Using Wasserstein GAN to Generate Adversarial Examples

Zhihan Xiong
Department of Physics and Mathematics
zxiong9@illinois.edu

Pierre Moulin
Department of Electrical and Computer Engineering
moulin@ifp.uiuc.edu

Abstract

Although Deep Neural Networks (DNNs) have state-of-the-art performance in various machine learning tasks, in recent years, they are found to be vulnerable to so-called adversarial examples. Specifically, take $\mathbf{x} \in \mathcal{D}$ on which a neural network has very high classification accuracy. It is possible to find some small perturbation $\Delta\mathbf{x}$ so that even though the difference between $\mathbf{x}$ and $\mathbf{x} + \Delta\mathbf{x} = \mathbf{x}'$ is almost imperceptible to humans, the given neural network is very likely to incorrectly classify $\mathbf{x} + \Delta\mathbf{x}$.

Currently there have been proposed several gradient and optimization based methods to create such adversarial examples $\mathbf{x}'$, but many of them cannot achieve high speed and high quality $\mathbf{x}'$ simultaneously. In this paper, we proposed a new algorithm to generate adversarial examples based on Generative Adversarial Networks (GANs), specifically, a modification to the training algorithm of the Improved Wasserstein GAN. The trained generator is able to create $\mathbf{x}'$ very similar to the original $\mathbf{x}$ while keeping the classification accuracy of the target model as low as the state-of-the-art attack. Furthermore, although training a GAN might be slow, after it is trained, it can generate adversarial examples much faster than previous optimization-based methods. Our goal is for this work to be used for further research on robust neural networks.

1 Introduction

Starting from 2012, in which AlexNet was proposed [1], deep neural networks (DNNs) begins to succeed in many different machine learning tasks including image classification [2], machine translation [3], image generation [4] and game playing [5]. Many of these works have comparable or even better performance than humans.

However, more recent research shows that although these networks are strong, they are far from robust. In particular, they can be easily attacked by deliberately crafted data, which are called *adversarial examples*. These examples, usually in image classification tasks, are created by adding a small perturbation on the original data. Although the difference is often imperceptible to humans, it is possible to make state-of-the-art neural networks incorrectly classify them with high confidence. Figure 1 shows a classical instance of this phenomenon [6].

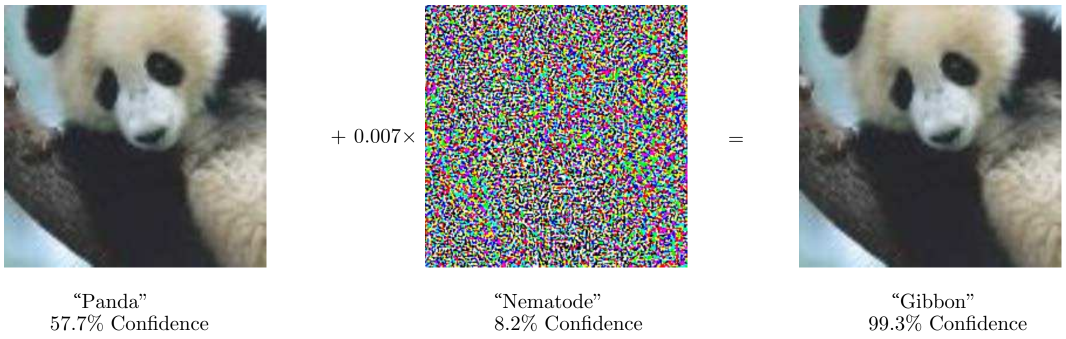

Figure 1: Goodfellow's demonstration [6] of adversarial example applied to GoogLeNet [7] on ImageNet. Even though the two pictures of panda are almost identical to human, GooLeNet is not able to identify the panda in the right picture.

It is possible to maliciously utilize this property of neural networks. For example, in the scenario of self-driving car, if the stop sign is adversarially perturbed, even though humans will still recognize it as a stop sign, the car may identify it as something else. As a result, some severe traffic accidents may happen. Furthermore, it is showed by Kurakin et al. that even the ordinary objects perceived through camera may become adversarial examples [8]. Therefore, this phenomenon is important and needs thorough exploration.

Currently, there have been proposed many effective approaches to generate adversarial examples including Goodfellow's fast gradient sign method [6], Papernot's Jacobian-based saliency map approach [9] and Carlini and Wagner's (CW's) optimization-based method [10]. However, for most current attacks, the issue is about the tradeoff between speed and quality. On one hand, one-step gradient-based methods like fast gradient sign method can craft adversarial examples very efficiently, but its attack is not strong and sometimes human can perceive the perturbation. On the other hand, optimization-based methods like the one proposed by Carlini and Wagner, also the currently known strongest attack, can create adversarial examples with very high quality, but each creation may need hundreds or thousands of iterations.

In this paper, to address these two issues, inspired by the fact that Generative Adversarial Networks (GANs) use a network to fool another network [11], we propose a method for generating adversarial examples using a GAN, specifically a Wasserstein GAN [12]. We show that our method can reach state-of-the-art success rate while keeping the speed of generating adversarial examples very fast.

This thesis will be organized as follows: in chapter 2, we will provide some background knowledge about DNNs and adversarial machine learning; in chapter 3, we will give a brief introduction to generative adversarial networks (GANs) and some other attack methods. Our Wasserstein GAN-based method will be introduced in chapter 4 and the experimental results of its comparison to other methods will be provided in chapter 5. Finally, in chapter 6, we will present our conclusion.

2 Backgrounds

2.1 Neural Networks

We first provide a brief introduction to neural networks, specifically those being used as classifiers. A neural network classifier F with k hidden layers is a composition of $k+1$ functions such that $F = F_{k+1} \circ F_k \circ \cdots \circ F_1 : \mathbb{R}^n \to \mathbb{R}^m$, where n is the dimension of input vector and m is the number of class.

Here, we will adopt the notation of Papernot and Carlini [10, 13]. Let $Z = F_k \circ F_{k-1} \circ \cdots \circ F_1 : \mathbb{R}^n \to \mathbb{R}^m$, which is often called as logits. We view $F(\mathbf{x}) = \mathbf{y} \in \mathbb{R}^m$ as a probability over m classes and in order for $\mathbf{y}$ to be a valid distribution, i.e. $\sum_{i=1}^{m} y_i = 1$, we use

$$F_{k+1}(Z(\mathbf{x}))_i = \frac{e^{Z(\mathbf{x})_i}}{\sum_{j=1}^{m} e^{Z(\mathbf{x})_j}} \tag{1}$$

where F_{k+1} is called as the softmax function.

Since $\mathbf{y}$ is viewed as a probability distribution over m classes, the label that F assigns to $\mathbf{x}$ will be defined as $C(\mathbf{x}) = \underset{i}{\operatorname{argmax}}\, F(\mathbf{x})_i$. Since $F(\mathbf{x})_i \propto e^{Z(\mathbf{x})_i}$, we can immediately get $\underset{i}{\operatorname{argmax}}\, F(\mathbf{x})_i = \underset{i}{\operatorname{argmax}}\, Z(\mathbf{x})_i = C(\mathbf{x})$.

The choice of functions $F_i, i \in \{1, 2, ..., k\}$ depends on the specific structure of neural network. In this paper, we will focus on image classification, in which we often choose $F_i = f_i\left(\mathbf{W}_i F(\mathbf{x})_{i-1} + \mathbf{b}_i\right)$. Here $\mathbf{W}_i$ is either a full matrix of a sparse matrix equivalent to operations like convolution or pooling. f_i is some nonlinear activation function, which is often chosen to be ReLU, hyperbolic tangent or sigmoid function. Among these functions, ReLU function, which is defined as $f(x) = \max\{0, x\}$, is the most widely used activation function in image classification area [10, 13].

2.2 Adversarial Examples and Attacks

We first give an informal definition of adversarial example.

Definition 2.1 *Given an input* $\mathbf{x}$ *and a neural network classifier* C *that* $C(\mathbf{x}) = y$ *is what a human perceives, an adversarial example is defined as some* $\mathbf{x}'$ *that is close to* $\mathbf{x}$ *such that* $C(\mathbf{x}') \neq C(\mathbf{x})$.

The closeness between $\mathbf{x}'$ and $\mathbf{x}$ can be defined in various way, but the most widely used definition is based on Euclidean distance, i.e. $\operatorname{distance}(\mathbf{x}', \mathbf{x}) = \|\mathbf{x}' - \mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n}(x_i' - x_i)^2}$. However, the appropriateness of this definition for closeness should have further research. The reason is that we want $\mathbf{x}'$ to look natural or the difference to $\mathbf{x}$ to be imperceptible to humans, but the Euclidean distance may not reflect such difference. Figure 2 shows an example of this argument.

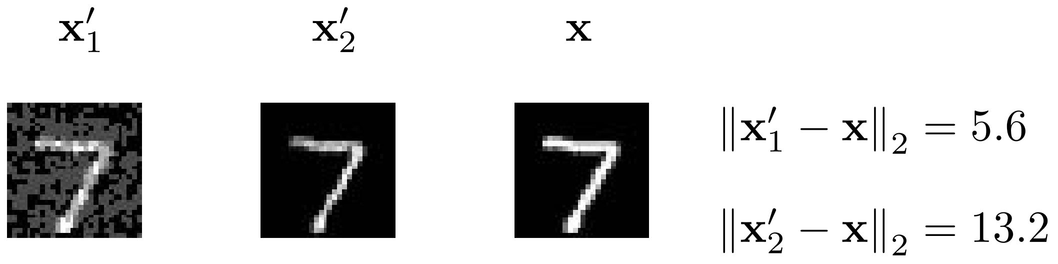
$\mathbf{x}_1'$
$\mathbf{x}_2'$
$\mathbf{x}$
$\|\mathbf{x}_1' - \mathbf{x}\|_2 = 5.6$
$\|\mathbf{x}_2' - \mathbf{x}\|_2 = 13.2$

Figure 2: $\mathbf{x}_1'$ is closer to $\mathbf{x}$ than $\mathbf{x}_2'$ based on Euclidean distance. However, $\mathbf{x}_2'$ looks much more like a digit someone may write than $\mathbf{x}_1'$. The original figure comes from the MNIST dataset [14].

The attacks can be classified as untargeted and targeted. The behavior of crafting the above mentioned $\mathbf{x}'$ is called an untargeted attack. The targeted attack is a stronger version of untargeted attack, in which we can choose some target label t with $t \neq C(\mathbf{x})$ and craft a similar $\mathbf{x}'$ so that $C(\mathbf{x}') = t$. Most currently proposed attacks have both untargeted and targeted version [10, 13, 15, 16, 17] and it is usually easy to transform the later version to the former version. Hence, in this paper, we will mainly focus on targeted attacks.

The attacks can also be classified based on its scenario. Specifically, we have white-box attack and black-box attack. The former one means that the attacker has full access to the target model and knows its model structure and parameters. The later one means that the attacker has only limited access to the target model. Usually, the attacker can only send limited number of queries to the target model and get labels assigned to the sent examples by the target model [17].

Although the later case is much more realistic for the attacker, Papernot et al. have shown that because of the existence of transferability, it will be easy to train a substitute model based on these queries so that attacks to this substitute model will still be very effective to the target model. [17]. Therefore, in this paper, we will only consider the white-box attack.

3 Related Works

3.1 Other Attack Methods

Currently, many methods for crafting adversarial examples have been proposed including one-step gradient-based methods, iterative gradient-based methods, optimization-based methods and gradient-free methods [6, 18, 15, 19, 16, 20, 21, 9, 10, 22].

Here, we will introduce two attack methods that are used for comparison in this paper. They are the fast gradient sign method [6] and CW's optimization-based method [10].

- **Fast Gradient Sign Method (FGSM)**: The untargeted version of FGSM was first proposed by Goodfellow et al. [6]. Given a valid input $\mathbf{x}$ and a target model F with its logits output $Z(\mathbf{x})$ and assigned label $y = C(\mathbf{x})$, the adversarial example is crafted based on the following formula:

  $$\mathbf{x}' = \mathbf{x} - \epsilon \cdot \operatorname{sign}\left(\nabla_{\mathbf{x}} Z(\mathbf{x})_y\right) \tag{2}$$

  where ϵ is a hyperparameter for controlling the strength of attack.

  Suppose $t \neq y$ is our target label, its targeted version is provided by the following formula [23]:

  $$\mathbf{x}' = \mathbf{x} + \epsilon \cdot \operatorname{sign}\left(\nabla_{\mathbf{x}} Z(\mathbf{x})_t\right) \tag{3}$$

- **CW's Optimization-based Method**: Based on Carlini and Wagner's proposition, crafting an adversarial example targeted on $t \neq y$ was formulated as the following optimization problem [10]:

  $$\begin{array}{cl} \underset{\boldsymbol{\delta}}{\text{minimize}} & \|\boldsymbol{\delta}\|_2^2 \\ \text{such that} & f(\mathbf{x}+\boldsymbol{\delta}) \le 0 \\ & \mathbf{x}+\boldsymbol{\delta} \in [0,1]^n \end{array} \tag{4}$$

  where f is some function that $f(\mathbf{x}+\boldsymbol{\delta}) \le 0$ if and only if $C(\mathbf{x}+\boldsymbol{\delta}) = t$. The function they choose is

  $$f(\mathbf{x}) = \max\left\{\max_{j \neq t} Z(\mathbf{x})_j - Z(\mathbf{x})_t, -\kappa\right\} \tag{5}$$

  where κ is a hyperparameter. The alternative formulation gives that [10]

  $$\begin{array}{cl} \underset{\boldsymbol{\delta}}{\text{minimize}} & \|\boldsymbol{\delta}\|_2^2 + c \cdot f(\mathbf{x}+\boldsymbol{\delta}) \\ \text{such that} & \mathbf{x}+\boldsymbol{\delta} \in [0,1]^n \end{array} \tag{6}$$

  where $c > 0$ is some user-chosen constant.

  In order to preserve the constraint that $\mathbf{x}+\boldsymbol{\delta} \in [0,1]^n$, which is a condition specifically for gray-scale image, they used the change of variables that $\boldsymbol{\delta} = \frac{1}{2}(\tanh(\mathbf{w})+1) - \mathbf{x}$. With all of the equations above, the optimization problem we need to solve is [10]

  $$\underset{\mathbf{w}}{\text{minimize}} \left\|\frac{1}{2}(\tanh(\mathbf{w})+1) - \mathbf{x}\right\|_2^2 + c \cdot f\left(\frac{1}{2}(\tanh(\mathbf{w})+1)\right) \tag{7}$$

  and $\frac{1}{2}(\tanh(\mathbf{w})+1) = \mathbf{x}+\boldsymbol{\delta}$ gives the adversarial example.

3.2 Generative Adversarial Networks (GANs)

GAN was first proposed by Goodfellow et al. in 2014 [11] and currently it is still one of the best generative models. It is composed by two networks, the generator G and discriminator D. Conceptually, during the training procedure, on the one hand, discriminator D will try to differentiate the real data and the data generated by G; on the other hand, the generator G will try to generate data similar to the real data so that D cannot tell the difference. Figure 3 shows how GAN works [24].

Mathematically, we can treat the real dataset as a sample from some unknown distribution, denoted as $p_{\text{data}}(\mathbf{x})$, and the task of generator G is to approximate this distribution. The distribution defined by G, denoted as p_g, is $G(\mathbf{z})$ obtained when $\mathbf{z} \sim p_{\mathbf{z}}$, where $\mathbf{z}$ is some random noise like standard

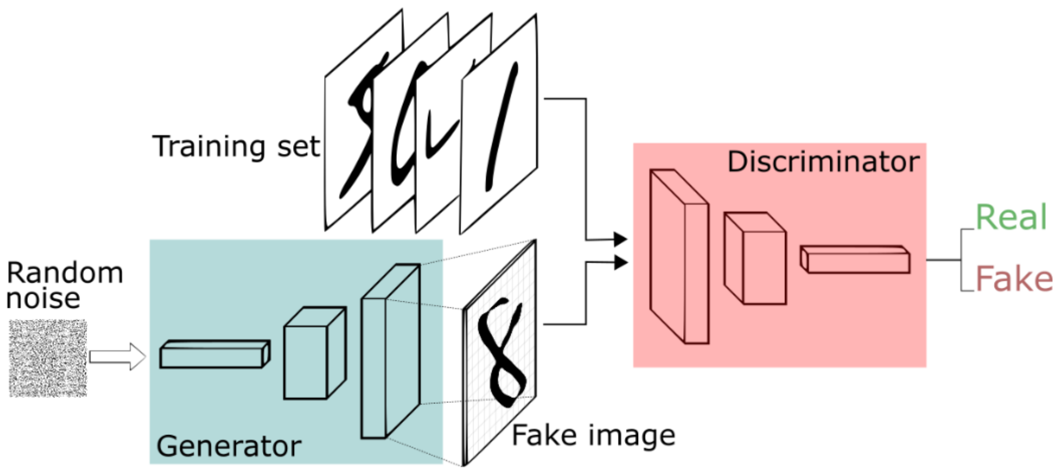

Figure 3: A conceptual explanation of how GAN works [24].

Gaussian [11]. Based on the proposition of Goodfellow et al., the training procedure of GAN is to solve the following optimization problem [11]:

$$\min_G \max_D L(D,G) = \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}\left[\log D(\mathbf{x})\right] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})}\left[\log\left(1 - D(G(\mathbf{z}))\right)\right] \tag{8}$$

where $D(\mathbf{x})$ represents the probability that $\mathbf{x}$ comes from p_{data}.

It is possible to show that the above problem is equivalent to optimize the Jensen-Shannon Divergence between p_{data} and p_g [25]. However, training a GAN based on this procedure is difficult and unstable. To address this issue, Arjovsky et al. proposed a new training algorithm for GAN based on minimizing the Wasserstein distance between p_{data} and p_g [12]. Later, the algorithm of training a Wasserstein GAN (WGAN) was further improved by Gulrajani et al. through adding a gradient penalty [26] during the training step of discriminator D. Below is the new proposed loss function [26]:

$$\max_G \min_D L_{\text{WGAN}}(D,G) = \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})}\left[D(G(\mathbf{z}))\right] - \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}\left[D(\mathbf{x})\right] \tag{9}$$

The gradient penalty term added when updating D is [26]

$$\lambda \cdot \mathbb{E}_{\hat{\mathbf{x}}\sim p_{\hat{\mathbf{x}}}(\hat{\mathbf{x}})}\left[\left(\left\|\nabla_{\hat{\mathbf{x}}} D(\hat{\mathbf{x}})\right\|_2 - 1\right)^2\right] \tag{10}$$

where λ is a hyperparameter and $\hat{\mathbf{x}} = \epsilon\mathbf{x} + (1-\epsilon)G(\mathbf{z})$ for some $\epsilon \sim U(0,1)$, the uniform distribution over $(0,1)$.

Algorithm 2 gives the details for training a WGAN, which is provided in the Appendix for reference. In this paper, our algorithm for generating adversarial examples will be a modification based on the improved algorithm for training WGAN, which will be called as adversarial WGAN.

4 Our Methodology

Based on the previous discussion, we can notice that during training a GAN, what a generator G does is essentially creating some fake data that is similar to the original in order to fool another network, the discriminator D. This is quite similar to something an adversarial attack will do. Therefore, it is natural to consider the possibility for letting GAN generate adversarial examples.[1] Specifically, we will use the Wasserstein GAN because it is much easier to train than the ordinary GAN.

Intuitively, given the original data $\mathbf{x}$, the task for generator G is still to generate $\mathbf{x}'$ that is similar to $\mathbf{x}$ so that the discriminator D cannot differentiate them. Thus, the loss function also contains the part for ordinary WGAN $L_{\text{WGAN}}(D,G)$.

[1]Although considering independently, we are unfortunately not the first to propose using GAN to generate adversarial examples [27]. However, our method can still be viewed as an improvement of their method.

The first difference is that the input of generator G now is not some random noise but the original data $\mathbf{x}$ because the adversarial example by definition is crafted based on some clean data.

The second difference is that this time, G needs to make sure that $\mathbf{x}'$ can also fool the target model so that $C(\mathbf{x}') = t \neq y = C(\mathbf{x})$. Thus, we need another term $L_{\text{adv}}(G, F, t)$ such that minimizing it is equivalent to fool the target model F for some target t. Here, we borrow the function suggested by Carlini et al. [10] that possesses this property. Thus, we have

$$L_{\text{adv}}(G, F, t) = \mathbb{E}_{\mathbf{x}' \sim p'}\left[\max\left\{\max_{j \neq t} Z(\mathbf{x}')_j - Z(\mathbf{x}')_t, -\kappa\right\}\right] \tag{11}$$

Furthermore, in order to preserve the constraint that $\mathbf{x}' \in [0,1]^n$ (when using RGB representation, the constraint will become $\mathbf{x}' \in [0,255]^n$, but this re-scaling is easy), we can apply the transformation that $\mathbf{x}' = \frac{1}{2}(\tanh(G(\mathbf{x})) + 1)$, which is the idea inspired by CW's method [10]. Figure 4 shows the whole data flow procedure described above.

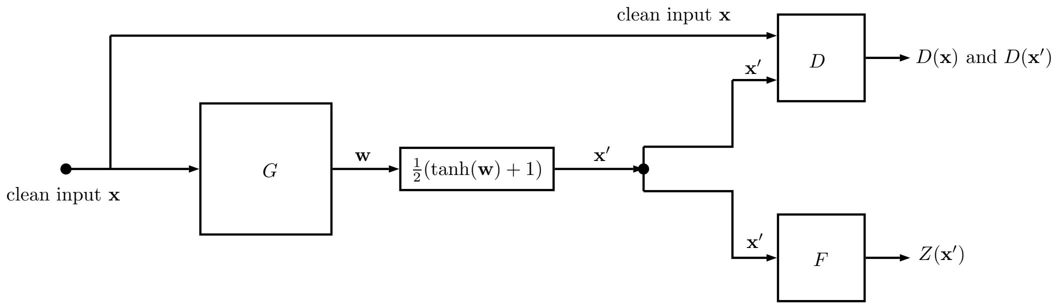

Figure 4: The whole data flow for adversarial WGAN (drawing style inspired by [27]). Here, G is the generator and $\mathbf{x}'$ is the generated adversarial example. F is the target model and $Z(\mathbf{x}')$ is the logits output with $\mathbf{x}'$ as input. D is the discriminator and $D(\mathbf{x})$ represents how probable D believes $\mathbf{x}$ is from the clean data.

Finally, we also want the difference between $\mathbf{x}'$ and $\mathbf{x}$ to be small. Thus, an extra penalty term on the magnitude of perturbation will be added. Here, we borrow Xiao's choice that $L_{\text{magnitude}}(G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})}[\max\{0, \|\mathbf{x}' - \mathbf{x}\|_2 - c\}]$ [27], where c is a hyperparameter.

Based on the above, the final loss function will be

$$L(D, G, F, t) = \alpha L_{\text{WGAN}}(D, G) - \beta L_{\text{adv}}(G, F, t) - \gamma L_{\text{magnitude}}(G) \tag{12}$$

where α, β and γ are hyperparameters for determining their relative importance. Thus, training an adversarial WGAN is equivalent to solving the optimization problem that $\max_G \min_D L(D, G, F, t)$. Algorithm 1 shows the full algorithm for training an adversarial WGAN, which is modified based on Algorithm 2 [26].

5 Experimental Results

5.1 Training Setup

The experiments we perform are mainly based on the MNIST [14] dataset because of the limitation of hardware. In experiments, we prepare three target models, which will be called them model A, B and C. The details of their architecture are provided in Table 5 in Appendix **??**. Among them, model A comes from the CNN tutorial of TensorFlow [28]. Model B is a variation of a target model used in Xiao's paper [27] and model C is designed by the author. Since nowadays, training a MNIST classifier is not difficult, they are not designed and trained very carefully. Table 1 provides their performance on the original MNIST test dataset.

The structures of generator and discriminator are also provided in Table 6, which are the variation of structures in Xiao's paper [27].

During the experiment, there are two notable differences between the actual implementation and the Algorithm 1. First, the three discriminators at line 8 in Algorithm 1 share convolutional layers

Algorithm 1 Adversarial Wasserstein GAN with Gradient Penalty [26]

```
Require: A target model's logits output $Z$ and a target label $t$; Learning rate $\eta$; $n_{\text{critic}}$ and batch size $m$; Penalty coefficient $\lambda$; Confidence $\kappa$; Hinge loss bound $c$; Relative importance factors $\gamma$, $\beta$, $\alpha$
1:  while $\boldsymbol{\theta}$ has not converged do
2:      for $t = 1, \ldots, n_{\text{critic}}$ do
3:          for $i = 1, \ldots, m$ do
4:              Sample real data $\mathbf{x} \sim p_{\text{data}}(\mathbf{x})$ and a random number $\epsilon \sim U(0, 1)$.
5:              $\mathbf{w} \leftarrow G_{\boldsymbol{\theta}}(\mathbf{x})$
6:              $\tilde{\mathbf{x}} \leftarrow \frac{1}{2}(\tanh(\mathbf{w}) + 1)$
7:              $\hat{\mathbf{x}} \leftarrow \epsilon\mathbf{x} + (1 - \epsilon)\tilde{\mathbf{x}}$
8:              $L_D^{(i)} \leftarrow D_{\boldsymbol{\phi}}(\hat{\mathbf{x}}) - D_{\boldsymbol{\phi}}(\mathbf{x}) + \lambda\left(\|\nabla_{\hat{\mathbf{x}}} D_{\boldsymbol{\phi}}(\hat{\mathbf{x}})\|_2 - 1\right)^2$
9:          end for
10:         $\boldsymbol{\phi} \leftarrow \text{ADAM}\left(\frac{1}{m}\nabla_{\boldsymbol{\phi}}\sum_{i=1}^{m} L_D^{(i)}, \boldsymbol{\phi}, \eta\right)$
11:     end for
12:     for $i = 1, \ldots, m$ do
13:         Sample real data $\mathbf{x} \sim p_{\text{data}}(\mathbf{x})$.
14:         $\tilde{\mathbf{x}} \leftarrow \frac{1}{2}(\tanh(G_{\boldsymbol{\theta}}(\mathbf{x})) + 1)$
15:         $L_{\text{adv}}^{(i)} \leftarrow \max\left\{\max_{j \neq t} Z(\tilde{\mathbf{x}})_j - Z(\tilde{\mathbf{x}})_t, -\kappa\right\}$
16:         $L_{\text{magnitude}}^{(i)} \leftarrow \max\{0, \|\tilde{\mathbf{x}} - \mathbf{x}\|_2 - c\}$
17:         $L_G^{(i)} \leftarrow \gamma L_{\text{adv}}^{(i)} + \beta L_{\text{magnitude}}^{(i)} - \alpha D_{\boldsymbol{\phi}}(\tilde{\mathbf{x}})$
18:     end for
19:     $\boldsymbol{\theta} \leftarrow \text{ADAM}\left(\frac{1}{m}\nabla_{\boldsymbol{\theta}}\sum_{i=1}^{m} L_G^{(i)}, \boldsymbol{\theta}, \eta\right)$
20: end while
```

Model A	Model B	Model C
99.07%	99.04%	99.37%

Table 1: Accuracies of three models on clean MNIST test dataset

but not the last dense layer. Although this variation indeed increases the model's expressivity, the specific reason that makes results better should need further exploration.

The second difference is that the transformation we used to preserve the constraint that $\mathbf{x}' \in [0, 1]^n$ is actually $\frac{1}{2}(\tanh(\mathbf{w}/\mu) + 1)$. Here, the extra hyperparameter μ is added so that the generator G tends to generate $\mathbf{w}$ with larger value. In the original setting without μ, the parameters of some layer in G are likely to become all zero because the values in each pixel are relatively small for grayscale image. After an all-zero layer appears, backpropagation cannot continue and adding a hyperparameter μ can effectively prevent this from happening. The values of μ we choose as well as all of other hyperparameters are provided in Table 2. As for the hyperparameters in ADAM optimizer, we use the default setting in TensorFlow implementation [28].

5.2 Attack Evaluation

In evaluation of our attack, the adversarial examples are crafted based on testing images of the MNIST dataset [14]. Table 3 shows the accuracies of each model on each adversarial generator.

Based on Table 3, we can see that our attack not only makes the accuracy of the target model very low, but also possesses strong transferability. In other words, the adversarial examples can also be used to attack a non-target model, which makes the black-box attack feasible [17].

For each attack, we also measure the time necessary for crafting adversarial examples of all 10000 testing images in the MNIST dataset. We further implement the FGSM and CW's optimization-based method and measure the amount of time they take for comparison. For FGSM, we choose

	G_A	G_B	G_C
Learning Rate η	1×10^{-5}	1×10^{-5}	1×10^{-6}
n_{critic}	5	5	5
Batch Size m	50	50	50
Penalty Coefficient λ	10	10	10
Confidence κ	0	0	0
Hinge Loss Bound c	0.3	0.3	0.3
Scaling Factor μ	10	10	40
Relative Importance Factor α	5	5	10
Relative Importance Factor β	17	17	17
Relative Importance Factor γ	1	1	10
target t	3	3	3

Table 2: Choice of hyperparameters for training. Here G_A means the generator used to attack model A and G_B, G_C follow the similar meaning.

	G_A	G_B	G_C
Model A	21.01%	12.48%	13.90%
Model B	19.03%	28.07%	9.58%
Model C	24.71%	17.21%	10.1%

Table 3: The accuracies when using model i to classify the adversarial examples generated by G_j for $i, j \in \{A, B, C\}$. For instance, when using model A to classify the adversarial examples generated by G_A, the accuracy is 21.01%.

$\epsilon = 0.2$ and for CW's method, we choose the default ADAM optimizer in TensorFlow [28] and learning rate being 3×10^{-3}. The GPU used to measure the time is GTX 1080Ti. Table 4 gives the numerical results and Figure 5 presents it in a bar graph.

G_A	G_B	G_C	FGSM$_A$	FGSM$_B$	FGSM$_C$	CW$_A$	CW$_B$	CW$_C$
0.803 s	0.785 s	0.749 s	0.960 s	0.912 s	3.435 s	349.3 s	305.9 s	2929.9 s

Table 4: Time (in seconds) necessary for each attack method to craft adversarial examples of 10000 MNIST tesing images.

We can see that for each target model, our method always has the fastest speed to craft adversarial examples. Furthermore, especially for CW's method, we can see that the amount of time for attack will significantly increase as the complexity of target model increases. By contrast, the amount of time necessary for our attack will not depend on the target model complexity. This property of our method can greatly benefits the procedure of adversarial training because it needs to incessantly generate adversarial examples [6].

Besides all above, the adversarial examples generated by our method also look natural to humans, especially compared to the examples presented in Xiao's paper [27]. Figure 6 shows such a comparison. For reference, in Figure 7, we also show a comparison between our adversarial examples and the corresponding clean images.

Finally, when evaluating how successful the targeted property of our attack is, we find some intriguing problems. In particular, although our attack is indeed targeted, the resulting target becomes different from the target we set. Furthermore, to the best of our knowledge, this is the first transferable targeted attack. However, the transferred target is again not the target we set during the training. Figure 8 uses a bar graph to show the details of this phenomenon based on adversarial examples created by G_A.

Roughly speaking, although we choose target label $t = 3$ to attack the model A, we can see that model A classifies most examples as 8, model B classifies most examples as 6 and only model

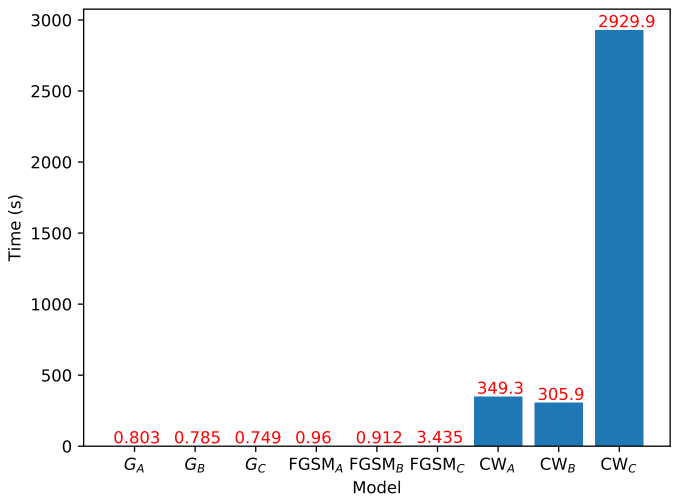

Figure 5: Bar graph showing the time data presented in Table 4

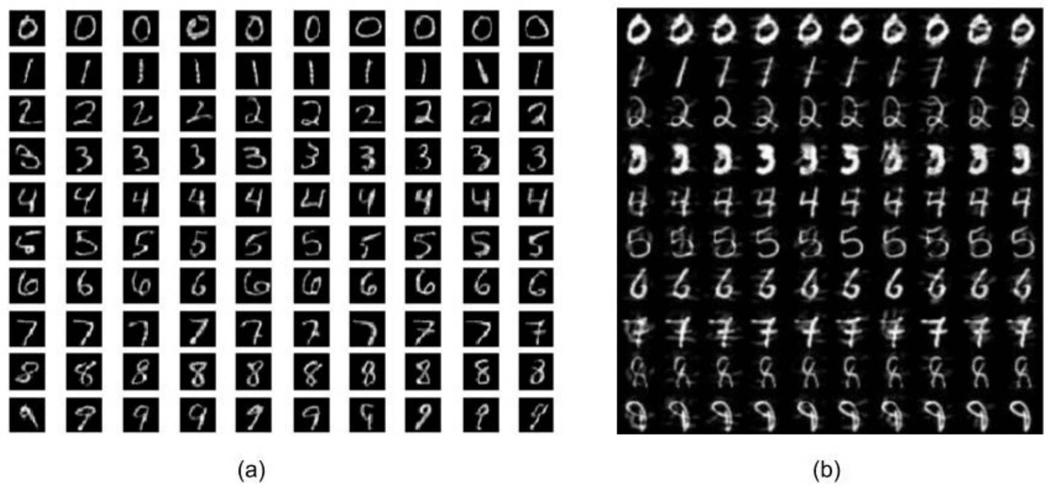

Figure 6: (a) are examples generated by G_A and (b) are examples presented in Xiao's paper [27]. It is clear that the images on left side look more natural to human.

C classifies most examples as 3. The current guess is that the problem may lie on the choice of $L_{\text{adv}}(G, F, t)$, but a further exploration will be left as future work.

6 Conclusion

In this paper, we presented a new method to generate adversarial examples by using Wasserstein GAN, which is an improvement of the method proposed by Xiao et al. [27]. Our method not only can generate state-of-the-art adversarial examples, which make the target model's accuracy very low and keep the example looking natural, but also have a very short running time that is independent of target model. To the best of our knowledge, this is the fastest attack among currently proposed methods. This property can greatly benefit the procedure of adversarial training.

In future work, besides addressing the target issue indicated in Figure 8, there are several other possible extensions to this method. First, an interesting question is if we really need the discriminator. CW's method can craft vivid adversarial examples by only applying penalty on Euclidean distance to

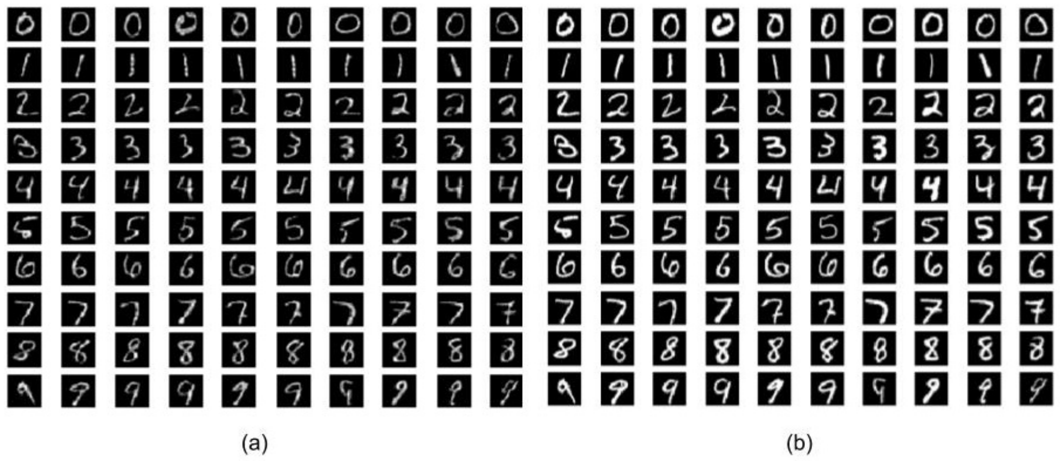

Figure 7: (a) are the same adversarial examples shown in Figure 6 and (b) are the corresponding clean images.

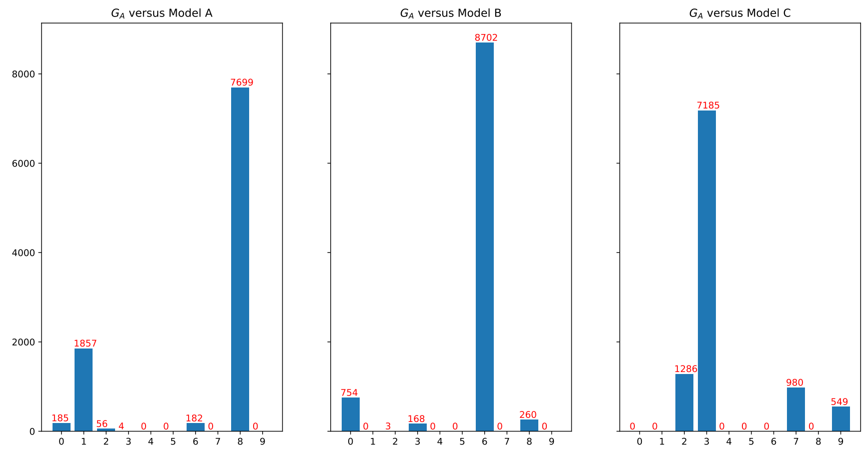

Figure 8: Here, the "G_A versus Model A" represents the result of using model A to classify the adversarial examples generated by G_A and the other two titles have similar meaning. In the subplot, the bar with (x, n) represents that there are n adversarial examples being classified as label x. The whole adversarial examples set is created based on the 10000 MNIST testing images.

the original [10], so it is possible that the discriminator is actually redundant in our method. Second, if we do not fix the parameters in the target model but train it and the generator simultaneously, can we end with a good generator and a robust neural network? Finally, we will also evaluate the strength of our attack on some currently proposed defense technique like distillation [13] and MagNet [29].

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[4] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of GANs for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*, 2017.

[5] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[6] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[7] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, et al. Going deeper with convolutions. CVPR, 2015.

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

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

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

[11] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, pages 2672–2680, 2014.

[12] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein GAN. *arXiv preprint arXiv:1701.07875*, 2017.

[13] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 582–597. IEEE, 2016.

[14] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[15] Pin-Yu Chen, Yash Sharma, Huan Zhang, Jinfeng Yi, and Cho-Jui Hsieh. EAD: elastic-net attacks to deep neural networks via adversarial examples. *arXiv preprint arXiv:1709.04114*, 2017.

[16] Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Xiaolin Hu, and Jun Zhu. Discovering adversarial examples with momentum. *CoRR*, abs/1710.06081, 2017.

[17] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 506–519. ACM, 2017.

[18] Seyed Mohsen Moosavi Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: A simple and accurate method to fool deep neural networks. In *Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, number EPFL-CONF-218057, 2016.

[19] Sara Sabour, Yanshuai Cao, Fartash Faghri, and David J Fleet. Adversarial manipulation of deep representations. *arXiv preprint arXiv:1511.05122*, 2015.

[20] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[21] Jonathan Uesato, Brendan O'Donoghue, Aaron van den Oord, and Pushmeet Kohli. Adversarial risk and the dangers of evaluating against weak attacks. *arXiv preprint arXiv:1802.05666*, 2018.

[22] Jiawei Su, Danilo Vasconcellos Vargas, and Sakurai Kouichi. One pixel attack for fooling deep neural networks. *arXiv preprint arXiv:1710.08864*, 2017.

[23] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial machine learning at scale. *arXiv preprint arXiv:1611.01236*, 2016.

[24] Adam Gibson, Chris Nicholson, and Josh Patterson. Deeplearing4j, 2017.

[25] Martin Arjovsky and Léon Bottou. Towards principled methods for training generative adversarial networks. *arXiv preprint arXiv:1701.04862*, 2017.

[26] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of Wasserstein GANs. In *Advances in Neural Information Processing Systems*, pages 5769–5779, 2017.

[27] Chaowei Xiao, Bo Li, Jun-Yan Zhu, Warren He, Mingyan Liu, and Dawn Song. Generating adversarial examples with adversarial networks. *arXiv preprint arXiv:1801.02610*, 2018.

[28] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[29] Dongyu Meng and Hao Chen. Magnet: A two-pronged defense against adversarial examples. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 135–147. ACM, 2017.

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

[31] Dmitry Ulyanov, Andrea Vedaldi, and Victor S. Lempitsky. Instance normalization: The missing ingredient for fast stylization. *CoRR*, abs/1607.08022, 2016.

Appendix

Architecture of Target Models

Table 5 for model architecture will follow these naming rules:

- "Conv($k, [k_1, k_2]$)" means a convolutional layer with k out channels, kernel size $[k_1, k_2]$ and stride $[1, 1]$.
- "BN" means a layer of batch normalization [30] with default parameters implemented in TensorFlow [28].
- "ReLU" means a ReLU activation function and "Softmax" means a softmax transformation function.
- "Dropout(p)" means a dropout layer with keep probability p.
- "MaxPooling($[k_1, k_2]$)" means a max pooling layer with kernel size $[k_1, k_2]$ and stride $[k_1, k_2]$.
- "FC(k)" means a fully connected layer with k output units.

A	B	C
Conv$(32, [5,5])$+ReLU	Conv$(32, [3,3])$+ReLU	Conv$(64, [8,8])$+BN+ReLU
MaxPooling$([2,2])$	Conv$(32, [3,3])$+ReLU	Conv$(128, [6,6])$+BN+ReLU
Conv$(64, [5,5])$+ReLU	MaxPooling$([2,2])$	Dropout$([0.5)$
FC(1024)+ReLU	Conv$(64, [3,3])$+ReLU	Conv$(128, [4,4])$+BN+ReLU
Dropout(0.5)	Conv$(64, [3,3])$+ReLU	FC(50)+ReLU
FC(10)+Softmax	MaxPooling$([2,2])$	Dropout(0.5)
	FC(100)+ReLU	FC(10)+ReLU+Softmax
	Dropout(0.5)	
	FC(10)+ReLU+Softmax	

Table 5: Model structure for target models. Among them, model A comes from the TensorFlow tutorial [28].

Architechture of GAN

We add the following naming rules for the architecture of generator and discriminator:

- "IN" means a layer of instance normalization [31] with default parameters implemented in TensorFlow [28].
- "Conv$(k, [k_1, k_2], [s_1, s_2])$" means a convolution layer with k out channels, kernel size $[k_1, k_2]$ and stride $[s_1, s_2]$.
- "HalfResi$(k, [k_1, k_2], [s_1, s_2])$" means a half residual block such that HalfResi$(\mathbf{x})$ =ReLU(Conv$(\mathbf{x})+\mathbf{x}$) and $k, [k_1, k_2], [s_1, s_2]$ are parameters of this convolution.
- "DeConv$(k, [k_1, k_2], [s_1, s_2])$" means a deconvolution layer with k out channels, kernel size $[k_1, k_2]$ and stride $[s_1, s_2]$.
- "LReLU(λ)" means a Leaky ReLU activation function which is defined as

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \lambda x & \text{if } x < 0 \end{cases} \tag{13}$$

Generator	Discriminator
Conv$(8, [3,3], [1,1])$+IN+ReLU	Conv$(8, [4,4], (22))$+IN+LReLU(0.2)
Conv$(16, [3,3], [2,2])$+IN+ReLU	Conv$(16, [4,4], (22))$+IN+LReLU(0.2)
Conv$(32, [3,3], [2,2])$+IN+ReLU	Conv$(32, [4,4], (22))$+IN+LReLU(0.2)
HalfResi$(32, [3,3], [1,1])$	FC(1)+ReLU
HalfResi$(32, [3,3], [1,1])$	
HalfResi$(32, [3,3], [1,1])$	
HalfResi$(32, [3,3], [1,1])$	
DeConv$(16, [3,3], [2,2])$+IN+ReLU	
DeConv$(8, [3,3], [2,2])$+IN+ReLU	
Conv$(1, [3,3], [1,1])$	

Table 6: Model structures for generator and discriminator

Algorithm for training WGAN

The full algorithm for training a Wasserstein GAN is provided in Algorithm 2 for comparison with Algorithm 1.

Algorithm 2 WGAN with gradient penalty [26]

```
Require: The gradient penalty coefficient $\lambda$, the number of critic iterations per generator iteration $n_{\text{critic}}$, the batch size $m$, Adam hyperparameters $\alpha$, $\beta_1$, $\beta_2$.
Require: initial discriminator parameters $\boldsymbol{\phi}_0$, initial generator parameters $\boldsymbol{\theta}_0$.
 1: while $\boldsymbol{\theta}$ has not converged do
 2:     for $t = 1, \ldots, n_{\text{critic}}$ do
 3:         for $i = 1, \ldots, m$ do
 4:             Sample real data $\mathbf{x} \sim \mathbb{P}_r$, latent variable $\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})$ and a random number $\epsilon \sim U(0,1)$.
 5:             $\tilde{\mathbf{x}} \leftarrow G_{\boldsymbol{\theta}}(\mathbf{x})$
 6:             $\hat{\mathbf{x}} \leftarrow \epsilon \mathbf{x} + (1-\epsilon)\tilde{\mathbf{x}}$
 7:             $L_D^{(i)} \leftarrow D_{\boldsymbol{\phi}}(\hat{\mathbf{x}}) - D_{\boldsymbol{\phi}}(\mathbf{x}) + \lambda\left(\left\|\nabla_{\hat{\mathbf{x}}} D_{\boldsymbol{\phi}}(\hat{\mathbf{x}})\right\|_2 - 1\right)^2$
 8:         end for
 9:         $\boldsymbol{\phi} \leftarrow \text{Adam}\left(\frac{1}{m}\nabla_{\boldsymbol{\phi}} \sum_{i=1}^{m} L_D^{(i)}, \boldsymbol{\phi}, \alpha, \beta_1, \beta_2\right)$
10:     end for
11:     Sample a batch of latent variables $\left\{\mathbf{z}^{(i)}\right\}_{i=1}^{m} \sim p_{\mathbf{z}}(\mathbf{z})$
12:     $\boldsymbol{\theta} \leftarrow \text{Adam}\left(\frac{1}{m}\nabla_{\boldsymbol{\theta}}\left[\sum_{i=1}^{m} -D_{\boldsymbol{\phi}}\left(G_{\boldsymbol{\theta}}\left(\mathbf{z}^{(i)}\right)\right)\right], \boldsymbol{\theta}, \alpha, \beta_1, \beta_2\right)$
13: end while
```