POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Enhancing Malware Detection in Executable Files using LSTM and BiLSTM-based Deep Learning Models with Word Embedding

Supervisors

Prof. Stefano DI CARLO

Prof. Ramon CANAL

Candidate

Andreu GIRONES

November 2023

Summary

In the realm of cybersecurity, the detection of malware in executable files represents a pressing challenge. Conventional signature-based methods often struggle to keep pace with evolving threats, necessitating innovative solutions. This research investigates the application of advanced machine learning techniques, specifically Long Short-Term Memory (LSTM) and Bidirectional LSTM (BiLSTM) architectures, augmented by word embedding methodologies, for robust malware detection.

The research initiates with a systematic investigation of fundamental machine learning principles and rigorous data processing methodologies, forming a robust foundation for subsequent phases. Leveraging this acquired knowledge, the study embarks on the creation and refinement of a specialized deep learning model intricately designed for the accurate detection of concealed malware within executable files.

Every aspect of model construction receives meticulous attention, encompassing data collection, preprocessing, rigorous experimentation, and the fine-tuning of hyperparameters through hyperparameter optimization (HPO).

The HPO process systematically explores and refines various model configurations. The results of this optimization process unveil the most effective model configurations, with thorough analysis of performance metrics. The evaluation of the final model provides a comprehensive assessment of its capabilities in malware detection.

In summary, this research presents an adaptive and robust deep learning model for malware detection, strengthened by LSTM and BiLSTM architectures and enriched by word embedding techniques. It offers a comprehensive account of the research process, encompassing data collection, preprocessing, hyperparameter optimization, and model evaluation. This work contributes valuable insights to the dynamic field of cybersecurity and underscores the potential of machine learning in fortifying the security of digital systems.

Acknowledgements

I am profoundly grateful to the individuals and institutions whose support and guidance have been instrumental in the completion of this thesis during the second year of my double degree program.

First and foremost, I extend my deepest appreciation to Professor Ramon Canal, who played a pivotal role as my initial supervisor during the first year. His guidance and introduction led me to Stefano Di Carlo, whose expertise and mentorship became invaluable as my supervisor during my second year at Politecnico di Torino. I am indebted to Stefano for welcoming me into the dynamic research environment of LAB 6 at the DAUIN Department of Control and Computer Engineering, where I conducted my thesis work.

I extend my heartfelt appreciation to Professor Ramon Canal for his exceptional and unwavering supervision throughout the entirety of my thesis. His continuous support and readiness to assist whenever I needed guidance were invaluable. I am particularly grateful to Professor Canal for not only guiding me through my thesis but also for providing me with the invaluable opportunity to contribute to the Vitamin-V project. Additionally, I appreciate Professor Canal for introducing me to outstanding professionals who not only demonstrated expertise but also provided invaluable insights and encouragement. I also extend my thanks to Stefano Di Carlo for his excellent supervision and warm welcome to LAB 6 at the DAUIN Department of Control and Computer Engineering, where I had the pleasure of connecting with remarkable people.

I extend my gratitude to the colleagues at LAB 6 for their warm welcome, camaraderie, and assistance throughout my time there. Their collaborative spirit significantly enriched my research experience.

Special thanks are due to Juanjo Costa, Eva Rodriguez, and Beatriz Otero, professors of the Department of Computer Architecture at UPC, for their guidance and support at various stages of my thesis. Their expertise has been invaluable in shaping the direction of my research.

I would also like to express my appreciation to the individuals at the SERT cluster at UPC who provided assistance during the execution of my project on their hardware. Their expertise and resources were crucial to the success of my research.

I extend my heartfelt thanks to my family and friends for their unwavering support and encouragement throughout my academic journey. Your belief in me has been a constant source of motivation.

Finally, I acknowledge the financial support received from the EU Horizon programme under grant GA.101093062 (Vitamin-V project). This funding played a crucial role in facilitating the completion of this thesis.

In conclusion, I am deeply thankful to all those who have contributed to this academic endeavor, and I am grateful for the collaborative and supportive environment that has shaped my research experience.

Gironés de la Fuente, Andreu

Table of Contents

List of Tables

List of Figures

Acronyms

ELF
: Executable and Linkable Format

AMD
: Advanced Micro Devices

LSTM
: Long Short-Term Memory

AI
: Artificial Intelligence

ML
: Machine Learning

RNN
: Recurrent Neural Network

CNN
: Convolutional Neural Networks

SGD
: Stochastic Gradient Descent

ADAM
: Adaptive Moment Estimation

BPTT
: Backpropagation Through Time

BiLSTM	Bidirectional Long Short-Term Memory
BRNN	Bidirectional Recurrent Neural Network
ROC	Receiver Operating Characteristic
AUC	Area Under the Curve
NLP	Natural Language Processing
CBOW	Continuous Bag of Word
SG	Skip-gram
ISA	Instruction Set Architecture
HP	Hyperparameter
HPO	Hyperparameter Optimization
HPC	Hyperparameter Configuration
MF	Multifidelity
SH	Successive Halving
HB	Hyperband

Chapter 1

Introduction

1.1 Motivation and Background

In the landscape of modern cybersecurity, the detection of malware has emerged as a critical pursuit to safeguard digital systems. Traditional signature-based methods face challenges in identifying evolving threats, prompting a shift towards innovative solutions. Among these, machine learning stands out as a promising approach, offering the potential to recognize nuanced patterns indicative of malicious activity.

Machine learning has indeed proven to be a valuable asset for enhancing malware detection capabilities. By harnessing the power of algorithms and data-driven insights, machine learning models have demonstrated their ability to discern subtle patterns, anomalies, and behaviors indicative of malicious activity. This transformative approach shifts the focus from predefined rules to adaptive learning, enabling security systems to evolve with the changing threat landscape.

This thesis delves into the realm of malware detection using machine learning models, with a specific focus on Long Short-Term Memory (LSTM) and Bidirectional LSTM (BiLSTM) architectures. LSTMs are a type of recurrent neural network known for their ability to capture temporal dependencies in sequential data. The bidirectional variant extends this capability by incorporating information from both past and future time steps, making it particularly well-suited for tasks like sequence classification and anomaly detection.

The motivation behind this research lies in the urgent need to develop robust and accurate malware detection systems. While machine learning holds immense potential, its application to cybersecurity presents unique challenges. The dynamic nature of malware, coupled with the vast diversity of attack vectors, necessitates the collection of comprehensive and representative datasets. Furthermore, effective feature extraction and preprocessing techniques are essential to enable machine learning models to discern relevant patterns from the raw data.

By incorporating LSTM and BiLSTM models, we strive to tackle these challenges. Our approach involves enhancing these recurrent neural networks with word embedding techniques to create an effective system for detecting malware within executable files. Through an exhaustive process of experimentation, hyperparameter optimization, and thorough performance evaluation, we aim to reveal the true potential of these models in accurately classifying and identifying malicious software. The outcomes of this research hold promise in strengthening our defense mechanisms against cyber threats and advancing the development of resilient and adaptive security systems.

1.2 Problem Statement

The rapid proliferation of malware poses a significant challenge to the security of digital systems. Malicious software is becoming increasingly sophisticated, employing evasive techniques to evade detection by traditional cybersecurity measures. Signature-based approaches, which rely on known patterns and attributes of malware, struggle to keep pace with the evolving tactics of cybercriminals. This dynamic landscape underscores the need for advanced and adaptable detection methods.

At the core of this research lies the central challenge: the development of robust and precise malware detection mechanisms through the utilization of machine learning models. Specifically, this inquiry delves into the application of LSTM and BiLSTM architectures, advanced neural networks designed to handle sequential data.

This pursuit entails an exploration of multifaceted dimensions. A foundational aspect involves the comprehensive grasp of fundamental machine learning concepts. This encompasses the inner workings of LSTM cells, the mathematics behind backpropagation, and the intricacies of hyperparameter optimization—a cornerstone upon which meaningful model construction is built.

Practical challenges in data acquisition also come into focus. Compiling datasets for machine learning purposes introduces complexities. The process of gathering a representative collection for training and testing is vital, yet the attainment of a diverse and legally sound dataset, particularly malicious samples, presents a significant hurdle.

In addition, the life cycle of a malware detection model mirrors the evolving threat landscape it aims to counter. The challenge is to facilitate model evolution—updating it to adapt to new attack vectors while preserving its acquired knowledge, all within the constraints of available resources.

Technical intricacies pertinent to the training of neural networks also come to the forefront. The resource-intensive nature of deep learning necessitates adept memory management. Effectively training LSTM and BiLSTM models requires a balanced orchestration of computational resources to optimize performance while adhering to memory limitations.

In summary, this thesis embarks on a comprehensive exploration of machine learning-powered malware detection models. With an in-depth understanding of machine learning concepts as a foundational bedrock, this research navigates the intricacies of data collection, model adaptation, and technical training challenges. The overarching aim is to contribute to the realm of adaptive cybersecurity, where emerging threats are met with sophisticated and responsive defenses.

1.3 Research Objectives

The primary objective of this project is to develop an effective deep learning model for the identification of malware executable files. To achieve this overarching goal, the research is structured into the following specific objectives:

1. **Comprehensive Understanding of Machine Learning and Data Processing:** Acquire a profound understanding of machine learning techniques and data processing methods as the foundational knowledge for subsequent research stages.

2. **Development of an Effective Malware Detection Model:** Create a deep learning model designed specifically for the detection of malware in executable files, including data collection, preprocessing, and model development.

3. **Hyperparameter Optimization:** Fine-tune the deep learning model's hyperparameters to enhance its performance through systematic exploration and adjustment of various model configurations.

4. **Model Evaluation:** Rigorously evaluate the final deep learning model's ability to accurately identify malware executable files using key performance metrics.

These research objectives collectively contribute to the development of a robust and accurate deep learning solution for malware detection. The subsequent chapters of this thesis detail the methodology, results, and conclusions derived from pursuing these objectives.

1.4 Thesis Outline

This thesis is structured to provide a comprehensive exploration of the development and optimization of a deep learning model for the identification of malware executable files. The following chapters outline the research journey, from the foundation laid in the literature review to the final evaluation of the proposed model:

1. **Introduction:** Introduces the research context, outlines the objectives, and provides an overview of the thesis structure.

2. **Literature Review:** Offers a concise yet comprehensive review of state-of-the-art techniques in malware detection. It presents an analysis of existing methodologies and serves as a foundation for the proposed model.

3. **Background:** Delves into the fundamental concepts and principles essential for understanding the proposed model. It provides in-depth explanations of machine learning, data processing, and other key components.

4. **Proposed Model:** Outlines the architecture and design of the proposed deep learning model specifically tailored for malware detection. Details of model structure, layers, and components are discussed.

5. **Methodology:** Presents the step-by-step methodology employed in the research. It covers data collection, data preprocessing, model training, execution, and evaluation. The chapter also elaborates on the hyperparameter optimization process.

6. **Results:** Provides a detailed analysis of the results obtained throughout the research. This includes the outcomes of hyperparameter optimization (HPO) as well as the final evaluation of the chosen model.

7. **Conclusion:** Summarizes the key findings of the research, discusses their implications, and offers recommendations for future work in the field of malware detection using deep learning models.

Chapter 2

Literature Review

In this chapter, we delve into the vast landscape of malware detection, exploring its fundamental concepts, approaches, and the evolving trends in the field. Malware, malicious software designed to compromise computer systems and data, poses significant threats to cybersecurity. As cybercriminals constantly innovate and adapt their techniques, the development of effective malware detection methods becomes paramount.

We commence by providing an overview of malware detection and its significance in safeguarding digital environments. We highlight the challenges posed by the ever-evolving nature of malware, which calls for dynamic and adaptable detection techniques. The chapter then delves into the foundational taxonomy of malware analysis, categorizing methods into static analysis, dynamic analysis, and hybrid analysis. These categories lay the groundwork for comprehending the various strategies employed in the detection of malware.

Moving forward, we explore the historical landscape of malware detection approaches. We examine primitive methods that were the precursors of more advanced techniques. Subsequently, we delve into conventional machine learning-based methods that utilize features extracted from malware samples to build detection models. We discuss the limitations of these approaches, particularly in coping with the intricacies of modern malware.

In the era of deep learning, we witness a paradigm shift in malware detection. Deep learning techniques, such as neural networks, convolutional neural networks (CNNs), and recurrent neural networks (RNNs), have revolutionized the field by autonomously learning complex patterns from raw data. We emphasize the role of these techniques in enhancing detection capabilities, even in the face of obfuscation and polymorphism.

Throughout this chapter, we navigate the ever-evolving landscape of malware detection, where researchers continuously strive to keep pace with emerging cyber threats. Our exploration will underscore the importance of a comprehensive and adaptive approach to malware detection, incorporating both historical knowledge and cutting-edge methodologies.

2.1 Introduction

In the contemporary landscape of information technology, cybersecurity has become an increasingly critical concern. The rapid expansion of digital technologies has led to a proliferation of cyber threats [1], with malicious actors employing sophisticated techniques to compromise systems, steal sensitive data, and disrupt operations. Among the various cybersecurity challenges, the detection of malicious software, or malware, poses a significant threat to the integrity and security of digital systems.

Malware encompasses a broad range of malicious software designed to infiltrate, exploit, or compromise computer systems, networks, and applications. Traditional approaches to malware detection have often relied on signature-based methods that identify known patterns of malicious code. However, these methods are limited in their ability to detect new and evolving malware variants, as cyber attackers continuously evolve their tactics to evade detection.

In response to the dynamic nature of modern cyber threats, deep learning has emerged as a promising approach for enhancing malware detection. Deep learning leverages artificial neural networks with multiple layers to automatically learn intricate patterns and representations from large volumes of data. By harnessing the power of deep learning, cybersecurity researchers and practitioners aim to develop more robust and adaptive malware detection systems capable of identifying both known and previously unseen threats.

This literature review explores the application of deep learning techniques to the realm of cybersecurity, with a specific focus on the detection of malicious software. By surveying the existing literature, this review aims to provide a comprehensive overview of the state-of-the-art in deep learning-based malware detection. The review will examine various methodologies that have been utilized to improve the accuracy and efficiency of malware detection systems.

The upcoming sections provide a comprehensive taxonomy for malware analysis and discuss various approaches employed in malware detection. The intention behind these sections is to equip the reader with a thorough understanding of the current state-of-the-art in this domain.

2.1.1 Taxonomy of Malware Analysis

Malware analysis is a fundamental practice in cybersecurity aimed at understanding and categorizing malicious software. It involves various techniques to dissect and comprehend the behavior, functionality, and potential threats posed by malware.

The taxonomy of malware analysis is broadly categorized into three main approaches: Static Analysis, Dynamic Analysis, and Hybrid Analysis.

Static Analysis

Static analysis focuses on examining malware without executing it. It involves analyzing the binary or source code of malware to identify patterns, characteristics, and potential malicious activities. This approach includes techniques such as disassembly, decompilation, and code inspection. Static analysis helps in identifying common malware indicators, extracting signatures, and detecting code vulnerabilities. However, it may not provide insights into the runtime behavior and evasion techniques employed by more sophisticated malware.

The work presented in this thesis primarily falls within the domain of Static Analysis, where the focus is on analyzing malware without its execution. This includes the examination of executable file's binary code, utilizing techniques such as disassembly, decompilation, and code inspection. The objective is to identify patterns and characteristics indicative of malware, enhancing our ability to detect and categorize malicious software.

Dynamic Analysis

Dynamic analysis involves executing malware within a controlled environment to observe its behavior during runtime. This approach captures interactions between the malware and its environment, monitoring system calls, network traffic, and file activities. Dynamic analysis enables the detection of hidden actions, evasion techniques, and malware's response to various inputs. It is particularly useful for understanding malware's execution flow, identifying communication channels, and detecting malicious payloads. However, dynamic analysis might not be effective against highly evasive malware that can detect and evade analysis environments.

Hybrid Analysis

Hybrid analysis combines both static and dynamic analysis techniques to enhance the accuracy and effectiveness of malware analysis. By leveraging the strengths of both approaches, hybrid analysis aims to overcome the limitations of individual methods. It can provide a comprehensive view of malware's behavior, including its static attributes and dynamic execution characteristics. Hybrid analysis is particularly beneficial for detecting polymorphic and obfuscated malware, as well as capturing both static and runtime indicators of compromise.

In the realm of cybersecurity, selecting the appropriate malware analysis technique depends on the nature of the malware, the goals of the analysis, and the available resources. Each approach contributes to a holistic understanding of malware, enabling security professionals to develop effective countermeasures and mitigate potential threats.

2.1.2 Trends in Malware Detection

The field of malware detection has evolved over the years, driven by advancements in technology and the evolving threat landscape. This subsection presents an overview of the trends in malware detection, highlighting different approaches employed to detect and mitigate the risks posed by malicious software.

Statistical Analysis

In the early stages of cybersecurity, malware detection predominantly relied on primitive methods such as signature-based detection and pattern matching [2, 3, 4, 5, 6]. Signature-based detection involves comparing files or code segments against a database of known malware signatures. While effective against well-known malware, this approach struggled with polymorphic and metamorphic malware that can evade static signatures through minor modifications. Additionally, simple pattern matching had limitations in detecting emerging threats with novel code patterns.

Machine Learning

As malware became more sophisticated, conventional machine learning methods started playing a crucial role in malware detection. These methods involve training models on labeled datasets, allowing them to learn patterns and characteristics of both benign and malicious software. Feature engineering, which involves selecting relevant attributes from the data, is a key component of this approach. Conventional machine learning techniques include K-means [7], decision trees [8], support vector machines [9, 10], and random forests [11]. While effective, these methods can struggle with the ever-changing nature of malware and require continuous updates to stay relevant.

Deep Learning

The emergence of deep learning has brought a paradigm shift to malware detection. Deep learning techniques, such as neural networks and convolutional neural networks (CNNs), can automatically learn complex features from raw data, reducing the need for manual feature engineering [12]. This approach excels at capturing intricate patterns, even in the presence of obfuscation and polymorphism. Recurrent neural networks (RNNs) are utilized to capture sequential dependencies in malware behavior. Hybrid architectures, combining CNNs and RNNs, further enhance detection capabilities [13]. Deep learning models, such as LSTM networks and Transformer-based architectures, have demonstrated promising results in accurately detecting known and unknown malware.

LSTM networks have proven to be effective in capturing long-range dependencies in sequential data, making them suitable for modeling the dynamic behavior of malware [14, 15]. Transformer-based architectures, with their attention mechanisms, excel at processing and understanding intricate relationships within malware code [16, 17]. These deep learning models are capable of learning complex behaviors exhibited by malware and adapting to evolving attack strategies.

The trends in malware detection reflect an ongoing effort to stay ahead of evolving cyber threats. As attackers continuously develop new evasion techniques, the combination of advanced machine learning algorithms and deep learning models equips cybersecurity professionals with powerful tools to combat the ever-growing variety of malware.

Chapter 3

Background

In the pursuit of effective malware detection using machine learning, it is essential to establish a comprehensive understanding of the foundational concepts, evaluation techniques, and the application of deep learning methods. This literature review section provides an in-depth exploration of these crucial aspects.

3.1 Artificial Intelligence

The quest for machines that can emulate human intelligence dates back to the inception of computing. Artificial Intelligence (AI) is the interdisciplinary domain that aims to create systems capable of reasoning, learning, and performing tasks that typically require human intelligence. The history of AI is marked by significant milestones, from the Dartmouth Workshop in 1956 [18] that birthed the term 'artificial intelligence' to the modern era of neural networks and deep learning.

In Figure 3.1, a taxonomy of Artificial Intelligence is depicted. Each segment of this taxonomy will be briefly described in the subsequent sections, elucidating their significance in the landscape of machine learning and malware detection.

3.2 Machine Learning

While Artificial Intelligence encompasses the broader concept of creating intelligent systems, Machine Learning (ML) constitutes a vital subset within AI. ML focuses on the development of algorithms and models that enable computers to learn patterns and make predictions based on data without explicit programming.

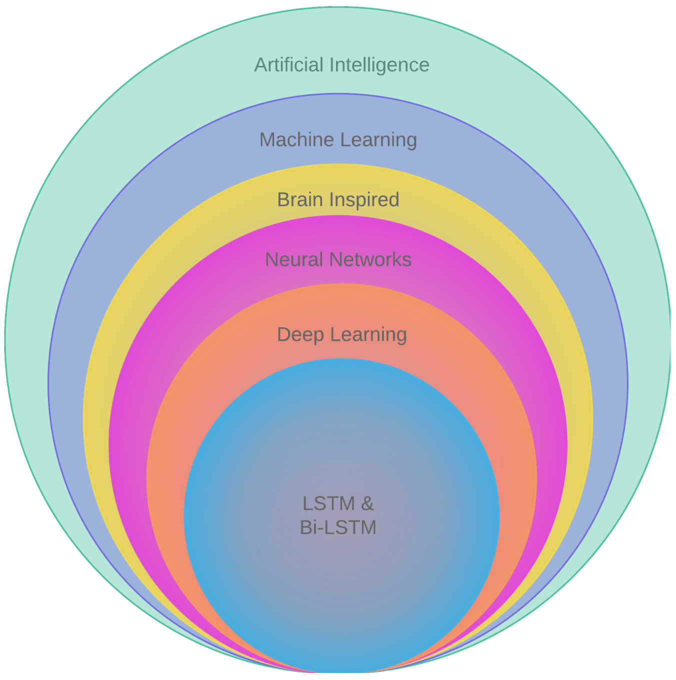

Figure 3.1: Taxonomy of Artificial Intelligence showing LSTM's placement

3.3 Neural Networks

The development of AI has been profoundly shaped by the complexities of the human brain. The human brain's remarkable capacity to process information, adapt to novel scenarios, and acquire knowledge from experiences has captivated researchers for decades. This profound intrigue has driven the investigation of brain-inspired principles within the domain of AI, culminating in the emergence of Neural Networks.

Neural Networks draw inspiration from the structure and functioning of biological neurons in the brain. These networks consist of interconnected nodes, or "neurons", that process and transmit information through weighted connections. The collective behavior of these interconnected neurons enables the network to learn patterns, recognize features, and make predictions from data.

In the context of machine learning, Neural Networks have demonstrated remarkable capabilities in tasks ranging from image recognition to natural language processing. Convolutional Neural Networks (CNNs) excel at image analysis, while Recurrent Neural Networks (RNNs) and variants like Long Short-Term Memory (LSTM) networks are particularly effective for sequential data analysis.

3.3.1 Neuron

At the heart of neural networks lies the concept of neurons, inspired by their biological counterparts in the human brain. Neurons serve as information processing units within the network, contributing to the network's ability to learn patterns from data.

The concept of artificial neurons can be traced back to the mid-20th century. Warren McCulloch and Walter Pitts proposed a simplified model of a neuron in their seminal paper "A Logical Calculus of Ideas Immanent in Nervous Activity" [19], published in 1943. In their work, they introduced a model of a neuron that could take binary inputs, apply weights to those inputs, sum them up, and produce an output based on a threshold. This groundbreaking paper laid the foundation for the concept of artificial neural networks.

Further development of the artificial neuron concept was carried out by Frank Rosenblatt, who introduced the perceptron in 1957 [20]. The perceptron was a single-layer neural network designed to recognize and classify patterns. Although perceptrons had limitations in solving certain types of problems, Rosenblatt's work marked a significant step forward in the field of neural networks.

The concept of artificial neurons, initially proposed by McCulloch, Pitts, and later developed by Rosenblatt, has paved the way for the complex neural network architectures that we see today, including multi-layer perceptrons, convolutional neural networks, recurrent neural networks, and more. These architectures have demonstrated remarkable capabilities in tasks ranging from image recognition to natural language processing and reinforcement learning.

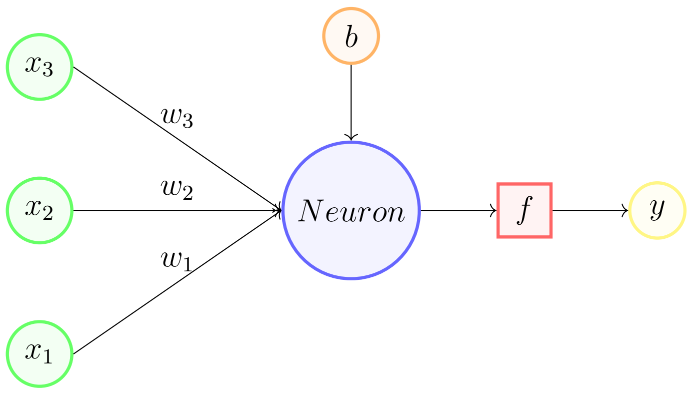

Figure 3.2: Neuron overview

A neuron comprises several key components, see Fig. 3.2:

- **Input data**: Neurons receive input data from other neurons or external sources. These inputs are multiplied by corresponding weights, which represent the strength of the connections.

- **Weights**: Each input is associated with a weight, representing the significance of that input in influencing the neuron's output. These weights are adjusted during training to learn meaningful patterns.
- **Summation Function**: The weighted inputs, including the bias terms, are summed together, incorporating both positive and negative influences.
- **Activation Function**: The summed value undergoes an activation function that introduces non-linearity to the neuron's output. Common activation functions include sigmoid, ReLU (Rectified Linear Unit), and tanh (Hyperbolic Tangent).
- **Output**: The output of the activation function represents the neuron's final response. This output is then forwarded to other neurons in subsequent layers. The output is calculated through the following formula:

  $$y = f\left(\sum_{i=1}^{n}(x_i \cdot w_i) + b\right)$$

  where n refers to the number of inputs, x_i signifies the value of the i-th input, w_i indicates the weight associated with the i-th input, b denotes the bias term, and f represents the activation function applied element-wise to the weighted sum.

  This computation is often presented in a vectorized format:

  $$y = f\left(\mathbf{w} \cdot \mathbf{x} + b\right)$$

  where $\mathbf{y}$ is the output vector, $\mathbf{w}$ is the weight vector, $\mathbf{x}$ is the input vector, and b is the bias. This compact form simplifies calculations and is widely used in the implementation of neural networks.

3.4 Fully Connected Layer

A *fully connected layer*, or also referred as a *dense layer*, represents a fundamental component of various artificial neural networks, including feed-forward neural networks and deep learning architectures. Within a fully connected layer, each neuron establishes connections with every neuron in the preceding layer. This arrangement results in a dense matrix of interconnections between neurons, facilitating the learning of intricate relationships from the input data.

Mathematically, in a fully connected layer, the output of each neuron is computed, as described above, by taking a weighted sum of the outputs from the previous layer, adding the bias, and applying an activation function. The weights and biases associated with each connection are the learnable parameters of the layer, and they determine the strength and bias of each connection.

A fully connected layer can be represented as follows:

$$\mathbf{y} = f\left(\mathbf{W} \cdot \mathbf{x} + \mathbf{b}\right)$$

where:

- $\mathbf{W}$ is the weight matrix that encodes the connection strengths between neurons,
- $\mathbf{x}$ represents the input to the network or the output of the preceding layer,
- $\mathbf{b}$ is the bias vector that accounts for the bias of each neuron.
- f is the activation function.

Fully connected layers are versatile and can capture complex patterns in data, but they can also lead to a large number of parameters in deep networks. Regularization techniques and architectural design choices are often used to mitigate overfitting and enhance the generalization ability of fully connected networks.

Fully connected layers are frequently used in conjunction with other layers, such as activation functions, pooling layers, and output layers, to build diverse neural network architectures capable of tackling various tasks, including classification, regression, and more complex tasks like image recognition and natural language processing.

3.5 Loss Function

The loss function, denoted as $\mathcal{L}$ in a neural network context, is a mathematical measure that quantifies the discrepancy between the predicted output of the network and the actual target values. Its purpose is to guide the learning process by providing a measure of how well or poorly the network is performing with respect to the training data.

In the context of training a neural network, the loss function computes a single scalar value that represents the error or cost associated with the current predictions made by the network. The goal during training is to minimize this loss function, as a lower value indicates that the network's predictions are closer to the desired targets.

Different types of neural network architectures and tasks may require different loss functions. Common loss functions include mean squared error (MSE) for regression problems and categorical cross-entropy for classification problems. The choice of the loss function depends on the specific problem being solved and the nature of the target values.

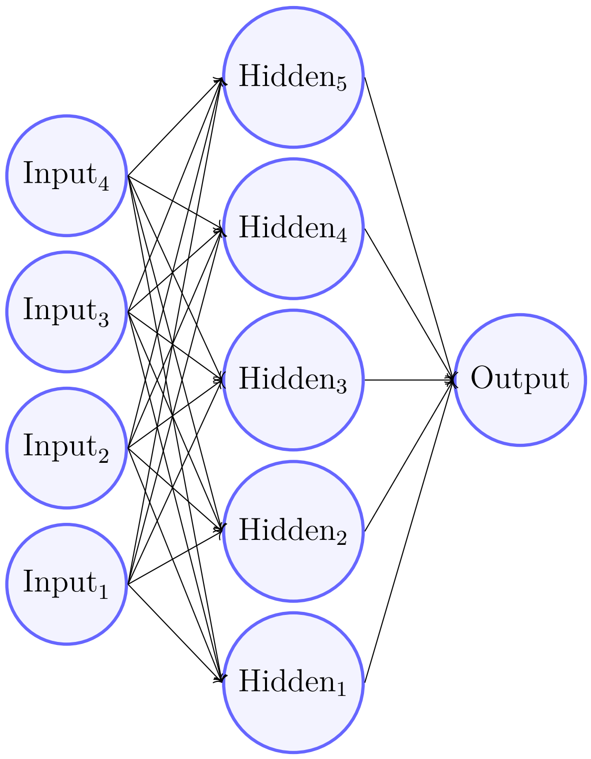

Figure 3.3: Illustration of a fully connected network architecture with 4 inputs, 1 hidden layer containing 5 neurons, and 1 output.

3.6 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a fundamental optimization algorithm used to train machine learning models, including neural networks. It's a variant of the traditional gradient descent algorithm designed to efficiently handle large datasets by processing data in small batches.

In SGD, rather than computing the gradient of the loss function over the entire dataset, the gradient is estimated using a single data point or a mini-batch of data at a time. This introduces randomness into the optimization process, as each batch provides a noisy estimate of the true gradient.

The main idea behind SGD is to iteratively update the model's parameters in the direction that reduces the loss function. For example, in a neural network, parameters include the weights and biases. The update formula for a parameter θ (which could represent a weight or bias) is given by:

$$\theta \leftarrow \theta - \eta \cdot \nabla_\theta \mathcal{L}$$

where:

- η is the learning rate, which determines the step size of the updates.
- $\nabla_\theta \mathcal{L}$ is the gradient of the loss with respect to the parameter θ.

Key concepts and benefits of SGD include:

- **Efficiency**: SGD processes a small subset of the data at each iteration, making it computationally efficient and suitable for large datasets.
- **Generalization**: The noise introduced by mini-batch sampling can help prevent overfitting by adding regularization.
- **Escape from Local Minima**: The noise in the gradient estimates can sometimes help the optimization process escape from local minima.

However, SGD also has challenges, such as sensitivity to learning rate and convergence to suboptimal solutions. To address these challenges, various techniques have been proposed, including learning rate schedules, momentum, and adaptive learning rate algorithms like Adam.

In practice, the choice of learning rate, batch size, and optimization algorithm plays a crucial role in determining the convergence and performance of neural networks during training.

3.6.1 Adam Optimization Algorithm

Adam (short for Adaptive Moment Estimation) [21] is an optimization algorithm that combines the advantages of both stochastic gradient descent (SGD) and momentum. It's designed to provide adaptive learning rates for individual parameters, making it well-suited for a wide range of machine learning tasks, including training neural networks.

The key idea behind Adam is to adaptively adjust the learning rates based on the historical gradients and squared gradients of the parameters. This helps the algorithm automatically adjust the learning rates for different parameters, allowing it to converge faster and handle various types of data and loss landscapes.

The update formula for parameter θ using Adam is given by:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t$$

where:

- η is the learning rate.
- $\hat{m}_t$ is the first moment estimate (similar to the momentum term in other algorithms).

- $\hat{v}_t$ is the second moment estimate.
- ϵ is a small constant to prevent division by zero.

The first moment estimate $\hat{m}_t$ and the second moment estimate $\hat{v}_t$ are computed as exponentially decaying moving averages of past gradients and squared gradients, respectively:

$$\hat{m}_t = \beta_1 \cdot \hat{m}_{t-1} + (1 - \beta_1) \cdot \nabla_t$$

$$\hat{v}_t = \beta_2 \cdot \hat{v}_{t-1} + (1 - \beta_2) \cdot (\nabla_t)^2$$

where ∇_t is the gradient of the loss with respect to parameter θ.

The hyperparameters β_1 and β_2 control the decay rates of the moving averages. Adam also introduces a bias correction step to mitigate the initialization bias of the moving averages.

Adam offers the benefits of efficient optimization, automatic adaptation of learning rates, and robustness to different loss landscapes.

3.7 Backpropagation

The backpropagation algorithm, has its origins in the work of several researchers in the 1970s and 1980s. Paul Werbos (1974) introduced the concept of backpropagation as a method for adjusting network weights and formulated it as an optimization problem [22]. David Rumelhart, Geoffrey Hinton, and Ronald Williams (1986) further developed the algorithm and demonstrated its effectiveness for training multi-layer neural networks [23].

Backpropagation is a pivotal algorithm for training neural networks by optimizing their parameters to minimize a chosen loss function. It is an acronym for *backward propagation of errors*, enabling iterative learning from data.

The backpropagation process consists of two main phases: the forward pass and the backward pass.

3.7.1 Forward Pass

During the forward pass, input data is propagated through the network, layer by layer. Activations are computed using learned weights and biases. The output of each layer becomes the input for the next layer, resulting in predictions or output values.

3.7.2 Backward Pass

The backward pass is where the crucial gradient computations occur. It calculates gradients of the loss function with respect to network parameters. These gradients quantify the sensitivity of the loss to changes in each parameter. The chain rule from calculus is employed to efficiently compute these gradients.

Gradients are computed layer by layer, initiated from the output layer and propagated backward towards the input layer. The chain rule permits the decomposition of the gradient of the loss with respect to the output into a product of gradients at each layer.

Mathematically, consider a simple neural network layer (see Fig. 3.3) with activation function f, weight matrix $\mathbf{W}$, biases $\mathbf{b}$, and input vector $\mathbf{x}$. The equations for the forward and backward passes are as follows:

Forward Pass:

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

$$\mathbf{y} = f(\mathbf{z})$$

Backward Pass:

$$\nabla_{\mathbf{z}}\mathcal{L} = \nabla_{\mathbf{y}}\mathcal{L} \cdot f'(\mathbf{z})$$

$$\nabla_{\mathbf{W}}\mathcal{L} = \nabla_{\mathbf{z}}\mathcal{L} \cdot \mathbf{x}^T$$

$$\nabla_{\mathbf{b}}\mathcal{L} = \nabla_{\mathbf{z}}\mathcal{L}$$

Here, $\mathcal{L}$ represents the loss function, and f' signifies the derivative of the activation function.

3.8 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a type of artificial neural network architecture designed to handle sequential data and capture temporal dependencies within it. Unlike feedforward neural networks, which process input data in a fixed and isolated manner, RNNs maintain an internal state that evolves as new input is processed. This enables RNNs to effectively model sequences, making them well-suited for tasks such as natural language processing, speech recognition, and time series analysis.

The key feature of RNNs is their ability to maintain memory of previous time steps, allowing them to capture and utilize information from earlier parts of a sequence. This is achieved by introducing recurrent connections that create a loop within the network, enabling information to be passed from one step to the next.

Mathematically, an RNN can be represented as follows:

$$\mathbf{h_t} = f\left(\mathbf{W_{hx}}\mathbf{x_t} + \mathbf{W_{hh}}\mathbf{h_{t-1}} + \mathbf{b_h}\right)$$

$$\mathbf{y} = \mathbf{W_{yh}h_t} + \mathbf{b_y}$$

where:

- $\mathbf{x_t}$ is the input at time step t,
- $\mathbf{h_t}$ is the hidden state at time step t,
- $\mathbf{W_{hx}}$ is the weight matrix associated with the input values,
- $\mathbf{W_{hh}}$ is the weight matrix associated with the previous hidden state values,
- $\mathbf{W_{oh}}$ is the weight matrix associated with the output values,
- $\mathbf{b_h}$ and $\mathbf{b_y}$ are bias vectors,
- f is the activation function.

The output at each time step can be used for prediction, classification, or further processing. Additionally, the hidden state $\mathbf{h_t}$ acts as a memory that stores relevant information from previous time steps.

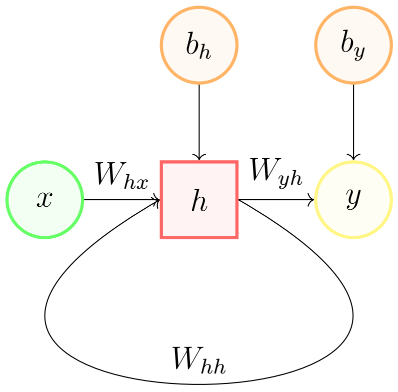

Figure 3.4: RNN overview

The architecture of an RNN can be visualized as shown in Figure 3.4. This figure depicts the sequential nature of RNNs, where the hidden state is shared across time steps, allowing the network to maintain memory of past inputs.

For a more detailed view of the temporal relationships within an RNN, an unrolled representation can be used, as seen in Figure 3.5.

The unrolled RNN diagram in Figure 3.5 illustrates the internal connections between time steps, providing a clearer depiction of how the network processes sequential data and maintains hidden states across multiple steps. This unrolled view helps in understanding the flow of information within the network over time.

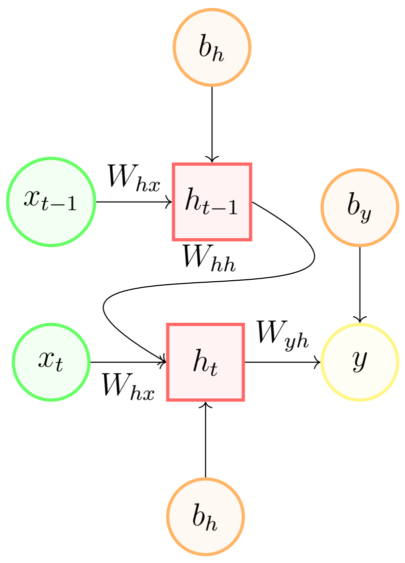

Figure 3.5: Unrolled representation of an RNN for two consecutive time steps

RNNs suffer from the vanishing gradient problem, where gradients become very small during backpropagation through time, leading to difficulties in learning long-range dependencies. To address this, more advanced RNN variants have been developed, such as Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs), which incorporate mechanisms to better capture long-term dependencies while mitigating the vanishing gradient problem.

3.9 Vanishing and Exploding Gradient Problem in RNNs

RNNs are powerful models for processing sequential data. However, they suffer from the vanishing and exploding gradient problem during training. This issue arises from the nature of backpropagation through time (BPTT) [24], the algorithm used to compute gradients in RNNs.

When backpropagating gradients through multiple time steps, the gradients can either become extremely small (vanishing gradient) or extremely large (exploding gradient). This occurs because the gradients are multiplied across multiple layers in the network, which can cause them to either diminish or explode exponentially.

The vanishing gradient problem leads to slow convergence and difficulty in capturing long-range dependencies in sequences. On the other hand, the exploding gradient problem can cause training instability and make it challenging to find suitable model parameters.

3.10 Long Short-Term Memory

Long Short-Term Memory (LSTM) [25] networks are a specialized type of recurrent neural networks designed to address the limitations of standard RNNs, particularly for capturing long-range dependencies in sequences.

3.10.1 LSTM Cell

At the core of an LSTM network is the LSTM cell, see Fig. 3.6. The LSTM cell possesses a unique architecture that enables it to remember information over long sequences and selectively retain or discard information through gating mechanisms. Each LSTM cell consists of various components, including input gates, forget gates, memory cells, and output gates.

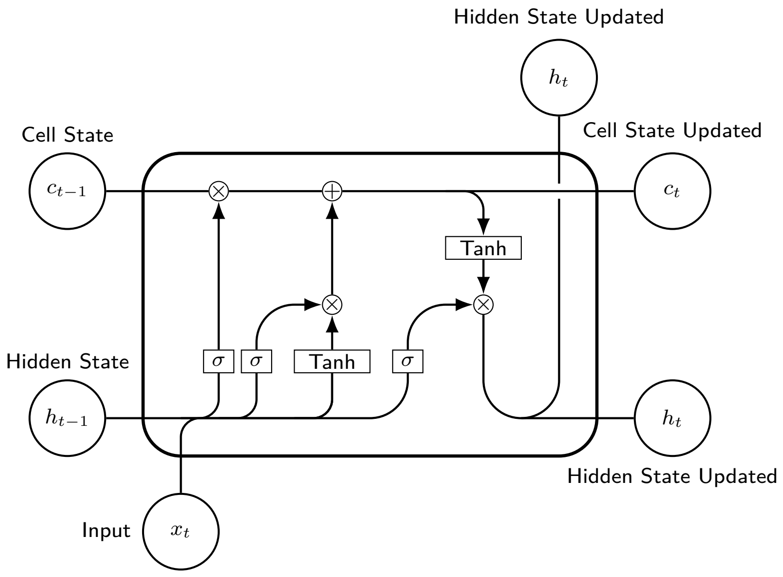

Figure 3.6: Architecture of an LSTM cell.

3.10.2 Working of an LSTM Cell

An LSTM cell's operation involves a delicate interplay of its components. The input gate controls which information should be updated, the forget gate determines what information to discard, and the output gate regulates the information to be outputted. The activation functions used within the LSTM cell are provided in Annex A.

Forget Gate (f): The forget gate decides what information from the previous cell state (c_{t-1}) should be discarded. It is controlled by the sigmoid activation function and is calculated as:

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$$

Potential Cell State Update ($\tilde{c}$): The cell state update captures the new information to be stored in the cell state. It is determined by the *tanh* activation function and is given by:

$$\tilde{c}_t = \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c)$$

Input Gate (i): The input gate determines how much of the new information ($\tilde{c}$) should be added to the cell state (c). It is controlled by the sigmoid activation function and is given by:

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$$

Cell State (c) Update: The cell state (c_t) is updated using the input gate (i_t), forget gate (f_t), and the potential cell state update ($\tilde{c}_t$):

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

Output Gate (o): The output gate controls how much of the cell state should contribute to the hidden state. It is determined by the sigmoid activation function and is calculated as:

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$$

Hidden State (h) Update: The hidden state (h_t) is updated using the output gate (o_t) and the cell state (c_t):

$$h_t = o_t \odot \tanh(c_t)$$

3.10.3 Addressing the Gradient Problem

Long Short-Term Memory networks stand out in their ability to effectively mitigate the vanishing and exploding gradient problem, which is a challenge faced by regular Recurrent Neural Networks. The challenge arises from the fact that during the training process, gradients can become extremely small or large as they are propagated backward through the network. This leads to issues such as slow or unstable learning, hindering the network's ability to capture long-term dependencies in sequences.

Traditional RNNs suffer from the vanishing gradient problem, where gradients associated with early time steps can become exceedingly small as they are multiplied by the same weights in each time step. This diminishes the impact of early inputs on later stages of learning, making it challenging for the network to capture long-range relationships. On the other hand, the exploding gradient problem results in gradients becoming significantly larger as they are multiplied by the same weights, leading to unstable and divergent training.

LSTMs introduce gating mechanisms that address these problems effectively. Each LSTM cell contains gating units that regulate the flow of information. These gating units, including the input, forget, and output gates, enable the LSTM to selectively remember or forget information over multiple time steps. Importantly, these gates control the flow of gradients during backpropagation.

The forget gate allows the LSTM to discard irrelevant information from the cell state, preventing it from being overwhelmed by irrelevant data that can contribute to the exploding gradient problem. The input gate then enables the LSTM to selectively update the cell state with new information. These gating mechanisms provide a level of control over the gradient flow that is not present in regular RNNs.

By addressing the gradient problem, LSTMs enhance the stability and effectiveness of training. They facilitate the flow of relevant gradients, ensuring that updates to the model's parameters are appropriate and well-scaled. This results in more efficient learning and improved gradient flow through time. Overall, the gating mechanisms in LSTMs play a crucial role in allowing the network to learn and capture dependencies in sequences, making them a superior choice for tasks involving long sequential data compared to regular RNNs.

LSTMs have revolutionized sequence modeling by addressing the limitations of traditional RNNs. Their ability to handle long sequences, coupled with memory cells and gating mechanisms, enables them to capture both short-term and long-term dependencies effectively.

3.11 Bidirectional Long Short-Term Memory

Bidirectional Long Short-Term Memory (BiLSTM) [26] networks are an extension of the traditional LSTM architecture, designed to capture not only past but also future information in sequences. Standard LSTMs process sequences in a left-to-right manner, which may limit their ability to capture patterns that depend on both earlier and later context. BiLSTMs address this limitation by incorporating two separate LSTM layers, one processing the sequence in a forward direction and the other in a backward direction.

The concept of Bidirectional Recurrent Neural Networks (BRNN) lays the foundation for BiLSTMs. Introduced by Schuster and Paliwal in 1997 [27], BRNNs process sequences bidirectionally, capturing context from both earlier and later time steps. This bidirectional processing allows the network to capture complex dependencies in sequential data more effectively than traditional RNNs.

The architecture of a BiLSTM network Fig. 3.7 comprises two LSTM layers: one that processes the input sequence in a forward direction (from the beginning to the end) and the other that processes the sequence in a backward direction (from the end to the beginning). The outputs of these two LSTM layers are often combined through concatenation or other methods to create a holistic representation of the input sequence. This combined representation retains information from both past and future time steps, enabling the network to make more informed predictions and capture intricate patterns.

By processing sequences in both directions, BiLSTMs can effectively capture dependencies that span across time steps. This feature is particularly advantageous in scenarios such as natural language processing, where understanding the context of words and phrases in both directions is crucial for accurate understanding and generation of text.

The bidirectional processing in BiLSTMs has proven valuable in various applications, including speech recognition, sentiment analysis, and machine translation. These networks excel in scenarios where contextual information plays a significant role in achieving accurate and meaningful predictions.

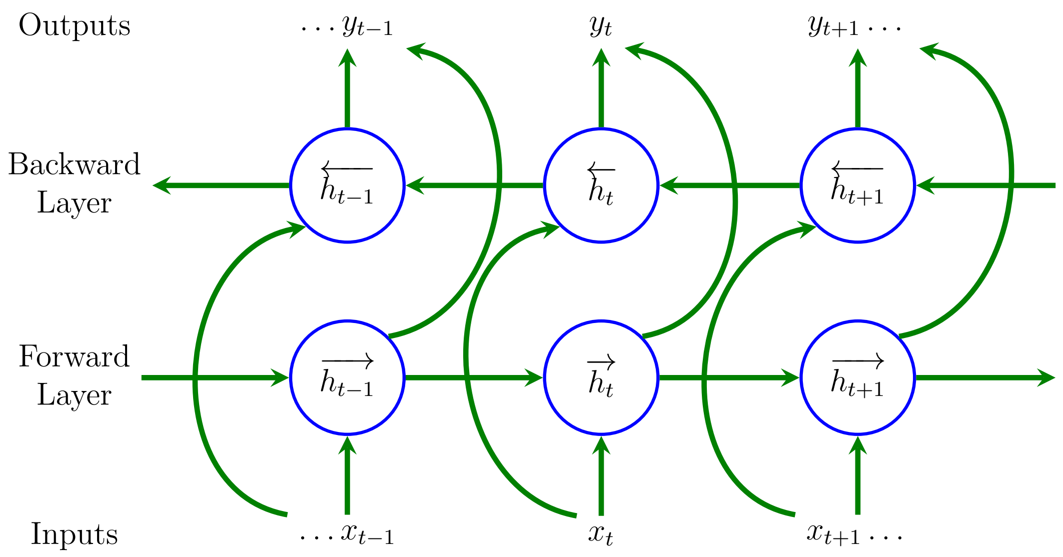

Figure 3.7: Architecture of a Bidirectional LSTM network.

3.11.1 Information Fusion

The key idea behind a BiLSTM is that the hidden states of both the forward and backward LSTMs at a given time step are concatenated. This combined representation captures the contextual information from both directions. The output at each time step is typically obtained by combining the outputs of the forward and backward LSTMs, which can be as simple as concatenating them or applying a weighted sum.

Mathematically, let $\overrightarrow{h_t}$ represent the hidden state of the forward LSTM and $\overleftarrow{h_t}$ represent the hidden state of the backward LSTM at time step t. The combined hidden state at time step t is given by $\overleftrightarrow{h_t} = [\overrightarrow{h_t}, \overleftarrow{h_t}]$, where $[,]$ denotes concatenation.

In summary, BiLSTM networks leverage the bidirectional processing capability of BRNNs and the memory-enhancing properties of LSTMs to capture dependencies in sequences from both past and future directions. This dual-context processing makes BiLSTMs a powerful tool for tackling complex sequence modeling tasks across various domains.

3.12 Word Embedding

Natural language is rich in meaning and context, making it a complex and intricate medium for communication. In the realm of natural language processing (NLP), accurately representing words in a manner that captures their semantic relationships and contextual nuances is a fundamental challenge. Traditional approaches, which treat words as discrete symbols, often struggle to capture the inherent complexities and similarities among words. This limitation spurred the development of word embeddings, a transformative technique that bridges the gap between the symbolic nature of words and their underlying semantic structures.

The effectiveness of word embeddings can be attributed to a foundational linguistic principle that asserts the similarity of words is reflected in their contextual usage. This concept finds its roots in the maxim "you shall know a word by the company it keeps," attributed to British linguist J.R. Firt [28]. Alongside his American counterpart Zellig Harris, Firth is often recognized for pioneering the concept of "distributional semantics." This idea underscores the notion that words with comparable meanings often appear in similar contexts. The development of word embeddings was heavily influenced by this linguistic theory, as it provided a framework to capture semantic relationships between words in a quantitative and meaningful manner. Through this innovative approach, words are no longer treated as isolated entities but rather as dynamic components intricately connected to the linguistic environment in which they occur. This principle has been a driving force behind the creation of word embedding techniques that enhance the ability of machine learning models to decipher and represent the intricate nuances of human language.

Word embeddings are distributed representations of words in a continuous vector space. Unlike traditional symbolic representations, where words are represented as isolated tokens, word embeddings encode semantic information by positioning words in a high-dimensional space. The key idea behind word embeddings is to map words that share similar contexts and meanings to nearby points in this vector space. This representation captures the inherent semantic relationships between words, enabling machines to process and understand natural language more effectively.

The foundation of word embeddings lies in the distributional hypothesis, which posits that words that appear in similar contexts tend to have similar meanings. Leveraging large text corpora, word embedding models learn to capture these contextual relationships and encode them as dense vectors. Through this process, words with similar meanings become closer in the vector space, while words with dissimilar meanings are separated by larger distances.

The advent of word embeddings has revolutionized various NLP tasks, including text classification [29], sentiment analysis [30], machine translation [31], and information retrieval [32]. The ability to represent words in a continuous space has enabled models to generalize better across languages, contexts, and domains, leading to substantial improvements in performance.

This section serves as an introduction to the word embedding technique employed in this project, namely Word2Vec [33]. We will delve into the details of Word2Vec, encompassing its two fundamental approaches: Continuous Bag of Words (CBOW) and Skip-gram (SG). Through a comprehensive exploration of these approaches, we aim to provide a clear understanding of how Word2Vec generates word embeddings by capturing semantic relationships and contextual associations within a given corpus. By focusing on these specific methods, we aim to shed light on the intricate processes that enable Word2Vec to transform words into high-dimensional vectors capable of encapsulating the semantic essence of language.

3.12.1 Word2Vec

Word2Vec, proposed by Tomas Mikolov and his team at Google in 2013 [34], revolutionized the field of natural language processing by introducing an efficient and scalable method to learn distributed word representations from large text corpora. This technique leverages the insight that words with similar meanings often appear in similar contexts. The fundamental principle of Word2Vec is rooted in the idea of "distributional semantics," which posits that the meaning of a word can be inferred from its distributional patterns within a corpus.

Word2Vec offers two primary approaches i.e. CBOW and SG. In the CBOW approach, the model predicts the target word based on its surrounding context words, treating the context words as input features. Conversely, in the SG approach, the model predicts the context words given a central target word. Both approaches employ a neural network architecture that learns to optimize word embeddings that capture semantic relationships among words.

By transforming words into numerical vectors that reflect their semantic essence, Word2Vec enables downstream machine learning models to operate more effectively on textual data. Overall, Word2Vec's elegant and data-driven approach to learning word embeddings has significantly advanced the field of natural language understanding and remains a cornerstone in modern deep learning techniques for text analysis.

3.13 Training Process

The training process of a neural network involves iteratively updating its parameters to minimize the chosen loss function. This process aims to improve the model's ability to make accurate predictions on unseen data. Several key components play a role in the training process, and their interactions are essential for achieving a well-trained neural network. The following sections delve into these components, providing a comprehensive understanding of each element and its impact on the model's performance.

3.13.1 Data Split

Before training, the dataset is typically split into three subsets: training, validation, and test sets. The training set is used to update the model's parameters, while the validation set helps monitor the model's performance during training. The test set is used to evaluate the final performance of the trained model on unseen data.

3.13.2 Epochs

An epoch refers to a single pass through the entire training dataset. During each epoch, the model's parameters are adjusted based on the training data to reduce the loss. Multiple epochs are performed to iteratively refine the model's performance.

3.13.3 Batch Size

Training on the entire dataset at once may require a substantial amount of memory and computation. Instead, training is often performed in batches. The batch size defines the number of data samples processed together before updating the model's parameters. This approach introduces stochasticity into the training process and can lead to faster convergence.

3.13.4 Learning Rate

The learning rate determines the step size taken in the direction that reduces the loss function. It is a hyperparameter that controls the magnitude of parameter updates during each iteration. A higher learning rate may lead to faster convergence, but it can also result in overshooting the optimal parameter values. Conversely, a lower learning rate may lead to slower convergence.

3.13.5 Optimization Algorithm

Gradient-based optimization algorithms, such as Stochastic Gradient Descent (SGD) and its variants (e.g., Adam, RMSProp), are used to update the model's parameters. These algorithms leverage the gradients of the loss function with respect to the parameters to adjust them in the direction that reduces the loss.

3.13.6 Validation Split

To monitor the model's performance during training and prevent overfitting, the validation set is used. The validation set is not used for training but is used to evaluate the model's performance on data it has not seen before. After each epoch, the model's performance on the validation set is measured. By evaluating the model's generalization ability on this separate dataset, we can determine whether the model is learning relevant patterns or memorizing the training data, aiding in the selection of the most optimal model.

In summary, the training process involves finding the right parameters to achieve optimal model performance while avoiding overfitting.

3.14 Model Evaluation Metrics

Evaluation metrics serve as indispensable tools for assessing the effectiveness of a machine learning model. They offer valuable insights into the model's proficiency in handling the provided dataset. The choice of specific metrics can vary based on the unique characteristics of the problem at hand.

3.14.1 Confusion Matrix

A confusion matrix is a tabular representation that shows the classification results of a model on a dataset. It provides insight into the true positive, true negative, false positive, and false negative predictions, allowing a deeper understanding of the model's performance.

		Prediction outcome		
		p	**n**	**total**
actual value	**p′**	True Positive	False Negative	P′
	n′	False Positive	True Negative	N′
	total	P	N	

Table 3.1: Confusion Matrix for Binary Classification

3.14.2 Accuracy

Accuracy is a widely used metric for evaluating the overall performance of a classification model. It measures the proportion of correctly predicted instances out of the total number of instances in the dataset. The accuracy metric provides insights into the model's proficiency in correctly classifying data points. Mathematically, accuracy can be represented as:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Instances}}$$

Alternatively, accuracy can also be computed in terms of True Positive and True Negative as follows:

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Number of Instances}}$$

3.14.3 Precision

Precision serves as a foundational evaluation metric to gauge the accuracy of positive predictions made by a classification model. It quantifies the ratio of true positive predictions to the total positive predictions generated by the model. Mathematically, precision is formulated as:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

A high precision value indicates that the model makes fewer false positive errors and possesses an enhanced capability to accurately identify positive instances. However, a higher precision may lead to a decreased ability to capture all true positive instances, possibly resulting in false negatives.

3.14.4 Recall

Recall, also referred to as sensitivity or the true positive rate, directs its focus on the model's competence in capturing all actual positive instances present within the dataset. It is calculated by assessing the ratio of true positive predictions to the total number of actual positive instances. Mathematically, recall is defined as:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

A higher recall value implies the model's effectiveness in identifying positive instances; however, it may potentially lead to an increase in false positive errors, thereby affecting precision. Balancing the trade-off between precision and recall is imperative for achieving an optimal classification performance.

3.14.5 F1-Score

The F1-score, a comprehensive evaluation metric, takes into account both precision and recall, offering a balanced assessment of a classification model's effectiveness. It is calculated as the harmonic mean of precision and recall, thus striking a balance between the two metrics. The F1-score is mathematically expressed by the formula:

$$\text{F1-Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

The F1-score aims to achieve a compromise between precision and recall, ensuring the model performs well on both positive and negative instances. This metric proves particularly valuable when dealing with imbalanced class distributions, necessitating a balanced evaluation measure to gauge overall performance.

3.14.6 Receiver Operating Characteristic (ROC) Curve

The Receiver Operating Characteristic (ROC) curve is a graphical representation that showcases the trade-off between the true positive rate (recall) and the false positive rate (1-specificity) at various threshold values. The ROC curve provides valuable insights into a model's ability to discriminate between positive and negative instances, across a spectrum of threshold settings.

In an ROC curve, the x-axis represents the false positive rate (FPR), also known as the probability of a false alarm or Type I error, while the y-axis represents the true positive rate (TPR), also referred to as recall or sensitivity. Each point on the ROC curve corresponds to a specific threshold setting, indicating the compromise between correctly identifying positive instances and incorrectly labeling negative instances.

The ROC curve's shape can help identify the model's performance. A curve that is closer to the upper-left corner indicates superior classification performance, as it implies a higher true positive rate while keeping the false positive rate low. Conversely, a diagonal line from the bottom-left to the top-right would represent a random guessing classifier.

3.14.7 Area Under the ROC Curve

The Area Under the ROC Curve (AUC-ROC) is a numerical measure of the ROC curve's performance. It quantifies the ability of a model to distinguish between positive and negative instances. An AUC-ROC value of 0.5 indicates random performance, while a value of 1.0 suggests perfect discrimination between the classes.

3.15 Hyperparameter Optimization

Hyperparameter optimization (HPO) is a fundamental process in machine learning model development, aimed at finding the optimal configuration of hyperparameters (HP) that results in the best model performance. HP are parameters of the model that are not learned during training but are set before the training process begins. These settings influence the learning process and can significantly impact a model's effectiveness.

The goal of HPO is to systematically search through different combinations of HP values to identify the configuration that leads to the highest performance on a specific task or dataset. This process is crucial as it can significantly affect a model's ability to generalize and make accurate predictions.

Common HP include learning rates, batch sizes, the number of layers in a neural network, dropout rates, and the choice of optimization algorithms, among others. The optimal values of these HP can vary depending on the dataset and the specific machine learning task.

HPO is typically performed using specialized libraries and tools that automate the search process, such as grid search, random search, or Bayesian optimization. These methods aim to efficiently explore the HP space and find the best combination.

For a detailed and comprehensive understanding of HPO techniques, refer to Bischl et al. [35], where the concept and various strategies are thoroughly explained.

3.15.1 Fidelity Level in Hyperparameter Optimization

The concept of fidelity level, denoted as λ_{fid}, represents a crucial parameter within the realm of HPO. It serves as a pivotal determinant in the allocation of computational resources to individual evaluations.

Typically, λ_{fid} is chosen to exhibit a linear relationship with the computational costs incurred during an evaluation. This linear correlation implies that the sum of all λ_{fid} values across evaluations provides an estimation of the total computational cost for an entire optimization run. Higher values of λ_{fid} are associated with evaluations that more closely approximate the true objective but at the expense of increased computational resources. Conversely, lower λ_{fid} values lead to more cost-efficient but potentially less accurate evaluations.

In practice, this introduces a delicate balance between the explorative nature of optimization and the fidelity of individual evaluations. One common approach is to initially allocate the computational budget to less expensive Hyperparameter Configurations (HPCs) with lower λ_{fid} values to explore a broader search space. Subsequently, the focus shifts towards more promising HPCs with higher λ_{fid} values.

This strategic allocation of computational resources based on λ_{fid} underscores the trade-off between the breadth of exploration and the depth of evaluation within the context of HPO.

3.15.2 Successive Halving

Successive Halving (SH) [36] is a pivotal technique within the field of HPO that plays a fundamental role in the foundation of the Hyperband optimization method. SH operates under the premise of a predetermined fidelity-budget B, which represents the cumulative sum of λ_{fid} values across all evaluations.

The SH methodology commences by considering a fixed number of candidate HPCs, denoted as $\lambda^{(i)}$, initially marked as $p^{[0]}$. It then proceeds to iteratively narrow down this candidate pool to identify the single most promising configuration through a series of evaluation stages denoted as t. This iterative process is driven by an incrementally increasing fidelity schedule.

The scheduling of fidelity increments within SH is often regulated by the η_{HB} control multiplier, a critical component of the Hyperband algorithm. Typically, η_{HB} is set to a value greater than 1, commonly 2 or 3. After each batch evaluation t involving the current population of size $p^{[t]}$, the selection process focuses on the top-performing $\frac{1}{\eta_{HB}}$ fraction of configurations. The fidelity for subsequent candidate evaluations is then adjusted to $\eta_{HB} \times \lambda_{fid}$, reflecting the heightened fidelity. As a result, configurations displaying promise receive increased fidelity allocations, while less promising ones are pruned early in the optimization process.

To ensure efficient fidelity budget allocation, the starting fidelity $\lambda_{fid}^{[0]}$ and the number of stages $s+1$ are calculated in a manner that approximately distributes $B/(s+1)$ fidelity units across each batch evaluation within the SH procedure. This strategic allocation strategy guarantees that, cumulatively, no more than the specified fidelity budget B is expended during the entirety of SH:

$$\sum_{t=0}^{s} \left\lfloor p^{[0]} \eta_{HB}^{-t} \right\rfloor \lambda_{fid}^{[0]} \eta_{HB}^{t} \leq B$$

It's worth noting that the effectiveness of SH hinges on a judicious selection of both the initial number of candidate configurations and the resulting fidelity schedule. When operating within a fixed fidelity-budget framework for HPO, practitioners must make a strategic decision between pursuing either a larger number of configurations with lower fidelity, or a smaller set of configurations but with higher fidelity. This choice carries significant implications for the optimization process and its outcomes.

3.15.3 Hyperband

Hyperband (HB), introduced by Li et al. in 2018 [37], represents a highly efficient approach to HPO. It can best be conceptualized as the iterative application of SH. Hyperband orchestrates multiple SH executions, referred to as *brackets*, each initiated with different numbers of starting configurations denoted as $p_s^{[0]}$.

To initialize a bracket, Hyperband requires two key parameters: η_{HB} and the upper limit of fidelity, λ_{fid}^{upp}, with the constraint $\lambda_{fid}^{upp} > \eta_{HB}$.

The process of constructing a fidelity budget B for each bracket begins by identifying the most exploratory bracket. This bracket sets the foundation for determining the fidelity budgets for the remaining brackets. Here, the number of batch evaluations $s_{max}+1$ is chosen such that:

$$s_{max} + 1 = \left\lfloor \log_{\eta_{HB}}(\lambda_{fid}^{upp}) \right\rfloor + 1$$

$$\lambda_{fid}^{[0]} = \lambda_{fid}^{upp} \eta_{HB}^{-s_{max}} \in (\eta_{HB}^{-1}, \eta_{HB})$$

$$\lambda_{fid}^{[s_{max}]} = \lambda_{fid}^{upp}$$

These fidelity values are collected in the vector:

$$\mathbf{r} = (\lambda_{fid}^{upp} \eta_{HB}^{-s_{max}}, \lambda_{fid}^{upp} \eta_{HB}^{-s_{max}+1}, \lambda_{fid}^{upp} \eta_{HB}^{-s_{max}+2}, \dots, \lambda_{fid}^{upp}) \in \mathbb{R}^{s_{max}+1}$$

.

The objective is to ensure roughly equal total fidelity expenditure and the reduction of candidates to one winning HPC in each batch evaluation. Consequently, the fidelity budget of each bracket is determined as $B = (s_{max} + 1)\lambda_{fid}^{upp}$.

For every $s \in \{0, \ldots, s_{max}\}$, a bracket is defined by establishing the starting fidelity $\lambda_{fid}^{[0]} \geq \lambda_{fid}^{low}$ for the bracket $\mathbf{r}^{(1+s_{max}-s)}$. This results in a total of $s_{max} + 1$ brackets, with an overall fidelity budget expended by HB amounting to $(s_{max}+1)B$. Consequently, each bracket s comprises $s + 1$ batch evaluations, and the initial population size $p_s^{[0]}$ is determined as the maximum value that satisfies the SH equation.

In summary, Hyperband optimizes the process of Hyperparameter Optimization by orchestrating multiple Successive Halving executions across brackets, each starting with a different number of configurations. This approach efficiently explores the hyperparameter search space, making it a powerful tool in the field of machine learning model development.

3.16 Instruction Set Architecture

In the realm of computer architecture, the term Instruction Set Architecture (ISA) defines the interface between hardware and software in a computer system. It acts as a bridge, specifying how instructions are encoded, executed, and interacted with by software programs. The ISA outlines a processor's set of instructions, including their formats, operations, and addressing modes.

For our project, which involves analyzing executables to generate feature vectors, a solid grasp of ISA is essential. The ISA defines the universe of possible instructions within these files, significantly influencing our approach to feature extraction. In our case, we evaluate x86-64 ELF files, where the x86-64 ISA serves as the reference. All instructions within these files, along with their encoding and execution, adhere to the x86-64 ISA.

3.16.1 x86-64 (AMD64) ISA

The x86-64 ISA, also known as x64, x86_64, AMD64, and Intel 64, is a 64-bit extension of the original x86 ISA. Initially developed by AMD (Advanced Micro Devices) and later adopted by Intel under the names "Intel 64" or "IA-32e".

The x86-64 ISA significantly expands processor capabilities by introducing 64-bit general-purpose registers and extended memory addressing. It maintains compatibility with legacy 32-bit x86 code, allowing seamless coexistence of 32-bit and 64-bit software on the same system.

The widespread adoption of the x86-64 ISA in the computing industry was a compelling factor in our choice to analyze ELF files of this particular ISA. It ensures that our research remains relevant and applicable in contemporary computing environments.

For reference to the instructions found in the x86-64 ELF files within our dataset, we turn to Section 2.5 of the AMD64 manual [38]. This manual serves as an invaluable resource, providing a comprehensive guide to the instruction set, a cornerstone of our project's analysis.

3.17 ELF Format

The ELF (Executable and Linkable Format) is a common file format used for executables, object code, shared libraries, and more. It provides a standardized way to organize and represent various components of a binary program, see Fig. 3.8.

The ELF format consists of several key components, including:

- **ELF Header**: The ELF header contains basic information about the file, such as its type, architecture, and entry point.

- **Program Header Table**: The program header table describes the various segments of the file, such as code, data, and stack. Each entry in the table specifies the type of the segment, its offset in the file, its virtual memory address, and other attributes.

- **Segments**: ELF files typically include segments for both code and data. The code segment contains the actual machine instructions to be executed, while the data segment holds initialized and uninitialized data used by the program.

- **Section Header Table**: Similar to the program header table, this table provides information about sections, including their sizes and file offsets.

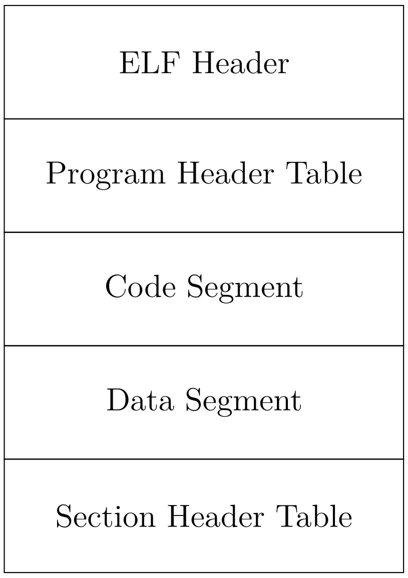

Figure 3.8: Overview of the ELF File Format

The code segment (also known as the text section) within the ELF file is of particular interest in our thesis. This segment contains the assembly instructions that define the behavior of each sample in our dataset. Analyzing the code segment allows us to gain insights into the underlying functionality and potential behavior of the programs represented by the ELF files.

Chapter 4

Proposed Model

4.1 Model Architecture

The proposed malware detection model consists of two primary components: a Word Embedding layer and a Deep Learning model. The architecture is designed to effectively capture the semantic information within the executable files and learn complex patterns for malware detection. This section provides an overview of the model's structure.

4.1.1 Word Embedding Layer

At the core of the model lies the Word Embedding layer, which plays a pivotal role in transforming raw executable files into a format suitable for deep learning. This layer utilizes Word2Vec, a popular word embedding technique, to convert sequences of binary instructions into continuous-valued vectors. Each instruction within an executable file is mapped to a unique vector representation. The Word Embedding layer enhances the model's ability to discern meaningful features from the raw data.

4.1.2 Deep Learning Model

The Deep Learning model comprises several LSTM or BiLSTM layers, followed by a softmax dense layer. Two aspects of the architecture are considered as hyperparameters: the type of layers (LSTM or BiLSTM) and the number of layers. Its optimal value is determined through a thorough hyperparameter optimization process, as depicted in section 5.6.

LSTM and BiLSTM Layers

LSTM layers are employed to capture sequential dependencies within the embedded feature vector. These layers maintain memory over time, allowing the model to understand the context and relationships between different features.

BiLSTM layers enhance the model's ability to capture both past and future information from the input sequences. By processing the data in both forward and backward directions, the model gains a comprehensive understanding of the contextual relationships.

The stacking of LSTM or BiLSTM layers allows the model to learn hierarchical features and patterns, for accurate malware detection.

Softmax Dense Layer

The model is designed as a binary classifier, classifying executable files into two categories i.e. malicious or benign. To accomplish this, the final layer is a single-node softmax dense layer. This layer computes the probability that a given program is malicious. The output is a single scalar value $p \in [0, 1]$, where values closer to 1 indicate a higher probability of being a malicious program, and values closer to 0 indicate a higher probability of being benign.

Loss Function: Binary Cross-Entropy

For the proposed deep learning model, we employ the binary cross-entropy loss function. Binary cross-entropy is a common choice for binary classification tasks, such as the one at hand. This loss function quantifies the dissimilarity between the predicted binary outputs and the actual target labels, encouraging the model to make accurate binary predictions. The formula for binary cross-entropy is given by:

$$\mathcal{L}(\mathbf{y}, \mathbf{p}) = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log(p_i) + (1 - y_i) \log(1 - p_i)]$$

Here, N represents the number of samples, y_i denotes the true binary label for the i-th sample, and p_i is the predicted probability that the i-th sample belongs to the positive class. Minimizing this loss function during training guides the model to make accurate binary predictions.

Adam

To optimize our deep learning model during training, we employ the Adam optimization algorithm. Adam stands for "Adaptive Moment Estimation" and is a widely used optimization technique for training neural networks. It combines elements of both the momentum and RMSprop optimization algorithms, making it well-suited for a variety of deep learning tasks.

Adam adapts the learning rates for each parameter individually based on their past gradients and squared gradients. This adaptability allows it to converge faster and handle a wide range of learning rates, making it a robust choice for optimizing our model. The algorithm is known for its effectiveness in accelerating convergence and improving overall training efficiency.

4.1.3 Model Overview

Fig. 4.1 is an overview of the proposed model architecture.

The depicted architecture highlights the flow of information through the Word Embedding and Deep Learning components, demonstrating how the model processes and analyzes the input data.

The number of LSTM or BiLSTM layers and other hyperparameters are determined through rigorous hyperparameter optimization to ensure the model's optimal performance in malware detection. This approach allows for a flexible and adaptable architecture that can effectively capture the complexities of malware behavior.

In the subsequent sections, we will delve deeper into the feature extraction, training process, hyperparameter optimization, and model evaluation to provide a comprehensive understanding of the proposed model's capabilities and performance.

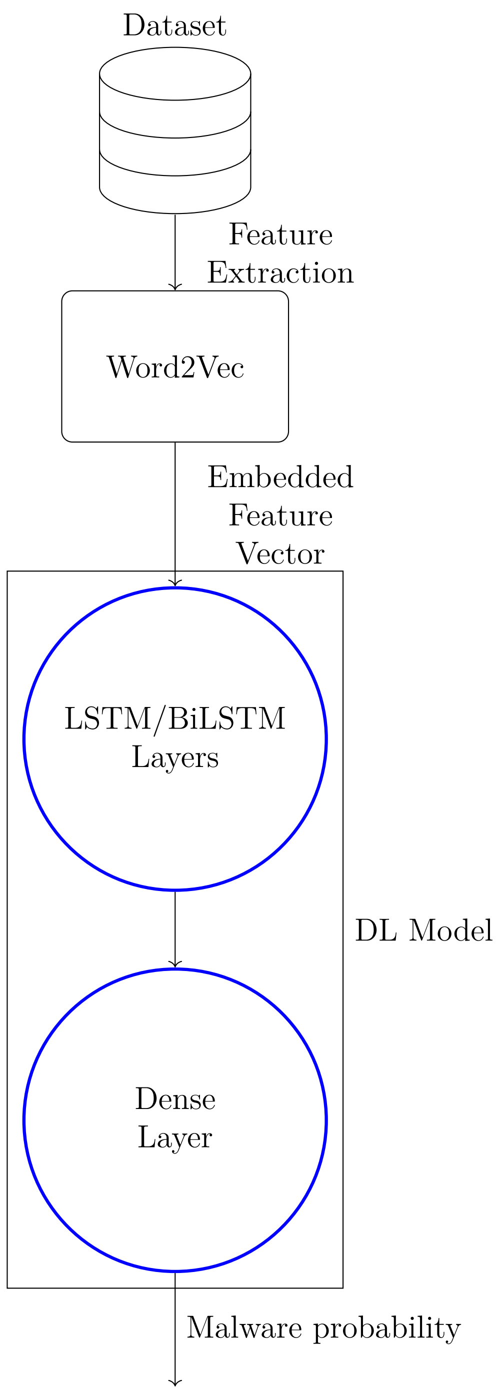

Figure 4.1: Overview of the Proposed Model Architecture

Chapter 5

Methodology

This chapter outlines the comprehensive methodology employed in this study to address the core objectives and tasks. It encompasses the entire research workflow, spanning from data collection to model evaluation. The process commences with data collection, where we gather the necessary dataset for analysis. Subsequently, we delve into data preprocessing techniques to prepare the dataset for model training. A crucial aspect of this research involves the disassembly of AMD64 ELF files, which is explained in detail. To facilitate model evaluation, we discuss the data splitting methodology to ensure the robustness of our models. The core of our research revolves around model development, encompassing the training, execution, and testing phases. Finally, the implemented hyperparameter optimization technique is discussed. This chapter provides an in-depth overview of the steps undertaken to achieve our research objectives.

5.1 Data collection

The data collection is done differently for the benign and malicious files. In the following sections, it could be found how we obtained the files for the experiment.

5.1.1 Benign files collection

This section outlines the methodology employed to obtain the benign files for the dataset. Our approach aimed to create a representative dataset and utilized information from the Debian Popularity Contest [39].

This study aimed to collect information on Debian machines that volunteered to participate in the project. The project website contains various types of information, including several ordered lists. One such list details the most commonly installed packages. To compile our dataset, we utilized this list to download packages in descending order of frequency and extract the available binaries. The number of downloaded files was contingent upon the number of previously collected malicious files, with the ultimate goal of creating a balanced dataset comprised of an equal number of benign and malicious binaries.

For each package in the list, we performed the following actions:

1. Installed the package using a specified method.
2. Identified the location of the binary using a specific technique.
3. Add the binary to the dataset.
4. Finally, removed the package using a designated process.

Given that the experiment was conducted on a 2020 M1 MacBookAir, an arm64 virtual machine was chosen to facilitate virtualization.

During the experiment, we faced several errors while processing the 3066 considered packages. Out of these, we were able to download 728 packages successfully and extract their binaries. However, we encountered difficulties in locating binaries for 2338 packages due to various reasons. The primary issue we encountered was that the package could not be located by `apt`, which occurred 1567 times. Additionally, an `apt` error message stating `E: Can’t select candidate version from package` was encountered for 346 packages, which can occur due to unmet dependencies for the chosen package to install or upgrade. In such cases, `apt` may not be able to determine a suitable candidate version because it can’t find a version that satisfies all dependencies. Moreover, we were able to download some packages successfully, but binaries were not found for 347 packages.

We provide the detailed error messages and their respective occurrences in the following table:

Error Message	Occurrences
Extraction Successful	728
Unable to locate package	1567
Binaries not found	347
Can’t select candidate version from package	346
Couldn’t find any package by	66
Download is performed unsandboxed as root as file	6
Handler silently failed	6

Table 5.1: Errors encountered when collecting benign files.

From the 728 extracted packages, we decided to include only 2742 binaries to maintain the dataset’s entropy. This decision was made due to the number of malicious files we were able to collect.

5.1.2 Malicious file collection

The experiment utilized malicious files obtained from VirusShare [40]. These files, mainly in the ELF format, covered three different years (2014, 2019, and 2020) and were collected for analysis. To understand the architecture of each executable, the tool called `objdump` was used. However, a small portion of these files posed problems as the tool couldn't determine their architecture. Among the 56757 files downloaded, 1151 files, approximately 2% of the dataset, were affected by this issue. To ensure the analysis's reliability, these problematic files were excluded, leaving only the unaffected files for further investigation. The remaining files contained executables for a range of architectures.

To select an appropriate architecture for the model, a comprehensive analysis of the malicious files by their architecture was conducted. In Figure 5.1, you can observe a visualization of the different architectures present in the collected dataset. This analysis played a pivotal role in guiding the choice of an architecture that could effectively handle the diverse range of malicious programs encountered during the experiment.

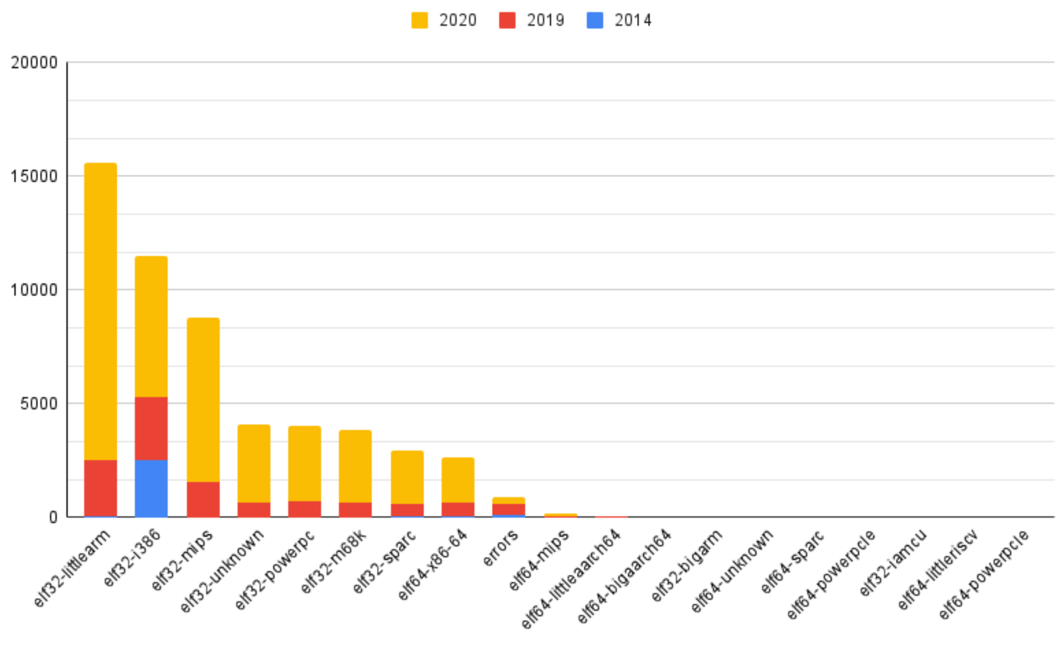

Figure 5.1: ISA histogram for the VirusShare files.

We curated a dataset of malicious X86-64 ELF files, as we believe this architecture is commonly used, making our work of potential interest to a wider audience. The table 5.2 presents statistics of the dataset for a better understanding.

Year	Total files	x86-64 files	Probability of x86-64 file
2014	2579	31	0.01
2019	9802	569	0.06
2020	43225	1912	0.04
Total	55606	2512	0.05

Table 5.2: Collected x86-64 files.

The majority of files in the dataset are from 2020, whereas the number of 2014 files is negligible.

Additionally, it is important to note that there is a possibility that `objdump` recognizes the architecture of a file but is unable to disassemble it. This was observed in four files, leaving the total number of collected files at 2512. This number is a bottleneck for our dataset, as it is generally easier to obtain benign files than malicious ones. Nevertheless, we believe that our dataset is sufficiently comprehensive to conduct the experiment.

5.1.3 Malicious Binaries Filtering by Architecture

In addressing the diverse architecture landscape of malicious binaries within our dataset, a meticulous process of filtering by architecture was implemented. Given the heterogeneous nature of the malicious files, containing binaries of various architectures, a specialized chain of Bash scripts was devised for this purpose.

- **`main.sh`:** Serving as the orchestrator of the filtering process, this script initializes essential variables and orchestrates the execution of subsequent scripts. It begins by locating non-hidden binary files in the specified path and proceeds to invoke `check_architecture.sh` for each of them. After the completion of `check_architecture.sh` for all binaries, the script calls `pick_binaries.sh`.

- **`check_architecture.sh`:** This script employs `objdump` to determine the architecture of each binary and assess whether the binary can be successfully disassembled. The results are systematically logged into a file, including the binary's name and its architecture if identified.

- **`pick_binaries.sh`:** Leveraging the log file generated by `check_architecture.sh`, this script discerns which binaries align with the desired architecture. Subsequently, it copies the identified binaries into a specified path, effectively creating a dataset exclusively comprising binaries of the desired architecture.

Upon executing this chain of scripts, the malicious binaries were successfully filtered based on the chosen architecture. The resulting dataset consisted of a total of 5254 files, maintaining a balanced distribution between benign and malicious files. All files in the refined dataset adhered to the x86-64 architecture, ensuring uniformity for subsequent analysis and model training.

5.2 Disassembly of AMD64 ELF Files

Malware analysis often requires a low-level examination of binary executables. We adopt the process of disassembly to convert AMD64 ELF (Executable and Linkable Format) files into human-readable assembly instructions. This facilitates a deeper understanding of the code's logic, aiding in the identification of potentially malicious behaviors.

Disassembly involves translating machine code into assembly instructions, resulting in a textual representation of the program's operations. This step enables the inspection of control flow, function calls, and data manipulation within the binary. By disassembling AMD64 ELF files, we gain insights into the code's functionality, which is essential for subsequent analysis and detection techniques.

We leverage the `objdump` tool, a widely-used utility in the field of reverse engineering, to perform this task.

5.2.1 Process Overview

Figure 5.2 provides an overview of the disassembly process. Here's a step-by-step breakdown:

1. **Input**: The process begins with an AMD64 ELF file, which is the target of analysis. This file could be a potentially malicious binary that requires thorough examination.

2. **`objdump` Invocation**: We execute the `objdump` tool with appropriate command-line options on the target ELF file. The tool dissects the binary into its constituent sections, headers, and instructions.

3. **Disassembly Output**: `objdump` generates a disassembly output that provides a human-readable representation of the binary's assembly instructions. Each instruction is accompanied by its hexadecimal opcode, memory addresses, and corresponding assembly code, see the example in 5.1.

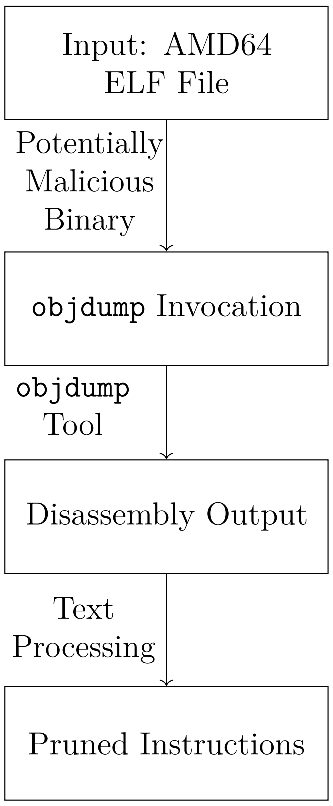

Figure 5.2: Disassembly Process using `objdump`

4. **Text Processing and Pruning**: Following the generation of the disassembly output, the text undergoes a series of processing steps. The `sed` tool, along with pattern matching techniques, is applied to precisely extract the instruction names from the disassembly output. The extracted instruction names are compiled into a structured CSV file, where each row corresponds to a distinct instructions sequence in the dataset. The columns in this file include:

   - **File:** The name of the executable file.
   - **Function:** The name of the function in which the instructions were found. If a function name is not identified, it is denoted as ".text."
   - **Sequence:** A sequence of instructions separated by blanks, representing the processed disassembled code.
   - **Label:** A binary indicator (1 or 0) specifying whether the file is malicious or benign, respectively.
   - **Length:** An integer indicating the length of the instruction sequence.

This CSV file serves as a comprehensive resource for subsequent analysis and data preparation tasks. Notably, the initial division based on function names is rendered obsolete for the project. When the CSV is read in further steps, all rows belonging to the same file are grouped together, considering the entire file as a single unit. This modification was implemented to streamline the data representation, as an alternative approach involving function-wise separation was found to be unnecessary and is not further described in the thesis. Example 5.2 showcases a snippet of the generated CSV file, providing insights into the formatted data structure.

Listing 5.1: Example Disassembled Program

```

zsh:     file format elf64-x86-64


Disassembly of section .text:

0000000000017000 <.text>:
   17000:   48 83 ec 08             sub    $0x8,%rsp
   17004:   48 8b 05 dd af 0b 00    mov    0xbafdd(%rip),%rax
   # d1fe8 <__gmon_start__>
   1700b:   48 85 c0                test   %rax,%rax
   1700e:   74 02                   je     17012 <
   __ctype_toupper_loc@plt-0x1e>
   17010:   ff d0                   call   *%rax
   17012:   48 83 c4 08             add    $0x8,%rsp
   17016:   c3                      ret
```

Listing 5.2: Example Pruned Instructions

```
File,Function,Sequence,Label,Length
/<path_of_the_file>/<file_name>,.text,sub mov test je call add ret
    ,0,7
```

In order to provide a better understanding of the dataset, we generated histograms showcasing the 20 most frequent instructions for both the benign and malicious files.

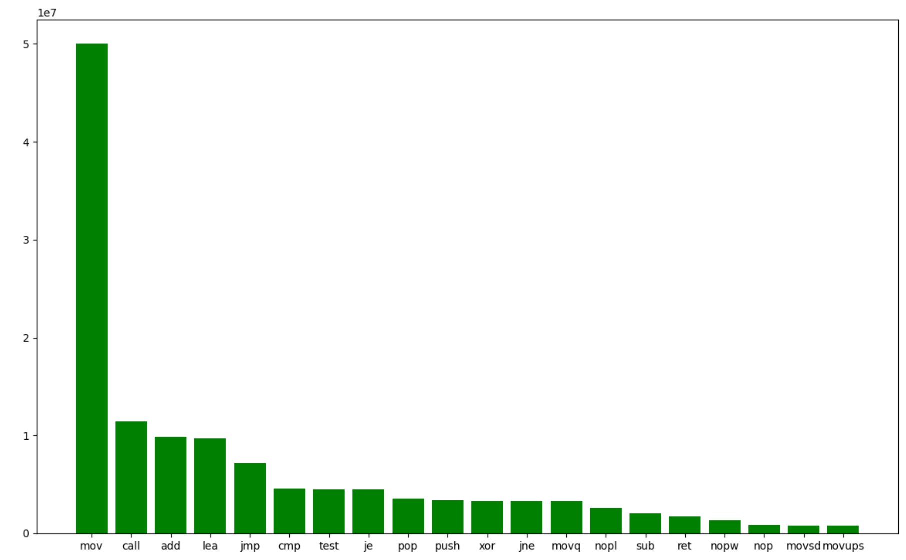

Figure 5.3: Histogram of the 20 most frequent machine instructions among the benign files.

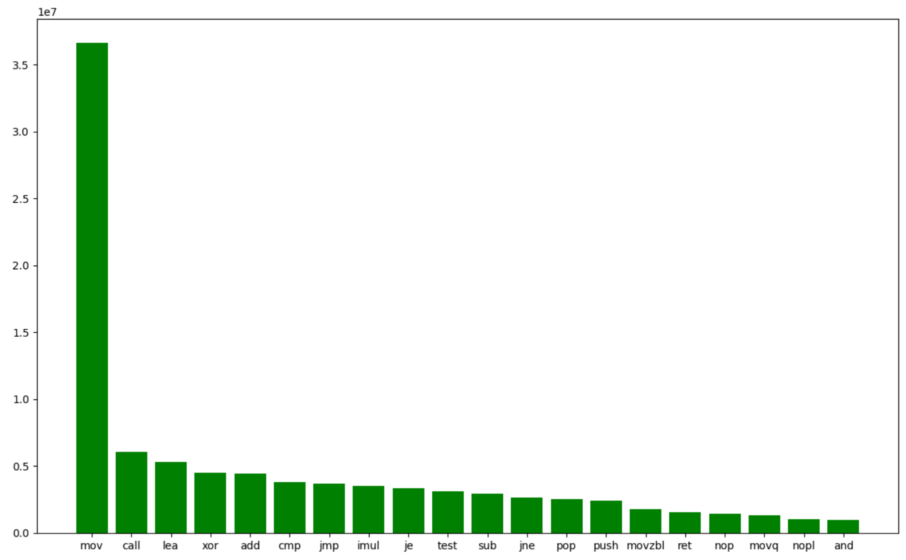

Figure 5.4: Histogram of the 20 most frequent machine instructions among the malicious files

5.3 Data Splitting

Before training the malware detection model, the dataset underwent a careful data splitting process to ensure effective training, validation, and evaluation of the model. The dataset, comprising executable files, was divided into three distinct subsets:

- **Training Set (70%):** This set, the largest of the three, consisted of 70% of the dataset. It was used to train the model, allowing it to learn from a substantial portion of the available data.
- **Validation Set (20%):** To ensure the model's robustness and to prevent overfitting, a validation set containing 20% of the dataset was created. The validation set was used to fine-tune hyperparameters and monitor the model's performance during training.
- **Testing Set (10%):** The testing set comprised 10% of the dataset. It remained separate from the training process and the HPO. It was reserved exclusively for evaluating the model's performance. This provided an unbiased assessment of the model's effectiveness.

The careful splitting of the dataset into these subsets facilitated a comprehensive evaluation of the model's capabilities and allowed for effective training, testing, and validation procedures.

Dataset	Benign Files	Malicious Files
Training	1919	1758
Validation	576	527
Test	247	227

Table 5.3: Dataset split overview.

5.4 Model Training

5.4.1 Data Preparation: Tokenization and Indexing

The foundation of training an effective malware detection model lies in the preparation of the dataset. In this section, we delve into the steps being taken to process the raw instruction data before it is fed into the deep learning model.

The training and validation datasets consist of sequences of assembly instructions, representing the executable code of various programs. To enable the model to work with this textual data, a series of preprocessing steps are performed.

Tokenization

First, the instructions within the training and validation datasets are tokenized. Tokenization is the process of breaking down the text into individual units or tokens, in this case, individual instruction names. This step is crucial to ensure that the model can understand and work with the discrete elements of the assembly instructions.

Indexing

Once tokenized, each unique instruction name needs to be assigned an integer index. This indexing process generates a vocabulary denoted as V, where each instruction name present in the training or validation datasets is included. Each unique instruction name in the vocabulary is assigned an integer index ranging from 1 to $|V|$, where $|V|$ represents the vocabulary size.

This indexing of instruction names transforms the textual data into a numerical format, making it amenable to processing by deep learning models. It provides the model with a structured representation of the instructions, allowing it to learn patterns and associations between different instructions during training.

5.4.2 Word Embedding

The next crucial step in preparing our data for effective deep learning-based malware detection is word embedding. Word embedding is a technique that maps discrete tokens, in our case, instruction names, into continuous vector spaces. This enables the model to understand the semantic relationships between words or, in our context, instructions. We employ the Gensim Word2Vec tool [41] to accomplish this task.

Our Word2Vec model is trained using the indexed training and validation data. It's configured with specific hyperparameters to generate embeddings that best capture the characteristics of our assembly instructions. In particular, we set the dimension of the embedding vectors to 300 [14]. Each instruction will be represented as a 300-dimensional vector. The window size is defined as 100 instructions [15]. This parameter determines how many instructions before and after a given instruction are considered when learning its context. The Word2Vec model is set to work as a Skip-Gram word embedding model, focusing on predicting the surrounding instructions based on a given instruction.

After the training process, the Word2Vec model yields an embedding matrix. This matrix has dimensions $|V| \times 300$, where $|V|$ represents the vocabulary size. Each row in the matrix corresponds to a unique instruction in the vocabulary, and each column represents one of the 300 dimensions in the embedding space. Mathematically, the embedding matrix can be expressed as:

$$\text{Embedding Matrix} \in \mathbb{R}^{|V| \times 300}$$

This matrix encodes the relationships and similarities between different instructions within our vocabulary. It forms the foundation upon which our deep learning model will learn to distinguish between benign and malicious programs based on these vector representations.

5.4.3 Feature Extraction

With our word embeddings in place, the next crucial step is feature extraction. This process involves generating a feature vector for each of the samples in our dataset, which will serve as input to our deep learning model.

To extract meaningful features from each sample, we begin by processing the assembly instructions contained within it. For this purpose, we employ the Tokenizer class provided by TensorFlow [42]. The Tokenizer class tokenizes and indexes each instruction in the sequence. As a result, we obtain a list of tuples, where each tuple consists of an instruction and the number of occurrences of that instruction in the sample.

The list of tuples is then sorted in descending order of instruction frequency, ensuring that the most frequent instructions are placed at the beginning of the list. This sorting is performed to prioritize the most informative instructions in the feature vector.

Furthermore, to maintain the feature vector's manageable size and avoid excessive computational overhead, we truncate the list if it exceeds a predetermined threshold, such as 600 instructions. This threshold ensures that our model focuses on the most relevant instructions while keeping the input dimensionality within reasonable bounds.

In essence, this feature extraction process transforms the raw assembly code of each sample into a structured representation that captures the frequency and relevance of individual instructions, preparing it for input into our deep learning network.

5.4.4 Feature Embedding

The next critical step in our pipeline is the embedding of the previously generated feature vectors. These feature vectors, representing the frequency and relevance of instructions within each sample, need to be transformed into a suitable input format for our deep learning model.

To achieve this, we leverage the word embeddings we acquired earlier through the Word2Vec model. The process entails maintaining the original order of instructions in the feature vector list and embedding each instruction using the pre-trained Word2Vec embeddings. This operation effectively replaces each instruction with a corresponding embedding vector.

As a result, we obtain a sequence of embedding vectors, where each vector represents an instruction within the feature vector. Mathematically, this can be envisioned as a matrix with dimensions:

$$\text{Embedded Feature Vector} \in \mathbb{R}^{f \times 300}$$

where f is the lenght of the feature vector. Since we have configured our Word2Vec model to generate embeddings of dimension 300, this matrix will have 300 columns.

In essence, this step converts our feature vectors into a more interpretable and computationally effective format, aligning them for seamless integration into our deep learning model.

5.4.5 Training the Deep Learning Model

The training of our deep learning model is a pivotal phase in our malware detection pipeline, where the neural network learns to make predictions based on the input data. This process involves several key steps:

1. **Data Input via tf.data.Dataset**: We begin by inputting our data using the TensorFlow `tf.data.Dataset` pipeline [43]. This allows us to efficiently feed the data into the model in batches of 64 padded samples, ensuring that GPU memory consumption remains manageable. To enhance the randomness of our data, we shuffle the samples before feeding them into the model.

2. **Model Configuration**: Before training, we configure the model with the predefined hyperparameters. These hyperparameters, which determine the model's architecture and learning strategy, have been meticulously tuned for optimal performance (for detailed hyperparameter descriptions, please refer to the "Hyperparameter Optimization" section in this chapter).

3. **Training Process**: The training process commences with the model being trained using the pairs of embedded feature vectors and their corresponding labels from the training dataset. Here, we employ the binary cross-entropy loss function as the optimization objective, aiming to minimize the discrepancy between predicted and actual labels.

4. **Validation**: To assess the model's performance during training and prevent overfitting, we leverage the validation dataset. After each epoch, the model is evaluated using the validation data to gauge its ability to generalize to unseen samples.

5. **Early Stopping:** Throughout the training process, we implement Early Stopping to prevent overfitting and optimize training efficiency. This technique continuously monitors the model's performance on the validation dataset after each epoch, specifically focusing on validation accuracy. If the validation accuracy fails to improve for three consecutive epochs, the training process is halted prematurely. Early Stopping ensures that the model does not continue learning when it ceases to benefit from further training, ultimately leading to a more efficient and well-generalized model.

In summary, this step involves configuring the model, inputting data efficiently, and training the neural network with the specified hyperparameters. The validation dataset plays a crucial role in ensuring the model's robustness and generalization capabilities throughout the training process.

5.5 Execution and Testing of the Model

In this section, we delve into the execution and testing of our deep learning model. This phase is crucial as it allows us to evaluate the model's performance on unseen data, ultimately assessing its ability to accurately classify executables as malicious or benign. The execution and testing process closely mirrors the training process described earlier, but with some key distinctions that we will explore in the subsequent sections.

Throughout this section, we will outline the steps involved in processing test data, feature extraction, feature embedding, and the forward pass through the deep learning model. These steps collectively provide insights into how the model performs on real-world executables and its effectiveness in malware detection. By drawing parallels with the training process and highlighting the differences, we aim to provide a comprehensive understanding of the model's execution and testing phase.

5.5.1 Feature Extraction

During the execution phase, the feature extraction process remains consistent with that of the training phase.

As previously described, each sample, which represents an executable program, undergoes the same procedure. First, the program's instructions are processed using the TensorFlow tokenizer class, resulting in a tokenized and indexed sequence. From this tokenization, we extract a word count for each instruction, creating a list of tuples pairing each instruction with its frequency of occurrence within the sample.

To ensure uniformity and manageable input sizes, we follow the same procedure as in training by ordering this list of instruction-frequency tuples from the most frequent to the least frequent. If the list exceeds a predefined length, typically set to 600, it is truncated to maintain consistency with the model's input requirements. This feature extraction process guarantees that each executable program is represented as a consistent feature vector, preserving the order and frequency of its instructions.

5.5.2 Feature Embedding

In the execution phase, the feature embedding process closely resembles that of the training phase. Its purpose remains to transform the feature vectors, which represent executable programs, into input suitable for the deep learning model.

As previously established, the Gensim Word2Vec model, trained on the training and validation data, has generated an embedding matrix that associates each instruction in the vocabulary with a unique embedding vector. This matrix has dimensions $|V| \times 300$, where $|V|$ represents the vocabulary size, and 300 signifies the chosen embedding dimension.

During feature embedding, we use this pre-trained embedding matrix to map each instruction in the feature vector to its corresponding embedding vector. However, one significant difference arises during execution. If an instruction within a feature vector was not previously encountered during the word embedding model's training, it is discarded. This omission ensures that the input to the model adheres to the vocabulary learned during training and maintains consistency with the Word2Vec embeddings.

Ultimately, this feature embedding process generates a sequence of embedding vectors for each feature vector, representing the instructions within the executable program. These embedded sequences serve as the model's input for the execution phase, facilitating the classification of the programs.

5.5.3 Evaluating and Executing th Deep Learning Model

In the execution phase of the deep learning model, the previously trained model is utilized to generate predictions regarding whether a given sample is malicious or benign based on its embedded feature vector. Consequently, during model evaluation, the primary function of the model is to predict the class labels of test samples. These predictions are then compared with the true labels of the samples to assess the model's performance.

In the execution phase, the process commences with the embedded feature vectors of the test samples. These feature vectors, undergo a feedforward process through the previously trained model. During this feedforward pass, the sequences of embedding vectors, are sequentially processed by the model's layers, including LSTM or BiLSTM layers, as well as the dense layer. Notably, unlike the training phase, this execution phase does not involve the adjustment of model parameters.

At the end of this feedforward process, the model outputs a probability for each sample. This probability reflects the model's assessment of the likelihood that the given sample is a malicious program. If the probability is greater than 0.5, the model predicts the sample as malicious; otherwise, it predicts it as benign.

When evaluating the model with the test data, this predicted probability is compared with the true label of the sample. The true label denotes whether the sample is genuinely a benign or malicious program. Using this comparison, various evaluation metrics are calculated to assess the model's performance. These metrics include accuracy, precision, recall, F1-score, and the area under the ROC curve (AUC-ROC).

The combination of the predicted probabilities and true labels, along with these evaluation metrics, forms the basis for evaluating the deep learning model's performance in identifying and classifying malicious software.

5.6 Hyperparameter Optimization

5.6.1 Search Space

Defining a well-structured search space is a pivotal step in the process of HPO. The search space determines the range and diversity of configurations explored during HPO, ultimately influencing the performance and efficiency of the machine learning model. In our study, we meticulously crafted the search space, considering parameters that significantly impact the deep learning model's architecture and training process.

To establish the boundaries of our search space, we drew inspiration from previous works in the field [15, 14]. These works provided valuable insights into the ranges of HPCs that have shown promise in similar tasks.

Table 5.4 summarizes the HPs included in our search space and their respective ranges. Each HP, such as epochs, batch size, depth, bidirectional layers, number of neurons, and dropout rate, was carefully selected to cover a wide spectrum of potential configurations. This comprehensive search space serves as the foundation for our HPO process, allowing us to explore a diverse array of model configurations and identify the most effective ones.

Table 5.4: HPO search space

Hyperparameter	**Range**
Epochs	1 - 100
Batch Size	1 - 128
Depth	1 - 3
Bidirectional	True, False
LSTM Dimension	1 - 320
Dropout Rate	0, 0.1, 0.2, 0.3, 0.4, 0.5

Here, we provide detailed descriptions for each hyperparameter in our search space:

- **Epochs:** The number of training epochs determines how many times the deep learning model iteratively processes the entire training dataset during training. It influences the model's capacity to learn patterns from the data but can also lead to overfitting if set too high.
- **Batch Size:** The batch size specifies how many data samples are processed in each forward and backward pass through the neural network during one training epoch. It affects the computational efficiency and convergence behavior of the model.
- **Depth:** The depth refers to the number of LSTM or BiLSTM layers in the model architecture. It determines the complexity and depth of the model's memory and contextual understanding.
- **Bidirectional:** This binary hyperparameter controls whether the model is build with LSTM or BiLSTM. Bidirectional layers consider context from both past and future data points, but increase the complexity of the model.
- **LSTM Dimension:** The LSTM dimension defines the width or capacity of the LSTM or BiLSTM layers. It impacts the model's ability to capture complex relationships in the data while increasing model complexity.
- **Dropout Rate:** Dropout is a regularization technique that helps prevent overfitting by randomly dropping a fraction of neurons during training. The dropout rate specifies the probability of dropping neurons during each training iteration.

These hyperparameters collectively influence the architecture and behavior of the deep learning model, making their optimization a critical step in achieving optimal performance.

5.6.2 Hyperparameter Optimization Setup

In the pursuit of optimizing hyperparameters for our deep learning model, we employed the Hyperband algorithm, as previously elucidated in subsection 3.15.3. This section outlines the configuration of the Hyperband algorithm and the optimization process we employed.

Hyperband Configuration

The Hyperband algorithm offers an efficient approach to hyperparameter optimization (HPO). To set up the Hyperband algorithm, we configured two crucial parameters: the hyperband control multiplier (η_{HB}) and the upper limit of fidelity (λ_{fid}^{upp}). We set η_{HB} to 3, controlling the schedule of evaluations within Hyperband. This choice balances exploration and exploitation during the optimization process.

The upper limit of fidelity (λ_{fid}^{upp}) is dynamically determined by the Optuna HPO framework [44]. Optuna ensures that the fidelity budget is allocated efficiently, adapting λ_{fid}^{upp} based on the optimization progress. This adaptability enables us to efficiently allocate computational resources, focusing on the most promising HPCs.

Optimization Tool: Optuna

Leveraging the Optuna framework [44], the HPO process was streamlined and automated. Optuna's capabilities, including optimization algorithms and support for result visualization, offered an effective and reliable solution.

Initially, we considered using Keras Tuner for HPO, but encountered challenges when visualizing the results. Optuna's support for result visualization offered a more effective and reliable solution for our needs.

Objective Function

Throughout the HPO process, the objective centered on maximizing the validation accuracy of the deep learning model. In each HPC evaluation, a model underwent training using the training dataset and subsequent evaluation with the validation dataset. The accuracy attained on the validation dataset guided the algorithm's exploration of the hyperparameter space.

Evaluation on Test Data

Upon identifying the HPC configuration yielding the highest validation accuracy, this model underwent evaluation on previously unseen data, specifically the test dataset. Results from this evaluation, including metrics encompassing accuracy, precision, recall, and F1-score, await presentation in the subsequent chapter.

This configuration ensured an effective and systematic exploration of HPC, encompassing 350 trials, ultimately culminating in a well-optimized deep learning model tailored to the task at hand.

5.7 Experimental Setup and Infrastructure

In this section, we provide an overview of the experimental setup and infrastructure used for training and executing the proposed deep learning model. The experiments were conducted on the cluster SERT [45] provided by the Department of Computer Architecture at the Polytechnic University of Catalonia.

5.7.1 Hardware Configuration

The cluster is composed of multiple nodes. The primary components of the node used in this research are detailed below:

Processor

The node is equipped with Intel Xeon Silver 4210R processors, each running at 2.40 GHz. These processors, known for their multicore architecture, contribute to efficient parallel processing.

Memory

32 GB of RAM were allocated for the job execution. This memory allocation ensures sufficient resources for the specific tasks carried out during the job, contributing to smooth execution and efficient processing.

Graphics Processing Unit (GPU)

The job made use of one of the NVIDIA RTX 2080TI GPUs available on the node. With 11 GB of GDDR6 memory and connected via PCIe, this GPU played a pivotal role in accelerating the deep learning computations associated with the job. The allocation of this high-performance GPU enhances the speed and efficiency of both training and inference tasks related to the deep neural network.

Node Specifications

The specific node used for job execution features the following specifications:

- 2x Intel Xeon Silver 4210R processors at 2.40 GHz

- 128 GB of RAM
- 2 x 480 GB SSDs
- 2 x 2TB NVMEs
- 2 x 10 Gigabit Ethernet network cards
- 8 NVIDIA RTX 2080TI GPUs with 11 GB GDDR6 memory each, connected via PCIe

These robust hardware configurations collectively enable efficient parallelization and accelerated processing for the computational demands of our research.

5.7.2 Software Configuration

The success of the experiments hinges on the careful selection and configuration of software tools and libraries for model development, training, and evaluation. The following tools, along with their respective versions, were integral to the project:

Programming Language

- **Python** [46] (version 3.8.18): Programming language providing a flexible and powerful environment for machine learning tasks.

Machine Learning Libraries

- **TensorFlow** [42] (version 2.11.1): Primary library for constructing and training the neural network model.
- **Scikit-learn** [47] (version 1.3.2): Utilized for various machine learning tasks, including preprocessing, modeling, and evaluation.
- **Gensim** [48] (version 4.3.2): Employed for the word embedding.
- **Optuna** [44] (version 3.4.0): Used for hyperparameter optimization.

Machine Learning Components

- **Word2Vec** from Gensim: Applied for word embedding tasks.
- Various layers including **Bidirectional, LSTM, Dense, Dropout** from tensorflow.keras.layers: Configured for building the neural network architecture.
- **Tokenizer** from tensorflow.keras.preprocessing.text: Used for tokenizing text data.

- **EarlyStopping** from tensorflow.keras.callbacks: Implemented for early stopping during model training.
- **train_test_split** from sklearn.model_selection: Employed for splitting the dataset into training and testing sets.
- Various metrics including **accuracy_score, precision_score, recall_score, f1_score, roc_curve, roc_auc_score** from sklearn.metrics: Utilized for evaluating model performance.

General-Purpose Libraries

- **Pandas** [49] (version 2.0.3): Used for data manipulation and analysis.
- **Matplotlib** [50] (version 3.7.3): Employed for data visualization and plotting.
- **Joblib** [51] (version 1.3.2): Used for efficient parallel computing and data caching.

Command-Line Tools

- **objdump**: Used for disassembling executable files and extracting information about the binaries.
- **sed**: Employed for stream editing, facilitating text transformations in the preprocessing stage.
- **awk**: Used for pattern scanning and text processing.
- **curl**: Utilized for HTTP-based file downloads in the project.

5.7.3 Cluster Configuration

The cluster resources were managed using Slurm [52]. This system facilitated job scheduling, resource allocation, and parallel computing across multiple machines.

By providing a detailed overview of the experimental setup and infrastructure, we aim to ensure transparency and reproducibility of the research outcomes.

Chapter 6

Results

6.1 Hyperparameter Optimization Results

Following a meticulous HPO process, we generated valuable insights into the critical factors influencing the performance of our deep learning model tailored for the detection of malware within executable files. Two graphical representations shed light on these findings, providing a comprehensive understanding of the HPs' importance and the objective function values achieved during the HPO.

6.1.1 Hyperparameter Importance Analysis

Figure 6.1 illustrates the relative importance of each HP in optimizing our model's performance. The analysis reveals that the choice between LSTM and BiLSTM layers holds the highest importance, accounting for a substantial 72% of the optimization process. Following, the dropout HP plays a moderate role with an importance rating of 17%. Meanwhile, other HP, including the dimension of the first layer (`lstm_units_0`), batch size, number of epochs, and depth, collectively contribute to the remaining 11%. Although these HPs exhibit lower individual importance, they collectively impact the model's performance by fine-tuning various aspects. This comprehensive understanding of HP importance guides our model's optimization process effectively.

6.1.2 Objective Function Analysis

Figure 6.2 provides a detailed insight into the objective function values, specifically the validation accuracy, obtained during the HPO for each HP.

Batch Size: The batch size tends to be consistently below 50 during the HPO, suggesting a preference for smaller batch sizes.

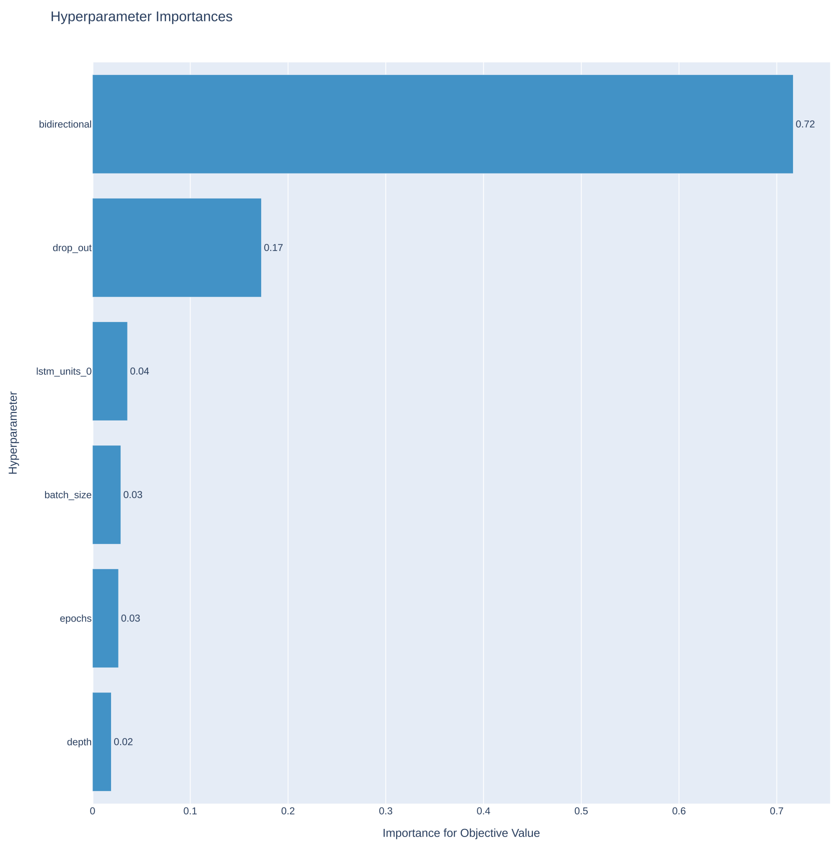

Figure 6.1: Relative Importance of Hyperparameters

Bidirectional: The analysis indicates a clear preference for BiLSTM layers over LSTM layers. The configurations with LSTM layers resulted in lower validation accuracies, notice that all the trials that obtained a validation accuracy below 0.8 where build with LSTM layers.

Depth: Optimal results continue to be achieved for all three depth options of this HP. The HPO frequently selects 3 layers, indicating its ability to achieve slightly favorable validation accuracies with a complex model.

Epochs: The number of epochs tends to hover around 25. This behavior can be attributed to the inclusion of Early Stopping in the training process, which prevents overfitting. Consequently, the model typically converges within approximately 25 epochs, regardless of the initially configured number of epochs.

LSTM Dimension: The dimension of LSTM or BiLSTM layers varies across the layers. The first layer predominantly falls within the range of 100 to 320, suggesting the importance of capturing complex relationships in the initial layer.

In contrast, the second layer tends to have lower dimensions, typically below 100. Similarly, the third layer follows this trend, with dimensions typically below 200. These observations reflect the diversity in layer dimensions, potentially enhancing the model's ability to capture different levels of abstraction in the data.

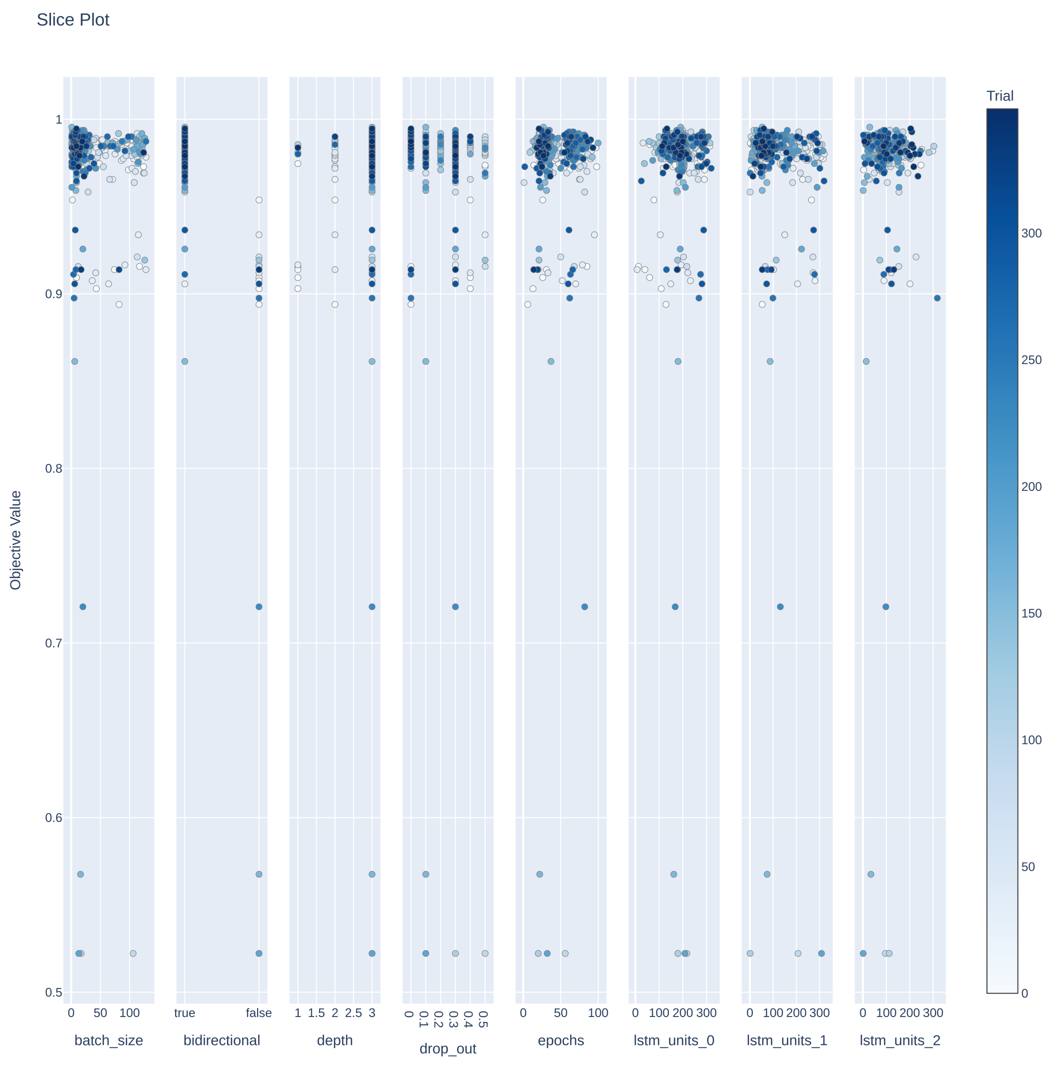

Figure 6.2: Validation Accuracy Values for Hyperparameters

These findings contribute valuable insights into the configuration of our deep learning model and guide its optimization for effective malware detection.

6.1.3 Validation Accuracy Progression

Figure 6.3 provides a visual representation of the progression of validation accuracy throughout the HPO trials. In this plot, each trial is depicted along the x-axis, with the corresponding validation accuracy on the y-axis.

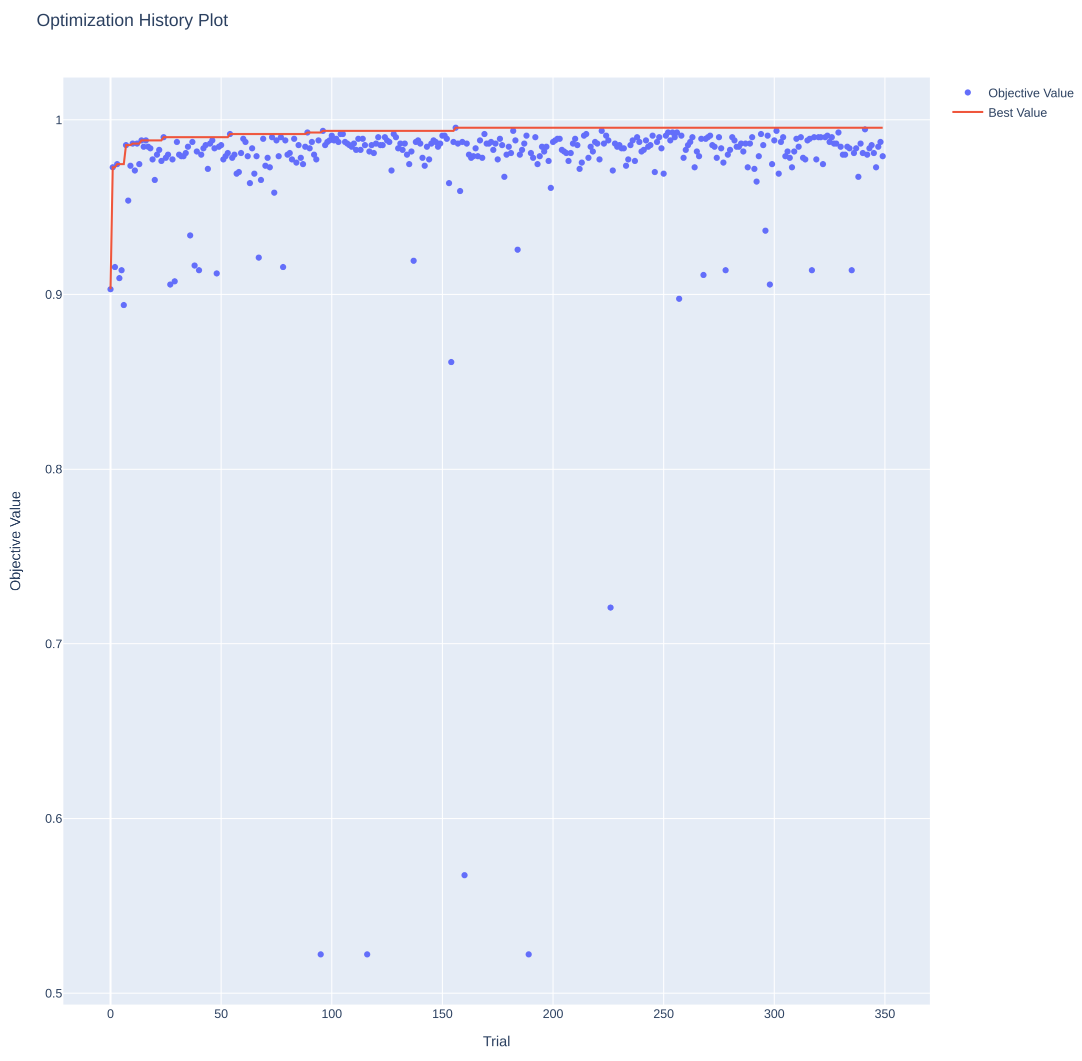

Figure 6.3: Validation Accuracy and Best Validation Accuracy Progression During Hyperparameter Optimization

Within the same plot, an additional representation showcases the highest validation accuracy achieved up to the corresponding trial. This secondary line offers valuable insights into the model's performance evolution throughout the optimization process, revealing whether it experienced substantial improvements or reached a stable plateau. Notably, this line demonstrates a rapid ascent, ultimately leveling off around the 20th trial. This suggests that a substantial reduction in the number of trials could potentially yield nearly identical results, given the convergence of validation accuracy observed during the early stages of optimization.

It's noteworthy that the majority of the trials consistently achieve validation accuracy scores above 0.95. This observation indicates that the model's performance approaches a high level of accuracy, leaving limited room for further significant improvements. These results underscore the effectiveness of the HPO in fine-tuning the model for optimal performance within the selected configuration.

6.1.4 Time Analysis

The time analysis involves an examination of the time taken for each individual trial during the HPO. Figure 6.4 illustrates the time progression, with the x-axis representing time and the y-axis representing the trial number. Notably, there is a consistent duration for each trial, with minimal variations in execution time. This pattern can be attributed to the model's implementation with early stopping, which ensures that trials do not extend beyond the necessary training time.

The entire hyperparameter optimization process, encompassing 350 trials, was completed in less than 6 hours.

6.2 Second Hyperparameter Optimization

The initial HPO results have unequivocally highlighted the paramount importance of the choice between BiLSTM and LSTM layers, with BiLSTM consistently delivering superior performance. With this crucial insight in mind and recognizing that further improvements may be challenging due to the already high validation accuracy, we decided to embark on a second HPO.

In this subsequent HPO, we exclusively utilized BiLSTM layers, reducing the search space while aiming to fine-tune the remaining HPs for optimal results. This strategic choice is driven by the strong indications from the prior HPO that the BiLSTM architecture holds the key to our model's effectiveness. The search space for the rest of the HPs remains the same, shown in table 5.4.

In the following sections, we present the results of this second HPO, comparing them with the outcomes of the initial HPO to gain a comprehensive understanding of how this focused optimization effort further enhances our deep learning model for malware detection.

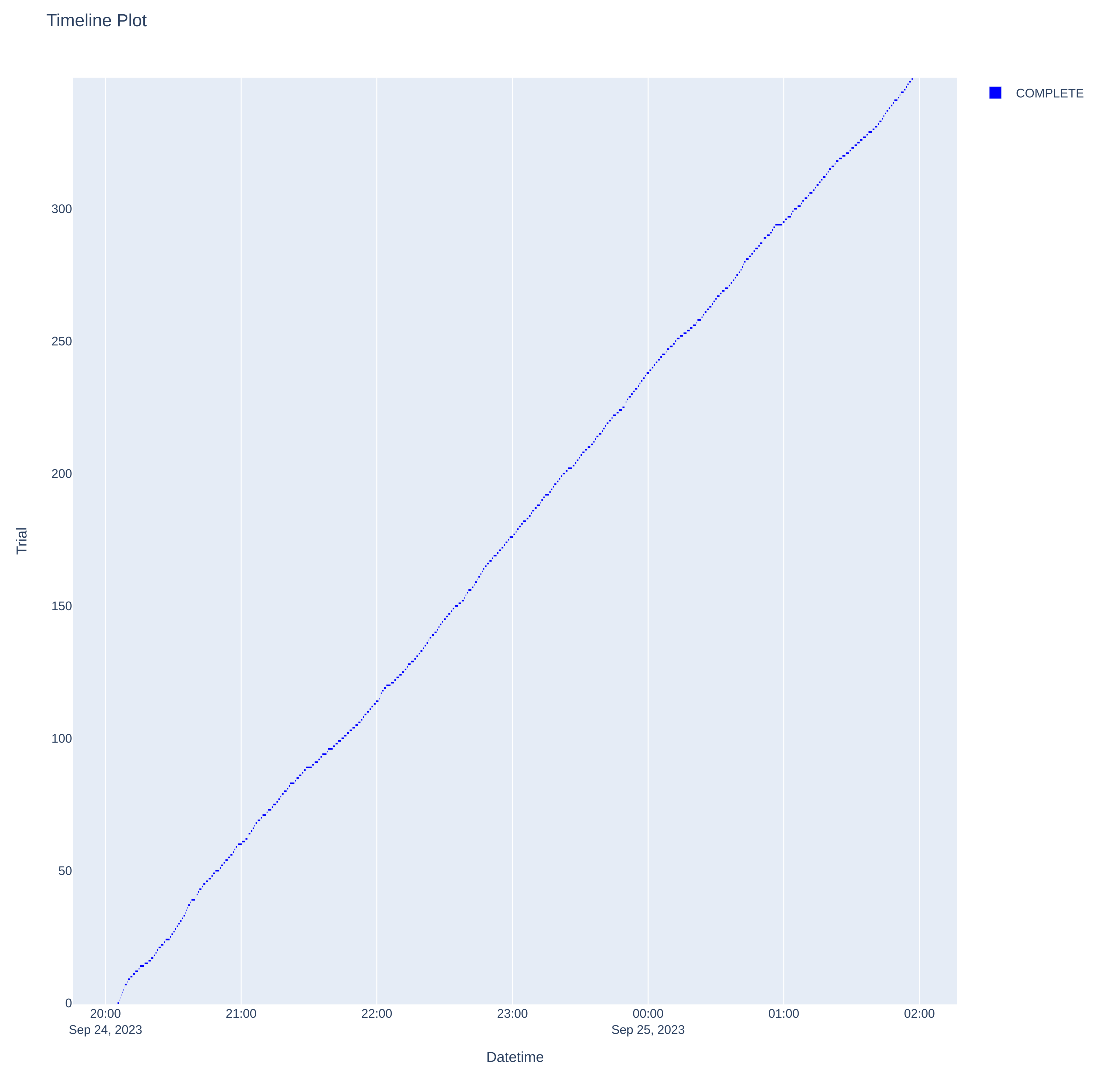

Figure 6.4: Time Taken for Each Trial During Hyperparameter Optimization

6.2.1 Hyperparameter Importance Analysis

The hyperparameter importance plot, depicted in Figure 6.5, reveals notable shifts in HP importance compared to the previous HPO, where the choice between LSTM and BiLSTM layers held paramount significance. In this second round of optimization, as the model architecture is constrained to BiLSTM, other HP come to the forefront.

The analysis shows that the most pivotal HP is the number of epochs, accounting for a substantial 64% of the optimization process. Following closely, the dimension

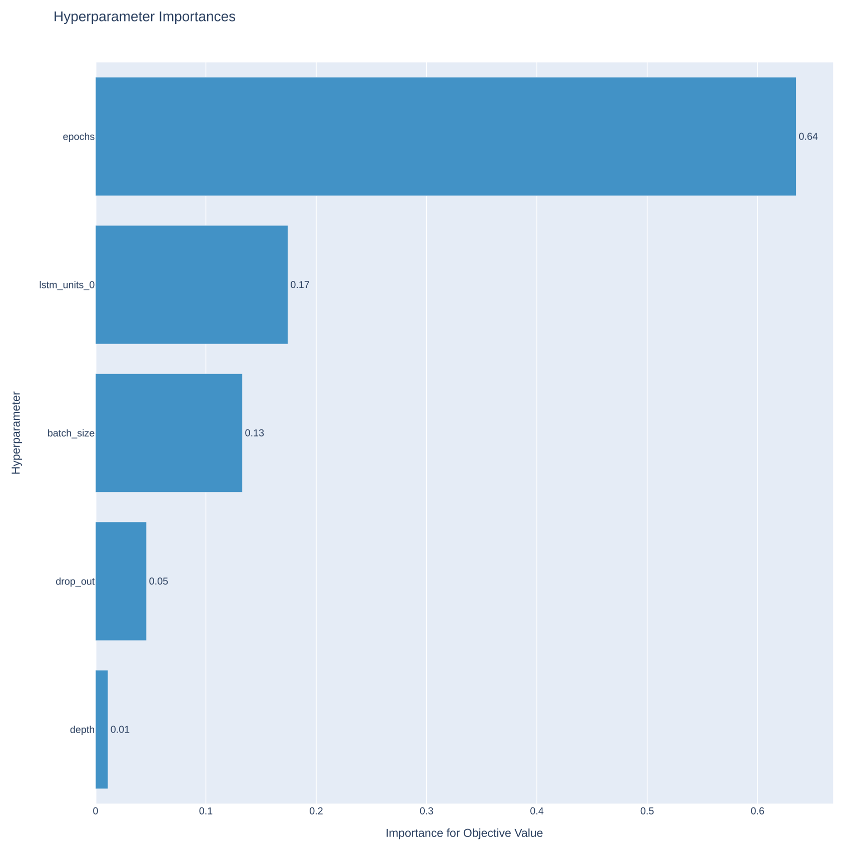

Figure 6.5: Relative Importance of Hyperparameters in the Second HPO

of the first BiLSTM layer plays a significant role with a weight of 17%, while the batch size contributes 13%. In contrast, the dropout rate and depth exhibit lower importance levels, at 5% and 1%, respectively.

These altered HPs importances underscore the model's adaptability to different architectural constraints and the evolving focus of the optimization process towards enhancing training dynamics, batch processing, and layer dimensions.

6.2.2 Objective Function Analysis

Figure 6.6 provides a detailed insight into the validation accuracy across trials for each HP. The results indicate a consistent preference for a two-layer model in the HPO. In this configuration, the first BiLSTM layer typically exhibited dimensions within the range of 50 to 200, while the second BiLSTM layer featured smaller dimensions, primarily below 100. This configuration highlights the importance of effectively managing the model's complexity while retaining its capacity to capture critical patterns within the data.

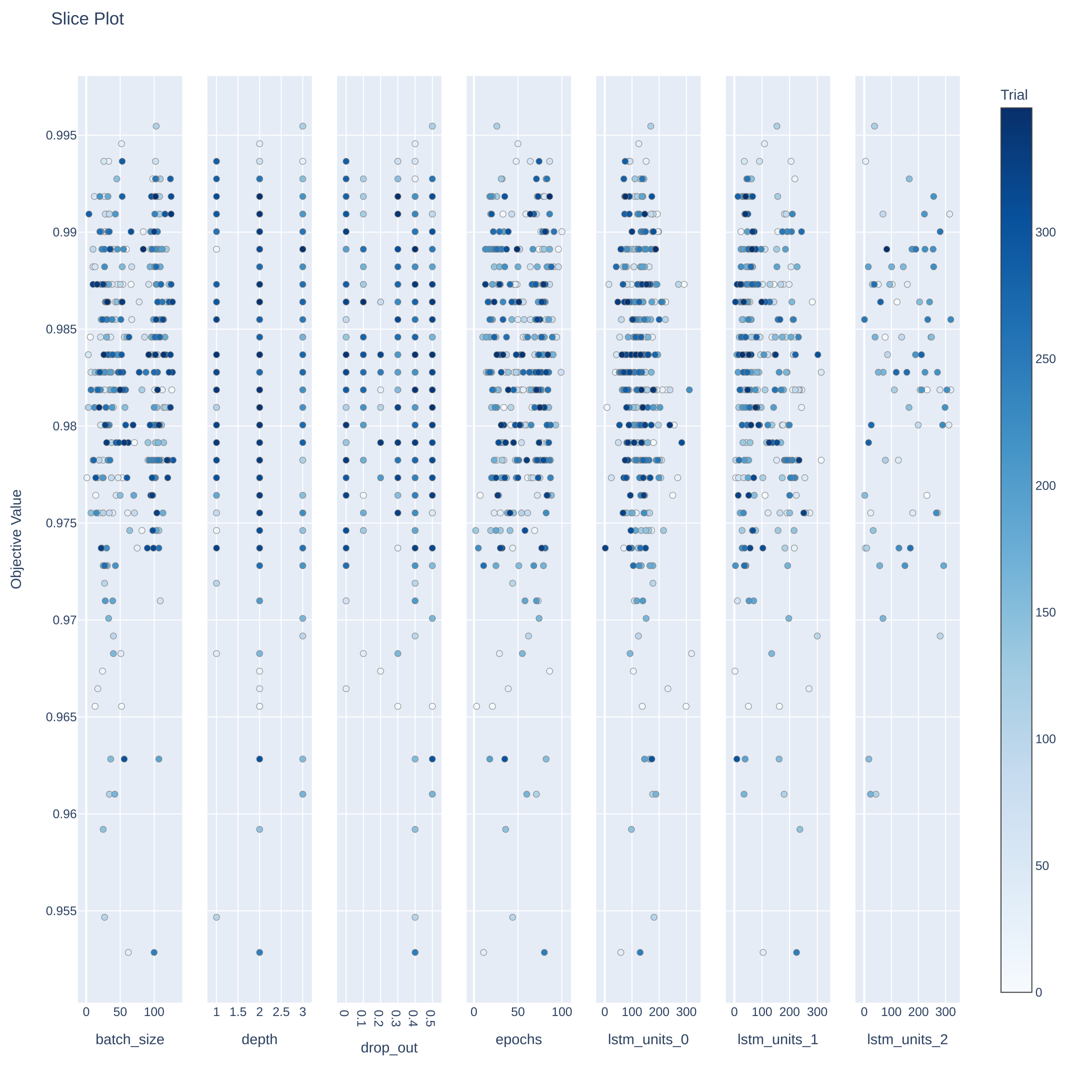

Figure 6.6: Validation Accuracy Values for Hyperparameters in the Second HPO

6.2.3 Validation Accuracy Progression

The Validation Accuracy Progression plot, presented in Figure 6.7, offers insight into the model's performance across trials during the second HPO.

In contrast to the previous HPO, where the validation accuracies exhibited a wider range of performance, all trials in this optimization cycle consistently achieve

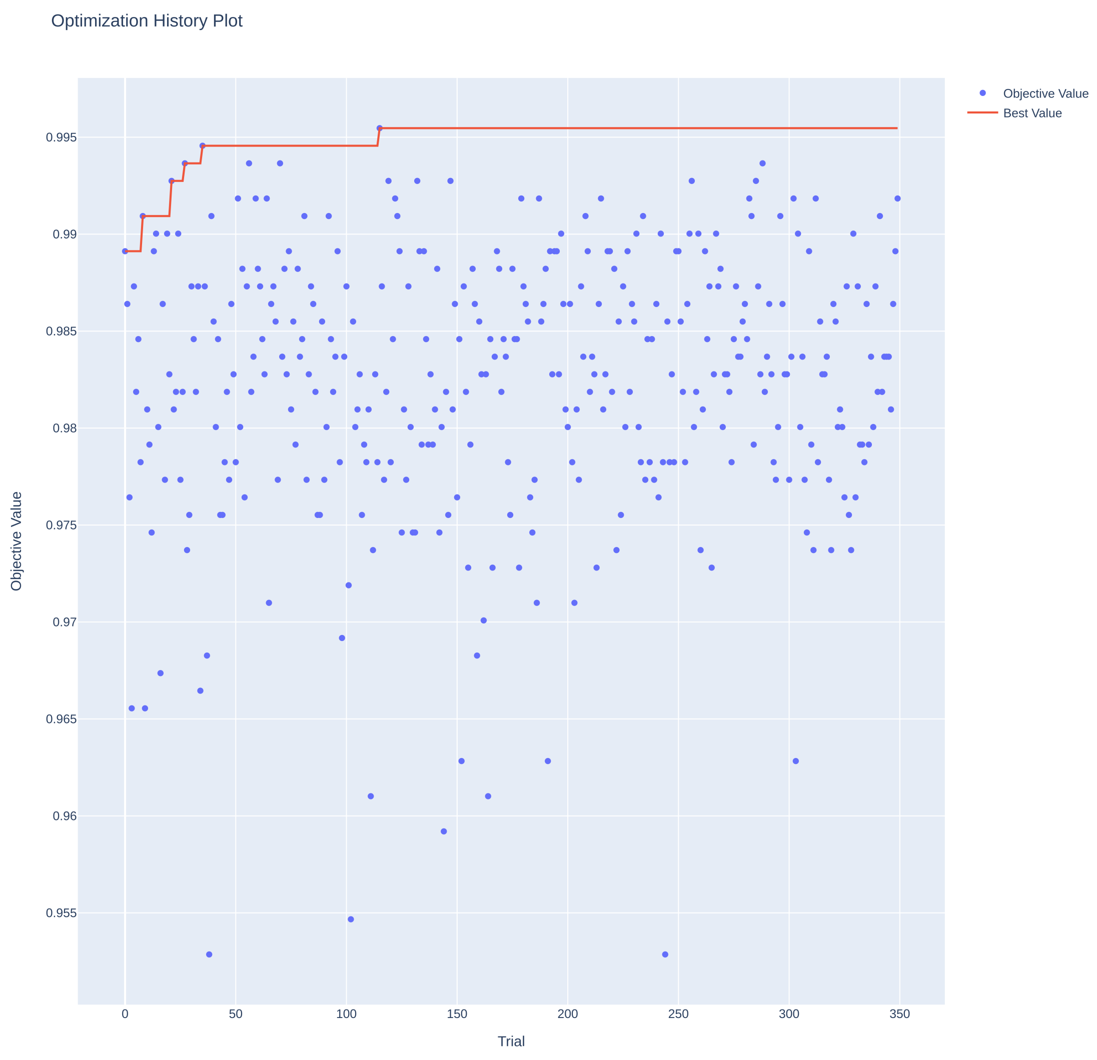

Figure 6.7: Validation Accuracy and Best Validation Accuracy Progression During Hyperparameter Optimization in the Second HPO

validation accuracies above 0.94. This indicates that the model's constrained architecture, based on BiLSTM layers, consistently performs well, leaving less room for performance variation.

Notably, similar to the earlier HPO, the red line denoting the best accuracy encountered during the optimization process ascends rapidly and then reaches a plateau around the 30th trial. This observation suggests that while the model performs consistently well, achieving further improvements in accuracy becomes challenging beyond this point.

These findings highlight the reliability and consistency of the BiLSTM model's performance, showcasing its ability to generalize effectively across various trials while also demonstrating the practical limits of performance gains in this particular configuration.

6.2.4 Time Analysis

The Timeline plot, as depicted in Figure 6.8, offers an overview of the time taken for each trial during the second HPO.

The timeline for this HPO exhibits a similar pattern to the first optimization cycle, with trials taking a comparable amount of time. However, a notable improvement is observed; this second HPO was completed more efficiently, taking only 5 hours instead of the previous 6 hours. This reduction in time can likely be attributed to the constrained search space, as the HPO is now exclusively focused on models utilizing BiLSTM layers.

This efficiency suggests that narrowing down the scope of the hyperparameter search not only maintains consistent performance but also streamlines the optimization process, making it more time-effective while achieving similar results. Such efficiency is a valuable consideration when deploying models in real-world scenarios, where computational resources and time are often limited.

6.3 Selected Model Evaluation

Following the HPO process, the model's optimal hyperparameters were determined as detailed in Table 6.1. To assess the model's performance, it was constructed using these hyperparameters and subsequently trained. The evaluation was conducted using previously unseen data, specifically the test dataset, and the results are presented below.

Two figures were generated for the evaluation. The first pair of graphs shows the accuracy and loss during the training and validation of the model for each epoch. Notably, after the 8th epoch, the model exhibits consistent behavior, maintaining an accuracy consistently above 0.98 and a loss consistently below 0.1. This observation demonstrates that the selected hyperparameter configuration, determined through HPO, effectively generalizes well to both the training and validation datasets.

With confidence in the model's performance on the training and validation datasets, the evaluation using the test dataset was carried out. The following metrics were calculated to assess the model's effectiveness:

The model demonstrates impressive performance even when applied to previously unseen data, a testament to its robustness and generalization capabilities. Additionally, the ROC curve, depicted in Figure 6.10, swiftly reaches a True Positive

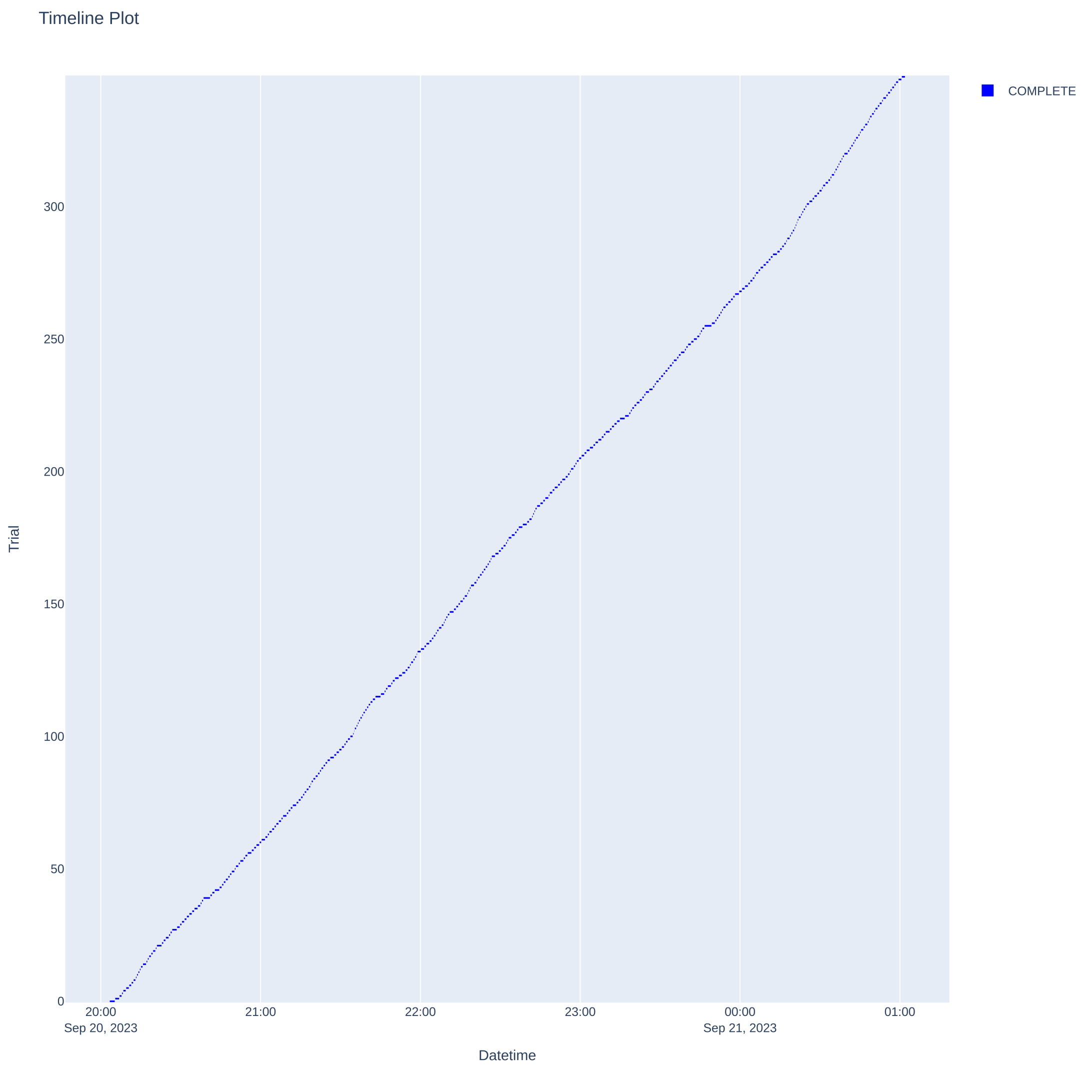

Figure 6.8: Time Taken for Each Trial During Hyperparameter Optimization in the Second HPO

rate of 1, reaffirming the exceptional performance of the selected hyperparameter configuration.

Table 6.1: Selected Hyperparameters After HPO

Hyperparameter	**Value**
Epochs	26
Batch Size	103
Depth	3
Bidirectional	True
Layer 1 Dimension	169
Layer 2 Dimension	154
Layer 3 Dimension	38
Dropout Rate	0.5

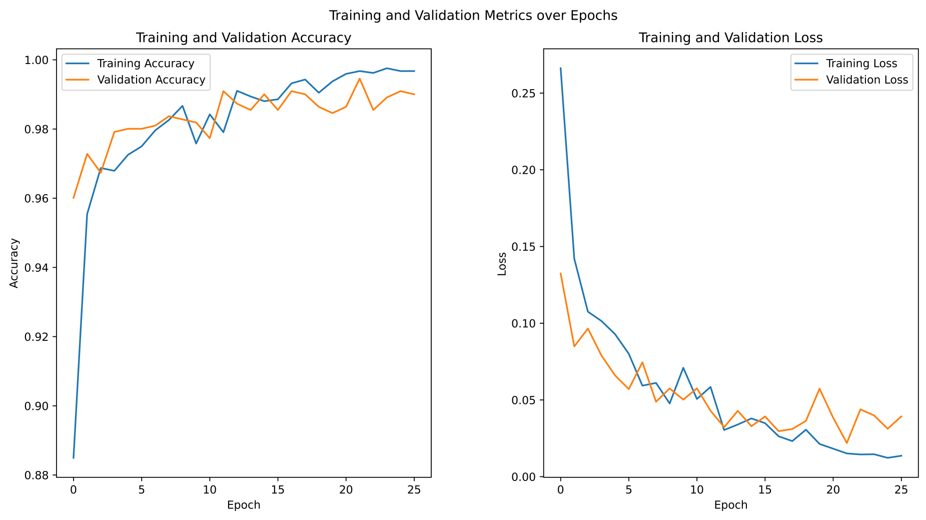

Figure 6.9: Accuracy and Loss during Training and Validation

Table 6.2: Model Evaluation Metrics

Metric	**Value**
Accuracy	0.983
Precision	0.995
Recall	0.969
F1-score	0.982
ROC AUC	0.998

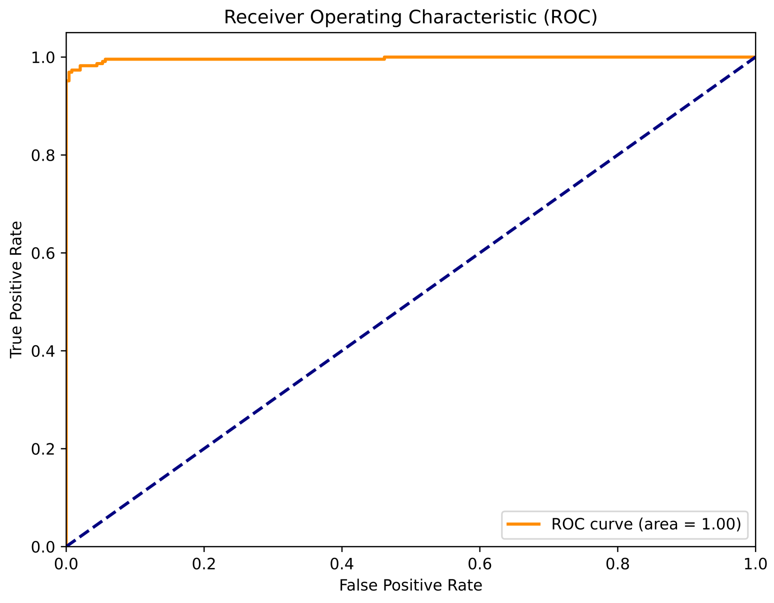

Figure 6.10: Receiver Operating Characteristic (ROC) Curve

Chapter 7

Conclusion

This research endeavor embarked on a quest to advance the field of malware detection through the application of deep learning techniques. Our journey has unveiled significant insights and contributions that underscore the potential of machine learning models in addressing the ever-evolving landscape of cyber threats.

The fundamental contributions of this study can be summarized as follows:

- **Deep Learning Model Development:** We designed, developed, and optimized a deep learning model tailored explicitly for the identification of malware within executable files. This model, based on LSTM and BiLSTM architectures enhanced with word embedding, has demonstrated exceptional capabilities in discerning intricate patterns indicative of malicious activity.
- **Hyperparameter Optimization:** Our meticulous HPO process offered valuable insights into the pivotal role of hyperparameters in model performance. Notably, the selection between LSTM and BiLSTM layers, its dimension, and the number of training epochs emerged as critical factors influencing the model's accuracy.
- **Evaluation and Generalization:** The selected optimal hyperparameters, determined through HPO, facilitated the construction of a model that not only excelled during training and validation but also exhibited remarkable performance when evaluated with unseen data, as indicated by high accuracy, precision, recall, F1-score, and AUC-ROC.
- **Practical Implications:** The results of this research have practical implications for bolstering cybersecurity measures. Our model's ability to effectively detect malware within executable files contributes to the development of more robust and adaptive security systems, critical for safeguarding digital ecosystems.

Appendix A

Functions

In this annex, we provide some key mathematical functions used in our thesis. These functions are essential components of various neural network architectures and data processing techniques.

A.1 Sigmoid

The sigmoid function, also known as the logistic function, is a widely used activation function in neural networks. It maps any real-valued number to a value between 0 and 1, making it suitable for modeling binary classification problems and introducing non-linearity to neural network layers.

The sigmoid function is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Figure A.1 shows a graphical representation of the sigmoid function.

A.2 Hyperbolic Tangent

The hyperbolic tangent function, often denoted as $\tanh(x)$, is another commonly used activation function in neural networks. It maps real-valued numbers to values between -1 and 1, similar to the sigmoid function. The tanh function introduces non-linearity and is particularly useful for modeling data with zero-centered properties.

The hyperbolic tangent function is defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Figure A.2 illustrates the graph of the hyperbolic tangent function.

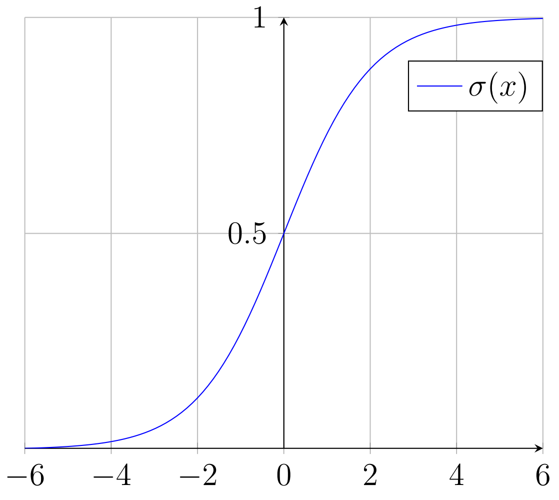

Figure A.1: Plot of the Sigmoid Function

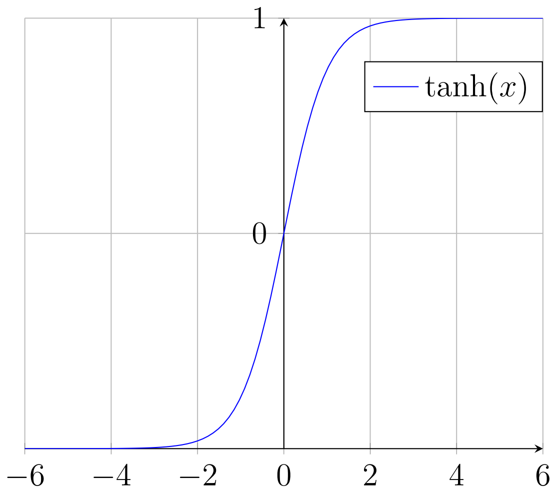

Figure A.2: Plot of the Hyperbolic Tangent Function

Bibliography

[1] McAfee Labs. *McAfee COVID-19 Report Reveals Pandemic Threat Evolution.* Accessed on Date. Year of publication. URL: `https://www.mcafee.com/blogs/other-blogs/mcafee-labs/mcafee-covid-19-report-reveals-pandemic-threat-evolution/` (cit. on p. 6).

[2] Mihai Christodorescu and Somesh Jha. «Static analysis of executables to detect malicious patterns». In: *12th USENIX Security Symposium (USENIX Security 03).* 2003 (cit. on p. 8).

[3] Igor Santos, Felix Brezo, Javier Nieves, Yoseba K Penya, Borja Sanz, Carlos Laorden, and Pablo G Bringas. «Idea: Opcode-sequence-based malware detection». In: *Engineering Secure Software and Systems: Second International Symposium, ESSoS 2010, Pisa, Italy, February 3-4, 2010. Proceedings 2.* Springer. 2010, pp. 35–43 (cit. on p. 8).

[4] Ammar AE Elhadi, Mohd A Maarof, and Ahmed H Osman. «Malware detection based on hybrid signature behaviour application programming interface call graph». In: *American Journal of Applied Sciences* 9.3 (2012), p. 283 (cit. on p. 8).

[5] Dan Fleck, Arnur Tokhtabayev, Alex Alarif, Angelos Stavrou, and Tomas Nykodym. «Pytrigger: A system to trigger & extract user-activated malware behavior». In: *2013 International Conference on Availability, Reliability and Security.* IEEE. 2013, pp. 92–101 (cit. on p. 8).

[6] Konstantin Berlin, David Slater, and Joshua Saxe. «Malicious behavior detection using windows audit logs». In: *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security.* 2015, pp. 35–44 (cit. on p. 8).

[7] M Elif Karsligl, A Gökhan Yavuz, M Amaç Güvensan, Khadija Hanifi, and Hasan Bank. «Network intrusion detection using machine learning anomaly detection algorithms». In: *2017 25th Signal Processing and Communications Applications Conference (SIU).* IEEE. 2017, pp. 1–4 (cit. on p. 8).

[8] Anıl Utku, İbrahim Alper Doğru, and M Ali Akcayol. «Decision tree based android malware detection system». In: *2018 26th Signal Processing and Communications Applications Conference (SIU)*. IEEE. 2018, pp. 1–4 (cit. on p. 8).

[9] Philip O'Kane, Sakir Sezer, Kieran McLaughlin, and Eul Gyu Im. «SVM training phase reduction using dataset feature filtering for malware detection». In: *IEEE transactions on information forensics and security* 8.3 (2013), pp. 500–509 (cit. on p. 8).

[10] Hyoil Han, SeungJin Lim, Kyoungwon Suh, Seonghyun Park, Seong-je Cho, and Minkyu Park. «Enhanced android malware detection: An svm-based machine learning approach». In: *2020 IEEE International Conference on Big Data and Smart Computing (BigComp)*. IEEE. 2020, pp. 75–81 (cit. on p. 8).

[11] Carlos Domenick Morales-Molina, Diego Santamaria-Guerrero, Gabriel Sanchez-Perez, Hector Perez-Meana, and Aldo Hernandez-Suarez. «Methodology for malware classification using a random forest classifier». In: *2018 IEEE International Autumn Meeting on Power, Electronics and Computing (ROPEC)*. IEEE. 2018, pp. 1–6 (cit. on p. 8).

[12] Danish Vasan, Mamoun Alazab, Sobia Wassan, Babak Safaei, and Qin Zheng. «Image-Based malware classification using ensemble of CNN architectures (IMCEC)». In: *Computers & Security* 92 (2020), p. 101748 (cit. on p. 8).

[13] Guosong Sun and Quan Qian. «Deep learning and visualization for identifying malware families». In: *IEEE Transactions on Dependable and Secure Computing* 18.1 (2018), pp. 283–295 (cit. on p. 9).

[14] Jungho Kang, Sejun Jang, Shuyu Li, Young-Sik Jeong, and Yunsick Sung. «Long short-term memory-based malware classification method for information security». In: *Computers & Electrical Engineering* 77 (2019), pp. 366–375 (cit. on pp. 9, 50, 55).

[15] Hamed HaddadPajouh, Ali Dehghantanha, Raouf Khayami, and Kim-Kwang Raymond Choo. «A deep recurrent neural network based approach for internet of things malware threat hunting». In: *Future Generation Computer Systems* 85 (2018), pp. 88–96 (cit. on pp. 9, 50, 55).

[16] Yujie Fan, Mingxuan Ju, Shifu Hou, Yanfang Ye, Wenqiang Wan, Kui Wang, Yinming Mei, and Qi Xiong. «Heterogeneous temporal graph transformer: An intelligent system for evolving android malware detection». In: *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 2021, pp. 2831–2839 (cit. on p. 9).

[17] Abir Rahali and Moulay A Akhloufi. «MalBERT: Using transformers for cybersecurity and malicious software detection». In: *arXiv preprint arXiv:2103.03806* (2021) (cit. on p. 9).

[18] John McCarthy, Marvin L Minsky, Nathaniel Rochester, and Claude E Shannon. «A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955». In: *AI magazine* 27.4 (2006), pp. 12–12 (cit. on p. 10).

[19] Warren S McCulloch and Walter Pitts. «A logical calculus of the ideas immanent in nervous activity». In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133 (cit. on p. 12).

[20] Frank Rosenblatt. «The perceptron: a probabilistic model for information storage and organization in the brain.» In: *Psychological review* 65.6 (1958), p. 386 (cit. on p. 12).

[21] Diederik P Kingma and Jimmy Ba. «Adam: A method for stochastic optimization». In: *arXiv preprint arXiv:1412.6980* (2014) (cit. on p. 16).

[22] Paul Werbos. «Beyond regression: New tools for prediction and analysis in the behavioral sciences». In: *PhD thesis, Committee on Applied Mathematics, Harvard University, Cambridge, MA* (1974) (cit. on p. 17).

[23] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. «Learning representations by back-propagating errors». In: *nature* 323.6088 (1986), pp. 533–536 (cit. on p. 17).

[24] Paul J Werbos. «Applications of advances in nonlinear sensitivity analysis». In: *System Modeling and Optimization: Proceedings of the 10th IFIP Conference New York City, USA, August 31–September 4, 1981.* Springer. 2005, pp. 762–770 (cit. on p. 20).

[25] Sepp Hochreiter and Jürgen Schmidhuber. «Long short-term memory». In: *Neural computation* 9.8 (1997), pp. 1735–1780 (cit. on p. 21).

[26] Alex Graves and Jürgen Schmidhuber. «Framewise phoneme classification with bidirectional LSTM and other neural network architectures». In: *Neural Networks* 18.5 (2005). IJCNN 2005, pp. 602–610. ISSN: 0893-6080. DOI: `https://doi.org/10.1016/j.neunet.2005.06.042`. URL: `https://www.sciencedirect.com/science/article/pii/S0893608005001206` (cit. on p. 23).

[27] Mike Schuster and Kuldip K Paliwal. «Bidirectional recurrent neural networks». In: *IEEE transactions on Signal Processing* 45.11 (1997), pp. 2673–2681 (cit. on p. 23).

[28] John Firth. «A synopsis of linguistic theory, 1930-1955». In: *Studies in linguistic analysis* (1957), pp. 10–32 (cit. on p. 25).

[29] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. «Bag of tricks for efficient text classification». In: *arXiv preprint arXiv:1607.01759* (2016) (cit. on p. 26).

[30] Kumar Ravi and Vadlamani Ravi. «A survey on opinion mining and sentiment analysis: tasks, approaches and applications». In: *Knowledge-based systems* 89 (2015), pp. 14–46 (cit. on p. 26).

[31] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. «Neural machine translation by jointly learning to align and translate». In: *arXiv preprint arXiv:1409.0473* (2014) (cit. on p. 26).

[32] Christopher D Manning. *An introduction to information retrieval.* Cambridge university press, 2009 (cit. on p. 26).

[33] Tomas Mikolov and Google Research. *Word2Vec: Efficient Estimation of Word Representations in Vector Space.* `https://code.google.com/archive/p/word2vec/`. Accessed: August 2023 (cit. on p. 26).

[34] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. «Efficient estimation of word representations in vector space». In: *arXiv preprint arXiv:1301.3781* (2013) (cit. on p. 26).

[35] Bernd Bischl et al. «Hyperparameter optimization: Foundations, algorithms, best practices, and open challenges». In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 13.2 (2023), e1484 (cit. on p. 31).

[36] Kevin Jamieson and Ameet Talwalkar. «Non-stochastic best arm identification and hyperparameter optimization». In: *Artificial intelligence and statistics.* PMLR. 2016, pp. 240–248 (cit. on p. 32).

[37] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. «Hyperband: A novel bandit-based approach to hyperparameter optimization». In: *The journal of machine learning research* 18.1 (2017), pp. 6765–6816 (cit. on p. 33).

[38] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual, Volume 2: System Programming.* Section 2.5. 2015. URL: `https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf` (cit. on p. 35).

[39] *Debian Popularity Contest.* `https://popcon.debian.org/`. [Online; accessed 30-Jan-2023] (cit. on p. 41).

[40] *VirusShare.* `https://virusshare.com`. [Online; accessed 5-Feb-2023] (cit. on p. 43).

[41] Radim Řehřek and Petr Sojka. «Software framework for topic modelling with large corpora». In: (2010) (cit. on p. 50).

[42] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/` (cit. on pp. 51, 59).

[43] Derek G. Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. *tf.data: A Machine Learning Data Processing Framework.* 2021. arXiv: `2101.12127 [cs.LG]` (cit. on p. 52).

[44] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. «Optuna: A next-generation hyperparameter optimization framework». In: *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining.* 2019, pp. 2623–2631 (cit. on pp. 57, 59).

[45] Universitat Politècnica de Catalunya. *SERT Cluster Wiki of UPC.* 2023-11-24. 2023. URL: `https://www.ac.upc.edu/app/wiki/serveis-tic/Clusters/Sert/FuncionamentGeneral` (cit. on p. 58).

[46] Python Core Team. *Python: A dynamic, open source programming language.* Python version 3.8.18. Python Software Foundation. 2019. URL: `https://www.python.org/` (cit. on p. 59).

[47] F. Pedregosa et al. «Scikit-learn: Machine Learning in Python». In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on p. 59).

[48] Radim Řehůřek and Petr Sojka. «Software Framework for Topic Modelling with Large Corpora». English. In: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks.* `http://is.muni.cz/publication/884893/en`. Valletta, Malta: ELRA, May 2010, pp. 45–50 (cit. on p. 59).

[49] The pandas development team. *pandas-dev/pandas: Pandas.* Version v2.0.3. June 2023. DOI: `10.5281/zenodo.8092754`. URL: `https://doi.org/10.5281/zenodo.8092754` (cit. on p. 60).

[50] Thomas A Caswell et al. *matplotlib/matplotlib: REL: v3.7.3.* Version v3.7.3. Sept. 2023. DOI: `10.5281/zenodo.8336761`. URL: `https://doi.org/10.5281/zenodo.8336761` (cit. on p. 60).

[51] *Joblib: Tools to provide lightweight pipelining in Python.* Version 1.3.2. URL: `https://joblib.readthedocs.io/` (cit. on p. 60).

[52] Andy B. Yoo, Morris A. Jette, and Mark Grondona. «SLURM: Simple Linux Utility for Resource Management». In: *Job Scheduling Strategies for Parallel Processing.* Ed. by Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. ISBN: 978-3-540-39727-4 (cit. on p. 60).