Generating Adversarial Examples with Graph Neural Networks

Florian Jaeckle and M. Pawan Kumar
Department of Engineering Science
University of Oxford
{florian,pawan}@robots.ox.ac.uk

Abstract

Recent years have witnessed the deployment of adversarial attacks to evaluate the robustness of Neural Networks. 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; we call it AdvGNN. Our GNN architecture closely resembles the network we wish to attack. 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, reducing the time required to generate adversarial examples with small perturbation norms by over 65% while also achieving good generalization performance on unseen networks.

1 Introduction

It is common practice to evaluate Neural Networks on their robustness to 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; Madry et al., 2018). 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 & Fischer, 2017; Poursaeed 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. 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. 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. 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 our case, the size of the allowed perturbation is deliberately chosen for each element in the dataset leading to a very high level of difficulty allowing for a more efficient and meaningful comparison of different adversarial attacks. 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.

2 Related Work

Many optimization based attacks have been proposed that given an allowed perturbation aim to find an adversarial example that gets misclassified with the highest level of confidence: FGSM (Goodfellow et al., 2015), I-FGSM (Kurakin et al., 2016), MI-FGSM (Dong et al., 2018), and PGD (Madry et al., 2018). Other methods aim to find the minimum perturbation required to change the prediction of the network (Szegedy et al., 2013; Carlini & Wagner, 2017; Moosavi-Dezfooli et al., 2016). 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 such as ATNs (Baluja & Fischer, 2017), GAPs (Poursaeed et al., 2018), and AdvGAN Xiao et al. (2018). 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 & Kumar, 2020) and to estimate better bounds (Dvijotham et al., 2018; Gowal et al., 2019), but to the best of our knowledge they have not yet been used to generate adversarial examples.

3 Problem Definition

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. 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 $\mathbf{x}'$ that maximizes

$$\max_{\mathbf{x}' \in \mathcal{B}(\mathbf{x},\epsilon)} L(\mathbf{x}', y, \hat{y}) = f(\mathbf{x}')_{\hat{y}} - f(\mathbf{x}')_{y}, \tag{1}$$

where $\mathcal{B}(\mathbf{x}, \epsilon) := \{x' \mid d(\mathbf{x}, \mathbf{x}') \leq \epsilon\}$ is an ϵ-sized norm-ball around $\mathbf{x}$. We refer to L as the adversarial loss from now on. If $L(\mathbf{x}', y, \hat{y}) > 0$ then $\mathbf{x}'$ is considered an adversarial example. PGD attack (Madry et al., 2018) aims to solve (1) by iteratively running Projected Gradient Descent on the negative adversarial loss using the sign of the gradient:

$$\mathbf{x}^{t+1} = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)} \left(\mathbf{x}^t + \alpha \operatorname{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}', y, \hat{y})) \right). \tag{2}$$

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 thus leading to a more efficient update step.

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. 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. We propose to use a GNN for the efficient computation of adversarial examples. In what follows, we first describe the GNN's main components, before explaining how the forward-backward passes are implemented, and how to transform the output of the GNN into a new direction.

4.1 GNN Components

Nodes. We create a node $\boldsymbol{v}_k[i]$ 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 $\boldsymbol{v}_k[i]$ 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.

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 use a learned function g to compute a corresponding p-dimensional embedding vector $\boldsymbol{\mu}_k[i] := g(\mathbf{f}_k[i]; \Theta^{init}) \in \mathbb{R}^p$. In our case g is a simple multilayer perceptron (MLP).

4.2 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. During the forward update, for $k = 1, \ldots, L-1$, we have, for all possible i,

$$\boldsymbol{\mu}_k[i] \longleftarrow F(\mathbf{f}_k[i], \boldsymbol{\mu}_{k-1}, \boldsymbol{\mu}_k[i], e^k; \Theta_1). \tag{3}$$

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

$$\boldsymbol{\mu}_k[i] \longleftarrow B(\mathbf{f}_k[i], \boldsymbol{\mu}_{k+1}, \boldsymbol{\mu}_k[i], e^{k+1}; \Theta_2). \tag{4}$$

4.3 Update Step

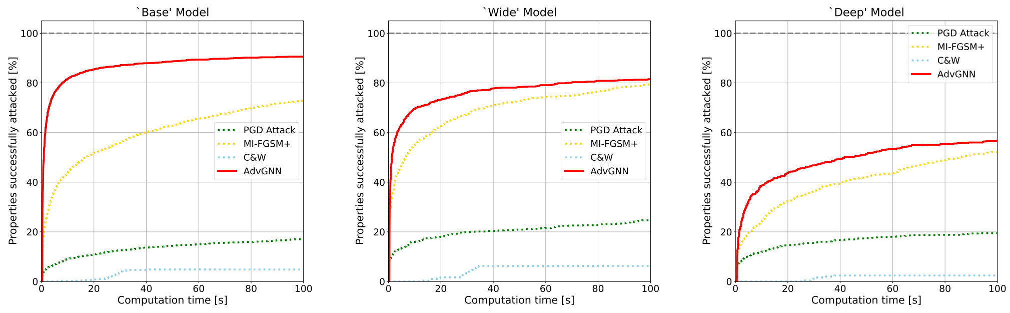
Finally, we need to transform the p-dimensional embedding vector of the input layer to get a new direction $\tilde{\mathbf{x}}$. We use a learnt function $g^{out} : \mathbb{R}^q \to \mathbb{R}^d$ to get $\tilde{\mathbf{x}} = g^{out}(\boldsymbol{\mu}_0; \Theta^{out})$. As it is unlikely that the GNN manages to output a new ascent direction that will lead us directly to the global optimum of equation (1), 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:

$$\mathbf{x}^{t+1} = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)} \left(\mathbf{x}^t + \alpha \ \mathrm{sgn}(\tilde{\mathbf{x}}) \right). \tag{5}$$

For a more detailed description of the GNN architecture see Appendix C.

4.4 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 ($\mathbf{x}$); 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 ($W, \mathbf{b}$). 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 and also solve a standard relaxation of the robustness problem (see Appendix C.1 and C.2). Finally, we generate s different starting points which we sample uniformly at random from the input domain $\mathcal{B}(\mathbf{x}, \epsilon)$. In order for the training procedure to closely resemble its behaviour at inference time, it is crucial to



(a) 'Base' model (b) 'Wide' model (c) 'Deep' model

Figure 1: Cactus plots for experiments on the 'Base' model (left), 'Wide' model (middle) and the 'Deep' model (right). For each, we compare the different attacks by plotting the percentage of successfully attacked images as a function of runtime. AdvGNN beats all baselines on all three models for any chosen timeout value.

train the GNN using a loss function that takes into account the adversarial loss across a large number of iterations K using a decay factor γ. Given the i-th training sample $d_i \in \mathcal{D}$, and the j-th initial starting point we define the loss $\mathcal{L}_{i,j}$ to be:

$$\mathcal{L}_{i,j} = -\sum_{t=1}^{K} L(\mathbf{x}^{i,j,t}, y^i, \hat{y}^i) * \gamma^t. \tag{6}$$

We sum over the individual loss values corresponding to each data point and each initial starting point to get the final training objective $\mathcal{L} = \sum_{i=1}^{|D|} \sum_{j=1}^{s} \mathcal{L}_{i,j}$. In our experiments we train the GNN using the Adam optimizer (Kingma & Ba, 2015) with a small weight decay. The training procedure is described in greater detail in Appendix D.

5 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 run experiments on the dataset described in Appendix B. We evaluate our methods by comparing it against PGD-Attack, MI-FGSM+, a modified version of MI-FGSM, and the Carlini and Wagner attack (C&W). We run an extensive hyper-parameter search for all three baselines which along with a more detailed description of each method can be found in Appendix F.

Results - 'Base' Model. We run all four methods on the 'Base' model with a timeout of 100 seconds and record the percentage of properties successfully attacked as a function of time (Figure 1a). 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. AdvGNN has not seen this network during training and before running these experiments. 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 (Figure 1c). The AdvGNN parameters have been fine-tuned on this model for 15 minutes to achieve better results (further details in Appendix D). 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 1c shows that AdvGNN is still the best performing method even if we pick a shorter timeout of less than 100 seconds.

6 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.

References

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

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*, pp. 370–379. PMLR, 2020.

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

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*, pp. 9185–9193, 2018.

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*, 2018.

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

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*, pp. 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.

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

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

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

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

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.

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*, pp. 2574–2582, 2016.

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

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.

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

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*, pp. 3905–3911, 2018.

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

A Network Architectures

We now describe the three models used in this work in greater detail. They have been trained robustly on the CIFAR-10 dataset (Krizhevsky et al., 2009) using the method introduced by Wong & Kolter (2018) to achieve robustness against l_∞ perturbations of size up to $\epsilon = 8/255$ (the amount typically considered in empirical works). The 'Base' and the 'Wide' model both have two convolutional layers, followed by two fully connected ones. The 'Deep' model has two further convolutional layers. All three networks use ReLU activations and all three models have been used in previous work (Lu & Kumar, 2020; Bunel et al., 2020).

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 1: Network Architectures.

B 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 & 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 exist a large number of different adversarial examples. In the first case, we don't 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. All three are trained robustly 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). Our dataset is inspired by the work of Lu & Kumar (2020) who created a verification dataset to compare defense methods on the same three models. The different network architectures are explained in greater detail in Appendix A.

For each of the three models we generate properties to attack, using the method described in Algorithm 1. 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 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.

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.

Algorithm 1 Generating Dataset

```
1: function GENERATING_DATASET($f, D, \eta, R, PGD\_hparams$)
2:     Provided: a trained network $f : \mathbb{R}^d \mapsto \mathbb{R}^m$, a set $D$ of $N$ pairs of images and their respective classes $(\mathbf{x}^i, y^i)$, a confidence parameter $\eta$, a restart parameter $R$, as well as parameters for PGD.
3:     for $i = 1, \ldots, N$ do:
4:         if $\arg\max f(\mathbf{x}^i)! = y^i$ then
5:             continue    ▷ If the network misclassifes the image, skip to the next one
6:         end if
7:         $\hat{y}^i \leftarrow$ random number from $\{0, \cdots, m-1\} \setminus \{y^i\}$ ▷ Pick a random incorrect class as target
8:         $\mathbf{l} \leftarrow 0$    ▷ highest perturbation value for which we have failed to find an adversarial example
9:         $\mathbf{u} \leftarrow 0.5$    ▷ lowest perturbation value for which we have found an adversarial example
10:        while $\mathbf{u} - \mathbf{l} \geq \eta$ do
11:            $\epsilon^i \leftarrow \frac{\mathbf{l}+\mathbf{u}}{2}$
12:            for $j = 1, \ldots, R$ do
13:                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
14:                if attack successful then
15:                    Break
16:                end if
17:            end for
18:            if found adversarial example then
19:                $\mathbf{u} \leftarrow \epsilon^i$    ▷ Update $\mathbf{u}$ as $\epsilon^i$ is now the lowest perturbation for which we have found an adversarial example
20:            else
21:                $\mathbf{l} \leftarrow \epsilon^i$ ▷ Update $\mathbf{l}$ as $\epsilon^i$ is now the highest perturbation for which we have failed to find an adversarial example
22:            end if
23:        end while
24:        Record $(\mathbf{x}^i, y^i, \hat{y}^i, \epsilon^i)$
25:    end for
26: end function
```

C 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.

C.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

$$\hat{\mathbf{x}}_{i+1} = W^{i+1}\mathbf{x}_i + \mathbf{b}^{i+1}, \qquad \text{for } i = 0, \ldots, L-1, \tag{7}$$

$$\mathbf{x}_i = \sigma(\hat{\mathbf{x}}_i), \qquad \text{for } i = 1, \ldots, L-1. \tag{8}$$

The adversarial problem can then be written as

$$\min \hat{\mathbf{x}}_L[y] - \hat{\mathbf{x}}_L[\hat{y}] \tag{9}$$

$$\hat{\mathbf{x}}_{i+1} = W^{i+1}\mathbf{x}_i + \mathbf{b}^{i+1}, \qquad \text{for } i = 0, \ldots, L-1, \tag{10}$$

$$\mathbf{x}_i = \sigma(\hat{\mathbf{x}}_i), \qquad \text{for } i = 1, \ldots, L-1, \tag{11}$$

$$\mathbf{x}_0 \in \mathcal{C} \subseteq \mathbb{R}^d \tag{12}$$

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:

$$\min \hat{\mathbf{x}}_k[j] \tag{13}$$

$$\hat{\mathbf{x}}_{i+1} = W^{i+1}\mathbf{x}_i + \mathbf{b}^{i+1}, \qquad \text{for } i = 0, \ldots, k-1, \tag{14}$$

$$\mathbf{x}_i = \sigma(\hat{\mathbf{x}}_i), \qquad \text{for } i = 1, \ldots, k-1, \tag{15}$$

$$\mathbf{x}_0 \in \mathcal{C} \subseteq \mathbb{R}^d. \tag{16}$$

We solve this using the method by Wong & 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.

C.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 equation 9 is strictly positive.

We now summarize the work of Bunel et al. (2020) who solve this problem using standard relaxations. First they relax the non-linear ReLU activation functions using the so-called Planet relaxation (Ehlers, 2017) 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:

$$cvx\_hull_\sigma(\hat{\mathbf{z}}_k, \mathbf{x}_k, \mathbf{l}_k, \mathbf{u}_k) \equiv \begin{cases} \mathbf{x}_k[i] \geq 0 \quad \mathbf{x}_k[i] \geq \hat{\mathbf{z}}_k[i] & \\ \mathbf{x}_k[i] \leq \frac{\mathbf{u}_k[i](\hat{\mathbf{z}}_k[i]-\mathbf{l}_k[i])}{\mathbf{u}_k[i]-\mathbf{l}_k[i]} & \text{if } \mathbf{l}_k[i] < 0 \text{ and } \mathbf{u}_k[i] > 0 \\ \mathbf{x}_k[i] = 0 & \text{if } \mathbf{u}_k[i] \leq 0 \\ \mathbf{x}_k[i] = \hat{\mathbf{z}}_k[i] & \text{if } \mathbf{l}_k[i] \geq 0. \end{cases} \tag{17}$$

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

$$\mathcal{P}_0(\mathbf{x}_0, \hat{\mathbf{z}}_1) \equiv \begin{cases} \mathbf{x}_0 \in C \\ \hat{\mathbf{z}}_1 = W_1\mathbf{x}_0 + \mathbf{b}_1 \end{cases} \qquad \mathcal{P}_k(\hat{\mathbf{z}}_k, \hat{\mathbf{z}}_{k+1}) \equiv \begin{cases} \exists \mathbf{x}_k \text{ s.t.} \\ \mathbf{l}_k \leq \hat{\mathbf{z}}_k \leq \mathbf{u}_k \\ cvx\_hull_\sigma(\hat{\mathbf{z}}_k, \mathbf{x}_k, \mathbf{l}_k, \mathbf{u}_k) \\ \hat{\mathbf{z}}_{k+1} = W_{k+1}\mathbf{x}_k + \mathbf{b}_{k+1}. \end{cases} \tag{18}$$

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

$$\min_{\mathbf{x},\hat{\mathbf{z}}} \hat{\mathbf{z}}_n \text{ s.t. } \mathcal{P}_0(\mathbf{x}_0, \hat{\mathbf{z}}_1); \mathcal{P}_k(\hat{\mathbf{z}}_k, \hat{\mathbf{z}}_{k+1}) \text{ for } k \in [1, \dots, L-1]. \tag{19}$$

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 equation 9 is strictly positive, then it follows that equation 9 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. (2020) we will use the Lagrangian decomposition Guignard & Kim (1987). To this end, we first create two copies $\hat{\mathbf{z}}_{A,k}, \hat{\mathbf{z}}_{B,k}$ of each variable $\hat{\mathbf{z}}_k$:

$$\begin{aligned} \min_{\mathbf{x},\hat{\mathbf{z}}} \hat{\mathbf{z}}_{A,n} \text{ s.t. } & \mathcal{P}_0(\mathbf{x}_0, \hat{\mathbf{z}}_{A,1}); \mathcal{P}_k(\hat{\mathbf{z}}_{B,k}, \hat{\mathbf{z}}_{A,k+1}) && \text{for } k \in [1, \dots, L-1] \\ & \hat{\mathbf{z}}_{A,k} = \hat{\mathbf{z}}_{B,k} && \text{for } k \in [1, \dots, L-1]. \end{aligned} \tag{20}$$

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

$$\begin{aligned} q(\boldsymbol{\rho}) = \min_{\mathbf{x},\hat{\mathbf{z}}} \quad & \hat{\mathbf{z}}_{A,n} + \sum_{k=1,\dots,n-1} \boldsymbol{\rho}_k^\top (\hat{\mathbf{z}}_{B,k} - \hat{\mathbf{z}}_{A,k}) \\ \text{s.t.} \quad & \mathcal{P}_0(\mathbf{x}_0, \hat{\mathbf{z}}_{A,1});\ \mathcal{P}_k(\hat{\mathbf{z}}_{B,k}, \hat{\mathbf{z}}_{A,k+1}) \text{ for } k \in [1, \dots, L-1]. \end{aligned} \tag{21}$$

Solving the Relaxation using Supergradient Ascent We solve the dual problem equation 21 using the supergradient ascent method proposed by Bunel et al. (2020). 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 $\mathbf{x}^{lp}$.

C.3 Node Features

For each node $\boldsymbol{v}_k[i]$ 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:

$$\mathbf{f}_0[i] := \left(\mathbf{x}^t[i], \nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y')[i], \mathbf{l}_0[i], \mathbf{u}_0[i], \mathbf{x}^{lp}[i]\right)^\top, \tag{22}$$

and for the hidden and final layers as:

$$\mathbf{f}_k[i] := \left(\mathbf{l}_k[i], \mathbf{u}_k[i], \boldsymbol{\rho}_k[i]\right)^\top. \tag{23}$$

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 (§C.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 §C.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.

C.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:

$$\boldsymbol{\mu}_k[i] := g(\mathbf{f}_k[i]). \tag{24}$$

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:

$$\Theta_0^{inp} \in \mathbb{R}^{5\times p}, \quad \Theta_0 \in \mathbb{R}^{3\times p} \quad \Theta_1^{inp}, \dots, \Theta_{T_1}^{inp},\ \Theta_1, \dots, \Theta_{T_1} \in \mathbb{R}^{p\times p} \tag{25}$$

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

$$\boldsymbol{\mu}_0^0 = \text{relu}(\Theta_0^{inp} \cdot \mathbf{f}_0), \qquad \boldsymbol{\mu}_0^{l+1} = \text{relu}(\Theta_{l+1}^{inp} \cdot \boldsymbol{\mu}_0^l), \qquad \text{for } l = 1, \ldots, T_1 - 1 \tag{26}$$

$$\boldsymbol{\mu}_k^0 = \text{relu}(\Theta_0 \cdot \mathbf{f}_k), \qquad \boldsymbol{\mu}_k^{l+1} = \text{relu}(\Theta_{l+1} \cdot \boldsymbol{\mu}_k^l), \qquad \text{for } l = 1, \ldots, T_1 - 1;\ k = 1, \ldots, L. \tag{27}$$

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.

C.5 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:

$$\boldsymbol{\mu}'_k[i] = \text{relu}\left(\Theta_1^{for}\boldsymbol{\mu}_k[i] + \Theta_2^{for}\left(W_k\boldsymbol{\mu}_{k-1} + \mathbf{b}_{k-1}\right)[i] + \Theta_3^{for}\left(\sum_{j\in N(i)} \boldsymbol{\mu}_{k-1}[j]/Q_{k+1}[j]\right)[i]\right). \tag{28}$$

Similarly, we perform a backward pass as follows:

$$\begin{aligned}\boldsymbol{\mu}_k[i] = \text{relu}\Bigg(&\Theta_1^{back}\boldsymbol{\mu}'_k[i] + \Theta_2^{back}(W_{k+1}^T\left(\boldsymbol{\mu}'_{k+1} - \mathbf{b}_{k+1}\right))[i] \\ &+ \Theta_3^{back}\left(\sum_{j\in N'(i)} \boldsymbol{\mu}'_{k+1}[j]/Q'_{k+1}[j]\right)[i]\Bigg).\end{aligned} \tag{29}$$

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. Both equation 28 and equation 29 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 $T_2 = 1$.

C.6 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 Θ^{out} to get:

$$\tilde{\mathbf{x}} = \Theta^{out} \cdot \boldsymbol{\mu}_0. \tag{30}$$

Ideally the GNN would output a new ascent direction that will lead us directly to the global optimum of equation (1). 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:

$$\mathbf{x}^{t+1} = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)}\left(\mathbf{x}^t + \alpha\ \text{sgn}(\tilde{\mathbf{x}})\right). \tag{31}$$

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.

D 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 & Kolter (2018) which is explained in greater detail in Appendix C.1. We further solve a standard relaxation of the robustness problem via methods from the verification literature equation C.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:

$$\mathcal{L}_{i,j} = -\sum_{t=1}^{K} L(\mathbf{x}^{i,j,t}, y^i, \hat{y}^i) * \gamma^t. \tag{32}$$

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 $\mathcal{L}$:

$$\mathcal{L} = \sum_{i=1}^{|D|} \sum_{j=1}^{s} \mathcal{L}_{i,j}. \tag{33}$$

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 (§33) with a horizon of 40 and with decay factor $\gamma = 0.9$. The training loss function is minimized using the Adam optimizer (Kingma & 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.

E 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).*

FGSM aims to generate an adversarial example with the following update step:

$$\mathbf{x}' = \mathbf{x} + \epsilon \ \text{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}', y, \hat{y})). \tag{34}$$

Let Θ_0 be the zero-matrix with non-zero elements $\Theta_0[1,4] = 1$, $\Theta_0[2,4] = -1$. Moreover, setting $T_1 = 1$, $\Theta_1 = \mathbb{1}$ and $\mathbf{b}_0 = \mathbf{b}_1 = \mathbf{0}$, we get

$$\mathbf{f}_0[i] := \Big(\mathbf{x}^t, \nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y'), \mathbf{l}_k[i], \mathbf{u}_k[i], \mathbf{x}_k^{lp}[i]\Big)^\top, \tag{35}$$

$$\boldsymbol{\mu}_k^0 = \big(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y'), -\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y'), \mathbf{0}, \dots, \mathbf{0}\big)^\top, \tag{36}$$

$$\boldsymbol{\mu} = \Big(\big(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y')\big)_+, -\big(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y')\big)_-, \mathbf{0}, \dots, \mathbf{0}\Big)^\top. \tag{37}$$

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 don't change the embedding vector. We now just need to set $\Theta^{out} = (1, -1, 0, \dots, 0)^\top$ to get the new direction:

$$\tilde{\mathbf{x}} = \Theta^{out} \cdot \boldsymbol{\mu}_0 = \big(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y')\big)_+ + \big(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y')\big)_- = \nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y'). \tag{38}$$

We now update as follows

$$\mathbf{x}^{t+1} = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)}\left(\mathbf{x}^t + \alpha \ \text{sgn}(\tilde{\mathbf{x}})\right) = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)}\left(\mathbf{x}^t + \alpha \ \text{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y'))\right). \tag{39}$$

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.

F Hyper-parameter Analysis for Baselines

F.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 $L(\mathbf{x}^t, y, \hat{y}) > 0$:

$$\mathbf{x}^{t+1} = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)}\left(\mathbf{x}^t + \alpha \ \text{sgn}(\nabla L(\mathbf{x}^t, y, \hat{y}))\right). \tag{40}$$

We need to pick optimal values for the hyper-parameters T and α. We run a hyper-parameter analysis on the validation dataset described in section §B. 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 2). 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 2: 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

F.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 2. 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 3. 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 3). 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 2 MI-FGSM (Dong et al., 2018)

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

Algorithm 3 MI-FGSM+

```
1: function MI-FGSM+($f, \mathbf{x}, y, \hat{y}, \mu, T, \alpha$)
2:     sample $\mathbf{x}^0$ from $\mathcal{B}(\mathbf{x}, \epsilon)$                                  ▷ 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:         $\mathbf{x}^{t+1} = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)} \left(\mathbf{x}^t + \alpha \cdot sgn(\mathbf{g}^{t+1})\right)$     ▷ Update the current point and project
7:     end for
8:     return $\mathbf{x}^T$
9: end function
```

Table 3: 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

F.3 Carlini and Wagner Attack

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

$$\min_{\delta} \; c \cdot h(x+\delta) + \sum_i [(\delta_i - \tau)_+], \tag{41}$$

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 4.

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 4 C&W

```
1:  function C&W ($h, \mathbf{x}, y, \hat{y}, T, c_{init}, c_{fin}, \gamma_\tau, \gamma_c, \alpha$)
2:      $c \leftarrow c_{init}$
3:      $\tau \leftarrow 1.0$
4:      while $\tau < 0.1$ and $c < c_{fin}$ do
5:
            $$\min_\delta \; c \cdot h(x+\delta) + \sum_i [(\delta_i - \tau)_+] \quad (42)$$
6:          Optimize 42 using the Adam optimizer with a learning rate of $\alpha$, and a step
    number of $T$
7:          if found a counter example with $\delta_i \leq \tau \; \forall i$ then
8:              $\tau \leftarrow \tau * \gamma_\tau$                ▷ Decay $\tau$ using the decay factor $\gamma_\tau$
9:              $c \leftarrow c * 1/2$                          ▷ Decay $c$ using factor $\gamma_c$
10:         else
11:             $c \leftarrow c * \gamma_c$
12:         end if
13:     end while
14:      return Best $\delta$ found
15: end function
```

	T	α	c_{init}	c_{fin}	γ_τ	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 4: Hyper-parameter analysis for C&W attack on the Validation Set

G Further Experimental Results

G.1 Main Experiments

All methods apart from C&W use random initialization. We therefore run every experiment in this paper three times, each time with a different random seed (using the Pytorch implementation of random seeds). We manually set the time taken to 100 if a method times out on a property. We summarize the results in Table 6 and Figure 2. We can see that even though the random seed makes a significant different for a single attack, when taking the average over the entire dataset the differences are very small. In particular, the difference between the results for the same attack with different seeds is much smaller than the difference between methods. This shows that our results are statistically significant.

	Base		Wide		Deep	
Method	Time(s)	Timeout(%)	Time(s)	Timeout(%)	Time(s)	Timeout(%)
PGD Attack	87.375	83.073	80.421	75.413	84.347	80.200
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
AdvGNN	**13.527**	**9.412**	**24.089**	**18.482**	61.703	54.133

Table 5: 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. The best performing method for each subcategory is highlighted in bold.

		'Base' Model		'Wide' Model		'Deep' Model	
Method	Seed	Time(s)	Timeout(%)	Time(s)	Timeout(%)	Time(s)	Timeout(%)
PGD Attack	2222	87.354	82.995	80.542	74.917	83.764	79.2
PGD Attack	3333	87.396	83.151	80.301	75.908	84.930	81.2
PGD Attack	4444	87.488	82.839	80.404	75.248	84.355	81.2
MI-FGSM+	2222	39.897	26.677	31.583	21.122	59.887	46.4
MI-FGSM+	3333	39.763	26.053	30.761	20.462	61.380	49.2
MI-FGSM+	4444	41.655	28.705	31.087	19.802	60.467	48.0
C&W	0	97.385	95.164	96.366	93.729	99.321	97.6
AdvGNN	2222	14.152	10.296	24.429	18.812	52.337	43.6
AdvGNN	3333	**12.937**	**8.580**	**23.501**	**17.822**	**50.054**	**42.0**
AdvGNN	4444	13.490	9.360	24.338	18.812	52.616	44.0

Table 6: 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 and the random Pytorch seeds specified. The best performing method for each subcategory is highlighted in bold. AdvGNN is the best performing method as every single run of AdvGNN beats every other run by any of the other methods on each model.

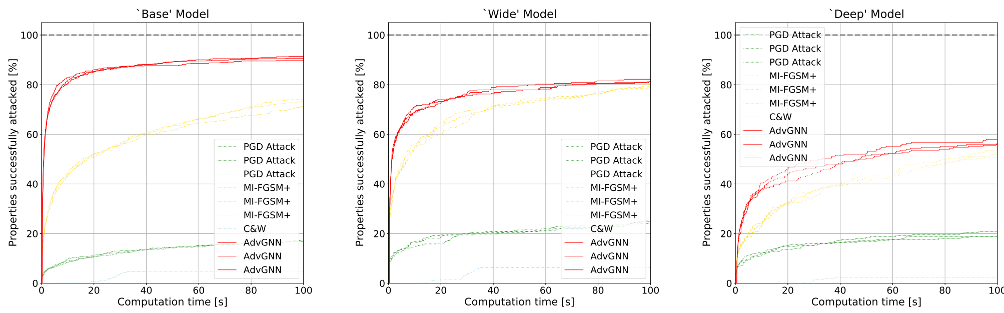


Figure 2: Cactus plots for the main datasets on the 'Base' , 'Wide' and 'Deep' models. For each, we compare the attack methods by plotting the percentage of successfully attacked images as a function of runtime.

G.2 Further Experiments on a Simpler 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 (0.01) 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. The results are summarized in Tables 7 and 8 and Figure 3. All methods manage to find adversarial examples more quickly 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. We provide a more in-depth analysis of the results on the original and the easier dataset in Appendix G.

		'Base' -Easy		'Wide' -Easy		'Deep' -Easy	
Method	Seed	Time(s)	Timeout(%)	Time(s)	Timeout(%)	Time(s	Timeout(%)
PGD Attack	2222	4.698	15.445	2.509	7.261	4.166	11.2
PGD Attack	3333	4.714	14.353	2.109	5.611	3.655	8.0
PGD Attack	4444	4.719	15.133	2.830	9.571	4.073	11.2
MI-FGSM+	2222	1.123	2.340	0.810	1.320	1.703	4.0
MI-FGSM+	3333	1.398	3.432	0.712	0.660	1.570	2.0
MI-FGSM+	4444	1.343	3.120	0.813	0.990	1.461	2.8
C&W	0	17.030	69.111	15.978	60.396	17.487	76.0
AdvGNN	2222	0.509	0.156	**0.550**	**0.330**	1.443	**0.8**
AdvGNN	3333	**0.505**	**0.000**	0.569	**0.330**	**1.351**	**0.8**
AdvGNN	4444	0.538	**0.000**	0.665	**0.330**	1.603	1.2

Table 7: 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 and the random Pytorch seeds specified. The best performing method for each subcategory is highlighted in bold.

	'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 8: 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.

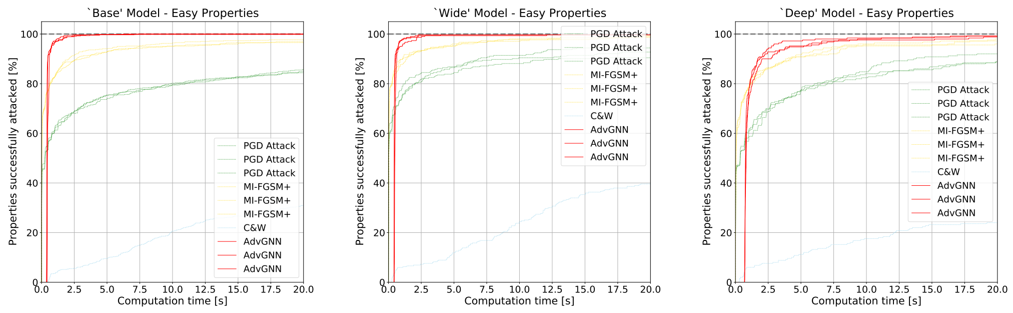
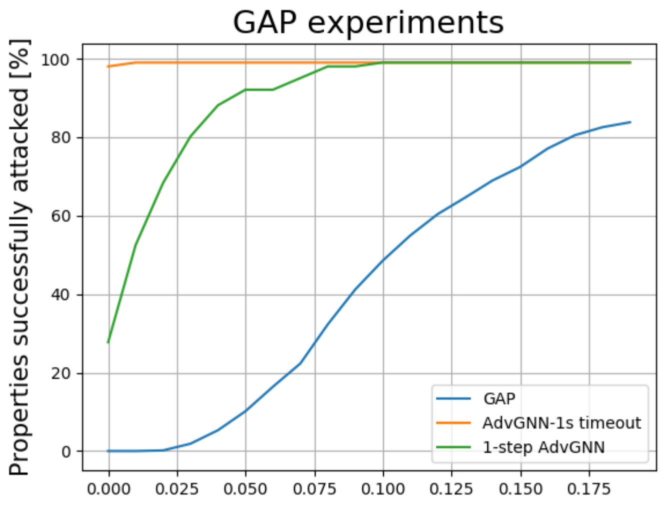


Figure 3: Cactus plots for the easy datasets on the 'Base' , 'Wide' and 'Deep' models. For each, we compare the attack methods by plotting the percentage of successfully attacked images as a function of runtime.

G.3 GAP Comparison

We compare our model against GAP, the learnt method proposed by Zhao et al. (2018). We use the image dependent targeted version of their attack. We train 10 models, one for each target class. We train using the parameters on 3000 images of the Cifar-10 training dataset on the 'Base' model described above for a total of 20 epochs, using a learning rate of 0.0002 and using the Adam optimizer.

We then test if on the easier version of our dataset on the 'Base' model. As GAPs fails to counter examples for any properties of the easy dataset described above, we run it on easier properties as well. We incrementally increase the allowed perturbation and show the percentage of successful attacks. The more the allowed perturbation increases, the easier the problem becomes and the higher the proportion of succesful attacks. As seen in Figure 4 AdvGNN clearly outperforms GAP. AdvGNN manages to find a counter example for 98% of all properties, whereas GAP fails to find a single one for the easy dataset. When increasing the allowed perturbation GAP manages to find more adversarial examples but doesn't manage to match the performance of AdvGNN. One of the main advantages of our method is that we can apply it iteratively, and thus get better results. However, even when restricting our method to a single iteration to speed up inference time, we still significantly outperform GAP.



(a) 'Base' model

Figure 4: We compare GAP, the original AdvGNN with a 1 second timeout, and a faster 1-step version of AdvGNN. We run experiments on the 'Base' model and incrementally simplify the dataset by adding a constant to the allowed perturbation (x axis) and plot the percentage or properties successfully attacked (y axis).