Generating Adversarial Examples with Graph Neural Networks (Supplementary material)

Florian Jaeckle and M. Pawan Kumar

Department of Engineering Science
University of Oxford
{florian,pawan}@robots.ox.ac.uk

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 and 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 and 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 GENERATING THE DATASET

We generate a dataset for three different models: the 'Base' model, the 'Wide' model, and the 'Deep' model. 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.

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 $i = 1, \dots, N$ 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 $j = 1, \dots, R$ 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

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

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

The adversarial problem can then be written as

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

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

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

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

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:

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

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.

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 (3) 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:

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

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

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

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

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

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 (3) is strictly positive, then it follows that (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. [2020] 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 $\hat{\mathbf{z}}_k$:

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

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

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

Solving the Relaxation using Supergradient Ascent We solve the dual problem (15) 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 $\mathbf{v}_k[i]$ 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:

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

and for the hidden and final layers as:

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

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 (§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:

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

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:

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

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

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

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

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.

D RUNNING STANDARD ALGORITHMS USING AdvGNN

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:

$$\mathbf{x}' = \mathbf{x} + \epsilon\, \text{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}', y, \hat{y})). \tag{22}$$

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

$$\mathbf{f}_0[i] := \Big(\mathbf{x}^t,\ \text{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y')), \mathbf{l}_k[i], \mathbf{u}_k[i], \mathbf{x}_k^{lp}[i]\Big)^\top, \tag{23}$$

$$\boldsymbol{\mu}_k^0 = \left(\operatorname{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y')), -\operatorname{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y')), \mathbf{0}, \dots, \mathbf{0} \right)^\top, \tag{24}$$

$$\boldsymbol{\mu} = \Big(\left(\operatorname{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y')) \right)_+, -\left(\operatorname{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y')) \right)_-, \mathbf{0}, \dots, \mathbf{0} \Big)^\top. \tag{25}$$

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 $\boldsymbol{\Theta}^{out} = (1, -1, 0, \dots, 0)^\top$ to get the new direction:

$$\tilde{\mathbf{x}} = \boldsymbol{\Theta}^{out} \cdot \boldsymbol{\mu}_0 = \left(\operatorname{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y')) \right)_+ + \left(\operatorname{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y')) \right)_- = \operatorname{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y')). \tag{26}$$

We now update as follows

$$\mathbf{x}^{t+1} = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)} \left(\mathbf{x}^t + \alpha \tilde{\mathbf{x}} \right) = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)} \left(\mathbf{x}^t + \alpha \operatorname{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}^t, y, y')) \right). \tag{27}$$

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.

E HYPER-PARAMETER ANALYSIS FOR BASELINES

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

$$\mathbf{x}^{t+1} = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)} \left(\mathbf{x}^t + \alpha \operatorname{sgn}(\nabla L(\mathbf{x}^t, y, \hat{y})) \right). \tag{28}$$

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 $\S B$. 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

E.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

E.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

$$\min_{\delta} \; c \cdot h(x+\delta) + \sum_i [(\delta_i - \tau)_+], \tag{29}$$

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:
            $\min_{\delta} \; c \cdot h(x+\delta) + \sum_i [(\delta_i - \tau)_+]$        (30)
 6:         Optimize 30 using the Adam optimizer with a learning rate of $\alpha$, and a step number of $T$
 7:         if found a counter example with $\delta_i \le \tau \; \forall i$ 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

F FURTHER EXPERIMENTAL RESULTS

F.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 5 and Figure 1. 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' 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 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 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.

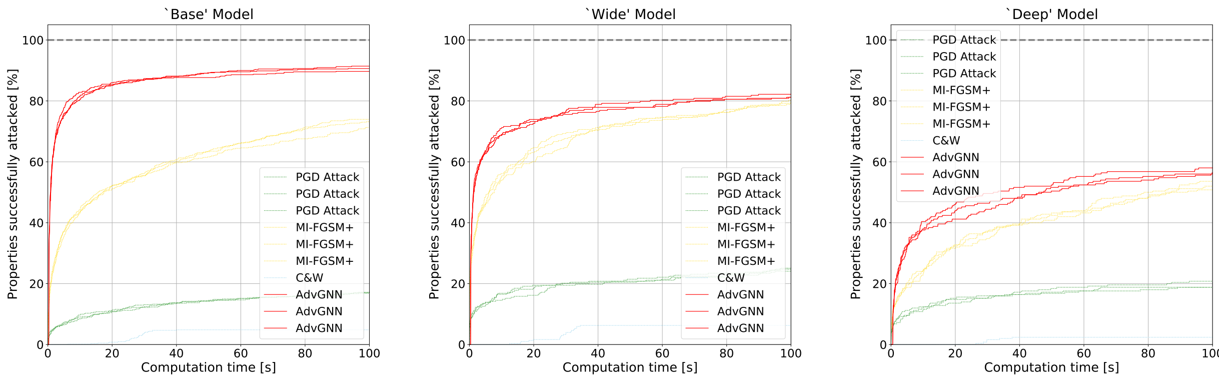

Figure 1: Cactus plots for the main datasets on the 'Base' , 'Wide' and 'Deep' models. For each, we compare the attack methods by plotting the percentage of successfully attacked images as a function of runtime.

F.2 EASY EXPERIMENTS

As mentioned above, we also run experiments on an easier dataset. In practice there may be use cases where we want to generate easy adversarial examples very quickly, hence it is beneficial for strong methods to also work well on easier tasks. As all methods will generate adversarial examples more quickly on this easier version of the dataset we reduce the timeout to 20 seconds. The results are summarized in Tables 6 and 7 and Figure 2. AdvGNN outperforms all baselines on all three models. On the 'Base' model in particular we reduce the percentage of properties on which our method times out by over 98% compared to each of the three baselines. When comparing the results for the different seeds we see that every single run of AdvGNN beat every other run of any of the baselines, again showing that changing the random seed does not change the outcome significantly.

		'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 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 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 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. The best performing method for each subcategory is highlighted in bold.

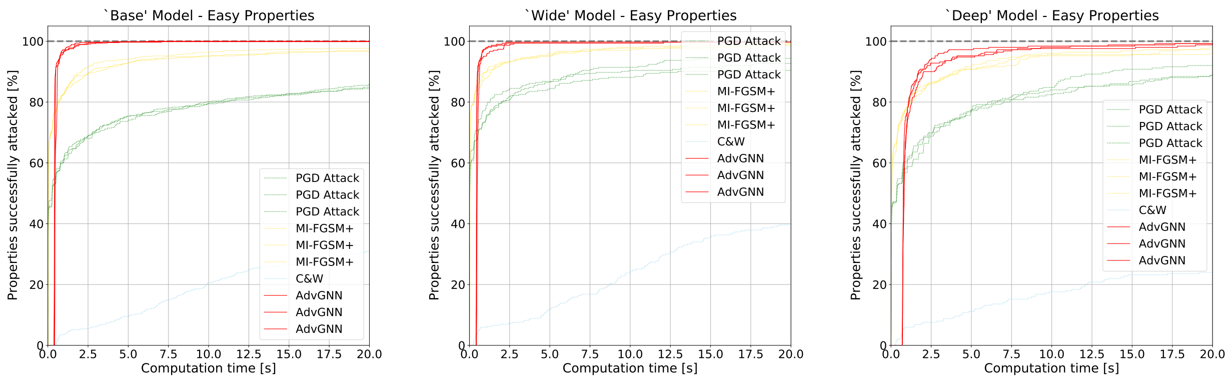

Figure 2: Cactus plots for the easy datasets on the 'Base' , 'Wide' and 'Deep' models. For each, we compare the attack methods by plotting the percentage of successfully attacked images as a function of runtime.

F.3 EXPERIMENTS ON AN 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 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 8.

F.4 ABLATION STUDY - SIMPLED FEATURE VECTORS

Computing the features vectors (Equations (32) and (33)) requires solving a linear program (Equation (31)). However, if we use a simpler approach as proposed by Kolter and Wong (2018) instead of super-gradient ascent our method still outperforms all baselines, successfully attacking 86% of all properties on the base model compared to 5%, 17%, and 73% for the three baselines, respectively (Table 9). 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	Seed	Time(s)	Timeout(%)
PGD Attack	2222	16.922	79.2
PGD Attack	3333	16.222	78.2
PGD Attack	4444	16.382	77.2
MI-FGSM+	2222	5.771	27.8
MI-FGSM+	3333	5.773	18.8
MI-FGSM+	4444	5.847	20.8
AdvGNN	2222	4.079	12.9
AdvGNN	3333	3.739	12.9
AdvGNN	4444	3.851	14.9

Table 8: 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 and the random Pytorch seeds specified.

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 9: '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 is the main method described; AdvGNN-s uses the simple KW method rather than the iterative supergradient ascent method to compute the feature vector (23)

References

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, 2020.

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.

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.

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, 2017.

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.

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

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

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.

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