Adversarial Attacks on Classifiers

Pradyot Prakash

Computer Sciences

University of Wisconsin-Madison

December 20, 2018

Approval Sheet

A THESIS SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCES OF THE UNIVERSITY OF WISCONSIN-MADISON IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE.

Name	**Signature**
Prof. Dimitris Papailiopoulos Electrical and Computer Engineering (Advisor)	_____
Prof. Yingyu Liang Computer Sciences (Committee Member)	_____
Prof. Theodoros Rekatsinas Computer Sciences (Committee Member)	_____

Abstract

While Deep Neural Networks are powerful learning models that have widespread usage in pattern recognition applications, they are susceptible to adversarial examples — small, often imperceptible perturbations to images—that can cause a modern-day classifier to misclassify with a high probability. Through this project, we investigate two methods: one to perform adversarial attacks on one of the robust defense algorithm, the Robust Manifold Defense and the other to make the classifier robust using codes.

In the first part, our attack methods include approaches such as the Fast Gradient Sign Method (FGSM), Iterative FGSM (I-FGSM), Single-Pixel Attacks, Max-Min based approach and an algorithm that exploits the geometry of a Variational Autoencoder. Although we were not able to break the defense completely, we were able to force the algorithm to misclassify on MNIST digits 36% of the time using I-FGSM with $\epsilon = 0.2$. Furthermore, our FGSM attack achieved an improvement of at least 8 to 10% in misclassification as compared to the approach taken in the Robust Manifold Defense paper

In the second part, we try to use codes to make a linear classifier more robust against adversarial attacks. The results for this section weren't too encouraging as we had hoped for. For the l_∞ attack model, we observe the code based model is more robust for some cases while not adding any additional defense for some other cases. For the l_2 Carlini-Wagner attack, we didn't see any improvements.

Acknowledgements

I want to thank Prof. Dimitris Papailiopoulos for guiding me and providing valuable direction and inputs throughout the course of the study. A big shout out to Michael Fernandes and Geoffrey Lau for their assistance in the first part of the project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep Neural Networks (DNNs) have demonstrated excellent performance and proven to be powerful learning models that achieve state-of-the-art pattern recognition performance in areas such as Computer Vision, Speech Recognition and other domains such as drug discovery and genomics. However, due to imperfections during the training phase, DNNs are highly vulnerable to adversarial examples. Recent research [9] has revealed that the outputs of a DNN can easily be altered by adding some small, often imperceptible perturbations of the input data and lead to misclassifications in modern-day classifiers with high probability. Existence of these perturbations are especially problematic when these classifiers are deployed in the real world, such as in self-driving cars. One reason why defending against these adversarial attacks can be hard is because the natural image is perturbed into some space which is far from the manifold of natural images. Since classifiers aren't trained on images far from the natural images, it does not learn the representation of adversarial examples.

There have been several attempts to make DNNs robust to these adversarial attacks. However, defending against a wide range of attacks remains a challenge. After surveying the literature, the common techniques used to defend against adversarial attacks are through the use of gradient masking techniques. Athayle et al. [1] proposed techniques to overcome these obfuscated gradients and successfully attacked these defenses.

This motivated Ilyas et al. [6] to implement the Robust Manifold Defense. Their claims of being completely robust to gradient-based attacks motivated us to find several means to undo their defenses. These include other gradient-based attacks not assessed in their work such as the Iterative Fast Gradient Sign Method (I-FGSM), a game-theoretic approach that formulates a max-min optimization problem, and brute force single-pixel and N-pixel attacks.

These ideas all rely on the existence of a single classifier which performs multiclass classification. In the second part of the project, borrowing ideas from coding theory, we construct an ensemble of weak classifiers and use that to perform multiclass labelling. The way it works is by using a code to depict a class. Supposing we use length k codes, we train k classifiers, each one predicting the corresponding coordinate. Using all the predicted outputs, we assign the input a label.

Chapter 2

Attack models

2.1 l_∞ attacks

2.1.1 Fast gradient sign methods

The Fast Gradient Sign Method is a simple and quick gradient-based method to generate adversarial images, as described in the work by Goodfellow et al [4]. The method linearizes the cost function and solves for the perturbation that maximizes this cost subject to an l_∞ constraint. The equation to generate adversarial examples is:

$$x_{adv} = x + \epsilon \cdot \text{sign}\left(\nabla_x \ell(x, y, \theta)\right) \tag{2.1}$$

where ϵ is a hyper-parameter that determines the maximum perturbation to be applied to the original image. ℓ stands for the loss function defined in terms of the input x, the true label y and the parameters of the classifier, θ. In all our experiments, we have used the cross-entropy loss which is a common loss function used for classification.

A variant of the FGSM involves reducing the perturbation to smaller steps α and applying it over a specified number of iterations, otherwise known as Iterative-FGSM [7]. In performing the iterations, we make sure to clip pixel intensity values if they exceed some interval $[a, b]$. This is to ensure that the perturbation does not go beyond the ϵ-neighbourhood of the original image. In our experiments, we used images whose intensities were in $[0, 1]$. Iterative-FGSM can be understood by the following equation where $x_{adv,t}$ represents the adversarial value at an intermediate time instant t:

$$x_{adv,0} = x, \quad x_{adv,t+1} = \Pi_{x,\epsilon}\left(x_{adv,t} + \alpha \cdot \ \text{sign}(\nabla_x \ell(x_{adv,t}, y, \theta))\right) \tag{2.2}$$

where generally $\alpha < \epsilon$ where Π is the projection operator defined as follows, in the case where pixel values lie in the interval $[0, 1]$:

$$\Pi_{x,\epsilon}(x') = \min(1, x + \epsilon, \max(0, \ x - \epsilon, \ x')). \tag{2.3}$$

2.2 l_2 attacks

In this threat model, instead of looking at the l_∞ norm, the l_2 norm is used to impose constraints on the distance between the perturbed image and the original

image. It is modelled as the constrained minimization problem:

$$\begin{aligned} \text{minimize}_{xadv} \quad & \|x_{adv} - x\|_2^2 \\ \text{such that} \quad & C(x_{adv}) \neq c \\ & x_{adv} \in [0,1]^d \end{aligned} \tag{2.4}$$

where C is the classifier of interest and c is the true label of x. However, this problem can be very hard to solve and a relaxed version of it is often used:

$$\begin{aligned} \text{minimize}_{xadv} \quad & \|x_{adv} - x\|_2^2 + \tau \cdot \text{ penalty}(x_{adv}) \\ \text{such that} \quad & x_{adv} \in [0,1]^d \end{aligned} \tag{2.5}$$

where penalty(.) is some function responsible for enforcing incorrect classification and $\tau > 0$ is the trade-off parameter between the two terms.

2.2.1 Carlini-Wagner (CW) attack

The explicit constraint of $x_{adv} \in [0,1]^d$ can be painful to implement externally, so Carlini and Wagner [2] uses the hyperbolic tan function to enforce that the output lies in the required region. They solve the following optimization:

$$\text{minimize}_w \quad \|\frac{1}{2}(\tanh(w)+1) - x\|_2^2 + \tau \cdot f\left(\frac{1}{2}(\tanh(w)+1)\right) \tag{2.6}$$

with f defined as

$$f(x) = \max\{C(x)_i \mid i \neq c\} - C(x)_c \tag{2.7}$$

Intuitively if C outputs the probabilities or the logits, then the f function tries to increase the chances of selecting a class other than the correct class, c.

2.3 Pixel attacks

In pixel attacks, only a handful of the pixels in an image are modified by adding relatively small perturbations to the targeted pixels. Su, Vargas and Sakurai demonstrated in [8] that it is possible to alter the output of a DNN in an extremely limited scenario where only one pixel is modified in an image from the CIFAR-10 and ImageNet datasets. The targeted pixel is determined by Differential Evolution - a population-based optimization algorithm. An advantage of this algorithm is that gradient information is not required for optimization, hence it works even for objective functions that are non-differentiable or even unknown.

Chapter 3

Robust Manifold Defense

3.1 Motivation

The space of "natural" images is assumed to lie in a low dimensional manifold of the entire real space. The intuition behind the existence of adversarial examples is that they lie close to the space of the "natural" images but their intrinsic dimensionality is high. Since, the adversarial images are close to the original images, if we can somehow project the noisy image to a natural looking image and feed the denoised image to the classifier, the classifier should do a good job at classification. The Robust Manifold Defense paper exploits this idea and proposes a projection operator to make a classifier robust against adversarial examples.

Finding a projection operator onto a space S requires one to know the geometry of S. That is too strict a demand to make for the space of real looking images which one can expect to be highly non-convex. But some of the recent work in generative modelling tries to solve exactly this problem by using Variational Autoencoders (VAE) and Generative Adversarial Networks (GAN). VAEs and GANs have two parts to them – Encoder (E) and Decoder (D). E takes an input vector v and outputs a low-dimensional representation of v, say $z \in Z$. D uses z to construct a vector which is close to v. Training VAE and GANs require different approaches but they end up identifying the latent space Z quite well. The generator, though not mimicking a projection operator (because the input and output space of G are not of the same dimension), can be used to construct a projection operator.

Given an adversarial example, x_{adv}, we would like to identify a $z_{adv} \in Z$ such that $G(z_{adv}) \approx x_{adv}$. This can be modelled as solving $(\arg\min_{z_{adv}} \|G(z_{adv}) - x_{adv}\|)$. Once we have such a z_{adv}, $G(z_{adv})$ gives us a "natural" looking image such that it is similar to x_{adv}. With this, we define our projection operator, P, as follows:

$$P(x) = G\left(\arg\min_{z \in Z} \|G(z) - x\|_2^2\right). \tag{3.1}$$

This operator operates in two steps: (1) finds the closest latent space representation of x and (2) uses that to construct a real looking image close to x. For any adversarial example, this is essentially trying to approximate it with a smoothened version of the image which we hope looks like a "natural" image because the decoder G has been trained to generate such images.

3.2 The Robust Manifold Defense Model

We reproduce the Robust Manifold Defense Model described by Ilyas et al. [6]. The paper protects a classifier C by projecting its inputs onto the range of a given generator G (decoder of a Variational Autoencoder). The paper claims that this step provides the classifier with robustness against first-order and combined adversarial attacks. This step can be considered as a prior $P(x)$ defined on the space of natural looking images. The authors call this formulation the **Invert and Classify Algorithm** (*INC*).
The *INC* algorithm can be described as follows,

- z^* is obtained performing gradient descent in the z space to minimize $\|G(z) - x\|$. Ideally, $z^* = \arg\min_z \|G(z) - x\|$.
- The classifier is is applied on the projected input $G(z^*)$ which outputs the label of x.

The INC algorithm is described in Algorithm 1.

Algorithm 1: Invert and Classify

Input : Image x, Generator G, Classifier C
Output : Predicted label

begin
$z_0 \sim \mathcal{N}(0, 1)$
for $t \leftarrow 1$ *to* T **do**
  $z_t \leftarrow z_{t-1} - \eta \left(\nabla_z \|x - G(z)\|_2^2\right)$
end
$z^* \leftarrow z_t$
return $C(G(z^*))$

The invert and classify algorithm works as follows: given an input image x, the algorithm solves a minimization problem to find a z^* such that $G(z^*)$ is close to the given image x and is a smoothened version of the input image. $G(z^*)$ is fed to the classifier. The paper solves the minimization problem using gradient descent which makes it a non-differentiable method of projecting on the manifold. This would not be easy to attack since a non-fixed number of gradient steps are required [6].

3.3 Model configuration

Dataset The dataset we use for our project is the MNIST handwritten digit database. The images are grayscale images with pixel intensities lying in the interval $[0, 1]$. The MNIST images used in our experiments were not pre-processed.

Naive Classifier (NC) We looked at two classifier models for our experiments. The first one is a simple tensorflow classifier with the following network configuration:

- Input size: 28×28 images vectorized to 784 dimensional vectors

- Convolutional layer with ReLU activation followed by a max pooling layer: 5×5 kernel, 32 filters, 2×2 strides. Output size: $14 \times 14 \times 32$
- Convolutional layer with ReLU activation followed by a max pooling layer: 5×5 kernel, 64 filters, 2×2 strides. Output size: $7 \times 7 \times 64$
- Fully connected layer followed by a dropout: 1024 width
- Fully connected layer followed by softmax: 10 nodes with each node representing the probability of that class
- Argmax on the output of the previous layer to get the predicted class

The loss function was chosen to be the cross-entropy loss and the Adam optimization algorithm was employed to train the network to over 97% accuracy. We used a batch size of 32, 2000 epochs and a dropout rate of 0.5.

Generative modelling For the generator G, we consider the decoder of a Variational Autoencoder (VAE). The VAE architecture is as follows:

Encoder

- Input size: 784 dimensional vector
- Linear layer followed by a softplus layer: 500 width
- Linear layer followed by a softplus layer: 500 width
- Latent space of dimension 20, i.e., encoder outputs $\mu \in \mathbb{R}^{20}$ and $\Sigma \in \mathbb{R}^{20\times 20}$

Decoder

- Input sampled from μ and Σ to get a vector in $\mathbb{R}^{20}$
- Linear layer followed by a softplus layer: 500 width
- Linear layer followed by a softplus layer: 500 width
- Linear layer followed by sigmoid to get a vector in $[0, 1]^{784}$

Robust classifier (RC) The robust approach is exactly as defined in Algorithm 1.

We elaborate on the various attack techniques performed on the Robust Manifold Defense model in this section. Our attacks can be broadly divided into two major classes: gradient-based attacks and non-gradient based attacks. All our experiments were conducted using Tensorflow.

3.4 Gradient-Based Attacks

The novelty of Robust Manifold Defense stems from the fact that the gradients are not directly accessible. This makes it difficult to directly perform FGSM on it. The projection step is performed iteratively and the number of steps taken is not fixed. Hence, in order to attack the algorithm, we would have to attack the projection operation (Eq. 3.1). We compute x_{adv} as follows,

$$x_{adv} = x + \epsilon \cdot \text{sign}(\nabla_x \ell(C_\theta(P(x)), y)) \tag{3.2}$$

where C_θ is our classifier defined using the parameters θ. One idea would be to rely on numerical methods to get the gradient. We can use first principles and go by the definition of gradient and use that instead of the analytic gradient. To simplify notation, let, $f(x) = \ell(C_\theta(P(x)), y)$ and $x \in \mathbb{R}^d$. Let h be a small scalar which is used in the definition of the gradient. Also, let h_j denote the d dimensional vector with all coordinates but the jth coordinate set to h. With these, the algorithm has been described in Algorithm 2.

Algorithm 2: Numerical gradient

Input : function f, $h \in \mathbb{R}$, $x \in \mathbb{R}^d$
Output : $\nabla_x f(x)$
for $j \leftarrow 1$ *to* d **do**
    $(\nabla f(x))_j \leftarrow \frac{f(x+h_j) - f(x-h_j)}{2h}$
end
return $\nabla_x f(x)$

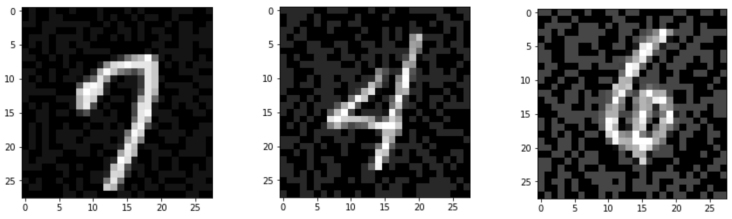
3.4.1 Fast Gradient Sign Method (FGSM)

In our experiments, we picked $\epsilon \in \{0.5, 1.0, 2.0\}$ and $h = 0.001$. Table 3.1 summarizes the inaccuracy in classification of the adversarial examples generated.

ϵ	# of adversarial examples generated	Misclassification rate (%)
0.05	150	19.3
0.1	150	22
0.2	150	24.7

Table 3.1: Misclassification rate using FGSM on the robust classifier

The success rate for misclassifying adversarial examples on the MNIST dataset is better than that reported by Ilyas et al. [6] for their FGSM attack. In their experiments, misclassification rates were at most 10% for $\epsilon \leq 0.2$. It should also be observed that the larger ϵ is, the noisier the generated image is and the more perceptible the perturbation is to the human eye. Some examples of MNIST digit images generated by FGSM are shown in Figure 3.1.



(a) $\epsilon = 0.05$, original label 7, classified as 9

(b) $\epsilon = 0.1$, original label 4, classified as 7

(c) $\epsilon = 0.2$, original label 6, classified as 2

Figure 3.1: Adversarial examples generated by FGSM on the robust classifier

3.4.2 Iterative Fast Gradient Sign Method (I-FGSM)

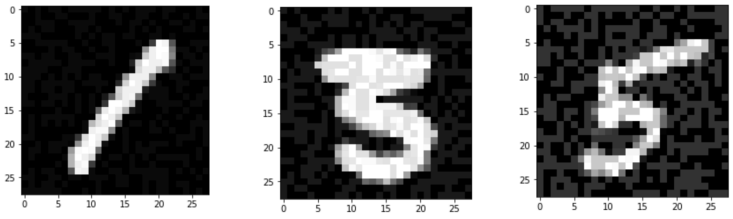
In order to try and improve the misclassification rate, a separate code was run to determine an average number of iterations of I-FGSM to be performed to generate adversarial examples with high confidence of misclassification. This average number of iterations turns out to be 6, but for the purpose of our experiments with I-FGSM, 5 iterations were performed. Table 3.2 summarizes the results obtained using I-FGSM to generate adversarial examples with $h = 0.001$.

ϵ	# of adversarial examples generated	Misclassification rate (%)
0.05	25	8
0.1	25	20
0.2	25	36

Table 3.2: Misclassification rate using I-FGSM on the robust classifier

Compared to the vanilla FGSM, I-FGSM produces adversarial images that achieves larger misclassification rate for larger ϵ but is unable to match the performance when ϵ is less than 0.1. Sample adversarial images generated by I-FGSM are presented in Figure 3.2. Adversarial examples generated by I-FGSM are perceivably less noisy as compared to their counterparts created through FGSM (as in Figure 3.1), and this difference is more obvious at larger values of ϵ.

I-FGSM seems to be a better algorithm to create adversarial examples. However, there is a trade-off as we need to perform more computation to create a single example. This is further made worse because the numerical gradient computation is inherently slow because we need to iterate over each pixel of the image.



(a) $\epsilon = 0.05$, original label 1, classified as 3

(b) $\epsilon = 0.1$, original label 3, classified as 8

(c) $\epsilon = 0.2$, original label 5, classified as 8

Figure 3.2: Adversarial Examples Generated by I-FGSM on the robust classifier

3.5 Max-Min attack

The above approaches used the gradient information to construct adversarial examples using the FGSM algorithm. However, the projection step (Eq. 3.1) is robust enough and the misclassification rate is low. If we dig deeper into the potential reason for this, it might be because the projection operator is able to remove the adversarial perturbations for most of the adversarially constructed images. So to defeat the Robust Classifier (RC), if we can find ways to misdirect the generator G to a "bad" z, then one can hope that the classifier following that would predict an incorrect label.

This motivates our game theoretic formulation. Let v be the adversarial perturbation that is added to our original image x and hence $x_{adv} = x + v$. The projection step tries to solve the optimization problem,

$$\min_z \|G(z) - x_{adv}\|_2^2 = \min_z \|G(z) - (x + v)\|_2^2. \tag{3.3}$$

If we can design our v such that whatever the G does, we try to make it worse. This can be better understood through the following optimization problem,

$$\max_{\|v\|_\infty \leq \epsilon} \min_z \|G(z) - (x + v)\|_2^2. \tag{3.4}$$

Hence, we try to find a perturbation v which tries to make the projection step worse. By imposing the constraint that $\|v\|_\infty \leq \epsilon$, we ensure that the adversarial perturbations are small and the image $x_{adv} = x + v$ still resembles the original image x. Thus, when such a $x + v$ is fed to RC, we might hope that the projection operation doesn't go through as nicely one would have hoped and the classifier would misclassify with a higher probability. Note that this formulation is not optimal because we would want to maximize the misclassification rate and this does not factor that in. But this is the best we can do since any formulation involving the classifier C would be intractable experimentally because the projection operator doesn't have a closed form expression.

Relaxing the l_∞ constraint, the above optimization problem is equivalent to solving two sub-problems (SP_1 and SP_2),

$$\begin{aligned} SP_1: &\quad \min_z \|G(z) - (x + v)\|_2^2 \\ SP_2: &\quad \min_v -\|G(z) - (x + v)\|_2^2 + \lambda \|v\|_\infty \end{aligned} \tag{3.5}$$

where λ regulates the size of v.

3.5.1 Experiments

The VAE architecture described earlier was used for the experiments. Tensorflow's Gradient Descent Optimizer with a learning rate of 0.01 was used to optimize the two sub-problems. This experiment was not too stable due to the difficulty involved in solving two sub problems of varying difficulties. Note that solving this problem is similar to the training of GANs which is known to be unstable and highly dependent on the parameter initialization. Furthermore, since SP_1 is generally a harder problem to optimize (because of the presence of a non-convex G), for every optimization step of SP_2, we take multiple gradient descent steps for SP_1. We choose this number to be 5.

To solve both SP_1 and SP_2, we used a $\gamma = 50$. Such a large value of γ was important to constrain the size of v to be small. With such hyper-parameter settings, we iterated for 10000 times until the value of $\|G(z) - (x + v)\|$ had dropped down to about 0.03. Even with tuning γ extensively, we were able to find v which had high $\|v\|_\infty$. Figure 3.3 shows an image generated by this process. Unfortunately, even with such a high perturbation, the RC did not misclassify.

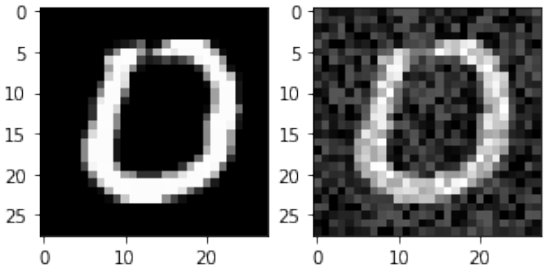


Figure 3.3: Max-min attack. Left: original image, right: image obtained by solving the optimization problem 3.4. $\|v\|_\infty = 0.25$, original label: 1, predicted label: 1

3.6 Attack by exploiting the geometry of Variational Autoencoders

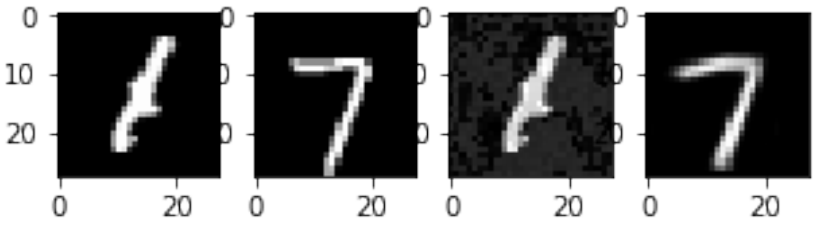
Variational Autoencoders identify parts of the latent space which correspond to objects which resemble each other closely. Since we wish to misclassify an image x, if we can add some small noise v to it such that the encoder E of VAE identifies a latent space for $x + v$ which is different from that of x, we can possibly try to fool the decoder G. So if we solve the optimization problem,

$$\min_{\|v\|_\infty \leq \epsilon} \|VAE(x + v) - x'\|_2^2 \tag{3.6}$$

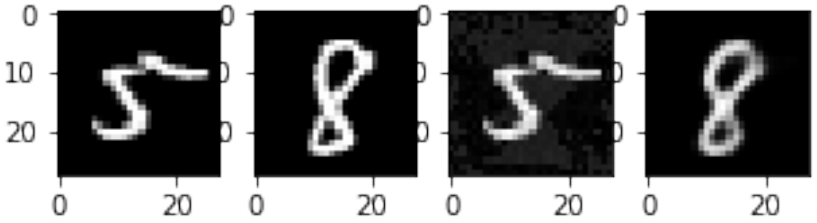
where $VAE(.)$ represents the output of the Variational Autoencoder and $x' \neq x$ is the target image that we want the VAE to output. By keeping the l_∞ norm of v to be small, we ensure that $x + v$ looks similar to the original image x. Relaxing the l_∞ constraint, we get the problem,

$$\min_v \|VAE(x + v) - x'\|_2^2 + \gamma\|v\|_\infty \tag{3.7}$$

where γ controls the size of v.



(a) $\|v\|_\infty = 0.10$, original label: 1, predicted label: 8



(b) $\|v\|_\infty = 0.08$, original label: 5, predicted label: 5

Figure 3.4: From left to right: original image (x), target image (x'), adversarial image ($x_{adv} = x + v$), and output of VAE ($VAE(x_{adv})$). (a) shows an image which got misclassified by RC and (b) shows an image which was classified correctly by RC

3.6.1 Experiments

The same VAE architecture described in a previous section was used. The experiments were performed with $\gamma = 0.5$ as it led to the best value of $\|v\|_\infty \leq 0.1$. Eq. 3.7 was solved by using the Gradient Descent Optimizer of Tensorflow by running the optimization for 10000 iterations. The iterations converged quite quickly with a learning rate of 0.1. $x_{adv} = x + v$ was fed as input to RC and the misclassification rate turned out to be around **17%**.

Figure 3.4 plots images for two digits generated in the experiment. It is evident that the optimization program is able to find small perturbations which can make the VAE predict a different digit for these two cases. This was our observation for all the digits too—there exist small perturbations which can help us get a target image $x_\prime$. However, as we noted earlier that the misclassification rate with this procedure was only 17% suggesting that the robust method is able to remove the adversarial noise for most of the inputs. This can be explained by noting that even though we are able to fool the encoder to represent the input digit differently, the projection step $P(x)$ is quite robust and multiple iterations involved in the projection step are able to remove the adversarial noise quite well so that the classification turns out to be correct for the majority of digits.

3.7 Single-Pixel and N-Pixel Attacks

In our pixel attack, we explore two brute-force variations of the pixel attack. For each experiment, a proportion of pixels in the MNIST digit images were randomly chosen and modified in two separate ways:

- Saturation: of the pixels to be attacked in the original image, if their intensities

are less than 0.5, they are set to 1. Conversely, if the pixel's intensity is greater than 0.5, they are set to 0.

$$new\_pixel\_value = \begin{cases} 1 & if\ pixel\_value < 0.5 \\ 0 & if\ pixel\_value \geq 0.5 \end{cases} \tag{3.8}$$

- Midpoint: regardless of pixel intensity in the original image, the pixel intensity is set to 0.5.

$$new\_pixel\_value = 0.5 \tag{3.9}$$

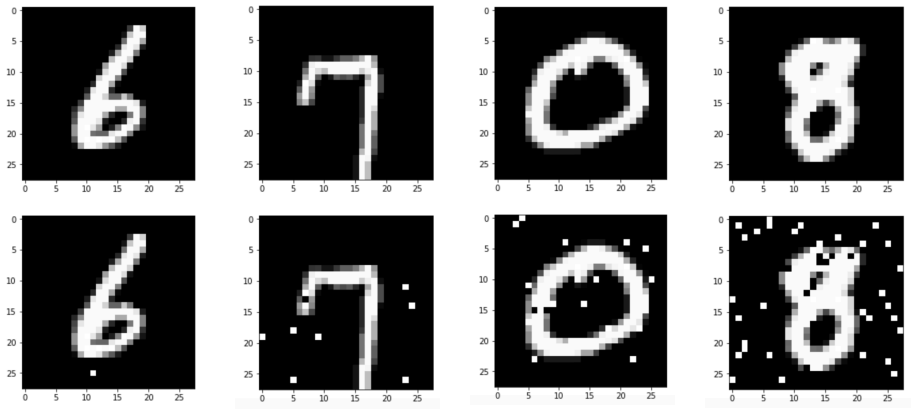
We perform pixel attacks on $N \in \{1, 8, 16, 40, 78\}$ pixels of the MNIST digits which comprise 784 pixels each. This corresponds to a proportion of 0.001%, 1%, 2%, 5% and 10% of the pixels in each image respectively. 1000 adversarial examples were generated in each experimental run and the results are reported in Tables 3.3 and 3.4. Sample adversarial images which we generated are shown in Figures 3.5 and 3.6.

Pixels attacked (%)	# of pixels attacked (N)	# of adversarial examples generated	Misclassification rate (%)
0.001	1	1000	18.5
1	8	1000	19.5
2	16	1000	20.3
5	40	1000	21.8
10	78	1000	28.2

Table 3.3: Misclassification rate of adversarial examples generated by N-pixel attack with varying N, using Saturation method

From Table 3.3, misclassification rate increases as we increase the percentage of pixels that undergo perturbations. However, the difference in misclassification rate is only 3.3% for a nearly 5% increase in the number of pixels under attack.

In Table 3.4, we observe that the Midpoint method for N-pixel attack does not result in perceivable impact to the classifier's performance on the adversarial examples. Comparing the adversarial images generated by the two variants of the N-pixel attack method in Figures 3.5 and 3.6, it is obvious that the Saturation method creates adversaries that are more perceptible to the human eye.

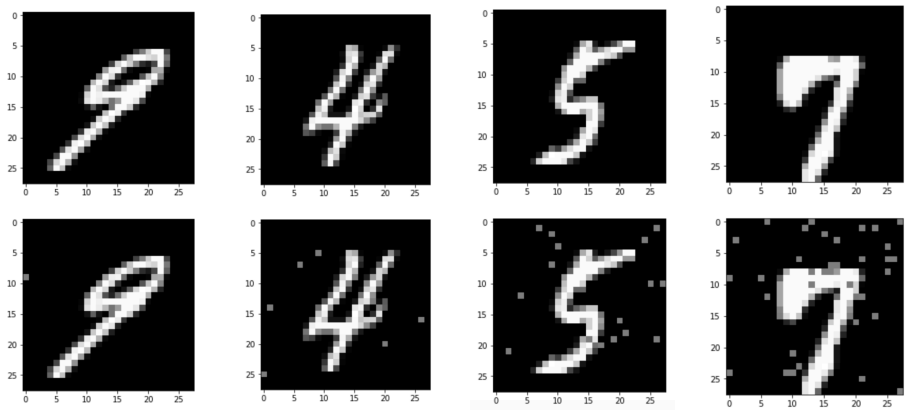


(a) 1-pixel attack (b) 8-pixel attack (c) 16-pixel attack (d) 40-pixel attack

Figure 3.5: Adversarial examples by N-pixel attack. Top row: original images, bottom row: adversarial examples using the Saturation method

% of pixels attacked	# of pixels attacked (N)	# of adversarial examples	Misclassification Rate (%)
0.001	1	1000	20.5
1	8	1000	19.0
2	16	1000	20.2
5	40	1000	19.2
10	78	1000	22.7

Table 3.4: Misclassification rate using N-pixel attack using the Midpoint method



(a) 1-pixel attack (b) 8-pixel attack (c) 16-pixel attack (d) 40-pixel attack

Figure 3.6: Adversarial examples by N-pixel attack. Top row: original images, bottom row: adversarial examples using the Midpoint method

Chapter 4

Ensemble of classifiers

The earlier chapter involved working with a single classifier which did multiclass labelling. To cause the classifier to predict an incorrect label, we were given an ϵ budget which controlled the distance between the input (x) and the adversarial image corresponding to it (x_{adv}) such that $\|x - x_{adv}\| \leq \epsilon$. We looked at different models where the $\|.\|$ function varied.

Creating adversarial examples imposes the restriction that x_{adv} should be close to x. With only a single classifier, the adversary gets to use all the ϵ energy on the one classifier. This motivates the idea that perhaps by using more than one classifiers, we can make the job of the adversary harder. If we can force the adversary to split the ϵ budget into smaller fractions such that the individual classifiers are robust enough, then perhaps the whole attack would fail.

4.1 A collection of linear classifiers

To motivate this better, let's look at simple case of binary classification using the linear function, $f(x) = \langle \theta, x \rangle$. We output the label to be the sign of $f(x)$, i.e., $C^0(x) = \text{sign}\ (f(x))$. Under the l_2 attack model, the best adversarial example is,

$$x_{adv} = x - \frac{y\epsilon\theta}{\|\theta\|_2} \tag{4.1}$$

where $y \in \{1, -1\}$ is the true label of x. On feeding this as input to f, we get $f(x_{adv}) = \langle \theta, x_{adv} \rangle = \left\langle \theta, x - \frac{y\epsilon\theta}{\|\theta\|_2} \right\rangle = f(x) - y\epsilon\|\theta\|_2$. A misclassification will be caused if $\epsilon\|\theta\|_2$ is large enough. This tells us that if we can somehow manage to keep the size of θ small, then the model has a higher degree of robustness.

Now consider an alternate story where we have three linear functions of the form $f_i(x) = \langle \theta_i, x \rangle$. The classifier $C^1(x)$ outputs the majority label of the three functions. So to cause misclassification, at least two functions needs to change signs. Being a linear model, we can guess that the best adversarial example will take the form,

$$x_{adv} = x + y\epsilon \left(\alpha_1 \frac{\theta_1}{\|\theta_1\|} + \alpha_2 \frac{\theta_2}{\|\theta_2\|} + \alpha_3 \frac{\theta_3}{\|\theta_3\|} \right) \tag{4.2}$$

such that $\|x_{adv} - x\| = \epsilon$. Without loss of generality, let's say that we wish for f_1 and f_2 to change signs and don't care about f_3. Let's look at two cases:

1. $\langle \theta_i, \theta_j \rangle = 0 \; \forall \; i \neq j$. This tells us that $\sum_i \alpha_i^2 = 1$ from the closeness constraint. We also want $f_j(x_{adv}) < f_j(x)$ for $j \in \{1, 2\}$. From this we have, $f_j(x_{adv}) = \langle \theta_j, x_{adv} \rangle = f_j(x) + y\epsilon\alpha_j \|\theta_j\|$ and so we want $y\alpha_j < 0$. Whatever way we choose the αs, the magnitude of $y\epsilon\alpha_j \|\theta_j\|$ would be smaller than the case with just one classifier. This makes the possibility of attack less likely because with a smaller perturbation, one may not be able to cross the decision boundary.

2. In the other case where the decision boundaries, θ_i, are not orthogonal, because of non-zero inner products, $\langle \theta_i, \theta_j \rangle$, the magnitude of the αs would even be smaller making the attack even harder. If we wanted f_3 to also change signs, that would be even harder because then $\alpha_3 \neq 0$ making the magnitudes of other αs further smaller.

This simple analysis suggests that by using multiple classifiers, we might be able to make the classifications more robust. The key takeaway is that the attack budget per classifier gets reduced. We also learn the following lessons as to how we should probably proceed:

1. try to keep the norm of the θ's smaller, and;

2. make the θ's orthogonal. For general gradient based attacks, this is equivalent to saying that the gradients of the various losses (w.r.t., x) should be orthogonal. For the linear case, the gradient is simply θ.

However, these present a few practical difficulties. While increasing the number of classifiers makes the model more robust, the memory cost also increases proportionally. Also we want all the independent classifiers to have good prediction powers. Coming up with a large number of equally accurate classifiers might not be always possible. While different random initialization might help, the learning algorithm determines to a large extent the exact classifier parameters learned. In a sufficiently high dimensional space, it is relatively less difficult to find orthogonal and equally capable parameters defining the classifiers, but is still not easy.

4.2 Multiclass learning

The above sections described a few simple ideas for robust classification but they worked only for the binary case. It is unclear as to how should one extend the idea to the multiclass setting. One way to extend these binary classifiers to the multiclass setting is through the use of codes.

Let's say we have m classes over which we want to build a classifier. It is interesting to note that there are multiple ways to describe the m classes of interest. The simplest and perhaps the most common is to call them numbers from the set $\{1, 2, \ldots, m\}$. Another way would be to represent them in binary and use them as labels. Or use any other arbitrary number system. The interesting thing to note here is that however we represent them, we are defining a mapping from the set $\{1, 2, \ldots, m\}$ to their representation. For instance, let $m = 7$ and we use the base 2 and base 3 number systems to represent them. Table 4.1 succinctly summarizes these mappings.

The idea to note here is that if the representation takes k characters then we can construct k functions, each mapping the decimal number to the corresponding bit in

Decimal	Binary	Ternary
1	001	01
2	010	02
3	011	10
4	100	11
5	101	12
6	110	20
7	111	21

Table 4.1: Binary and ternary representations of the integers 1 to 7

the chosen representation. Since we know how to do binary classification and work with two classes, let's pick the binary system. Suppose that we choose to represent our m classes using k bits. These bits need not represent the base 2 versions of the decimal numbers. We define k functions, $(f_1, f_2, \ldots, f_k)$ such that for each i,

$$f_i : \{1, 2, \ldots, m\} \to \{0, 1\} \tag{4.3}$$

Using our learning algorithms, we can learn an estimator for each f_i based on the training dataset using the binary classification ideas and from that a multiclass learning model can be devised. The following sections details this.

4.3 Multiclass classification model

Suppose we have a classification task over m classes with the code length being k. This means that we need to output a vector of k 0-1 values. Let the m labels be represented by $L = (l^1, l^2 \ldots, l^m)$. Each of l^i is a vector of length k over $\Sigma^k = \{0, 1\}^k$. We have a collection of k classification functions, $C_1, C_2, \ldots, C_k$ where C_i outputs the probability of of the i^{th} bit being a 1. The classification algorithm works as follows:

$$x \to C(x) = \begin{bmatrix} C_1(x) \\ C_2(x) \\ \vdots \\ C_k(x) \end{bmatrix} \to \hat{y}(x) = \begin{bmatrix} \hat{y}_1(x) \\ \hat{y}_2(x) \\ \vdots \\ \hat{y}_k(x) \end{bmatrix} \to D(\hat{y}(x)) = \text{the output class} \tag{4.4}$$

where

$$\hat{y}_i(x) = \begin{cases} 1 & C_i(x) \geq 0.5 \\ 0 & o/w \end{cases} \tag{4.5}$$

with D being a decoder which outputs an index $\in [m]$ defined as follows,

$$D(v) \in \arg\min_{i \in [m]} d_H(v, l^i). \tag{4.6}$$

$d_H(u, v)$ is defined to be the Hamming distance between $u, v \in \Sigma^k$. It is possible that there are multiple classes which are equidistant from $v \in \Sigma^k$, in which case D outputs any of the possible labels arbitrarily. To avoid such a scenario, we would want codes $l^1, \ldots, l^m$ to be well spaced out so that only one class minimizes the Hamming distance. Assume that the Hamming distance of L is d.

$$d = \min_{i \neq j} d_H(l^i, l^j) \tag{4.7}$$

4.3.1 Probability measure

In our model, we can define the probability of a input belonging to a particular class using the Hamming distance, d_H. Let $P(c|v)$ represent the probability of $v \in \Sigma^k$ belonging to the class c. We want P to have certain properties:

1. The probability should be maximum for the class to which the Hamming distance is minimum for v. This tells us that the probability should be a decreasing function of the Hamming distance.

2. For well separated code words, the true class should have a probability close to 1 and other classes should have extremely small probabilities.

A function of the form,

$$P(c|v) = \frac{e^{k - d_H(v, l^c)}}{\sum_{i=1}^{m} e^{k - d_H(v, l^i)}} = \frac{e^{-d_H(v, l^c)}}{\sum_{i=1}^{m} e^{-d_H(v, l^i)}} \tag{4.8}$$

is a probability measure and satisfies the above two requirements. This clearly is a decreasing function of d_H. Since, d_H counts the number of differing bits, the values of d_H changes by multiples of 1. Because the exponential function changes quite rapidly, it squashes the probability of a farther class label and so satisfies requirement (2). One may note that this is similar to be softmax function which enjoys some nice desirable properties. We will use this probability model to construct an attack on our classifiers.

4.3.2 Codes

We want the labels, L, to be unique and that requires at least $k \geq \lceil \log_2(m) \rceil$ bits. There are multiple choices to select the codes. Some of the common ideas are:

- **Semantic codes**: The labels could be chosen semantically to highlight the structure of the object to be classifier. For instance, [3] designs the labels for the MNIST dataset accounting for the shape of the digits. While an interesting idea, this becomes increasingly harder task as the classes of interest become more complex, as is the case for the CIFAR or ImageNet datasets.

- **Identity codes**: This is the one-hot encoding for the m classes. All the coordinates of l^i are 0 except the i^{th} coordinate which is 1.

- **Random codes**: Each coordinate of l^i is sampled randomly from the Bernoulli distribution with parameter p. These are well separated from each other in the Hamming distance and serve our purpose well. One potential downside could be that since the codes are random, there is no semantic information captured in the codes.

- **Error correcting codes**: Codes such as the Hamming codes, BCH codes, etc.

We use Identity codes and Random codes (with $p = \frac{1}{2}$) in our experiments.

4.3.3 Notion of adversarial attack

Our goal is to find an x_{adv} which solves the optimization:

$$\begin{aligned} &\text{minimize} \quad \|x_{adv} - x\| \\ &\text{subject to} \quad D(\hat{y}(x_{adv})) \neq D(\hat{y}(x)) \end{aligned} \tag{4.9}$$

For an adversarial example to exist in this model, at least $\lfloor \frac{d-1}{2} \rfloor$ of the classifiers should output incorrect labels. Therefore to design a robust classification algorithm, we need to have at most $\lfloor \frac{d-1}{2} \rfloor$ of the classifiers be fooled by the adversary. Note that the adversarial attack model and the class of classifiers would help define (k, L) and consequently d. Working with this definition is not easy because of combinatorial constraints.

Let's look at a different formulation. To train such a joint classifier, a typical idea is to look at the sum of the independent losses along with a regularizer. Suppose the loss for the i^{th} classifier is denoted by $\ell_i(x, y_i, \theta_i)$ where y_i is the i^{th} coordinate of the true label of x and θ_i is the parameter defining the i^{th} classifier. Putting everything together, we get,

$$\ell = \sum_{i=1}^{k} \beta_i \ell_i(x, y_i, \theta_i) + \mu R(\theta_1, \ldots, \theta_k) \tag{4.10}$$

$$\beta_i \geq 0, \ \sum_{i=1}^{k} \beta_i = 1$$

where R is a regularization function on the weights and the β_i represent the weights given to the different losses. Since each of the classifiers are doing binary classification, the cross-entropy loss is a natural choice for ℓ_i.

4.3.4 Two l_∞ attack models

For this section, assume no regularization, i.e., $R = 0$. Constructing an FGSM adversarial example for a single classifier would be straightforward, but to do it for multiple classifiers requires some thought. Two potential attacks are:

1. Taking motivation from Equation 4.2 of section 4.1, we can construct an adversarial example for the k different classifiers independently and take their linear combination to come up with an adversarial example for the joint model.

$$x_{adv} = x + \epsilon \cdot \sum_{i=1}^{k} \beta_i \ \text{sign} \ (\nabla_x \ell_i(x, y_i, \theta_i)) \tag{4.11}$$

2. Alternately, we can look at the joint loss function ℓ and perform an attack of the kind,

$$x_{adv} = x + \epsilon \cdot \text{sign} \ \left(\sum_{i=1}^{k} \beta_i \nabla_x \ell_i(x, y_i, \theta_i) \right) \tag{4.12}$$

Both of these attacks respect the $\epsilon - l_\infty$ budget constraint and thus, are valid attacks. For our case, since all the k classifiers are equivalent (by virtue of the bits being sampled from Bernoulli with $p = \frac{1}{2}$ or the codes being Identity codes), all the $\beta_i = \frac{1}{k}$.

4.3.5 Logistic classifier

Let us make these ideas concrete for the case of the linear logistic classifier. Each C_i independently learns the parameters for the i^{th} coordinate. Suppose each C_i is represented by a parameter θ_i which gives us a family of classifiers, $C_i(x) = \sigma(x; \theta_i)$. Here, σ is the logistic function defined as,

$$\sigma(x; \theta) = \frac{1}{1 + e^{-\langle \theta, x \rangle}} \tag{4.13}$$

The cross-entropy loss gives us,

$$\ell_i(x, y_i, \theta_i) = -y_i \log\left(\sigma(x; \theta_i)\right) - (1 - y_i) \log\left(1 - \sigma(x; \theta_i)\right) \tag{4.14}$$

and the gradient of the loss with respect to the input becomes,

$$\nabla_x \ell_i(x, y_i, \theta_i) = \left(\sigma(x; \theta_i) - y_i\right) \theta_i \tag{4.15}$$

Hence, for the i^{th} classifier, $\nabla_x \ell_i(x, y_i, \theta_i) = \left(C_i(x) - y_i\right) \theta_i$ where y_i is the i^{th} coordinate of the true label of x.

We trained two models: one with identity codes and the other with random codes. The accuracies of the classifiers have been detailed in Table 4.2. Since the random codes could be different for two different runs, the two sub-columns under the 'Random codes' column list the accuracy of the model with two different code words.

4.4 A probabilistic attack model

Earlier we described a model to construct adversarial examples and using that discussed a direct gradient based approach to computing adversarial examples. This required assuming that the adversary had access to the gradient of the loss functions and that made it very simple to use that. What if the adversary doesn't have access to the gradient? In this section we define a new attack model where instead of the gradient, the adversary is only given a function which outputs the probabilities of an input x belonging to the m different classes.

Formally, the adversary has access to a probability function, $P(y|x)$ which outputs the probability that x belongs to the class y. At first thought it would seem weird to use this to come up with an adversarial example. But if we think about the problem of trying to minimize the probability of the correct class, then we have a potential direction of approach. Let the true label of x be c.

As we know that the gradient of a function gives the direction in which the function increases by the maximum value and consequently the negative gradient is the direction of maximum descent. So if we can somehow compute $-\nabla_x P(c|x)$, then by perturbing x slightly in that direction, we can potentially make the classifier make a false prediction. This is a strong insight and can be made to work if the gradient of the probability function can be obtained. As in the Robust Manifold Defense chapter, the numerical gradient comes to our rescue!

k	Identity code	Random code	
		Expt. 1	Expt. 2
4	-	49.7	41.66
5	-	53.71	55.09
6	-	55.1	76.04
7	-	63.1	75.76
8	-	68.64	76.04
9	-	78.89	79.81
10	80.01	79.09	79.96
11	-	80.58	80.12
12	-	82.56	79.76
13	-	83.3	80.7
14	-	83.44	83.44
15	-	83.01	83.42
16	-	83.36	83.18
17	-	82.89	84.16
18	-	82.9	84.25
19	-	83.51	84.11
20	-	84.57	84.3

Table 4.2: Accuracy (%) of logistic classifier with different code lengths. The two sub-columns under 'Random codes' denote two runs with different randomly generated codes

4.4.1 NES gradient estimate

Earlier we went with the definition of the gradient and used that to estimate the numerical gradient. That approach although correct, can be replaced with better computationally tractable algorithms such as the Natural Evolutionary Strategies (NES) as used in [5]. NES has been explained in detail in [10] and we summarize it in Algorithm 3. NES basically samples perturbation values around the input x from a Gaussian and uses that to get the gradient estimate.

4.4.2 Experimental setup

The Tensorflow library has been used to perform all the experiments. We use the MNIST digits dataset for our results. In all of the experiments below, we trained using a batch size of 32 and for 5 epochs over the entire MNIST dataset. To compute

Algorithm 3: NES gradient estimate

Input : Function $P(y|x), x \in \mathbb{R}^d$
Output : Numerical estimate of $\nabla_x P(y|x)$
$g \leftarrow 0_d$ **for** $j \leftarrow 1$ *to* n **do**
  $u_i \leftarrow \mathcal{N}(0_d, I_{d\times d})$
  $g \leftarrow g + P(y|x + \sigma \cdot u_i) \cdot u_i$
  $g \leftarrow g + P(y|x - \sigma \cdot u_i) \cdot u_i$
end
return $\frac{g}{2n\sigma}$

the NES gradients, we iterated for 10000 steps, i.e., $n = 10000$ and σ was set to 0.1 in Algorithm 3.

The baseline to compare our model is a simple feed-forward linear neural network. This model does not use any codes and rather uses the softmax function to generate probabilities for the 10 classes. Thus, the probability function looks like,

$$P(y|x) = (\text{softmax}\,(Wx + b))_y \tag{4.16}$$

The loss function used is the cross-entropy function along with a $\|W\|_F$ regularization term,

$$\text{cross-entropy loss} + \mu\|W\|_F \tag{4.17}$$

The accuracy of training the model with different values of μ have been listed in Table 4.3. We see that the accuracy decreases as μ increases, which is expected.

μ	Accuracy (%)
0.01	89.27
0.03	89.7
0.05	90.53
0.08	90.17
0.1	87.12
0.3	79.48
0.5	74.48
0.1	62.67

Table 4.3: Accuracy of baseline model with varying μ

For the codeword model, the loss function was the sum of sum of individual k cross-entropy losses. With identity codes to represent the 10 classes, the accuracy obtained was 81.55%. From recall from earlier discussions that having orthogonal parameters is helpful. By initializing the weights using random Gaussian sampling, the initial set of weights were mostly orthogonal. We noted that at the end of

training period, the weights were quite orthogonal. No regularization was used. For the case where random codes were used, Table 4.4 lists the accuracies obtained.

Code size	Accuracy (%)
5	60.18
6	53.32
7	63.93
8	77.09
9	73.81
10	73.06
11	75.89
12	76.04
13	67.71
14	73.56
15	67.2
16	79.45
17	77.26
18	79.18
19	81.94

Table 4.4: Accuracy of code model with random codes

It is interesting to note that for the random code model, $k = 10$ yields an accuracy of 73.06% which is lower than the identity code model. Repeating the experiments did not change their relative orders of accuracy.

4.4.3 l_∞ attack

To construct the l_∞ attack under this framework, we combine the probability measure defied in Equation 4.8 with the NES approach of computing gradients from Algorithm 3. As mentioned earlier, we set $n = 10000$ and $\sigma = 0.1$. Each call to Algorithm 3 returns an estimate of the gradient of the true class c of an input x. Applying the sign operation gives us the FGSM attack. This was projected onto the $[0, 1]^{784}$ hypercube. We did this 5 times to get the projected I-FGSM. In the tables below, ϵ denotes the magnitude of the perturbation, i.e., $\|x_{adv} - x\|_\infty = \epsilon$. Tables 4.5, 4.6 and 4.7 summarize the results of this attack for the baseline model, code model with identity code and code model with random codes respectively.

μ	ϵ	Accuracy (%)	Misclass rate (%)
0.0	0.1	90	90
0.01	0.1	92	92
0.03	0.1	89	89
0.05	0.1	83	83
0.08	0.1	88	82
0.1	0.1	89	32
0.3	0.1	82	23
0.5	0.1	73	26
0.1	0.1	60	15

Table 4.5: I-FGSM on the baseline model

ϵ	Misclass rate (%)
0.01	6
0.02	20
0.03	23
0.04	31
0.05	39
0.06	33
0.07	41
0.08	37
0.09	32
0.1	68

Table 4.6: I-FGSM on the code model with identity code. The accuracy of the model is 81.55%

4.4.4 l_2 Carlini-Wagner (CW) attack

We performed the CW attack using the formulation discussed in Section 2.2.1. The optimization is described by:

$$\text{minimize}_w \quad \|\frac{1}{2}(\tanh(w)+1) - x\|_2^2 + \tau \cdot f\left(\frac{1}{2}(\tanh(w)+1)\right) \tag{4.18}$$

Code size	Accuracy (%)	Misclass rate (%)
5	60.18	57
6	53.32	51
7	63.93	55
8	77.09	77
9	73.81	76
10	73.06	73
11	75.89	77
12	76.04	71
13	67.71	63
14	73.56	65
15	67.2	64
16	79.45	72
17	77.26	63
18	79.18	72
19	81.94	71

Table 4.7: I-FGSM on the code model with random code with $\epsilon = 0.1$

with f defined as

$$f(x) = \max\{C(x)_i \mid i \neq c\} - C(x)_c \tag{4.19}$$

Different values of the hyperparameter τ determines the relative strength given to the two competiting terms. We use a strategy which uses binary search to find the most suitable τ which definitely leads to a misclassification. With this approach, we were able to always find an adversarial example pertaining to the given input. Hence this attack turned out to be extremely powerful to the rate that any meaningful comparisons could not be made. Tables 4.8 and 4.9 summarize the results for these attacks.

4.4.5 Takeaways

The table in this section collect the results of all the experiments we performed. One will note that increasing the perturbation magnitude, ϵ, increases the misclassification rate, which is in-line with the expectations. Let's divert our attention to Table 4.5. At first glance we notice that increasing the regularization parameter μ decreases the misclass rate keeping ϵ constant. However, with an increase in μ, the accuracy of the model goes down. For a less accurate model, the misclass rate is less exciting to work with as the model itself may be predicting a lot of incorrect labels to being with.

μ	Accuracy (%)	Misclass rate (%)
0.0	89.05	100
0.01	89.4	100
0.03	89.86	100
0.05	90.32	100
0.08	89.95	100
0.1	86.93	100
0.3	79.46	100
0.5	74.49	100
1.0	62.67	100

Table 4.8: CW attack on the baseline model

Code size	Accuracy (%)	Misclass rate (%)
5	71.14	100
6	66.81	100
7	67.3	100
8	74.52	100
9	75.36	100
10	76.89	100
11	76.78	100
12	79.03	100
13	77.3	100
14	77.9	100
15	74.65	100
16	78.46	100
17	81.35	100
18	78.58	100
19	82.8	100
20	82.11	100

Table 4.9: CW attack on the random code model

From Table 4.7 we see that increasing the code word size leads to an improvement in the classification accuracy/ However, the trend in misclassification rate is quite random and doesn't show a clear trend. Repeating the experiment a few times as well failed to give a guiding principle for the misclass rates with the code size as well.

Chapter 5

Conclusion

In this project, we have explored various gradient-based, game-theoretic and pixel attack methods in our attempt to defeat the Robust Manifold Defense by Ilyas et al [6]. Out of all these attacks, our FSGM attack was more effective than the approach explored by Ilyas et al. - increasing misclassification rates by at least 8 to 10%. Furthermore, iterative-FGSM caused the classifier to misclassify on 36% of adversarial images when the maximum allowed perturbation was set to $\epsilon = 0.2$. While this result is better than what was reported in the original paper, we believe that a more thorough investigation of our approaches would lead to a stronger attack on the defense algorithm. We also have some unexplored ideas revolving around the use of GANs to learn to create adversarial examples in an automated way.

For the second part of the project involving an ensemble of classifiers, we saw a new approach of constructing an adversarial attack model. We defined a new probability model involving the Hamming distance and that came handy while performing the gradient based attacks. We did not see very strong signs of robustness which could justify the additional memory cost incurred by maintaining copies of a single classifier as originally hoped. However, the l_2 Carlini-Wagner attacks presents possible avenues of exploration with more carefully designed experiments to leverage more insights from the model.

Bibliography

[1] Anish Athalye, Nicholas Carlini, and David A. Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. *CoRR*, abs/1802.00420, 2018.

[2] 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.

[3] Thomas G Dietterich and Ghulum Bakiri. Solving multiclass learning problems via error-correcting output codes. *Journal of artificial intelligence research*, 2:263–286, 1994.

[4] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations*, 2015.

[5] Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin. Black-box adversarial attacks with limited queries and information. *arXiv preprint arXiv:1804.08598*, 2018.

[6] Andrew Ilyas, Ajil Jalal, Eirini Asteri, Constantinos Daskalakis, and Alexandros G. Dimakis. The robust manifold defense: Adversarial training using generative models. *CoRR*, abs/1712.09196, 2017.

[7] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *CoRR*, abs/1607.02533, 2016.

[8] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. One pixel attack for fooling deep neural networks. *CoRR*, abs/1710.08864, 2017.

[9] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *CoRR*, abs/1312.6199, 2013.

[10] Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber. Natural evolution strategies. *J. Mach. Learn. Res.*, 15(1):949–980, January 2014.