
23             nn.BatchNorm2d(num_features=out_channels),
24             nn.ReLU(inplace=True),
25             nn.Conv2d(in_channels=out_channels, out_channels=out_channels, kernel_size=3, stride=stride,
                   padding=1),
26             nn.BatchNorm2d(num_features=out_channels),
27             nn.ReLU(inplace=True),
28             nn.Conv2d(in_channels=out_channels, out_channels=(out_channels * self.expansion),
                   kernel_size=1, stride=1, padding=0),
29             nn.BatchNorm2d(num_features=(out_channels * self.expansion)),
30         )
31         self.i_downsample = i_downsample  # Store the value of i_downsample for downsampling
32         self.relu = nn.ReLU(inplace=True)  # ReLU activation for the residual connection
```

Inside the `__init__()` method, the block's operations are defined using a `nn.Sequential` container. It consists of convolutional layers, batch normalisation layers, and ReLU activation functions, similar to what was seen in Figure 12. The first convolutional layer performs a 1x1 convolution on the input data, followed by batch normalisation and ReLU activation. The second convolutional layer applies a 3x3 convolution with a specified stride, padding, and output channels. Another batch normalisation and ReLU activation follow it. The third convolutional layer employs a 1x1 convolution to adjust the number of output channels based on the expansion factor, which is set to 4 in ResNet (`(out_channels * self.expansion)`). Batch normalisation is applied again after this layer.

The `i_downsample` attribute holds an optional downsampling block, allowing for downsampled connections within the block. Lastly, the `relu` attribute represents an additional ReLU activation

function used within the block, as seen in Listing 18. This code block represents the `forward()` function of the `ResBlock` class and it's sequential processing.

Listing 18 ResBlock forward function.

```
34 def forward(self, X: torch.Tensor) -> torch.Tensor:
35     """Forward method for the ResNet block
36     Args:
37         X (torch.Tensor): Input tensor
38     Returns:
39         torch.Tensor: Output tensor
40     """
41     X_skip = X.clone()  # Create a copy of the input tensor for the skip connection
42     X = self.conv(X)  # Pass the input tensor through the convolutional
43     # Apply downsampling to the skip connection if i_downsample is not None
44     if self.i_downsample is not None:
45         X_skip = self.i_downsample(X_skip)
46     X = X + X_skip  # Add the convolved tensor and the skip connection tensor
47     return self.relu(X)  # Apply ReLU activation to the combined tensor and return the result
```

The ResNet creation function (Listing 19), represented by the `ResNet` class, constructs a ResNet network architecture with a specified number of layers, following the structure presented in Figure 11. It utilises a `_make_layer` function to create the desired layer blocks based on the given number of layers for each filter size. The architecture includes an initial input block, a sequence of layer blocks, and concludes with a fully connected layer (`nn.Linear`) for classification, where the number of input features is determined by the expansion factor of the last ResNet block (`512 * ResBlock.expansion`) and the number of output classes specified.

The `_make_layer` function shown in Listing 20 creates a feature layer within the ResNet architecture. It takes the number of total blocks in the layer (`res_blocks`), the number of filters for the blocks (`filters`), and an optional stride size (`stride`) as inputs. The function begins by initialising a variable `ii_downsample` to track whether downsampling is required. It then creates an empty `nn.ModuleList()` named layers to store the individual blocks in the layer.

The function checks if downsampling is needed by comparing the stride and the number of input channels (`self.in_channels`) with the appropriate values. If downsampling is necessary, it creates a downsampling block (`ii_downsample`) consisting of a 1x1 convolutional layer and batch normalisation. After that, the `_make_layer` function includes the first block of the layer by creating an instance of the ResBlock class. The `self.in_channels` determines the number of input channels, the number of output channels is set to `filters`, the downsampling block is assigned to `ii_downsample`, and the stride value is set to `stride`.

The function then updates the number of input channels (`self.in_channels`) by multiplying the filters by the expansion factor of the `ResBlock` class (`ResBlock.expansion`), which is always 4 for

Listing 19 ResNet network initialisation.

```
class ResNet(nn.Module):
    ResNet50_LAYERS = [3, 4, 6, 3]
    ResNet101_LAYERS = [3, 4, 23, 3]
    ResNet152_LAYERS = [3, 8, 36, 3]

    def __init__(self, res_layers: list, num_classes: int, num_channels: int = 3) -> None:
        """ResNet initialization
        Args:
            res_layers (list): list with the number of layers for each filter size in the network
            num_classes (int): number of output classes
            num_channels (int, optional): channels of the image (3 if 'RGB', 1 if 'Grayscale'). Defaults to 1.
        """
        super(ResNet, self).__init__()
        self._num_channels = num_channels
        self._num_classes = num_classes
        self.in_channels: int = 64  # Initial number of channels
        # Input layer: Convolution, BatchNorm, ReLU, and MaxPool
        self.input = nn.Sequential(
            nn.Conv2d(in_channels=num_channels, out_channels=self.in_channels, kernel_size=7, stride=2, padding=3, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1),
        )
        # List of residual layers
        self.network = nn.ModuleList(
            [
                self._make_layer(res_layers[0], 64),
                self._make_layer(res_layers[1], 128, stride=2),
                self._make_layer(res_layers[2], 256, stride=2),
                self._make_layer(res_layers[3], 512, stride=2),
            ]
        )
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))  # Adaptive average pooling layer
        self.fc = nn.Linear(in_features=(512 * ResBlock.expansion), out_features=num_classes)  # Fully connected layer for classification
```

the ResNet models used (ResNet50, ResNet101, ResNet152). Finally, the remaining residual blocks are added to the layers list using a loop that iterates `res_blocks - 1` time. Each iteration creates a `ResBlock` with the updated input channels (`self.in_channels`) and the filters specified.

The `forward()` function, as depicted in Listing 21, defines the forward pass of the ResNet classifier. The function takes an input tensor `X` and applies initial layers to preprocess the input. It then iterates through each layer in the network, passing the input tensor through them. After the layers, adaptive average pooling is applied to the tensor. The tensor is then reshaped into a 2D shape to be fed into the fully connected layer for classification. Finally, the output of the fully connected layer is returned as the result of the forward pass. The final version of this method will be further discussed in the Storage System section since this is the function responsible for triggering node connection in memory, as described previously during the design.

The ResNet network, like the AttentionUNet implementation, supports changes in the number of input channels. In Listing 22, the code demonstrates how the input layer is redefined to accommodate the

Listing 20 ResNet network layer creation function.

```
def _make_layer(self, res_blocks: int, filters: int, stride: int = 1) -> nn.Sequential:
    """Makes a feature layer
    Args:
        res_blocks (int): number of total blocks in the layer (1 full block and res_blocks-1 of residual blocks)
        filters (int): number of filters for the blocks in the layer
        stride (int, optional): stride size. Defaults to 1.
    Returns:
        nn.Sequential: specified layer
    """
    ii_downsample: ResBlock = None
    layers = nn.ModuleList()
    # Check if downsampling is needed
    if stride != 1 or self.in_channels != filters * ResBlock.expansion:
        # Create downsampling layer using 1x1 convolution and BatchNorm
        ii_downsample = nn.Sequential(
            nn.Conv2d(in_channels=self.in_channels, out_channels=(filters * ResBlock.expansion),
                kernel_size=1, stride=stride),
            nn.BatchNorm2d(num_features=(filters * ResBlock.expansion)),
        )
    # Create the first residual block and add it to the layers
    layers.append(ResBlock(self.in_channels, out_channels=filters, i_downsample=ii_downsample,
        stride=stride))
    self.in_channels = filters * ResBlock.expansion  # Update the number of input channels for the remaining blocks
    # Adds the remaining residual blocks to the network
    layers.extend([ResBlock(self.in_channels, filters) for _ in range(res_blocks - 1)])
    return nn.Sequential(*layers)  # Return the sequential module containing all the layers
```

Listing 21 ResNet network forward function.

```
def forward(self, X: torch.Tensor) -> torch.Tensor:
    """Forward pass of the network.
    Args:
        X (torch.Tensor): Input tensor.
    Returns:
        torch.Tensor: Output tensor after passing through the network.
    """
    X = self.input(X)  # Pass the input through the initial layers
    # Pass the input through each layer in the network
    for layer in self.network:
        X: torch.Tensor = layer(X)
    X = self.avgpool(X)  # Apply adaptive average pooling
    X = X.reshape(X.shape[0], -1)  # Reshape the tensor to a 2D shape for the fully connected layer
    # Pass the reshaped tensor through the fully connected layer for classification
    return self.fc(X)
```

desired number of channels. The `num_channels` property retrieves the current number of channels, and the corresponding setter method updates both the `_num_channels` attribute and the input layer with the specified number of channels. This flexibility allows the ResNet network to adapt to different input channel configurations easily.

Additionally, the ResNet network also provides the ability to adjust the number of classes it can classify. The code shown in Listing 23 demonstrates how the `num_classes` property allows retrieval and modification of the current number of classes. When the number of classes is changed, the corresponding setter method updates both the `_num_classes` attribute and the fully connected layer of the network. By doing so, the ResNet network can dynamically adapt to different classification tasks

Listing 22 ResNet network number of channels property.

```
85  @property
86  def num_channels(self) -> int:
87      """Get the number of channels.
88      Returns:
89          int: The number of channels.
90      """
91      return self._num_channels
92
93  @num_channels.setter
94  def num_channels(self, num_channels: int) -> None:
95      """Set the number of channels.
96      Args:
97          num_channels (int): The number of channels.
98      """
99      self._num_channels = num_channels  # Set the number of channels
100     self.input[0] = nn.Conv2d(in_channels=num_channels, out_channels=64, kernel_size=7, stride=2,
            padding=3, bias=False)  # Replace the input layer with the given number of channels
```

with varying numbers of classes.

Listing 23 ResNet network number of classes property.

```
102 @property
103 def num_classes(self) -> int:
104     """Get the number of classes.
105     Returns:
106         int: The number of classes.
107     """
108     return self._num_classes
109
110 @num_classes.setter
111 def num_classes(self, num_classes: int) -> None:
112     """Set the number of classes.
113     Args:
114         num_classes (int): The number of classes.
115     """
116     self._num_classes = num_classes  # Set the number of classes
117     self.fc = nn.Linear(in_features=(512 * ResBlock.expansion), out_features=num_classes)  # Update the
            fully connected layer with the new number of classes
```

The code implemented to regulate the training for this neural network structure is similar to the previously presented for the Segmentation Network in Listings 8, 10 and 11 with slight differences only in the `batch_train()` method. In the case of the Classification Network, it is possible to use a previously trained segmentation model to isolate the foreground from the background for incoming images, thus facilitating the classification process. This is evident in Listing 24 where if the `self.segmenter` attribute is not `None`, indicating that a segmentation model is available, the input data `b_data_x` is passed through the segmentation model. This is done using a with `torch.no_grad()` block, which disables gradient computation for memory efficiency. Additionally, the metrics employed to train the neural network are also different, as seen from the `_calculate_metrics()` method in Listing 25.

The `forward()` function of as seen in Listing 26 the classifier takes an input tensor X and returns the output tensor. The function first moves the input tensor to the specified device and then checks if

Listing 24 Classification Network training using segmentation.

```
if self.segmenter:  # forwards the segmentation model if available
    with torch.no_grad():  # avoid the computation of gradients
        b_data_x = self.segmenter(b_data_x)
```

Listing 25 Classification Network training metrics calculation.

```
def _calculate_metrics(self, preds: torch.Tensor, labels: torch.Tensor) -> list[torch.Tensor]:
    """Calculates various performance metrics based on the predicted values and the ground truth labels.
    Args:
        preds (torch.Tensor): Predicted values.
        labels (torch.Tensor): Ground truth labels.
    Returns:
        list[torch.Tensor]: List of performance metrics, including class accuracy, precision, recall, F1 score, and Matthews correlation coefficient (MCC).
    """
    _conf_matrix: torch.Tensor = LearningSystem.confusion_matrix(preds, labels)  # Compute confusion matrix
    return [
        LearningSystem.class_accuracy(_conf_matrix).mean(),  # Calculate mean class accuracy
        LearningSystem.precision(_conf_matrix).mean(),  # Calculate mean precision
        LearningSystem.recall(_conf_matrix).mean(),  # Calculate mean recall
        LearningSystem.f1_score(_conf_matrix).mean(),  # Calculate mean F1 score
        LearningSystem.mcc(_conf_matrix).mean(),  # Calculate mean Matthews correlation coefficient (MCC)
    ]
```

a segmenter is available. If a segmenter is present, it applies segmentation to the input tensor before passing it through the classifier's model. The function then returns the output tensor obtained from the model.

Listing 26 Classification Network forward method.

```
def forward(self, X: torch.Tensor) -> torch.Tensor:
    """Classification network forward method
    Args:
        X (torch.Tensor): Input tensor
    Returns:
        torch.Tensor: Output tensor
    """
    X = X.to(self.device)  # Move input tensor to the specified device
    # Check if a segmenter is available and apply segmentation
    if self.segmenter:
        X = self.segmenter(X)
    return self.model(X)  # Pass the segmented tensor through the model and return the result
```

It is now possible to define the full Bottom Level structure, following the code in Listing 27. It initialises an instance of `ClassificationNetwork` and `SegmentationNetwork` with the appropriate arguments, such as the number of classes, whether attention is used, the type of network architecture. The `use_segmentation` parameter controls the usage of segmentation. If segmentation is enabled, the `segmenter` instance is assigned to the `segmenter` attribute of the `ClassificationNetwork`. Finally, the module is sent to the specified device using the `to()` method.

Listing 27 Bottom Level __init__function.

```python
class BottomLevel(object):
    def __init__(self, num_classes: int, attention: bool, net_type: str, use_segmentation: bool) -> None:
        """Bottom level interface constructor
        Args:
            num_classes (int): number of classification classes
            attention (bool, optional): if the segmentation will use attention. Defaults to True.
            net_type (str, optional): name of the network. Defaults to "resnet50".
            use_segmentation (bool, optional): if the classifier will use segmentation. Defaults to True.
        """
        # Initialize the classifier network
        self.classifier = ClassificationNetwork(num_classes=num_classes, net_type=net_type)
        self.segmenter = SegmentationNetwork(attention=attention)  # Initialize the segmentation network
        self._use_segmentation = use_segmentation
        if self._use_segmentation:
            # Connect the segmentation network to the classifier if segmentation is enabled
            self.classifier.segmenter = self.segmenter
```

In order to provide a higher level of abstraction in the training process of the bottom level, the `fit()` method is implemented as seen in Listing 28. This method allows for training both the segmentation and classification tasks. The method takes two optional arguments: `segmentation` and `classification`, which are tuples containing the `DataLoader` and the number of epochs for training each task. If the `segmentation` argument is provided, the `segmenter` is trained using the specified data and number of epochs. Similarly, if the `classification` argument is provided, the `classifier` is trained using the specified data and number of epochs. The method also checks if segmentation is enabled and whether the `segmenter` has been previously trained. If segmentation is enabled but no previous training has been found, a warning is issued to alert the user about this mismatch.

Since the Bottom-Level has a direct connection with the Top-Level of the Computation System, the `forward()` method of this module (Listing 29) was adapted to support two types of data structures commonly used at this stage. To achieve this, and as a mechanism to overload functions or methods in Python, the decorator `@singledispatchmethod` was employed to have the same method behave differently depending on the data type received.

`@singledispatchmethod` is a decorator in Python's `functools` module used to define a single-dispatch generic function in a class. It allows you to define multiple methods with the same name in a class, where each method handles a specific type of the first argument. When invoking the method, the appropriate implementation is automatically selected based on the type of the argument. It simplifies the implementation of polymorphic behaviour in a class by dispatching the method based on the argument's type.

The forward method is initially defined to handle a single input tensor `X` and return a tensor after passing it through the segmentation and classification models. However, by using the

Listing 28 Bottom Level fit method.

```
def fit(self, segmentation: Tuple[DataLoader, int] = None, classification: Tuple[DataLoader, int] =
    None):
    """Trains the BottomLevel model. Can be used for both segmentation and classification.
    The training can be done for both segmentation and classification.
    Args:
        segmentation (Tuple[DataLoader, int], optional): segmentation tuple with the DataLoader and the
    number of epochs for training. Defaults to None.
        classification (Tuple[DataLoader, int], optional): classification tuple with the DataLoader and
    the number of epochs for training. Defaults to None.
        use_segmentation (bool, optional): enables if the classification network will use segmentation as
    a pre-processor for it's data. Defaults to True.
    """
    if segmentation:
        segmentation_data, seg_epochs = segmentation
        assert isinstance(segmentation_data, DataLoader) and isinstance(seg_epochs, int)
    if classification:
        classification_data, class_epochs = classification
        assert isinstance(classification_data, DataLoader) and isinstance(class_epochs, int)
    # Perform segmentation training if specified
    if segmentation:
        self.segmenter.fit(segmentation_data, epochs=seg_epochs)
    # Check if segmentation is enabled but no previous training is found
    if self.use_segmentation and self.segmenter.prev_training is None:
        warnings.warn("Segmentation is enabled but no previous training was found.")
    # Perform classification training if specified
    if classification:
        self.classifier.fit(classification_data, epochs=class_epochs)
```

`@forward.register` decorator, an additional implementation of the forward method is defined to handle input of type `DataLoader`. When the `forward()` method is called with an argument of type `DataLoader`, the registered implementation is automatically invoked. Inside this implementation, it iterates over the batches of the `DataLoader`, passes each batch through the segmentation and classification models, and compiles the probabilities and labels. The final output is a tuple containing the concatenated probabilities and labels, to be used by the Top-Level in it's processing.

Listing 29 Bottom Level foward overloaded methods.

```
@singledispatchmethod
def forward(self, X: torch.Tensor) -> torch.Tensor:
    # Check if segmentation is enabled but no previous training was found
    if self.use_segmentation and self.segmenter.prev_training is None:
        warnings.warn("Segmentation is enabled but no previous training was found.")
    # Check if classifier training is available
    if self.classifier.prev_training is None:
        warnings.warn("No Classifier training is available.")
    X = self.classifier(X)  # Apply classification to the input tensor
    X = torch.softmax(X, dim=1)  # Convert output scores to probabilities using softmax
    return X  # Return the processed tensor

@forward.register
def _(self, X: DataLoader) -> Tuple[torch.Tensor, torch.Tensor]:
    # Iterate over batches and compile probabilities
    X_probs, y_labels = zip(*[(self(X_data), y_true) for X_data, y_true in tqdm(X, desc="Compiling
        probabilities...")])
    # Concatenate the collected probabilities and labels
    X_probs = torch.cat(X_probs, dim=0)
    y_labels = torch.cat(y_labels, dim=0)
    return X_probs, y_labels  # Return the compiled probabilities and labels
```

4.1.2 Top-Level Subsystem

This subsection provides a detailed exploration of the implementation of Decision Forests and Decision Trees. Decision Forests, regarded as an ensemble learning technique, employ the collective intelligence of multiple Decision Trees to achieve precise predictions. The induction mechanism of these algorithms relies on the probabilities derived from the bottom-level, resulting in an enhanced overall predictive performance. By examining the code and underlying principles, valuable insights into the intricate workings of Decision Forests and Decision Trees can be gained, elucidating their effectiveness in addressing complex learning tasks through collaborative decision-making.

Starting with the Decision Forest implementation, the code in Listing 30 showcases the `__init__()` function of the `DecisionForest` class. This function is responsible for creating an instance of the Decision Forest. It takes two optional parameters: `min_samples_split`, which specifies the minimum number of samples required for a split during tree building, and `max_depth`, which defines the maximum depth that the Decision Trees within the forest can reach. Setting `max_depth` to None implies no depth limit.

Listing 30 Decision Forest __init__function.

```
7  class DecisionForest:
8      def __init__(self, min_samples_split: int, max_depth: int) -> None:
9          """Create a DecisionForest, the parameters are used in the decision tree building
10         Args:
11             min_samples_split (int, optional): Minumum number of samples per split. Defaults to 10.
12             max_depth (int, optional): maximum depth the Decision Trees can reach, None mean no cap.
       Defaults to None.
13         """
14         self.max_depth = max_depth  # maximum depth the Decision Trees can reach
15         self.min_samples_split = min_samples_split  # minimuym number of samples to split
16         self.dts: list[DecisionTree] = []  # list to hold the decision trees
```

Furthermore, the generation of the Decision Forest using Decision Trees can be observed in the `fit()` method, as seen in Listing 31. This method takes as input the feature matrix `X_data` and the corresponding target vector `y_data`, ensuring that their shapes are compatible. To parallelise the process, a multiprocessing pool is then created using `mp.Pool(num_processes)`, where `num_processes` represents the number of CPU cores, determined using the `mp.cpu_count()` function from the `multiprocessing` module. The map function is used to distribute the workload across multiple processes. The auxiliary method `_create_dt()` defined in Listing 32 is invoked within the map function, passing a set of arguments for each Decision Tree in the forest. These arguments consist of the index, the feature matrix, a specific column of the target vector, the `min_samples_split`, and the `max_depth`.

Listing 31 Decision Forest fit function.

```
def fit(self, X_data: np.ndarray, y_data: np.ndarray):
    """Fit method for the decision forest, to create decision trees
    Args:
        X_data (np.ndarray), y_data (np.ndarray)
    """
    assert X_data.shape == y_data.shape  # verifies that the data has the same number of instances and
        features
    num_processes = mp.cpu_count()  # number of processes the cpu is capable of running simultaneously
    pool = mp.Pool(num_processes)
    self.dts = pool.map(
        self._create_dt,
        [(i, X_data, column, self.min_samples_split, self.max_depth) for i, column in
            enumerate(y_data.T)],
    )  # splits the data into columns, each column generating a process to create a decision tree
    pool.close()  # indicates that no more tasks will be added to the pool
    pool.join()  # waits for all the processes in the pool to finish their execution
```

Listing 32 Decision Forest auxiliary tree creation function.

```
@staticmethod
def _create_dt(args: Tuple[np.ndarray, np.ndarray, int, int]) -> DecisionTree:
    """Auxiliary function to create a new decision tree
    Args:
        args (Tuple[np.ndarray, np.ndarray, int, int]): Collection of arguments
    Returns:
        DecisionTree: generated decision tree
    """
    i, X, column, min_samples_split, max_depth = args  # retrieves the arguments
    print(f"Building DecisionTree on class {i}")
    # Creates a DecisionTree instance from the arguments
    dt = DecisionTree(min_samples_split=min_samples_split, max_depth=max_depth)
    dt.fit(X, column)  # Fit the decision tree
    return dt  # Returns the constructed decision tree
```

The `_create_dt()` is a static method decorated with the `@staticmethod` decorator. The static method is advantageous in this scenario as it can be executed independently within a parallelised context without relying on the instance or any shared state.

Once all the Decision Trees have been called to generate, the pool of processes is closed using `pool.close()`, and the program waits for all the processes to complete using `pool.join()`. The resulting Decision Trees are stored in the `self.dts` attribute of the Decision Forest. This mechanism allows for efficient and parallelised construction of the Decision Forest, leveraging the collective intelligence of multiple Decision Trees to achieve accurate predictions.

Similarly, the prediction process in the Decision Forest is parallelised using multiple processes as seen in Listing 33. In the call method, a multiprocessing pool is created to distribute the prediction task among the trained Decision Trees. The `_predict_dt()` method in Listing 34 is called for each Decision Tree, where it performs the prediction using the feature matrix. The results are collected and returned as a `numpy` array, providing efficient and collective predictions from the Decision Forest.

Before moving to the implementation of the Decision Tree class, it is essential to understand the

Listing 33 Decision Forest parallelised prediction.

```
def __call__(self, X: np.ndarray) -> np.ndarray:
    """Forward method for the decision forest
    Args:
        X (np.ndarray): feature array
    Returns:
        np.ndarray: array of predicted labels
    """
    num_processes = mp.cpu_count()
    pool = mp.Pool(num_processes)
    preds = pool.map(self._predict_dt, [(dt, X) for dt in self.dts])  # predicts for each of the decision
        tree
    pool.close()  # waits for the process to finish
    pool.join()
    return np.array(preds).T  # returns the predictions array, in a list wise shape
```

Listing 34 Decision Forest prediction function for each Decision Tree.

```
@staticmethod
def _predict_dt(args: Tuple[DecisionTree, np.ndarray]) -> np.ndarray:
    """Auxiliary function to predict from a decision tree
    Args:
        args (Tuple[np.ndarray, np.ndarray, int, int]): Collection of arguments
    Returns:
        np.ndarray: predicted from the decision tree
    """
    dt, X = args  # gets the decision tree argument and the feature array
    return dt(X)  # call predict, same as forward methods for neural networks
```

fundamental building blocks that comprise the structure of the tree. These building blocks, known as nodes, play a crucial role in the decision-making process of the tree. The `DecisionNode` presented in Listing 35 represents an internal node in the tree. It contains crucial information such as the splitting criteria, which determines how the tree branches out at that node. Additionally, the `DecisionNode` holds references to its left and right child nodes, indicating the subsequent branches in the tree. Each `DecisionNode` also keeps track of the number of samples associated with it, providing insights into the data distribution at that specific point in the tree.

Listing 35 Decision Node for Decision Tree building.

```
class DecisionNode:
    _id: int = 1

    def __init__(self, split_data: SplitNode, left: Union[DecisionNode, LeafNode], right:
        Union[DecisionNode, LeafNode], num_samples: int) -> None:
        self.split_data = split_data
        self.left = left
        self.right = right
        self.num_samples = num_samples
        self.id: int = DecisionNode._id
        DecisionNode._id += 1
```

On the other hand, the `LeafNode` in Listing 36 serves as a terminal or leaf in the Decision Tree. It signifies the endpoint of a particular branch and carries a label assigned to it. The label in the LeafNode

represents the predicted class or outcome associated with the samples that reach that leaf. Similar to the `DecisionNode`, the `LeafNode` also stores the number of samples it represents, offering valuable information about the data distribution at the leaf level.

Listing 36 Leaf Node for Decision Tree building.

```
17 class LeafNode:
18     _id: int = 1
19
20     def __init__(self, label: int, num_samples: int) -> None:
21         self.label = label
22         self.num_samples = num_samples
23         self.id: int = LeafNode._id
24         LeafNode._id += 1
```

Additionally, the `SplitNode` (Listing 37) is a specialised data structure used to store the information related to a split in the Decision Tree. It is implemented as a `TypedDict`, a dictionary subclass introduced in Python that provides type hints for the keys and their associated values. The `SplitNode` includes the feature index, indicating the feature used for the split, the threshold value representing the decision boundary, and the metric gain, which quantifies the improvement achieved by the split. The usage of a `TypedDict` in this context ensures type safety and provides clarity about the expected structure of the split data.

Listing 37 Split Node with the best split at a certain stage of the Decision Tree building.

```
27 class SplitNode(TypedDict):
28     feature_index: int
29     threshold: float
30     metric_gain: float
```

Having defined the nodes that the tree will encompass, the decision tree's construction process can now be explained. It begins with the initialisation method, `__init__()` in Listing 38, of the `DecisionTree` class. This method sets up various attributes of the decision tree. The `root` attribute represents the tree's root node, initially set to `None` as no tree has been constructed yet. The `lowest_depth` attribute tracks the lowest depth achieved by the tree during construction.

The stopping conditions of the decision tree construction are also specified. The `min_samples_split` parameter determines the minimum number of samples to split at a node. If the number of samples is below this threshold, further splitting is halted, and a leaf node is created. The `max_depth` parameter defines the maximum depth that the decision tree can reach. If the depth exceeds this limit, the construction process stops, and leaf nodes are created. Additionally, the `METRIC` dictionary is defined, which associates different metrics for evaluating the quality of splits.

Listing 38 Decision Tree __init__function.

```
56 class DecisionTree:
57     METRIC = {"info_gain": info_gain, "info_gain_ratio": info_gain_ratio}
58
59     def __init__(self, min_samples_split: int, max_depth: int) -> None:
60         """Initializes a decision tree with the specified parameters.
61         Args:
62             min_samples_split (int): The minimum number of samples required to split a node.
63             max_depth (int): The maximum depth of the decision tree.
64         """
65         self.root: Union[LeafNode, DecisionNode] = None  # root node of the tree, initialy a None since no
               tree was built yet
66         self.lowest_depth = 0  # lowest depth achieved by the tree during the construction process
67         # Stopping conditions
68         self.min_samples_split = min_samples_split
69         self.max_depth = max_depth
```

These metrics, namely information gain and information gain ratio, serve as evaluators for assessing the performance of splits in the decision tree. They rely on the concept of entropy, which plays a fundamental role in quantifying the impurity or disorder in a set of labels. Entropy can be defined as the measure of uncertainty or randomness in the distribution of labels within a subset.

To calculate the entropy of a given subset of labels, the `entropy()` function is utilised. This function takes in a *numpy* array of labels and performs the following steps: First, it calculates the total count of each class by utilising the `np.unique` function. Then, it computes the probability of each class by dividing the class counts by the total number of labels. Finally, the entropy is calculated by taking the negative sum of the element-wise product of the normalised class probabilities and their logarithms base 2. This process is seen in the code segment in Listing 39.

Listing 39 Entropy metric calculation.

```
7  def entropy(labels: np.ndarray):
8      """Calculate the entropy for a given subset of labels
9      Args:
10         labels (np.ndarray): subset of labels
11     Returns:
12         np.ndarray: entropy of the given labels
13     """
14     _, counts = np.unique(labels, return_counts=True)  # calculate total count of each class
15     norm_counts = counts / len(labels)  # calculate the probability of each class
16     return -np.sum(norm_counts * np.log2(norm_counts))  # calculate the probability of each class
```

The `info_gain()` function in Listing 40 takes three numpy arrays as input: the parent label array (`labels`), the labels corresponding to the left branch obtained from thresholding a feature (`left_labels`), and the labels corresponding to the right branch obtained from the same thresholding (`right_labels`). This threshold process is better described ahead when explaining the splitting function during the decision tree construction.

To calculate the information gain, the function first computes the entropy of the parent labels by

invoking the `entropy()` function. It then subtracts from the parent entropy the weighted sum of the entropies of the left and right branches. The weights are determined by the ratios of the number of samples in each branch to the total number of samples in the parent.

The resulting information gain reflects the reduction in entropy achieved by the split. A higher information gain indicates that the split effectively separates the data into more homogeneous subsets, which is desirable for building an accurate decision tree model.

Listing 40 Information Gain calculation.

```
19 def info_gain(labels: np.ndarray, left_labels: np.ndarray, right_labels: np.ndarray) -> float:
20     """Info gain for splitting the labels into two branches
21     Args:
22         labels (np.ndarray): parent label array
23         left_labels (np.ndarray): left generated branch from thresholding a feature
24         right_labels (np.ndarray): right generated branch from thresholding a feature
25     Returns:
26         float: information gain for the split
27     """
28     return entropy(labels) - ((len(left_labels) / len(labels)) * entropy(left_labels)) -
           ((len(right_labels) / len(labels)) * entropy(right_labels))
```

The process of calculating the information gain ratio, as discussed earlier, is implemented in the code presented in Listing 41. This code snippet defines the `info_gain_ratio()` function, which takes the same input parameters as the `info_gain()` function: the parent label array (`labels`), the labels corresponding to the left branch obtained from thresholding a feature (`left_labels`), and the labels corresponding to the right branch obtained from the same thresholding (`right_labels`).

Listing 41 Information Gain Ratio calculation.

```
44 def info_gain_ratio(labels: np.ndarray, left_labels: np.ndarray, right_labels: np.ndarray) -> float:
45     """Information Gain ratio for splitting the labels into two branches
46     Args:
47         labels (np.ndarray): parent label array
48         left_labels (np.ndarray): left generated branch from thresholding a feature
49         right_labels (np.ndarray): right generated branch from thresholding a feature
50     Returns:
51         float: information gain ratio
52     """
53     return abs(np.nan_to_num(info_gain(labels, left_labels, right_labels) / split_info(labels,
           left_labels, right_labels)))
```

Inside the `info_gain_ratio()` function, the information gain is computed by invoking the `info_gain()` function, and the result is divided by the split information, which is calculated using the `split_info()` function defined in Listing 42.

The `split_info()` function calculates the split information for a given split. It takes the same input parameters as the `info_gain()` and `info_gain_ratio()` functions. To avoid numerical instability, a small constant (`epsilon`) is added to the denominator when calculating the logarithms. The split

Listing 42 Split Info calculation.

```
def split_info(labels: np.ndarray, left_labels: np.ndarray, right_labels: np.ndarray) -> float:
    """Split infotmation for splitting the labels into two branches
    Args:
        labels (np.ndarray): parent label array
        left_labels (np.ndarray): left generated branch from thresholding a feature
        right_labels (np.ndarray): right generated branch from thresholding a feature
    Returns:
        float: split information
    """
    epsilon = 1e-8
    return -((len(left_labels) / len(labels)) * np.log2(len(left_labels) / len(labels) + epsilon)) +
        ((len(right_labels) / len(labels)) * np.log2(len(right_labels) / len(labels) + epsilon))
```

information is obtained by applying the formula that combines the proportions of samples in the left and right branches.

The information gain ratio is computed as the absolute value of the information gain divided by the split information. This ratio provides a normalised measure that accounts for the inherent bias of the information gain towards splits with many outcomes.

The specific building process of the decision tree starts in the `fit()` method, as seen in Listing 43, where the recursive function `_build_tree()` is called. This method takes two parameters: `X`, representing the feature values of the dataset, and `y`, corresponding to the labels. Within the `fit()` method, the number of features (`n_features`) is determined from the shape of the feature matrix `X`, and the number of classes (`n_classes`) is obtained by finding the number of unique labels in the target vector `y`. The root of the decision tree is then set to the result of calling `_build_tree()` with `X` and `y` as arguments. This recursive function handles the splitting and node creation process, ultimately constructing the complete decision tree.

Listing 43 Decision Tree fit function.

```
def fit(self, X: np.ndarray, y: np.ndarray) -> None:
    """Fits the decision tree to the given data set
    Args:
        X (np.ndarray): X values of the dataset
        y (np.ndarray): y values of the dataset (labels)
    """
    self.n_features = X.shape[1]  # number of features in the dataset
    self.n_classes = len(np.unique(y))  # number of unique classes in the dataset
    self.root = self._build_tree(X, y)  # builds the decision tree recursively using the dataset
```

The `_build_tree()` method is a recursive function responsible for constructing the decision tree, presented in Listing 44. It takes three parameters: `X`, representing a subset of features, `y`, corresponding to a subset of labels, and `depth`, indicating the current depth of the tree, initially set to 0. The function also asserts that the number of rows in `X` matches the number of elements in `y`. The `lowest_depth` attribute of the decision tree is updated to the maximum value between the current depth and the existing

lowest depth. The `num_samples` variable is assigned the length of the label subset `y`, representing the number of samples in the current split.

Listing 44 Decision Tree building function.

```
81 def _build_tree(self, X: np.ndarray, y: np.ndarray, depth: int = 0) -> Union[DecisionNode, LeafNode]:
82     """Recursively build a decision tree
83     Args:
84         X (np.ndarray): subset of features
85         y (np.ndarray): subset of labels
86         depth (int, optional): current depth of the tree. Defaults to 0.
87     Returns:
88         Union[DecisionNode, LeafNode]: either a DecisionNode or a LeafNode, depending on the outcome
89     """
90     assert X.shape[0] == y.shape[0]  # verifies that the number of instances in X and y are equal
91     self.lowest_depth = max(depth, self.lowest_depth)  # updates the lowest_depth attribute with the
           maximum value between depth and the current lowest_depth
92     num_samples = len(y)  # calculates the number of samples in the dataset
```

Then, several conditions are tested to determine if the splitting process should end and a leaf node should return. These conditions include reaching the maximum depth specified by `max_depth`, the subset of data being too small to be split according to `min_samples_split`, or all the labels in the subset `y` being the same. In such cases, a `LeafNode` is created, with the label being the most common in `y` and using the total number of samples for the current split.

Listing 45 Decision Tree building: stopping conditions verification.

```
93     # return a leaf when:
94     # - the max deapth is reached
95     # - the current data is too short to be split
96     # - every label on the data is the same
97     if (self.max_depth and depth >= self.max_depth) or (num_samples <= self.min_samples_split) or
           (np.all(y == y[0])):
98         return LeafNode(np.bincount(y).argmax(), num_samples)
```

When none of these criterion are met, it is necessary to evaluate the best splitting conditions for the current data. For this, the function `_get_best_split()` evaluates which are the threshold and feature values for splitting as seen in *99*. If the `metric_gain` from the best split is greater than 0, indicating sufficient improvement from splitting the data, the function recursively calls `_build_tree()` to construct the left and right subtrees. The left subtree is built using the subset of `X` and `y` where the values are less than or equal to the split threshold, while the right subtree is built using the subset where the values are greater than the split threshold. The depth parameter is incremented by 1 in each recursive call.

Then, before creating the subsequent `DecisionNode` using the split data as seen in *Line 115*, another evaluation is made, verifying that the Nodes created from the previous recursive execution are not representing the same label. If this is the case, these two nodes that would originate a `DecisionNode` are replaced by a single `LeafNode` representing the matching label.

Listing 46 Decision Tree building: recursive call for branches.

```
    best_split: SplitNode = self._get_best_split(X, y)
    if best_split["metric_gain"] > 0:  # if enough improvement from splitting the data
        left_subtree = self._build_tree(
            X=X[X[:, best_split["feature"]] <= best_split["threshold"]],
            y=y[X[:, best_split["feature"]] <= best_split["threshold"]],
            depth=depth + 1,
        )  # build left subtree
        right_subtree = self._build_tree(
            X=X[X[:, best_split["feature"]] > best_split["threshold"]],
            y=y[X[:, best_split["feature"]] > best_split["threshold"]],
            depth=depth + 1,
        )  # build right subtree
```

Listing 47 Decision Tree building: same labeled branches verification and Decision Node creation.

```
        if isinstance(left_subtree, LeafNode) and isinstance(right_subtree, LeafNode):
            if left_subtree.label == right_subtree.label:
                return LeafNode(np.bincount(y).argmax(), num_samples)
        return DecisionNode(best_split, left_subtree, right_subtree, num_samples)
```

The `_get_best_split()` function is responsible for computing the best split for the given labels. It takes three parameters: `X`, representing the feature array, `y`, corresponding to the label array, and metric, which specifies the metric to be used for calculating the best split. The code implementation can be seen in Listing 48.

Listing 48 Decision Tree best split finding.

```
def _get_best_split(self, X: np.ndarray, y: np.ndarray, metric: str = "info_gain") -> SplitNode:
    """Computes the best split for the given labels
    Args:
        X (np.ndarray): feature array
        y (np.ndarray): label array
        metric (str, optional): metric to be used to calculate the best split ("info_gain",
    "info_gain_ratio"). Defaults to "info_gain".
    Returns:
        SplitNode: object with the data for selecting the best split
    """
    best_split = SplitNode(metric_gain=-float("inf"))
    for feature in range(self.n_features):  # for each feature
        for threshold in np.unique(X[:, feature]):  # for each unique feature value
            left_y = y[X[:, feature] <= threshold]  # left subtree with features bellow or equal to
                threshold
            right_y = y[X[:, feature] > threshold]  # right subtree with features above threshold
            if len(left_y) == 0 or len(right_y) == 0:  # if one of the branches is empty is invalid
                continue
            with ignore_warning(RuntimeWarning):  # throws a warning when a branch is pure
                mg = self.METRIC[metric](y, left_y, right_y)
            if mg > best_split["metric_gain"]:  # if the metric for the split is better, replace the best
                split
                best_split["metric_gain"] = mg
                best_split["threshold"] = threshold
                best_split["feature_index"] = feature
    return best_split
```

The function initialises `best_split` as a `SplitNode` object with an initial `metric_gain` value of negative infinity (`-float("inf")`). This object will store the data related to the best split found. The

function then iterates over each feature, using the `range(self.n_features)` loop. For each feature, it iterates over the unique values in that feature using `np.unique(X[:, feature])`. These unique feature values will serve as potential split thresholds. For each unique threshold value, the function splits the label array `y` into the left subset (`left_y`) and the right subset (`right_y`) based on whether the corresponding feature values in `X` are less than or equal to the threshold or greater than the threshold, respectively. The function checks if either the left or right subset is empty (`len(left_y) == 0 or len(right_y) == 0`). If either of the branches is empty, it continues to the next iteration, as an empty branch is not a valid split.

Inside the `ignore_warning(RuntimeWarning)` context manager, the function calculates the metric for the split, using the specified metric and the `self.METRIC[metric]` function. This metric function is either `info_gain()` or `info_gain_ratio()`, described earlier, depending on the chosen metric. The `ignore_warning()` context manager suppresses a warning that may be raised when one of the branches is pure (contains only samples of a single class). This happens in situations where calculating the metric for a pure branch may involve dividing by zero. The calculated metric gain (`mg`) is then compared to the `metric_gain` of the current best split. If the metric gain is greater, indicating a better split, the `best_split` object is updated with the new `metric_gain`, `threshold`, and `feature`. After evaluating all possible splits for each feature and threshold, the function returns the `best_split` object, which contains the data for selecting the best split, including the feature index, threshold value, and metric gain.

The `TopLevel` class seen in Listing 49 serves as an interface to the Decision Tree/Forest processing, encapsulating the functionality of these sub-modules. It provides a simplified way to utilise the Decision Forest by providing methods for initialising, fitting, and calling the forest. It is worth mentioning that the `TopLevel` class does not significantly contribute to the program's functionalities. However, as a measure of conformity and organisation, it was included in the codebase.

Listing 49 Top Level class methods.

```
5  class TopLevel(object):
6      def __init__(self, min_samples_split: int, max_depth: int) -> None:
7          # Initializes a DecisionForest instance with the provided min_samples_split and max_depth
8          self.forest = DecisionForest(min_samples_split=min_samples_split, max_depth=max_depth)
9
10     def fit(self, X: np.ndarray, y: np.ndarray) -> None:
11         # Fits the DecisionForest to the given dataset X and labels y
12         self.forest.fit(X, y)
13
14     def __call__(self, X: np.ndarray) -> np.ndarray:
15         # Calls the DecisionForest on the input data X and returns the predicted labels
16         return self.forest(X)
```

In concluding the Computation System, the final interface for working with this module can be introduced. The `ComputationSystem` class in Listing 50 serves as the main interface, encompassing the functionality of both the `BottomLevel` and `TopLevel` classes. Upon initialisation, it takes the number of classes as a parameter and creates instances of the `BottomLevel` and `TopLevel` classes.

Listing 50 Computation System __init__function.

```
9  class ComputationSystem(object):
10     def __init__(self, num_classes: int, attention: bool, net_type: str, use_segmentation: bool,
           min_samples_split: int, max_depth: int) -> None:
11         """Initializes the ComputationSystem.
12         Args:
13             num_classes (int): Number of classes for classification.
14             attention (bool): Flag indicating whether attention mechanism is enabled.
15             net_type (str): Type of network architecture.
16             use_segmentation (bool): Flag indicating whether segmentation is used as a pre-processor for
       classification.
17             min_samples_split (int): Minimum number of samples required to split a node in the decision
       tree.
18             max_depth (int): Maximum depth of the decision tree.
19         """
20         # Create the BottomLevel and TopLevel components
21         self._bl = BottomLevel(num_classes, attention, net_type, use_segmentation)
22         self._tl = TopLevel(min_samples_split, max_depth)
```

The fit method within the `ComputationSystem` class (shown in Listing 51) is responsible for training the entire system using the provided segmentation and classification datasets. It begins by setting the `BottomLevel` component to training mode using the `train()` method. Next, it calls the `fit()` method of the `BottomLevel` instance, passing the segmentation and classification datasets. This step trains the `BottomLevel` using the given datasets.

Listing 51 Computation System fit method.

```
24 def fit(self, segmentation: Tuple[DataLoader, int] = None, classification: Tuple[DataLoader, int] = None)
       -> None:
25     """Fits the ComputationSystem to the given datasets.
26     Args:
27         segmentation (Tuple[DataLoader, int], optional): Segmentation dataset and number of epochs for
       training. Defaults to None.
28         classification (Tuple[DataLoader, int], optional): Classification dataset and number of epochs for
       training. Defaults to None.
29     """
30     self._bl.train(True)  # sets the bottom level to training mode
31     self._bl.fit(segmentation, classification)  # fits the bottom level to the datasets given
32     with torch.no_grad():  # no need to compute gradients
33         X, y = self._bl(classification[0])  # forwards the bottom-level with the training data
34     X = X.cpu().numpy()  # converts the tensors to arrays
35     y = y.cpu().numpy().astype(int)
36     self._tl.fit(X, y)  # fits the top-level with the training data from the bottom-level
37     self._bl.train(False)  # sets the bottom level to inference mode
```

To obtain the training data from the `BottomLevel`, the method performs a forward pass by invoking the `BottomLevel` instance with the classification data. This operation returns the predictions and labels

as tensors, which are then converted to NumPy arrays using the `cpu().numpy()` method. These arrays are assigned to the variables `X` and `y`, respectively. Moving on to the `TopLevel`, the method calls its `fit()` method, passing the training data obtained from the `BottomLevel`. This step trains the `TopLevel` using the training data, finally setting the bottom-level back to inference mode.

The `__call__()` method in the `ComputationSystem` class enables the system to generate predictions for input data `X`, as seen in Listing 52. Within this method, the bottom-level component is invoked with `X`, executing a forward pass and producing a tensor probability that represents the predicted probabilities for each class. By using the `torch.no_grad()` context manager, gradient computation is disabled during this forward pass. The probability tensor is then converted to a list, capturing the probabilities for each class. Subsequently, this list is passed to the `TopLevel` instance, which computes and determines the final predicted labels. The resulting labels are converted to a list, and the method returns this list as the prediction output for the given input data.

Listing 52 Computation System forward method.

```
39 def __call__(self, X: torch.Tensor) -> Tuple[List[int], List[int]]:
40     """Performs the inference on the given input.
41     Args:
42         X (torch.Tensor): Input tensor.
43     Returns:
44         Tuple[List[int], List[int]]: Tuple containing the probabilities and labels.
45     """
46     with torch.no_grad():  # no need to compute gradients
47         probability: torch.Tensor = self._bl(X)  # forwards the bottom-level
48     probability = probability.cpu().tolist()[0]  # converts the probability tensor to a list
49     labels: np.ndarray = self._tl(np.array(probability))  # forwards the probability in the top-level
50     labels = labels.tolist()[0]  # converts the labels numpy array to a list
51     return (probability, labels)
```

4.2 Learning System

The Learning System's module provides insights and mechanisms for better training the Bottom Level's neural networks. It offers a range of optimisation algorithms, such as gradient descent variants (e.g., Adam, SGD), which adaptively adjust the network's weights to minimise the training loss. Additionally, the Learning System includes learning rate schedulers, which dynamically modify the learning rate during training to ensure optimal convergence. These schedulers adjust the learning rate based on predefined rules, such as reducing it gradually or responding to specific conditions, allowing for more effective training. Furthermore, the Learning System incorporates various evaluation metrics to assess the network's performance, such as accuracy, precision, recall, and F1-score. These metrics provide quantitative measures of the network's predictive capabilities and help gauge its effectiveness in

handling different tasks and datasets. By leveraging these optimisers, schedulers, and metrics, the Learning System enables users to fine-tune the Bottom Level's neural networks and achieve improved training outcomes.

This code snippet as in Listing 53 defines a class called `LearningSystem` that represents the learning system component. Upon initialisation, it sets up the learning configurations for a classifier and a segmenter. The learning configurations are stored in dictionaries `self.classifier_learn` and `self.segmenter_learn`.

An SGD optimiser is created for the classifier with a learning rate of 1e-4 and momentum of 0.9. The optimiser is assigned to `classifier.optim` and also stored in `self.classifier_learn["optimiser"]`. A step-based learning rate scheduler is created with a step size of 4 and a decay factor of 0.1. The scheduler is assigned to `classifier.scheduler` and stored in `self.classifier_learn["scheduler"]`.

An Adam optimiser is created for the segmenter with a learning rate of 1e-4. Similarly to the classifier, the optimiser is assigned to `segmenter.optim` and stored in `self.segmenter_learn["optimiser"]`. A learning rate scheduler based on plateaus is created using `ReduceLROnPlateau`, with the mode set to "min" (indicating reduction on loss plateau) and a patience of 2. The scheduler is assigned to `segmenter.scheduler` and stored in `self.segmenter_learn["scheduler"]`.

Listing 53 Learning System __init__function.

```
20 class LearningSystem(object):
21     def __init__(self, classifier: ClassificationNetwork, segmenter: SegmentationNetwork) -> None:
22         """Initializes a LearningSystem object with a classifier and segmenter.
23         Args:
24             classifier (ClassificationNetwork): The classifier network.
25             segmenter (SegmentationNetwork): The segmenter network.
26         """
27         self.classifier_learn = {}  # Dictionary to store classifier learning-related information
28         self.segmenter_learn = {}  # Dictionary to store segmenter learning-related information
29         # Configures optimizer and scheduler for the classifier
30         classifier.optim = self.classifier_learn["optimiser"] =
               torch.optim.SGD(classifier.model.parameters(), lr=1e-4, momentum=0.9)
31         classifier.scheduler = self.classifier_learn["scheduler"] =
               torch.optim.lr_scheduler.StepLR(self.classifier_learn["optimiser"], step_size=4, gamma=0.1)
32         # Configures optimizer and scheduler for the segmenter
33         segmenter.optim = self.segmenter_learn["optimiser"] =
               torch.optim.Adam(segmenter.model.parameters(), lr=1e-4)
34         segmenter.scheduler = self.segmenter_learn["scheduler"] =
               torch.optim.lr_scheduler.ReduceLROnPlateau(self.segmenter_learn["optimiser"], "min",
               patience=2)
```

To avoid recursive imports in the Learning System module, the code snippet in 54 checks if the `TYPE_CHECKING` constant is **`True`**. The `TYPE_CHECKING` constant is a special constant provided by the typing module that is only **`True`** during static type checking (e.g., using tools like *mypy*) but **`False`**

during run-time. Thus, when the code is not in run-time, the modules are imported, and only work to provide the coder with type checking capabilities from the IDE.

Listing 54 Learning System recursive imports avoidance.

```
5  from typing import TYPE_CHECKING
6
7  if TYPE_CHECKING:
8      from cs.bl.classification import ClassificationNetwork
9      from cs.bl.segmentation import SegmentationNetwork
```

4.2.1 Training Metrics

This section will present the implementation of metrics, focusing on their role in evaluating the performance of classification and segmentation tasks. These metrics can be divided into two categories: classification metrics and segmentation metrics. Classification metrics assess the accuracy and quality of classification models by measuring precision, recall, and F1-score, as seen in Listing 25. Segmentation metrics, conversely, evaluate the accuracy and consistency of segmentation models using metrics such as intersection over union (IoU), Dice coefficient, and pixel-wise accuracy, evident in Listing 13. By examining the implementation of these metrics within the system, valuable insights can be gained regarding the performance of the models, and informed decisions can be made regarding their training and optimisation.

Regarding the segmentation metrics, the first metric presented is the `mask_accuracy()` function in Listing 55. This function calculates the accuracy of the given predictions and masks. It takes two 2D tensors as inputs: `preds` representing the predicted masks and `masks` representing the true masks. The function computes the element-wise comparison between the predictions and masks and calculates the sum of the correct predictions. This sum is then divided by the total number of elements in the predictions tensor using the `torch.numel()` function. The result is a one-element tensor representing the mask accuracy.

Listing 55 Mask Accuracy metric.

```
132 def mask_accuracy(preds: torch.Tensor, masks: torch.Tensor) -> torch.Tensor:
133     """Calculates the accuracy of the given predictions and masks.
134     Args:
135         preds (torch.Tensor): 2D tensor with the mask predictions.
136         masks (torch.Tensor): 2D tensor with the true masks.
137     Returns:
138         torch.Tensor: One element tensor with the mask accuracy.
139     """
140     return (preds == masks).sum() / torch.numel(preds)
```

Next, the `dice_score()` function is available in Listing 56, serving the purpose of calculating the Dice Score for a given set of predictions and masks. This function expects two 2D tensors, namely `preds` for the predicted masks and `masks` for the true masks. The computation involves multiplying the element-wise product of the predictions and masks by 2, followed by summing up the results. This sum is then divided by the sum of the predictions and masks, adding a small epsilon value to avoid potential division by zero, defined in Listing 57.

Listing 56 Dice Score metric.

```
143 def dice_score(preds: torch.Tensor, masks: torch.Tensor) -> torch.Tensor:
144     """Calculates the Dice Score of the given predictions and masks.
145     Args:
146         preds (torch.Tensor): 2D tensor with the mask predictions.
147         masks (torch.Tensor): 2D tensor with the true masks.
148     Returns:
149         torch.Tensor: One element tensor with the Dice Score.
150     """
151     return (2 * (preds * masks).sum()) / ((preds + masks).sum() + EPS)
```

Listing 57 Epsilon value definition.

```
11 EPS = 1e-8  # small value to avoid division by zero
```

Finally, the `iou_score()` function is provided in Listing 58 to compute the Intersection over Union (IoU) Score for a given set of predictions and masks. The computation involves calculating the sum of the element-wise product of the predictions and masks. This sum is divided by the difference between the sum of the predictions and masks and the sum of the element-wise product of the predictions and masks, with the addition of a small epsilon value to prevent division by zero.

Listing 58 IoU metric.

```
154 def iou_score(preds: torch.Tensor, masks: torch.Tensor) -> torch.Tensor:
155     """Calculates the IoU Score of the given predictions and masks.
156     Args:
157         preds (torch.Tensor): 2D tensor with the mask predictions.
158         masks (torch.Tensor): 2D tensor with the true masks.
159     Returns:
160         torch.Tensor: One element tensor with the IoU Score.
161     """
162     return (preds * masks).sum() / ((preds + masks).sum() - ((preds * masks).sum()) + EPS)
```

The `confusion_matrix()` function as seen in Listing 59 is employed in classification metrics to establish the necessary framework for metric calculations. By taking the predicted and true labels as 1D tensors, this function generates a 2D tensor called `confusion` with a shape of (2, 2), specifically designed for binary classification scenarios. The function examines pairs of predictions and targets through

an iterative process, updating the corresponding coordinates in the `confusion` tensor based on the relationship between the prediction and the true label. In cases where the prediction aligns with the true label, the diagonal elements are incremented, indicating correct predictions. Conversely, discrepancies between the prediction and the true label signify instances of false positives or false negatives. Finally, the resulting `confusion` tensor, capturing the distribution of predictions and targets, is returned as the outcome of the function.

Listing 59 Confusion Matrix computation.

```
37 def confusion_matrix(preds: torch.Tensor, targets: torch.Tensor) -> torch.Tensor:
38     """Calculates the confusion matrix for the given predictions and targets.
39     Args:
40         preds (torch.Tensor): 1D tensor of predicted labels.
41         targets (torch.Tensor): 1D tensor of true labels.
42     Returns:
43         torch.Tensor: 2D tensor with the confusion matrix.
44     """
45     # Creates a tensor with the shape (2, 2) for binary classification
46     confusion = torch.zeros(2, 2, dtype=int)
47     # Iterates over the prediction/target pairs and increments to the coordinates of these pairs
48     # When the prediction is the same as the true label, the value is incremented in the diagonal meaning
           that the prediction is correct
49     # The difference between the prediction and the true label means either a false positive or a false
           negative.
50     preds = preds.flatten()
51     targets = targets.flatten()
52     for p, t in zip(preds, targets):
53         confusion[p, t] += 1
54     return confusion
```

The computation logic for the specific parameters related to confusion matrix analysis is outlined in the comment block accompanying this module, given the Listing 60. True positives (TP) are identified when both the true label and the predicted label are the same, whereas false positives (FP) are determined by elements in the same column that were predicted as positive but are not true positives. Similarly, false negatives (FN) are identified by elements in the same row that belong to a specific class but were predicted incorrectly. True negatives (TN) are computed by subtracting the sum of TP, FP, and FN from the total number of instances.

Listing 60 TP, FP, FN and TN calculation from a confusion matrix.

```
14  = [i, i]                  # true positives have the same true label and predicted label
15  = [i, :] - TP             # false positives are the ones that are in the same collumn (were predicted
       but are not true)
16  = [:, i] - TP             # false negatives are the ones that are in the same row (are from that class
       but were predicted wrong)
17  = total - TP - FP - FN    # true negatives are everything that is not in the same row collumn
```

By utilising the obtained confusion matrix, it becomes possible to calculate the overall accuracy of the classification. The accuracy metric measures the proportion of correctly classified instances out of

the total number of instances. The provided function in Listing 61, `accuracy()`, takes in the confusion matrix as a 2D tensor and returns a one-element tensor representing the overall accuracy. The calculation is performed by summing the diagonal elements of the confusion matrix, which correspond to the true positives, and dividing it by the sum of all elements in the matrix plus a small epsilon value (`EPS`) to avoid division by zero. The resulting accuracy value is then returned, ensuring it is on the same device as specified by the device parameter.

Listing 61 Accuracy Metric.

```
57 def accuracy(conf_matrix: torch.Tensor) -> torch.Tensor:
58     """Gives the overall accuracy of the given confusion matrix.
59     Args:
60         conf_matrix (torch.Tensor): 2D tensor with the confusion matrix.
61     Returns:
62         torch.Tensor: One element tensor with the overall accuracy.
63     """
64     return conf_matrix.diagonal().sum() / (conf_matrix.sum() + EPS)
```

Similarly, precision and recall metrics can be calculated, as seen in Listings 62 and 63. The `precision()` function, shown in Listing 62, takes in the confusion matrix as a 2D tensor and returns a 1D tensor containing the precision value for each class. Precision measures the proportion of true positive predictions out of all positive predictions (true positives and false positives). To compute precision, the function extracts the true positives (TP) from the diagonal of the confusion matrix. False positives (FP) are obtained by summing the elements in each column of the confusion matrix and subtracting the corresponding TP value. The precision for each class is then computed by dividing TP by the sum of TP, FP, and a small epsilon value (EPS) to prevent division by zero.

Listing 62 Precision Metric.

```
81 def precision(conf_matrix: torch.Tensor) -> torch.Tensor:
82     """Gives the precision for each class of the given confusion matrix.
83     Args:
84         conf_matrix (torch.Tensor): 2D tensor with the confusion matrix.
85     Returns:
86         torch.Tensor: 1D tensor with the precision for each class.
87     """
88     TP = torch.diag(conf_matrix)
89     FP = conf_matrix.sum(dim=0) - TP
90     return TP / (TP + FP + EPS)
```

The `recall()` function, shown in Listing 63, calculates the recall for each class using the provided confusion matrix. Recall, also known as sensitivity or true positive rate, measures the proportion of true positive predictions out of all actual positive instances (true positives and false negatives). The function begins by extracting the TP values from the diagonal of the confusion matrix. False negatives (FN) are

obtained by summing the elements in each row of the confusion matrix and subtracting the corresponding TP value. The recall for each class is then computed by dividing TP by the sum of TP, FN, and EPS.

Listing 63 Recall Metric.

```
93  def recall(conf_matrix: torch.Tensor) -> torch.Tensor:
94      """Gives the recall for each class of the given confusion matrix.
95      Args:
96          conf_matrix (torch.Tensor): 2D tensor with the confusion matrix.
97      Returns:
98          torch.Tensor: 1D tensor with the recall for each class.
99      """
100     TP = torch.diag(conf_matrix)
101     FN = conf_matrix.sum(dim=1) - TP
102     return TP / (TP + FN + EPS)
```

Using these two metrics, the `f1_score()` function shown in Listing 64 calculates the F1 Score for each class based on the precision and recall values. The F1 Score provides a balanced evaluation of a classifier's performance. By calling the `precision()` and `recall()` functions, the function obtains the precision and recall values for each class using the given confusion matrix. The F1 Score is then computed using these values.

Listing 64 F1 Score Metric.

```
105 def f1_score(conf_matrix: torch.Tensor) -> torch.Tensor:
106     """Gives the F1 Score for each class of the given confusion matrix.
107     It uses the precision and recall for each class.
108     Args:
109         conf_matrix (torch.Tensor): 2D tensor with the confusion matrix.
110     Returns:
111         torch.Tensor: 1D tensor with the F1 Score for each class.
112     """
113     precisions = LearningSystem.precision(conf_matrix)
114     recalls = LearningSystem.recall(conf_matrix)
115     return 2 * ((precisions * recalls) / (precisions + recalls + EPS))
```

Concluding this section, the `mcc()` function in Listing 65 calculates the Matthews Correlation Coefficient (MCC) for each class based on the given confusion matrix. The MCC is a measure of the quality of binary classification models, taking into account true positives, true negatives, false positives, and false negatives. By computing the values of TP, FP, FN, and TN using the confusion matrix, the function then applies the MCC formula to obtain the MCC values for each class.

4.3 Storage System

The Storage System module in BorisCAD's cognitive architecture focuses on efficiently storing and manipulating long-term and working memory, which is crucial for performing complex Computer-Aided

Listing 65 Matthews Correlation Coefficient Metric.

```
118 def mcc(conf_matrix: torch.Tensor) -> torch.Tensor:
119     """Calculates the Matthews Correlation Coefficient for each class of the given confusion matrix.
120     Args:
121         conf_matrix (torch.Tensor): 2D tensor with the confusion matrix.
122     Returns:
123         torch.Tensor: 1D tensor with the MCC for each class.
124     """
125     TP = torch.diag(conf_matrix)
126     FP = conf_matrix.sum(dim=0) - TP
127     FN = conf_matrix.sum(dim=1) - TP
128     TN = conf_matrix.sum() - (TP + FP + FN)
129     return ((TP * TN) - (FP * FN)) / (torch.sqrt((TP + FP) * (TP + FN) * (TN + FP) * (TN + FN)) + EPS)
```

Design (CAD) tasks. By simplifying certain biological complexities, the Storage System module provides a streamlined proof of concept while capturing the fundamental aspects of memory processes.

In BorisCAD, the Storage System module is a repository for storing and retrieving essential design elements, such as image features and their contents, enabling the architecture to make informed decisions based on past experiences. Long-term memory retains information over an extended period, while working memory actively manipulates this stored information in real-time during decision-making.

The Storage System module incorporates a graph structure into its memory design to optimise information storage and manipulation. This graph structure efficiently represents and organises complex relationships between design elements, such as parts, sub-assemblies, and assemblies. Each memory node within the graph represents an image perceived by the system's bottom level and contains multiple abstraction layers, including high and low-level feature maps that provide visual or sensory representations of the image.

Starting with the high-level interface, the `StorageSystem` class depicted in Listing 66 connects the working and long-term memory within the cognitive architecture. It facilitates various operations in encoding, recognition, retrieval, and decision-making processes.

This class is implemented as a singleton, ensuring that there is only one instance of the class throughout the system. This design allows multiple components within the system to access the `StorageSystem` without creating new instances, promoting efficiency and consistency.

Upon initialisation, the `__init__()` method takes in a parameter `wm_size`, representing the working memory's size. It creates an instance of the `WorkingMemory` class, representing the working memory object with the specified size. Additionally, it creates an instance of the `LongTermMemory` class, representing the long-term memory object. The `_most_recent_node` attribute of the `StorageSystem` class is used to store the most recent memory node. This attribute refers to the `MemoryNode` object representing the most recently encoded information in the memory system.

Listing 66 Storage System __init__function.

```
class StorageSystem(metaclass=Singleton):
    """
    This class works as a connection between the working and long-term memory.
    It calls all operations that need to be performed during the encoding, recognition, retrieval, and
    decision-making process.
    It is implemented as a singleton class, allowing it to be accessed by multiple places in the system
    without creating a new instance.
    """

    def __init__(self, wm_size: int) -> None:
        """Initialize the StorageSystem object.
        Args:
            wm_size (int): The size of the working memory.
        """
        self.working_memory = WorkingMemory(wm_size)  # Create a working memory object
        self.long_term_memory = LongTermMemory()  # Create a long-term memory object
        self._most_recent_node: MemoryNode = None  # Store the most recent memory node
```

To implement the `StorageSystem` class as a singleton, the auxiliary type class `Singleton` is utilised. This class, depicted in Listing 67, acts as a metaclass that creates a base class with singleton behaviour when called. The Singleton class includes a dictionary attribute `_instances`, which is used to store instances of classes that implement the singleton pattern. The `__call__()` method within the class overrides the default behaviour of creating a new instance when a class is called. Instead, it enforces the singleton behaviour by checking if an instance of the class already exists in the `_instances` dictionary. If the class doesn't have an instance yet, a new one is created and stored in the dictionary. Subsequent calls to the class will then return the stored instance rather than creating a new one.

Listing 67 Singleton class implementation.

```
class Singleton(type):
    """A metaclass that creates a Singleton base class when called."""

    _instances = {}  # Dictionary to store instances of Singleton classes

    def __call__(cls, *args, **kwargs):
        """Overrides the __call__ method to enforce singleton behavior.
        Args:
            cls: The class being instantiated.
            *args: Positional arguments passed to the class constructor.
            **kwargs: Keyword arguments passed to the class constructor.
        Returns:
            The instance of the Singleton class.
        """
        if cls not in cls._instances:
            # If the class doesn't have an instance yet, create a new one and store it
            cls._instances[cls] = super(Singleton, cls).__call__(*args, **kwargs)
        return cls._instances[cls]  # Return the stored instance for subsequent calls
```

To enable the creation of a new node in memory, a property was added to the `StorageSystem` class. When this property is assigned a value, it triggers adding the node to the memories, as illustrated in Figure 26. This triggering point is achieved by modifying the `forward()` function in the ResNet architecture, which is responsible for gathering feature maps during inference time. The modified `forward()` function,

depicted in Listing 68, demonstrates this process. In this modified function, the low-level and high-level features are stored from the first and last layers of the network, respectively. When the network is not in the training mode, a `MemoryNode` is created using the stored features, and it is assigned to the most recent node in the `StorageSystem`.

Listing 68 Modified ResNet network forward method.

```
120 def forward(self, X: torch.Tensor) -> torch.Tensor:
121     """Forward pass of the network.
122     Args:
123         X (torch.Tensor): Input tensor.
124     Returns:
125         torch.Tensor: Output tensor after passing through the network.
126     """
127     X = self.input(X)  # Pass the input through the initial layers
128     for i, layer in enumerate(self.network):
129         X: torch.Tensor = layer(X)  # Pass the input through each layer in the network
130         if self.training:
131             continue
132         if i == 0:
133             _low_level_features = X.clone()  # Store the low-level features at the first layer
134         elif i == len(self.network) - 1:
135             _high_level_features = X.clone()  # Store the high-level features at the last layer
136     if not self.training:
137         # Create a MemoryNode with the stored low-level and high-level features and assign it to the most
                recent node in the StorageSystem
138         StorageSystem().most_recent_node = MemoryNode((_low_level_features, _high_level_features))
139     X = self.avgpool(X)  # Apply adaptive average pooling
140     X = X.reshape(X.shape[0], -1)  # Reshape the tensor to a 2D shape for the fully connected layer
141     # Pass the reshaped tensor through the fully connected layer for classification
142     return self.fc(X)
```

The memory addition process in the `StorageSystem` class is fully implemented, as depicted in Listing 69. This implementation involves the `most_recent_node` setter method, which sets the provided `MemoryNode` as the `_most_recent_node` attribute of the `StorageSystem` class. Furthermore, it triggers adding the new node to the memories.

First, the method adds the new node to the LTM by calling the `add_node()` method of the `LongTermMemory` class. This step establishes connections between the new node and similar nodes in the LTM that surpass a predefined threshold during recognition. Next, the method populates the WM with similar nodes retrieved from the LTM. Using the `get_population()` method of the `LongTermMemory` class, a list of similar nodes is obtained based on the specified similarity criteria. The method then iterates through the retrieved nodes in reverse order and adds them to the WM. This ensures that the most similar node is added last, closely positioned to the newly inferred node. Finally, the method adds the most recent node to the beginning of the working memory by invoking the `add_node()` method of the `WorkingMemory` class.

To gain a clearer understanding of the system's structure, it is essential to visualise the composition of a "memory." The `MemoryNode` class provides a representation of a memory node, as shown in

Listing 69 Implementation of the memory addition process in the StorageSystem class.

```
@most_recent_node.setter
@add_logging
def most_recent_node(self, node: MemoryNode):
    """Setter method for the most_recent_node property.
    Args:
        node (MemoryNode): The memory node to set as the most recent node.
    """
    self._most_recent_node = node
    # Create/add new node to the memories
    # Connect the new node with nodes in the LTM that are similar and above the threshold (first stage of recognition)
    self.long_term_memory.add_node(self._most_recent_node)
    # Populate the working memory with similar nodes (n nodes, the most similar, depending on the working memory size)
    population = self.long_term_memory.get_population(self._most_recent_node)
    # Add the similar nodes to the working memory (maximum number of nodes to be added is one less than the working memory size)
    for node in reversed(population[: self.working_memory.size - 1]):
        # By reversing this list, the most similar node is added last and closest to the inferred node
        self.working_memory.add_node(node)
    # Add the most recent node to the beginning of the working memory
    self.working_memory.add_node(self._most_recent_node)
```

Listing 70. This class consists of a nested `Data` class, defined using the `dataclass` decorator. This class encapsulates the attributes associated with a memory node, including `low_level_features` and `high_level_features`, represented as tensor objects. Additionally, the probabilities, classification, and final decision attributes are optional lists, assigned to the node throughout the execution of the program.

The `MemoryNode` class also contains other attributes and functionalities. The `connected_nodes` attribute is a `SortedDict` that stores the connected nodes, facilitating efficient retrieval. The data attribute is initialised using the `Data` class, taking the provided data set as arguments. An ID is generated for each node using `uuid.uuid4().hex`. Lastly, the hot index attribute is set to 50, representing the default value.

The `add_node()` method in the LTM utilises the properties of `MemoryNode` objects during the recognition task. It compares the similarity between the new node and existing nodes in the memory, based on a recognition threshold. If the similarity meets the threshold, a connection is established between the new node and the existing node. In addition, the `connected_nodes` dictionary, which stores these connections, is automatically sorted when a new similar node is found and inserted. This sorting ensures that the connected nodes are ordered based on their similarity to the new node, allowing for efficient retrieval and decision-making processes. This process is displayed in Listing 71.

The calculation of node similarity is achieved through the implementation of the overloaded `__sub__()` function, as shown in Listing 72. This function computes the similarity score between two `MemoryNode` instances. It calculates the similarity for both the low-level features and high-level features

Listing 70 Implementation of the Memory Node class.

```python
class MemoryNode:
    @dataclass
    class Data:
        low_level_features: torch.Tensor
        high_level_features: torch.Tensor
        probabilities: Optional[list] = None
        classification: Optional[list] = None
        final_decision: Optional[list] = None

        def __repr__(self) -> str:
            return f"Data(low_level_features={self.low_level_features.shape},
                high_level_features={self.high_level_features.shape}, probabilities={self.probabilities},
                classification={self.classification})"

    def __init__(self, data: set):
        """Initialize a MemoryNode instance.
        Args:
            data (set): A set containing the required data attributes for the MemoryNode.
        """
        self.connected_nodes = SortedDict()  # Dictionary to store connected nodes
        self.data = MemoryNode.Data(*data)  # Initialize the data attribute
        self.id = uuid.uuid4().hex  # Generate a unique ID for the node
        self.hot_index: int = 50  # Set the hot index to 50 (default value)
```

Listing 71 Node addition method in the Long-Term Memory.

```python
def add_node(self, node: MemoryNode) -> None:
    """Add a node to the memory.
    Args:
        node (MemoryNode): The node to be added to the memory.
    """
    # Perform recognition process and update connected nodes' hotness
    for node_id in self.memory.keys():
        # Compute the similarity between the new node and existing nodes
        similarity = node - self.memory[node_id]
        # If the similarity meets the recognition threshold, add the connection
        if similarity >= self.T_RECOGNITION:
            node.connected_nodes[node_id] = similarity
    # Add the new node to the memory
    self.memory[node.id] = node
```

by measuring the Euclidean distance between them. The similarity scores are then combined and averaged to obtain the overall similarity score.

Listing 72 Memory Node subtraction method overload.

```python
def __sub__(self, other: MemoryNode) -> float:
    """Compute the similarity between two MemoryNode instances.
    Args:
        other (MemoryNode): The other MemoryNode instance to compare similarity with.
    Returns:
        float: The similarity score between the two MemoryNode instances.
    """
    low_level_similarity: torch.Tensor = 1 / (1 + torch.norm(self.data.low_level_features -
        other.data.low_level_features))
    high_level_similarity: torch.Tensor = 1 / (1 + torch.norm(self.data.high_level_features -
        other.data.high_level_features))
    return ((low_level_similarity + high_level_similarity).mean()).item()
```

The addition process in the Working Memory is demonstrated in Listing 73. When a new node is

added, the memory is updated accordingly. If the node already exists in the memory, it is removed and then re-added at the most recent location. In the case where the memory is already full, the oldest node (at index 0) is replaced with the new node. Finally, the new node is appended to the memory, ensuring it is included in the working memory for subsequent operations. As one can see, the maximum size the WM can have is represented by the size attribute, initialised at the object creation.

Listing 73 Implementation of the addition method in the Working Memory.

```
15 def add_node(self, node: MemoryNode) -> None:
16     """Updates the memory with a new node.
17     If the node is already in the memory, it is moved to the most recent location.
18     If the memory is full, the oldest node is replaced with the new node.
19     Args:
20         node (Node): The node to update the memory with.
21     """
22     if node in self.memory:
23         # If the node is already in the memory, remove it to re-add at a more recent location
24         self.memory.remove(node)
25     elif len(self.memory) >= self.size:
26         self.memory.pop(0)  # If the memory is full, remove the oldest node (at index 0)
27     self.memory.append(node)  # Add the new node to the memory
```

Listing 74 Implementation of method for similar nodes retrieval in the Long-Term Memory.

```
28 def get_population(self, node: MemoryNode) -> List[MemoryNode]:
29     """Retrieve the population of recognized nodes from memory based on the provided node.
30     Args:
31         node (MemoryNode): The node used for retrieval.
32     Returns:
33         List[MemoryNode]: The population of recognized nodes from memory.
34     """
35     # Retrieve the recognized nodes from memory based on the provided node
36     population = [self.memory[key] for key in node.connected_nodes.keys() if self.memory[key].hot_index >
           self.T_REMEMBER]
37     # Update the node core with the new hotness (increase the hotness of accessed items and decrease the
           hotness of non-accessed items)
38     # When retrieving the nodes for the population, those are the nodes that need to be updated,
           everything else in memory needs to decay in hotness.
39     # Identify the IDs of nodes that need to decay in hotness
40     decay_hotness_ids = self.memory.keys() - set(node.connected_nodes.keys()) - {node.id}
41     # Decrement the hotness of non-accessed nodes
42     for key in decay_hotness_ids:
43         self.memory[key].hot_index = max(self.memory[key].hot_index - 1, 0)
44     # Increment the hotness of accessed nodes
45     for key in node.connected_nodes.keys():
46         self.memory[key].hot_index = min(self.memory[key].hot_index + 1, 100)
47     return population
```

To complete the memory encoding process, it is necessary to define the retrieval of nodes from the long-term memory, which are added alongside the most recent node and utilised for decision-making. The following code snippet, depicted in Listing 74, illustrates the implementation of the `get_population()` method. This method retrieves a group of recognised nodes from the memory based on a provided node. It identifies nodes with a hot index exceeding a certain threshold, indicating their relevance and recent access. The method also manages the hotness of nodes by incrementing the hot index for accessed

nodes and decrementing it for non-accessed nodes.

4.4 Modular Integration: Decision-Making

The decision-making process in this architecture incorporates information from multiple levels and incorporates the influence calculated by the working memory for a given input image. To facilitate this process, the attributes of the current node, including probabilities and labels, are transformed into weighted versions of the module outputs. These weights are determined by the class accuracy, giving more importance to accurately predicted classes and reducing the impact of classes that the models struggle to learn effectively. The calculation of these weights relies on analysing a forward pass on a test set, generating a confusion matrix for each label and calculating said class accuracies.

The complete validation process is succinctly illustrated in Listing 75. In this context, a Tensor containing images is employed to perform a forward pass on both the bottom and top levels of the model. The provided labels are utilised as a means of validating the model's performance. The data employed to validate the top-level model consists of the forward pass output obtained from the bottom-level model.

Listing 75 Computation System validation.

```
56 def validate(self, X: torch.Tensor, y: torch.Tensor) -> None:
57     """Validate the computation system using a set of validation data
58     Args:
59         X (torch.Tensor): Test data to use for validation
60         y (torch.Tensor): True labels to use for validation
61     """
62     self._bl.validate(X, y)
63     with torch.no_grad():  # no need to compute gradients
64         X = self._bl(X)  # forwards the bottom-level with the testing data
65     self._tl.validate(X, y)
```

To generate the confusion matrices and subsequent calculation of the accuracy for each class, the same process is done in both models. The Listing 76 shows the line used to calculate said metric. It achieves this by utilising the `zip()` function to iterate over the transpose of `predictions` and , pairing corresponding elements together. Within the iteration, it calls the `confusion_matrix()` function from the `LearningSystem` class, passing in the paired `preds` and `targets` as arguments. This function computes a confusion matrix, which evaluates the performance of a classification model by comparing predicted labels against true labels. The resulting confusion matrix is then passed to the `accuracy()` function to determine the accuracy. This process is carried out for each pair of predictions and targets, and the resulting accuracy values are stored in a list called self.accuracies. This list is then used to weight

the predictions from these models in order for them to be used in the decision-making calculation as seen in Listing 78.

Listing 76 Calculation of accuracies using a confusion matrix for each class.

```
self.accuracies = [LearningSystem.accuracy(LearningSystem.confusion_matrix(preds, targets)) for preds,
    targets in zip(predictions.T, y.T)]
```

Once the weighted values are obtained, they are assigned to the attributes of the current node in the Computation System's forward call, as seen in Listing 77, building upon the code previously presented in Listing 52. The final decision is made in the forward call of the `Boris` class, where the equation 3.21 is applied. The predefined weights assigned to each component in this final decision are then used during validation to fine-tune the model's decision-making process.

Listing 77 Computation System forward method with memory access and assignments.

```
def __call__(self, X: torch.Tensor) -> Tuple[List[int], List[int]]:
    """Performs the inference on the given input.
    Args:
        X (torch.Tensor): Input tensor.
    Returns:
        Tuple[List[int], List[int]]: Tuple containing the probabilities and labels.
    """
    with torch.no_grad():  # no need to compute gradients
        probability: torch.Tensor = self._bl(X)  # forwards the bottom-level
    probability = probability.cpu().tolist()[0]  # converts the probability tensor to a list
    StorageSystem().most_recent_node.data.probabilities = probability  # updates the probabilities in the
        memory node
    labels: np.ndarray = self._tl(np.array(probability))  # forwards the probability in the top-level
    StorageSystem().most_recent_node.data.classification = labels  # updates the classifications in the
        memory node
    labels = labels.tolist()[0]  # converts the labels numpy array to a list
    return (probability, labels)
```

The `__call__()` method in the `Boris` class depicted in Listing 78 plays a crucial role in determining the ultimate decision based on an input image. When invoked with an input tensor X, this method triggers the Computation System to fetch the probabilities and labels associated with the input. These probabilities denote the predicted class probabilities, while the labels represent the predicted class labels themselves. Subsequently, the method retrieves the influence value from the working memory, reflecting the impact or significance assigned to the input image based on past experiences.

The weights, denoted as alpha, beta, and omega, govern the respective contributions of each module's output to the final decision. By combining the outputs using these weights, a comprehensive decision is derived. This approach allows the Boris class to provide a holistic decision by considering the predicted probabilities and labels as well as the influence obtained from the working memory.

Listing 78 Boris decision-making implementation.

```
def __call__(self, X: torch.Tensor) -> List[int]:
    """Perform the forward pass of the Boris instance.
    Args:
        X (torch.Tensor): Input tensor.
    Returns:
        list: Final decision based on probabilities, labels, and working memory influence.
    """
    (probabilities, labels) = self._cs(X)  # Call the ComputationSystem to obtain probabilities and labels
    wm_influence = self._ss.working_memory.get_influence()  # Get the influence from the working memory
    alpha, beta, omega = 0.33, 0.33, 0.33  # Define weights for combining the outputs
    # Calculate and store the final decision in the current node's data
    StorageSystem().most_recent_node.data.final_decision = ((np.array(probabilities) * alpha +
        np.array(labels) * beta + np.array(wm_influence) * omega) / 3).tolist()
    return StorageSystem().most_recent_node.data.final_decision  # Return the final decision
```

4.5 Data Management

Effective data management is crucial for successful machine learning tasks such as segmentation and classification. In this subsection, the implementation of two Dataset classes is presented, namely `ImageMaskDataset` and `CustomDataset`, that provide convenient and efficient ways to handle data for segmentation and classification tasks, respectively. These classes encapsulate the necessary functionalities to load, preprocess, and organise the data required for training and evaluation. Additionally, the specific file structures required for each dataset can be outlines, ensuring proper alignment between images, masks, and labels. Utilising these Dataset classes and adhering to the prescribed file structures makes the data management process streamlined and facilitates seamless integration with the machine learning models.

The `ImageMaskDataset` class is specifically designed to handle the data required for training the segmentation network. It expects a specific file structure where each image and its corresponding mask are stored together. The dataset folder should contain image files and their corresponding mask files. The image files should be named as desired, while the mask files should have the same name as their corresponding image file but with `"_mask"` appended to it. For example, if an image file is named `"image1.jpg"`, its corresponding mask file should be named `"image1_mask.jpg"`. This naming convention ensures that each image is associated with its respective mask. The class takes a folder path, the expected size of the images and masks, and optional transformation functions for images and masks. It utilises the *torchvision* library to load and preprocess the image and mask data. The `__len__()` method returns the length of the dataset, and the `__getitem__()` method retrieves an item from the dataset. It opens the image and mask files, converts them to tensors, and applies transformations if specified. Assertions are included to ensure that the image and mask sizes match the expected size.

The `CustomDataset` class caters to the data requirements of the classification network. It expects a specific file structure where the image files are accompanied by a CSV file containing the corresponding labels. The dataset folder should contain the image files and a CSV file named `"image_labels.csv"` that stores the labels. The CSV file should have the structure seen in Table 4.

Table 4: Classification dataset CSV structure.

Image Name	Label 1	Label 2	...	Label N
image1.jpg	0	1	...	1
image1.jpg	1	0	...	0
...	...	...	...	...
imageN.jpg	1	0	...	1

The CSV file should have the image names in the first column and the corresponding labels in subsequent columns. Each row represents the image-label pair. The `CustomDataset` class takes a folder path, the expected size of the images, and an optional transformation function as inputs. It reads image labels from the CSV file in the dataset folder and stores them internally. The class initialises other attributes, such as the base directory, class names, and image folder, based on the images' size and expected size. The `__len__()` method returns the dataset's length, corresponding to the number of images. The `__getitem__()` method retrieves an item from the dataset. It opens the image file, converts it to a tensor, and applies transformations if specified. The labels associated with the item are also retrieved and converted to a tensor. Assertions are included to ensure that the image size matches the expected size.

The specific methods for retrieving an item from each dataset are depicted in Listings 79 and 80. The first code snippet loads an image and its corresponding mask for image segmentation. It constructs file paths for the image and mask, opens them, converts them to tensors, checks their sizes, applies transformations if needed, and returns the transformed image and mask tensors. The second code snippet loads an image and associated labels for classification. It constructs the image path, retrieves the labels, opens the image, checks its size, applies transformations if provided, converts the labels to a tensor, and returns the transformed image tensor and label tensor.

Listing 79 Implementation of the method used to retrieve an image-mask object for segmentation.

```
137     image_path = os.path.join(self.dataset_folder, self.image_files[idx])  # Get the path of the image
138     mask_path = os.path.join(self.dataset_folder, self.image_files[idx].split(".")[0] + "_mask." +
            self.image_files[idx].split(".")[1])  # Get the path of the corresponding mask
139     image = to_tensor(Image.open(image_path).convert("RGB"))  # Open the image and convert it to a tensor
140     mask = to_tensor(Image.open(mask_path).convert("L"))  # Open the mask and convert it to a tensor
141     assert (image.shape[1], image.shape[2]) == self.expected_size  # Check if the image size matches the
            expected size
142     assert (mask.shape[1], mask.shape[2]) == self.expected_size  # Check if the mask size matches the
            expected size
143     if self.transform:
144         image = self.transform(image)  # Apply the data transformation function to the image
145     if self.target_transform:
146         mask = self.target_transform(mask)  # Apply the data transformation function to the mask
147     return image, mask
```

Listing 80 Implementation of the method used to retrieve an image-labels object for classification.

```
179     if torch.is_tensor(idx):
180         idx: list = idx.tolist()  # Convert the index to a list if it's a tensor
181     img_path = os.path.join(self.base_dir, self.image_folder, self.data.iloc[idx, 0])  # Get the image
            path
182     labels = self.data.iloc[idx, 1:]  # Get the labels for the item
183     img = to_tensor(Image.open(img_path).convert("RGB"))  # Open the image and convert it to a tensor
184     # Check if the image size matches the expected size
185     assert (img.shape[1], img.shape[2]) == self.expected_size
186     if self.transform:
187         img = self.transform(img)  # Apply the data transformation function to the image
188     label_tensor = torch.tensor(labels, dtype=torch.float32)  # Convert the labels to a tensor
189     return img, label_tensor
```

Chapter 5

Experimental Analysis and Verification

This chapter will explore the testing process of the BorisCAD architecture. At this stage, the primary objective is to ensure the system performs optimally and reliably during training and inference. Given that the system handles real-world input data, the ability to make accurate predictions becomes a crucial requirement.

Encompassing neural networks for segmentation and classification, the bottom level of the computation system forms the foundation of sub-symbolic data in this architecture. The segmentation network, responsible for partitioning input data into distinct regions, facilitates precise location and identification of relevant features. The classification network then uses this to assign accurate labels or classes to the previously segmented regions.

At the top level, the system incorporates a decision forest composed of multiple decision trees. Decision trees offer a structured and hierarchical approach to decision-making, handling symbolic data to derive accurate conclusions. The decision forest integrates the outputs of individual decision trees to make robust and informed predictions. The main objective of testing this part of the system is to evaluate the generalisation of the induction mechanism based on the given dataset while avoiding overcommitting resources to overfitting the training data.

Furthermore, the evaluation will occur during the inference phase involving memory. As the architecture processes new images over time, it actively constructs memory to enhance the final decision-making, resulting in more accurate information identification from previous data.

During this testing phase, the focus will be on evaluating the overall performance and reliability of the system by simulating different input conditions, varying data and configuration parameters. By doing so, it will be graphically visible where the system needs to be optimised or improved.

Thus, this chapter will outline the methodologies and techniques employed to conduct this

comprehensive testing of the system during training and inference time. Evaluation metrics and other relevant indicators will be explored to quantify the system's effectiveness.

In order to establish a benchmark, the testing framework incorporates a system comprising a NVIDIA GeForce RTX 3060 GPU housing 12GB of dedicated memory. This GPU is synergistically coupled with an AMD Ryzen 7 3700X processor and an 48GB of DDR4 RAM operating at a clock speed of 3600 MHz. The collective configuration presents a cumulative memory capacity of 36GB, ensuring optimal utilisation of the GPU's capabilities.

5.1 Modular Testing

Due to the cognitive architecture of this system, which employs bottom-up learning, the testing process is applied sequentially. It follows a modular approach, where each component of the system is individually assessed before evaluating their integration. This ensures a thorough examination of the performance and functionality of each module. Past inferences from lower methods are then utilised in higher methods, further emphasising the need for testing and validation before evaluating subsequent components in the chain. Therefore, the learning process is presented hierarchically, with each model utilising the previous one during both the training and inference stages.

The testing process focuses on different modules within the system, starting with the segmentation network. This module undergoes independent training and testing to determine its effectiveness in partitioning input data into distinct regions. The assessment evaluates its accuracy and precision in identifying relevant features.

Next, the evaluation shifts to the classification network. This module utilises the output of the segmentation network and assigns accurate labels or classes to the segmented regions. The analysis includes variations with and without the segmentation pre-processing step, aiming to understand the influence of this step on the final classification results.

Furthermore, the classification network generates probabilities that play a crucial role in constructing the decision forest. These probabilities, derived from the classification module, enhance the robustness and reliability of the decision forest's predictions.

The image sizes and number of epochs used for training all the models in the computation system were selected based on previous research findings and to allow for a comprehensive range of evaluations without excessive resource allocation.

5.1.1 Bottom-level: segmentation and classification

In the context of testing the segmentation network, a series of experiments has been designed to evaluate its performance and robustness. These tests involve varying image sizes and numbers of training epochs, aiming to accurately assess the network's ability to partition input data into distinct regions. The testing configuration is explicit in Table 5.

Table 5: Segmentation tests planning.

Test Configuration	Image Size	Number of Epochs
Test 1	224×224	8
Test 2	224×224	16
Test 3	224×224	32
Test 4	480×480	8
Test 5	480×480	16
Test 6	480×480	32
Test 7	512×512	8
Test 8	512×512	16
Test 9	512×512	32

The number of training epochs for each test configuration was carefully determined based on extensive experimentation and validation. Through iterative training and evaluation, the optimal point where the network achieves peak performance without overfitting or diminishing returns is found. Furthermore, the network's capability to generalise across different image sizes and adapt to various training durations is of significant interest. Several metrics will be employed to measure the segmentation network's performance, including mask accuracy, dice score, and Intersection over Union (IoU).

Mask accuracy quantifies the pixel-wise precision of the generated segmentation masks. It quantitatively assesses how closely the network aligns with ground truth annotations, indicating its proficiency in identifying and classifying distinct regions within the input data. The dice score is a metric that measures the similarity between the predicted segmentation masks and the ground truth masks. It assesses the overlap between the two masks, considering both true and false positives. The dice score comprehensively evaluates the network's ability to balance precision and recall in its segmentation outputs. IoU calculates the ratio of the area of overlap between the predicted segmentation masks and the ground truth masks to the total area encompassed by both masks. IoU provides a robust measure of segmentation accuracy, reflecting how well the network delineates regions of interest.

Employing these metrics allows for a thorough evaluation of the segmentation network's performance across various test configurations. By considering different image sizes and training

epochs, the network's adaptability to varying input dimensions and training durations can be analysed, aiding in identifying optimal settings. Additionally, only a single dataset of image-masks will be used to evaluate the performance of the segmentation system. This dataset has been selected to represent a diverse range of images and provides the necessary ground truth masks for evaluation.

To train the model, a configuration file can be created as seen bellow. In here, the segmentation dataset is defined, as well as the specific size that the images will have. Additionally, the network selected will include the attention mechanisms. To train the model, a save file is defined, as well as a number of epochs and the batch size. The same process will be done for every other module tested within the system, with specific configurations for the operation at hand.

config_segmenter.yaml

```
Boris:
  segmentation: K:\Dataset\masks
  size: [224, 224]
  Computation:
    Bottom:
      attention: true
Training:
  epochs: [8]
  save_file: segmenter_224_224_16.boris
  batch_size: 32
```

A comprehensive display of the obtained results for the segmentation network training are displayed in Table 6.

Table 6: Segmentation tests results.

Test Configuration	Loss	Accuracy (%)	Dice Score	IoU
Test 1	0.1223	96.62%	0.9567	0.9170
Test 2	0.1056	96.87%	0.9621	0.9269
Test 3	0.0857	97.42%	0.9738	0.9490
Test 4	0.1677	97.56%	0.9620	0.9268
Test 5	0.1253	97.73%	0.9654	0.9332
Test 6	0.0794	97.99%	0.9708	0.9432
Test 7	0.1719	96.98%	0.9495	0.9040
Test 8	0.1298	97.36%	0.9574	0.9184
Test 9	0.1115	96.80%	0.9468	0.8999

In Test 1, Test 2, and Test 3, the image size was kept constant at 224x224 pixels, while the number of epochs increased from 8 to 32. The loss values decreased gradually, indicating improved performance as the model was trained longer. The accuracy, Dice Score, and IoU also showed a positive trend, with higher values achieved with more epochs. Notably, the differences in these metrics between Test 2 and Test 3 were relatively minor compared to the improvement observed between Test 1 and Test 2.

In Test 4, Test 5, and Test 6, the image size was increased to 480x480 pixels, maintaining the same number of epochs as in the previous tests. Surprisingly, the performance did not significantly improve despite the larger image size. The loss values were slightly higher, indicating a slightly less accurate prediction. The accuracy, Dice Score, and IoU values were slightly improved compared to the smaller image size tests, but the gains were not substantial, given the increase in computational time. Increasing the image size beyond a certain threshold may contribute little to the segmentation performance.

In Test 7, Test 8, and Test 9, the image size was further increased to 512x512 pixels, while the batch size was reduced to 16. Similar to the previous tests, the loss values increased slightly compared to the smaller image size tests. The accuracy, Dice Score, and IoU values slightly decreased compared to the 480x480 pixel tests.

In conclusion, the segmentation tests demonstrated that increasing the image size beyond a certain threshold did not substantially improve the model's performance. While larger image sizes were expected to provide more detailed information for segmentation, the gains in accuracy, Dice Score, and IoU were relatively modest compared to the required computational time. These findings suggest a limit to the benefits of increasing image size in the context of segmentation. Therefore, it is essential to carefully consider the trade-off between computational resources and performance when deciding on the optimal image size for segmentation tasks.

During the training, a verbose similar to what is seen in Figure 40 is presented. In here, the metrics associated with the model are seen, in addition to information referring the current batch and completion times. The adaptive learning rate is also displayed whenever there is a change. Additionally, an example of a graph generated using the training metrics is depicted in appendix Figure 51.

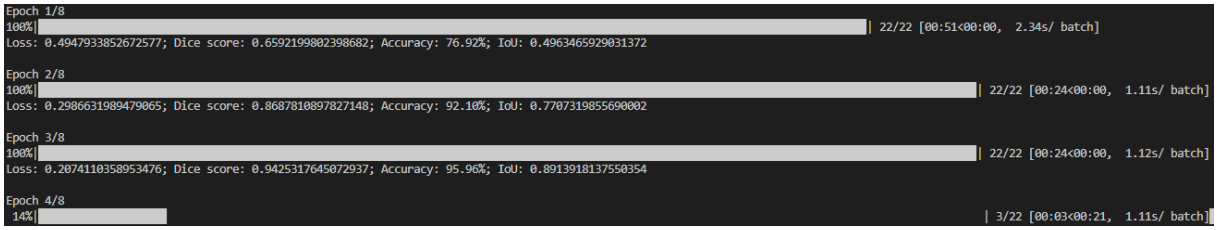


Figure 40: Segmentation Network verbose during training and validation.

The evaluation of the classification network focuses on its ability to assign labels or classes to input images accurately. This assessment is conducted in two scenarios: with and without the segmentation network, analysing the model's behaviour when given pre-processed data that isolates important image sections provides valuable insights. This comprehensive testing approach aims to evaluate the system's capability to accurately identify and classify various objects or entities without relying solely on the input data.

During the testing phase, multiple evaluation metrics are employed to assess the performance of the classification network. These metrics include accuracy, precision, recall, F1 score, and Mathews Correlation Coefficient (MCC). The network's ability to correctly classify raw data and segmented regions in an image can be determined by measuring these metrics, enabling informed predictions.

The testing involves three datasets: CheXpert, NIH Chest X-ray, and pneumonia-related X-ray images. The CheXpert dataset comprises a vast collection of chest X-ray images with annotations provided by radiologists, serving as valuable ground truth for evaluating the network's ability to classify various thoracic pathologies accurately. The NIH Chest X-ray dataset consists of diverse chest X-ray images sourced from the National Institutes of Health, enabling comprehensive evaluation of the classification network's performance across various conditions. Lastly, the pneumonia-related X-ray images dataset focuses specifically on pneumonia-related cases, allowing for an assessment of the network's proficiency in accurately identifying and distinguishing pneumonia-related patterns.

Similar to the segmentation network testing, the testing procedure for the classification network with the segmentation preprocessing is presented in Table 7. This procedure is repeated for each of the three datasets, ensuring the evaluation of the system's performance when encountering different data types. Additionally, and as a mean to compare results, Table 8 was elaborated to compare these results with a classification model trained without the use of the segmentation preprocessing. Since this is only used as an extra comparison and justification for the full usage of the segmentation network, the pneumonia dataset was used for it's low complexity and size.

The results obtained from training the classification network for each of the datasets are presented in Tables 12, 11, and 10. The additionally tested model without the segmentation processing is depicted in Table 9.

In the given Table 9, the results of the pneumonia dataset training without segmentation pre-processing are presented. For Test 1, which utilised an image size of 224x224 and trained for 32 epochs, the obtained results include a loss of 0.4183, an accuracy of 79.34%, precision of 0.7207, recall of 0.7983, F1 score of 0.7575, and MCC of 0.5126.

Table 7: Classification tests with segmentation planning.

Test Configuration	Image Size	Number of Epochs
Test 1	224×224	16
Test 2	224×224	32
Test 3	224×224	48
Test 4	480×480	16
Test 5	480×480	32
Test 6	480×480	48
Test 7	512×512	16
Test 8	512×512	32
Test 9	512×512	48

Table 8: Classification tests without segmentation and using the pneumonia dataset.

Test Configuration	Image Size
Test 1	224×224
Test 2	480×480
Test 3	512×512

Similarly, Test 2 employed an image size of 480x480 and trained for 32 epochs. The results show a lower loss of 0.3590 and an improved accuracy of 82.97%. The precision, recall, F1 score, and MCC were calculated as 0.7739, 0.8330, 0.8023, and 0.4634, respectively.

In Test 3, the image size was increased to 512x512 while training for 32 epochs. The results demonstrate a further reduction in loss to 0.3575 and an increased accuracy of 83.06%. The precision, recall, F1 score, and MCC were computed as 0.7760, 0.8327, 0.8033, and 0.4556, respectively.

Table 9: Classification tests results on pneumonia dataset without segmentation.

Test Configuration	Loss	Accuracy (%)	Precision	Recall	F1 Score	MCC
Test 1	0.4183	79.34%	0.7207	0.7983	0.7575	0.5126
Test 2	0.3590	82.97%	0.7739	0.8330	0.8023	0.4634
Test 3	0.3575	83.06%	0.7760	0.8327	0.8033	0.4556

Analysing the provided results for the pneumonia classification tests, it is possible to observe the system's performance across different image sizes and the number of epochs depicted in Table 10. For the tests conducted with an image size of 224x224, increasing the number of epochs from 16 to 32 (Test 1 to Test 2) led to a slight improvement in most metrics. The loss decreased from 0.4104 to 0.3942, and the accuracy increased from 79.55% to 80.48%. The precision, recall, F1 score, and MCC also demonstrated improvements. Similarly, increasing the number of epochs from 32 to 48 (Test 2 to Test 3) slightly decreased accuracy, precision, recall, F1 score, and MCC. However, the overall performance remained relatively consistent.

Moving to the larger image size of 480x480, the improvements across the board are noticeable. Increasing the number of epochs from 16 to 32 (Test 4 to Test 5) resulted in a significant increase in accuracy from 79.05% to 83.37%. The precision, recall, F1 score, and MCC also improved. Further increasing the number of epochs from 32 to 48 (Test 5 to Test 6) led to a continued enhancement in performance, with accuracy reaching 84.56%.

For the most prominent image size of 512x512, the overall performance is slightly lower compared to the other image sizes. However, increasing the number of epochs from 32 to 48 (Test 8 to Test 9) showed improvements across most metrics, including accuracy, precision, recall, F1 score, and MCC.

Table 10: Classification tests results on pneumonia dataset.

Test Configuration	Loss	Accuracy (%)	Precision	Recall	F1 Score	MCC
Test 1	0.4104	79.55%	0.7246	0.7990	0.7600	0.5178
Test 2	0.3942	80.48%	0.7369	0.8099	0.7717	0.5413
Test 3	0.3960	80.19%	0.7330	0.8061	0.7678	0.5337
Test 4	0.4121	79.05%	0.7173	0.7934	0.7535	0.5043
Test 5	0.3321	83.37%	0.7565	0.8368	0.7946	0.5318
Test 6	0.3123	84.56%	0.8065	0.8488	0.8271	0.5689
Test 7	0.4121	76.25%	0.6946	0.7231	0.7086	0.4324
Test 8	0.3231	85.57%	0.7795	0.8115	0.7952	0.4852
Test 9	0.3013	85.63%	0.7800	0.8121	0.7957	0.4855

Comparing the achieved results, including a segmentation network in the classification system results in improved performance across all image sizes. In Test 2, with an image size of 224x224 and 32 epochs, the accuracy increases from 79.34% to 80,48% with the segmentation network. This improvement is consistent with Test 5 (480x480, 32 epochs) and Test 8 (512x512, 32 epochs), where accuracy, precision, recall, F1 score, and MCC show notable enhancements when segmentation is incorporated. These results indicate that the segmentation network enhances the system's ability to detect and classify pneumonia-related features in the datasets accurately. Consequently, integrating the segmentation network positively impacts the overall performance of the pneumonia classification system. The testing done with the next two datasets (NIH and CheXpert) will always have the segmentation network, in order to increase the testing metrics obtained.

The classification experiments depicted in Table 11 were conducted using the NIH dataset, and the results were obtained for different image sizes and varying numbers of epochs, as shown in Table 7.

Test 1, performed on images of size 224x224 and trained for 16 epochs, yielded a loss of 0.6503. The accuracy achieved was 63.55%, with precision, recall, F1 score, and MCC values of 0.6700, 0.6990, 0.6842, and 0.2013, respectively. The relatively lower performance of Test 1 suggests that the model

Table 11: Classification test results on NIH dataset.

Test Configuration	Loss	Accuracy (%)	Precision	Recall	F1 Score	MCC
Test 1	0.6503	63.55%	0.6700	0.6990	0.6842	0.2013
Test 2	0.6233	65.23%	0.7162	0.7162	0.7162	0.5721
Test 3	0.5739	65.32%	0.7052	0.7183	0.7117	0.5710
Test 4	0.5335	81.23%	0.7219	0.7300	0.7259	0.4701
Test 5	0.5191	85.13%	0.7566	0.7650	0.7608	0.4927
Test 6	0.4648	86.00%	0.7643	0.7729	0.7686	0.4977
Test 7	0.5293	81.63%	0.7302	0.7414	0.7357	0.4501
Test 8	0.4547	85.03%	0.8092	0.8183	0.8137	0.5242
Test 9	0.3604	85.98%	0.8182	0.8092	0.8137	0.5521

still needs to be optimised for this dataset and thus requires further training.

Test 2, conducted with the same image size but trained for 32 epochs, exhibited improvements across all metrics. The loss decreased to 0.6233, and the accuracy increased to 65.23%. The remaining metrics achieved 0.7162, 0.7162, 0.7162, and 0.5721, respectively.

Continuing with Test 3, where the number of epochs increases to 48, the loss dropped to 0.5739 while the accuracy remained consistent at 65.32%. The precision, recall, F1 score, and MCC improved slightly compared to Test 2, reaching values of 0.7052, 0.7183, 0.7117, and 0.5710, respectively. Tests 2 and 3 demonstrate that increasing the number of epochs enhances the model's ability to extract relevant features from the segmented images, improving the gathered metrics but stabilising near the end of the training.

Test 4 was conducted on images of size 480x480 and trained for 16 epochs, an image size larger than the previous tests. This configuration yielded a significantly reduced loss of 0.5335 and an improved accuracy of 81.23%. The remaining metrics reached 0.7566, 0.7650, 0.7608, and 0.4927, respectively. In Test 5, with the same image size but trained for 32 epochs, the loss further decreased to 0.5191, and the accuracy significantly improved to 85.13%.

Test 6 was performed on images of size 480x480 and trained for 48 epochs, resulting in a reduced loss of 0.4648 and an accuracy of 86.00%. The precision, recall, F1 score, and MCC achieved values of 0.7643, 0.7729, 0.7686, and 0.4977, respectively. These findings demonstrate the effectiveness of a larger image size combined with increased epochs in achieving superior classification performance.

The subsequent tests, conducted on images of size 512x512, exhibited similar trends. Test 9, trained for 48 epochs on images of size 512x512, showcased superior performance. The loss reached its lowest value at 0.3604, and the accuracy achieved an outstanding 85.98%. Precision, recall, F1 score, and MCC obtained values of 0.8182, 0.8092, 0.8137, and 0.5521, respectively.

The experiments performed on the NIH dataset demonstrated that increasing the number of epochs and utilising larger image sizes improved metrics, with the best results achieved at the highest image size and for the most epochs. Although verified, these parameters make the system more resource heavy and might only sometimes be achievable in real-world tasks.

Moving to the final dataset used to test the bottom-level performance, the results obtained using the CheXpert dataset are depicted in Table 12.

Table 12: Classification test results on CheXpert dataset.

Test Configuration	Loss	Accuracy (%)	Precision	Recall	F1 Score	MCC
Test 1	0.6493	61.32%	0.6712	0.6754	0.6733	0.3292
Test 2	0.6306	63.87%	0.6980	0.7024	0.7002	0.3423
Test 3	0.5956	63.90%	0.6983	0.7027	0.7005	0.3425
Test 4	0.5713	81.23%	0.7312	0.7464	0.7387	0.4731
Test 5	0.5315	85.13%	0.7647	0.7806	0.7725	0.4948
Test 6	0.5084	86.00%	0.7724	0.7885	0.7804	0.4998
Test 7	0.5569	81.63%	0.7302	0.7914	0.7596	0.5301
Test 8	0.5491	86.13%	0.7683	0.8327	0.7992	0.5578
Test 9	0.4968	86.38%	0.7705	0.8352	0.8015	0.5594

Similar to previous tests, from increasing the number of epochs the model trains for, the metrics significantly improve between 16 and 32 epochs and stabilise afterwards. Test 1, trained for 16 epochs, resulted in a loss of 0.6493 and an accuracy of 61.32%. The precision, recall, F1 score, and MCC values were 0.6712, 0.6754, 0.6733, and 0.3292, respectively. Test 2, trained for 32 epochs, showed improvements across all metrics. The loss decreased to 0.6306, and the accuracy increased to 63.87%. With Test 3, trained for 48 epochs, the loss decreased to 0.5956, while the accuracy remained relatively consistent at 63.90%. Consistently with what happened in the previous testing for the image size of 224x224, the metrics obtained are significantly low compared to larger sizes. This decrease can be explained by the network being unable to capture the image's intricacies with such low resolution and only being able to bias towards the most seen label within the dataset.

Tests 4, 5 and 6 obtained improved results, with the maximum obtained in all metrics achieved at 48 epochs into training. The final loss obtained was 0.5084, and the maximum accuracy took a value of 86%. The remaining analysed metrics also had the highest values, with 0.7724, 0.7885, 0.7804 and 0.4998, respectively.

Test 7, with an image size of 512x512 trained for 16 epochs, displayed a loss value of 0.5569 and an accuracy of 81.63%. The precision, recall, F1 score, and MCC values were 0.7302, 0.7914, 0.7596, and 0.5301, respectively. Test 8, trained for 32 epochs, resulted in a lower loss of 0.5491 and a higher

accuracy of 86.13%. The remaining metrics continue to improve, and, at the end of Test 9, with the duration of 48 epochs, their values achieved the best performance. The model achieved the lowest loss at 0.4968 and an accuracy of 86.38%.

In conclusion, the experiments conducted on the CheXpert dataset demonstrate the significant impact of increasing the number of epochs and utilising larger image sizes on the bottom-level performance of cognitive architecture. The results consistently show that the metrics improve significantly as the number of epochs increases, reaching their peak and stabilising around 48 epochs of training. However, it is worth noting that the limitations of lower image resolutions, as observed in the smaller image size, hinder the model's ability to capture intricate details, resulting in relatively lower performance. Therefore, optimising the trade-off between image resolution and computational resources is crucial when deploying the model in real-world tasks.

Similar to what is seen in the segmentation network, a verbose is displayed during training, depicted in Figure 41. This verbose elaborates on the gathered metrics after every epoch, and has periodic updates on the current learning rate the model is using. Additionally, an example of a graph generated using the training metrics is depicted in appendix Figure 52.

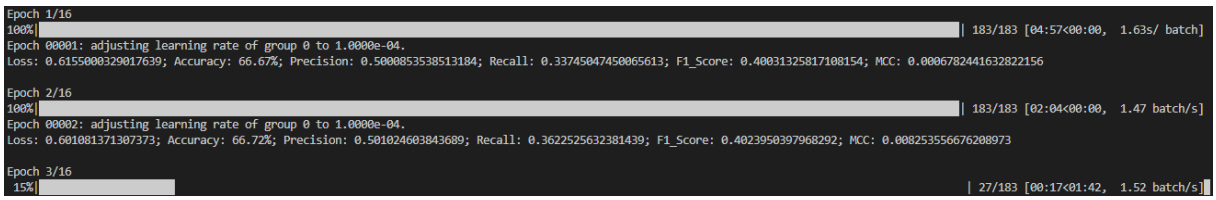


Figure 41: Classification Network verbose during training and validation.

5.1.2 Top-level: Decision Forest building

During the upcoming testing phase, the primary objective will be to evaluate the performance of the decision forest, which serves as the top-level structure of the system. These tests are designed to verify that the decision forest, operating as the symbolic component, can deliver reliable and accurate classifications for the assigned tasks, thereby enhancing the predictions made by the bottom-level component.

In addition, the sub-symbolic model will be utilised to study the improvements brought about by the symbolic approach. By comparing the results obtained from both models, a comprehensive analysis can be conducted to assess the advancements achieved by incorporating the symbolic part in the system.

In the subsequent step, the evaluation will involve utilising the probabilities generated by the bottom-level subsystem, trained explicitly for images with sizes of 224x224 and 48 epochs, on the pneumonia

dataset. These initial metrics will serve as a baseline for assessing the effectiveness of the decision forest in improving classification performance. The impact and improvement brought about by the top-level structure can be quantitatively measured and evaluated by comparing the metrics obtained from the low-accuracy model with the enhanced results achieved through the decision forest.

Additionally, similar evaluation procedures will be conducted on images of larger sizes, specifically 480x480 and 512x512, to analyse the continued relevance and effectiveness of the top-level structure. By testing the decision forest on different image sizes, it will be possible to determine if the improvement provided by the top-level component remains consistent across various image dimensions.

The choice of the pneumonia dataset for these tests is deliberate, as it exhibits a relatively small length and a limited number of classification classes. This selection enables more accessible and less resource-intensive testing, allowing a more efficient evaluation of the decision forest's performance.

Table 13 outlines the test configuration for evaluating the decision forest. The main parameter being varied in these tests is the minimum number of samples required for a split in a decision tree node. By conducting these tests, it becomes possible to observe the impact of altering the minimum samples per split on the decision forest's performance and its ability to generalise and make accurate classifications.

Table 13: Decision Forest tests by varying the minimum number of splits.

Test Configuration	Minimum Samples per Split
Test 1	1
Test 2	4
Test 3	8
Test 4	16

Table 14 depicts the metrics gathered by testing various decision forests on different configurations. Comparing these results to the metrics achieved without the decision tree, specifically for Test 3, it can be observed that the decision tree configuration led to some improvements. The decision tree configuration with a minimum of 8 samples per split showed a higher accuracy of 84.80%, indicating that the decision tree improved the overall classification performance compared to the baseline accuracy of 80.19% without the decision tree. Additionally, the decision tree configuration exhibited a slightly higher precision of 0.7439, suggesting that the decision tree's predictions were more precise in identifying positive instances. However, the decision tree configuration showed a slightly lower recall of 0.7978 compared to the baseline recall of 0.8061, indicating that the decision tree may have missed a few positive instances. The F1 score, which balances precision and recall, was slightly higher for the decision tree configuration at 0.7699 compared to 0.7678 for the baseline. Furthermore, the decision tree configuration achieved a higher MCC of 0.5752, indicating better overall agreement between predicted and true labels, compared to the

MCC of 0.5337 for the baseline configuration.

Table 14: Decision Forest tests using a validated pneumonia model for image size 224x224.

Test Configuration	Accuracy (%)	Precision	Recall	F1 Score	MCC
Test 1	83.90%	0.7365	0.7932	0.7638	0.5637
Test 2	85.20%	0.7476	0.7991	0.7725	0.5801
Test 3	84.80%	0.7439	0.7978	0.7699	0.5752
Test 4	74.90%	0.6753	0.7654	0.7175	0.4598
Baseline	80.19%	0.7330	0.8061	0.7678	0.5337

Similarly, the same tests were done by using the pneumonia model validated on 480x480 and 512x512 images. The results are then presented in Tables 15 and 16. As seen before, there are substantial improvements when using the decision forest as an extra processing step for the classification network. The decision tree configurations for the 480x480 images exhibited varying performance, depicted in Table 15. Test 1 achieved an accuracy of 87.12%, while Test 2 showed improvement with 87.87% accuracy and higher metrics overall. Test 3, with a minimum of 8 samples per split, reached an accuracy of 88.35% and maintained competitive metrics. However, Test 4, with 16 samples per split, experienced a decrease in accuracy. Comparing these configurations to the baseline, the decision tree consistently outperformed it in terms of accuracy, precision, recall, F1 score, and MCC.

Table 15: Decision Forest tests using a validated pneumonia model for image size 480x480.

Test Configuration	Accuracy (%)	Precision	Recall	F1 Score	MCC
Test 1	87.12%	0.7904	0.8517	0.8199	0.6743
Test 2	87.87%	0.8062	0.8632	0.8337	0.7019
Test 3	88.35%	0.7986	0.8574	0.8270	0.6882
Test 4	81.53%	0.8123	0.7369	0.7728	0.5546
Baseline	84.56%	0.8065	0.8488	0.8271	0.5689

Table 16 depicts the metrics for the decision tree on the 512x512 images and, in Test 1, with a minimum of 1 sample per split, the decision tree obtained an accuracy of 86.25%. Test 2, with a minimum of 4 samples per split, showed improvement with an accuracy of 87.53% and higher metrics, suggesting better classification performance. Similarly, Test 3, with 8 samples per split, demonstrated competitive accuracy and metrics, highlighting the decision tree's ability to capture meaningful patterns. However, in Test 4, with a higher minimum of 16 samples per split, the decision tree's accuracy decreased to 80.76%, accompanied by lower performance metrics. Comparatively, the baseline with 48 epochs, achieved a lower accuracy of 85.63% and relatively lower precision, recall, F1 score, and MCC values. These results

indicate that the decision tree configurations (Test 1 to Test 4) generally outperformed the baseline in terms of accuracy and classification metrics on the 512x512 images.

Table 16: Decision Forest tests using a validated pneumonia model for image size 512x512.

Test Configuration	Accuracy (%)	Precision	Recall	F1 Score	MCC
Test 1	86.25%	0.7752	0.8318	0.8025	0.6412
Test 2	87.53%	0.7861	0.8452	0.8146	0.6617
Test 3	86.98%	0.7813	0.8385	0.8089	0.6529
Test 4	80.76%	0.7126	0.7843	0.7467	0.5168
Baseline	85.63%	0.7800	0.8121	0.7957	0.4855

Overall, the decision tree tests showcased the potential of the decision forest, particularly on the larger image sizes, to enhance classification performance. Fine-tuning the hyperparameters, such as the minimum samples per split, proved crucial in optimising the decision tree's effectiveness in accurately classifying pneumonia cases.

For the decision forest building, the verbose in Figure 42 can be seen. Although more simple than the ones previously presented, it shows the probabilities being compiled to be used in training and the start message when each of the decision tree are being built.

Figure 42: Verbose during forest building.

5.2 Inference Testing

Having successfully trained the entire computation system of the architecture for a specific set of labels, the focus now shifts to evaluating its execution during inference. This crucial phase involves integrating the memory component, vital in storing past information and establishing connections with current input data.

To ensure the optimal functioning of the memory component, it is imperative to understand its behaviour comprehensively in this context. This entails investigating how it effectively retrieves and utilises stored memories and fine-tuning its associated parameters. By carefully adjusting these parameters, the system can better capture relevant similarities and associations between memories, enhancing decision-making capabilities.

The entire decision-making process is subjected to rigorous testing following the thorough comprehension and optimisation of the memory component. This evaluation applies the same metrics

for classification and inference, allowing for a comprehensive analysis of how the memory component influences the final decision. Additionally, this testing phase provides insights into the overall performance and effectiveness of the decision-making process.

By systematically examining the interplay between the memory component and the decision-making mechanism using the computation system data, this evaluation aims to unveil the intricate dynamics at play and identify opportunities for improvement. The results obtained from this comprehensive assessment will not only shed light on the impact of the memory component on the system's decision-making capabilities but also contribute to further enhancements in the architecture's overall performance.

5.2.1 Memory Building

The evaluation of the memory component involves a comprehensive set of tests, focusing on long-term memory while the working memory is assessed during inference time. These tests aim to examine the associativity properties of the memory, the connectivity between nodes, and the evolution of "hotness" over time.

Different similarity thresholds, specifically 0.8, 0.85, 0.9, and 0.95, were employed to evaluate the associativity patterns. A higher similarity threshold indicates that images need a more remarkable resemblance to be connected, generating "memory chunks" composed of interconnected nodes but with less density since the images must be nearly identical to connect. Conversely, a lower similarity threshold leads to excessive generalisation, causing unnecessary connections between different images and, thus, larger groups of connected nodes.

Given the standard features present in X-ray images, such as edges and shape, a baseline level of similarity is consistently met. However, the memory tests seek to identify the appropriate threshold that balances capturing meaningful associations and avoiding indiscriminate connections. These thresholds ensure that the memory accurately captures and connects images that exhibit similar patterns besides the obvious ones, enabling the system to make informed decisions based on relevant information.

The same dataset was utilised to conduct the long-term memory tests, employing varying similarity thresholds. This approach allows for an in-depth analysis of how the memory associates and connects nodes based on the given data. The unshuffled data was forwarded through the trained model, ensuring consistent outcomes while solely observing changes in memory properties and structure. In the Figures depicting the tests, the colour assigned to each node represents the "remember" threshold, indicating whether a node is successfully retrieved from the working memory when needed. This additional analysis

material enhances the interpretation of the memory's behaviour and functionality.

For the lowest threshold of 0.8 depicted in Figure 43, the memory construction reveals significant connections even within the first 5 images. These connections stem from the shared similarities inherent in x-ray images, such as similar edges and shapes. As the number of images increases to 20, all nodes become interconnected, forming a single chunk of well-activated memory. With 50 images, the memory remains highly connected, and only a few nodes appear as "forgotten." By the time we reach 100 images, the memory displays a dense structure with a single chunk of interconnected nodes, indicating the presence of strong associations.

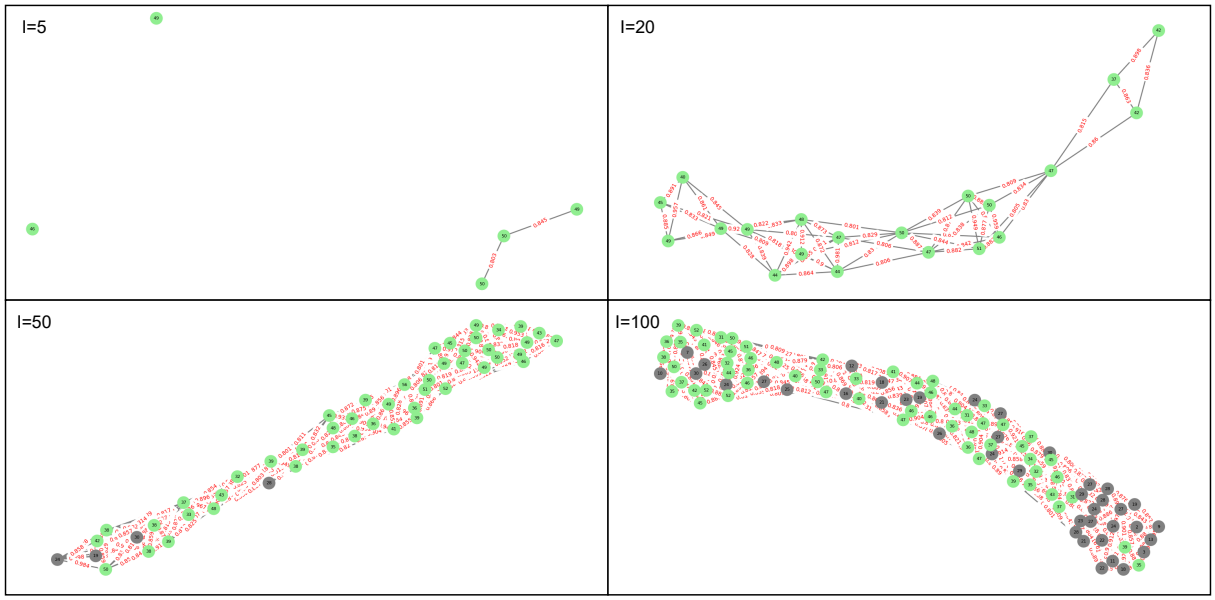


Figure 43: LTM construction (5, 20, 50 and 100 images) with a similarity threshold of 0.8.

At the 0.85 threshold delineated in Figure 44, the memory construction demonstrates a distinct pattern. Even at the 50-image mark, the memory exhibits split chunks of data. A significant number of nodes are forgotten and not activated, resulting in disconnected clusters. However, as we reach 100 images, these clusters gradually connect, forming a unified structure, albeit with fewer connections compared to the lower threshold.

Moving to the 0.9 threshold, the memory displays a unique pattern characterised by a long line of connected nodes with relatively low density. This indicates that for a connection to be established, the images must exhibit a higher degree of similarity. While the memory still forms clusters, the overall structure highlights the need for stronger resemblances between images. This is described in Figure 45.

Finally, at the edge case of the 0.95 threshold shown in Figure 46, the memory construction showcases distinct characteristics. Numerous lonely nodes with no similarity are observed, emphasising the stringent threshold. However, small groups of highly similar nodes are present, indicating a high

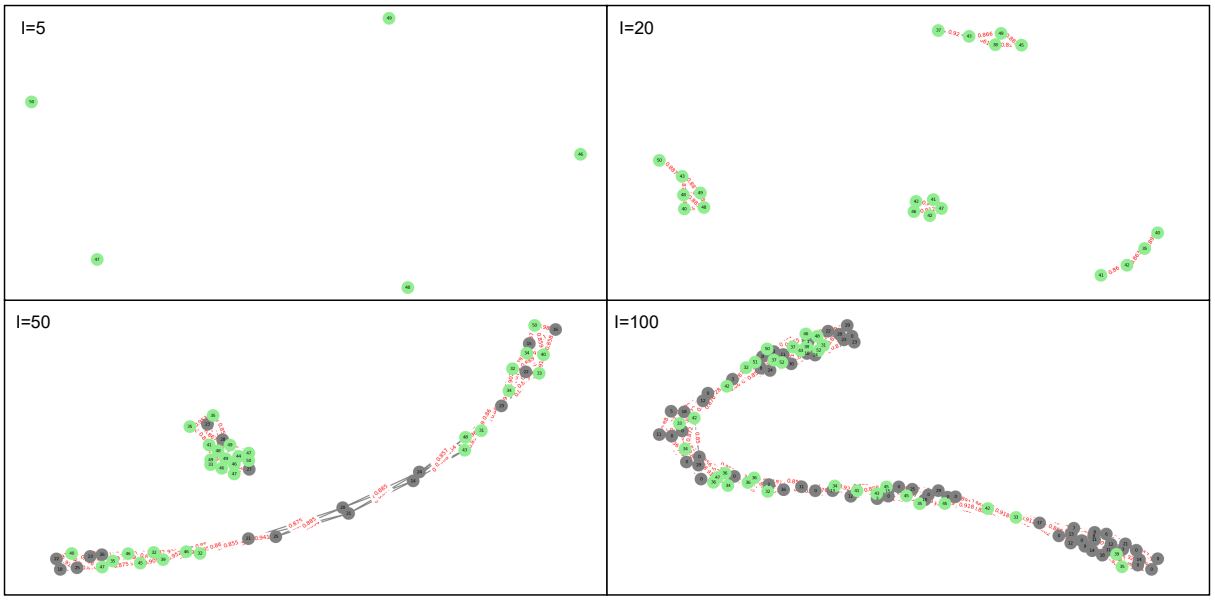


Figure 44: LTM construction (5, 20, 50 and 100 images) with a similarity threshold of 0.85.

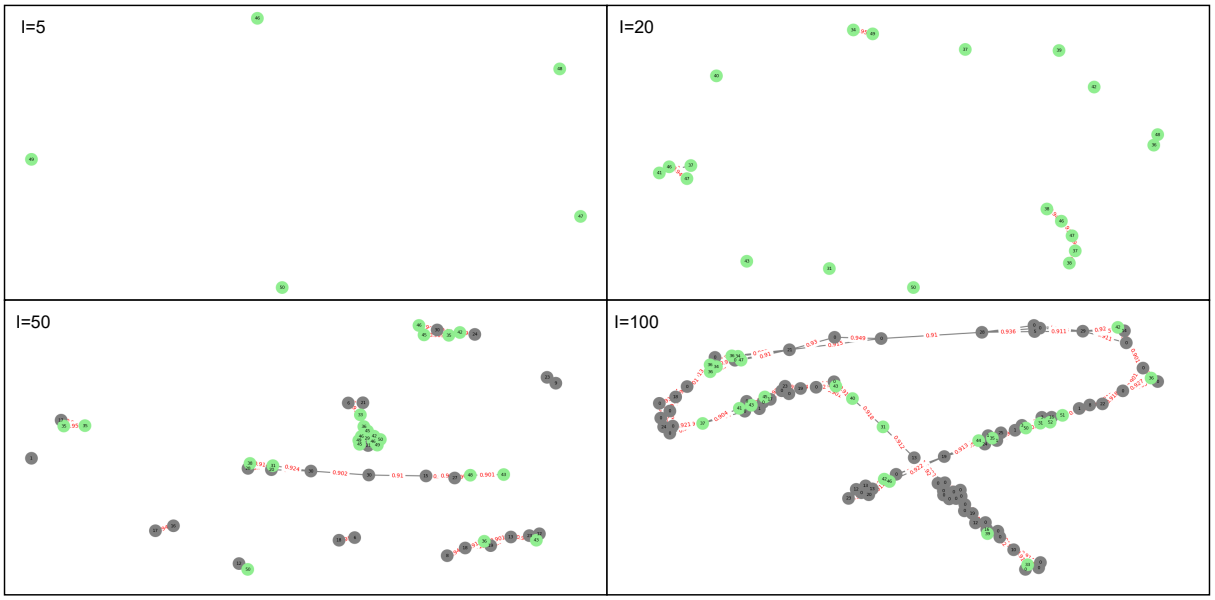


Figure 45: LTM construction (5, 20, 50 and 100 images) with a similarity threshold of 0.9.

level of resemblance within these clusters. The memory structure at this threshold reflects the challenge of establishing connections due to the requirement for almost identical images.

In conclusion, evaluating memory construction at different similarity thresholds sheds light on the intricate dynamics of the memory component within the system. The observations demonstrate that lower similarity thresholds, such as 0.8, result in widespread connections and the formation of a dense, interconnected memory structure. As the similarity threshold increases, the memory becomes more discerning, with connections occurring among images that exhibit higher levels of resemblance. The memory's construction at higher thresholds, such as 0.95, showcases the challenge of establishing

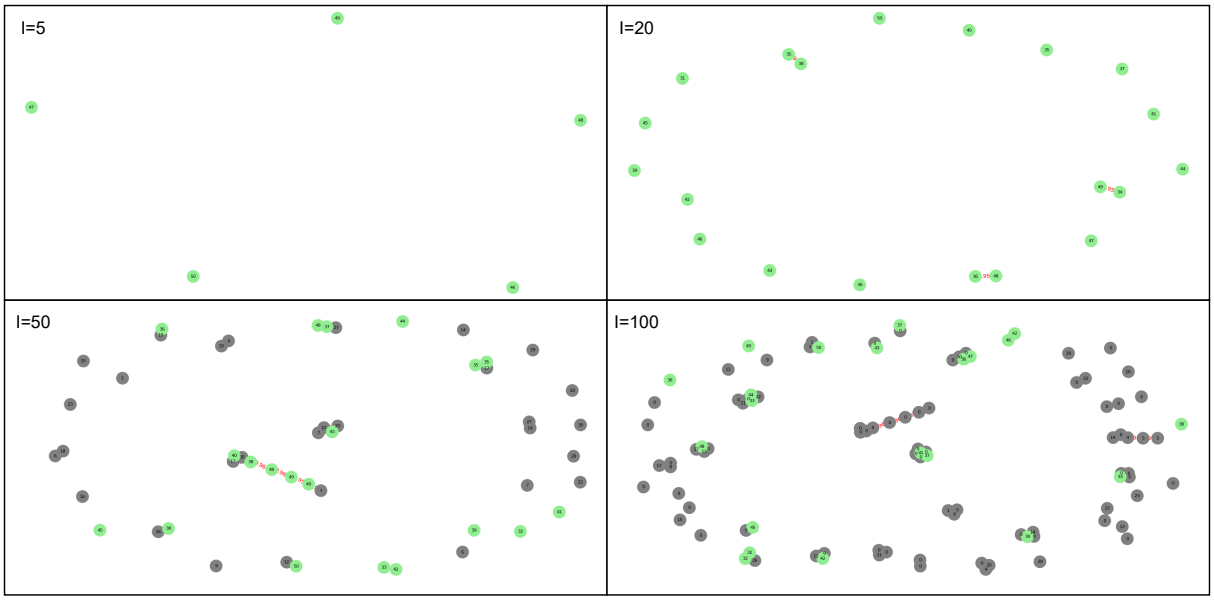


Figure 46: LTM construction (5, 20, 50 and 100 images) with a similarity threshold of 0.95.

connections, with fewer associations forming and emphasising near-identical images. These findings highlight the critical role of similarity thresholds in determining the structure and functionality of the memory, as well as the delicate balance required to capture meaningful associations while avoiding excessive or inadequate connections. Understanding these nuances contributes to the overall understanding of the system's memory capabilities and aids in optimising its performance for real-world inference tasks.

In order to evaluate the functioning of the working memory component, rigorous tests were conducted under controlled conditions, employing a similarity threshold of 0.8 and a memory size of 7. The primary objective of these tests was to investigate the dynamic insertion, removal, and sorting mechanisms within the working memory, guided by connections established in the long-term memory. This thorough analysis ensures the effective encoding, organisation, and retrieval of relevant information within the cognitive architecture.

Figure 47 provides a comprehensive visual representation depicting the progressive behaviour of the working memory as six sequential images are successively inserted. Initially, nodes are added to the working memory successively, reflecting the order of their insertion and exhibiting a decay in their hotness levels. However, a turning point occurs upon introducing the fifth image, where a connection is forged between the latest inserted node and existing nodes. This critical connection triggers a systematic reorganisation of the working memory, wherein the connected nodes are repositioned closer to the top and sorted based on their similarity to the current node.

Continuing the process, Figure 48 depicts the working memory's state as the long-term memory

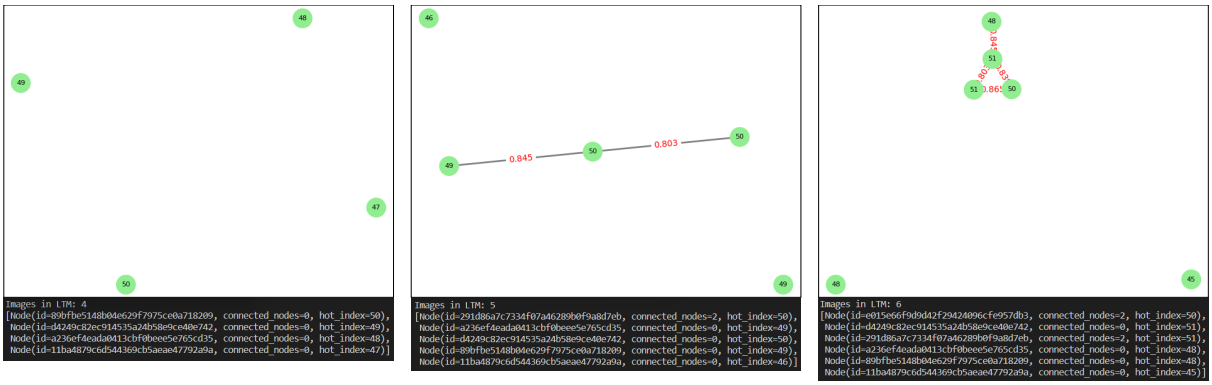


Figure 47: LTM with 4,5 and 6 images and WM orderly structure.

accommodates nine images. Considering the predetermined memory size constraint of 7, a deliberate node removal process is implemented after each iteration to accommodate new connections and nodes. Remarkably, a solitary unconnected node persists within the working memory upon reaching the threshold of seven images and is excluded from the working memory upon introducing the eighth image. However, as this previously unconnected node establishes a subsequent connection with the most recent node, it is reintegrated into the working memory, heralding a profound reorganisation of the memory structure to accommodate these newly connected nodes.

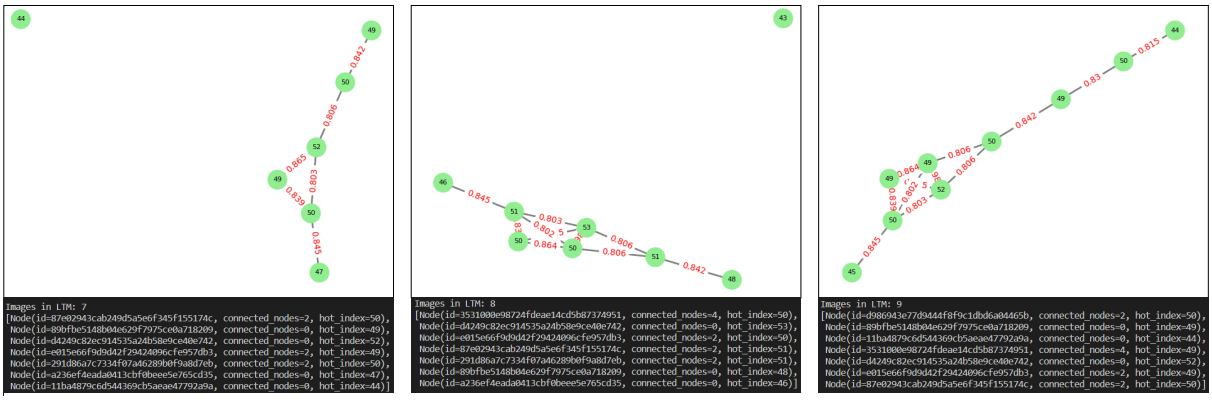


Figure 48: LTM with 7,8 and 9 images and WM orderly structure.

These extensive tests provide compelling evidence of the dynamic nature of the working memory component, which adeptly adjusts its organisation based on the evolving connections established in long-term memory. Such adaptability ensures the seamless integration and utilisation of information, facilitating robust decision-making capabilities within the overarching cognitive architecture.

5.3 Cognitive Architecture: Overtime Improvement

It is now time to move towards full system integration during inference time. The current architecture has implemented the only method for continuous learning related to memory conducibility and the ability to make continuous associations. This evolution aims to improve decision-making by testing new image insertion and connections on the Long-Term Memory.

Until now, all tests have focused on evaluating how well a model performs in assessing a specific label or class within a set of labels for an image. While this aspect is crucial, especially in medical image diagnosis, evaluating how well a model performs when assessing all labels simultaneously is equally essential. This comprehensive evaluation becomes critical since correctly identifying or missing a pathology can have significant consequences. With the full system integration being tested, only fully correct predictions will be counted as valid outcomes. This stringent criterion will be used to validate the accuracy of the architecture.

The testing conditions for this process have been precisely defined to accurately evaluate the performance of the cognitive architecture in this task. The bottom-level analysis will involve 224x224 images trained with segmentation and classification using the previously evaluated highest metrics (32 and 48 epochs, respectively). At the top level, the decision forest will be constructed with a setting of 8 minimum samples per split. Additionally, the system's Working Memory will have a memory size of 7. Both WM and LTM will begin the inference process empty. Five hundred images will be inserted into memory while varying the similarity threshold for this test, repeating the full test after each variation. This variation aims to clarify the effectiveness of connections made and their contribution to the final decision.

Figure 49 illustrates the verbose display after the inference of each image. The image is initially forwarded to the bottom level, creating a new memory node once it reaches the classification network and the feature maps are extracted. Subsequently, the decision forest is inferred, and a set of labels is extracted, which is then combined with the previously obtained classification probabilities. Before making the final decision, the memory gathers information from previous relevant experiences and uses it as a weighting factor in its decision-making process. Initially, each of these predictions will be equally considered in the decision-making. However, these weights will be adjusted, and their influence will be evaluated.

As the first image is inserted, the WM does not influence the final decision, as everything in the Storage System is empty. Consequently, even though the bottom-level predicts both true labels with high probabilities, an error in the top-level prediction leads to an incorrect final decision, resulting in an accuracy

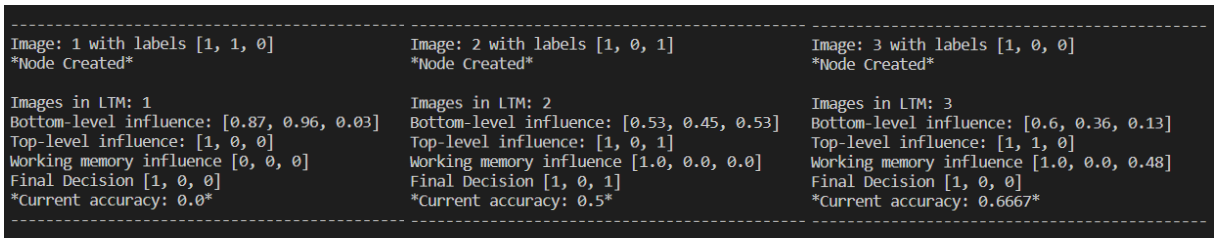


Figure 49: First three images evaluated with the full system.

of 0.

Moving on to the second image, the same process is repeated. In this case, the working memory influences since the previously inserted image is still in memory. Despite the incorrect prediction in the working memory, the correct predictions from both the bottom and top levels result in a correct final decision, thereby increasing the overall accuracy to 50%.

Similarly, as the third image is inserted, the top level incorrectly predicts its labels. However, due to the established influence of the working memory, which now has multiple experiences stored, it assists in correcting this mistake and contributes to the final decision. As a result, the accuracy is improved to 66.6%.

A comprehensive analysis of the accuracy results is depicted in Table 17. The table presents the results of the experiment, showcasing the performance metrics at different image insertion quantities and thresholds. The column headers represent the number of images (100, 200, 300, 400, and 500) inserted into memory, while the rows represent various similarity thresholds. Additionally, the last two rows are labeled "bottom-level" and "top-level" and serve as baseline values for comparison.

Table 17: Accuracy evaluation for varying thresholds.

	Images in memory				
	100	200	300	400	500
0.95	0.738	0.711	0.708	0.720	0.724
0.9	0.800	0.782	0.770	0.781	0.786
0.85	0.874	0.862	0.851	0.852	0.852
0.8	0.755	0.737	0.734	0.748	0.748
0.85 weighted	0.908	0.913	0.913	0.915	0.914
bottom-level					0.683
top-level					0.725

These metrics, as explained before, indicate a complete prediction of the labels existing at a given image. The results demonstrate said success rates achieved by the model under various conditions. For instance, at a threshold of 0.95, the model achieved success rates of 0.738, 0.711, 0.708, 0.720, and 0.723 for 100, 200, 300, 400, and 500 images, respectively. The memory structure at 500 images

for this threshold is depicted in appendix Figure 60. Additionally, the accuracy evolution as the memory is populated can be seen in appendix Figure 61. As the threshold decreased to 0.9, the success rates improved to 0.800, 0.782, 0.770, 0.781, and 0.786 for the corresponding image quantities. The accuracy evolution is presented in appendix Figure 59 and the memory structure in 58. Similarly, for the threshold of 0.85, the success rates were 0.874, 0.862, 0.851, 0.852, and 0.852, as seen in Figure 57. The results varied further as the threshold decreased to 0.8, with success rates of 0.755, 0.737, 0.734, 0.748, and 0.748 for the respective image quantities, described in Figure 55. The visual representation of the LTM for the similarity thresholds of 0.85 and 0.8 are depicted in Figure 56 and in Figures 54 and 53, respectively. Compared to the baseline, it is clear that the bottom and top levels alone do not perform as well as when the memory is introduced.

A final test depicted in Figure 62 was made. In this case, the decision-making equation does not have weights equally distributed across all modules. Instead, following the nomenclature provided in equation 3.21, the α and β weights, referring to the classification and induction mechanisms, respectively, were elevated from 0.33 to 0.4. On the other hand, the ω weight, defining the working memory influence, was decreased to 0.2, giving less weight to the working memory decision but still considering its influence. By analysing the data presented in Table 17, it becomes evident that the application of this approach led to a significant increase in accuracy compared to the previously determined best similarity threshold.

A better visual of this relation between the tests presented in Table 17 is also depicted in the bar plot in Figure 50, where it is clear that this weighted version of the decision-making substantially improved the process when compared to both baseline options and to the other working memory influenced decisions.

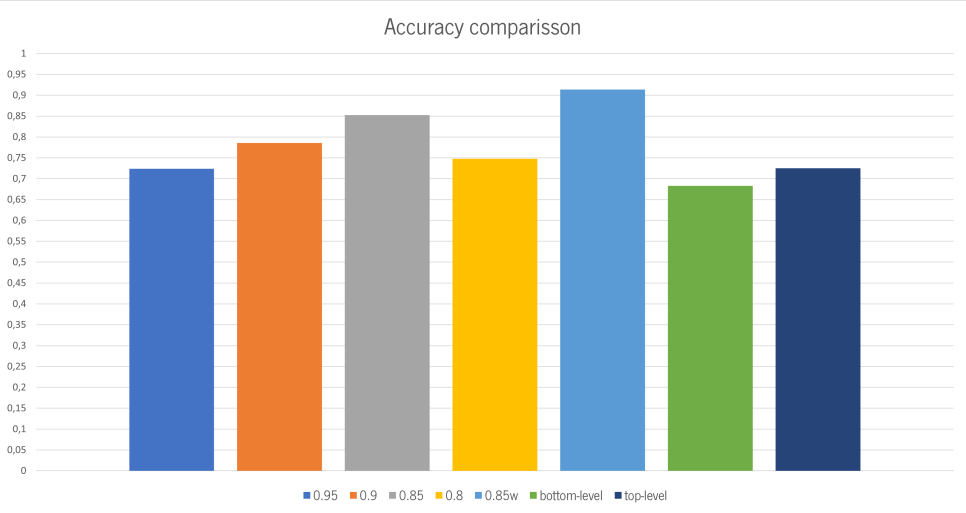


Figure 50: Accuracy comparison for different modules and varying similarity threshold.

Through this analysis, it becomes evident that the threshold at which nodes connect within the memory influences the outcome of the decision memory prediction. The most plausible explanation for the observed performance loss in these scenarios lies in the similarities among the nodes. A low similarity threshold allows nodes to connect more easily, significantly impacting decision-making. Nodes with low similarity will still influence the decision, leading to inaccurate predictions. The low similarity threshold facilitates connections based on shared characteristics, such as lung edges and overall thorax structure, rather than meaningful characteristics, like pathologies.

Conversely, a high similarity threshold makes it more challenging for two images to establish a connection. This scenario proves helpful when two images demonstrate a high level of similarity due to a specific pathology, for instance. While this would enhance prediction accuracy in such cases, it also means that the working memory will contain previously inferred images instead of similar ones, as the latter does not exist within that threshold. Consequently, predictions from the working memory under these circumstances become more random. They are not the result of a recall operation from long-term memory but rather from current images in memory.

5.4 Results Overview

The results overview of the presented cognitive architecture reveals several critical aspects of its performance and capabilities. One fundamental aspect is the integration of segmentation and classification networks within the architecture, which serves as a mechanism to define perception in its cognitive framework. By seamlessly combining these two networks, the cognitive architecture achieves increased efficiency and effectiveness at the bottom level of the model.

Integrating segmentation and classification networks allows for a more comprehensive understanding of visual information. The segmentation network is crucial in identifying and delineating objects or regions of interest within an image. It provides the necessary context for subsequent processing by the classification network, which assigns labels to the segmented regions. This integration gives the architecture a holistic perception of the input data, improving performance in various tasks.

Furthermore, the cognitive architecture acknowledges the importance of analysing and optimising different network configurations for segmentation and classification methods. By carefully tuning the network parameters and architectures, the architecture can be adapted and optimised to perform exceptionally well in specific tasks. This flexibility ensures the cognitive architecture remains versatile and adaptable to different domains or datasets.

During the evaluation process, it became evident that the choice of datasets and image sizes significantly impacted the cognitive architecture's performance. Different datasets encompass varying characteristics, such as image quality, diversity of pathology, and imaging modalities. Moreover, image size can influence the architecture's ability to capture fine details or encompass the entire object of interest. These factors are crucial in shaping the architecture's performance, highlighting the need for comprehensive testing and evaluation.

For instance, in the context of medical image diagnosis, the datasets used in the evaluation may have focused on specific pathologies or imaging modalities, limiting the generalisability of the findings. Given the tailored optimisation and analysis for those particular cases, the cognitive architecture's performance in identifying and diagnosing these specific pathologies may have been excellent. However, a deployable version of the architecture for medical image diagnosis would require further testing to ensure a global standpoint.

To achieve a more comprehensive and deployable architecture for medical image diagnosis, including a broader range of pathologies and imaging modalities in the evaluation process would be necessary. This would involve expanding the dataset to encompass more diverse medical conditions, ensuring the architecture can accurately detect and classify a broader spectrum of diseases. By incorporating a more extensive and diverse dataset, the cognitive architecture could be better equipped to handle real-world scenarios and provide more robust and reliable diagnoses.

Additionally, the evaluation of image size impact revealed the need for a comprehensive assessment of different image resolutions and scales. Some pathologies may require a more detailed examination, necessitating higher-resolution images, while others may be identifiable even at lower resolutions. By evaluating the architecture's performance across a range of image sizes, it becomes possible to determine the optimal settings for deployment in different clinical settings.

Regarding reasoning methods, the cognitive architecture employs decision forests to enhance accurate label prediction. Decision forests leverage the classification probabilities generated by the neural network and consider the relationships between these probabilities. This approach enables the architecture to make informed predictions by considering the spatial characteristics within an image and the overall context provided by the classification probabilities. As a result, the cognitive architecture achieves more accurate label predictions, improving its overall performance.

Additionally, the cognitive architecture incorporates a storage system consisting of working and long-term memories to enhance its prediction capabilities further. By incorporating memory's temporal and relational properties, the architecture can learn from past experiences and refine its predictions

accordingly. This integration of memory adds a valuable dimension to the cognitive architecture, enabling it to leverage previous encounters and adapt its responses based on learned patterns. Including the storage system contributes to the architecture's overall predictive power.

However, during the evaluation process, it was observed that while the memory system positively impacted the cognitive architecture's performance, its influence was negligible in some instances. It became apparent that when more importance was given to the memory system, the accuracy of the architecture began to decay. This raised the need for further investigation into the underlying mechanisms of the memory system and its impact on overall performance.

Upon closer examination, it was found that the method used to measure similarity within the memory system needed to be more suitable for the information it was comparing. In this particular case, the architecture utilised feature maps retrieved from a neural network, and the similarity was measured using the Euclidean distance metric. However, it became evident that the Euclidean distance failed to capture essential aspects that distinguish two images as unique or similar.

Further evaluation and exploration of feature-matching techniques used in computer vision or similar fields are necessary to address this limitation and improve the similarity calculation within the memory system. These methods are specifically designed to measure image similarity by considering various visual cues and patterns. By incorporating a more sophisticated feature-matching approach, the cognitive architecture can enhance its memory system and improve the accuracy of its predictions. Therefore, in future iterations of the cognitive architecture, a more suitable method for measuring similarity within the memory system should be implemented, leveraging feature-matching techniques or computer vision algorithms. This enhancement in the similarity calculation would enable the memory system to contribute more effectively to the architecture's predictive power, leading to more accurate and reliable predictions.

In summary, while the evaluation of the cognitive architecture demonstrated its effectiveness in the specific context of the tested datasets and image sizes, it is essential to acknowledge that developing a deployable version for medical image diagnosis requires further testing and validation. Expanding the evaluation to include a broader range of pathologies and imaging modalities is crucial to ensure the architecture's robustness and adaptability in diverse clinical scenarios.

By incorporating a more diverse set of pathologies and imaging modalities into the evaluation process, the cognitive architecture can undergo rigorous testing to assess its performance across various medical conditions. This expanded evaluation will help identify potential limitations and areas for improvement, ensuring that the architecture can deliver accurate and reliable diagnoses across a broader spectrum of medical conditions.

The evaluation should also consider factors such as image acquisition parameters, image quality, and potential variations in medical imaging protocols. These factors can significantly impact the architecture's performance, and thus, it is essential to assess its robustness in handling real-world variations.

Furthermore, it is vital to conduct validation studies using independent datasets to verify the generalisability of the cognitive architecture's performance. This validation process would involve evaluating the architecture on unseen data, which can provide a more comprehensive understanding of its capabilities and limitations. It would also help validate the architecture's performance against benchmarks and standards established in the medical imaging field.

Through further testing, validation, and refinement, the cognitive architecture can be enhanced to meet the requirements of a deployable version for medical image diagnosis. This iterative process of development and improvement will ensure that the architecture can deliver accurate, reliable, and consistent diagnoses across a wide range of medical conditions, ultimately benefiting healthcare professionals and patients alike.

In conclusion, while the initial evaluation of the cognitive architecture showcased its effectiveness in the tested context, further testing, validation, and refinement are necessary to develop a deployable version for medical image diagnosis. The cognitive architecture can be refined and enhanced by expanding the evaluation to encompass a broader range of pathologies and imaging modalities, considering real-world variations, and conducting validation studies using independent datasets to provide accurate and reliable diagnoses across diverse clinical scenarios.

Chapter 6

Conclusion

Developing a cognitive architecture for medical image classification and computer-assisted diagnosis holds great promise in revolutionising healthcare by enhancing the accuracy, efficiency, and accessibility of medical imaging analysis. This thesis has demonstrated the effectiveness of a comprehensive framework that combines advanced machine learning algorithms, deep neural networks, and cognitive principles to process and interpret medical images.

The proposed cognitive architecture leverages the power of artificial intelligence to surpass traditional image classification techniques by emulating human cognitive processes. It integrates domain-specific knowledge, context awareness, and reasoning capabilities, resulting in a higher level of understanding and more accurate diagnoses.

The cognitive architecture is adaptable and scalable, handling diverse medical imaging modalities and accommodating technological advancements. Its versatility allows for its application across various medical specialities, benefiting radiologists, clinicians, and patients.

Robust validation and evaluation methodologies have been highlighted to ensure the reliability and clinical applicability of the cognitive architecture. Rigorous testing, benchmarking, and comparative analysis have validated its performance, showcasing significant accuracy, speed, and diagnostic precision improvements.

Moreover, the cognitive architecture effectively addresses fundamental challenges in medical image analysis, including data variability, limited labelled datasets, and deep learning model interpretability. Incorporating explainability techniques provides insights into decision-making, enabling clinicians to trust and understand the diagnostic outcomes.

Furthermore, developing a cognitive architecture for medical image classification and computer-assisted diagnosis opens new avenues for interdisciplinary research and collaboration within

medical imaging. It serves as a foundation for innovative approaches, bringing together medical, computer science, and cognitive science experts to leverage artificial intelligence in healthcare.

While acknowledging the immense potential of developing a cognitive architecture for medical image classification and computer-assisted diagnosis, it is essential to address its limitations and challenges. Integrating such a complex system into clinical workflows requires careful consideration of regulatory compliance, ethical considerations, and legal implications. Safeguarding patient privacy, ensuring algorithm transparency, and establishing robust validation protocols are crucial for responsible deployment in clinical settings.

Future research can focus on refining the cognitive architecture by incorporating different cognitive processes, such as natural language understanding and context awareness, to enhance interpretability and communication capabilities. Exploring federated learning approaches to leverage decentralised data sources while maintaining privacy and security can advance scalability and effectiveness.

In conclusion, developing a cognitive architecture for medical image classification and computer-assisted diagnosis represents a significant milestone in medical imaging. Its potential to revolutionise healthcare, improve diagnostic accuracy, and enhance patient outcomes is immense. Continued research, collaboration, and responsible implementation can reshape the medical imaging landscape, ushering in a future where artificial intelligence plays a vital role in precision medicine and patient care.

6.1 Future Directions and Challenges

While developing a cognitive architecture for medical image classification and computer-assisted diagnosis has shown remarkable progress, several avenues for future work merit exploration. These directions aim to address existing challenges, refine the architecture, and maximise its potential impact on healthcare.

1. Continual Learning and Adaptability

One crucial aspect of the future development of cognitive architectures is incorporating continual learning mechanisms. It can stay up-to-date with the evolving nature of diseases, imaging techniques, and treatment options by enabling the architecture to adapt and learn from new data and emerging medical knowledge. Continual learning techniques, such as incremental and transfer learning, can be explored to ensure the architecture's adaptability and long-term effectiveness.

For this, transfer learning techniques could be used to enable the bottom-level structure of the

architecture to adapt to new datasets or new data. Additionally, a top-down learning approach could be added, where correct predictions from the system (verified by a professional), could be used in run-time to retrain the Computation System according to this new information.

2. Integration of Multimodal Data

Expanding the cognitive architecture's capabilities to handle multimodal data can enhance its diagnostic accuracy and comprehensive understanding of medical images. Integrating information from multiple imaging modalities, such as combining magnetic resonance imaging (MRI) and positron emission tomography (PET), can provide a more complete view of a patient's condition. Incorporating non-imaging data, such as clinical notes or genomics data, can further augment the architecture's diagnostic capabilities.

Transfer learning can also be leveraged to facilitate the integration of multimodal data. By pre-training the architecture on existing multimodal datasets, it can learn representations that capture the shared information across different modalities, allowing it to process and interpret diverse data types effectively.

3. Explainability and Trustworthiness

Enhancing the explainability and interpretability of the cognitive architecture is crucial for building trust and acceptance among healthcare professionals. Future research should focus on developing techniques that provide transparent and interpretable explanations for the architecture's decision-making processes. This would allow clinicians to understand and validate the reasoning behind the generated diagnoses, ultimately leading to increased confidence in the system's recommendations.

Specifically, research efforts can be directed towards designing post-hoc interpretability methods to provide insights into the architecture's internal representations and decision-making processes. Techniques such as feature visualisation can help highlight the regions of interest and important features in medical images that contribute to the architecture's predictions. Developing algorithms that quantify uncertainty or confidence in the architecture's outputs can provide further insights into its reliability and trustworthiness.

4. Clinical Validation and Integration

Rigorous clinical validation studies are essential to facilitate the adoption of cognitive architectures. Future research should involve large-scale, multi-centre studies involving diverse patient populations to assess the architecture's performance, generalisability, and real-world clinical utility. Collaborations with healthcare institutions and stakeholders are crucial to ensure seamless integration into existing clinical workflows, adherence to regulatory standards, and compliance with ethical guidelines.

To enable comprehensive clinical validation, it is essential to establish standardised evaluation

protocols and benchmarks. These protocols should include appropriate metrics for assessing the architecture's performance, such as sensitivity, specificity, accuracy, and area under the receiver operating characteristic curve (AUC-ROC). Additionally, the architecture should be tested on diverse patient populations to evaluate its robustness and generalisability across different demographics, diseases, and imaging setups.

5. Human-AI Collaboration and User Experience

Exploring effective strategies for human-AI collaboration and optimising the user experience are essential areas for future work. Designing user interfaces and interactive visualisation tools that facilitate seamless clinician interaction and cognitive architectures can enhance usability and workflow integration. Understanding the needs, expectations, and concerns of end-users through user-centred design approaches will be crucial for maximising the architecture's acceptance and usability in clinical practice.

To improve human-AI collaboration, the architecture should be designed to provide accurate predictions, actionable insights, and decision support. The user interface should present the architecture's outputs intuitively and interpretably, making it easier for clinicians to understand and integrate the information into their decision-making process. Furthermore, iterative feedback loops should be established to gather user feedback and refine the architecture based on real-world usage scenarios, providing continuous learning characteristics.

In conclusion, future research endeavours should address the challenges mentioned above and explore new directions to advance the development of cognitive architectures for medical image classification and computer-assisted diagnosis. By refining the architecture's capabilities, ensuring regulatory compliance, and fostering interdisciplinary collaborations, the full potential of this technology can be unlocked, leading to improved patient care, enhanced diagnostic accuracy, and transformative advancements in medical imaging and healthcare as a whole.

Bibliography

[1] S. H. Park and K. Han, "Methodologic Guide for Evaluating Clinical Performance and Effect of Artificial Intelligence Technology for Medical Diagnosis and Prediction," *Radiology*, vol. 286, no. 3, pp. 800–809, mar 2018. [Online]. Available: http://pubs.rsna.org/doi/10.1148/radiol.2017171920

[2] J. R. Anderson, M. Matessa, C. Lebiere, J. R. Anderson, M. Matessa, C. L. A.-r. A, J. R. Anderson, and M. Matessa, "Human – Computer Interaction ACT-R : A Theory of Higher Level Cognition and Its Relation to Visual Attention ACT-R : A Theory of Higher Level Cognition and Its Relation to Visual Attention," vol. 0024, 2009.

[3] R. Sun, P. Slusarz, and C. Terry, "The interaction of the explicit and the implicit in skill learning: A dual-process approach," *Psychological Review*, vol. 112, no. 1, pp. 159–192, 2005.

[4] P. Laird, J. E., & Rosenbloom, "The evolution of the Soar cognitive architecture," *Mind matters A tribute to Allen Newell*, no. August 1987, pp. 1–50, 1996. [Online]. Available: http://books.google.com/books?hl=en{&}lr={&}id=3D-KX8vZNccC{&}oi=fnd{&}pg=PA1{&}dq=evolution+of+the+Soar+cognitive+architecture{&}ots=DB3tS3WI8A{&}sig=u-FJ2HQ7wHsbCA7tNGCe8ax4lR0

[5] J. E. Laird, "Extending the soar cognitive architecture," *Frontiers in Artificial Intelligence and Applications*, vol. 171, no. 1, pp. 224–235, 2008.

[6] H. Sofian, J. C. M. Than, S. Mohammad, and N. M. Noor, "Calcification detection of coronary artery disease in intravascular ultrasound image: Deep feature learning approach," *International Journal of Integrated Engineering*, vol. 10, pp. 43–57, 2018.

[7] S. Helie and R. Sun, "How the core theory of CLARION captures human decision-making," *Proceedings of the International Joint Conference on Neural Networks*, pp. 173–180, 2011.

[8] J. R. Anderson, *How Can the Human Mind Occur in the Physical Universe?* Oxford University Press, oct 2007. [Online]. Available: https://oxford.universitypressscholarship.com/view/10.1093/acprof:oso/9780195324259.001.0001/acprof-9780195324259

[9] M. Gori, B. Machine, D. L. Model, D. N. Network, F. Extraction, N. L. Processing, and A. Recognition, "Deep Architecture."

[10] L. M. Prevedello, S. S. Halabi, G. Shih, C. C. Wu, M. D. Kohli, F. H. Chokshi, B. J. Erickson, J. Kalpathy-Cramer, K. P. Andriole, and A. E. Flanders, "Challenges Related to Artificial Intelligence Research in Medical Imaging and the Importance of Image Analysis Competitions," *Radiology: Artificial Intelligence*, vol. 1, no. 1, p. e180031, jan 2019. [Online]. Available: https://pubs.rsna.org/doi/abs/10.1148/ryai.2019180031

[11] S. Deepa and B. Aruna Devi, "A survey on artificial intelligence approaches for medical image classification," *Indian Journal of Science and Technology*, vol. 4, no. 11, 2011. [Online]. Available: http://www.indjst.org

[12] A. N. Ramesh, C. Kambhampati, J. R. Monson, and P. J. Drew, "Artificial intelligence in medicine," pp. 334–338, sep 2004. [Online]. Available: /pmc/articles/PMC1964229/?report=abstracthttps://www.ncbi.nlm.nih.gov/pmc/articles/PMC1964229/

[13] S. Pacilè, J. Lopez, P. Chone, T. Bertinotti, J. M. Grouin, and P. Fillard, "Improving Breast Cancer Detection Accuracy of Mammography with the Concurrent Use of an Artificial Intelligence Tool," *Radiology: Artificial Intelligence*, vol. 2, no. 6, p. e190208, nov 2020. [Online]. Available: https://pubs.rsna.org/doi/abs/10.1148/ryai.2020190208

[14] P. Lakhani and B. Sundaram, "Deep learning at chest radiography: Automated classification of pulmonary tuberculosis by using convolutional neural networks," *Radiology*, vol. 284, no. 2, pp. 574–582, aug 2017.

[15] S. Dutta and P. P. Bonissone, "Integrating case- and rule-based reasoning," *International Journal of Approximate Reasoning*, vol. 8, no. 3, 1993.

[16] R. Saraiva, M. Perkusich, L. Silva, H. Almeida, C. Siebra, and A. Perkusich, "Early diagnosis of gastrointestinal cancer by using case-based and rule-based reasoning," *Expert Systems with Applications*, vol. 61, pp. 192–202, nov 2016.

[17] F. Pesapane, C. Volonté, M. Codari, and F. Sardanelli, "Artificial intelligence as a medical device in radiology: ethical and regulatory issues in Europe and the United States," pp. 745–753, oct 2018. [Online]. Available: https://link.springer.com/articles/10.1007/s13244-018-0645-yhttps://link.springer.com/article/10.1007/s13244-018-0645-y

[18] C. C. Weems, S. P. Levitan, A. R. Hanson, E. M. Riseman, D. B. Shu, and J. G. Nash, "The image understanding architecture," *International Journal of Computer Vision*, vol. 2, no. 3, pp. 251–282, jan 1989. [Online]. Available: https://link.springer.com/article/10.1007/BF00158166

[19] C. Wang, N. Komodakis, and N. Paragios, "Markov Random Field modeling, inference & learning in computer vision & image understanding: A survey," *Computer Vision and Image Understanding*, vol. 117, no. 11, pp. 1610–1627, nov 2013.

[20] N. Zheng, G. Loizou, X. Jiang, X. Lan, and X. Li, "Preface: Computer vision and pattern recognition," pp. 1265–1266, sep 2007. [Online]. Available: https://doi.org/10.1080/00207160701303912

[21] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*. International Conference on Learning Representations, ICLR, sep 2015. [Online]. Available: http://www.robots.ox.ac.uk/

[22] "The human-computer interaction handbook: Fundamentals, evolving technologies ... - google livros." https://books.google.pt/books?hl=pt-PT&lr=&id=A8TPF_O385AC&oi=fnd&pg=PA93&dq=cognitive+architecture&ots=fnSJIVlRYc&sig=h-RqRvV777DaqE0rfuljHPN4-sE&redir_esc=y#v=onepage&q=cognitivearchitecture&f=false, accessed Jan. 03, 2021.

[23] R. Sun, "Desiderata for cognitive architectures," *Philosophical Psychology*, vol. 17, pp. 341–373, 2004.

[24] J. W. Tweedale, "A review of cognitive decision-making within future mission systems," in *Procedia Computer Science*, vol. 35, no. C. Elsevier B.V., jan 2014, pp. 1043–1052.

[25] A. Chella, M. Frixione, and S. Gaglio, "A cognitive architecture for artificial vision," *Artificial Intelligence*, vol. 89, no. 1-2, pp. 73–111, jan 1997.

[26] J. E. Laird, K. R. Kinkade, S. Mohan, and J. Z. Xu, "Cognitive Robotics using the Soar Cognitive Architecture," Tech. Rep., 2012. [Online]. Available: www.aaai.org

[27] J. R. Anderson, "ACT: A simple theory of complex cognition." *American Psychologist*, vol. 51, no. 4, pp. 355–365, 1996. [Online]. Available: http://doi.apa.org/getdoi.cfm?doi=10.1037/0003-066X.51.4.355

[28] J. P. Borst and J. R. Anderson, "A step-by-step tutorial on using the cognitive architecture ACT-R in combination with fMRI data," *Journal of Mathematical Psychology*, vol. 76, 2017.

[29] S. Hélie, N. Wilson, and R. Sun, "The CLARION Cognitive Architecture : A Tutorial," *Proceedings of the 30th Annual Meeting of the Cognitive Science Society*, pp. 9–10, 2008.

[30] R. Sun, "The importance of cognitive architectures: An analysis based on CLARION," *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 19, no. 2, pp. 159–193, 2007.

[31] F. Toates, "Motivational systems," 1986. [Online]. Available: https://books.google.com/books?hl=pt-PT{&}lr={&}id=JVg7AAAAIAAJ{&}oi=fnd{&}pg=PR9{&}dq=+F.+Toates{&}ots=gMWfsDu3nn{&}sig=X{_}1Dq3ucuP4ClPkcVP776j3QfjE

[32] R. Sun, "A Tutorial on CLARION 5.0," vol. 66, pp. 37–39, 2012.

[33] R. Sun, T. Peterson, and C. Sessions, "Beyond simple rule extraction: Acquiring planning knowledge from neural networks," pp. 288–300, 2002.

[34] R. Sun and C. Giles, *Sequence learning: Paradigms, algorithms, and applications*, 2003.

[35] J. Lehman, J. Laird, and P. Rosenbloom, "A gentle introduction to Soar, an architecture for human cognition," *Invitation to cognitive science*, vol. 4, no. May 2014, pp. 212–249, 1996.

[36] A. Newell, "Unified theories of cognition," Tech. Rep., 1994. [Online]. Available: https://www.google.com/books?hl=pt-PT{&}lr={&}id=1IbY14DmV2cC{&}oi=fnd{&}pg=PA1{&}dq=allen+newell+unified+theories+of+cognition+mendeley{&}ots=odLt00C{_}Jc{&}sig=JokBLYWxEGTs{_}NBxFKJWPXX5T6E

[37] A. M. Nuxoll and J. E. Laird, "Extending cognitive architecture with episodic memory," *Proceedings of the National Conference on Artificial Intelligence*, vol. 2, pp. 1560–1565, 2007.

[38] R. M. Young and R. L. Lewis, "The Soar Cognitive Architecture and Human Working Memory," *Models of Working Memory*, pp. 224–256, 2012.

[39] J. E. Laird, *The Soar Cognitive Architecture*, 2018. [Online]. Available: https://direct.mit.edu/books/book/2938/The-Soar-Cognitive-Architecture

[40] J. Y. Puigbo, A. Pumarola, C. Angulo, and R. Tellez, "Using a cognitive architecture for general purpose service robot control," *Connection Science*, vol. 27, no. 2, pp. 105–117, 2015. [Online]. Available: https://doi.org/10.1080/09540091.2014.968093

[41] S. Zhong, H. Ma, L. Zhou, X. Wang, S. Ma, and N. Jia, "Guidance Compliance Behavior on VMS Based on SOAR Cognitive Architecture," *Mathematical Problems in Engineering*, vol. 2012, pp. 1–21, 2012. [Online]. Available: http://www.hindawi.com/journals/mpe/2012/530561/

[42] D. F. Lucentini and R. R. Gudwin, "A comparison among cognitive architectures: A theoretical analysis," *Procedia Computer Science*, vol. 71, pp. 56–61, 2015.

[43] C. Lemke, M. Budka, and B. Gabrys, "Metalearning: a survey of trends and technologies," *Artificial Intelligence Review*, vol. 44, no. 1, pp. 117–130, jun 2015. [Online]. Available: /pmc/articles/PMC4459543/?report=abstracthttps://www.ncbi.nlm.nih.gov/pmc/articles/PMC4459543/

[44] C. Giraud-Carrier, "Metalearning-a tutorial," *Tutorial at the 2008 International Conference on Machine Learning and Applications (ICMLA)*. [Online]. Available: https://www.academia.edu/2621381/Metalearning{_}a{_}tutorial

[45] V. J, *Understanding Machine Learning Performance with Experiment Databases*, 2010, no. May.

[46] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *34th International Conference on Machine Learning, ICML 2017*, vol. 3. International Machine Learning Society (IMLS), mar 2017, pp. 1856–1868. [Online]. Available: https://arxiv.org/abs/1703.03400v3

[47] J. Snell, K. Swersky, and T. R. Zemel, "Prototypical Networks for Few-shot Learning," Tech. Rep.

[48] S. Ravi and H. Larochelle, "OPTIMIZATION AS A MODEL FOR FEW-SHOT LEARNING," Tech. Rep.

[49] H. Wei, H. Jia, Y. Li, and Y. Xu, "Verify and measure the quality of rule based machine leaning," *Knowledge-Based Systems*, vol. 205, p. 106300, oct 2020. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S095070512030472X

[50] A. Majkowska, S. Mittal, D. F. Steiner, J. J. Reicher, S. M. McKinney, G. E. Duggan, K. Eswaran, P. H. C. Chen, Y. Liu, S. R. Kalidindi, A. Ding, G. S. Corrado, D. Tse, and S. Shetty, "Chest radiograph interpretation with deep learning models: Assessment with radiologist-adjudicated reference standards and population-adjusted evaluation," *Radiology*, vol. 294, no. 2, pp. 421–431, 2020.

[51] A. K. Jaiswal, P. Tiwari, S. Kumar, D. Gupta, A. Khanna, and J. J. Rodrigues, "Identifying pneumonia in chest X-rays: A deep learning approach," *Measurement: Journal of the International Measurement Confederation*, vol. 145, pp. 511–518, oct 2019.

[52] S. Chilamkurthy, R. Ghosh, S. Tanamala, M. Biviji, N. G. Campeau, V. K. Venugopal, V. Mahajan, P. Rao, and P. Warier, "Deep learning algorithms for detection of critical findings in head CT scans: a retrospective study," *The Lancet*, vol. 392, no. 10162, pp. 2388–2396, dec 2018.

[53] "Automated chest ct scan analysis with deep learning classifier." https://www.rsipvision.com/automated-chest-ct-scan-analysis/, accessed Jan. 03, 2021.

[54] "Artificial intelligence and machine learning for x-ray imaging - esr connect." https://connect.myesr.org/course/artificial-intelligence-and-machine-learning-for-x-ray-imaging/, accessed Jan. 03, 2021.

[55] J. Gao, Y. Yang, P. Lin, and D. S. Park, "Editorial: Computer vision in healthcare applications," *Journal of Healthcare Engineering*, vol. 2018, 2018.

[56] A. Esteva, K. Chou, S. Yeung, N. Naik, A. Madani, A. Mottaghi, Y. Liu, E. Topol, J. Dean, and R. Socher, "Deep learning-enabled medical computer vision," *npj Digital Medicine*, vol. 4, 12 2021.

[57] A. Criminisi and J. Shotton, "Decision forests for computer vision and medical image analysis," 2013. [Online]. Available: http://link.springer.com/10.1007/978-1-4471-4929-3

[58] B. Coifman, D. Beymer, P. Mclauchlan, and J. Malik, "A real-time computer vision system for vehicle tracking and tracking surveillance," 1998.

[59] B. Bhanu and A. Kumar, "Deep learning for biometrics," 2017. [Online]. Available: http://link.springer.com/10.1007/978-3-319-61657-5

[60] G. Papari and N. Petkov, "Edge and line oriented contour detection: State of the art," *Image and Vision Computing*, vol. 29, pp. 79–103, 2011.

[61] C. Schüldt, I. Laptev, and B. Caputo, "Recognizing human actions: A local svm approach," *Proceedings - International Conference on Pattern Recognition*, vol. 3, pp. 32–36, 2004.

[62] S. Huang, C. A. Nianguang, P. P. Pacheco, S. Narandes, Y. Wang, and X. U. Wayne, "Applications of support vector machine (svm) learning in cancer genomics," *Cancer Genomics and Proteomics*, vol. 15, pp. 41–51, 1 2018.

[63] G. Guo, H. Wang, D. Bell, Y. Bi, and K. Greer, "Knn model-based approach in classification," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 2888, pp. 986–996, 2003.

[64] J. J. Bird and D. R. Faria, "A study on cnn transfer learning for image classification engagement through ai-based interactive games: Neurocognitive training for children with learning difficulties view project sim2real: From simulation to real robotic application using deep reinforcement learning and knowledge transfer view project," 2018. [Online]. Available: https://www.researchgate.net/publication/325803364

[65] Q. Li, W. Cai, X. Wang, Y. Zhou, D. D. Feng, and M. Chen, "Medical image classification with convolutional neural network," *2014 13th International Conference on Control Automation Robotics and Vision, ICARCV 2014*, pp. 844–848, 2014.

[66] J. Wang, Y. Yang, J. Mao, Z. Huang, C. Huang, and W. Xu, "Cnn-rnn: A unified framework for multi-label image classification."

[67] D. Dablain, K. N. Jacobson, C. Bellinger, M. Roberts, and N. V. Chawla, "Understanding cnn fragility when learning with imbalanced data," *Machine Learning*, pp. 1–26, 2023.

[68] K. Weiss, T. M. Khoshgoftaar, and D. D. Wang, "A survey of transfer learning," *Journal of Big Data*, vol. 3, 12 2016.

[69] C. A. Ferreira, T. Melo, P. Sousa, M. I. Meyer, E. Shakibapour, P. Costa, and A. Campilho, "Classification of breast cancer histology images through transfer learning using a pre-trained inception resnet v2," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10882 LNCS, pp. 763–770, 2018.

[70] A. S. B. Reddy and D. S. Juliet, "Transfer learning with resnet-50 for malaria cell-image classification," *Proceedings of the 2019 IEEE International Conference on Communication and Signal Processing, ICCSP 2019*, pp. 945–949, 4 2019.

[71] M. D. Zeiler, "Adadelta: An adaptive learning rate method," 12 2012. [Online]. Available: http://arxiv.org/abs/1212.5701

[72] T. Takase, S. Oyama, and M. Kurihara, "Effective neural network training with adaptive learning rate based on training loss," *Neural Networks*, vol. 101, pp. 68–78, 5 2018.

[73] L. Liu, H. Jiang, P. He, W. Chen, X. Liu, J. Gao, and J. Han, "On the variance of the adaptive learning rate and beyond," 8 2019. [Online]. Available: http://arxiv.org/abs/1908.03265

[74] Y. Lecun and Y. Bengio, "Convolutional networks for images, speech, and time-series," 1995.

[75] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, pp. 84–90, 6 2012.

[76] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," 10 2014. [Online]. Available: http://arxiv.org/abs/1410.0759

[77] O. Abdel-Hamid, A. R. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, "Convolutional neural networks for speech recognition," *IEEE Transactions on Audio, Speech and Language Processing*, vol. 22, pp. 1533–1545, 10 2014.

[78] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, "A convolutional neural network for modelling sentences," 4 2014. [Online]. Available: http://arxiv.org/abs/1404.2188

[79] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 9 2016. [Online]. Available: http://arxiv.org/abs/1609.02907

[80] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," 5 2015. [Online]. Available: http://arxiv.org/abs/1505.04597

[81] J. Zhang, Z. Jiang, J. Dong, Y. Hou, and B. Liu, "Attention gate resu-net for automatic mri brain tumor segmentation," *IEEE Access*, vol. 8, pp. 58 533–58 545, 2020.

[82] O. Oktay, J. Schlemper, L. L. Folgoc, M. Lee, M. Heinrich, K. Misawa, K. Mori, S. McDonagh, N. Y. Hammerla, B. Kainz, B. Glocker, and D. Rueckert, "Attention u-net: Learning where to look for the pancreas," 4 2018. [Online]. Available: http://arxiv.org/abs/1804.03999

[83] J. Schlemper, O. Oktay, L. Chen, J. Matthew, C. Knight, B. Kainz, B. Glocker, and D. Rueckert, "Attention-gated networks for improving ultrasound scan plane detection," 2018.

[84] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2016. [Online]. Available: http://image-net.org/challenges/LSVRC/2015/

[85] D. A. Waterman, J. Paul, and M. Peterson, "Expert systems for legal decision making," *Expert Systems*, vol. 3, no. 4, pp. 212–226, 1986.

[86] E. A. Feigenbaum, "Expert systems in the 1980s," *State of the art report on machine intelligence. Maidenhead: Pergamon-Infotech*, 1981.

[87] D. Michie, "Current developments in expert systems," in *Proceedings of the Second Australian Conference on Applications of expert systems*, 1987, pp. 137–156.

[88] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, pp. 81–106, 1986.

[89] L. Rokach and O. Maimon, "Decision trees," 2015.

[90] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, "Classification and regression trees (wadsworth, belmont, ca)," *ISBN-13*, pp. 978–0 412 048 418, 1984.

[91] J. R. Quinlan and R. L. Rivest, "Inferring decision trees using the minimum description lenght principle," *Information and computation*, vol. 80, no. 3, pp. 227–248, 1989.

[92] J. Mingers, "An empirical comparison of pruning methods for decision tree induction," *Machine learning*, vol. 4, pp. 227–243, 1989.

[93] G. Biau and E. Scornet, "A random forest guided tour," *Test*, vol. 25, pp. 197–227, 2016.

[94] M. Belgiu and L. Dra□gut□, "Random forest in remote sensing: A review of applications and future directions," *ISPRS journal of photogrammetry and remote sensing*, vol. 114, pp. 24–31, 2016.

[95] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.

[96] J. R. Quinlan, *C4. 5: programs for machine learning*. Elsevier, 2014.

[97] B. Hssina, A. Merbouha, H. Ezzikouri, and M. Erritali, "A comparative study of decision tree id3 and c4. 5," *International Journal of Advanced Computer Science and Applications*, vol. 4, no. 2, pp. 13–19, 2014.

[98] S. Singh and P. Gupta, "Comparative study id3, cart and c4. 5 decision tree algorithm: a survey," *International Journal of Advanced Information Science and Technology (IJAIST)*, vol. 27, no. 27, pp. 97–103, 2014.

[99] R. J. Lewis, "An introduction to classification and regression tree (cart) analysis," in *Annual meeting of the society for academic emergency medicine in San Francisco, California*, vol. 14. Citeseer, 2000.

[100] H. Eichenbaum, "Memory: organization and control," *Annual review of psychology*, vol. 68, pp. 19–45, 2017.

[101] L. R. Squire, "Memory and brain systems: 1969–2009," *Journal of Neuroscience*, vol. 29, no. 41, pp. 12 711–12 716, 2009.

[102] A. Baddeley, "Working memory: Theories, models, and controversies," *Annual review of psychology*, vol. 63, pp. 1–29, 2012.

[103] E. R. Kandel, "The molecular biology of memory storage: a dialogue between genes and synapses," *Science*, vol. 294, no. 5544, pp. 1030–1038, 2001.

[104] P. Goyal and E. Ferrara, "Graph embedding techniques, applications, and performance: A survey," *Knowledge-Based Systems*, vol. 151, pp. 78–94, 2018.

[105] J. M. Gardiner, B. Gawlik, and A. Richardson-Klavehn, "Maintenance rehearsal affects knowing, not remembering; elaborative rehearsal affects remembering, not knowing," *Psychonomic Bulletin & Review*, vol. 1, pp. 107–110, 1994.

[106] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[107] G. Montavon, G. Orr, and K.-R. Müller, *Neural networks: tricks of the trade*. springer, 2012, vol. 7700.

[108] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010: 19th International Conference on Computational StatisticsParis France, August 22-27, 2010 Keynote, Invited and Contributed Papers*. Springer, 2010, pp. 177–186.

[109] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[110] R. Sun and X. Zhang, "Accounting for a variety of reasoning data within a cognitive architecture," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 18, no. 2, pp. 169–191, 2006.

[111] S. Candemir, S. Jaeger, K. Palaniappan, J. P. Musco, R. K. Singh, Z. Xue, A. Karargyris, S. Antani, G. Thoma, and C. J. McDonald, "Lung segmentation in chest radiographs using anatomical atlases with nonrigid registration," *IEEE transactions on medical imaging*, vol. 33, no. 2, pp. 577–590, 2013.

[112] S. Jaeger, A. Karargyris, S. Candemir, L. Folio, J. Siegelman, F. Callaghan, Z. Xue, K. Palaniappan, R. K. Singh, S. Antani *et al.*, "Automatic tuberculosis screening using chest radiographs," *IEEE transactions on medical imaging*, vol. 33, no. 2, pp. 233–245, 2013.

[113] J. Irvin, P. Rajpurkar, M. Ko, Y. Yu, S. Ciurea-Ilcus, C. Chute, H. Marklund, B. Haghgoo, R. Ball, K. Shpanskaya *et al.*, "Chexpert: A large chest radiograph dataset with uncertainty labels and expert comparison," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 590–597.

[114] X. Wang, Y. Peng, L. Lu, Z. Lu, M. Bagheri, and R. M. Summers, "Chestx-ray8: Hospital-scale chest x-ray database and benchmarks on weakly-supervised classification and localization of common thorax diseases," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 2097–2106.

[115] D. Kermany, "Labeled optical coherence tomography (oct) and chest x-ray images for classification," 2018. [Online]. Available: https://data.mendeley.com/datasets/rscbjbr9sj/2

Appendix A

Appendix

A.1 Figures

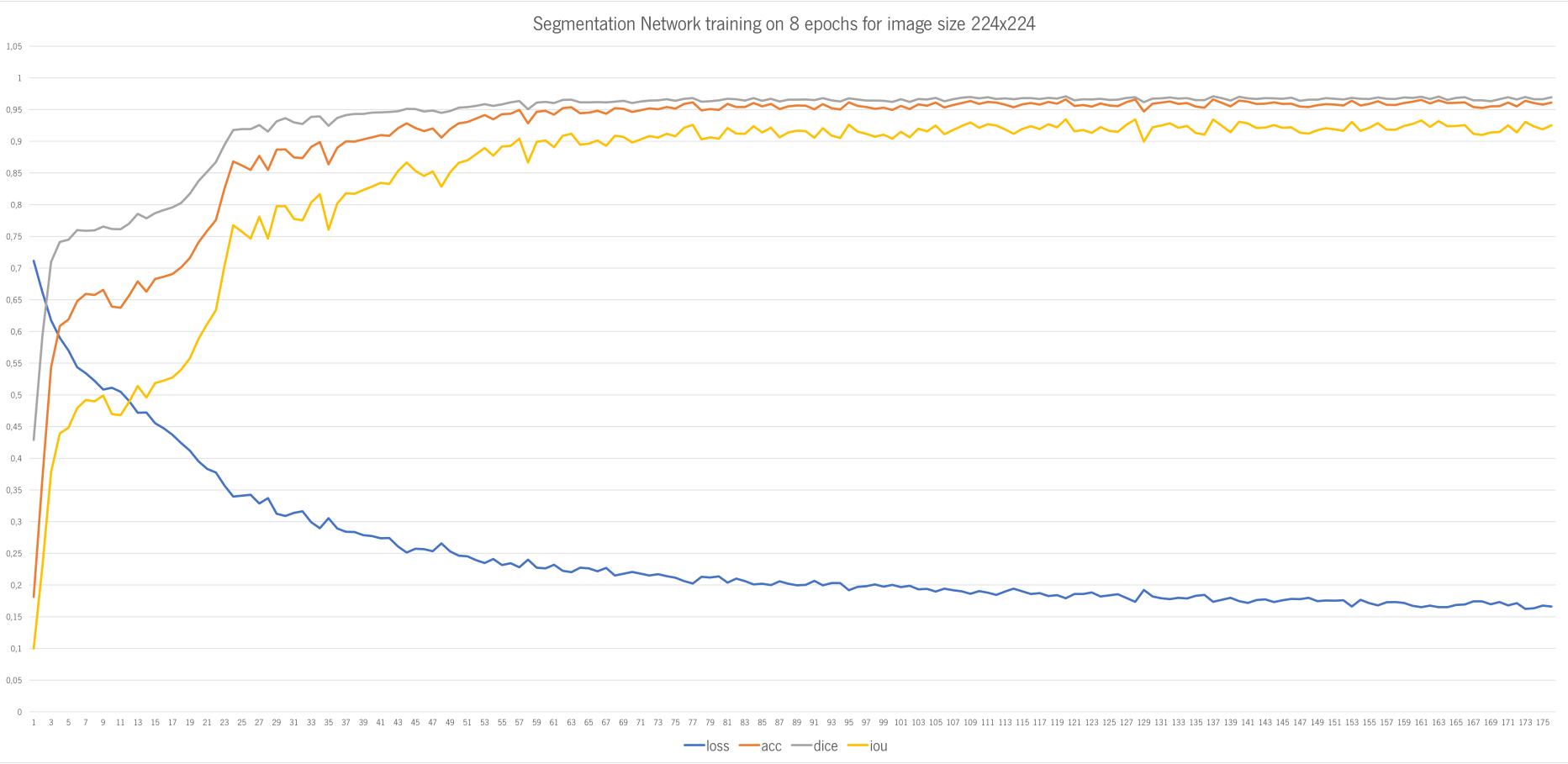


Figure 51: Segmentation metrics chart for a model trained for 8 epochs on an image size 224x224.

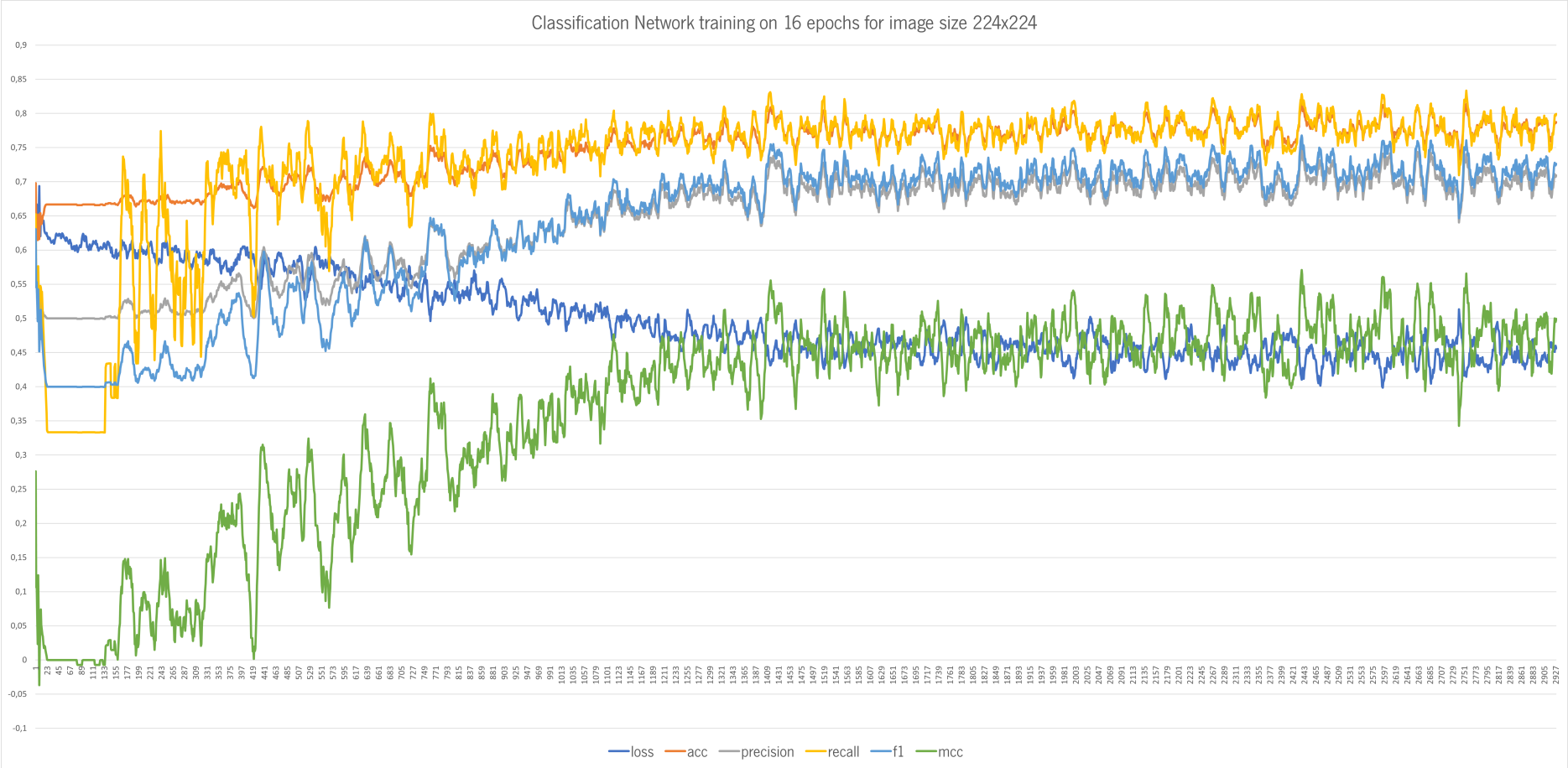


Figure 52: Classification metrics chart for a model trained for 16 epochs on an image size 224x224.

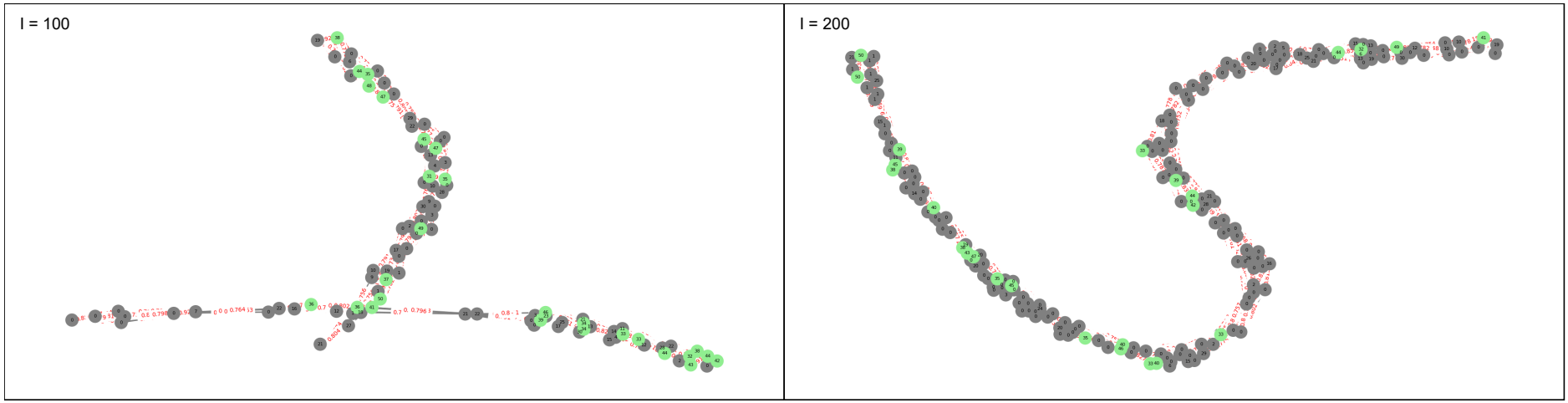


Figure 53: LTM nodes and connections with 100 and 200 images building on a 0.8 similarity threshold.

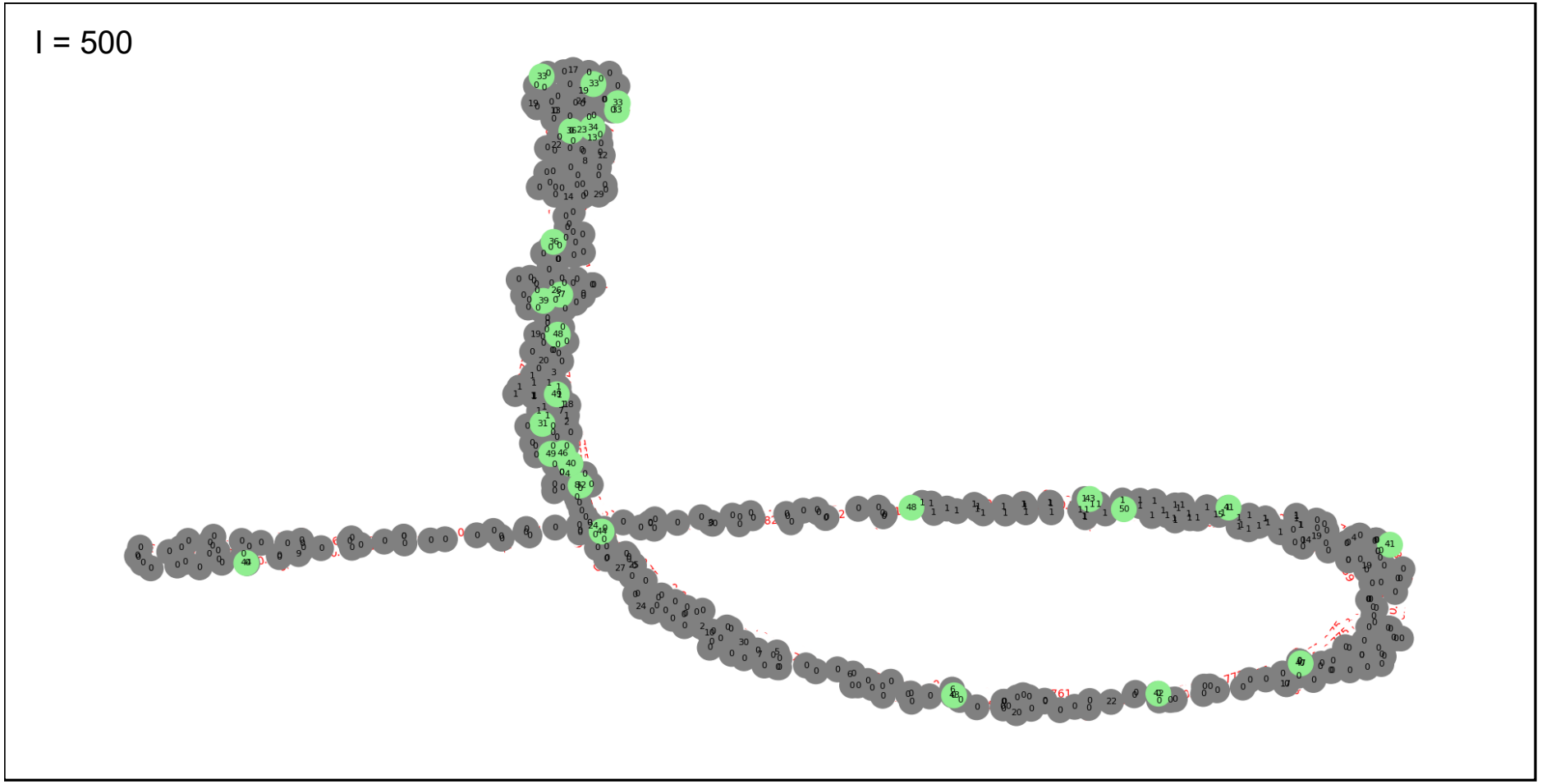


Figure 54: LTM nodes and connections with final 500 images building on a 0.8 similarity threshold.

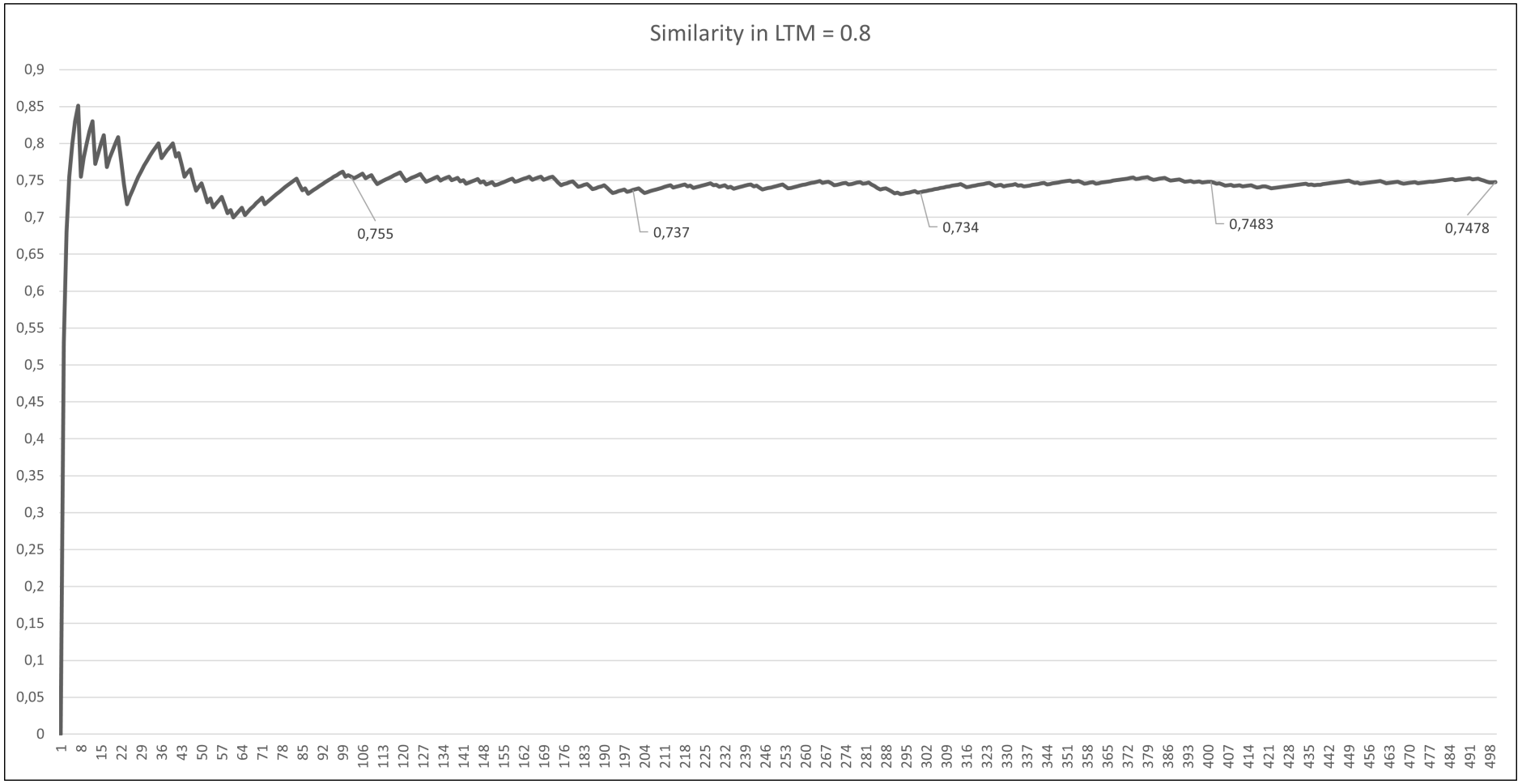


Figure 55: Accuracy evolution as more images are added to memory with similarity=0.8.

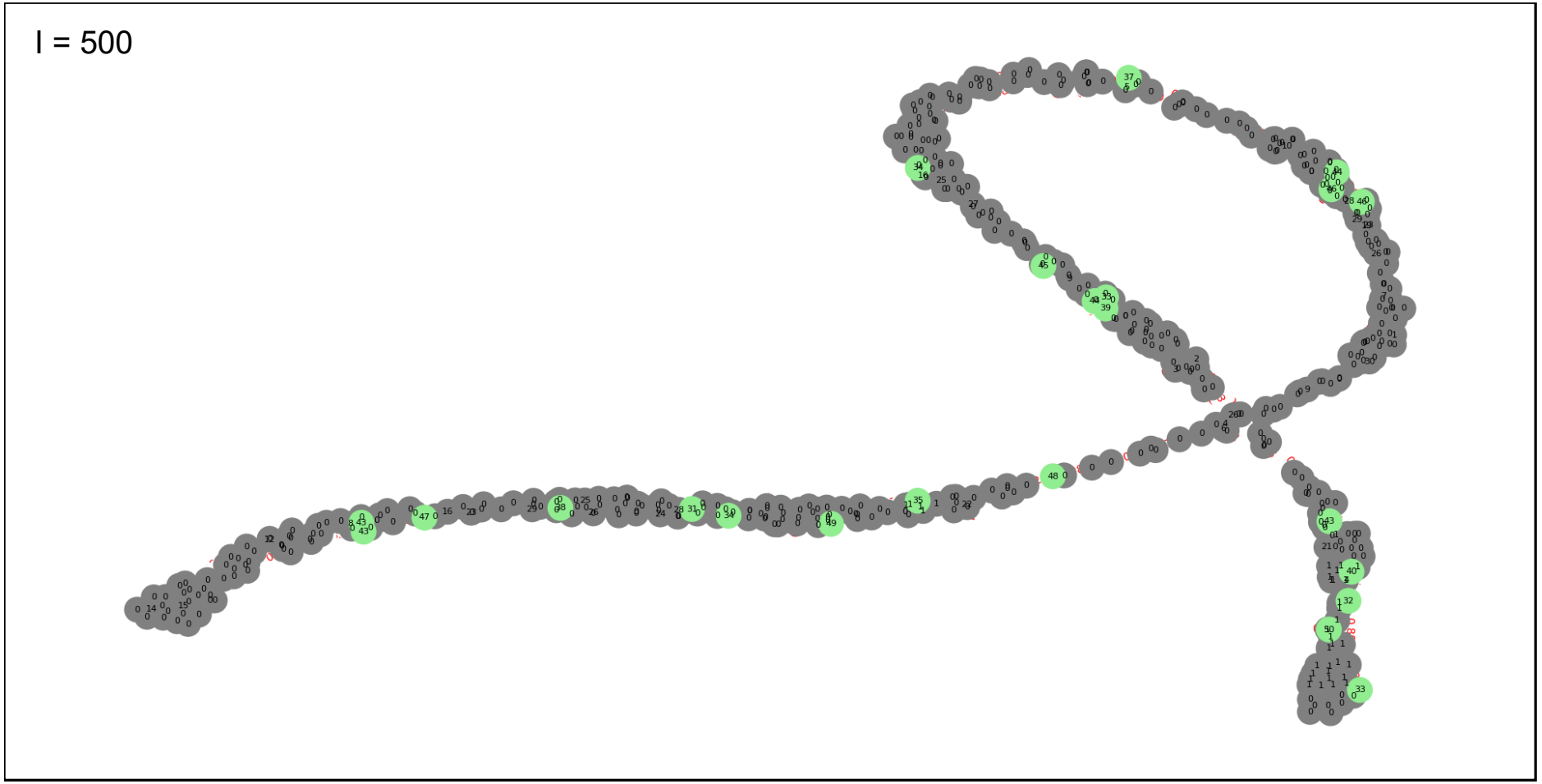


Figure 56: LTM nodes and connections with final 500 images building on a 0.85 similarity threshold.

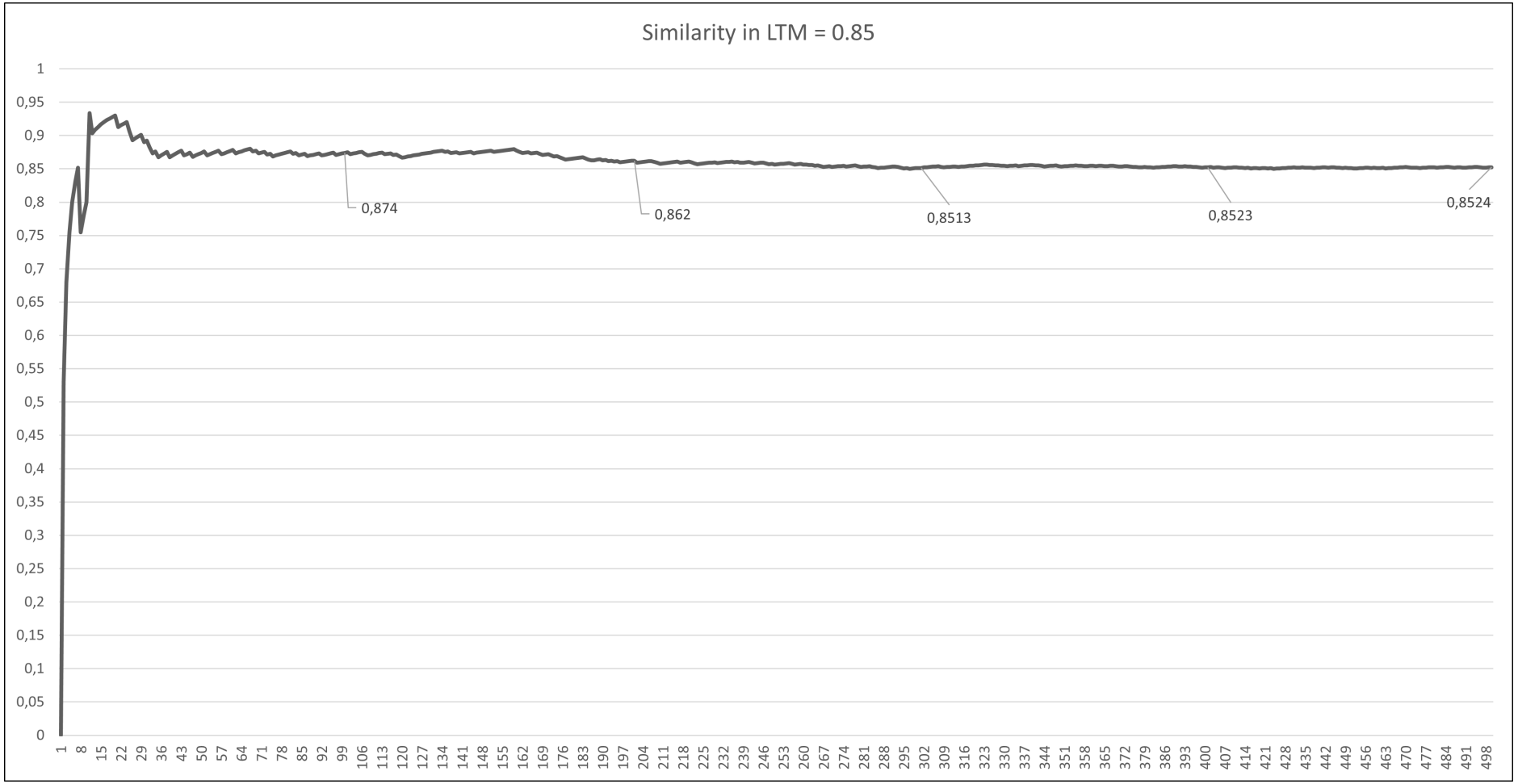


Figure 57: Accuracy evolution as more images are added to memory with similarity=0.85.

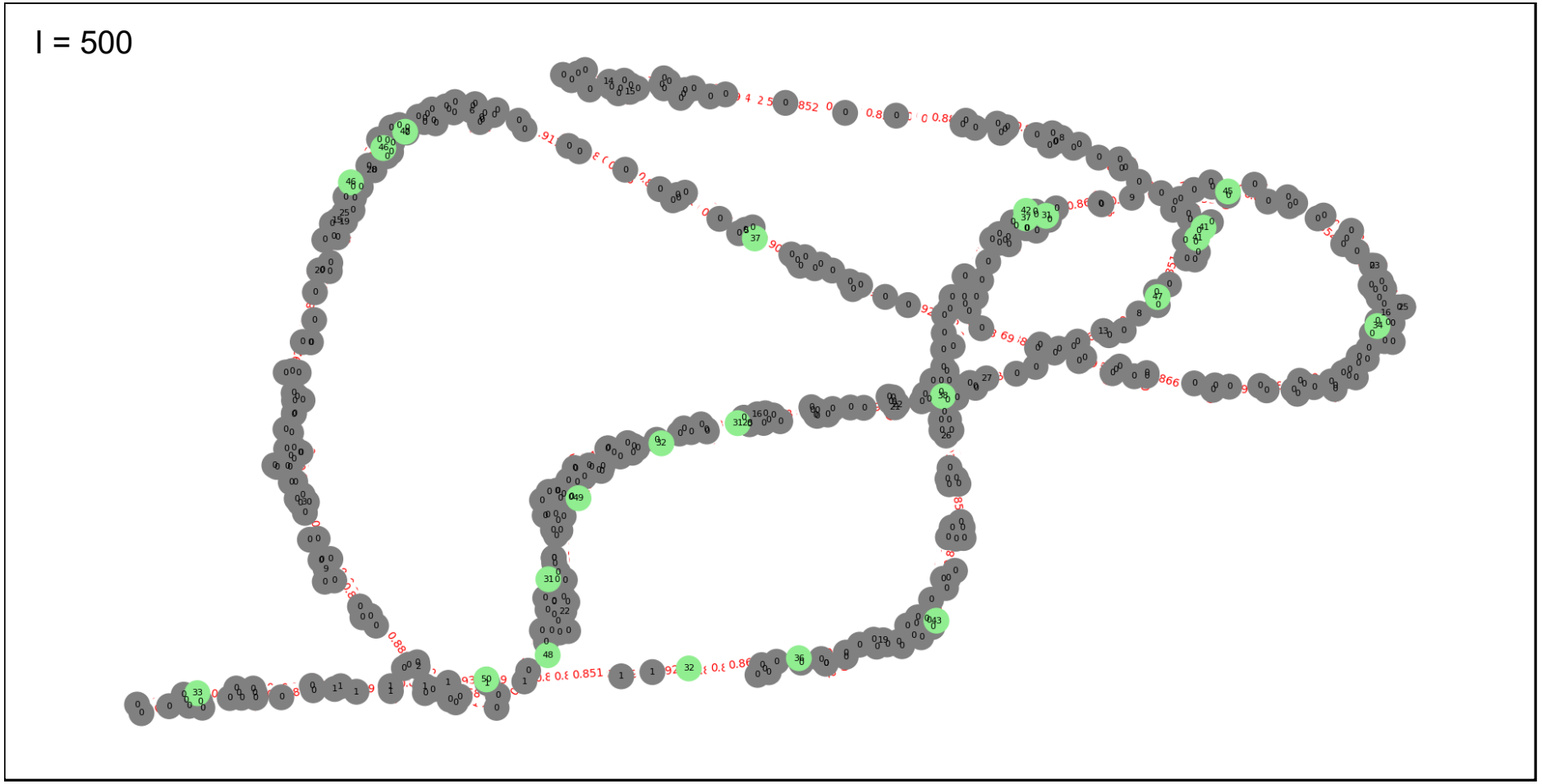


Figure 58: LTM nodes and connections with final 500 images building on a 0.9 similarity threshold.

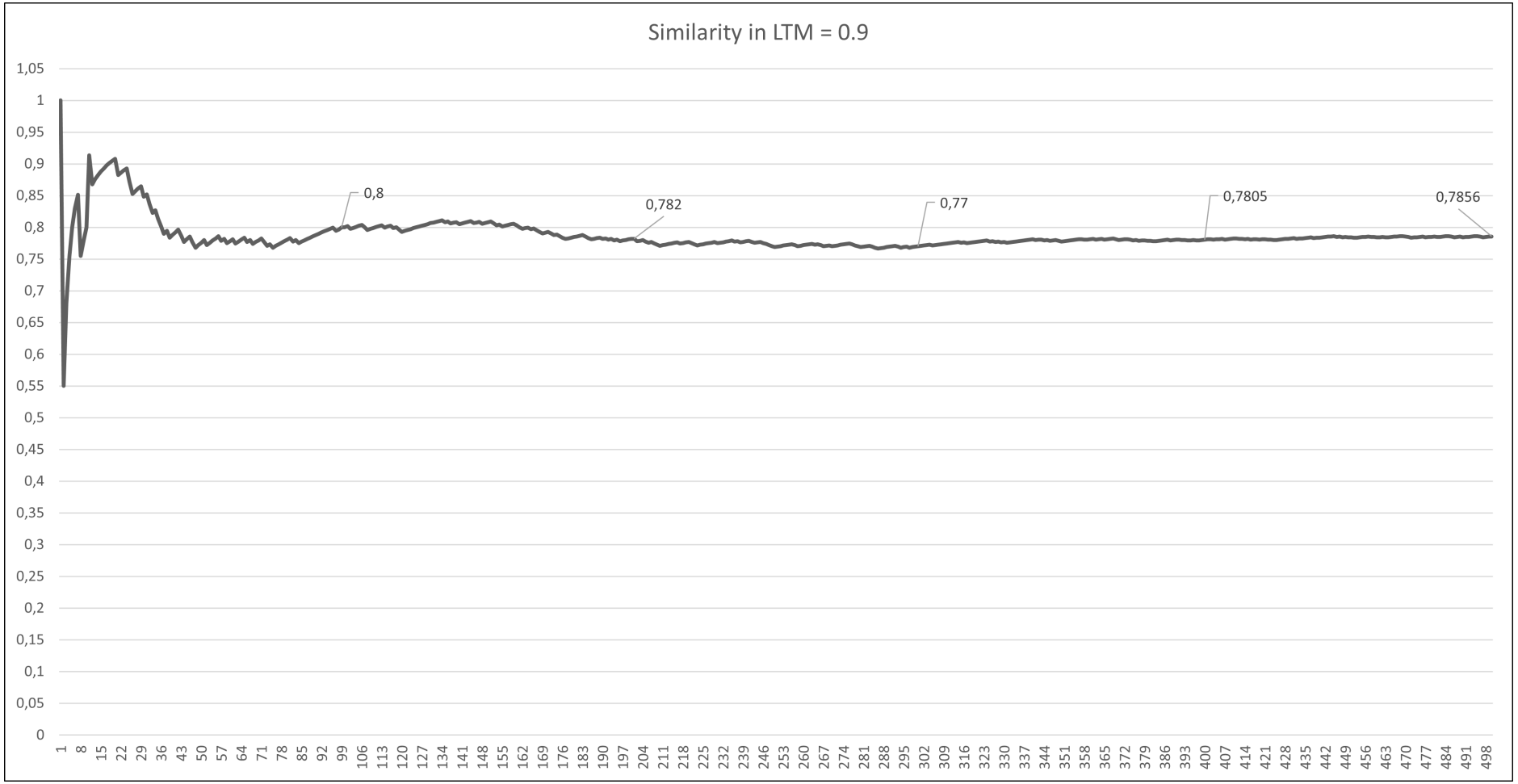


Figure 59: Accuracy evolution as more images are added to memory with similarity=0.9.

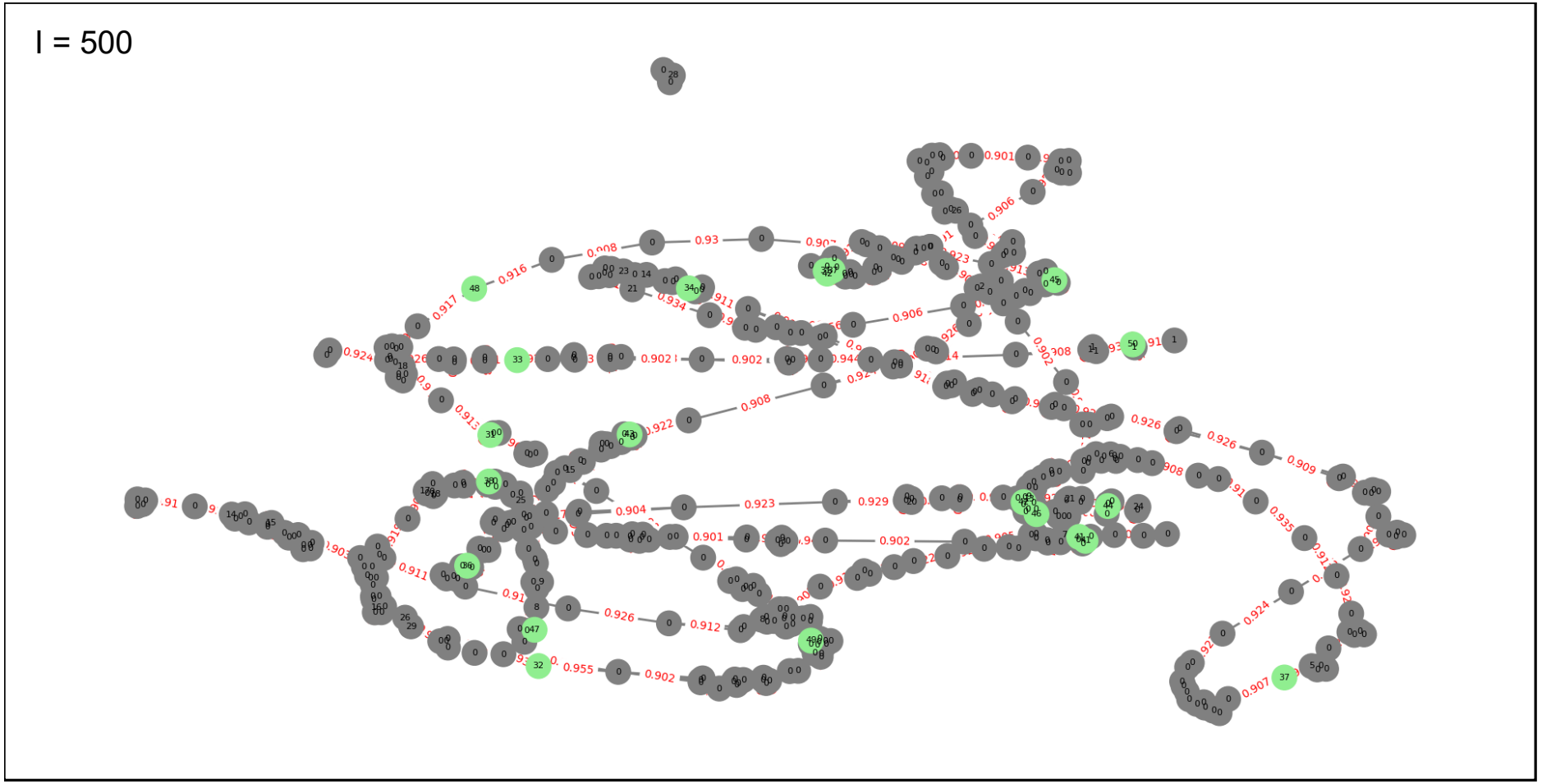


Figure 60: LTM nodes and connections with final 500 images building on a 0.95 similarity threshold.

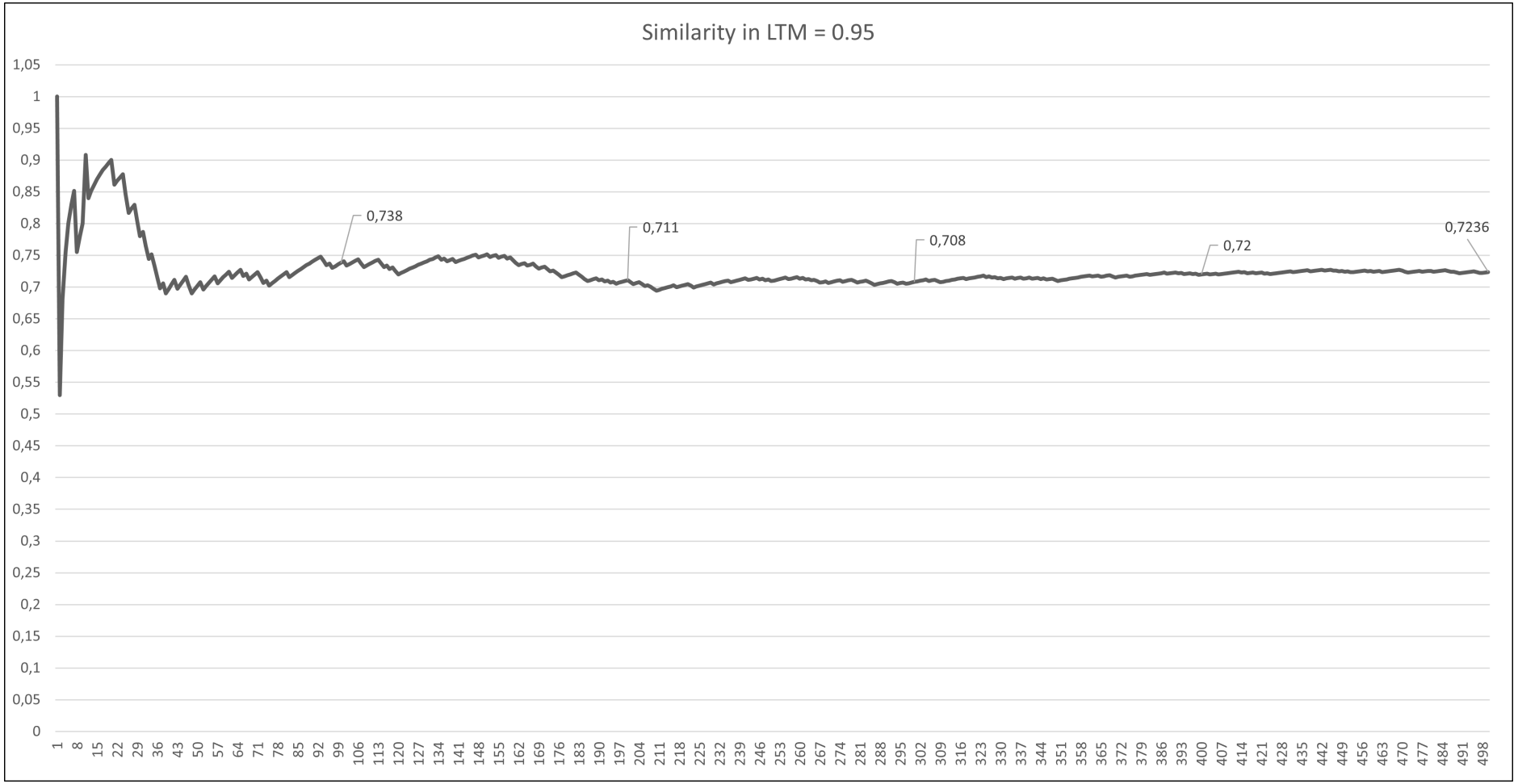


Figure 61: Accuracy evolution as more images are added to memory with similarity=0.95.

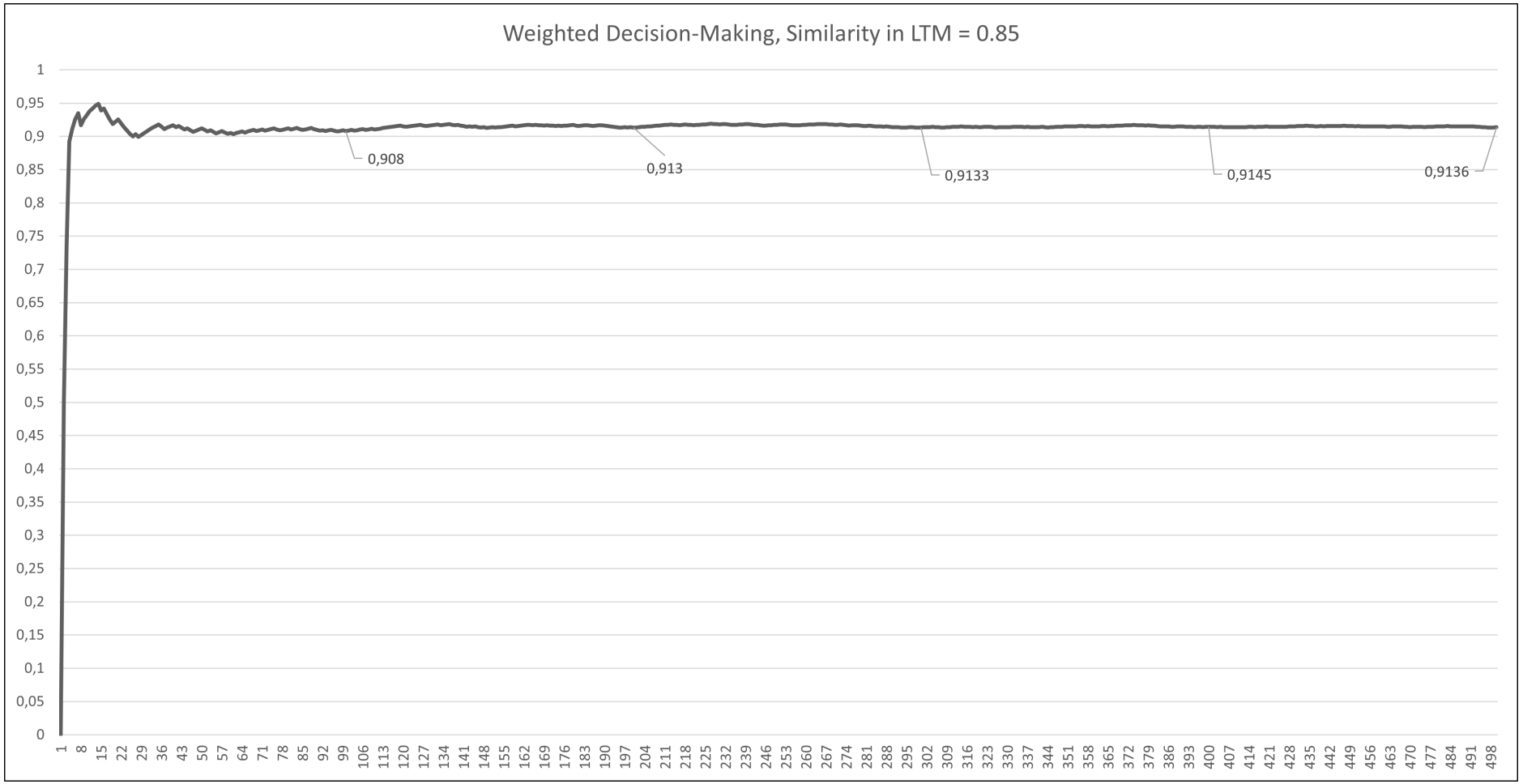


Figure 62: Accuracy evolution as more images are added to memory with similarity=0.85 and adjusted weights.