Crafting Label Noise to Increase Adversarial Vulnerability in Deep Learning Models

Acknowledgements

Without the support and sacrifices of my mother, I could never have dreamt of receiving an education. I would like to thank my supervisors, Professor Varun Kanade, and Professor Philip Torr for their guidance throughout this project, and also for access to the computational resources I required. I am grateful for the interesting discussions I got to have with them throughout the summer. I am grateful to Amartya Sanyal, for being a mentor for the project and my closest friend. My year at Oxford, in the midst of a pandemic would have been very empty without the warm group of friends at St Hugh's College. I am also thankful to my course supervisor, Professor Bill Roscoe for being a guiding light, and for priceless advice.

Abstract

Deep Neural Networks have achieved the state of the art in many tasks. However, recently there have been concerns regarding their robustness, and hence reliability, when being deployed in mission critical applications. It seems that Neural Networks can be easily fooled, by adding small perturbations to the input. There has been a recent flow of literature in the direction of purposefully making modifications to datasets in order to increase the vulnerability of Neural Networks trained on them. Such modifications to datasets with malicious intent, is called dataset poisoning.

Label flipping is one such dataset poisoning attack, where certain samples from a dataset are chosen, and mislabeled on purpose. Surprisingly, Neural Networks manage to generalise very well, even after memorising label noise. However, their adversarial vulnerability takes a huge fall. In this thesis, we will study the nature of decision boundaries learnt by networks as they memorise label noise. Our attempt is to then utilise this to craft label noise, i.e., select points to flip in a dataset, in order to maximise the adversarial vulnerability of networks trained on it. The number of points we can select to label-flip is naturally constrained by a budget. Further, a suprising experimental result we observed, is that placing mislabeled points in the highest probability density regions of the data distribution has minimal effect on the adversarial accuracy of a network that memorises it. We also extend some previous theoretical results regarding uniform label noise, with additional reasonable assumptions about networks, such as lipschitzness.

Contents

Chapter 1

Introduction

1.1 The Problem

Deep Neural Networks (DNNs) have gained a lot of popularity in the Machine Learning Community. The fact that they are universal approximators [10], coupled with good training algorithms, and an abundance of data and powerful hardware, are some factors that have contributed to their popularity. These have helped deep learning achieve the state of the art in many tasks, from computer vision [12], to machine translation [22, 24], to achieving superhuman performance in board and computer games [21, 25]. However, the reliability of neural networks is still in question. The work of Szegedy et al. [23] opened up a new line of research on the robustness of neural networks. Neural Networks seem to be sensitive to small perturbations in the input.

Recent works [3, 27] have pointed out that several over-parameterized machine learning models can memorize label noise (mislabelled points), which is surprisingly already present in widely used datasets [19] such as MNIST and CIFAR-10, without showing a noticeable drop in their test accuracies. This has been investigated with artificially induced label noise in standard datasets as well [19], and is referred to as *benign overfitting*.

However, memorizing noisy data does come with downsides. Sanyal et al. [19] investigated the consequences of memorizing *uniform* label noise on the adversarial error of the model. They showed that the adversarial error increases in the presence of such label noise, because the model becomes increasingly vulnerable in regions where a wrong label has been memorized. So, a model that is trained on data with label noise might perform very well on unseen test data, but it will be highly adversarially vulnerable, much more than what a model trained on clean data would be. This means we, as attackers, can add human imperceptible perturbations to the inputs of a network and get the network to misclassify them.

While Sanyal et al. [19] experimentally and theoretically investigated a setting where models memorize *uniform* label noise, there has not been any study towards carefully *crafting* label noise such that if it is memorised, robustness is hurt massively. More precisely, the question we seek to answer is: mislabeling which points in a dataset would cause a greater fall in the adversarial vulnerability? The answer to this question could be used to select points in a dataset which we could mislabel, and render a victim's model, trained on this "poisoned" data, to be highly adversarially vulnerable.

The bound proven in Theorem 1 of Sanyal et al. [19] makes no assumption regarding the classifier that memorises a dataset with uniform label noise. The worst case for high adversarial error is assumed, in order to arrive at the bound: the case where mislabeled points get memorised discontinuously. In this worst case, the classifier gets the highest possible test accuracy, getting the wrong answer only at mislabeled points, and nowhere else. The bound naturally follows from this. There exist l_p balls of radius ϵ (the attack radius) around mislabeled points within which all points are vulnerable. In practice, networks learned with SGD are much smoother, and end up learning regions around these mislabelled points where the network will produce the wrong label, rather than just at the mislabelled points.

The problem at hand is in the following direction. Given a dataset, we are to find out which points in the dataset can be label-flipped , so that a neural network trained on the dataset becomes highly adversarially vulnerable while not showing an appreciable drop in its test accuracy. The number of points we can flip is constrained by a budget. Victims who might deploy their model trained on such a poisoned dataset, will have vulnerabilities in their system. This study is also in the hope that we can uncover some properties of neural networks fitting mislabeled points, and observe the effects of memorizing label noise, so that we might be able to come up with defences to the same.

When mislabeled points are introduced into the dataset, deep neural networks trained on the dataset learn regions where they misclassify the input, and our aim is to also study the nature of these regions. The experiments in this thesis reveal whether these regions are seperate pockets, such that nearby points become vulnerable, or are extensive regions which render regions far away from the mislabeled training point vulnerable.

1.2 Contributions

Our contributions in this thesis are threefold:

1. Extending a previous theoretical result on benign overfitting, with added Lipschitzness assumptions on networks, in the context of uniform label noise.

2. Showing that placing mislabeled points at the highest probability density regions does not hurt adversarial vulnerability, indicating that the proven theoretical bound in literature is vacuous in practice.

3. Coming up with an experimental setup that helps identify points which are best to be mislabeled, for hurting the adversarial accuracy of a model trained on them. The experiments we did also indicate why randomly selecting points to label-flip in a popular dataset like MNIST is still very good. Points which massively hurt adversarial accuracy on being mislabeled turn out to be high in number. This also identifies a significant property of mislabeled points that causes such a massive drop in the adversarial accuracy: they draw and stretch the decision boundary, and this will be explained in detail in chapter 5. The selection of such points seems to be dependent on the dataset under study. While our final procedure is not a function into which we can simply plug in a dataset and receive a poisoned version, we do have an impactful toolset that guides the selection of training points to label-flip.

Chapter 2

Background

2.1 The supervised classification problem

Consider a distribution $\mathcal{P} : \mathbb{R}^d \to \mathbb{R}_{\geq 0}$ from which datapoints are sampled, and a function $c : \mathbb{R}^d \to \Omega$, called the *ground truth* function. Ω is a discrete set of *classes* that instances from $\mathbb{R}^d$ need to be mapped into. Given n points sampled i.i.d. from $\mathcal{P}$, in a *training set* $D = \{\mathbf{x_i}\}_{i=1}^{n}$, the supervised classification problem is to find a function with high probability, that classifies samples from $\mathcal{P}$ reliably as c would have. This notion of reliability is captured formally through the *natural risk*, which we will define shortly.

The *natural risk* $R_{\text{Nat}}(\mathcal{C})$ of a classifier $\mathcal{C} : \mathbb{R}^d \to \Omega$, is defined as

$$R_{\text{Nat}}(\mathcal{C}) = \mathbb{P}_{\mathbf{x}\sim\mathcal{P}}[c(\mathbf{x}) \neq \mathcal{C}(\mathbf{x})] = \int_{\mathbb{R}^d} \mathcal{P}(\mathbf{x}) \cdot [\![c(\mathbf{x}) \neq \mathcal{C}(\mathbf{x})]\!] \, \mathbf{dx} \tag{2.1}$$

In other words, the natural risk is the probability of the classifier getting a wrong answer. In practice, we carry out a Monte-Carlo estimation of the natural risk, by finding the fraction of points in an unseen set of points sampled i.i.d. from $\mathcal{P}$ (called the "test set") for which the classifier gets the incorrect output. This estimate is also called the "test error". Precisely, let this test set be denoted as $D_{\text{test}} = \{(\mathbf{x_i}, y_i = c(\mathbf{x_i}))\}_{i=1}^{m}$. Then,

$$\hat{R}_{\text{Nat}}(\mathcal{C}) = \frac{1}{|D_{\text{test}}|} \cdot \sum_{(\mathbf{x},y)\in D_{\text{test}}} [\![y \neq \mathcal{C}(\mathbf{x})]\!]$$

2.2 Neural networks

Deep Neural Networks are a family of over-parametrised models. They are proven *universal approximators* [10], which means with enough parameters, they can approximate any function with arbitrarily small error.

DNNs are composed of *layers*, with each layer parameterized by *weights* and *biases*. Each layer takes in a vector and produces another vector. This is done by first doing an affine transformation of the input vector, which is entirely specified by the weight matrix and bias of that layer. This transformation yields a vector which is then passed through an *activation function*. DNNs acquire their non-linearity through these activation functions. More precisely, let the ith layer of a network be denoted as $L_i : \mathbb{R}^{d_i} \to \mathbb{R}^{d_{i+1}}$. Let its weight matrix be $W_i \in \mathbb{R}^{d_i \times d_{i+1}}$, bias be $\mathbf{b}_i \in \mathbb{R}^{d_{i+1}}$ and activation function be σ. Then the layer L_i is defined as the following function:

$$L_i(\mathbf{x}) = \sigma(W_i\mathbf{x} + \mathbf{b}_i)$$

Then the entire DNN is simply a composition of all of its layers. So a DNN $\mathcal{C} : \mathbb{R}^{d_{\text{in}}} \to \mathbb{R}^{d_{\text{out}}}$ having k layers $L_1, ..., L_k$ is defined as:

$$\mathcal{C} = L_k \circ L_{k-1} ... \circ L_1$$

The series of computations we perform while finding the outputs of each layer for a given input is called a *forward pass*.

The ReLU activation function is the most popular one. It is defined as

$$\text{ReLU}(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases}$$

Two other popular activation functions are the sigmoid and tanh activations

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

and

$$\tanh(x) = 2 \cdot \sigma(x) - 1$$

The above definitions involve scalar inputs to these functions. We will overload vector inputs for the activation functions defined above, such that each dimension in the input vector is passed through the activation function considered.

For classification problems, we require the network to output a probability distribution over classes in Ω. This is achieved by using the *softmax* activation function, defined as:

$$\text{softmax} : \mathbb{R}^k \to \Delta^k \text{ ; } \text{softmax}(\mathbf{x}) = \mathbf{x}' \text{ s.t. } \mathbf{x}'_i = \frac{e^{\mathbf{x}_i}}{\sum_{j=1}^{d} e^{\mathbf{x}_j}}$$

where Δ^k refers to the k dimensional simplex.

The softmax function is essentially a soft, smooth version of the one-hot-argmax function, which returns a one-hot vector of same dimensionality, with a

1 at the index with the maximal value in a vector, and 0 elsewhere. The above activation produces a probability distribution over the dimensions. Hence, in networks used for classification problems, we generally augment the final layer, having $|\Omega|$ dimensional outputs, with the softmax activation function. Ω here is the set of classes the classifier has to map into.

The dimensionalities of each layer, activations, number of layers, etc. are design choices, and these constitute the *architecture* of the network. Any such design choice, be it the number of layers or the activation functions used, induces inductive biases. We assume that there exists a function that is a good enough approximation of the function we want, in the family of functions we search over, referred to as the *hypothesis class.* For instance, the hypothesis class could be the set of neural networks with different values for their parameters, but having the same architecture. Now, we will need to search over this family of functions with the same architecture for the "best" one, having certain values for its parameters, which is what the *training algorithm* does.

Convolutional Neural Networks (CNNs) are DNNs that perform very well on image data [12, 13], by capturing spacial correlations in images. All experiments on image datasets done in this thesis utilise CNNs.

2.2.1 Training algorithms

Gradient Descent is an optimisation technique that assumes a first order approximation of the objective function. Given a function $f(\mathbf{x})$, which needs to be *minimised* over $\mathbf{x}$, gradient descent attempts to find a minimum by iteratively updating the current best as:

$$\mathbf{x_{i+1}} = \mathbf{x_i} - \alpha \cdot \nabla_{\mathbf{x}} f(\mathbf{x})\big|_{\mathbf{x}=\mathbf{x_i}}$$

where α is a hyperparameter called the *learning rate.* Hence, gradient descent requires gradients of f computed w.r.t $\mathbf{x}$.

In machine learning, we might frequently require optimising over functions of the form $\frac{1}{m}\sum_{i=1}^{m} g_i(\mathbf{x})$. Often, each term of this sum is the contribution of each individual point in the dataset to the objective. This requires gradient descent to evaluate

$$\nabla_{\mathbf{x}} \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}_i(\mathbf{x}) = \frac{1}{m} \sum_{i=1}^{m} \nabla_{\mathbf{x}} \mathcal{L}_i(\mathbf{x})$$

This is usually computationally expensive to evaluate, since one gradient descent step will require computing gradients for each term, which frequently evaluates to computing average gradients over the entire training set. Stochastic Gradient Descent (SGD) involves sampling a small batch of samples from the training set, and executing a gradient descent step using the mean gradients for that batch, rather than the entire training set.

There are many other first-order gradient based optimizers, the most popular one being the Adam optimizer [11]. Adam has much higher convergence rates in practice, compared to vanilla SGD. In all of the experiments in this thesis, Adam is the optimizier used.

2.2.2 Loss functions

To train networks, we need to have a metric that captures how "good" a network is. We could then optimise over this metric, and tweak the parameters of the model hoping that the resultant model will have a low natural risk. Such metrics are called loss functions.

For classification problems, including the ones in this thesis, we will be using the *Cross Entropy Loss*, which is standard for classifiers. For a "target" distribution P, and a distribution Q both over a set Ω, the cross entropy loss produces the lowest value when $P = Q$, and has high values when Q is very different from P.

$$H[Q, P] = -\sum_{\omega \in \Omega} P(\omega) \log Q(\omega)$$

Given an input example $\mathbf{x}$, labeled with $y = c(\mathbf{x})$, to a classifier $\mathcal{C}$, which produces a distribution over classes $\mathcal{C}(\mathbf{x})$, the cross-entropy loss evaluates to $H[\mathcal{C}(\mathbf{x}), \mathbf{y}]$ where $\mathbf{y}_j = [\![j == y]\!]$.

2.2.3 Backpropagation

To train a neural network $\mathcal{C}$ with the algorithms mentioned in the section 2.2.1, we require to compute $\mathbb{E}_{\mathbf{x} \sim \mathcal{P}}[\nabla_\theta \mathcal{L}(\mathcal{C}(\mathbf{x}), c(\mathbf{x}))]$ where $\mathcal{P}$ is the data distribution. In practice, this is done by sampling a batch of b samples $X = [\mathbf{x_i} \mid i \in [1..b]]$ uniformly from the test set, and computing $\frac{1}{b} \sum_{i=1}^{i=b} \nabla_\theta \mathcal{L}(\mathcal{C}(\mathbf{x_i}), c(\mathbf{x_i}))$.

In order to do this computation in neural networks, we require to methodically apply the chain rule. The backpropagation algorithm does this. We first record all computations we do during the forward pass, and the following recursive relation is utilised to find the gradients of the loss with respect to one value in this forward pass. Let this value be v, and all values dependent on v be $u_1 ... u_k$. Let L denote the final value whose gradient we are to compute, w.r.t. v, assuming we already have $\frac{\partial L}{\partial u_1} ... \frac{\partial L}{\partial u_k}$ and we know how to compute $\frac{\partial v}{\partial u_i}$. Then,

$$\frac{\partial L}{\partial v} = \sum_{i=1}^{k} \frac{\partial v}{\partial u_i} \cdot \frac{\partial L}{\partial u_i}$$

So we start from $\frac{\partial L}{\partial L} = 1$, and work "backwards" from there to compute all gradients.

The relation above can be applied to any network such that for each operation, we can compute gradients of the operation's outputs w.r.t. its inputs, and allows for far more flexibility than than just working for the model architectures we discussed.

2.3 Adversarial risk

Szegedy et al. [23] discovered a significant property of deep neural networks: they are sensitive to changes in the input. Networks trained on data drawn from a distribution perform very well on unseen data from the same distribution, for example that of handwritten digits, but are extremely sensitive to small, almost human imperceptible input perturbations. This is generally referred to in literature as *Adversarial Vulnerability*. [8] and [15] showed some methods to craft such perturbations, and these are called *adversarial attacks*. These attacks will be discussed later.

This sensitivity to input perturbations is a significant vulnerability, since the changes in the input are mostly imperceptible to humans. Deep Learning models, with their improving real-world performance [12, 9], are begining to get deployed in mission critical systems [4], and such vulnerabilities raise questions about the reliability of neural networks for such applications.

The adversarial vulnerability of a classifier $\mathcal{C}$ is quantified through the *adversarial risk*. While there are a multitude of ways in literature in which adversarial vulnerability has been defined, we are concerned with adversarial vulnerability in the context of l_p perturbations. The adversarial risk $R^{\rho}_{\text{Adv}}(\mathcal{C})$ of a classifier $\mathcal{C} : \mathbb{R}^d \to \Omega$ in a classification problem where datapoints are drawn from the distribution $\mathcal{P} : \mathbb{R}^d \to \mathbb{R}_{\geq 0}$ where d is the data dimensionality, and ground truth labeling function $c : \mathbb{R}^d \to \Omega$, where Ω is the set of classes, is defined as

$$\begin{aligned} R^{\rho}_{\text{Adv}}(\mathcal{C}) &= \mathbb{P}_{\mathbf{x}\sim\mathcal{P}}[\exists \tilde{\mathbf{x}} \in \mathcal{B}^p_{\rho}(\mathbf{x}) \text{ s.t. } c(\mathbf{x}) \neq \mathcal{C}(\tilde{\mathbf{x}})] \\ &= \int_{\mathbb{R}^d} \mathcal{P}(\mathbf{x}) \ [\![\exists \tilde{\mathbf{x}} \in \mathcal{B}^p_{\rho}(\mathbf{x}) \text{ s.t. } c(\mathbf{x}) \neq \mathcal{C}(\tilde{\mathbf{x}})]\!] \ \mathbf{dx} \end{aligned} \tag{2.2}$$

where $\mathcal{B}^p_{\rho}(\mathbf{x})$ is the set of all points inside an l_p ball around $\mathbf{x}$, given by

$$\mathcal{B}^p_{\rho}(\mathbf{x}) = \{\mathbf{x}' \in \mathbb{R}^d \mid \rho \geq \|\mathbf{x}' - \mathbf{x}\|_p\}$$

for some norm p.

The adversarial risk intuitively captures the probability that we get an input such that it can be perturbed within a constrained region, so that the classifier gives the wrong output. In practice, this risk is estimated by utilizing some standard attack-strategy to calculate perturbations, such as those explained next, and finding the fraction of test-set points for which we could successfully find a perturbation that could make the classifier misclassify.

Given a classifier $\mathcal{C}$ and a point $\mathbf{x} \sim \mathcal{P}$, the problem of finding an adversarial example for $\mathbf{x}$, i.e. finding an $\tilde{\mathbf{x}} \in \mathcal{B}_{\rho}^{p}(\mathbf{x})$ s.t. $c(\mathbf{x}) \neq \mathcal{C}(\tilde{\mathbf{x}})$ can be written as a constrained optimization problem:

$$\tilde{\mathbf{x}} = \underset{\mathbf{u} \in \mathcal{B}_{\rho}^{p}(\mathbf{x})}{\arg\max}\ \mathcal{L}(\mathcal{C}(\mathbf{u}), c(\mathbf{x}))$$

where $\mathcal{L}$ is a loss function, and could be the same one used to train the model. So we maximize this loss inside the l_p ball around $\mathbf{x}$ to find an adversarial example. The solution to the above optimisation problem is an *untargeted* adversarial example. This means that the resultant adversarial example could be classified by the model as any wrong class, which is why we call algorithms that solve or approximate the above as *untargeted attacks*.

We could also have *targeted attacks*, which provide greater control to attackers. Targeted attacks involve creating adversarial examples such that the classifier classifies it to be of a particular *target class*. Let the target class be t. Then the constrained optimisation problem for a targeted attack around $\mathbf{x}$ is

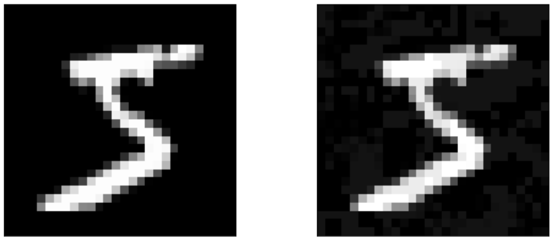
$$\tilde{\mathbf{x}}_{\mathbf{t}} = \underset{\mathbf{u} \in \mathcal{B}_{\rho}^{p}(\mathbf{x})}{\arg\min}\ \mathcal{L}(\mathcal{C}(\mathbf{u}), t)$$

The brief review on attack strategies that follows this discussion is based on different ways to go about computationally solving the above maximization problem.

2.4 Standard attacks for neural networks in literature

Here, we discuss 2 popular attack techniques in literature. When we use the term *attack*, the problem we attempt to solve is finding an adversarial example, i.e., given a point from the data distribution, we are to find out a point within an l_p ball (of some radius ρ) around that point, such that the model gives the wrong output for that point. For a *targeted attack*, given a point $\mathbf{x}$ with label $c(x)$ from the data distribution, we are to find a point $\mathbf{x}'$ inside an l_p ball around $\mathbf{x}$ such that the model classifies $\mathbf{x}'$ as a target class t.

Reiterating, our objective will be to solve a constrained optimisation problem, minimising or maximising the loss, depending on whether or not we are doing a targeted, or untargeted attack.



(a) Classified as a ”5” (b) Classified as a ”3”

Figure 2.1: A convolutional neural network trained on the MNIST dataset, attained 99.19% test accuracy. The classifier predicts the image of ”5” on the left correctly with 100% confidence. The same ”5” was perturbed, within a 0.04 l_∞ radius of the original ”5”. The same classifier classifies this ”5” as a ”3”, with more than 60% confidence. This classifier has a 0% adversarial accuracy for an 0.3 l_∞ radius, which means for each point in the test set, we can find a small enough perturbation to add to the original image, and get the classifier to misclassify. The reader might need to view the page from different angles to perceive the difference between the two images.

Let $\mathcal{L} : \Gamma(\Omega) \times \Omega \to \mathbb{R}$ be the loss function considered, where Ω is the set of classes and $\Gamma(\Omega)$ is the set of all probability distributions over Ω. For our discussion, it does not matter which loss is used, as long as it penalizes, or has high values when the model produces the wrong class prediction, and low value when it gives the right one. The cross entropy loss is the most popular choice. Given a data-point $\mathbf{x}$ and neural network classifier $\mathcal{C}_\theta$ parameterized by θ, these methods rely on computing the loss value by forward propagating through $\mathcal{C}_\theta$, and then backpropagating, computing gradients all the way to the inputs, finding the gradient of the loss with respect to the input.

Essentially, we compute the following:

$$\boldsymbol{\Delta}\mathbf{x} = \nabla_{\mathbf{p}} \mathcal{L}(\mathcal{C}_\theta(\mathbf{p}), y)\Big|_{\mathbf{p}=\mathbf{x}}$$

we use the above computed value to carry out gradient ascent for untargeted attacks, and gradient descent for targeted attacks, setting $y = t$ where t is the target class. Since we require access to the model and all the computations done by it to compute the gradient above, these attacks are also referred to as *white box gradient based attacks.* There are many *black box* attacks as well, where access to the model is not assumed. These black box attacks are however, out of the scope of our discussion.

2.4.1 Fast Gradient Sign Method

The Fast Gradient Sign Method (FGSM) [8], is a one step procedure, and requires only one forward-backward pass through the network. The constraint assumed for the attack is ϵ radius in l_∞ norm, which allows the attacker to produce adversarial examples for $\mathbf{x}$ within an ϵ radius l_∞ ball around $\mathbf{x}$. The adversarial example produced by FGSM is given by:

$$\mathbf{x}' = \mathbf{x} + \epsilon \cdot \text{sign}(\nabla_{\mathbf{p}} \mathcal{L}(\mathcal{C}_\theta(\mathbf{p}), y)\big|_{\mathbf{p}=\mathbf{x}})$$

for an untargeted attack, and

$$\mathbf{x}' = \mathbf{x} - \epsilon \cdot \text{sign}(\nabla_{\mathbf{p}} \mathcal{L}(\mathcal{C}_\theta(\mathbf{p}), t)\big|_{\mathbf{p}=\mathbf{x}})$$

for a targeted attack. The sign function is:

$$\text{sign}(x) = \begin{cases} 0 & x = 0 \\ \frac{x}{|x|} & x \neq 0 \end{cases}$$

and the above function is overloaded to vectors as an elementwise operation. The sign of gradients, scaled by ϵ are used for updates to ensure that the steps taken are small.

2.4.2 Projected Gradient Descent

The Projected Gradient Descent (PGD) [15] attack is the most popular adversarial attack in literature. Throughout this thesis, wherever we refer to attacks in experiments, we refer to PGD attacks. In literature, and in this thesis as well, whenever we have to evaluate the adversarial accuracy of a network, we iterate through the test set, and find out the fraction of the test set for which the PGD attack was unsuccessful.

The PGD attack involves a few hyperparameters:

1. Attack-steps m: This is the number of iterations the PGD algorithm is going to take.

2. Step-size μ: This is the size of each update the algorithm will take. The impact of this hyperparameter will be shown in the update equation below.

3. Attack-norm p: Perturbations are constrained to be within a region around the input $\mathbf{x}$. This distance constraint is in the form of a norm, which we will call the *attack-norm*. For instance, perturbations could be constrained within an l_∞ ball ($p = \infty$) of the input.

4. Attack-radius ϵ: This is the radius of the l_p ball within which the perturbed image should lie around the original image.

We repreatedly perturb the current perturbed image using FGSM, and project it back onto the l_p ball around the original input. Using the PGD algorithm with attack norm p is generally also referred to as ”using an l_∞ PGD-adversary. The PGD algorithm is as shown in algorithm 13.

Algorithm 1 Projected Gradient Descent

```
procedure Targeted-PGD($\mathcal{C}_\theta, \mathbf{x}, t, \epsilon, p, m, \mu$)
    $\mathbf{x_0} \leftarrow \mathbf{x}$
    for $i \leftarrow [1..m]$ do
        $\mathbf{x_i} = $ Targeted-FGSM($\mathcal{C}_\theta, \mathbf{x_{i-1}}, t, \mu$)
        $\mathbf{x_i} \leftarrow$ Project($\mathbf{x_i}, \mathcal{B}^p_\rho(\mathbf{x})$)    ▷ Project onto $l_p$ ball around $\mathbf{x}$
    return $\mathbf{x_m}$

procedure Untargeted-PGD($\mathcal{C}_\theta, \mathbf{x}, y, \epsilon, p, m, \mu$)
    $\mathbf{x_0} \leftarrow \mathbf{x}$
    for $i \leftarrow [1..m]$ do
        $\mathbf{x_i} = $ Untargeted-FGSM($\mathcal{C}_\theta, \mathbf{x_{i-1}}, y, \mu$)
        $\mathbf{x_i} \leftarrow$ Project($\mathbf{x_i}, \mathcal{B}^p_\rho(\mathbf{x})$)    ▷ Project onto $l_p$ ball around $\mathbf{x}$
    return $\mathbf{x_m}$

procedure Targeted-FGSM($\mathcal{C}_\theta, \mathbf{x}, t, \epsilon$)
    $d \leftarrow \mathcal{C}_\theta(\mathbf{x})$    ▷ Forward pass. $d \in \Gamma(\Omega)$ is the network's output
    $\ell \leftarrow \mathcal{L}(d, t)$    ▷ Operations are recorded to support backward-pass
    $\mathbf{\Delta x} = \nabla_\mathbf{p} \mathcal{L}(\mathcal{C}_\theta(\mathbf{p}), t)\big|_{\mathbf{p}=\mathbf{x}}$    ▷ Backpropagate
    $\mathbf{x}' = \mathbf{x} - \epsilon \cdot \mathrm{sign}(\mathbf{\Delta x})$
    return $\mathbf{x}'$

procedure Untargeted-FGSM($\mathcal{C}_\theta, \mathbf{x}, y, \epsilon$)
    $d \leftarrow \mathcal{C}_\theta(\mathbf{x})$    ▷ Forward pass. $d \in \Gamma(\Omega)$ is the network's output
    $\ell \leftarrow \mathcal{L}(d, t)$    ▷ Operations are recorded to support backward-pass
    $\mathbf{\Delta x} = \nabla_\mathbf{p} \mathcal{L}(\mathcal{C}_\theta(\mathbf{p}), y)\big|_{\mathbf{p}=\mathbf{x}}$    ▷ Backpropagate
    $\mathbf{x}' = \mathbf{x} + \epsilon \cdot \mathrm{sign}(\mathbf{\Delta x})$
    return $\mathbf{x}'$
```

2.5 Benign Overfitting and label noise

As showed by Zhang et al. [27], networks show respectable generalisation error even after memorising training data that has high levels of label noise. This phenomenon of DNNs is referred to in literature as *benign-overfitting.* Bartlett et al. [2] study this phenomenon in the context of linear regression problems.

Sanyal et al. [19] showed that while benign overfitting does not significantly harm the natural accuracy of classifiers, it massively harms the adversarial accuracies of the same models, studying the problem mainly in the context of uniform label noise. All experiments in this thesis also show the same: classifiers trained on noisy data have high natural accuracy but low adversarial accuracies.

A significant theoretical result shown by Sanyal et al. [19] is the following:

Theorem 1. *Let $\mathcal{C}$ be a classifier, and $\mathcal{D} : \mathbb{R}^d \to \mathbb{R}_{\geq 0}$ the data distribution, and $c : \mathbb{R}^d \to \Omega$ be the ground truth labeling function, where Ω is the set of classes. If there exist constants $c_1 \geq c_2 > 0, \rho > 0$ and a set $\zeta \subset \mathbb{R}^d$, such that*

1. $\mathbb{P}_{\mathbf{x}\sim\mathcal{D}}\left[\bigcup_{\mathbf{s}\in\zeta} \mathcal{B}_\rho^p(\mathbf{s})\right] \geq c_1$
2. $\forall \mathbf{s} \in \zeta, \quad \mathbb{P}_{\mathbf{x}\sim\mathcal{D}}[\mathcal{B}_\rho^p(\mathbf{s})] \geq \frac{c_2}{|\zeta|}$
3. $\forall \mathbf{s} \in \zeta, \ \forall \mathbf{u}, \mathbf{v} \in \mathcal{B}_\rho^p(\mathbf{s}), \ c(\mathbf{u}) = c(\mathbf{v})$

where $\mathcal{B}_\rho^p(\mathbf{x})$ is a ρ-radius l_p ball around $\mathbf{x}$. Let S be a training set, obtained by taking n i.i.d. samples from $\mathcal{P}$, and labeling each with $c(.)$ with η probability, or some incorrect class with probability $1 - \eta$. Let $\mathcal{C}$ be a classifier such that it has a 100% training accuracy on S. This means it gets the correct class on unflipped points, and has also memorized the wrong label on flipped points. Then, if $n \geq \frac{|\zeta|}{\eta c_2} \log(|\zeta|/\delta)$, with at least $1 - \delta$ probability, $\mathcal{R}_{Adv}^{2\rho}(\mathcal{C}) \geq c_1$.

Theorem 1 establishes that if the data distribution has regions of high probability density, then a classifier trained on data sampled from that distribution, injected with uniform label noise will have high adversarial error, with high probability. Secondly, the result does not require any assumption on the classifier. $\mathcal{C}$ could be a 1-nearest-neighbour classifier or a massive neural network, as long as it memorises the entire training set. Further, the theorem intuitively says that when a point is mislabeled and also memorised by a classifier, it will render a small region around it adversarially vulnerable, considering the worst case when the network memorises the mislabeled point discontinuously. This means that the network will have 100% test accuracy, and will get the wrong answers only exactly at the milabeled points. The adversarial risk in the bound comes from the nearby probability mass, around these mislabeled regions.

This result inspired the first few algorithms we tried out, which were based on the idea that placing mislabeled points at the highest probability density regions might be optimal to harm the adversarial accuracy of models trained on the data. This is for the case where there is no uniform label noise, rather we as attackers, are allowed to change the labels of a few points in the dataset.

2.6 Dataset poisoning

Dataset poisoning [20, 5] is a general term referring to the manipulation of a dataset by an agent, in order to cause vulnerabilities in any system that relies on this data. Modifications could be adding new malicious samples, changing existing samples, or perhaps changing labels.

Backdoor attacks [5, 18, 28] are popular dataset poisoning attacks which involve crafting and placing data-points such that if a neural network classifier ends up memorising this point, it will have certain regions that are adversarially vulnerable. The name "backdoor" comes from the fact that samples that would seem to be of class A to us humans, will be memorised by the classifier to be of class B, creating a region (a "backdoor") where we as attackers, can later query inputs of one class, while the classifier predicts those inputs to be of another class.

Label flipping attacks [26, 17] are poisoning attacks where we as attackers change the labels of points in the dataset. For example, we might flip malicious emails and label them as spam in an email dataset, and any spam detection network that learns from this dataset will have poor accuracy (natural or adversarial, depending on the attack), at least for a subset of emails, which we can later use to our advantage when the model gets deployed.

Dataset poisoning attacks constitute a relevant problem. This is because most practical machine learning systems utilise data that is scraped, or curated by another party. For the former case, we can craft malicious data-points, and upload them online, waiting for someone to scrap it off and train their networks on it [28]. For the latter case, the opportunity to inject malicious data-points into the dataset is more straightforward. Unfortunately, we might not always be able to refrain from using data from untrusted sources. Studying these attacks has the end-goal that we could eventually device training schemes that produce classifiers that are robust to these poisoning attacks.

The objective of this thesis is to study how label-flipping attacks can be crafted to maximise the adversarial error of networks trained on poisoned data. From this point, points in the dataset that we mislabel, will be refered to as *poisons*, or *flipped* points.

Chapter 3

Theoretical Results

Sanyal et al. [19] showed a theoretical result regarding *uniform* label noise, stated above in 2.5. The proof assumes the best case scenario where the network trained on poisoned data memorizes mislabeled points discontinuously, as dictionaries. This will have no effect on the test accuracy of the network, and points only in an l_p ball around the mislabelled points will be vulnerable. Practically, networks are smooth, and this smoothness can be formalized using Lipschitzness assumptions.

A function $f : X \to Y$ is locally lipschitz-continuous if there exists a constant κ s.t. $\forall \mathbf{x} \in X$, there exists a neighbourhood U around $\mathbf{x}$, s.t. $\forall \mathbf{x_1}, \mathbf{x_2} \in U$,

$$\frac{\|\mathrm{f}(\mathbf{x_1}) - \mathrm{f}(\mathbf{x_2})\|_q}{\|\mathbf{x_1} - \mathbf{x_2}\|_p} \leq \kappa$$

for some distance metrics $\|.\|_p$ and $\|.\|_q$.

Deep Neural Networks are lipschitz, and including this assumption in Theorem-1 of [19], gives the same bounds for a weaker adversary, i.e., we require a smaller attack-radius to get the same lower bound on the adversarial risk obtained by [19]. Below is the extended theorem, with the same assumptions (restated) from [19], with the additional assumption of lipschitzness.

3.1 Improved bound with Lipschitzness Assumptions

Theorem 2. *Let $\mathcal{C}$ be a classifier, and $\mathcal{D} : \mathbb{R}^d \to \mathbb{R}_{\geq 0}$ the data distribution, and $c : \mathbb{R}^d \to \Omega$ be the ground truth labeling function, where Ω is the set of classes. If there exist constants $c_1 \geq c_2 > 0, \rho > 0$ and a set $\zeta \subset \mathbb{R}^d$, such that*

1. $\mathbb{P}_{\mathbf{x} \sim \mathcal{D}}\left[\bigcup_{\mathbf{s} \in \zeta} \mathcal{B}_\rho^p(\mathbf{s}) \right] \geq c_1$

2. $\forall \mathbf{s} \in \zeta, \quad \mathbb{P}_{\mathbf{x}\sim\mathcal{D}}[\mathcal{B}^p_\rho(\mathbf{s})] \geq \frac{c_2}{|\zeta|}$

3. $\forall \mathbf{s} \in \zeta, \ \forall \mathbf{u}, \mathbf{v} \in \mathcal{B}^p_\rho(\mathbf{s}), \ c(\mathbf{u}) = c(\mathbf{v})$

where $\mathcal{B}^p_\rho(\mathbf{x})$ is a ρ-radius l_p ball around $\mathbf{x}$. Let S be a training set, obtained by taking n i.i.d. samples from $\mathcal{P}$, and labeling each with $c(.)$ with η probability, or some incorrect class with probability $1-\eta$. If a classifier $\mathcal{C}$ is such that

1. *$\mathcal{C}$ has a 100% training accuracy on S. This means it gets the correct class on unflipped points, and has also memorized the wrong label on flipped points. Further, on each training point, the classifier $\mathcal{C}$ is confident on the result at least χ more than the second largest output probability.*

2. *$\mathcal{C}$ is locally lipschitz, so for each point $\mathbf{x} \in \mathbb{R}^d$ and $\mathbf{x}'$ in its neighbourhood,*

$$\frac{[\![\mathcal{C}(\mathbf{x}) \neq \mathcal{C}(\mathbf{x}')]\!]}{\|\mathbf{x} - \mathbf{x}'\|_p} \leq \kappa$$

then if $n \geq \frac{|\zeta|}{\eta c_2} \log(|\zeta|/\delta)$, with at least $1-\delta$ probability, $\mathcal{R}^{2\rho-\tau}_{Adv}(\mathcal{C}) \geq c_1$ where $\tau = \frac{1}{\kappa}$.

Proof. Consider some $\mathbf{x} \in \zeta, \ \text{s.t.} \ \exists(\mathbf{x}', y) \in S, \ \mathbf{x}' \in \mathcal{B}^p_\rho(\mathbf{x}), \ y \neq c(\mathbf{x}')$, i.e., there exists a training point in S inside the l_p ball around $\mathbf{x}$ which has been mislabeled. The classifier would have memorized this point. We need to find out the attack radius ρ required to render all of the points inside $\mathcal{B}^p_\rho(\mathbf{x})$ vulnerable. The worst case is when this point $\mathbf{x}'$ lies on the surface of $\mathcal{B}^p_\rho(\mathbf{x})$ since this is the case where the attack radius required is going to be the largest. Now, any mislabeled point that is memorized by the classifier is going to create a region of radius at least τ where the classifier produces the same wrong label y. This means in the worst case, when $\mathbf{x}'$ lies on the surface, we will require an attack radius of $2\rho - \tau$ for all points in $\mathcal{B}^p_\rho(\mathbf{x})$ to be vulnerable.

If there is one mislabeled point placed inside each l_p ball around every point in ζ, then $\mathcal{R}^{2\rho-\tau}_{\text{Adv}} \geq c_1$, since all of the mass inside every l_p ball considered contributes to the adversarial risk.

The rest of the proof proceeds same as [19]. We will now lower bound the probability that there is one mislabeled point placed inside each l_p ball around every point in ζ.

$$\mathbb{P}\Big[\bigwedge_{\mathbf{s}\in\zeta}\exists \mathbf{s}' \in \mathcal{B}^p_\rho(\mathbf{s}), (\mathbf{s}', y) \in S \text{ for some } y \text{ s.t. } y \neq c(\mathbf{s}')\Big]$$
$$= 1 - \mathbb{P}\Big[\bigvee_{\mathbf{s}\in\zeta}\nexists \mathbf{s}' \in \mathcal{B}^p_\rho(\mathbf{s}), (\mathbf{s}', y) \in S \text{ for some } y \text{ s.t. } y \neq c(\mathbf{s}')\Big]$$

Using the union bound,

$$\geq 1 - |\zeta|\mathbb{P}\Big[\text{for a particular } \mathbf{s} \in \zeta,\ \nexists \mathbf{s}' \in \mathcal{B}^p_\rho(\mathbf{s}), (\mathbf{s}', y) \in S \text{ for some } y \text{ s.t. } y \neq c(\mathbf{s}')\Big]$$
$$= 1 - |\zeta|\Big(1 - \eta\frac{c_2}{|\zeta|}\Big)^n \ \geq\ 1 - |\zeta|e^{-n.\eta\frac{c_2}{|\zeta|}} \ \geq\ 1 - \delta$$

$n \geq \frac{|\zeta|}{\eta c_2}\log\big(\frac{|\zeta|}{\delta}\big)$ samples suffice for this. This completes the proof. □

The above theorem shows that if there are regions of very high probability mass in the data distribution, and there is uniform label noise at some rate, then a lipschitz classifier that fits 100% of the training data with some threshold confidence will have high adversarial error with high probability.

3.2 Lipschitzness bounded by distance to the nearest point, of different label

Theorem 3. *Let $\mathcal{C}$ be a classifier, and $\mathcal{D} : \mathbb{R}^d \to \mathbb{R}_{\geq 0}$ the data distribution, and $c : \mathbb{R}^d \to \Omega$ be the ground truth labeling function, where Ω is the set of classes. If there exist constants $c_1, c_2, c_{in}, c_{total}$ and $R \geq r$ and a set $\zeta \subset \mathbb{R}^d$, such that:*

1. $c_{ceil} \geq \mathbb{P}_{\mathbf{x}\sim\mathcal{D}}\Big[\bigcup_{\mathbf{s}\in\zeta}\mathcal{B}^p_R(\mathbf{s})\Big] \geq c_{total}$
2. $\forall \mathbf{s} \in \zeta,\quad \mathbb{P}_{\mathbf{x}\sim\mathcal{D}}[\mathcal{B}^p_r(\mathbf{s})] \geq \frac{c_{in}}{|\zeta|}$
3. $\forall \mathbf{s} \in \zeta,\ \forall \mathbf{u}, \mathbf{v} \in \mathcal{B}^p_R(\mathbf{s}),\ c(\mathbf{u}) = c(\mathbf{v})$

where $\mathcal{B}^p_\rho(\mathbf{x})$ refers to a ρ-radius l_p ball around $\mathbf{x}$. Let S be a training set, obtained by taking n i.i.d. samples from $\mathcal{P}$, and labeling each with $c(.)$ with η probability, or some incorrect class with probability $1 - \eta$. If a classifier $\mathcal{C}$ is such that

1. *$\mathcal{C}$ has a 100% training accuracy on S. This means it gets the correct class on unflipped points, and has also memorized the wrong label on flipped points. Further, on each training point, the classifier $\mathcal{C}$ is confident on the result at least χ more than the second largest output probability.*

2. *$\mathcal{C}$ is locally lipschitz. We assume that this lipschitzness constant is a decreasing function κ of the distance to the nearest point of the opposite label. So for each point $\mathbf{x} \in \mathbb{R}^d$ and $\mathbf{x}'$ in its neighbourhood, with the nearest point to $\mathbf{x}$ with a different label being at a distance greater than L in l_p norm,*

$$\frac{[\![\mathcal{C}(\mathbf{x}) \neq \mathcal{C}(\mathbf{x}')]\!]}{\|\mathbf{x} - \mathbf{x}'\|_p} \leq \kappa(L)$$

then,

$$\mathbb{P}\left[\mathcal{R}_{Adv}^{R+r-\frac{1}{\kappa(R-r)}}(\mathcal{C}) \geq c_{total}\right] \;\geq\; (1 - c_{ceil})^{n-|\zeta|}\left(\frac{\eta n c_{in}}{e|\zeta|}\right)^{|\zeta|}\sqrt{2\pi|\zeta|}$$

The lower bound on the probability might seem highly vacuous at first, but it might not be so. With increasing sample complexity, the training set will be having a large number of correctly labeled points around an incorrectly labeled point, when the lipschitzness assumption also breaks down if the classifier has to memorise the mislabeled point. The above theorem, hence should be considered in the light of low sample complexity. If there are only a few samples taken from $\mathcal{D}$, then while the highest probability density regions will receive points with high probability, there will be "intermediate" probability density regions that can receive a mislabeled point, with a correctly labeled point very far off from them. This means the network will learn larger regions with the wrong label, around such mislabeled points. The above theorem formalises this phenomenon.

Proof. If the sampling (and adding label noise) process results in one mislabeled point inside each l_p ball of radius r around each point in ζ, and no other point inside each l_p ball of radius R around each point in ζ, then,

$$\mathcal{R}_{\text{Adv}}^{R+r-\frac{1}{\kappa(R-r)}}(\mathcal{C}) \geq c_{\text{total}}$$

We need to lower bound the probability that this event, denoted as $\mathcal{E}$, happens. Since we sample n times from $\mathcal{D}$, the probability that exactly $|\zeta|$ samples result inside each of the l_p balls of radius r, and the rest lie outside the l_p balls of radius R, is

$$\mathbb{P}(\mathcal{E}) \geq \; (1 - c_{\text{ceil}})^{n-|\zeta|}(\eta c_{\text{in}})^{|\zeta|} \cdot \binom{n}{|\zeta|} \cdot |\zeta|!$$

Using the inequalities $\binom{n}{k} \geq (n/k)^k$ and$k! \geq \sqrt{2\pi k}(k/e)^k$,

$$\begin{aligned}\mathbb{P}(\mathcal{E}) \geq & \,(1 - c_{\text{ceil}})^{n-|\zeta|}(\eta c_{\text{in}})^{|\zeta|} \cdot (n/|\zeta|)^{|\zeta|} \cdot \sqrt{2\pi|\zeta|} \cdot (|\zeta|/e)^{|\zeta|} \\ = & (1 - c_{\text{ceil}})^{n-|\zeta|}\left(\frac{\eta n c_{\text{in}}}{e|\zeta|}\right)^{|\zeta|}\sqrt{2\pi|\zeta|}\end{aligned}$$

which is the bound we need. □

Chapter 4

Early Unsuccessful Experiments

The first subsection of this Chapter covers many of the initial experiments we did, and failed at. The rest of the subsections detail the other experiments that shed light on what happens when networks memorize label noise, and eventually with a few final experiments showing the nature of the worst points to mislabel.

4.1 Building graphs on training points

Sanyal et al. [19] theoretically gives bounds for the adversarial risk by reasoning that if mislabeled points are placed at high density regions - regions where if the classifier memorizes the wrong label, will render a large amount of probability mass in the vicinity of this mislabeled point vulnerable. This was the inspiration for the algorithm we studied initially: flipping the labels of training points which come from high probability density areas. The ideas in this section involve flipping the labels of training points in regions which can render as much probability mass in their neighbourhood vulnerable as possible. For datasets like MNIST, there is no reliable way to model the probability density function given training points, which is why we attempt to resort to heuristics, based on various combinations of representations of points with different ways of selecting points. This will be explained in detail in the following paragraphs. The algorithms discussed in this section require a few definitions.

Definition 1. *Given a set of points, $S = \{\mathbf{x_i}\}_{i=1}^m$, the Class-Graph of this set of points $G_\gamma^p(S)$ is defined as the undirected graph $G_\gamma^p(S) = (\{1..m\}, E_\gamma^p(S))$ where*

$$E_\gamma^p(S) = \{\{i,j\} \ \mid 0 < i < j \leq m \ \wedge \ \|\mathbf{x_i} - \mathbf{x_j}\|_p \leq \gamma\}$$

Intuitively, a class-graph of a set of points is an undirected graph connecting every two points closer than a threshold γ, under some norm p.

The algorithm is based on the following idea: For each class of the dataset, we build class-graphs in some representation space. The degrees of various points are now considered to be a heuristic for the probability density at each point. We then use one of the two selection strategies outlined below to select vertices from class graphs, which will then be label-flipped. When we refer to label-flipping, we simply will change each label selected to be of the next class, cyclically. The adversarial accuracy of the resultant model will then be evaluated in a targeted manner, by carrying out targeted PGD attacks from each class to the next one.

Selection Algorithms

The following two algorithms are the selection algorithms we experimented with. Each of them take in an undirected graph, and select a given number of vertices from it. Let Δ be the maximum degree in the given graph.

1. Algorithm 2: We simply select b nodes with the highest degrees

2. Algorithm 3: A $(\Delta + 2)$-approximation to the dominating set problem (Δ is the largest degree in the input graph), for a budget number of vertices in the dominating set. [1] proves the approximation factor for the full dominating set problem, and the proof for this budget-version follows form it.

Algorithm 2 Greedily Selecting vertices

```
procedure GREEDY-SELECTION($G, b$)
    Degrees ← [degree(i) for i in $V(G)$]
    Selected-Vertices ← Sort-By-Val($\{1..|V(G)|\}$, Val=Degrees)[-b:]
    return Selected-Vertices
```

It is a valid question as to why we ran 2 different algorithms: one where only high degree nodes are selected (greedy strategy), and one which is an approximate solution to the modified dominating set problem discussed before. When we select only select high degree points, we might be rendering points from the same neighbourhood vulnerable, which the second algorithm avoids.

4.1.1 Building graphs on input space

The first attempt was to build *Class Graphs* on each class of the MNIST dataset, and for each class, pick vertices from the graph to flip labels. For now, we do not concern ourselves with the label we flip to. Each label that has to be flipped is simply set to the next value cyclically.

The procedure we followed was as follows:

Algorithm 3 ($\log \Delta + 2$)-approximation to the dominating set problem

```
procedure Smart-Greedy-Selection($(G, b)$)
    $n \leftarrow |V(G)|$
    $S \leftarrow \phi, D \leftarrow \phi, U \leftarrow \{1..n\}$        ▷ $\phi$ refers to the empty set
    while $b > 0 \wedge U \neq \phi$ do
        choose $v \in \arg\max_{x \in D \cup U} |U \cap (\{x\} \cup N_G(x))|$ ▷ $N_G(x)$ is the set of points in $G$ neighbouring to $x$
        $S \leftarrow S \cup \{v\}$
        $U \leftarrow U \setminus (\{v\} \cup N(v))$
        $D \leftarrow D \cup (U \cap (\{v\} \cup N_G(v)))$
        $b \leftarrow b - 1$
    return S
```

1. Produce *Class Graphs* for all classes on the MNIST dataset

2. Select b points to flip using one of the 2 selection algorithms

3. Change the labels of the selected points

4. Train networks on the *poisoned* datasets

5. Randomly select b points to flip, and train and evaluate the networks on them to have a random baseline to compare with

As previously mentioned, the idea behind the above algorithms on the *Class Graphs* is that we want to select points that can render as much amount of probability mass *adversarially vulnerable* as possible. Since we do not have a reliable method of density estimation, we select points that have a large number of neighbouring points, as a heuristic.

Surprisingly, the results showed that not only did the above strategies yield networks that had adversarial errors comparable to the random strategy (i.e., they are no better), rather, they were strictly *inferior*. Networks trained on poisoned datasets modified using the above strategies are more robust than the ones trained on the same dataset but with random label noise. The next few subsections illustrate attempts to do the same in different subspaces which are more feature rich, since it might very well be the case that looking at the proximity of points in the input representation is sub-optimal for this. One strong reason why this might be the case is the inter-point distances of the points in practical datasets we experimented with: MNIST. The distances between points is far, far more than the perturbation budget ρ. Figure 4.1 illustrates this.

Figure 4.1: Figure showing the distribution of the inter-point distances in class 0 of the MNIST dataset

4.1.2 Using Low Rank representations

Principal Component Analysis (PCA) is a popular dimensionality reduction method. It aims to find $d_{\text{low-rank}}$ orthogonal directions, given d dimensional data points, such that the dataset has the highest variance along these $d_{\text{low-rank}}$ directions. The dimensionality reduction is then done by first centering the data to have 0 mean, and then projecting the data points onto each of the $d_{\text{low-rank}}$ vectors, getting $d_{\text{low-rank}}$ dimensional representations for each of the points. These $d_{\text{low-rank}}$ directions are obtained by finding out the eigenvectors corresponding to the $d_{\text{low-rank}}$ highest eigenvalues of the covariance matrix of the data distribution. In practice, the mean and the covariance matrix are *estimated* from the given data points. These low-dimensional representations yielded by PCA are generally referred to as *low-rank* representations as well.

We calculated the low-rank representations of points in the MNIST training set, and built Class-Graphs on these low-rank representations. This experiment did not yield any conclusions. In certain cases, it performed better and in certain cases worse, than the random baseline, which is not conclusive of anything.

4.1.3 Using the feature representations of a trained network

The next attempt was to consider the representations learnt by a trained network on the dataset. Networks are able to extract rich features, that heavily correlate with the label. It might be the case that selecting points that are closer in *feature space* might perform well. This attempt too, did not yield anything.

This was rather a bit short sighted, since we later realised, and experimentally verified as well, that networks memorize the false labels by learning very different feature representations, compared to other points in the vicinity, which have a different (correct) label.

4.1.4 Using the feature representations of a trained network as well as input space

The final experiment in this direction was to select points that are high-degree both in the input as well as feature space. We select $2b$ number of points from input as well as feature space. The greedy algorithm was employed for this, since the other one almost selected the same set of points, for the small budgets we had. We then select b points from the intersection of the set of points selected by both the algorithms.

The above also did not yield definitive results.

4.2 Min Coloring trick

If we could fit one model with each training set point flipped, then we could iterate over all such models and check which mislabelled points cause the largest drop in adversarial accuracy. In this experiment, we build graphs over the training set points of MNIST which belong to class 5, having one edge between points closer than γ in l_∞ distance. We then solve an extension of the minimum coloring problem on this graph, where even two adjacent nodes cannot be assigned the same color, and so cant second neighbours. We only need an approximate solution, which ensures that neighbours or second neighbours are not assigned the same color. The idea is to fit a network $\mathcal{N}$ mapping these points to the colors. After $\mathcal{N}$ memorizes the colors, we iterate over every point in class 5, attempting to assign a score to each point. The score for a point x is determined by the number of its neighbours which were rendered vulnerable to x, which is estimated by the number of x's neighbours such that we can carry out a *targeted* attack from them, and flip the network's output to the color assigned to x.

The idea behind the above experiment is that the network might be memorizing similar regions of the wrong label for mislabelled points in the poisoned dataset setup as different coloured regions when we assign colors to the training points. Further, the reason why we ensure that neighbours and second neighbours are not assigned the same color is that when we start out with a training point x, with an assigned color $c(x)$, we need to carry out a targeted attack from this point to the color of one of its neighbours, hence if 2 neighbours share the same color, we cannot be sure which region (pertaining to which of these neighbours) the attack settled at.

Next, we flip the labels of the training points in class 5 with the largest scores assigned by the above algorithm. The resulting network's adversarial accuracy is estimated on class 5 only, and that too in a targeted fashion, attempting attacks that flip the network's output from 5 to 6, precisely.

The above experiment was compared to the random strategy, where randomly chosen points from class 5 (same fraction of points mislabeled, for comparison). The random strategy still performed better, compared to points selected by the above algorithm. This hints towards the idea that there are points, and a large number of them, such that the network not only learns a wrong region around them, but probably learns stretched out, far reaching regions from the decision boundary that it would have learnt on a clean dataset.

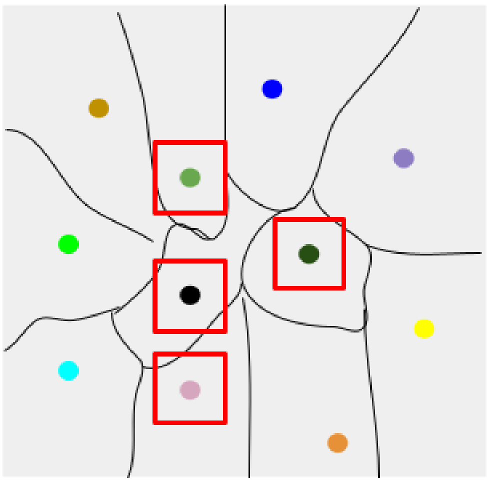


Figure 4.2: Idea behind using different color labels for each point in a specific class, and getting a network to memorize the assigned labels. The black lines show the decision boundaries learnt for the different colors, and red boxes show the points which are vulnerable because of a boundary passing through the l_∞ balls around those points.

Chapter 5

Experimental results

5.1 Toy Distribution

To develop a better understanding of how neural networks fit mislabeled points, what exactly is the cause of the huge drop in their vulnerability when *random* label noise is introduced into the dataset, and also why the highest probability density regions are not the best to label-flip, we carry out some controlled experiments in this chapter. Some of the following experiments utilize a toy dataset which consists of two classes, with the instance space being $\mathbb{R}^d$. n samples are generated from the gaussian mixture

$$\mathcal{P}_{\text{toy}}(\mathbf{x}) = \frac{1}{2}\mathcal{N}(\mathbf{x}| - \mu\mathbf{1}_d, \sigma^2 I_d) + \frac{1}{2}\mathcal{N}(\mathbf{x}|\mu\mathbf{1}_d, \sigma^2 I_d)$$

and labelled with the ground truth labeling function

$$c_{\text{toy}}(\mathbf{x}) = [\![\mathbf{x}^T.\mathbf{1_d} \geq 0]\!]$$

to create a dataset. This dataset is denoted as $\mathcal{D}_{\text{toy}}$. From this, we create a *poisoned* dataset $\mathcal{D}^r_{\text{toy}}$ where we place b mislabeled points for each class, at r distance from the means of the 2 classes, (uniformly sampled at r distance). We drop many of these details from the notation of $\mathcal{D}_{\text{toy}}$ for clarity. We are *placing* mislabeled points, and not *flipping* since that provides greater control over toy experiments.

Figure 5.1 shows samples from $\mathcal{D}_{\text{toy}}$ with $\mu = 10, \sigma^2 = 1, d = 2$.

5.1.1 Toy distribution with points far away from the decision boundary

Here, we aim to study the vulnerability of networks when $\mu \gg \sigma$. The decision boundary given by $\mathbf{x}^T.\mathbf{1_d} = 0$ is far away from where the points are located. For

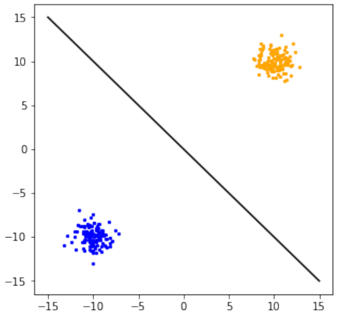


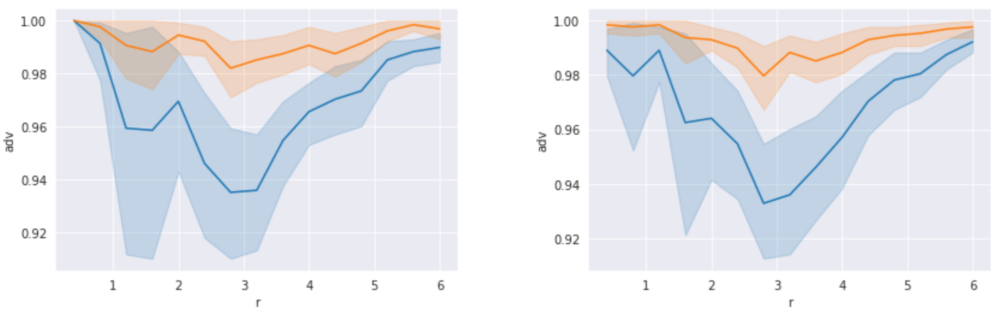
Figure 5.1: Toy data distribution with $d = 2,\ \mu = 10,\ \sigma^2 = 1,\ d = 2$

now, this notion of being far away is not well defined. But what we want to study is the effect on the adversarial vulnerability of networks when poisons are very far away from the boundary that these networks would have learnt otherwise in the absence of poisons. This decision boundary that the classifier would have learnt in the absence of poisons will be refered to as the *natural boundary*. We attempt to keep the boundary far away so that the networks do not stretch the natural boundary, in order to accommodate poisons. The networks should be learning clean, new pockets around the mislabeled points, where they give the wrong answer. It is a valid question as to how we might know whether the affect we want to avoid is actually not happening, and that networks are learning closed pockets around mislabeled points. For this, we found an experimental trick: mislabeling points to a third, new class that is not present in the dataset. Since this class is not present in the training set, when the network memorises them, it cannot stretch the natural decision boundary to do so. The training setup for this case, with the new class introduced, is the exact same, with one extra logit added to the output layer of the network.

The adversarial accuracy of models trained in this experiment were evaluated using an l_∞ PGD adversary, with an attack radius of 0.2.

Figure 5.2 shows the results of two runs $(d = 16, b = 2, \sigma^2 = 1, \mu = 10)$ with 64 points in the training set (including the mislabeled points). We fit networks to different toy datasets with the r parameter being varied. Reiterating, r is the distance from the means from which the mislabeled points are placed (in this case, 2 mislabeled points per class). The results are summarised as follows:

1. Two data points with label 0 were placed near $\mu\mathbf{1}_d$ and two points labelled with 1 were placed near $-\mu\mathbf{1}_d$. Classifiers were then trained to overfit to this dataset (achieve 100% training accuracy). The curve shows that adversarial accuracy first decreases and then increases, as poisons are placed farther and farther away from the means of the Gaussians. This is even more evidence for the conjecture that placing mislabeled points in the highest probability



(a) Poisons have labels set to the opposite class

(b) Poisons have labels set to a third, new class

Figure 5.2: 🟧 is the test accuracy 🟦 is the adversarial accuracy. r axis represents the distance from the means of the gaussians where mislabelled points (poisons) were placed. Setting wrong labels to a third class produces a curve very similar to the case where labels were set to the opposite class for poisons. Multiple models were trained, and randomisation was also done on the poisons placed for each run, which is what the confidence intervals and mean curves are representative of.

density regions does not hurt the adversarial accuracy by the largest factor.

2. The above experiment was repeated, but the label-flipping is done to a third, new class. This was to see whether the drop in adversarial accuracy primarily stems from learning more complex decision boundaries in the first experiment where the poisons were labeled with the opposite class, or the vulnerability is equally bad when the classifier has to learn new regions with the third class as the label. The curve obtained in this case is very similar to the previous experiment, indicating that regions memorized to be of the opposite class is not worse for the adversarial accuracy, in comparison to the case where the network is forced to memorize new regions

For the exact same setup, when the dimensionality d is increased, the adversarial accuracy quickly shoots up to one and the curve becomes flatter (as shown in the next section). Since standard datasets are generally high dimensional, the next section involves 64 dimensional toy data.

5.1.2 Toy distribution with points close to the decision boundary

Here, we further carry out experiments on the toy dataset, this time with higher dimensionality (64 dimensions), and vary μ and r, to observe how the adversarial accuracy of a fit network changes. This time, we are interested in observing the affects when mislabeled points are placed close to the decision boundary as well. The increase in dimensionality is because we need to ensure that the effects we see are not merely artefacts of low dimensionality (the results of the previous section are such artefacts, since for the same strength of the adversary, the accuracy turns out to be 100% for $d = 64$).

Differences in the adversarial accuracies, between the cases where we flip to the other class, and when we flip to a 3rd class, are *larger* when the means are *closer*.

We vary μ and r, fit networks for each case, and estimate the adversarial accuracy of the fit network. We carry out these experiments for $d = 64$, with a 4 layer fully connected, ReLU activated network. After creating the poisoned dataset, we fit networks with the stated architecture to these poisoned datasets, and evaluate their adversarial as well as natural accuracies. The results clearly indicate that placing mislabeled poisons near the mean is the *worst* for having high adversarial error. There is high variance in the adversarial accuracy of models given vanilla training, hence we ran multiple seeds for each setup.

For the case where the means were very far apart, the adversarial accuracy was the highest, and the poisons had almost no effect. Closer the class-means get, lower the adversarial accuracy.

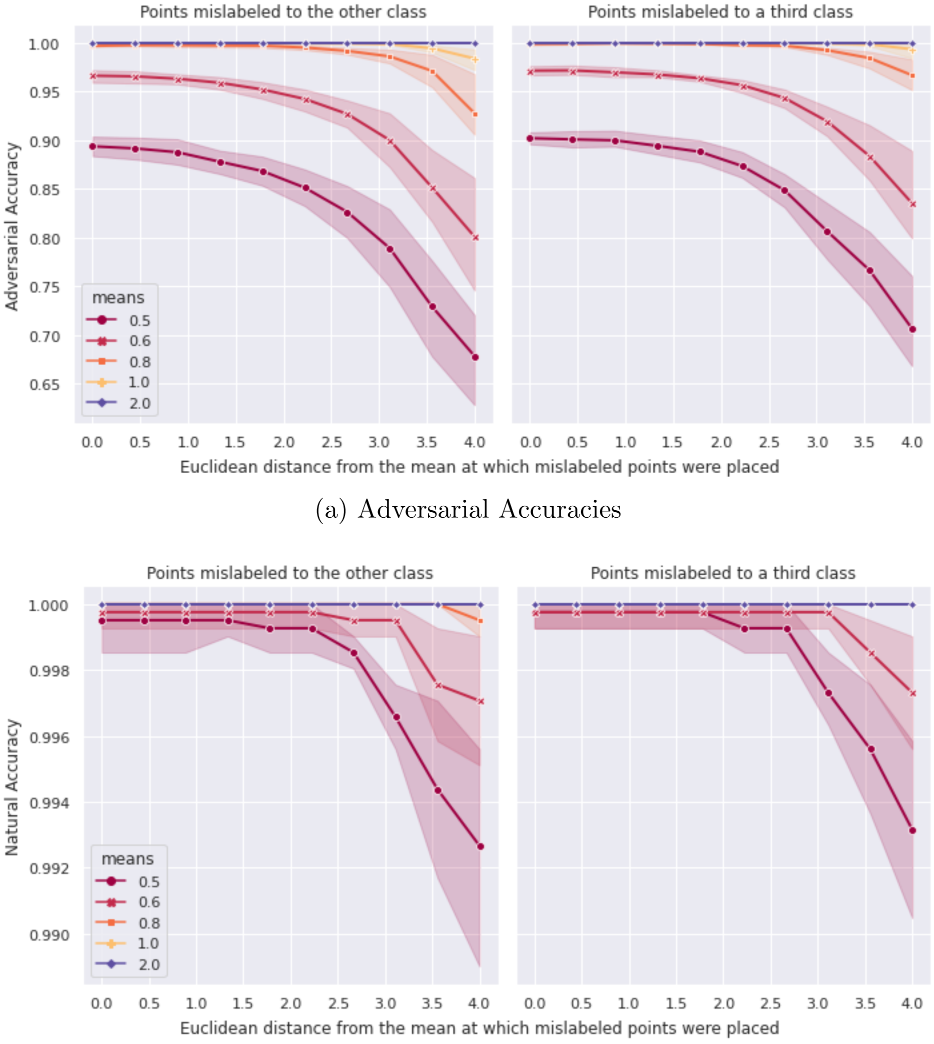
We also made another change to the setup, like in the previous section, and repeated the above experiments. The training setup is the exact same, except that this time, we mislabel points to a *third* class, which is not present in the dataset. The network is given one extra unit in its output layer, and all hypterparameters are preserved. All of the relative trends we observed above were observed again, but this time the networks had *higher* adversarial accuracies for cases where mislabeling was done to a third class.

Another point to note is that while the natural accuracies of the networks also showed the same trends as the adversarial accuracies, these are on a very different scale, with all values being upwards the 99% accuracy mark.

We can derive the following conclusions from the above experiments:

1. Adversarial accuracy is hurt more by mislabeling to a class whose decision boundary lies close. This is indicated by the fact that when we flipped the label to a new, 3rd class, the adversarial accuracy was better than when we flipped the label to the other class.

2. Placing the mislabeled points at the highest probability density regions is ineffective when our intention is to increase the adversarial error. In fact, in many of the above experiments, the networks simply did not fit the mislabeled points placed at these high density regions. This is further motivation to not place mislabeled points in such regions, since we cannot expect the



(a) Adversarial Accuracies

(b) Natural Accuracies

Figure 5.3: $d = 64, b = 2, \sigma^2 = 1.0$; 2 mislabeled points were placed per class. As before, randomisation is over training multiple models with different initialisations, and placing poisons randomly sampled at a given r from the means.

victims' models to memorize these points. They can simply use *early stopping*, and their models will be as good in adversarial accuracy as models that would have been trained on clean datasets.

3. When the class-means are closer to each other, the adversarial error is *higher*. This indicates that the most adversarially vulnerable networks simply do not memorize a closed region around mislabeled points, rather they tweak the decision boundary to include the mislabeled points, rendering large amounts

of probability mass vulnerable “on the way”. Figure 5.4 shows a visualisation of this. Further experiments add strong evidence in support of this claim.

Till this point in this chapter, we do not have enough insights to exactly understand what might be an optimal strategy to flip labels.

5.2 The worst poisons are not the ones producing pockets with the wrong label

5.2.1 Flipping to an 11th class in MNIST

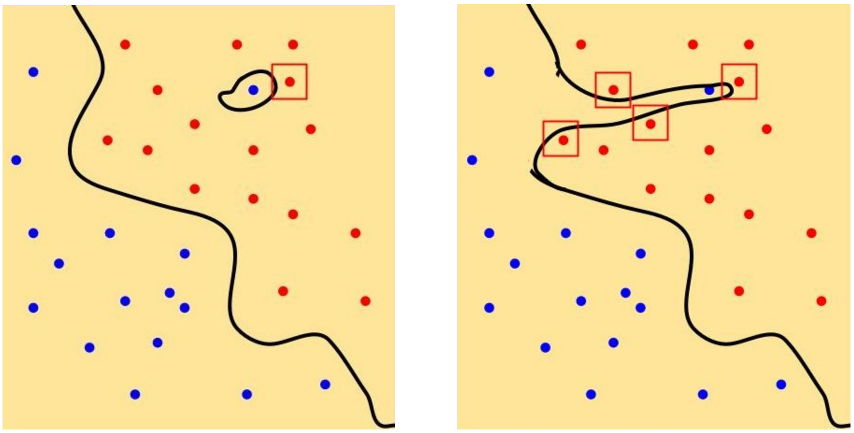
Similar to the approach we took for toy experiments, here we attempt to study classifiers which memorise the poisoned MNIST dataset. We first randomly select 10 points per class from the MNIST dataset which will be label-flipped, and then we fit 2 different classifiers: one classifier is fit with the mislabled points set to the next class in the dataset, and the other classifier, with one extra logit added in its output layer, is fit with the mislabeled points set to an 11th class, which is not present in the dataset. Adversarial evaluation is untargeted for these experiments.

Poisoning Experiment	Adversarial Accuracy(%)
Mislabeling to next class	36.46 ± 4.42
Mislabeling to an 11th class	41.51 ± 4.26

Table 5.1: Adversarial Accuracies of networks when poisoning is done to a new class compared to the case when poisoning is done to the next class cyclically. The accuracy was evaluated against an l_∞ PGD adversary with an attack radius of 0.1. Randomisation in the above results is over model initialisation. For each run, the same points in MNIST are label-flipped.

Table 5.1 shows the results of this experiment. *The network trained with points flipped to the 11th class had a much higher adversarial accuracy compared to the other network.* This indicates that the poor adversarial accuracy of models trained with random label noise is probably because of the mislabeled points stretching the decision boundary, and rendering large amounts of probability mass vulnerable on the way. We ran the above experiment with other randomly selected points to label-flip as well, and the results show the same trend.

5.3 Back to MNIST: Analysis using adversarial paths



(a) Network learning a ”pocket” around a mislabeled point

(b) Network teaking the decision boundary to ”accommodate” a mislabaled point

Figure 5.4: The above figure shows two different ways in which a classifier could memorize a mislabeled point. If the decision boundary is tweaked to accommodate a mislabeled point, it could lead to much higher adversarial errors, compared to the other case.

The following section descibes a series of experiments, which reveal why the random strategy performs surprisingly well in searching for poisons, and also sheds light on the nature of training points which when mislabeled, cause a massive drop in the adversarial accuracy of the model.

The first experiment shows the result for a binary classification problem, with the MNIST dataset's classes 0 and 1 only. The other case shows the same experiment done on the full MNIST dataset. The experiments also reveal why we will always get good enough results (i.e., high adversarial error) while randomly selecting points to label-flip, because it turns out that points responsible for huge drop in adversarial accuracy when mislabeled are very large in number.

5.3.1 Finding a path from a point to the boundary

A requirement to the experiments that follow is a procedure to find a path from a data point to the decision boundary of a given classifier. For instance, if we are given a classifier trained on the full MNIST dataset, and are given a datapoint $\mathbf{x}$ (i.e., some handwritten digit), then we need to find a path (the shortest path ideally) from this datapoint to the decision boundary. We might also have a constraint: finding a path to the decision boundary of the current data point's class, and a target class. Further, this path is unconstrained, and we are not concerning ourselves with finding such a path that completely lies inside a certain region, such as an l_p ball of specific radius around the input $\mathbf{x}$. We will carry out an *unbounded* gradient descent attack, and record all the adversarial examples we get on the way.

For a small step size ω, we make gradient ascent updates on the input image that is plugged into a model, by backpropagating gradients from the loss, all over to the inputs. During these gradient updates, unlike Projected Gradient Descent, we do not project the updated input back into an l_p ball of radius ρ around the original input. After every update, we record the updated input. This procedure terminates when the output of the classifier is flipped, or when some maximum number of steps have already been taken. We will call this procedure *untargeted-unbounded-gradient-descent*.

When we have several classes, it makes sense to study the effect of poisons by evaluating the adversarial accuracy of the model using targeted attacks from the original class of the poisons, to the class the poisons were mislabeled to. For instance, if some points in class 9 of the MNIST dataset were mislabeled to class 0, then it makes sense to evaluate the the impact of these poisons by estimating the robustness of the resultant model using targeted adversarial attacks from class 9 to class 0. We hence, find out paths in multi-class settings by recording the path from a given data point to a target decision boundary. We will call this procedure *targeted-unbounded-gradient-ascent* 8.

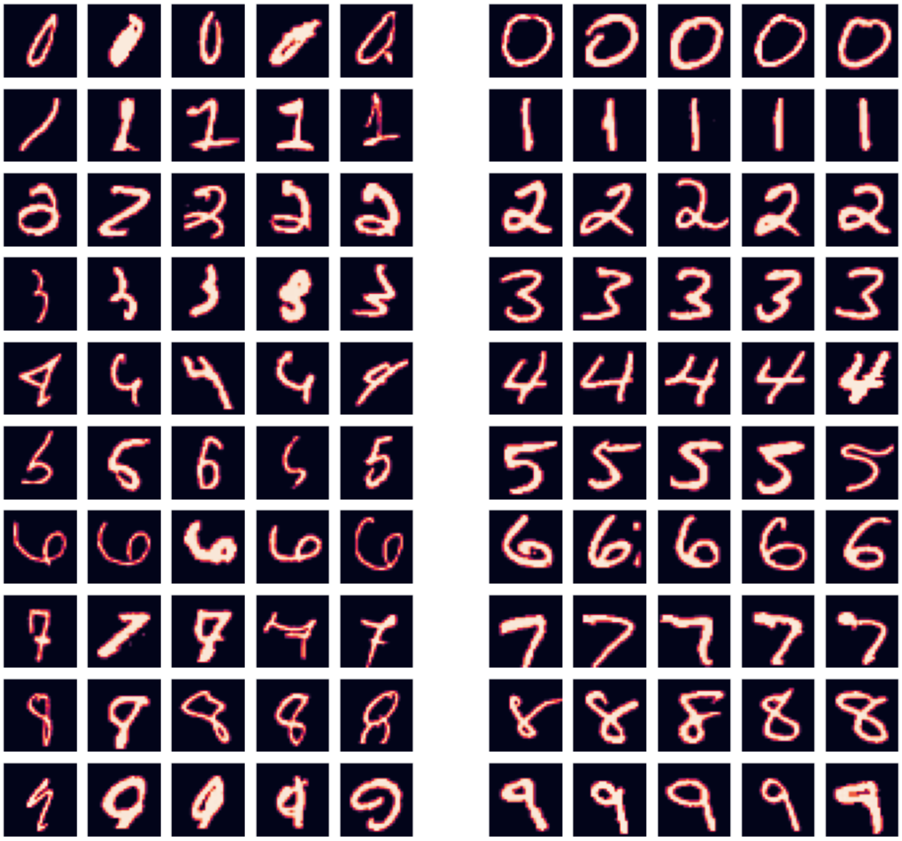
We will refer to the paths traced out by the above algorithms as *adversarial paths*.

Interestingly, when we found out the adversarial paths for some of the below experiments, we came across label noise already present in MNIST. This is illustrated in figure 5.5. We selected points closest to and farthest from the decision boundary, and plotted them out. Many of the points closest to the decision boundary already have label noise. Adversarial paths were found out from each class to the next, in a targeted manner, for this analysis.

5.3.2 Using adversarial paths to analyse poisons

For a given dataset, we first train a classifier on the clean datasets, without placing any mislabeled poisons. Let us call this the *clean classifier*. We then iterate over the training set, and find out adversarial paths for each point.

The hypothesis is that if points are too close to the decision boundary, then it is not very optimal to mislabel them (optimal in the sense that the adversarial *error* of a model trained on the dataset with mislabeled points has to be maximal),



(a) 5 points per class closest to the boundary of the next class

(b) 5 points per class farthest from the boundary of the next class

Figure 5.5: After finding out the adversarial paths from each training point to the decision boundary of a model trained on MNIST, we visualized some points closest to and farthest from the decision boundary. Examples pertaining to each class have been put on different rows: 0 to 9 from top to bottom. Note that the path to the next class was found by untargeted-unbounded-gradient-ascent, which is why label noise is such that some of the 0s look like 1s, 1s like 2s, and so on. Constratingly, the images on the right are far away from the boundary, again evaluated in a targeted manner. These images seem very crisp.

since the decision boundary is going to slightly be modified to accommodate the poison. If it is too far away, then it might be the case that the classifier resorts to memorising the poison using an alternative pocket, rather than modify the natural decision boundary.

Algorithm 4 Targeted Unbounded Gradient Ascent

```
procedure ADVERSARIAL-PATH($\mathcal{C}, \mathbf{x}, t, m, \epsilon$)
    Path ← [$\mathbf{x}$]
    Output ← $\mathcal{C}(\mathbf{x})$
    while len(Path) $\leq m$ and $t \neq$ Output do
        $\mathbf{x}$ = TARGETED-FGSM($\mathcal{C}, \mathbf{x}, t, \epsilon$)
        Path.append($\mathbf{x}$)
        Output ← $\mathcal{C}(\mathbf{x})$
    return Path
```

5.3.3 The experiments

0-1 MNIST

We only consider a 2-class dataset, composed of the classes “0” and “1”, from MNIST.

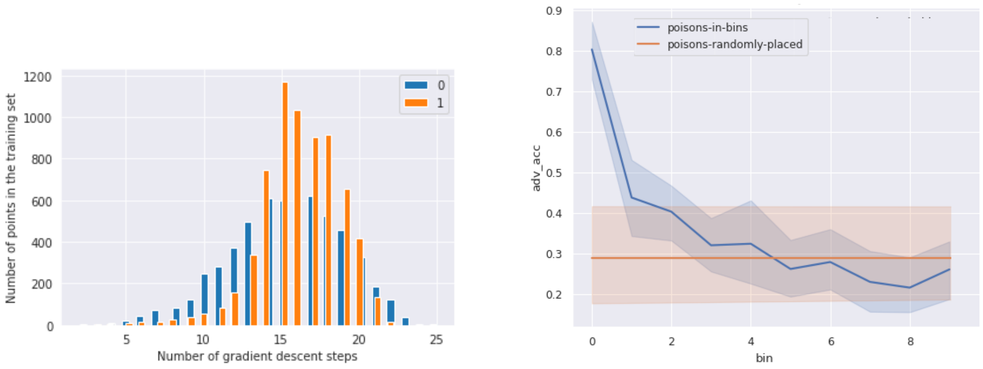
A binary classifier is trained on this dataset. We then find out the adversarial paths for each point in the training set. Figure 5.6 shows an illustration of the distribution of the length of these paths over the training set points, per class.

Next, we divide the training set points into various disjoint buckets, each bucket holding points from a different range of the length of the adversarial path. Essentially, the x-axis of 5.6 is divided into ranges, and points falling into each range are put into the bucket for that range.

We train several models for each bucket, each with randomly selected poisons from that bucket. Figure 5.6 illustrates the results. Closer the mislabeled points are to the original decision boundary, less the fall in the adversarial accuracy of the model trained after mislabeling that point. As we go farther from the decision boundary, the poisons have a greater impact. The next experiment will shed more light on placing poisons even farther from the boundary (in the full MNIST case, where some classes were observed to be even farther from the decision boundary), and the impact of the poisons starts wearing away.

MNIST

We now consider the full MNIST dataset, and any points selected to be mislabeled, will be cyclically set to the next class in [0..9]. We first, like we did previously,



(a) Distribution of the length of adversarial paths, over each class of 0-1 MNIST.

(b) Adversarial accuracy of network trained on a poisoned 0-1 MNIST dataset, with poisons selected at increasing distances from the decision boundary.

Figure 5.6: (a) shows the distribution over the number of gradient ascent updates it took for the classifier to flip its output, for 0-1 MNIST, for the points in the training set. The y-axis indicates the number of points in the training set. The step size used for the above was 0.01, on $[0, 1]$ normalized input images. (b) The x axis only denotes an ordinal value: higher values represent buckets holding points for which the adversarial paths were lengthier. The points were segregated into different buckets for (b), each bucket having points at a certain distance range from the boundary. The test accuracy of all of the above models are greater than 99%.

find out the adversarial paths for each training point. But since we are flipping the label to the next class, we will find out adversarial paths for each point in a targeted manner, with the next class being the target.

Figure 5.7 shows the distribution of the number of gradient descent steps each adversarial path took, i.e. the length of the targeted-adversarial paths for each point in the training set. First of all, we observe that we cannot have universal buckets for all classes, since some classes are much farther from the decision boundary than others. For instance, points in class 1 are much closer to the boundary than points in class 9. So it is not possible to do experiments where we flip points in the same range for all classes. There is a solution however, which is that we only place mislabeled poisons for *one class* of the dataset, flipping the label cyclically to the next class. The adversarial accuracy of the resultant network will only be evaluated on targeted attacks from that selected class, to the next one (the class the labels were flipped to).

The results of this experiment show that the closer the mislabeled points are to the decision boundary, lower their impact. As we mislabel points farther from the boundary, the impact increases, until a certain point. After a point, selecting points farther from the decision boundary worsens the impact. The curves are as shown in figure 5.8, and the 3 graphs shown are for classes 1, 4, and 9 flipped to 2, 5, and 0 respectively. These classes were chosen since they are at increasing distances from the decision boundary. Points in class 1 are very close, and points in class 9 are very far. So they capture cases at different regions in the spectrum.

Also, the minima observed for all the curves is where most of the training set points lie, which explains why the random strategy manages to cause a huge fall in the adversarial accuracy. Randomly selecting points to flip still has a high chance of selecting these impactful points, since there are so many of them.

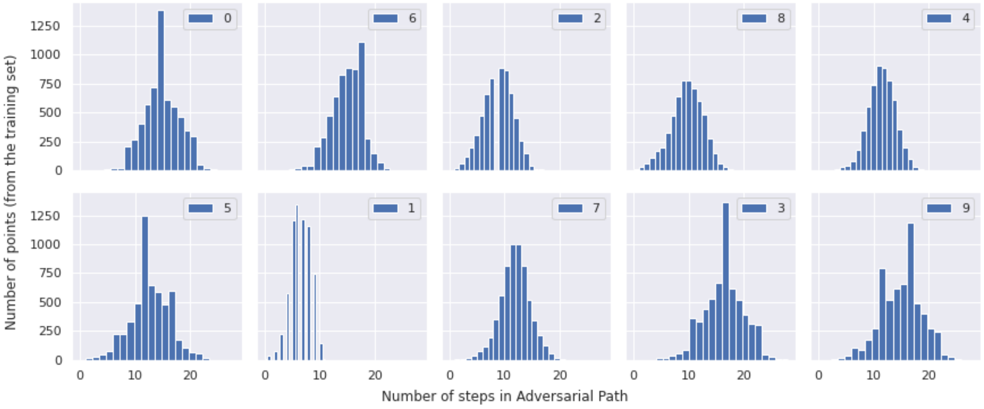


Figure 5.7: Distribution over the number of gradient ascent updates it took for the classifier to flip its output, for the full MNIST dataset, for the points in the training set. The y-axis indicates the number of points in the training set. The step size used for the above was 0.01, on $[0, 1]$ normalized input images. Note that for class ”1”, it takes not more than 10 steps for almos the entire training set, which means for an attack radius of $\epsilon = 0.1$, the entire class must be vulnerable, which is indeed the case. The adversarial accuracy of the model trained on the clean dataset, on targeted attacks from 1 to 2 is 0.

This provides a way to select points to flip, given a budget: we can fit a clean model to the dataset, and find out adversarial paths for each point. The targeted or untargeted nature of the adversarial path will depend on the kind of vulnerabity we want in the victim's model. For instance, if we decide to flip a class A to a class B, then the adversarial paths should be targeted from A to B, after which the victim's model will be vulnerable to targeted attacks from class A to class B. We can then perform an analysis over the different adversarial path lengths, fitting models for each case, which will guide towards refining our search towards a smaller set of points to search over.

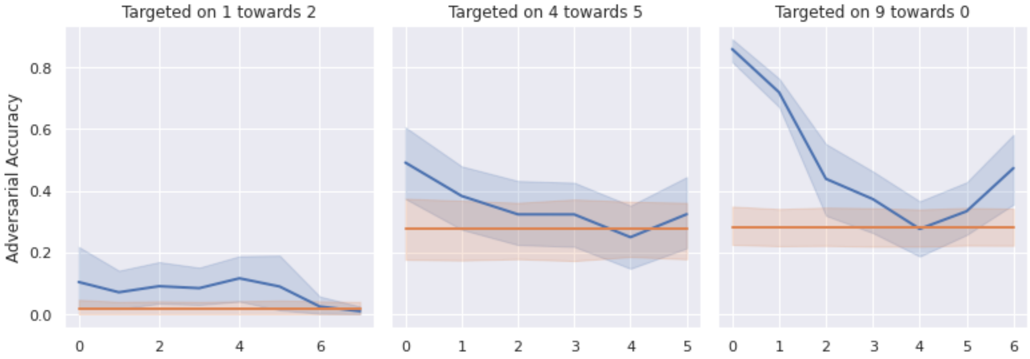


Figure 5.8: Full MNIST dataset. The x axis only denotes an ordinal value: higher values represents represents buckets holding points for which the adversarial paths were lengthier. The orange line is a baseline showing the adversarial accuracy when same number of points are randomly selected to poison. The 3 graphs shown are for classes 1, 4, and 9 flipped to 2, 5, and 0 respectively. These classes are at increasing distances from the decision boundary: points in class 1 are very close, and points in class 9 are very far, class 4 being an intermediate case. The adversarial accuracy evaluation for each case is done using targeted attacks from classes 1, 4, and 9 to 2, 5, 0 respectively to study the effects of poisons. 10 poisons were placed per class for these runs. The test accuracy of all of the networks evaluated in the above diagram is more than 99%.

Chapter 6

Future Work: A Meta-Learning Approach

The problem of *crafting* label noise can be framed as a multi-level optimisation problem. While meta-learning is a broadly used term, we will be referring to the *gradient-based* meta-learning setup [6, 16, 7]. Under this setup, we have an inner objective, $\mathcal{L}^{\text{in}}$ minimised inside a loop, and outer objective $\mathcal{L}^{\text{out}}$, minimised in an outer-loop, nesting the inner objective inside of it. There are certain parameters ϕ utilised by the training setup of the inner loop, which the outer loop has control over, and parameters θ which the inner loop has control over. The aim is to compute gradients $\nabla_\phi \mathcal{L}^{\text{out}}$ to make gradient-descent updates to ϕ. The way this gradient computation is done, is by unfolding training steps in the inner loop, and backpropagating through the gradient updates made in this inner loop.

For simplicity, let us assume we have to solve a binary classification problem. There are only 2 classes, and we are given a dataset with n points sampled from $\mathcal{P}$, and labeled with the ground truth labeling function c to produce a dataset $S_{\text{clean}} = \{(\mathbf{x}_i, c(\mathbf{x}_i))\}_{i=1}^n$.

We are to find one point in this dataset, i.e. some $i \in \{1..n\}$. This point should be such that if the given point's label is flipped to the other class, the adversarial accuracy of a classifier trained on this modified, or *poisoned* dataset is as low as possible.

The following are different agents optimising over different variables in the problem we are concerned with:

1. The *victim* optimises over the model parameters, attempting to minimise the training loss

2. The *input adversary*, as we will call it, optimises over constrained perturbations to the input, attempting to maximise the training loss

3. The *poisoning adversary*, as we will call it, optimises over which points to carry out a label change, given a budget constraint over the number of labels it can modify, and attempts to minimise the adversarial accuracy of the model

Note that in each case, the agents are concerned with maximising or minimising *accuracy*, but the loss is the only metric under control. An appropriate loss, when minimised or maximised, can be expected to increase/decrease the accuracy of the model.

Let the size of the training set be n. Let the label noise the poisoning adversary adds to the training set be represented as $\mathbf{\Delta y} \in \{0,1\}^n$ such that $\sum_i \mathbf{\Delta y}_i = 1$, $\mathbf{y}$ being the vector of labels in the dataset, ith label being $\mathbf{y}_i$, X being a matrix storing the training points as its row vectors, with the ith data point being $\mathbf{x_i}$. Let $\mathcal{H}$ be the hypothesis class, i.e. the set of functions the *victim* is optimising over (one example of a hypothesis class is the set of all neural networks of a fixed architecture, with different parameter values). The problem can then be framed as:

$$\mathbf{\Delta y}^* = \underset{\mathbf{\Delta y} \text{ is one-hot}}{\arg\max}\ \mathcal{R}^{\rho}_{\text{Adv}}\left(\underset{\mathcal{C}\in\mathcal{H}}{\arg\min}\ \mathcal{L}(\mathcal{C}, X, \mathbf{y} \oplus \mathbf{\Delta y})\right) \tag{6.1}$$

where $\oplus$ is the "xor" operation, i.e. allows label flipping to happen wherever $\mathbf{\Delta y}$ is one-hot at. This can be practically implemented as

$$\mathbf{\Delta y}^* = \underset{\mathbf{\Delta y} \text{ is one-hot}}{\arg\max}\ \max_{X' \in \mathcal{B}^p_\rho(X)} \mathcal{L}\left(\underset{\mathcal{C}\in\mathcal{H}}{\arg\min}\ \mathcal{L}(\mathcal{C}, X, \mathbf{y} \oplus \mathbf{\Delta y}), X', \mathbf{y}\right) \tag{6.2}$$

where $\mathcal{B}^p_\rho(X)$ is defined as the set of all matrices M such that $\forall i \in \{1..n\}$, $\mathbf{m_i} \in \mathcal{B}^p_\rho(\mathbf{x_i})$.

While implementing, we can have a learnable real valued vector $\mathbf{h}$ with dimensionality n where n is the training set size. Let us call this the "flip logit vector". We derive a one-hot vector over the training set using $\mathbf{h}$, by using the Gumbel-Softmax trick used to create a "smooth" function that behaves like argmax, but that allows gradients to backpropagate [14]. Given a logit vector $\mathbf{h}$, the Gumbel-Softmax trick produces the almost one-hot vector $\mathbf{f}$ such that:

$$\mathbf{f}_i = \frac{\exp((\mathbf{h}_i + \mathbf{g}_i)/\tau)}{\sum_{j=1}^{n} \exp((\mathbf{h}_j + \mathbf{g}_j)/\tau)}$$

where $\tau > 0$ is a hyperparameter called the temperature and $\mathbf{g}$ is an n-dimensional vector such that each entry of it, $\mathbf{g}_i$ is sampled from the Gumbel Distribution i.i.d.

Now that we have $\mathbf{f}$, such that we can backpropagate gradients through it, we show the model we are training on (we do not need the exact same model that the *victim* might be using. Most attacks in literature which make use of models to craft the attack assume a model and hope that the attack generalizes, which works very well practically) the modified label vector

$$\mathbf{y}' = \mathbf{y} \odot (1 - \mathbf{f}) + \mathbf{y} \odot (1 - \mathbf{f})$$

Now we make gradient updates on the network we have, simulating training with these new labels, and recording all operations we do, use the final network produced by these updates to craft adversarial examples from the training set itself, and finally computing the loss of this final model on the adversarial examples. This loss has to be *maximised.* So we backpropagate from this loss, and receive gradients for **h** through **f**, and we do a *gradient ascent* on **h**.

We could not find a computationally feasible way to implement this. Practical datasets have thousands of training examples per class, and the inner loop of this method involves multiple gradient descent steps, which means we will have to calculate and backpropagate second order gradients through these steps, while involving the entire training set in the update steps, so as to get gradients back to each dimension of **f**, since we require exactly one dimension selected by **f**. Further, after the updates in the inner loop, we also need to compute adversarial examples for each point in the training set, with the final updated network, and the outer loop's maximisation is done over these adversarial examples. Figure 6.1 shows an example for a small dataset with just 2 batches of size 4.

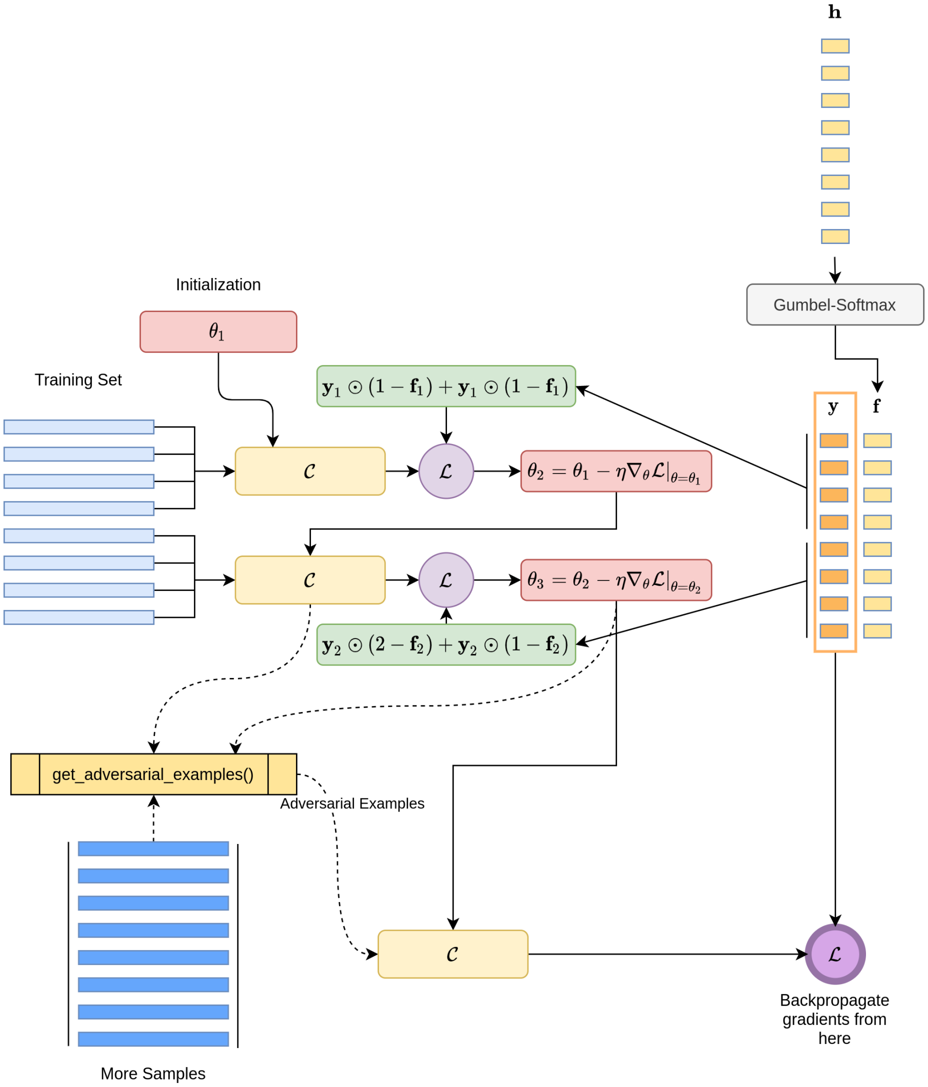

h
Gumbel-Softmax
y
f
Initialization
θ_1
Training Set
$\mathbf{y}_1 \odot (1 - \mathbf{f}_1) + \mathbf{y}_1 \odot (1 - \mathbf{f}_1)$
$\mathcal{C}$
$\mathcal{L}$
$\theta_2 = \theta_1 - \eta\nabla_\theta\mathcal{L}|_{\theta=\theta_1}$
$\mathcal{C}$
$\mathcal{L}$
$\theta_3 = \theta_2 - \eta\nabla_\theta\mathcal{L}|_{\theta=\theta_2}$
$\mathbf{y}_2 \odot (2 - \mathbf{f}_2) + \mathbf{y}_2 \odot (1 - \mathbf{f}_2)$
get_adversarial_examples()
Adversarial Examples
$\mathcal{C}$
$\mathcal{L}$
Backpropagate gradients from here
More Samples

Figure 6.1: An example of the multi-level optimisation case, where we have a small dataset of 8 examples, and two gradient updates are made in the innermost loop, with each batch update having 4 examples. The function get_adversarial_examples () produces adversarial exapmles for a given network and inputs. Dotted lines indicate that the operations are not to be recorded, and bold lines indicate that the operations should be recorded, so that we can pass gradients through them during the backward pass. $\odot$ refers to elementwise multiplication.

Chapter 7

Conclusion

The ideas discussed in this thesis uncover properties of neural networks as they memorise label noise. We establish that placing mislabeled points at the highest probability density regions is sub-optimal to hurt the adversarial accuracy of models, and also provide a way to select points to label flip, by analysing their adversarial paths for models trained on clean data.

While we extend and also provide a new bound for the adversarial risk, when classifiers memorise label noise, it is clear from the experiments in this thesis that these bounds, as well as the one in literature, with regards to uniform label noise are highly vacuous in practice. This is because networks seem to be much more smoother in memorising poisons than these analyses assume. The regions wrongly learnt by networks are much larger in size, rendering a large amount of probability mass vulnerable.

The adversairal path experiments conducted show that the points that affect the adversarial accuracy the most when mislabeled, are the ones which force the network to learn complex decision boundaries. If, in order to memorise a mislabeled point, networks tweak the decision boundary they would have learnt otherwise, in the absence of that mislabeled point, the adversarial accuracy is massively hurt.

Bibliography

[1] Dominating set (chapter 7). URL `http://ac.informatik.uni-freiburg.de/teaching/ss_12/netalg/lectures/chapter7.pdf`.

[2] Peter L Bartlett, Philip M Long, Gábor Lugosi, and Alexander Tsigler. Benign overfitting in linear regression. *Proceedings of the National Academy of Sciences*, 2020.

[3] Mikhail Belkin, Siyuan Ma, and Soumik Mandal. To understand deep learning we need to understand kernel learning. *arXiv preprint arXiv:1802.01396*, 2018.

[4] Adith Boloor, Xin He, Christopher Gill, Yevgeniy Vorobeychik, and Xuan Zhang. Simple physical adversarial examples against end-to-end autonomous driving models. pages 1–7, 06 2019. doi: 10.1109/ICESS.2019.8782514.

[5] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *CoRR*, abs/1712.05526, 2017. URL `http://arxiv.org/abs/1712.05526`.

[6] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 1126–1135. PMLR, 2017. URL `http://proceedings.mlr.press/v70/finn17a.html`.

[7] Luca Franceschi, Michele Donini, Paolo Frasconi, and Massimiliano Pontil. Forward and reverse gradient-based hyperparameter optimization. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 1165–1173. PMLR, 2017. URL `http://proceedings.mlr.press/v70/franceschi17a.html`.

[8] I. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *CoRR*, abs/1412.6572, 2015.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015. URL `http://arxiv.org/abs/1502.01852`.

[10] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[11] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL `http://arxiv.org/abs/1412.6980`.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL `https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf`.

[13] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1(4):541–551, 12 1989. ISSN 0899-7667. doi: 10.1162/neco.1989.1.4.541. URL `https://doi.org/10.1162/neco.1989.1.4.541`.

[14] Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL `https://openreview.net/forum?id=S1jE5L5gl`.

[15] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL `https://openreview.net/forum?id=rJzIBfZAb`.

[16] Alex Nichol, Joshua Achiam, and J. Schulman. On first-order meta-learning algorithms. *ArXiv*, abs/1803.02999, 2018.

[17] Elan Rosenfeld, Ezra Winston, Pradeep Ravikumar, and J. Zico Kolter. Certified robustness to label-flipping attacks via randomized smoothing. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 8230–8241. PMLR, 2020. URL `http://proceedings.mlr.press/v119/rosenfeld20b.html`.

[18] Aniruddha Saha, Akshayvarun Subramanya, and Hamed Pirsiavash. Hidden trigger backdoor attacks. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 11957–11965. AAAI Press, 2020. URL `https://aaai.org/ojs/index.php/AAAI/article/view/6871`.

[19] Amartya Sanyal, Varun Kanade, Puneet K. Dokania, and Philip H.S. Torr. How benign is benign overfitting ? In *Submitted to International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=g-wu9TMPODo`. under review.

[20] Avi Schwarzschild, Micah Goldblum, Arjun Gupta, John P. Dickerson, and Tom Goldstein. Just how toxic is data poisoning? A unified benchmark for backdoor and data poisoning attacks. *CoRR*, abs/2006.12557, 2020. URL `https://arxiv.org/abs/2006.12557`.

[21] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 10 2017. doi: 10.1038/nature24270.

[22] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3104–3112, 2014. URL `https://proceedings.neurips.cc/paper/2014/hash/a14ac55a4f27472c5d894ec1c3c743d2-Abstract.html`.

[23] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *International Conference on Learning Representations*, 2014. URL `http://arxiv.org/abs/1312.6199`.

[24] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017. URL `https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html`.

[25] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Çaglar Gülçehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft II using multi-agent reinforcement learning. *Nat.*, 575(7782):350–354, 2019. doi: 10.1038/s41586-019-1724-z. URL `https://doi.org/10.1038/s41586-019-1724-z`.

[26] Han Xiao, Huang Xiao, and C. Eckert. Adversarial label flips attack on support vector machines. In *ECAI*, 2012.

[27] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning (still) requires rethinking generalization. *Commun. ACM*, 64(3):107–115, 2021. doi: 10.1145/3446776. URL `https://doi.org/10.1145/3446776`.

[28] Chen Zhu, W. Ronny Huang, Hengduo Li, Gavin Taylor, Christoph Studer, and Tom Goldstein. Transferable clean-label poisoning attacks on deep neural nets. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 7614–7623. PMLR, 2019. URL `http://proceedings.mlr.press/v97/zhu19a.html`.