Towards Robust Neural Networks: Evaluation and Construction

Jingyue Lu

St Peter's College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy

Michaelmas 2021

Abstract

With their supreme performance in dealing with a large amount of data, neural networks have significantly benefited our lives in many aspects, from home assistants to autonomous driving cars. However, it is found that neural networks are brittle. By slightly perturbing the inputs in a way imperceptible to humans, neural networks can barely make any correct predictions. This seriously limits their applications in safety-critical areas, such as health care and finance. In this thesis, we study robust neural networks in the hope to facilitate a wider and more reliable use of neural networks. Specifically, we focus on evaluating and training robust neural networks.

We first consider robustness evaluation. A common approach for assessing the robustness of a neural network is through formal verification, which is often computationally expensive. We make several contributions to speed up the process. In brief, we adopt the idea that a majority of verification methods can be reformulated under a unified branch and bound framework. By dealing with the unified framework directly, we propose high-level improvements, including heuristics and a learned framework, for the branching and bounding components. Furthermore, we introduce new datasets to enable comprehensive comparison analyses of our methods with other existing ones.

In terms of constructing robust neural networks, we develop a new algorithm for efficient robust training. Many popular robust training methods rely on strong adversaries, which are costly to compute when the model complexity and input dimension are high. We design a novel framework that allows a more effective use of adversaries. As a result, to achieve similar performance, cheap and weak adversaries can be used instead. Based on the framework, we introduce the algorithm ATLAS. We demonstrate the effectiveness and efficiency of ATLAS by showing its outstanding performance on several standard datasets.

Towards Robust Neural Networks: Evaluation and Construction

Jingyue Lu

St Peter's College

University of Oxford

Supervised by Prof M. Pawan Kumar

A thesis submitted for the degree of

Doctor of Philosophy

Michaelmas 2021

Abstract

With their supreme performance in dealing with a large amount of data, neural networks have significantly benefited our lives in many aspects, from home assistants to autonomous driving cars. However, it is found that neural networks are brittle. By slightly perturbing the inputs in a way imperceptible to humans, neural networks can barely make any correct predictions. This seriously limits their applications in safety-critical areas, such as health care and finance. In this thesis, we study robust neural networks in the hope to facilitate a wider and more reliable use of neural networks. Specifically, we focus on evaluating and training robust neural networks.

We first consider robustness evaluation. A common approach for assessing the robustness of a neural network is through formal verification, which is often computationally expensive. We make several contributions to speed up the process. In brief, we adopt the idea that a majority of verification methods can be reformulated under a unified branch and bound framework. By dealing with the unified framework directly, we propose high-level improvements, including heuristics and a learned framework, for the branching and bounding components. Furthermore, we introduce new datasets to enable comprehensive comparison analyses of our methods with other existing ones.

In terms of constructing robust neural networks, we develop a new algorithm for efficient robust training. Many popular robust training methods rely on strong adversaries, which are costly to compute when the model complexity and input dimension are high. We design a novel framework that allows a more effective use of adversaries. As a result, to achieve similar performance, cheap and weak adversaries can be used instead. Based on the framework, we introduce the algorithm ATLAS. We demonstrate the effectiveness and efficiency of ATLAS by showing its outstanding performance on several standard datasets.

Statement of Originality

I hereby declare that, except where made explicitly clear, the contents of this dissertation are my own original work, and to the best of my knowledge do not contain materials submitted in whole, or in part, for consideration for any other degree or qualification at the University of Oxford or any other academic or professional institution.

Jingyue Lu
Oct 30, 2021

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor Prof. M. Pawan Kumar for his continuous support through my DPhil studies. The completion of the thesis would not have been possible without his thought-provoking comments, constructive advice, enthusiasm and, especially, his encouragement and patience during the challenging time of the pandemic COVID-19.

I would like to extend my sincere thanks to all members of the Oval group, particularly, Rudy, Leo, Florian, Alasdair, Alessandro and Harkirat, for all the interesting and insightful discussions we had during reading groups, the help and support in tackling a wide variety of issues that came up along the research journey and the good fun we had over lunch breaks. Thanks also to Harkirat and Francisco for proofreading the thesis.

Furthermore, I am deeply indebted to my parents. The research in this thesis would have taken far longer without their unrelenting support and their profound belief in my abilities.

Finally, I gratefully acknowledge the financial support that I received through the Clarendon Fund and Statistical Science (EPSRC and MRC Centre for Doctoral Training) scholarship.

Contents

1 Introduction

Contents

1.1 Motivation

In recent years, deep learning has gained enormous popularity. It has demonstrated state-of-the-art performance in various tasks, including computer vision [70, 92] (e.g., facial recognition systems), speech recognition [24, 118] (e.g., Apple Siri, Amazon Alexa), board game programs [81, 91] (e.g., AlphaGo) and medical image analyses [59, 74] (e.g., lesion detection). Several reasons have contributed to the success of deep learning. Firstly, the outstanding performance of deep learning in dealing with a large amount of data makes it the perfect tool for the 'Big Data Era' we are in. Secondly, unlike traditional machine learning, which requires careful feature engineering, deep learning algorithms solve a problem end to end. This attribute enables deep learning techniques to effectively deal with complex issues,

where the domain understanding for feature extraction is limited. Finally, although deep learning often requires training large neural networks on big datasets, the total training time has significantly reduced with the advent of GPU computing. The highly parallel structure of GPUs has made deep learning much more available.

However, despite their outstanding performance on diverse tasks, deep neural networks are found to be vulnerable to adversarial examples [36, 87]. Adversarial

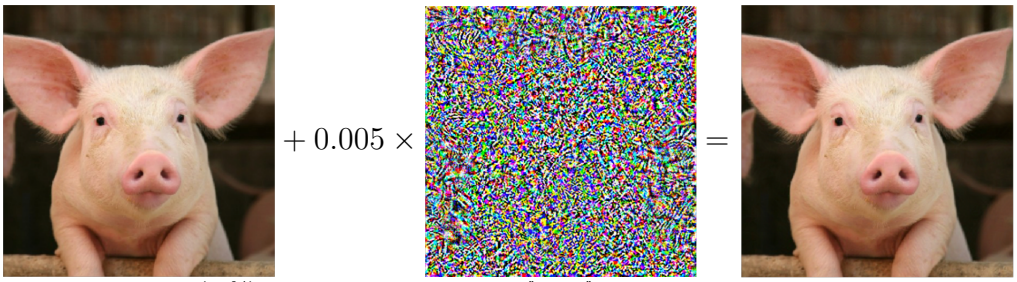


Figure 1.1: An adversarial example created by adding noise to the original image. The trained neural network (NN) that originally outputs the correct label Pig with 91% confidence, fails on the perturbed image by predicting it as an airliner with 99% confidence. In stark contrast to the neural network, to the human eye, the left and the right images look identical. Images are adopted from [56].

examples are inputs that are specifically designed to confuse networks. They are normally generated by adding to an input a small perturbation imperceptible to the human eye, such that a well-trained network no longer makes a correct prediction on the perturbed input. One such example is provided in Figure 1.1. With more companies relying on deep learning for analysis and decision-making processes, the brittleness of neural networks can have costly consequences, especially in safety-critical settings such as autonomous driving, health care and finance.

For the sake of illustration, we give two possible scenarios. Autonomous driving with its promised high-speed, efficiency and reduced environmental impacts could be an ideal replacement for traditional driving. However, one of the biggest issues for self-driving cars is safety. When on the road, self-driving cars employ deep neural networks to process data from sensors and cameras and make driving decisions accordingly. It is of extreme importance for networks to correctly read signs and to

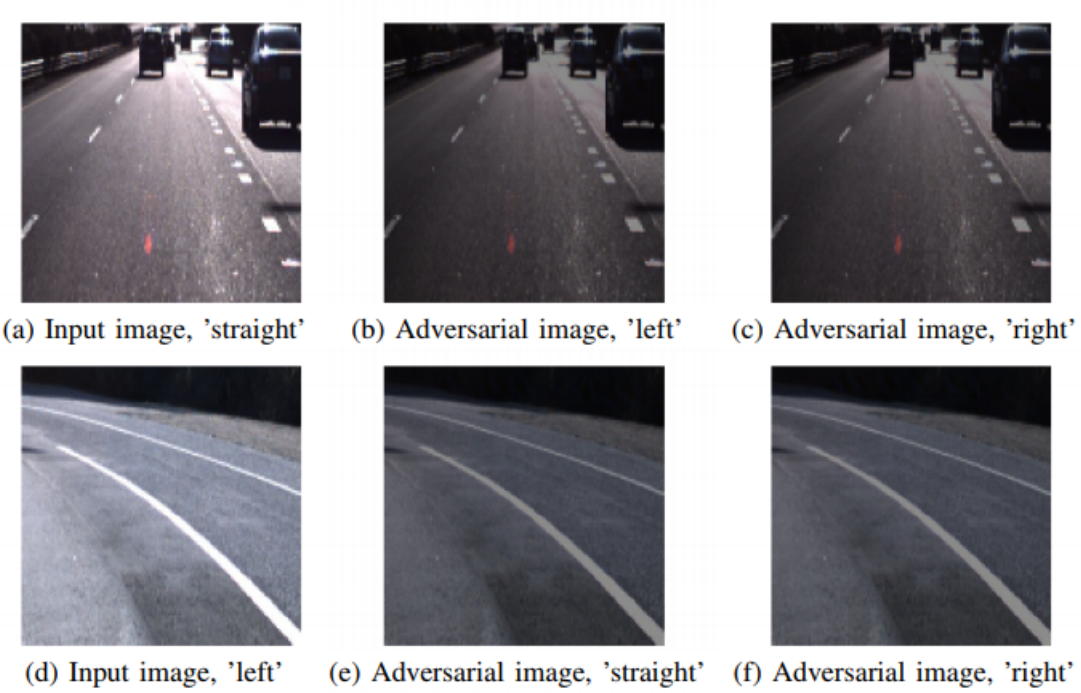


Figure 1.2: Input images and adversarial images with their corresponding outputs from a trained classification model. On the first row, the network gives the correct output "straight" for the original input road image. However, by adding small and non-essential changes, the network can be fooled to give any other output labels. A similar example for an input image “left" is provided in the second row. Those images are adopted from [17].

identify pedestrians and other objects. If the employed networks are not robust, attackers can modify the behaviour of an autonomous driving car, resulting in severe car accidents. Some of the effects of adversarial attacks have been studied in [17], as shown in Figure 1.2, by adding a small perturbation to camera images, the route of a self-driving car can be completely changed in a disastrous way. In addition, adversarial attacks could lead to financial losses. At the moment, many people rely on smart assistants such as Apple Siri, Amazon Alexa for daily tasks and have often stored their bank details in them. Users give voice commands by talking to these assistants. To recognize an audio wave and perform a speech-to-text transform, neural networks are applied. However, Carlini and Wagner [14] show that for any audio waveform, they can produce a waveform that is over 99.9% similar but transcribes as any phrase of their choice, see Figure 1.3. With this form of attack, it is possible for a hacker to steal people's money by manipulating voice commands in a specific way. There are many other potential threats of deep

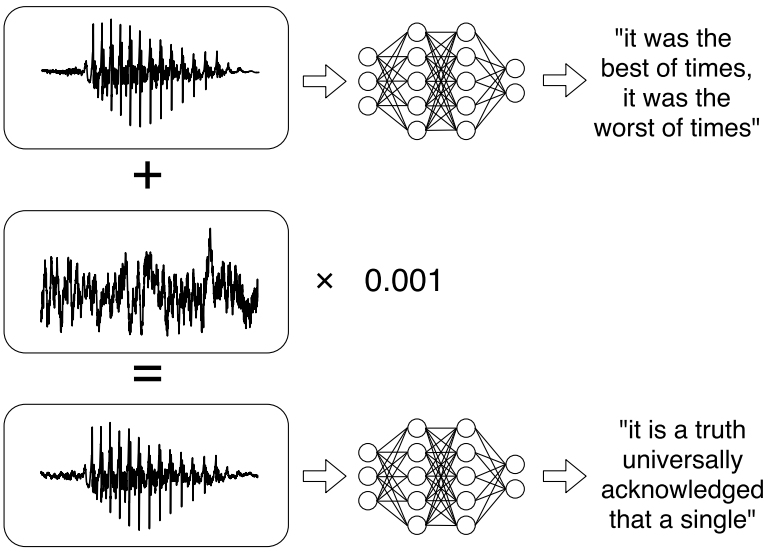


Figure 1.3: An example of targeted attack on speech-to-text. Given any waveform (top row), an arbitrary desired phrase (bottom row) can be transcribed by adding a small perturbation (middle row). The example is adopted from [14].

neural networks. Therefore, for wider and reliable applications of this powerful tool, it is crucial to understand robust neural networks.

In this thesis, we focus on evaluating and training robust neural networks. We develop techniques for examining the robustness of a trained network, so a more informed and reliable decision can be made using network outputs. We also propose strategies for training networks with improved robustness, in the hope to extend their applications to safety-critical areas.

1.2 Key Challenges

We identify some key challenges in studying robust neural networks. By discussing each one of them, we point out which problems we focus on to present a complete and overall picture of the thesis.

Defining Robustness The challenge of defining robustness mainly comes from the difficulty of characterizing perturbations. To illustrate the idea, we use image classification tasks, which are also the tasks considered in this thesis. Intuitively, we expect a state-of-the-art neural network that generalises well to be robust.

That is, the network should give the same consistent output for an image even when it is slightly perturbed, as small perturbations do not change the object category of an image in human eyes. As a result, to translate the intuitive understanding of robustness into a mathematical definition, we need to formally define the perturbations that are allowed. In other words, we need to precisely and rigorously define what we humans perceive to be visually similar. This is a daunting task and so far there have been no conclusive results. Previous works have mainly simplified the task by adopting a specific well-formulated perturbation. Popular choices include perturbations measured by ℓ_0 , ℓ_1, ℓ_2 and ℓ_∞ norms and spatial translations, for example rotation and translation. In this thesis, we carry out most of the studies by assuming the perturbation is measured by the ℓ_∞ norm.

Evaluating Robustness Once a perturbation measure is determined, we have a clear definition for robustness. To measure how robust a model is, we estimate the percentage of data points that the model can make the consistent correct prediction on a data point itself and all its allowed perturbed points. These data points should be from the same data distribution the model is trained on. Generally, each standard dataset we dealt with is already split into a training dataset and a test dataset. We directly treat the test dataset as a representation of the data distribution and measure the model performance on it. Under this setting, the difficulty for evaluating robustness lies in the fact that neural network functions are non-smooth and for each data point, there are uncountable allowed perturbed points. Thus, it is a non-trivial task to determine whether a model is robust at any given point. There are generally two ways to deal with this issue: adversarial attacks and formal verification. Adversarial attacks focus on identifying possible adversarial points within a given time limit. Because these methods are easily scalable and have preset time limits, they are frequently used for evaluating model robustness. However, results from adversarial attacks can provide a false sense of security. Models performing well on adversarial attacks may not actually be robust but simply appear so due to the "not-strong-enough" attacks used, as demonstrated

in [6]. Verification, on the other hand, formally proves whether all perturbed points can be predicted correctly by the model. We mention that formal verification is a general type of problem and evaluating a model's robustness can be seen as a special instance of it. To tackle the non-smooth issue of a network, one common strategy adopted by verification methods is to convexify the network and transform the verification problem into an optimization problem. The robustness decision is based on the solution of the optimization problem. For example, the neural network is robust on a domain if the minimum over the domain is above 0. A major drawback for this type of methods is that they are time-consuming and can only deal with networks that are much smaller than those used in real-life applications. We have particularly focused on formal verification in this work.

Improving Robustness The final challenge and the end goal is to develop strategies so that neural networks make robust predictions. Although, up till now, neural networks are still far from achieving human-like robustness performances, small steps have been made. Researchers have attempted the challenge from various perspectives. Some works [47, 61, 108, 119] have focused on pre-processing images. By recognizing and removing adversarial noise from an input image, networks are able to make consistent predictions. Other works study the training process. They identify possible causes of models being non-robust within the training process and come up with a specific solution for an identified cause. This type of work includes investigating the loss surface, such as developing loss functions for a more flattened loss surface [25, 63, 66, 72]; analysing the intermediate activation of neural networks, for instance, modifying activation functions or their outputs [73, 97, 105, 106]; examining the architecture of neural networks, for example, assessing the effects of skip connections and batch normalisation in defending adversarial examples [32, 102]. Furthermore, it has been shown in [78] that due to the sample complexity, robust learning has increased difficulty than standard learning in generalising its training results to unseen datasets. As a result, several works have been dedicated to narrowing the generalisation gap of robust learning for improved robust performance

[31, 78, 113, 115]. Finally, a few works have proposed ways to refine the robust learning process, such as speeding up the robust training [79, 101, 114, 117].

1.3 Thesis Outline

For a clear illustration, we provide a thesis outline below. Chapter 2 contains preliminary materials. Chapter 3 and Chapter 4 focus on robustness evaluation by developing verification methods while Chapter 5 centers on robustness construction by designing robust training strategies.

Chapter 2 In this chapter, we present basic mathematical notations and preliminary materials that will be used throughout the thesis. We provide a general definition for robustness and the specific one adopted in this thesis. Then, based on the definition, we give mathematical formulations for evaluating model robustness. We show the exact mathematical problems that attack methods and verification methods try to solve, respectively. In terms of attack methods, we discuss some frequently employed strong attacks and their underlying ideas. These attacks will be used in Chapter 5 to test the performance of our robust training strategies. In terms of verification methods, we present different formulations of the verification process and introduce a general Branch and Bound framework [10] comprising many existing verification methods. Both Chapter 3 and Chapter 4 are based on the Branch and Bound framework. Finally, we give a brief introduction to Graph Neural Networks (GNN), which is the main technique employed in Chapter 4.

Chapter 3 With the unified Branch and Bound framework, we identify the potential weakness of previous verification methods and propose several heuristics for improvement. New heuristics are developed for both bounding and branching components. In addition, we introduce comprehensive datasets consisting of trained as well as synthetic networks with fully connected and/or convolutional layers for verification. On these datasets, we conduct a thorough and extensive analysis of the available verification methods. We also demonstrate the outstanding performance

of our proposed heuristics.

Chapter 4 To make further improvements, we propose a novel framework for designing an effective branching strategy for the Branch and Bound algorithm to replace the heuristic proposed in Chapter 3. Specifically, we learn a GNN to imitate the strong branching heuristic behaviour. Empirically, our framework achieves roughly 50% reduction in both the number of branches and the time required for verification on various convolutional networks when compared to the best available hand-designed branching strategy. In addition, we show that our GNN model enjoys both horizontal and vertical transferability: horizontally, the model trained on easy properties performs well on the properties of increased difficulty levels; vertically, the model trained on small neural networks achieves similar performance on large neural networks. New datasets have also been proposed.

Chapter 5 In this chapter, we focus on strategies for constructing robust neural networks. We start by giving a brief overview of the various perspectives that have been taken on tackling the robustness problem and their related works. We note that many of the robust training strategies rely on strong adversaries. Strong adversaries can be prohibitively expensive to generate when the input dimension is high and the model structure is complicated. We adopt a new perspective on robustness and propose a novel training algorithm that allows a more effective use of adversaries. Specifically, with weak adversaries, our method reaches a similar robust accuracy level to the state-of-the-art approaches trained with strong adversaries on MNIST, CIFAR-10, and CIFAR-100. As a result, the overall training time is reduced. Furthermore, when trained with strong adversaries, our method matches with the current state-of-the-art on MNIST and outperforms them on CIFAR-10 and CIFAR-100.

Chapter 6 In the final chapter, we summarize key findings and identify possible future works.

1.4 Contributions and Publications

Overall, we have made the following contributions to the evaluation and construction of robust neural networks, which have led to several peer-reviewed publications.

1.4.1 Evaluating Neural Network Robustness

By focusing on formal verification and dealing with the general Branch and Bound framework [10] directly,

- We proposed high-level improvements for the bounding and branching components. In terms of bounding, we suggest problem-dependent reapproximation. When branching is considered, we developed effective heuristics for input branching and activation branching.

- We designed a learned branching strategy for a further improvement. We observed that the initially developed branching heuristic loses its effectiveness at later stages of the verification process. Strong branching, another branching heuristic, is capable of giving consistent high-quality branching decisions, but is computationally inhibitive even for small branching problems. To obtain both computational efficiency and branching quality, we proposed a learned branching strategy by employing GNNs to imitate the behaviour of the strong branching heuristic. We demonstrated the effectiveness of our strategy on various datasets.

- We constructed and introduced new datasets to facilitate the development of verification methods. We were the first to construct datasets on medium-sized convolutional networks and datasets with various internal properties. The resulting datasets have been employed in various studies [11, 22, 96] and served as one of the test sets in the first and second International Verification of Neural Networks Competition.

The above contributions are included in Chapter 3 and Chapter 4. The resulting publications are:

Rudy Bunel*, **Jingyue Lu***, Ilker Turkaslan, Philip H.S. Torr, Pushmeet Kohli and M. Pawan Kumar. 2020. 'Branch and Bound for Piecewise Linear Neural Network Verification'. *Journal of Machine Learning Research.*

Jingyue Lu and M. Pawan Kumar. 2020. 'Neural Network Branching for Neural Network Verification'. *The International Conference on Learning Representations (ICLR).* (**Long Talk** 1.85%)

1.4.2 Constructing Robust Neural Networks

One drawback of current robust training strategies is that most of them rely on strong adversaries. Strong adversaries are expensive to compute and require long training time. To improve on this,

- We proposed a new robust training framework that allows a more effective use of each adversary generated. By designing algorithms based on our framework, the final algorithm can achieve similar robustness performance using computationally cheap adversaries to those relying on expensive strong adversaries. As a result, our framework enables fast robust training.

The above contribution is included in Chapter 5 and the resulting publication is:.

Jingyue Lu and M. Pawan Kumar. 2021. 'Improving Local Effectiveness for Global Robust Training'. *ICLR RobustML Workshop.*

*Equal Contribution

2 Preliminaries

Contents

2.1 Neural Networks

Neural networks are sets of connected neurons that take an input $\boldsymbol{x}$ from a space $\mathcal{X} \subset \mathbb{R}^{|\mathcal{X}|}$ and produce an output $\boldsymbol{y}$ in a space $\mathcal{Y} \subset \mathbb{R}^{|\mathcal{Y}|}$. Depending on the nature

of spaces $\mathcal{X}$ and $\mathcal{Y}$, neural networks can be employed to perform various tasks. For example, neural networks can be utilized in image classification tasks when $\mathcal{X}$ is an image data distribution with total C distinct categories. Given an image input $\boldsymbol{x}$, we design a neural network to output a vector $\boldsymbol{g}$ such that $\boldsymbol{g} \in \mathbb{R}^{|C|}$. The predicted label c can be derived as $c \coloneqq \arg\max_{i \in C} g_i$. The output g_i of a network is also referred to as a logit.

2.1.1 Neural Network Layers

In terms of structure, a neural network consists of an input layer, multiple hidden layers, and an output layer. An example is provided in Figure 2.1. Each layer

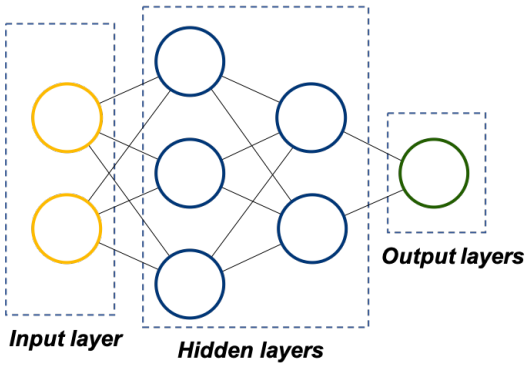


Figure 2.1: A simple neural network that takes a two-dimensional vectors as an input and outputs a scalar. It consists an input layer (yellow neurons), multiple hidden layers (blue neurons) and an output layer (green neuron).

can take various forms and the most commonly used layers include fully connected layers, convolutional layers, and activation layers, which we discuss below.

2.1.1.1 Fully Connected Layers

A fully connected layer connects every neuron on a layer to every neuron on the following layer. A simple illustration is shown in Figure 2.2. Every input node value passing through an edge is multiplied by the weight associated with the edge and all values arriving at the same node are summed up. The final output for each node on the next layer is the addition of the sum and its related bias. Mathematically, assume

the input is a vector $\boldsymbol{x}_i$. Let W^{i+1} be the weight matrix corresponding to the edges and $\boldsymbol{b}^{i+1}$ be the bias vector. A fully connected layer is a linear function in the form of

$$\hat{\boldsymbol{x}}_{i+1} = W^{i+1}\boldsymbol{x}_i + \boldsymbol{b}^{i+1}, \tag{2.1}$$

where weight W^{i+1} and bias $\boldsymbol{b}^{i+1}$ are learnable parameters.

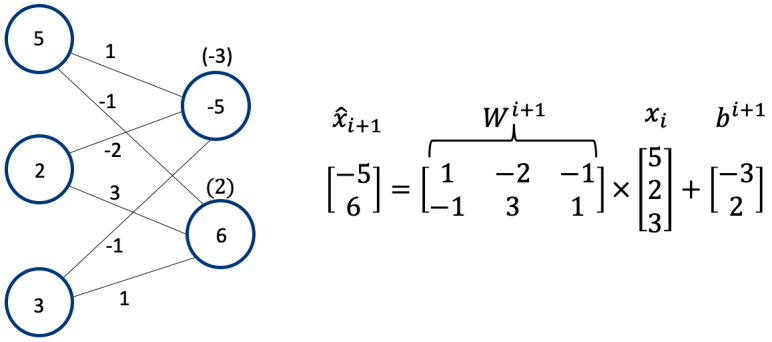


Figure 2.2: On the right is an instance of fully connected layers with an input vector of $\mathbb{R}^3$ and an output of $\mathbb{R}^2$. Input values or output values are shown in its corresponding neurons. A scalar weight is associated to each edge. Numbers in brackets are biases. On the left is its corresponding mathematical representation.

2.1.1.2 Convolutional Layers

Due to their capacity to capture spatial dependencies, convolutional layers have a crucial role in vision and other related areas. The key attribute of convolutional layers is to use filters to scan through only a few neurons at a time. This not only allows convolutional layers to efficiently and effectively detect various features, but also to differentiate the detected features by assigning distinct importance to them. We explain how convolutional layers work through a simple example, shown in Figure 2.3. As indicated in the example, kernel size and stride value are predetermined. In addition, in this example, only one channel of output is generated. Convolutional layers can support multiple channels of output. To obtain another channel of output, we simply provide a different set of kernels and a bias value. Formally, for a 2-dimensional (2D) convolutional layer, we compute the output as

$$\hat{X}_{i+1}^{out_j} = b_{out_j} + \sum_{p=1}^{|in|} K_p^{out_j} \star X^{in_p}, \tag{2.2}$$

where $\star$, in, out represent sliding inner product, input channels and output channels respectively. For a given output channel out_j, the set $\{K_p^{out_j} \mid p = 1, \ldots, |in|\}$ is its corresponding set of kernels and b_{out_j} is its associated bias. Using the similar idea, it is also possible to design convolutional layers for 1D and 3D cases. We mention that all kernels and biases are learnable parameters.

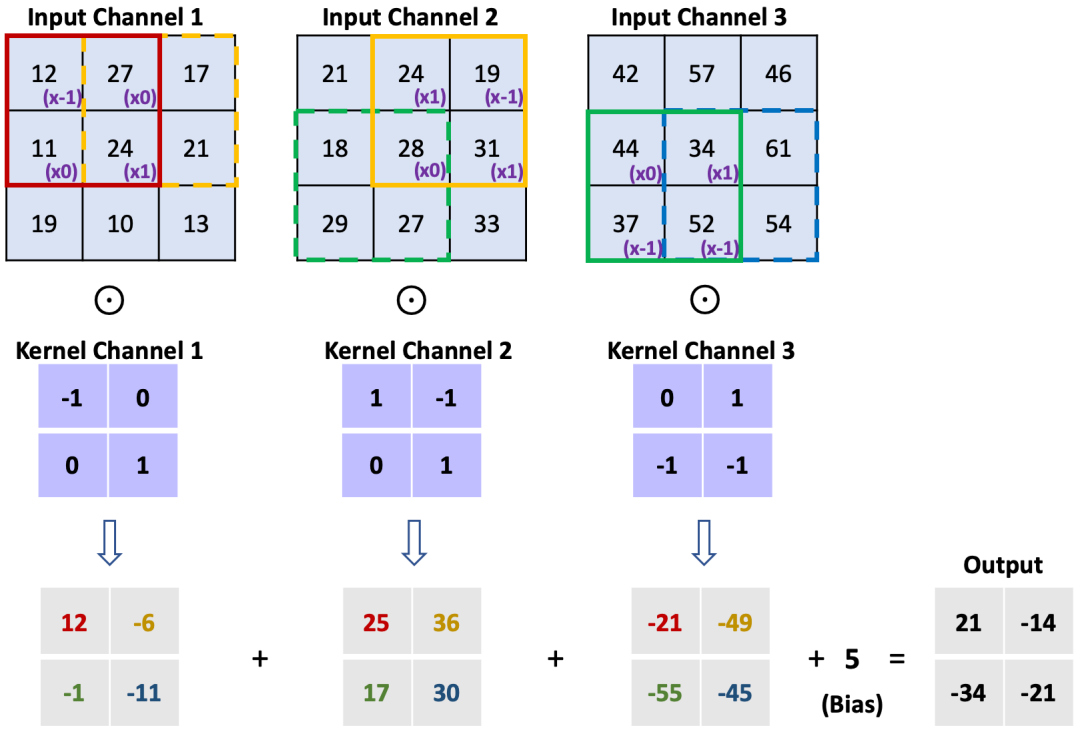


Figure 2.3: A simple example of applying a convolutional layer. Assume we have a 3 by 3 RBG image with three colour channels. The first row of Figure 2.3 shows the image with each square represents one channel. A convolutional layer carries out convolution operation on each channel respectively with each associated kernel/filter. Let kernel size be 2. The convolution operation starts from the upper-left corner and performs element-wise multiplication between the first 2 by 2 square of pixels (indicated by the red square) and the kernel (also shown by the purple number at the right-bottom of each pixel square). The final product is 12. We record it in red to specify it is the result for pixels in the red square (the same is done for all other colors). We then shift the filter to the right to perform another convolution operation. Here, we set the stride to be 1, so the filter is shifted one step to the right, as suggested by the yellow dotted square. Once a row is done, the filter will move one step downwards, as shown by the dotted green square. The final operation is indicated by the dotted blue square. We note that to facilitate the visibility, we only draw two color-squares on each channel. In practice, a filter moves from red to yellow to green and finally to blue on all channels. Once all computation is finished, we add results together with a bias value to obtain the final output.

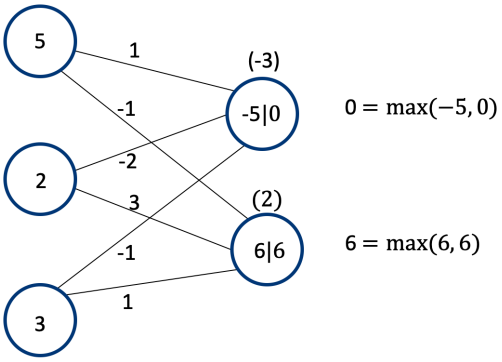


Figure 2.4: A fully connected layer as shown in Figure 2.2, followed by a ReLU activation layer.

2.1.1.3 Activation Layers

Activation layer applies an activation function to a neuron's output. Activation functions do not contain learnable parameters and many of them are nonlinear. As a result, apart from obtaining a more controlled behaviour of neurons, activation layers can be applied to add nonlinearities in networks. Doing so enables neural networks to deal with a wider range of problems. Frequently used activation functions include:

- ReLU: $\sigma(x) = \max(0, x)$,
- Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$,
- tanh: $\sigma(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$.

Activation layer normally follows after a fully connected layer or a convolutional layer. Because of this, we often integrate an activation function into a neuron node when using graph representations, as shown in Figure 2.4.

2.1.2 Neural Network Architecture

By combining layers in different ways, we obtain various neural network architectures.

2.1.2.1 Feed-forward Neural Networks

One basic type of neural networks are feed-forward neural networks. For these networks, information moves in one direction with no loops nor cycles. Common feed-forward neural networks consist of fully connected layers and convolutional layers, with each of the hidden layers followed by an activation layer. We give the mathematical representation for one such feed-forward neural network. We first note that since a convolutional layer is a linear operation, we can find its corresponding matrix operator and write it into a matrix multiplication form. We use this idea to simplify the notations. Assume $f(;\boldsymbol{\theta})$ is an L layer feed-forward neural network, $f : \mathcal{X} \to \mathbb{R}^C$, with ReLU activation units. Then for any $\boldsymbol{x}$, we have $f(\boldsymbol{x}) = \hat{\boldsymbol{x}}_L \in \mathbb{R}^C$ where

$$\hat{\boldsymbol{x}}_{i+1} = W^{i+1}\boldsymbol{x}_i + \boldsymbol{b}^{i+1}, \qquad \text{for } i = 0, \ldots, L-1, \tag{2.3a}$$

$$\boldsymbol{x}_i = \max(\hat{\boldsymbol{x}}_i, 0), \qquad \text{for } i = 1, \ldots, L-1. \tag{2.3b}$$

We will mainly deal with feed-forward neural networks with ReLU activation in the above form in Chapters 3 and 4 due to the complexity of verification methods.

We consider a more complicated feed-forward network architecture like wide residual network (WRN) in Chapter 5. WRN is a variant of residual network (ResNet) [41]. ResNet uses skip connections, shown in Figure 2.5, to jump over some layers, so training problems such as vanishing gradients can be avoided. This structure is particularly useful in constructing large-scale deep neural networks. WRN differs from ResNet in decreased depth but increased width of layers. As they are commonly utilized in other robustness-related papers, we directly adopted the WRN structure to conduct a fair comparison with existing robust training strategies. We refer interested readers to [111] for a more detailed description of WRN.

2.1.2.2 Convolutional Neural Networks

Another commonly used neural network architecture is convolutional neural networks (CNNs). Any neural network that employs one or more convolutional layers can be

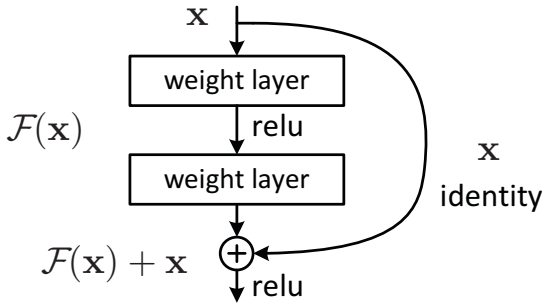


Figure 2.5: A residual block with a skip connection. The figure is adopted from [41].

referred to as a CNN. There is an overlap between feed-forward neural networks and convolutional neural networks.

2.1.3 Graph Neural Networks

In this section, we give an introduction to a special type of neural networks: graph neural networks (GNN), which are the main tools used in Chapter 4. GNNs are developed for data inputs that are in the form of graphs. Graph inputs are common in various areas. For instance, graphs are required to present molecules in chemistry, to summarize social connections in social networks, and to model interactions between users and products in recommendation systems. However, due to the irregularity and interconnections of graphs, standard neural networks are no longer applicable and specially designed GNNs are needed.

To facilitate the discussion, we first introduce some basic notations. A graph G is described by the set of vertices V and the set of edges E, such that $G = (V, E)$. Assume there are n nodes in V and m edges in E. We denote v for each node within V and $e_{uv} = (u, v)$ for each edge connecting nodes v and u in E. Moreover, we let $N(v)$ be the set of nodes that are in the neighbourhood of v. That is $N(v) = \{u \in V \,|\, (v, u) \in E\}$. The adjacency matrix $\mathbf{A}$ is a $n \times n$ matrix with the property $A_{ij} = 1$ if $e_{v_i v_j}$ is in E and $\mathbf{A}_{ij} = 0$ otherwise. A graph is directed, if all edges are directed from one node to another while a graph is undirected, if there is a pair of edges with opposing directions for any two connecting nodes. In terms of the adjacency matrix, a graph is undirected if and only if the adjacency

matrix is symmetric. On each node v, there are several node features. We combine those features into a d-dimensional vector $\mathbf{z}_v \in \mathbb{R}^d$. Collectively, we use the matrix $\mathbf{Z} \in \mathbb{R}^{n \times d}$ to present features for all nodes. Similarly, for each edge connecting u, v, we have a k-dimensional feature vector $\mathbf{z}^e_{uv} \in \mathbb{R}^k$ and for all edges, we have the feature matrix $\mathbf{Z}^e \in \mathbb{R}^{m \times k}$.

When constructing GNNs, it is common to use network embedding vectors. We assign a low-dimensional embedding vector to each node within the graph and denote the embedding vector for a node v as $\boldsymbol{\mu}_v$. Embedding vectors are generally set as zero vectors or randomly initialised. These low-dimensional vectors are representations that aim to capture its associated node features, the graph topology and other related neighbouring information. They are often the only vectors that get updated by GNN. After we finish updating embedding vectors, we apply desired graph analyses on these vectors only. In terms of tasks relating to nodes, such as node classification, node embedding vectors can be directly used. When results about edges are considered, a function that takes the embedding vectors of an edge's end nodes as an input can be employed. Finally, if the graph-level information is required, pooling and readout operations on embedding vector can be applied. For example, a graph-level embedding vector $\boldsymbol{\mu}_G$ can be generated with a readout function R as

$$\boldsymbol{\mu}_G = R(\{\boldsymbol{\mu}_v \,|\, v \in V\}), \tag{2.4}$$

where the set $\{\boldsymbol{\mu}_v \,|\, v \in V\}$ is the set of all updated embedding vectors and R can be a neural network with learnable parameters.

As summarized in [103], GNNs can be categorized into four types: namely, recurrent GNNs; convolutional GNNs; graph autoencoders and spatial-temporal GNNs. We will briefly discuss the first two types and refer interested readers to [39, 53, 62, 82, 93] for graph autoencoders and to [109, 110] for spatial-temporal GNNs.

2.1.3.1 Recurrent GNNs

Recurrent GNNs are the type of GNNs that are first developed in the field. The key underlying idea of recurrent GNNs (RecGNNs) is that by constantly exchanging

information with its neighbours, a node should eventually reach an equilibrium in terms of its embedding vector. Specifically, the process is carried out using recurrent neural architectures with iterative updates of embedding vectors. Mathematically, at each step t, the embedding vector of a node v is updated as follows

$$\boldsymbol{\mu}_v^{(t)} = \sum_{u \in N(v)} f(\boldsymbol{z}_v, \boldsymbol{z}_{uv}^e, \boldsymbol{z}_u, \boldsymbol{\mu}_u^{(t-1)}), \tag{2.5}$$

where f is the recurrent network [77]. We point out that, to ensure the convergence, f should be a contraction mapping, which could be achieved by imposing a penalty term on the Jacobian matrix of parameters.

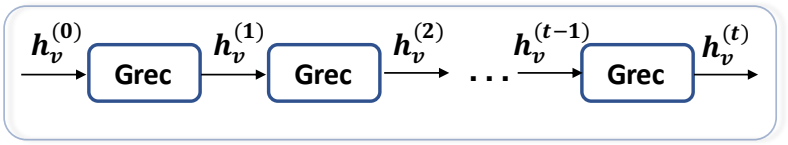
Although RecGNNs are considered computationally expensive, they hold theoretical importance for later work.

2.1.3.2 Convolutional GNNs

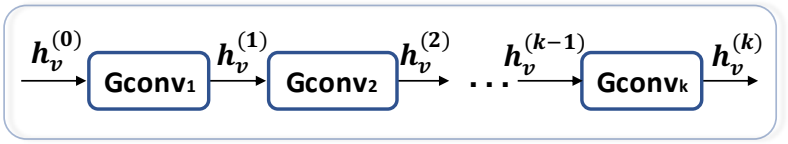
The development of convolutional GNNs (ConvGNNs) is inspired by RecGNNs and, at the same time, motivated by the success of convolutional neural networks in various tasks including computer vision. Instead of contraction maps, ConvGNNs use convolutional layers. In addition, the parameters of these layers can be changed at each update step. A clear illustration of the difference between RecGNNs and ConvGNNs has been provided in [103] and is shown in Figure 2.6. ConvGNNs can be further divided into spectral-based ConvGNNs and spatial-based ConvGNNs.

Spectral-based ConvGNNs

The theoretical foundation of spectral-based ConvGNNs lies in graph signal processing [80]. It starts by computing the normalised graph Laplacian matrix using the associated adjacency matrix and node degrees. Then by factorising the Laplacian matrix, we compute its eigenvalues and eigenvectors. The eigenvectors form an orthonormal space. For every embedding vector $\boldsymbol{\mu}$, we treat it as a graph input signal. To update it, we first use graph Fourier transform to redefine it in the orthonormal space; then update it with a filter, which is a set of learnable parameters; and finally, apply inverse graph Fourier transform to map the updated signal to its original space.



(a) Recurrent Graph Neural Networks (RecGNNs). RecGNNs use the same graph recurrent layer in updating node embedding vectors.



(b) Convolutional Graph Neural Networks (ConvGNNs). ConvGNNs use a different graph convolutional layer in updating node embedding vectors.

Figure 2.6: RecGNNs v.s. ConvGNNs. The graphs are adopted from [103]. The authors have used $\boldsymbol{h}_v$ to represent a node embedding vector, which is denoted as $\boldsymbol{\mu}_v$ in this study.

Due to the eigen-decomposition applied in spectral-based ConvGNNs, it is straightforward to note that these methods are expensive in computation and are sensitive to perturbations. Furthermore, once a graph with different orthonormal space is encountered, a new set of parameters for the filter should be learned. Several improvements have been made in tackling these issues, we refer interested readers to [23, 60].

Spatial-based ConvGNNs

As its name suggests, spatial-based ConvGNN updates an embedding vector using the corresponding node's spatial relationships. Similar to RecGNNs, the update iteration follows the idea of information propagation. Various spatial-based ConvGNN designs have been proposed, starting from the first spatial-based ConvGNN [65], to a general framework for spatial-based ConvGNN [34] and to the more refined networks that assign different importance to neighbouring nodes [39, 90]. Compared to spectral-based ConvGNNs, spatial-based ConvGNNs enjoy more flexibility and computational efficiency.

We focus on the general framework for ConvGNN, that is the Message Passing Neural Network (MPNN) [34]. Based on it, we designed our own GNN for verification in Chapter 4. By performing messaging passing on each node using its neighbouring

information, MPNN updates the embedding vectors as follows:

$$\boldsymbol{\mu}_v^{(t)} = U_t(\boldsymbol{\mu}_v^{(t-1)}, \sum_{u \in N(v)} M_t(\boldsymbol{\mu}_v^{(t-1)}, \boldsymbol{\mu}_u^{(t-1)}, \boldsymbol{z}_{uv}^e)). \tag{2.6}$$

Here, $\boldsymbol{\mu}_v^{(0)} = \boldsymbol{z}_v$ and $U_t(\cdot)$, $M_t(\cdot)$ are often multilayer perceptrons with learnable parameters. The updates are carried out for a predetermined number of times, denoted by T. Theoretically, the radius of information propagation depends on T with large T allowing information to propagate further. Once the update is finished, node-level, edge-level, and graph-level analyses can be conducted accordingly as described at the beginning of the section.

There are several possible issues of the MPNN. Firstly, the over-squashing issue for information propagation between distant nodes [2]. The issue is not specific for MPNN-based GNNs but also for RecGNNs. Aggregating information along long paths could be problematic, as it is impossible to have a high-quality information representation when exponentially growing information along the path has to be integrated into a fixed sized vector (consider a tree graph). Secondly, it has been found in [107] that MPNN-based GNNs may not have sufficient expressive power to be able to learn to distinguish certain graph structures. When faced with this problem, Graph Isomorphism Network (GIN) [107] can be applied. Different from MPNN, GIN learns an additional parameter $\epsilon^{(t)}$ and updates embedding vectors as

$$\boldsymbol{\mu}_v^{(t)} = M_t((1+\epsilon^{(t)})\boldsymbol{\mu}_v^{(t-1)}, \sum_{u \in N(v)} \boldsymbol{\mu}_u^{(t-1)}). \tag{2.7}$$

Here, $M_t(\cdot)$ stands for multilayer perceptrons. When designing a MPNN-based GNN specifically for verification problems in Chapter 4, we have taken careful consideration of these potential issues.

2.2 Evaluating Neural Networks

We evaluate a neural network's performance from three perspectives: nominal accuracy, empirical robust accuracy, and formal verification. In this thesis, we consider classification tasks only. Assume we are given an image dataset $\mathcal{X}$ and $\mathcal{X}$ contains a training dataset $\mathcal{X}_{train}$ and a test dataset $\mathcal{X}_{train}$.

2.2.1 Evaluating Nominal Accuracy

The basic and fundamental evaluation of a neural network $f(\cdot;\boldsymbol{\theta})$ is to test whether it makes correct predictions on the original image dataset $\mathcal{X}$. Since the network is trained on the training dataset, for an unbiased evaluation, we measure its nominal accuracy by counting the percentage of points such that the following condition is satisfied on the test dataset:

$$\arg\max_{i\in\mathcal{C}} f_i(\boldsymbol{x};\boldsymbol{\theta}) = c, \qquad \text{where } \boldsymbol{x} \in \mathcal{X}_{test}, \tag{2.8}$$

where c is the true label for $\boldsymbol{x}$.

2.2.2 Evaluating Robust Accuracy

In this work, we focus on the robustness of neural networks. A neural network $f(\cdot;\boldsymbol{\theta})$ is considered to be robust if for every $\boldsymbol{x} \in \mathcal{X}$ with label c, the network satisfies that

$$\arg\max_{i\in\mathcal{C}} f_i(\boldsymbol{x}';\boldsymbol{\theta}) = c, \qquad \forall \boldsymbol{x}' \in \mathcal{V}(\boldsymbol{x}). \tag{2.9}$$

Here, $\mathcal{V}(\boldsymbol{x})$ represents the set of all points that are visually similar to $\boldsymbol{x}$. In order to employ this abstract definition for robustness evaluation, we need to mathematically quantify the set $\mathcal{V}(\boldsymbol{x})$. However, it is extremely difficult to understand and to rigorously define visual similarity under the perception of humans. Researchers have resorted to simplified but mathematically well-defined perturbations. One type of widely studied perturbations is those measured by ℓ_p norms. Given an ϵ, we consider all images within ϵ distance to the original image are visually similar to the original image. Mathematically, we replace $\mathcal{V}(\boldsymbol{x})$ by

$$\mathcal{B}_\epsilon^p(\boldsymbol{x}) = \{\boldsymbol{x}' \mid \|\boldsymbol{x}' - \boldsymbol{x}\|_p \leq \epsilon\}.$$

Popular choices of p are $p = 0, 1, 2, \infty$. Similarly, one can replace $\mathcal{V}(\boldsymbol{x})$ by other types of perturbations, for instance, sets of images generated by rotating or translating the original image, as studied in [30].

Once the perturbation is determined, we evaluate the robustness of a neural network f on the test dataset $\mathcal{X}_{test}$ for an unbiased measure.

2.2.2.1 Empirical Robustness Metric

One frequently used metric is the empirical robustness metric, which measures networks' robustness empirically through adversarial attacks (discussed in Section 2.3). To be specific, we use the same fixed ϵ for data points within the test set. Then for every data point $\boldsymbol{x}$ in the test set, we attempt to find an adversary point $\boldsymbol{x}' \in \mathcal{B}_\epsilon^\infty(\boldsymbol{x})$ such that

$$\arg\max_{i \in \mathcal{C}} f_i(\boldsymbol{x}') \neq c, \tag{2.10}$$

within a given time limit. Adversarial attacks are designed to find such points $\boldsymbol{x}'$. If within the time limit, no such $\boldsymbol{x}'$ can be found, we say the network is empirically robust at the point $\boldsymbol{x}$. We measure the percentage of points at which the network is empirically robust and use the percentage as the final metric. The higher the percentage, the more robust a neural network is empirically.

We note that this metric has no theoretical guarantees. We cannot formally claim that the network is robust at a point $\boldsymbol{x}$ even when no adversary can be found. It could simply be that the adversarial attack used is not strong enough. However, this metric is computationally efficient. When dealing with large neural networks, this metric is often the only suitable choice. For the same reason, we adopt this metric only when testing our robust training strategy in Chapter 5.

2.2.2.2 Formal Verification

Formal verification is a broad concept. It focuses on formally evaluating whether a network satisfies a given property. Mathematically, given a network that implements a function $\hat{\boldsymbol{x}}_{\boldsymbol{L}} = f(\boldsymbol{x}; \boldsymbol{\theta})$, a bounded input domain $\mathcal{D}$ and a property P, we want to prove

$$\forall \mathbf{x} \in \mathcal{D}, \quad \hat{\mathbf{x}}_{\mathbf{L}} = f(\mathbf{x}; \boldsymbol{\theta}) \implies P(\hat{\mathbf{x}}_{\mathbf{L}}) \text{ is true.} \tag{2.11}$$

Depending on how the input domain and the property are defined, formal verification can be applied in a wide range of settings.

Verifying whether the robustness condition

$$\arg\max_{i \in \mathcal{C}} f_i(\boldsymbol{x}'; \boldsymbol{\theta}) = c, \qquad \forall \boldsymbol{x}' \in \mathcal{B}_\epsilon^\infty(\boldsymbol{x}). \tag{2.12}$$

holds is one instance of formal verification problems with $\mathcal{D} = \mathcal{B}_\epsilon^\infty(\boldsymbol{x})$ and $P(\hat{\mathbf{x}}_\mathbf{L}) = \left\{\hat{x}_{L[c]} > \hat{x}_{L[i]}, \quad \forall i \text{ s.t. } i \neq c\right\}$. We use $x_{i[j]}$ to denote the jth element of $\mathbf{x_i}$ from here onwards. We note that when the robustness at a point $\boldsymbol{x}$ is formally verified, it is guaranteed that no adversarial point $\boldsymbol{x}'$ can exist. Formal verification can be formulated in various forms and depending on the formulation, different verification methods have been proposed. We describe some verification methods in the Section 2.4.

It is desirable to have a theoretically guaranteed robustness metric. However, all verification methods are too expensive computationally to be carried out for every point within the test set. Thus, in this study, instead of directly applying verification methods for robustness evaluation, we focus on improving and developing verification methods in the hope to accelerate the verification process. This is the key theme of Chapter 3 and Chapter 4. To facilitate the study of verification methods, we focus on Piecewise-Linear neural networks (PL-NN). That is, feed-forward networks consist of fully connected or convolutional layers with ReLU or Maxpooling activation to simplify the verification process. We construct several datasets of verification properties based on the networks. Since the final goal is to formally verify the condition 2.12, verification properties constructed are mainly in the form: $P(\hat{\mathbf{x}}_\mathbf{L}) = \left\{\hat{x}_{L[c]} > \hat{x}_{L[i]}\right\}$ for $\mathcal{D} = \mathcal{B}_\epsilon^\infty(\boldsymbol{x})$, where c is the true label, i is a randomly selected targeted label and ϵ is chosen through binary search. We pick one test class i for each image to balance the large number of test images in $\mathcal{X}_{test}$ and the high computational cost for verifying a property. The value of ϵ determines how difficult a property is. We use binary search to control the overall difficulty of a verification property, so it is not easy to solve but solvable within certain time constraints. More details on the datasets can be found in the experiment sections in Chapter 3 and Chapter 4. We evaluate the performance of a verification method by counting the total number of properties it can verify within a time limit (generally

two hours per property).

Before we give a detailed introduction to the existing adversarial attacks and verification methods, we emphasise again the different settings employed in Chapters 3 and 4 and Chapter 5. Chapters 3 and 4 focus on developing verification methods for improved computational efficiency. New datasets consisting of verification properties are constructed for testing verification methods. Verification properties are intentionally generated to imitate the condition 2.12. We generate one verification property for one image for a subset of the whole image test dataset. For a comprehensive study, other verification properties are also used (More details in Chapter 3). All networks considered are PL-NN. Chapter 5 centres on training robust neural networks. There is no restriction on the network structure and considerably large network architectures are used. Due to their large size, the final trained neural networks are evaluated using the empirical robustness metric on the whole image test dataset $\mathcal{X}_{test}$ and verification methods are not applicable.

2.3 Adversarial Attacks

We first discuss adversarial attacks. Adversarial attacks disapprove condition 2.12 by finding counterexamples. Given a time or computational limit, adversarial attacks attempt to find an adversarial example $\boldsymbol{x}_{adv}$ (also known as adversary) such that

$$\arg\max_{i\in\mathcal{C}} f_i(\boldsymbol{x}_{adv};\boldsymbol{\theta}) \neq c, \qquad \boldsymbol{x}_{adv} \in \mathcal{B}_\epsilon^\infty(\boldsymbol{x}). \tag{2.13}$$

The percentage of points that no such example can be found is used as the robustness measure. Depending on the amount of network information the attacker has access to, attacks can be further divided into white box attacks and black box attacks. White box attacks are attacks that have access to all information of a target model, including model weights and architecture. They are thus able to compute the target model's gradient and perform back-propagation. On the other hand, black box attacks have the limited access to the logits output (also called the soft output) of the target model.

2.3.1 White Box Attacks

We discuss popular and powerful white box attacks in the following. These attacks are also used in Chapter 5 for evaluating a model's robustness.

PGD Attacks, introduced in [63], find adversaries by searching for a point that maximises the objective loss function ℓ. The maximization problem is approached by performing multistep project gradient descent (PGD). We point out that PGD is developed upon the Fast Gradient Sign Method (FGSM) [36], where the following one-step update is performed:

$$\boldsymbol{x}' = \boldsymbol{x} + \epsilon \cdot \operatorname{sgn}(\nabla_{\boldsymbol{x}} \ell(f(\boldsymbol{x}; \boldsymbol{\theta})), c)). \tag{2.14}$$

We use sgn to represent the sign function. In terms of its multi-step variant PGD, the algorithm starts with $\boldsymbol{x}_0$, which could either be the original point $\boldsymbol{x}$ or its slightly perturbed variant within $\mathcal{B}_\epsilon^\infty(\boldsymbol{x})$. Then at any given step t, we update $\boldsymbol{x}_t$ through a PGD step

$$\boldsymbol{x}_{t+1} = \Pi_{\mathcal{B}_\epsilon^\infty(\boldsymbol{x})}(\boldsymbol{x}_t + \xi \cdot \operatorname{sgn}(\nabla_{\boldsymbol{x}_t} \ell(f(\boldsymbol{x}_t; \boldsymbol{\theta})), c)), \qquad \xi < \epsilon. \tag{2.15}$$

Here, $\Pi_{\mathcal{B}_\epsilon^\infty}$ is the projection operator projecting an input onto the ball $\mathcal{B}_\epsilon^\infty(\boldsymbol{x})$ and ξ is the pre-determined step-size. The total number of PDG steps is fixed at the beginning and the process terminates when the current point $\boldsymbol{x}_{t+1}$ is verified to be an adversary. Generally, the more steps of PGD allowed, the stronger the attack is. It is also found that starting point has an impact on the strength of the attack. As a result, we could conduct multiple restarts with a different starting point at each time (any point within the ball $\mathcal{B}_\epsilon^\infty$ is a feasible starting point) for a fixed number of PGD steps, which could potentially lead to a strengthened attack.

Un-targeted Attacks is introduced in [72]. It is very similar to PGD attacks, but a different maximization objective is used. Instead of maximizing the loss function, the untargeted attack maximizes the function

$$f_u(\boldsymbol{x}; \boldsymbol{\theta}) - f_c(\boldsymbol{x}; \boldsymbol{\theta}), \tag{2.16}$$

where c is the correct class and $u = \arg\max_{i \neq c} f_i(\boldsymbol{x})$. We allow u to change during each PGD step.

Multi-targeted Attacks, also adopted from [72], is a stronger attack than un-targeted attacks. It performs extensive search for each possible wrong class $i \in \mathcal{C}$ such that $i \neq c$. For each i, we perform PGD steps with the maximization objective

$$f_i(\boldsymbol{x}; \boldsymbol{\theta}) - f_c(\boldsymbol{x}; \boldsymbol{\theta}). \tag{2.17}$$

Since this attack considers every wrong class, it is much more expensive than other attacks with the computational cost growing linearly with the total number of classes. As a result, we do not perform this attack on CIFAR100 dataset.

2.3.2 Black Box Attacks

In general, black box attacks are weaker than white box attacks due to their limited access to a model's information. The common approach for performing black box attacks is to generate adversaries with a surrogate model and then attack the original model with those generated adversaries.

We have also adopted this strategy in performing black box attacks in Chapter 5. We first trained a surrogate model on the same training dataset. Since the parameters of the surrogate model are known, we then perform white box attacks on the surrogate model to produce adversarial examples.

2.3.3 Ensemble Attacks

Finally, we discuss an ensemble of parameter-free attacks. This ensemble of attacks is proposed in [20] and is referred to as AutoAttack. It consists of two variants of PGD attacks and two other complementary attacks with one being white box and the other being black box. The two variants of PGD attacks are developed to improve on two potential causes of failure for PGD, identified in [20], namely the limitation of the fixed step size used in PGD and the widely used cross-entropy loss. The other two attacks are included for diversity reasons. AutoAttack are comparatively

strong attacks. They are also easily applicable, due to their parameter-free nature.

So far, we have listed all adversarial attacks that are used in Chapter 5 for measuring models' robustness. We emphasize that given the incomplete nature of adversarial attacks, their results can provide an inaccurate impression of a model's robustness. As demonstrated in [6, 20], several robust training strategies, which claim to have state-of-the-art performance under attacks, are found to be ineffective under stronger or different types of attacks. Bearing in mind the possible insufficiency of adversarial attacks, we hope to make fair and sound robustness evaluations in Chapter 5 by including a large range of attacks.

2.4 Formal Verification

Overall, verification algorithms can be divided into three categories: algorithms are **unsound** if they can only prove some of the false properties are false; algorithms are **incomplete** if they can only prove some of the true properties are true; and algorithms are **complete** if they are able to report all correct properties. In this work, we will only focus on **complete** algorithms. For unsound methods, we refer interested readers to [1, 13, 45, 98] and for incomplete methods, we refer interested readers to [27, 84, 99, 104]. In addition, among complete methods, the scope of this work does not include approaches relying on additional assumptions such as twice differentiability of the network [42, 112], limitation of the activation to binary values [16, 69] or restriction to a single linear domain [8].

2.4.1 Verification Formalism

In this section, we reduce all instances of verification problems to canonical representations and present different formulations of the verification process. We consider the general formal verification problem:

$$\forall \mathbf{x} \in \mathcal{D}, \quad \hat{\mathbf{x}}_{\mathbf{L}} = f(\mathbf{x}; \boldsymbol{\theta}) \implies P(\hat{\mathbf{x}}_{\mathbf{L}}) \text{ is true}, \tag{2.18}$$

where f represents a network such that $\hat{\boldsymbol{x}}_{\boldsymbol{L}} = f(\boldsymbol{x}; \boldsymbol{\theta})$, D is a bounded input domain and the property P is restricted to a Boolean formula over linear inequalities.

2.4.1.1 Verification as a Satisfiability Problem

We first consider satisfiability problems. We note the whole satisfiability problem can be transformed into a global optimisation problem where the decision will be obtained by checking the sign of the minimum. If the property is a simple inequality $P(\hat{\mathbf{x}}_{\mathbf{L}}) \coloneqq \mathbf{c}^T \hat{\mathbf{x}}_{\mathbf{L}} > b$, it is sufficient to add to the network a final fully connected layer with one output, with weight of $\mathbf{c}$ and a bias of $-b$. If the global minimum of this network is positive, it indicates that for all $\hat{\mathbf{x}}_{\mathbf{L}}$, the original network output, we have $\mathbf{c}^T \hat{\mathbf{x}}_{\mathbf{L}} - b > 0 \implies \mathbf{c}^T \hat{\mathbf{x}}_{\mathbf{L}} > b$, and as a consequence the property is true. On the other hand, if the global minimum is negative, then the minimizer provides a counterexample. Clauses `OR` and `AND` in the property can similarly be expressed as additional layers, using MaxPooling units. Specifically, clauses specified using `OR` (denoted by $\bigvee$) can be encoded by using a MaxPooling unit. If the property is $P(\hat{\mathbf{x}}_{\mathbf{L}}) = \bigvee_i \left[\mathbf{c}_i^T \hat{\mathbf{x}}_{\mathbf{L}} > b_i\right]$, this is equivalent to $\max_i \left(\mathbf{c}_i^T \hat{\mathbf{x}}_{\mathbf{L}} - b_i\right) > 0$. Clauses specified using `AND` (denoted by $\bigwedge$) can be encoded similarly: the property $P(\hat{\mathbf{x}}_{\mathbf{L}}) = \bigwedge_i \left[\mathbf{c}_i^T \hat{\mathbf{x}}_{\mathbf{L}} > b_i\right]$ is equivalent to $\min_i \left(\mathbf{c}_i^T \hat{\mathbf{x}}_{\mathbf{L}} - b_i\right) > 0 \iff -\left(\max_i \left(-\mathbf{c}_i^T \hat{\mathbf{x}}_{\mathbf{L}} + b_i\right)\right) > 0$. We can formulate any Boolean formula over linear inequalities on the output of the network as a sequence of additional linear and max-pooling layers. From now on, we assume that the property is in a canonical form. Specifically, the output of the network is a scalar, and the property is true if the output is positive for all inputs in a given domain, and false otherwise. Assuming the network only contains ReLU activations between each layer, the satisfiability problem to find a counterexample can be expressed as:

$$\mathbf{l_0} \leq \mathbf{x} \leq \mathbf{u_0} \quad (2.19a) \qquad \hat{\mathbf{x}}_{\mathbf{i+1}} = W_{i+1}\mathbf{x_i} + \mathbf{b_{i+1}} \quad \forall i \in \{0,\ L-1\} \quad (2.19c)$$

$$\hat{x}_L \leq 0 \quad (2.19b) \qquad \mathbf{x_i} = \max\left(\hat{\mathbf{x}}_{\mathbf{i}}, 0\right) \quad \forall i \in \{1,\ L-1\}. \quad (2.19d)$$

Equation 2.19a represents the constraints on the input and Equation 2.19b on the neural network output. Equation 2.19c encodes the linear layers of the network and Equation 2.19d the ReLU activation functions. If an assignment

to all values can be found, this represents a counterexample. If this problem is unsatisfiable, no counterexample can exist, implying that the property is true. We emphasise that we are required to prove that no counterexamples can exist, and not simply that none could be found.

While for clarity of explanation, we have limited ourselves to the specific case where only ReLU activation functions are used, this is not restrictive. MaxPooling units can also be dealt with by converting each of them into a combination of linear layers and ReLU activation functions. To do so, we decompose the element-wise maximum into a series of pairwise maximum

$$\begin{aligned}\max\left(x_1, x_2, x_3, x_4\right) = \max(\ & \max\left(x_1, x_2\right), \\ & \max\left(x_3, x_4\right))\end{aligned} \tag{2.20}$$

and decompose the pairwise maximums as sum of ReLUs:

$$\max\left(x_1, x_2\right) = \max\left(x_1 - x_2,\ 0\right) + \max\left(x_2 - l_{x_2}, 0\right) + l_{x_2}, \tag{2.21}$$

where l_{x_2} is a pre-computed lower bound of the value that x_2 can take. As a result, the only requirement is to be able to compute necessary lower bounds, for which the methods discussed later can help.

Reformulating a verification problem into this canonical representation does not make its resolution simpler since the addition of the ReLU non-linearities Equation 2.19d transforms a problem that would have been solvable by simple Linear Programming into an NP-hard problem [50]. However, it does provide a formalism advantage. Specifically, it allows us to prove complex properties, containing several `OR` clauses, with a single procedure rather than having to decompose the desired property into separate queries as was done in previous work [50]. Operationally, a valid strategy for dealing with verification problems in the canonical form is to impose the constraints Equations 2.19a-2.19d and minimise the value of $\hat{x}_L$. Finding the exact global minimum is not necessary for verification. However, it provides a measure of satisfiability or unsatisfiability. If the value of the global minimum is positive, it will correspond to the margin by which the property is satisfied.

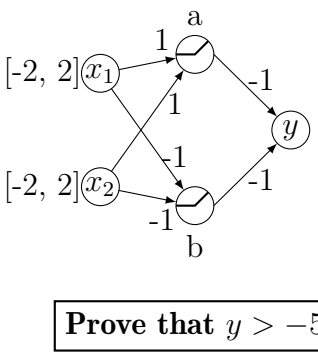


Figure 2.7: Example Neural Network. We attempt to prove the property that the network output is always greater than -5.

Toy Example *A toy-example of the Neural Network verification problem is given in Figure 2.7. On the domain* $\mathcal{D} = [-2; 2] \times [-2; 2]$*, we want to prove that the output* y *of the one hidden-layer network always satisfies the property* $P(y) = [y > -5]$*. We will use this as a running example to illustrate different formulations of the problem and discuss methods that can be reframed in our unified framework.*

For the network of Figure 2.7, the problem is formulated as follows. The variables would be $\{x_1, x_2, a_{in}, a_{out}, b_{in}, b_{out}, y\}$ *and the set of constraints would be:*

$$-2 \leq x_1 \leq 2 \qquad -2 \leq x_2 \leq 2 \tag{2.22a}$$

$$\hat{a} = x_1 + x_2 \qquad \hat{b} = -x_1 - x_2 \tag{2.22b}$$

$$a = \max(\hat{a}, 0) \qquad b = \max(\hat{b}, 0) \tag{2.22c}$$

$$y = -a - b \tag{2.22d}$$

$$y \leq -5. \tag{2.22e}$$

Here, $\hat{a}, \hat{b}$ *is the input value to hidden unit, while* a, b *is the value after the ReLU. Any point satisfying the above constraints would be a counterexample to the property, as it would imply that it is possible to drive the output to -5 or less.*

2.4.1.2 Mixed Integer Linear Programming Formulation

The main difficulty for the verification of PL-NN is the nonlinearities brought by ReLU units. A possible way to eliminate the nonlinearities is to encode them with the help of binary variables, transforming the PL-NN verification problem Equation 2.19 into a Mixed Integer Linear Programming (MIP) problem. This can be done with the use of "big-M" encoding [38]. The following encoding is from [88]. Assuming we have access to lower and upper bounds on the values that can be taken by the coordinates of $\hat{\mathbf{x}}_\mathbf{i}$, which we denote $\mathbf{l_i}$ and $\mathbf{u_i}$, we can replace the nonlinearities:

$$\mathbf{x_i} = \max(\hat{\mathbf{x}}_\mathbf{i}, 0) \quad \Rightarrow \quad \boldsymbol{\delta_i} \in \{0,1\}^{h_i}, \quad \mathbf{x_i} \geq 0, \quad \mathbf{x_i} \leq \mathbf{u_i} \cdot \boldsymbol{\delta_i} \tag{2.23a}$$

$$\mathbf{x_i} \geq \hat{\mathbf{x}}_\mathbf{i}, \quad \mathbf{x_i} \leq \hat{\mathbf{x}}_\mathbf{i} - \mathbf{l_i} \cdot (1 - \boldsymbol{\delta_i}). \tag{2.23b}$$

It is easy to verify that $\delta_{i[j]} = 0 \Leftrightarrow x_{i[j]} = 0$ (replacing $\delta_{i[j]}$ in Equation 2.23a) and $\delta_{i[j]} = 1 \Leftrightarrow x_{i[j]} = \hat{x}_{i[j]}$ (replacing $\delta_{i[j]}$ in Equation 2.23b). From now on, we refer to $\mathbf{l_0}$ and $\mathbf{u_0}$, the lower and upper bounds for the input domain, as input bounds and $\mathbf{l_i}$ and $\mathbf{u_i}$ for $i \in \{1,\ L-1\}$, the lower and upper bounds for hidden units $\hat{\mathbf{x}}_\mathbf{i}$, as intermediate bounds.

Toy Example (MIP formulation) *In our example, the nonlinearity of Equation 2.22c would be replaced by the following conditions. Here, we give the detailed description for $\hat{a}$. The same is applied to $\hat{b}$.*

$$\begin{aligned} &a \geq 0 && a \geq \hat{a} \\ &a \leq \hat{a} - l_a(1-\delta_a) && a \leq u_a \delta_a \\ &\delta_a \in \{0,1\}. \end{aligned} \tag{2.24}$$

where l_a is a lower bound of the value that $\hat{a}$ can take (such as -4) and u_a is an upper bound (such as 4). The binary variable δ_a indicates which phase the ReLU is in: if $\delta_a = 0$, the ReLU is blocked and $a = 0$, else the ReLU is passing and $a = \hat{a}$. The problem remains difficult due to the integer constraint on δ_a.

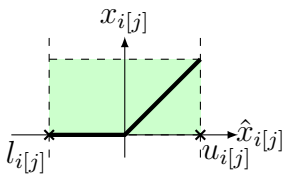


Figure 2.8: Feasible domain corresponding to a naive relaxation for a single ReLU.

We now consider bounds computation. By taking advantage of the feed-forward structure of the neural network, lower and upper bounds $\mathbf{l_i}$ and $\mathbf{u_i}$ can be obtained by applying interval arithmetic [43] to propagate the bounds to the inputs, one layer at a time. When convex relaxation is considered, we mention that interval arithmetic applies a naive relaxation, as shown in Figure 2.8. Specifically, if we have bounds $\mathbf{l_i}, \mathbf{u_i}$ for a vector $\mathbf{x_i}$, we can derive the bounds $\hat{\mathbf{l}}_{\mathbf{i+1}}, \hat{\mathbf{u}}_{\mathbf{i+1}}$ for a vector $\hat{\mathbf{x}}_{\mathbf{i+1}} = W_{i+1}\mathbf{x_i} + \boldsymbol{b}_{i+1}$:

$$\hat{l}_{i+1[j]} = \sum_k \left(W^+_{i+1[j,k]} l^+_{i[k]} + W^-_{i+1[j,k]} u^+_{i[k]} \right) + b_{i+1[j]} \tag{2.25a}$$

$$\hat{u}_{i+1[j]} = \sum_k \left(W^+_{i+1[j,k]} u^+_{i[k]} + W^-_{i+1[j,k]} l^-_{i[k]} \right) + b_{i+1[j]} \tag{2.25b}$$

with the notation $a^+ = \max(a, 0)$ and $a^- = \min(a, 0)$. Propagating the bounds through a ReLU activation is simply equivalent to applying the ReLU to the bounds.

Thanks to this particular feed-forward structure of the problem, the generic nonlinear, nonconvex problem has been rewritten into a MIP. Optimization of MIP is well studied and highly efficient off-the-shelf solvers exist. As solving them is NP-hard, the performance is going to be dependent on the quality of both the solver used and the encoding. We now ask the following question: how much efficiency can be gained by using a bespoke solver rather than a generic one? In order to answer this, we present specialised solvers under the unified Branch and Bound framework for the PL-NN verification task.

2.4.2 Branch and Bound for Verification

As described in the discussion of satisfiability problems, the verification problem can be rephrased as a global optimisation problem. For nonconvex problems, algorithms such as first order methods like gradient descent are not appropriate as they have

no way of guaranteeing whether or not a stationary point is a global minimum. In this section, we present an approach to estimate the global minimum, based on the Branch and Bound paradigm [57, 68]. We also show that several published methods fit this framework. Among these methods, detailed studies are conducted on the previous state-of-the-art methods Reluplex and Planet, which are introduced as examples of Satisfiability Modulo Theories (SMT).

Algorithm 1 Branch and Bound

```
1: function BAB(net, problem, ϵ)
2:   global_ub ← inf
3:   global_lb ← − inf
4:   probs ← [(global_lb, problem)]
5:   while global_ub − global_lb > ϵ do
6:     (_ , prob) ← pick_out(probs)
7:     [subprob_1, . . . , subprob_s] ← split(prob)
8:     for i = 1 . . . s do
9:       prob_ub ← compute_UB(net, subprob_i)
10:      prob_lb ← compute_LB(net, subprob_i)
11:      if prob_ub < global_ub then
12:        global_ub ← prob_ub
13:        prune_problems(probs, global_ub)
14:      end if
15:      if prob_lb < global_ub then
16:        problems.append((prob_lb, subprob_i))
17:      end if
18:    end for
19:    global_lb ← min{lb | (lb, prob) ∈ probs}
20:  end while
21:  return global_ub
22: end function
```

Algorithm 1 describes its generic form. The original problem is repeatedly split into sub-problems (either split the input domain into sub-domains or an unfixed ReLU activation unit [1] into different phases) (line 7), over which the lower and upper bounds of the minimum are computed (lines 9-10). The best upper bound found so far serves as a candidate for the global minimum. Any sub-problem whose lower bound is greater than the current global upper bound can be pruned away as it cannot contain the global minimum (line 13, lines 15-17). By iteratively splitting the

[1] We refer to a ReLU activation unit $x_i[j] = \max(\hat{x}_{i[j]}, 0)$ as unfixed if, given the upper and lower bounds $u_{i[j]}, l_{i[j]}$ of $x_{i[j]}$, $x_{i[j]}$ can take either the value of $\hat{x}_{i[j]}$ or 0.

(sub-)problems, it is possible to compute tighter lower bounds. We keep track of the global lower bound on the minimum by taking the minimum over the lower bounds of all sub-problems (line 19). When the global upper bound and the global lower bound differ by less than a small scalar ϵ (line 5), we consider that we have converged.

Algorithm 1 shows how to optimise and obtain the global minimum. If all that we are interested in is the satisfiability problem, the procedure can be simplified by initialising the global upper bound to 0 (in line 2). Any sub-problem with a lower bound greater than 0 (and therefore not eligible to contain a counterexample) will be pruned out (by line 15). The computation of the lower bound can therefore be replaced by the feasibility problem (or its relaxation) imposing the constraint that the output is below zero without changing the algorithm. If it is feasible, there might still be a counterexample and further branching is necessary. If it is infeasible, the sub-problem can be pruned out. In addition, if any upper bound improving on 0 is found on a sub-problem (line 11), it is possible to stop the algorithm as this already indicates the presence of a counterexample.

The description of the verification problem as optimisation and the pseudo-code of Algorithm 1 are generic and would apply to verification problems beyond the specific case of PL-NN. To obtain a practical algorithm, it is necessary to specify several elements.

A search strategy, defined by the `pick_out` function, which chooses the next problem to branch on. Several heuristics are possible, for example, those based on the results of previous bound computations. For satisfiable problems or optimisation problems, this allows us to discover good upper bounds, enabling early pruning.

A branching rule, defined by the `split` function, which takes a problem `prob` and returns its partition into subproblems such that $\bigcup_i \texttt{subprob\_i} = \texttt{prob}$ and that $(\texttt{subprob\_i} \cap \texttt{subprob\_j}) = \emptyset,\ \forall i \neq j$. This will determine the attributes of the (sub-)problems, which impacts the hardness of computing bounds. In addition, choosing the right partition can greatly impact the quality of the resulting bounds.

Bounding methods, defined by the `compute_`$\{$`UB`, `LB`$\}$ functions. These procedures estimate respectively upper bounds (`prob_ub`) and lower bounds (`prob_lb`)

over the minimum output that the network `net` can reach over a given (sub-)problem. We want the lower bound to be as high as possible, so that this (sub-)problem can be pruned easily. This is usually done by introducing convex relaxations of the (sub-)problem and minimising them. Generally, tighter relaxations lead to higher computational cost. On the other hand, the computed upper bound should be as small as possible, to allow pruning out other (sub-)problems or discovering counterexamples. As any feasible point corresponds to an upper bound on the minimum, heuristic methods are sufficient.

2.4.3 Branch and Bound Reformulations

We now give a general discussion of published works in the verification literature through the lens of the Branch and Bound framework. We first briefly mention some methods that do not rely on SMT solvers and then conduct detailed studies on SMT based methods Reluplex and Planet. ReluVal is a complete method introduced by Wang et al. [95]. In ReluVal, an input domain branching rule is used and the split function decides which domain dimension to split on by computing influence metrics based on input-output gradient. For the bounding method, it uses a novel technique called symbolic interval propagation, which replaces lower and upper bounds by linear equations of input variables and propagates these linear equations in a layer by layer order. By doing so, symbolic intervals can preserve more dependency information than interval arithmetic, and are thus able to achieve tighter final bounds. Inspired by the branching rule used by ReluVal, Royo et al. [76] proposed a verification procedure via shadow prices. This method also adopts a Branch and Bound framework. In detail, for the branching rule, the method makes an input split decision through sensitivity studies of the bounds of unfixed ReLU nodes to the change of the input domain. The bounding method is not specified, but the same logic has been used to check whether a sub-domain should be stored and further split or should be pruned. Also based on ReluVal is a complete method called Neurify [94]. Neurify is an improved version of ReluVal, which also uses gradient metrics to make a split decision but it splits on unfixed ReLU nodes.

Neurify uses a different bounding method than ReluVal by calling an LP solver instead of computing symbolic interval bounds. For all the methods mentioned above, no specific search strategy is implemented. All sub-problems are simply enumerated. We now shift our focus to SMT-based methods.

2.4.3.1 Reluplex

Katz et al. [50] present a procedure named Reluplex to verify properties of neural network containing linear functions and ReLU activation units. Reluplex functions as an SMT solver using the splitting-on-demand framework [7]. The principle of Reluplex is to always maintain an assignment to all variables, even if some of the constraints are violated.

Starting from an initial assignment, it attempts to fix some violated constraints at each step. It prioritises fixing linear constraints (Equations 2.19a, 2.19b, 2.19c and some relaxation of Equation 2.19d) using a simplex algorithm, even if it leads to violated ReLU constraints. If no solution to this relaxed problem containing only linear constraints exists, the counterexample search is unsatisfiable. Otherwise, either all ReLus are respected, which generates a counterexample, or Reluplex attempts to fix one of the violated ReLu, potentially leading to newly violated linear constraints. This process is not guaranteed to converge. Thus, to make progress, the nonlinearities that get fixed too often are split into two cases. Two new problems are generated, each corresponding to one of the phases of the ReLU. In the worst setting, the problem will be split completely over all possible combinations of activation patterns, at which point the sub-problems will all be simple LPs.

This algorithm is a special case of Branch and Bound for satisfiability. The **search strategy** is handled by the SMT core and to the best of our knowledge does not prioritise any (sub-)problems. The **branching rule** is implemented by the ReLU-splitting procedure: when neither the upper bound search nor the detection of infeasibility are successful, one nonlinear constraint over the j-th neuron of the i-th layer $x_{i[j]} = \max\left(\hat{x}_{i[j]}, 0\right)$ is split out into two sub-problems:

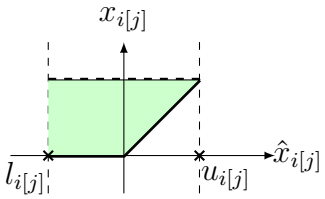


Figure 2.9: Feasible domain corresponding to the Reluplex relaxation for a single ReLU.

$\{x_{i[j]} = 0, \hat{x}_{i[j]} \leq 0\}$ and $\{x_{i[j]} = \hat{x}_{i[j]}, \hat{x}_{i[j]} \geq 0\}$. The prioritisation of the ReLUs that has been frequently fixed is a heuristic.

As Reluplex only deals with satisfiability, the analogue of the lower bound computation is an over-approximation of the satisfiability problem. The **bounding method** used is a convex relaxation, obtained by dropping some of the constraints. The following relaxation (also see Figure 2.9) is applied to ReLU units for which the sign of the input is unknown ($l_{i[j]} < 0$ and $u_{i[j]} > 0$),

$$\mathbf{x_i} = \max(\hat{\mathbf{x}}_\mathbf{i}, 0) \Rightarrow \mathbf{x_i} \geq \hat{\mathbf{x}}_\mathbf{i} \quad (2.26a) \qquad \mathbf{x_i} \geq 0 \quad (2.26b) \qquad \mathbf{x_i} \leq \mathbf{u_i}. \quad (2.26c)$$

If this relaxation is unsatisfiable, this indicates that the subdomain cannot contain any counterexample and can be pruned out. The search for an assignment satisfying all the ReLU constraints by iteratively attempting to correct the violated ReLUs is a heuristic that is equivalent to the search for an upper bound lower than 0: success implies the end of the procedure.

Toy Example (Running Reluplex) *Figure 2.10 shows the initial steps of a run of the Reluplex algorithm on the example of Figure 2.7. Starting from an initial assignment, it attempts to fix some violated constraints at each step. It prioritises fixing linear constraints (Equations 2.22a, 2.22b and 2.22e in our illustrative example) using a simplex algorithm, even if it leads to violated ReLU constraints Equation 2.22c. This can be seen in step 1 and 3 of the process.*

If no solution to the problem containing only linear constraints exists, this shows that the counterexample search is unsatisfiable. Otherwise, all linear constraints are fixed and Reluplex attempts to fix one violated ReLU at a time, such as in step

Step	x_1	x_2	$\hat{a}$	a	$\hat{b}$	b	y
1	0	0	0	0	0	0	0
	Fix linear constraints						
	0	0	0	1	0	4	-5
2	0	0	0	1	0	4	-5
	Fix a ReLU						
	0	0	0	1	4	4	-5
3	0	0	0	1	4	4	-5
	Fix linear constraints						
	-2	-2	-4	1	4	4	-5
	...						

Figure 2.10: Evolution of the Reluplex algorithm. Red cells corresponds to value violating linear constraints, and orange cells corresponds to value violating ReLU constraints. Resolution of violation of linear constraints are prioritised.

2 of Figure 2.10 (fixing the ReLU b), potentially leading to newly violated linear constraints. In the case where no violated ReLU exists, this means that a satisfiable assignment has been found and that the search can be terminated early.

2.4.3.2 Planet

Ehlers [28] also proposed an approach based on SMT. Unlike Reluplex, the proposed tool, named Planet, operates by explicitly attempting to find an assignment to the phase of the nonlinearities. Reusing the notation of Section 2.4.1.2, it assigns a value of 0 or 1 to each $\delta_{i[j]}$ variable, verifying at each step the feasibility of the partial assignment to prune infeasible partial assignment early.

As in Reluplex, the **search strategy** is not explicitly encoded and simply iterates over the (sub-)problems that have not yet been pruned. The **branching rule** is the same as for Reluplex, as fixing the decision variable $\delta_{i[j]} = 0$ is equivalent to choosing $\{x_{i[j]} = 0, \hat{x}_{i[j]} \leq 0\}$ and fixing $\delta_{i[j]} = 1$ is equivalent to $\{x_{i[j]} = \hat{x}_{i[j]}, \hat{x}_{i[j]} \geq 0\}$. Note however that Planet does not include any heuristic to prioritise which decision variables should be split over. As a result, there is no mechanism that based on a heuristic search of a feasible point to encourage early termination in Planet.

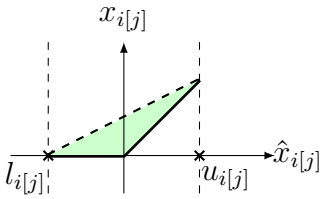


Figure 2.11: Feasible domain corresponding to the Planet relaxation for a single ReLU.

For satisfiable problems, only when a full complete assignment is identified is a solution returned. To detect incoherent assignments, Ehlers [28] introduces a global linear approximation to a neural network, which is used as a **bounding method** to over-approximate the set of values that each hidden unit can take. In addition to the existing linear constraints (Equations 2.19a, 2.19band 2.19c), the non-linear constraints are approximated by sets of linear constraints representing the convex hull of each nonlinearity treated independently.

Specifically, a ReLU with an input of unknown signs are replaced by a set of equations:

$$\mathbf{x_i} = \max(\hat{\mathbf{x}}_\mathbf{i}, 0) \;\Rightarrow\; \mathbf{x_i} \geq \hat{\mathbf{x}}_\mathbf{i} \tag{2.27a}$$

$$\mathbf{x_i} \geq 0 \tag{2.27b}$$

$$x_{i[j]} \leq u_{i[j]} \frac{\hat{x}_{i[j]} - l_{i[j]}}{u_{i[j]} - l_{i[j]}}. \tag{2.27c}$$

Recall that $x_{i[j]}$ corresponds to the value of the j-th coordinate of $\mathbf{x_i}$. An illustration of the convex hull is provided in Figure 2.11.

Compared with the relaxation of Reluplex Equation 2.26, Planet relaxation is tighter. Specifically, Equations 2.26a and 2.26b are identical to Equations 2.27a and 2.27b but Equation 2.27c implies Equation 2.26c. Indeed, given that $\hat{x}_{i[j]}$ is smaller than $u_{i[j]}$, the fraction multiplying $u_{i[j]}$ is necessarily smaller than 1, indicating that this provides a tighter bounds on $x_{i[j]}$.

To use this approximation to compute better bounds than the ones given by simple interval arithmetic, it is possible to leverage the feed-forward structure of the neural networks and obtain bounds one layer at a time. Having included all constraints up until the i-th layer (not including the i-th layer), it is possible to optimize over the resulting linear programming problem and obtain bounds for all

the units of the i-th layer, which in turn will allow us to create the constraints Equation 2.27 for the next layer.

In addition to the pruning obtained by the convex relaxation, both Planet and Reluplex make use of conflict analysis [64] to discover combinations of splits that cannot lead to satisfiable assignments, allowing them to perform further pruning of (sub-)problems.

Toy Example (Running Planet) *Planet first computes initial bounds via interval arithmetic for nodes a, b and y. Then it builds a linear program to approximate the network by using the linear constraints Equations 2.27a, 2.27b and 2.27c. Upper and lower bounds of a, b and y are refined by calling an LP solver. In this toy example, after refinement, the lower bound and upper bound of y are replaced with* -3 *and* 5 *respectively. Since it is sufficient to prove the property with the refined lower bound, the algorithm exits before entering the main loop where a Satisifiability solver is called.*

In the next chapter, we will conduct a comprehensive evaluation of the available verification methods using the unified Branch and Bound framework. With the unified framework, we are also able to identify potential places for improvements and hence propose heuristics to address those deficiencies accordingly. Due to their high-level nature, these proposed strategies can be easily adopted by any verification method that falls under the framework.

2.5 Learning Neural Networks

Once a network architecture has been decided, we need to learn for the parameter $\boldsymbol{\theta}$, which are the weights and biases of the network. However, due to the large number of parameters and the nonlinearity of the network, it is often impossible to find the perfect solution. Instead, we cast the problem as an optimisation question and solve for a set of parameters that gives satisfactory performance. To be specific, we first design a loss function that we aim to minimize. We then update neural network

weights by performing gradient descent algorithms, which can be efficiently done with the help of GPUs. In the following, we focus on image classification tasks.

2.5.1 Learning for Nominal Accuracy

Assume $\mathcal{X} \subset \mathbb{R}^{|\mathcal{X}|}$ is an image data distribution with total C distinct categories. We need to learn $\boldsymbol{\theta}$ for a neural network $f(\cdot;\boldsymbol{\theta}) : \mathcal{X} \to \mathbb{R}^C$. Let $\boldsymbol{x}$ be an arbitrary image data point and c be its label. Also, let $\boldsymbol{y_x} \in \mathbb{R}^C$ be an one hot encoding vector with $y^c_{\boldsymbol{x}} = 1$. A typical choice of loss function for classification is the cross-entropy loss, defined as

$$\ell^{CE}\left(f(\boldsymbol{x};\boldsymbol{\theta})\,,\,c\right) = -\sum_{i=1}^{C} y^i_{\boldsymbol{x}} \log(p_i(\boldsymbol{x};\boldsymbol{\theta})), \tag{2.28}$$

where $p_i(\boldsymbol{x};\boldsymbol{\theta}) = \frac{\exp(f_i(\boldsymbol{x};\boldsymbol{\theta}))}{\sum_j \exp(f_j(\boldsymbol{x};\boldsymbol{\theta}))}$ is the probability of an image being class i. For simplified notations, we use $\ell^{CE}(\boldsymbol{x};\boldsymbol{\theta})$ to indicate $\ell^{CE}\left(f(\boldsymbol{x};\boldsymbol{\theta})\,,\,c\right)$.

We apply iterative optimisation algorithms to solve for

$$\arg\min_{\boldsymbol{\theta}} \; \mathbb{E}[\,\ell^{CE}(\boldsymbol{x};\boldsymbol{\theta})\,]. \tag{2.29}$$

Generally, we start the optimisation procedure by randomly initializing $\boldsymbol{\theta}$. In practice, for the minimization objective, we use the empirical average on a provided training dataset instead of the expectation of the whole data distribution.

2.5.2 Learning for Robust Accuracy

Although training with cross-entropy leads to high nominal accuracy, it is often found to result in close to 0 robust accuracy. Since parameter updates are guided by the loss function, new loss functions that specially designed to encourage the robust behaviour of networks are proposed.

Among these losses, the most popular ones are the adversarial training loss designed by Madry et al. [63] and TRADES presented by Zhang et al. [115]. Specifically, adversarial training loss solves for

$$\arg\min_{\boldsymbol{\theta}} \; \mathbb{E}[\max_{\boldsymbol{x}' \in \mathcal{B}^{\infty}_{\epsilon}(\boldsymbol{x})} \ell^{CE}(\boldsymbol{x}';\boldsymbol{\theta})\,] \tag{2.30}$$

and TRADES deals with the problem

$$\arg\min_{\boldsymbol{\theta}} \mathbb{E}[\, \ell^{CE}(\boldsymbol{x};\boldsymbol{\theta}) + \max_{\boldsymbol{x}' \in \mathcal{B}_\epsilon^\infty} \mathrm{KL}(\boldsymbol{p}(\boldsymbol{x};\theta) \| \boldsymbol{p}(\boldsymbol{x}';\theta))\,], \tag{2.31}$$

where KL stands for Kullback-Leibler divergence. In both losses, we need to solve for an inner maximization problem. We recall that this is precisely the problem PDG attacks are designed for. We thus use the iterative PGD algorithm to find $\boldsymbol{x}'$ in both cases.

We note that the quality of $\boldsymbol{x}'$ found by PGD depends on the total number of steps taken. In general, a certain number of PGD steps are required to ensure effective robust training. This could be prohibitively expensive when the input dimension is high and the network architecture is complicated. In Chapter 5, we adopt a new perspective on robust accuracy and propose a novel training algorithm that allows efficient robust training.

2.5.3 Datasets

Several standard image datasets have been created for image classification tasks. For consistency and comparability, we use the standard datasets in our study as well. Specifically, we focus on MNIST [58], CIFAR10 [54], CIFAR100 [54].

MNIST is a collection of handwritten single digits from 0 to 9, so there are 10 classes in total. The dataset consists of a training dataset with 6,000 images per class and a test dataset with 1,000 images per class. It is a relatively easy dataset, as all images are small in size (28 by 28 pixel) and are grayscale (single colour channel). Sample images can be found in Figure 2.12.

Figure 2.12: Sample images of MNIST dataset.

CIFAR10 is composed of 10 classes 32 by 32 pixel colour images (three colour channels). For each class, there are 5000 training images and 1000 test images.

Some sample images are shown in Figure 2.13. **CIFAR100** is almost the same as CIFAR10 but with 100 classes, so there are 500 training images and 100 test images for each class.

Figure 2.13: Sample images from CIFAR10 dataset.

3

Branch and Bound for Formal Verification

Contents

3.1 Introduction

In this chapter, we improve on Branch and Bound based neural network verification. We work with the general Branch and Bound framework directly. By doing so, we identify high-level improvements that are readily applicable to all methods under the framework. In addition, we introduce several new datasets, each of which

focuses on testing a distinct perspective of all verification methods, to perform a comprehensive study. We report the detailed analyses in the final section.

3.2 Contributions

We summarise the main contributions made within the chapter below.

1. By directly working with the general Branch and Bound framework, we identified high-level improvements for the bounding and the branching components. In terms of bounding, we suggested a problem-dependent re-approximation to balance the tradeoff between tightness and computational cost. When branching is considered, we proposed new input branching heuristic and ReLU activation branching heuristic. We demonstrated the effectiveness of our strategies on various datasets.

2. We introduced comprehensive data sets consisting of trained as well as synthetic networks with fully connected and/or convolutional layers. At the time of the study, the only exisiting datasets are on small networks with fully-connected layers. However, convolutional net-works are widely used in computer vision tasks and should be an indispensable component for fair and complete evaluations of verification methods. Based on this need, we constructed datasets **Robust MNIST** and **reduced Robust MNIST** on medium sized convolutional networks. We note that unlike the general set-up using the same ϵ value for the allowed perturbation, we use binary search to pick a suitable value of ϵ. Our decision is supported by two observations. Firstly, the difficulty of a verification property relies not only on the size of the network, but also on the value of ϵ. Secondly, one bottleneck for BaB based methods is the time required for solving linear programs (LPs), which could increase significantly with the size of the network. Constrained by the LP, we deal with medium-sized convolutional networks and adjust the difficulty level of verification properties through ϵ to have a broad coverage.

3. We provided suggestions on how to choose branching and bounding components when dealing with problems with special structure. To study problems with special structure, we introduced the synthetic **PCAMNIST** and **TwinStream**. **PCAMNIST** consists of properties on networks with varying input dimension, depth and width. **TwinStream** is constructed on networks with highly correlated layers. By evaluating verification methods on these special networks, we gained a deep understanding of the dynamics between the components of the BaB framework. As a result, we give guidance on how to best utilize the framework to accommodate the problem at hand.

In the following, we will first illustrate our improvements on the bounding and branching component respectively. We will give more details on the datasets the Section 3.5.2. All proposed datasets can be found on `https://github.com/JingyueLu/PLNN-verification-journal`.

3.3 Better Bounding

Recall that $\boldsymbol{l}_0$ and $\boldsymbol{u}_0$ are the lower and upper bounds for an input $\boldsymbol{x}$. We also denote an intermediate layer lower bound as $\boldsymbol{l}_i$ and an upper bound as $\boldsymbol{u}_i$. Bounding depends on the underlying convex relaxation applied. In Chapter 2, we have introduced relaxations employed by interval arithmetic, Reluplex and Planet. Apart from those, we mention a new type of relaxation, proposed by Weng et al. [99]. As shown in Figure 3.1, this relaxation uses two parallel lines, determined by the upper and lower bounds of the pre-activation unit $\hat{x}$, to present the upper and lower bounds for the post-activation unit x. We will use this relaxation for computing bounds in some of our methods. We point out the final formulas for computing bounds based on this relaxation are exactly the same as those used by Kolter and Wong [100], although Kolter and Wong derived their bounds from a completely different idea. Among all proposed convex relaxations, Planet [28] is the tightest and should therefore give the best bounds while interval arithmetic is the cheapest computationally.

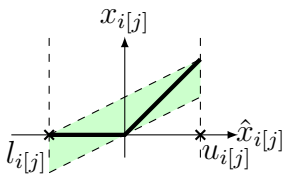


Figure 3.1: Feasible domain corresponding to the relaxation introduced in [99] for a single ReLU.

So far, many verification methods compute intermediate bounds once at the beginning of the verification process. We improve the bounding component of the Branch and Bound framework by constructing a new and tighter intermediate bounds approximation for each sub-problem generated after a split. The key idea is to incorporate the constraints from the split. For instance, on each sub-problem, we can refine all the $\mathbf{l_i}, \mathbf{u_i}$ bounds by formulating corresponding linear programming problems with added constraints from the split. Then, we solve for minimum for $l_{i[j]}$ and maximum for $u_{i[j]}$ in a layer by layer order. With refined upper and lower bounds, we are able to introduce smaller convex relaxation hulls to obtain tighter relaxations for later layers. We show the importance of this in the experiments section with the **reluBaB** method that performs splitting on the activation like Planet but updates its intermediate bounds approximation completely at each sub-problem.

However, it should be noted there is a trade-off between the benefits of tighter relaxation and the overall computational efficiency. Since updating all the $\mathbf{l_i}, \mathbf{u_i}$ bounds could be computationally expensive, we show in the experiments sections that, depending on the problem at hand, it is sometimes sufficient to update bounds for some of the layers. For other layers, bounds of Weng et al. [99] or equally Kolter and Wong [100], which are computational cheap but outperforms those of interval arithmetic, could be used. The overall gain from a tighter relaxation is not in a linear relationship with the number of intermediate bounds updated.

3.4 Better Branching

In the following, we discuss two possible ways to improve branching strategies.

3.4.1 Branching on Input Domains

Both Planet and Reluplex adopt the decision to split on the activation of the ReLU nonlinearities. Although the decision is intuitive as it provides a clear set of decision variables to fix, these methods ignore another natural branching strategy, namely, splitting on the input domain. There are two main advantages of input domain splitting strategies. Firstly, it is simple and straightforward to apply. Once an input dimension to split on is decided, we only need to modify the associated input constraints for each subdomain, generated by the split step of the BAB algorithm. There is no need to deal with potential conflicts (e.g. infeasible sub-problem) that could be introduced by fixing a ReLU node. Secondly, it can be argued that since the function encoded by the neural networks are piecewise linear in their input, splitting the input domain could result in the computation of high quality upper and lower bounds. With tighter input bounds, tighter intermediate bounds at all layers can be easily re-evaluated, which might not be the case for splitting on a ReLU node, at least for layers prior to the ReLU node we branch on.

To demonstrate the benefits of input domain splitting, we propose two novel input split algorithms. We will show in the experiments sections that domain splitting strategies incorporating our proposed heuristics are highly effective for small scale verification problems with low dimensional input. The first and the most direct algorithm is **BaB** method. Based on a domain with input constrained by $\boldsymbol{l}_0 \leq \boldsymbol{x} \leq \boldsymbol{u}_0$, the `split` function would return two subdomains where bounds would be identical in all dimension except for the dimension with the largest length, denoted $i^\star$. The bounds for each subdomain for dimension $i^\star$ are given by $l_{0[i^\star]} \leq x_{0[i^\star]} \leq \frac{l_{0[i^\star]}+u_{0[i^\star]}}{2}$ and $\frac{l_{0[i^\star]}+u_{0[i^\star]}}{2} \leq x_{0[i^\star]} \leq u_{0[i^\star]}$.

In order to exploit the benefits of input domain splitting to the fullest, we introduce the second splitting heuristic by using highly efficient lower bound computation approaches developed through dual methods. Recall that in Chapter 2, we have

showed that verification problem can be formulated as an optimization problem:

$$\min \hat{x}_L \qquad \text{subject to} \tag{3.1a}$$

$$\mathbf{x} \leq \mathbf{u_0} \tag{3.1b}$$

$$\mathbf{x} \leq -\mathbf{l_0} \tag{3.1c}$$

$$\hat{\mathbf{x}}_{\mathbf{i+1}} = W_{i+1}\mathbf{x_i} + \mathbf{b_{i+1}} \qquad \forall i \in \{0,\ L-1\} \tag{3.1d}$$

$$\mathbf{x_i} = \max\left(\hat{\mathbf{x}}_\mathbf{i}, 0\right) \qquad \forall i \in \{1,\ L-1\}. \tag{3.1e}$$

To convexify the nonlinear ReLU, we could use any feasible convex relaxation. We adopt the Planet convex relaxation here and, for ambiguous ReLUs, replace the equality $x_{i[j]} = \max\left(\hat{x}_{i[j]}, 0\right)$ with inequality constraints

$$x_{i[j]} \geq \hat{x}_{i[j]} \tag{3.2a}$$

$$x_{i[j]} \geq 0 \tag{3.2b}$$

$$x_{i[j]} \leq u_{i[j]} \frac{\hat{x}_{i[j]} - l_{i[j]}}{u_{i[j]} - l_{i[j]}}. \tag{3.2c}$$

We note that for fixed ReLUs, we can simply replace $x_{i[j]} = \max\left(\hat{x}_{i[j]}, 0\right)$ with $x_{i[j]} = 0$ $(u_{i[j]} < 0)$ or $x_{i[j]} = \hat{x}_{i[j]}$ $(l_{i[j]} > 0)$ respectively. By associating dual variables with each of the constraints, it is straightforward to deduce the dual form of the above convexified optimization problem:

$$\max_{\alpha} l^L, \qquad \text{subject to } \forall \alpha_{k[j]} \in [0,1] \tag{3.3}$$

where

$$l^L = -\sum_{k=1}^{L} \nu_{k+1}^T b_k - \mathbf{u}_0^T[\hat{\nu}_1]_+ + \mathbf{l}_0^T[\hat{\nu}_1]_- + \sum_{k=2}^{L}\sum_{j\in\mathcal{I}_i} \frac{u_{k[j]} l_{k[j]}}{u_{k[j]} - l_{k[j]}}[\hat{\nu}_{k[j]}]_+$$

$$\nu_{L+1} = -1$$

$$\hat{\nu}_k = W_k^T \nu_{k+1}, k = L, \ldots, 1$$

$$\nu_{k,j} = \begin{cases} 0 & \text{if } u_{k[j]} > 0 \quad (j \in \mathcal{I}_k^-) \\ \hat{\nu}_{k[j]} & \text{if } l_{k[j]} > 0 \quad (j \in \mathcal{I}_k^+) \\ \frac{u_{k[j]}}{u_{k[j]} - l_{k[j]}}[\hat{\nu}_{k[j]}]_+ - \alpha_{k[j]}[\hat{\nu}_{k[j]}]_- & \text{otherwise} \quad (j \in \mathcal{I}_k) \end{cases}$$

$$\text{for } k = n, \ldots, 2.$$

Here, $[v]_-, [v]_+ \geq 0$ represent negative and positive parts of an element v respectively. The term I_k^- denotes the set of activation nodes whose upper bounds are negative while I_k^+ contains activation nodes whose lower bounds are positive. The rest of the activation nodes belong to I_k. For all possible values of α, l^L is a valid lower bound. Therefore, instead of solving for the optimal α values, we can use pre-fixed values for α to obtain a lower bound. This approach is computationally efficient as one computation of l^L is equivalent to one backward pass due to the recursive nature of v_k and $\hat{v}_k$. Here, we have directly used all intermediate bounds $u_{k[j]}$ and $l_{k[j]}$. These intermediate bounds are actually computed beforehand in a similar fashion with above formula. Each $l_{k[j]}$ can be treated as a lower bound to a sub-network consisting of layers prior to it and $u_{k[j]}$ are obtained by negating the signs of the sub-network. Given the input constraints, we proceed layer by layer to compute all intermediate bounds $u_{k[j]}$ and $l_{k[j]}$. Using the dual form to obtain efficient lower bounds has been adopted in various studies of verification methods. However, different studies choose different values for α: Kolter and Wong [100] and Weng et al. [99] compute the lower bound by setting $\alpha_{k[j]} = \frac{u_{k[j]}}{u_{k[j]} - l_{k[j]}}$ while Zhang et al. [116] sets $\alpha_{k[j]} = 0$ for $|l_{k[j]}| > u_{k[j]}$ and $\alpha_{k[j]} = 1$ for $|l_{k[j]}| \leq u_{k[j]}$.

With the dual formulation and predetermined values of α, we are able to efficiently compute lower bound estimations $l_{i_0}^L$ and $l_{i_1}^L$ of subdomains sub_{i_0} and sub_{i_1} respectively. For clarity, $l_{i_0}^L$ is a lower bound estimation of the minimum output of the network **net** (see Algorithm 1) can reach on sub_{i_0}. Finally, we make the split decision by choosing the dimension that improves the domain's lower bound the most after the split, which is the solution of $\arg\max_i(\min(l_{i_0}^L, l_{i_1}^L))$. Here, we have used the minimum of $l_{i_0}^L, l_{i_1}^L$ to represent the improvement achieved over the split on a input dimension. It is possible to replace it with other criteria such as the $\max(l_{i_0}^L, l_{i_1}^L)$ or the product of $l_{i_0}^L$ and $l_{i_1}^L$, as used in Khalil et al. [52] in the context of other MIP problems.

Once a domain splitting decision is made, two subdomains are generated. On each subdomain, we use an LP solver to refine intermediate bounds, as mentioned in Section 3.3. Finally, we call the LP solver again to compute the domain lower

bound (`prob_lb`). Performance of **BaB** and **BaBSB** are included in the experiments section. In this study, we have followed [100] and [99] by setting $\alpha_{k[j]} = \frac{u_{k[j]}}{u_{k[j]} - l_{k[j]}}$ for all dual formulation computations. Other choices of α could also be used.

3.4.2 Branching on ReLU Activation Nodes

Despite their success in small-scale verification problems, input domain splitting methods are often found to be inadequate for large-scale networks, as there are several limitations of input domain branching strategies. Firstly, for some methods (e.g. **BaBSB**), their computation cost for making a branching decision increases at least linearly with input dimensions. High computational cost renders these methods infeasible for high input dimensional problems. Secondly and more importantly, potential input branching decisions constitute a fairly small portion of all potential branching decisions for large-scale network architecture, which contains a sizable number of unfixed ReLU activation nodes (each is a valid potential branching point). Focusing on input domain splitting alone significantly limits the power of verification methods. Given the wide and almost dominant usage of deep convolutional networks in various tasks, developing an effective and computationally cheap heuristic for branching on ReLU nonlinearities is of considerable importance for verification.

In this section, we propose a new smart branching algorithm **BaBSR** on the activation of the ReLU non-linearities. So far, to the best of our knowledge, existing ReLU node splitting methods are **Reluplex**, **Planet** and **Neurify**. We show that **BaBSR** enjoys various benefits over the existing methods, although all methods use the same **branching rule**: for a given ReLU node (a node $x_{i[j]} = \max\left(\hat{x}_{i[j]}, 0\right)$ is split out into two subdomains:$\{x_{i[j]} = 0, \hat{x}_{i[j]} \leq 0\}$ and $\{x_{i[j]} = \hat{x}_{i[j]}, \hat{x}_{i[j]} \geq 0\}$). To start, unlike **Planet** which does not have any heuristic to make splitting decisions, **BaBSR** uses a simple and fast heuristic to prioritise which unfixed ReLU node to split on. **Reluplex** uses the SMT core to handle the splitting order and **Neurify** computes gradient scores to prioritise ReLU nodes. We show in experiments that the prioritising strategy of **BaBSR** is much more successful than that of **Reluplex** and **Neurify** in selecting an effective ReLU node. Additionally, **BaBSR** has a convergence

guarantee and encourages early termination, which means verification problems can be solved completely and efficiently. Once a ReLU node has been selected, **BaBSR** calls a commercial solver to obtain a lower bound for each sub-problem with the added constraint $\hat{x}_{i[j]} \leq 0$ or $\hat{x}_{i[j]} \geq 0$ respectively. The algorithm continues until line 5 in Algorithm 1 is satisfied. Convergence guarantee is inherently supported by the algorithm while early termination through finding adversarial examples is assisted by the prioritisation we used in generating sub-problems.

The heuristic used for prioritising ReLU nodes is based on the similar idea to the one used in **BaBSB**. For an arbitrary picked-out sub-problem, we refer to the lower bound of the network minimum on the sub-problem as l^L. To decide which unfixed ReLU nodes to split on, for each potential splitting option (any unfixed ReLU node), we attempt to compute a rough estimate of the potential improvement to the lower bound. We then make the splitting decision by choosing the unfixed node with the largest estimated improvement.

In detail, we estimate the improvement on splitting an arbitrary unfixed ReLU node via a modified application of Equation 3.3. We first observe that when imposing a constraint on an arbitrary unfixed ReLU node $x_{i[j]}$, we will force $\nu_{i[j]}$ to be either 0 (ReLU is in a blocking state) or $\hat{\nu}_{i[j]}$ (ReLU is in a completely passing state). As a direct result, the associated terms $\nu_{i[j]}b_{i-1[j]}$ and $\frac{u_{i[j]}l_{i[j]}}{u_{i[j]}-l_{i[j]}}[\hat{\nu}_{i[j]}]_+$ in Equation 3.3 will change accordingly. Furthermore, ν_k, $\hat{\nu}_k$ for $k < i$ as products of ν_i will take different values as well and so do their associated terms in Equation 3.3. It is possible to compute lower bounds l^L by assuming $\nu_{i[j]} = 0$ and $\nu_{i[j]} = \hat{\nu}_{i[j]}$ and then take the minimum (or maximum, product etc.) of the two cases to represent the improvement made if splitting on $x_{i[j]}$. However, doing this would require two full or partial (only ν_k, $\hat{\nu}_k$ for $k < i$ need to be updated) backward passes for one ReLU splitting choice, which can be computationally expensive when the number of unfixed ReLU nodes is large. To deal with this issue, we use a key observation when dealing with different datasets: when the weights (similar for bias) on each layer are of same magnitudes, the potential improvement of each $x_{i[j]}$ is generally dominated by the changes of terms $\nu_{i[j]}b_{i-1[j]}$ and $\frac{u_{i[j]}l_{i[j]}}{u_{i[j]}-l_{i[j]}}[\hat{\nu}_{i[j]}]_+$. Since a rough

estimation is sufficient in this scenario, we thus propose to evaluate each ReLU split choice $x_{i[j]}$ by computing a ReLU score $s_{i[j]}$, defined as

$$s_{i[j]} = \left| \min \left(\frac{u_{i[j]}}{u_{i[j]} - l_{i[j]}} \hat{\nu}_{i[j]} b_{i-1[j]},\ (\frac{u_{i[j]}}{u_{i[j]} - l_{i[j]}} - 1) \hat{\nu}_{i[j]} b_{i-1[j]} \right) - \frac{u_{i[j]} l_{i[j]}}{u_{i[j]} - l_{i[j]}} [\hat{\nu}_{i[j]}]_+ \right|. \tag{3.4}$$

We decide by picking the ReLU with the largest score. One major benefit of this heuristic is that it is highly computationally efficient. As the same v_i and $\hat{v}_i$, for $1 < i < L - 1$, are used for computing all unfixed ReLUs scores, the recursive formulations of v_i and $\hat{v}_i$ enable us to compute all ReLU scores within one single backward pass, regardless the total number of unfixed ReLU nodes.

To maximize the performance of the heuristic, we have also incorporated into the heuristic other useful observations. Firstly, we refer to a convolution layer as sparse if it contains mostly zeros when linearlized. We found that splitting on the sparsest layer is often ineffective in terms of improving the global lower bound when compared to choices on other layers. In addition, ReLU scores are likely to fail in giving good indications when all of them are close to zero. Thus, given a network that contains a large convolution layer with a small kernel, we do not consider the unfixed ReLU nodes on this layer until all other improvements computed are relatively small. When this happens, we consider the heuristic used in prioritising ReLU nodes to be no longer effective. Hence, other selecting strategies should be used, such as random selection with a preference over non-sparse layers.

Finally, unlike **BaBSB**, **BaBSR** does not call an LP solver to compute tight intermediate bounds. The main applications of **BaBSR** should be networks with large convolution layers. For these networks, computing tight intermediate bounds via an LP solver is computationally infeasible. Thus, once a ReLU split choice is made, we explicitly replace the upper or lower bound of the corresponding ReLU node to 0 for each newly generated sub-problem and then update intermediate bounds by the better of interval arithmetic bounds and bounds computed using the method of Wong and Kolter [100]. Overall, with rough improvement estimations of ReLU choices and loose intermediate bounds, **BaBSR** is significantly cheap for each

branching step. While many more branches might be required to solve a property, we often find this trade-off is worthwhile as will be seen in the experiments sections.

3.5 Experiments: General Setup

The problem of PL-NN verification has been shown to be NP-complete [50]. Meaningful comparison between approaches therefore needs to be experimental. Before we conduct a comprehensive evaluation of our proposed methods and other available ones, we introduce our experimental setup in the following.

3.5.1 Methods

The simplest baseline we refer to is **BlackBox**, a direct encoding of Equation 2.19 into the Gurobi solver, which will perform its own relaxation, without taking advantage of the problem's structure.

For the SMT based methods, **Reluplex** and **Planet**, we use the publicly available versions [29, 49]. Both tools are implemented in C++. We wrote software to support conversion between the input formats of both solvers in both directions. However, it is worth noting that we do not change the underlying GLPK solver for linear programming. All the other methods use a potentially faster Gurobi LP solver. The reader is reminded to take this key difference into account when studying our results.

We also evaluate the potential of using MIP solvers, based on the formulation of Equation 2.23. Due to the lack of availability of open-sourced methods at the time of our experiments, we reimplemented the approach in Python, using the Gurobi MIP solver. We report results for a variant called **MIPplanet**, which uses bounds derived from Planet's convex relaxation rather than simple interval arithmetic. Both the **MIPplanet** and **BlackBox** are not treated as simple feasibility problems but are encoded to minimize the output $\hat{x}_L$ of Equation 2.19b, with a callback interrupting the optimization as soon as a negative value is found.

In addition, we include the methods derived from our Branch-and-Bound analysis. Our implementation follows Algorithm 1 faithfully, is implemented in Python, and uses Gurobi to solve LPs. The `pick_out` strategy consists in prioritising the

(sub-)problem that currently has the smallest lower bound. Upper bounds are generated by randomly sampling points on the considered (sub-)problem or directly computing the network value at the lower bound solution provided by Gurobi, for which we use the convex approximation of Ehlers [28] to obtain lower bounds. As suggested by the better bounding Section 3.3, we update the linear approximation for sub-problems instead of using a single approximation generated at the beginning (e.g., Planet [28]). Bearing in mind the trade-off between the benefits of tighter relaxation and computational costs, we rebuild the approximation via calling an LP solver to recompute none or partial or all intermediate bounds for each sub-problem. This decision should be made on a case by case basis depending on the size of a network architecture, the number of input dimensions, and the magnitude and correlation of layer weights. To better study the trade-off, we have incorporated different approximation strategies into different algorithms and compared them on several datasets.

For `split`, we focus on two types of split: input domain split and ReLU activation split. In each case, we consider naive split methods and improved versions. Specifically, in terms of input domain split, the naive methods (e.g. **BaB**) simply performs branching by splitting the input domain in half along its longest edge, while the improved methods (e.g. **BaBSB**) does it by splitting the input domain along the dimension that gives the estimated best improvement to the global lower bound. Estimations are made through the fast bounds formula provided in Wong and Kolter [100]. Regarding the ReLU node split, we study methods which always split on the first or a random unfixed ReLU node on the first layer containing unfixed ReLU nodes. More advanced methods (e.g. **BaBSR** and **BaBSRL**) prioritize ReLU nodes through the criteria introduced in Equation 3.4. Each specific method is a combination of the choice of approximation strategies (how to rebuild approximations) and branching strategies (what branching heuristics to use). We summarize all Branch-and-Bound methods considered in Table 3.1. We use abbreviations **imb** for intermediate bounds and `prob_ub` and `prob_lb` are the same as those defined in Algorithm 1.

Method	Branching Type	Branching Heuristics	Approximations
BaBSB	**Input**	fast bounds Wong and Kolter [100]	**imb:** LP solver `prob_lb` LP solver `prob_ub` random sampling
BaB		longest edge	**imb:** LP solver `prob_lb` LP solver `prob_ub` random sampling
reluBaB	**ReLU**	the first unfixed ReLU node on the first layer containing unfixed ReLU nodes	**imb:** LP solver `prob_lb` LP solver `prob_ub` random sampling
BaBSR		prioritization via the criteria defined in Equation 3.4	**imb:** better of interval bounds and bounds of Wong and Kolter [100] `prob_lb` LP solver `prob_ub` solution of the LP
BaBSRL		prioritization via the criteria defined in Equation 3.4	**imb:** update the bounds of the layer right before the ReLU node selected via an LP solver; update the rest of the layers via better of interval bounds and bounds of Wong and Kolter [100] `prob_lb` LP solver `prob_ub` solution of the LP

Table 3.1: All Branch-and-Bound methods considered in the experiments section.

3.5.2 Datasets

In order to obtain an inclusive analysis, we perform verification on six datasets of properties. Among them, 2 datasets are existing datasets directly adopted from previous work and the other 4 datasets are newly proposed datasets by us. For the newly proposed datasets, **Robust MNIST** and **reduced Robust MNIST** are considerably larger than the existing datasets and start to include convolutional layers. **PCAMNSIT** and **TwinStream** are the first to allow us to study varying structures of networks.

The **CollisionDetection** dataset [28] attempts to predict whether two vehicles with parameterized trajectories are going to collide. 500 properties are extracted from problems arising from a binary search to identify the size of the region around the training examples in which the prediction of the network does not change. The network used is relatively shallow but due to the process used to generate the properties, some lie extremely close to the decision boundary between satisfiable (SAT) and unsatisfiable (UNSAT). Recall that satisfiable refers to properties that are false (a counterexample found) while unsatisfiable refers to properties that are true (no counterexample exists).

The **Airborne Collision Avoidance System (ACAS)** dataset, as released by Katz et al. [50] is a neural network based advisory system recommending horizontal manoeuvres for an aircraft to avoid collisions, based on sensor measurements. Each of the five possible manoeuvres is assigned a score by the neural network and the action with the minimum score is chosen. The 188 properties to verify are based on some specifications describing various scenarios. Networks used in ACAS are deeper than those employed by CollisionDetection dataset, but both contain fully connected layers only.

The **Robust MNIST Network** is created based on the work of Wong and Kolter [100]: we use the same network architecture and the network is trained with their proposed robust training strategies. The network contains 2 convolution layers followed by 2 fully connected layers with a total number of 4804 activation nodes. Since on a network of this size each LP requires more than 2 seconds, the total number of branches that could be taken within a timeout is low. To better evaluate the performance of the branching heuristic used in **BaBSR**, we also introduce a reduced version, the **reduced Robust MNIST Network**. The reduced network has the same structure as the original one but fewer hidden nodes on each hidden layer. The total number of ReLU nodes in the reduced network is 1226. On the reduced network, each LP requires only 0.17 seconds, which allows a large number of branching decisions to be made before timeout. For MNIST networks, the natural properties to verify are whether the predicted label changes if each input image is allowed to be perturbed within an ϵ-infinity norm ball. Since each combination of an image in the MNIST test set and an ϵ constitute a valid property, we randomly select test images and verify properties at a set of pre-specified epsilons ranging from 0.14 to 0.175 for the Robust MNIST Network and from 0.11 to 0.14 for the reduced Robust MNIST Network. Note that, on the same network, epsilon values determine the difficulty level of verification properties. We use a set of epsilon values for both networks to allow comprehensive evaluations. Higher value of ϵ is used for the large network, as the network is more robust. Due to the large number of properties, we restrict the timeout to be one hour on these two datasets.

The above datasets do not allow us to explore the impact of various problem/model parameters such as depth, number of hidden units, input dimensionality, and correlation between hidden nodes on the same layer. As a result, we propose datasets, **PCAMNIST** and **TwinStream** to remove this deficiency. We give a detailed introduction to each of them below.

The **PCAMNIST** is a novel dataset that we introduce to get a better understanding of what factors influence the performance of various methods. It is generated in a way to give control over different architecture parameters. The networks take k features as input, corresponding to the first k eigenvectors of a Principal Component Analysis decomposition of the digits from the MNIST dataset. We also vary the depth (number of layers), width (number of hidden unit in each layer) of the networks. We train a different network for each combination of parameters on the task of predicting the parity of the presented digit. The properties that we attempt to verify are whether there exists an input for which the score assigned to the odd class is greater than the score of the even class plus a large confidence. By tweaking the value of the confidence in the properties, we can make the property either true or false, and we can choose by how much is it true. This gives us the possibility of tweaking the "margin", which represent a good measure of the difficulty of a network.

The **TwinStream** dataset consists of networks containing two separate streams, where each of the streams has the same architecture, weights, and inputs. The final layer of the network computes the difference between the outputs of the two streams and adds a positive bias term, which we will refer to as the margin, denoted as m. As a result, the output is always equal to the value of the final bias. We give explicit formulations of the weights and biases of the TwinStream networks. Given a N-layer stream with weights $W_1^s, \ldots, W_N^s$ and biases $b_1^s, \ldots, b_{N-1}^s$, the corresponding N-layer TwinStream network consists of following weights $W_1, \ldots, W_N$ and biases $b_1, \ldots, b_N$:

$$W_1 = \begin{bmatrix} W_1^s \\ W_1^s \end{bmatrix}, \qquad W_i = \begin{bmatrix} W_i^s & \mathbf{0} \\ \mathbf{0} & W_i^s \end{bmatrix} \quad \text{for } i \in \{2, \ldots, N-1\}, \qquad W_N = \begin{bmatrix} W_N^s & -W_N^s \end{bmatrix}.$$

$$b_i = \begin{bmatrix} b_i^s \\ b_i^s \end{bmatrix} \quad \text{for } i \in \{2, \ldots, N-1\}, \qquad b_N = m.$$

On each of those networks, we attempt to prove the property that the output of the network is positive. We generate streams with random weights using Glorot initialisation [35]. Various TwinStream Networks are constructed by varying the depth, the number of hidden units in each of the streams, the number of inputs, and the value of the margin. Note that as opposed to the other datasets, the weights are not the result of an optimization process and therefore may not be representative of real use cases. The TwinStream dataset is specially designed such that the hidden nodes on the same layer are highly correlated. It allows us to explore the trade-off between different bounding strategies.

Finally, we summarize in Table 3.2 the characteristics of all of the datasets used for the experimental comparison.

3.5.3 Evaluation Criteria

For each of the datasets, we compare different methods using the same protocol. We attempt to verify each property with a timeout of two hours (with an exception of one-hour timeout for the reduced Robust Network experiment and the Robust Network experiment due to the large number of properties), and a maximum allowed memory usage of 20GB, on a single core of a machine with an i7-5930K CPU. We measure the time taken by the solvers to either prove or disprove the property. If the property is false and the search problem is therefore satisfiable, we expect the solver to exhibit a counterexample. If the returned input is not a valid counterexample, we do not count the property as successfully proven, even if the property is indeed satisfiable. All code and data necessary to replicate our analysis have been released.

3.6 Experiments: Results and Analyses

We perform an ablation study to evaluate the performance of different methods on various datasets.

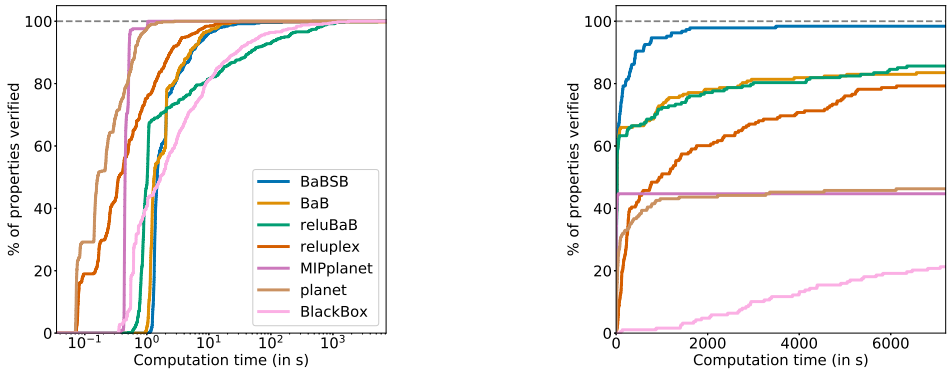
Data set	Count	Model Architecture
Collision Detection	500	6 inputs 40 hidden unit layer, MaxPool 19 hidden unit layer 2 outputs
ACAS	188	5 inputs 6 layers of 50 hidden units 5 outputs
reduced ROBUST MNIST	1200	28 by 28 inputs Conv2d(1,4,4, stride=2, padding=1) Conv2d(4,8,4, stride=2, padding=1) linear layer of 50 hidden units linear layer of 10 hidden units $\mathcal{L}_\infty$ ball radius selected are {0.11, 0.115, 0.12, 0.125, 0.127, 0.13, 0.14}
ROBUST MNIST	1000	28 by 28 inputs Conv2d(1,16,4, stride=2, padding=1) Conv2d(16,32,4, stride=2, padding=1) linear layer of 100 hidden units linear layer of 10 hidden units $\mathcal{L}_\infty$ ball radius selected are {0.14, 0.15, 0.155, 0.16, 0.165, 0.17, 0.175}
PCAMNIST	27	10 or {5, 10, 25, 100, 500, 784} inputs 4 or {2, 3, 4, 5, 6, 7} layers of 25 or {10, 15, 25, 50, 100} hidden units, 1 output, with a margin of +1000 or {-1e4, -1000, -100, -50, -10, -1 ,1, 10, 50, 100, 1000, 1e4}
TwinStream	81	{5, 10, 25} inputs {2, 4, 5} layers of {5, 10, 25} hidden units, 1 output, with a margin of {1e2, 1, 10}

Table 3.2: Details of all the datasets. For PCAMNIST, we use a base network with 10 inputs, 4 layers of 25 hidden units and a margin of 1000. We generate new problems by changing one parameter at a time, using the values inside the brackets.

3.6.1 Small Networks

We first consider small networks with low dimensional input. These are networks of CollisionDetection and ACAS. When networks are small, computing all intermediate bounds via an LP solver is computationally affordable. In these cases, gains from tighter relaxations are often significant and the tightest intermediate bounds should be used to achieve an ideal performance. As a result, methods like **BaBSR** and **BaBSRL** that employ rough estimated intermediate bounds are not included in

this section.

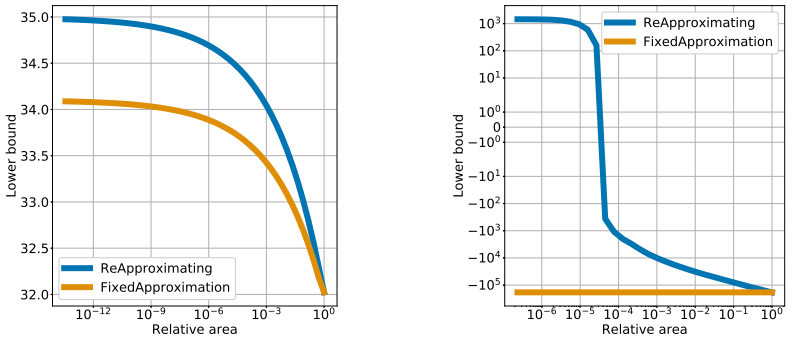


(a) CollisionDetection dataset. **(b)** ACAS dataset.

Figure 3.2: Proportion of properties verifiable for varying time budgets depending on the methods employed. A higher curve means that for the same time budget, more properties will be solvable. All methods solve CollisionDetection quite quickly except **reluBaB**, which is much slower and **BlackBox** which produces several incorrect counterexamples. On ACAS, **BaBSB** performs the best.

In Figure 3.2a, on the shallow networks of CollisionDetection, most solvers succeed on all properties in about 10 seconds. In particular, the SMT inspired solvers **Planet**, **Reluplex** and the MIP solver **MIPplanet** are extremely fast.

On the deeper networks of ACAS, in Figure 3.2b, no errors are observed, but most methods timeout on the most challenging testcases. The best baseline is **Reluplex**, which reaches 79.26% success rate at the two hour timeout, while our best method, **BaBSB**, already achieves 98.40% with a budget of one hour. To reach Reluplex's success rate, the required runtime is two orders of magnitude smaller. We are also able to identify the factors that allow our methods to perform well. We point out that the only difference between **BaBSB** and **BaB** is the smart branching, which represents a significant part of the performance gap. Furthermore, on networks with low dimensional inputs, branching over the ReLU activation nodes rather than over the inputs does not contribute much, as shown by the small difference between the naive input split method **BaB** and the naive ReLu split method **reluBaB**. The rest of the performance gap can be attributed to using better bounds: **reluBaB** significantly outperforms **planet** while using the same branching strategy and the same convex relaxations. This advantage of rebuilding the approximation at each



(a) Approximation on a CollisionDetection net.

(b) Approximation on a deep net from ACAS.

Figure 3.3: Quality of the linear approximation, depending on the size of the input domain. We plot the value of the lower bound as a function of the area on which it is computed (higher is better). The domains are centered around the global minimum and repeatedly shrunk. Rebuilding completely the linear approximation at each step allows to create tighter lower-bounds thanks to better $\mathbf{l_i}$ and $\mathbf{u_i}$, as opposed to using the same constraints and only changing the bounds on input variables. This effect is even more significant on deeper networks.

step is also confirmed by the results shown in Figure 3.3, which demonstrates the significant improvements re-approximation brings especially for deeper networks.

Figure 3.4 presents an additional analysis on a 20-property subset of the ACAS dataset, showing how the methods used impacts the need for branching. Smart branching and the use of better lower bounds reduce heavily the number of subdomains to explore. Given that there is a difference in the way verification works for SAT problems vs. UNSAT problems, we also report the comparison results on the subset of data sorted by decision type in Figure 3.5 and Figure 3.6.

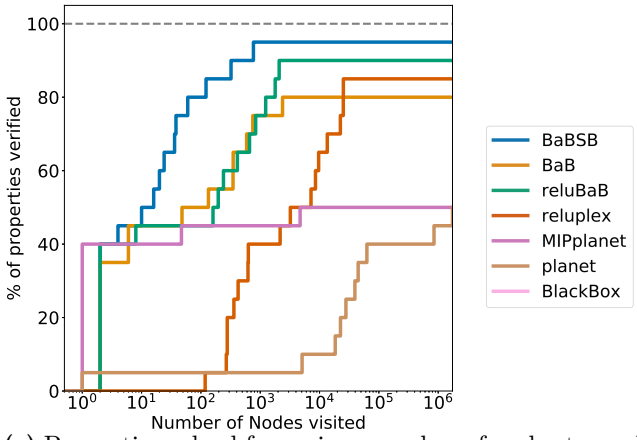
3.6.2 Large Networks

We then study the performance of various methods on the reduced Robust MNIST Network and the Robust MNIST Network. Due to the large number of properties (1200 and 1000 respectively to cover a wide range of difficulties), we treat **MIPplanet**[1] as the benchmark and rule out methods which could not perform at least at a similar level to **MIPplanet** over simple properties, that is, those that could be

[1]For large networks, calling an LP solver to compute all intermediate bounds in computationally expensive. To ensure a fair comparison between **BaBSR** and **MIPplanet**, the same intermediate bounds as that of **BaBSR** are used for building the LP model.

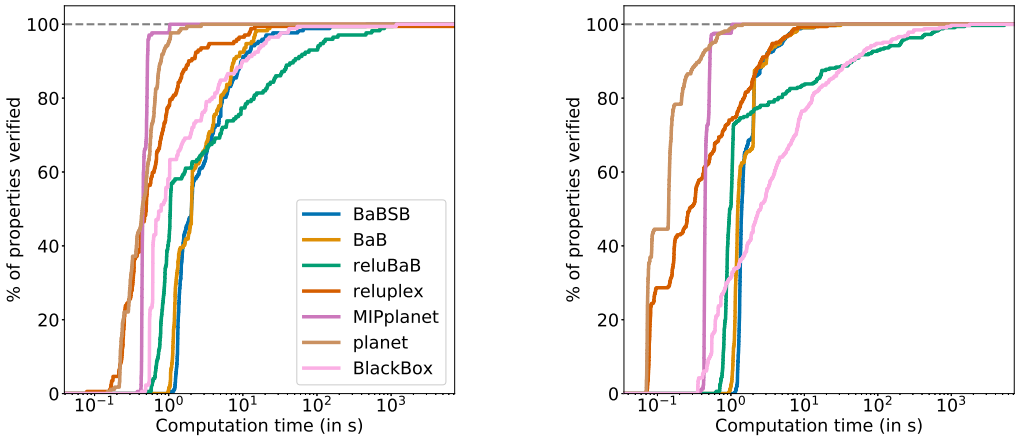
Method	Average time per Node
BaBSB	1.81s
BaB	2.11s
reluBaB	1.69s
reluplex	0.30s
MIPplanet	0.017s
planet	1.5e-3s

Table 3.3: Average time to explore a node for each method.



(a) Properties solved for a given number of nodes to explore (log scale).

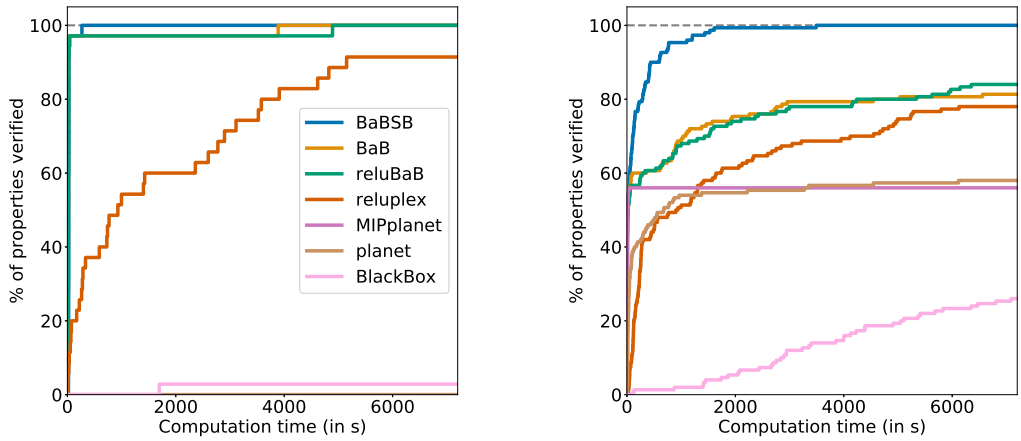
Figure 3.4: Trade-off between bounding cost and total number of branches required. Figure 3.4a shows how many subdomains needs to be explored before verifying properties while Table 3.3 shows the average time cost of exploring each subdomain. Our methods have a higher cost per node but they require significantly less branching, thanks to better bounding. Note also that between **BaBSB** and **BaB**, the smart branching reduces by an order of magnitude the number of nodes to visit.



(a) On SAT properties **(b)** On UNSAT properties

Figure 3.5: Proportion of properties verifiable by different methods under varying time budgets on the **CollisionDetection** data set. We can identify that all the errors that **BlackBox** makes are on SAT properties, as it returns incorrect counterexamples.

solved within 100 seconds by **MIPplanet**. For a comprehensive study of available complete methods, we have also included **ERAN**, ETH Robustness Analyser for neural networks. It is developed on a series of work [84–86] that apply abstract



(a) On SAT properties **(b)** On UNSAT properties

Figure 3.6: Proportion of properties verifiable by different methods under varying time budgets on the **ACAS** data set. We observe that **planet** doesn't succeed in solving any of the SAT properties, while our proposed methods are extremely efficient at it, even if there remains some properties that they can't solve.

interpretation for neural network verification. **ERAN**[2] mainly focuses on incomplete verification of MNIST, CIFAR10 and a subset of ACAS properties but also supports a complete mode, rendering itself a fair candidate for comparison studies. We observe from Figure 3.7 that most methods fail even on simple properties. **BaBSB** becomes incompetent when the input dimension is high. Although **planet**, **reluplex** and **BaBSR** use the same branching rule and loose bounds of different extent for approximation, the significantly improved performance of **BaBSR** is attributed to its effective ReLU prioritization heuristic and early termination feature. The **reluBaB** method is the only ReLU split method that uses the tightest bounds available throughout the procedure. However, the fact that it could not solve a single property reemphasizes the issue of finding a balance between the computational cost and the quality of the relaxation introduced. With limited computing resources, more gains might be achieved via a good branching strategy than a tighter relaxation.

We thus only compare among **BaBSR**, **ERAN** and **MIPplanet** on all properties of reduced Robust MNIST Network and the Robust MNIST Network. Results

[2]For our experiments, we have used the complete version of **ERAN** with refinepoly domain and 10 seconds MILP timeout, as suggested in Singh et al. [86]. All the rest of hyper-parameters are set as default values. Each process is restricted to a single cpu core.

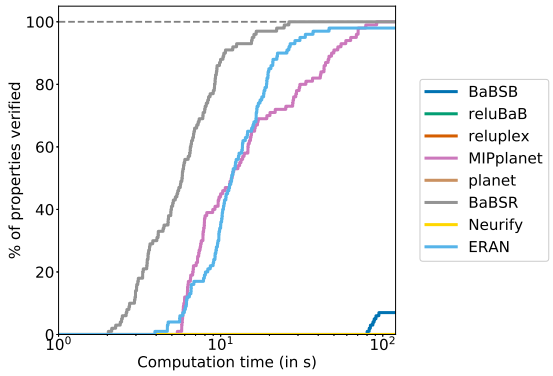
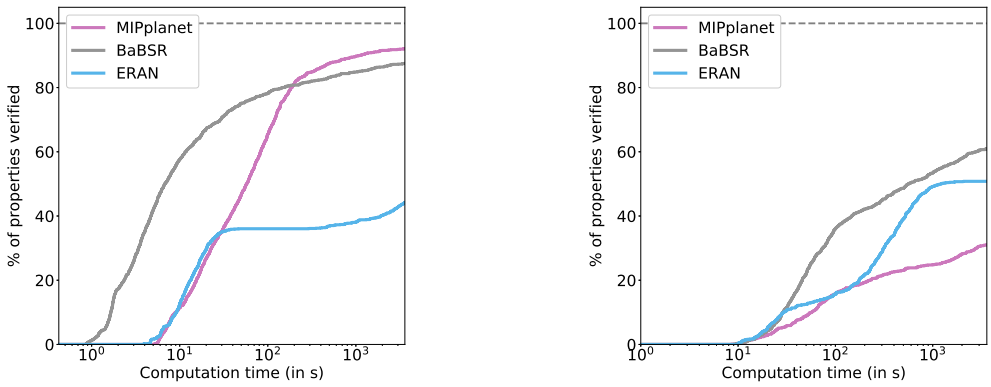


Figure 3.7: On simple properties, all **planet**, **reluBaB**, **reluplex** and **Neurify** timed out on every single property for 100s time limit. **BaBSB** managed to solve some properties but its performance is significantly worse than that of **MIPplanet**. Only **BaBSR** and **ERAN** perform better than and similarly to **MIPplanet** and are thus ran on all properties of the reduced Robust MNIST Network and the Robust MNIST Network.

are shown in Figure 3.8a and Figure 3.8b, respectively. Overall, **BaBSR** achieved the best performance on both datasets. In detail, when the reduced Robust MNIST network is considered, **BaBSR** outperforms **MIPplanet** significantly on easy properties, but the performance gap decreases when properties become more and more difficult. On the most challenging ones, **MIPplanet** slightly wins over **BaBSR**. A likely cause for the declined performance of **BaBSR** could be the split heuristic used. After a certain number of branches, the rough estimations used by the split heuristic are no longer fair representations of potential improvements that could be made by available branching decisions. This conjecture is consistent with what we observe on the Robust MNIST Network, where **BaBSR** significantly outperforms **MIPplanet** on all properties. Since each LP is expensive (requires more than 5 seconds) on the large network, the total number of branches that could be taken within the time limit is at least 10 times smaller than that on the reduced network, which means the heuristic of **BaBSR** probably remains effective throughout the verification procedure. **ERAN** is not as competent as the other two methods on the reduced Robust dataset but it beats **MIPplanet** on the Robust dataset. Compared to **MIPplanet**, **ERAN**, in its complete mode, adopts a similar idea of solving a MIP instance. However, different convex relaxations and intermediate bounds are used. Specifically, in these experiments, **ERAN** uses intermediate bounds collected by running RefinePoly analysis. Those computationally more expensive but potentially tighter intermediate bounds used by **ERAN** might explain its varying performance to that of **MIPplanet** on these datasets. In addition, when the network

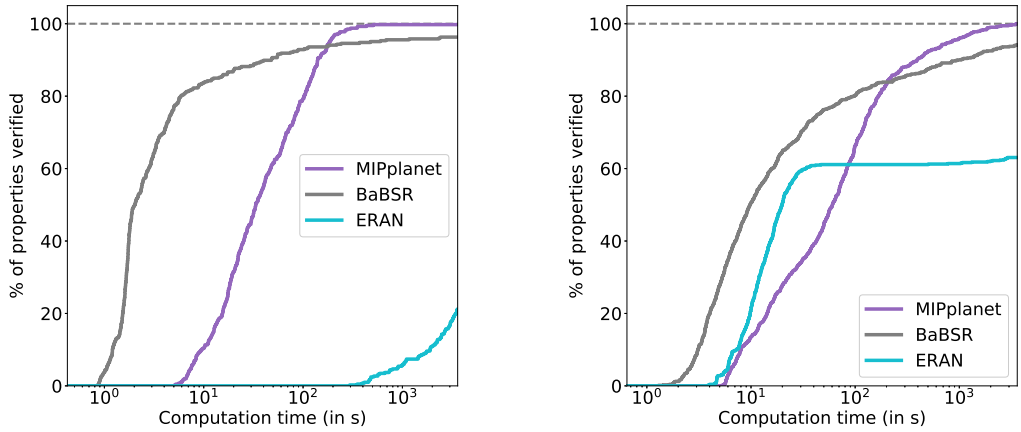
size increases, the time required by the LP solver increases exponentially, which becomes the main bottleneck in verifying the properties of large networks in the Branch-and-Bound framework. Thus, to tackle real-life verification problems, which often involve networks considerably larger than the Robust MNIST Network, it is important to develop an efficient LP solver that, by exploiting the special structure of neural networks, scales well with their size. At the same time, developing a better split strategy that is computationally cheap yet capable of giving high-quality decisions throughout the whole Branch-and-Bound process is also the key to the success of Branch-and-Bound methods in dealing with large network verification problems. In the next chapter, we demonstrate some success in achieving the desired branching strategies by employing graph neural networks to imitate strong branching decisions. Additional results based on the decision type of problems are reported in Figure 3.9 and Figure 3.10.



(a) All properties of reduced Robust MNIST Network.

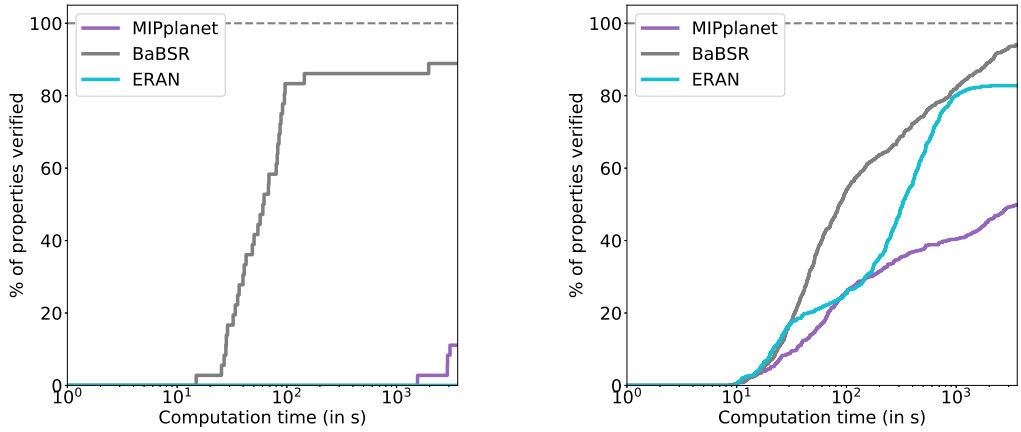
(b) All properties of the Robust MNIST Network.

Figure 3.8: Cactus plots of properties solved by **MIPplanet**, **BaBSR** and **ERAN** on the reduced Robust MNIST Network (left) and the Robust MNIST Network (right). **BaBSR** outperforms **MIPplanet** except on the difficult properties of the reduced Robust MNIST Network. The declined performance of **BaBSR** might be caused by the ineffectiveness of the branching heuristics at later stage. The better performance of **ERAN** than that of **MIPplanet** on the Robust MNIST Network could be explained by the possible tighter intermediate bounds used by **ERAN**.



(a) On SAT properties **(b)** On UNSAT properties

Figure 3.9: Proportion of properties verifiable by different methods under varying time budgets on the **reduced Robust network** data set. Similar performances can be observed on both SAT and UNSAT properties. In terms of the total number of properties solved, **MIPplanet** slightly outperforms **BaBSR** on challenging UNSAT problems. However, on simple problems, **BaBSR** are much more time efficient than **MIPplanet**.



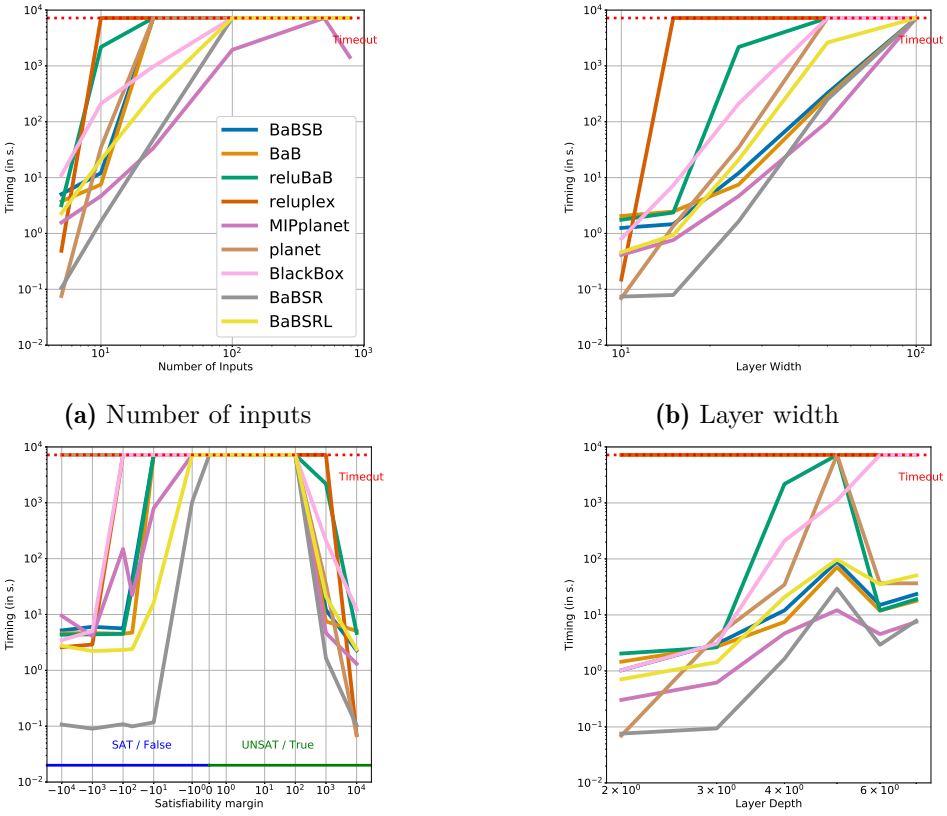
(a) On SAT properties **(b)** On UNSAT properties

Figure 3.10: Proportion of properties verifiable by different methods under varying time budgets on the **Robust Network** data set. **BaBSR** outperforms **MIPplanet** significantly in both cases. The huge performance gap on SAT properties indicates that Branch-and-Bound is an effective algorithm for finding counterexamples on large networks.

3.6.3 Varying Parameters

Finally, we study how the performance of each method is impacted by various parameters. Firstly, consider the case of various network architectures of the PCAMNIST dataset. In the graphs of Figure 3.11, the trend for almost all methods are similar, which seems to indicate that hard properties are intrinsically hard and not just hard for a specific solver. Figure 3.11a shows an expected trend: the larger the number of inputs, the harder the problem is. Similarly, Figure 3.11b shows unsurprisingly that wider networks require more time to solve, which can be explained by the fact that they have more non-linearities. The impact of the margin, as shown in Figure 3.11c is also clear. Properties that are true or false with large satisfiability margins are easy to prove, while properties that have small satisfiability margins are significantly harder. It is interesting to see the inconsistent performance of **MIPplanet**, which could be due to the different strategies used by Gurobi for different sized problems. In addition, consistent results of **BaBSR** outperforming **MIPplanet** on easy problems can be observed. We point out that the PCAMNIST dataset is a small dataset with only 27 networks. There is a possibility that the observations made may not generalise well.

The TwinStream dataset sheds some light on how to select branching and bounding strategies for the Branch and Bound based methods or other methods to best solve the properties at hand. In Figure 3.12, we see that **MIPplanet** performed the best over all properties and input split methods **BaBSB** and **BaB** performed the worst. Despite the fact that all twinladder networks are small, ReLU split strategy should be preferred when highly correlated layers are present. In terms of ReLU split based methods, the method with the tightest relaxation **reluBaB** and the method with the supposedly effective ReLU prioritizing heuristics **BaBSR** performed the worst. This is expected, as when the correlation among the hidden nodes of the same layer is high (by which we mean if we write each node as a linear combination of hidden nodes on the previous layer, the coefficient vectors are highly correlated), the ReLU score used in **BaBSR** will become way too loose to be of any use. For example, if several ReLU nodes $x_{i[p]}$ are highly correlated with the node $x_{i[j]}$, forcing



(a) Number of inputs

(b) Layer width

(c) Margin

(d) Network depth

Figure 3.11: Impact of the various parameters over the runtimes of the different solvers. The base network has 10 inputs and 4 layers of 25 hidden units, and the property to prove is true with a margin of 1000. Each of the plots corresponds to a variation of one of this parameters.

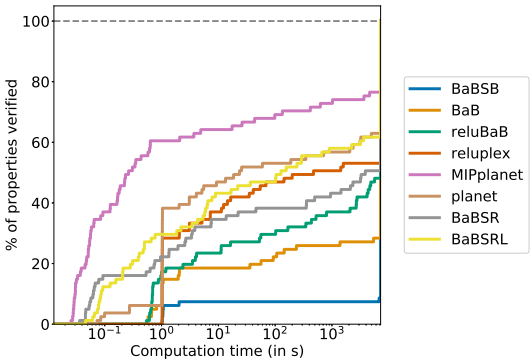


Figure 3.12: By design, the TwinStream dataset consists of UNSAT properties only. All methods returned correct results for properties verified apart from **Reluplex**, which return SAT for several properties. These properties are treated as unsolved for these methods. **MIPplanet** outperforms all other methods on the TwinStream dataset.

$\hat{x}_{i[j]} \leq 0$ or $\hat{x}_{i[j]} \geq 0$ will lead to notable changes to the upper or lower bound of $x_{i[p]}$ respectively. In extreme cases, it means $\hat{x}_{i[p]} \leq 0$ or $\hat{x}_{i[p]} \geq 0$ if the weights associate with $x_{i[j]}$ and $x_{i[k]}$ have a correlation of one. Estimating improvements by keeping all other terms the same is thus irrational in this case. The **reluBaB** method mainly suffers from its computational cost. The method **BaBSRL** lies between **BaBSR** and **reluBaB**. Since only one layer is updated via the LP, **BaBSRL** has tighter relaxations than **BaBSR** so the ReLU prioritising heuristic of **BaBSR** can make better branching decisions in the following steps. Yet, the computational cost of **BaBSRL** is much lower than that of **reluBaB**. An improved performance of **BaBSRL** over **BaBSR** and **reluBaB** can be observed. Overall, on the small networks with highly correlated layers like networks of Twinstream, **MIPplanet** is a definite winner. However, as we have observed previously, **MIPplanet** is likely to suffer from scalability issues. We expect **BaBSRL** might be a better option for large networks with highly correlated layers.

3.7 Discussion

Through the lens of a unified Branch-and-Bound framework, we have identified the weakness of existing methods and proposed new methods to improve on it. The effectiveness of the proposed methods are demonstrated through their enhanced performance on our comprehensive datasets. However, there is still much room for improvement. We illustrate a few ideas under the Branch and Bound framework.

In terms of bounding strategies, we observe tighter intermediate bounds could lead to faster convergence, but they are expensive to obtain. Given the layer-wise structure of neural networks, exploiting the GPU computing power might be a potential way to compute tighter intermediate bounds cheaply. Since the publication of the journal paper containing the main discussion of the chapter, several works have been produced exploring the similar idea. Bunel et al. [11] proposed a novel bound computation approach based on Lagrangian decomposition to take advantage of a network structure and GPU computing power. The same Planet relaxation is applied in Bunel et al. [11]. For a further improvement, De Palma et al. [22]

focuses on adopting an even tighter relaxation, introduced by Anderson et al. [4]. This stronger relaxation jointly constrains groups of ReLUs, rather than linearising them independently. The main reason the relaxation is rarely used, despite its improved tightness than Planet, is due to its exponentially increasing number of constraints involved. De Palma et al. [22] managed to alleviate this deficiency by introducing a customised solver that keeps an active set of constraints. By doing so, the solver is capable to utilize GPUs for parallel computation of bounds employing such a tighter relaxation.

For branching strategies, the methods discussed mainly rely on heuristics, which are likely to fail when problem changes. Cheaper while effective strategies should be possible. We introduce one learning-based solution based in the next chapter. Furthermore, Branch and Bound based methods require solving LPs, the main bottleneck hindering their development. Since LPs are mostly solved to decide whether a branching should be conducted on a (sub-)problem, learning to imitate LP decisions could be a potential way to speed up the whole process by orders of magnitude.

Furthermore, one advantage of the Branch-and-Bound framework is that it is not restricted to piecewise linear networks, which is not true for methods such as Reluplex, Planet or the MIP encoding. Any type of networks for which an appropriate bounding function can be found will be verifiable with a Branch-and-Bound based method. In order for Branch-and-Bound to achieve good performance on various kinds of networks, developing appropriate bounding functions is necessary. Recent advances on bounds for activations such as sigmoid or hyperbolic tangent have been made in Dvijotham et al. [27]. While their focus is on incomplete methods, our Branch-and-Bound framework makes it readily usable for complete verification as well.

We encourage the development of new methods to address these issues and hope that our inclusive and various datasets could facilitate the process by allowing comprehensive evaluation and comparison studies.

4

Neural Network Branching

Contents

4.1 Introduction

The Branch and Bound algorithm consists of two key components: branching and bounding. Branching strategies decide how the search space is recursively split into smaller spaces. Bounding methods compute bounds of each subspace to tighten the bounds of the final objective function over the whole search space. In the previous chapter, we have proposed several heuristics for each component to speed up the verification process. In this chapter, we make further improvements on the branching strategies. We propose a novel machine learning framework for designing a branching strategy. By directly working with a general framework, our identified algorithmic improvements can be combined with any bounding technique, leading to potential performance advancement for any Branch and Bound based verification method. We show that our learned branching strategy not only largely outperforms other heuristics on various datasets but is also computationally efficient.

4.2 Improved Branching Strategies with GNN

Branching strategies have significant impacts on the overall problem-solving process, as they directly decide the total number of steps, and consequently the total time, required to solve the problem at hand. The quality of a branching strategy is even more important when neural network verification problems are considered, which generally have a very large search space. Each input dimension or each activation unit can be a potential branching option and neural networks of interest often have high dimensional inputs and thousands of hidden activation units. With such a large search space, an effective branching strategy could mean a large reduction of the total number of branches required, and consequently of the time required to solve a problem. Developing an effective strategy is thus of significant importance to the success of Branch and Bound based neural network verification.

So far, to the best of our knowledge, branching rules adopted by Branch and Bound based verification methods are either random selection [28, 50] or hand-designed heuristics, including ones proposed in the previous chapter and those of [76, 95]. Random selection is generally inefficient as the distribution of the best branching decision is rarely uniform. In practice, this strategy often results in exhaustive search to make a verification decision. On the other hand, hand designed heuristics often involve a trade-off between effectiveness and computational cost. Empirically, strong branching is generally the best performing heuristics for Branch and Bound methods in terms of the number of branches, but it is computationally prohibitive as each branching decision requires an expensive exhaustive search over all possible options. To be more specific, strong branching adopts a greedy strategy by finding the locally optimal choice at each branching step. It is computationally expensive because it needs to compute the lower bound for available branching choices to identify the local optimal one. It should be noted that strong branching does not guarantee to find the global optimal solution (a solution that requires the minimum number of branches). Due to its computational cost, strong branching is rarely used even for medium-sized problems. Instead, heuristics generating branching choices with lower quality but high computational efficiency are used. The heuristics that are currently employed in practice are either inspired by the corresponding dual problem when verification is formulated as an optimization problem or incorporating the gradient information of the neural network [95].

Since for large size neural network verification problems, a slight reduction in the quality of the branching strategy could lead to substantial increase in the total number of branches required to solve the problem. A computationally cheap but high quality branching strategy is thus much needed. We propose a framework that employs GNNs to imitate the strong branching. GNNs allows us to make use of the inherent structure of the problem. Specifically, we make the following contributions:

- We use a GNN to exploit the structure of the neural network we want to verify. The embedding vectors of the GNN are updated by a novel schedule, which is both computationally cheap and memory efficient. In detail, we mimic the

forward and backward passes of the neural network to update the embedding vectors. In addition, the proposed GNN allows a customised schedule to update embedding vectors via shared parameters. That means, once training is done, the trained GNN model is applicable to various verification properties on different neural network structures.

- We train GNNs via supervised learning. We provide ways to generate training data cheaply but inclusive enough to represent branching problems at different stages of a Branch and Bound process for various verification properties. With the ability to exploit the neural network structure and a comprehensive training data set, our GNN is easy to train and converges fast.
- Our learned GNN also enjoys transferability both horizontally and vertically. Horizontally, although trained with easy properties, the learned GNN gives similar performance on medium and difficult level properties. More importantly, vertically, given that all other parts of Branch and Bound algorithms remain the same, the GNN trained on small networks performs well on large networks. Since the network size determines the total cost for generating training data and is positively correlated with the difficulty of learning, this vertical transferability allows our framework to be readily applicable to large scale problems.
- We further enhance our framework via online learning. For a learned branching strategy, it is expected that the strategy can fail to output satisfactory branching decisions from time to time. To deal with this issue, we provide an online scheme for fine-tuning the GNN along the Branch and Bound process in order to best accommodate the verification property at hand.
- Finally, we supply a dataset on convolutional neural network verification problems, covering problems at different difficulty levels over neural networks of different sizes. We hope that by providing a large problem dataset it could allow easy comparisons among existing methods and additionally encourage the development of better methods.

We mention that although we focus on neural networks with ReLU activation units in this chapter, we point out that our framework is applicable to any neural network architecture.

4.3 Related Work

Learning has already been used in solving combinatorial optimization problems [9, 21] and mixed integer linear programs (MILP) [3, 33, 40, 52]. In these areas, instances of the same underlying structure are solved multiple times with different data values, which opens the door for learning. Among them, Khalil et al. [52], Alvarez et al. [3], Hansknecht et al. [40], and Gasse et al. [33] proposed learned branching strategies for solving MILP with Branch and Bound algorithms. These methods imitate the strong branching strategy. Specifically, Khalil et al. [52] and Hansknecht et al. [40] learn a ranking function to rank potential branching decisions while Alvarez et al. [3] uses regression to assign a branching score to each potential branching choice. Apart from imitation, Anderson et al. [4] proposed utilizing Bayesian optimization to learn verification policies. There are two main issues with these methods. Firstly, they rely heavily on hand-designed features or priors and secondly, they use a generic learning structure which is unable to exploit the neural network architecture.

The approach most relevant to ours is the concurrent work by Gasse et al. [33]. They managed to reduce feature reliance by exploiting the bipartite structure of an MILP through a GNN. The bipartite graph is capable of capturing the network architecture, but cannot exploit it effectively. Specifically, it treats all the constraints the same and updates them simultaneously using the same set of parameters. This limited expressiveness can result in a difficulty in learning and hence in a high generalization error for neural network verification problems. Our proposed framework is specifically designed for neural network verification problems. By exploiting the neural network structure, and designing a customized schedule for embedding updates, our framework is able to scale elegantly both in terms of computation and memory. Finally, we mention that the recently

proposed verification methods [4, 51, 83] are not explicitly formulated in Branch and Bound. Since our focus is on branching, we mainly use the methods discussed in the previous chapter for comparison.

4.4 GNN Framework

To facilitate the discussion, recall that we treat a verification task as a global optimization problem. Assume $f : \mathbb{R}^{|\mathcal{X}|} \to \mathbb{R}$ is an L layer feed-forward neural network with ReLU activation units such that for any $\boldsymbol{x}_0 \in \mathcal{D} \subset \mathbb{R}^{|\mathcal{X}|}$, $f(\boldsymbol{x}_0) = \hat{\boldsymbol{x}}_L \in \mathbb{R}$ where

$$\hat{\boldsymbol{x}}_{i+1} = W^{i+1}\boldsymbol{x}_i + \boldsymbol{b}^{i+1}, \qquad \text{for } i = 0, \ldots, L-1, \tag{4.1a}$$

$$\boldsymbol{x}_i = \max(\hat{\boldsymbol{x}}_i, 0), \qquad \text{for } i = 1, \ldots, L-1. \tag{4.1b}$$

We want to find the minimum of $f(\boldsymbol{x}_0)$ over $\mathcal{D}$. We compare the minimum with the threshold 0 to check whether a property is verified (UNSAT) or not (SAT; counter-examples exist).

We present an overview of our GNN framework for solving neural network verification problems. We construct our framework based on a GNN, which takes the neural network f as a graph input G_f. An embedding vector $\boldsymbol{\mu}_v$ is initialised for each node in $v \in V$. Given the layer-by-layer formulation of verification problems, we design a GNN based on MPNN framework. As the result, the GNN updates each node's embedding vector by aggregating its own node features and all its neighbours' embedding vectors. After several rounds of updates, we obtain an updated embedding vector (a learned representation) of each node. To make a branching decision, we treat the updated embedding vectors as inputs to a score function, which assigns a score for each node that constitutes a potential branching option. A branching decision is made based on the scores of potential branching decision nodes. Our framework is visualised in Figure 4.1. In the following, we first introduce each component of the GNN. Then, we present a general updating schedule for neural network verification problems. A detailed implementation used in the experiments is provided in a later section. Finally, we illustrate the score function that is employed to make final branching decision.

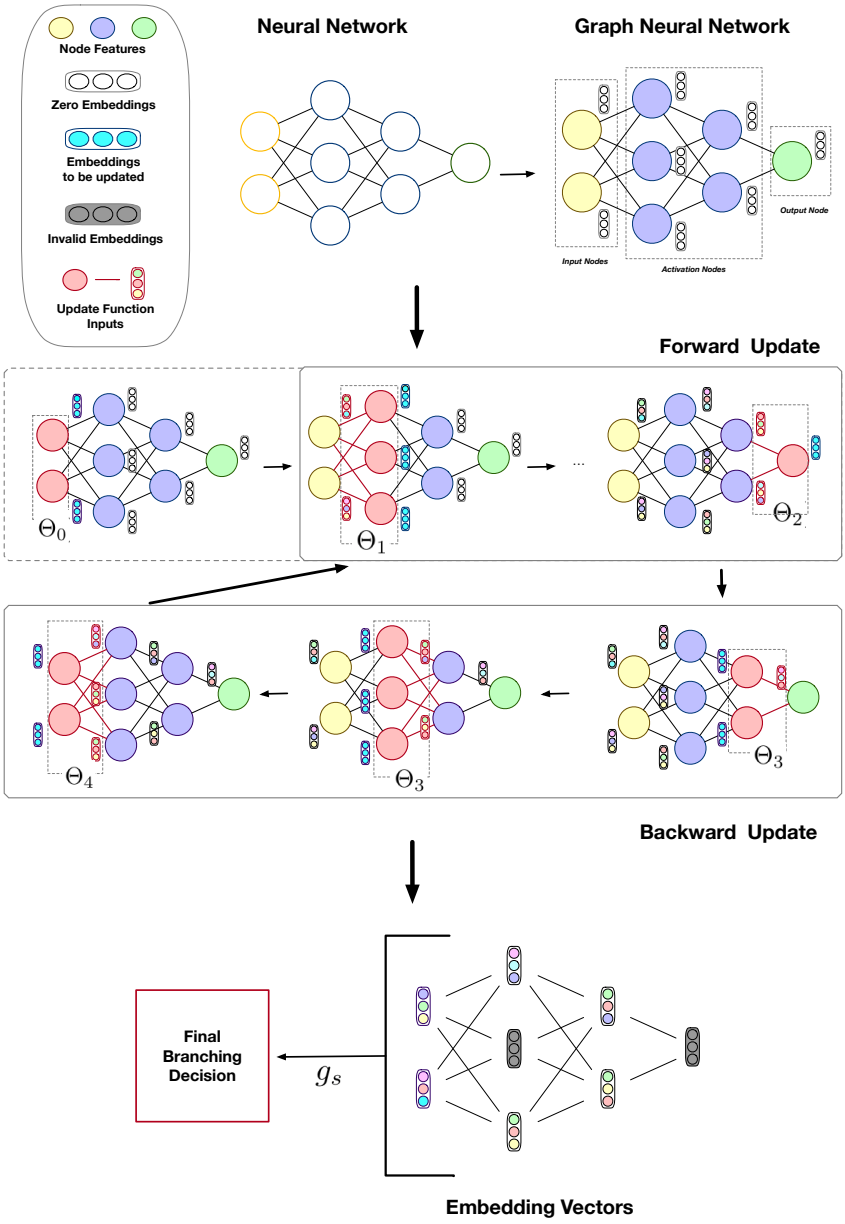


Figure 4.1: Illustration of our proposed GNN framework. An all zeros embedding network mimicking the neural network is initialised. Embedding vectors are updated via several rounds of forward backward passes using updating Equations 4.2-4.6. We obtain the final branching decision by calling a score function g_s over all embedding vectors of the potential branching decision nodes.

4.4.1 GNN Components

We now give descriptions for each component of the GNN.

Nodes Given the neural network f, V consists of all input nodes $v_{0[j]}$, all hidden activation nodes $v_{i[j]}$ and an output node v_L. In our framework, we combine every pre-activation variable and its associated post-activation variable and treat them as a single node. Pre- and post-activation nodes together contain the information about the amount of convex relaxation introduced at this particular activation unit, so dealing with the combined node simplifies the learning process. In terms of the Equation 4.1, let $x'_{i[j]}$ denote the combined node of $\hat{x}_{i[j]}$ and $x_{i[j]}$. The nodes $v_{0[j]}$, $v_{i[j]}$ and v_L are thus in one-to-one correspondence with $x_{0[j]}$, $x'_{i[j]}$ and x_L. We note that V is larger than the set of all potential branching decisions as it includes unambiguous activation nodes and an output node.

Node features Different types of nodes have different sets of features. In particular, input node features contain the corresponding domain lower and upper bounds and the primal solution. For activation nodes, the node features consist of associated intermediate lower and upper bounds, the layer bias, primal and dual solutions and new terms computed using previous features. Finally, the output node has features including the associated output lower and upper bounds, the layer bias and the primal solution. Other types of features could be used and some features could be excluded if they are expensive to compute. We denote input node features as $\boldsymbol{z}_{0[j]}$, activation node features as $\boldsymbol{z}_{i[j]}$ and output node features as $\boldsymbol{z}_L$. Our framework uses simple node features and does not rely on extensive feature engineering. Nonetheless, by relying on the powerful GNN framework, it provides highly accurate branching decisions.

Edges E consists of all edges connecting nodes in V, which are exactly the connecting edges in f. Edges are characterized by the weight matrices that define the parameters of the network f such that for an edge e^i_{jk} connecting $x'_{i[j]}$ and $x'_{i+1[k]}$, we assign $e^i_{jk} = W^i_{jk}$.

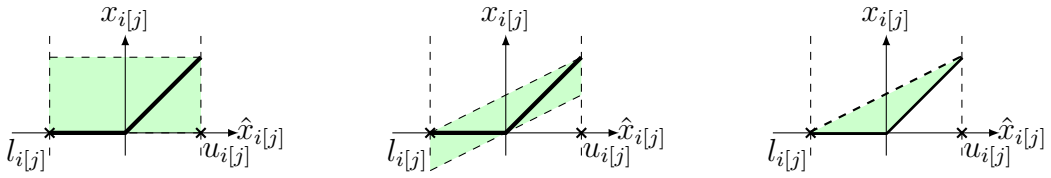
Embeddings We associate a p-dimensional embedding vector $\boldsymbol{\mu}_v$ for each node $v \in V$. All embedding vectors are initialised as zero vectors.

4.4.2 Forward and Backward Embedding Updates

In general, under the MPNN framework, all node embedding vectors are updated at the same time and there is no particular order between nodes. In this work, instead, we propose an update scheme where only the nodes corresponding to the same layer of the network f are updated at the same time, so embedding vector updates are carried out in a layer-by-layer forward-backward way.

We argue that the forward-backward updating scheme is a natural fit for our problem. In detail, for a given problem $\mathcal{D}$, each branching decision (an input node or an ambiguous activation node) will generate two sub-problems sub_{i_0} and sub_{i_1}, with each sub-domain having an output lower bound $l^L_{i_0}$ and $l^L_{i_1}$ respectively, equal to or higher than $l^L_{\mathcal{D}}$ the lower bound that of $\mathcal{D}$. Strong branching heuristic uses a predetermined function to measure the combined improvement of $l^L_{i_0}$ and $l^L_{i_1}$ over $l^L_{\mathcal{D}}$ and makes the final branching decision by selecting the node that gives the largest improvement. Thus, to maximise the performance of a graph neural network, we want a node embedding vector to maximally capture all information related to the computation of $l^L_{i_0}$ and $l^L_{i_1}$. For estimating $l^L_{i_0}$, $l^L_{i_1}$ of splitting on a potential branching decision node v, we note that these values are closely related to two factors. The first factor is the amount of convex relaxations introduced at a branching decision node v, when v corresponds to an ambiguous activation node. The second factor considers that the impact that splitting node v will have on the convex relaxations introduced to nodes on layers after that of v. Recall that, if there are no ambiguous activation nodes, the neural network f is simply a linear operator, whose minimum value can be easily obtained. When ambiguous activation nodes are present, the total amount of relaxation introduced determines the tightness of the lower bound to f. We thus treat embedding vectors as a measure of local convex relaxation and its contribution to other nodes' convex relaxation.

As shown in Figure 4.2, at each ambiguous activation node $x'_{i[j]}$, the area of convex relaxation introduced is determined by the lower and upper bounds of the pre-activate node $\hat{x}_{i[j]}$. We observe that intermediate lower and upper bounds of a node $\hat{x}_{i[j]}$ are significantly affected by the layers prior to it and have to be computed



(a) Naive relaxation **(b)** Linear bounds relaxation **(c)** Planet relaxation [28]

Figure 4.2: Different convex relaxations introduced. For each plot, the black line shows the output of a ReLU activation unit for any input value between $l_{i[j]}$ and $u_{i[j]}$ and the green shaded area shows the convex relaxation introduced. Naive relaxation (a) is the loosest relaxation. Linear bounds relaxation (b) is tighter and is introduced in [99]. Finally, Planet relaxation (c) is the tightest linear relaxation among the three considered [28]. Among them, (a) and (b) have closed form solutions which allow fast computations while (c) requires an iterative procedure to obtain an optimal solution.

in a layer-by-layer fashion. Based on the observation, we utilise a forward layer-by-layer update on node embedding vectors. This should allow these embedding vectors to capture the local relaxation information. In terms of the impact of local relaxation change to that of other nodes, we note that by splitting an ambiguous node into two fixed cases, all intermediate bounds of nodes on later layers will be affected, leading to relaxation changes at those nodes. We thus employ a backward layer-by-layer update to account for the impact the local change has over other nodes. Theoretically, by fixing an ambiguous ReLU node, intermediate bounds of nodes at previous layers and on the same layer might change as well. For a naturally trained neural network, the changes for these nodes should be relatively small compared to nodes on the later layers. To account for these changes, we rely on multiple rounds of forward-and-backward updates.

In summary, during the forward update, for $i = 1, \ldots, L-1$, we have, for all possible j,

$$\boldsymbol{\mu}_{0[j]} \longleftarrow F_{inp}(\boldsymbol{z}_{0[j]}; \boldsymbol{\theta}_0), \quad \text{if } \boldsymbol{\mu}_{0[j]} = \mathbf{0}, \tag{4.2}$$

$$\boldsymbol{\mu}_{i[j]} \longleftarrow F_{act}(\boldsymbol{z}_{i[j]}, \boldsymbol{\mu}_{i-1}, e^i; \boldsymbol{\theta}_1), \tag{4.3}$$

$$\boldsymbol{\mu}_L \longleftarrow F_{out}(\boldsymbol{z}_L, \boldsymbol{\mu}_{L-1}, e^L; \boldsymbol{\theta}_2). \tag{4.4}$$

During the backward update, for $i = L-1, \ldots, 1$, we have

$$\boldsymbol{\mu}_{i[j]} \longleftarrow B_{act}(\boldsymbol{z}_{i[j]}, \boldsymbol{\mu}_{i+1}, e^{i+1}; \boldsymbol{\theta}_3), \tag{4.5}$$

$$\boldsymbol{\mu}_{0[j]} \longleftarrow B_{inp}(\boldsymbol{z}_{0[j]}, \boldsymbol{\mu}_1, e^1; \boldsymbol{\theta}_4). \tag{4.6}$$

Update functions F and B take the form of multi-layered fully-connected networks with ReLU activation functions or composites of these simple update networks. The terms $\boldsymbol{\theta}_i$ denote the parameters of the networks. In the above, we have provided general updating formulas. Exact implementation details should be decided based on the problem at hand. For our neural network verification problems, we provide the exact formula design in Section 4.7.1.

We point out that our forward-backward update scheme does not depend on the underlying neural network structure and thus should be generalizable to network architectures that differ from the one we use for training. However, it does rely on the information used to compute convex relaxations, so underlying data distribution, features and bounding methods are assumed to be the same when the trained model is applied to different networks. Furthermore, our forward-backward update is memory efficient, as we are dealing with one layer at a time and only the updated embedding vectors of the layer are used to update the embedding vectors in the next (forward-pass) and the previous (backward-pass) layer. This makes it readily applicable to large networks.

4.4.3 Score Function

At the end of the forward-backward updates, embedding vectors for potential branching decision nodes (all input nodes and ambiguous activation nodes) are gathered and treated as inputs of a score function $g_s(\cdot; \boldsymbol{\theta}_5) : \mathbb{R}^p \to \mathbb{R}$, which takes the form of a fully-connected network with parameters $\boldsymbol{\theta}_5$. It assigns a scalar score for each input embedding vector. The final branching decision is determined by picking the node with the largest score.

4.5 Parameter Estimation

Given the GNN, we apply supervised learning to learn parameters

$$\boldsymbol{\Theta} \coloneqq (\boldsymbol{\theta}_0, \boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_3, \boldsymbol{\theta}_4, \boldsymbol{\theta}_5).$$

4.5.1 Training

We propose a new hinge rank loss function that is specifically designed for our framework. Before we give details of the loss, we introduce a relative improvement measure m first. Given a domain $\mathcal{D}$, for each branching decision node v, the two generated sub-problems have output lower bounds $l^L_{i_0}$ and $l^L_{i_1}$. We measure the relative improvement of splitting at the node v over the output lower bound $l^L_{\mathcal{D}}$ as follows

$$m_v \coloneqq (\min(l^L_{i_0}, 0) + \min(l^L_{i_1}, 0) - 2 \cdot l^L_{\mathcal{D}})/(-2 \cdot l^L_{\mathcal{D}}). \tag{4.7}$$

Intuitively, m $(0 \leq m \leq 1)$ measures the average relative sub-problem lower bound improvement to the maximum improvement possible, that is $-l^L_{\mathcal{D}}$. Any potential branching decision node v can be compared and ranked via its relative improvement value m_v. Since we are only interested in branching nodes with large improvement measures, ranking loss is a natural choice. A direct pairwise rank loss might be difficult to learn for neural network verification problems, given the large number of branching decision nodes on each domain $\mathcal{D}$. In addition, many branching decisions may give similar performance, so it is redundant and potentially harmful to the learning process if we learn a ranking among these similar nodes. To deal with these issues, we develop our loss by first dividing all potential branching nodes into M classes (M is much smaller than the total number of branching decision nodes) through the improvement value m_v of a node. We denote the class label as Y_v for a node v. Labels are assigned in an ascending order such that $Y_v >= Y_{v'}$ if $m_v > m_{v'}$. We then compute the pairwise hinge-rank loss on these newly assigned labels as

$$loss_{\mathcal{D}}(\boldsymbol{\Theta}) = \frac{1}{K} \sum_{i=1}^{N} \Big(\sum_{j=1}^{N} \phi(g_s(\boldsymbol{\mu}_j; \boldsymbol{\Theta}) - g_s(\boldsymbol{\mu}_i; \boldsymbol{\Theta})) \cdot \mathbf{1}_{\mathrm{Y_j > Y_i}} \Big), \tag{4.8}$$

where $\phi(z) = (1 - z)_+$ is the hinge function, N is the total number of branching decision nodes and K is the total number of pairs where $Y_j > Y_i$ for any branching

decision nodes v_i, v_j. The loss measures the average hinge loss on score difference $(g_s(\boldsymbol{\mu}_j; \boldsymbol{\Theta}) - g_s(\boldsymbol{\mu}_i; \boldsymbol{\Theta}))$ for all pairs of branching decision nodes v_i, v_j such that $Y_j > Y_i$. Finally, we evaluate $\boldsymbol{\Theta}$ by solving the following optimization problem:

$$\boldsymbol{\Theta} = \arg\min_{\boldsymbol{\Theta}} \frac{\lambda}{2}\|\boldsymbol{\Theta}\|^2 + \frac{1}{n}\sum_{i}^{n} loss_{\mathcal{D}_i}(\boldsymbol{\Theta}), \tag{4.9}$$

where the $loss_{\mathcal{D}_i}$ is the one introduced in Equation 4.8 and n is the number of training samples.

4.5.2 Fail-safe Strategy

We introduce a fail-safe strategy employed by our framework to ensure that consistent high-quality branching decisions are made throughout a Branch and Bound process. The proposed framework uses a GNN to imitate the behavior of the strong branching heuristic. Although computationally cheap, in some cases, the output decision by the learned graph neural network might be suboptimal. When this happens, it could lead to considerably deteriorated performance for two reasons. Firstly, we observed that for certain problems, which requires multiple splits to reach a conclusion on this problem, if a few low-quality branching decisions are made at the beginning or the middle stage of the branching process, the total number of splits required might increase substantially. The total Branch and Bound path is thus, to some extent, sensitive to the quality of each branching decision apart from those made near the end of the Branch and Bound process. Secondly, once a low-quality decision is made on a given problem, a decision of similar quality is likely to be made on the two newly generated sub-problems, leading to exponential decrease in performance. Features for newly generated sub-problems are normally similar to those of the parent problem, especially in the cases where the branching decision of the parent problem is made on the later layers and loose intermediate bounds are used. Thus, it is reasonable to expect the GNN fails again on the resulting sub-problems.

To deal with this issue, we keep track of the output lower bound improvement for each branching decision, as introduced in Equation 4.7. We then set a pre-determined threshold parameter. If the improvement is below the threshold, a

computationally cheap heuristic is called to make a branching decision. Generally, the back-up heuristic is able to give an above-threshold improvement and generate sub-problems sufficiently different from the parent problem to allow the learned GNN to recover from the next step onwards.

4.5.3 Online Learning

Online learning is a strategy to fine-tune the network for a particular property after we have learnt $\boldsymbol{\Theta}$. It can be seen as an extension of the fail-safe strategy employed. Every time a heuristic branching decision node v_h is used instead of the node v_{gnn} chosen by the GNN, we can use v_h and v_{gnn} to update the GNN accordingly. Since a correct GNN model should output an embedding vector $\boldsymbol{\mu}_h$ resulting in a higher score $g_s(\boldsymbol{\mu}_h; \boldsymbol{\Theta})$ for the heuristic decision, a loss can be developed based on the two scores $g_s(\boldsymbol{\mu}_h; \boldsymbol{\Theta})$ and $g_s(\boldsymbol{\mu}_{gnn}; \boldsymbol{\Theta})$ to generate optimization signals for correcting the GNN behaviour. For example, the loss used in our experimental setting is:

$$loss_{online}(\boldsymbol{\Theta}) = g_s(\boldsymbol{\mu}_{gnn}; \boldsymbol{\Theta}) - g_s(\boldsymbol{\mu}_h; \boldsymbol{\Theta}) \cdot (1 + (\gamma - 1) \cdot ((m_h - m_{gnn}) > t)\,. \quad (4.10)$$

The last term is used to amplify ($\gamma > 0$) the loss if the relative improvement made by the heuristic decision is more than t percent higher than that by the GNN. We update $\boldsymbol{\Theta}$ of the GNN by taking one gradient step with a small learning rate of the following minimization problem.

$$\boldsymbol{\Theta} = \arg\min_{\boldsymbol{\Theta}} \frac{\lambda}{2}\|\boldsymbol{\Theta}\|^2 + loss_{online}(\boldsymbol{\Theta}). \quad (4.11)$$

Online learning is property specific: it uses the decisions made by heuristics to fine tune the GNN model so it can best accommodate the property at hand. As will be shown in our experiments, a small but significant improvement in performance is achieved when online learning is used.

4.6 Experiments: General Setup

We now give details for our experiment setup. We are interested in verifying properties on large network architectures with convolutional layers. In the previous

chapter, existing neural network methods are compared on a robustly trained convolutional network on MNIST. For an increased difficulty level, in this chapter, we include and put emphasis on the more challenging dataset CIFAR-10.

4.6.1 Baselines

We decide our baselines based on the experiment results of Chapter 3. In the chapter, methods including MIPplanet, BaBSR, Planet [28], reluBaB, Neurify [94], ERAN [84–86] and Reluplex [50] are compared on a small convolutional MNIST network. Among them, BaBSR significantly outperform other methods while MIPplanet, supported by the sophisticated commercial solver Gurobi, gives consistent satisfactory performance over various datasets. We thus evaluate our methods against these two methods only in the experiments section[1].

4.6.2 BaB Algorithm Details

We give details for the branching strategy and bounding methods employed.

Bounding methods Intermediate bounds are determined by the better of bounds computed using linear bounds relaxations (Figure 4.2(b)) and interval arithmetic. For the output lower bound, we use Planet relaxation (Figure 4.2(c)) and solve the corresponding LP with Gurobi. For the output upper bound, we compute it by directly evaluating the network value at the input provided by the LP solution.

Branching strategy We focus on ReLU split only in our experiments. As shown in the previous chapter, domain split only outperforms ReLU split on low input dimensional and small scale networks. Also, since one of the Baseline method BaBSR employs a ReLU-split heuristic, we consider ReLU split only for a fair

[1]We note that, to strengthen our baseline, we also compared our MIPplanet baseline against a new MIP based algorithm [89]. To test these two methods, we randomly selected 100 verification properties from the CIFAR Base experiment with timeout 3600s. In terms of solving time, MIPplanet requires 1732.18 seconds on average while the new MIP algorithm requires 2736.60 seconds. Specifically, MIPplanet outperforms the new MIP algorithm on 78 out of 100 properties. MIPplanet is therefore a strong commerical solver backed baseline for comparison.

comparison. However, we emphasize that our framework is readily applicable to work with a combined domain and ReLU split strategy.

4.6.3 Network Structures

Three neural network structures will be studied for CIFAR-10 and MNIST respectively. The base one (Base) is of the similar structure and size to the one used in Chapter 3. It has two convolutional layers, followed by two fully connected layers and is trained robustly using the method provided in [100]. This particular choice of network size is made because the time required for solving each LP increases substantially with the size of the network. To best evaluate the performance of the branching strategy, we have to work with a medium sized network so that within the given timeout, a sufficient amount of branching decisions can be made to allow effective comparisons. When testing the transferability of the framework, two larger networks will be employed but their sizes are still restricted by the LP bottleneck. Specifically, for these two larger networks, one has the same layer structure as the Base model but has more hidden units on each layer, which we refer to as the Wide model. The other has a similar number of hidden units on each layer but more layers. We refer to it as the Deep model. A detailed description of network architecture for CIFAR-10 is provided in Table 4.1 and for MNIST in Table 4.2.

4.6.4 Verification Properties

Finally, we consider the following verification properties. Given an image $\boldsymbol{x}$ for which the model correctly predicted the label c, we randomly choose a label c' such that for a given ϵ, we want to prove $(\boldsymbol{y}^c - \boldsymbol{y}^{c'})^T f(\boldsymbol{x}') > 0$, $\forall \boldsymbol{x}'$ s.t $\|\boldsymbol{x} - \boldsymbol{x}'\|_\infty \leq \epsilon$. Here, f is the original neural network, $\boldsymbol{y}^c$ and $\boldsymbol{y}^{c'}$ are one-hot encoding vectors for labels c and c'. We want to verify that for a given ϵ, the trained network will not make a mistake by labelling the image as c'. Since BaBSR is claimed to be the best performing method on convolutional networks, we use it to determine the ϵ values, which govern the difficulty level of verification properties. Small ϵ values mean that most ReLU activation units are fixed so their associated verification properties are easy to prove while large ϵ values could lead to easy detection of

Network Name	No. of Properties	Network Architecture
BASE	Easy: 467 Medium: 773 Hard: 426	Conv2d(3,8,4, stride=2, padding=1) Conv2d(8,16,4, stride=2, padding=1) linear layer of 100 hidden units linear layer of 10 hidden units (Total ReLU activation units: 3172)
WIDE	300	Conv2d(3,16,4, stride=2, padding=1) Conv2d(16,32,4, stride=2, padding=1) linear layer of 100 hidden units linear layer of 10 hidden units (Total ReLU activation units: 6244)
DEEP	250	Conv2d(3,8,4, stride=2, padding=1) Conv2d(8,8,3, stride=1, padding=1) Conv2d(8,8,3, stride=1, padding=1) Conv2d(8,8,4, stride=2, padding=1) linear layer of 100 hidden units linear layer of 10 hidden units (Total ReLU activation units: 6756)

Table 4.1: For each CIFAR experiment, the network architecture used and the number of verification properties tested.

Network Name	No. of Properties	Network Architecture
BASE	100	Conv2d(1,4,4, stride=2, padding=1) Conv2d(4,8,4, stride=2, padding=1) linear layer of 50 hidden units linear layer of 10 hidden units (Total ReLU activation units: 1226)
WIDE	100	Conv2d(1,16,4, stride=2, padding=1) Conv2d(16,32,4, stride=2, padding=1) linear layer of 100 hidden units linear layer of 10 hidden units (Total ReLU activation units: 4804)
DEEP	100	Conv2d(1,8,4, stride=2, padding=1) Conv2d(8,8,3, stride=1, padding=1) Conv2d(8,8,3, stride=1, padding=1) Conv2d(8,8,4, stride=2, padding=1) linear layer of 100 hidden units linear layer of 10 hidden units (Total ReLU activation units: 5196)

Table 4.2: For each MNIST experiment, the network architecture used and the number of verification properties tested.

counter-examples. The most challenging ϵ values are those at which a large number of activation units are ambiguous. We use binary search with BaBSR method to

find suitable ϵ values. We only consider ϵ values that result in UNSAT properties and timed out properties. Binary search process is simplified by our choice of robustly trained models. Since these models are trained to be robust over a δ ball, the predetermined value δ can be used as a starting value for binary search.

Based on the difficulty level of properties and their underlying network architecture, we divide verification properties into 2 groups to test the horizontal and vertical transferability of our framework. Recall that horizontal transferability examines whether the GNN trained on easy properties can perform well on harder properties while vertical transferability tests whether the GNN trained on properties of a small network can work for properties of larger networks.

Horizontal transferability For horizontal transferability test, we use the same Base model structure but change the difficulty level for properties. In terms of CIFAR-10, testing verification properties are generated by binary search with BaBSR and 3600s timeout. We categorise verification properties solved within 800s as **easy**, which is consistent with training data generated, between 800s and 2400s as **medium** and more than 2400s as **hard**. In total, we generated 467 easy properties, 773 medium properties and 426 hard properties. We do not perform horizontal performance evaluation on MNIST due the its simplicity.

Vertical transferability For vertical transferability test, we varies the underlying network architecture. When CIFAR-10 is considered, we use BaBSR and a timeout of 7200s to generate 300 properties for the **Wide** model and 250 properties for the **Deep** model. For these two models, each LP called for solving a sub-problem output lower bound is much more time consuming, especially for the Deep model. This is reason that the average number of branches required is fewer that those of the Base model within the given time limit. For MNIST, we set the binary search time limit to be 1800 seconds for the Base model and 3600 seconds for the other two models. We generate 100 properties for each network architecture.

4.7 Experiments: Training a GNN

We give exact details how we implement forward and backward updates; generate training datasets; and set the hyper-parameters for the GNN used in the experiment section.

4.7.1 Implementation of Forward and Backward Passes

Based on previous analyses, choices of forward and backward update functions are based on the bounding methods used. In our experiments, we used linear bound relaxations for computing intermediate bounds and Planet relaxation for computing the final output lower bound. We start with a graph neural network mimicking the structure of the network we want to verify. Since an LP solver is called for the final output lower bound, we have primal values for all nodes of V and dual values for all ambiguous nodes of V.

4.7.1.1 Forward Pass

Unless otherwise stated, all functions F_* are 2-layer fully connected networks with ReLU activation units.

Input nodes We update the embedding vectors of input nodes during the first round of forward pass only. That is, we update $\boldsymbol{\mu}_{0[j]}$ when it is zero for all j. After that, input nodes embedding vectors are updated only in backward pass. For each input node, we form the feature vector $\boldsymbol{z}_{0[j]}$ as a vector of $l_{0[j]}$, $u_{0[j]}$ and its associated primal solution. The input node embedding vectors are computed as

$$\boldsymbol{\mu}_{0[j]} = F_{inp}(\boldsymbol{z}_{0[j]}; \boldsymbol{\theta}_0). \tag{4.12}$$

Activation nodes The update function F_{act} can be broken down into three parts: (1) compute information from local features; (2) compute information from neighbourhood embedding vectors; and (3) combine information from (1) and (2) to update current layer's embedding vectors.

1. *compute information from local features* Since we compute the final lower bound with the Planet relaxation (Figure 4.2(c)), we introduce a new feature related to the relaxation: the intercept of the relaxation triangle, shown in Figure 4.3. We denote an intercept as β and compute it as

$$\beta_{i[j]} = \frac{-l_{i[j]} \cdot u_{i[j]}}{u_{i[j]} - l_{i[j]}}. \tag{4.13}$$

The intercept of a relaxation triangle can be used as a measure of the amount of relaxation introduced at the current ambiguous node.

Therefore, the local feature vector $\boldsymbol{z}_{i[j]}$ of an ambiguous node $x'_{i[j]}$ consists of $l_{i[j]}$, $u_{i[j]}$, $\beta_{i[j]}$, its associated layer bias value, primal values (one for pre-activation variable and one for post-activation variable) and dual values. We obtain information from local features via

$$Q_{i[j]} = \begin{cases} F_{act-lf}(\boldsymbol{z}_{i[j]}; \boldsymbol{\theta}_1^0) & \text{if } x'_{i[j]} \text{ is ambiguous,} \\ \mathbf{0} & \text{otherwise.} \end{cases} \tag{4.14}$$

where $Q_{i[j]} \in \mathbb{R}^p$.

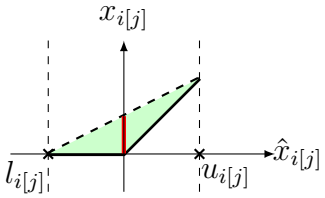
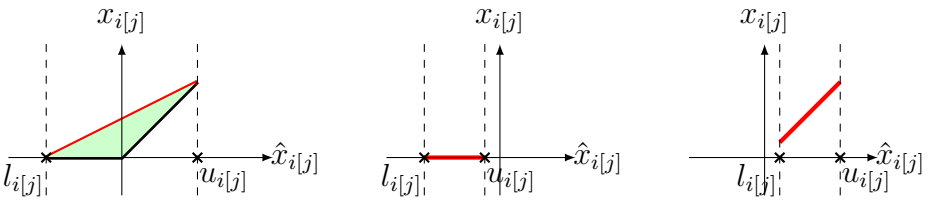


Figure 4.3: Red line represents the intercept of the convex relaxation. It is treated as a measure of the shaded green area.

2. *Information from neighbourhood embedding vectors* During the forward pass, we focus on embedding vectors of the previous layer only. To update an embedding vector on layer i, we first combine embedding vectors of the previous layer with edge weights via

$$E_{i[j]} = \sum_k W^i_{kj} \cdot \boldsymbol{\mu}_{i-1[k]}. \tag{4.15}$$

To compute the information from neighbourhood embedding vectors to an arbitrary activation node $x'_{i[j]}$, we consider each activation unit as a *gate*. We observe that the amount of the information from neighbourhood embedding



(a) Ambiguous Node (b) Complete Blocking (c) Complete Passing

Figure 4.4: Depending on the value of $l_{i[j]}$ and $u_{i[j]}$, relaxed activation function can take three forms. The left figure shows the case where $l_{i[j]}$ and $u_{i[j]}$ are of different signs. In this case, for any input value between $l_{i[j]}$ and $u_{i[j]}$, the maximum output achievable is indicated by the red line. The middle figure shows the case where both $l_{i[j]}$ and $u_{i[j]}$ are no greater than zero. In this case, the activation function completely blocks all input information by outputting zero for any input value. The right figure shows the case where $l_{i[j]}$ and $u_{i[j]}$ are greater or equal to zero. In this case, the activation function allows complete information passing by outputting a value equal to the input value.

vectors that remains after passing through a *gate* is dependent on the its lower bound $l_{i[j]}$ and upper bound $u_{i[j]}$. When $l_{i[j]}$ and $u_{i[j]}$ are of different signs, $x'_{i[j]}$ is an ambiguous node. With relaxation, for any input value between $l_{i[j]}$ and $u_{i[j]}$, the maximum output achievable after passing an activation unit is shown by the red slope in Figure 4.4(a). The red slope $s_{i[j]}$ is computed as

$$s_{i[j]}(\hat{x}_{i[j]}) = \frac{u_{i[j]}}{u_{i[j]} - l_{i[j]}} \cdot \hat{x}_{i[j]} + \beta_{i[j]}. \tag{4.16}$$

Thus, the amount of information from neighbourhood embedding vectors that remains after passing through an ambiguous *gate* is related to the ratio $\alpha \coloneqq \frac{u_{i[j]}}{u_{i[j]} - l_{i[j]}}$. When $u_{i[j]}$ is no greater than zero, the activation node $x'_{i[j]}$ completely blocks all information. For any input value, the output value is zero after passing the activation unit, as shown by the red line in Figure 4.4(b). We have $\alpha = 0$ in this case. Finally, when $l_{i[j]}$ is no less than 0, the activation node $x'_{i[j]}$ allows a complete passing of information and $\alpha = 1$. It is shown by the red line in Figure 4.4(c). We incorporate these observations into our evaluations and compute the information from neighbourhood embedding vectors as

$$N_{i[j]} = f_{act-nb}([\alpha \cdot E_{i[j]}, \alpha' \cdot E_{i[j]}]; \boldsymbol{\theta}_1^1), \tag{4.17}$$

where $\alpha' = 1 - \alpha$ when $0 < \alpha < 1$ and $\alpha' = \alpha$ otherwise. Here, we use $[\boldsymbol{a}, \boldsymbol{b}]$ to denote the concatenation of two vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^p$ into a vector of $\mathbb{R}^{2p}$. We introduce α' to be more informative. We do not consider the information that relate to the intercept $\beta_{i[j]}$ in the ambiguous case for the sake of simplicity. Improved performance could be expected if the $\beta_{i[j]}$ related information is incorporated as well.

3. *Combine previous information* Finally, we combine the information from local features and the information from neighbourhood embedding vectors to update the embedding vectors of activation nodes. Specifically,

$$\boldsymbol{\mu}_{i[j]} = F_{act-com}([Q_{i[j]}, N_{i[j]}]; \boldsymbol{\theta}_1^2). \tag{4.18}$$

Output node The embedding vector of the output node is updated in a similar fashion to that of activation nodes. We first compute information from local features.

$$Q_{Lj} = F_{out-lf}(\boldsymbol{z}_{Lj}; \boldsymbol{\theta}_2^0) \tag{4.19}$$

For the output node, the vector of local features $\boldsymbol{z}_L$ consists of the output lower bound, the output upper bound, the primal solution and the layer bias. F_{out-lf} is a one-layer fully connected network with ReLU activation units. We then compute information from neighbourhood embedding vectors. Since the output node does not have an activation unit associated with it, we directly compute the information of neighbourhood embedding vectors as

$$E_{L[j]} = \sum_k W_{kj}^L \cdot \boldsymbol{\mu}_{L-1[k]}. \tag{4.20}$$

Finally, we update the embedding vector of the output node as

$$\boldsymbol{\mu}_{Lj} = F_{out-com}([Q_{L[j]}, E_{L[j]}]; \boldsymbol{\theta}_2^1). \tag{4.21}$$

4.7.1.2 Backward Pass

During backward message passing, for $i = L - 1, \ldots, 1$, we update the embedding vectors for activation nodes and input nodes. Again, all functions B_* are 2-layer fully connected networks unless specified otherwise.

Activation nodes Similar to the updates of embedding vectors carried out for activation nodes in a forward pass, we update the embedding vectors of activation nodes using the same three steps in the backward pass, but with minor modifications.

1. *Information from local features* We use the same feature $\boldsymbol{z}_{i[j]}$ as the one used in the forward pass and compute the information from local features as

$$Q^b_{i[j]} = \begin{cases} B_{act-lf_1}(\boldsymbol{z}_{i[j]}; \boldsymbol{\theta}^0_3) & \text{if } x'_{i[j]} \text{ is ambiguous,} \\ \mathbf{0} & \text{otherwise.} \end{cases} \tag{4.22}$$

We recall that a dual value indicates how the final objective function is affected if its associated constraint is relaxed by a unit. To better measure the importance of each relaxation to the final objective function, we further update the information from local features by

$$Q^{b'}_{i[j]} = \begin{cases} B_{act-lf_2}([\boldsymbol{d}_{i[j]} \odot Q^b_{i[j]}, Q^b_{i[j]}]; \boldsymbol{\theta}^1_3) & \text{if } Q^b_{i[j]} \neq \mathbf{0} \\ \mathbf{0} & \text{otherwise.} \end{cases} \tag{4.23}$$

Here, $\boldsymbol{d}_{i[j]}$ is the vector of dual values corresponding to the activation node $x'_{i[j]}$. We use $\odot$ to mean that we multiply $Q^b_{i[j]}$ by each element value of $\boldsymbol{d}_{i[j]}$ and concatenate them as a singe vector.

2. *Information from neighbourhood embedding vectors* During the backward pass, we focus on embedding vectors of the next layer only. In order to update an embedding vector on layer i, we compute the neighbourhood embedding vectors as

$$E^b_{i[j]} = \sum_k W^{i+1}_{jk} \cdot \boldsymbol{\mu}_{i+1[k]}. \tag{4.24}$$

We point out that there might be an issue with computing $E^b_{i[j]}$ if the layer $i+1$ is a convolutional layer in the backward pass. For a convolutional layer, depending on the padding number, stride number and dilation number, each node $x'_{i[j]}$ may connect to a different number of nodes on the layer $i+1$. Thus, to obtain a consistent measure of $E^b_{i[j]}$, we divide $E^b_{i[j]}$ by the number of

connecting node on the layer $i+1$. We denote the resulted average value as $E^{b'}_{i[j]}$ and use it instead. Let

$$E^{b*}_{i[j]} = \begin{cases} E^{b'}_{i[j]} & \text{if layer i+1 convolutional,} \\ E^{b}_{i[j]} & \text{otherwise.} \end{cases} \tag{4.25}$$

The following steps are the same as the forward pass. We first evaluate

$$N^{b}_{i[j]} = B_{act-nb}([\alpha \cdot E^{b*}_{i[j]}, \alpha' \cdot E^{b*}_{i[j]}]; \boldsymbol{\theta}^2_3), \tag{4.26}$$

and the update embedding vectors as

$$\boldsymbol{\mu}_{i[j]} = B_{act-com}([Q^{b'}_{i[j]}, N^{b}_{i[j]}]; \boldsymbol{\theta}^3_3). \tag{4.27}$$

Input nodes Finally, we update the input nodes. We use the feature vector $\boldsymbol{z}^b_0$, which consists of domain upper bound and domain lower bound. Information from local features is evaluated as

$$Q_{0j} = B_{inp-lf}(\boldsymbol{z}^b_{0[j]}; \boldsymbol{\theta}^0_4). \tag{4.28}$$

We compute the information from neighbourhood embedding vectors in the same manner as we do for activation nodes in the backward pass, shown in Equation 4.25. Denote the computed information as $E^{b*}_{0[j]}$. The embedding vectors of input nodes are updated by

$$\boldsymbol{\mu}_{0[j]} = B_{inp-com}([Q^{b'}_{0[j]}, E^{b*}_{0[j]}]; \boldsymbol{\theta}^1_4). \tag{4.29}$$

4.7.2 Generate Training Datasets

We collect training data along the Branch and Bound process for solving a verification property. For each given domain, given the large number of potential branching decisions, we perform the strong branching heuristic on a selected subset of all potential branching decisions. The subset consists of branching decisions that are estimated to be of high quality by the BaBSR heuristic and randomly selected ones, which ensure a minimum 5% coverage on each layer.

To construct a training dataset that is representative enough of the whole problem space, we need to cover a large number of properties. In addition, it is

important to include training data at different stages of a Branch and Bound process. However, running a complete Branch and Bound process with the strong branching heuristic for hundreds of properties is computationally expensive and considerably time consuming. We thus propose the following procedure for generating a training dataset to guarantee a wide coverage both in terms of the verification properties and Branch and Bound stages. From all verification properties, we randomly select around 25% of non-timeout property to conduct a complete Branch and Bound process with the strong branching heuristic. For the rest of the properties, we try to generate at least $B = 20$ training data for each verification property. Given the maximum number of branches $q = 10$ and an effective and computationally cheap heuristic, we first produce a random integer k from $[0, q]$. Then, we run a Branch and Bound process with the selected cheap heuristic for k steps. Finally, we call the strong branching heuristic to generate a training sample. We repeat the process until B training samples are collected or the Branch and Bound process is terminated. Algorithm 2 outlines the procedure for generating the training dataset.

CIFAR-10 To generate a training dataset, 565 random images are selected. Binary search with BaBSR and 800 seconds timeout are used to determine ϵ on the Base model. Among the 565 verification properties determined, we use 430 properties to generate 17958 training samples and the rest of the properties to generate 5923 validation samples.

For a typical epsilon value, each sub-problem generally contains 1300 ambiguous ReLU nodes. Among them, approximately 140 ReLU nodes are chosen for strong branching heuristics, which leads to roughly 200 seconds for generating a training sample. We point out that the total amount of time required for generating a training sample equals 2*(per LP solve time)*(number of ambiguous ReLU nodes chosen). Although both the second and the third terms increase with the size of the model used for generating the training dataset, the vertical transferability of our GNN, as demonstrated later, enables us to efficiently generate a training dataset by working with a small substitute of the model we are interested in.

Algorithm 2 Generating Training Dataset

Provided: total P properties; minimum B training data for each property; a maximum q branches between strong branching decisions
for $p = 1, \ldots, P$ **do**:
    $\alpha \longleftarrow$ random number from $[0, 1]$
    if p is not a timed out property and $\alpha \leq 0.25$ **then**
        Running a complete Branch and Bound process with the Strong Branching Heuristic
    else
        $b = 0$
        while $b \leq B$ **do**
            $k \longleftarrow$ random integer from$[0, q]$
            while $k > 0$ **do**
                Call a computationally cheap heuristic
                if BaB algorithm terminates **then return**
                end if
                $k = k - 1$
            end while
            Call the strong branching heuristic and generate a training sample
            if BaB algorithm terminates **then return**
            end if
            $b = b + 1$
        end while
    end if
end for

MNIST To generate a training dataset, 538 random MNIST images are selected. Binary search with BaBSR and 600 seconds timeout are used to determine ϵ on the Base model. Among the 538 verification properties determined, we use 403 properties to generate 18231 training samples and the rest of the properties to generate 5921 validation samples. For a typical epsilon value, each sub-problem generally contains 480 ambiguous ReLU nodes. Among them, approximately 80 ReLU nodes are chosen for strong branching heuristics, which leads to roughly 45 seconds for generating a training sample.

4.7.3 Training Details

CIFAR-10 We initialise a GNN by assigning each node a 64-dimensional zero embedding vector. GNN updates embedding vectors through two rounds of forward and backward updates. To train the GNN, we use hinge rank loss (Equation 4.8) with

$M = 10$. Parameters $\mathbf{\Theta}$ are computed and updated through the Adam optimizer with the weight decay rate $\lambda = 1e^{-4}$ and the learning rate $1e^{-4}$. If the validation loss does not decrease for 10 consecutive epochs, we decrease the learning rate by a factor of 5. If the validation loss does not decrease for 20 consecutive epochs, we terminate the learning procedure. The batch size is set to 2. In our experiments, each training epoch takes less than 400 seconds and the GNN converges within 60 epochs.

In terms of the training accuracy, we first evaluate each branching decision using the metric defined by Equation 4.7 [2]. Since there are several branching choices that give similar performance for each sub-problem, we consider all branching choices that have m_v above 0.9 as correct decisions. Under this assumption, our trained GNN achieves 85.8% accuracy on the training dataset and 83.1% accuracy on the validation dataset.

MNIST The same set of hyper-parameters and training procedure are used for training a GNN for MNIST dataset. The GNN converges in 70 epochs with each epoch taking less than 400 seconds. The trained GNN reached 86.5% accuracy on the training dataset and 83.1% accuracy on the validation dataset.

4.7.4 Implementation of the Fail-safe Strategy

Since, to the best of our knowledge, the branching heurisitc of BaBSR is the best performing one on convolutional neural networks so far, we choose it for our fail-safe strategy. The threshold is set to be 0.2. Every time when the relative improvement m_{gnn} of a GNN branching decision v_{gnn} is less than 0.2, we call the heuristic to make a new branching decision v_h. We solve the corresponding LPs for the new branching decision and compute its relative improvement m_h. The node with higher relative improvement is chosen to be the final branching decision.

[2] we have tried various other metrics, including picking the minimum of the two sub-problem lower bounds and the maximum of the two lower bounds. Among these metrics, metric defined by Equation 4.7 performs the best.

4.7.5 Implementation of Online Learning

We take a conservative approach in terms of online learning. We refer to a GNN decision as a failed decision if the relative improvement offered by heuristic branching is better than the one offered by the GNN. We record all GNN failed decisions and only update the GNN model online when the same failed decision is made at least twice. To update the GNN model, we use Adam optimizer with the weight decay rate $\lambda = 1e^{-4}$ and the learning rate $1e^{-4}$. The GNN model is updated with one gradient step only with respect to the optimization problem Equation 4.11, where $\gamma = 1$ and $t = 0.1$ in the loss function $loss_{online}$, defined in Equation 4.10.

4.8 Experiments: Results and Analyses

All verification experiments are run in parallel on 16 CPU cores, with one property being verified on one CPU core. We focus on studying the results for the more challenging dataset CIFAR-10 and then briefly discuss the results for MNIST.

4.8.1 CIFAR-10

We first conduct analyses on the dataset CIFAR-10.

4.8.1.1 Horizontal Transferability

We show the horizontal transferability results in the Table 4.3. Methods are compared in three perspectives: the average time over all properties, the average number of branches required over the properties that are solved by all methods (we exclude timed-out properties), and the ratio of timed-out properties. Since the properties are generated based on BaBSR, the timed-out ratios of BaBSR on easy and medium properties are not comparable with those of other methods. All other numbers should give a fair evaluation of the effectiveness of our branching strategy. BaBSR, GNN, and GNN-online only differ in the branching strategy used.

On all three sets of properties, we see that our learned branching strategy has led to a more than 50% reduction in the total average number of branches required for a property. As a direct result, the average time required achieves at least a

	Easy		
Method	time(s)	branches	%Timeout
BaBSR	545.457	578.828	0.0
MIPplanet	1499.375		0.165
GNN	272.695	285.682	0.002
GNN-online	**208.821**	**252.807**	**0.002**
	Medium		
Method	time(s)	branches	%Timeout
BaBSR	1370.395	1405.301	0.0
MIPplanet	2240.980		0.430
GNN	592.118	583.210	0.004
GNN-online	**556.163**	**501.595**	**0.001**
	Hard		
Method	time(s)	branches	%Timeout
BaBSR	3127.995	2493.870	0.413
MIPplanet	2250.623		0.462
GNN	1577.156	995.437	0.216
GNN-online	**1369.326**	**813.502**	**0.183**

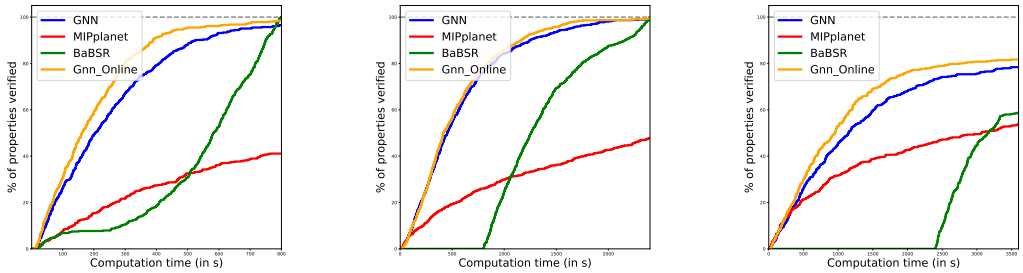
Table 4.3: Methods' performance on the Base model trained on CIFAR-10. For easy, medium and difficult level verification properties, we compare methods' average solving time, average number of branches required and the percentage of timed out properties. GNN-Online outperforms other methods in all aspects.

50% reduction as well. Our framework is thus an effective scheme and enjoys horizontal transferability. A further performance improvement is obtained through instance-specific online learning. Among all 1666 tested verification properties, GNN with online learning solves 61.52% of properties with fewer number of branches and 60.20% of properties in less time when compared to the standard GNN.

Time cactus plots for each category of properties can be found in Figure 4.5. All these time cactus plots look similar. We also provide a time cactus plot for all properties of the Base model in Figure 4.6a. Although BaBSR performs better than the commercial solver encoded method MIPplanet overall, MIPplanet wins on a subset of properties. The learned model GNN, however, is capable of giving consistent high quality performance over all properties tested.

4.8.1.2 Vertical Transferability

The model learned on the Base network is tested on the verification properties of large networks. Experimental results are given in the Table 4.4 and time cactus plots



(a) Easy properties **(b)** Medium properties **(c)** Hard properties

Figure 4.5: Cactus plots for easy properties (left), medium properties (middle) and hard properties (right) on the Base model trained on CIFAR-10. For each category, we plot the percentage of properties solved in terms of time for each method. BaBSR beats MIPplanet on easy and medium properties overall. On hard properties, although BaBSR manages to solve more properties, its average performance is worse than MIPplanet in terms of time. GNN consistently outperforms BaBSR and MIPplanet on all three levels of properties, demonstrating the horizontal transferability of our framework. Again, further small improvement can be achieved through online-learning.

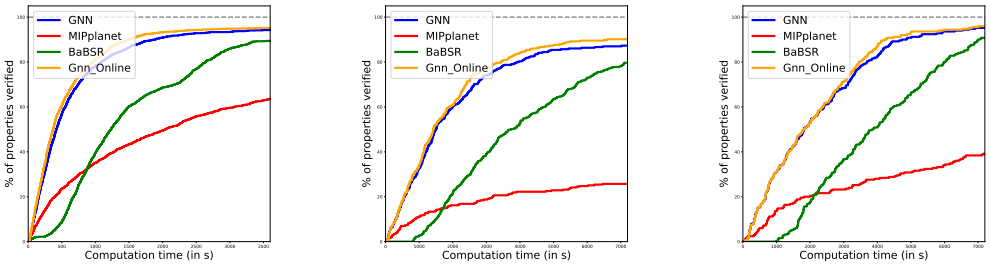
	Wide			Deep		
Method	time	branches	%Timeout	time	branches	%Timeout
BaBSR	4137.467	843.476	0.0	4016.336	416.824	0.0
MIPplanet	5855.059		0.743	5426.160		0.608
GNN	2367.693	387.403	0.127	2308.612	208.760	0.048
GNN-online	**2179.306**	**353.879**	**0.095**	**2220.351**	**199.032**	**0.040**

Table 4.4: Methods' performance on large models trained on CIFAR-10. For verification properties on the Wide model and the Deep model, we compare methods' average solving time, average number of branches required and the percentage of timed out properties. GNN-Online outperforms other methods in all aspects.

(Figures 4.6b, 4.6c) are also provided. All results are similar to what we observed on the Base model, which show that our framework enjoys vertical transferability.

4.8.1.3 Fail-safe Heuristic Dependence

We note that removing the GNN and using the fail-safe heuristic only is equivalent to BaBSR. The fact that GNN significantly outperforms BaBSR demonstrates that GNN is doing most of the job. To better evaluate the GNN's reliance on a fail-safe heuristic, we study the ratio of times that a GNN branching decision is used for each verification property of a given model. Results are listed in Table 4.5. For all three models, GNN accounts for more than 90% of branching decisions employed on average, ensuring the effectiveness of our GNN framework.



(a) Base model **(b)** Wide large model **(c)** Deep large model

Figure 4.6: Cactus plots for the Base model (left), Wide large model (middle) and Deep large model (right) trained on CIFAR-10. For each model, we plot the percentage of properties solved in terms of time for each method. Consistent performances are observed on all three models. BaBSR beats MIPplanet on the majority of properties. GNN consistently outperforms BaBSR and MIPplanet. Further small improvement can be achieved through online-learning.

Model	GNN(avg)	BaBSR(avg)	BaBSR(min)	BaBSR(max)
Base	0.934	0.066	0.0	0.653
Wide	0.950	0.050	0.0	0.274
Deep	0.964	0.036	0.0	0.290

Table 4.5: Evaluating GNN's dependence on the fail-safe strategy. Given a model trained on CIFAR-10, we collected the percentage of times GNN branching decision is used and the percentage of times the fail-safe heuristic (BaBSR in our case) is employed for each verification property. We report the average ratio of all verification properties of the same model. To account for extreme cases, we also list the minimum and maximum usage ratios of the fail-safe heuristic for each model.

4.8.1.4 GNN Feature Analysis

We evaluate the importance of different features used in GNN. We note that two types of features are used in GNN. The first type (including intermediate bounds, network weights, and biases) can be collected at negligible costs. The other type is LP features (primal and dual values) that are acquired by solving a strong LP relaxation, which are expensive to compute but potentially highly informative. To evaluate their effects, we trained a new GNN with LP features removed and tested the new GNN on 260 randomly selected verification properties on the Base model. Among the selected properties, 140 are categorised as easy, 70 as medium, and 50 as hard. We denote the model trained on all features as GNN and the newly trained model as GNN-R (we use R to indicate reduced features).

	Easy		
Method	time(s)	branches	%Timeout
BaBSR	429.589	641.300	0.0
GNN	268.592	319.386	0.0
GNN-R	348.482	441.043	0.0
	Medium		
Method	time(s)	branches	%Timeout
BaBSR	1622.669	1504.366	0.0
GNN	724.883	529.070	0.0
GNN-R	898.011	720.958	0.0
	Hard		
Method	time(s)	branches	%Timeout
BaBSR	2466.712	1931.098	0.0
GNN	1025.826	772.667	0.0
GNN-R	1340.559	967.804	0.0

Table 4.6: Measuring the importance of features used by GNN. For easy, medium and difficult level verification properties, we compare methods' average solving time, average number of branches required and the percentage of timed out properties.

From Table 4.6, we observe that removing primal and dual information deteriorates the GNN performance, but GNN-R still outperforms the baseline heuristic BaBSR. We believe cheap features are the most important. Depending on the cost of LP, potential users can either remove expensive LP features or train a GNN with a smaller architecture.

4.8.1.5 MIPplanet Branching Number

MIPplanet is implemented with the commercial solver Gurobi. Since Gurobi outputs an internal branch number, we recorded MIPplanet branch number for a subset of verification properties for each model. In detail, we randomly selected 120 properties of various difficulty levels for the Base model and 27 properties each for the Wide and Deep model. Results are summarised in Table 4.7.

One key observation we made is that Gurobi branch number is not positively related to the solving time. For instance, on timed-out properties of the Wide model, MIPplanet branch number varies between 1 and 7479. We suspect Gurobi performs cutting before branching, so the time spent on branching varies between properties, leading to inconsistent branch number and solving time. As a result, the MIPplanet branch number is not comparable with that of BaBSR, GNN, and

	Base		
Method	time(s)	branches	%Timeout
BABSR	1472.508	1420.839	0.067
MIPPLANET	1783.800	3780.408	0.258
GNN	714.224	817.017	0.017
GNN-ONLINE	582.738	642.387	0.008
	Wide		
Method	time(s)	branches	%Timeout
BABSR	2985.199	918.167	0.111
MIPPLANET	5254.134	2949.625	0.556
GNN	996.811	268.333	0.074
GNN-ONLINE	949.694	257.667	0.033
	Deep		
Method	time(s)	branches	%Timeout
BABSR	3811.712	482.167	0.111
MIPPLANET	4566.080	4332.375	0.407
GNN	1893.081	201.500	0.0
GNN-ONLINE	1776.278	192.167	0.0

Table 4.7: Methods' performance on randomly selected properties on CIFAR-10. We show methods' average solving time, average number of branches required and the percentage of timed out properties. We emphasize that MIPplanet branch number is not comparable with those of other methods.

GNN-online. This is also the reason that we did not include MIPplant branch number in Table 4.3 and Table 4.4.

4.8.1.6 LP Solving Time and GNN Computing Time

We mention that LP solving time is the main bottleneck for Branch and Bound based verification methods. Although both GNN evaluation time and LP solving time increase with the size of the network, LP solving time grows at a significantly faster speed. For instance, in CIFAR experiments, GNN requires on average 0.02, 0.03, and 0.08 seconds to make a branching decision on Base, Wide, and Deep models, respectively, but the corresponding one LP solving times on average are roughly 1.1, 4.9, 9.6 seconds. GNN evaluation is almost negligible for large neural networks when compared to LP solving time.

4.8.1.7 Geometric Mean

For all our experiments, we based our analyses on the statistics of average solving time and branching number. To ensure the reported numbers are not biased by

	Easy		
Method	time(s)	branches	%Timeout
BaBSR	471.547	468.117	0.0
MIPplanet	836.049		0.165
GNN	180.638	191.514	0.002
GNN-online	**143.968**	**180.575**	**0.002**
	Medium		
Method	time(s)	branches	%Timeout
BaBSR	1304.032	1236.398	0.0
MIPplanet	1401.273		0.430
GNN	412.880	417.970	0.004
GNN-online	**399.786**	**380.463**	**0.001**
	Hard		
Method	time(s)	branches	%Timeout
BaBSR	3094.794	2271.796	0.413
MIPplanet	1374.794		0.462
GNN	985.142	691.599	0.216
GNN-online	**833.343**	**605.443**	**0.183**

Table 4.8: Methods' performance on the Base model trained on CIFAR-10. For easy, medium and difficult level verification properties, we compare methods' geometric average solving time, geometric average number of branches required and the percentage of timed out properties. GNN-Online outperforms other methods in all aspects.

potential outliers, we measure the methods' performance with the geometric mean as well and summarize results in Table 4.8 and Table 4.9. Statistics of the geometric mean are consistent with those of the arithmetic mean, validating previous analyses.

	Wide			Deep		
Method	time	branches	%Timeout	time	branches	%Timeout
BaBSR	3510.960	699.826	0.0	3580.469	359.464	0.0
MIPplanet	4430.098		0.743	4014.469		0.608
GNN	1403.841	296.821	0.127	1529.542	158.498	0.048
GNN-online	**1316.455**	**279.298**	**0.095**	**1502.349**	**155.523**	**0.040**

Table 4.9: Methods' performance on large models trained on CIFAR-10. For verification properties on the Wide model and the Deep model, we compare methods' geometric average solving time, geometric average number of branches required and the percentage of timed out properties. GNN-Online outperforms other methods in all aspects.

4.8.2 MNIST

We perform similar analyses on MNIST and discuss the results below.

4.8.2.1 Tranferability

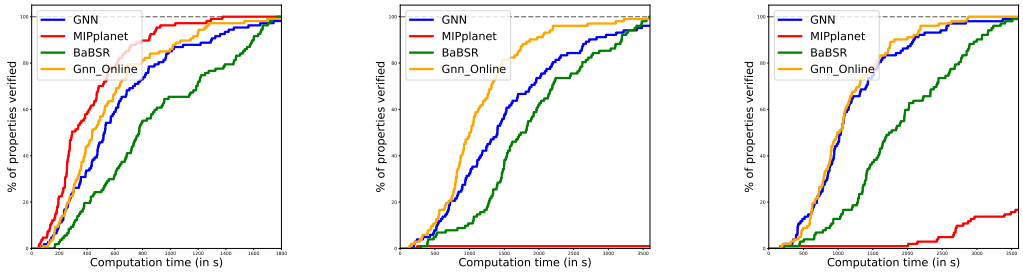
Results are summarised by cactus plots in Figure 4.7. We observe that MIPplanet outperforms all Branch and Bound based methods on the verification properties of the Base model. Given that the network size of the Base model is particularly small (1226 hidden units), we believe that MIP algorithms backed by commercial solvers could be the most effective tool for verification problems of small size. Our conjecture is further confirmed by the fact that MIPplanet timed out on almost all properties of both the Wide and Deep models. On all three models, GNN consistently outperforms BaBSR, demonstrating the transferability of our framework. Finally, when online learning is considered, we found it is effective in fine-tuning the trained GNN and enabling further performance improvements, especially on the Wide model.

	Base		
Method	time(s)	branches	%Timeout
BaBSR	878.226	2659.276	0.0
MIPplanet	**411.167**		**0.0**
GNN	623.291	1549.867	0.019
GNN-online	547.102	**1363.057**	0.009
	Wide		
Method	time(s)	branches	%Timeout
BaBSR	1888.084	399.571	0.0
MIPplanet	3567.109		0.990
GNN	1526.381	322.510	0.040
GNN-online	**1133.429**	**283.673**	**0.010**
	Deep		
Method	time(s)	branches	%Timeout
BaBSR	1895.032	211.644	0.0
MIPplanet	3437.918		0.833
GNN	1189.271	144.238	0.010
GNN-online	**1121.066**	**139.980**	**0.0**

Table 4.10: Methods' performance on different models trained on MNIST. For the Base, Wide and Deep model, we compare methods' average solving time, average number of branches required and the percentage of timed out properties respectively. MIPplanet performs the best on the Base model while GNN-Online outperforms other methods on the Wide and the Deep model.

4.8.2.2 Fail-safe Heuristic Dependence

Results of Table 4.11 ensure that the trained GNN indeed accounts for the most branching decisions.



(a) Base model **(b)** Wide large model **(c)** Deep large model

Figure 4.7: Cactus plots for the Base model (left), Wide large model (middle) and Deep large model (right) for MNIST. For each model, we plot the percentage of properties solved in terms of time for each method. MIPplanet outperforms all Branch and Bound based method on the Base model. However, once model size becomes larger, MIPplanet's performance deteriorates quickly. ON the Wide and Deep model, MIPplanet timed out on most properties and is outperformed by BaBSR. GNN consistently outperforms BaBSR on all three models, demonstrating the transferability of our framework. Again, further small improvement can be achieved through online-learning.

Model	GNN(avg)	BaBSR(avg)	BaBSR(min)	BaBSR(max)
Base	0.962	0.038	0.0	0.204
Wide	0.884	0.116	0.0	0.376
Deep	0.938	0.062	0.0	0.407

Table 4.11: Evaluating GNN's dependence on the fail-safe strategy. Given a MNIST model, we collected the percentage of times GNN branching decision is used and the percentage of times the fail-safe heuristic (BaBSR in our case) is employed for each verification property. We report the average ratio of all verification properties of the same model. To account for extreme cases, we also list the minimum and maximum usage ratios of the fail-safe heuristic for each model.

4.8.2.3 Geometric Mean

The consistency between the results of Table 4.12 and Table 4.10 confirm that our analyses based on arithmetic mean are not biased by outliers.

4.8.3 Transferability Between Datasets

To evaluate whether our framework can generalize further to support transferring between datasets, we tested the CIFAR-10 trained GNN on MNIST verification properties. In detail, we have tested the GNN on 20 randomly picked verification properties of MNIST Base model. We found that BABSR outperforms CIFAR trained GNN on all properties, so the CIFAR trained GNN model does not transfer

	Base		
Method	time(s)	branches	%Timeout
BaBSR	740.740	2210.248	0.0
MIPplanet	**330.455**		**0.0**
GNN	501.966	1268.780	0.019
GNN-online	450.033	**1123.867**	0.009
	Wide		
Method	time(s)	branches	%Timeout
BaBSR	1683.816	362.171	0.0
MIPplanet	3506.400		0.990
GNN	1267.736	284.452	0.040
GNN-online	**973.015**	**254.641**	**0.0**
	Deep		
Method	time(s)	branches	%Timeout
BaBSR	1694.555	187.723	0.0
MIPplanet	3356.780		0.833
GNN	1022.466	131.171	0.010
GNN-online	**991.552**	**128.115**	**0.0**

Table 4.12: Methods' performance on different models trained on MNIST. For the Base, Wide and Deep model, we compare methods' geometric average solving time, geometric average number of branches required and the percentage of timed out properties respectively. MIPplanet performs the best on the Base model while GNN-Online outperforms other methods on the Wide and the Deep model.

to MNIST dataset. This is expected as MNIST and CIFAR images differ significantly from each other.

4.9 Discussion

The key observation of our work is that the neural network we wish to verify can be used to design a GNN to improve branching strategies. This observation can be used in enhancing the performance of other aspects of the Branch and Bound process. Possible future works include employing GNNs to find fast-converging starting values for solving LPs on a neural network and utilising GNNs to develop a lazy verifier that only solves the corresponding LP on a domain when it could lead to pruning.

Before we end, we mention that our GNN framework requires generating the training data set, which incurs high offline computational cost. In our setting, we need to apply the GNN to solve verification problems of similar type many times after the GNN is trained. The high offline cost is thus justified. We do not recommend our framework should the resulting GNN be employed only a few times.

5

Fast Robust Training

Contents

5.1 Introduction

In this chapter, we study the construction of robust neural networks. Instead of focusing on methods to verify whether a given neural network is robust, we aim to devise training strategies that produce robust neural networks, especially in an efficient way. Since we deal with much larger networks than those considered in the previous chapters, the verification methods developed are no longer applicable. We evaluate our training strategies via empirical robust accuracy.

As discussed in Chapter 2, neural networks trained with standard loss functions are often fragile. Several robust training strategies have been proposed [25, 48, 63, 66, 71, 72, 115]. Among them, adversarial training (ADV) [63] and TRADES [115] are two of the most frequently used training methods. However, although developed upon different ideas, both ADV and TRADES require using strong adversarial attacks, generally computed through several steps of projected gradient descent. Such attacks can quickly become prohibitive when model complexity and input dimensions increase, thereby limiting their applicability. We attempt to improve on this limitation. In the following, we propose a novel algorithm that allows efficient robust training. We demonstrate the performance of our method on several standard datasets.

5.2 Effective Use of Weak Adversaries

In this context, we refer to weak adversaries as adversaries obtained using a single gradient step and strong adversaries as adversaries employing multiple gradient steps (more than 5 steps). Using adversaries generated through multiple gradient steps often leads to large performance improvement [63]. However, this is not guaranteed. In addition, the performance gain from adding an extra gradient step is likely to decrease after a certain number of steps. Thus, weak and strong used here are not mathematically validated but more for the sake of naming convenience.

We note that the cost of finding strong adversaries is mainly due to the high number of gradient steps performed. One potential approach to alleviate the problem is to make better use of cheap but weak adversaries. Based on this idea, Wong et al. [101] argue that by using random initialization and a larger step size, adversarial training with weak adversaries found via one gradient step is sufficient to achieve a satisfactory level of robustness. We term this method as one-step ADV from here onwards.

While one-step ADV does indeed exhibit robustness, there is still a noticeable gap when compared to its multi-step counterpart. Adopting the same underlying idea, we further bridge the gap by proposing a new robust training algorithm: Adversarial Training via LocAl Stability (ATLAS). Local stability, in our context, implies the stability of prediction and is the same as local robustness. Specifically, we make the following contributions:

- We take on a novel perspective on robust accuracy and introduce a framework for constructing robust training losses that allow more effective use of adversaries. The framework consists of a local component and a global component. The local component maximizes the effectiveness of an adversary by improving the network's robustness on both the adversary and the points around it. In other words, the local component attempts to increase the radius of a ball, centered at the adversary, on which the network is being robust. The global component combines all local balls in a regularized way to achieve the desired overall robust performance. We emphasise the local and global used here are different from those in other literature. We adopt a finer view by considering every single image instead of the whole distribution: global robustness means robustness on a perturbation ball wrapping around the image and local robustness means robustness on a small ball wrapping around the weak adversary point found for the image.

- Based on the framework and guided by the need of fast robust training, we propose our novel robust training algorithm ATLAS.

- We show that ATLAS makes a more effective use of weak adversaries by favourably comparing it against one-step ADV on three datasets: MNIST, CIFAR-10 and CIFAR-100. Furthermore, with a one-step weak adversary, ATLAS manages to achieve comparable levels of robust accuracy to multi-step state-of-the-art methods on all datasets.

- Finally, we show that when strong adversaries are used, ATLAS matches with the current state-of-the-art on MNIST and outperforms them on CIFAR-10 and CIFAR-100.

5.3 Related Works

Robust training aims to learn a network such that it is able to give the same correct output even when the input is slightly perturbed. Existing robust training algorithms can be divided into two categories: natural image based methods and adversary based methods. Within the first category, the common form of loss is a natural loss term plus a regularizer computed at natural images. We briefly mention some of these methods. Moosavi-Dezfooli et al. [66] observed empirically that reducing the curvature of the loss function and the decision boundary could lead to robust models. The authors thus propose a regularizer based on the Hessian of the loss function. Closely related is the regularizer, introduced in [44, 46, 75], that penalizes the Frobenius norm of the Jacobian of the loss function. Jacobian regularizer can also be seen as a way of reducing the curvature of the decision boundary [46]. Although calculating the norm is computationally expensive, a fast estimator with empirically high accuracy has been developed in Hoffman et al. [44].

We focus on adversary based robust training, as they generally perform better in terms of robust accuracy. Under this category, an early fundamental work is the Fast Gradient Sign Method (FGSM) by Goodfellow et al. [36]. Adversarial Training (ADV) [63] is a multi-step variant of FGSM. Rather than using one-step FGSM, ADV employs multi-step projected gradient descent (PGD) [55] with smaller step sizes to generate perturbed inputs. These modifications have allowed ADV to become one

of the most effective robust training methods so far [6]. Another frequently used robust training method is TRADES [115]. Motivated by its theoretical analyses of the trade-off between natural accuracy and robust accuracy in a binary classification example, TRADES encourages model robustness by adding to the natural loss a regularizer involving adversaries to push away the decision boundary.

Recently, Qin et al. [72] suggest that a robust model can be learned through promoting linearity in the vicinity of the training examples. They designed a local linearity regularizer (LLR), in which adversaries are used to maximize the penalty for nonlinearity. Applying LLR also allows efficient robust training. We note that the underlying idea of LLR is complementary to ATLAS.

One major drawback of adversary based methods [26, 63, 72, 97, 115] is that most of them rely on strong adversaries, computed via expensive PGD. When the input dimension is high and the model structure is complicated, finding adversaries can be too expensive for these methods to work effectively. Several works have researched possible ways to speed up the process. Algorithmically, Zhang et al. [114] cut down the total number of full forward and backward passes for generating multi-step PGD adversaries, while Zhang et al. [117] introduce a parameter to allow early stop PGD. Shafahi et al. [79] reduce adversary overhead by combining weight update and input perturbation update within a single back propagation and use a single step FGSM adversary. Wong et al. [101] argue that the main reason that one-step FGSM, generally regarded as not robust, is effective in [79] is due to the non-zero initialization used. As a result, Wong et al. [101] proposed that with random initialization and a larger step size, the weak adversaries generated by FGSM could lead to models with a high level of robust accuracy.

5.4 Notations

Before we delve into the details of our method, we introduce the following notation to assist the discussion. We first assume that the neural network f contains ReLU activation only for the sake of clarity. This implies that the neural network is piecewise linear. As a consequence, at each $\boldsymbol{x} \in \mathcal{X}$, it is easy to find a weight

matrix $W^{\boldsymbol{x}} \in \mathbb{R}^{\mathcal{C}\times N}$ and a constant vector $\boldsymbol{b}^{\boldsymbol{x}} \in \mathbb{R}^{\mathcal{C}}$ such that $f(\boldsymbol{x};\boldsymbol{\theta}) = W^{\boldsymbol{x}}\boldsymbol{x} + \boldsymbol{b}^{\boldsymbol{x}}$. To simplify the notation, we define

$$\breve{W}^{\boldsymbol{x}} = [\, W^{\boldsymbol{x}} \,|\, \boldsymbol{b}^{\boldsymbol{x}} \,] \in \mathbb{R}^{\mathcal{C}\times(N+1)}, \qquad \breve{\boldsymbol{x}}^T = [\, \boldsymbol{x}^T \,|\, 1 \,] \in \mathbb{R}^{1\times(N+1)}. \tag{5.1}$$

As a result, at each point $\boldsymbol{x}$, we have $f(\boldsymbol{x};\boldsymbol{\theta}) = \breve{W}^{\boldsymbol{x}}\breve{\boldsymbol{x}}$.

In addition, we use a ball $\mathcal{B}_\epsilon(\boldsymbol{x})$ to represent the allowed perturbation. If f is robust on $\mathcal{B}_\epsilon$, we call $\mathcal{B}_\epsilon$ a robust ball.

5.5 The General Framework

Unlike the optimization perspective taken by ADV and the regularization viewpoint adopted by TRADES, our new loss framework is motivated from a geometric standpoint. We start by briefly mentioning its closely related regularization type approaches. Generally, in such approaches, we start from a natural image $\boldsymbol{x}$. By including a loss term for $\boldsymbol{x}$, regularization type losses ensure the model gives the correct prediction $y_{\boldsymbol{x}}$ at $\boldsymbol{x}$. Assume the network makes the correct prediction at the natural image $\boldsymbol{x}$. It follows that, there exists a robust ball $\mathcal{B}_\gamma(\boldsymbol{x})$, where $\gamma \geq 0$ is the maximum radius achievable. Previous use of adversary regularization term encourages the radius γ of the ball $\mathcal{B}_\gamma(\boldsymbol{x})$, centered at $\boldsymbol{x}$, to increase.

We take a different direction in our new loss framework. Instead of focusing on one global ball centered at $\boldsymbol{x}$, we focus on local balls centered at various points $\boldsymbol{x}' \in \mathcal{B}_\epsilon(\boldsymbol{x})$ and hopefully, by combining them in a regularized way, we achieve the desired robustness $\gamma \geq \epsilon$. To facilitate the illustration, we introduce our loss framework through a binary classification problem. Let there be two classes $\mathcal{C} = \{c_1, c_2\}$. For an arbitrary point $\boldsymbol{x} \in \mathcal{X}$, the correct class label is $y_{\boldsymbol{x}} = c_1$. Since we are interested in a network's performance in classifying labels, we compute the logits difference as

$$f_1(\boldsymbol{x};\boldsymbol{\theta}) - f_2(\boldsymbol{x};\boldsymbol{\theta}) = \boldsymbol{d}\breve{W}^{\boldsymbol{x}}\breve{\boldsymbol{x}} \in \mathbb{R},$$

where $\boldsymbol{d} = [1, -1]$ is a row vector. For a binary classification problem with cross-entropy loss, the loss decreases monotonically with the increase of the value $\boldsymbol{d}\breve{W}^{\boldsymbol{x}}\breve{\boldsymbol{x}}$.

To achieve the desired robustness, we require $\boldsymbol{d}\breve{W}^{\boldsymbol{x}'}\breve{\boldsymbol{x}}' > 0$ for all $\boldsymbol{x}' \in \mathcal{B}_\epsilon(\boldsymbol{x})$. In terms of the cross entropy loss ℓ, we need $\ell(\boldsymbol{x}') < -\log(0.5)$ for all $\boldsymbol{x}' \in \mathcal{B}_\epsilon(\boldsymbol{x})$. In Figure 5.1, we show a potential loss curve on $\mathcal{B}_\epsilon(\boldsymbol{x})$ by fixing the value of $\boldsymbol{x}$ on all dimensions apart from one. Our loss framework consists of a local component and a global component. Both components update the parameters of the network model.

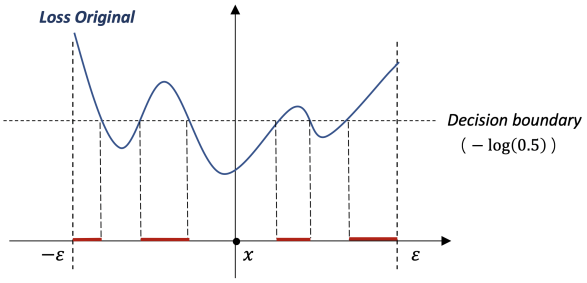


Figure 5.1: Original loss curve on $\mathcal{B}_\epsilon(\boldsymbol{x})$ by fixing the value of $\boldsymbol{x}$ on all dimensions apart from one. Points with the loss above the decision boundary $-\log 0.5$ are successful adversarial points and marked as red.

5.5.1 Local Robustness

For the local component, at each given local point $\boldsymbol{x}' \in \mathcal{B}_\epsilon(\boldsymbol{x})$, the goal is to attain a robust ball $\mathcal{B}_{\gamma'}(\boldsymbol{x}')$ with large radius. We maximize the use of adversaries by treating them as local center points $\boldsymbol{x}'$. In other words, given an adversary $\boldsymbol{x}' \in \mathcal{B}_\epsilon(\boldsymbol{x})$, we need to satisfy two requirements: the model predicts the required label $c_{\boldsymbol{x}}$ at $\boldsymbol{x}'$ and the radius γ' of $\mathcal{B}_{\gamma'}(\boldsymbol{x}')$ should be enlarged. When compared with ADV, the additional second requirement allows us to achieve improved robustness performance even when weak adversaries are used.

The first requirement can be easily met by applying a standard loss term at $\boldsymbol{x}'$ with the label $c_{\boldsymbol{x}}$. For the second requirement, to push away the decision boundary, various methods including Jacobian regularizers [46, 75], MMA [26] and MMR [19] have been introduced. An expected loss curve after introducing the local component is shown in Figure 5.2.

5.5.2 Global Regularization

In terms of the global component, we combine local balls together in a controlled way to further improve the robustness. We first make the following observation.

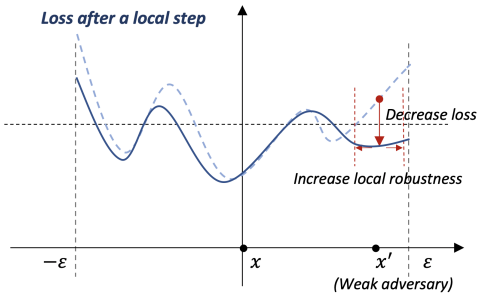


Figure 5.2: Loss curve after a local step. The dashed line is the original loss curve. Provided with a weak adversary $\boldsymbol{x}'$, the local step aims to decrease the loss at $\boldsymbol{x}'$ while increase local robustness so the local robust ball centered at $\boldsymbol{x}'$ has a large radius.

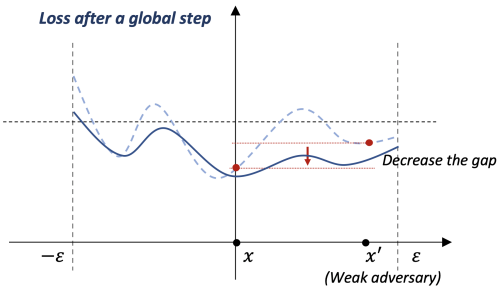


Figure 5.3: Loss curve after a global step. The dashed line is the loss curve after a local step. To further smooth the loss curve and to remove adversaries between $\boldsymbol{x}$ and $\boldsymbol{x}'$, we penalize the difference $p(\boldsymbol{x};\boldsymbol{\theta}) - p(\boldsymbol{x}';\boldsymbol{\theta})$. As a result, the gap between $\ell(\boldsymbol{x})$ and $\ell(\boldsymbol{x}')$ is decreased.

Proposition 1. *Consider a ball $\mathcal{B}_\epsilon(\boldsymbol{x})$ at an arbitrary point $\boldsymbol{x} \in \mathcal{X}$. For any $\boldsymbol{x}^* \in \mathcal{B}_\epsilon(\boldsymbol{x})$, we define $\breve{W}^{\boldsymbol{x}^*}$ and $\breve{\boldsymbol{x}}^*$ accordingly, as in equation (5.1). Let u be a constant such that, for all possible $\breve{W}^{\boldsymbol{x}^*}$, the following is satisfied $\|\breve{W}^{\boldsymbol{x}^*}\|_F \leq u$. Assume $\boldsymbol{x}^1$ and $\boldsymbol{x}^2$ are arbitrary chosen points in $\mathcal{B}_\epsilon(\boldsymbol{x})$. Then for any*

$$\boldsymbol{x}_\eta = \eta \cdot \boldsymbol{x}^1 + (1-\eta) \cdot \boldsymbol{x}^2, \tag{5.2}$$

we have

$$\boldsymbol{d}\breve{W}^{\boldsymbol{x}_\eta}\breve{\boldsymbol{x}}_\eta \geq \frac{1}{2}(\boldsymbol{d}\breve{W}^{\boldsymbol{x}^1}\breve{\boldsymbol{x}}^1 + \boldsymbol{d}\breve{W}^{\boldsymbol{x}^2}\breve{\boldsymbol{x}}^2) - u \cdot L, \tag{5.3}$$

where $L = \sqrt{2}(2\|\breve{\boldsymbol{x}}_\eta\|_2 + \frac{1}{2}\|\breve{\boldsymbol{x}}^1 - \breve{\boldsymbol{x}}^2\|_2)$ is a constant.

Proof. The proof is straightforward. To simplify the notation, we define $s^1 \coloneqq \boldsymbol{d}\breve{W}^{\boldsymbol{x}^1}\breve{\boldsymbol{x}}^1$, $s^2 \coloneqq \boldsymbol{d}\breve{W}^{\boldsymbol{x}^2}\breve{\boldsymbol{x}}^2$ and $s_\eta \coloneqq \boldsymbol{d}\breve{W}^{\boldsymbol{x}_\eta}\breve{\boldsymbol{x}}_\eta$. Also, let l be the lower bound $\boldsymbol{d}\breve{W}^{\boldsymbol{x}^*}\breve{\boldsymbol{x}}^* \geq l$ for any $\boldsymbol{x}^* \in \mathcal{B}_\epsilon(\boldsymbol{x})$. Such l exits because $\mathcal{B}_\epsilon(\boldsymbol{x})$ is a closed and bounded ball and f is continuous. We first note that

$$|s^1 - l| - |s^1 - s_\eta| \leq |s_\eta - l|.$$

Since $s^1 - l > 0$ and $s_\eta - l \geq 0$, we can remove the absolute sign and have

$$s^1 - |s^1 - s_\eta| \leq s_\eta. \tag{5.4}$$

The term $|s^1 - s_\eta|$ can be upper bounded as

$$\begin{aligned} |s^1 - s_\eta| &= |\boldsymbol{d}\breve{W}^{\boldsymbol{x}^1}\breve{\boldsymbol{x}}^1 - \boldsymbol{d}\breve{W}^{\boldsymbol{x}_\eta}\breve{\boldsymbol{x}}_\eta| \\ &\leq |(\boldsymbol{d}\breve{W}^{\boldsymbol{x}^1} - \boldsymbol{d}\breve{W}^{\boldsymbol{x}_\eta})\breve{\boldsymbol{x}}_\eta| + |\boldsymbol{d}\breve{W}^{\boldsymbol{x}^1}(\breve{\boldsymbol{x}}^1 - \breve{\boldsymbol{x}}_\eta)| \\ &\leq 2u\|\boldsymbol{d}\|_2\|\breve{\boldsymbol{x}}_\eta\|_2 + u(1-\eta)\|\boldsymbol{d}\|_2\|\breve{\boldsymbol{x}}^1 - \breve{\boldsymbol{x}}^2\|_2. \end{aligned} \tag{5.5}$$

We further denote $m = 2u\|\boldsymbol{d}\|_2\|\breve{\boldsymbol{x}}_\eta\|_2$ and $v = u\|\boldsymbol{d}\|_2\|\breve{\boldsymbol{x}}^1 - \breve{\boldsymbol{x}}^2\|_2$. By replace $|s^1 - s_\eta|$ with its upper bounds equation (5.5), it follows that

$$s^1 - m - (1-\eta)v \leq s_\eta. \tag{5.6}$$

Similarly for s^2, we have

$$s^2 - m - \eta v \leq s_\eta. \tag{5.7}$$

Finally, combining equation (5.6) and equation (5.7), we get

$$\frac{1}{2}(s^1 + s^2) - m - \frac{1}{2}v \leq s_\eta, \tag{5.8}$$

the desired inequality. □

We now consider the combination of local robust balls. At each loss computation, we have two points: the natural image $\boldsymbol{x}$ and the adversary $\boldsymbol{x}'$. Assume the model makes the correct prediction at the natural image and, after the local component loss, at $\boldsymbol{x}'$, so we have two non-empty local robust balls $\mathcal{B}_\gamma(\boldsymbol{x})$ and $\mathcal{B}_{\gamma'}(\boldsymbol{x}')$. We further assume that these two balls are disjoint for the sake of simplicity. The same underlying idea should work for more general cases. The goal is that, with the global regularization step, points that are misclassified even after the local step can be correctly predicted.

To do so, given that local robust balls $\mathcal{B}_\gamma(\boldsymbol{x})$ and $\mathcal{B}_{\gamma'}(\boldsymbol{x}')$ are disjoint, we assume there exists an adversary point $\boldsymbol{x}^* \in \mathcal{B}_\epsilon(\boldsymbol{x})$ satisfying

$$\boldsymbol{x}^* = \eta^* \cdot \boldsymbol{x} + (1-\eta^*) \cdot \boldsymbol{x}', \tag{5.9}$$

where $\eta^* \in (0,1)$ is a constant depending on $\mathcal{B}_\gamma(\boldsymbol{x})$ and $\mathcal{B}_{\gamma'}(\boldsymbol{x}')$. Then a direct application of Proposition 1 gives

$$\begin{aligned} \boldsymbol{d}\breve{W}^{\boldsymbol{x}^*}\breve{\boldsymbol{x}}^* &\geq \frac{1}{2}(\boldsymbol{d}\breve{W}^{\boldsymbol{x}}\breve{\boldsymbol{x}} + \boldsymbol{d}\breve{W}^{\boldsymbol{x}'}\breve{\boldsymbol{x}}') - u \cdot L \\ &= \frac{1}{2}\boldsymbol{d}(\breve{W}^{\boldsymbol{x}}\breve{\boldsymbol{x}} - \breve{W}^{\boldsymbol{x}'}\breve{\boldsymbol{x}}') + \boldsymbol{d}\breve{W}^{\boldsymbol{x}'}\breve{\boldsymbol{x}}' - u \cdot L \end{aligned} \tag{5.10}$$

where u and L are defined accordingly.

To make $\boldsymbol{x}^*$ no longer an adversary (that is, to encourage $\boldsymbol{d}\breve{W}^{\boldsymbol{x}^*}\breve{\boldsymbol{x}}^* > 0$) after the global combination, we should make the lower bound on the right as high as possible. Since both $\boldsymbol{d}\breve{W}^{\boldsymbol{x}}\breve{\boldsymbol{x}}$ and $\boldsymbol{d}\breve{W}^{\boldsymbol{x}'}\breve{\boldsymbol{x}}'$ have the same max value theoretically, we can increase their sum by first increasing the value of $\boldsymbol{d}\breve{W}^{\boldsymbol{x}'}\breve{\boldsymbol{x}}'$ and then decrease the distance between $\boldsymbol{d}\breve{W}^{\boldsymbol{x}}\breve{\boldsymbol{x}}$ and $\boldsymbol{d}\breve{W}^{\boldsymbol{x}'}\breve{\boldsymbol{x}}'$. We recall that during the local robustness step, we have introduced cross-entropy loss at $\boldsymbol{x}'$, which directly maximizes the value of $\boldsymbol{d}\breve{W}^{\boldsymbol{x}'}\breve{\boldsymbol{x}}'$. In addition, enlarging local robust balls often leads to a decreased value of u. Given that $\boldsymbol{d}\breve{W}^{\boldsymbol{x}'}\breve{\boldsymbol{x}}'$ and u are already optimized, a sound global step is to minimise the distance between $\boldsymbol{d}\breve{W}^{\boldsymbol{x}}\breve{\boldsymbol{x}}$ and $\boldsymbol{d}\breve{W}^{\boldsymbol{x}'}\breve{\boldsymbol{x}}'$. We note that, for the distance to be zero, it is sufficient to have the relative differences $f_2(\boldsymbol{x}';\boldsymbol{\theta}) - f_1(\boldsymbol{x}';\boldsymbol{\theta})$ and $f_2(\boldsymbol{x};\boldsymbol{\theta}) - f_1(\boldsymbol{x};\boldsymbol{\theta})$ to be equal rather than requiring an equality between logits $f(\boldsymbol{x};\boldsymbol{\theta})$ and $f(\boldsymbol{x}';\boldsymbol{\theta})$. To account for this fact, we use prediction probabilities instead. The global regularization step requires minimizing the prediction probability difference $p(\boldsymbol{x};\boldsymbol{\theta}) - p(\boldsymbol{x}';\boldsymbol{\theta})$. As a result, we are likely to turn $\boldsymbol{x}^*$ into a correctly predicted point after the global step, as illustrated in Figure 5.3.

When the loss surface is considered, the global component loss encourages the loss surface to be smoothened and flattend on $\mathcal{B}_\epsilon(\boldsymbol{x})$, so the global component is helpful even when no such adversaries $\boldsymbol{x}^*$ exist. As suggested in [18, 67, 75, 115], these properties of the loss surface are desired and are possibly indispensable for models to be robust.

Overall, various losses can be constructed under this framework. Depending on the problem at hand, one can choose an appropriate regularizer for increasing local robust balls while selecting a suitable metric for penalizing the prediction probability difference.

5.6 ATLAS

Guided by the goal of effective fast robust training, we propose a new training algorithm ATLAS, which stands for Adversarial Training via LocAl Stability,

based on the framework. We introduce and explain our choices for the local and global components.

5.6.1 ATLAS-local

Many existing approaches for increasing robust ball radius are computationally expensive. In this work, we adopt the standard Jacobian approach to meet our local component requirements based on the fact that an efficient estimation algorithm for Jacobian has been developed in [44]. We briefly introduce how Jacobian can be used to increase local robustness.

Given a local point $\boldsymbol{x}'$, let the correct class label be $c = c_{\boldsymbol{x}}$. Then for any $c' \neq c$, the boundary hyper-surface separating classes c and c' consists of points $\boldsymbol{x}^b$ satisfying

$$f_c(\boldsymbol{x}^b; \boldsymbol{\theta}) - f_{c'}(\boldsymbol{x}^b; \boldsymbol{\theta}) = 0. \tag{5.11}$$

Applying the standard formula [15] for computing the distance between a point and a hyper-plane [1], we get the first order approximation distance $d_{c'}$ of $\boldsymbol{x}'$ to the boundary hyper-surface in equation (5.11) under the l_2 norm as

$$d_{c'} = \frac{|f_c(\boldsymbol{x}'; \boldsymbol{\theta}) - f_{c'}(\boldsymbol{x}'; \boldsymbol{\theta})|}{\|\nabla_{\boldsymbol{x}'} f_c(\boldsymbol{x}'; \boldsymbol{\theta}) - \nabla_{\boldsymbol{x}'} f_{c'}(\boldsymbol{x}'; \boldsymbol{\theta})\|_2}. \tag{5.12}$$

Since the above equation holds true for any c', we conclude the model is robust on an l_2 norm ball centered at $\boldsymbol{x}'$ with radius $d \coloneqq \min_{c' \neq c} d_{c'}$. To maximize the radius d, we borrow the following proposition from [46], which introduced a Jacobian regularizer to the natural loss, to provide a lower bound for d.

Proposition 2. *Assume the model is making the correct prediction $c = c_{\boldsymbol{x}}$ for $\boldsymbol{x}'$ and the distance metric is measured via l_2 norm. The first order approximation of the minimum perturbation d that is required to find an adversary example is lower bounded by*

$$d \geq \frac{1}{\sqrt{2}\|J(\boldsymbol{x}')\|_F} \min_{c' \neq c} |f_c(\boldsymbol{x}'; \boldsymbol{\theta}) - f_{c'}(\boldsymbol{x}'; \boldsymbol{\theta})|, \tag{5.13}$$

where $J(\boldsymbol{x}') = \nabla_{\boldsymbol{x}'} f(\boldsymbol{x}'; \boldsymbol{\theta})$ is the Jacobian matrix computed at $\boldsymbol{x}'$ and $\|\cdot\|_F$ is the Frobenius norm.

[1] in this case, the point should be $\boldsymbol{x}'$ and the hyper-plane is tangent to the boundary hyper-surface.

For a larger distance d, we need to both increase the value of $|f_c(\boldsymbol{x}';\boldsymbol{\theta}) - f_{c'}(\boldsymbol{x}';\boldsymbol{\theta})|$ and decrease the Frobenius norm of $J(\boldsymbol{x}')$. The first term can be easily taken care of by using a cross-entropy loss on $\boldsymbol{x}'$ while for the second term, we can include a cheap approximation of $\|J(\boldsymbol{x}')\|_F^2$, denoted by $\|J(\boldsymbol{x}')\|_F^{approx}$ as a penalty term in our loss. Using the idea of random projection, Hoffman et al. [44] shows theoretically and empirically that $\|J(\boldsymbol{x}')\|_F^{approx}$ can be estimated with high quality by one backward pass regardless the total number of classes C.

To combine the above analyses, for the local robustness component, we introduce the following loss

$$\ell_{\text{local}}(\boldsymbol{x}') = \ell^{CE}(\boldsymbol{x}') + \alpha \cdot \|J(\boldsymbol{x}')\|_F^{approx}, \tag{5.14}$$

where α is a positive scalar. It is worth noting that our local loss shown in equation (5.14) is different from that of [44, 46, 75] as the loss is evaluated at an adversary $\boldsymbol{x}'$ instead of the image $\boldsymbol{x}$. In the fast robust training setting, $\boldsymbol{x}'$ is a weak adversary obtained through one-step FGSM.

5.6.2 ATLAS-global

To penalize the prediction probability difference, a natural and frequently used choice is Kullback–Leibler (KL) distance. We thus formulate the global loss as

$$\ell_{\text{global}}(\boldsymbol{x}') = \beta \cdot \text{KL}(p(\boldsymbol{x}';\boldsymbol{\theta}) \| p(\boldsymbol{x};\boldsymbol{\theta})), \tag{5.15}$$

where β is a positive constant.

Although generally $\text{KL}(p(\boldsymbol{x}';\boldsymbol{\theta}) \| p(\boldsymbol{x};\boldsymbol{\theta})) \neq \text{KL}(p(\boldsymbol{x},\boldsymbol{\theta}) \| p(\boldsymbol{x}',\boldsymbol{\theta}))$ due to the asymmetric nature of KL distance, the same underlying nature of both terms means it is sufficient to use one direction for the penalty. In our loss, we have use the prediction distribution at $\boldsymbol{x}'$ to be the base distribution. There are two reasons for this choice: firstly, we want it to be consistent with the fact that the local component loss is computed on $\boldsymbol{x}'$; and secondly, we hope it could mitigate a possible over-fitting issue. If a model over-fits at the original image $\boldsymbol{x}$ causing an element $p_i(\boldsymbol{x};\boldsymbol{\theta})$ of $p(\boldsymbol{x};\boldsymbol{\theta})$ to be close to 0, the distance between $p_i(\boldsymbol{x};\boldsymbol{\theta})$ and $p_i(\boldsymbol{x}';\boldsymbol{\theta})$ would

not be penalized effectively. In extreme cases, the difference will not be penalized at all if $p_i(\boldsymbol{x};\boldsymbol{\theta}) = 0$. Using $p(\boldsymbol{x}';\boldsymbol{\theta})$ as the base might alleviate this issue, as $\boldsymbol{x}'$ keeps changing during each epoch. Under the same reasoning, we point out TRADES relies on finding strong adversaries. Since it uses the natural image $\boldsymbol{x}$ to compute cross-entropy loss and as the base distribution in the KL regularizer, TRADES cannot use adversaries effectively when they are weak. This is consistent with what we see in the experiments section that is, TRADES with one step FGSM is largely outperformed by other methods on challenging datasets. Furthermore, in the later section, we show that when putting more emphasis on adversaries by replacing $\boldsymbol{x}$ with $\boldsymbol{x}'$ in TRADES, improved robust accuracy can be achieved.

5.6.3 Final Loss

Integrating the above analyses, we propose the following as the final loss for ATLAS. Given $\alpha > 0$ and $\beta > 0$, the loss is formulated as

$$\ell_{\text{ATLAS}} = \frac{1}{|\mathcal{X}|}\sum_{\boldsymbol{x}\in\mathcal{X}} \underbrace{\ell^{\text{CE}}(\boldsymbol{x}') + \alpha \cdot \|J(\boldsymbol{x}')\|_F^{approx}}_{\text{local}} + \underbrace{\beta \cdot \text{KL}(p(\boldsymbol{x}';\boldsymbol{\theta})\|p(\boldsymbol{x};\boldsymbol{\theta}))}_{\text{global}}, \quad (5.16)$$

where $\boldsymbol{x}' \in \mathcal{B}_\epsilon(\boldsymbol{x})$. We mention that although our analyses are carried out in l_2 norm, our results are easily generalizable to other norms.

5.7 Experiments: General Setup

We evaluate the performance of ATLAS on three datasets: MNIST, CIFAR-10 and CIFAR-100. We consider the cases of training robustness models with weak adversaries generated by FGSM and with strong adversaries by multi-step PGD. Here, we fixed the number of gradient steps instead of the time budget for generating adversaries. There are two reasons for this choice. Firstly, it is straightforward to compare the total time required for the generation. Secondly, the budget time may not be divisible by the time required for each gradient step for some methods. This leads to under-usage of the time budget for some methods and hence unfair comparison among methods. To be consistent with the general experimental setting, we adopt l_∞ norm and use the perturbation value $\epsilon = 0.3$ for MNIST and $\epsilon = 8/255$

for CIFAR-10 and CIFAR-100. MNIST is trained on a 4-layer CNN, which consists of 2 convolutional layers followed by 2 fully connected layers. CIFAR-10 and CIFAR-100 are trained on Wide-ResNet-28-8 [111].

5.7.1 Methods

We compare our method with ADV and TRADES. We term methods that use weak FGSM adversaries as **one-step methods**. Following the advice from Wong et al. [101], we apply FGSM, combined with random initialization and a pre-determined step size (details for determining the FGSM step size is provided later) to generate adversarial examples. One-step ADV is an implementation of Wong et al. [101]. To be consistent with their multi-step variants, we use cross-entropy loss for computing the gradient in FGSM for ADV and ATLAS while employ KL distance for TRADES. **Multi-step methods** are trained on strong adversaries generated by PGD: 20 steps for the MNIST and 10 steps for CIFAR-10 and CIFAR-100. We also include the Jacobian penalty loss [44, 46, 75] due to its close relation to our local loss component. We refer to it as zero-step JAC to emphasize the fact that it does not require adversaries.

5.7.2 Robustness Evaluation

Extensive robustness evaluations are conducted to avoid a false sense of security. Black box attack generates adversaries on surrogate models with 1000 PGD steps. In terms of white box attacks, apart from an initial robustness evaluation with PGD-20, we further employ three strong white attacks: PGD-1000, Untargeted (Un-T) attack [12], and Multi-targeted (Multi-T) attack [37]. On the big Wide-ResNet model, we run 10 restarts with 100 steps for Untargeted attack and 5 restarts with 20 steps for Multi-target attack. We reduce the number of restarts and steps for the Multi-target attack to account for the fact that each incorrect class needs to be tested. Due to the large number of classes, Multi-target is not performed on CIFAR-100. On the other hand, we perform 20 restarts with 50 steps for both attacks on the small CNN model. Random initialization is applied in all white box attacks. On the challenging datasets CIFAR-10 and CIFAR-100, one more robust evaluation AutoAttack [20] is

carried out to obtain a more comprehensive understanding of the models' robustness performance. The related results are separately reported in Section 5.8.4.

For each method, attack results are reported for a model at the epoch that gives the best validation robust accuracy during the training. Details for computing the validation robust accuracy are provided in the following subsection.

5.7.3 Training

We apply the same training set-up for all methods. We use batch size 128 for MNIST and 64 for CIFAR datasets. During training, we start with $\epsilon = 0$ and gradually increase its value to 0.3 for MNIST or 8/255 for CIFAR-10 and CIFAR-100 in the first 15 epochs.

We employ SGD optimizer on all three datasets. On MNIST, we choose 0.01 for the starting learning rate while on CIFAR-10 and CIFAR-100, we set the starting learning rate to be 0.1, the momentum to be 0.9 and the weight decay to be $2e^{-4}$. To encourage efficient robust training, instead of a fixed learning rate scheduler that decreases the learning rate at pre-specified epoch numbers, we rely on the robust accuracy of a validation dataset to guide the training process. To be more specific, we first randomly select 5000 images from the training dataset to form a validation set before the training starts. During training, after each epoch, we run a multi-step PGD attack to evaluate the model's robust accuracy on the validation set. If the validation robust accuracy does not improve for a fixed number of consecutive epochs (we refer to the number as plateau epoch number), we decrease the learning rate by five. If the robust accuracy does not improve for ten consecutive epochs, we terminate the training process. For MNIST, we apply a 20-step PGD attack with step size 0.01 to determine the validation robust accuracy. In terms of CIFAR-10 and CIFAR-100, 10-step PGD attack with step size 0.007 is used.

5.7.3.1 Plateau Epoch Number and FGSM Step Size

Due to the simplicity of the MNIST dataset and similar behaviour on both CIFAR-10 and CIFAR-100, we determine the plateau epoch number and the FGSM step

size by experimenting with one-step ADV on CIFAR-10.

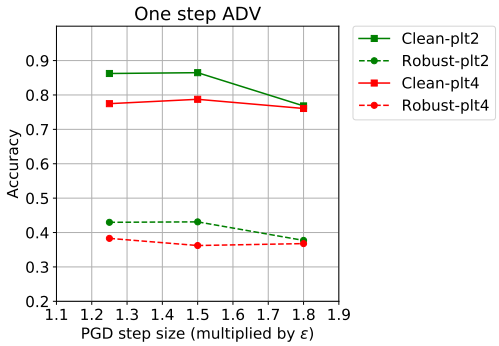


Figure 5.4: Clean and robust accuracy for one-step ADV at different step sizes on the training dataset

We use the strongest Multi-T attack for evaluating model's robust accuracy on a randomly selected subset of the training dataset. In Figure 5.4, the green lines show the results for the plateau epoch to be 2 (plt2) while the red lines are for the plateau epoch to be 4 (plt4). In terms of step sizes, we tested 3 possible values: 1.25ϵ (the suggested value by Wong et al. [101]), 1.5ϵ and 1.8ϵ. It is clear to see that with plt2, the models perform better in both nominal accuracy and robust accuracy at all three step sizes. We thus use plateau epoch number 2 to adjust the learning rate in all our experiments for comparable results. When the step size is considered, both 1.25ϵ and 1.5ϵ give satisfactory results. Since other methods show more stable performance with the step size 1.25ϵ, we use it for generating weak adversaries in our experiments.

5.7.4 Final Chosen Parameters

For each method, the set of parameters that gives the best validation robust accuracy is selected. Models trained with these parameters are evaluated under attacks and their performance are reported in Table 5.1. The final chosen parameters are listed below.

MNIST: zero-step JAC is trained with $\alpha_{\text{JAC}} = 0.5$; one-step and multi-step TRADES are trained with $\beta_{\text{TRADES}} = 1$; one-step ATLAS is trained with $\alpha = 1e{-}05$

and $\beta = 0.3$ while multi-step ATLAS is trained with $\alpha = 5e-06$ and $\beta = 0.2$.

CIFAR-10: zero-step JAC is trained with $\alpha_{\text{JAC}} = 0.5$; one-step TRADES is trained with $\beta_{\text{TRADES}} = 2$; multi-step TRADES is trained with $\beta_{\text{TRADES}} = 4$; one-step ATLAS is trained with $\alpha = 0.0002$ and $\beta = 5$ and multi-step ATLAS is trained with $\alpha = 0.0001$ and $\beta = 5.0$.

CIFAR-100: zero-step JAC is trained with $\alpha_{\text{JAC}} = 0.5$; one-step TRADES is trained with $\beta_{\text{TRADES}} = 2$; multi-step TRADES is trained with $\beta_{\text{TRADES}} = 4$; one-step ATLAS is trained with $\alpha = 0.0002$ and $\beta = 5.0$ and multi-step ATLAS is trained with $\alpha = 0.0001$ and $\beta = 10.0$.

5.8 Experiments: Results and Analyses

We mention that with the learning rate scheduler guided by the robust accuracy on the validation set, all models achieved their best performance around 30 epochs, except 12 epochs for zero-step JAC on MNIST. In terms of robustness performance, slight performance improvements for all methods and narrower performance gaps are observed when all models are trained for 70 epochs with a fixed learning rate scheduler on MNIST. On the other hand, on CIFAR-10 and CIFAR-100, we found that all methods perform better with the guided learning rate scheduler. Since we are interested in efficient robust learning, we use the guided learning rate scheduler for all experiments. We summarize the attack results in Table 5.1.

5.8.1 CIFAR-10

We study the challenging CIFAR-10 first. For zero-step Jac, we observe that to achieve a roughly 30% robust accuracy, nominal accuracy is dropped to below 65%. Directly including a Jacobian penalty at natural images could thus lead to over-regularization issues. This may also be the reason the method is used as a post-processing technique in [46]. On the other hand, by using local adversaries instead of natural images, ATLAS with the local component only manages to obtain robust

accuracy without sacrificing the nominal accuracy much, which is demonstrated in Section 5.8.5. When TRADES is considered, we find that its effectiveness relies on strong adversaries: the robust accuracy increases sharply from one-step case to multi-step case. Since TRADES computes the loss at natural images and use them as the base distribution in the KL penalty, weak adversaries are not effectively used.

Overall, one-step ATLAS outperforms other one-step methods in all strong white box attacks. Even when compared with methods trained with 10 steps PGD, the performance gap between one-step ATLAS and multi-step ADV under the strongest Multi-T attack is below 1.5%. Although adding local and global penalties results in extra computational time for one-step ATLAS (0.73s) when compared to one-step ADV (0.25s), the fact that ATLAS with weak adversaries achieves comparable high robust accuracy to that of multi-step ADV (1.38s) still allows a roughly 50% speed-up in training. For a fair comparison, we also evaluate an ADV model with 5 PGD steps. The 5-step ADV model requires slightly more time (0.75s per batch) but is less robust than one-step ATLAS by reaching 44.93% accuracy under the Multi-T attack. When trained against strong adversaries, multi-step ATLAS wins over the other two multi-step methods in all strong white box attacks. Finally, we observe that increased robustness accuracy has led to decreased nominal accuracy. This is consistent with reported results in other robust training literature [48, 72, 115]. It is possibly caused by the fact that many methods, including ours, achieve robustness by smoothening the loss surface through regularization.

5.8.2 CIFAR-100 and MNIST

Similar performances are observed on the CIFAR-100. The key fact to notice here is that the per batch training cost required for CIFAR-100 is similar to that of CIFAR-10. Since we apply an efficient approximation to evaluate $\|J(\boldsymbol{x})\|_F^2$, the increase of total class number does not result in a rise of total computational cost. Again, we evaluate ADV model with 5 PGD steps, which requires 0.75s per batch and reaches 25.21% robust accuracy under Un-T attack. On MNIST, one-step

Table 5.1: Robustness performance for MNIST on a small CNN and for CIFAR-10 and CIFAR-100 on Wide-Resnet 28-8. The higher, the better.

	Model	Clean accuracy	Black box attack	White box attack: PGD-20 attack	White box attack: PGD-1000 attack	White box attack: Un-T attack	Time: Multi-T attack	Time: per batch
MNIST	zero-step JAC	98.34%	89.84%	58.09%	2.48%	28.83%	28.72%	0.012s
	one-step ADV	**99.49**%	96.26 %	96.29%	83.25%	91.02%	90.62%	0.009s
	one-step TRADES	99.40%	96.21%	96.28%	83.05%	90.66%	90.02%	0.016s
	one-step ATLAS	99.47%	**96.36**%	**96.84**%	**89.56**%	**92.44**%	**92.04**%	0.018s
	twenty-step ADV	99.48%	96.40%	97.16%	**92.32**%	93.01%	**92.87**%	0.077s
	twenty-step TRADES	**99.49**%	96.21%	97.01%	90.51%	92.49%	92.21%	0.097s
	twenty-step ATLAS	99.44%	**96.45**%	**97.19**%	92.20%	**93.03**%	92.74%	0.086s
CIFAR-10	zero-step JAC	63.24%	60.45%	34.40%	34.34%	31.13%	30.32%	0.47s
	one-step ADV	86.24%	82.53%	45.73%	44.98%	45.14%	42.97%	0.25s
	one-step TRADES	**86.84**%	**82.63**%	40.03%	39.41%	39.31%	38.14%	0.55s
	one-step ATLAS	84.52%	81.06%	**49.01**%	**48.59**%	**48.54**%	**46.55**%	0.73s
	ten-step ADV	**84.77**%	82.53%	50.60%	50.22%	49.74%	47.96%	1.38s
	ten-step TRADES	84.01%	**82.63**%	52.70%	52.45%	50.29%	49.66%	2.02s
	ten-step ATLAS	82.94%	81.06%	**53.45**%	**53.10**%	**51.51**%	**50.16**%	1.88s
CIFAR-100	zero-step JAC	47.96%	44.67%	15.39%	15.18%	14.35%	-	0.47s
	one-step ADV	48.88%	45.59%	19.89%	19.72%	19.04%	-	0.25s
	one-step TRADES	**62.58**%	**57.64**%	16.93%	16.21%	14.83%	-	0.53s
	one-step ATLAS	60.98%	57.04%	**26.30**%	**25.94**%	**26.28**%	-	0.73s
	ten-step ADV	**60.72**%	**58.15**%	27.71%	27.43%	26.83%	-	1.37s
	ten-step TRADES	59.76%	57.78%	26.27%	26.10%	23.86%	-	2.00s
	ten-step ATLAS	59.62%	57.83%	**29.12**%	**28.89**%	**27.77**%	-	1.87s

ATLAS performs the best as well. Since MNIST is a simple dataset, all multi-step methods obtain high robust accuracy on the small CNN model.

5.8.3 Additional Loss

We study an additional loss to demonstrate ATLAS's effective use of weak adversaries. We recall that within our framework, the key to maximize the use of weak adversaries is to treat them as center points of local balls in the local component. We thus consider the following loss, for $\boldsymbol{x}' \in \mathcal{B}_\epsilon(\boldsymbol{x})$,

$$\ell_{\text{TradesJac}} = \frac{1}{|\mathcal{X}|} \sum_{\boldsymbol{x} \in \mathcal{X}} \underbrace{\ell^{\text{CE}}(\boldsymbol{x}) + \alpha_{\text{tj}} \cdot \|J(\boldsymbol{x})\|_F^{approx}}_{\text{local}} + \underbrace{\beta_{\text{tj}} \cdot \text{KL}(p(\boldsymbol{x};\boldsymbol{\theta}) \| p(\boldsymbol{x}';\boldsymbol{\theta}))}_{\text{global}} .$$

We term the loss as TradesJac, as it can also be seen as a combination of the TRADES loss and the zero-step JAC loss. Adversaries are computed by using the

gradient of the KL-distance. The main difference between TradesJac and ATLAS is in the local component, where TradesJac uses natural images as center points for local balls instead of weak adversaries.

We train models using TradesJac with various sets of values for α_{tj} and β_{tj} and reported results for the model with the best validation robust accuracy. Results can be found in Table 5.2. On both CIFAR-10 and CIFAR-100, we have set $\alpha_{\text{tj}} = 0.0002$ and $\beta_{\text{tj}} = 0.5$. We have also included other one-step methods for easy comparison. It is clear to see that ATLAS outperforms TradesJac in all attacks, confirming that ATLAS is able to employ weak adversaries more effectively in achieving robust accuracy.

5.8.4 Additional Attacks

We perform an additional robustness evaluation AutoAttack [20] on CIFAR-10 and CIFAR-100. AutoAttack is an ensemble of parameter-free attacks. We have used the standard version of AutoAttack for all evaluations. Results for CIFAR-10 and CIFAR-100 can be found in Table 5.3 and Table 5.4 respectively. ATLAS gives the overall best robust performance among all methods in both weak and strong adversary cases.

Table 5.2: Robustness performance for one-step methods on CIFAR-10 and CIFAR-100. All models are trained on Wide-Resnet 28-8. The higher, the better.

				White box attack				Time
	Model	Clean accuracy	Black box attack	PGD-20 attack	PGD-1000 attack	Un-T attack	Multi-T attack	per batch
CIFAR-10	one-step ADV	86.24%	82.53%	45.73%	44.98%	45.14%	42.97%	0.25s
	one-step TRADES	**86.84**%	**82.63**%	40.03%	39.41%	39.31%	38.14%	0.55s
	one-step ATLAS	84.52%	81.06%	**49.01**%	**48.59**%	**48.54**%	**46.55**%	0.73s
	one-step TradesJac	84.38%	81.54%	46.04%	45.75%	44.60%	43.59%	1.14s
CIFAR-100	one-step ADV	48.88%	45.59%	19.89%	19.72%	19.04%	-	0.25s
	one-step TRADES	**62.58**%	**57.64**%	16.93%	16.21%	14.83%	-	0.53s
	one-step ATLAS	60.98%	57.04%	**26.30**%	**25.94**%	**26.28**%	-	0.73s
	one-step TradesJac	58.15%	54.56%	21.22%	20.89%	19.78%	-	1.15s

Table 5.3: Robustness performance for CIFAR-10 on Wide-Resnet 28-8. The higher the better.

	Model	Auto attack
CIFAR-10	zero-step JAC	30.03%
	one-step ADV	42.49%
	one-step TRADES	37.74%
	one-step ATLAS	**46.16**%
	one-step TradesJac	43.16%
	ten-step ADV	47.56%
	ten-step TRADES	49.40%
	ten-step ATLAS	**49.82**%

Table 5.4: Robustness performance for CIFAR-100 on Wide-Resnet 28-8. The higher the better.

	Model	Auto attack
CIFAR-100	zero-step JAC	11.85%
	one-step ADV	17.17%
	one-step TRADES	12.75%
	one-step ATLAS	**23.42**%
	one-step TradesJac	17.99%
	ten-step ADV	24.64%
	ten-step TRADES	22.78%
	ten-step ATLAS	**25.77**%

5.8.5 Ablation Studies

We perform ablation studies for ATLAS. We consider the one-step setting. Recall that ATLAS consists of two components: a local component and a global component. We test the effect of the local component by setting $\beta = 0$ to get **ATLAS-l**:

$$\ell_{\text{ATLAS-l}} = \frac{1}{|\mathcal{X}|} \sum_{\boldsymbol{x} \in \mathcal{X}} \ell^{\text{CE}}(\boldsymbol{x}') + \alpha \cdot \mathcal{A}(J(\boldsymbol{x}')), \quad \boldsymbol{x}' \in \mathcal{B}_\epsilon(\boldsymbol{x}). \tag{5.17}$$

and the global component by setting $\alpha = 0$ to get **ATLAS-g**:

$$\ell_{\text{ATLAS-g}} = \frac{1}{|\mathcal{X}|} \sum_{\boldsymbol{x} \in \mathcal{X}} \ell^{\text{CE}}(\boldsymbol{x}') + \beta \cdot \text{KL}(p(\boldsymbol{x}'; \boldsymbol{\theta}) \| p(\boldsymbol{x}; \boldsymbol{\theta})), \quad \boldsymbol{x}' \in \mathcal{B}_\epsilon(\boldsymbol{x}). \tag{5.18}$$

We apply the strongest Multi-T attack on CIFAR-10 and Un-T attack on CIFAR-100. Results are summarised in Table 5.5. When CIFAR-10 is considered, both ATLAS-l and ATLAS-g alone are effective in improving the model's robust accuracy. Combining the local and global components (one-step ATLAS) leads to a further robustness improvement. In terms of time per batch, apart from the cross-entropy loss term at the adversary in both ATLAS-l and ATLAS-g, ATLAS-l requires computing the gradient of a Jacobian approximation term and is computationally more expensive than ATLAS-g, which calculates the gradient of a KL penalty term instead. Furthermore, the fact that ATLAS-g is computationally cheaper than TRADES is because it uses cross-entropy loss to derive a FGSM adversary while TRADES uses KL distance. Similar performance is observed on CIFAR-100.

Table 5.5: Robustness performance for CIFAR-10 and CIFAR-100 on Wide-Resnet 28-8. The higher, the better.

	Model	Clean accuracy	Un-T attack	Multi-T attack	Time per batch
CIFAR-10	one-step ADV	86.24%	-	42.97%	0.25s
	one-step TRADES	**86.84%**	-	38.14%	0.55s
	one-step ATLAS-l	85.24%	-	44.46%	0.60s
	one-step ATLAS-g	82.07%	-	44.43%	0.38s
	one-step ATLAS	84.52%	-	**46.55%**	0.73s
CIFAR-100	one-step ADV	48.88%	19.04%	-	0.25s
	one-step TRADES	**62.58%**	14.83%	-	0.53s
	one-step ATLAS-l	58.46%	24.73%	-	0.60s
	one-step ATLAS-g	51.88%	19.71%	-	0.37s
	one-step ATLAS	60.98%	**26.28%**	-	0.73s

We show the model's performance when trained with ATLAS-l and ATLAS-g at different parameter values. Results for CIFAR-10 are summarised in Figure 5.5 and 5.6 while results for CIFAR-100 are summarised in Figure 5.7 and 5.8. For CIFAR-10, we see that the clean accuracy decreases with the increase of α value for ATLAS-l, but the robust accuracy retains at the similar level. In terms of ATLAS-g, the same trend is observed and when the value of β is large, both clean and robust accuracy decline. On CIFAR-100, there is a big drop in robust accuracy for ATLAS-l when α rises while robust accuracy for ATLAS-g is relatively insensitive to the change of β value.

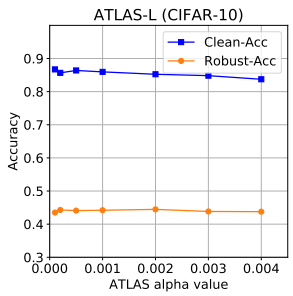


Figure 5.5: Clean and robust accuracy for ATLAS-l at different α on CIFAR-10

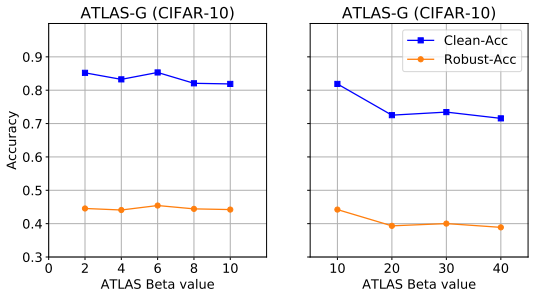


Figure 5.6: Clean and robust accuracy for ATLAS-g at different β on CIFAR-10

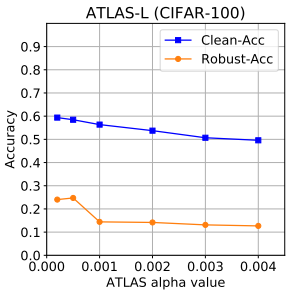


Figure 5.7: Performance of ATLAS-l at different α on CIFAR-100

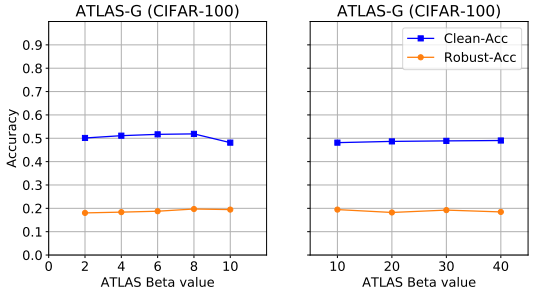


Figure 5.8: Clean and robust accuracy for ATLAS-g at different β on CIFAR-100

5.8.6 ATLAS-g vs. TRADES

ATLAS-g and TRADES take similar forms. The only differences between these two methods are: firstly, ATLAS-g computes cross-entropy loss at the adversary while TRADES at the natural image; secondly, ATLAS-g uses cross-entropy loss to find a gradient in FGSM while TRADES employs KL distance.

On CIFAR-10, it is clear that ATLAS-g outperforms TRADES on both clean and robust accuracy for all tested β values. This observation supports the fact that ATLAS-g, by computing the loss at weak adversaries, allows a more effective use of them. Maximizing the use of weak adversaries is important in the fast robust training setting. In terms of CIFAR-100, ATLAS-g outperforms TRADES on robust accuracy. However, TRADES achieves higher clean accuracy with small β values. The regularization effect of ATLAS-g could be higher than TRADES.

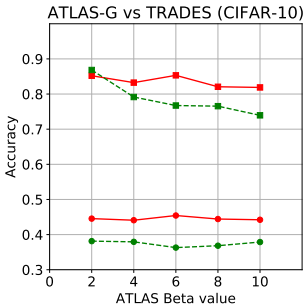


Figure 5.9: ATLAS-g vs. TRADES on CIFAR-10

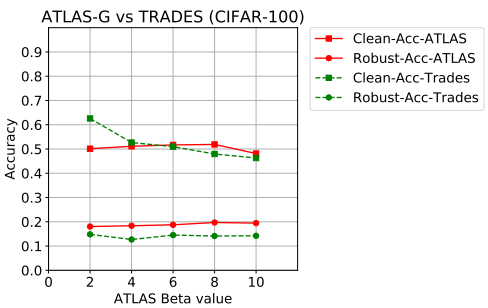


Figure 5.10: ATLAS-g vs. TRADES on CIFAR-100

5.8.7 Catastrophic Overfitting

On CIFAR-10, we constantly find that for one-step TRADES, after a certain number of epochs, there is a sudden drop of more than 10% on validation robust accuracy. This phenomenon is referred to as catastrophic overfitting [5, 101]. We observe that when this behaviour occurs, the approximated value $\|J(\boldsymbol{x}')\|_F^{approx}$ increases steeply, which is consistent with what has been observed in [5]. When dealing with catastrophic overfitting is the main concern, Andriushchenko and Flammarion [5] argue the key is to increase local linearity and they introduce a regularizer to maximize gradient alignment for points within the perturbation ball. In our case, the issue could be similarly resolved by increasing the coefficient α to encourage local linearity. However, since the goal of this project is to achieve high robust accuracy fast, we do not restrict ourselves to models without catastrophic overfitting only, which are likely to compensate the robust accuracy for stability. Instead, we use early termination and consider a wider range of models. We mention that the same phenomenon is observed on one-step ADV and on one-step ATLAS when α is small but less frequently.

We give more details on the catastrophic overfitting behaviour in our experiments. We first show typical training plots for both cases. The upper row of Figure 5.11 contains plots for a model when catastrophic overfitting is avoided and the bottom row shows the plots for a model when catastrophic overfitting happens. It is clear to see that catastrophic overfitting is highly associated with a sharp increase of Jacobian value, which is consistent with the observations made in [5]. As a result, for both ATLAS-l and ATLAS, the catastrophic overfitting can be effectively avoided by increasing the value for α as it directly penalizes large Jacobian value. Finally, we note that, for a stable training performance, the choice of α depends on the value of β. That is, a large β value requires a large α value.

5.8.8 Parameter Choices: One-Step Methods

To better understand the trade-off between clean accuracy and robust accuracy and to test the methods' sensitivity to parameters, we train each method with various

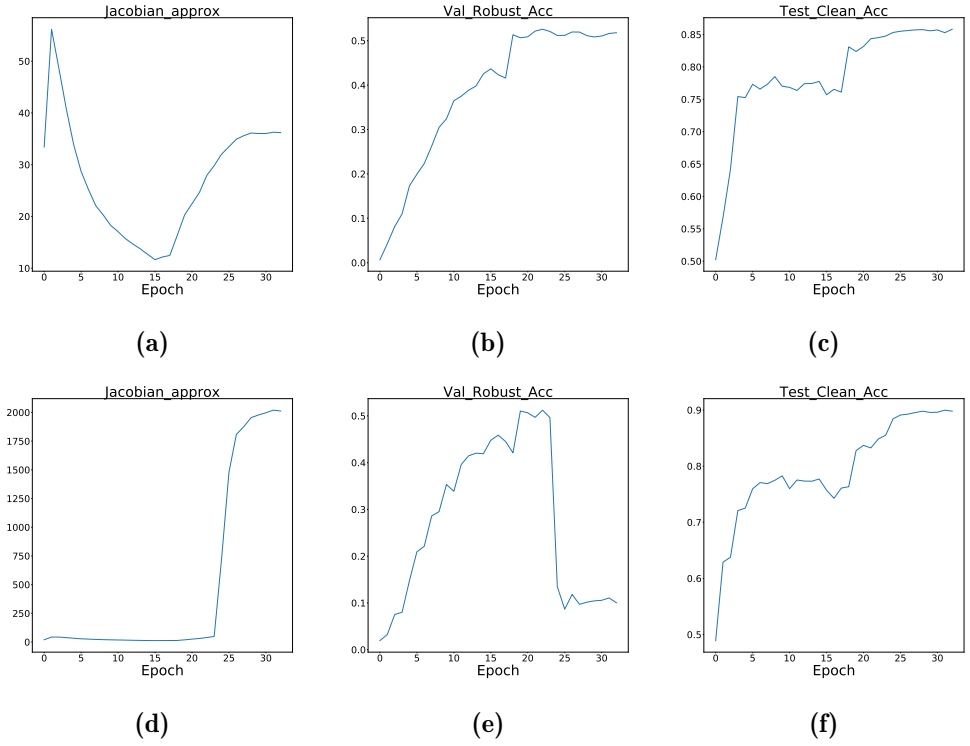


Figure 5.11: We show typical training plots for a model on CIFAR-10 when catastrophic overfitting is avoided in the upper row and when catastrophic overfitting happens in the bottom row. Upper row plots are for the ATLAS model with $\alpha = 0.0002$ and $\beta = 5.0$ while the bottom row plots are for the ATLAS model with $\alpha = 0$ and $\beta = 8.$. The figures on the left shows the value of $\|J(\boldsymbol{x}')\|_F^{\text{approx}}$ throughout the training. Catastrophic overfitting is associated with a steep increase of the Jacobian value. The figures in the middle gives the trend for robust accuracy on the validation dataset. Validation robust accuracy drops sharply with the sudden increase of the Jacobian value. Finally, on the right, we have plots for clean accuracy on the test set after each epoch. A further increase to almost 90% clean accuracy is experienced in the catastrophic overfitting case.

parameter choices. The final models are evaluated on the testing dataset.

5.8.8.1 Zero-Step Jac

Although zero-step Jac does not require adversaries, we used 2 epochs for adjusting the learning rate via validation robust accuracy to be consistent. Model's performance over a range of α_{Jac} values is shown in Figure 5.12 for CIFAR-10 and 5.13 for CIFAR-100. There is a large trade-off between clean and robust accuracy for small α_{Jac} values. Then both clean and robust accuracy decrease with the increase of α_{Jac}.

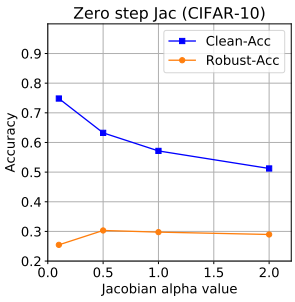


Figure 5.12: Clean and robust accuracy for zero-step JAC at different α_{Jac} on CIFAR-10

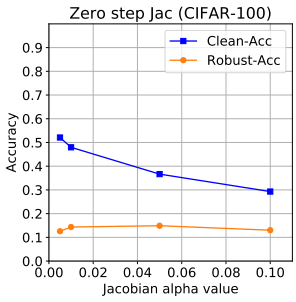


Figure 5.13: Clean and robust accuracy for zero-step JAC at different α_{Jac} on CIFAR-100

5.8.8.2 One-Step TRADES

In Figure 5.14 (CIFAR-10) and 5.15 (CIFAR-100), we show one-step TRADES at various β_{TRADES} values. It is easy to see that increasing the value of β_{TRADES} mainly hurts the clean accuracy without improving robust accuracy.

5.8.8.3 One-Step ATLAS

In Figure 5.16, we give clean and robust accuracy for models trained at different α and β values on CIFAR-10. In Figure 5.17, we give clean and robust accuracy for models trained at different α and β values on CIFAR-100. A slight trade-off between clean and robust accuracy can be observed.

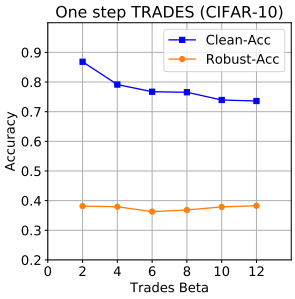


Figure 5.14: Clean and robust accuracy for one-step TRADES at different β_{TRADES} on CIFAR-10

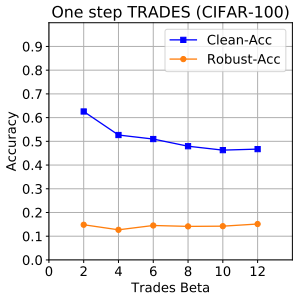


Figure 5.15: Clean and robust accuracy for one-step TRADES at different β_{TRADES} on CIFAR-100

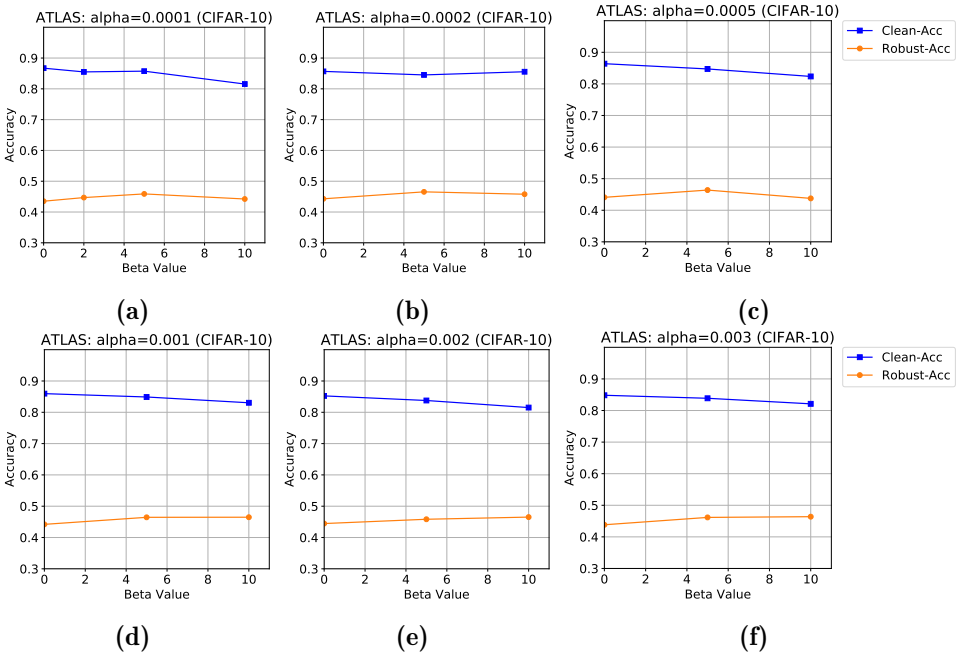


Figure 5.16: Clean and robust accuracy for one-step ATLAS at different α, β on CIFAR-10

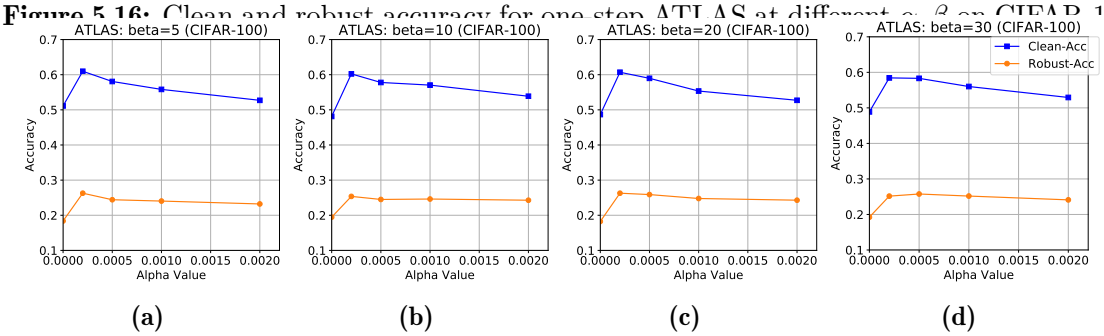


Figure 5.17: Clean and robust accuracy for one-step ATLAS at different α, β on CIFAR-100

5.9 Discussion

We have adopted a different perspective and proposed a new framework for constructing losses that allow more effective use of adversaries. Specifically, based on the framework, we introduce a novel algorithm ATLAS for fast robust training. ATLAS can be used as an initialisation technique for other complicated tasks. For instance, a model trained with ATLAS can be employed as the starting point for layer-wise adversarial training in improving certified robustness. Apart from fast robust training, we believe other losses could be constructed from the framework to accommodate different problems. We leave the explorations of potential applications to various robust training settings to future research.

In the previous two chapters, we have studied robust neural networks in terms of robustness evaluation. In this chapter, we shift our focus to the construction of robust neural networks. Evaluation and construction are two sides of the investigation of robust neural networks. Associated methods are generally developed in different settings respectively. For instance, when dealing with verification methods, we use small to medium sized networks, as these methods are still in the early stage of development. Researchers are currently working on increasing the scalability of verification methods. On the other hand, robust training algorithms are readily applicable to large scale network architectures used in real-life, because most of them only require modifying the final loss function. These algorithms attempt to achieve robust performance by sending desired signals to the underlying weight-updating process through a carefully designed loss function. Although verification methods and robust training are studied independently, it is frequently observed that robustly trained neural networks are much easier to verify than those trained in a standard way. The exact causes for this phenomenon are interesting topics for researchers to explore in the future.

6 Conclusion

Contents

Neural networks with their supreme performance in dealing with a large amount of data have benefited our lives in various aspects, from phone's face unlock feature to virtual assistant tools like Apple Siri. However, neural networks are easily fooled. Their accuracy drops considerably when the inputs are perniciously perturbed in a way imperceptible to humans. Such brittle nature of neural networks has significantly limited their applications, especially in safety-critical areas. For a wider and more reliable use of neural networks, we study the robustness of neural networks. We conclude the thesis by summarising the key findings we made in evaluating and training robust neural networks. In addition, we propose opportunities for future research.

6.1 Key Findings

There are three main challenges in the field of robust neural networks that need to be addressed: defining robustness, evaluating robustness, and improving robustness. In this thesis, we focused on tackling the last two challenges: evaluating and training robust neural networks. Specifically, Chapter 3 and Chapter 4 improved on robustness evaluation in terms of formal verification, while chapter 5 worked on the efficient construction of robust neural networks. We list the main findings of each chapter below.

In Chapter 3, we proposed several heuristics for improving the overall formal verification process based on a general framework. Formal verification formally evaluates whether a property is satisfied on a given domain. It has been suggested by Bunel et al. [10] that many verification methods can be reformulated as instances of a Branch and Bound framework. By directly working with the general Branch and Bound framework, we identify high-level improvements that are readily applicable to methods under the framework. In terms of bounding, we suggested case-dependent strategies in updating the intermediate bounds. When branching is considered, we designed heuristics for branching on both input domains and ReLu activation nodes. For a comprehensive comparison study of our presented heuristics against other existing methods, we introduced several new datasets including the Robust MNIST dataset and its reduced version, the PCAMNIST dataset and the TwinStream dataset, with each focuses on examining a different perspective of methods' performance. We demonstrated the effectiveness of our proposed heuristics and made observations on how to choose bounding and branching strategies to best solve a given problem.

In Chapter 4, we extended the branching strategy by introducing a machine learning framework. In essence, we employed GNNs to imitate the strong branching heuristic. Strong branching heuristic is generally considered the best performing heuristic overall. However, due to its significantly high computational cost, the strong

branching heuristic is rarely used on neural networks. Our learned framework works as a computationally cheap surrogate. To help with the learning process, we used GNNs to exploit the structure of the neural network we want to verify. In addition, we designed a novel schedule for updating embedding vectors. By mimicking the forward and backward passes of neural networks, our updating schedule enjoys both computational and memory efficiency. To best evaluate the performance of our framework, we also introduced a new dataset of verification problems, covering problems at different difficulty levels over convolutional neural networks of different sizes. Through experiments, we showed the outstanding performance of our framework by beating the best existing heuristic with a large gap. Our framework also provides several other benefits. Firstly, with the feature of shared parameters, it has been demonstrated empirically that our framework enjoys horizontal and vertical transferabilities. That means GNN training can be done on datasets of easy verification problems on small neural networks to save the total computational cost. Furthermore, our framework supports online learning for improved performance.

In Chapter 5, we developed an algorithm for training robust neural networks efficiently. We noted that many widely used robust training methods require strong adversaries, which are generated by taking multiple PGD steps. Such process can quickly become prohibitive when model complexity and input dimensions increase. One way to deal with this issue is to make an more effective use of cheap but weak adversaries. Based on this idea, we introduce a loss framework for robust training. The framework consists of a local component and a global component. The local component focuses on maximizing the effectiveness of a given adversary, while the global component combines local components in a regularized way for an overall robust performance. Guided by the framework, we proposed a novel algorithm ATLAS. We tested ATLAS using both weak adversaries generated by one-step PGD and strong adversaries generated by at least 10 steps of PGD. ATLAS makes a more effective use of weak adversaries by outperforming other methods

on all three datasets: MNIST, CIFAR-10 and CIFAR-100. We saw that, with one-step weak adversaries, ATLAS also manages to achieve comparable levels of robust accuracy to state-of-the-art methods employing multi-step strong adversaries on all datasets. Finally, we mention that when strong adversaries are applied, ATLAS gives similar performance to state-of-the-art methods on MNIST and better results on CIFAR-10 and CIFAR-100.

6.2 Suggestions for Future Works

In the following, we present potential future research directions.

6.2.1 Branch and Bound based Formal Verification

Due to the large number of intermediate bounds, the computational cost is a key factor to consider when improving on intermediate bounds quality. Several works have utilized GPUs to reduce the total computational cost in order to attain tighter intermediate bounds. We expect a further improvement could be achieved by conducting a refined treatment of bounds. Depending on the structure of the network and each layer's weight correlation, we believe the quality of intermediate bounds at different layers has varied impacts on the computation of the final bound. Therefore, we can develop a heuristic to determine on which layers tight bounds are essential and on which are not. Then only on those essential layers should the computationally expensive but high-quality bounding methods be applied. Through this customised way, we could best utilize the computational power.

In the Branch and Bound algorithm, the final bound of a sub-problem is used to decide whether the domain should be pruned or not. Many Branch and Bound methods compute the final bound with LP solvers, which have high computational costs and have trouble to scale up when dealing with deep neural network. These limitations hinder the application of these methods to large neural networks. To resolve this issue, we could design a similar strategy that indicates on which sub-problems a high-accuracy final bound is needed. We mention that we only need

high quality bounds on sub-problems that we are likely to prune away.

For SAT properties, early detection of adversaries is the key. Finding an effective upper bound for the output minimum of a sub-problem is thus crucial. At the moment, we compute an upper bound either by running PGD attacks on a sub-problem or by evaluating at its LP solution. We believe more effective strategies can be developed in searching for adversaries on a sub-problem. In addition, identifying sub-problems that have higher chances of containing adversaries and dealing with them first could also lead to an improved performance. In all implemented Branch and Bound methods, we have used the same sorting strategy (pick-up function in Algorithm 1) for sub-problems. That is, the next sub-problem to be picked is the one with the lowest final lower bound. We encourage the development of better sorting rules to facilitate the search process for adversaries.

6.2.2 Learning with GNN

By treating neural networks as graph inputs, we observed that GNNs are effective in capturing network information to assist with network-related decision making. As a result, following the discussion in the previous subsection, we could use GNNs to identify layers, on which high-quality intermediate bounds should be performed. Furthermore, we could apply GNN to accelerate the search of adversaries by learning to generate effective input perturbations or gradient update directions.

6.2.3 Robust Training

In this thesis, we propose a new framework that allows more effective use of weak FGSM adversaries. With an emphasis on efficiency, we made specific choices on the local and global parts to develop the algorithm ATLAS. We believe our framework can be applied to many other robust training settings. By making careful decisions for the local and global parts, we could utilize the framework to construct losses that best accommodate the problem at hand.

References

[1] Michael Akintunde et al. “Reachability analysis for neural agent-environment systems”. In: *Sixteenth International Conference on Principles of Knowledge Representation and Reasoning* (2018).

[2] Uri Alon and Eran Yahav. “On the bottleneck of graph neural networks and its practical implications”. In: *arXiv preprint arXiv:2006.05205* (2020).

[3] Alejandro Marcos Alvarez, Quentin Louveaux, and Louis Wehenkel. “A machine learning-based approximation of strong branching”. In: *INFORMS Journal on Computing* 29.1 (2017), pp. 185–195.

[4] Ross Anderson et al. “Strong mixed-integer programming formulations for trained neural networks”. In: *Proceedings of IPCO 2019* (2019).

[5] Maksym Andriushchenko and Nicolas Flammarion. “Understanding and improving fast adversarial training”. In: *Advances in Neural Information Processing Systems* 33 (2020).

[6] Anish Athalye, Nicholas Carlini, and David Wagner. “Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples”. In: *International conference on machine learning*. PMLR. 2018, pp. 274–283.

[7] Clark Barrett et al. “Splitting on demand in SAT modulo theories”. In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning* (2006).

[8] Osbert Bastani et al. “Measuring neural net robustness with constraints”. In: *Conference on Neural Information Processing Systems* (2016).

[9] Irwan Bello et al. “Neural combinatorial optimization with reinforcement learning”. In: *arXiv preprint arXiv:1611.09940* (2016).

[10] Rudy Bunel et al. “A Unified View of Piecewise Linear Neural Network Verification”. In: *Conference on Neural Information Processing Systems* (2018).

[11] Rudy Bunel et al. “Lagrangian decomposition for neural network verification”. In: *Conference on Uncertainty in Artificial Intelligence*. PMLR. 2020, pp. 370–379.

[12] Nicholas Carlini and David Wagner. “Adversarial examples are not easily detected: Bypassing ten detection methods”. In: *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*. 2017, pp. 3–14.

[13] Nicholas Carlini and David Wagner. “Towards evaluating the robustness of neural networks”. In: *2017 IEEE Symposium on Security and Privacy (SP)* (2017).

[14] Nicholas Carlini and David Wagner. “Audio adversarial examples: Targeted attacks on speech-to-text”. In: *2018 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2018, pp. 1–7.

[15] Ward Cheney and David Kincaid. "Linear algebra: Theory and applications". In: 110 (2009), pp. 450–451.

[16] Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. "Verification of Binarized Neural Networks". In: *arXiv:1710.03107* (2017).

[17] Alesia Chernikova et al. "Are self-driving cars secure? evasion attacks against deep neural networks for steering angle prediction". In: *2019 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2019, pp. 132–137.

[18] Moustapha Cisse et al. "Parseval networks: Improving robustness to adversarial examples". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 854–863.

[19] F. Croce and M. Hein. "Provable robustness against all adversarial l_p-perturbations for $p \geq 1$". In: *International Conference on Learning Representations* (2019).

[20] Francesco Croce and Matthias Hein. "Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks". In: *arXiv preprint arXiv:2003.01690* (2020).

[21] Hanjun Dai et al. "Learning Combinatorial Optimization Algorithms over Graphs". In: *Conference on Neural Information Processing Systems* (2017).

[22] Alessandro De Palma et al. "Scaling the convex barrier with active sets". In: *International Conference on Learning Representations*. 2020.

[23] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. "Convolutional neural networks on graphs with fast localized spectral filtering". In: *Advances in neural information processing systems* 29 (2016), pp. 3844–3852.

[24] Li Deng, Geoffrey Hinton, and Brian Kingsbury. "New types of deep neural network learning for speech recognition and related applications: An overview". In: *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE. 2013, pp. 8599–8603.

[25] G.W. Ding et al. "Max-margin adversarial (mma) training: Direct input space margin maximization through adversarial training". In: *International Conference on Learning Representations* (2020).

[26] Gavin Weiguang Ding et al. "MMA Training: Direct Input Space Margin Maximization through Adversarial Training". In: *International Conference on Learning Representations*. 2019.

[27] Krishnamurthy Dvijotham et al. "A dual approach to scalable verification of deep networks". In: *The Conference on Uncertainty in Artificial Intelligence* (2018).

[28] Ruediger Ehlers. "Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks". In: *Automated Technology for Verification and Analysis* (2017).

[29] Ruediger Ehlers. *Planet*. `https://github.com/progirep/planet`. 2017.

[30] Logan Engstrom et al. "A rotation and a translation suffice: Fooling cnns with simple transformations". In: (2018).

[31] Farzan Farnia, Jesse M Zhang, and David Tse. "Generalizable adversarial training via spectral normalization". In: *arXiv preprint arXiv:1811.07457* (2018).

[32] Angus Galloway et al. “Batch normalization is a cause of adversarial vulnerability”. In: *ICML Workshop* (2019).

[33] Maxime Gasse et al. “Exact Combinatorial Optimization with Graph Convolutional Neural Networks”. In: *arXiv preprint arXiv:1906.01629* (2019).

[34] Justin Gilmer et al. “Neural message passing for quantum chemistry”. In: *International conference on machine learning*. PMLR. 2017, pp. 1263–1272.

[35] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *AISTATS* (2010).

[36] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. “Explaining and harnessing adversarial examples”. In: *International Conference on Learning Representations* (2015).

[37] Sven Gowal et al. “An alternative surrogate loss for pgd-based adversarial testing”. In: *arXiv preprint arXiv:1910.09338* (2019).

[38] Igor Griva, Stephen G Nash, and Ariela Sofer. *Linear and nonlinear optimization*. Vol. 108. Siam, 2009.

[39] William L Hamilton, Rex Ying, and Jure Leskovec. “Inductive representation learning on large graphs”. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 2017, pp. 1025–1035.

[40] Christoph Hansknecht, Imke Joormann, and Sebastian Stiller. “Cuts, primal heuristics, and learning to branch for the time-dependent traveling salesman problem”. In: *arXiv preprint arXiv:1805.01415* (2018).

[41] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[42] Matthias Hein and Maksym Andriushchenko. “Formal Guarantees on the Robustness of a Classifier against Adversarial Manipulation”. In: *Conference on Neural Information Processing Systems* (2017).

[43] Timothy Hickey, Qun Ju, and Maarten H Van Emden. “Interval arithmetic: From principles to implementation”. In: *Journal of the ACM (JACM)* (2001).

[44] J. Hoffman, D.A. Roberts, and S Yaida. “Robust learning with Jacobian Regularization”. In: *https://arxiv.org/abs/1908.02729* (2019).

[45] Xiaowei Huang et al. “Safety verification of deep neural networks”. In: *International Conference on Computer Aided Verification* (2017).

[46] Daniel Jakubovitz and Raja Giryes. “Improving dnn robustness to adversarial attacks using jacobian regularization”. In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 514–529.

[47] Guoqing Jin et al. “Ape-gan: Adversarial perturbation elimination with gan”. In: *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2019, pp. 3842–3846.

[48] H. Kannan, A. Kurakin, and I. Goodfellow. “Adversarial logit pairing”. In: *arXiv preprint arXiv:1803.06373* (2018).

[49] Guy Katz et al. *Reluplex*. `https://github.com/guykatzz/ReluplexCav2017`. 2017.

[50] Guy Katz et al. “Reluplex: An efficient SMT solver for verifying deep neural networks”. In: *CAV* (2017).

[51] Guy Katz et al. “The marabou framework for verification and analysis of deep neural networks”. In: *International Conference on Computer Aided Verification* (2019).

[52] Elias Boutros Khalil et al. “Learning to branch in mixed integer programming”. In: *Thirtieth AAAI Conference on Artificial Intelligence* (2016).

[53] Thomas N Kipf and Max Welling. “Variational graph auto-encoders”. In: *arXiv preprint arXiv:1611.07308* (2016).

[54] Alex Krizhevsky, Geoffrey Hinton, et al. “Learning multiple layers of features from tiny images”. In: (2009).

[55] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. “Adversarial examples in the physical world”. In: *arXiv preprint arXiv:1607.02533* (2016).

[56] Madry Lab. *A brief introduction to adversarial examples*. URL: `http://gradientscience.org/intro_adversarial/`.

[57] Ailsa H. Land and Alison G. Doig. “An automatic method for solving discrete programming problems”. In: *Econometrica 28.3* (1960).

[58] Yann LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[59] June-Goo Lee et al. “Deep learning in medical imaging: general overview”. In: *Korean journal of radiology* 18.4 (2017), pp. 570–584.

[60] Ruoyu Li et al. “Adaptive graph convolutional neural networks”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018.

[61] Fangzhou Liao et al. “Defense against adversarial attacks using high-level representation guided denoiser”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 1778–1787.

[62] Tengfei Ma, Jie Chen, and Cao Xiao. “Constrained generation of semantically valid graphs via regularizing variational autoencoders”. In: *arXiv preprint arXiv:1809.02630* (2018).

[63] A. Madry et al. “Towards deep learning models resistant to adversarial attacks”. In: *International Conference on Learning Representations* (2018).

[64] João P Marques-Silva and Karem A Sakallah. “GRASP: A search algorithm for propositional satisfiability”. In: *IEEE Transactions on Computers* (1999).

[65] Alessio Micheli. “Neural network for graphs: A contextual constructive approach”. In: *IEEE Transactions on Neural Networks* 20.3 (2009), pp. 498–511.

[66] S.M. Moosavi-Dezfooli et al. “Robustness via curvature regularization, and vice versa”. In: *In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2020).

[67] Seyed-Mohsen Moosavi-Dezfooli et al. “Analysis of universal adversarial perturbations”. In: *arXiv preprint arXiv:1705.09554* (2017).

[68] D. R. Morrison et al. “Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning”. In: *Discrete Optimization 19* (2016).

[69] Nina Narodytska et al. “Verifying Properties of Binarized Deep Neural Networks”. In: *arXiv:1709.06662* (2017).

[70] Niall O’Mahony et al. “Deep learning vs. traditional computer vision”. In: *Science and Information Conference*. Springer. 2019, pp. 128–144.

[71] N. Papernot et al. “Distillation as a defense to adversarial perturbations against deep neural networks”. In: *IEEE Symposium on Security and Privacy (SP) (pp. 582-597)* (2016).

[72] C. Qin et al. “Adversarial robustness through local linearization”. In: *In Advances in Neural Information Processing Systems* (2019).

[73] Adnan Siraj Rakin et al. “Defend deep neural networks against adversarial examples via fixed and dynamic quantized activation functions”. In: *arXiv preprint arXiv:1807.06714* (2018).

[74] Muhammad Imran Razzak, Saeeda Naz, and Ahmad Zaib. “Deep learning for medical image processing: Overview, challenges and the future”. In: *Classification in BioApps* (2018), pp. 323–350.

[75] Andrew Slavin Ross and Finale Doshi-Velez. “Improving the adversarial robustness and interpretability of deep neural networks by regularizing their input gradients”. In: *Thirty-second AAAI conference on artificial intelligence*. 2018.

[76] Vicenc Rubies Royo et al. “Fast Neural Network Verification via Shadow Prices”. In: *arXiv:1902.07247* (2019).

[77] Franco Scarselli et al. “The graph neural network model”. In: *IEEE transactions on neural networks* 20.1 (2008), pp. 61–80.

[78] Ludwig Schmidt et al. “Adversarially robust generalization requires more data”. In: *arXiv preprint arXiv:1804.11285* (2018).

[79] Ali Shafahi et al. “Adversarial training for free!” In: *Advances in Neural Information Processing Systems*. 2019, pp. 3353–3364.

[80] David I Shuman et al. “The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains”. In: *IEEE signal processing magazine* 30.3 (2013), pp. 83–98.

[81] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *nature* 529.7587 (2016), pp. 484–489.

[82] Martin Simonovsky and Nikos Komodakis. “Graphvae: Towards generation of small graphs using variational autoencoders”. In: *International conference on artificial neural networks*. Springer. 2018, pp. 412–422.

[83] Gagandeep Singh et al. “Boosting Robustness Certification of Neural Networks”. In: *International Conference on Learning Representations* (2018).

[84] Gagandeep Singh et al. “Fast and effective robustness certification”. In: *Conference on Neural Information Processing Systems* (2018).

[85] Gagandeep Singh et al. “An Abstract Domain for Certifiying Neural Networks”. In: *Proc. ACM Program. Lang. 3* (2019).

[86] Gagandeep Singh et al. “Boosting Robustness Cerfication of Neural Networks”. In: *International Conference on Learning Representations* (2019).

[87] Christian Szegedy et al. “Intriguing properties of neural networks”. In: *arXiv preprint arXiv:1312.6199* (2013).

[88] Vincent Tjeng and Russ Tedrake. “Verifying Neural Networks with Mixed Integer Programming”. In: *International Conference on Learning Representations* (2019).

[89] Vincent Tjeng, Kai Xiao, and Russ Tedrake. “Evaluating robustness of neural networks with mixed integer programming”. In: *International Conference on Learning Representations* (2019).

[90] Petar Veličković et al. “Graph attention networks”. In: *arXiv preprint arXiv:1710.10903* (2017).

[91] Oriol Vinyals et al. “Grandmaster level in StarCraft II using multi-agent reinforcement learning”. In: *Nature* 575.7782 (2019), pp. 350–354.

[92] Athanasios Voulodimos et al. “Deep learning for computer vision: A brief review”. In: *Computational intelligence and neuroscience* 2018 (2018).

[93] Daixin Wang, Peng Cui, and Wenwu Zhu. “Structural deep network embedding”. In: *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 2016, pp. 1225–1234.

[94] Shiqi Wang et al. “Efficient formal safety analysis of neural networks”. In: *Conference on Neural Information Processing Systems* (2018).

[95] Shiqi Wang et al. “Formal security analysis of neural networks using symbolic intervals”. In: *27th {USENIX} Security Symposium ({USENIX} Security 18)* (2018), pp. 1599–1614.

[96] Shiqi Wang et al. “Beta-crown: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification”. In: *arXiv preprint arXiv:2103.06624* (2021).

[97] Yisen Wang et al. “Improving adversarial robustness requires revisiting misclassified examples”. In: *International Conference on Learning Representations*. 2019.

[98] Stefan Webb et al. “A statistical approach to assessing neural network robustness”. In: *International Conference on Learning Representations* (2019).

[99] Tsui-Wei Weng et al. “Towards fast computation of certified robustness for relu networks”. In: *International Conference on Machine Learning* (2018).

[100] Eric Wong and Zico Kolter. “Provable defenses against adversarial examples via the convex outer adversarial polytope”. In: *International Conference on Machine Learning* (2018).

[101] Eric Wong, Leslie Rice, and J. Zico Kolter. “Fast is better than free: Revisiting adversarial training”. In: *International Conference on Learning Representations* (2020).

[102] Dongxian Wu et al. “Skip connections matter: On the transferability of adversarial examples generated with resnets”. In: *International Conference on Learning Representations* (2020).

[103] Zonghan Wu et al. “A comprehensive survey on graph neural networks”. In: *IEEE transactions on neural networks and learning systems* 32.1 (2020), pp. 4–24.

[104] Weiming Xiang, Hoang-Dung Tran, and Taylor T Johnson. "Output Reachable Set Estimation and Verification for Multi-Layer Neural Networks". In: *arXiv:1708.03322* (2017).

[105] Chang Xiao, Peilin Zhong, and Changxi Zheng. "Enhancing adversarial defense by k-winners-take-all". In: *arXiv preprint arXiv:1905.10510* (2019).

[106] Kaidi Xu et al. "Interpreting adversarial examples by activation promotion and suppression". In: *arXiv preprint arXiv:1904.02057* (2019).

[107] Keyulu Xu et al. "How powerful are graph neural networks?" In: *arXiv preprint arXiv:1810.00826* (2018).

[108] Weilin Xu, David Evans, and Yanjun Qi. "Feature squeezing: Detecting adversarial examples in deep neural networks". In: *arXiv preprint arXiv:1704.01155* (2017).

[109] Sijie Yan, Yuanjun Xiong, and Dahua Lin. "Spatial temporal graph convolutional networks for skeleton-based action recognition". In: *Thirty-second AAAI conference on artificial intelligence*. 2018.

[110] Bing Yu, Haoteng Yin, and Zhanxing Zhu. "Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting". In: *arXiv preprint arXiv:1709.04875* (2017).

[111] Sergey Zagoruyko and Nikos Komodakis. "Wide residual networks". In: *arXiv preprint arXiv:1605.07146* (2016).

[112] Radosiaw R Zakrzewski. "Verification of a trained neural network accuracy". In: *IJCNN* (2001).

[113] Runtian Zhai et al. "Adversarially robust generalization just requires more unlabeled data". In: *arXiv preprint arXiv:1906.00555* (2019).

[114] Dinghuai Zhang et al. "You only propagate once: Painless adversarial training using maximal principle". In: *Advances in Neural Information Processing Systems*. 2019.

[115] Hongyang Zhang et al. "Theoretically principled trade-off between robustness and accuracy". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 7472–7482.

[116] Huan Zhang et al. "Efficient neural network robustness certification with general activation functions". In: *arXiv preprint arXiv:1811.00866* (2018).

[117] Jingfeng Zhang et al. "Attacks Which Do Not Kill Training Make Adversarial Learning Stronger". In: *International Conference on Machine Learning*. 2020.

[118] Zixing Zhang et al. "Deep learning for environmentally robust speech recognition: An overview of recent developments". In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 9.5 (2018), pp. 1–28.

[119] Dawei Zhou et al. "Towards Defending against Adversarial Examples via Attack-Invariant Features". In: *arXiv preprint arXiv:2106.05036* (2021).