Towards Robust Machine Learning with Graph Neural Networks

Florian Jaeckle

Wadham College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy

Trinity 2022

Abstract

In order to apply Neural Networks in safety-critical settings, such as healthcare or autonomous driving, we need to be able to analyse their robustness against adversarial attacks. These attacks perturb natural images by adding small, carefully chosen perturbations to them that are imperceptible to the human eye. Trained neural networks with high training and validation accuracy often misclassify a large number of these perturbed images. In this thesis we propose several new methods aimed at analysing the robustness of trained neural networks to adversarial attacks.

In the first part, we improve upon existing methods to generate adversarial examples more efficiently. We note that past work in this field has relied on optimization methods that ignore the inherent structure of the problem and data, or generative methods that rely purely on learning and often fail to generate adversarial examples where they are hard to find. To alleviate these deficiencies, we propose a novel stand-alone attack based on a GNN that takes advantage of the strengths of both approaches. Our GNN computes descent directions to guide an iterative procedure towards adversarial examples.

Our next contribution is inspired by the observation that many state-of-the-art adversarial attacks require many random restarts to generate adversarial examples. Each time we perform a restart we ignore all previous unsuccessful runs. In order to alleviate this deficiency, we propose a method that learns from its mistakes. Specifically, our method uses GNNs as an attention, to greatly reduce the search space for future iterations of the attacks.

For our final contribution, we note that adversarial attacks may fail, even where adversarial examples exist. We thus focus on formal complete neural network verification which returns a sound and complete proof of robustness. Recent years have witnessed the deployment of branch-and-bound (BaB) frameworks for formal verification in deep learning. The main computational bottleneck of BaB is the estimation of lower bounds. Past work in this field has relied on traditional optimization algorithms whose inefficiencies have limited their scope. To alleviate this deficiency, we propose a novel graph neural network (GNN) based approach. Our GNN aims to compute a dual solution of the convex relaxation, thereby providing a valid lower bound, which, if positive, proves robustness.

Towards Robust Machine Learning with Graph Neural Networks

Florian Jaeckle

Wadham College

University of Oxford

Supervised by Prof M. Pawan Kumar

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2022

Abstract

In order to apply Neural Networks in safety-critical settings, such as healthcare or autonomous driving, we need to be able to analyse their robustness against adversarial attacks. These attacks perturb natural images by adding small, carefully chosen perturbations to them that are imperceptible to the human eye. Trained neural networks with high training and validation accuracy often misclassify a large number of these perturbed images. In this thesis we propose several new methods aimed at analysing the robustness of trained neural networks to adversarial attacks.

In the first part, we improve upon existing methods to generate adversarial examples more efficiently. We note that past work in this field has relied on optimization methods that ignore the inherent structure of the problem and data, or generative methods that rely purely on learning and often fail to generate adversarial examples where they are hard to find. To alleviate these deficiencies, we propose a novel stand-alone attack based on a GNN that takes advantage of the strengths of both approaches. Our GNN computes descent directions to guide an iterative procedure towards adversarial examples.

Our next contribution is inspired by the observation that many state-of-the-art adversarial attacks require many random restarts to generate adversarial examples. Each time we perform a restart we ignore all previous unsuccessful runs. In order to alleviate this deficiency, we propose a method that learns from its mistakes. Specifically, our method uses GNNs as an attention, to greatly reduce the search space for future iterations of the attacks.

For our final contribution, we note that adversarial attacks may fail, even where adversarial examples exist. We thus focus on formal complete neural network verification which returns a sound and complete proof of robustness. Recent years have witnessed the deployment of branch-and-bound (BaB) frameworks for formal verification in deep learning. The main computational bottleneck of BaB is the estimation of lower bounds. Past work in this field has relied on traditional optimization algorithms whose inefficiencies have limited their scope. To alleviate this deficiency, we propose a novel graph neural network (GNN) based approach. Our GNN aims to compute a dual solution of the convex relaxation, thereby providing a valid lower bound, which, if positive, proves robustness.

Acknowledgements

First and foremost, I would like to thank my supervisor, Pawan, for his continued support throughout the last four years. Thank you for introducing me to the field of AI robustness, for helping me find suitable topics to research, and for your very detailed feedback that helped me improve my scientific writing. The frequent advice you gave me and the incredible patience you showed me, especially during the early stages of my academic journey, were incredibly helpful.

I would further like to thank my colleagues in the OVAL group: Rudy, Alban, Leo, Prateek, Jodie, Alasdair, Alessandro, Harkirat, and Francisco. I learned a lot from all of you and particularly enjoyed our lunches at the maths department and St Anne's (thank you Alasdair!). A special shout out to Leo who helped me a lot in my first year, teaching me more good coding practices than I had previously learned in my four-year Computer Science undergrad. And thank you Alessandro and Jodie for your work on the verification workshops, it was great to be part of a team during the PhD. Also, a special thanks to Wendy for all your support throughout all 4 years and thank you to the entire AIMS cohort for a fun first year together. I would also like to thank my undergrad tutors at Worcester: Rich, Hongseok, and Endre, who taught me so much during my undergrad and who encouraged me to apply for the PhD.

Next, I would like to thank everyone who I was lucky enough to live with in 14 West Street. The original Crib: Jake, Paddy, and Lulu, as well as Stu and Nick for a great final year despite living through a pandemic. The Doppelkopf sessions in particular were a real highlight during lockdowns and so was @weststreetfood. Also thank you to Jess and Nayana for being great honorary West Street members.

Also, a massive thank you to all the friends I made at St Cross. First to Allie, Olivia, and Alex, you made the second year of my DPhil my favourite year. Thank you, Allie, for going on long runs with me and for running half marathons together (one day we'll run a full marathon together). Thank you also to Sorcha, Jodie, and Meghan for your support and friendship in the last two years of my PhD in particular. And thank you of course to Lulu for taking me along to the St Cross freshers week in the first place.

A big thank you to everyone in the Worcester College Football Teams, especially co-captain Jake, Paddy, Stu and Jeff. Playing MCR football on a rainy Saturday morning was always the highlight of every week. Thank you also to all the medics: Tom, Ben, Katie and Louisa. I really enjoyed spending time with all of you for the extra two years you stayed in Oxford and thank you for letting me join so many medic socials!

A special thank you to Isi for your incredible support during my final year of my PhD! I don't think I would have managed to write-up my thesis without your amazing support. I am very grateful for all the great times we got to spend in Oxford together before moving to London, especially going for walks to Port Meadow, cooking nice dinners together and getting a bonus year at Worcester. I'm inspired by you every day and can't wait to see what the future brings!

Most of all, I would like to thank my family for the massive support you have provided me with for my entire life. Thank you to all of my brothers: Mario, Sandro, and Rico! I really enjoyed that I was able to spend so much more time together again during the lockdowns, just like in the good old days! Thank you to my parents in particular to whom I owe most of my achievements. You are the main reason why I got accepted for my undergrad degree without which I would have never gotten the PhD place. But more importantly, the main reason why I'm able to live such a happy and fulfilling life is because of everything you have done for me. Danke!

Contents

List of Abbreviations

Adam	Adaptive Moment Estimation.
BaB	Branch-and-Bound.
BaBSR	Branch-and-Bound with Smart Branching on ReLUs.
CNN	Convolutional Neural Network.
ConvGNN	Convolutional Graph Neural Network.
CPU	Central Processing Unit.
C&W	Carlini Wagner Attack.
FGSM	Fast Gradient Sign Method.
GAN	Generative Adversarial Network.
GD	Gradient Descent.
GNN	Graph Neural Network.
GRU	Gated Recurrent Unit.
GPU	Graphics Processing Unit.
I-FGSM	Iterative Fast Gradient Sign Method.
lb	Lower Bound.
LP	Linear Programming.
lr	Learning Rate.
MI-FGSM	Iterative Fast Gradient Sign Method with Momentum.
MIP	Mixed-Integer Programming.
MLP	Multilayer Perceptron.
NN	Neural Network.
NP-hardness	Non-deterministic polynomial-time hardness.
PGD	Projected Gradient Descent.
RecGNN	Recurrent Graph Neural Network.
ReLU	Rectified Linear Unit.

RNN	Recurrent Neural Network.
SAT	Satisfiable.
SGD	Stochastic Gradient Descent.
ub	Upper Bound.
UNSAT	Unsatisfiable.

1 Introduction

Contents

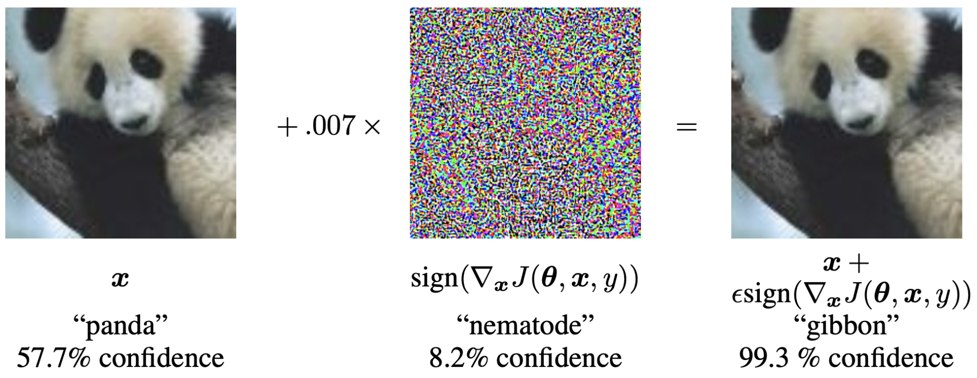

$+ .007 \times$
$=$
$\boldsymbol{x}$
"panda"
57.7% confidence
$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"nematode"
8.2% confidence
$\boldsymbol{x} + \epsilon\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"gibbon"
99.3 % confidence

Figure 1.1: An adversarial example being generated with the Fast Gradient Sign Method [Goodfellow et al., 2015] on GoogleLeNet [Szegedy et al., 2015] on ImageNet. Adding a very small perturbation to the clean image changes the classification of the neural network.

1.1 Motivation

Deep Learning has become a fundamental part of our lives in the last 15 years. The latest resurgence of neural networks began in 2007 driven by work by Ranzato et al. [2006], Bengio et al. [2007], and Hinton [2007]. Since then, neural networks have been applied to areas such as autonomous driving [Farabet et al., 2011, Ciregan et al., 2012, Sermanet et al., 2013], healthcare [Johnson et al., 2018, Ahmad et al., 2018], and drug discovery [Dahl et al., 2014, Segler et al., 2018], often achieving super-human performance. Despite their outstanding performances on various tasks, neural networks are found to be vulnerable to adversarial examples [Szegedy et al., 2013, Goodfellow et al., 2015] — examples that are similar to real inputs but ones which the neural network misclassifies with a high probability. They are obtained by applying small but deliberately chosen perturbations that are often imperceptible to the human eye (Figure 1.1). The brittleness of neural networks can have costly consequences in safety-critical areas such as autonomous vehicles [Bojarski et al., 2016] and personalized medicine [Weiss et al., 2012]. When one requires robustness to adversarial examples, traditional model evaluation approaches, which test the trained model on a hold-out set, do not suffice. Instead, formally verifying adversarial robustness becomes necessary. The required property to verify is that the underlying neural network outputs the same correct prediction for all points within a norm ball whose radius is determined by the maximum perturbation allowed. However, this verification problem is very difficult to solve as it is highly non-convex and scales poorly with the size of the neural network we are verifying.

1.2 Key Challenges

In this thesis we aim to improve existing methods that evaluate the adversarial robustness of neural networks. We introduce a new adversarial attack than can be used to generate adversarial examples more efficiently. Subsequently, we propose an attention mechanism that learns from previous mistakes and can be applied to existing attacks to enhance their performance. Finally, we present a new bounding subroutine which can be used to improve the commonly used Branch-and-Bound algorithm for complete neural network verification.

Generating Adversarial Examples. In order to estimate the adversarial robustness of a trained neural network, we take a set of natural images, and a corresponding set of allowed perturbations (in our case norm balls). For each image and perturbation set, we run a single or multiple adversarial attacks. If the attack manages to return an adversarial example then this can be used to prove that the network is not robust around that particular image. If the adversarial attack fails to find such an example, we believe the network to be robust around the image. However, attacks may be unsuccessful at finding existing adversarial examples, thereby erroneously indicating robustness. Improving the adversarial attacks we use will thus lead to more accurate robustness estimations. However, generating adversarial examples requires solving a highly non-convex problem, a difficult task to solve. Past work in this field has relied on traditional optimization algorithms that ignore the inherent structure of the problem and data, or generative methods that rely purely on learning and often fail to generate adversarial examples where they are hard to find. To alleviate these deficiencies, we propose a novel attack based on a graph neural network (GNN) that takes advantage of the strengths of both approaches.

Furthermore, we note that many state-of-the-art adversarial attacks often require many random restarts to generate adversarial examples. Each time we perform a restart we ignore all previous unsuccessful runs. In order to alleviate this deficiency, we propose a method that learns from its mistakes.

Formal Neural Network Verification. For our final contribution, we note that adversarial attacks may fail, even where adversarial examples exist. We thus focus on formal complete neural network verification which returns a sound and complete proof of robustness. Formal verification is typically carried out using the Branch-and-Bound (BaB) framework. The BaB framework solves a mixed integer programming (MIP) formulation of the verification problem. It works by hierarchically splitting the domain of the MIP into subdomains via a routine known as branching. For each subdomain it computes an upper and a lower bound of the MIP objective. If the upper bound of a subdomain is less than the lower bound of another, the latter subdomain can be pruned thereby reducing the search space for the optimal solution. Branching is usually performed using an efficient heuristic that is either hand-designed or learnt [Hansknecht et al., 2018, Khalil et al., 2016, Lu and Kumar, 2020b]. The upper bound is also efficient to compute as it involves evaluating the objective for any feasible solution. In contrast, the lower bound computation requires solving a large convex relaxation. Typically, the relaxation is solved using either commercial solvers such as Gurobi [Gurobi Optimization, 2020], or traditional optimization algorithms such as subgradient descent or proximal minimization [Bunel et al., 2020a]. However, neither approach scales elegantly with the size of the relaxation, which prevents current formal verification methods from being applied to deep state-of-the-art networks. In other words, lower bound estimation forms the main computational bottleneck for BaB.

1.3 Thesis Outline

We now present the structure of this thesis and summarize the content of each chapter.

Chapter 2 In the second chapter we first introduce neural networks. We describe the main building blocks that make up most neural networks including linear, convolutional, activation, and pooling layers. Next, we outline how to train neural network, defining different objective functions and optimization algorithms, before focusing on regularization. Following this, we proceed by covering the concept of adversarial robustness and outlining different types of adversarial attacks. We then discuss formal neural network verification, mainly focusing on the Branch-and-Bound method which is one of the most common frameworks for verification. Finally, we describe Graph Neural Networks, neural networks that are used to solve graph based problems. We mainly focus on Recurrent Graph Neural Networks and Convolutional Graph Neural Networks as they are the most relevant to our work.

Chapter 3 In this chapter we focus our attention on generating adversarial examples more effectively. Most existing attacks rely on traditional optimization algorithms that ignore the inherent structure of the problem and data, or generative methods that rely purely on learning and often fail to generate adversarial examples where they are hard to find. We propose a new method (AdvGNN) that takes advantage of the strengths of both methods. We learn to compute descent directions to guide an iterative procedure towards adversarial examples. We first analyse the structure of our GNN, before describing the training procedure of the GNN. We compare our method to several state-of-the-art attacks and show that we can generate adversarial examples with small perturbation norms more efficiently and successfully. Moreover, we provide a new challenging dataset specifically designed to allow for a more illustrative comparison of adversarial attacks.

Chapter 4 In the fourth chapter we propose an attention mechanism that can be applied to existing iterative adversarial attacks to boost their performances. Our method uses a GNN to learn from previous unsuccessful attacks and outputs a new, smaller search space for future attacks to focus on. We illustrate the architecture of our GNN, and how the message-passing algorithm is implemented, before detailing how the GNN is trained. We finally analyse how our GNN can be applied to many different attacks in an experimental setting.

Chapter 5 In chapter five, we focus on complete formal verification. Many verification algorithms use Branch-and-Bound to either generate an adversarial example or return a certificate of robustness. In this chapter we focus on the bounding part of BaB. In particular, we propose a new GNN based bounding method, that generates better dual variables for a relaxation of the adversarial problem. We first describe the framework of our GNN, before detailing how we train the parameters of the GNN. Finally, we run a number of experiments to analyse the performance benefits of our new method compared to existing bounding methods.

Chapter 6 In the sixth and final chapter of this thesis we summarize the key aspects of our work and outline the potential for future work.

1.4 Contributions and Publications

We now summarize the main contributions we have made during the time of the PhD and highlight the peer-reviewed publications they have led to.

1.4.1 Generating Adversarial Examples

We have made the following contributions that focus on generating adversarial examples more effectively:

- We proposed AdvGNN, a GNN based standalone adversarial attack that generates adversarial examples more quickly by combining ideas from the general optimization literature with elements from generative methods.

- We showed how we can use a GNN based attention mechanism to boost existing iterative adversarial attacks by learning from previous unsuccessful attacks.

- We created a dataset that is specifically designed to allow for a clear and illustrative comparison of adversarial attacks.

This work is covered in Chapters 3 and 4 and has led to the following publications:

Florian Jaeckle and M. Pawan Kumar. 2021. ‘Generating Adversarial Examples with Graph Neural Networks’. *Conference on Uncertainty in Artificial Intelligence (UAI).*
Also published at *RobustML Workshop at ICLR 2021.*

Florian Jaeckle, Aleksandr Agadzhanov, Jingyue Lu, and M. Pawan Kumar. 2022. ‘Attention for Adversarial Attacks: Learning from your Mistakes’. *The AAAI-22 Workshop on Adversarial Machine Learning and Beyond.* (Won the Best Paper Award).

1.4.2 Verifying Neural Networks

On the topic of formally verifying neural networks we made the following contribution:

- We showed how we can improve the bounding part of the BaB algorithm by generating more accurate lower bounds more quickly. Specifically, we propose using a GNN to output better ascent directions for the dual variables for the relaxation of the adversarial problem.

The above contribution is presented in Chapter 5 and has resulted in the following publications:

Florian Jaeckle and M. Pawan Kumar. 2023. ‘Neural Lower Bounds for Verification’. *IEEE Conference on Secure and Trustworthy Machine Learning.*
Also published at *RobustML Workshop at ICLR 2021.*

1.4.3 Contributions to Other Subjects

We now summarize some of the other work done during the time of the PhD, that is not included in this thesis.

1.4.3.1 Verification of Neural Networks Competition

Along with other members of the OVAL group, I participated in the first and second international verification of neural networks competition (VNN-COMP'20 and VNN-COMP'21). We finished third out of the 12 participating groups. Our contributions to the competitions were twofold:

- We provided a benchmark upon which different methods can be compared. We identified that most verification benchmarks consider image-independent perturbation radii, often resulting in properties being either too easily verifiable or containing many adversarial examples that are easy to find. To alleviate these shortcomings we provided a dataset consisting of three CIFAR10 networks and a list of unique perturbation values associated with each image-network pair that result in challenging verification properties.
- We also submitted our own verification method which is based on an optimized version of the Branch-and-Bound Algorithm. Our method aims to optimize the three different parts of the BaB algorithm by combining some methods proposed in the literature with our own optimized branching strategy and bounding method.

We further contributed to writing the following report, summarizing the results of the competition:

Stanley Bak, Changliu Liu, and Taylor Johnson. The second international verification of neural networks competition (vnn-comp 2021): Summary and results. *arXiv preprint arXiv:2109.00498*, 2021.

1.4.3.2 Compatible Model Updates

We also worked on the problem of compatible model updates for image retrieval systems. In many retrieval systems the original high dimensional data (e.g. images) is mapped to a lower dimensional feature through a learned embedding model. The task of retrieving the most similar member of a gallery set to a given query image is performed through a similarity comparison on features. When the embedding model is updated, it might produce features that are not compatible or comparable with features already in the gallery computed with the old model. Subsequently, all features in the gallery must be re-computed using the new embedding model – a computationally expensive process called backfilling. Recently, compatible representation learning methods have been proposed to avoid backfilling. Despite their relative success, there is an inherent trade-off between new model's stand-alone performance and its compatibility with the old model. To alleviate this deficiency, we proposed FastFill; we summarize its main contributions as below:

- We demonstrate the importance of the training objective to the online backfilling performance and propose using a new training objective for feature alignment that leads to improved retrieval performance at all levels of partial backfilling compared to previous works.
- We further use uncertainty to compute an improved backfilling ordering that promptly elevates retrieval performance, by backfilling less compatible features in the gallery set first.
- We demonstrate how FastFill can update biased embedding models with minimum compute, crucial to efficiently fix large-scale, biased retrieval systems.

As this work is not related to robust machine learning we do not include it in this thesis. This work will appear at ICLR2023:

Florian Jaeckle, Fartash Faghri, Ali Farhadi, Oncel Tuzel, and Hadi Pouransari. Fastfill: Efficient compatible model update. *The Eleventh International Conference on Learning Representations (ICLR), 2023.* `https://openreview.net/forum?id=rnRiiHw8Vy`.

2
Preliminaries

Contents

In this thesis we mainly focus on image classification problems: we are given a set of images and want a program to classify the main object in each of them. We will first describe how to use neural networks for image classification. We will then define adversarial examples which present a significant weakness of neural networks, and summarize a few methods used in the literature to find them. Next, we will cover formal neural network verification which aims to analyse the robustness of networks towards adversarial examples. Finally, we will cover Graph Neural Networks, as most contributions in this work utilize them.

2.1 Neural Networks

Image classification problems require a function $f : \mathbb{R}^d \mapsto \mathbb{R}^m$ that takes as input a d-dimensional image and returns an m-dimensional output vector that indicates the confidence value that the input belongs to each of the m different classes. We can use f to make a prediction by using the arg max function: given an image $\mathbf{x} \in \mathbb{R}^d$, we say that f classifies it as $y \in [1, \dots, m]$ if:

$$y = \arg\max f(\mathbf{x}). \tag{2.1}$$

Colour images $\tilde{\mathbf{x}} \in \mathbb{R}^{3\times h\times w}$ are often saved as three-dimensional matrices. The first dimensional (also called channel) corresponds to the red, green, and blue values for each pixel, the second dimension corresponds to the rows and the third to the columns of the image. We flatten the images to get a one dimensional vector $\mathbf{x} \in \mathbb{R}^d$, where $d = 3 * h * w$. There are many different ways to implement the classifier f. In this work we focus on Convolutional Neural Networks (CNNs). We will now define the different building blocks (layers) that make up most CNNs.

2.1.1 Neural Network Architectures

Linear Layers. The simplest implementation for the classifier f is a linear function. Given a weight matrix $W \in \mathbb{R}^{d\times m}$ and a bias vector $b \in \mathbb{R}^m$, we can define f to be:

$$f(\mathbf{x}) = W\mathbf{x} + b. \tag{2.2}$$

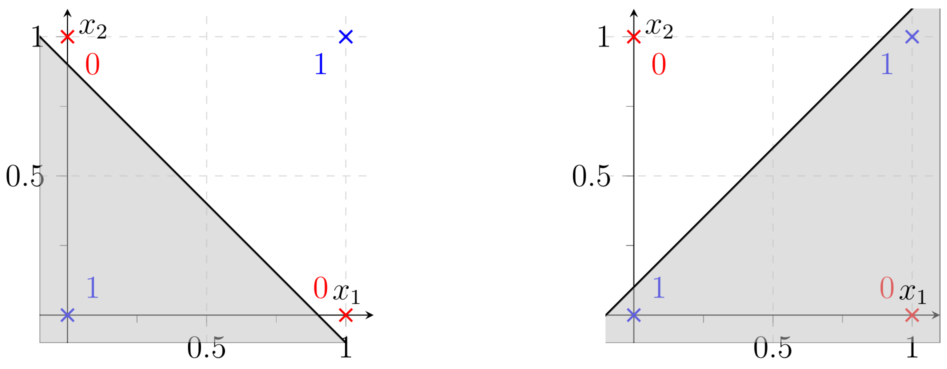


Figure 2.1: Here we show the XOR problem: $y = x_1 \oplus x_2$. It is impossible to separate the two classes using a single line. As linear functions can only separate a two dimensional grid along a line, they are unable to express the XOR function.

A linear function is simple to train and takes up little memory. However, it is also very limited in its expressiveness. It is, for example, unable to learn the simple XOR function (see Figure 2.1). In order to increase the power of our function we add non-linear activation functions.

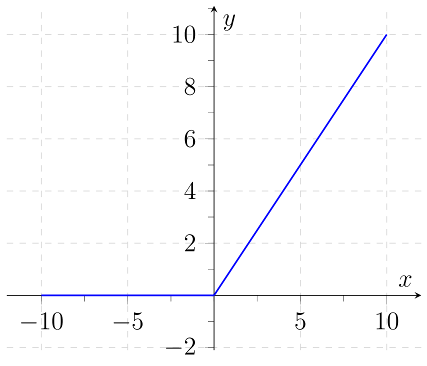
Non-linear Activations. In order to improve our function we cannot simply concatenate two linear layers, as that would simply create another linear function. We thus need non-linear activation functions $\sigma : \mathbb{R} \mapsto \mathbb{R}$ that we can add between two linear layers. The most common activation is the Rectified Linear Unit (ReLU) activation:

$$\sigma(x) = max\{x, 0\}. \tag{2.3}$$

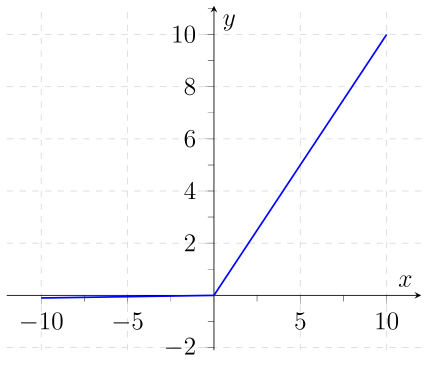
We can create a non-linear function by concatenating a linear layer ($W_1 \in \mathbb{R}^{d \times d_1}, b_1 \in \mathbb{R}^{d_1}$), a ReLU layer, and another linear layer ($W_2 \in \mathbb{R}^{d_1 \times m}, b_2 \in \mathbb{R}^m$), to get:

$$f(x) = W_2\,\sigma(W_1\mathbf{x} + b_1) + b_2. \tag{2.4}$$

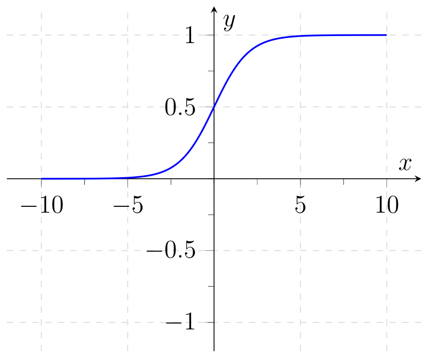
Here the ReLU function is applied element-wise. We can create significantly stronger functions by stacking two linear layers in this way. To give but one example: we can express the XOR function using this architecture [Goodfellow et al., 2016]. The observation that we can create more expressive functions by stacking several simpler layers together forms the foundation of deep learning.



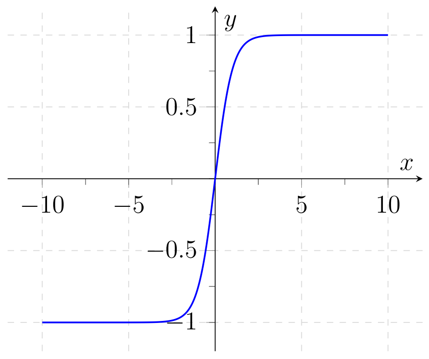
(a) ReLU



(b) PReLU



(c) Sigmoid



(d) Tanh

Figure 2.2: An illustration of four commonly used activation functions.

Other non-linear activations are also widely used in the literature. The main disadvantage of ReLU activations is that its gradient is zero when the input is negative, thus decreasing the ability to learn. Maas et al. [2013] thus proposed using leaky ReLUs, that multiply the input by a small constant (0.01) instead of zero when the input is negative:

$$\sigma(x) = \begin{cases} \mathbf{x} & if\ \mathbf{x} >= 0 \\ 0.01\mathbf{x} & otherwise. \end{cases} \tag{2.5}$$

Similarly He et al. [2015] used parametric ReLUs (PReLUs) that learn the scaling parameter α:

$$\sigma(x) = \begin{cases} \mathbf{x} & if\ \mathbf{x} >= 0 \\ \alpha\mathbf{x} & otherwise. \end{cases} \tag{2.6}$$

There are other activation functions used in machine learning that are not piecewise linear. Some examples include the sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{2.7}$$

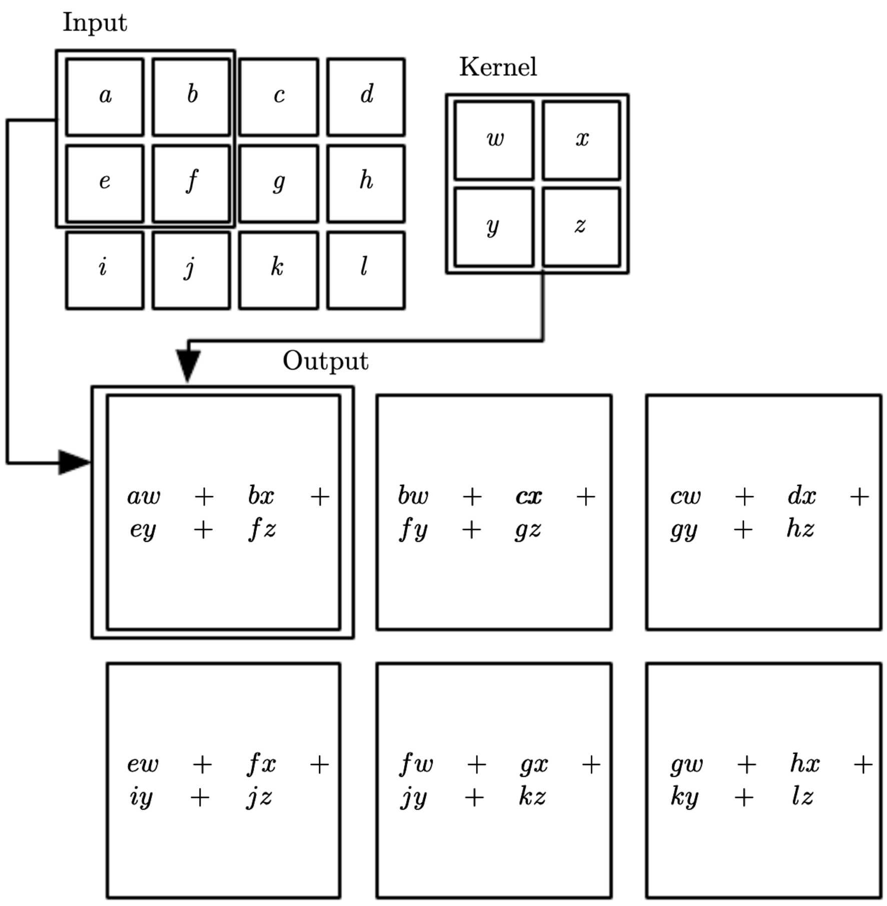


Figure 2.3: Illustration of the convolutional operation by Goodfellow et al. [2016]. The lower-dimensional kernel, sometimes also referred to as the filter, or the feature detector, slides over the larger input and uses matrix multiplications to compute the output, also called a feature map.

or the hyperbolic tangent activation function (tanh):

$$\sigma(x) = \frac{2e^{2x} - 1}{2e^{2x} + 1}. \tag{2.8}$$

Some of these activation functions are illustrated in Figure 2.2.

Convolutional Layers. Convolutional Neural Networks (CNNs) [LeCun et al., 1989] have been one of the most widely used types of functions for computer vision problems. Compared to fully connected linear layers described in the previous paragraph, convolutional layers have a number of inductive biases, which make them suitable for image classification problems. They assume that nearby pixels are more likely to be correlated than pixels far apart from each other. They are also equivariant to translations, meaning that as the input image shifts, so does the output of the convolution. Given a two dimensional input $I \in \mathbb{R}^{n_1 \times m_1}$ and a weight matrix $K \in \mathbb{R}^{n_2 \times m_2}$, the convolutional operation $(*)$ is defined as:

$$S_{i,j} = (K * I)_{i,j} = \sum_{n=1}^{n_2} \sum_{m=1}^{m_2} I_{i+n,j+m} K_{m,n}, \tag{2.9}$$

for any $i \in [1, \ldots, n_1 - n_2 + 1]$ and $j \in [1, \ldots, m_1 - m_2 + 1]$. From now on we refer to the weight matrix as a filter. The size of the filter needs to be no greater than the size of the input (i.e. $n_1 \geq n_2$ and $m_1 \geq m_2$). In practice it is often significantly smaller than the input. We further illustrate the convolutional method in Figure 2.3.

The convolutional operation decreases the length and width of the input by one pixel less than the length and width of the kernel matrix. In order to be able to change the output size independently of the size of the kernel, we add artificial rows to the top and bottom of each image and similarly add columns to the left and right. As we set the value of these additional rows and columns to zero, this is commonly referred to as zero-padding. Using padding not only changes the size of the output but also ensures that we visit the pixels at the boundary of the image more than once.

We sometimes want to downsample an input using convolutions. Instead of moving the filter by one pixel at a time, we can shift it by s number of pixels in either the horizontal or vertical direction. We refer to the parameter s as the stride of the convolution. Using strides, we can decrease the width and height of the output by a factor of s compared to the convolution's input.

One of the main advantages of using convolutional layers, compared to fully connected linear ones, is the number of parameters required. Images often have thousands or even millions of pixels, but we can often learn useful features with kernels that have only thousands of learnable parameters.

Pooling Layers. Once we have passed an input through a convolutional layer and a non-linear activation layer (such as ReLU), we often apply what is known as a pooling operation. It aims to summarize the values of several neighbouring

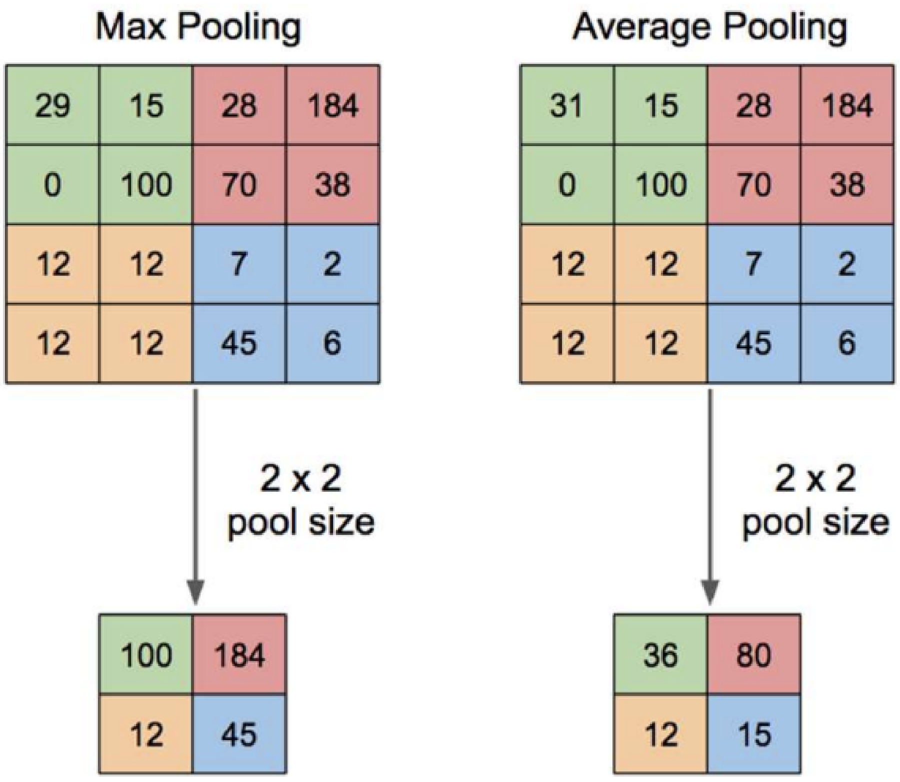


Figure 2.4: An explanation of the max pooling (left) and average pooling (right) operations taken from Yani et al. [2019]. Both methods reduce the size of the input and introduce a small level of translation invariance.

values by a single new value. The most commonly used pooling operations are max pooling [Zhou and Chellappa, 1988], which outputs the maximum value of a square neighbourhood, and average pooling, which computes the average of a square patch. Both types of pooling are illustrated in Figure 2.4. Pooling layers add a small translation invariance. If we translate the image by a small amount, some of the values in the post-pooling layer will remain constant. This is a desirable property for image recognition, as translating an image by a small amount rarely changes the meaning of it. There are further types of layers commonly used in the literature including skip connections [He et al., 2015, 2016] and batch normalization [Ioffe and Szegedy, 2015]; however, we do not focus on either of them in this work.

2.1.2 Training and Evaluating Neural Networks

Having summarized the main building blocks of convolutional neural networks we will now describe how we can train them to perform a desired task. This is included for the sake of completeness and does not form a major part of any of our contributions. We do not focus on improving training algorithms but instead focus on evaluating the robustness of already trained networks.

Datasets. Given a labelled dataset that consists of a set of images each with a corresponding label (describing the main object in each image), we first separate it into three parts: the training, validation and test sets. The test set often consists of about 20% of all images, the validation set is 20% of the remaining images, and the training dataset makes up the rest. We train different models on the training dataset and then evaluate them on the validation dataset. Based on the performance on the validation dataset we might make changes to the model architectures or other design decisions and train them again. Once we are satisfied with our model's performance, we test it on the previously unseen test set. The test set should only be used once to ensure that it provides an accurate estimation of the model's true performance on new data.

Loss Function. In order to train a neural network, we need to define a function that indicates how well the current version is doing. This function, which is commonly referred to as the loss function, evaluates the network's performance on the training dataset. We can train a neural network $f(\cdot;\boldsymbol{\theta})$ with learnable parameters $\boldsymbol{\theta}$ by minimizing the loss function. Given an image, label pair $(\mathbf{x}, y)$, recall that the i-th element of the output of $f(\mathbf{x};\boldsymbol{\theta})$, corresponds to the confidence parameter that image $\mathbf{x}$ belongs to class i. As the values in the output vector aren't necessarily all positive and do not sum to one, we need to change those confidence values into probabilities. The probability of image $\mathbf{x}$ belonging to class i as defined by $f(\cdot;\boldsymbol{\theta})$, can be interpreted as:

$$p_i(f(\mathbf{x};\boldsymbol{\theta})) = \frac{\exp(f(\mathbf{x};\boldsymbol{\theta})_i)}{\sum_j \exp(f(\mathbf{x};\boldsymbol{\theta})_j)}. \tag{2.10}$$

The cross-entropy loss, the most commonly used loss in multi-class classification, is the negative logarithm of the probability returned by f of $\mathbf{x}$ belonging to the true class y:

$$l^{CE}(f(\mathbf{x};\boldsymbol{\theta}), y) = -log\left(p_y(f(\mathbf{x};\boldsymbol{\theta}))\right) = -log\left(\frac{\exp(f(\mathbf{x};\boldsymbol{\theta})_y)}{\sum_j \exp(f(\mathbf{x};\boldsymbol{\theta})_j)}\right). \tag{2.11}$$

Minimizing the cross-entropy loss results in maximizing the output value corresponding to the true class y, while simultaneously minimizing all other elements of the output. The training loss is simply the mean cross-entropy loss over all samples in the training dataset $\mathcal{X}$:

$$J(\mathcal{X};\boldsymbol{\theta}) = \sum_{(x,y)\in\mathcal{X}} l^{CE}(f(\mathbf{x};\boldsymbol{\theta}), y). \tag{2.12}$$

For ease of notation we write $J(\boldsymbol{\theta})$ instead of $J(\mathcal{X};\boldsymbol{\theta})$ from here on, as the training dataset is often fixed.

Optimization Algorithms. Minimizing (2.12) with respect to the parameters $\boldsymbol{\theta}$ is a difficult problem to solve as it is non-convex when f is a multi-layered neural network. Finding the closed form solution is infeasible, which leaves us with iterative optimization algorithms that first randomly initialize the parameters ($\boldsymbol{\theta}^0$) and then aim to arrive at a good solution after several update steps. Gradient Descent (GD) takes small steps towards the gradient of the training loss ($\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$):

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \epsilon \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}). \tag{2.13}$$

Here ϵ is the stepsize parameter that determines how large the update steps are. It is important for the algorithm's success to choose an adequate value of it. If ϵ is too small, then the GD takes a very long time to converge to a good solution, whereas if it is too large, then it might oscillate around the solution or even diverge. If the training dataset is large, as is often the case in computer vision problems, computing $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ is very expensive as it requires computing the gradient at every single datapoint in the training set. We can estimate the true gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ by sampling only a few images ($\mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(b)} \sim \mathcal{X}$) from the training set uniformly at random (without replacement) and computing the gradient over those b points only. This set of points is often called a mini-batch. The method is called Stochastic Gradient Descent (SGD) and its update step is similar to GD:

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^{t} - \epsilon \mathbf{g}, \tag{2.14}$$

where $\mathbf{g}$ is the gradient over the minibatch:

$$\mathbf{g} = \frac{1}{b} \sum_{j=1}^{b} \nabla_{\boldsymbol{\theta}} l^{CE}(f(\mathbf{x}^{(j)}; \boldsymbol{\theta}), y^{(j)}). \tag{2.15}$$

This approach is motivated by the observation that the expectation of the mean gradient of these b points is the same as the gradient over the entire training dataset ($\mathbb{E}[\mathbf{g}] = \nabla J(\boldsymbol{\theta})$).

Sometimes the convergence rate of SGD can be slow. Using momentum [Polyak, 1964] can greatly speed up the optimization procedure [Qian, 1999]. It is particularly useful when we face either high curvature or noisy gradients [Sutton, 1986]. High curvature leads to the gradient changing significantly over small regions of search space. The momentum method computes a momentum term ($\mathbf{v}$) which is a weighted average of past gradients. We use this momentum term to update the parameters instead of the gradient itself:

$$\mathbf{v}^{t+1} = \gamma \mathbf{v}^{t} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{2.16}$$

$$\boldsymbol{\theta} = \boldsymbol{\theta} + \mathbf{v}_{t+1}. \tag{2.17}$$

Other optimizers used in the literature include Adagrad [Duchi et al., 2011], Adadelta [Zeiler, 2012], RMSProp [Hinton et al., 2012], and Adam [Kingma and Ba, 2015].

Regularization. When training a neural network we focus on maximizing performance on the training set. Our ultimate goal, however, is to create a model that performs well on previously unseen data, for example the test set. A problem that often occurs in machine learning is networks performing very well on the training dataset but not generalizing well to unseen data; this is commonly referred to as overfitting. There are many different approaches that can be used to decrease the level of overfitting. We can, for example, increase the size of the training dataset, either by collecting more data or where this is infeasible by using data augmentation. This involves artificially creating new images using operations such as rotating, translating, or flipping. Changing the neural network architecture can also improve overfitting. Generally, larger networks are more expressive and able to perform well on the training set but are also more prone to overfitting. Thus decreasing the size of our model can reduce overfitting. However, changing the network architecture is often impractical as it can require retraining the model from scratch and involves human decision making. This has motivated the use of regularization, which aims to improve generalization performance without increasing the error on the training dataset. Regularization adds an assumption that certain parameter assignments are better than others. The most commonly used form of regularization is weight decay, which encourages smaller parameter values. We add a term ($\Omega(\boldsymbol{\theta})$) to our loss function that increases when the norm of our learnable parameters ($\boldsymbol{\theta}$) increases:

$$\widetilde{J}(\mathcal{X};\boldsymbol{\theta}) = J(\mathcal{X};\boldsymbol{\theta}) + \lambda\Omega(\boldsymbol{\theta}). \tag{2.18}$$

The most popular implementations for Ω is L^2 regularization:

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2}\,\|\boldsymbol{\theta}\|_2^2 = \frac{1}{2}\boldsymbol{\theta}^\top\boldsymbol{\theta}. \tag{2.19}$$

Similarly, we can also use the L^1 norm for regularization:

$$\Omega(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|_1 = \sum_i |\boldsymbol{\theta}_i|. \tag{2.20}$$

Both types of regularization encourage parameters with small magnitudes, but they differ in their exact effect on $\boldsymbol{\theta}$. The gradient of the regularization term ($\nabla_{\boldsymbol{\theta}}\Omega(\boldsymbol{\theta})$) tends to zero as $\boldsymbol{\theta}$ tends to zero when using L^2 regularization; however this is not the case for L^1 regularization ($\nabla_{\boldsymbol{\theta}}\Omega(\boldsymbol{\theta}) = \operatorname{sgn}(\boldsymbol{\theta})$). This is why L^1 regularization encourages sparsity, where many values of the learnable parameters become zero. L^2 weight decay is more popular, mainly because its gradient is smooth, which makes optimization easier. Other methods to reduce overfitting that we do not focus on in this work include dropout [Srivastava et al., 2014], early stopping [Caruana et al., 2000], and feature selection [Guyon and Elisseeff, 2003].

2.2 Adversarial Robustness

State-of-the-art neural networks have achieved remarkable performance on many computer vision tasks in recent years, outperforming humans on a number of them [Buetti-Dinh et al., 2019, Murphy et al., 2022]. However, Szegedy et al. [2013] realised that seemingly high performing networks are vulnerable to so-called adversarial attacks. These attacks add a small, carefully chosen perturbation to a natural image, aiming for the neural network to misclassify it. These perturbations are small enough so that humans cannot tell the difference between the clean and modified images, yet the neural network makes very different predictions for both. This vulnerability is a major problem when neural networks are being applied to safety-critical situations.

There are many different methods called adversarial attacks that try to generate these adversarial perturbations. White box attacks have access to the neural network architecture and weights, whereas black box attacks do not. Universal attacks aim to generate a single perturbation that can be applied to many different images, while image dependent attacks compute a different perturbation for each image it is trying to attack. Untargeted attacks return perturbations that make the network misclassify the image in any way, compared to targeted attacks that aim for the neural network to misclassify the perturbed image as a particular incorrect class.

Formally, we can define the adversarial robustness problem as follows. Given a neural network $f(\cdot;\boldsymbol{\theta})$, an image $\mathbf{x}$ with true label y, and an allowed perturbation set $\mathcal{B}$, does there exist a perturbed image $\mathbf{x}' \in \mathcal{B}$, such that

$$\arg\max_i f(\mathbf{x}';\boldsymbol{\theta}) \neq y. \tag{2.21}$$

If such an $\mathbf{x}'$ exists, we call it an adversarial example. For ease of notation we omit $\boldsymbol{\theta}$ from here onwards as the network parameters are often fixed for the adversarial robustness problem. Different type of perturbations have been studied in the literature. The most common ones are norm perturbations, where $\mathcal{B}$, is a $1-$, $2-$, or $\infty-$norm ball around the true image $\mathbf{x}$:

$$\mathcal{B}_p(\mathbf{x},\epsilon) = \{\mathbf{x}' \mid \|\mathbf{x}-\mathbf{x}'\|_p \leq \epsilon\} \qquad p \in [1,2,\infty]. \tag{2.22}$$

Other perturbations that have been studied in the literature include rotations, translations, shearing, and scaling [Fawzi and Frossard, 2015, Pei et al., 2017, Xiao et al., 2018b, Kanbak et al., 2018, Engstrom et al., 2019].

2.3 Adversarial Attacks

We focus on white box attacks in this thesis as they are the strongest forms of attacks. White box attacks can be separated into three main categories: the most common attacks aim to maximize the level of confidence with which the network misclassifies an image given an allowed perturbation norm. Another set of attacks tries to find an adversarial example with the smallest possible perturbation. The final set of attacks uses generative methods to output possible adversarial examples.

2.3.1 Attacks Maximizing the Adversarial Loss

The first category of adversarial attacks aims to maximize the adversarial loss L. The formulation of the adversarial loss depends on whether we are running targeted or untargeted attacks. For untargeted attacks the adversarial loss is simply the training objective of the neural network (which we are now maximizing rather than minimizing). For targeted attacks the adversarial loss is defined as

$$L(\mathbf{x}', y, \hat{y}) = f(\mathbf{x}')_{\hat{y}} - f(\mathbf{x}')_{y}, \tag{2.23}$$

given a natural image $\mathbf{x}$ with true label y, an incorrect target class $\hat{y} \neq y$, and a domain $\mathcal{B}(\mathbf{x}, \epsilon)$. We call L the adversarial loss and $\mathbf{x}'$ an adversarial example if $L(\mathbf{x}', y, \hat{y}) > 0$. Many algorithms which attempt to solve (2.23) have been proposed in the literature. The first was the Fast Gradient Sign Method (FGSM) [Goodfellow et al., 2015] that takes a single step towards the sign of the adversarial gradient $(\nabla_{\mathbf{x}} L(\mathbf{x}', y, \hat{y}))$:

$$\mathbf{x}' = \mathbf{x} + \epsilon \operatorname{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}', y, \hat{y})). \tag{2.24}$$

Madry et al. [2018] proposed what is commonly referred to as the PGD attack (also called I-FGSM), a method that applies the FGSM step iteratively. This corresponds to running Projected Gradient Descent on the negative adversarial loss. The $(t+1)^{th}$ update step is defined as:

$$\mathbf{x}^{t+1} = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)}\left(\mathbf{x}^t + \alpha\ \mathrm{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}', y, \hat{y}))\right). \tag{2.25}$$

Dong et al. [2018] proposed boosting the performance of the I-FGSM with momentum. They iteratively apply the following two update steps:

$$\mathbf{g}^{t+1} \leftarrow \mu \cdot \mathbf{g}^t + \frac{\nabla_x L(\mathbf{x}^t, y, \hat{y})}{\|\nabla_x L(\mathbf{x}^t, y, \hat{y})\|_1} \tag{2.26}$$

$$\mathbf{x}^{t+1} = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)}\left(\mathbf{x}^t + \alpha \cdot sgn(\mathbf{g}^{t+1})\right). \tag{2.27}$$

Here $\mathbf{g}$ is the momentum term. As this problem is non-convex, the solution of any algorithm depends heavily on the initialization. To improve the success rate of the attack, we can perform random restarts where we initialize $\mathbf{x}^0 \in \mathcal{B}(\mathbf{x}, \epsilon)$ randomly each time.

2.3.2 Attacks Minimizing the Perturbation Norm

A similar line of research aims to find an adversarial example with the smallest possible perturbation. Szegedy et al. [2013] proposed using limited-memory box constrained optimization (based on the Broyden–Fletcher–Goldfarb–Shanno algorithm (BFGS)) to find the smallest perturbation required to change the prediction of the network. Carlini and Wagner [2017] approximate the objective function using a simpler linear function that can be solved using standard optimization algorithms. Moosavi-Dezfooli et al. [2016] introduced the Deepfool attack that exploits the assumption that the network behaves linearly near the original input.

2.3.3 Generative Attacks

A third class of attacks includes generative methods. Baluja and Fischer [2017] train a second neural network (ATN) that, given an input, aims to output an adversarial example. Poursaeed et al. [2018] trained a generative method that also learns to generate image-specific perturbations. Xiao et al. [2018a] propose the use of a Generative Adversarial Network (GAN) that learns to approximate the distribution of the original images.

2.3.4 Black Box Attacks

In the black box setting, the attacker has very limited knowledge of the model it is attacking. In most scenarios it can query the model on inputs of its choice and observe the output. Bhambri et al. [2019] separate black box attacks into four main categories: gradient estimation, transferability, local search, and combinatorics. Gradient estimation methods aim to predict the gradient of the target model [Chen et al., 2017, Ilyas et al., 2018]. Transferability attacks train a surrogate model to attack [Papernot et al., 2017, Dong et al., 2019]. Local search based attacks [Narodytska and Kasiviswanathan, 2016, Brendel et al., 2017] use the optimization method initially proposed by Feige et al. [2011]. And the combinatorics approach by Moon et al. [2019] treats the problem as a discrete surrogate problem and uses combinatorial optimization to solve it.

2.4 Formal Neural Network Verification

Formal verification of neural networks refers to the problem of proving or disproving a property over a bounded input domain. Properties are functions of neural network outputs. When a property can be expressed as a boolean expression over linear forms, we can modify the neural network in a suitable way so that the property can be simplified to check the sign of the neural network output [Bunel et al., 2018a]. Note that all the properties studied in previous works satisfy this form, thereby allowing us to use the aforementioned simplification. Mathematically, given the modified neural network f, a bounded convex input domain $\mathcal{C}$, formal verification examines the truthfulness of the following statement:

$$\forall \mathbf{x} \in \mathcal{C}, \qquad f(\mathbf{x}) \geq 0. \tag{2.28}$$

If the above statement is true, the property holds. Otherwise, the property does not hold. Some state-of-the-art methods to solve this problem that have performed well in a recent neural network verification benchmark [Bak et al., 2021] include α-β-CROWN [Zhang et al., 2018], VeriNet [Henriksen and Lomuscio, 2019], ERAN [Singh et al., 2018], and Marabou [Katz et al., 2019].

2.4.1 Branch-and-Bound

Verification tasks are often treated as a global optimization problem. We want to find the minimum of a function $f(\mathbf{x})$ over a domain $\mathcal{C}$ in order to compare it with the threshold 0. Specifically, we consider an L layer feed-forward neural network, $f : \mathbb{R}^d \to \mathbb{R}$, with non-linear activations σ such that for any $\mathbf{x}_0 \in \mathcal{C} \subseteq \mathbb{R}^d$, $f(\mathbf{x}_0) = \hat{\mathbf{x}}_L \in \mathbb{R}$, where

$$\hat{\mathbf{x}}_{i+1} = W^{i+1}\mathbf{x}_i + \mathbf{b}^{i+1}, \qquad \text{for } i = 0, \dots, L-1, \tag{2.29a}$$

$$\mathbf{x}_i = \sigma(\hat{\mathbf{x}}_i), \qquad \text{for } i = 1, \dots, L-1. \tag{2.29b}$$

The terms W^i and $\mathbf{b}^i$ refer to the weights and biases of the i-th layer of the neural network f. Domain $\mathcal{C}$ can be an ℓ_p norm ball with radius ϵ. In our case, we use the ReLU activation defined as $\sigma(x) = \max(x, 0)$ as it is widely used in machine learning in general [Krizhevsky et al., 2012, Maas et al., 2013] and in neural network verification in particular [Bunel et al., 2018b, Dvijotham et al., 2018a, Ehlers, 2017b]. Finding the minimum of f is a challenging task, as the optimization problem is generally NP hard [Katz et al., 2017a]. To deal with the inherent difficulty of the optimization problem itself, the Branch-and-Bound (BaB) framework [Bunel et al., 2018a] is generally adopted. In detail, BaB based methods divide the feasible domain defined by equations (2.29a) and (2.29b) for all $\mathbf{x} \in \mathcal{C}$ into sub-domains, each of which defines a new sub-problem (branching). They then compute a relaxed lower bound of the minimum on each sub-problem (bounding). The minimum of the lower bounds of all the generated sub-domains constitutes a valid global lower bound of the global minimum over $\mathcal{C}$. As a recursive process, BaB keeps partitioning the sub-domains to tighten the global lower bound. The process terminates when the computed global lower bound is above zero (property is true) or when an input with a negative output is found (property is false). A detailed description of the BaB method is provided in Algorithm (1). In what follows, we provide a brief description of the two components, bounding methods and branching strategies.

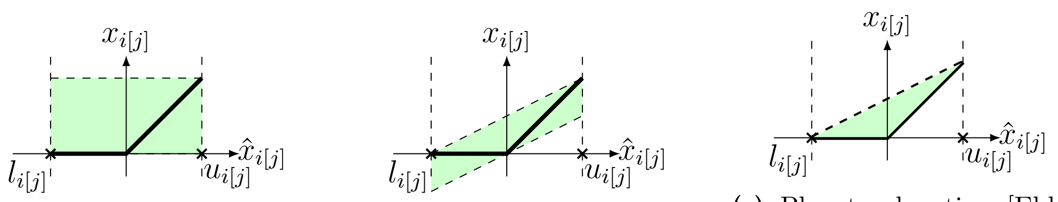
Algorithm 1 Branch-and-Bound

```
Input: net, problem
 1: global_lb ← compute_LB(net, problem) {global lower bound}
 2: global_ub ← compute_UB(net, problem) {global upper bound}
 3: probs ← [(global_lb, problem)] {set of all current domains}
 4: while probs is not empty do
 5:     (_ , prob) ← pick_out(probs) {the pick_out function picks an ambiguous ReLU
        to split on}
 6:     [subprob_1, subprob_2] ← split(prob)
 7:     for $i = 1, 2$ do
 8:         sub_lb ← compute_LB(net, subprob_i)
 9:         sub_ub ← compute_UB(net, subprob_i)
10:         if sub_ub < 0 then
11:             return SAT {we've found an adversarial example}
12:         end if
13:         if sub_lb < 0 then
14:             probs.append((sub_lb, subprob_i))
15:         end if {if sub_lb >= 0 then the subdomain gets pruned away}
16:     end for
17:     global_lb ← min{lb | (lb, prob) ∈ probs} {If probs is non-empty then
        global_lb is negative}
18: end while
19: return UNSAT {all subproblems have a positive lower bound, therefore global_lb
    is positive}
```

2.4.2 Bounding

The bottleneck of most BaB algorithms is the estimation of a lower bound on the output of the neural network we are trying to verify on a given subdomain. As the neural network is highly non-convex and thus hard to optimize over we use convex relaxations. Different linear-sized relaxations have been proposed in the literature such as naïve relaxation, linear bounds relaxation, and Planet relaxation. In our work we use the Planet relaxation as it is the tightest relaxation as can be seen in Figure 2.5. Specifically, we focus on the decomposition based approach of Bunel et al. [2020a] which is described in more detail below.

Planet Relaxation. We denote the output of the i-th layer before the application of the ReLU as $\hat{\mathbf{x}}_i$ and the output of applying the ReLU to $\hat{\mathbf{x}}_i$ as $\mathbf{x}_i$. Given the lower bounds $\mathbf{l}_i$ and upper bounds $\mathbf{u}_i$ of the values of $\hat{\mathbf{x}}_i$, we relax the ReLU activations



(a) Naïve relaxation **(b)** Linear bounds relaxation **(c)** Planet relaxation [Ehlers, 2017a]

Figure 2.5: 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. Naïve relaxation (a) is the loosest relaxation. Linear bounds relaxation (b) is tighter and is introduced in Weng et al. [2018a]. Finally, Planet relaxation (c) is the tightest linear relaxation among the three considered [Ehlers, 2017a]. Among them, (a) and (b) have closed form solutions which allow fast computations while (c) requires an iterative procedure to obtain an optimal solution. Figure taken from Lu and Kumar [2020a]

$\mathbf{x}_i = \sigma(\hat{\mathbf{x}}_i)$ to its convex hull $cvx\_hull_\sigma(\hat{\mathbf{z}}_i, \mathbf{x}_i, \mathbf{l}_i, \mathbf{u}_i)$, defined as follows:

$$cvx\_hull_\sigma(\hat{\mathbf{z}}_i, \mathbf{x}_i, \mathbf{l}_i, \mathbf{u}_i) \equiv \begin{cases} x_{i[j]} \geq 0 \quad x_{i[j]} \geq \hat{\mathbf{z}}_{i[j]} & \\ x_{i[j]} \leq \frac{u_{i[j]}(\hat{\mathbf{z}}_{i[j]} - l_{i[j]})}{u_{i[j]} - l_{i[j]}} & \text{if } l_{i[j]} < 0 \text{ and } u_{i[j]} > 0 \\ x_{i[j]} = 0 & \text{if } u_{i[j]} \leq 0 \\ x_{i[j]} = \hat{\mathbf{z}}_{i[j]} & \text{if } l_{i[j]} \geq 0. \end{cases} , \forall j \tag{2.30}$$

Here, $x_{i[j]}$ denotes the j-th element of $\mathbf{x}_i$. Note that the computation of the convex hull requires the knowledge of the lower and upper bounds (i.e. $\mathbf{l}_i$ and $\mathbf{u}_i$) for each intermediate node. The bounds do not have to be optimal. However, the tighter the bounds are, the tighter the relaxation will be as well. There are different ways of computing said bounds that have been proposed in the literature [Gowal et al., 2018, Raghunathan et al., 2018, Wong and Kolter, 2018]. For the sake of clarity, we introduce the following notations for the constraints corresponding to the input and the i-th layer respectively:

$$\mathcal{P}_0(\mathbf{x}_0, \hat{\mathbf{x}}_1) \equiv \begin{cases} \mathbf{x}_0 \in C \\ \hat{\mathbf{x}}_1 = W_1\mathbf{x}_0 + \mathbf{b}_1 \end{cases} \qquad \mathcal{P}_i(\hat{\mathbf{x}}_i, \hat{\mathbf{x}}_{i+1}) \equiv \begin{cases} \exists \mathbf{x}_i \text{ s.t.} \\ \mathbf{l}_i \leq \hat{\mathbf{x}}_i \leq \mathbf{u}_i \\ cvx\_hull_\sigma(\hat{\mathbf{z}}_i, \mathbf{x}_i, \mathbf{l}_i, \mathbf{u}_i) \\ \hat{\mathbf{x}}_{i+1} = W_{i+1}\mathbf{x}_i + \mathbf{b}_{i+1}. \end{cases} \tag{2.31}$$

Using the above notation, the Planet relaxation for computing the lower bound can be written as:

$$\min_{\mathbf{x}, \hat{\mathbf{x}}} \hat{\mathbf{z}}_L \text{ s.t. } \mathcal{P}_0(\mathbf{x}_0, \hat{\mathbf{x}}_1); \mathcal{P}_i(\hat{\mathbf{x}}_i, \hat{\mathbf{x}}_{i+1}) \text{ for } i \in [1, \ldots, L-1]. \tag{2.32}$$

Lagrangian Decomposition. We often merely need approximations of the bounds rather than the precise values of them. We can therefore make use of the primal-dual formulation of the problem as every feasible solution to the dual problem provides a valid lower bound for the primal problem. Following the work of Bunel et al. [2020a] we will use the Lagrangian decomposition [Guignard and Kim, 1987]. To this end, we first create two copies $\hat{\mathbf{x}}_{A,i}, \hat{\mathbf{x}}_{B,i}$ of each variable $\hat{\mathbf{x}}_i$:

$$\begin{aligned} \min_{\mathbf{x},\hat{\mathbf{x}}} \hat{\mathbf{x}}_{A,L} \text{ s.t. } & \mathcal{P}_0(\mathbf{x}_0, \hat{\mathbf{z}}_{A,1}); \mathcal{P}_i(\hat{\mathbf{x}}_{B,i}, \hat{\mathbf{x}}_{A,i+1}) && \text{for } i \in [1, \ldots, L-1], \\ & \hat{\mathbf{x}}_{A,i} = \hat{\mathbf{x}}_{B,i} && \text{for } i \in [1, \ldots, L-1]. \end{aligned} \tag{2.33}$$

Next we obtain the dual by introducing Lagrange multipliers $\boldsymbol{\rho}$ corresponding to the equality constraints of the two copies of each variable:

$$\begin{aligned} q(\boldsymbol{\rho}) = \min_{\mathbf{x},\hat{\mathbf{x}}} \quad & \hat{\mathbf{x}}_{A,n} + \sum_{i=1,\ldots,n-1} \boldsymbol{\rho}_i^\top (\hat{\mathbf{x}}_{B,i} - \hat{\mathbf{x}}_{A,i}) \\ \text{s.t.} \quad & \mathcal{P}_0(\mathbf{x}_0, \hat{\mathbf{z}}_{A,1});\ \mathcal{P}_i(\hat{\mathbf{x}}_{B,i}, \hat{\mathbf{x}}_{A,i+1}) \ \text{ for } i \in [1, \ldots, L-1]. \end{aligned} \tag{2.34}$$

Problem (2.34) is unconstrained with respect to $\boldsymbol{\rho}$. In other words, any possible $\boldsymbol{\rho}$ is a feasible solution and thus by duality provides a lower bound for the primal problem (2.33). We therefore aim to maximize $q(\boldsymbol{\rho})$ to get the tightest possible lower bound.

2.4.3 Branching

We now discuss branching strategies which form the second important part of the Branch-and-Bound algorithm. Especially for large scale networks f, each branching step has a large number of putative choices. In these cases, the effectiveness of a branching strategy directly determines the possibility of verifying properties over these networks within a given time limit. On neural networks, two types of branching decisions are used: input domain split and hidden activation unit split.

Assume we want to split a parent domain $\mathcal{D}$. Input domain split selects an input dimension and then makes a cut on the selected dimension while the rest of the dimensions remain the same. The common choice is to cut the selected dimension in half and the dimension to cut is decided by the branching strategy used. Available input domain split strategies are Bunel et al. [2018a] and Royo et al. [2019]. The strategy of Royo et al. [2019] is based on a sensitivity test of the LP on $\mathcal{D}$. Whereas Bunel et al. [2018a] use the formula provided in Wong and Kolter [2018] to estimate final output bounds for subdomains after splitting on each input dimension and selects the dimension that results in the highest output lower bound estimates.

In our setting, we refer to a ReLU activation unit $x_{i[j]} = \max(\hat{x}_{i[j]}, 0)$ as ambiguous over $\mathcal{D}$ if the upper bound $u_{i[j]}$ and the lower bound $l_{i[j]}$ for $\hat{x}_{i[j]}$ have different signs. The activation unit split chooses among ambiguous activation units and then divides the original problem into cases of different activation phases of the chosen activation unit. If a branching decision is made on $x_{i[j]}$, we divide the ambiguous case into two determinable cases: $\{x_{i[j]} = 0, l_{i[j]} \leq \hat{x}_{i[j]} \leq 0\}$ and $\{x_{i[j]} = \hat{x}_{i[j]}, 0 \leq \hat{x}_{i[j]} \leq u_{i[j]}\}$. After the split, the originally introduced convex relaxation is removed, since the above sets are themselves convex. We expect large improvements on the output lower bounds of the newly generated sub-problems if a good branching decision is made. Apart from random selection, employed in Ehlers [2017a] and Katz et al. [2017a], available ReLU split heuristics are Wang et al. [2018] and Bunel et al. [2020b]. Wang et al. [2018] compute scores based on gradient information to prioritise ambiguous ReLU nodes. Bunel et al. [2020b] use scores to rank ReLU nodes that are computed with a formula developed on the estimation equations in Wong and Kolter [2018]. We note that for both branching strategies, after the split, intermediate bounds are updated accordingly on each new sub-problem. For neural network verification problems, either domain split or ReLU split can be used at each branching step. When compared with each other, ReLU split is a more effective choice for large scale networks, as shown in Bunel et al. [2020b].

2.5 Graph Neural Networks

Problems in many different areas of research have an underlying structure that resembles a graph. Examples include proteomics [Baldi and Pollastri, 2003], genetics and genomics [Friedman, 2004], scene description [Krahmer et al., 2003], software engineering [Collberg et al., 2003], and social network analysis [Carrington et al., 2005]. We first describe the mathematical formulation of graphs, before summarizing methods that are used to solve graph based problems.

Many Graphs can be presented as a tuple $G = (V, E)$, where V is a set of nodes (sometimes also referred to as vertices), and E is a set of edges. An edge $e_{i,j} = (v_i, v_j) \in E$ connects two nodes $v_i, v_j \in E$. Edges always point from one node (v_i) to another node (v_j). A graph is said to be undirected if for any edge $e_{i,j} = (v_i, v_j) \in E$, the reverse edge $e_{j,i} = (v_j, v_i)$ is also in E, otherwise a graph is directed. The list of vertices V can be represented as an ordered list $(v_1, \ldots, v_n)$. The set of edges E can be written using an $n \times n$ adjacency matrix $\mathbf{A}$, where $\mathbf{A}_{i,j} = 1$ if $e_{i,j} \in E$ and $\mathbf{A}_{i,j} = 0$ otherwise. A graph is undirected if and only if $\mathbf{A}$ is symmetric. In some cases each node has a set of attributes which are represented in the form of a p-dimensional vector $\mathbf{z} \in \mathbb{R}^p$. All node attribute vectors are put together in a single node attribute matrix $\mathbf{X} \in \mathbb{R}^{n \times p}$. Similarly we sometimes have edge attributes $\mathbf{X}^e \in \mathbb{R}^{m \times q}$, where $\mathbf{z}^e_{i,j} = \mathbf{X}^e_{ij} \in \mathbb{R}^q$ represents the attribute of edge $e_{i,j}$.

Some of the most well known computer science algorithms are based on graphs such as Dijkstra's shortest path algorithm, the Bellman–Ford algorithm, Floyd cycle detection algorithm, and Ford-Fulkerson algorithm. Inspired by deep learning's success in other areas, recent work in the literature has applied machine learning based methods to graph problems. Scarselli et al. [2008] proposed the Graph Neural Network (GNN) model, which can be applied to many different types of graphs and uses a recursive neural network (a neural network that can be applied recursively). Since this seminal paper, many different types of GNNs have been created.

Wu et al. [2020] divides GNNs into four main categories: Recurrent Graph Neural Networks (RecGNNs), Convolutional Graph Neural Networks (ConvGNNs), Graph Autoencoders (GAEs), and Spatial-Temporal Graph Neural Networks (STGNNs). We will now focus on ConvGNNs as they are the most relevant to our work. We refer the interested reader to [Hochreiter and Schmidhuber, 1997, Scarselli et al., 2008, Gallicchio and Micheli, 2010, Cho et al., 2014, Li et al., 2015, Dai et al., 2018], [Cao et al., 2016, Kipf and Welling, 2016b, Wang et al., 2016, Li et al., 2018b, You et al., 2018, Simonovsky and Komodakis, 2018] , and [Jain et al., 2016, Li et al., 2017, Yu et al., 2017, Seo et al., 2018, Yan et al., 2018, Wu et al., 2019, Guo et al., 2019] for an overview of RecGNNs, GAEs and STGNNs respectively.

2.5.1 Convolutional Graph Neural Networks (ConvGNNs)

Convolutional GNNs have been influenced by RecGNNs. Whereas, RecGNNs apply the same (contractive) function (f) either until convergence, or for a fixed number of times, ConvGNNs use a fixed number of layers, each with different weights, to update the embeddings. However, the functions corresponding to each layer are only used once. ConvGNNs can be separated into two categories: Spectral-based ConvGNNs and Spatial-based ones.

Spectral-Based ConvGNNs. Spectral-based ConvGNNs are based on methods that have been used in graph signal processing and focus on undirected graphs. They use the graph Laplacian which is defined as $\mathbf{L} = \mathbf{I}_n - \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-(1/2)}$, where $\mathbf{D}$ is the diagonal matrix of node degrees $\mathbf{D}_{ii} = \sum_j(\mathbf{A}_{i,j})$. Let $\mathbf{U}$ be the vector of eigenvectors of L, then we can define a Fourier transform $\mathcal{F}(\mathbf{x}) = \mathbf{U}^\top\mathbf{x}$. Now, given any filter $\mathbf{g} \in \mathbb{R}^n$, we can define a graph convolution for an input $\mathbf{x}$ that is based on this Fourier transform:

$$\mathbf{x} *_G \mathbf{g} = \mathcal{F}^{-1}\left(\mathcal{F}(\mathbf{x}) \odot \mathcal{F}(\mathbf{g})\right), \tag{2.35}$$

where $\odot$ is the elementwise product. All Spectral ConvGNNs are based on this definition; they vary in how to choose or compute $\mathbf{g}$. Bruna et al. [2013], Henaff et al. [2015], Defferrard et al. [2016], Kipf and Welling [2016a], Zhuang and Ma [2018], and Li et al. [2018a] all proposed GNNs that make use of this operation.

Spatial-Based ConvGNNs. Similarly to the convolutional operation in image based CNNs, spatial-based ConvGNNs perform updates based on a node's spatial relations. Graph convolutions compute a weighted sum of all neighbouring nodes. However, for graph convolutions the neighbouring nodes are unordered and their number is not fixed unlike CNNs which apply constant sized square filters by taking a weighted average over the neighbouring pixels of the central pixel. Many different spatial-based ConvGNNs have been proposed in the literature [Micheli, 2009, Atwood and Towsley, 2016, Niepert et al., 2016, Ying et al., 2018, Xu et al., 2018, Chiang et al., 2019, Velickovic et al., 2019]. The most relevant to our work is the message-passing neural network (MPNN) [Gilmer et al., 2017]. It provides a generalization of many different ConvGNN implementations. It assumes that we send messages along the edges of the graphs to update the node embeddings. Unlike with RecGNNs, the number of update iterations is fixed and the update function is unique for every iteration:

$$\boldsymbol{\mu}_v^{(k)} = U_k \left(\boldsymbol{\mu}_v^{(k-1)}, \sum_{u \in N(v)} M_k \left(\boldsymbol{\mu}_v^{(k-1)}, \boldsymbol{\mu}_u^{(k-1)}, \mathbf{z}_{vu}^e \right) \right). \tag{2.36}$$

In the original implementation, the embedding vectors are initialized to equal the node attributes: $\boldsymbol{\mu}_v = \mathbf{z}_v$. The functions $U_k(\cdot)$ and $M_k(\cdot)$ have learnable parameters. Once the message passing has been completed, the embedding vectors are often passed to an output function that completes the task that we want to carry out on the graph.

3 Generating Adversarial Examples with Graph Neural Networks

Contents

3.1 Introduction

Ever since Szegedy et al. [2013] showed that Neural Networks (NNs) are susceptible to adversarial attacks, it has become common practice to evaluate their robustness to various types of adversarial attacks. Most attack schemes use standard techniques from the optimization literature without significant adaptation for the specific problem at hand [Szegedy et al., 2013, Moosavi-Dezfooli et al., 2017, Goodfellow et al., 2015, Madry et al., 2018, Papernot et al., 2016]. At the other end of the spectrum are purely machine learning based techniques, which aim to learn the underlying probability distribution of adversarial perturbations to generate adversarial examples [Baluja and Fischer, 2017, Zhao et al., 2018, Poursaeed et al., 2018, Song et al., 2018]. However, the inductive bias incorporated in the network architectures of generative models ignores the iterative structure of optimization-based attacks. As a result, generative models often fail to match the performance of iterative optimization-based methods on finding minimal perturbations leading to adversarial examples. We therefore introduce a novel attacking method that combines the optimization based approach with learning.

Specifically, we propose the use of a graph neural network (GNN) that assists an iterative procedure resembling standard optimization techniques. The architecture of the GNN closely mirrors that of the network we wish to attack. Given an image, its true class and an incorrect target class, at each iteration the GNN proposes a direction for potentially maximizing the difference between the logits of the incorrect class and the correct class. Henceforth, we refer to the objective function we wish to maximize via the GNN as the adversarial loss function. Every single evaluation of the GNN is made up of one or more forward and backward passes that mimic a run of the network that we are attacking. When training the GNN we consider a horizon with a decay factor to output a direction of movement that maximizes the adversarial loss function. By using a parameterization of the GNN that depends only on the type of neurons and layers and not on the underlying architecture, we can train a GNN using one network and test it on another.

Our other main contribution is introducing a new method to assess the strength and efficiency of adversarial attacks. In the literature adversarial attacks are often compared using a trained model and some fixed allowed perturbation size. The method that manages to find an adversarial example for the highest number of images is considered to be the strongest one. However, the network to be attacked is often robust for a significant proportion of images. All attacks on these images will therefore fail. Conversely, for other images adversarial perturbations are very easy to find, again not demonstrating significant differences between methods. We therefore introduce a challenging dataset on three different neural networks of different sizes that are solely made up of properties for which adversarial examples exist. The size of the allowed perturbation is deliberately chosen for each element in the dataset leading to a very high level of difficulty. We hope that providing this new dataset will allow for a more efficient and meaningful comparison of different adversarial attacks in the future.

We compare our method, which we call AdvGNN, against various attacks on this dataset. AdvGNN reduces the average time required to find adversarial examples by more than 65% compared to several state-of-the-art attacks and also significantly reduces the rate of unsuccessful attacks. AdvGNN also achieves good generalization performance on unseen larger models.

3.2 Related Work

In this work we focus on white-box image-dependent targeted attacks, the strongest form of adversarial attacks.

Studying adversarial attacks, and white-box attacks in particular, has become an active field of research over the last few years. Adversarial attacks can be separated into three main categories [Serban et al., 2020]. One class of attacks aims to find an adversarial example that lies within some allowed perturbation and that the network misclassifies with a high level of confidence. Goodfellow et al. [2015] proposed the Fast Gradient Sign Method (FGSM) that takes a single step towards the gradient of the adversarial loss function. The Iterative Fast Gradient

Method (I-FGSM) [Kurakin et al., 2016] and Projected Gradient Attack (PGD) [Madry et al., 2018] both apply FGSM iteratively, taking several steps towards the sign of the gradient. Dong et al. [2018] proposed adding momentum to I-FGSM, thus significantly improving its performance (MI-FGSM).

A similar line of research aims to find an adversarial example with the smallest possible perturbation. Szegedy et al. [2013] proposed using limited-memory box constrained optimization (BFGS) to find the smallest perturbation required to change the prediction of the network. Carlini and Wagner [2017] approximate the objective function using a simpler linear function that can be solved using standard optimization algorithms. Moosavi-Dezfooli et al. [2016] introduced the Deepfool attack that exploits the assumption that the network behaves linearly near the original input. Both of these types of attacking strategies ignore the rich inherent structure of the problem and the data, information that can be used to come up with better ascent directions.

A third class of attacks includes generative methods. Baluja and Fischer [2017] train a second neural network (ATN) that, given an input, aims to output an adversarial example. Poursaeed et al. [2018] trained a generative method that also learns to generate image-specific perturbations. Xiao et al. [2018a] propose the use of a GAN that learns to approximate the distribution of the original images. All of these methods ignore the iterative nature of many optimization algorithms, resulting in a lower success rate in generating adversarial examples that are very close to original images.

We propose using a Graph Neural Network (GNN) to combine the strengths of both the optimization based and learning based methods to generate adversarial examples more efficiently. GNNs have been used in Neural Network Verification to learn the branching strategy in a Branch-and-Bound algorithm [Lu and Kumar, 2020a] and to estimate better bounds [Dvijotham et al., 2018a, Gowal et al., 2019], but to the best of our knowledge they have not yet been used to generate adversarial examples. We show in this work how they can be employed successfully for this task.

3.3 Problem Definition

In this section we define the problem of finding adversarial examples, and outline some of the most popular approaches to solving it.

We are given a neural network $f : \mathbb{R}^d \mapsto \mathbb{R}^m$ that takes a d-dimensional input and outputs a confidence score for m different classes. Specifically, we consider an L layer feed-forward neural network, with non-linear activations σ such that for any $\mathbf{x}_0 \in \mathcal{C} \subseteq \mathbb{R}^d$, $f(\mathbf{x}_0) = \hat{\mathbf{x}}_L \in \mathbb{R}^m$, where

$$\hat{\mathbf{x}}_{i+1} = W^{i+1}\mathbf{x}_i + \mathbf{b}^{i+1}, \qquad \text{for } i = 0, \dots, L-1, \tag{3.1}$$

$$\mathbf{x}_i = \sigma(\hat{\mathbf{x}}_i), \qquad \text{for } i = 1, \dots, L-1. \tag{3.2}$$

The terms W^i and $\mathbf{b}^i$ refer to the weights and biases of the i-th layer of the neural network f, and $\mathcal{C}$ is some convex input domain. Every convolutional filter can be rewritten as a linear layer; hence for the sake of clarity we treat convolutional layers like we do linear ones. Given an image $\mathbf{x}$, its true class y, an incorrect class $\hat{y}$, and an allowed perturbation ϵ, a targeted attack aims to find $\mathbf{x}'$, such that

$$f(\mathbf{x}')_{\hat{y}} \geq f(\mathbf{x}')_y \ \text{ and } \ d(\mathbf{x}, \mathbf{x}') \leq \epsilon, \tag{3.3}$$

for some distance measure d. In other words we aim to find an adversarial example $\mathbf{x}'$ that is close to the original input but is misclassified as $\hat{y}$. Problem (3.3) is often reformulated as follows:

$$max_{\mathbf{x}' \in \mathcal{B}(\mathbf{x}, \epsilon)} \ L(\mathbf{x}', y, \hat{y}) = f(\mathbf{x}')_{\hat{y}} - f(\mathbf{x}')_y, \tag{3.4}$$

where $\mathcal{B}(\mathbf{x}, \epsilon)$ is an ϵ-sized norm-ball around $\mathbf{x}$, that is,

$$\mathcal{B}(\mathbf{x}, \epsilon) := \{x' \mid d(\mathbf{x}, \mathbf{x}') \leq \epsilon\}. \tag{3.5}$$

We refer to L as the adversarial loss from now on. If $L(\mathbf{x}', y, \hat{y}) \geq 0$ then $\mathbf{x}'$ is considered an adversarial example. FGSM [Goodfellow et al., 2015], a fast attack on the l_∞ norm, aims to solve (3.4) by using the sign of the gradient of the adversarial loss:

$$\mathbf{x}' = \mathbf{x} + \epsilon \ \text{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}', y, \hat{y})). \tag{3.6}$$

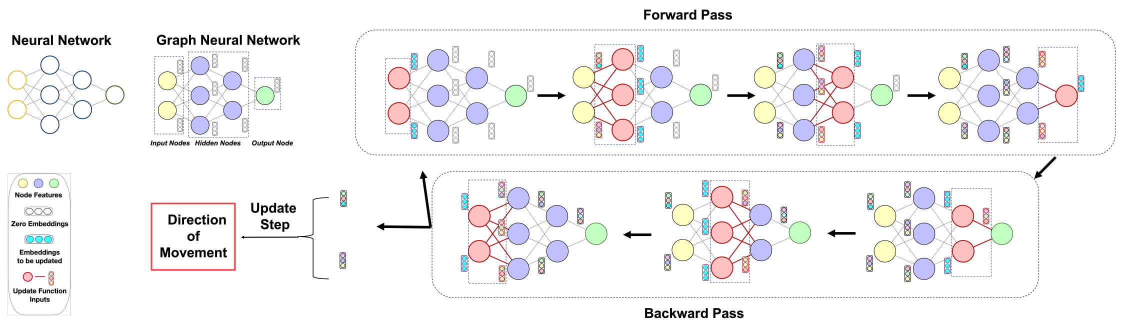


Figure 3.1: We illustrate the GNN Framework (Jingyue Lu helped make this figure). The structure of the GNN mimics that of the Neural Network we are attacking. We perform forward and backward message passing to update the embedding vectors. Finally, we perform an update step to transform the embedding vectors of the input level into a new direction of movement.

Madry et al. [2018] proposed applying this step iteratively, which equates to running Projected Gradient Descent (PGD) on the negative adversarial loss:

$$\mathbf{x}^{t+1} = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)} \left(\mathbf{x}^t + \alpha \ \mathrm{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}', y, \hat{y})\right). \tag{3.7}$$

Using the sign of the gradient of the adversarial loss as the direction of movement is effective when we do not have access to more information about the problem. However, we argue that in the white-box setting, where we have access to more information, the effectiveness of this approach is limited. We aim to replace the gradient by a more informed direction that, along with the gradient, takes the inherent structure of the problem and the data into consideration.

3.4 GNN Framework

The key observation of our work is that several previously known attacks can be thought of as performing forward-backward style passes through the network to compute an ascent direction for the adversarial loss function. Examples include, PGD, I-FGSM, and C&W (the method proposed by Carlini and Wagner [2017]). However, the exact form of the passes is restricted to those suggested by standard optimization algorithms, which are agnostic to the special structure of adversarial attacks. This observation suggests a natural generalization: parameterize the forward and backward passes, and estimate the parameters using a training dataset so as to exploit the problem and data structure more successfully. In what follows, we first provide an overview of our approach that achieves this generalization through graph neural networks (GNN). The remaining subsections describe the various components of the GNN and the forward and backward passes in greater detail.

3.4.1 Overview

We propose to use a GNN for the efficient computation of adversarial examples. Since previous attacks perform forward and backward passes on the network they wish to attack, it makes sense to use a GNN that mimics the architecture of that network as closely as possible. To this end, we treat the neural network as a graph $G_{NN} = (V_{NN}, E_{NN})$ and provide it as input for the GNN. We denote the GNN as an isomorphic graph to G_{NN}, that is, $G_{GNN} = (V_{GNN}, E_{GNN})$ where there is a one-to-one correspondence between the nodes V_{NN} and V_{GNN}, and edges E_{NN} and E_{GNN}. For every node $v \in V_{GNN}$ we first compute a feature vector $\mathbf{f}$, which contains local information about the node. We then use this feature vector and a learned function g to compute an embedding vector $\boldsymbol{\mu}$. The high-dimensional embedding vector encapsulates a lot of the important information about the corresponding node, the structure of the neural network, and the state of the optimization algorithm. The embedding vectors are initialized based on the node features and then updated using forward and backward passes in the GNN. Exchanging information with its neighbours ensures that the embedding vectors capture the global information of the structure of the problem. Once we have gotten a learned representation of each node we will convert the embedding vectors into a direction of movement. Having provided an overview we will now describe the GNN's main elements in greater detail.

3.4.2 GNN Components

Nodes. We create a node $\mathbf{v}_k[i]$ in our GNN for every node in the original network, where k indexes the layer and i the neuron. We denote the set of all nodes in the GNN by V_{GNN}.

Node Features. For each node $\mathbf{v}_k[i]$ we define a corresponding q-dimensional feature vector $\mathbf{f}_k[i] \in \mathbb{R}^q$ describing the current state of that node. Its exact definition depends on the task we want to solve. In our experiments the feature vectors consist of three parts: the first part captures the gradient at the current point; the second part includes the lower and upper bounds for each neuron in the original network based on the bounded input domain; and the third part encapsulates information that we get from solving a standard relaxation of the adversarial loss from the incomplete verification literature. A more detailed analysis can be found in Appendix A.1. While more complex features could be included, we deliberately chose the simple features described above and rely on the power of GNNs to efficiently compute an accurate direction of movement.

Edges. We denote the set of all the edges connecting the nodes in V_{GNN} by E_{GNN}. The edges are equivalent to the weights in the neural network that we are trying to attack. We define e^k_{ij} to be the edge connecting nodes $v_k[i]$ and $v_{k+1}[j]$ and assign it the value of W^k_{ij}.

Embeddings. For every node $v_k[i]$ we compute a corresponding p-dimensional embedding vector $\boldsymbol{\mu}_k[i] \in \mathbb{R}^p$ using a learned function g:

$$\boldsymbol{\mu}_k[i] := g(\mathbf{f}_k[i]). \tag{3.8}$$

In our case g is a simple multilayer perceptron (MLP), which is made up of a set of linear layers Θ_i and non-linear ReLU activations. We have the following set of trainable parameters:

$$\Theta^{init}_0 \in \mathbb{R}^{p \times q}, \quad \Theta^{init}_1, \dots, \Theta^{init}_{T_1} \in \mathbb{R}^{p \times p}. \tag{3.9}$$

Given a feature vectors $\mathbf{f}_k$, we compute the following set of vectors:

$$\boldsymbol{\mu}^0_k = \text{relu}(\Theta^{init}_0 \cdot \mathbf{f}_k), \quad \boldsymbol{\mu}^{l+1}_k = \text{relu}(\Theta^{init}_{l+1} \cdot \boldsymbol{\mu}^l_k). \tag{3.10}$$

We initialize the embedding vectors to be $\boldsymbol{\mu}_k = \boldsymbol{\mu}^{T_1}_k$, where $T_1 + 1$ is the depth of the MLP.

3.4.3 Forward and Backward Passes

So far, the embedding vector $\boldsymbol{\mu}$ solely depends on the current state of that node and does not take the underlying structure of the problem or the neighbouring nodes into consideration. We therefore introduce a method that updates the embedding vectors by simulating the forward and backward passes in the original network. The forward pass consists of a weighted sum of three parts: the first term is the current embedding vector, the second is the embedding vector of the previous layer passed through the corresponding linear or convolutional filters, and the third is the average of all neighbouring embedding vectors:

$$\begin{aligned}\boldsymbol{\mu}'_k[i] = \text{relu}\Bigg(&\Theta_1^{for}\boldsymbol{\mu}_k[i] + \Theta_2^{for}\left(W_k\boldsymbol{\mu}_{k-1} + \mathbf{b}_{k-1}\right)[i] + \\ &\Theta_3^{for}\left(\sum_{j\in N(i)} \boldsymbol{\mu}_{k-1}[j]/Q_{k+1}[j]\right)[i]\Bigg). \end{aligned} \tag{3.11}$$

Similarly, we perform a backward pass as follows:

$$\begin{aligned}\boldsymbol{\mu}_k[i] = \text{relu}\Bigg(&\Theta_1^{back}\boldsymbol{\mu}'_k[i] + \Theta_2^{back}(W_{k+1}^T\left(\boldsymbol{\mu}'_{k+1} - \mathbf{b}_{k+1}\right))[i] + \\ &\Theta_3^{back}\left(\sum_{j\in N'(i)} \boldsymbol{\mu}'_{k+1}[j]/Q'_{k+1}[j]\right)[i]\Bigg). \end{aligned} \tag{3.12}$$

Here $\Theta_1^{for}, \Theta_2^{for}, \Theta_3^{for}, \Theta_1^{back}, \Theta_2^{back}, \Theta_3^{back} \in \mathbb{R}^{p\times p}$ are all learnable parameters and W and b are the weights and biases of the target network as defined in equations (3.1) and (3.2). Both (3.11) and (3.12) can be implemented using existing deep learning libraries. To ensure better generalization performance to unseen neural networks with a different network architecture we include normalization parameters Q and Q'. These are matrices whose elements are the number of neighbouring nodes in the previous and following layer respectively for each node. We repeat this process of running a forward and backward pass T_2 times. The high-dimensional embedding vectors are now capable of expressing the state of the corresponding node taking the entire problem structure into consideration as they are directly influenced by every other node, even if we set $T_2 = 1$.

3.4.4 Update Step

Finally, we need to transform the p-dimensional embedding vector of the input layer to get a new direction $\tilde{\mathbf{x}}$. We simply use a linear output function $\boldsymbol{\Theta}^{out} \in \mathbb{R}^{1\times p}$ to get:

$$\tilde{\mathbf{x}} = \boldsymbol{\Theta}^{out} \cdot \boldsymbol{\mu}_0. \tag{3.13}$$

Ideally the GNN would output a new ascent direction that will lead us directly to the global optimum of equation (3.4). However, as the problem is complex this may not be feasible in practice without making the GNN very large, thereby resulting in computationally prohibitive inference. Instead, we propose to run the GNN a small number of times to return directions that gradually move towards the optimum.

Given a step size α, our previous point $\mathbf{x}^t$, and the new direction $\tilde{\mathbf{x}}$ we update as follows:

$$\mathbf{x}^{t+1} = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)} \left(\mathbf{x}^t + \alpha\tilde{\mathbf{x}}\right). \tag{3.14}$$

The hyper-parameters for the GNN computation of new directions of movement are the depth of the MLP (T_1), how many forward and backward passes we run (T_2), the embedding size (p), and the stepsize parameter α.

3.5 GNN Training

Having described the structure of the GNN we will now show how to train its learnable parameters. Our training dataset $\mathcal{D}$ consists of a set of samples $d_i = (\mathbf{x}^i, y^i, \hat{y}^i, \epsilon^i, W^i, \mathbf{b}^i)$, each with the following components: a natural input to the neural network we wish to attack ($\mathbf{x}$), for example an image; the true class (y); a target class ($\hat{y}$); the size of the allowed perturbation (ϵ), which in our case is an ℓ_∞ ball; and the weights and biases of the neural network ($W, \mathbf{b}$). We note that the allowed perturbation can be unique for each datapoint.

In order to get the individual components that make up the feature vectors, we first compute the intermediate bounds of each node in the network using the method by Wong and Kolter [2018] which is explained in greater detail in Appendix A.1.1.

We further solve a standard relaxation of the robustness problem via methods from the verification literature (A.1.2). Finally, we generate s different starting points which we sample uniformly at random from the input domain $\mathcal{B}(\mathbf{x}, \epsilon)$.

Recall that we do not use the GNN to directly compute the optimum adversarial example. Instead, we run it iteratively, where each iteration computes a new direction of movement. In order for the training procedure to closely resemble its behaviour at inference time, it is crucial to train the GNN using a loss function that takes into account the adversarial loss across a large number of iterations K.

Given the i-th training sample $d_i = (\mathbf{x}^i, y^i, \hat{y}^i, \epsilon^i, W^i, \mathbf{b}^i) \in \mathcal{D}$, and the j-th initial starting point we define the loss $\mathcal{L}_{i,j}$ to be:

$$\mathcal{L}_{i,j} = -\sum_{t=1}^{K} L(\mathbf{x}^{i,j,t}, y^i, \hat{y}^i) * \gamma^t. \tag{3.15}$$

Instead of maximizing over the adversarial loss, we minimize over the negative loss. If the decay factor $\gamma \in (0, 1)$ is low then we encourage the model to make as much progress in the first few steps as possible, whereas if γ is closer to 1, then more emphasis is placed on the final output of the GNN, sacrificing progress in the early stages. Readers familiar with reinforcement learning may be reminded of the discount rates used in algorithms such as Q-learning and policy-gradient methods. We sum over the individual loss values corresponding to each data point and each initial starting point to get the final training objective $\mathcal{L}$:

$$\mathcal{L} = \sum_{i=1}^{|D|} \sum_{j=1}^{s} \mathcal{L}_{i,j}. \tag{3.16}$$

In our experiments we train the GNN using the Adam optimizer [Kingma and Ba, 2015] and with a small weight decay.

Running Standard Algorithms using AdvGNN. As mentioned earlier, the motivation behind our GNN framework is to offer a parameterized generalization of previous attacks. We now formalize the generalization using the following proposition.

Proposition 1. *AdvGNN can simulate FGSM [Goodfellow et al., 2015], PGD attack [Madry et al., 2018], and I-FGSM [Kurakin et al., 2016].*

Proof. We show that our method is strictly more expressive than FGSM, I-FGSM, and PGD by showing that it can simulate each of them exactly. FGSM aims to generate an adversarial example with the following update step:

$$\mathbf{x}' = \mathbf{x} + \epsilon \ \mathrm{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}', y, \hat{y})). \tag{3.17}$$

Set $p = 2$ $q = 5$ and let $\Theta_0^{init} \in \mathbb{R}^{2\times 5}$ be the zero-matrix with non-zero elements $\Theta_0^{init}[1,2] = 1$, $\Theta_0^{init}[2,2] = -1$. Moreover, setting $T_1 = 1$, $\Theta_1^{init} = \mathbb{1}$ and $\mathbf{b}_0 = \mathbf{b}_1 = \mathbf{0}$, we get feature vectors

$$\mathbf{f}_0[i] := \left(\mathbf{x}^t,\ \mathrm{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y')), \mathbf{l}_k[i], \mathbf{u}_k[i], \mathbf{x}_k^{lp}[i]\right)^\top, \tag{3.18}$$

and embedding vectors

$$\begin{aligned}\boldsymbol{\mu}_0^0 &= \mathrm{relu}(\Theta_0^{init}\mathbf{f}_0) & (3.19)\\ &= \mathrm{relu}\left(\ \mathrm{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y')), -\ \mathrm{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y'))\right)^\top, & (3.20)\\ \boldsymbol{\mu} = \boldsymbol{\mu}_0^1 &= \mathrm{relu}(\Theta_1^{init}\boldsymbol{\mu}_0^0) = \boldsymbol{\mu}_0^0. & (3.21)\end{aligned}$$

See Appendix A.1 for a more detailed explanation of the design of the feature vectors. If we set $\Theta_2^{for} = \Theta_3^{for} = \Theta_2^{back} = \Theta_3^{back} = \mathbf{0}$ and $\Theta_1^{for} = \Theta_1^{back} = \mathbb{1}$, then the forward and backward passes do not change the embedding vector. We now just need to set $\boldsymbol{\Theta}^{out} = [1, -1]$ to get the new direction:

$$\begin{aligned}\tilde{\mathbf{x}} &= \boldsymbol{\Theta}^{out} \cdot \boldsymbol{\mu}_0 & (3.22)\\ &= \left(\ \mathrm{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y'))\right)_+ + \left(\ \mathrm{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y'))\right)_- & (3.23)\\ &=\ \mathrm{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y')). & (3.24)\end{aligned}$$

We now update as follows

$$\mathbf{x}^{t+1} = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)}\left(\mathbf{x}^t + \alpha\tilde{\mathbf{x}}\right) = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)}\left(\mathbf{x}^t + \alpha\ \mathrm{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y'))\right). \tag{3.25}$$

Setting $\alpha = \epsilon$ we get the same update as FGSM. We have shown that we can simulate FGSM using our GNN architecture by running AdvGNN once. Moreover, we can also simulate T iterations of PGD or I-FGSM by running AdvGNN T times. □

Network Name	**No. of Properties**	**Network Architecture**

‘Base’ Model	Training: 2500
	Validation: 50
	Testing: 641
		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

‘Wide’ Model	303	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

‘Deep’ Model	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

Table 3.1: Network Architectures. These are the three convolutional neural networks that we run our experiments on. One which we call the ‘Base’ model, one with the same layer structure but more hidden nodes which we call the ‘Wide’ model, and one with more hidden layers which we refer to as the ‘Deep’ model. All three use ReLU activation functions and are trained robustly using the methods of Wong and Kolter [2018] against l_∞ perturbations of size up to $\epsilon = 8/255$.

3.6 A New Dataset for Comparing Adversarial Attacks

In this section we describe our new dataset that has been specifically designed to compare state-of-the-art adversarial attacks. Previously, adversarial attacks were compared on how well they attack a trained neural network on a set number of images for a fixed allowed perturbation [Madry et al., 2018, Dong et al., 2018, Carlini and Wagner, 2017, Moosavi-Dezfooli et al., 2016]. However, for many of the images there either does not exist an adversarial example in the allowed perturbed input space or there exists a large number of different adversarial examples. In the first case, we do not learn anything about the differences between different methods as none of them return an adversarial example, and for the latter case all attacks will terminate very quickly, again not providing any insights. In practice only a small proportion of test cases affect the differences in performance between the various methods. To alleviate this problem we provide a dataset where the allowed input perturbation is uniquely determined for every image in the dataset.

This ensures that for every property there exist adversarial examples, but so few that only efficient attacks manage to find them.

We generate a dataset based on the CIFAR-10 dataset [Krizhevsky et al., 2009] for three different neural networks of various sizes. One which we call the 'Base' model, one with the same layer structure but more hidden nodes which we call the 'Wide' model, and one with more hidden layers which we refer to as the 'Deep' model. The exact network architectures are summarized in Table 3.1. All three are trained robustly using the method of Wong and Kolter [2018] against l_∞ perturbations of size up to $\epsilon = 8/255$ (the amount typically considered in empirical works). Our dataset is inspired by the work of Lu and Kumar [2020a] who created a verification dataset to compare defense methods on the same three models. They computed the largest possible perturbation norms for which the model is robust. More specifically, given a neural network f, a set of images x^i, they aim to find the maximum perturbation norms ϵ^i, such that there does not exist an adversarial example in $\mathcal{B}(\mathbf{x}, \epsilon^i)$. We want to generate a similar dataset but for comparing adversarial attacks rather than defense methods.

We generate the dataset by repeatedly picking an image from the CIFAR-10 test set, asserting that the network classifies the image correctly, and picking an incorrect class at random. We then aim to compute the smallest perturbation for which there exists an adversarial example by running an expensive binary search using PGD attacks with a large number of steps and restarts. We also generate a second dataset on the 'Base' model which we call the validation dataset and use to optimize various hyper-parameters for the attacks used in the next section.

A more detailed description of the algorithm can be found in Algorithm 2. The algorithm runs binary search together with PGD-attack to find the smallest perturbation for each image for which there exists at least one adversarial example. We generate a dataset setting the confidence parameter η to $1e{-}3$, the restart number to 20,000, and run PGD for 2,000 steps with a learning rate of $1e - 2$. We generate a dataset consisting of 641 properties for the 'Base' model, 303 properties for the 'Wide' model, and 250 properties for the 'Deep' model. We also create a validation

Algorithm 2 Generating Dataset

Input: a trained network $f : \mathbb{R}^d \mapsto \mathbb{R}^m$, a set D of N pairs of images and their respective classes $(\mathbf{x}^i, y^i)$.

Parameters: a binary search parameter η, a restart parameter R, as well as parameters for PGD.

Output:A set of tuples each consisting of an image $(\mathbf{x}^i)$, its class (y^i), an incorrect target class $(\hat{y}^i)$, and a perturbation parameter (ϵ^i)

```
1:  for $i = 1, \ldots, N$: do
2:    if $\arg\max f(\mathbf{x}^i) \neq y^i$ then
3:      continue {If the network misclassifies the image, skip to the next one}
4:    end if
5:    $\hat{y}^i \leftarrow$ random number from $\{0, \cdots, m-1\} \setminus \{y^i\}$ {Pick a random incorrect class as target}
6:    $\mathbf{l} \leftarrow 0$ {highest perturbation value for which we have failed to find an adversarial example}
7:    $\mathbf{u} \leftarrow 1.0$ {lowest perturbation value for which we have found an adversarial example}
8:    while $\mathbf{u} - \mathbf{l} \geq \eta$ do
9:      $\epsilon^i \leftarrow \frac{\mathbf{l}+\mathbf{u}}{2}$
10:     for $j = 1, \ldots, R$ do
11:       Run PGD with $(f, \mathbf{x}^i, y^i, \hat{y}^i, \epsilon^i)$ {Run PGD with $R$ restart or until found an adversarial example}
12:       if attack successful then
13:         Break
14:       end if
15:     end for
16:     if found adversarial example then
17:       $\mathbf{u} \leftarrow \epsilon^i$ {Update $\mathbf{u}$ as $\epsilon^i$ is now the lowest perturbation for which we have found an adversarial example}
18:     else
19:       $\mathbf{l} \leftarrow \epsilon^i$ {Update $\mathbf{l}$ as $\epsilon^i$ is now the highest perturbation for which we have failed to find an adversarial example}
20:     end if
21:   end while
22:   Record $(\mathbf{x}^i, y^i, \hat{y}^i, \epsilon^i)$
23: end for
```

dataset with the same parameters used as for the test dataset on the 'Base' model consisting of 50 properties; we further create a training dataset also on the 'Base' model with 2500 properties using $R = 100$ restarts and running PGD for 1,000 steps.

To make our dataset compatible with the verification dataset by Lu and Kumar

[2020a] we adopt their problem definition; there is, however, one big deficiency. They take an image $\mathbf{x} \in [0,1]^{32\times32\times3}$, normalize it to get $\bar{\mathbf{x}} = \frac{\mathbf{x}-\mu}{\sigma}$(as it common practice in computer vision) and then define the problem constraint using this normalized image: $\mathcal{B}(\bar{\mathbf{x}},\epsilon) := \{\mathbf{x}' \mid d(\bar{\mathbf{x}},\mathbf{x}') \leq \epsilon\}$. However, this norm ball contains inputs that do not correspond to natural images; in other words the unnormalized norm ball is not necessarily contained in the valid image space:

$$\{(\mathbf{x}' \times \sigma) + \mu \mid \mathbf{x}' \in \mathcal{B}(\bar{\mathbf{x}},\epsilon)\} \nsubseteq [0,1]^{32\times32\times3}. \tag{3.26}$$

Mathematically, the problem is still well defined, and it can still be used to compare adversarial attacks, but some solution to the problem are not in fact valid adversarial examples, as they cannot be interpreted as an image. We thus propose to apply the perturbation before normalization and also clamp the bounds to lie in the feasible regions:

$$\mathcal{B}(\mathbf{x},\epsilon) := \{\frac{\mathbf{x}'-\mu}{\sigma} \mid d(\mathbf{x},\mathbf{x}') \leq \epsilon \,\&\, \mathbf{x}' \in [0,1]^{32\times32\times3}\}. \tag{3.27}$$

When using (3.27) any solution to the adversarial problem is a valid image. We now generate a dataset using the same algorithm as before, but with different hyper-parameters. We generate two different datasets each on three different models ('Base', 'Wide' , 'Deep') and each consisting of 250 properties per model. The easier one of the two datasets is generated with the following hyper-parameters: $\eta = 0.001, R = 20000, lr = 0.1, iters = 2000$, whereas the harder one uses the following set of hyper-parameters: $\eta = 0.0005, R = 20000, lr = 0.1, iters = 8000$. The datasets are expensive to generate; it takes about 2 hours per property if run on a single GPU.

Finally, we note that in the literature only the success rate is reported when comparing different methods. The time taken by different methods is not analysed and the efficiency of the attacks is thus sometimes hard to determine. We propose to compare methods by reporting the success rate over running time to show both the speed and the strength of adversarial attacks.

3.7 Experiments

We now describe an empirical evaluation of our method by comparing it to several state-of-the-art attacks on the CIFAR-10 dataset. We first outline the experimental setting (§3.7.1), before describing the attacks we compare our method to (§3.7.2), and finally analysing the results (§3.7.3). Code for all experiments is available at `https://github.com/oval-group/AdvGNN`.

3.7.1 Setup

We run experiments on the dataset described in the previous section. The dataset is based on the CIFAR-10 dataset and includes three different networks to attack. All properties are SAT, meaning that there exists at least one adversarial example in the given input domain for each image and an overall success rate of 100% is theoretically achievable. We use a timeout of 100 seconds for each property. As most of the attacks we use rely on random initialisations the performance varies depending on the random seed. We thus run every experiment three times with three different seeds and report the average over the different runs.

We use the version of the dataset that normalises the images before applying the perturbation to achieve consistency with the dataset used in formal neural network verification [Lu and Kumar, 2020a]. This way, we have the potential to easily incorporate the GNN into verification algorithms, such as the Branch-and-Bound verification method that we will cover in Chapter 5. We will leave this for future work.

All the experiments were run under Ubuntu 16.04.4 LTS. All attacks were run on a single Nvidia Titan V GPU and three i9-7900X CPUs each. The implementation of our model as well as all baselines is based on Pytorch [Paszke et al., 2017].

3.7.2 Methods

We evaluate our methods by comparing it against PGD-Attack, MI-FGSM+, a modified version of MI-FGSM, and Carlini and Wagner attack, which according to several surveys on adversarial examples are all state-of-the-art methods [Akhtar and Mian, 2018, Chakraborty et al., 2018, Serban et al., 2020, Huang et al., 2020].

PGD. The first baseline we run is PGD-attack [Madry et al., 2018]. As described before, PGD-attack picks an initial starting point uniformly at random and then iteratively performs Projected Gradient Descent on the negative adversarial loss (3.7). Based on an extensive hyper-parameter analysis (see Appendix $A.2.1$) we pick the stepsize parameter $\alpha = 0.1$, and set the number of iterations to $T = 100$. We perform random restarts until we have either managed to find an adversarial example or the time limit has been reached.

MI-FGSM+. MI-FGSM is I-FGSM with an added momentum term. MI-FGSM starts at the image $\mathbf{x}$ and takes T steps of size ϵ/T. Defining the stepsize as such ensures that the current point lies in the feasible region throughout the entire algorithm without the need to project. To strengthen the attack we perform random restarts as we do for PGD. To ensure that not all runs of MI-FGSM on the same image are identical we therefore have to choose the initial point randomly as well. Furthermore, we perform a hyper-parameter search not only over the momentum term μ and the number of iterations T as done in the original paper, but also over the stepsize α (see Appendix A.2.2 for details). We run it with the following optimized parameters: $\alpha = 0.1$, $\mu = 0.5$, and $T = 100$. This modified version of MI-FGSM is denoted as MI-FGSM+.

C&W. The third baseline we use is C&W, the optimization-based attack proposed by Carlini and Wagner [2017]. C&W aims to find the smallest perturbation required to find an adversarial example by minimizing a loss function of the form $l(v) = c \cdot F(x+v) + \|(v-\tau)_+\|_1$ for some surrogate function F, and constants c and τ. There are a total of six hyper-parameters which we optimize over on a validation dataset and which we describe in greater detail in Appendix $A.2.3$. We note that in the original implementation C&W is often run until the minimum perturbation for which there exist at least one adversarial example is found. However, to be able to compare it to the other methods we stop the C&W attack as soon as an adversarial example is found for the given perturbation value or when the time limit is reached.

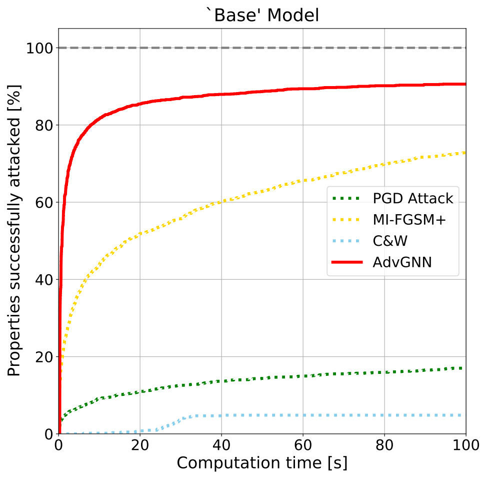
AdvGNN. The final attack we run is AdvGNN. We train our AdvGNN on the 'Base' model and on 2500 images of the CIFAR-10 test set that are not part of the dataset we test on. The ϵ values which define the allowed perturbation for each training sample are computed in a similar procedure to the test datasets described above. We train the GNN using the loss function described in section §3.5 with a horizon of 40 and with decay factor $\gamma = 0.9$. The training loss function is minimized using the Adam optimizer [Kingma and Ba, 2015] with a weight decay of 0.001. The initial learning for Adam is 0.01, and is manually decayed by a factor of 0.1 at epochs 20, 30, and 35. We pick the following values for the hyper-parameters of our AdvGNN: the stepsize α is 1e-2, the embedding size is $p = 32$, and we perform a single forward and backward pass ($T_1 = T_2 = 1$). To improve the performance on the 'Deep' model we fine-tune our AdvGNN for 15 minutes on the 'Deep' model before running the attack. Fine-tuning is run on 300 images that are not included in the 'Deep' test set. We use a fixed ϵ value of 0.25 for all images.

3.7.3 Results

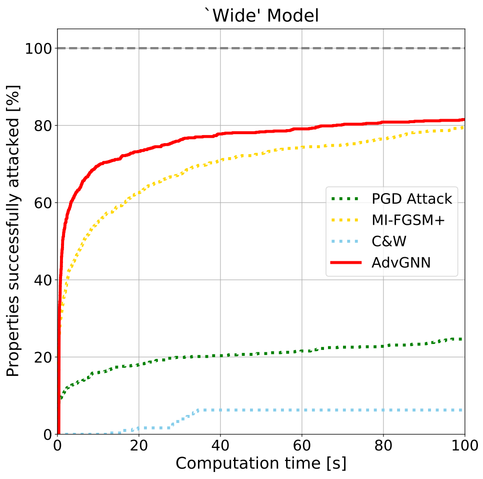
Method	Time(s)	Timeout(%)
PGD Attack	87.412	82.995
MI-FGSM+	40.438	27.145
C&W	97.385	95.164
AdvGNN	**13.527**	**9.412**

Table 3.2: 'Base' Model. We compare average (mean) solving time and the percentage of properties that the methods time out on when using a cut-off time of 100s. AdvGNN significantly outperforms all three baselines by reducing the average time taken to attack a property and by increasing the number of successfully attacked images.

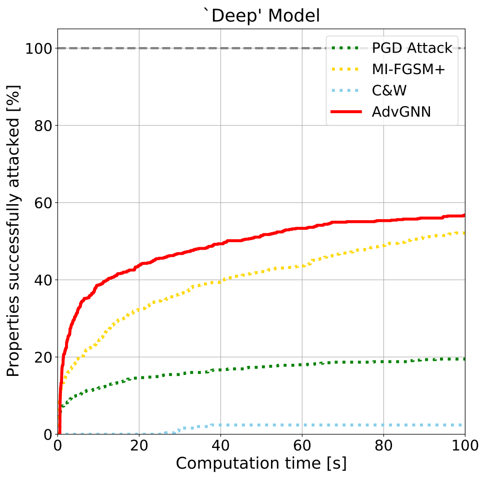
'Base' Model. We run all four methods described in the previous section on the 'Base' model with a timeout of 100 seconds and record the percentage of properties successfully attacked as a function of time (Figure 3.2a and Table 3.2). C&W only manages to find an adversarial advantage for less than 5% of all images. PGD outperforms C&W but still only manages to solve 17% of all properties. MI-FGSM+ outperforms PGD, timing out on 26% of all images with an average time of 40



(a) 'Base' model



(b) 'Wide' model



(c) 'Deep' model

Figure 3.2: Cactus plots for experiments on the 'Base' model (top), 'Wide' model (bottom left) and the 'Deep' model (bottom right). For each, we compare the different attacks by plotting the percentage of successfully attacked images as a function of runtime. Baselines are represented by dotted lines. AdvGNN beats all baselines for any chosen timeout value on the 'Base' model, which it has been trained on. Moreover, it shows good generalization performance by also outperforming all baselines on the previously unseen 'Wide' and 'Deep' models.

seconds. AdvGNN beats all three methods reducing both the average time taken and the proportion of properties timed out on by more than 65%.

'Wide' Model. Next we compare the methods on the 'Wide' model (Figure 3.2b and Table 3.3). AdvGNN has not seen this network during training and before

Method	Time(s)	Timeout(%)
PGD Attack	80.415	75.358
MI-FGSM+	31.144	20.462
C&W	96.366	93.729
AdvGNN	**24.089**	**18.482**

Table 3.3: 'Wide' Model. We compare the methods on the previously unseen 'Wide' model. We again compare average (mean) solving time and the percentage of properties that the methods time out on when using a cut-off time of 100s. AdvGNN significantly outperforms all three baselines thus showing good generalization performance on larger networks it has not seen during training.

running these experiments. MI-FGSM+ is again the best performing baseline, and AdvGNN the best performing method overall both in terms of average solving time and percentage of properties successfully attacked. AdvGNN reduces the time required to find an adversarial example by over 70% compared to PGD and C&W , and by 20% compared to MI-FGSM+. This demonstrates that AdvGNN achieves good generalization performance and can be trained on one model and used to run attacks on another.

'Deep' Model. We also run experiments on the 'Deep' model (Table 3.4, Figure 3.2c). We remind the reader that the AdvGNN parameters have been fine-tuned on this model for 15 minutes to achieve better results. AdvGNN outperforms all three other attacks on this larger 'Deep' model both with respect to the total number of successful attacks and the average time of each attack. Figure 3.2c shows that AdvGNN is still the best performing method even if we pick a shorter timeout of less than 100 seconds.

Easy Dataset. As some of the baselines, C&W in particular, struggle to successfully attack most of the properties in the previous experiment, we further compare the methods on a simpler dataset. We add a constant delta (0.001) to each epsilon value in the above dataset and reduce the timeout to 20 seconds. Increasing the allowed perturbation simplifies the task of finding an adversarial example as can be seen in Figure 3.3. All methods manage to find adversarial examples more quickly

Method	Time(s)	Timeout(%)
PGD Attack	84.349	80.533
MI-FGSM+	60.578	47.867
C&W	99.321	97.600
AdvGNN	**51.669**	**43.200**

Table 3.4: 'Deep' Model. We compare the methods on the previously unseen 'Deep' model. We again compare average (mean) solving time and the percentage of properties that the methods time out on when using a cut-off time of 100s. AdvGNN significantly outperforms all three baselines thus showing good generalization performance on larger networks it has not seen during training.

than on the original dataset and time out on significantly fewer properties. The relative order of the methods is the same on all three models in both the original and the simpler dataset. In particular, AdvGNN outperforms the baselines on all three models, reducing the percentage of unsuccessful attacks by at least 98% on the 'Base' model and by more than 65% on the 'Wide' and 'Deep' model.

	'Base' -Easy		'Wide' -Easy		'Deep' -Easy	
Method	Time(s)	Timeout(%)	Time(s)	Timeout(%)	Time(s)	Timeout(%)
PGD Attack	4.710	14.977	2.483	7.481	3.965	10.133
MI-FGSM+	1.288	2.964	0.778	0.990	1.578	2.933
C&W	17.030	69.111	15.978	60.396	17.487	76.000
AdvGNN	**0.518**	**0.052**	**0.595**	**0.330**	**1.465**	**0.933**

Table 3.5: Easy Dataset. We run experiments on the easier dataset. We compare average (mean) solving time and the percentage of properties that the methods time out on when using a cut-off time of 20s. The best performing method for each subcategory is highlighted in bold. AdvGNN outperforms all three baselines on all three models.

Adversarially Trained Model. We can confirm that our approach also works for adversarially trained models. We train a neural network that has the same artchitecture as the 'Wide' model used above using the method by Madry et al. [2018]. After finetuning our GNN on an this adversarially trained CIFAR10 model, AdvGNN outperforms both PGD and MI-FGSM+. We run all three methods on 101 properties with a timeout of 20 seconds and repeat the experiment three times with three different random seeds. AdvGNN clearly outperforms both baselines

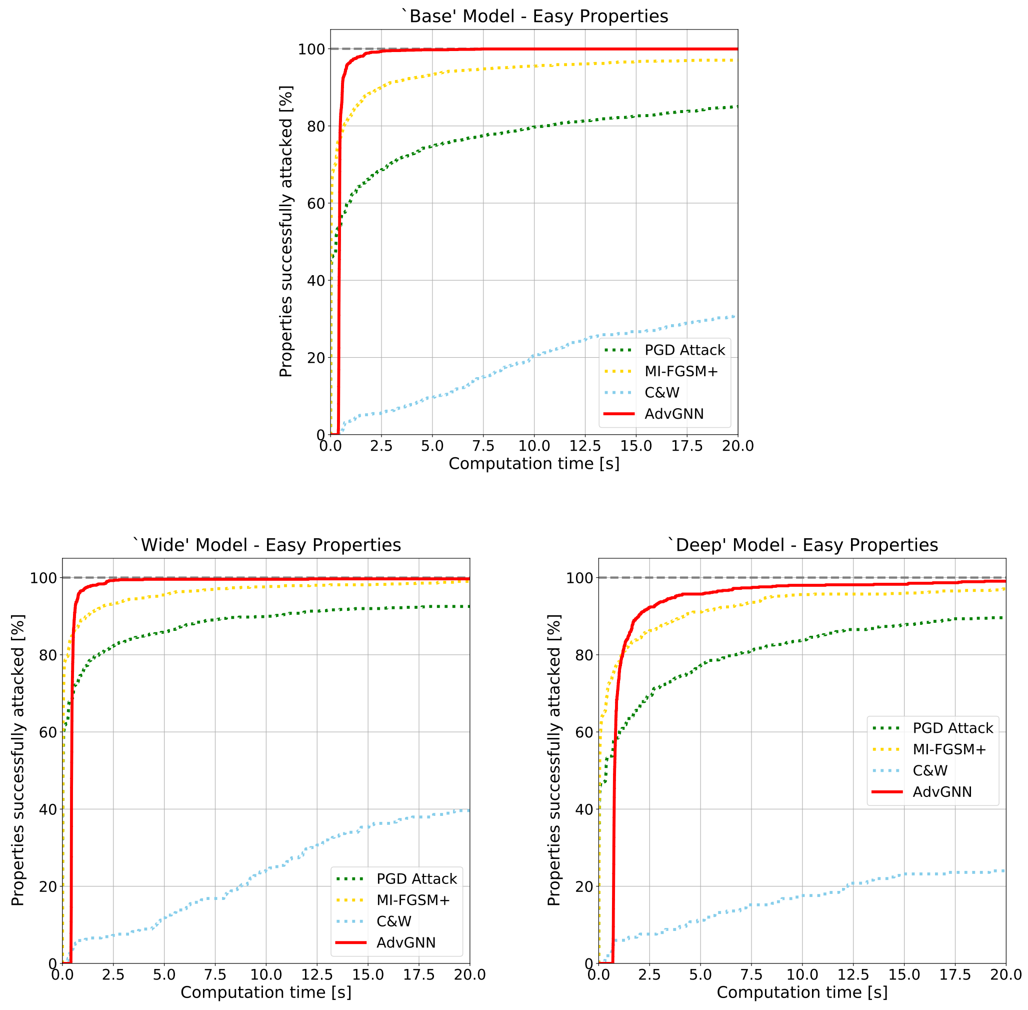


Figure 3.3: Cactus plots for experiments on the easier version of the dataset on the 'Base' model (left), 'Wide' model (middle) and the 'Deep' model (right). We add a small constant $\delta = 0.001$ to each perturbation size ϵ^i to make the adversarial problem easier to solve. For each model, we compare the attacks by plotting the percentage of successfully attacked images as a function of runtime. Similarly to the original dataset,AdvGNN is the best performing attack on all three models, again showing good generalization performance on unseen networks.

timing out on 14% of all properties compared to 21% for MI-FGSM+ and 78% for PGD, reducing average solving time by over 30% (see Table 3.6).

Ablation Study - Simpled Feature Vectors. Computing the features vectors (Equations (A.16) and (A.17)) requires solving a linear program (Equations A.15) as we describe in Appendix A.1. However, if we use a simpler approach as proposed by Wong and Kolter [2018] instead of super-gradient ascent our method still outperforms

Method	Time(s)	Timeout(%)
PGD Attack	16.509	78.2
MI-FGSM+	5.797	22.47
AdvGNN	**3.890**	**13.57**

Table 3.6: Adversarially Trained Model. We run experiments on the adversarially trained 'Wide' model. We compare average (mean) solving time and the percentage of properties that the methods time out on when using a cut-off time of 20s. Despite AdvGNN having been trained on the 'Base' model that has been trained robustly using the method of Wong and Kolter [2018], AdvGNN still outperforms all baselines on the adversarially trained 'Wide' model.

all baselines, successfully attacking 86% of all properties on the base model compared to 5%, 17%, and 73% for the three baselines, respectively (Table 3.7). The reduced performance compared to the original AdvGNN performance shows that the feature vector plays a significant role in generating better directions. At the same time the modified AdvGNN method still outperforms all baselines indicating that the KW can be used when we run our method on larger networks.

Method	Time(s)	Timeout(%)
PGD Attack	87.412	82.995
MI-FGSM+	40.438	27.145
C&W	97.385	95.164
AdvGNN-s	19.788	13.885
AdvGNN	**13.527**	**9.412**

Table 3.7: Ablation Study. We compare average (mean) solving time and the percentage of properties that the methods time out on when using a cut-off time of 100s. AdvGNN is the main method described; AdvGNN-s uses the simple KW method rather than the iterative supergradient ascent method to compute the feature vector (3.18). AdvGNN-s performs worse than the original AdvGNN method showing the importance of the design of the feature vector. Nevertheless, AdvGNN-s still outperforms all baselines.

3.8 Discussion

We introduced AdvGNN, a novel method to generate adversarial examples more efficiently that combines elements from both optimization based attacks and generative methods. We show that AdvGNN beats various state-of-the-art baselines reducing the average time taken to find adversarial examples by between 65 and 85 percent. We further show that AdvGNN generalizes well to unseen methods. Moreover, we introduced a novel challenging datasets for comparing different adversarial attacking methods. We show how it enables an illustrative comparison of different attacks and hope it will encourage the development of better attacks in the future.

Future work might include using AdvGNN for adversarial training, or for adversarial image detection. Furthermore, one could try incorporating AdvGNN into a complete verification method.

4 Attention for Adversarial Examples: Learning from Mistakes

Contents

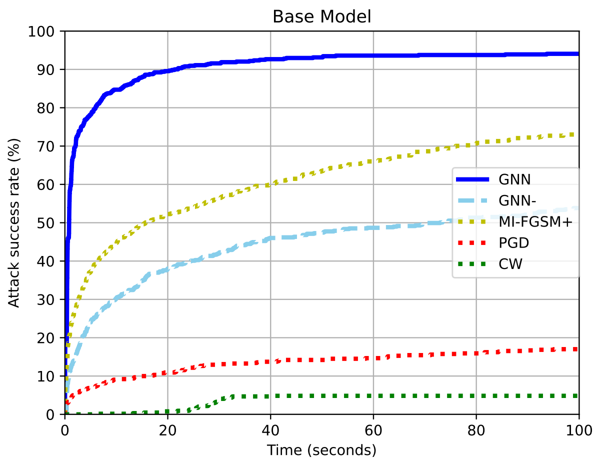


Figure 4.1: We compare our GNN based attention mechanism method against various adversarial attacks. We show the percentage of properties successfully attacked for any given time in seconds. Our method (solid line) outperforms all baselines (dotted lines). We also show that a similar GNN method that does not learn from past mistakes (dashed line), does not outperform all baselines, thereby demonstrating the importance of learning from previous unsuccessful attacks.

4.1 Introduction

Despite AI's high level of performance, often beating humans on tasks like computer vision, researchers both in academia and industry have called for machine learning based approaches to be regulated more heavily, especially due to their lack of explainability and vulnerability towards malicious attacks. Szegedy et al. [2013] showed that neural networks are susceptible to so-called adversarial attacks — inputs that are close to natural images, but ones that a trained neural network misclassifies, often with a high level of confidence. Tiny perturbations that are imperceptible to the human eye are often enough to trick the network. Numerous different methods to generate these adversarial examples have been proposed in the literature.

In this work we focus on white box attacks, one of the most well-studied problems in the adversarial attacks literature. In this setting the attacker has access to the network architecture, as well as its weights. Most of the state-of-the-art approaches are iterative methods using techniques from the standard optimziation literature [Moosavi-Dezfooli et al., 2016, Carlini and Wagner, 2017, Dong et al., 2018, Madry et al., 2018]. They start at an (often random) initial point and aim to arrive at an adversarial example using many optimization steps. To improve performance most of these attacks use random restarts. We highlight two challenges or weaknesses of this approach. Firstly, the search space is often large as it tends to be high-dimensional, and secondly at every restart we ignore all previous unsuccessful optimization attempts. We aim to alleviate both of these points using a new attention mechanism. Our method aims to greatly reduce the search space for future attack, by learning from its past mistakes, thereby increasing its chances of finding an adversarial example more quickly.

To this end, we propose to use a Graph Neural Network to reduce the search space for adversarial attacks. We treat the Neural Network we are attacking as a graph. We use message passing in our GNN to mimic the forward-backward nature of the back-propagation algorithm, which forms a subroutine of many adversarial attacks. Our GNN takes as input the targeted neural network as well as information from previous unsuccessful PGD attacks and outputs a new input domain that is smaller than the previous one. Our approach outperforms several widely used adversarial attacks as seen in Figure 4.1. We manage to reduce the number of properties for which we fail to find an adversarial example by over 70% compared to using the standard PGD attack with random restarts. Code for all experiments will be made available after publication.

4.2 Related Work

Several different types of adversarial attacks exist. We focus on white-box image-dependent targeted attacks, as they can be seen as the strongest form of attacks and are widely used to asses the robustness of neural networks. Moreover, one can create an untargeted attack using an ensemble of targeted ones.

Adversarial Attack Methods. Serban et al. [2020] separate adversarial attacks into three main categories. The first, which we focus on in this work, aims to find an adversarial example given an allowed perturbation norm. Examples include *Projected Gradient Descent* (PGD) [Madry et al., 2018], the *Fast Gradient Sign*

Method (FGSM) [Goodfellow et al., 2015], *Iterative FGSM* (I-FGSM) [Kurakin et al., 2016], and *I-FGSM with Momentum* (MI-FGSM) [Dong et al., 2018]. A second type of attacks aims to find an adversarial example with the smallest possible perturbation. The first such attack was proposed by Szegedy et al. [2013] using limited-memory box constrained optimization. Other methods have been proposed by Moosavi-Dezfooli et al. [2016] and Carlini and Wagner [2017]. A third line of research focuses on attacks that use machine learning based methods to learn to generate better adversarial examples, such as ATNs [Fischetti and Jo, 2018], and GAPs [Poursaeed et al., 2018], AdvGANs [Xiao et al., 2018a].

Attention Mechanisms. Attention mechanisms have been widely used in computer vision [Itti et al., 1998, Ramachandran et al., 2019], natural language procressing [Vaswani et al., 2017] and computational biology [Nambiar et al., 2020]. To the best of our knowledge, very limited work has been done to use attention for generating adversarial examples. Chen et al. [2017] proposed several techniques to reduce the search space and Cui et al. [2020] use active subspaces to generate adversarial examples. Recently, Wang et al. [2022] published a method that uses attention information to generate universal adversarial perturbations. Concurrent work by Jia et al. [2021] uses generative networks to improve the initialization for adversarial attacks used for adversarial training. Unlike our method their learnt initialization method is only conditioned on the natural image as well as the gradient information from the target network and does not learn from past attacks.

Graph Neural Networks. We introduce an attention mechanism utilizing Graph Neural Networks (GNN). GNNs have recently been used for neural network verification in Branch-and-Bound based algorithms: both to learn a branching strategy [Lu and Kumar, 2020a] and to learn better bounds [Dvijotham et al., 2018a, Gowal et al., 2019]. Based on this evidence we argue that GNNs are well suited for our problems, as we can treat the neural network we are attacking as a graph and simulate the back-propagation algorithm of adversarial attacks using message-passing.

4.3 Problem Definition

We now outline the problem definition along with some standard algorithms to solve it. Throughout this work we write scalars in non-bold italic lowercase or uppercase letters (λ or L); vectors will be written as bold non-italic lowercase letters ($\mathbf{z}$); the i-th element of a vector $\mathbf{z}$ will be denoted as $\mathbf{z}_i$; matrices will be denoted as bold non-italic uppercase letters ($\mathbf{W}$); the element of the matrix $\mathbf{W}$ appearing in the i-th row and the j-th column will be denoted as $\mathbf{W}_{i,j}$.

We are given an L-layer (convolutional) Neural Network $f : \mathbb{R}^d \mapsto \mathbb{R}^m$, that takes as input a d-dimensional vector, in our case an image, and outputs an m-dimensional vector, corresponding to the m different classes of our classification problem. Given weights $\mathbf{W}^{(j)}$, biases $\mathbf{b}^{(j)}$, and a non-linear activation function σ, f can be defined as follows:

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

$$\mathbf{x}_i = \sigma(\hat{\mathbf{x}}_i), \qquad \text{for } i = 1, \ldots, L-1. \tag{4.2}$$

Here $\mathbf{x}^{(0)} \in \mathbb{R}^d$, is the input image, and $\mathbf{x}^{(L)}$ is the output vector. Throughout this work we use ReLU activations, as they are the most commonly used activation for feed-forward neural networks [Ramachandran et al., 2017]. As we are considering image classification problems in this work, $f(\mathbf{x})_j = \mathbf{x}_j^{(L)}$ can be interpreted as the confidence value that the input belongs to the j-th class; the image thus gets classified as $\arg\max_j f(\mathbf{x})_j$ by f.

An adversarial example is an input vector that is close to a natural image but one that gets misclassfied by our network. That is, given a real image $\mathbf{x}$ with true label y, $\mathbf{x}'$ is an adversarial example if it lies near $\mathbf{x}$ and $f(\mathbf{x})_y < f(\mathbf{x})_{\hat{y}}$ for some incorrect class $\hat{y}$. There are many ways to compute the similarity between two images; we use the infinity norm, as it has been widely used in the literature [Madry et al., 2018, Dong et al., 2018]. We require that the distance between $\mathbf{x}$ and $\mathbf{x}'$ is less than some given parameter ϵ, that is $d(\mathbf{x}, \mathbf{x}') := \|\mathbf{x} - \mathbf{x}'\|_\infty := max_j|\mathbf{x}_j - \mathbf{x}'_j| \le \epsilon$.

We can formulate the problem of finding adversarial examples as an optimization problem:

$$max_{\mathbf{x}' \in \mathcal{B}(\mathbf{x},\epsilon)} \; L(\mathbf{x}', y, \hat{y}) = f(\mathbf{x}')_{\hat{y}} - f(\mathbf{x}')_{y}, \tag{4.3}$$

where $\mathcal{B}(\mathbf{x}, \epsilon)$ is an ϵ-sized infinity norm-ball around $\mathbf{x}$:

$$\mathcal{B}(\mathbf{x}, \epsilon) := \{\mathbf{x}' \mid d(\mathbf{x}, \mathbf{x}') \leq \epsilon\}. \tag{4.4}$$

We call L the adversarial loss and $\mathbf{x}'$ an adversarial example if $L(\mathbf{x}', y, \hat{y}) > 0$. Many algorithms to solve (4.3) have been proposed in the literature. The first method was the *fast gradient sign method* (FGSM) [Goodfellow et al., 2015] that takes a single step towards the sign of the adversarial gradient:

$$\mathbf{x}' = \mathbf{x} + \epsilon \; \mathrm{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}', y, \hat{y})). \tag{4.5}$$

Madry et al. [2018] proposed what is commonly referred to as the PGD attack, a method that applies the FGSM step iteratively. This corresponds to running Projected Gradient Descent on the negative adversarial loss. The $(t+1)^{th}$ update step is defined as:

$$\mathbf{x}^{t+1} = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)} \left(\mathbf{x}^{t} + \alpha \; \mathrm{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}', y, \hat{y})) \right). \tag{4.6}$$

As problem (4.3) is non-convex, the solution of any algorithm depends heavily on the initialization. To improve the success rate of the attack, we can perform random restarts where we initialize $\mathbf{x}^0 \in \mathcal{B}(\mathbf{x}, \epsilon)$ randomly each time. However, each time we initialize $\mathbf{x}^0$, we ignore all previous unsuccessful attacks, as well as the structure of the input domain and the optimization problem in general. We propose using an attention mechanism, which significantly reduces the search space thus leading to better initializations and a more efficient attack. Our attention method utilizes Graph Neural Networks (GNNs). We will describe the GNN framework in the next section

4.4 GNN Framework

Our method is motivated by two observations: firstly, neural networks can be interpreted as graphs, with the neurons being nodes, and the weights being edges; secondly common adversarial attacks, such as PGD attack, can be described as taking a forward and a backward pass through this network to generate the adversarial gradient ($\nabla_{\mathbf{x}} L(\mathbf{x}', y, \hat{y})$). These observations naturally lead to the idea of using Graph Neural Networks to help with the generation of adversarial examples. The structure of our GNN is based on the neural network we are trying to attack and the message passing algorithm mimics the forward-backward steps used by the PGD attack. Our method is inspired by work by Lu and Kumar [2020a] who use GNNs to make better branching decisions for Neural Network complete verification problems.

4.4.1 Overview

We start by providing a high level overview of our approach. The details of the various components of our framework are provided in the remainder of the section. Given a Neural Network f that we are trying to attack, we create a corresponding Graph Neural Network $G_f = (V, E)$. The term V is the set of vertices in the GNN; we create one vertex for every neuron in the original network f. Similarly, E describes the set of edges, where we have an edge between two nodes if and only if the two corresponding neurons are connected in f. We further create a feature vector $\mathbf{z}_v \in \mathbb{R}^p$ for every $v \in V$. These contain important information about the node that we pass to the GNN. Next, we create a multi-dimensional learnt embedding vector $\boldsymbol{\mu}_v \in \mathbb{R}^d$ for every $v \in V$. Once we have created an embedding vector for every node, we start the forward-backward messaging passing algorithm that updates embedding vectors based on information from previous and later layers. Finally, we use another learnt function that takes the embedding vectors corresponding to the nodes in the input layer and outputs a new smaller search space.

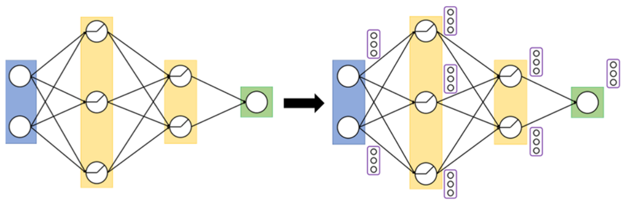


Figure 4.2: GNN Framework. On the left is the original neural network that we are trying to attack, and on the right is the GNN with the embedding vectors initialized for each node. There is a one-to-one correspondence between the nodes of the GNN and the neurons of the original neural network f, and between the GNN edges and the Neural Network weights. The nodes are separated into input, hidden, and output layers (Figure provided by Aleksandr Agadzhanov).

4.4.2 Components

The structure of our GNN including nodes, edges, and node embedding vectors is depicted in Figure 4.2. We now describe each part in more detail.

Nodes. We create one node for every neuron in the original network f. We denote the set of nodes in the GNN as V.

Edges. The set of edges E in the GNN is based on the weights in the original network. The edge weights correspond to the weights in f. The edges influence the message passing algorithm described below.

Node Features. We compute a feature vector $\mathbf{z}_v$ for every $v \in V$. We separate all nodes into three categories: input nodes, hidden nodes, and output nodes. We use different methods to generate the node features for each of them. The feature vectors for the input, hidden, and output layers are of dimension p_{inp}, p_{hid}, and p_{out} respectively. We aim to use data that encapsulates as much useful information of the respective nodes as possible, while at the same time being cheap to compute. Some of the features we use include current bounds of the node as well as information from previous unsuccessful PGD attacks. The former defines the search space for adversarial attacks. The latter is key to the success of our framework, as we aim to learn from the unsuccessful attempts to increase the chance of success in the future. This forms a contrasts to the traditional approach of using random restarts that ignore all previous runs. A detailed explanation of how we compute these features can be found in Appendix B.1.

Node Embeddings. Embedding vectors are multi-dimensional vectors that are generated using learnt functions. We initialize the embedding vectors for the input layer using a learnt function $F_{inp} : \mathbb{R}^{p_{inp}} \mapsto \mathbb{R}^d$ that is parameterized by $\boldsymbol{\theta}^0$ and takes as input the feature vectors for the input layer ($\mathbf{z}^0$):

$$\boldsymbol{\mu}_j^0 = F_{inp}(\mathbf{z}_j^0; \boldsymbol{\theta}^0). \tag{4.7}$$

We describe F_{inp} and the other learnt functions mentioned below in greater amount of detail in Appendix B.2. All other embedding vectors are initialized and updated using functions that take both local feature vectors as well as neighbouring embedding vectors as input. We describe this update procedure in the next subsection. Unlike the feature vectors, the embedding vectors are influenced by neighbouring nodes and thus include information about the state of the entire problem. In particular, once we have completed the message passing algorithm, the embedding vectors of the input layer are influenced by all other embeddings and can thus be used to generate a new smaller search space.

4.4.3 Message Passing

The power of the GNN lies in the message passing algorithm. We initialize and update embedding vectors in a forward-backward manner that is based on the gradient computation procedure used by PGD. Figure 4.3 illustrates the update procedure. We now describe these forward-backward update steps of the GNN in greater detail.

Forward Pass. We iteratively update one layer at a time, starting with the first hidden layer. We compute the embedding vector for the j-th node in the i-th layer using a learnt function $F_{hid} : \mathbb{R}^{p_{hid}+d} \mapsto \mathbb{R}^d$, that takes as input the local

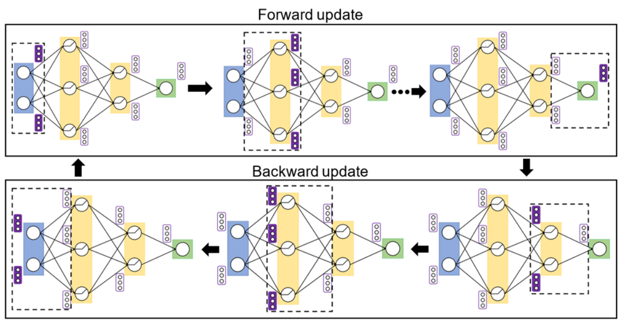


Figure 4.3: GNN Message Passing. The forward update steps are depicted in the top row, and the following backward pass in the bottom row. Embedding vectors are updated using learnt functions that take as input both feature vectors and embedding vectors from previous or later layers respectively. We can perform several rounds of this message passing cycle (Figure provided by Aleksandr Agadzhanov).

feature vectors of the i-th layer ($\mathbf{z}_j^i$), the embedding vectors from the previous layer ($\boldsymbol{\mu}^{i-1}$), and the edge matrix (E) as follows:

$$\boldsymbol{\mu}_j^i = F_{hid}(\mathbf{z}_j^i, \boldsymbol{\mu}^{i-1}, E; \boldsymbol{\theta}^1) \qquad \forall i \in \{1, \cdots, L-1\}. \tag{4.8}$$

Next, we compute the embedding vector for the j-th node of the output layer, using another learnt function $F_{out} : \mathbb{R}^{p_{out}+d} \mapsto \mathbb{R}^d$. It also takes the local feature vector of the corresponding node ($\mathbf{z}_j^i$) the embedding vectors from the final hidden layer ($\boldsymbol{\mu}^{L-1}$) and the edge matrix (E) to compute an embedding vector for the final layer:

$$\boldsymbol{\mu}_j^L = F_{out}(\mathbf{z}_j^L, \boldsymbol{\mu}^{L-1}; \boldsymbol{\theta}^2). \tag{4.9}$$

Once we have finished the forward passes, all embedding vectors have been influenced by all embeddings from previous layers, so as long as the corresponding nodes are connected via a path in the original network f. As our main focus lies on the embedding vectors of the input layer we now need to send the information backwards through the GNN.

Backward Pass. At this point we have computed an embedding vector for every node in the GNN, and every embedding vector is influenced by embedding vectors from all previous layers. We now perform a backward pass, to send information back from the output layer to the input layer, inspired by the nature of the the back-propagation algorithm that is used to compute the adversarial gradient for the PGD attack. For all hidden layers we update the embedding vectors as follows:

$$\boldsymbol{\mu}_j^i = B_{hid}(\mathbf{z}_j^i, \boldsymbol{\mu}^{L+1}, E; \boldsymbol{\theta}^3) \qquad \forall i \in \{1, \cdots, L-1\}. \tag{4.10}$$

Like for the forward pass, the backward function takes as input the feature vector, but instead of using the embedding vectors from the previous layer, we now use the embedding vector for the following layer, as we are passing information backwards. Once we have updated all hidden layers we update the embedding vector for the input layer:

$$\boldsymbol{\mu}_j^0 = B_{inp}(\mathbf{z}_j^0, \boldsymbol{\mu}^1, E; \boldsymbol{\theta}^4). \tag{4.11}$$

At this point all embedding vectors have been updated based on information from all other layers. We can repeat this forward-backward update scheme if we like. We note that at any point we only need to keep embedding vectors for one single layer in memory. In Appendix B.2 we define and describe the five different functions implementing the forward-backward messaging passing scheme in greater detail. All functions can be implemented efficiently using functions from standard deep learning packages. Next we will need to generate a new input domain based on the embedding vectors.

4.4.4 Reducing the Search Space

We now use the embedding vectors of the input layer ($\boldsymbol{\mu}^0$) to output a new search space. We aim to split each input node separately. For the j-th node, given a lower bound l_j and an upper bound u_j we want to generate new tighter bounds $\bar{l}_j$ and $\bar{u}_j$ such that $l_j \leq \bar{l}_j \leq \bar{u}_j \leq u_j$. We use a learnt function $g : \mathbb{R}^d \mapsto 2$ to get

$$\begin{bmatrix} \hat{l}_j \\ \delta_j \end{bmatrix} = g(\boldsymbol{\mu}_j^0; \boldsymbol{\theta}_5). \tag{4.12}$$

Here, $\hat{l}_j$ can be interpreted as the new lower bound, and δ_j as the offset parameter, which defines the difference between the new upper and lower bounds. To ensure feasibility we take

$$\bar{l}_j \leftarrow \max\{\min\{\hat{l}_j, u_j\}, l_j\}. \tag{4.13}$$

This leads to a new lower bound lying in between the old lower and upper bounds. Next we compute the new upper bound $\bar{u}_j$ using both the new lower bound and the offset parameter:

$$\bar{u}_j \leftarrow \max\{\min\{\bar{l}_j + \delta_j, u_j\}, \bar{l}_j\}. \tag{4.14}$$

This leads to a new upper bound lying in between the new lower bound and the old upper bound. In the next section we describe how to evaluate the strength of these new bounds and how to train a GNN successfully.

4.4.5 GNN Attention Algorithms

We now describe the GNN Attention in greater detail. Algorithm 4 summarizes the main method. We first run PGD attack (Algorithm 3) on the original input domain. If the attack is successful we return the adversarial example it found. If not, we call the GNN (Algorithm 5) to return a smaller search space. The GNN takes information from the previously unsuccessful attack as input ($\mathbf{v}$) in order to learn for future attacks. We then run PGD again on this smaller search input space. The GNN is made up of three main steps. It first initializes the feature vectors, it then performs forward and backward passes to create and update the embedding vectors, and finally it takes the embedding vector of the input domain to compute a new set of bounds. We then use these new bounds for future PGD attacks.

4.5 GNN Training

Having described the GNN framework along with its message passing algorithm, we now show how to train it. We first describe the loss function that we are trying to optimize over and that explains how well our GNN is performing. We then outline the training dataset used to learn the optimal GNN parameters.

4.5.1 Objective Function

Let us first denote the set of learnable parameters as

$$\mathbf{\Theta} = \begin{bmatrix} \boldsymbol{\theta}_0^T & \boldsymbol{\theta}_1^T & \boldsymbol{\theta}_2^T & \boldsymbol{\theta}_3^T & \boldsymbol{\theta}_4^T & \boldsymbol{\theta}_5^T \end{bmatrix}^T. \tag{4.15}$$

Algorithm 3 PGD Attack

Input: Neural Network f, natural image $\mathbf{x}$, true label y, incorrect target label $\hat{y}$, input domain B.
Parameters: step size α, iteration parameter T, restart parameter R.
Output: an adversarial example or gradient information from unsuccessfull attacks.

```
 1: initialize an empty dictionary $\mathbf{v}$ to store information of the PGD attack
 2: for $r = 0, \ldots, R$ do
 3:     sample $\mathbf{x}^0$ from B uniformly at random
 4:     for $t = 0, \ldots, T$ do
 5:         if $L(\mathbf{x}^t, y, \hat{y}) > 0$ then
 6:             Return: $(\mathbf{x}^t, None)$
 7:         else
 8:             $\mathbf{x}^{t+1} = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)}\left(\mathbf{x}^t + \alpha \operatorname{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, \hat{y})\right).$
 9:             add $\nabla L(\mathbf{x}^t, y, \hat{y})$ and $\mathbf{x}^{t+1}$ to $\mathbf{v}$
10:         end if
11:     end for
12: end for
13: Return: $(None, \mathbf{v})$
```

We use a supervised learning approach to train our GNN. Our training dataset contains adversarial examples that have been generated by successful but very expensive PGD attacks. We aim to train the GNN to output new bounds that contain these adversarial examples whilst being as tight as possible to make the adversarial example easy to find.

Given an adversarial example $\mathbf{x}_{\text{PGD}}$ returned by a successful PGD attack, and bounds $\bar{l}(\mathbf{\Theta})$ and $\bar{u}(\mathbf{\Theta})$ outputted by the GNN, we define a loss for each of the d input nodes. The loss consists of two parts. The first one checks whether the adversarial example lies within the bounds:

$$L_{1,i}(\mathbf{\Theta}) = \begin{cases} 0 & \text{if } \bar{l}_i(\mathbf{\Theta}) \leq x_{\text{PGD},i} \leq \bar{u}_i(\mathbf{\Theta}) \\ \bar{l}_i(\mathbf{\Theta}) - x_{\text{PGD},i} & \text{if } x_{\text{PGD},i} < \bar{l}_i(\mathbf{\Theta}) \\ x_{\text{PGD},i} - \bar{u}_i(\mathbf{\Theta}) & \text{if } x_{\text{PGD},i} > \bar{u}_i(\mathbf{\Theta}). \end{cases} \tag{4.16}$$

The loss is zero if the adversarial example lies within the bounds and increases linearly with the distance between the true adversarial example and the bounds returned by the GNN. Note, we simplified notation to improve clarity: the output of the GNN $\bar{l}_i(\mathbf{\Theta})$ does not only depend on the learnt parameters $\mathbf{\Theta}$, but also on information needed to initialize the feature vectors and the network f.

Algorithm 4 PGD Attack with GNN Attention

Input: Neural Network f, natural image $\mathbf{x}$, true label y, incorrect target label $\hat{y}$, perturbation size ϵ, a trained GNN parameterized by $\mathbf{\Theta}$
Parameters: Restart parameters (R, R_{GNN}), PGD parameters (α, T, R_{PGD}), GNN parameters
Output: an adversarial example or *None*

```
1: for $r = 0, \dots, R$: do
2:     $(\mathbf{x}_{PGD}, \mathbf{v}) \leftarrow PGD(f, \mathbf{x}, y, \hat{y}, \mathcal{B}(\mathbf{x}, \epsilon))$ {Initial PGD attack (Algorithm 3)}
3:     if $\mathbf{x}_{PGD}$ is not None then
4:         Return: $\mathbf{x}_{PGD}$
5:     end if
6:     for $r_{gnn} = 0, \dots, R_{GNN}$: do
7:         $B_{GNN} \leftarrow GNN(f, \mathbf{x}, y, \hat{y}, B_{GNN}, \mathbf{v})$ {Reduce search space (Algorithm 5)}
8:         $(\mathbf{x}_{PGD}, \mathbf{v}) \leftarrow PGD(f, \mathbf{x}, y, \hat{y}, B_{GNN})$ {Run attacks (Algorithm 3)}
9:         if $\mathbf{x}_{PGD}$ is not None then
10:            Return: $\mathbf{x}_{PGD}$
11:        end if
12:    end for
13: end for
14: Return: None
```

We also want to encourage the bounds to be as tight as possible, in order to significantly reduce the size of the search space. To this end, we define a second loss term that encourages tightness of the bounds:

$$L_{2,i}(\mathbf{\Theta}) = \frac{\bar{u}_i(\mathbf{\Theta}) - \bar{l}_i(\mathbf{\Theta})}{u_i - l_i}. \tag{4.17}$$

Our final loss function is a normalized combination of the two sums describing both the tightness of the bounds and whether they include the ground truth:

$$L(\mathbf{\Theta}) = \frac{1}{d} \sum_{i=1}^{d} L_{1,i}(\mathbf{\Theta}) + \lambda \cdot L_{2,i}(\mathbf{\Theta}). \tag{4.18}$$

Here, λ is a fixed parameter that determines the relative weighting of the two loss functions. If λ is small then the GNN focuses on minimizing the first loss term and in the process becomes more conservative to ensure that we do not exclude the ground truth from the new subspace. If, on the other hand, λ is large, then the GNN becomes more risky and aims to output a much smaller subspace. We fix the value of λ to be 0.033, having run a cross-validation analysis.We will try to minimize this loss using the Adam optimizer [Kingma and Ba, 2015] with a learning rate of 1e-4 and no weight decay.

Algorithm 5 GNN Attention

Input: Neural Network f, natural image $\mathbf{x}$, true label y, incorrect target label $\hat{y}$, input domain B, summary from an unsuccessful attack $\mathbf{v}$

Parameters: Number of forward and backward passes R.

Output: A smaller input domain B_{GNN}

```
 1: {Initialize Feature Vectors}
 2: $\mathbf{z} \leftarrow feat\_init(f, \mathbf{x}, y, \hat{y}, B, \mathbf{v})$
 3: {$R$ rounds of Forward and Backward Passes}
 4: for r = 0, ..., R: do
 5:     $\boldsymbol{\mu}_j^0 = F_{inp}(\mathbf{z}_j^0; \boldsymbol{\theta}^0)$ {Initialize input embedding vector.}
 6:     for $i \in \{1, \cdots, L-1\}$: do
 7:         $\boldsymbol{\mu}_j^i = F_{hid}(\mathbf{z}_j^i, \boldsymbol{\mu}^{i-1}, E; \boldsymbol{\theta}^1)$ {Forward Update Step for Hidden Layers.}
 8:     end for
 9:     $\boldsymbol{\mu}_j^L = F_{out}(\mathbf{z}_j^L, \boldsymbol{\mu}^{L-1}; \boldsymbol{\theta}^2)$ {Forward Update Step for Output Layer.}
10:     for $i \in \{1, \cdots, L-1\}$ do
11:         $\boldsymbol{\mu}_j^i = B_{hid}(\mathbf{z}_j^i, \boldsymbol{\mu}^{L+1}, E; \boldsymbol{\theta}^3)$ {Backward Update Step for Hidden Layers.}
12:     end for
13:     $\boldsymbol{\mu}_j^0 = B_{inp}(\mathbf{z}_j^0, \boldsymbol{\mu}^1, E; \boldsymbol{\theta}^4)$. {Backward Update Step for Input Layers.}
14: end for
15: {Generate Bounds}
16: $\begin{bmatrix} \hat{l}_j \\ \delta_j \end{bmatrix} = g(\boldsymbol{\mu}_j^0; \boldsymbol{\theta}_5)$
17: $\bar{l}_j \leftarrow \max\{\min\{\hat{l}_j, u_j\}, l_j\}$
18: $\bar{u}_j \leftarrow \max\{\min\{\bar{l}_j + \delta_j, u_j\}, \bar{l}_j\}$
19: Return: $(\bar{\mathbf{l}}, \bar{\mathbf{u}})$
```

4.5.2 Training Dataset

We now describe the training dataset we used to learn a successful GNN. It is based on the adversarial training dataset used for the AdvGNN method in the previous chapter. We attacked a convolutional neural network we call the 'Base' model. It has been trained robustly on the CIFAR10 dataset [Krizhevsky et al., 2009] using the methods of Madry et al. [2018] against l_∞ perturbations of size up to $\epsilon = 8/255$ (the amount typically considered in empirical works). We created a set of 4515 properties, each a tuple consisting of a natural image ($\mathbf{x}_i$), a true label (y_i), an incorrect target label ($\hat{y}_i$), and an allowed perturbation value (ϵ_i). The perturbation value is uniquely chosen for each tuple: it is large enough so that there exists at least one adversarial example in the infinity norm ball around the natural image that it defines; at the same time it is small enough so that the adversarial examples are hard to find. The 'Base' network classifies all of these images correctly, so $\epsilon_i > 0$ for all training points i.

Before we further describe the training dataset, we remind the reader of the experimental setting that our GNN will be used in: we aim to generate an adversarial example for a given network and image. If the first PGD run manages to find one, we can move on to the next image. However, if the first iteration of the PGD attack was unsuccessful, we want to use the GNN to learn from this and focus our attention to a smaller input domain, from which we run the next PGD attack. We now need to create a training dataset with which we can simulate this experimental setup.

Every data point in the training dataset will consist of 6 elements: We need a natural image ($\mathbf{x}_i$), along with its true class (y_i), an incorrect target class ($\hat{y}_i$) and an allowed perturbation (ϵ_i) to define the adversarial problem (4.3). We further need a valid adversarial example $\mathbf{x}_{PGD,i}$ for the training loss (4.16) and information from a failed PGD attack ($\mathbf{v}_i$) to initialize the feature vectors of the GNN as described above. We take ($\mathbf{x}_i, y_i, \hat{y}_i, \epsilon_i$) from the training dataset used above. We run PGD with random restarts until we succeed in finding an adversarial example $\mathbf{x}_{PGD,i}$; if we cannot find an adversarial example after a given timeout (120s) we skip to the next property. Finally, we need data from an unsuccessful PGD attack. We thus repeatedly run PGD attacks for a fixed number of iterations (100) until one such run is unsuccessful. We then save a summary of this unsuccessful run ($\mathbf{v}_i$) to complete a training sample. If after a given timeout (120s) we still have not managed to run an unsuccessful PGD attack we discard this property as it is too easy and the GNN is not needed to solve it. Once we have generated a big enough training dataset we can start training a GNN by optimizing over (4.18).

4.6 Experiments

We now evaluate the performance of our method by comparing it to various baselines used in the literature. We think of our method as a tool that can be applied to existing attacks, as opposed to being a stand-alone attack. We first show how GNN Attention is able to boost the performance of the PGD attack on the 'Base' model that is has been trained on before showing how it generalizes well to new unseen networks. We then show that GNN Attention can be applied not just to PGD but to a variety of different adversarial attacks it has not seen during training.

4.6.1 Experimental Set-Up

We use a similar experimental setup as in the previous chapter. We run white-box targeted image-dependent adversarial attacks on the CIFAR10 dataset [Krizhevsky et al., 2009]. We run experiments on the new dataset proposed in section (§3.6) as it allows for a more effective comparison of adversarial attacks. As is the case for the training dataset, every data point in the test set is a tuple consisting of a natural image ($\mathbf{x}_i$), a true label (y_i), an incorrect target label ($\hat{y}_i$), and an allowed perturbation value (ϵ_i). The perturbation norm is image dependent to ensure that at least one adversarial example exists in $\mathcal{B}(\mathbf{x}, \epsilon)$, but at the same time it is small enough so that it is difficult to find.

We use the version of the dataset that applies the perturbation before normalization. This way the resulting images, can be interpreted in a more natural way as even the perturbed images are still real, natural images. Unlike in the previous chapter, where we proposed a stand-alone attack that can be directly incorporated into verification algorithms, we now propose a method to boost existing attacks. As our method here is not an independent attack, we will focus on comparing many different attacks and the integration into verification algorithms is a less straight forward application. We thus picked the version of the dataset that seems more natural for comparing adversarial attacks at the expense of it being less compatible with the existing verification dataset.

We report the percentage of properties successfully attacked over time, for the baselines and our method. Both the baseline and our method are implemented in Pytorch [Paszke et al., 2017] and are run under Ubuntu 16.04.4 LTS and on a single Nvidia Titan V.

4.6.2 Baselines

We first compare our method against the standard PGD attack [Madry et al., 2018] with random initializations, against an optimized version of the Iterative Fast Gradient Sign Method (MI-FGSM+) [Dong et al., 2018] and the Carlini Wagner attack (CW) [Carlini and Wagner, 2017]. PGD tries to find an adversarial example by first choosing a starting point $\mathbf{x}^0 \in \mathcal{B}(\mathbf{x}, \epsilon)$ uniformly at random and then running the following update step for 100 iterations:

$$\mathbf{x}^{t+1} = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)} \left(\mathbf{x}^t + \alpha \ \mathrm{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}', y, \hat{y})\right). \tag{4.19}$$

We stop early if we have found an adversarial example, that is if for any t, we get $L(\mathbf{x}^t, y, \hat{y}) > 0$. MI-FGSM uses momentum to boost the performance of I-FGSM. CW aims to find the minimum perturbation for which there exists an adversarial example. We stop early as soon as the perturbation found is smaller than the given ϵ_i. We use the same hyper-parameters as we did in the previous chapter (§3.7.2)

For the second set of experiments we compare against various variations of the Iterative Fast Gradient Sign Method. All the different attack versions we will now summarize aim to minimize the adversarial loss (4.3). As mentioned above, PGD attack (almost identical to the Iterative Fast Gradient Sign Method (I-FGSM)) applies (4.19) repeatedly. Rather than using the signed gradient, we can alternatively just use the actual gradient; we call this the Iterative Fast Gradient Method (I-FGM):

$$\mathbf{x}^{t+1} = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)} \left(\mathbf{x}^t + \alpha \nabla_{\mathbf{x}} L(\mathbf{x}', y, \hat{y})\right). \tag{4.20}$$

Dong et al. [2018] proposed boosting the performance of the I-FGSM with momentum. They iteratively apply the following two update steps:

$$\mathbf{g}^{t+1} \leftarrow \mu \cdot \mathbf{g}^t + \frac{\nabla_x L(\mathbf{x}^t, y, \hat{y})}{\|\nabla_x L(\mathbf{x}^t, y, \hat{y})\|_1} \tag{4.21}$$

$$\mathbf{x}^{t+1} = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)} \left(\mathbf{x}^t + \alpha \cdot sgn(\mathbf{g}^{t+1})\right). \tag{4.22}$$

Here $\mathbf{g}$ is the momentum term. Just as we did for I-FGSM we can also create another attack by using the true gradient rather than the signed gradient. We call

	Base		Wide		Deep	
Method	**Time (s)**	**Timeout (%)**	**Time (s)**	**Timeout (%)**	**Time (s)**	**Timeout (%)**
PGD	87.412	82.995	80.415	75.358	84.349	80.533
MI-FGSM+	40.438	27.145	31.144	20.462	60.578	47.867
C&W	97.385	95.164	96.366	93.729	99.321	97.600
GNN	**9.397**	**5.928**	**15.244**	**11.551**	**43.384**	**32.800**

Table 4.1: We compare average (mean) solving time and the percentage of properties that the methods time out on when using a cut-off time of 100s. Our method outperforms all three baselines on all three models, including the previously unseen 'Wide' and 'Deep' models.

the following method MI-FGM (Iterative Fast Gradient Method with Momentum):

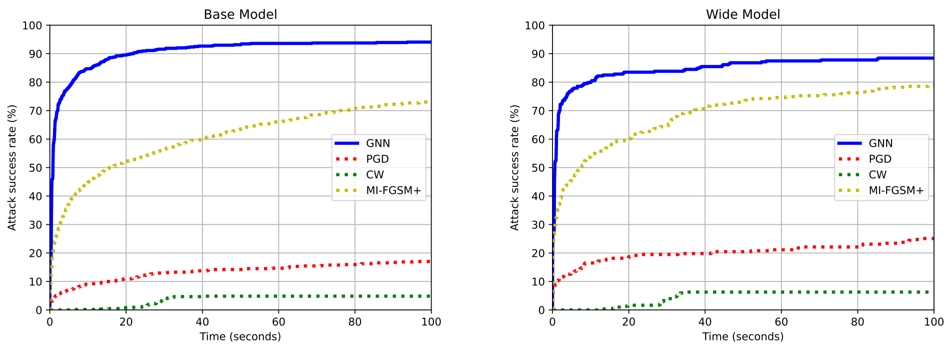
$$\mathbf{g}^{t+1} \leftarrow \mu \cdot \mathbf{g}^t + \frac{\nabla_x L(\mathbf{x}^t, y, \hat{y})}{\|\nabla_x L(\mathbf{x}^t, y, \hat{y})\|_1} \tag{4.23}$$

$$\mathbf{x}^{t+1} = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)} \left(\mathbf{x}^t + \alpha \cdot \mathbf{g}^{t+1})\right). \tag{4.24}$$

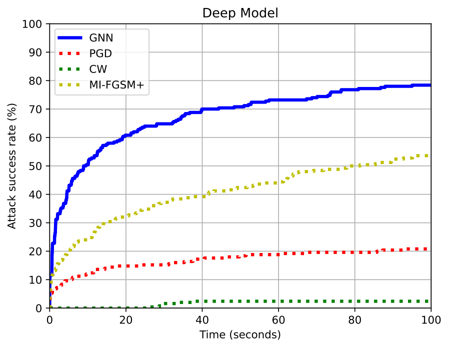
We create two further attacks MI-FGSM' and MI-FGM' by removing the normalizing term in equations (4.21) and (4.23) respectively. Finally, we create another attack that optimizes the adversarial loss using the adam optimizer Kingma and Ba [2015]. We call this attack PGD-Adam. It is the attack that is uses with the GNN throughout this thesis unless otherwise specified.

4.6.3 Our Method

Our method consists of three parts. We first run PGD with no restarts with the same hyper-parameters as above. If the attack has been successful we move on to the next image. If unsuccessful we then use the GNN, that has been trained as described in the previous section. The GNN takes as input the current bounds, the image we are trying to attack and data from the unsuccessful PGD attack, to output new, tighter bounds. We perform two iterations of the forward-backward message passing procedure described above as the embedding vectors tend to converge after two passes. We then run further attacks on using the new tighter bounds. For a more detailed description of our method see Algorithm 4 in section §4.4.5.



(a) Base Model (b) Wide Model



(c) Deep Model

Figure 4.4: Cactus plots for experiments on the 'Base' model (a), and the previously unseen 'Wide' model (b) and the 'Deep' model (c). For each, we compare the different attacks by plotting the percentage of successfully attacked images as a function of runtime. Baselines methods including PGD, MI-FGSM+, and CW are represented by dotted lines. Our GNN based attention method beats all baselines not only on the 'Base' model that is has been trained on, but also on the previously unseen 'Wide' and 'Deep' models thus showing good generalization performance.

4.6.4 Results

We compare our method against the baselines in Figure 4.4 and Table 4.1. We first run experiments on the 'Base' model, the same network that our GNN has been trained on. Our method significantly boosts the performance of the standard PGD attack. We increase the number of properties successfully attacked by over 75% while decreasing the average time taken to do so by 75%.

Generalization to Unseen Networks We then show that the trained GNN generalizes well to unseen networks by running it on the 'Wide' and the 'Deep' model.

	Time (s)		Timeout (%)	
Attack	**Baseline**	**GNN**	**Baseline**	**GNN**
PGD-Adam	16	**3.2**	9	**1**
I-FGM	97	**70.1**	97	**66**
I-FGSM	67.2	**16.2**	60.8	**10**
MI-FGM'	94.8	**51**	94.8	**47.6**
MI-FGM	98.8	**78.6**	98.8	**76.8**
MI-FGSM'	13.3	**4.4**	7.2	**2**
MI-FGSM	13	**4.4**	5.6	**1.6**

Table 4.2: We compare different attacks with and without the GNN Attention on the 'Base' model. We compare average (mean) solving time and the percentage of properties that the methods time out on when using a cut-off time of 100s. GNN Attention improves all seven attacks significantly despite having been trained on PGD-Adam only.

On both of these networks our method significantly outperforms all baselines. The GNN generalizing well to unseen networks is important as it stop us from having to retrain a GNN whenever we want to use it on different networks.

Generalization to Unseen Attacks. We now show how a GNN that has been trained with one attack generalizes well to many different unseen attacks without the need of retraining or even fine-tuning. We run experiments on the 'Base' model comparing seven different attacks. For each attack we compare the performance with and without using GNN Attention. As seen in Figure 4.5 and Table 4.2 the same GNN that has only been trained using PGD-Adam improves every single attack considerably. We set $\alpha = 0.1$ for all attacks, and $\mu = 0.5$ where required. These hyper-parameters are not necessarily all optimal. However, as we use the same hyper-parameters for the baseline and when combining it with GNN Attention, we get a fair experiments regardless. The GNN shows very good generalization performance on all unseen attacks: it improves both attacks that are largely unsuccessful (MI-FGM, I-FGM, MI-FGM'), as well as attacks that perform relatively well already (PGD-Adam, MI-FGSM, MI-FGSM').

One downside of our method compared to using random initializations is the one-off cost associated with training the GNN. However, we argue that in most applications, such as verification or adversarial training, we do not just call the

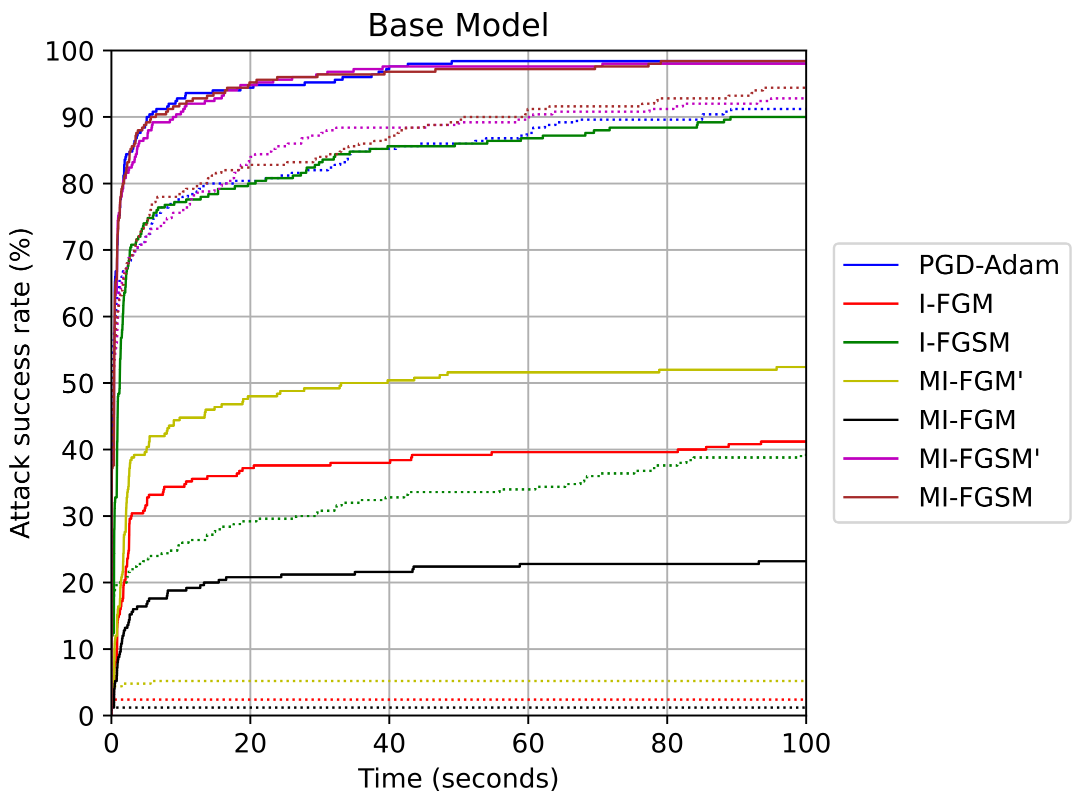


Figure 4.5: We run experiments with different adversarial attacks on the 'Base' model. Each colour presents a different attack: the plain attacks (baselines) are plotted in dotted lines and the attacks combined with the GNN attention are shown in solid lines. The same GNN that has only been trained with the PGD-Adam attack is able to improve all other attacks as well, thus showing good generalization performance to unseen attacks.

adversarial attack once, but a large number of times. The improved performance of the GNN thus makes up for the one time training cost. The GNN's ability to generalize well to unseen networks as well as to different adversarial attacks, reduces the need for training new GNNs.

Generalization to Different Dataset. We now run experiments on the version of the adversarial dataset that applies the perturbation norm (ϵ) before normalization as described in chapter §3.6. We compare our GNN method combined with PGD-Adam against PGD-Adam without attention. We show that our method outperforms the baseline not only on the 'Base' model which we used for training the GNN but also on the unseen 'Wide' and 'Deep' models (Figure 4.6 and Table 4.3). This shows that GNN generalizes to different definitions of the robustness problem.

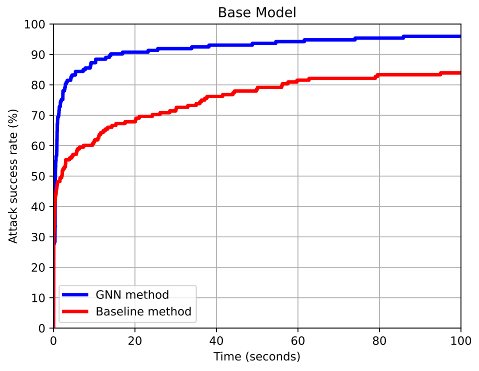
4.7 Conclusion

In this work we have shown how to improve existing adversarial attack methods using a Graph Neural Network as an attention mechanism. Our method learns from previous unsuccessful attacks to greatly decrease the search space for future attacks, thus improving the chances of finding an adversarial example in less time. Our method leads to a 70% decrease in the number of unsuccessful attacks on a published adversarial dataset, compared to using the standard PGD attack with random initializations. Our Method shows good generalization performance to unseen networks and can be combined with different adversarial attacks without the need of retraining or even fine-tuning.

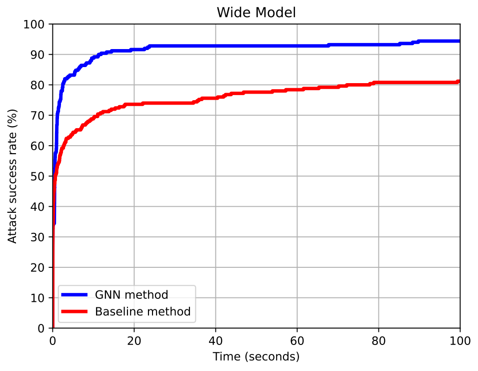
There is potential for future work to build on our method and to extend it. This could include using a similar approach to work for other attack methods that use random restarts such as the the Carlini Wagner attack [Carlini and Wagner, 2017], or autoattack [Croce and Hein, 2020]. One could also combine it with other attack methods that use learning to output adversarial examples such as AdvGAN [Poursaeed et al., 2018], ATN [Baluja and Fischer, 2017], or AdvGNN. Moreover, the GNN based approach could be extended to work on larger or deeper neural networks, or those containing residual connections.

	Base		Wide		Deep	
Method	**Time (s)**	**Timeout (%)**	**Time (s)**	**Timeout (%)**	**Time (s)**	**Timeout (%)**
PGD	24.627	16.071	24.215	18.8	35.276	25.108
GNN	**7.969**	**4.046**	**8.505**	**5.6**	**30.9**	**22.07**

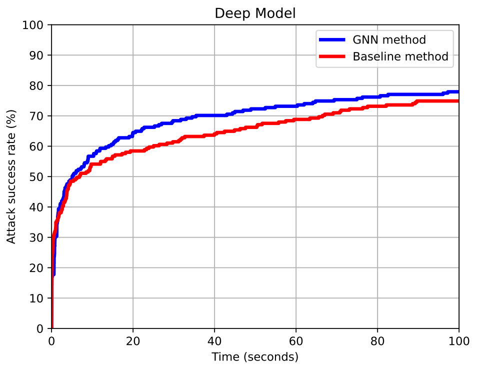
Table 4.3: We run experiments on the version of the adversarial dataset that applies the perturbation before normalization. We compare average (mean) solving time and the percentage of properties that the methods time out on when using a cut-off time of 100s. For the timed out properties we set the solving time 100s to compute the average time taken. The GNN manages to outperform the baseline on all three models both in terms of average time taken and number of properties successfully attacked.



(a) Base Model



(b) Wide Model



(c) Deep Model

Figure 4.6: Perturbation applied before normalization. Cactus plots for experiments on the 'Base' model (left), as well as the previously unseen 'Wide' model (middle) and the 'Deep' model (right). The properties are taken from the new dataset described above. For each experiment, we compare the different attacks by plotting the percentage of successfully attacked images as a function of runtime. PGD-Adam with attention (blue curve) outperforms PGD-Adam without attention (red curve) on all three models for any chosen timeout value.

5 Neural Network Branch-and-Bound for Neural Network Verification

Contents

5.1 Introduction

Despite their outstanding performances on various tasks, neural networks are found to be vulnerable to adversarial examples [Goodfellow et al., 2015, Szegedy et al., 2013] — examples that are similar to real inputs but ones which the neural network misclassifies with a high probability. They are obtained by applying small but deliberately chosen perturbations that are often imperceptible to the human eye. The brittleness of neural networks can have costly consequences in areas such as autonomous vehicles [Bojarski et al., 2016] and personalized medicine [Weiss et al., 2012]. When one requires robustness to adversarial examples, traditional model evaluation approaches, which test the trained model on a hold-out set, do not suffice. Instead, formal verification of properties such as adversarial robustness becomes necessary. For instance, to ensure self-driving cars make consistent correct decisions even when the input image is slightly perturbed, the required property to verify is that the underlying neural network outputs the same correct prediction for all points within a norm ball whose radius is determined by the maximum perturbation allowed.

Several methods have been proposed for verifying properties on neural networks. Bunel et al. [2018a] showed that many of the available methods can be viewed as instances of a unified Branch-and-Bound (BaB) framework. The BaB framework solves a mixed integer programming (MIP) formulation of the verification problem. In other words, the verification problem is formulated as the minimization of a linear objective under linear constraints over variables that can take real values or can be restricted to take only integral values. A BaB algorithm consists of two key components: branching strategies and bounding methods. Branching strategies decide how the feasible domain of the MIP is recursively split into smaller subdomains. For each subdomain the bounding method then computes an upper and a lower bound of the MIP objective. If the upper bound of a subdomain is less than the lower bound of another, the latter subdomain can be pruned thereby reducing the domain for the optimal solution. In this chapter we focus on improving the bounding method as it makes up the bottleneck of the Branch-and-Bound verification algorithm.

The bounding component of the BaB algorithm consists of two parts: finding upper and lower bounds. The former is efficient to compute as it involves evaluating the objective for any feasible solution. In contrast, the lower bound computation requires solving a large convex relaxation. Typically, the relaxation is solved using either commercial solvers such as Gurobi [Gurobi Optimization, 2020] or traditional optimization algorithms such as subgradient descent [Dvijotham et al., 2018b] or proximal minimization [Bunel et al., 2020a]. However, neither approach scales elegantly with the size of the relaxation, which prevents current formal verification methods from being applied to deep state-of-the-art networks. In other words, lower bound estimation forms a computational bottleneck for BaB. A natural question that arises is why do traditional algorithms fail? We argue that by their very nature, they ignore the rich inherent structure of lower bound estimation for the problem of verification. Specifically, all lower bounds that one wishes to estimate across multiple subdomains of the same property, across multiple properties of the same network, and across multiple networks share the same form of variables, constraints and objectives. Furthermore, the coefficients of the objective and constraints are determined from network weights, which themselves are not random but are estimated using real data. Traditional optimization algorithms are agnostic to this complex high-dimensional structure as it is not “visible” to human intelligence.

The aforementioned arguments suggest that, in order to scale-up verification, we require computationally cheap methods that can exploit the inherent structure of the problem and the data. To this end, we propose a novel machine learning framework that can be used for the estimation of lower bounds. Moreover, we use the same framework to return tighter lower bounds more quickly than other optimization methods. Specifically, we make the following contributions:

- We use a graph neural network (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.

- When training the GNN, we provide ways to generate training data cheaply but inclusive enough to represent problems at different stages of a BaB process for various verification properties. With the ability to exploit the neural network structure and a comprehensive training data set, our GNNs are easy to train and converge quickly.
- Our learnt GNNs also enjoy transferability both horizontally and vertically. Horizontally, although trained with easy properties, the learnt GNNs give similar performance on medium and difficult level properties. More importantly, vertically, given that all other parts of BaB algorithms remain the same, the GNNs trained on small networks perform 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.

Since most available verification methods work on ReLU-based deep neural networks, we focus on neural networks with ReLU activation units in this chapter. However, we point out that our framework is applicable to other neural network architectures using different non-linearities such as sigmoid or the hyperbolic tangent.

5.2 Related Works

Learning has already been used in solving combinatorial optimization problems [Bello et al., 2017a, Dai et al., 2017] and mixed integer linear programs (MILP) [Khalil et al., 2016, Alvarez et al., 2017, Hansknecht et al., 2018, Gasse et al., 2019]. 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. [2016], Alvarez et al. [2017], Hansknecht et al. [2018], and Gasse et al. [2019]

proposed learnt branching strategies for solving MILP with BaB algorithms. These methods imitate the strong branching strategy.

Most bounding methods create relaxations of the original problem. There are a variety of relaxations that have easily computable closed-form solutions such as Interval Bound Propagation [Gowal et al., 2018] or WK, the method introduced by Wong and Kolter [2018]. However, these relaxations tend to be quite loose and therefore lead to bad estimates of the final layer output of a neural network. Hence different linear programming (LP) relaxations were proposed that provide tighter bounds: Planet introduced by Ehlers [2017b], Reluplex by Katz et al. [2017b], or the Anderson relaxation [Anderson et al., 2019b]. However, these often require an iterative solver to optimize them, which tends to not scale well. We will therefore use machine learning approaches to come up with better bounds than the best current iterative methods. However, only limited work has been done on learning the bound computation in the BaB algorithm: Dvijotham et al. [2018a,b] propose several different learnt methods: one that treats every layer separately, and another that uses a simple forward-backward architecture. Gowal et al. [2019] propose a method called predictor-verifier (PVT) that learns a robust training procedure and a verifier simultaneously. However, these methods end up beating Interval Bound propagation, a comparatively weak baseline, by a small margin only. There is thus a large scope for improvement which we explore in our work.

5.3 Background

Formal verification of neural networks refers to the problem of proving or disproving a property over a bounded input domain. Properties are functions of neural network outputs. When a property can be expressed as a Boolean expression over linear forms, we can modify the neural network in a suitable way so that the property can be simplified to checking the sign of the neural network output [Bunel et al., 2018a]. Note that all the properties studied in previous works satisfy this form, thereby allowing us to use the aforementioned simplification. Mathematically, given the modified neural network f, a bounded convex input domain $\mathcal{C}$, formal verification examines the truthfulness of the following statement:

$$\forall \mathbf{x} \in \mathcal{C}, \qquad f(\mathbf{x}) \geq 0. \tag{5.1}$$

If the above statement is true, the property holds. Otherwise, the property does not hold.

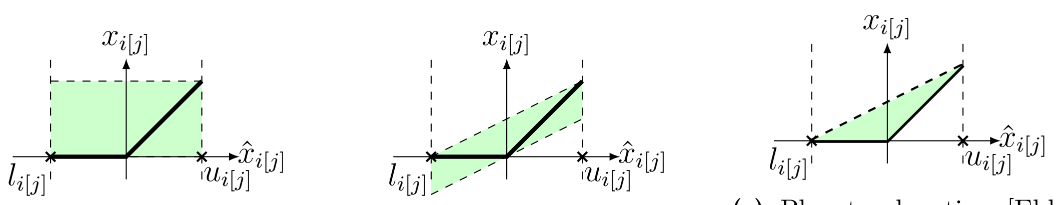
5.3.1 Branch-and-Bound

Verification tasks are often treated as a global optimization problem. We want to find the minimum of $f(\mathbf{x})$ over $\mathcal{C}$ in order to compare it with the threshold 0. Specifically, we consider an L layer feed-forward neural network, $f : \mathbb{R}^d \to \mathbb{R}$, with non-linear activations σ such that for any $\mathbf{x}_0 \in \mathcal{C} \subseteq \mathbb{R}^d$, $f(\mathbf{x}_0) = \hat{\mathbf{x}}_L \in \mathbb{R}$ where

$$\hat{\mathbf{x}}_{i+1} = W^{i+1}\mathbf{x}_i + \mathbf{b}^{i+1}, \qquad \text{for } i = 0, \ldots, L-1, \tag{5.2a}$$

$$\mathbf{x}_i = \sigma(\hat{\mathbf{x}}_i), \qquad \text{for } i = 1, \ldots, L-1. \tag{5.2b}$$

The terms W^i and $\mathbf{b}^i$ refer to the weights and biases of the i-th layer of the neural network f. Domain $\mathcal{C}$ can be an ℓ_p norm ball with radius ϵ. In our case, we use the ReLU activation defined as $\sigma(x) = \max(x, 0)$ as it is widely used in machine learning in general [Krizhevsky et al., 2012, Maas et al., 2013] and in neural network verification in particular [Bunel et al., 2018b, Dvijotham et al., 2018a, Ehlers, 2017b]. However, we note that our approach works for other non-linearities such as the sigmoid activation or the hyperbolic tangent [De Palma et al., 2021]. See Appendix C.2 for a more detailed description of using other non-linearities. Finding the minimum of f is a challenging task, as the optimization problem is generally NP hard [Katz et al., 2017a]. To deal with the inherent difficulty of the optimization problem itself, BaB [Bunel et al., 2018a] is generally adopted. In detail, BaB based methods divide the feasible domain defined by equation (5.2a) and (5.2b) for all $\mathbf{x} \in \mathcal{C}$ into sub-domains, each of which defines a new sub-problem (branching). They then compute a relaxed lower bound of the minimum on each sub-problem (bounding). The minimum of the lower bounds of all the generated



(a) Naive relaxation **(b)** Linear bounds relaxation **(c)** Planet relaxation [Ehlers, 2017a]

Figure 5.1: 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 Weng et al. [2018a]. Finally, Planet relaxation (c) is the tightest linear relaxation among the three considered [Ehlers, 2017a]. Among them, (a) and (b) have closed form solutions which allow fast computations while (c) requires an iterative procedure to obtain an optimal solution. Figure taken from Lu and Kumar [2020a]

sub-domains constitutes a valid global lower bound of the global minimum over $\mathcal{C}$. As a recursive process, BaB keeps partitioning the sub-domains to tighten the global lower bound. The process terminates when the computed global lower bound is above zero (property is true) or when an input with a negative output is found (property is false). A detailed description of the BaB algorithm is provided in the appendices. In what follows, we provide a brief description of the two components, bounding methods and branching strategies, that is necessary for the understanding of our novel learning framework.

5.3.2 Bounding

The bottleneck of most BaB algorithms is the estimation of a lower bound on the output of the neural network we are trying to verify on a given subdomain. As the neural network is highly non-convex and thus hard to optimize over we use convex relaxations. Different linear-sized relaxations have been proposed in the literature such as naive relaxation, linear bounds relaxation, and Planet relaxation. In our work we use the Planet relaxation as it is the tightest relaxation as can be seen in Figure 5.1. Specifically, we focus on the decomposition based approach of Bunel et al. [2020a] which is described in more detail below.

Planet Relaxation. We denote the output of the i-th layer before the application of the ReLU as $\hat{\mathbf{x}}_i$ and the output of applying the ReLU to $\hat{\mathbf{x}}_i$ as $\mathbf{x}_i$. Given the lower bounds $\mathbf{l}_k$ and upper bounds $\mathbf{u}_k$ of the values of $\hat{\mathbf{x}}_i$, we relax the ReLU activations $\mathbf{x}_i = \sigma(\hat{\mathbf{x}}_i)$ to its convex hull $cvx\_hull_\sigma(\hat{\mathbf{z}}_k, \mathbf{x}_k, \mathbf{l}_k, \mathbf{u}_k)$, defined as follows:

$$cvx\_hull_\sigma(\hat{\mathbf{z}}_k, \mathbf{x}_k, \mathbf{l}_k, \mathbf{u}_k) \equiv \begin{cases} x_{i[j]} \geq 0 \quad x_{i[j]} \geq \hat{\mathbf{z}}_{i[j]} & \\ x_{i[j]} \leq \frac{u_{i[j]}(\hat{\mathbf{z}}_{i[j]} - l_{i[j]})}{u_{i[j]} - l_{i[j]}} & \text{if } l_{i[j]} < 0 \text{ and } u_{i[j]} > 0 \\ x_{i[j]} = 0 & \text{if } u_{i[j]} \leq 0 \\ x_{i[j]} = \hat{\mathbf{z}}_{i[j]} & \text{if } l_{i[j]} \geq 0. \end{cases}, \forall j \tag{5.3}$$

Here, $x_{i[j]}$ denotes the j-th element of $\mathbf{x}_i$. Note that the computation of the convex hull requires the knowledge of the lower and upper bounds (i.e. $\mathbf{l}_i$ and $\mathbf{u}_i$) for each intermediate node. The bounds do not have to be optimal. However, the tighter the bounds are, the tighter the relaxation will be as well. There are different ways of computing said bounds that have been proposed in the literature [Gowal et al., 2018, Raghunathan et al., 2018, Wong and Kolter, 2018]. In our experiments we use the method proposed by Wong and Kolter [2018]. For the sake of clarity, we introduce the following notations for the constraints corresponding to the input and the i-th layer respectively:

$$\mathcal{P}_0(\mathbf{x}_0, \hat{\mathbf{x}}_1) \equiv \begin{cases} \mathbf{x}_0 \in C \\ \hat{\mathbf{x}}_1 = W_1 \mathbf{x}_0 + \mathbf{b}_1 \end{cases} \qquad \mathcal{P}_i(\hat{\mathbf{x}}_i, \hat{\mathbf{x}}_{i+1}) \equiv \begin{cases} \exists \mathbf{x}_i \text{ s.t.} \\ \mathbf{l}_k \leq \hat{\mathbf{x}}_i \leq \mathbf{u}_k \\ cvx\_hull_\sigma(\hat{\mathbf{z}}_k, \mathbf{x}_k, \mathbf{l}_k, \mathbf{u}_k) \\ \hat{\mathbf{x}}_{i+1} = W_{i+1} \mathbf{x}_i + \mathbf{b}_{i+1}. \end{cases} \tag{5.4}$$

Using the above notation, the Planet relaxation for computing the lower bound can be written as:

$$\min_{\mathbf{x}, \hat{\mathbf{x}}} \hat{\mathbf{z}}_L \text{ s.t. } \mathcal{P}_0(\mathbf{x}_0, \hat{\mathbf{x}}_1); \mathcal{P}_i(\hat{\mathbf{x}}_i, \hat{\mathbf{x}}_{i+1}) \text{ for } k \in [1, \dots, L-1]. \tag{5.5}$$

Lagrangian Decomposition. We often merely need approximations of the bounds rather than the precise values of them. We can therefore make use of the primal-dual formulation of the problem as every feasible solution to the dual problem provides a valid lower bound for the primal problem. Following the work of Bunel et al. [2020a] we will use the Lagrangian decomposition [Guignard and Kim, 1987]. To this end, we first create two copies $\hat{\mathbf{x}}_{A,i}, \hat{\mathbf{x}}_{B,i}$ of each variable $\hat{\mathbf{x}}_i$:

$$\begin{aligned} \min_{\mathbf{x},\hat{\mathbf{x}}} \hat{\mathbf{x}}_{A,L} \text{ s.t. } & \mathcal{P}_0(\mathbf{x}_0, \hat{\mathbf{z}}_{A,1}); \mathcal{P}_k(\hat{\mathbf{z}}_{B,k}, \hat{\mathbf{z}}_{A,k+1}) && \text{for } k \in [1, \dots, L-1] \\ & \hat{\mathbf{x}}_{A,i} = \hat{\mathbf{x}}_{B,i} && \text{for } k \in [1, \dots, L-1]. \end{aligned} \tag{5.6}$$

Next we obtain the dual by introducing Lagrange multipliers $\boldsymbol{\rho}$ corresponding to the equality constraints of the two copies of each variable:

$$\begin{aligned} q(\boldsymbol{\rho}) = \min_{\mathbf{x},\hat{\mathbf{x}}} \quad & \hat{\mathbf{x}}_{A,n} + \sum_{i=1,\dots,n-1} \boldsymbol{\rho}_i^\top (\hat{\mathbf{x}}_{B,i} - \hat{\mathbf{x}}_{A,i}) \\ \text{s.t.} \quad & \mathcal{P}_0(\mathbf{x}_0, \hat{\mathbf{z}}_{A,1});\ \mathcal{P}_k(\hat{\mathbf{z}}_{B,k}, \hat{\mathbf{z}}_{A,k+1}) \text{ for } k \in [1, \dots, L-1]. \end{aligned} \tag{5.7}$$

Problem (5.7) is unconstrained with respect to $\boldsymbol{\rho}$. In other words, any possible $\boldsymbol{\rho}$ is a feasible solution and thus by duality provides a lower bound for the primal problem (5.6). We therefore aim to maximize $q(\boldsymbol{\rho})$ to get the tightest possible lower bound. Given an assignment to the dual variables, Bunel et al. [2020a] showed that the minimization over $\mathbf{x}_0^*$, $\hat{\mathbf{x}}_A^*$, $\hat{\mathbf{x}}_B^*$ can be done efficiently. The supergradients can then be easily computed as $\nabla_{\boldsymbol{\rho}}(q) = \hat{\mathbf{x}}_B^* - \hat{\mathbf{x}}_A^*$. This is used to come up with lower bounds via supergradient ascent. Unfortunately, supergradient ascent is known to be quite slow. We therefore take a different approach that learns to estimate a better ascent direction, thereby providing larger lower bounds more efficiently.

5.4 Branch-and-Bound Algorithms

The following generic Branch-and-Bound Algorithm is provided in Bunel et al. [2020b]. Given a neural network *net* and a verification property *problem* we wish to verify, the BaB procedure examines the truthfulness of the property through an iterative procedure. During each step of BaB, we first use the *pick_out* function (line 6) to choose a problem *prob* to branch on. The split function (line 7) determines the branching strategy and splits the chosen problem *prob* into sub-problems. We compute output upper and lower bounds on each sub-problem with functions *compute_UB* and *compute_LB* respectively. Newly computed output upper bounds are used to tighten the global upper bound, which allows more sub-problems to be pruned. We prune a sub-problem if its output lower bound is greater than or equal to the global upper bound, so the smaller the global upper bound the better it is. Newly calculated output lower bounds are used to tighten the global lower bound, which is defined as the minimum of the output lower bounds of all remained sub-problems after pruning. We consider the BaB procedure converges when the difference between the global upper bound and the global lower bound is smaller than ϵ.

In our case, our interested verification problem Equation (5.1) is a satisfiability problem. We thus can simplify the BaB procedure by initialising the global upper bound *global_ub* as 0. As a result, we prune all sub-problems whose output lower bounds are above 0. In addition, the BaB procedure is terminated early when a below 0 output upper bound of a sub-problem is obtained, which means a counterexample exits.

Algorithm 6 Branch-and-Bound

```
Input: net, problem
1: global_lb ← compute_LB(net, problem) {global lower bound}
2: global_ub ← compute_UB(net, problem) {global upper bound}
3: probs ← [(global_lb, problem)] {set of all current domains}
4: while probs is not empty do
5:     (_ , prob) ← pick_out(probs) {the pick_out function picks an ambiguous ReLU
       to split on}
6:     [subprob_1, subprob_2] ← split(prob)
7:     for $i = 1, 2$ do
8:         sub_lb ← compute_LB(net, subprob_i)
9:         sub_ub ← compute_UB(net, subprob_i)
10:        if sub_ub $< 0$ then
11:            return SAT {we've found an adversarial example}
12:        end if
13:        if sub_lb $< 0$ then
14:            probs.append((sub_lb, subprob_i))
15:        end if {if sub_lb $> 0$ then the subdomain gets pruned away}
16:    end for
17:    global_lb ← min{lb | (lb, prob) ∈ probs} {If probs is non-empty then
       global_lb is negative}
18: end while
19: return UNSAT {all subproblems have a positive lower bound, therefore global_lb
    is positive}
```

We will now describe the parallelized version of the BaB algorithm for when we use supergradient ascent or the bounding GNN to compute final bounds. Compared to the standard BaB algorithm the number of times the *compute_UB* and *compute_LB* functions are called can be reduced by a factor of *batch_size*, which can lead to a significant speed up as these two functions tend to be the bottleneck of the algorithm.

Algorithm 7 Branch-and-Bound — parallelized version

```
Input: net, problem
 1: global_lb ← compute_LB(net, problem) {global lower bound}
 2: global_ub ← compute_UB(net, problem) {global upper bound}
 3: probs ← [(global_lb, problem)] {set of all current domains}
 4: while probs is not empty do
 5:   $s = \min\{$batch_size/2, len(probs)$\}$
 6:   subproblems = []
 7:   for $i = 1 \dots s$ do
 8:     (_ , prob) ← pick_out(probs) {the pick_out function picks an ambiguous
        ReLU to split on}
 9:     [subprob_i$_1$, subprob_i$_2$] ← split(prob)
10:     subproblems ← subproblems + [subprob_i$_1$, subprob_i$_2$]
11:   end for
12:   [sub_lb$_{1_1}$, sub_lb$_{1_2}$, ..., sub_lb$_{s_1}$, sub_lb$_{s_2}$] ←
      compute_LBs(net, subproblems)
13:   [sub_ub$_{1_1}$, sub_ub$_{1_2}$, ..., sub_ub$_{s_1}$, sub_ub$_{s_2}$] ←
      compute_UBs(net, subproblems)
14:   for $i = 1 \dots s$ do
15:     for $j = 1, 2$ do
16:       if sub_ub$_{i_j} < 0$ then
17:         return SAT {we've found an adversarial example}
18:       end if
19:       if sub_lb$_{i_j} < 0$ then
20:         probs.append((sub_lb$_{i_j}$, subprob_i$_j$))
21:       end if
22:     end for
23:   end for
24:   global_lb ← min{lb | (lb, prob) ∈ probs} {If probs is non-empty then
      global_lb is negative}
25: end while
26: return UNSAT {all subproblems have a positive lower bound, therefore global_lb
    is positive}
```

5.5 GNN Framework

The bounding part of the BaB algorithm aims to estimate the lower bound for the final layer of the neural network. Previously known lower bound computation techniques such as supergradient ascent and proximal maximization [Bunel et al., 2020a] can be thought of as performing forward-backward style passes through the network to update the dual variables. However, the exact form of the passes is restricted to those suggested by standard optimization algorithms, which are agnostic to the special structure of neural lower bound computation. This observation suggests using a GNN framework to parameterize the forward and backward passes, and estimate the parameters using a training data set so as to exploit the problem and data structure more successfully.

Nodes and Edges. A graph neural network G_{GNN} is represented by two components: a set of nodes V_{GNN} and a set of edges E_{GNN}, such that $G_{GNN} = (V_{GNN}, E_{GNN})$. The precise structure of G_{GNN} depends on its input. We treat the neural network f that we're trying to verify as a graph $G_f = (V_f, E_f)$ and provide it as input to the GNN. The set of nodes in the GNN corresponds to the nodes in the original neural network. We create a node $v_{i[j]}$ in our bounding GNN for every dual variable $\rho_{i[j]}$. Every dual variable corresponds to the output of the non-linear activation and the input to the next linear layer. The set E_{GNN} consists of all edges connecting nodes in V_{GNN}, 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 $v_{i[j]}$ and $v_{i+1[k]}$, we assign $e^i_{jk} = W^i_{jk}$.

Node Features. For every node $v \in V_{GNN}$ we first compute a feature vector $\mathbf{f}$, which contains local information about the node and will depend on the task that we want to solve. There is an inherent trade-off between using simple and easily computable features and using more informative but complex ones. Our methods use simple node features and do not rely on extensive feature engineering. Instead we rely on the powerful GNN framework - in particular the forward-backward passes described below - to generate accurate solutions. More specifically, for each node $v_{i[j]}$ we define a corresponding d-dimensional feature vector $\mathbf{f}_k[i] \in \mathbb{R}^d$ describing the current state of that node as follows:

$$\mathbf{f}_k[i] := \left(\rho_{i[j]}, \hat{\mathbf{z}}_{A,i[j]}, \hat{\mathbf{z}}_{B,i[j]}, \hat{\mathbf{z}}_{B,i[j]} - \hat{\mathbf{z}}_{A,i[j]}\right)^\top . \tag{5.8}$$

Here, $\rho_{i[j]}$ is the current assignment to the corresponding dual variable and $\hat{\mathbf{z}}_{A,i[j]}$ and $\hat{\mathbf{z}}_{B,i[j]}$ are the closed-form solutions to the inner minimization problem of the dual problem as explained above. The term $\hat{\mathbf{z}}_{B,i[j]} - \hat{\mathbf{z}}_{A,i[j]}$ corresponds to the supergradient of q. While more complex features could be included, we deliberately chose the simple features described above and rely on the power of GNNs to efficiently compute an accurate ascent direction.

Embeddings. For every node $v_i[j]$ we compute a corresponding p-dimensional embedding vector $\boldsymbol{\mu}_{i[j]} \in \mathbb{R}^p$ using a learnt function $g_{init}(\cdot;\boldsymbol{\theta}_0) : \mathbb{R}^d \to \mathbb{R}^p$ that takes the feature vector as input:

$$\boldsymbol{\mu}_{i[j]} := g_{init}(\mathbf{f}_k[i];\boldsymbol{\theta}_0). \tag{5.9}$$

In our case g_{init} is a multilayer perceptron (MLP).

Forward and Backward Passes. In general, a graph neural network learns signals from a graph by acting as a function of two inputs: a feature matrix consisting of the embedding vectors, and an adjacency matrix representing the graph structure. Under this formulation, 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. As described above, our approach is modelled on existing lower bound computation techniques: in order to pass information between the embedding vectors, we update them using a forward- and backward update schedule simulating runs through the original network. The update steps normally take as input the node feature vectors, the embedding vectors of all neighbouring nodes, as well as the edge values. At the end of the forward-backward updates, embedding vectors capture important information about the corresponding node, the structure of the neural network, and the state of the optimization algorithm. We point out that our forward-backward update scheme does not depend on the underlying neural network structure and thus should be applicable to network architectures that differ from the one we use for training. 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. Specifically, during the forward update, for $i = 1, \dots, L-1$, we have, for all possible j,

$$\boldsymbol{\mu}_{i[j]} \longleftarrow F(\mathbf{f}_k[i], \boldsymbol{\mu}_{i-1}, \boldsymbol{\mu}_{i[j]}, e^i; \boldsymbol{\theta}_1). \tag{5.10}$$

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

$$\boldsymbol{\mu}_{i[j]} \longleftarrow B(\mathbf{f}_k[i], \boldsymbol{\mu}_{i+1}, \boldsymbol{\mu}_{i[j]}, e^{i+1}; \boldsymbol{\theta}_3). \tag{5.11}$$

In other words, both the forward and the backward passes are performed by functions $F(\cdot; \boldsymbol{\theta}_1) : \mathbb{R}^p \mapsto \mathbb{R}^p$ and $B(\cdot; \boldsymbol{\theta}_2) : \mathbb{R}^p \mapsto \mathbb{R}^p$ with parameters $\boldsymbol{\theta}_0$ and $\boldsymbol{\theta}_2$, which can be estimated using a training data set.

Update Step and Output Function. Finally, we need to reduce each p-dimensional embedding vector to a single value to get an ascent direction $\bar{\boldsymbol{\rho}}_i^{t+1}$. We use an output function $g_{out}(\cdot; \boldsymbol{\theta}_3) : \mathbb{R}^p \to \mathbb{R}$ with learnable parameter $\boldsymbol{\theta}_3$ to get a set of dual variables: $\bar{\boldsymbol{\rho}}_i^{t+1} = g_{out}(\boldsymbol{\mu}_i; \boldsymbol{\theta}_3)$. Ideally the GNN would output a new ascent direction that will lead us directly to the global optimum of equation (5.7). However, as the dual problem is complex this may not be feasible in practice without making the GNN very large, thereby resulting in computationally prohibitive inference. Instead, we propose to run the GNN a small number of times to return ascent directions that gradually move towards the optimum. Given a step size η^{t+1}, previous dual variables $\boldsymbol{\rho}^t$, and the new ascent direction $\bar{\boldsymbol{\rho}}^{t+1}$ we update the dual variables as follows:

$$\boldsymbol{\rho}^{t+1} = \boldsymbol{\rho}^t + \eta^{t+1} \bar{\boldsymbol{\rho}}^{t+1}. \tag{5.12}$$

Similar to many iterative optimization methods we decay our stepsize as we want to take smaller steps the closer we get to the optimal solution. Given an initial step size η_0, we define the step size at time t as follows: $\eta^t = \eta_0 * \sqrt{t}$.

5.6 Parameter Estimation

Training. We now describe how to train the learnable parameters for the bounding GNN. We aim to maximize the dual values returned by the bounding GNN method. The dual value directly corresponds to the final layer lower bound for a given subdomain. In particular, if the dual value is strictly positive, then the corresponding lower bound is greater than zero as well and we can prune away that subdomain thereby reducing the size of the BaB tree.

Recall that we do not use the GNN to directly compute the optimum dual solution. Instead, we run it iteratively, where each iteration computes an update direction for the dual variables. In order for the training procedure to more closely resemble its behaviour at inference time, it is crucial to train the GNN using a loss function that takes into account the dual values across a large number of iterations K. In order to ensure that a single training sample does not dominate the loss by reaching a large positive value, we truncate the loss values for each sample. The natural point to clamp the individual losses at is the value returned by supergradient ascent (q^i_{SupG}) plus a small positive threshold κ. Inference time of our GNN is shorter than supergradient ascent as we run it for significantly fewer iterations (100 and 500 respectively); so as long as the duals returned by the GNN are as good as those returned by supergradient ascent, the GNN will outperform the baseline in the BaB setting. Given the i-th training sample d_i, the corresponding dual objective q^i, and the dual variables returned by the bounding GNN $\boldsymbol{\rho}^{i,t}_{GNN}$, we define its loss to be:

$$\mathcal{L}_i = -\sum_{t=1}^{K} q^i(\boldsymbol{\rho}^{i,t}_{GNN}) * \gamma^t * \mathbf{1}_{q^i(\boldsymbol{\rho}^{i,K}_{GNN})<q^i_{SupG}+\kappa}. \tag{5.13}$$

Instead of maximizing over the dual value, we minimize over the negative dual instead. If the decay factor $\gamma \in (0,1)$ is low then we encourage the model to make as much progress in the first few steps as possible, whereas if γ is closer to 1, then more emphasis is placed on the final output of the GNN, sacrificing progress in the early stages. Readers familiar with reinforcement learning may be reminded by the discount rates used in algorithms such as Q-learning and policy-gradient methods. We sum over the individual loss values corresponding to each data point to get the final training objective $\mathcal{L}$: $\mathcal{L} = \sum_{i=1}^{|D|} \mathcal{L}_i$.

Fail-safe Strategy. As our approach is based on learning, we do not have any convergence guarantees. For a few subdomains our GNN might diverge rather than improve on the value returned for its parent. We therefore introduce a fail-safe strategy that ensures our algorithm performs well even when our GNN fails. We compare whether the final bound of a given subdomain outputted by the GNN beats the bound returned for its parent domain by a given absolute threshold. If it fails to do so, then we add the subdomain into a second set of current subdomains. We use supergradient ascent to solve these subdomains on which our GNN performed poorly. This way we reduce the risk of our Branch-and-Bound algorithm timing out on certain properties.

Running Standard Algorithms using the GNN. As mentioned earlier, the motivation behind our GNN framework is to offer a parameterized generalization of previous methods for lower bound computation. We now formalize the generalization using the following proposition.

Proposition 2. *Our GNN architecture can simulate supergradient ascent [Bunel et al., 2020a].*

Proof. We show that our method is strictly more expressive than supergradient ascent by showing that it can simulate it exactly.

The supergradient ascent step is equivalent to update step $\boldsymbol{\rho}^{t+1} = \boldsymbol{\rho}^t + \eta^{t+1}\hat{\boldsymbol{\rho}}^{t+1}$, where

$$\hat{\boldsymbol{\rho}}_k^{t+1} = \hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}. \tag{5.14}$$

Let $p = 2$, $q = 4$, and $\Theta_0^{init} \in \mathbb{R}^{2\times 4}$ be the zero-matrix with non-zero elements $\Theta_0^{init}[1,4] = 1$, $\Theta_0^{init}[2,4] = -1$. Moreover, setting $T_1 = 1$, $\Theta_1^{init} = \mathbb{1}$ and $\mathbf{b}_0 = \mathbf{b}_1 = \mathbf{0}$, we get

$$\boldsymbol{\mu}_k^0 = \Theta_0 \cdot \mathbf{f}_k + b_0 = \left(\hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}, -\hat{\mathbf{z}}_{B,k} + \hat{\mathbf{z}}_{A,k}\right)^\top, \tag{5.15}$$

$$\boldsymbol{\mu}_k = \Theta_1 \cdot \text{relu}(\boldsymbol{\mu}_k^0) + \mathbf{b}_1 = \Big(\left(\hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}\right)_+, -\left(\hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}\right)_-\Big)^\top. \tag{5.16}$$

If we set $\Theta_2^{for} = \Theta_3^{for} = \Theta_2^{back} = \Theta_3^{back} = \mathbf{0}$ and $\Theta_1^{for} = \Theta_1^{back} = \mathbb{1}$, then the forward and backward passes do not change the embedding vector (see Appendix C.1 for a more detailed definition of the forward and backward passes). We now just need to set $\boldsymbol{\Theta}^{out} = (1, -1)^\top$ to get the final ascent direction:

$$\hat{\boldsymbol{\rho}}_k^{t+1} = \left(\hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}\right)_+ + \left(\hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}\right)_- = \hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}. \tag{5.17}$$

We have shown that we can simulate supergradient ascent using our GNN architecture. □

5.7 Experiments

We now validate the effectiveness of our proposed framework through comparative experiments against other available neural network verification methods. A comprehensive study of neural network verification methods has been done in Bunel et al. [2020b]. We thus design our experiments based on the results presented in their work. We start by describing the verification dataset proposed by Lu and Kumar [2020a], which consists of properties of varying levels of difficulty defined on three different network architectures. Next, we summarize the training dataset used for our GNN. We then show that our GNN method outperforms conventional lower bounding techniques on the previously described verification dataset. Finally, we combine our new bounding method with a machine learning based branching strategy to create a learnt Branch-and-Bound framework that beats the individual GNN methods as well as other state-of-the-art verification methods.

5.7.1 OVAL Verification Dataset

Many existing datasets consist of a neural network along with a set of inputs and a constant epsilon value [Balunovic and Vechev, 2020, Katz et al., 2017b, 2019]. However, this results in a lot of properties being either SAT, meaning that a counter-example exists, or they are very easily verifiable. In both cases the property does not enable us to compare the lower bounding part of different verification methods in a meaningful way. In the former case the run-time and outcome are determined by the sub-routine that generates adversarial examples rather than the one that generates lower bounds, and in the latter case all verification methods verify the property very quickly. To alleviate this inefficiency, Lu and Kumar [2020a] provide a dataset where the allowed input perturbation is uniquely determined for every image in the dataset. This ensures that all properties can be verified but doing so is challenging enough that it highlights the difference in performance between different methods —they call it the OVAL verification dataset.

The OVAL benchmark consists of sets of adversarial robustness properties specifically designed for three adversarially trained CIFAR-10 convolutional neural networks with ReLU activations. Two networks are composed of 2 convolutional layers followed by 2 fully connected layers: a 'Base' model, and a wider 'Wide' model. A 'Deep' model has 2 additional convolutional layers, with a width analogous to the 'Base' model. All three models are trained robustly using the method introduced by Wong and Kolter [2018] to achieve robustness against l_∞ perturbations of size up to $\epsilon = 8/255$ (the amount typically considered in empirical works). They run their experiments on adversarially trained models, as standard trained networks are not robust for most images, and hence not suited as well for comparing verification methods.

Finally, Lu and Kumar [2020a] consider the following verification properties. Given an image $\mathbf{x}$ for which the model correctly predicted the label y_c, they randomly choose a label $y_{c'}$ such that for a given ϵ, they want to prove $(\boldsymbol{e}^{(c)} - \boldsymbol{e}^{(c')})^T f'(\mathbf{x}') > 0, \forall \mathbf{x}'$ s.t $\|\mathbf{x} - \mathbf{x}'\|_\infty \leq \epsilon$. Here, f' is the original neural network, $\boldsymbol{e}^{(c)}$ and $\boldsymbol{e}^{(c')}$ are one-hot encoding vectors for labels y_c and $y_{c'}$. They want to verify that for a given ϵ, the trained network will not make a mistake by labelling the image as $y_{c'}$. Since BaBSR (Branch-and-Bound with Smart branching on ReLUs, [Bunel et al., 2020b]) is claimed to be the best performing method on convolutional networks, they 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 counter-examples. The most challenging ϵ values are those at which a large number of activation units are ambiguous. They use binary search with BaBSR method to find the largest ϵ values possible that result in true properties. They further include a few timed out properties, that with a high probability are true properties to make the dataset even more challenging which might be beneficial when comparing stronger verification methods in the future. The 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.

Properties are generated for the 'Base' model using binary search with BaBSR and a 3600s timeout. They categorise verification properties solved by BaBSR within 800s as easy, between 800s and 2400s as medium and more than 2400s as hard. In total, they generated 467 easy properties, 773 medium properties and 426 hard properties. They use a larger timeout of 7200s to generate 300 properties for the 'Wide' model and 250 properties for the 'Deep' model.

5.7.2 Training Dataset

We would like to train the GNN on the same samples that we will encounter during inference time. However, that is impossible as the structure and the elements of the BaB tree computed at test time depend on the lower bound computation and thus on the GNN. To resolve that problem we dynamically create a training dataset as follows. We first pick 100 images from the training dataset used by Lu and Kumar [2020a] together with the corresponding properties that we are verifying our network against and epsilon values defining the input domain. All of the training properties used are easy; that is BaBSR takes less than 800 seconds to solve them. We then create the first part of the training dataset by running a complete BaB algorithm on these properties using the supergradient method. We record the intermediate bounds and parent dual variables for each subdomain visited to create a dataset to train a first GNN on. Once we have finished training the first version of the GNN we extend the dataset by running another complete BaB algorithm on the same properties; this time using the first version of the GNN instead of supergradient ascent to compute the lower bounds. We subsequently resume training the first GNN on the extended dataset for a fixed number of epochs to get a second GNN. We then repeat this process of extending the dataset and further training the GNN three more times. For most properties in the training dataset we acquire a large number of samples over the different iterations. To speed up training, we reduce the proportion of the training dataset on which we train our GNNs by only picking a small subset of the samples for each property. We make sure to pick subdomains from different stages of the BaB algorithm in order to get a more diverse training dataset. More specifically, for each iteration we train the GNN on 10,000 subdomains for a total of 50 epochs. At the start of each of the three iterations, we randomly select the subdomains to train on, choosing the same number of subdomains for each property. We train the GNN by minimizing the loss function (5.13) using a horizon of 100 and a decay factor $\gamma = 0.99$. We train the GNN using the Adam optimizer Kingma and Ba [2015] with a learning rate of $1e^{-2}$ and no weight decay; we manually decay the learning rate by a factor of 10 if the loss function doesn't improve for two consecutive epochs. For the update step we use an initial step size of $\mu = 1e^{-3}$, and decay it as explained above. We set the embedding size to be 32 for all GNNs. Moreover, we set $T_1 = 1$ and $T_2 = 1$. That is, we use a 2-layer MLP to initialize the embedding vectors and perform just one set of forward and backward passes. At the beginning of the first iteration we create the dataset by running the BaB algorithm using supergradient ascent and Adam to compute the lower bounds; we set the learning rate to be $1e^{-4}$. For the second and third iterations we further extend the dataset, this time using the current version of the GNN to compute the final lower bounds.

The total training time for the GNN, including the generation of the training dataset is 72 hours. Similar to the branching GNN used by Lu and Kumar [2020a] there is an additional cost associated with our approach, that is training a GNN on a similar structure that we wish to verify. However, we argue that the significantly improved performance compared to state-of-the-art attack methods shown in various experiments below far outweighs this. In most use cases we want to repeatedly verify a network, or run lower bounding methods as a subroutine to calculate robustness. The more often we run our method, the more the improved performance of the GNN that is not just quicker but also more powerful than other verification methods makes up for the one-time cost.

5.7.3 Results

We will now show the practical effectiveness of our bounding GNN model by comparing its performance with several state-of-the-art bounding methods.

Baselines. We compare our work to the baselines used in both Bunel et al. [2018b] and Lu and Kumar [2020b]. We use MIPPlanet, which is a mixed integer solver by the commercial solver GUROBI, and BaBSR, a BaB based method that uses an LP solver by GUROBI to compute bounds for the subdomains. We also compare against supergradient ascent together with Adam as proposed by Bunel et al. [2020a] (for a more detailed description see Algorithm 8). We run supergradient ascent for 500 steps with a learning rate of 1e-4; both of these hyper-parameters have been optimized over the validation dataset. Note that whereas BaBSR aims to solve the subdomains to optimality, the supergradient and bounding GNN methods aim to trade accuracy for speed. As they return estimated bounds faster they lead to a quicker BaB algorithm despite generating more subdomains.

Algorithm 8 Supergradient method

Input: A trained network $f : \mathbb{R}^d \mapsto \mathbb{R}^m$, and a set of intermediate bounds $\{\mathbf{l}_k, \mathbf{u}_k\}_{k=1..n}$
Parameters: Parameters for the Adam Optimizer
Output: A dual value $q(\boldsymbol{\rho})$

```
1: Initialise dual variables $\boldsymbol{\rho}^0$ using the duals of the parent domain or the algo of
   Wong and Kolter [2018]
2: for nb_iterations do
3:     $\hat{\mathbf{z}}^*$, $\hat{\mathbf{z}}^*_A\, \hat{\mathbf{z}}^*_B \leftarrow$ inner minimization as proposed by Bunel et al. [2020a]
4:     Compute supergradient using $\nabla_{\boldsymbol{\rho}} q(\boldsymbol{\rho}^t) = \hat{\mathbf{z}}^*_B - \hat{\mathbf{z}}^*_A$
5:     $\boldsymbol{\rho}^{t+1} \leftarrow$ Adam's update rule Kingma and Ba [2015]
6: end for
7: return $q(\boldsymbol{\rho})$
```

To speed up the BaB algorithm we parallelize over the lower bound computation for the different subdomains (see Algorithm 7). We run the bounding GNN with a batch-size of 200 and the supergradient ascent method with the highest batch-size possible for all experiments: 1600 on the 'Base' model, 900 on the 'Deep', and 350 on the 'Wide' model. For the baseline, the maximum possible batch-size for a given memory constraint depends on the size of the model we are trying to verify, and the computation of the intermediate bounds, as well as the gradient. The actual update step is more memory efficient and thus does not influence the batch-size.

Our Method The bounding GNN method uses the hand-designed branching heuristic of BaBSR and the bounding GNN to return final lower bounds. If the embedding size p is small, then the maximum batch size is the same as for supergradient ascent. For larger embedding sizes, the maximum batch size is significantly lower. We use an embedding size of $p = 32$. We run our GNN for 100 steps with an absolute fail-safe threshold of 0.05 for the 'Base' experiment and a threshold of 0.1 for the larger models. This together with the significant reduction in subdomains visited in the BaB algorithm when using the GNN more than compensates for the fact that one iteration of the GNN takes longer than one iteration of supergradient ascent. If the GNN performs poorly on a subdomain and the fail-safe method is used, it is likely to also not do well on the child subdomains.

	Easy		Medium		Hard	
Method	Time (s)	Timeout(%)	Time (s)	Timeout(%)	Time (s)	Timeout(%)
Gurobi BaBSR	550.48	0.00	1374.32	0.00	3129.08	42.41
MIPplanet	1499.35	16.49	2240.92	42.95	2255.53	46.35
Supergradient	57.49	0.21	82.66	**0.00**	872.54	17.41
GNN-Branching	272.69	0.21	592.12	0.39	1573.16	21.65
GNN-Bounding (ours)	46.70	**0.00**	78.85	**0.00**	665.64	11.53
GNN-Combined (ours)	**37.09**	0.21	**57.97**	**0.00**	**367.45**	**5.65**

Table 5.1: We compare average (mean) solving time and the percentage of properties solved for easy, medium, and hard properties on the 'Base' model. The best performing method for each subcategory is highlighted in bold (note that by definition Gurobi BaBSR doesn't time out on easy and med experiments). The combined GNN approach significantly outperforms all other methods and at the same time shows strong generalization performance. The Bounding GNN also outperforms all baselines on all three types of properties.

We therefore use supergradient ascent to solve all subdomains that result from further subdividing the current one. For all experiment we use a batch-size of 300 for both the GNN and supergradient descent. For the base experiments we store all current subdomains in memory because it is quicker; for the deep and wide models we store them as files because the experiments are more memory expensive. We run both the supergradient baseline and the bounding GNN method on a single GPU and 4 CPUs each. One advantage of our method compared to off-the-shelf solvers is precisely that we can run it on GPUs and can therefore use more efficient parallelized implementations of mathematical operations.

Performance on the 'Base' model We compare our method on the verification dataset described in section §5.7.1. The dataset also includes images of the CIFAR10 test set, but uses different images to the training dataset. We first run experiments on the 'easy' part of the verification dataset that has been generated for the the base model. These are properties which GUROBI can solve in less than 800 seconds similar to the properties of the training dataset. Our method leads to an over 70% reduction in average time taken compared to BaBSR and MIPPlanet (Table 5.1) and it times out on significantly fewer properties. Our bounding GNN also outperforms supergradient ascent. It leads to an over 20% reduction in average verification time (Figure 5.2).

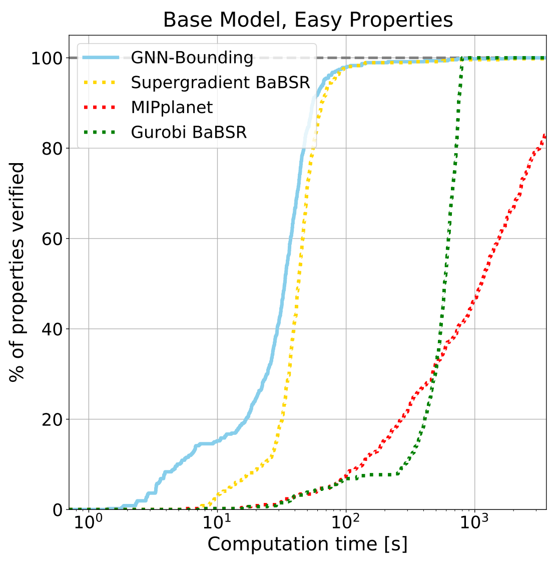
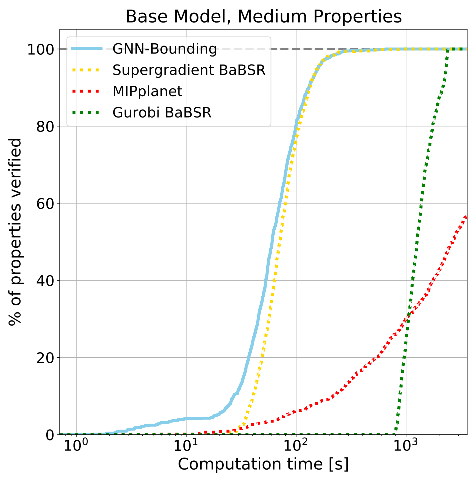
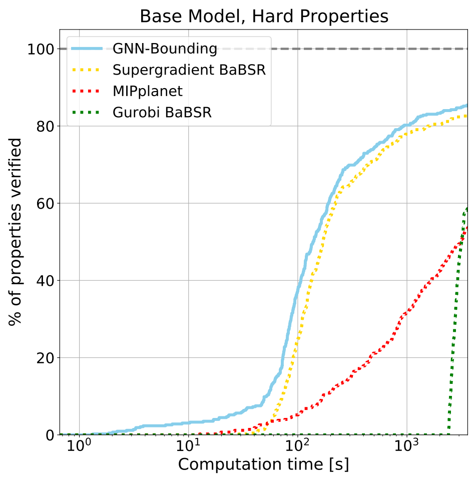


Figure 5.2: Cactus plots for easy properties on the 'Base' model - properties which GUROBI can solve in less than 800 seconds. We compare our bounding methods with other complete verification algorithms by plotting the percentage of solved properties as a function of runtime. Our method is faster than all three baselines.

Generalization to Harder Properties We now run experiments on harder properties on the Base model. We define medium properties to be the ones that GUROBI BaBSR solves between 800 and 2400 seconds and hard properties to be those for which it requires more than 2400 seconds. As can be seen in Figures 5.3a and 5.3b our method outperforms all baselines on both sets of properties. On the medium set of properties our method leads to a over 80% decrease in verification time compared to BaBSR and MIPPlanet and over 15% compared to Supergradient Ascent (Table 5.1). On the hard part of the dataset we get an over 40% reduction in the time taken to verify the properties compared to two of the baselines and get a small performance increase compared to Supergradient Ascent as well. We have thus highlighted that even though the GNN has been trained on easy properties only, it generalizes well to harder ones.



(a)



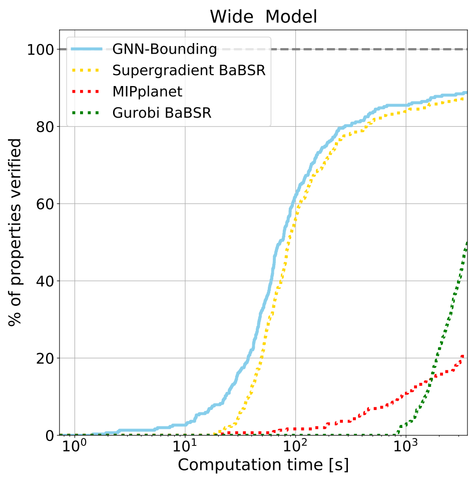
(b)

Figure 5.3: (a) Cactus plots for medium properties on the 'Base' model - properties which GUROBI can solve in between 800 and 2400 seconds. Our method is faster than all three baselines. (b) Experiments on hard properties on the 'Base' model - properties which GUROBI cannot solve in 2400 seconds. GNN-Bounding is the strongest method for any given timeout.

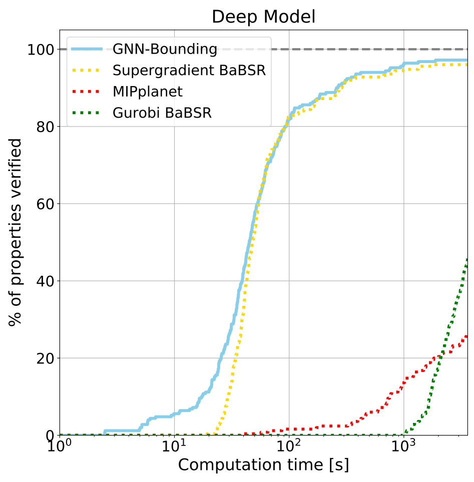
Generalization to Larger Networks We further show the generalization performance of our GNN without the need to perform fine-tuning or online learning by testing it on two larger neural networks. As shown in Figure 5.4 and Table 5.2, the GNN still outperforms all baselines on the unseen networks both in terms of average time taken and the percentage of properties that time out. We improve on MiPPlanet and Gurobi BaBSR on the 'Wide' and 'Deep' models by over 80% and 90%, respectively and reduce the time compared to supergradient ascent by 18% and 11%. These results demonstrate that our GNN based method does not only outperform the baselines on the setup it has been trained for, but also generalizes well to unseen networks and more difficult properties.

	Base		Wide		Deep	
Method	Time (s)	Timeout(%)	Time (s)	Timeout(%)	Time (s)	Timeout(%)
Gurobi BaBSR	1588.02	10.57	2917.95	51.11	3007.24	54.00
MIPplanet	2036.65	36.40	3108.50	79.37	2997.12	73.60
Supergradient	277.22	4.50	624.36	12.87	241.80	4.00
GNN-branching	752.94	5.77	1817.97	21.27	1966.90	20.80
GNN-bounding (ours)	219.62	2.94	513.63	10.23	214.55	2.80
GNN-combined (ours)	**131.11**	**1.50**	**269.11**	**4.95**	**76.10**	**0.40**

Table 5.2: We compare average (mean) solving time, and the percentage of properties that the methods time out on when using a cut-off time of 3600s on the 'Base', 'Wide', and 'Deep' models. The best performing method for each subcategory is highlighted in bold. GNN-Combined significantly outperforms all other methods, timing out on fewer properties and having the lowest average verification time. GNN-Bounding, although slower than GNN-Combined, also has a lower average verification time than all baselines.



(a)



(b)

Figure 5.4: (a) Cactus plots for experiments on the 'Wide' model - a neural network that has the same depth as the 'Base' model but whose convolutional layers are wider. The baselines are displayed with dotted lines. Our method solves more properties in any given time than the baselines. (b) Experiments on the 'Deep' model - a neural network that has the same width as the 'Base' model but has a more layers. GNN-Bounding is fast than all three baselines.

Combine with GNN-based Branching Having shown that our GNN-based bounding method beats various baselines for the estimation of the lower bounds we now combine it with a stronger branching strategy introduced by Lu and Kumar [2020a]. Similar to our work they also use a GNN, in their case to imitate the strong branching strategy. We train a new branching GNN using the bounding

GNN to compute lower bounds during the training procedure instead of the Gurobi based LP solver which the authors use in their paper. We run the combined GNN method on the same number of GPUs and CPUs as the bounding GNN and the supergradient ascent baseline.

As shown in Table 5.1 and Figure 5.5 the combined GNN method outperforms both of the individual GNN methods as well as all other baselines on the easy properties on the 'Base' model. It reduces the average verification time compared to all baselines by more than 35%. Moreover, as highlighted in Figure 5.6 it achieves good horizontal transferability: despite being trained on easy properties only, Combined-GNN performs particularly well on the medium and hard properties. On the medium properties it reduces mean solving time by more than 30% and on the hard dataset it reduces the number of properties timing out by over 65% compared to the best performing baseline.

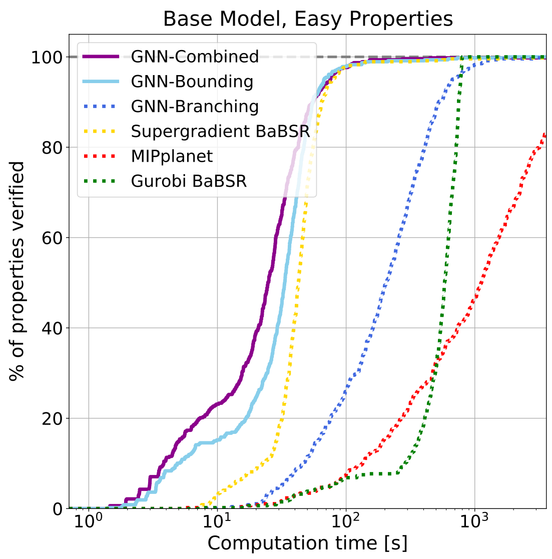
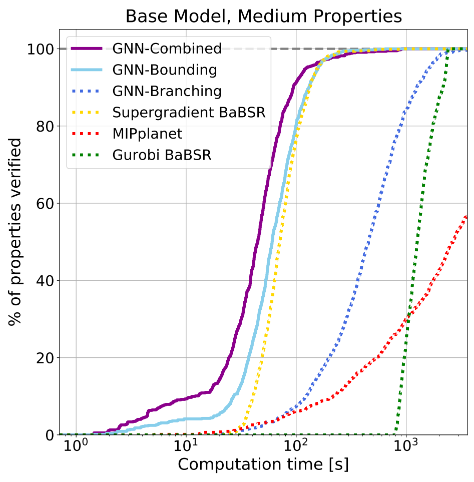
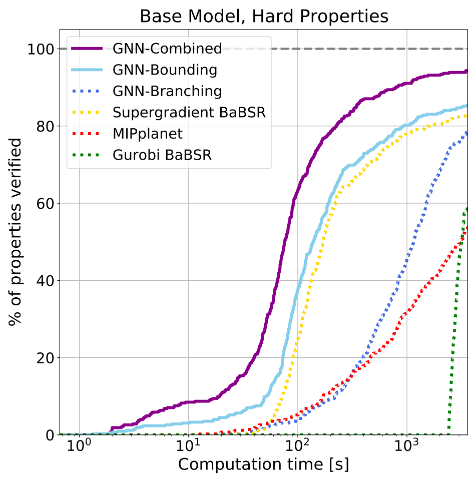


Figure 5.5: Cactus plots for experiments on easy properties on the 'Base' model - properties which GUROBI can solve in less than 800 seconds. GNN-Combined and GNN-Bounding both outperform all four baselines.



(a)

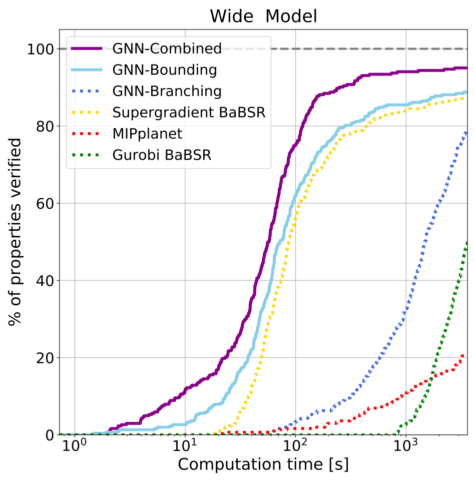


(b)

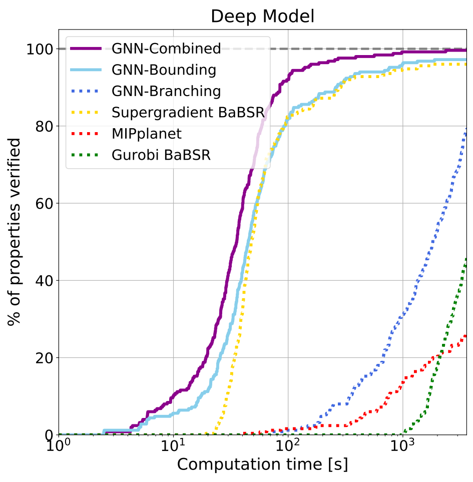
Figure 5.6: (a) Cactus plots for medium properties on the 'Base' model - properties which GUROBI solves in between 800 and 2400 seconds. The Combined GNN method is the fastest followed by the GNN Bounding one. (b) Experiments on hard properties on the 'Base' model - properties which GUROBI cannot solve in 2400 seconds. The combined GNN method manages to verify more properties for any given timeout.

Furthermore, the combined GNN also performs well on the previously unseen 'Wide' and 'Deep' models as shown in Table 5.2 and Figure 5.7. It decreases the number of unsuccessful verification attempts by over 60% and 90% respectively on the two models and reduces the average solving time by over 50% .

Good generalisation performance from easy properties to difficult ones and from small networks to larger ones is beneficial as the complexity of training the GNNs depends on both the difficulty of the training properties and the size of the model. Moreover, it allows us to train a single pair of GNNs and use them for different verification tasks on various different networks.



(a)



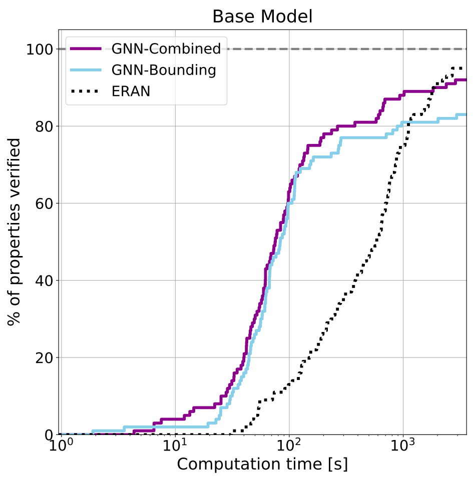
(b)

Figure 5.7: (a) Cactus plots for experiments on the 'Wide' model - a neural network that has the same depth as the 'Base' model but whose convolutional layers are wider. The baselines are displayed with dotted lines. The combined GNN method is significantly faster and stronger than all baselines. (b) Comparisons of the different methods on the 'Deep' model - a neural network that has the same width as the 'Base' model but has more layers. GNN-Combined and GNN-Bounding reduce average verification time compared to the other methods.

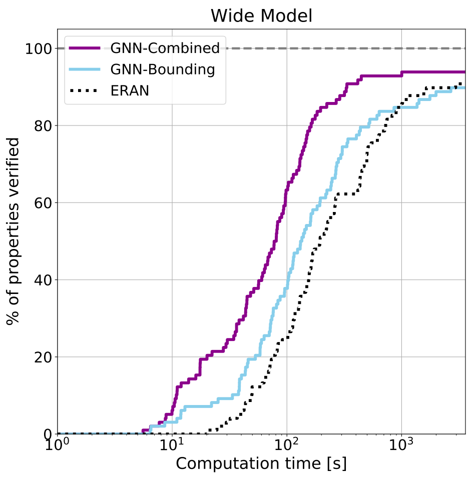
Further Baseline Comparison We will now compare our results against ERAN [Singh et al., 2020], a state-of-the-art complete verification method. We run ERAN as well as both of our methods on a subset of the OVAL dataset used in the VNN-COMP competition [VNN-COMP, 2020]. The results are summarized in Table 5.3 and Figure 5.8). Both the GNN-Bounding and the GNN-Combined methods have a lower average solving time than ERAN on all three models. In particular our combined GNN method leads to a 40%, 50%, and 80% reduction in verification time compared to ERAN on the 'Base', 'Wide' , and 'Deep' models respectively. While ERAN times out on fewer properties on the 'Base' model, the combined GNN method verifies more properties on the 'Wide' model. Neither method times out on any properties on the 'Deep' model.

	Base		Wide		Deep	
Method	Time (s)	Timeout (%)	Time (s)	Timeout (%)	Time (s)	Timeout (%)
ERAN	805.89	**5.00**	635.48	9.18	545.72	**0.00**
GNN-BOUNDING	758.28	17.00	599.50	10.20	285.25	4.00
GNN-COMBINED	**479.61**	8.00	**317.16**	**6.12**	**90.78**	**0.00**

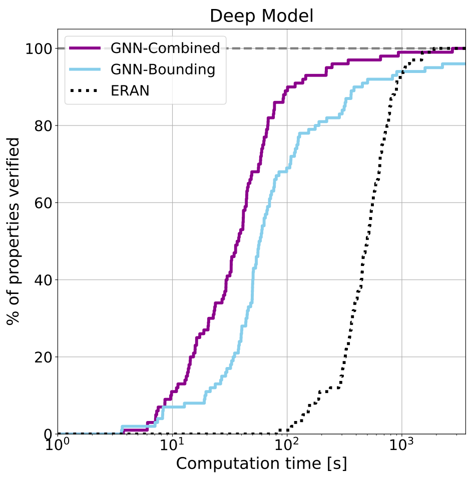
Table 5.3: We run experiments on the Oval Dataset from the VNN-Competition VNN-COMP [2020] and compare our methods against ERAN [Singh et al., 2020], a popular complete verification algorithm. Our methods are significantly faster than ERAN on all three neural networks. ERAN times out on fewer properties on the 'Base' model, whereas the Combined-GNN approach solves more property successfully on the 'Wide' model.



(a)



(b)



(c)

Figure 5.8: We compare our two methods against ERAN, a state-of-the-art complete verification method on the Oval dataset from the VNN-Comp 2020 [VNN-COMP, 2020]. Both the GNN-Bounding and the GNN-Combined methods are significantly faster than ERAN on the 'Base' (a), 'Wide' (b), and 'Deep' models (c).

Constant Epsilon Experiments We will now compare our method against the supergradient ascent method, the strongest baselines used in this thesis on a new dataset with constant perturbation norms. We run both methods on 100 properties on the 'Base' model. We set the epsilon value for all 100 properties to 0.1, 0.15, 0.2, and 0.25, respectively and plot the percentage of properties successfully verified for each experiment. The smaller the perturbation value the more properties are verified by both methods. As seen in Figure 5.9 Our method outperforms the baseline for all four epsilon values demonstrating that the improved performance seen in the experiments above does not depend on whether we use constant or unique ϵ values. We further note that our method generalizes well to perturbation norms that differ from the ones seen at training time.

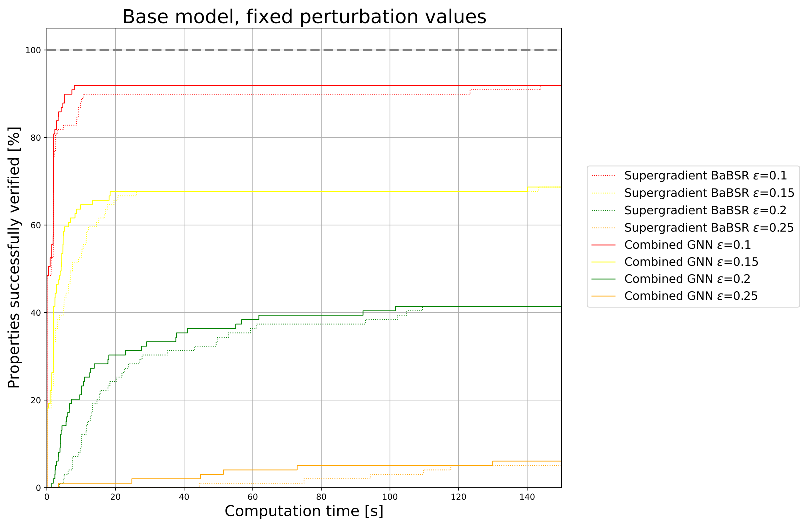


Figure 5.9: Cactus plots for the 'Base' model and for different constant ϵ values. We compare our method against Supergradient BaBSR, the strongest baseline, by plotting the percentage of properties that have been solved for any given time. The combined GNN method outperforms supergradient BaBSR for all four values of epsilon.

5.8 Discussion

We have shown how to improve the complete verification procedure using GNNs that learn how to use the underlying structure of the problem to return better bounds more quickly and to improve branching strategies. We show that our method consistently beats the existing state-of-the-art algorithms. Our GNNs trained on easy properties on a small network show good generalization performance on harder properties and on larger unseen networks. We've taken an important step towards creating verification methods for larger state-of-the-art networks. Further work might include extending our approach to work on different relaxations, such as the one proposed by Anderson et al. [2019b], which is tighter than Planet but has significantly more constraints. Alternatively, one could learn a lazy verifier that only solves subdomains for which there is a high chance of pruning and further divides them into more subdomains otherwise.

6
Summary and Extensions

Contents

6.1 Summary

The work presented in this thesis targets two problems: firstly, generating adversarial attacks more efficiently and successfully, and secondly, improving existing complete verification algorithms for neural network robustness. We summarize the outcomes of each chapter below, before detailing some potential avenues for extensions and future work.

In Chapter 3 we show how we can directly generate adversarial examples more efficiently with the help of a Graph Neural Network. Our method, which we term AdvGNN, combines two types of adversarial attacks: optimization based algorithms and generative methods. During inference, we perform forward-backward passes through the GNN layers to guide an iterative procedure towards adversarial examples. We show that our method beats state-of-the-art adversarial attacks, including PGD-attack, MI-FGSM, and Carlini and Wagner attack, reducing the time required to generate adversarial examples with small perturbation norms by over 65%. Moreover, AdvGNN achieves good generalization performance on unseen networks. Finally, we provided a new challenging dataset specifically designed to allow for a more illustrative comparison of adversarial attacks.

In Chapter 4 we present the use of a GNN as an attention mechanism to boost existing adversarial attacks. Our method, which learns from previous unsuccessful attacks, reduces the search space upon which future iterations of the attack can focus. Using our method, we boost the attack's performance: the GNN increases the success rate of PGD by over 70% , while simultaneously reducing the average computation time by 75%. Moreover, we show that our method generalizes well to unseen networks and can be combined with previously unknown attacks.

In Chapter 5 we focus on improving the bounding method of the Branch-and-Bound algorithm for complete neural network verification. We propose using a GNN to output better ascend directions for the dual solution of the convex relaxation, thereby providing a valid lower bound. We show that our approach provides a significant speed-up for formal verification compared to state-of-the-art solvers and achieves good generalization performance on unseen networks. Furthermore, we demonstrate that our method can be combined with a GNN-based branching strategy, reducing the average verification time by more than 35% and reducing the number of unsuccessful verification attempts by up to 65% compared to various baselines, including GUROBI BaBSR, MIPPlanet, and Supergradient Ascent.

6.2 Future Work

We now propose some directions for future work. As the third and fourth chapter both focus on generating adversarial examples more efficiently, there is significant overlap in the suggestions for future work. First and foremost, one could attempt to combine the two methods, since the Attention mechanism can be used with every iterative attack, such as AdvGNN. Moreover, the GNN based attention mechanism could also be applied to other attack methods that use random restarts such as the the Carlini Wagner attack [Carlini and Wagner, 2017] or the more recent AutoAttack [Croce and Hein, 2020]. One could also combine our Attention method with other attacks that use learning to output adversarial examples such as AdvGAN [Poursaeed et al., 2018], or ATN [Baluja and Fischer, 2017]. Furthermore, both of our GNN based methods can be extended to work on larger neural networks, those containing residual connections, or even those using attention. More work can be done to study the effect of the different components of the Graph Neural Networks on the efficiency of the two methods; this could lead to more efficient implementations or design decisions thus boosting the performance of the attacks. In addition, one might attempt to incorporate either AdvGNN or the Attention mechanism in complete verification, adversarial training methods, or adversarial image detection, as adversarial attacks can form a subroutine of each of them.

There is also scope for future work building on the work presented in Chapter five. In the years since this work was carried out, better Graph Neural Network architectures have been proposed in the literature; they could be implemented to both improve and scale up our method. Moreover, by combining our approach with more efficient methods, which estimate the upper bounds for each subdomain, we can improve our complete verification algorithm upon properties that the network is not robust. In addition, we can also extend our approach to work on different relaxations, such as the one proposed by Anderson et al. [2019a], which is tighter than Planet but has significantly more constraints. We could further study how different intermediate bound computation strategies effect the strength of our GNN. Alternatively, one could learn a lazy verifier that only solves subdomains for which there is a high chance of pruning and further divides them into more subdomains otherwise. One could also try and combine the GNN based framework with other BaB based verification methods that have been proposed in the literature.

Throughout this thesis, we studied neural network robustness to l_p-norm based perturbations; however, in order to apply this work to real-world applications, it would be interesting to study other, alternative natural image perturbations such as rotations, mirroring, or colour space augmentations. Finally, we have only focused on image classification problems in this thesis. However, adversarial examples exists in many different settings, such as the closely related object detection, or image (instance) segmentation problems, or more distant natural language, or video analysis tasks. In the future, we might aim to extend our work to these areas.

Bibliography

Muhammad Aurangzeb Ahmad, Carly Eckert, and Ankur Teredesai. Interpretable machine learning in healthcare. In *Proceedings of the 2018 ACM international conference on bioinformatics, computational biology, and health informatics*, pages 559–560, 2018.

Naveed Akhtar and Ajmal Mian. Threat of adversarial attacks on deep learning in computer vision: A survey. *IEEE Access*, 6:14410–14430, 2018.

Alejandro Marcos Alvarez, Quentin Louveaux, and Louis Wehenkel. A machine learning-based approximation of strong branching. *INFORMS Journal on Computing*, 29(1):185–195, 2017.

Greg Anderson, Shankara Pailoor, Isil Dillig, and Swarat. Chaudhuri. Optimization and abstraction: a synergistic approach for analyzing neural network robustness. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019a.

Ross Anderson, Joey Huchette, Christian Tjandraatmadja, and Juan Pablo Vielma. Strong mixed-integer programming formulations for trained neural networks. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 27–42. Springer, 2019b.

James Atwood and Don Towsley. Diffusion-convolutional neural networks. *Advances in neural information processing systems*, 29, 2016.

Stanley Bak, Changliu Liu, and Taylor Johnson. The second international verification of neural networks competition (vnn-comp 2021): Summary and results. *arXiv preprint arXiv:2109.00498*, 2021.

Pierre Baldi and Gianluca Pollastri. The principled design of large-scale recursive neural network architectures–dag-rnns and the protein structure prediction problem. *The Journal of Machine Learning Research*, 4:575–602, 2003.

Shumeet Baluja and Ian Fischer. Adversarial transformation networks: Learning to generate adversarial examples. *arXiv preprint arXiv:1703.09387*, 2017.

Mislav Balunovic and Martin Vechev. Adversarial training and provable defenses: Bridging the gap. In *International Conference on Learning Representations*, 2020.

Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *The International Conference on Learning Representations Workshop*, 2017a.

Yoshua Bengio, Yann LeCun, et al. Scaling learning algorithms towards ai. *Large-scale kernel machines*, 34(5):1–41, 2007.

Siddhant Bhambri, Sumanyu Muku, Avinash Tulasi, and Arun Balaji Buduru. A survey of black-box adversarial attacks on computer vision models. *arXiv preprint arXiv:1912.01667*, 2019.

Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

Wieland Brendel, Jonas Rauber, and Matthias Bethge. Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. *arXiv preprint arXiv:1712.04248*, 2017.

Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.

Antoine Buetti-Dinh, Vanni Galli, Sören Bellenberg, Olga Ilie, Malte Herold, Stephan Christel, Mariia Boretska, Igor V Pivkin, Paul Wilmes, Wolfgang Sand, et al. Deep neural networks outperform human expert's capacity in characterizing bioleaching bacterial biofilm composition. *Biotechnology Reports*, 22:e00321, 2019.

Rudy Bunel, Ilker Turkaslan, Philip H.S Torr, Pushmeet Kohli, and M. Pawan Kumar. A unified view of piecewise linear neural network verification. *Advances in Neural Information Processing Systems*, pages 4790–4799, 2018a.

Rudy Bunel, Alessandro De Palma, Alban Desmaison, Krishnamurthy Dvijotham, Pushmeet Kohli, Philip Torr, and M Pawan Kumar. Lagrangian decomposition for neural network verification. In *Conference on Uncertainty in Artificial Intelligence*, pages 370–379. PMLR, 2020a.

Rudy Bunel, Jingyue Lu, Ilker Turkaslan, P Kohli, P Torr, and M. Pawan Kumar. Branch and bound for piecewise linear neural network verification. *Journal of Machine Learning Research*, 21(2020), 2020b.

Rudy R Bunel, Ilker Turkaslan, Philip Torr, Pushmeet Kohli, and M Pawan Kumar. A unified view of piecewise linear neural network verification. In *Advances in Neural Information Processing Systems*, pages 4790–4799, 2018b.

Shaosheng Cao, Wei Lu, and Qiongkai Xu. Deep neural networks for learning graph representations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57. IEEE, 2017.

Peter J Carrington, John Scott, and Stanley Wasserman. *Models and methods in social network analysis*, volume 28. Cambridge university press, 2005.

Rich Caruana, Steve Lawrence, and C Giles. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. *Advances in neural information processing systems*, 13, 2000.

Anirban Chakraborty, Manaar Alam, Vishal Dey, Anupam Chattopadhyay, and Debdeep Mukhopadhyay. Adversarial attacks and defences: A survey. *arXiv preprint arXiv:1810.00069*, 2018.

Pin-Yu Chen, Huan Zhang, Yash Sharma, Jinfeng Yi, and Cho-Jui Hsieh. Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In *Proceedings of the 10th ACM workshop on artificial intelligence and security*, pages 15–26, 2017.

Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 257–266, 2019.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3642–3649, 2012. doi: 10.1109/CVPR.2012.6248110.

Christian Collberg, Stephen Kobourov, Jasvir Nagra, Jacob Pitts, and Kevin Wampler. A system for graph-based visualization of the evolution of software. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 77–ff, 2003.

Francesco Croce and Matthias Hein. Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks. In *International conference on machine learning*, pages 2206–2216. PMLR, 2020.

Chunfeng Cui, Kaiqi Zhang, Talgat Daulbaev, Julia Gusak, Ivan Oseledets, and Zheng Zhang. Active subspace of neural networks: Structural analysis and universal attacks. *SIAM Journal on Mathematics of Data Science*, 2(4):1096–1122, 2020.

George E Dahl, Navdeep Jaitly, and Ruslan Salakhutdinov. Multi-task neural networks for qsar predictions. *arXiv preprint arXiv:1406.1231*, 2014.

Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *Conference on Neural Information Processing Systems*, 2017.

Hanjun Dai, Zornitsa Kozareva, Bo Dai, Alex Smola, and Le Song. Learning steady-states of iterative algorithms over graphs. In *International conference on machine learning*, pages 1106–1114. PMLR, 2018.

Alessandro De Palma, Rudy Bunel, Alban Desmaison, Krishnamurthy Dvijotham, Pushmeet Kohli, Philip HS Torr, and M Pawan Kumar. Improved branch and bound for neural network verification via lagrangian decomposition. *arXiv preprint arXiv:2104.06718*, 2021.

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in neural information processing systems*, pages 3844–3852, 2016.

Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Hang Su, Jun Zhu, Xiaolin Hu, and Jianguo Li. Boosting adversarial attacks with momentum. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 9185–9193, 2018.

Yinpeng Dong, Tianyu Pang, Hang Su, and Jun Zhu. Evading defenses to transferable adversarial examples by translation-invariant attacks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4312–4321, 2019.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

Krishnamurthy Dvijotham, Sven Gowal, Robert Stanforth, Relja Arandjelovic, Brendan O'Donoghue, Jonathan Uesato, and Pushmeet Kohli. Training verified learners with learned verifiers. *arXiv preprint arXiv:1805.10265*, 2018a.

Krishnamurthy Dvijotham, Robert Stanforth, Sven Gowal, Timothy A Mann, and Pushmeet Kohli. A dual approach to scalable verification of deep networks. In *Conference on Uncertainty in Artificial Intelligence*, pages 550–559, 2018b.

Ruediger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. *Automated Technology for Verification and Analysis*, 2017a.

Ruediger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 269–286. Springer, 2017b.

Logan Engstrom, Brandon Tran, Dimitris Tsipras, Ludwig Schmidt, and Aleksander Madry. Exploring the landscape of spatial robustness. In *International Conference on Machine Learning*, pages 1802–1811. PMLR, 2019.

Clément Farabet, Yann LeCun, Koray Kavukcuoglu, Eugenio Culurciello, Berin Martini, Polina Akselrod, and Selcuk Talay. Large-scale fpga-based convolutional networks. *Scaling up machine learning: parallel and distributed approaches*, 13(3):399–419, 2011.

Alhussein Fawzi and Pascal Frossard. Manitest: Are classifiers really invariant? *arXiv preprint arXiv:1507.06535*, 2015.

Uriel Feige, Vahab S Mirrokni, and Jan Vondrák. Maximizing non-monotone submodular functions. *SIAM Journal on Computing*, 40(4):1133–1153, 2011.

Matteo Fischetti and Jason Jo. Deep neural networks and mixed integer linear optimization. *Constraints*, 23(3):296–309, 2018.

Nir Friedman. Inferring cellular networks using probabilistic graphical models. *Science*, 303(5659):799–805, 2004.

Claudio Gallicchio and Alessio Micheli. Graph echo state networks. In *The 2010 international joint conference on neural networks (IJCNN)*, pages 1–8. IEEE, 2010.

Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 15554–15566, 2019.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1263–1272. JMLR. org, 2017.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *The International Conference on Learning Representations*, 2015.

Sven Gowal, Krishnamurthy Dvijotham, Robert Stanforth, Rudy Bunel, Chongli Qin, Jonathan Uesato, Timothy Mann, and Pushmeet Kohli. On the effectiveness of interval bound propagation for training verifiably robust models. *arXiv preprint arXiv:1810.12715*, 2018.

Sven Gowal, Krishnamurthy Dvijotham, Robert Stanforth, Timothy Mann, and Pushmeet Kohli. A dual approach to verify and train deep networks. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pages 6156–6160. AAAI Press, 2019.

Monique Guignard and Siwhan Kim. Lagrangean decomposition: A model yielding stronger lagrangean bounds. *Mathematical programming*, 39(2):215–228, 1987.

Shengnan Guo, Youfang Lin, Ning Feng, Chao Song, and Huaiyu Wan. Attention based spatial-temporal graph convolutional networks for traffic flow forecasting. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 922–929, 2019.

LLC Gurobi Optimization. Gurobi optimizer reference manual, 2020. URL `http://www.gurobi.com`.

Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

Christoph Hansknecht, Imke Joormann, and Sebastian Stiller. Cuts, primal heuristics, and learning to branch for the time-dependent traveling salesman problem. *arXiv preprint arXiv:1805.01415*, 2018.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Mikael Henaff, Joan Bruna, and Yann LeCun. Deep convolutional networks on graph-structured data. *arXiv preprint arXiv:1506.05163*, 2015.

Patrick Henriksen and A Lomuscio. *Efficient Neural Network Verification via Adaptive Refinement and Adversarial Search*. PhD thesis, Imperial College London, 2019.

Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, 14(8):2, 2012.

Geoffrey E Hinton. To recognize shapes, first learn to generate images. *Progress in brain research*, 165:535–547, 2007.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Xiaowei Huang, Daniel Kroening, Wenjie Ruan, James Sharp, Youcheng Sun, Emese Thamo, Min Wu, and Xinping Yi. A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability. *Computer Science Review*, 37:100270, 2020.

Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin. Black-box adversarial attacks with limited queries and information. In *International Conference on Machine Learning*, pages 2137–2146. PMLR, 2018.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Laurent Itti, Christof Koch, and Ernst Niebur. A model of saliency-based visual attention for rapid scene analysis. *IEEE Transactions on pattern analysis and machine intelligence*, 20(11):1254–1259, 1998.

Ashesh Jain, Amir R Zamir, Silvio Savarese, and Ashutosh Saxena. Structural-rnn: Deep learning on spatio-temporal graphs. In *Proceedings of the ieee conference on computer vision and pattern recognition*, pages 5308–5317, 2016.

Xiaojun Jia, Yong Zhang, Baoyuan Wu, Jue Wang, and Xiaochun Cao. Boosting fast adversarial training with learnable adversarial initialization. *arXiv preprint arXiv:2110.05007*, 2021.

Kipp W Johnson, Jessica Torres Soto, Benjamin S Glicksberg, Khader Shameer, Riccardo Miotto, Mohsin Ali, Euan Ashley, and Joel T Dudley. Artificial intelligence in cardiology. *Journal of the American College of Cardiology*, 71(23):2668–2679, 2018.

Can Kanbak, Seyed-Mohsen Moosavi-Dezfooli, and Pascal Frossard. Geometric robustness of deep networks: analysis and improvement. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4441–4449, 2018.

Guy Katz, Clark Barrett, David Dill, Kyle Julian, and Mykel Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. *International Conference on Computer Aided Verification*, 2017a.

Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer, 2017b.

Guy Katz, Derek A Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, et al. The marabou framework for verification and analysis of deep neural networks. In *International Conference on Computer Aided Verification*, pages 443–452. Springer, 2019.

Elias Boutros Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016a.

Thomas N Kipf and Max Welling. Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308*, 2016b.

Emiel Krahmer, Sebastiaan van Erk, and André Verleg. Graph-based generation of referring expressions. *Computational Linguistics*, 29(1):53–72, 2003.

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533*, 2016.

Yann LeCun et al. Generalization and network design strategies. *Connectionism in perspective*, 19(143-155):18, 1989.

Ruoyu Li, Sheng Wang, Feiyun Zhu, and Junzhou Huang. Adaptive graph convolutional neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018a.

Yaguang Li, Rose Yu, Cyrus Shahabi, and Yan Liu. Diffusion convolutional recurrent neural network: Data-driven traffic forecasting. *arXiv preprint arXiv:1707.01926*, 2017.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. Learning deep generative models of graphs. *arXiv preprint arXiv:1803.03324*, 2018b.

Jingyue Lu and M Pawan Kumar. Neural network branching for neural network verification. In *International Conference on Learning Representations*, 2020a.

Jingyue Lu and M Pawan Kumar. Neural network branching for neural network verification. In *International Conference on Learning Representations*, 2020b.

Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*. Citeseer, 2013.

Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, 2018.

Alessio Micheli. Neural network for graphs: A contextual constructive approach. *IEEE Transactions on Neural Networks*, 20(3):498–511, 2009.

Seungyong Moon, Gaon An, and Hyun Oh Song. Parsimonious black-box adversarial attacks via efficient combinatorial optimization. In *International Conference on Machine Learning*, pages 4636–4645. PMLR, 2019.

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

Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Universal adversarial perturbations. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1765–1773, 2017.

EA Murphy, Beate Ehrhardt, CL Gregson, OA von Arx, April Hartley, MR Whitehouse, MS Thomas, Gregor Stenhouse, TJS Chesser, CJ Budd, et al. Machine learning outperforms clinical experts in classification of hip fractures. *Scientific reports*, 12(1): 1–11, 2022.

Ananthan Nambiar, Maeve Heflin, Simon Liu, Sergei Maslov, Mark Hopkins, and Anna Ritz. Transforming the language of life: Transformer neural networks for protein prediction tasks. In *Proceedings of the 11th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, pages 1–8, 2020.

Nina Narodytska and Shiva Prasad Kasiviswanathan. Simple black-box adversarial perturbations for deep networks. *arXiv preprint arXiv:1612.06299*, 2016.

Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. In *International conference on machine learning*, pages 2014–2023. PMLR, 2016.

Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 372–387. IEEE, 2016.

Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 506–519, 2017.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. *Automatic differentiation in pytorch*, 2017.

Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Towards practical verification of machine learning: The case of computer vision systems. *arXiv preprint arXiv:1712.01785*, 2017.

Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics*, 4(5):1–17, 1964.

Omid Poursaeed, Isay Katsman, Bicheng Gao, and Serge Belongie. Generative adversarial perturbations. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4422–4431, 2018.

Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. Certified defenses against adversarial examples. In *International Conference on Learning Representations*, 2018.

Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

Prajit Ramachandran, Niki Parmar, Ashish Vaswani, Irwan Bello, Anselm Levskaya, and Jon Shlens. Stand-alone self-attention in vision models. *Advances in Neural Information Processing Systems*, 32, 2019.

Marc'Aurelio Ranzato, Christopher Poultney, Sumit Chopra, and Yann Cun. Efficient learning of sparse representations with an energy-based model. *Advances in neural information processing systems*, 19, 2006.

Vicenc Rubies Royo, Roberto Calandra, Dusan M Stipanovic, and Claire Tomlin. Fast neural network verification via shadow prices. *arXiv preprint arXiv:1902.07247*, 2019.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

Marwin HS Segler, Thierry Kogej, Christian Tyrchan, and Mark P Waller. Generating focused molecule libraries for drug discovery with recurrent neural networks. *ACS central science*, 4(1):120–131, 2018.

Youngjoo Seo, Michaël Defferrard, Pierre Vandergheynst, and Xavier Bresson. Structured sequence modeling with graph convolutional recurrent networks. In *International Conference on Neural Information Processing*, pages 362–373. Springer, 2018.

Alex Serban, Erik Poll, and Joost Visser. Adversarial examples on object recognition: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 53(3):1–38, 2020.

Pierre Sermanet, Koray Kavukcuoglu, Soumith Chintala, and Yann LeCun. Pedestrian detection with unsupervised multi-stage feature learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3626–3633, 2013.

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

Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. Fast and effective robustness certification. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 10802–10813. Curran Associates, Inc., 2018. URL `http://papers.nips.cc/paper/8278-fast-and-effective-robustness-certification.pdf`.

Gagandeep Singh, Jonathan Maurer, Christoph Mller, Matthew Mirman, Timon Gehr, Adrian Hoffmann, Petar Tsankov, Dana Drachsler Cohen, Markus Pschel, and Martin Vechev. Eth robustness analyzer for neural networks (eran), 2020. URL `https://github.com/eth-sri/eran`.

Yang Song, Rui Shu, Nate Kushman, and Stefano Ermon. Constructing unrestricted adversarial examples with generative models. *Advances in Neural Information Processing Systems*, 31:8312–8323, 2018.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

Richard Sutton. Two problems with back propagation and other steepest descent learning procedures for networks. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society, 1986*, pages 823–832, 1986.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

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

Petar Velickovic, William Fedus, William L Hamilton, Pietro Liò, Yoshua Bengio, and R Devon Hjelm. Deep graph infomax. *ICLR (Poster)*, 2(3):4, 2019.

VNN-COMP. International verification of neural networks competition (vnn-comp), 2020. URL `https://sites.google.com/view/vnn20/vnncomp`.

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*, pages 1225–1234, 2016.

Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient formal safety analysis of neural networks. *Conference on Neural Information Processing Systems*, 2018.

Zifei Wang, Xiaolin Huang, Jie Yang, and Nikola Kasabov. Universal adversarial perturbation generated by using attention information. In *Advances in Intelligent Systems Research and Innovation*, pages 21–39. Springer, 2022.

Jeremy C Weiss, Sriraam Natarajan, Peggy L Peissig, Catherine A McCarty, and David Page. Machine learning for personalized medicine: Predicting primary myocardial infarction from electronic health records. *AI Magazine*, 33(4):33–33, 2012.

Tsui-Wei Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Duane Boning, Inderjit S Dhillon, and Luca Daniel. Towards fast computation of certified robustness for relu networks. *International Conference on Machine Learning*, 2018a.

Tsui-Wei Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Duane Boning, Inderjit S Dhillon, and Luca Daniel. Towards fast computation of certified robustness for relu networks. *arXiv preprint arXiv:1804.09699*, 2018b.

Eric Wong and Zico Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. *International Conference on Machine Learning*, 2018.

Zonghan Wu, Shirui Pan, Guodong Long, Jing Jiang, and Chengqi Zhang. Graph wavenet for deep spatial-temporal graph modeling. *arXiv preprint arXiv:1906.00121*, 2019.

Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 2020.

Chaowei Xiao, Bo Li, Jun-Yan Zhu, Warren He, Mingyan Liu, and Dawn Song. Generating adversarial examples with adversarial networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 3905–3911, 2018a.

Chaowei Xiao, Jun-Yan Zhu, Bo Li, Warren He, Mingyan Liu, and Dawn Song. Spatially transformed adversarial examples. *arXiv preprint arXiv:1801.02612*, 2018b.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

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.

Muhamad Yani et al. Application of transfer learning using convolutional neural network method for early detection of terry's nail. In *Journal of Physics: Conference Series*, volume 1201, page 012052. IOP Publishing, 2019.

Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. *Advances in neural information processing systems*, 31, 2018.

Jiaxuan You, Rex Ying, Xiang Ren, William Hamilton, and Jure Leskovec. Graphrnn: Generating realistic graphs with deep auto-regressive models. In *International conference on machine learning*, pages 5708–5717. PMLR, 2018.

Bing Yu, Haoteng Yin, and Zhanxing Zhu. Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting. *arXiv preprint arXiv:1709.04875*, 2017.

Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. Efficient neural network robustness certification with general activation functions. In *Advances in neural information processing systems*, pages 4939–4948, 2018.

Zhengli Zhao, Dheeru Dua, and Sameer Singh. Generating natural adversarial examples. In *International Conference on Learning Representations*, 2018.

Yi-Tong Zhou and Rama Chellappa. Computation of optical flow using a neural network. In *ICNN*, pages 71–78, 1988.

Chenyi Zhuang and Qiang Ma. Dual graph convolutional networks for graph-based semi-supervised classification. In *Proceedings of the 2018 World Wide Web Conference*, pages 499–508, 2018.

Appendices

A
Generating Adversarial Examples with Graph Neural Networks

Contents

A.1 GNN Architecture

Having described the main structure of the GNN above, as well as the implementation of the forward and backward passes, and the final update step, we will now explain in greater detail how the node features are computed. The node features consist of three pieces of information: the gradient at the current point, the intermediate bounds of the neurons in the original network, and information from solving a standard relaxation of the adversarial loss. We now describe in greater detail how each of those parts is defined and computed.

A.1.1 Intermediate Bounds

We recall the definition of the original network we are trying to attack: $f(\mathbf{x}_0) = \hat{\mathbf{x}}_L \in \mathbb{R}^m$, where

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

$$\mathbf{x}_i = \sigma(\hat{\mathbf{x}}_i), \qquad \text{for } i = 1, \ldots, L-1. \tag{A.2}$$

The adversarial problem can then be written as

$$\min \hat{\mathbf{x}}_L[y] - \hat{\mathbf{x}}_L[\hat{y}] \tag{A.3}$$

$$\hat{\mathbf{x}}_{i+1} = W^{i+1}\mathbf{x}_i + \mathbf{b}^{i+1}, \qquad \text{for } i = 0, \ldots, L-1, \tag{A.4}$$

$$\mathbf{x}_i = \sigma(\hat{\mathbf{x}}_i), \qquad \text{for } i = 1, \ldots, L-1, \tag{A.5}$$

$$\mathbf{x}_0 \in \mathcal{C} \subseteq \mathbb{R}^d \tag{A.6}$$

We now aim to compute bounds on the values that each neuron $\mathbf{x}_k[j]$ can take, where k indexes the layer, and j the neuron in that layer. The computation of the lower bound of a neuron can be described as finding a lower bound for the following minimization problem:

$$
\begin{aligned}
\min \hat{\mathbf{x}}_k[j] & & \text{(A.7)}\\
\hat{\mathbf{x}}_{i+1} = W^{i+1}\mathbf{x}_i + \mathbf{b}^{i+1}, & \quad \text{for } i = 0, \ldots, k-1, & \text{(A.8)}\\
\mathbf{x}_i = \sigma(\hat{\mathbf{x}}_i), & \quad \text{for } i = 1, \ldots, k-1, & \text{(A.9)}\\
\mathbf{x}_0 \in \mathcal{C} \subseteq \mathbb{R}^d. & & \text{(A.10)}
\end{aligned}
$$

We solve this using the method by Wong and Kolter [2018] and using Interval Bound Propagation [Gowal et al., 2018] and record the tighter of the two. We get the upper bound by changing the sign of the weights of the k-th layer function. We denote the lower and upper bounds for the j-th neuron in the k-th layer as $\mathbf{l}_k[j]$ and $\mathbf{u}_k[j]$, respectively.

A.1.2 Solving a Standard Relaxation with Supergradient Ascent

We now describe a standard relaxation of the adversarial problem from the verification literature. Neural Network verification methods aim to solve the opposite problem of adversarial attacks. They try to prove that for a given network f, an image $\mathbf{x}$, a convex neighbourhood around it, $\mathcal{C}$, a true class y, and an incorrect target class $\hat{y}$, there does not exists an example $\mathbf{x}' \in \mathcal{C}$ that the network misclassifies as $\hat{y}$. In other words, it aims to show that no adversarial attack would be successful at finding an adversarial example. This is equivalent to showing that the minimum in (A.3) is strictly positive.

We now summarize the work of Bunel et al. [2020a] who solve this problem using standard relaxations. First they relax the non-linear ReLU activation functions using the so-called Planet relaxation [Ehlers, 2017b] before computing lower bounds using a formulation based on Lagrangian decompositions.

Planet Relaxation. We denote the output of the k-th layer before the application of the ReLU as $\hat{\mathbf{z}}_k$ and the output of applying the ReLU to $\hat{\mathbf{z}}_k$ as $\mathbf{x}_k$. Given the lower bounds $\mathbf{l}_k$ and upper bounds $\mathbf{u}_k$ of the values of $\hat{\mathbf{z}}_k$, we relax the ReLU activations $\mathbf{x}_k = \sigma(\hat{\mathbf{z}}_k)$ to its convex hull $cvx\_hull_\sigma(\hat{\mathbf{z}}_k, \mathbf{x}_k, \mathbf{l}_k, \mathbf{u}_k)$, defined as follows:

$$cvx\_hull_\sigma(\hat{\mathbf{z}}_k, \mathbf{x}_k, \mathbf{l}_k, \mathbf{u}_k) \equiv \begin{cases} \mathbf{x}_k[i] \geq 0 \quad \mathbf{x}_k[i] \geq \hat{\mathbf{z}}_k[i] & \\ \mathbf{x}_k[i] \leq \frac{\mathbf{u}_k[i](\hat{\mathbf{z}}_k[i] - \mathbf{l}_k[i])}{\mathbf{u}_k[i] - \mathbf{l}_k[i]} & \text{if } \mathbf{l}_k[i] < 0 \text{ and } \mathbf{u}_k[i] > 0 \\ \mathbf{x}_k[i] = 0 & \text{if } \mathbf{u}_k[i] \leq 0 \\ \mathbf{x}_k[i] = \hat{\mathbf{z}}_k[i] & \text{if } \mathbf{l}_k[i] \geq 0. \end{cases} \tag{A.11}$$

To improve readability of our relaxation, we introduce the following notations for the constraints corresponding to the input and the k-th layer respectively:

$$\mathcal{P}_0(\mathbf{x}_0, \hat{\mathbf{z}}_1) \equiv \begin{cases} \mathbf{x}_0 \in C \\ \hat{\mathbf{z}}_1 = W_1 \mathbf{x}_0 + \mathbf{b}_1 \end{cases} \qquad \mathcal{P}_k(\hat{\mathbf{z}}_k, \hat{\mathbf{z}}_{k+1}) \equiv \begin{cases} \exists \mathbf{x}_k \text{ s.t.} \\ \mathbf{l}_k \leq \hat{\mathbf{z}}_k \leq \mathbf{u}_k \\ cvx\_hull_\sigma(\hat{\mathbf{z}}_k, \mathbf{x}_k, \mathbf{l}_k, \mathbf{u}_k) \\ \hat{\mathbf{z}}_{k+1} = W_{k+1}\mathbf{x}_k + \mathbf{b}_{k+1}. \end{cases} \tag{A.12}$$

Using the above notation, the Planet relaxation for computing the lower bound can be written as:

$$\min_{\mathbf{x}, \hat{\mathbf{z}}} \hat{\mathbf{z}}_n \text{ s.t. } \mathcal{P}_0(\mathbf{x}_0, \hat{\mathbf{z}}_1); \mathcal{P}_k(\hat{\mathbf{z}}_k, \hat{\mathbf{z}}_{k+1}) \text{ for } k \in [1, \ldots, L-1]. \tag{A.13}$$

Lagrangian Decomposition. We often merely need approximations of the bounds rather than the precise values of them: if we show that some valid lower bound of (A.3) is strictly positive, then it follows that (A.3) is also strictly positive and no adversarial example exists. We can therefore make use of the primal-dual formulation of the problem as every feasible solution to the dual problem provides a valid lower bound for the primal problem. Following the work of Bunel et al. [2020a] we will use the Lagrangian decomposition Guignard and Kim [1987]. To this end, we first create two copies $\hat{\mathbf{z}}_{A,k}, \hat{\mathbf{z}}_{B,k}$ of each variable $\hat{\mathbf{z}}_k$:

$$\begin{aligned} \min_{\mathbf{x},\hat{\mathbf{z}}} \hat{\mathbf{z}}_{A,n} \text{ s.t. } & \mathcal{P}_0(\mathbf{x}_0, \hat{\mathbf{z}}_{A,1}); \mathcal{P}_k(\hat{\mathbf{z}}_{B,k}, \hat{\mathbf{z}}_{A,k+1}) && \text{for } k \in [1, \ldots, L-1] \\ & \hat{\mathbf{z}}_{A,k} = \hat{\mathbf{z}}_{B,k} && \text{for } k \in [1, \ldots, L-1]. \end{aligned} \tag{A.14}$$

Next we obtain the dual by introducing Lagrange multipliers $\boldsymbol{\rho}$ corresponding to the equality constraints of the two copies of each variable:

$$\begin{aligned} q(\boldsymbol{\rho}) = \min_{\mathbf{x},\hat{\mathbf{z}}} \quad & \hat{\mathbf{z}}_{A,n} + \sum_{k=1,\ldots,n-1} \boldsymbol{\rho}_k^\top (\hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}) \\ \text{s.t.} \quad & \mathcal{P}_0(\mathbf{x}_0, \hat{\mathbf{z}}_{A,1});\ \mathcal{P}_k(\hat{\mathbf{z}}_{B,k}, \hat{\mathbf{z}}_{A,k+1}) \text{ for } k \in [1, \ldots, L-1]. \end{aligned} \tag{A.15}$$

Solving the Relaxation using Supergradient Ascent We solve the dual problem (A.15) using the supergradient ascent method proposed by Bunel et al. [2020a]. We run supergradient ascent together with Adam for 100 steps to get a set of dual variables $\boldsymbol{\rho}$, as well as a matching set of primal variables $\mathbf{x}_0$ which, henceforth, we denote as $\mathbf{x}^{lp}$.

A.1.3 Node Features

For each node $\mathbf{v}_k[i]$ we define a corresponding q-dimensional feature vector $\mathbf{f}_k[i] \in \mathbb{R}^q$ describing the current state of that node. We define the node features for the input layer as follows:

$$\mathbf{f}_0[i] := \left(\mathbf{x}^t[i],\ \operatorname{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y')[i]), \mathbf{l}_0[i], \mathbf{u}_0[i], \mathbf{x}^{lp}[i]\right)^\top, \tag{A.16}$$

and for the hidden and final layers as:

$$\mathbf{f}_k[i] := (\mathbf{l}_k[i], \mathbf{u}_k[i], \boldsymbol{\rho}_k[i])^\top. \tag{A.17}$$

Here, $\mathbf{x}^t$ is our current point, $\nabla_{\mathbf{x}} L(\mathbf{x}, y, y')$ is the gradient at the current point, and $\mathbf{l}_k[i]$, and $\mathbf{u}_k[i]$ are the bounds for each node as described above (§A.1.1). Further, $\boldsymbol{\rho}_k$ is the current assignment to the corresponding dual variables computed using supergradient ascent and $\mathbf{x}_k^{lp}$ is the input corresponding to the primal solution of the dual (see §A.1.2). Other features can be used depending on the exact task or experimental setup. We note that there exists a trade-off between using more expressive features that are difficult to compute or simpler ones that are faster to compute.

A.1.4 Embeddings.

For every node $v_k[i]$ we compute a corresponding p-dimensional embedding vector $\boldsymbol{\mu}_k[i] \in \mathbb{R}^p$ using a learned function g:

$$\boldsymbol{\mu}_k[i] := g(\mathbf{f}_k[i]). \tag{A.18}$$

In our case g is a simple multilayer perceptron (MLP), which is made up of a set of linear layers Θ_i and non-linear ReLU activations. We train two different MLPs, one for the input layer, g^{inp}, and one for all other layers g. We have the following set of trainable parameters:

$$\Theta_0^{inp} \in \mathbb{R}^{5\times p}, \quad \Theta_0 \in \mathbb{R}^{3\times p} \quad \Theta_1^{inp}, \dots, \Theta_{T_1}^{inp}, \ \Theta_1, \dots, \Theta_{T_1} \in \mathbb{R}^{p\times p} \tag{A.19}$$

Given feature vectors $\mathbf{f}_0, \dots, \mathbf{f}_L$ we compute the following set of vectors:

$$\boldsymbol{\mu}_0^0 = \mathrm{relu}(\Theta_0^{inp} \cdot \mathbf{f}_0), \qquad \boldsymbol{\mu}_0^{l+1} = \mathrm{relu}(\Theta_{l+1}^{inp} \cdot \boldsymbol{\mu}_0^l), \qquad \text{for } l = 1, \dots, T_1 - 1 \tag{A.20}$$

$$\boldsymbol{\mu}_k^0 = \mathrm{relu}(\Theta_0 \cdot \mathbf{f}_k), \qquad \boldsymbol{\mu}_k^{l+1} = \mathrm{relu}(\Theta_{l+1} \cdot \boldsymbol{\mu}_k^l), \qquad \text{for } l = 1, \dots, T_1 - 1;\ k = 1, \dots, L. \tag{A.21}$$

We initialize the embedding vector to be $\boldsymbol{\mu}_k = \boldsymbol{\mu}_k^{T_1}$, where $T_1 + 1$ is the depth of the MLP.

A.2 Hyper-parameter Analysis for Baselines

A.2.1 PGD Attack

PGD aims to generate adversarial examples by picking $\mathbf{x}^0 \in \mathcal{B}(\mathbf{x}, \epsilon)$ uniformly at random and then running the following update step for T steps or until $L(\mathbf{x}^t, y, \hat{y}) > 0$:

$$\mathbf{x}^{t+1} = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)} \left(\mathbf{x}^t + \alpha \ \mathrm{sgn}(\nabla L(\mathbf{x}^t, y, \hat{y})) \right). \tag{A.22}$$

We need to pick optimal values for the hyper-parameters T and α. We run a hyper-parameter analysis on the validation dataset described above. We try every combination of $T \in \{50, 100, 250, 1000\}$ and $\alpha \in \{1e-1, 1e-2, 1e-2\}$ and rank them both for the average time taken and the percentage of properties they time out. Taking the average of the two ranks we see that choosing $T = 100$ and $\alpha = 0.01$ is the best combination (Table A.1). We repeat the hyper-parameter on an easier version of the validation dataset which we get by adding a delta of 0.001 to the value of every perturbation. Just like for the original validation dataset, the following two combinations of hyper-parameters perform significantly better than all other combinations: $(T = 1000, \alpha = 0.001)$ and $(T = 100, \alpha = 0.01)$. They time out on the same number of properties but the former has a slightly lower average solving time this time.

Table A.1: Hyper-parameter analysis for PGD attack on the Validation Set

T	α	average_time	timeout	rank_time	rank_timeout	average_rank
100	0.01	87.740020	0.843137	1.0	2.0	1.50
1000	0.001	91.157906	0.862745	2.0	3.5	2.75
250	0.01	92.968972	0.823529	5.0	1.0	3.00
500	0.01	91.378347	0.862745	3.0	3.5	3.25
1000	0.01	91.607033	0.882353	4.0	5.0	4.50
50	0.01	93.659832	0.921569	6.0	6.5	6.25
500	0.001	94.735763	0.921569	7.0	6.5	6.75
100	0.1	99.852496	0.980392	8.0	8.0	8.00
1000	0.1	101.000000	1.000000	12.0	12.0	12.00
100	0.001	101.000000	1.000000	12.0	12.0	12.00
250	0.001	101.000000	1.000000	12.0	12.0	12.00
250	0.1	101.000000	1.000000	12.0	12.0	12.00
500	0.1	101.000000	1.000000	12.0	12.0	12.00
50	0.001	101.000000	1.000000	12.0	12.0	12.00
50	0.1	101.000000	1.000000	12.0	12.0	12.00

A.2.2 MI-FGSM+ Attack

Adding momentum to the MI-FGSM attack was first suggested by Dong et al. [2018]. The original implementation is described in Algorithm 9. This version does not perform well on our challenging dataset however. In fact it doesn't manage to find a single counter example on the validation dataset for any combination of hyper-parameters. One reason for this behaviour could be that often adversarial examples lie near the boundary of the input domain (at least in one dimension) and to reach those points every single update step needs to have the correct sign for that particular dimension (as we take T steps of the form $\pm{}^{\epsilon}/_{T}$) . In order to improve its performance on difficult datasets we run it with random restarts. However, as the original implementation has no statistical elements, every run on the same image with the same hyper-parameters would have the same outcome. We thus adapt MI-FGSM to initialize the starting point uniformly at random from the input domain rather than starting at the original image. We further observed that initializing α as done in the original implementation greatly reduces its rate of success. We thus treat it as a hyper-parameter and give it as input to the function. We denote this optimized version of MI-FGSM as MI-FGSM+ and describe it in greater detail in Algorithm 10. Similarly to PGD-Attack we now optimize over the hyper-parameters on the validation dataset. We try the following values: $T \in \{10, 100, 1000\}$, $\alpha \in \{1e{-}1, 1e{-}2, 1e{-}3\}$, $\eta \in \{0.0, 0.25, 0.5, 1.0\}$. As we did for PGD we rank the performance of all combinations of hyper-parameters with respect to the number of properties successfully attack and average time taken (Table A.2). We get the following optimal set of hyper-parameters: $T = 100, \alpha = 0.1, \eta = 0.5$.

We also perform a similar analysis on an easier version of the validation dataset, where we add a constant (0.001) to the allowed perturbation value for each image. We reach the same optimal assignment for the three hyper-parameters as before.

Algorithm 9 MI-FGSM [Dong et al., 2018]

```
Input: $f, \mathbf{x}, y, \hat{y}, \mu, T$
1: $\alpha \leftarrow \epsilon/T$ {Initialize stepsize parameter}
2: $\mathbf{x}^0 \leftarrow \mathbf{x}$ {Initialize starting point}
3: $\mathbf{g}^0 \leftarrow 0$ {Initialize momentum vector}
4: for $t = 1, \dots, T$: do
5:     $\mathbf{g}^{t+1} \leftarrow \mu \cdot \mathbf{g}^t + \frac{\nabla_x L(\mathbf{x}^t, y, \hat{y})}{\|\nabla_x L(\mathbf{x}^t, y, \hat{y})\|_1}$ {Update the momentum term}
6:     $\mathbf{x}^{t+1} = \mathbf{x}^t + \alpha \cdot sgn(\mathbf{g}^{t+1})$ {Update the current point}
7: end for
8: return $\mathbf{x}^T$
```

Algorithm 10 MI-FGSM+

```
Input: $f, \mathbf{x}, y, \hat{y}, \mu, T, \alpha$
1: sample $\mathbf{x}^0$ from $\mathcal{B}(\mathbf{x}, \epsilon)$ {Initialize starting point}
2: $\mathbf{g}^0 \leftarrow 0$ {Initialize momentum vector}
3: for $t = 1, \dots, T$: do
4:     $\mathbf{g}^{t+1} \leftarrow \mu \cdot \mathbf{g}^t + \frac{\nabla_x L(\mathbf{x}^t, y, \hat{y})}{\|\nabla_x L(\mathbf{x}^t, y, \hat{y})\|_1}$ {Update the momentum term}
5:     $\mathbf{x}^{t+1} = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)} (\mathbf{x}^t + \alpha \cdot sgn(\mathbf{g}^{t+1}))$ {Update the current point and project }
6: end for
7: return $\mathbf{x}^T$
```

Table A.2: Hyper-parameter analysis for MI-FGSM+ on the Validation Set

T	α	μ	average_time	timeout	rank_time	rank_timeout	average_rank
100	0.1	0.5	43.513870	0.305556	1.0	1.0	1.00
100	0.1	1.0	55.608030	0.500000	2.0	2.5	2.25
1000	0.01	0.5	59.396192	0.500000	3.0	2.5	2.75
1000	0.1	1.0	61.623909	0.527778	4.0	4.5	4.25
1000	0.1	0.5	63.214772	0.527778	5.0	4.5	4.75
1000	0.01	1.0	65.085730	0.583333	6.0	7.0	6.50
100	0.1	0.25	70.484347	0.555556	9.0	6.0	7.50
1000	0.01	0.25	67.918430	0.638889	7.0	8.5	7.75
100	0.01	0.5	69.199902	0.638889	8.0	8.5	8.25
100	0.01	0.25	75.356267	0.722222	10.0	10.5	10.25
1000	0.001	0.5	76.749888	0.722222	11.0	10.5	10.75
1000	0.001	0.25	82.939370	0.805556	12.0	12.5	12.25
10	0.1	0.5	83.524314	0.833333	13.0	14.5	13.75
100	0.01	1.0	83.739845	0.833333	14.0	14.5	14.25
1000	0.1	0.25	88.323196	0.805556	16.0	12.5	14.25
1000	0.001	1.0	87.845959	0.861111	15.0	16.0	15.50
10	0.1	1.0	90.158706	0.888889	17.0	17.0	17.00
10	0.1	0.25	94.782105	0.916667	18.0	18.0	18.00
10	0.01	0.25	100.012025	1.000000	19.0	23.0	21.00
10	0.001	0.5	100.012147	1.000000	20.0	23.0	21.50
10	0.001	1.0	100.012219	1.000000	21.0	23.0	22.00
10	0.01	1.0	100.012781	1.000000	22.0	23.0	22.50
10	0.01	0.5	100.013981	1.000000	23.0	23.0	23.00
10	0.001	0.25	100.015674	1.000000	24.0	23.0	23.50
100	0.001	1.0	100.119291	1.000000	25.0	23.0	24.00
100	0.001	0.5	100.124259	1.000000	26.0	23.0	24.50
100	0.001	0.25	100.134148	1.000000	27.0	23.0	25.00

A.2.3 Carlini and Wagner Attack

We run the l_∞ version of the Carlini and Wanger Attack ($C\%W$) [Carlini and Wagner, 2017]. C&W aims to repeatedly optimize

$$\min_\delta \ c \cdot h(x+\delta) + \sum_i [(\delta_i - \tau)_+], \tag{A.23}$$

for different values of c and τ, where h is a surrogate function based on the neural network we are trying to attack. The method is described in greater detail in Algorithm 11.

C&W has six hyper-parameters we search over: $T, c_{init}, c_{fin}, \gamma_\tau, \gamma_c, \alpha$. Running every possible combination of assignments to the hyper-parameters like we did for PGD and MI-FGSM+ becomes computationally too expensive as the number of assignments increases exponentially in the number of parameters. Instead we split the search into three rounds. We initialize the parameters with those suggested in the original paper. In the first round we change one parameter at a time, keeping all other parameters constant. At the end of the first round we record the optimal values for each parameter. We evaluate the performance by taking the average of the minimum perturbation for which C&W managed to return a successful attack for each image. We then repeat this process twice more: each time searching over the optimal hyper-parameter assignment one at a time, and updating the values at the end of each round. At the end of the third round we reach the following assignment: $T = 100$, $c_{init} = 1e-5$, $c_{fin} = 1000$, $\gamma_\tau = 0.99$, $\gamma_c = 1.5$, $\alpha = 1e-4$.

Algorithm 11 C&W

```
Input: $h, \mathbf{x}, y, \hat{y}, T, c_{init}, c_{fin}, \gamma_\tau, \gamma_c, \alpha$
 1: $c \leftarrow c_{init}$
 2: $\tau \leftarrow 1.0$
 3: while $\tau < 0.1$ and $c < c_{fin}$ do
 4:
        $$\min_{\delta} \; c \cdot h(x+\delta) + \sum_i [(\delta_i - \tau)_+] \qquad (A.24)$$
 5:     Optimize A.24 using the Adam optimizer with a learning rate of $\alpha$, and a
        step number of $T$
 6:     if found a counter example with $\delta_i \leq \tau \; \forall i$ then
 7:         $\tau \leftarrow \tau * \gamma_\tau$ {Decay $\tau$ using the decay factor $\gamma_\tau$}
 8:         $c \leftarrow c * 1/2$ {Decay $c$ using factor $\gamma_c$}
 9:     else
10:         $c \leftarrow c * \gamma_c$
11:     end if
12: end while
13:
14: return Best $\delta$ found
```

	T	α	c_{init}	c_{fin}	$\gamma_{\mathcal{T}}$	γ_c	Avg $(\epsilon_{val} - \epsilon_{C\&W})$
Round 1	1000	1e-2	1e-5	20	0.9	2.0	-
	10	-	-	-	-	-	-0.170515
	100	-	-	-	-	-	**-0.134574**
	1000	-	-	-	-	-	-0.146015
	-	1e-3	-	-	-	-	**-0.050695**
	-	1e-2	-	-	-	-	-0.146015
	-	1e-1	-	-	-	-	-0.717864
	-	-	1e-5	-	-	-	-0.146015
	-	-	1e-4	-	-	-	-0.140346
	-	-	1e-3	-	-	-	**-0.130972**
	-	-	1e-2	-	-	-	-0.149450
	-	-	-	0.1	-	-	-0.199197
	-	-	-	1	-	-	-0.197057
	-	-	-	10	-	-	-0.160842
	-	-	-	100	-	-	**-0.105023**
	-	-	-		0.5	-	-0.221801
	-	-	-		0.9	-	**-0.146015**
	-	-	-		0.99	-	-0.180077
	-	-	-		-	1.5	-0.161380
	-	-	-		-	2.0	**-0.146015**
	-	-	-		-	5.0	-0.162026
Round 2	100	1e-3	1e-3	100	0.9	2.0	-
	10	-	-	-	-	-	-0.068567
	100	-	-	-	-	-	**-0.051525**
	1000	-	-	-	-	-	-0.052210
	-	1e-4	-	-	-	-	-0.059814
	-	1e-3	-	-	-	-	**-0.051525**
	-	1e-2	-	-	-	-	-0.101191
	-	-	1e-5	-	-	-	-0.051074
	-	-	1e-4	-	-	-	**-0.050897**
	-	-	1e-3	-	-	-	-0.051525
	-	-	1e-2	-	-	-	-0.053127
	-	-	-	10	-	-	-0.053124
	-	-	-	100	-	-	-0.051525
	-	-	-	1000	-	-	**-0.051488**
	-	-	-		0.5	-	-0.180116
	-	-	-		0.9	-	-0.051525
	-	-	-		0.99	-	**-0.036093**
	-	-	-		-	1.5	**-0.051445**
	-	-	-		-	2.0	-0.051525
	-	-	-		-	5.0	-0.052622
Round 3	100	1e-3	1e-4	1000	0.99	1.5	-
	10	-	-	-	-	-	-0.045861
	100	-	-	-	-	-	**-0.035893**
	1000	-	-	-	-	-	-0.101963
	-	1e-4	-	-	-	-	**-0.033943**
	-	1e-3	-	-	-	-	-0.035893
	-	1e-2	-	-	-	-	-0.098903
	-	-	1e-5	-	-	-	**-0.035488**
	-	-	1e-4	-	-	-	-0.035893
	-	-	1e-3	-	-	-	0.035676
	-	-	-	10	-	-	-0.035957
	-	-	-	100	-	-	**-0.035893**
	-	-	-	1000	-	-	**-0.035893**
	-	-	-		0.9	-	-0.049217
	-	-	-		0.99	-	**-0.035893**
	-	-	-		0.999	-	-0.128462
	-	-	-		-	1.25	-0.037318
	-	-	-		-	1.5	**-0.035893**
	-	-	-		-	2.0	-0.037543

Table A.3: Hyper-parameter analysis for C&W attack on the Validation Set

B

Attention for Adversarial Examples: Learning from Mistakes

Contents

B.1 Node features

This section explains how the feature vectors can be constructed for the input, hidden and output layers. As mentioned above, we aim to design the features so that they capture the maximum possible information needed to make a good branching decision while keeping the computational complexity as low as possible. Some of the text in this section has been provided by Aleksandr Agadzhanov.

B.1.1 Input node features

Firstly, the original lower and upper bounds on the i-th input node l_i and u_i are selected as this node's features as they are very indicative of the influence of this node on the output of the network.

If the GNN framework is invoked after some initial adversarial PGD attack is unsuccessful, then the information obtained from this attack can be used to generate more features for the nodes of the network. It should be noted that if this initial adversarial PGD attack is successful, then the GNN framework does not need to be invoked at all since the given property was proven to be false by a single randomly initialised PGD attack. Otherwise, all the available information can be provided to the GNN for it to make a branching decision. The first and obvious choice of a feature which can easily be obtained for each input node from an unsuccessful PGD attack is the value of this node at the end of the PGD attack.

Finally, information about the gradients of the output of a given neural network with respect to all of its inputs throughout the unsuccessful adversarial PGD attack can also be indicative of the location of a valid adversarial attack within the input domain. One option would be to provide the gradients at each step of the unsuccessful PGD attack to the GNN. However, since the number of steps of a PGD attack can vary and sometimes be in the order of thousands or tens of thousands, it might be excessive to do that as this will introduce redundancy and lead to higher computational complexity which is undesirable. Hence, in this project the following information about the gradients of the network's output with respect to each input was selected to enter the feature vectors of the corresponding input nodes:

- The mean gradient over all steps of an unsuccessful PGD attack
- The median gradient over all steps of an unsuccessful PGD attack
- The maximum gradient over all steps of an unsuccessful PGD attack
- The minimum gradient over all steps of an unsuccessful PGD attack
- The standard deviation of the gradients over all steps of an unsuccessful PGD attack
- The gradient at the last step of an unsuccessful PGD attack

B.1.2 Hidden node features

The features of all the activation nodes of a given neural network should be designed to contain information about the propagation of the numerical values through the network which in its turn can help the GNN to deduce the overall effect of each of the network's inputs on its output. As in the case of the input node features, it is sensible to include the lower and upper bounds on each activation node in its feature vector. However, while the exact lower and upper bounds on the input nodes are directly available from input constraints, obtaining the bounds on all the activation nodes is much more difficult. Computing the exact minima and maxima of the nodes of the activation and output layers is in general an intractable problem which means that approximate solutions have to be used to calculate the reasonably tight lower and upper bounds on these nodes. We compute these intermediate bounds using linear bound relaxations [Weng et al., 2018b]. This function made it

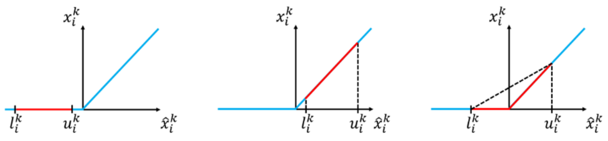


Figure B.1: Three cases of the state of a ReLU activation node based on this node's bounds. Left: blocking state due to both lower and upper bounds on a node being non-positive in which case the output of such ReLU activation node is always zero. Middle: passing state due to both lower and upper bounds on a node being non-negative in which case the output of such node is always equal to its input. Right: ambiguous state due to the lower and upper bounds being negative and positive respectively in which case the output can be either zero or equal to the input depending on the exact input to the node.

possible to obtain the approximate lower and upper bounds for all the activation and output layer nodes given the bounds on all the input layer nodes.

Since the unsuccessful PGD attack after which the GNN framework is initialised contains the values for each node of each layer at the end of the attack, a third feature which should enter the feature vectors of all the activation nodes is the value of the corresponding node at the end of the PGD attack.

The fourth feature which describes each activation node quite well and hence should be included in its feature vector is its associated bias in the original network. Mathematically, for i-th node of k-th activation layer where $k \in \{1, 2, ..., L-1\}$, the bias which should enter its feature vector is given by b_i^k.

The fifth feature is based on relaxations of the ReLU non-linearity and is taken from work by Lu and Kumar [2020a]. Activation nodes are quite specific due to their nonlinear nature and so specifying all of the above features might still not be enough to fully describe them. In case of the ReLU activation function, the state of any activation node can belong to one of the three cases. Denoting the lower and upper bounds of the i-th node of the k-th activation layer as as l_i^k and u_i^k respectively and the node values before and after the ReLU function is applied as $\hat{x}_i^k$ and x_i^k respectively, these cases can be visualised as shown in Figure B.1. In all parts of the figure, the ReLU function is plotted in blue and its part being considered based on the values of the lower and upper bounds is indicated in red.

In the first case, shown on the left, both lower and upper bounds on a particular activation node happen to be non-positive. Such activation node is referred to be in its blocking state as it has zero as its output for all possible inputs given by the lower and upper bounds, i.e. it blocks all the information. In the same way as it was done in Lu and Kumar [2020a], the final state which measures ambiguity, denoted by β_i^k, of each such activation node will be given by $\beta_i^k = 0$ as there is no ambiguity associated with this node.

The second case, shown in the middle, arises when both lower and upper bounds on a particular activation node are non-negative. In this case, the activation node is referred to be in its passing state because this time its output will be equal to its input for all possible inputs given by the lower and upper bounds. The final state of such activation node is also given by $\beta_i^k = 0$ because, as in the first case, there is no ambiguity involved in the propagation of information through such activation node.

In the final case, shown on the right, the lower and upper bounds on a particular activation node turn out to be negative and positive respectively. The state of such activation node is referred to as ambiguous since its output can be either zero or equal to its input depending on the exact input value. To define β_i^k, the same approach which was used in Lu and Kumar [2020a] can be followed whereby the intercept of the triangle formed by the section of the ReLU between l_i^k and u_i^k and the line connecting the ends of this section, indicated by the dashed line in the figure, is considered. The equation of this line can easily be shown to be:

$$x_i^k(\widehat{x}_i^k) = \frac{u_i^k}{u_i^k - l_i^k}\widehat{x}_i^k - \frac{u_i^k l_i^k}{u_i^k - l_i^k} \tag{B.1}$$

As $u_i^k \to 0$ and/or $l_i^k \to 0$, the intercept of the above line tends to zero, and as u_i^k becomes more positive and/or l_i^k becomes more negative, it increases. Hence, the intercept of the above line is a suitable measure of ambiguity of a ReLU node and therefore for each ambiguous activation node:

$$\beta_i^k = -\frac{u_i^k l_i^k}{u_i^k - l_i^k} \tag{B.2}$$

where β_i^k in the equation above is necessarily positive since for an ambiguous node $u_i^k > 0$ and $l_i^k < 0$.

B.1.3 Output node features

The output node features need to be informative of the state of the output of a given neural network to provide the GNN with the information about the input-output relationship of the network. The output feature vectors can be constructed using the same features as in the case of the activation nodes apart from the ambiguity descriptor. Hence, four features to enter the feature vector of each output node are:

- Lower bound on the output node
- Upper bound on the output node
- Node value at the end of an unsuccessful PGD attack
- Node bias in the original network

Having described in greater detail how we compute the node feature vectors $\mathbf{z}$, we now turn our attention to the implementation of the message passing algorithm.

B.2 Message Passing Functions

We now describe how the embedding vectors are initialize and updated using forward and backward passes. There are 5 learnt functions that implement the GNN message passing algorithm: F_{inp}, F_{hid}, F_{out}, B_{hid}, B_{inp}; and one learnt function that turns the embedding vectors into a branching decision: g. Some of these functions are inspired by the work of Lu and Kumar [2020a]. Each of the six functions will now be

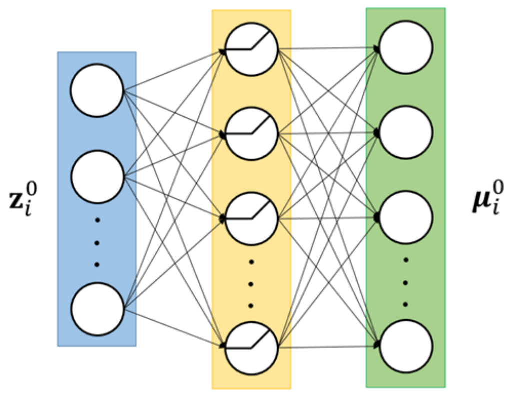


Figure B.2: Structure of the forward input embedding vector update network

discussed in detail in turn. From here on we refer to F_{hid} and B_{hid} as F_{hid} and B_{hid} respectively, to highlight that the state of the activation function, in our case the

B.2.1 Forward pass — input layer

The first network of interest is the one corresponding to the function F_{inp}. It has already been mentioned in the Section 4.4 that it, along with all the other networks, has a form of a multi-layered fully-connected network. This network in particular should simply process a local feature vector of the given input node and return an updated input embedding vector. Hence, this neural network will be designed to have a single stage with one activation layer containing ReLU activation functions, as depicted in Figure B.2.

It is important to make a few assumptions for the simplicity of the design of all the auxiliary neural networks, including the one above, in accordance with Lu and Kumar [2020a]. Firstly, it will be assumed that all the embedding vectors, i.e. the ones of the input, activation and output nodes, are of the same size. Secondly, all the activation layers of all the auxiliary neural networks will be assumed to have the same size. Finally, the two sizes from the previous two points will be assumed to be equal, as shown in Figure B.2.

B.2.2 Forward pass — hidden layer

The network corresponding to the function F_{act} should take a local feature vector of the activation node as well as all the embedding vectors of all the previous layer nodes propagated to this node as inputs and return an updated activation embedding vector. In contrast to the simple network described in Subsection B.2.1, in this case there are three stages to be considered which are explained below in turn.

The overall structure of the neural network which implements the update function F_{act} is shown in Figure B.3 where, since the network is fully-connected, the connections between layers are illustrated by single arrows for better visualisation purposes. The first stage, appearing in the top left corner of the figure and denoted by F_{act-lf}, is meant to process the local features of the i-th node of the k-th activation layer where $k \in \{1, 2, ..., L-1\}$. This stage has the exact same structure as the network implementing F_{inp}. It should be noted at this point that a further assumption has to be made for simplicity which says that the outputs of all the intermediate stages of all the auxiliary neural networks should have their size equal to that of the activation layers of these networks, as shown in Figure B.3. Denoting the parameters of the first stage of the network as $\boldsymbol{\theta}_1^0$ and its output as $\boldsymbol{R}_i^k$, the operation of F_{act-lf} can be written as:

$$\boldsymbol{R}_i^k = \begin{cases} F_{act-lf}(\mathbf{z}_i^k; \boldsymbol{\theta}_1^0) & \text{if} \quad \beta_i^k > 0 \\ \mathbf{0} & \text{otherwise} \end{cases} \tag{B.3}$$

where it is important to note that the condition for the first statement means that the pass through the first stage should only be made if the node is ambiguous, as explained in Subsection B.1.2. Otherwise, the output from the first stage is set to the zero vector of appropriate size, in accordance with Lu and Kumar [2020a].

The second stage of the network, denoted by F_{act-nb} and appearing in the bottom left corner of Figure B.3, needs to process the embedding vectors of the

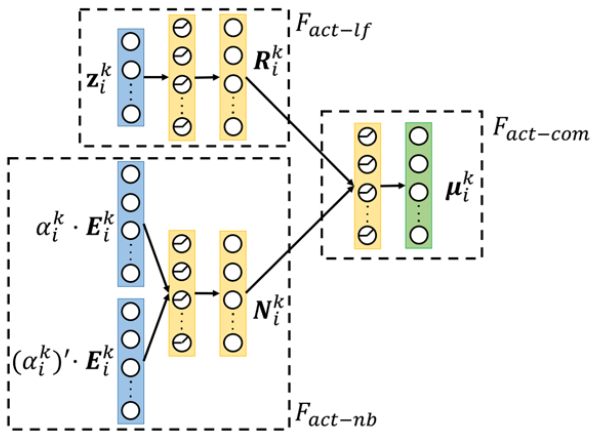


Figure B.3: Structure of the forward activation embedding vector update network

previous layer, i.e. those of the neighbouring nodes, hence the subscript. To do that, these should first be propagated forward and combined at the current node of the current activation layer. Considering the i-th node of the k-th activation layer where $k \in \{1, 2, ..., L-1\}$ and the weight matrix $\mathbf{W}_k$ connecting this layer with the previous $(k-1)$-th layer, this is done by taking j-th embedding vector of the previous layer in turn, multiplying it by the corresponding weight $W_{i,j}^k$ and then summing over j. Mathematically, the resulting vector which contains information about the combined embedding vectors of the previous layer, denoted by $\boldsymbol{E}_i^k$, can be computed as:

$$\boldsymbol{E}_i^k = \sum_j W_{i,j}^k \cdot \boldsymbol{\mu}_j^{k-1} \tag{B.4}$$

where it is very important to note that, in contrast to the conventional pass through the original neural network, the bias of the i-th node of the k-th activation layer is not applied when computing $\boldsymbol{E}_i^k$.

Once $\boldsymbol{E}_i^k$ is computed, the final processing step involves considering the amount of information which passes through the activation node. By looking at the equation (B.1) from Subsection B.1.2, it can be seen that as $u_i^k \to 0$, i.e. as the node tends to its blocking state, the slope of this equation, given by $\frac{u_i^k}{u_i^k - l_i^k}$, tends to 0. On the other hand, as $l_i^k \to 0$, i.e. as the node tends to its passing state, the slope tends to 1. When the node is in its ambiguous state, however, the slope lies in the range $(0, 1)$.

Hence, the slope of (B.1), which will be denoted by α_i^k, is a suitable and well-defined measure of information passing through the node. Using the method from Lu and Kumar [2020a] and denoting the parameters of the second stage network as $\boldsymbol{\theta}_1^1$ and its output as $\boldsymbol{N}_i^k$, the operation of F_{act-nb} can be mathematically defined as follows:

$$\boldsymbol{N}_i^k = F_{act-nb}\Big(\Big[\alpha_i^k \cdot (\boldsymbol{E}_i^k)^T \quad (\alpha_i^k)' \cdot (\boldsymbol{E}_i^k)^T\Big]^T ; \boldsymbol{\theta}_1^1\Big) \tag{B.5}$$

where the two vectors are concatenated to form one vector of double the size and $(\alpha_i^k)'$ is defined as:

$$(\alpha_i^k)' = \begin{cases} 1 - \alpha_i^k & \text{if} \quad 0 < \alpha_i^k < 1 \\ \alpha_i^k & \text{otherwise} \end{cases} \tag{B.6}$$

The third and final stage of the network, denoted by $F_{act-com}$ and appearing on the right in Figure B.3, combines the information obtained from the local features, given by $\boldsymbol{R}_i^k$, and that from the neighbouring embedding vectors, given by $\boldsymbol{N}_i^k$, to return the updated embedding vector of a particular activation node. Denoting the parameters of this stage as $\boldsymbol{\theta}_1^2$, its operation can be defined in the following way for all $k \in \{1, 2, ..., L-1\}$:

$$\boldsymbol{\mu}_i^k = F_{act-com}\Big(\Big[(\boldsymbol{R}_i^k)^T \quad (\boldsymbol{N}_i^k)^T\Big]^T ; \boldsymbol{\theta}_1^2\Big) \tag{B.7}$$

B.2.3 Forward pass — output layer

The final forward update network is the one associated with the function F_{out}. Its operation is completely analogous to that of F_{act} since it has to take a local output feature vector as well as all the embedding vectors of the last activation layer nodes propagated forward as inputs and return an updated output embedding vector. This is achieved by a network structure involving two stages, as shown in Figure B.4. The first stage, denoted by F_{out-lf} and appearing in the top left corner

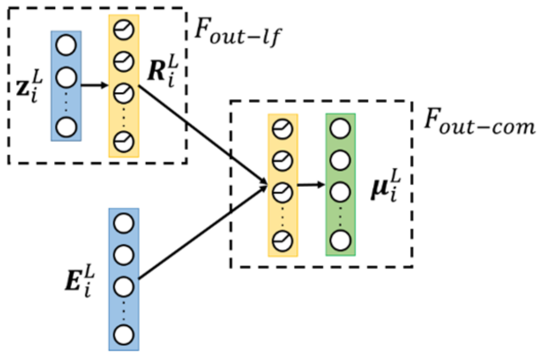


Figure B.4: Structure of the forward output embedding vector update network

of the figure, processes the local output features and has almost the same structure as the network implementing F_{act-lf} from Figure B.3. Denoting the parameters of the first stage as $\boldsymbol{\theta}_2^0$ and its output as $\boldsymbol{R}_i^L$, its operation is defined as:

$$\boldsymbol{R}_i^L = F_{out-lf}(\mathbf{z}_i^L; \boldsymbol{\theta}_2^0) \tag{B.8}$$

Since the ReLU activation function is not applied to the output nodes of the original neural network, there is no need for either a conditional statement in the equation above or for further processing applied to the embedding vectors of the last activation layer once they are propagated forward and combined to form the vector $\boldsymbol{E}_i^L$. This vector is obtained in exactly the same way as before according to equation (B.4). Hence, the second and last stage of this update network, denoted by $F_{out-com}$ and shown on the right in Figure B.4, concatenates $\boldsymbol{R}_i^L$ directly with $\boldsymbol{E}_i^L$ to produce an updated embedding vector at its output. Denoting the parameters of this stage as $\boldsymbol{\theta}_2^1$, its operation can be defined in the following way:

$$\boldsymbol{\mu}_i^L = F_{out-com}\left(\begin{bmatrix} (\boldsymbol{R}_i^L)^T & (\boldsymbol{E}_i^L)^T \end{bmatrix}^T ; \boldsymbol{\theta}_2^1 \right) \tag{B.9}$$

B.2.4 Backward pass — hidden layer

The first neural network which performs the backward update on the embedding vectors is the one corresponding to the update function B_{act}. The operation of

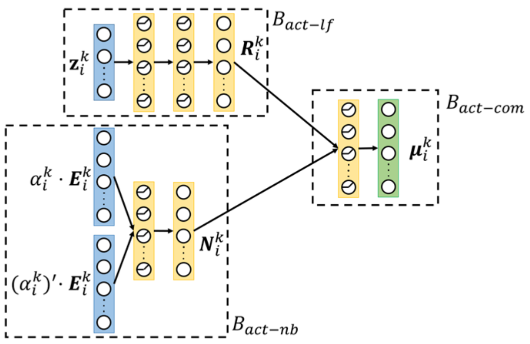


Figure B.5: Structure of the backward activation embedding vector update network

this function is very similar to that of F_{act} in that it also takes a local feature vector of a particular activation node as one of the inputs and returns an updated embedding vector for this node. The second input, however, unlike in the case of F_{act}, should consist of the propagated and combined embedding vectors of the next rather than previous layer nodes. In addition, due to the features selected in Subsection B.1.2 for the activation nodes in this project being slightly different to the features selected in Lu and Kumar [2020a], the design of the network implementing B_{act} will also be a bit different. This network, similarly to F_{act}, has three stages to it which are all shown in Figure B.5.

The first stage of this network, denoted by B_{act-lf}, is completely analogous to F_{act-lf}. Denoting the parameters of B_{act-lf} as $\boldsymbol{\theta}_3^0$ and its output, similarly to the case of the first stages of the forward update networks, as $\boldsymbol{R}_i^k$ where $k \in \{1, 2, ..., L-1\}$, the operation of the first stage can be defined as follows:

$$\boldsymbol{R}_i^k = \begin{cases} B_{act-lf}(\mathbf{z}_i^k; \boldsymbol{\theta}_3^0) & \text{if} \quad \beta_i^k > 0 \\ \mathbf{0} & \text{otherwise} \end{cases} \tag{B.10}$$

where the condition for the first statement is the same as in case of equation (B.3) and implies that the pass through the network defined by B_{act-lf} is only made if the activation node under consideration is ambiguous whereas otherwise the output from this stage is set to the zero vector of the appropriate size.

The structure of the second stage, denoted by B_{act-nb}, is exactly the same as the one of F_{act-nb}. The only difference is that to form the vector $\boldsymbol{E}_i^k$, the embedding vectors of all the nodes of the next layer now have to be propagated backwards and combined. Using the same notation as in Subsection B.2.2 and again noting that bias is not applied when propagating embedding vectors, $\boldsymbol{E}_i^k$ is obtained as:

$$\boldsymbol{E}_i^k = \sum_j W_{i,j}^{k+1} \cdot \boldsymbol{\mu}_i^{k+1} \tag{B.11}$$

Denoting the parameters of the second stage of the network as $\boldsymbol{\theta}_3^1$ and its output, similarly to the case of the output of F_{act-nb}, as $\boldsymbol{N}_i^k$, its operation is then defined in the same way as the operation of F_{act-nb}:

$$\boldsymbol{N}_i^k = B_{act-nb}\Big(\Big[\alpha_i^k \cdot (\boldsymbol{E}_i^k)^T \quad (\alpha_i^k)' \cdot (\boldsymbol{E}_i^k)^T\Big]^T ; \boldsymbol{\theta}_3^1\Big) \tag{B.12}$$

where α_i^k and $(\alpha_i^k)'$ are defined in the same way as before.

The third and final stage of this network, denoted by $B_{act-com}$, is again the same as the one associated with $F_{act-com}$. Denoting its parameters as $\boldsymbol{\theta}_3^2$, $B_{act-com}$ is defined can be the following way or $k \in \{1, 2, ..., L-1\}$:

$$\boldsymbol{\mu}_i^k = B_{act-com}\Big(\Big[(\boldsymbol{R}_i^k)^T \quad (\boldsymbol{N}_i^k)^T\Big]^T ; \boldsymbol{\theta}_3^2\Big) \tag{B.13}$$

B.2.5 Backward pass — input layer

The second network which performs the backward update on the embedding vectors and concludes one round of updates is the one associated with the function B_{inp}. In the same way the network implementing F_{out} was similar to the one implementing F_{act}, this network is very similar to the previous one implementing B_{act}. It should take a local feature vector of an input node together with all the embedding vectors of the first activation layer nodes propagated backwards as before and return an

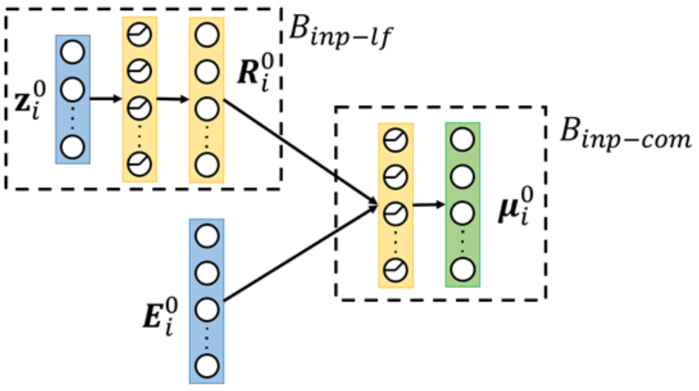


Figure B.6: Structure of the backward input embedding vector update network

updated embedding vector for this input node. The structure of the network implementing B_{inp} consists of two stages, as shown in Figure B.6, and can be observed to be almost identical to the one which appeared in Figure B.4.

The first stage of the above neural network, denoted by B_{inp-lf}, processes the local feature vector of the given input node in a similar way to all the networks mentioned above. Denoting the parameters of this stage as $\boldsymbol{\theta}_4^0$ and its output as $\boldsymbol{R}_i^0$, the operation of B_{inp-lf} can be defined in the following way:

$$\boldsymbol{R}_i^0 = B_{inp-lf}(\mathbf{z}_i^0; \boldsymbol{\theta}_4^0) \tag{B.14}$$

Again, since the ReLU activation functions are not applied at the input nodes, there is no need for either a conditional statement in the equation above or for further processing of the vector $\boldsymbol{E}_i^0$ of the propagated backwards and combined embedding vectors of the first activation layer nodes. The vector $\boldsymbol{E}_i^0$ is obtained in the same way as before in accordance with equation (B.11). The second stage of B_{inp} then becomes identical to the second stage of F_{out} so the vectors $\boldsymbol{R}_i^0$ and $\boldsymbol{E}_i^0$ can be directly concatenated and passed through this stage, denoted by $B_{inp-com}$, to obtain the updated embedding vector of a particular input node. Denoting the parameters of this stage as $\boldsymbol{\theta}_4^1$, $B_{inp-com}$ is defined as:

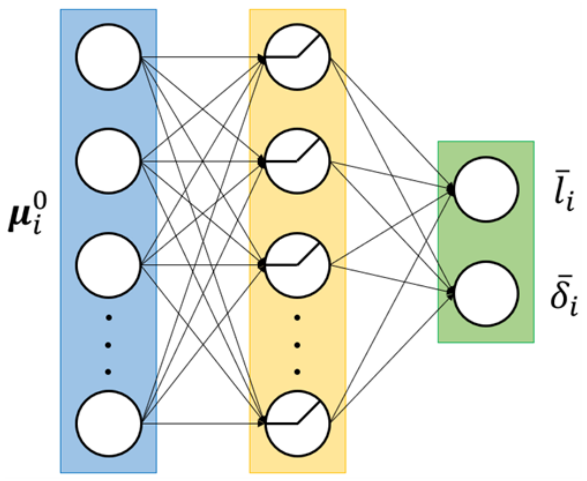


Figure B.7: Structure of the bounds update network

$$\boldsymbol{\mu}_i^0 = B_{inp-com}\Big(\Big[(\boldsymbol{R}_i^0)^T \quad (\boldsymbol{E}_i^0)^T\Big]^T; \boldsymbol{\theta}_4^1\Big) \tag{B.15}$$

B.2.6 Output Layer

The last auxiliary neural network involved in the GNN framework is the one which implements the function g by computing the new lower bound $\bar{l}_i$ and $\bar{\delta}_i$, the offset from it, which defines the new upper bound for each input node. By doing so, g can potentially greatly reduce the search space where a valid PGD attack is most likely to be and thus make the future PGD attacks more likely to succeed. The network implementing g, the structure of which is shown in Figure B.7, has only one stage which takes an embedding vector of a particular input node and returns a two-dimensional vector of $\bar{l}_i$ and $\bar{\delta}_i$.

C

Neural Network Branch-and-Bound for Neural Network Verification

Contents

C.1 Implementation of the Message Passing Algorithm

In our case g is a multilayer perceptron (MLP), which is made up of a series of linear layers Θ_i and non-linear activations σ. We have the following set of trainable parameters:

$$\Theta_0 \in \mathbb{R}^{d\times p}, \quad \Theta_1, \dots, \Theta_{T_1} \in \mathbb{R}^{p\times p}, \quad \mathbf{b}_0, \dots, \mathbf{b}_{T_1} \in \mathbb{R}^p. \tag{C.1}$$

Given a feature vector $\mathbf{f}$ we compute the following set of vectors:

$$\boldsymbol{\mu}^0 = \Theta_0 \cdot \mathbf{f} + b_0, \quad \boldsymbol{\mu}^{l+1} = \Theta_{l+1} \cdot \text{relu}(\boldsymbol{\mu}^l) + \mathbf{b}_{l+1}. \tag{C.2}$$

We initialize the embedding vector to be $\boldsymbol{\mu} = \boldsymbol{\mu}^{T_1}$, where $T_1 + 1$ is the depth of the MLP.

The hyper-parameters for the GNN computation of the duals are the depth of the MLP (T_1), how many forward and backward passes we run (T_2), and the embedding size (p).

The forward pass consists of a weighted sum of three parts: the first term is the current embedding vector, the second is the embedding vector of the previous layer passed through the corresponding linear or convolutional filters, and the third is the average of all neighbouring embedding vectors:

$$\boldsymbol{\mu}'_{i[j]} = \text{relu}\left(\Theta_1^{for}\boldsymbol{\mu}_{i[j]} + \Theta_2^{for}\left(W_i\boldsymbol{\mu}_{i-1} + \mathbf{b}_{i-1}\right)[j] + \Theta_3^{for}\left(\sum_{k\in N(j)} \boldsymbol{\mu}_{i-1[k]}/Q_{i[j]}\right)[j]\right). \tag{C.3}$$

Both the second and the third term can be implemented using existing deep learning functions. Similarly, we perform a backward pass as follows:

$$\boldsymbol{\mu}_{i[j]} = \text{relu}\left(\Theta_1^{back}\boldsymbol{\mu}'_{i[j]} + \Theta_2^{back}(W_{i+1}^T\left(\boldsymbol{\mu}'_{i+1} - \mathbf{b}_{i+1}\right))[j] + \Theta_3^{back}\left(\sum_{k\in N'(j)} \boldsymbol{\mu}'_{i+1[k]}/Q'_{i[j]}\right)[j]\right). \tag{C.4}$$

Here $\Theta_1^{for}, \Theta_2^{for}, \Theta_3^{for}, \Theta_1^{back}, \Theta_2^{back}, \Theta_3^{back} \in \mathbb{R}^{p \times p}$ are all learnable parameters. To ensure better generalization performance to unseen neural networks with a different network architecture we include normalization parameters Q and Q'. These are matrices whose elements are the number of neighbouring nodes in the previous and following layer respectively for each node. We repeat this process of running a forward and backward pass T_2 times. The high-dimensional embedding vectors are now capable of expressing the state of the corresponding node taking the entire problem structure into consideration as they are directly influenced by every single other node, even if we set $T_2 = 1$.

C.2 Extending our Method beyond Piece-wise Linearities

The Branch-and-Bound verification method with ReLU splitting is complete as the leaf nodes in the BaB tree form a convex problem. We note that in practice we run our method with a given timeout which makes our method incomplete. As complete neural verification is NP-hard it is unlikely that there exists an efficient algorithm that is complete for all cases when using a short timeout. Other versions of BaB including BaB with input domain splitting is also complete, as in most cases all the subdomains are small enough to make all ReLU nodes non-ambiguous thus making the problem convex and easy to solve. In the worst case we end up evaluating the network at every single input point which is possible as there is a finite number of input points due to floating-point arithmetic.

Our bounding method can be extended to other non-linearities such as the sigmoid activation or the hyperbolic tangent. De Palma et al. [2021] and Zhang et al. [2018] provide suitable relaxations for both functions given a pair of lower and upper bounds which allows us to form a dual formulation. A GNN can then estimate better dual directions as it has done for the ReLU case. Similar to our bounding method, our branching approach can also be extended to work on all piece-wise convex/concave functions. Rather than splitting the ReLU into 2 linear pieces when branching on a particular node we split it into k convex or concave pieces. We note that from a theoretical point of view the method is not complete anymore as the composition of a convex and a concave function is not guaranteed to be either convex or concave. However, if we allow repeatedly splitting on the same non-linear activations then our method stays complete.

In practice we do not focus on whether verification methods are complete as all methods become incomplete when using a short timeout. Instead we care about the efficiency of different methods shown empirically.