Research on Enhancing the Reliability of Neural Network-Based Systems using Testing and Verification

Dissertation Submitted in Partial Fulfillment for
the Degree of Doctor of Informatics and Data Science

LIU HAIYI

Under the supervision of
Professor Shaoying Liu

High-Quality Software Engineering Group,
Dependable Systems Laboratory,
Department of Informatics and Data Science,
Graduate School of Advanced Science and Engineering,
Hiroshima University, Higashi-Hiroshima, Japan

December 2023

Abstract

This dissertation discusses how to use testing and verification methods to enhance the reliability of systems incorporating neural networks. Specifically, this dissertation aims to investigate: 1) How to employ Testing-based methods to identify potential errors that may arise during the training process of neural networks. 2) How to combine testing and verification methods to improve the reliability of trained neural network models. 3) How to use a combination of testing and verification to explore the interpretability of trained neural networks. Particularly, three approaches are proposed to answer the above three questions. We will introduce them successively.

A Testing-based method to assess the GPU-memory consumption. During the training process of neural network models, a large amount of GPU computing resources is required, but it is difficult for developers to accurately calculate the GPU resources that the model may consume before running, which brings great inconvenience to the development of neural network-based systems. This is particularly important especially in today's cloud-based model training. Therefore, it is very important to estimate the GPU memory resources that the neural network model may use in a certain computing framework. Existing work has focused on static analysis methods to assess GPU memory consumption, highly coupled with the framework, and lack of research on low-coupled GPU memory consumption of the framework. In this article, we propose the Testing-Based Estimation Method (TBEM), which is a Testing-based method for estimating the memory usage of the neural network model. First, TBEM generates enough neural network models using an orthogonal array testing strategy and a classical neural network design pattern. Then, TBEM generates neural network model tested in a real environment to obtain the real-time GPU memory usage values corresponding to the model. After obtaining the data of different models and corresponding GPU usage values, the data is analyzed by regression.

A method utilizing Testing-Based Formal Verification for simplifying and verifying neural networks. Although the security of neural networks can be enhanced by verification, verifying neural networks is an NP-hard problem, making the application of verification algorithms to large scale neural networks a challenging task. For this reason, we propose NNTBFV, a framework that utilizes the principles of Testing-Based Formal Verification (TBFV) to simplify neural networks and verify the simplified networks. Unlike conventional neural network pruning techniques, this approach is based on specifications, with the goal of deriving approximate execution paths under given preconditions. To mitigate the potential issue of unverifiable conditions due to overly broad preconditions, we also propose a precondition partition method. Empirical evidence shows that as the range of preconditions narrows, the size of the execution paths also reduces accordingly. The execution path generated by NNTBFV is still a neural network, so it can be verified by verification tools. In response to the results from the verification tool, we provide a theoretical method for analysis.

We evaluate the effectiveness of NNTBFV on the ACAS Xu model project, choosing Verification-based and Random-based neural network simplification algorithms as the baselines for NNTBFV. Experiment results show that NNTBFV can effectively approximate the baseline in terms of simplification capability, and it surpasses the efficiency of the Random-based method.

An approach to provide localized interpretation of neural networks using the principle of Testing-Based Formal Verification. Although neural networks have been widely used in many fields such as NLP (natural language processing), image processing and even MMML (Multi-modal Machine Learning), their weak interpretability and poor reliability have been criticized by many users for a long time. Specifically, there are two aspects. One is that neural networks are difficult to be verified. The reason is that the architecture of neural networks is based on experience, and the parameter are constructed by back-propagation of the training data. It is difficult to give a formal specification like traditional software, and the verification of neural networks is an NP-hard problem, which makes it difficult to achieve complete verification of the large models. The other is that neural networks are difficult to be explained, that is, it is difficult for us to figure out what features the result of neural network reasoning is based on. For instance, although a neural network correctly recognizes the cat in the picture, we can not determine whether the neural network correctly recognizes the cat through its features or the watermark in the

picture.

These three aspects of research all revolve around the theme of enhancing the reliability of neural network-based systems. They focus on improving the reliability of neural networks during the training process, the reliability of trained neural networks, and their interpretability, respectively. Additionally, applying the theory of TBFV to neural networks also provides theoretical support for generating formal local interpretations of neural networks in the future.

Acknowledgements

I am a clumsy yet curious person about the world. During my undergraduate and master's studies, I learned a lot about mathematics. However, since I didn't know where this knowledge would be applied, I always felt confused about it and was curious about the knowledge itself—where it comes from and where it can be applied. Professor Shaoying Liu was my supervisor during my doctoral studies, and I would like to start by thanking him for his careful guidance. Whenever my thoughts were chaotic, he could always help me clarify my research ideas. The research topic he formulated for me was very forward-looking. It not only helped me comprehend the idea of 'inferring the greater from the lesser' through this subject, but it also fulfilled my curiosity about 'knowledge' itself. This approach of extrapolating from minor elements to broader perspectives has been immensely insightful for me. Secondly, I would like to thank Professor Hiroyuki Okamura for his insightful comments on my work, which inspired me to improve my current work. I would like to thank Hiroshima University for providing me with scholarship support and research funding, which is crucial for a student from a poor background, and without which my life in Japan would have been difficult. Finally, I would like to thank my family for their understanding, tolerance, and encouragement during my PhD program. They have been a solid support for me and a source of motivation for me to work hard.

Contents

Chapter 1

Introduction

1.1 Background

The development of artificial intelligence (AI) algorithms in recent years has far exceeded people's expectations and imagination, showing no signs of slowing down. Among these AI algorithms, neural networks are considered the quintessence of the AI field due to their exceptional performance, and their application is expanding into increasingly diverse areas. In the field of machine vision, neural networks have been pivotal in research branches like image classification, object detection, image generation, and video analysis. These algorithms have been progressively implemented in safety-critical systems such as facial recognition and autonomous driving. In the field of natural language processing, neural networks are also widely used in machine translation, speech recognition, and even in Question and Answer systems (Q&A systems) [1]. Although neural networks have made a big splash in the field of AI such as machine vision and natural language processing, the accompanying reliability problems have also intensified. For example, in the field of automated driving [46, 88], automated driving systems sometimes misjudge traffic signs, route planning errors, and even traffic behavior predicting lost objects. In Q&A systems, there are Jailbreak operations that make the Q&A system say answers that jeopardize the society. Such a series of problems will greatly interfere with the normal use of AI products, lose trust in AI products, and in serious cases, even bring personal safety hazards to users. The root of these problems comes from the fact that the construction process of neural network model is experience-driven (hyper-parameter setting), data-dependent (relying on the quality of training data) and no fixed rules to guide the network structure and other factors, resulting in neural network algorithms generally have adversarial example, formal verification difficulties, poor explanatory and other problems.

In addition, compared to small-scale models, large-scale neural networks [15] show better performance in a variety of tasks such as classification, prediction, and data generation, which has led to the increasing size of today's neural networks, with the current most complex natural language Large Language Model (LLM) even having 175 billion parameters [10]. However, while large-scale models bring stronger learning ability and better generalization ability, they also make it more and more difficult to guarantee the reliability of the models. Therefore, how to improve the construction quality of neural networks as well as how to improve the reliability and interpretability of neural networks becomes an urgent problem.

1.2 Motivation

Testing and verification are two important means to ensure the quality of traditional software. Typically, in the design and development of traditional software, programmers implement specific instructions and rules by writing code. After testing and verifying these instructions and rules, the software is released to the public. Due to the clear logic and rules of traditional software, testing and verification can effectively ensure its reliability. In contrast, the development process of systems based on neural networks is largely driven by experience and data, with rules and their interpretability not as clear as in traditional software. This makes the application of traditional software testing and verification methods to ensure the quality of neural network-based systems somewhat inappropriate. Our total motivation is to alleviate the situation where software testing and verification methods are not applicable in neural network-based systems and to enhance the reliability and stability of neural network-based systems. In this article, we focus on two types of issues, both of which arise from the characteristics of neural networks. However, the difference is that the first type of issue may arise **during the model training process**, specifically how to avoid GPU Out of Memory (OOM) errors during neural network training. This issue can be extended to the question of “how to assess GPU memory consumption before training neural networks.” The second type of issue may occur **after model training**. This specific issue is addressed by exploring how to accelerate the formal verification of neural networks, and how testing and verification can provide explanations for neural network behaviors. The motivation for each of these two problems will be described below.

The first issue: Neural network-based software systems often need to call a series of computing frameworks during operation. Especially when the invoked framework contains closed source software. GPU usage is a typical scenario involving calls across multiple frameworks. Utilizing GPUs to accelerate the construction and inference of neural networks requires the use of multiple components within a system, among which closed-source projects like CUDA are powerful tools for accelerating computations. However, improper use of GPUs can lead to issues such as GPU OOM, incorrect data parallelism on GPUs, and calling unsupported operations on CUDA tensors, among other errors [32]. Of these, GPU OOM is one of the most significant topics. To avoid GPU OOM issues during the training of neural networks, it is essential to estimate as accurately as possible the GPU memory consumption of the network before training begins. This not only prevents errors during training and speeds up the construction of neural networks, but also aids in making informed decisions regarding hardware and software purchases for systems based on neural networks. Typically, the training of neural networks is carried out either through cloud services or by building hardware systems, both of which involve considerations of computational resource procurement. Estimating GPU memory usage can help in reducing the over-purchase of computational power.

The second issue: For pre-trained neural networks or trained neural networks, how to ensure their reliability is an important research direction. Since neural networks are experience-driven and data-driven, although they have good results on unknown data, they also bring problems such as difficult to test [65] and verification [44], and poor interpretability [3,77]. These problems are intensified with the arrival of large models. Testing of neural networks is a powerful tool to ensure their reliability. The approach of generating adversarial examples [104] and performing adversarial training is a typical test-based way to ensure the reliability of neural networks. However, the test-based approach does not formally guarantee that the adversarial example are not included in a given input interval. Neural network formal verification algorithms can overcome the shortcomings of test-based approaches to some extent by guaranteeing that the neural network is reliable within a given formal specification. Unfortunately, the neural network verification problem itself is an NP-hard problem, so neural network formal verification algorithms are mostly used on top of smaller models. In addition, the poor interpretability of neural networks leads to the fact that the formal specification of neural networks is difficult to give in some cases. In the scenario of large-scale neural networks, the interpretability of the model also affects the user's trust in the model. To alleviate the above problems, we try the theory of testing-based formal verification applied to neural networks to accelerate the verification and interpretability of neural networks.

1.3 Current Challenges

Based on the categorization in section 1.2, we also divided the current challenge into two parts: how to ensure the reliability of model training and how to ensure the reliability of the trained model.

In terms of ensuring the reliability of the model during training, we mainly focus on how to evaluate the GPU usage before model training. Thereby preventing GPU OOM during model training caused by unreasonable hyperparameter settings. Current research results are mostly based on static analysis, which analyzes the GPU usage of neural network frameworks in computing computation graphs to estimate the GPU usage of neural network models during training. This idea is usually used in the field of GPU memory optimization. Although it can effectively estimate the GPU memory usage, the method relies heavily on the static analysis of the neural network computation framework. When there is an upgrade or memory optimization of the neural network framework, there is still a need for an expert to re-analyze the framework. Moreover, some neural network frameworks invoke tools with closed-source software, which makes it even more difficult for experts to analyze the framework.

In terms of ensuring the reliability of the trained model, We have focused on how the principles of testing-based formal verification can be utilized to accelerate the verification of neural networks as well as to give explanations of neural network models. While neural network models are getting larger and larger, their verification is becoming more and more difficult. Since the neural network verification problem is an NP-hard problem, the acceleration of hardware can alleviate the verification bottleneck of the verification algorithm, but it still has limitations. Therefore, it is imperative to propose algorithms to accelerate the verification of neural networks. While debugging a model, understanding its behavior is crucial. Current research on the interpretability of neural networks is typically based on heuristic algorithms. Formalizing local explanations for neural network models is a research topic that needs to be explored.

1.4 Summary of the Research

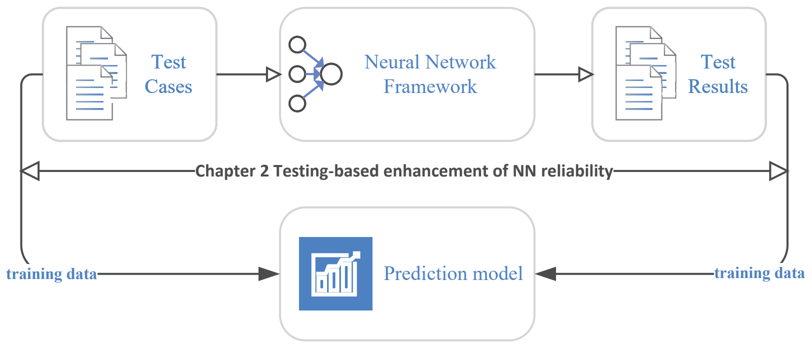


Figure 1.1: Overview of major work on the first issue

This doctoral dissertation has investigated how to use testing and verification methods in software engineering to alleviate the reliability problem of neural networks. Three main areas of work have been accomplished to address the two aspects mentioned above. They correspond to Chapters 2 to 4 of this paper, respectively.

In chapter 2, for the first issue in section 1.2, we propose a test-based approach to evaluate the GPU storage space that the model may use before the model is trained, referred to as TBEM. since the core idea of TBEM is the idea of black-box testing. This allows the approach to avoid static analysis that is completely program-dependent, alleviating the first challenge in section 1.3. We propose to construct a prediction model to help developers predict GPU memory utilization. The inputs of the prediction model are the parameters and hyperparameters of the neural network to be evaluated, such as the number of neural network layers, the number of neurons in each layer, the batch size, etc. The output is the GPU memory utilization of the neural network model to be evaluated. The output is the GPU memory utilization of the neural network model to be evaluated. The training data of the predictive model is obtained by testing method. Fig.1.1 summarizes the core idea of TBEM.

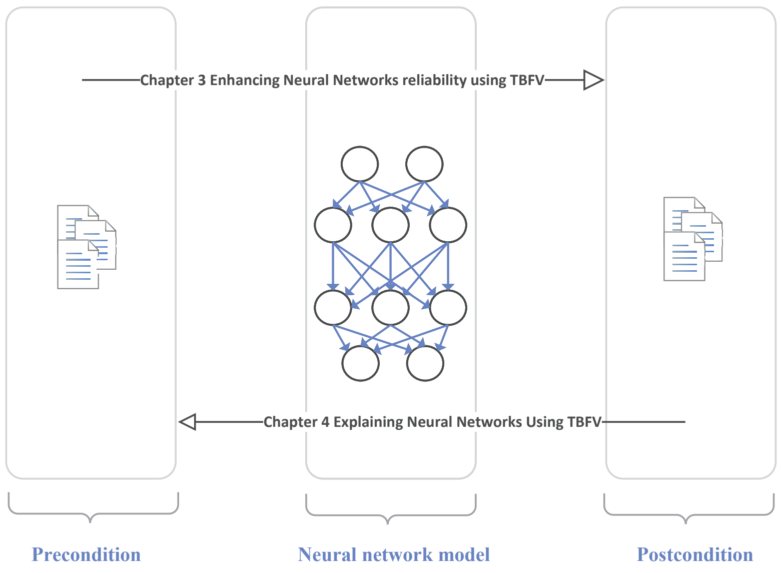

Chapter 3 Enhancing Neural Networks reliability using TBFV
Chapter 4 Explaining Neural Networks Using TBFV
Precondition
Neural network model
Postcondition

Figure 1.2: Overview of major work on the second issue

In chapter 3 , regarding the second research gap described in section 1.2, We propose NNTBFV, a method to simplify and verify neural networks using the TBFV. NNTBFV attempts to combine the methodology of testing with the methodology of verification to utilize testing to accelerate the verification of neural networks, thus alleviating the computational challenges associated with the verification of neural network models. First, the method generates test cases that meet the precondition in the formal specification, then, the neural network executes all the test cases and performs neural network pruning. Finally, the pruned neural network is verified. The core idea of this work is to accelerate the verification of neural network models by reducing the size of neurons. In addition, a theoretical method DeepTBFV is introduced in Chapter 4, which is used to give an explanation of the neural network and thus enhance the reliability of the neural network. Unlike NNTBFV, DeepTBFV utilizes the principle of testing-based formal specification to derive the precondition of the neural network from the post-condition.If the neural network does not have an artificially given precondition, then we take the result derived from the post-condition as the local explanation of the neural network model. post-condition as the local interpretation of the neural network model. Fig.1.2 intuitively shows the difference between the two efforts of NNTBFV and DeepTBFV.

Chapter 2

A Testing-Based Approach to Enhance the Neural Network Reliability of the Training

2.1 Introduction

2.1.1 Background and Motivation

In recent years, with the improvement of computer performance and the continuous accumulation of data, the research and engineering implementation of artificial intelligence algorithms have made rapid progress. Among them, deep learning module is the most applied and implemented system in artificial intelligence system. It is widely used in many scenes, such as image recognition, speech recognition, recommendation system and so on. Although the accuracy and breadth of artificial intelligence system are improving year by year, the hardware cost and time cost of constructing neural network system are also increasing year by year. In 2020, the Gpt-3 model [19] published by Open Artificial Intelligence has 175 billion parameters, and the cost of network training is as high as 12 million US dollars. The high cost of model training is a common phenomenon of the neural network system. Facing such a high cost of model training, how to estimate the amount of memory that a deep learning model will occupy and ensure that the model does not out of memory during the training phase has become an important issue. This error is caused by the fact that developers cannot accurately estimate the size of the video memory occupied by the model before the model runs, so they cannot find the upper and lower limits of the super parameters suitable for their own development environment. According to relevant research literature, among all program failures of deep learning jobs, out of memory(OOM) account for 9.1% (including GPU and CPU) [106], and often occur in training process [32], which makes all the previous efforts of ongoing model training wasted. This not only wastes GPU computing resources, but also affects the development progress of engineers. Therefore, the memory consumption of different deep learning models and various deep learning libraries becomes particularly important. In terms of deep learning model and memory consumption, many researchers have made great contributions and provided corresponding solutions from different angles. The main methods include memory exchange, memory sharing, recalculation, and compressed neural network, etc. these methods reduce the use of memory in the training process of deep learning model by analyzing the calculation graph model and using the technologies such as liveness analysis in static analysis or dynamic memory sharing and memory exchange. But their technology is usually used to make the built model input a larger batch size in the current hardware environment. Not to evaluate that the built model will cause memory overflow in a certain environment before model training.

2.1.2 Challenges and Proposed Methods

In terms of deep learning framework and memory consumption, Gao et al. [23] proposed the method of using static analysis and calculation diagram and resident buffer to predict the memory utilization before model training.

Although the above methods have made effective solutions, there are still the following problems in the memory consumption evaluation of deep learning model:

- Deep learning library (e.g., TensorFlow, Pytoch) [28] generally contains two main functions, automatic differentiation, and GPU acceleration. Automatic differentiation is usually implemented by deep learning library, while GPU accelerated process is usually implemented by calling multiple NVIDIA components(e.g., CUDA, cudnn), it is difficult to achieve static analysis for the cooperative calls of multiple non-open-source components. Because the components called by the framework are in the closed source state, users cannot carry out common memory analysis methods such as context analysis. It also makes the deep learning library a black box for users.

- Each framework of deep learning is iterating rapidly in months, and new deep learning frameworks emerge one after another. The method of static analysis requires experts to analyze the framework. Therefore, the static analysis method undoubtedly increases the labor cost and time cost of evaluating the memory consumption of the deep learning model [70].

  To solve the above problems, this paper proposes a method based on the combination of static analysis and dynamic test modeling analysis [40] [99]. Firstly, using the method of static analysis, the calculation graph of neural network is statically analyzed to pre-estimate the memory that may be consumed by the model. Then the pre-estimated model is run in the deep learning framework to obtain the real value of the model under the framework. Finally, Polynomial regression [93] is used to analyze the gap between the memory consumption estimated by static analysis and that of the real model, to deduce the possible memory consumption of the deep learning framework under different models.

2.2 Priliminary

In this section, we introduce three preliminaries used in TBEM, which are orthogonal array testing strategy, regression algorithm, and formal specification. The relationship between the above concepts and TBEM will be discussed in the overview section.

2.2.1 Orthogonal array

Orthogonal array generation method(OAGM), also known as Taguchi method, is a technology to generate orthogonal array(OA). The shape of the test case table depends on the number of factors and levels in the test [40].

Definition 1. *An Orthogonal array can be defined as $OA(n, f, l, s)$, where:*

- *n is the number of rows of an orthogonal array. In an orthogonal array, n is known as $runs$.*
- *f indicates how many parameters (factors) need to be tested. In an orthogonal array, f is known as $factors$.*
- *l represents the value range of each parameter. In an orthogonal array, l is known as $levels$.*
- *s represents the strength of the orthogonal array. Let the orthogonal array be an $n \times f$ matrix A. In any $n \times s$ sub-matrix in A, There are $w = l \cdot d$ possible d-tuple rows, each of which appears the same number of times.*

In Definition 1, $factors$ correspond to hyperparameters in the deep learning framework that we need to test. $Levels$ represent the range within which hyperparameters can be set. The number of $runs$ is usually determined by $strength$. The relationship between specific parameters is as follows

- Orthogonal arrays are usually written as the following pattern: $L_{runs}(levels^{factors})$
- The value of $runs$ is equal to $levels^{strength}$ when the levels of each factor are equal.
- When the number of $levels$ in each $factor$ of the orthogonal array is different, runs is equal to the product of the number of $levels$ in the last $strength$ column of the orthogonal array.

2.2.2 Polynomial regression

Definition 2. *we have a polynomial equation of degree n represented as:*

$$y_i = \delta_0 + \delta_1 x_i + \delta_2 x_i^2 + \cdots + \delta_n x_i^n + \epsilon \; (i = 1, 2, \ldots, m) \tag{2.1}$$

can be expressed in matrix form in terms of a design matrix X, a response vector $\vec{y}$, a parameter vector $\vec{\delta}$ and a vector $\vec{\epsilon}$ of random errors. The i-th row of X and $\vec{y}$ will contain the x and y value for the i-th data sample, then the model can be written as a system of linear equations:

$$\vec{y} = X\vec{\delta} + \vec{\epsilon} \tag{2.2}$$

where $\vec{y} = [y_1, y_2, \cdots y_n]$ and the vector of estimated polynomial regression coefficients is:

$$\vec{\delta} = (X^T X)^{-1} X^T \vec{y} \tag{2.3}$$

assuming $m < n$ which is required for the matrix to be invertible, then since X is a Vandermonde matrix, the invertibility condition is guaranteed to hold if all the x_i values are distinct. This is the unique least-squares solution.

2.2.3 Formal specification

In order to select test cases generated by TBEM, ensure that the final test cases generated can be recognized by the neural network framework. We need to investigate and summarize some neural network formal specifications. SOFL (Structured Object-Oriented Formal Language) as one of the Formal Engineering Methods for industrial software development [50]. In this paper, we use SOFL to write the formal specification of neural network. The reason is that the formal specification written by SOFL is easier for developers to understand and implement than the written by formal methods. In SOFL, the operation of

```
Input = set of nat = [W1,H1,D1]
Output = set of nat = [W2,H2,D2]
Hyperparameter = set of nat = [K,F,S,P]

process ShapeVerify(Input,Output,Hyperparameter:set) y:bool
pre forall[x:set] | x>0
post W2=(W1-F+2P)/S+1 and H2=(H1-F+2P)/S+1 and D2=K and
    y = true or W2<>(W1-F+2P)/S+1 or H2<>(H1-F+2P)/S+1 or
    D2<>K and y = false
end_process
```

Listing 2.1: *A formal specification of neural networks using SOFL*

filtering test cases that do not conform to the specification can be represented by process, where **process** and **end_process** is a pair of keywords used to mark the beginning and end of the process. **pre** is a keyword indicating the start of the precondition of the process, and the keyword **post** indicates the start of the postcondition. Record a process as S, and then record pre-condition and post-condition as S_{pre} and S_{post} respectively. If the input variable of a process S meets S_{pre}, according to the specification, the output variable defined based on the input variable must meet S_{post} after S. Then we can have the following definitions:

Definition 3. *Let* $S_{post} \equiv (C_1 \wedge D_1) \vee (C_2 \wedge D_2) \cdots \vee (C_n \wedge D_n)$ *where each* $C_i (i \in \{1, \ldots, n\})$ *is a predicate called a guard condition that contains no output variable and each* $D_i (i \in \{1, \ldots, n\})$ *is another predicate called defining condition that defines the output variables.*

Listing 2.1 shows a formal specification for convolutional neural network code using SOFL. $ShapeVerify$ is a process that used to verify the dimensional relationship between tensors in neural network. It has pre-condition $S_{pre} := true$ and post-condition $S_{post} := ((W2 = (W1 - F + 2P)/S + 1) \wedge (H2 = (H1 - F + 2P)/S + 1) \wedge D2 = K \wedge y = true) \vee ((W2 <> (W1 - F + 2P)/S + 1) \vee (H2 <> (H1 - F + 2P)/S + 1) \vee (D2 <> K) \wedge y = false)$. Then, we can write the post-condition in the form of Definition 3:

$G_1 := ((W2 = (W1 - F + 2P)/S + 1) \wedge (H2 = (H1 - F + 2P)/S + 1) \wedge D2 = K$

$D_1 := y = true$

$G_2 := ((W2 <> (W1 - F + 2P)/S + 1) \vee (H2 <> (H1 - F + 2P)/S + 1) \vee (D2 <> K))$

$D_1 := y = false$

According to the above definition, we will collect the specifications of different neural network models and express them using SOFL. Finally, the specification described in SOFL is accurately transformed into Python code to remove the test cases that do not meet the specification. The details about the syntax of SOFL can be found in the publication by Liu [50].

2.3 Problem formulation and Overview

In this chapter, we first formulate the problem [49] and the proposed method. Then, an overview of TBEM is given.

2.3.1 Formalization of problem

In order to more clearly describe the problem that we solve and the methods to be proposed. We formalize the deep learning framework and the operation process of graph model in the framework.

- Formalization of model operation process in deep learning framework [79]. Let's define set API as

$$\mathbf{I} = \{A_i\}_{i=1}^{n} = \{A_1, A_2, \ldots, A_n\} \tag{2.4}$$

Where A_i is the existing API in the neural network framework, and n is the number of **I**. At the same time, The set of hyper-parameters to be set for each **I** is defined as

$$\mathbf{HP_{A_i}} = \{p_{A_i}^{j}\}_{j=1}^{n} = \{p_{A_i}^{1}, p_{A_i}^{2}, \ldots, p_{A_i}^{n}\} \tag{2.5}$$

Where $p_{A_i}^{j}$ is the specific hyper-parameter to be set in each **I**. A_i is the element in set API and j is the number of all Hyper-parameters of the **I**. For example,

$$\sum_{i=1}^{n} |HP_{A_i}| \tag{2.6}$$

can represent the types of all settable hyper-parameters in the framework.

- Formal specification for deep learning model in framework [16]. Next, we describe the form of the model in the framework based on the definition of the deep learning framework. Given a set of **I**, We do a finite Cartesian product

$$\overbrace{I \times I \times \cdots \times I}^{K} \tag{2.7}$$

denoted as I^K and

$$I^K = \{< A_1, A_2, \ldots, A_n > | A_i \in I, 1 \leq i \leq K\} \tag{2.8}$$

Then, the model in the deep learning framework can be defined as

$$model \in I^* = \bigcup_{K=1}^{\infty} I^K \tag{2.9}$$

2.3.2 Formalization of method

Because in the DL framework, the model usually runs in the form of calculation graph, so we mark the calculation graph [12] set as G. Let $CG : model \to G$ represents the mapping between the model and the calculation graph, for a given input $m \in model$, there will be a corresponding calculation garaph $g \in G$.

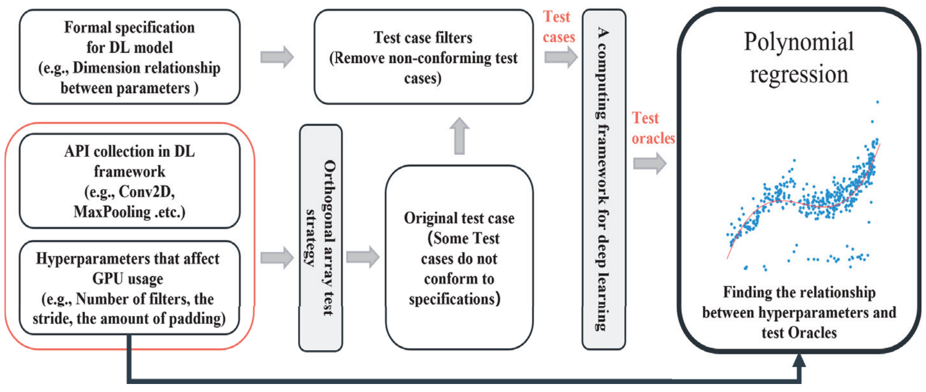

Formal specification for DL model (e.g., Dimension relationship between parameters)
Test case filters (Remove non-conforming test cases)
Test cases
A computing framework for deep learning
Polynomial regression
API collection in DL framework (e.g., Conv2D, MaxPooling .etc.)
Hyperparameters that affect GPU usage (e.g., Number of filters, the stride, the amount of padding)
Orthogonal array test strategy
Original test case (Some Test cases do not conform to specifications)
Test oracles
Finding the relationship between hyperparameters and test Oracles

Figure 2.1: Overview

Meanwhile, let GU be the set of interger means the size of the $GPU-memory$ consumed by the model, for each $m \in model$, a corresponding $GPU-memory$ usage can be obtained by running the model or static analysis for the graph [95]. Let $GV : model \rightarrow GU$ be a function of GU for the model. Through the investigation of previous studies, it can be seen that the static analysis of the calculation graph can roughly estimate the usage of the $GPU-memory$ of the model at run time. Therefore, let $SGV : graph \rightarrow GU$ be a function of GU for the model. There will be a certain gap between the GU obtained by static analysis of the model and the GU obtained by running the model. This gap can be defined as

$$Gap(model) = GV(model) - SGV(CG(model)) \tag{2.10}$$

It is critical to define the relationship between $Gap(model)$ and $model$. Not only can it be used to get a more accurate model $GPU-memory$ usage, but it can also be used to evaluate the execution efficiency of the DL framework [69].

In order to find the specific mathematical form of $Gap(model)$, we propose a data fitting method based on OAGM. First, a certain scale of deep learning model is generated through the orthogonal array test strategy, which is used as a test case to test the GU value (Test Oracle) of the DL framework at runtime. Next, use the regression algorithm to find the relationship between test case and test oracle, that is, to obtain $GV(model)$. However, if we want to know $Gap(model)$, we still need to know the specific value of SGV. This paper adopts the method of static analysis of the computational graph, and evaluates the specific value of $SGV(CG(model))$ [106]through the analysis of the tensor scale. Fig. 2.1 shows an overview of how TBEM works. The process is divided into test phase and data analysis phase. The test phase includes the generation of test cases and the collection of test data. In the data analysis phase, polynomial regression is used to solve the relationship between different hyperparameters of the neural network and GPU usage.

In the test phase, we need to automatically generate test cases. We believe that there are two principles for the generated test cases: (1) The number of generated test cases can be executed in a limited time. For example, suppose we select 10 Super parameters of neural network, and each super parameter has 3 values. The full test of such a neural network model will produce 59049 (3^{10}) test cases. Such a test scale may not be completed in a limited time. (2) Test cases need to be evenly distributed in the test space as much as possible, which can make the conclusion of polynomial regression more accurate.

In order to overcome the above difficulties, we propose a test case generation algorithm based on OAGM. Firstly, the structure of many classical neural network models is defined by string formatting. Then, the OAGM strategy is used to deform the layers of the classical neural network template to produce a sufficient number of neural network structures. Finally, using OAGM again, the super parameters corresponding to the neural network structure are generated, and the initial test cases are obtained.However, the test cases generated in this way can not guarantee that they all meet the requirements of neural network framework. The main problem is that the shape of tensor may be inconsistent. Therefore, we implement a filter to remove the non-conforming test cases, so as to get the final test cases that can be run.

After the test is complete, we can get the corresponding GPU usage for different test cases. Then, how to use the data and solve the mapping relationship between test cases and GPU usage is an important issue. We propose a polynomial regression solution, which abstracts test cases into a hyperparametric vector and establishes a mapping relationship between the hyperparametric vector and the GPU usage. This process enables TBEM to infer the GPU usage of different neural network models. Finally, we make an empirical study on the reasoning ability of TBEM, which proves the validity of TBEM.

2.4 Approach

In this section, we first introduce what OAGM is and how to use OAGM to generate test cases that can test the DL framework, and then analyze the feasibility of test cases generated based on OAGM and the ability of the generated test cases and test oracle to be applied to regression analysis feasibility. Finally, the regression model and static analysis model used in this method are introduced.

2.4.1 Test case generation based OAGM

The purpose of testing is to find out how much $GPU-memory$ is consumed by different models running under a certain framework. But there are many hyperparameters e.g., batch size [67].in the deep learning algorithm, and not all hyperparameter changes will have a huge impact on the $GPU-memory$. The static analysis of the deep learning calculation graph can filter out the APIs that have a greater impact on the memory consumption.

Observatios and motivations about API screening: GPU's advantage lies in parallel computing. In the process of training deep learning models, there are a large number of operators that need to be calculated in parallel. For example, feature mapping in forward propagation, gradient mapping in back propagation, etc. Therefore, we have screened APIs related to convolution operation, pool operation, and Batch Normalization that will generate a large number of parallel calculations. In each API, the input scale and output scale of each layer of neurons can be set, and different parameters correspond to different memory usage. In addition, the depth in deep learning is also a major factor in consuming memory. Therefore, in the test, models with different depths and different structures will be tested orthogonally [48]. Corresponding to the memory consumed by different models. Because the memory consumption of the underlying framework will not decrease with the increase of the influencing parameters in the model, that is, the direction of data change is known, and only the rate of change is unknown. Therefore, the data obtained by the orthogonal test is sufficient for multivariate polynomia regression analysis.

Let's take the VGG network as a example. Suppose that through the static analysis [72] of the neural network model, three representative hyperparameters are selected, namely Batch-size, Depth and Number of convolutional layers [56].

Test Number	Batch size	Depth	Number of convolutional layers
Case1	4	11	8
Case2	8	13	10
Case3	16	16	13
Case4	32	19	16

Table 2.1: Orthogonal test example of VGG network

These three hyperparameters constitute the factors in the orthogonal array. As shown in Table 2.1. In an orthogonal array, the range of values for each factor is called levels. Table 2.1 shows an orthogonal array with a factor of 3 and levels of 4. If a comprehensive experimental method is used for testing, up to 3^4 tests are required. And the number of tests increases exponentially with the value of levels. However, using orthogonal experiments to generate orthogonal arrays requires only 4^2 tests. In other words, when the levels become very large, a comprehensive test is impossible. Therefore, this article uses the OAGM to generate test cases [14]. Testing the deep learning framework and record the test oracle corresponding to each test case.

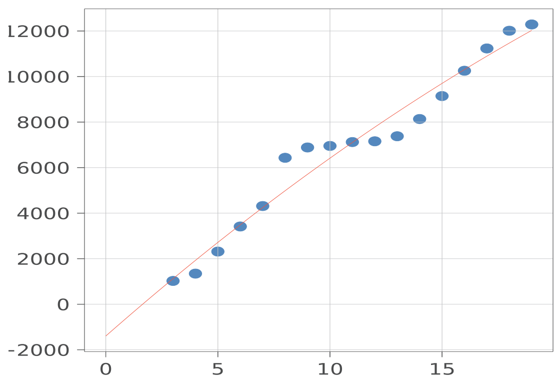


Figure 2.2: Relationship between the number of neural network layers and GPU utilization

2.4.2 Polynomial regression

we got a lot of pairs of test case and test Oracle, where test case is marked as *model* and test Oracle is marked as GU. Because the model is generated by transforming parameters. therefore, *model* can be denoted as $model(hyperparameter_1, \ldots, hyperparameter_n)$ and The hyperparameters are derived from the static analysis calculation graph.

Next, we use polynomial regression to find the relationship between hyperparameters and GU, which is equivalent to finding a way to solve $GV(model)$. Furthermore, $SGV(model)$ is known. We have also found a way to solve $Gap(model)$.

2.5 Case Study

```python
from tensorflow import keras
from tensorflow.keras import layers, models, Input
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Conv2D, MaxPooling2D,
    Dense, Flatten, Dropout

def VGG16(classes, input_shape):
    input = Input(shape=input_shape)
    # 1st block
    x = Conv2D(64, 3, activation='relu')(input)
    x = Conv2D(64, 3, activation='relu')(x)
    x = MaxPooling2D(2, strides=(2,2))(x)
    # 2nd block
    x = Conv2D(128, 3, activation='relu')(x)
    x = Conv2D(128, 3, activation='relu')(x)
    x = MaxPooling2D(2, strides=(2,2))(x)
    # 3rd block
    ...
    # full connection
    x = Flatten()(x)
    x = Dense(4096,activation='relu')(x)
    x = Dense(4096, activation='relu')(x)
    output_tensor = Dense(classes, activation='softmax')(x)

    model = Model(input, output)
    return model
```

Listing 2.2: VGG 16 model

To clarify the role of regression in this algorithm, we give an example of univariate polynomial regression in this subsection. From the method of static analysis, the number of layers of deep learning is directly related to the consumption of computing resources. The function of univariate polynomial regression is to analyze the tested data and get the mathematical expression of the relationship between the number of deep learning layers and the computing resources.Finally, it is worthwhile to estimate the display memory consumption of various neural networks by using the regression results.

Visual Geometry Group(VGG) [83] is a classical neural network model. Listing 2.2 shows the implementation of the VGG16 model in the TensorFlow framework. In order to test the memory usage of training in the TensorFlow framework for in-depth learning models of different layers, and to ensure that the model is true and effective as possible.We mutated the VGG16 model.Because case study is the reason, the convolution layer parameters (filters, kernel size, padding, strides operations, etc.) and pool layer parameters (pool size, strides, padding, data format, etc.) appearing in the model are set as uniform parameters when mutating VGG16, excluding the effect of Hyperparameters other than the number of layers on explicit memory consumption in the deep learning model above. Fig. 2.2 shows the relationship between the number of neural network layers and GPU usage (MB).

2.6 Related Work

2.6.1 GPU-memory estimation

So far, most of the research on memory management of DL accelerator-GPU focuses on how to optimize the use of GPU memory during model training. For example, Rhu et al. [73] proposed vDNN to formulate a memory swapping strategy between main memory and GPU memory by analyzing the computation graph, to reduce the footprint of GPU memory in the process of training. Gradient checkpoint [12] uses the idea of recomputation to implement an algorithm for training n layer network, which only consumes $\mathcal{O}(\sqrt{n})$ memory. SuperNeurons [95] and Capuchin [69] both combine memory sharing, memory swapping and recomputation techniques to varying degrees to further improve the optimization of GPU memory management in DL model training. However, unlike our work, these studies usually focus on how to optimize memory usage during DL model training. Rather than estimating how much GPU memory may be consumed by the model itself when the model is not trained.

DNNMem [23] is the most relevant work with TBEM. TBEM and DNNMem have a common purpose, that is, the GPU memory that may be consumed by the DL model is estimated before the DL model is executed. However, the TBEM method is based on the regression model generated from test data to evaluate the DL model. It is essentially different from DNNMem, which evaluates the GPU memory consumption of the DL model through the static analysis of the computation graph. Different working principles make TBEM overcome the following problems of static analysis methods in the following aspects: 1) the version update speed of each component of deep learning framework is fast, and the update cost of tools developed based on static analysis principle is large; 2) The dependency of deep learning framework is complex, and some components cannot be completely statically analyzed.

2.6.2 Testing of DL framework

In recent years, with the increasing demand for the stability of DL framework, the research on automatic test DL framework has gradually attracted the attention of researchers. Cradle [70] detects the inconsistency between the implementation of the same neural network model in multiple DL frameworks, determines that there may be errors in the inconsistent framework by using the way that the minority obeys the majority, and puts forward the relevant algorithm to locate the wrong location. Gao.et.al. [28] Proposed another DL framework testing method called Audee, which is different from cradle's practice of using existing DNNS as test cases. Audee tried to generate test cases by using search algorithm, and improved the bug type detection range and bug location accuracy.

For algorithms without multiple implementations, that is, when cross referencing cannot be used for Test Oracle comparison, Murphy et al. [60, 61] Tried to test the machine learning framework with the method of geometric relations, but the framework here is not a neural network framework.

The above methods mainly focus on how to detect the bugs in the framework. Although TBEM also needs to generate test cases for the framework to run, on the contrary, TBEM is correct based on the framework. We record the feedback given by the framework to prevent GPU memory overflow due to hyperparameter setting errors in neural network code implementation.

2.7 Summary

In this article, we propose a testing-based method to evaluate the memory usage of the DL framework. This method is different from the previous static analysis method. The possible errors of the static analysis method can be corrected through testing, and the possible GPU-memory usage of the deep learning model can be better evaluated before the deep learning model is running.

At present, it is only a theoretical framework. In the future, this method will be used to automatically generate a large number of test cases to test the mainstream DL framework, so as to prove the effectiveness of this method [45].

Chapter 3

Enhancing neural networks reliability using Testing-Based Formal Verification

3.1 Introduction

3.1.1 Background and Motivation

In recent years, neural networks have been increasingly used in security-critical systems, such as autonomous driving [25], financial payments [31], and even aviation scheduling systems [26]. At the same time, neural networks have also been shown to be potentially vulnerable to elaborate adversarial examples (AEs) [104]. The neural network verification algorithm [9] can use the given formal specification to verify the neural network. Unfortunately, the neural network verification problem has been proved to be NP-hard [36], and large scale neural networks are constantly being released. That makes it a greater challenge to ensure the reliability of neural networks.

Generally, when engineers are faced with complex tasks, they tend to build large scale model, such as more complex structures or more neurons in the model [92]. The reason is that larger scale models are usually more expressive [22]. Recently, large scale neural networks, such as ChatGPT [63], Midjourney [64], etc., have all showcased their strong capabilities in reasoning and generation. While larger models bring benefits, they also create the disadvantage of poor interpretability and difficulty to verification. For instance, when we want to verify the properties of the model, since the verification algorithm is usually NP-hard, many redundant models will inevitably lead to time out. To alleviate the security problems caused by large scale neural networks, in terms of testing, several AEs generation algorithms have been proposed by the security community and are widely used in industry. These methods enhance the reliability of neural networks to some extent [24, 38, 57]. At the same time, some pruning algorithms have tried to scale down the neural network size by means of testing with as little impact on accuracy as possible [27]. After all, smaller neural networks can be better deployed to obtain more efficient forward inference capabilities [8, 30]. However, the above algorithms do not explore how to guarantee the reliability of the neural network under the formal specification. Meanwhile, formal verification algorithms for neural networks are also continuously proposed by the formal verification community. This kind of algorithm has achieved success in small neural networks. To extend the verification algorithm to large neural networks, some studies have proposed using hardware such as GPUs [98] or parallel computing [102] to speed up the verification of neural networks. The efficiency of neural network verification is improved by distributed computing. However, at present, the speed of hardware accelerating the verification of neural network is far less than the expansion of the scale of neural network.

3.1.2 Challenges and Proposed Methods

3.1.2.1 The Challenge of Neural Networks Verification

The development of neural networks verification research has benefited from a large number of traditional software verification techniques, such as constraint solving and reachability analysis [55]. The verification process usually requires Boolean Satisfiability Problem (SAT) Encoding, Linear Programming (LP) Encoding or Mixed-Integer Linear Programming (MILP) Encoding of the neural networks. Although there are a large number of techniques to enhance the coding efficiency, streamline the constraint size, and speed up the process of constraint solving. However, the verification of neural networks is still an NP-hard problem [36]. As the size of neural networks continues to increase, the verification efficiency decreases substantially.

3.1.2.2 The Challenge of Neural Network Pruning

Currently, how to design the structure of deep neural network for different application scenarios is still an unsolved problem. Due to the lack of this design specification, the redundancy of neural network is widespread. Redundancy may improve the accuracy of neural network to a certain extent, but in most cases, it can only increase the training cost and reasoning cost of the network, which becomes particularly obvious in application scenarios such as edge computing, embedded systems. To overcome the redundancy of neural network structure, pruning has become an essential technology for neural network training and deployment. Neural network pruning contains two technical challenges. The first is how to create rules to locate which neurons, or which weights should be pruned. The second is how to ensure that the pruned neural network is equivalent or similar to the original neural network.

Existing neural network pruning algorithms often use a large number of test cases to test the trained neural network, and then prune the neural network according to the rules of neuron activation or weight size. Then, the pruned neural network is trained again. To achieve the same or even better effect on some test data sets. When given the formal specification of the input and output for a neural network model, the ensuing challenge is that the pruning algorithm must generate a model as similar as possible to the original (unpruned) model within the formal specification.

If we want the accuracy of the pruned neural network to be completely consistent with the original network, the execution path of the neural network for each test case in the input space should be known. However, the sample size of the input space to be tested to obtain the execution path is huge. Taking the FCNN as an example, it is assumed that the number of neurons in the input layer is n and the input space of each neuron is m. Then it takes m^n test cases to obtain the neural network execution path corresponding to the input space through the test method. Even if we quantify the neural network input, that is, m is represented by $int8$, m^n tests may still be an impossible task, and worse, n may also become larger with the development of training data (For example, clearer training picture data is used in computer vision).

3.1.2.3 Proposed Methods

TBFV is a test-based approach to obtain the pre-condition of execution paths and verify that paths satisfy the given specification by analyzing the constraints of execution paths. In this paper, we propose a testing-based formal verification method for neural networks, named NNTBFV, which is used to Simplifying and verifying neural networks using the formal specification. NNTBFV consists of two phases: neural networks simplification phase, and neural networks verification phase.

Simplification phase. Unlike the previous method mentioned above, we generate test cases in order to find as many inactive neurons as possible under the formal specification and use structured pruning to remove the inactive neurons. The purpose of this is to reduce the size of the neural network as much as possible under the formal specification and to accelerate the verification of the neural network. Specifically, for a given formal specification S for a neural network which contains a formal definition of the pre-condition PR and the post-condition PO. We are gradually generating a series of input values consistent with specifications to activate more neurons in the neural network. Here, we cannot guarantee the activation of all neurons that should be activated. The gradient ascending algorithm can approach the real activation number of neurons as much as possible. For instance, a fully connected $2 \times 8 \times 8 \times 2$ four layer neural network N with 16 neurons containing ReLU activation function. Assuming that 12 neurons of the neural network N must be activated for S, there are 4 neurons that cannot be activated in any case. Our algorithm is to find out as much as possible which of the 12 neurons can be activated by generating test cases, and prune the neurons that can not be activated structurally. Additionally, to prevent the intervals defined by PR from being too large, the original PR space is partitioned into several sub pre-condition. We constructed a set equivalent to the original PR using the Cartesian product, named PRP. Hence, verifying all elements in PRP means verifying the original pre-condition.

Verification phase. The simplified neural network is formally verified by the neural network verification algorithm. Upon completion of the verification, if the simplified neural network accurately represents its sub-domains without any counterexamples, we ascertain the execution path to be reliable. On the other hand, if counterexamples emerge, we utilize them as training data and subsequently fine-tune the neural network to further enhance its reliability. Furthermore, it's important to note that our method remains unaffected regardless of the underlying verification tool being utilized. Any verification tool can be integrated with our method.

To evaluate the effectiveness of NNTBFV, we developed a PyTorch-based prototype that adheres to a set of formal specifications, enabling the activation of a maximal number of neurons within the neural network. To facilitate the comparison with the baseline methods, we use the ACAS Xu model [36] as our baseline model, with specific details referred to in the section 3.5. Additionally, as the model refined by NNTBFV is built on PyTorch, it effortlessly integrates with any neural network verifiers compatible with the PyTorch output format. Although we have not yet found any work that is completely similar to the purpose of NNTBFV, in order to evaluate the effectiveness of NNTBFV, We adopt the "Pruning and Slicing Neural Networks using Formal Verification" methodology from Lahav et al [39] as our baseline. This is a verification-based method for judging redundant neurons, which we will refer to as the 'verification-based method' in the following sections. The experiment results demonstrate that NNTBFV can effectively approximate the results of verification-based method, and since NNTBFV is a testing-based algorithm, it can be more easily computed in parallel and avoid the time out that may occur in the verification-based approach.

Overall, this paper mainly makes the following contributions:

- We develop the first TBFV for neural networks, named NNTBFV. It can identify a class of redundant neurons whose removal has a little impact on the output of the whole network. The main contribution of this technology is to enhance the scalability of neural network verification by reducing the scale of neural networks under the formal specifications.

- We implemente a prototype to evaluate the effectiveness of NNTBFV. The experimental results show that NNTBFV can activate as many neurons in the neural network as possible within a given formal specification, approximating the baseline and compared to the random-based method. As the granularity of the partition increases, the activation rate of neurons decreases.

- we outline the theoretical methods for verifying neural networks and analyzing verification results based on the pre-condition of the execution path.

The rest of the chapter is organized as follows. In Section 3.2, we provide NNTBFV with the necessary preliminary. The overview and detail of NNTBFV are given in Section 3.3. A case study is shown in Section 3.4. We evaluate the effect of NNTBFV in Section 3.5, followed by a discussion of related work in Section 3.6. Finally, a conclusion and future work are placed in Section 3.7.

3.2 Priliminary

3.2.1 Neural Network

A neural network's structure typically comprises multiple layers of interconnected nodes, often referred to as "neurons". It's worth noting that neurons within the same layer usually lack direct interconnections. Each neuron processes incoming inputs, performs calculations, and yields outputs. These outputs then serve as the inputs for the succeeding layer of neurons. The information flows from the input layer, through one or more hidden layers, and ultimately to the output layer, thereby establishing a sophisticated network for information processing.

In a fully connected neural network (FCNN) with t layers, l_1 represents the input layer of the neural network and l_t represents the output layer of the neural network, then the neural network can be formally represented as a nonlinear function $f : l_1 \rightarrow l_t$, where $l_1 \subseteq \mathbb{R}^m$ and $l_t \subseteq \mathbb{R}^n$, m represents the number of input neurons and n represents the number of output neurons. Except for l_1, the input of each neuron in the neural network can be represented as

$$n^i_j = w^i_{1j} \cdot o^{i-1}_1 + w^i_{2j} \cdot o^{i-1}_2 + \cdots + w^i_{sj} \cdot o^{i-1}_s + b^i_j \tag{3.1}$$

where n^i_j denotes the jth neuron in the l_i, and s denotes the number of neurons in the l_{i-1} layer. o^{i-1}_1 is the output of the 1th neuron in layer l_{i-1} and w^i_{1j} is the weight of o_1^{i-1} to the jth neuron in the l_i. The meaning of o_k^{i-1} and w_{kj}^i is similar to that of o_1^{i-1} and w_{1j}^i, where $k \in 2 \ldots s$. We consider the $FCNN$ with Rectified linear Unit ($ReLU$) as the activation function. The output of each neuron in the neural network can be denoted as :

$$o_j^i = ReLU(n_j^i) = max(n_j^i, 0) \tag{3.2}$$

Furthermore, let the number of neurons in layer i be u. We can express the operation of the $FCNN$ hidden layer in the form of matrix product, as follows,

$$\begin{pmatrix} o_1^i \\ o_2^i \\ \vdots \\ o_u^i \end{pmatrix} = \begin{pmatrix} ReLU(n_1^i) \\ ReLU(n_2^i) \\ \vdots \\ ReLU(n_u^i) \end{pmatrix} = ReLU\left(\begin{pmatrix} w_{11}^i & w_{12}^i & \dots & w_{1s}^i \\ w_{21}^i & w_{22}^i & \dots & w_{2s}^i \\ \vdots & \vdots & \ddots & \vdots \\ w_{u1}^i & w_{u2}^i & \dots & w_{us}^i \end{pmatrix}\begin{pmatrix} o_1^{i-1} \\ o_2^{i-1} \\ \vdots \\ o_s^{i-1} \end{pmatrix} + \begin{pmatrix} b_1^i \\ b_2^i \\ \vdots \\ b_u^i \end{pmatrix}\right) \tag{3.3}$$

FCNN is the basic idea of many neural networks, and the fully connected layer can be converted into convolution layers by a simple transformation [53], and the verification theory of FCNN using ReLU function is most intensively studied because of its functional properties being easy to verify.

3.2.2 Testing-Based Formal Verification

The testing-based formal verification [43, 47] is proposed to ensure the correctness of all traversed program paths in traditional software. The first step of TBFV is to generate a test case (Tc) based on the test condition in the formal specification. The second step is to obtain a traversed program *path* by executing the Tc on program P, where the *path* contains a series of conditions. The third step is to verify the reliability of the *path* under the formal specification by using symbolic execution or Hoare logic [71].

3.2.3 Neuron Coverage

Neuron coverage [68] is the proportion of neurons activated in the neural network to the total number of neurons when the neural network runs a test suite. It is a derivative of the traditional software testing concept of statement coverage in neural networks. Specific definitions are as follows:

$$NC(T, x) = \frac{|\{n | \forall x \in T, ACout(n, x) \geq t\}|}{|N|} \tag{3.4}$$

where the set $N = \{n_1, n_2, \ldots, n_s\}$ represents all neurons in the neural network and $|N| = s$. $T = \{x_1, x_2, \ldots, x_k\}$ represents a test suite for a neuron network and k is the number of test cases. $ACout(n, x_i)$ is the output of neuron n given the input test case x_i. t is the threshold value for determining whether a neuron is activated, usually $t = 0$.

3.2.4 Gradient Ascent

Definition 4. *If $f(x_1, x_2, \ldots, x_n)$ has a partial derivative at point $p(x_1, x_2, \ldots, x_n)$ with respect to all independent variables, then the vector $(f_{x_1}(p), f_{x_2}(p), \ldots, f_{x_n}(p))$ is the gradient of the function f at the point p, denoted as:*

$$\nabla_f(p) = (f_{x_1}(p), f_{x_2}(p), \ldots, f_{x_n}(p)) . \tag{3.5}$$

Since the directional derivative can be expressed as

$$f_l\,(p) = \nabla_f(p) \cdot l = |\nabla_f(p)| \cdot \cos\theta \tag{3.6}$$

where l is the direction vector, θ is the angle between gradient vector and l. If $\theta = 0$, the the direction of the gradient is the same as the direction of l, and the value of $f_l(p)$ is maximum. Then the direction of the gradient of f at point p is the direction in which the value of f grows fastest. In general, gradient ascent is used to solve for the extreme value of a function in a certain space.

3.2.5 Verification of Neural Networks

Deep neural network is a mapping in a high-dimensional space, which can be formally expressed as $f : \mathbb{R}^m \to \mathbb{R}^n$. If there is a set of constraints ϕ which is the pre-condition of f on $\mathbb{R}^m$, and existence a set of constraints φ which is the post condition of f on $\mathbb{R}^n$. Then, the problem of neural network verification is transformed into proving that $\forall x \in \mathbb{R}^m : \phi(x) \to \varphi(f(x))$ is satisfied or not.

3.2.6 Cartesian product

Definition 5. *the Cartesian product of two sets A and B, denoted $A \times B$, is the set of all ordered pairs (a, b) where a is in A and b is in B. In terms of set theory, that is $A \times B = \{(a, b) \mid a \in A \wedge b \in B\}$*

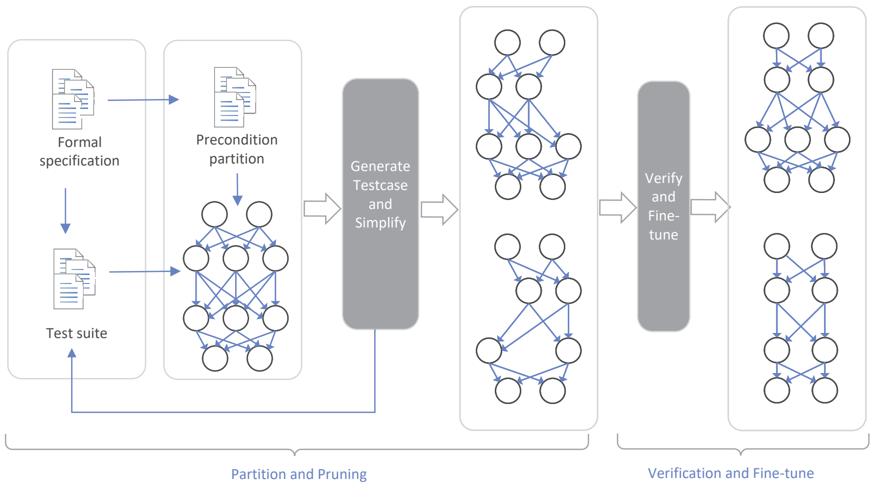


Figure 3.1: An overview of NNTBFV

3.3 Theoretical Method

To solve the above challenges, we have introduced NNTBFV. Section 3.3.1 presents an overview of NNTBFV. There are details about how to use test cases to prune in section 3.3.2. Then, section 3.3.3 describes the process of formal verification and retraining of neural networks in detail.

3.3.1 Overview

The overview of NNTBFV is described in Fig. 3.1. The proposed method includes two phases: simplification of neural networks and verification of neural networks. The neural network studied is trained models in which trainable parameters are fixed.

Simplification phase. The formal specification of the neural network model and the test suite need to be prepared. The test suite can be randomly generated based on formal specification or using existing training data and testing data. When the range of pre-conditions in the formal specification is large, the pre-condition partition can be used to obtain a smaller execution path. All samples in the test suite are fed into the neural network, and each sample generates one execution path. We merge all the execution paths to generate the initial path. The initial path is used as input to NNTBFV to generate the execution path corresponding to each execution path. Next, we can use gradient ascent to activate as many neurons as possible under pre-condition.

Verification phase, NNTBFV verifies that each execution path satisfies the formal specification. If the verification result is unsatisfied, the stability of the execution path on the interval is guaranteed. If it is satisfied, we will look for the counterexample of the execution path and retrain the execution path on the formal specification. In the following chapters, the technical details of each step will be described in detail.

Table 3.1: Definitions of common symbols in this paper.

Symbol	Significance
$PR_i = [\ rl_i,\ ru_i\ ]$	Constraints on the i-th neuron in the input layer of the neural network
$PR = \{PR_1, PR_2, \ldots, PR_n\}$	The set of constraints of all neurons in the input layer of the neural network
$PR^p = \{PR_1^p, PR_2^p, \ldots, PR_n^p\}$	The set of interval partition of each interval D_i
$Ts = \{(x_1, y_1), \ldots, (x_t, y_t)\}$	Initial test suite.
$Ts^I = \{x_1, x_2, \ldots, x_t\}$	A set of inputs in a training dataset.
$Ts^O = \{y_1, y_2, \ldots, y_t\}$	A set of outputs in a training dataset.
PRP	The set PRP is defined as an pre-condition partition

3.3.2 Simplification of Neural Networks

3.3.2.1 Mathematical Formulation of Pre-condition Partition

Here, we need to make an assumption, that is, all the training data sets are consistent with the formal specification. This assumption is reasonable because data that obviously does not conform to the formal specification should not be trained by the neural network. The trained neural network model (NN) obtains the weight from the training data set, and verifies the validity of the weight in the test data set. After that, the execution path of the neural network corresponding to each sample is determined.

To explain the pre-condition partition for neural networks, we first provide a general form of formal specification.

```
process The neural network
pre-condition $PR_1 \wedge PR_2 \cdots \wedge PR_m$
post-condition $PO_1 \wedge PO_2 \cdots \wedge PO_n$
end_process
```

Listing 3.1: *Formal Specification illustration*

In Listing.1, the pre-condition can be represented by the conjunction of all constraints of neurons in the input layer. The post-condition can be represented by the conjunction of constraints on the output layer. Further, we can define the partition of constraint for each neuron.

Definition 6. *partition of neuron constraint*
Let's partition an constraint PR_i into a set of disjoint partitions $PR_i^p = \{PR_i^1, PR_i^2, \ldots, PR_i^s\}$ where each $PR_i^j = [rl_i^j,\ ru_i^j]$ is a sub constraint of PR_i. Then the constraint PR_i can be expressed as $PR_i^1 \vee PR_i^2, \cdots \vee PR_i^s$.

In the input layer of NN, the constraint of each neuron corresponds to the element in PR. For example, if the number of neurons in the input layer is m, then $|PR| = m$. According to definition 6, each constraint in PR can be divided. We divide each element PR_i in PR into s sub constraints, where s is the hyperparameter, which needs to be set artificially. The purpose is to prune the neural network more efficiently. In addition, we can rewrite the formal specification in Listing 3.1 to the form of Listing 3.2 according to definition 6.

Definition 7. *$pre - condition$ partition*
*If the interval partition is performed on each element of the set PR, then the set $PR^p = \{PR_1^p, PR_2^p, \ldots, PR_n^p\}$ can be obtained. We can constructe a set $PRP = PR_1^p \times PR_2^p \cdots \times PR_n^p$ by **Cartesian product**. The set PRP is defined as an pre-condition partition.*

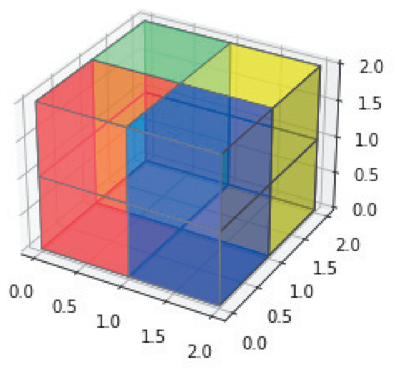
We perform partition using $definition$ 7 on the pre-condition of NN, denote as PRP. From the definition of formal specification, we can get: $\forall x_i \in Ts^I, \exists prp \in PRP\ \ s.t.\ xi \in prp$. That is, there is a function $f : Ts^I \rightarrow PRP$

```
process The neural network
pre-condition ($PR_1^1 \vee PR_1^2, \ldots, \vee PR_1^s$) $\wedge$
              ($PR_2^1 \vee PR_2^2, \ldots, \vee PR_2^s$) $\wedge$
              ⋮
```

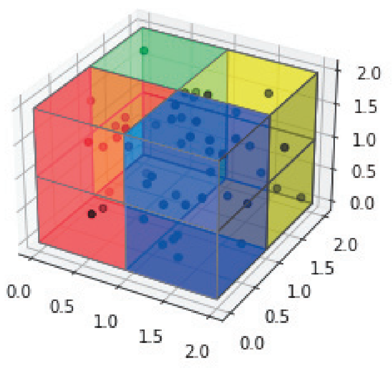
```
                $(PR_m^1 \vee PR_m^2, \ldots, \vee PR_m^s)$
post-condition  $PO_1 \wedge PO_2 \cdots \wedge PO_n$
end_process
```

Listing 3.2: *Formal Specification Illustration*

Fig. 3.2 is a case study of test case partition for input space of neural network. Let the input layer of NN have three neurons and $|PR_i^p| = 2\ where\ i = 1, 2, 3$, then the pre-condition partition of NN can be represented as a cube C in three dimensional space. The length, width and height of the cube are $|PR_1|$, $|PR_2|$ and $|PR_3|$ respectively. The set composed of all sub cubes is set PRP. Firstly, the pre-condition is partitioned by definition 7, as shown in sub Fig.3.2(a). The sub cubes in cube C are elements in PRP. Secondly, each element in Ts^I is mapped to different sub cubes, as shown in sub Fig.3.2(b).



(a) Pre-condition Partition



(b) Mapping of test suite

Figure 3.2: Test case segmentation for input space of neural network

The above is a mathematical description of pre-condition partition. It is worth noting that the number of pre-condition partitions is decided manually, i.e., when we cannot verify the pre-condition due to its too large range, we can find the corresponding smaller execution paths by means of pre-condition Partitions.

3.3.2.2 Identify Execution Paths

The execution path of pre-condition on a neural network must be a sub graph of the whole neural network. When we feed a test case from Ts into the neural network, we can obtain the execution path of that test case. We merge the exe-

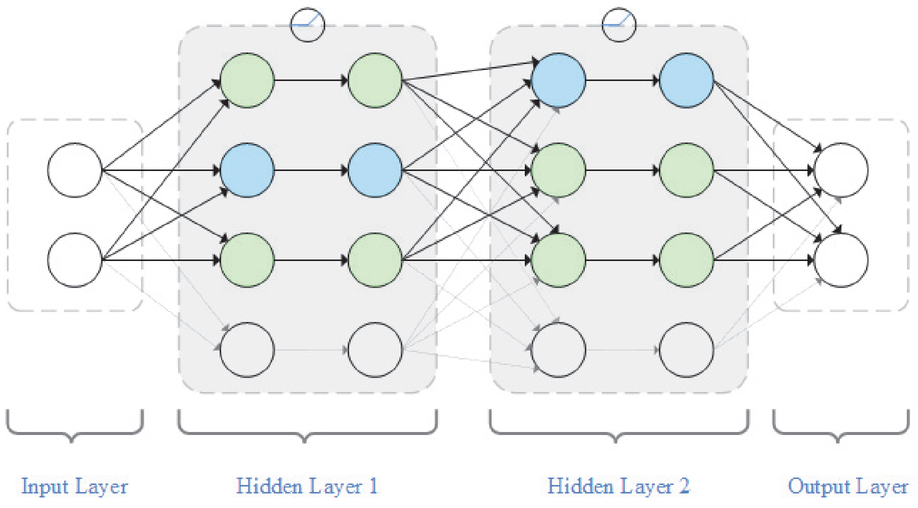


Figure 3.3: Execution path generation on pre-condition

cution paths of all test cases in pre-condition on together, which is the execution path on pre-condition.

To calculate the execution path for each test case, we need to record the behavior of each neuron. The calculation process is shown in Equation 3.7,

$$\begin{pmatrix} assert(o_1^i > h) \\ assert(o_2^i > h) \\ \vdots \\ assert(o_u^i > h) \end{pmatrix} = assert \begin{pmatrix} ReLU(n_1^i) \\ ReLU(n_2^i) \\ \vdots \\ ReLU(n_u^i) \end{pmatrix} \tag{3.7}$$

The meaning of the symbols in the equation is the same as in Preliminary. 3.2.1. By asserting the output values of the neurons in each hidden layer of the neural network, a boolean vector recording the state of each neuron's activation is finally saved for each hidden layer. We denote this Boolean vector as $AS(ts)$, indicating that the Boolean vector is about the test case ts. If we provide a Ts3.1 that meets the pre-condition, then the corresponding boolean vector for this set of test cases is $AS(Ts) = AS(tc_1) \vee AS(tc_2) \cdots AS(tc_t)$, where $|Ts| = t$.

To explain this process more clearly, we take the neural network in Fig. 3.3 as an example. We set the activation threshold to 0. The formal representation of Hidden Layer 1 is as follows,

$$\begin{pmatrix} o_1^2 \\ o_2^2 \\ o_3^2 \\ o_4^2 \end{pmatrix} = \begin{pmatrix} ReLU(n_1^2) \\ ReLU(n_2^2) \\ ReLU(n_3^2) \\ ReLU(n_4^2) \end{pmatrix} = ReLU\left(\begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{pmatrix}\begin{pmatrix} o_1^1 \\ o_2^1 \end{pmatrix} + \begin{pmatrix} b_1^2 \\ b_2^2 \\ b_3^2 \\ b_4^2 \end{pmatrix}\right) \tag{3.8}$$

the formal representation of Hidden Layer 2 is as follows,

$$\begin{pmatrix} o_1^3 \\ o_2^3 \\ o_3^3 \\ o_4^3 \end{pmatrix} = \begin{pmatrix} ReLU(n_1^3) \\ ReLU(n_2^3) \\ ReLU(n_3^3) \\ ReLU(n_4^3) \end{pmatrix} = ReLU\left(\begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \end{pmatrix}\begin{pmatrix} o_1^2 \\ o_2^2 \\ o_3^2 \\ o_4^2 \end{pmatrix} + \begin{pmatrix} b_1^3 \\ b_2^3 \\ b_3^3 \\ b_4^3 \end{pmatrix}\right) \tag{3.9}$$

The green node indicates the execution path of the first test case. Then the execution path of the $AS(tc_1)$ can be calculated as,

$$AS(tc_1) = \begin{pmatrix} assert(o_1^2 > 0) = True, & assert(o_2^2 > 0) = False, \\ assert(o_3^2 > 0) = True, & assert(o_4^2 > 0) = False, \\ assert(o_1^3 > 0) = False, & assert(o_2^3 > 0) = True, \\ assert(o_3^3 > 0) = True, & assert(o_4^3 > 0) = False \end{pmatrix} \tag{3.10}$$

Similarly, the blue node indicates the execution path of the first test case.

$$AS(tc_2) = \Big(False, True, False, False, True, False, False, False\Big) \tag{3.11}$$

Finally, we perform a disjunction operation on the execution paths of two test cases to obtain the overall execution path as fellow.

$$\begin{aligned} AS(ts_1, ts_2) &= AS(tc_1) \vee AS(tc_2) \\ &= (True, True, True, False, True, True, True, False) \end{aligned} \tag{3.12}$$

3.3.2.3 Generating Test Cases

In order to minimize the impact of pruning on the accuracy of the neural network and make the pruned neural network more suitable for verification. We used the gradient ascent algorithm to generate test cases. In this context, the symbol conventions are consistent with the preliminaries. Assuming that the result of $assert(o_j^i > h)$ is False, we can define the objective function 3.13,

$$X^* = \arg\max_X n_j^i(X; W; B) \tag{3.13}$$

where X are the input vectors, and W and B represent the set of weight vectors and the set of biases, respectively. By employing the computation method in the preliminary 3.2.4, we can calculate the gradient of the objective function with respect to X. Finally, we employ formula $X = X + \alpha \cdot \nabla_X n_j^i$ to generate test cases that satisfy the pre-condition constraint, by setting an appropriate learning rate α.

3.3.2.4 Network Pruning and Pre-condition Execution Path Generation

If the execution path of the neural network is recorded as computational graphs $G = \langle V, E \rangle$, then, the execution path of a pre-condition is the sub graph $G_p = \langle V_p, E_p \rangle$. where $V_p \subset V$, $E_p \subset E$. The points in set V_p are the activated neurons, and the edges in E_p are the edges connected to V_p. Therefore, the process of neural network pruning is the process of deleting points in $V_s = V - V_p$ and edges in $E_s = E - E_p$ in G.

Algorithm 1: NNTBFV

```
   Input:  $Its$ // Initial test suite
 1         $nn$ // Neural networks to be pruned
 2         $pre-condition$
 3         $postcondition$
 4         $count$ // number of iterations
   Output: $Execution\ Path$
 5 for $test\_case$ in $Its$ do
 6     $nn(test\_case)$
 7     $Neuron\_statement = AS(test\_case)$
 8 for $neuron\_state$ in $Neuron\_statement$ do
 9     if $neuron\_state$ is $False$ then
10         while $loop \leq count$ do
11             $test\_case = Generate(neuron)$
12             if $test\_case$ satisfies $pre-condition$ AND $nn(test\_case)$
                 satisfies $postcondition$ then
13                 $Neuron\_statement = AS(test\_case)$
14                 $loop += 1$
15             else
16                 $loop += 1$
17 $Execution\ Path = Pruning(Neuron\_statement)$
18 return $sub\_neural\_network$
```

The specific details of generating the execution path for Pre condition are shown in **Algorithm 1**, we first feed the initial test suite into the neural network that should be pruned. The activation of each neuron is recorded during the stage of performing the test as shown in line 7 of algorithm 1. In lines 8 to 11 of the algorithm 1, we find the neuron that is not activated in the neural network, and use the output value of the inactive neuron as the dependent variable and the input value of the neural network as the independent variable to perform the gradient ascent calculation and generate the test case. As in line 12 to 16, if the test case is found to satisfy the formal specification within a certain number of iterations, the neuron is successfully activated, otherwise, the neuron will be pruned. Finally, when the number of iterations of the algorithm 1 reaches the given count, the algorithm stops and returns $sub_neural_network$ in line 17 and 18.

3.3.3 Verification of Neural Networks

In this chapter, we formally outline the theoretical methods for verifying neural networks and analyzing verification results based on the PR of the execution path. The process of converting an execution path into constraints for counterexample resolution is beyond the scope of this paper, as mature research and tools already exist [36, 81, 97, 102] for this purpose[1].

Let the set PRP contain w elements, then $PRP = prp_1, prp_2, \cdots, prp_w$ and each element is in and each element is the Disjunctive Normal Form (DNF). The following constraints can be obtained:

$$\begin{pmatrix} \neg prp_1 \wedge constraints(path_1) \wedge PO \\ \neg prp_2 \wedge constraints(path_2) \wedge PO \\ \vdots \\ \neg prp_w \wedge constraints(path_w) \wedge PO \end{pmatrix} \tag{3.14}$$

Where $constraints$ refers to the process of encoding paths into constraints through the neural network verification tool. $path_i$ is the execution path of prp_i in the neural network, obtained by Algorithm 1. If no counterexamples exist in the $\neg prp_i \wedge constraints(path_i) \wedge PO$, the $path_i$ is proved to be reliable for its precondition. Conversely, if counterexamples exist, the counterexamples can be collected for fine-tune.

[1]https://github.com/dlshriver/dnnv

At this point, we have comprehensively demonstrated how to simplify and verify neural networks. However, we did not delve into the fine-tuning based on counterexamples. The reason is that fine-tuning of neural networks based on counterexamples is a separate topic in itself. Therefore, we will treat this part as future work.

3.4 Case Study

We use a three-layer FCNN to explain the work flow of the algorithm and deduce its effectiveness. It is a trained neural network, named FN. In FN, the weight tensor from the input layer to the hidden layer is denoted as A, and the weight tensor from the hidden layer to the output layer is denoted as B.

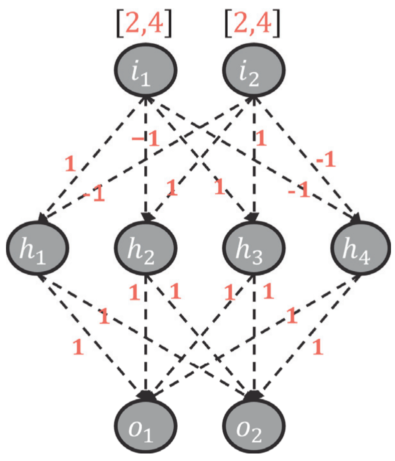


Figure 3.4: Network structure of FN

$$A = \begin{array}{c} \begin{array}{cccc} h_1 & h_2 & h_3 & h_4 \end{array} \\ \begin{pmatrix} 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{pmatrix} \end{array} \begin{array}{c} \\ i_1 \\ i_2 \end{array} \qquad B = \begin{array}{c} \begin{array}{cccc} h_1 & h_2 & h_3 & h_4 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \end{array} \begin{array}{c} \\ o_1 \\ o_2 \end{array} \tag{3.15}$$

where $i_{1,2}$, $h_{1,2,3,4}$ and $o_{1,2}$ represents the marker of neurons in the input layer, hidden layer and output layer, respectively. The network structure of FN can be graphically represented as Fig.3.4.

We formally specify the interval property of neural network using SOFL. It is given in the Listing 3.3.

```
process FN(i₁:real,i₂:real) o₁,o₂:real
```

```
pre  $2 \leq i_1 \leq 4$  and  $2 \leq i_2 \leq 4$
post  $6 \leq o_1 \leq 8$  and  $6 \leq o_2 \leq 8$
end_process
```

Listing 3.3: The interval property of FN using SOFL

For the pre-conditions of the FN, PR_1 and PR_2 are divided into $PR_1^p = \{PR_1^1, PR_1^2\}$ and $PR_2^p = \{PR_2^1, PR_2^2\}$ by definition 1, where $PR_1^1 = [2,3)$, $PR_1^2 = [3,4]$ and $PR_2^1 = [2,3)$, $PR_2^2 = [3,4]$. The set PRP is the Cartesian product of PR_1^p and PR_2^p. According to definition 2, we can get:

$$\begin{aligned} PRP =&\{PR^1 = \{PR_1^1, PR_2^1\}, PR^2 = \{D_1^1, D_2^2\}, \\ &PR^3 = \{PR_1^2, PR_2^1\}, PR^4 = \{PR_1^2, PR_2^2\}\} \end{aligned} \tag{3.16}$$

The value interval is substituted into ISS, which can be expressed as:

$$prp_1 = \{[2,3),[2,3)\} \quad prp_2 = \{[2,3),[3,4]\} \tag{3.17}$$

$$prp_3 = \{[3,4],[2,3)\} \quad prp_4 = \{[3,4],[3,4]\} \tag{3.18}$$

Next, we assign two test cases to each pre-condition. See Tab 3.2 for details. If a neuron in FN is activated during the forward propagation of both test cases, the neuron is marked with green. Red indicates that the neuron is not activated in both test cases. If one test case activates the neuron and the other test case does not activate the neuron, it is indicated in yellow.

Table 3.2: Test case corresponding to each *prp*

PRP	**Test case**	
$prp_1 = \{[2,3),[2,3)\}$	$i_1 = 2.0$	$i_2 = 2.5$
	$i_1 = 2.5$	$i_2 = 2.0$
$prp_2 = \{[2,3),[3,4]\}$	$i_1 = 2.0$	$i_2 = 3.0$
	$i_1 = 2.5$	$i_2 = 4.0$
$prp_3 = \{[3,4],[2,3)\}$	$i_1 = 3.0$	$i_2 = 2.0$
	$i_1 = 4.0$	$i_2 = 2.5$
$prp_4 = \{[3,4],[3,4]\}$	$i_1 = 3.0$	$i_2 = 4.0$
	$i_1 = 4.0$	$i_2 = 3.0$

Fig.3.5 is the execution path of test cases in *prp*$_2$ and *prp*$_3$, where the red neurons are inactive neurons. When the neural network pruning is performed,

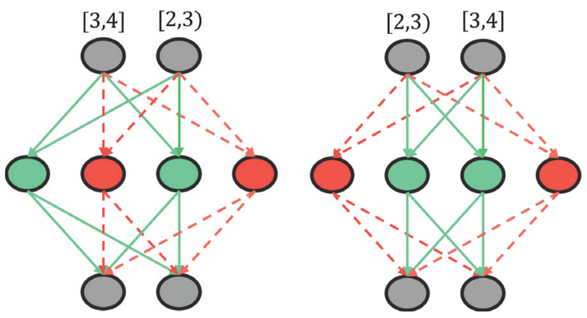


Figure 3.5: Execution paths on prp_2 and prp_3

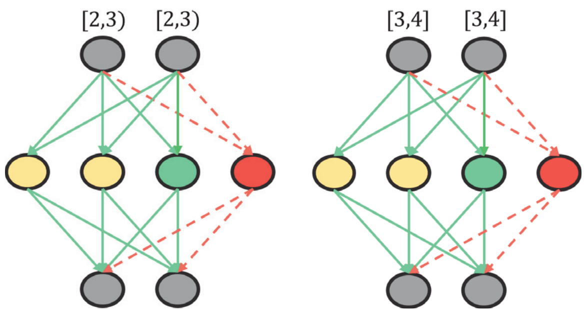


Figure 3.6: Execution paths on prp_1 and prp_4

the neuron and its connected edges will be deleted. Since the test case execution paths in prp_2 and prp_3 are the same, no neurons marked as yellow appear.

The difference is that in prp_1 and prp_4, as shown in Fig.3.6, the execution paths of the two test cases are different, and the neurons activate in different states when different test cases are executed, and we mark this class of neurons as yellow. Unlike the inactive neurons, such neurons will be retained in the pruning of FN.

Finally, we will verify each execution path. This step is usually given to the neural network verification tool. Specifically, we perform symbolic execution on FN, and then on prp_2, $o_1 = 2x, o_2 = 2x$, on prp_3, the output is $o_1 = 2y, o_2 = 2y$. FN is reliable on prp_2 and prp_3. However, when $input_1 = 2, input_2 = 2$ is executed on FN, the result is $output_1 = 4 \leq 6$ and $output_2 = 4 \leq 6$. Therefore, we consider $output_1 = 4 \leq 6$ and $output_2 = 4 \leq 6$ as a counterexample. The counterexample is made into a training set and the sub network is retrained.

3.5 Evaluation

We use the implementation of the concept of NNTBFV to test the ACAS Xu system. This test is mainly aimed at evaluating the following research questions (**RQ**s):

- **RQ1**: Can NNTBFV activate as many neurons as possible under the formal specification? How does it compare with verification-based pruning?
- **RQ2**: Does the simplification effect of NNTBFV change with the partitioning of the pre-condition?

In addition, we conducted theoretical and empirical analyses to answer RQ3.

- **RQ3**: Is the time complexity of neural network verification using NNTBFV better than that of verification conducted solely using neural network verification tools?

3.5.1 Baseline Models and Experiment Setup

3.5.1.1 Baseline models:ACAS Xu

ACAS Xu is a variant of Airborne Collision Avoidance System X for unmanned aircraft. Its purpose is to avoid collisions between aircraft and unmanned aircraft. The Model contains 45 ReLU-based FCNNs. Each neural network has 8 layers: an input layer with 5 neurons, an output layer with 5 neurons, and 6 hidden layers, each with 50 neurons. The input of the neural network can be recorded as a vector $X = (\rho, \theta, \psi, v_{own}, v_{intr})$. Fig.3.7 shows the significance of each variable in the horizontal scenario. The meaning and units of each dimension of X are:

- ρ : The distance between ownship and the intruder, unit is $feet$.
- θ : The angle between ownship heading direction and intruder, measured counterclockwise, unit is $radians$.
- ψ : The heading angle of intruder relative to the heading direction of ownship, measured counterclockwise, unit is $radians$.
- v_{own}, v_{intr} : Flight speed of each aircraft, unit is $feet\ per\ second$

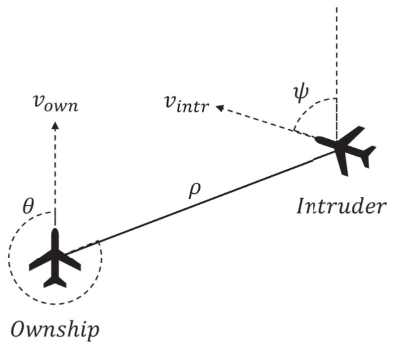


Figure 3.7: Geometry for horizontal scenarios of ACAS Xu Model

The output of the ACAS Xu model includes 5 scenarios, namely Clear of Conflict (COC), weak right, strong right, weak left, or strong left. It is a navigation suggestion for ownship. We record the states output by the ACAS Xu model as a set:

$$Y = [COC, weakright, strongright, weakleft, strongleft]$$

Due to the navigation suggestions for ownship, not only the characteristics in X need to be considered, it is related to the time until loss of vertical separation $T = [0, 1, 5, 10, 20, 40, 60, 80, 100]$ and Y of the previous moment. ACAS Xu has 45 neural networks (let Cartesian product $M = Y \times T$, then $|M| = 45$), each neural network can be denoted as N_{ij}, where i represents the i-th element of Y, j represents the j-th element of T, for instance, Neural network No.N_{24} represents the previous action as $weakright$ and $time = 10$.

3.5.1.2 Experiment Setup

All experiments are conducted on a windows 10 computer, equipped with a Intel i9-11900, 32GB of memory and a Nvidia RTX3090 24G graphics card.

3.5.2 Baseline Methods

To compare the performance of our proposed method, we selected two baseline methods, as follows:

- Verification-based pruning: A neural network pruning method based entirely on formal verification, which only needs to generate corresponding queries for the specification.

- Random-based pruning: Generate test cases that conform to the precondition in the formal specification in a uniformly distributed manner, and perform neural network pruning with the execution of the test cases.

The Verification-based method is the most rigorous pruning method among all the baselines, and can accurately identify whether there will be activation of each neuron under a given formal specification. Therefore, we use this method as our main baseline method, thereby demonstrating that the NNTBFV is a good approximation to the Verification-based method in a limited time. The Uniform distribution based method is a pruning algorithm without adding a search strategy. The effectiveness of the NNTBFV is explored by comparing it with the Uniform distribution based method.

3.5.3 Results

3.5.3.1 Answer to RQ1

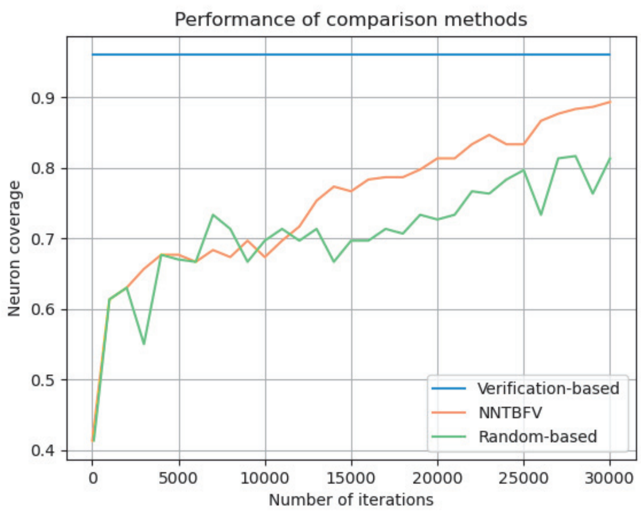


Figure 3.8: Comparison of neuron coverage

We iteratively generated test cases conforming to the formal specification and executed them on 45 ACAS Xu networks. The x-axis of Fig.3.8 represents the number of iterations to generate test cases and the y-axis represents the neuron coverage in Preliminary3.2.3 in ACAS Xu networks. The proportion is the average activation value of the neural network obtained after testing the

ACAS Xu networks system 10 times using NNTBFV.

The blue horizontal line in the figure indicates that the neurons identified by the Verification-based method for ACAS Xu networks under the formal specification that will definitely activate account for 95.8% of all neurons in ACAS Xu networks, i.e., even if 4.2% of the neurons in the ACAS Xu networks are removed, the model still conforms to the given formal specification. We use this method as our baseline. The yellow line represents the effectiveness of the method in NNTBFV. From the graph, it can be seen that as the number of iterations increases, the pruning method in NNTBFV can better approximate the baseline and outperform the removal rate of redundant neurons in the neural network by randomly generating test cases. After 30000 iterations, NNTBFV can ultimately activate 90.67% of neurons in ACAS Xu under formal specification. The method of relying on randomly generated test cases for neural network pruning can only activate 81.67% of neurons in ACAS Xu after 30000 iterations.

From the above comparion, NNTBFV can approximate the baseline for a given formal specification. This means that our method can activate as many neurons as possible under the formal specification. And NNTBFV has a higher recognition of neuronal activation rate and is more stable than Random-based method.

3.5.3.2 Answer to RQ2

In order to explore the relationship between pre-condition partition and neuron coverage, demonstrate the effectiveness of using pre-condition partition to reduce the size of neural networks in a larger proportion. we tested the neuron coverage corresponding to different pre condition partition levels, as shown in Fig 3.9. The horizontal axis represents the partition level for a given pre condition, and the vertical axis represents the change in ACAS Xu model neuron coverage in Preliminary 3.2.3 as the partition level increases.

We choose the formal specification ϕ_1 as the pre-condition to be split, and the model N_{ij} as the neural network model under test.

The limiting case for pre-condition partition is the neuronal coverage of a single test case in the forward propagation of a given neural network. Therefore, we use the neural network coverage of a single test case as the baseline for forward-propagating pre-condition partition.

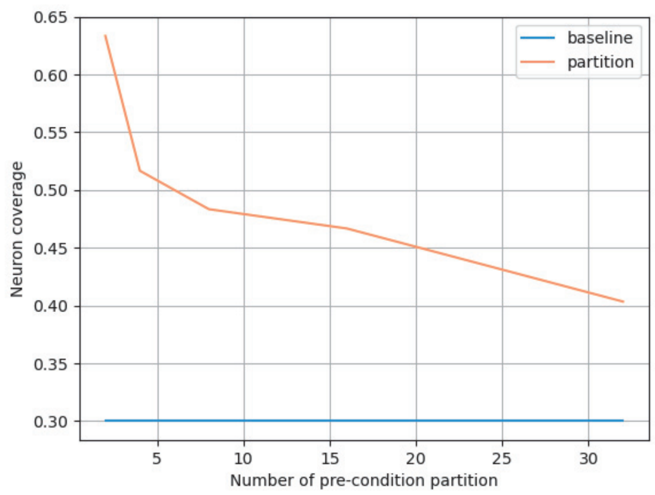


Figure 3.9: Comparison of pre-condition partition

3.5.3.3 Answer to RQ3

In order to answer the time complexity of evaluating NNTBFV and thereby answer RQ3, we introduce the following lemma.

Lemma 1 (See [36]). *Neural network verification is an NP-hard problem, and a 3-SAT problem can be reduced to a neural network verification problem.*

From Lemma 1, we know that the time complexity of the neural network verification algorithm is greater than the time complexity of the 3-SAT algorithm. We only need to show that the time complexity of NNTBFV is superior to the time complexity of the 3-SAT algorithm! Therefore, in the theoretical derivation, we directly take the time complexity of the 3-SAT solving algorithm as the time complexity of the verification algorithm.

In Figure 3.10, we use the horizontal axis to represent the scale of the neural network to be verified, and the vertical axis to represent the time complexity of the algorithm. It is known that the time complexity of the current 3-SAT solving algorithm is $\mathcal{O}(1.3^n)$ marked as blue line. The Time complexity of NNTBFV simplified neural network is marked as red line, and its Time complexity is $\mathcal{O}(n^2)$.

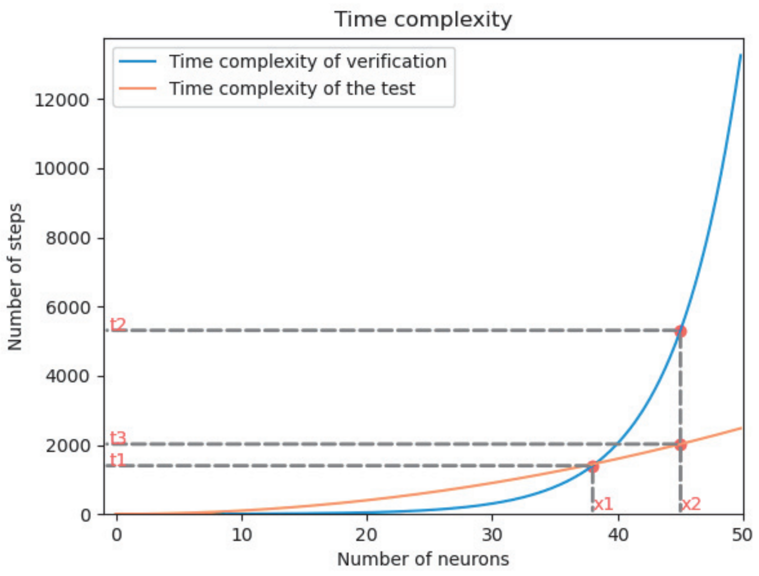


Figure 3.10: Comparison of Time Complexity

When the scale of the neural network is x_2, the time spent using the verification algorithm to verify the neural network is t_2, while the time spent using NNTBFV to verify the neural network is the sum of the time t_3 required to simplify the network from scale x_2 to x_1 and the time t_1 required to verify the neural network of scale x_1. Therefore, we only need to prove that $t_2 > t_1 + t_3$ (or $t_2 - t_1 > t_3$) to demonstrate that the time complexity of NNTBFV is better to that of the verification algorithm.

Assuming that NNTBFV can remove $k\%$ neurons, as the scale of the verification problem increases, there must be $\mathcal{O}(k\%1.3^n) > \mathcal{O}(n^2)$.

3.5.4 Threats to Validity

The primary threats to the validity of our experiments stem from two sources: the selected benchmark project, and the inherent randomness associated with a test-based algorithm.

Although we used the well-known ACAS Xu model as our benchmark project in order to ensure the reliability and comparability of the experimental results, the training data of this project are not publicly available, only the weight and bias of the model are published, so there is a possible threat to the validity when reproducing the Pytorch version of the network. In addition, although our experiments present results as the average of multiple solutions, the test-based algorithm is still somewhat stochastic.

3.5.5 Discussion

Our paper discusses the problem of how to obtain the corresponding execution path by a test-based approach given a specification. It is experimentally confirmed that the NNTBFV can effectively approximate the true execution path of the neural network for a given formal specification.

As the NNTBFV yields approximate solutions, it effectively addresses the frequent timeouts encountered by verification-based algorithms when verifying large scale models. Moreover, our algorithm is better suited for parallel computing due to its test-based nature, enabling algorithm acceleration through cluster computing.

We propose an algorithm to partition the pre-condition in the formal specification and find that the size of the execution path neuron corresponding to that execution path decreases as the refinement level increases. This is intuitive. This is because the limit of the refinement level is the execution path of a single test case. As the refinement level increases, the number of activated neurons in the neural network tends to decrease. However, the relationship between different division strategies and neuron coverage is not discussed in this paper and left for future work.

The above analysis shows that NNTBFV can alleviate the difficulties of large scale model verification. Due to the size of the model, it is not possible to verify the entire model, and if a portion of the inputs require high reliability, we can restrict the input intervals of those that require high reliability to a small range, and then use NNTBFV to obtain a convenient execution path for verification.

3.6 Related Work

The application of traditional testing techniques and traditional verification techniques to enhance the reliability of neural networks are two separate directions. Since NNTBFV is derived from traditional TBFV and combines both techniques, we briefly review the existing work on testing and verification[2] in deep learning, as well as recent advances in TBFV.

Currently, the research of neural network testing mainly focuses on how to attack and defend, that is, how to generate adversarial example and how to prevent being attacked by it. Search-based methods such as FGSM, IGSM [24, 38] attack neural networks by searching for adversarial examples. Subsequently, deepfool, JSMA [57, 66] etc. have made different degrees of improvement in search efficiency and approach. In the area of neural network defense testing, the field is inspired by traditional software testing methods and proposes the criterion of neuron coverage [68]. Other works that use neuron coverage in different dimensions to improve the robustness of neural networks include DeepGauge [86], DeepConcolic [86] etc.

In terms of neural network verification, many scholars have proposed different algorithms for neural network verification. Each algorithm tries to reduce the output range of neural network as much as possible from the perspectives of reachability, optimization and search. Optimization-based neural network verification usually transforms the neural network into a constraint solving problem, such as NSVerify [51], MIPVerify [89]. Reluplex [36] and Planet [18] et al. studied in combining search and optimization like to improve the accuracy and efficiency of verification. In addition, methods that combine search with Reachability include Neurify [97] etc.

Unlike the above work, we want to combine testing and verification to provide a method to verify and enhance the reliability of neural networks under a formal specification. Similar to this in traditional software engineering research are TBFV [47], TBFV-SE [96] etc.

3.7 Summary

NNTBFV is proposed as a method for implifying and verifying neural networks. In NNTBFV, testing-based neural network pruning are introduced for reducing the neural network neurons that need to be verified. Our case studies and experiments have largely confirmed the feasibility of NNTBFV.

[2]https://github.com/stanleybak/vnncomp2023/

In the further work, we will focus on two main themes. The first is fine-tuning of neural networks based on counterexamples, specifically, how to effectively fine-tune execution paths. The second theme is the precise functional relationship between the neuron coverage and the scale of precondition partition, and whether this relationship is related to the stability of the neural network.

Chapter 4

A Theoretical Approach: Locally Interpreting Neural Networks using Testing-Based Formal Verification

4.1 Introduction

4.1.1 Background and Motivation

When a neural network model has a corresponding formal specification, the neural network verification algorithm can give the model to satisfy the corresponding formal specification or not within a certain period of time. If the verification result can not be provided within the specified time frame, the verification algorithm can also notify the engineers that the verification problem has *timedout*. Engineers can further address the *timeout* issue encountered during verification by adopting neural network pruning or enhancing computer performance, among other methods.

The premise of applying these methods is that we provide the formal specification of the model to be verified, including both the precondition and postcondition. The verification process is similar to that of traditional software, where the formal specification and the neural network model are jointly transformed into an SMT (Satisfiability Modulo Theories) problem through formal methods. However, unlike traditional software, there are many neural network models in reality for which it is difficult to provide corresponding formal specifications. This challenge arises because neural network models are trained on data, rather than being constructed based on rules like traditional software. This also leads to disadvantages such as weak interpretability and opaque decision-making processes in neural network models, which are particularly evident in fields such as image recognition and Natural language processing.

The training process of a neural network involves searching through a training dataset, which can be represented as a set of $(case, result)$ pairs, to obtain a model that maps the input space to the output space. This model is then used to infer many cases that it has not seen before. Due to the complexity and diversity of the training data, the greatest challenge lies in formally defining and providing the precondition.Fortunately, in the application domain of neural networks, especially in classification tasks, we can relatively easily define the output range of each neuron in the output layer (in the field of formal methods, this definition of output range is also known as providing the postcondition). Therefore, how to deduce, based on the postcondition of the neural network as well as the model's own parameters and structure, which features in the potential input space will lead the neural network model's output to conform to the postcondition is a question worth investigating. At the same time, this issue is exactly equivalent to the study of neural network interpretability. That is, research on how to use neural network verification algorithms to enhance the reliability of neural network models without a precondition is equivalent to research on how to use these algorithms to explain the behavior of neural networks.

More generally speaking, if we view the training of neural networks as an induction process (i.e., inferring the model from the case and result), then explaining neural network models can be seen as an abduction process (i.e., inferring the precondition from the model and a result). Through the abduction process, feature importance or saliency maps can be generated for neural network models to highlight the inputs that are most influential for a given output. This can be applied in many scenarios.

In software engineering, developers can use the feature importance or saliency maps explained from neural networks to verify whether the network meets the users' needs. Furthermore, they can use these insights to test and verify neural network models, thereby enhancing the reliability of the neural network models post-deployment.

Researchers can use neural network explainability to understand the behavior of AI and design better neural network structures. Data scientists can even leverage neural network explainability to comprehend AI model decision-making processes. For example, clinicians can review AI diagnostic results through saliency maps.

Overall, the application of neural network explainability is very broad and is increasingly attracting the attention of researchers.

4.1.2 Challenges and Proposed Methods

Although neural network interpretability has a wide range of applications in many fields, there are still numerous challenges that exist.

Challenge One: How to address the phenomenon of model saturation that occurs in Testing-based approaches (In the field of neural network interpretability, the methods are known as perturbation-based approach) and Gradient-based approaches. These methods typically view the neural network model as a black box, applying perturbation strategies to the model's inputs to generate appropriate test cases and observing the impact of these test cases on the model's predictions. Although these test-based methods are relatively easy to understand, the quality of their explanations highly depends on the choice of perturbation strategy. Moreover, these test-based explanation algorithms may be affected by model saturation, where the model's output no longer changes in response to input perturbations, leading to a decrease in interpretability.

Unlike testing-based methods, which require the generation of a large number of test cases through perturbations, gradient-based approaches determine the impact of features on the model's behavior by utilizing the gradients of the model with respect to its input features. Theoretically, it is believed that the larger the gradient value of a feature, the greater its influence on the model's behavior, and vice versa. The feature importance derived from these methods takes advantage of the model properties to some extent, enhancing interpretability. However, model saturation can lead to vanishing gradients, which in turn may lead to misleading interpretations by gradient-based methods. For instance, in a cat image classification model, if the saliency map shows a very uniform distribution of gradients across pixels, it indicates that no particular pixels are important. However, in reality, features such as the cat's ears and eyes are critical for the classification decision.

Challenge Two: How to view neural network models as white boxes and to intuitively explain them from a formal methods perspective. Current research on neural network interpretability often regards the model itself as a black box, explaining the behavior of the neural networks by the characteristics displayed during the execution of test cases. Although these algorithms provide explanations that are easy to understand, the heuristic nature of treating the models as black boxes means that the explanations provided by the algorithms cannot guarantee correctness and conciseness. Some methods of abductive explanations have been proposed by the formal methods community and are considered useful. However, for those utilizing interpretability, it is generally desired that the explanatory algorithms provide explanations of the model that are more intuitive and easier to understand. Furthermore, methods based solely on formal verification, since they rely on the neural network verifier to provide answers, require the establishment of a strategy for querying the neural network verifier in order to obtain explanations of the model. The difficulty lies in the fact that the neural network verifier itself requires substantial computational power, and the formal verification of neural networks has been proven to be an NP-hard problem, making explanation methods based on formal verification difficult to apply to large-scale models.

Although there is some research on heuristic algorithms and approximation methods to obtain approximate results of model explanations, which has played a certain accelerating effect, those improved methods only alleviate the shortcomings of methods based on formal verification.

To derive explanations from neural network models that can contribute to the enhancement of neural network reliability, and to mitigate the challenges faced by existing explanatory algorithms to some extent, we propose a theoretical approach. That is, utilizing testing-based formal verification to interpret the behavior of neural networks.

In this chapter, we propose an algorithm to derive the precondition of a pre-trained neural network from its post-condition, named DeepTBFV. The precondition can be used to verify and explain the behavior of the neural network and assist developers to increase the reliability of the neural network. Firstly, we forward propagate a test case on a neural network, and generate the execution path of the test case. Secondly, we propose a technique to encode the execution path as a multivariate linear inequality system, and since the activation state of neurons is fixed in the execution path, the linear constraints generated by the execution path do not need to introduce the relaxation variables. Finally, we use the classification results obtained from this test case to define the post-condition of the test case and derive the precondition in reverse. The process of defining the post-condition is based on an assumption that will be described in the subsection 4.3.4.

4.2 Preliminary

4.2.1 Floyd-Hoare Logic

Floyd Hoare Logic [71], also known as Hoare Logic, represents predicate logic and a set of axioms in the form of Hoare triples, and then defines the semantics of the programming language. The specific form of Hoare triples is as follows:

$$\{pre\}\ c\ \{post\} \tag{4.1}$$

where c is a specific program code, pre indicates the preconditions in the program, which describes the program state before executing c, and $post$ indicates the post condition in the program, which describes the program state after executing c. Such a Hoare Logic triple indicates that if an input of the program c meets its pre, the output of c should meets $post$ after executing program c. Otherwise, program c must have errors.

To formalize the program, Hoare Logic defines inference rules [35] for each of grammar clauses. In the inference process of the neural network model, there is no iteration. Therefore, when using Hoare logic to deduce the preconditions of a neural network model from its postconditions, we only need to consider the rules of assignment, condition and sequence in Hoare logic. The detailed rules are as follows.

$$Assign \quad \frac{}{\{Q(E/x)\}\ x := E\ \{Q\}} \tag{4.2}$$

Assignment statement rule is to perform the assignment statement $x := E$ on all x in the precondition $\{Q(E/x)\}$, that is, to replace all x in the precondition with E to obtain the post condition $\{Q\}$.

$$If \quad \frac{\{P \wedge S\}c_1\{Q\} \quad \{P \wedge \neg S\}c_2\{Q\}}{\{P\}if\{s\}\{c_1\}else\{c_2\}\{Q\}} \tag{4.3}$$

The rule for conditional statements is used to formally describe the expression of conditional statements in Hoare logic, particularly in TBFV. When the execution path of the program is fixed, the rule for conditional statements can be expressed as:

$$\frac{}{\{S \wedge Q\}\ S\ \{Q\}} \tag{4.4}$$

4.2.2 Symbolic Execution

The basic idea of symbolic execution is to use symbols to replace the concrete input values required for program execution, and to symbolically simulate the execution of each instruction of the program, to form corresponding symbolic expressions for the execution paths of the program, i.e., symbolic execution generates a series of constraints on the execution paths of the program.

Symbolic execution can be divided into static and dynamic symbolic execution [5, 37, 62], depending on whether the source program is executed or not. Static symbolic execution does not execute the source program. Instead, it first uses static analysis to determine the program's execution paths. Then, a solver is used to determine if these paths are reachable. Finally, it outputs all the reachable paths of the program. On the other hand, dynamic symbolic execution [13] records the execution paths of the program while it is actually running and outputs these paths as a set of constraint conditions. In this chapter, we apply the concept of dynamic symbolic execution to fully connected neural networks.

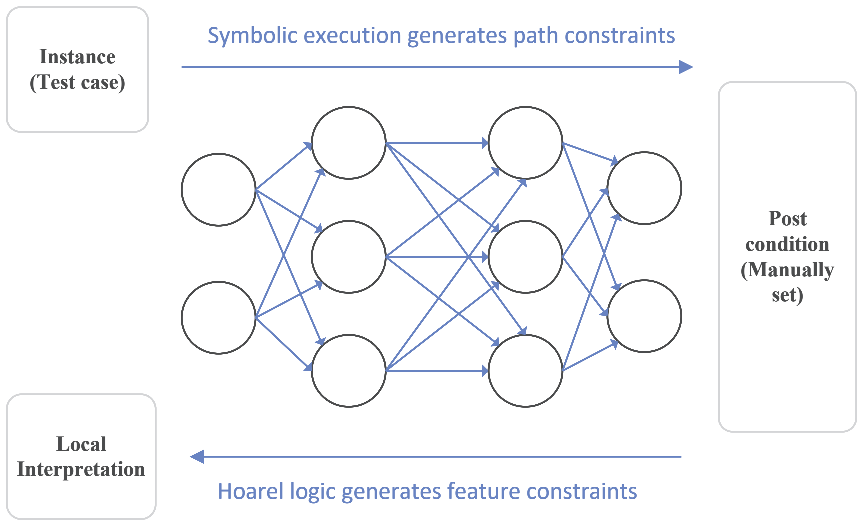

Instance (Test case)
Symbolic execution generates path constraints
Post condition (Manually set)
Local Interpretation
Hoarel logic generates feature constraints

Figure 4.1: Execution path generation

4.3 Theoretical Method

4.3.1 Formal Local Interpretation and Overview

In this chapter, our focus is on the local interpretability in classification tasks of fully connected neural networks. The task can be formally represented as a four-element tuple $\langle F, D, N, C\rangle$. Here, $F = \{f_1, f_2, \ldots, f_m\}$ is denoted as the set of neurons in the input layer (also referred to as features). The set $D = \{d_1, d_2, \ldots, d_m\}$ is denoted as the input space of each neuron in F and $C = \{c_1, c_2, \ldots, c_n\}$ means a set of classes. N is the neural network model, which can be donated as a map $N : \mathcal{F} = \{d_1 \times d_2 \times \cdots \times d_m\} \rightarrow C$.

Typically, given a $s \in \mathcal{F}$, we can compute its category $c \in C$, that is, $N(v) = c$, but we do not know which parts of s have played a decisive role in determining the computation result as c. When the s is executed in N, its execution path p, being a set of constraints generated by symbolic execution, is deterministic. Then, there exists the set of intervals $T = ([t^l_1, t^u_1], [t^l_2, t^u_2], \ldots [t^l_m, t^u_m])$, if $s \in \mathcal{T} = [t^l_1, t^u_1] \times [t^l_2, t^u_2] \times \cdots \times [t^l_m, t^u_m]$ and $\forall x \in \mathcal{T}$, $N(s) = N(x)$. We then consider the set T as a formal interpretation of N under the path p. Since T is constrained to the execution path p of the sample s in N, it is referred to as the local interpretation of s.

In order to realize the above mentioned formal local interpretation, DeepTBFV is mainly divided into two phases, in the first phase, the neural network model executes the instance to be interpreted, and obtains the execution path of the instance in the model, i.e., a series of constraints on the execution path. As shown in Fig. 4.1, this process is similar to dynamic symbolic execution to generate path constraints. In the second phase, we artificially set the output into the range of neurons, i.e., we give the post condition of the neural network model. next, as shown in Fig. 4.1, DeepTBFV generate the path constraints based on the post condition and the path via Hall logic. constraints to derive a set of constraints on the neurons in the input layer. We consider this set of constraints as the formal local interpretation of the instance under the specified neural network model.

4.3.2 Testing-based neural network execution path generation

The execution of a test case by the neural network means that the test case is used as the input for forward propagation. In the process of forward propagation, the state of each neuron is fixed. That is, we consider a neuron n_i in a neural network (NN) to be activated for the test case if its output is greater than a threshold value. Conversely, if n_i is less than the threshold, then the neuron n_i is not activated. We record the activation state of each neuron when the neural network forward propagate for the test case.

Formally, we define all neurons in the neural network as the set $neurons$. Then, the neurons activated in the neural network can be defined as the set $ActiveN = \{n \mid n \in neurons \land Out(n) > \theta\}$, where θ is the threshold in neurons and $out(n)$ records the output value of neuron. Similarly, the inactive neurons in the neural network can be defined as the set $InactiveN = \{n \mid n \in neurons \land Out(n) \leq \theta\}$. As shown in Fig. 4.2, in the forward propagation of neural networks, we use blue for the neurons activated in the test case and red for the inactive neurons. Then, the process of assigning activation states to the neuron in $neurons$ through the test cases is called execution path generation.

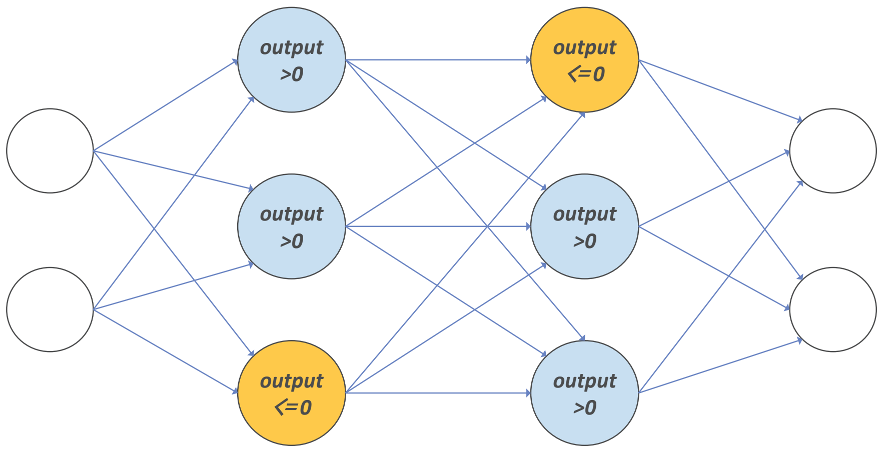


Figure 4.2: Execution path generation

4.3.3 Formal modeling of neural network paths

Since the state of activation of each neuron is fixed, we can convert the state of each neuron into a multiple linear inequality. Let $Neuron_j^i = \{x_1^{i-1}, x_2^{i-1}, \cdots x_n^{i-1}\}$ is the set of inputs of the j-th neuron in the i-th layer of the neural network. If the activation function of Neuron is ReLU and is activated by the test case, the activation state of the neuron can be modeled as follows,

$$If : Neuron_j^i \in ActiveN$$

$$Then : Out(Neuron_j^i) = w_1^{i-1} \cdot x_1^{i-1} \cdots + w_n^{i-1} \cdot x_n^{i-1} > 0$$

Similarly, if the neuron is not activated by the test case, then the activation state of the neuron can be modeled as follows,

$$If : Neuron_j^i \in InactiveN$$

$$Then : Out(Neuron_j^i) = w_1^{i-1} \cdot x_1^{i-1} \cdots + w_n^{i-1} \cdot x_n^{i-1} \leq 0$$

According to the above rules, we can model the process of forward propagation of test cases in the neural network into a system of multivariate linear inequalities. Algorithm 2 specifically describes how to model the execution path of the neural network.

Algorithm 2: Formal modeling of the neural network pathes

```
   Input: $Neurons$ ; $ActiveN$ ; $InactiveN$
   // List of linear constraints corresponding to neural
      networks
   Output: $Constraint$
1  $Constraint = list()$
2  for $neuron$ in $Neurons$ do
3      if $neuron$ in $ActiveN$ then
4          $Constraint.append(Out(neuron) > 0)$ else
5              $Constraint.append(Out(neuron) \leq 0)$
6  return $Constraint$
```

4.3.4 Deriving Preconditions from Post-conditions

When the state of each neuron in the neural network is fixed, we only need to carry out the Floyd-Hoare Logic assignment statement for the constraints of each neuron. This means that the constraints of each neuron are replaced by variables through the assignment statement. The constraints of each neuron we can get only include the input of the neural network.

Specifically, let $Inp = \{x_1, x_2, \cdots x_n\}, Inp \in R^n$ as the input of the neural network and $Out = \{y_1, y_2, \cdots y_m\}, Out \in R^m$ as the output of the neural network. Moreover, We record the replacement expression of the j-th neuron in the i-th layer as f_j^i. Then the constraint corresponding to neurons can be denoted as

$$f_j^i(x_1, x_2, \cdots x_n) \leq 0 \tag{4.5}$$

or

$$f_j^i(x_1, x_2, \cdots x_n) > 0 \tag{4.6}$$

The constraints of the output layer of the neural network are different from those of the middle layer. It should conform to the user-defined post-conditions. Here, the constraints of neurons in the output layer can be denoted as,

$$\alpha_j \leq y_j(x_1, x_2, \cdots x_n) \leq \beta_j \tag{4.7}$$

Where $y_j(x_1, x_2, \cdots x_n)$ is a multivariate linear polynomial and j represents the j-th output of the output layer. The interval $O_j = [\alpha_j, \beta_j]$ means the post-condition of y_j.

In addition, we also propose a method to construct post-conditions for the path generated by test cases. If the test case activates a neuron in the output layer, it means that the output value of the activated neuron is greater than that of other inactive neurons. Therefore, we can formally record this statement as $y_j^a > y_i^{in}$, where $i \in \{1, 2 \cdots m\} \backslash \{j\}$, a represents activated neuron, in represents inactive neurons. The activated neurons are marked as j. Since the output value of a neuron has an upper and lower limit, such as $INT8$ quantization of a neural network, the output values of neurons are quantized to $[-128, 127]$. Therefore, we can also give the upper and lower limits of the output value of neurons in the output layer. Let the upper and lower bounds of the output layer neurons be $[l, u]$, where the l denotes upper bound, and u denotes lower bound. Then,

$$\forall y_i \in Out, \exists\ \alpha_i \leq y_i \leq \beta_i \tag{4.8}$$

where $i \in \{1, 2, \cdots m\}$, and m is the number of output layer.

4.4 Case Study

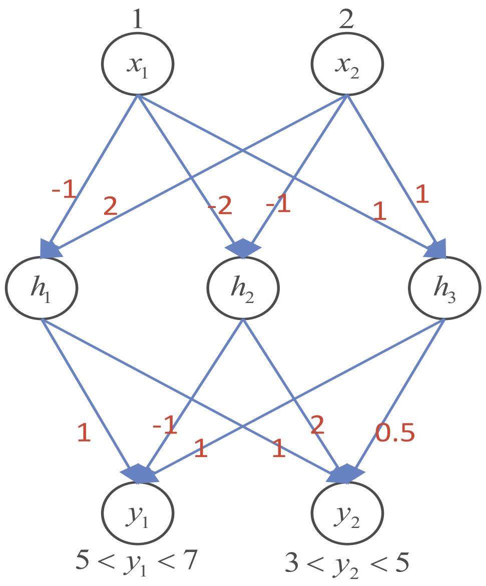


Figure 4.3: Example showing how to execute DeepTBFV

In this section, we use a three-layer neural network to show how DeepTBFV works and to demonstrate its effectiveness. Consider a ReLU-based neural network G. The structure and weights of this network are shown in Fig 4.3. Without loss of generality, we can assume that $bias = 0$ for each neuron in G. We denote the neurons of the input layer, the neurons of the hidden layer and the neurons of the output layer as $\{x_1, x_2\}$, $\{h_1, h_2, h_3\}$, $\{y_1, y_2\}$ respectively.

To execute DeepTBFV for G, all the steps are given in sequence as follows:

Step 1: Select the test case of interest as input to the neural network, suppose we use $x_1 = 1$, $x_2 = 2$ as the test case.

Step 2: Record the activation of each neuron in the hidden layer of the neural network,

$$h_1 = ReLU(-x_1 + 2x_2 = 3) \in ActiveN$$

$$h_2 = ReLU(-2x_1 - x_2 = 0) \in InactiveN$$

$$h_3 = ReLU(x_1 + x_2 = 3) \in ActiveN$$

Step 3: Generate neural network middle layer constraints,

$$h_1 = -x_1 + 2x_2 > 0$$

$$h_2 = 2x_1 - x_2 \leq 0$$

$$h_3 = x_1 + x_2 > 0$$

Step 4: Generate output layer constraints, here we assume that the output layer has constraints,

$$5 < y_1 = h_1 - h_2 + h_3 < 7$$

$$3 < y_1 = h_1 + 2h_2 + 0.5h_3 < 5$$

Step 5: Through Floyd-Hoare Logic, the system of inequality equations for each neuron in the input layer is derived backwards,

$$2x_1 - x_2 \leq 0$$

$$-x_1 + 2x_2 > 0$$

$$-x_1 - x_2 < 0$$

$$3 < -0.5x_1 + 2.5x_2 < 5$$

$$5 < 3x_2 < 7$$

4.5 Related Work

As neural network technology is progressively applied in an increasing number of fields, the issue of its interpretability has also gradually become a hot topic of research. In this section, we will categorize and summarize the research achievements related to the interpretability of neural networks. Given that our work is an attempt to provide local interpretation for neural network models, we will, after an overview of the research outcomes, discuss in detail the progress related to local interpretation methods for neural networks, as well as their theoretical contributions and the challenges they face. Finally, we will summarize the differences between DeepTBFV and existing research. Due to the overlapping definitions of interpretability (focusing on the 'what and why') and explainability (focusing on the 'how') [3,77,90] in academia, the terms interpretability and explainability will be used interchangeably in the literature.

Taxonomy and Overview: Based on the purpose of neural network explainability algorithms, they can generally be divided into two types of methods: global interpretation and local interpretation [42]. The purpose of global interpretation approaches is to gain a comprehensive understanding of the behavior of neural networks by seeking explanations across the entire input domain, through analysis of the neural network model's training data, architecture, and weights. Common methods involve using tree-based models [2,78,100,101] or rule-based models [11,41,59,87,91] to approximate a trained neural network model and then interpreting the decision-making process of the neural network model. The method of using interpretable models to approximate a neural network model can provide a global interpretation, but it is also limited by the scale of the model being interpreted. When faced with large-scale neural network models, global interpretation methods will become difficult to understand due to an excess of features (such as decision trees with very great depth). Additionally, this method will also consume a considerable amount of computational power. In contrast, local interpretation methods focus on understanding why the neural network model makes a particular prediction or classification for a single instance. Because local interpretation approaches do not require interpreting the entire neural network model, they have the advantages of lower computational power consumption and ease of implementation. Additionally, there are also some methods that can utilize local interpretation to understand neural network models globally [33]. Our work primarily focuses on how to generate local interpretations for neural network models, so we will next elaborate on and compare work related to local interpretations or explanations.

Local Interpretation: In the field of local interpretation of neural networks, it can generally be divided into test-based methods and model-based methods. Among them, research on testing-based methods ignores the internal working mechanisms of the neural network model, relying instead on generating test case strategies to quantitatively analyze the impact of changes in input data on the model. The methods based on testing can be further divided into perturbation-based methods and adversarial-based methods, with the main difference being the strategy used to generate test cases. Preturbation-based methods [20, 21, 75, 103, 105] mainly obtain a test suite by masking different areas of the model's input data. Then, using the forward propagation of the neural network model, the different test cases in the test suite are evaluated. This process allows for the analysis of which parts of the input data have a significant impact on the output of the neural network model. The idea of perturbation testing is embodied in the ZFNet [105] by Zeiler and Fergus. However, ZFnet, due to its relatively simple strategy in generating test cases, can lead to incorrect interpretations in some situations. This has led to subsequent research [20, 21, 103] that aims to improve the test case generation strategy of ZFnet. Local Interpretable Model-Agnostic Explanations(LIME) [75] and its improved algorithms [52, 74] extends the concept of ZFnet from being model-specific to model-agnostic. Although perturbation-based methods are lightweight and easy to understand, they encounter issues with the poor generalization [24, 104] of neural network models. In some cases, this can lead to the failure of perturbation-based algorithms in interpreting neural network models. For this reason, a series of adversarial-based methods [58, 76, 94] have been proposed. The core idea of this type of algorithm is to use a strategy for generating adversarial examples to create test data, and then use the generated test cases to explore the behavior of the model. Since the test data used are adversarial examples, it can effectively enhance the robustness of the interpretation of the neural network model.

Model-based methods, though sometimes employing strategies for generating test cases, differ from testing-based methods [7, 29, 105]. The model-based approach emphasizes the use of the neural network model's internal information, such as structure, weights, and biases. They provide explanations for the model through techniques like gradient computation, class activation mapping, and formal modeling. In general, model-based approaches can be subdivided into three types: gradient-based methods, methods based on neuronal states, and methods based on formal verification. The core idea of gradient-based [17, 85] methods is to treat the trained neural network model as a function and calculate the gradient of each independent variable in the input layer at a given point (the instance that needs to be interpreted). If the independent variable has a large gradient, it is considered that this feature has a significant impact on the output of the neural network model, and vice versa. This idea was first applied in the field of machine vision [82]. Subsequently, springenberg et al. [84] proposed a different processing strategy for gradients. Another method that similarly modifies the processing strategy for gradients is the deconvnet [54]. Although these gradient-based methods can utilize the intrinsic information within neural networks to some extent to obtain model interpretations, their reliance on gradients means that when faced with the problem of gradient vanishing, these methods may fail to provide interpretations of the model, or even yield incorrect explanations. Neuron state-based methods can avoid the issue of gradient vanishing that comes with solving gradients. Typically, neurons in a neural network have only two states: activated or not activated. The core idea of neuron state-based methods is to analyze the activation states of neurons when executing different test cases. DeepLIFT [80], a representative algorithm of this concept, assigns scores by comparing the activation of neurons to their reference activations and evaluating the differences. Similar work includes LRP [4] and and its improved variants. Furthermore, there are some interpretive strategies that use class activation mapping to explain neural networks, which have also yielded satisfactory explanatory results. These algorithms though overcome the problem of inaccurate model interpretation brought about by the vanishing gradient. However, most of them are heuristic algorithms with the disadvantage of weak theoretical basis. The activation state of neurons intuitively represents the properties of the neural network. Tt is still unknown what kind of neuron activation state represents the specific characteristics of the neural network model. For example, does a high neuron activation rate necessarily mean that the neural network is robust? There is still no definite answer to this question.

Formal interpretation: Unlike heuristic-based neural network interpretation algorithms, since the formal interpretation [7, 29, 34] is typically based on a neural network verifier, it can endow the explanation with formal guarantees.

Formal interpretations are very useful, but they still face two challenges. The first challenge is that the objects to be verified need to be set up manually and the human involvement may affect the interpretation given by the algorithm [6]. The second challenge is that the generation of existing Formal interpretations frequently invokes the neural network verifier which need cost a lot of arithmetic power. In contrast, DeepTBFV does not need to call the neural network verifier; we fix the execution paths in the neural network, which eliminates the need to utilize the neural network verifier to estimate the output range of each neuron. In addition, DeepTBFV's scheme of directly utilizing the constraint solver also reduces manual intervention in the algorithm.

4.6 Summary

We propose DeepTBFV, a test-based approach to generate execution paths for a test case in a neural network and derive the precondition of the input to the neural network backwards by means of a custom post-condition. To our best knowledge, we are the first to apply the idea of TBFV to the field of verification and interpretation of neural networks.

Although we utilize the principle of TBFV to give the formal local interpretation of the neural network model, these interpretations exist in the form of linear constraints. However, DeepTBFV still has potential shortcomings. Firstly, the formal local interpretation in the form of linear constraints cannot be intuitively understood by users, and further parsing is needed to generate the importance of features. Second, the setting of post condition is an important factor that affects the effectiveness of interpretation, and there is no theoretical method to set the post condition.

In future work, we will focus on the following two parts:

- How to automatically generate post conditions suitable for DeepTBFV based on the output results of the instance is an unresolved issue.
- How to analyze the constraints generated by DeepTBFV so that the constraint results can help people intuitively understand the behavior of the neural network model.

Chapter 5

Conclusion

The main research objective of this dissertation is how to improve the reliability of neural network based systems by both means of testing and verification. Due to the characteristics of neural networks, such as data-driven, empirical construction, and poor interpretability, many errors or adversarial examples can occur in the process of constructing neural networks, making the construction of neural network-based systems interrupted or even failed. These issues, based on the construction process of neural networks, can be divided into reliability risks that arise during model training and reliability risks that occur after model training. Over the years, significant progress has been made in the areas of testing and verification of neural network models, respectively. However, research is still relatively scant on how to ensure the reliability of models during the training process and how to combine testing and verification to guarantee the reliability of models.

To be more specific, we explore three problems in two areas: reliability assurance during model training and reliability assurance after training, one of which belongs to reliability assurance during model training and two of which belong to reliability assurance after model training. In response to these three specific issues, this dissertation achieves the following innovative results:

In Chapter 2, this dissertation proposes a test-based method for assessing GPU memory usage, aimed at enhancing the reliability of model training. It generates training data that can be used to train a model for predicting GPU memory usage through testing methods, and uses the trained GPU memory usage prediction model to assess the potential GPU memory consumption of the neural network model to be trained. Ultimately, this aims to avoid the problem of OOM during the training process of the neural network model. The case study indicate that this method can identify the functional relationship between GPU memory consumption and specified features.

In chapter 3, this dissertation introduces a method that leverages the concept of TBFV to accelerate neural network verification, aiming to improve the reliability of the model post-training. First, test cases are generated under a given formal specification and the execution paths of the neural network under the corresponding specification are recorded. Finally, the size of the neural network is reduced by pruning to achieve the purpose of accelerating the verification of the neural network. Experiments and theoretical analysis show that the test-based method can effectively identify the inactive neurons in the neural network, and the reduction of the neural network size can effectively improve the verification speed of the neural network.

In Chapter 4, this dissertation proposes a theoretical framework using the principles of TBFV to explain neural network behavior, aiming to enhance the local interpretability of the trained model. Firstly, the execution path of the sample being interpreted in the neural network is recorded. Then, this execution path is transformed into linear constraints, and the constraints for each neuron in the input layer of the neural network model are solved. Finally, we propose a theoretical framework for interpreting the neural network model using these constraints. A case study shows that this theoretical method has the capability to interpret neural network models.

Although this dissertat contributes to the reliability of neural network-based systems, these methods still have limitations and threats. In Chapter 2, the prediction results of TBEM are influenced by the selected GPU memory usage features, which is also the main aspect threatening TBEM. Secondly, the pre-trained neural networks and the post-training neural networks mentioned in Chapters 3 and 4 are fully connected neural networks. Although fully connected layers can be converted into various neural network layer structures such as convolutional layers, this does not guarantee that the methods proposed in Chapters 3 and 4 are applicable to convolutional neural networks, recurrent neural networks, or even graph neural networks. This limits the scalability of

NNTBFV and DeepTBFV.

Based on the contributions and limitations of the above descriptions, we believe that the methods of testing and verification in traditional software can still be adapted to achieve improved reliability of neural network-based systems. There are many issues that need to be further refined and researched in future work. In terms of ensuring reliability during the training process of the neural network model, although test-based methods have many advantages of black-box testing, further improving the accuracy of predicting GPU memory usage still requires additional integration of static analysis of the operational mechanisms of the neural network framework. In terms of ensuring the reliability of the trained model, the idea of combining testing and verification methods has a lot of research space and value in guaranteeing the reliability of pre-trained models or trained neural networks. First, testing has the disadvantages of not being able to prove that the system is completely bug-free, limited coverage, etc., which happen to be the advantages of formal verification. Similarly, formal verification has the disadvantages of ignoring user requirements and relying on formal specification, which are the advantages of testing. Therefore, how to better combine the advantages of testing and verification to improve the reliability and interpretability of neural network-based systems before deployment is a future research direction. For example, how to determine the order of alternation between testing and verification, that is, whether to test first then verify, or to verify first then test.

To sum up, we believe that testing and verification can enhance the reliability of neural network-based systems and in particular, combining testing and verification can efficiently ensure the reliability and interpretability of neural network algorithms. In future research, the combination of testing and verification also brings new possibilities for lightweight, interpretable neural networks.

Appendix

Formal Specification of The ACAS Xu Using SOFL

We use Structured Object-Oriented Formal Language (SOFL) to define the formal specification of ACAS Xu. On the one hand, SOFL is a familiar formal language, and on the other hand, the three-step specification provided by SOFL is well suited to describe the formal specification for neural network verification[1].

In SOFL, a process represents a transformation from input to output and can describe any Operation in the programming language. Its functional behavior can be defined by a formal specification with pre-conditions and post-conditions. Let NN denote a neural network, then we use *pre* and *post* to denote the pre-condition and post-condition of the neural network, respectively. Furthermore, We can describe the neural network verification problem as follows: if the input of the NN satisfies the constraints in *pre*, then the output of the NN should satisfy the constraints in *post*. In section 3.5.1.1, we introduce the 45 neural networks of ACAS Xu. In the following, each formal specification, corresponds to a different neural network in ACAS Xu.

```
process $N_{ij}$, $1 \leq i \leq 5$ and $1 \leq j \leq 9$
pre $55047.691 \leq \rho \leq 60760$ and $1145 \leq v_{own} \leq 1200$ and $0 \leq v_{int} \leq 60$
post $COC \leq 1500$
end_process
```

Formal Specification : ϕ_1

```
process $N_{ij}$, $1 \leq i \leq 5$ and $1 \leq j \leq 9$ except $N_{42}$ and $N_{53}$
pre $55047.691 \leq \rho \leq 60760$ and $1145 \leq v_{own} \leq 1200$ and $0 \leq v_{int} \leq 60$
post $COC$ is not the maximal score.
end_process
```

Formal Specification : ϕ_2

```
process $N_{ij}$, $1 \leq i \leq 5$ and $1 \leq j \leq 9$ except $N_{17}$ and $N_{18}$ and $N_{19}$
```

[1]http://www.vnnlib.org/

```
pre $1500 \leq \rho \leq 1800$ and $-0.06 \leq \theta \leq 0.06$ and $\psi \geq 3.10$ and
    $1200 \geq v_{own} \geq 980$ and $1200 \geq v_{intr} \geq 960$
post $COC$ is not the minimal score.
end_process
```

Formal Specification : ϕ_3

```
process $N_{ij}$, $1 \leq i \leq 5$ and $1 \leq j \leq 9$ except $N_{17}$ and $N_{18}$ and $N_{19}$
pre $1500 \leq \rho \leq 1800$ and $-0.06 \leq \theta \leq 0.06$ and $\psi = 0$ and
    $1200 \geq v_{own} \geq 1000$ and $700 \leq v_{intr} \leq 800$
post $COC$ is not the minimal score.
end_process
```

Formal Specification : ϕ_4

```
process $N_{11}$
pre $250 \leq \rho \leq 400$ and $0.2 \leq \theta \leq 0.4$ and
    $-3.141592 \leq \psi \leq -3.141592 + 0.005$ and $100 \leq v_{own} \leq 400$ and
    $0 \leq v_{intr} \leq 400$
post $strongright$ is the minimal score
end_process
```

Formal Specification : ϕ_5

```
process $N_{11}$
pre $12000 \leq \rho \leq 62000$ and
    ($0.7 \leq \theta \leq 3.141592$ or $-3.141592 \leq \theta \leq -07$) and
    $-3.141592 \leq \psi \leq -3.141592 + 0.005$ and
    $100 \leq v_{own} \leq 1200$ and
    $0 \leq v_{intr} \leq 1200$
post $COC$ is the minimal score
end_process
```

Formal Specification : ϕ_6

```
process $N_{19}$
pre $0 \leq \rho \leq 60760$ and
    $-3.141592 \leq \theta \leq 3.141592$ and $-3.141592 \leq \psi \leq 3.141592$ and
    $100 \leq v_{own} \leq 1200$ and $0 \leq v_{intr} \leq 1200$
post $strongright$ and $strongleft$ are never the minimal scores.
end_process
```

Formal Specification : ϕ_7

```
process $N_{29}$
pre $0 \leq \rho \leq 60760$ and $-3.141592 \leq \theta \leq -075 \times 3.141592$ and
    $-0.1 \leq \psi \leq 0.1$ and $600 \leq v_{own} \leq 1200$ and $600 \leq v_{intr} \leq 1200$
post $weakleft$ is minimal or $COC$ is minimal
end_process
```

Formal Specification : ϕ_8

```
process $N_{33}$
```

```
pre $2000 \leq \rho \leq 7000$ and $0.7 \leq \theta \leq 3.141592$ and
    $-3.141592 \leq \psi \leq -3.141592 + 0.01$ and $900 \leq v_{own} \leq 1200$ and
    $600 \leq v_{intr} \leq 1200$
post $strongleft$ is minimal
end_process
```

Formal Specification : ϕ_9

```
process $N_{45}$
pre $36000 \leq \rho \leq 60760$ and $0.7 \leq \theta \leq 3.141592$ and
    $-3.141592 \leq \psi \leq -3.141592 + 0.01$ and $900 \leq v_{own} \leq 1200$ and
    $600 \leq v_{intr} \leq 1200$
post $COC$ is minimal
end_process
```

Formal Specification : ϕ_{10}

Bibliography

[1] Basemah Alshemali and Jugal Kalita. Improving the reliability of deep neural networks in nlp: A review. *Knowledge-Based Systems*, 191:105210, 2020.

[2] David Alvarez-Melis and Tommi S. Jaakkola. Towards robust interpretability with self-explaining neural networks. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 7786–7795, 2018.

[3] Alejandro Barredo Arrieta, Natalia Díaz Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador García, Sergio Gil-Lopez, Daniel Molina, Richard Benjamins, Raja Chatila, and Francisco Herrera. Explainable artificial intelligence (XAI): concepts, taxonomies, opportunities and challenges toward responsible AI. *Information fusion*, 58:82–115, 2020.

[4] Sebastian Bach, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PloS one*, 10(7):e0130140, 2015.

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[6] Pablo Barceló, Mikaël Monet, Jorge Pérez, and Bernardo Subercaseaux. Model interpretability through the lens of computational complexity. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[7] Shahaf Bassan and Guy Katz. Towards formal xai: Formally approximate minimal explanations of neural networks. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 187–207. Springer, 2023.

[8] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? *Proceedings of machine learning and systems*, 2:129–146, 2020.

[9] Rudy R Bunel, Ilker Turkaslan, Philip Torr, Pushmeet Kohli, and Pawan K Mudigonda. A unified view of piecewise linear neural network verification. *Advances in Neural Information Processing Systems*, 31, 2018.

[10] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Kaijie Zhu, Hao Chen, Linyi Yang, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. A survey on evaluation of large language models. *arXiv preprint arXiv:2307.03109*, 2023.

[11] Simin Chen, Soroush Bateni, Sampath Grandhi, Xiaodi Li, Cong Liu, and Wei Yang. DENAS: automated rule generation by knowledge extraction from neural networks. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 813–825. ACM, 2020.

[12] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[13] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. Dysy: Dynamic symbolic execution for invariant inference. In *Proceedings of the 30th international conference on Software engineering*, pages 281–290, 2008.

[14] Bang Di, Jianhua Sun, Dong Li, Hao Chen, and Zhe Quan. Gmod: a dynamic gpu memory overflow detector. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–13, 2018.

[15] Francesco Di Giovanni, Lorenzo Giusti, Federico Barbero, Giulia Luise, Pietro Lio, and Michael M Bronstein. On over-squashing in message passing neural networks: The impact of width, depth, and topology. In *International Conference on Machine Learning*, pages 7865–7885. PMLR, 2023.

[16] Tommaso Dreossi, Shromona Ghosh, Alberto Sangiovanni-Vincentelli, and Sanjit A Seshia. A formalization of robustness for deep neural networks. *arXiv preprint arXiv:1903.10033*, 2019.

[17] Mengnan Du, Ninghao Liu, Qingquan Song, and Xia Hu. Towards explanation of dnn-based prediction with guided feature inversion. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1358–1367, 2018.

[18] Rüdiger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In Deepak D'Souza and K. Narayan Kumar, editors, *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings*, volume 10482 of *Lecture Notes in Computer Science*, pages 269–286. Springer, 2017.

[19] Luciano Floridi and Massimo Chiriatti. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30(4):681–694, 2020.

[20] Ruth Fong, Mandela Patrick, and Andrea Vedaldi. Understanding deep networks via extremal perturbations and smooth masks. In *Proceedings of*

the IEEE/CVF international conference on computer vision, pages 2950–2958, 2019.

[21] Ruth C Fong and Andrea Vedaldi. Interpretable explanations of black boxes by meaningful perturbation. In *Proceedings of the IEEE international conference on computer vision*, pages 3429–3437, 2017.

[22] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9,2019*. OpenReview.net, 2019.

[23] Yanjie Gao, Yu Liu, Hongyu Zhang, Zhengxian Li, Yonghao Zhu, Haoxiang Lin, and Mao Yang. Estimating gpu memory consumption of deep learning models. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1342–1352, 2020.

[24] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, May, 2015, Conference Track Proceedings*, CA, USA, 2015.

[25] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 37(3):362–386, 2020.

[26] Guan Gui, Fan Liu, Jinlong Sun, Jie Yang, Ziqi Zhou, and Dongxu Zhao. Flight delay prediction based on aviation big data and machine learning. *IEEE Transactions on Vehicular Technology*, 69(1):140–150, 2019.

[27] Dario Guidotti, Francesco Leofante, Luca Pulina, and Armando Tacchella. Verification of neural networks: Enhancing scalability through pruning. In Giuseppe De Giacomo, Alejandro Catalá, Bistra Dilkina, Michela Milano, Senén Barro, Alberto Bugarín, and Jérôme Lang, editors, *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020*

- Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020), volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 2505–2512. IOS Press, 2020.

[28] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. Audee: Automated testing for deep learning frameworks. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 486–498. IEEE, 2020.

[29] Xingwu Guo, Ziwei Zhou, Yueling Zhang, Guy Katz, and Min Zhang. Occrob: Efficient smt-based occlusion robustness verification of deep neural networks. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 208–226. Springer, 2023.

[30] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *CoRR*, abs/1607.03250, 2016.

[31] Anzhong Huang, Lening Qiu, and Zheng Li. Applying deep learning method in tvp-var model under systematic financial risk monitoring and early warning. *Journal of Computational and Applied Mathematics*, 382:113065, 2021.

[32] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1110–1121, 2020.

[33] Mark Ibrahim, Melissa Louie, Ceena Modarres, and John Paisley. Global explanations of neural networks: Mapping the landscape of predictions. In *Proceedings of the 2019 AAAI/ACM Conference on AI, Ethics, and Society*, pages 279–287, 2019.

[34] Alexey Ignatiev, Nina Narodytska, and João Marques-Silva. Abduction-based explanations for machine learning models. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019,*

The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019, pages 1511–1519. AAAI Press, 2019.

[35] Ievgen Ivanov and Mykola S Nikitchenko. On the sequence rule for the floyd-hoare logic with partial pre-and post-conditions. In *ICTERI Workshops*, pages 716–724, 2018.

[36] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex an efficient smt solver for verifying deep neural networks. In *Computer Aided Verification: 29th International Conference, CAV , Proceedings, Part I 30*, pages 97–117, Heidelberg, Germany, July 2017.

[37] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[38] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial machine learning at scale. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[39] Ori Lahav and Guy Katz. Pruning and slicing neural networks using formal verification. In *2021 Formal Methods in Computer Aided Design (FMCAD)*, pages 183–192, New Haven, CT, USA, 2021.

[40] LJUBOMIR Lazić. Use of orthogonal arrays and design of experiments via taguchi methods in software testing. *Recent Advances in Applied and Theoretical Mathematics*, pages 256–67, 2013.

[41] Benjamin Letham, Cynthia Rudin, Tyler H. McCormick, and David Madigan. Interpretable classifiers using rules and bayesian analysis: Building a better stroke prediction model. *CoRR*, abs/1511.01644, 2015.

[42] Yu Liang, Siguang Li, Chungang Yan, Maozhen Li, and Changjun Jiang. Explaining the black-box model: A survey of local interpretation methods for deep neural networks. *Neurocomputing*, 419:168–182, 2021.

[43] Ai Liu and Shaoying Liu. Enhancing the capability of testing-based formal verification by handling operations in software packages. *IEEE Transactions on Software Engineering*, 49(1):304–324, 2023.

[44] Changliu Liu, Tomer Arnon, Christopher Lazarus, Christopher Strong, Clark Barrett, Mykel J Kochenderfer, et al. Algorithms for verifying deep neural networks. *Foundations and Trends® in Optimization*, 4(3-4):244–404, 2021.

[45] Haiyi Liu, Shaoying Liu, and Ai Liu. Testing-based gpu-memory consumption estimation for deep learning. In *Software Engineering Symposium 2021 Proceedings*, volume 2021, pages 196–199, aug 2021.

[46] Liangkai Liu, Sidi Lu, Ren Zhong, Baofu Wu, Yongtao Yao, Qingyang Zhang, and Weisong Shi. Computing systems for autonomous driving: State of the art and challenges. *IEEE Internet of Things Journal*, 8(8):6469–6486, 2020.

[47] Shaoying Liu. Testing-based formal verification for theorems and its application in software specification verification. In *International Conference on Tests and Proofs*, pages 112–129, Vienna, Austria, July 2016.

[48] Shaoying Liu and Shin Nakajima. A “vibration” method for automatically generating test cases based on formal specifications. In *2011 18th Asia-Pacific Software Engineering Conference*, pages 73–80. IEEE, 2011.

[49] Shaoying Liu and Shin Nakajima. Automatic test case and test oracle generation based on functional scenarios in formal specifications for conformance testing. *IEEE Transactions on Software Engineering*, 2020.

[50] Shaoying Liu, A Jefferson Offutt, Chris Ho-Stuart, Yong Sun, and Mitsuru Ohba. Sofl: A formal engineering methodology for industrial applications. *IEEE Transactions on Software Engineering*, 24(1):24–45, 1998.

[51] Alessio Lomuscio and Lalit Maganti. An approach to reachability analysis for feed-forward relu neural networks. *CoRR*, abs/1706.07351, 2017.

[52] Scott M. Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In Isabelle Guyon, Ulrike von Luxburg, Samy Ben-

gio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 4765–4774, 2017.

[53] Wei Ma and Jun Lu. An equivalence of fully connected layer and convolutional layer. *CoRR*, abs/1712.01252, 2017.

[54] Aravindh Mahendran and Andrea Vedaldi. Understanding deep image representations by inverting them. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5188–5196, 2015.

[55] Mark Huasong Meng, Guangdong Bai, Sin Gee Teo, Zhe Hou, Yan Xiao, Yun Lin, and Jin Song Dong. Adversarial robustness of deep neural networks: A survey from a formal verification perspective. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2022.

[56] Sparsh Mittal and Shraiysh Vaishay. A survey of techniques for optimizing deep learning on gpus. *Journal of Systems Architecture*, 99:101635, 2019.

[57] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2574–2582, Las Vegas, NV, USA, June 2016.

[58] Pramod Kaushik Mudrakarta, Ankur Taly, Mukund Sundararajan, and Kedar Dhamdhere. Did the model understand the question? In Iryna Gurevych and Yusuke Miyao, editors, *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, pages 1896–1906. Association for Computational Linguistics, 2018.

[59] W James Murdoch and Arthur Szlam. Automatic rule extraction from long short term memory networks. *arXiv preprint arXiv:1702.02540*, 2017.

[60] Christian Murphy, Gail E Kaiser, and Marta Arias. An approach to software testing of machine learning applications. Technical Report CUCS-014-07, Department of Computer Science, Columbia University, New York, 2007.

[61] Christian Murphy, Gail E Kaiser, and Lifeng Hu. Properties of machine learning applications for use in metamorphic testing. Technical Report CUCS-011-08, Department of Computer Science, Columbia University, New York, 2008.

[62] Duc-Anh Nguyen, Kha Do Minh, Minh Le Nguyen, and Pham Ngoc Hung. A symbolic execution-based method to perform untargeted attack on feed-forward neural networks. *Automated Software Engineering*, 29(2):46, 2022.

[63] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023.

[64] Jonas Oppenlaender. The creativity of text-to-image generation. In *Proceedings of the 25th International Academic Mindtrek Conference*, pages 192–202, Nagoya, Japan, Nov 2022.

[65] Andrei Paleyes, Raoul-Gabriel Urma, and Neil D Lawrence. Challenges in deploying machine learning: a survey of case studies. *ACM Computing Surveys*, 55(6):1–29, 2022.

[66] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *Proceedings of the 2016 IEEE European Symposium on Security and Privacy*, pages 372–387, Saarbrucken, Germany, March 2016.

[67] Matteo Papini, Matteo Pirotta, and Marcello Restelli. Adaptive batch size for safe policy gradients. In *Advances in Neural Information Processing Systems*, volume 30, page 3594–3603. Curran Associates, Inc., 2017.

[68] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: automated whitebox testing of deep learning systems. *Communications of the ACM*, 62(11):137–145, 2019.

[69] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 891–905, 2020.

[70] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. Cradle: cross-backend validation to detect and localize bugs in deep learning libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1027–1038. IEEE, 2019.

[71] Vaughan R Pratt. Semantical considerations on floyd-hoare logic. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*, pages 109–121, 1730 Massachusetts Ave., NW Washington, DCUnited States, October 1976.

[72] Pavlo M Radiuk et al. Impact of training set batch size on the performance of convolutional neural networks for diverse datasets. *Information Technology and Management Science*, 20(1):20–24, 2017.

[73] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[74] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. Nothing else matters: Model-agnostic explanations by identifying prediction invariance. *CoRR*, abs/1611.05817, 2016.

[75] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you? " explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2016.

[76] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Anchors: High-precision model-agnostic explanations. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[77] Rabia Saleem, Bo Yuan, Fatih Kurugollu, Ashiq Anjum, and Lu Liu. Explaining deep neural networks: A survey on the global interpretation methods. *Neurocomputing*, 513:165–180, 2022.

[78] Gregor P. J. Schmitz, Chris Aldrich, and F. S. Gouws. ANN-DT: an algorithm for extraction of decision trees from artificial neural networks. *IEEE Transactions on Neural Networks*, 10(6):1392–1401, 1999.

[79] Sanjit A Seshia, Ankush Desai, Tommaso Dreossi, Daniel J Fremont, Shromona Ghosh, Edward Kim, Sumukh Shivakumar, Marcell Vazquez-Chanlatte, and Xiangyu Yue. Formal specification for deep neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 20–34, Cham, 2018. Springer.

[80] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. Learning important features through propagating activation differences. In *International conference on machine learning*, pages 3145–3153. PMLR, 2017.

[81] David Shriver, Sebastian G. Elbaum, and Matthew B. Dwyer. DNNV: A framework for deep neural network verification. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 137–150. Springer, 2021.

[82] K Simonyan, A Vedaldi, and A Zisserman. Deep inside convolutional networks: visualising image classification models and saliency maps. In *Proceedings of the International Conference on Learning Representations (ICLR)*. ICLR, 2014.

[83] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[84] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

[85] Suraj Srinivas and François Fleuret. Full-gradient representation for neural network visualization. *Advances in neural information processing systems*, 32, 2019.

[86] Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. Deepconcolic: testing and debugging deep neural networks. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 111–114. IEEE / ACM, 2019.

[87] Ismail A. Taha and Joydeep Ghosh. Symbolic interpretation of artificial neural networks. *IEEE Transactions on knowledge and data engineering*, 11(3):448–463, 1999.

[88] Emil Talpes, Debjit Das Sarma, Ganesh Venkataramanan, Peter Bannon, Bill McGee, Benjamin Floering, Ankit Jalote, Christopher Hsiong, Sahil Arora, Atchyuth Gorti, and Gagandeep S. Sachdev. Compute solution for tesla's full self-driving computer. *IEEE Micro*, 40(2):25–35, 2020.

[89] Vincent Tjeng, Kai Yuanqing Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[90] Erico Tjoa and Cuntai Guan. A survey on explainable artificial intelligence (XAI): toward medical XAI. *IEEE transactions on neural networks and learning systems*, 32(11):4793–4813, 2021.

[91] Hiroshi Tsukimoto. Extracting rules from trained neural networks. *IEEE transactions on neural networks and learning systems*, 11(2):377–389, 2000.

[92] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[93] Marian Vesely. Computer curve fitting of polynomials. *CoordinatedScience Laboratory Report R-595, University of Illinois*, 1972.

[94] Jorg Wagner, Jan Mathias Kohler, Tobias Gindele, Leon Hetzel, Jakob Thaddaus Wiedemer, and Sven Behnke. Interpretable and fine-grained visual explanations for convolutional neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9097–9107, 2019.

[95] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 41–53, 2018.

[96] Rong Wang and Shaoying Liu. TBFV-SE: testing-based formal verification with symbolic execution. In *2018 IEEE International Conference on Software Quality, Reliability and Security, QRS 2018, July 16-20, 2018*, pages 59–66, Lisbon, Portugal, 2018. IEEE.

[97] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient formal safety analysis of neural networks. *CoRR*, abs/1809.08098, 2018.

[98] Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J Zico Kolter. Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. *Advances in Neural Information Processing Systems*, 34:29909–29921, 2021.

[99] Hao Wu. Application of orthogonal experimental design for the automatic software testing. In *Applied mechanics and materials*, volume 347, pages 812–818. Trans Tech Publ, 2013.

[100] Mike Wu, Michael C. Hughes, Sonali Parbhoo, Maurizio Zazzi, Volker Roth, and Finale Doshi-Velez. Beyond sparsity: Tree regularization of deep models for interpretability. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Ed-*

ucational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018, pages 1670–1678. AAAI Press, 2018.

[101] Mike Wu, Sonali Parbhoo, Michael C. Hughes, Ryan Kindle, Leo A. Celi, Maurizio Zazzi, Volker Roth, and Finale Doshi-Velez. Regional tree regularization for interpretability in deep neural networks. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 6413–6421. AAAI Press, 2020.

[102] Kaidi Xu, Huan Zhang, Shiqi Wang, Yihan Wang, Suman Jana, Xue Lin, and Cho-Jui Hsieh. Fast and complete: Enabling complete neural network verification with rapid and massively parallel incomplete verifiers. In *International Conference on Learning Representation (ICLR)*, Vienna, Austria, Jan 2021.

[103] Chih-Kuan Yeh, Joon Kim, Ian En-Hsu Yen, and Pradeep K Ravikumar. Representer point selection for explaining deep neural networks. *Advances in neural information processing systems*, 31, 2018.

[104] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. Adversarial examples: Attacks and defenses for deep learning. *IEEE transactions on neural networks and learning systems*, 30(9):2805–2824, 2019.

[105] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I 13*, pages 818–833. Springer, 2014.

[106] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. An empirical study on program failures of deep learning jobs. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1159–1170. IEEE, 2020.

Publication List of the Author

Referred Journals

[J-1] Haiyi Liu, Shaoying Liu, Guangquan Xu, Ai Liu, and Dingbang Fang. NNTBFV: Simplifying and Verifying Neural Networks using Testing-Based Formal Verification. International Journal of Software Engineering and Knowledge Engineering, 34(02):273–300, 2024.

[J-2] Haiyi Liu, Shaoying Liu, Chenglong Wen, and W. Eric Wong. "TBEM: Testing-Based GPU-Memory Consumption Estimation for Deep Learning." IEEE Access, 10:39674-39680, 2022.

Referred Conferences

[J-3] Haiyi Liu, Shaoying Liu, Ai Liu, Dingbang Fang, and Guangquan Xu. "Verifying and Improving Neural Networks Using Testing-Based Formal Verification." In International Workshop on Structured Object-Oriented Formal Language and Method, pp. 126-141. Cham: Springer International Publishing, 2022.

[J-4] Haiyi Liu, Shaoying Liu, Guangquan Xu, Ai Liu, Yujun Dai. "Utilizing Testing-Based Formal Verification in Neural Networks: A Theoretical Approach." In 2023 13th International Conference on Software Technology and Engineering (ICSTE), pp.151-155. IEEE, 2023.