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. During training, its parameters are estimated via a loss function that encourages the efficient computation of adversarial examples over a time horizon. We show that our method beats state-of-the-art adversarial attacks, including PGD-attack, MI-FGSM, and Carlini and Wagner attack, reducing the time required to generate adversarial examples with small perturbation norms by over 65%. Moreover, AdvGNN achieves good generalization performance on unseen networks. Finally, we provide a new challenging dataset specifically designed to allow for a more illustrative comparison of adversarial attacks.

1 INTRODUCTION

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

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

Accepted for the 37th Conference on Uncertainty in Artificial Intelligence (UAI 2021).

architecture, we can train a GNN using one network and test it on another.

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

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

2 RELATED WORK

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

Studying adversarial attacks, and white-box attacks in particular, has become an active field of research over the last few years. Adversarial attacks can be separated into three main categories [Serban et al., 2020]. One class of attacks aims to find an adversarial example that lies within some allowed perturbation and that the network misclassifies with a high level of confidence. Goodfellow et al. [2015] proposed the Fast Gradient Sign Method (FGSM) that takes a single step towards the gradient of the adversarial loss function. The Iterative Fast Gradient Method (I-FGSM) [Kurakin et al., 2016] and Projected Gradient Attack (PGD) [Madry et al., 2018] both apply FGSM iteratively, taking several steps towards the sign of the gradient. Dong et al. [2018] proposed adding momentum to I-FGSM, thus significantly improving its performance (MI-FGSM).

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

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

We propose using a Graph Neural Network (GNN) to combine the strengths of both the optimization based and learning based methods to generate adversarial examples more efficiently. GNNs have been used in Neural Network Verification to learn the branching strategy in a Branch-and-Bound algorithm [Lu and Kumar, 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. We show in this work how they can be employed successfully for this task.

3 PROBLEM DEFINITION

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

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

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

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

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

$$f(\mathbf{x}')_{\hat{y}} \geq f(\mathbf{x}')_y \quad \text{and} \quad d(\mathbf{x}, \mathbf{x}') \leq \epsilon, \tag{3}$$

for some distance measure d. In other words we aim to find an adversarial example $\mathbf{x}'$ that is close to the original input but is misclassified as $\hat{y}$. Problem (3) is often reformulated as follows:

$$max_{\mathbf{x}' \in \mathcal{B}(\mathbf{x},\epsilon)} L(\mathbf{x}', y, \hat{y}) = f(\mathbf{x}')_{\hat{y}} - f(\mathbf{x}')_y, \tag{4}$$

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

$$\mathcal{B}(\mathbf{x}, \epsilon) := \{x' \mid d(\mathbf{x}, \mathbf{x}') \leq \epsilon\}. \tag{5}$$

We refer to L as the adversarial loss from now on. If $L(\mathbf{x}', y, \hat{y}) \geq 0$ then $\mathbf{x}'$ is considered an adversarial example.

FGSM [Goodfellow et al., 2015], a fast attack on the l_∞ norm, aims to solve (4) by using the sign of the gradient of the adversarial loss:

$$\mathbf{x}' = \mathbf{x} + \epsilon \ \text{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}', y, \hat{y})). \tag{6}$$

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

$$\mathbf{x}^{t+1} = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)} \left(\mathbf{x}^t + \alpha \ \text{sgn}(\nabla_{\mathbf{x}} L(\mathbf{x}', y, \hat{y})) \right). \tag{7}$$

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

4 GNN FRAMEWORK

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

4.1 OVERVIEW

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

4.2 GNN COMPONENTS

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

Node Features. For each node $\mathbf{v}_k[i]$ we define a corresponding q-dimensional feature vector $\mathbf{f}_k[i] \in \mathbb{R}^q$ describing the current state of that node. Its exact definition depends on the task we want to solve. In our

experiments the feature vectors consist of three parts: the first part captures the gradient at the current point; the second part includes the lower and upper bounds for each neuron in the original network based on the bounded input domain; and the third part encapsulates information that we get from solving a standard relaxation of the adversarial loss from the incomplete verification literature. A more detailed analysis can be found in Appendix C.

While more complex features could be included, we deliberately chose the simple features described above and rely on the power of GNNs to efficiently compute an accurate direction of movement.

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

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

$$\boldsymbol{\mu}_k[i] := g(\mathbf{f}_k[i]). \tag{8}$$

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

$$\Theta_0 \in \mathbb{R}^{q \times p}, \quad \Theta_1, \ldots, \Theta_{T_1} \in \mathbb{R}^{p \times p}. \tag{9}$$

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

$$\boldsymbol{\mu}^0_k = \text{relu}(\Theta_0 \cdot \mathbf{f}_k), \quad \boldsymbol{\mu}^{l+1}_k = \text{relu}(\Theta_{l+1} \cdot \boldsymbol{\mu}^l_k). \tag{10}$$

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

4.3 FORWARD AND BACKWARD PASSES

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

$$\boldsymbol{\mu}'_k[i] = \text{relu}\left(\Theta^{for}_1 \boldsymbol{\mu}_k[i] + \Theta^{for}_2 \left(W_k \boldsymbol{\mu}_{k-1} + \mathbf{b}_{k-1}\right)[i] + \Theta^{for}_3 \left(\sum_{j \in N(i)} \boldsymbol{\mu}_{k-1}[j] / Q_{k+1}[j]\right)[i]\right). \tag{11}$$

Similarly, we perform a backward pass as follows:

$$\boldsymbol{\mu}_k[i] = \text{relu}\left(\Theta^{back}_1 \boldsymbol{\mu}'_k[i] + \Theta^{back}_2 (W^T_{k+1} \left(\boldsymbol{\mu}'_{k+1} - \mathbf{b}_{k+1}\right))[i] + \Theta^{back}_3 \left(\sum_{j \in N'(i)} \boldsymbol{\mu}'_{k+1}[j] / Q'_{k+1}[j]\right)[i]\right). \tag{12}$$

Here $\Theta^{for}_1, \Theta^{for}_2, \Theta^{for}_3, \Theta^{back}_1, \Theta^{back}_2, \Theta^{back}_3 \in \mathbb{R}^{p \times p}$ are all learnable parameters and W and b are the weights and biases of the target network as defined in equations (1) and (2). Both (11) and (12) can be implemented using existing deep learning libraries. To ensure better generalization performance to unseen neural networks with a different network architecture we include normalization parameters Q and Q'. These are matrices whose elements are the number of neighbouring nodes in the previous and following layer respectively for each node. We repeat this process of running a forward and backward pass T_2 times. The high-dimensional embedding vectors are now capable of expressing the state of the corresponding node taking the entire problem structure into consideration as they are directly influenced by every other node, even if we set $T_2 = 1$.

4.4 UPDATE STEP

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

$$\tilde{\mathbf{x}} = \boldsymbol{\Theta}^{out} \cdot \boldsymbol{\mu}_0. \tag{13}$$

Ideally the GNN would output a new ascent direction that will lead us directly to the global optimum of equation (4). However, as the problem is complex this may not be feasible in practice without making the GNN very large, thereby resulting in computationally prohibitive inference. Instead, we propose to run the GNN a small number of times to return directions that gradually move towards the optimum.

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

$$\mathbf{x}^{t+1} = \Pi_{\mathcal{B}(\mathbf{x},\epsilon)}\left(\mathbf{x}^t + \alpha\tilde{\mathbf{x}}\right). \tag{14}$$

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

5 GNN TRAINING

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

In order to get the individual components that make up the feature vectors, we first compute the intermediate bounds of each node in the network using the method by Wong and Kolter [2018] which is explained in greater detail in Appendix C.1. We further solve a standard relaxation of the robustness problem via methods from the verification literature (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:

$$\mathcal{L}_{i,j} = -\sum_{t=1}^{K} L(\mathbf{x}^{i,j,t}, y^i, \hat{y}^i) * \gamma^t. \tag{15}$$

Instead of maximizing over the adversarial loss, we minimize over the negative loss. If the decay factor $\gamma \in (0, 1)$ is low then we encourage the model to make as much progress in the first few steps as possible, whereas if γ is closer to 1, then more emphasis is placed on the final output of the GNN, sacrificing progress in the early stages. Readers familiar with reinforcement learning may be reminded of the discount rates used in algorithms such as Q-learning and policy-gradient methods.

We sum over the individual loss values corresponding to each data point and each initial starting point to get the final training objective $\mathcal{L}$:

$$\mathcal{L} = \sum_{i=1}^{|D|}\sum_{j=1}^{s}\mathcal{L}_{i,j}. \tag{16}$$

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

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

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

6 A NEW DATASET FOR COMPARING ADVERSARIAL ATTACKS

In this section we describe our new dataset that has been specifically designed to compare state-of-the-art adversarial attacks.

Previously, adversarial attacks were compared on how well they attack a trained neural network on a set number of images for a fixed allowed perturbation [Madry et al., 2018, Dong et al., 2018, Carlini and Wagner, 2017, Moosavi-Dezfooli et al., 2016]. However, for many of the images there either does not exist an adversarial example in the allowed perturbed input space or there 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 and 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.

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

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.

7 EXPERIMENTS

We now describe an empirical evaluation of our method by comparing it to several state-of-the-art attacks on the CIFAR-10 dataset. We first outline the experimental setting (§7.1), before describing the attacks we compare our method to (§7.2), and finally analysing the results (§7.3).

7.1 SETUP

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

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

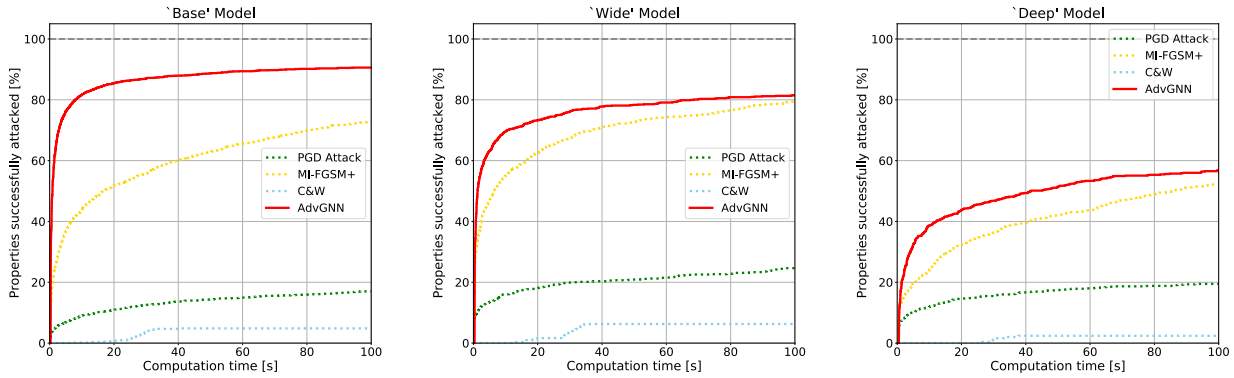
7.2 METHODS

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

PGD. The first baseline we run is PGD-attack [Madry et al., 2018]. As described before, PGD-attack picks an initial starting point uniformly at random and then iteratively performs Projected Gradient Descent on the negative adversarial loss (7). Based on an extensive hyper-parameter analysis (see Appendix E.1) we pick the stepsize parameter $\alpha = 0.1$, and set the number of iterations to $T = 100$. We perform random restarts until we have either managed to find an adversarial example or the time limit has been reached.

MI-FGSM+. MI-FGSM is I-FGSM with an added momentum term. MI-FGSM starts at the image $\mathbf{x}$ and takes T steps of size ϵ/T. Defining the stepsize as such ensures that the current point lies in the feasible region throughout the entire algorithm without the need to project. To strengthen the attack we perform random restarts as we do for PGD. To ensure that not all runs of MI-FGSM on the same image are identical we therefore have to choose the initial point randomly as well. Furthermore, we perform a hyper-parameter search not only over the momentum term μ and the number of iterations T as done in the original paper, but also over the stepsize α (see Appendix E.2 for details). We run it with the following optimized parameters: $\alpha = 0.1$, $\mu = 0.5$, and $T = 100$. This modified version of MI-FGSM is denoted as MI-FGSM+.

C&W. The third baseline we use is C&W, the optimization-based attack proposed by Carlini and Wagner [2017]. C&W aims to find the smallest perturbation required to find an adversarial example by minimizing a loss function of the form $l(v) = c \cdot F(x + v) + \|(v - \tau)_+\|_1$ for some surrogate function F, and constants c and τ. There are a total of six hyper-parameters which we optimize over on a validation dataset and which we describe in greater detail in Appendix E.3. We note that in the original implementation C&W is often run until the minimum perturbation for which there exist at least one adversarial example is found. However, to be able to compare it to the other methods we stop the C&W attack as soon as an



(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. Baselines are represented by dotted lines. AdvGNN beats all baselines on all three models for any chosen timeout value.

adversarial example is found for the given perturbation value or when the time limit is reached.

AdvGNN. The final attack we run is AdvGNN. We train our AdvGNN on the 'Base' model and on 2500 images of the CIFAR-10 test set that are not part of the dataset we test on. The ϵ values which define the allowed perturbation for each training sample are computed in a similar procedure to the test datasets described above. We train the GNN using the loss function described in section §5 with a horizon of 40 and with decay factor $\gamma = 0.9$. The training loss function is minimized using the Adam optimizer [Kingma and Ba, 2015] with a weight decay of 0.001. The initial learning for Adam is 0.01, and is manually decayed by a factor of 0.1 at epochs 20, 30, and 35. We pick the following values for the hyper-parameters of our AdvGNN: the stepsize α is 1e-2, the embedding size is $p = 32$, and we perform a single forward and backward pass ($T_1 = T_2 = 1$). To improve the performance on the 'Deep' model we fine-tune our AdvGNN for 15 minutes on the 'Deep' model before running the attack. Fine-tuning is run on 300 images that are not included in the 'Deep' test set. We use a fixed ϵ value of 0.25 for all images.

7.3 RESULTS

'Base' Model. We run all four methods described in the previous section on the 'Base' model with a timeout of 100 seconds and record the percentage of properties successfully attacked as a function of time (Figure 1a and Table 1). C&W only manages to find an adversarial advantage for less than 5% of all images. PGD outperforms C&W but still only manages to solve 17% of all properties. MI-FGSM+ outperforms PGD, timing out on 26% of all images with an average time of 40 seconds. AdvGNN beats all three methods reducing both the average time taken and the proportion of properties timed out on by more than 65%.

Table 1: '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.

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

Table 2: 'Wide' Model. We compare the methods on the 'Wide' model.

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

'Wide' Model. Next we compare the methods on the 'Wide' model (Figure 1b and Table 2). AdvGNN has not seen this network during training and before running these experiments. MI-FGSM+ is again the best performing baseline, and AdvGNN the best performing method overall both in terms of average solving

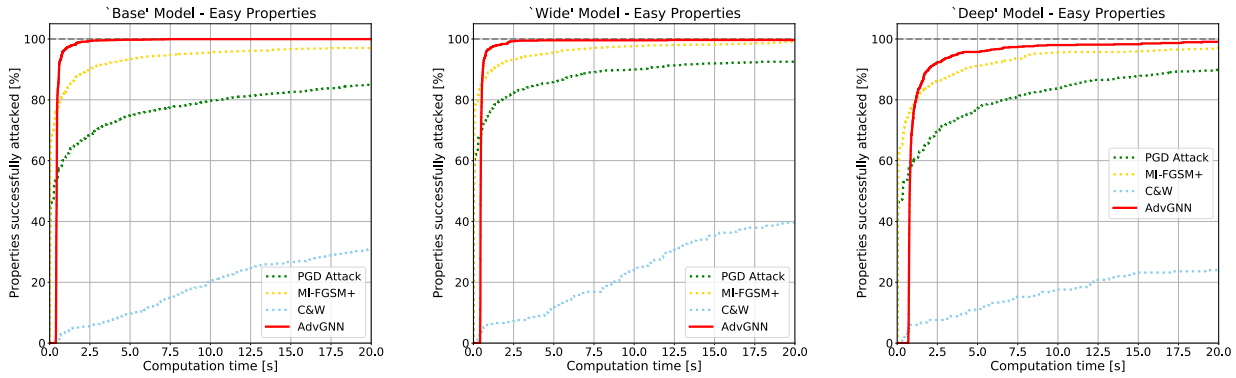


Figure 2: Cactus plots for experiments on the easier version of the dataset on the 'Base' model (left), 'Wide' model (middle) and the 'Deep' model (right). We add a small constant $\delta = 0.001$ to each perturbation size ϵ^i. For each model, we compare the attacks by plotting the percentage of successfully attacked images as a function of runtime. AdvGNN is the best performing attack on all three models.

time and percentage of properties successfully attacked. AdvGNN reduces the time required to find an adversarial example by over 70% compared to PGD and C&W , and by 20% compared to MI-FGSM+. This demonstrates that AdvGNN achieves good generalization performance and can be trained on one model and used to run attacks on another.

Table 3: 'Deep' Model. We compare the different methods on the 'Deep' model.

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

'Deep' Model. We also run experiments on the 'Deep' model (Table 3, Figure 1c). We remind the reader that the AdvGNN parameters have been fine-tuned on this model for 15 minutes to achieve better results. AdvGNN outperforms all three other attacks on this larger 'Deep' model both with respect to the total number of successful attacks and the average time of each attack. Figure 1c shows that AdvGNN is still the best performing method even if we pick a shorter timeout of less than 100 seconds.

Easy Dataset. As some of the baselines, C&W in particular, struggle to successfully attack most of the properties in the previous experiment, we further compare the methods on a simpler dataset. We add a constant delta (0.001) to each epsilon value in the above dataset and reduce the timeout to 20 seconds. Increasing the allowed perturbation simplifies the task of finding an adversarial example as can be seen in Figure 2. 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 F.

8 DISCUSSION

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

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

References

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

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*, 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.

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

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.

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*, pages 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*, pages 6156–6160. AAAI Press, 2019.

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

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

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*, pages 2574–2582, 2016.

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

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

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

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

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

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

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*, pages 3905–3911, 2018.

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