Testing Deep Neural Networks for Classification Tasks Through Adversarial Perturbations on Test Datasets

by

Rajasekhar Ponakala

A research study submitted in partial fulfillment of the requirements for the
degree of Master of Engineering in
Information and Communication Technology

Examination Committee: Prof. Matthew N. Dailey (Chairperson)
Dr. Mongkol Ekpanyapong
Dr. Attaphongse Taparugssanagorn

Nationality: Indian
Previous Degree: Bachelor of Technology in Electronics and Communication Engineering
Jawaharlal Nehru Technological University Hyderabad
Telagana, India

Scholarship Donor: AIT Fellowship

Asian Institute of Technology
School of Engineering and Technology
Thailand
December 2019

ACKNOWLEDGMENTS

I would like to thank Prof. Matthew N. Dailey for accepting me to do research under him. I gained lot of knowledge from him throughout my period at AIT. His belief in me has never let me down, instead it made me push my limits to work hard and achieve my goal. I will be grateful for his advice throughout my work.

I would also like to thank my committee members, Dr. Attaphongse Taparugssanagorn and Dr. Mongkol Ekpanyapong, for their valuable comments and suggestions on my work. I would also like to thank my friends in the ICT department at AIT for their support.

Last but not least, I would also like to thank my parents, for their love, support, and encouragement.

Rajasekhar Ponakala
May 2019

ABSTRACT

Deep learning models often provide high predictive accuracy on test datasets. However, sometimes, changes in the real-world distribution of the data in production may result in incorrect functioning. Such problems have motivated research in the safety and security-critical systems domains. In this research study, I test CNN models for vehicle type classification from the AIT VISION Lab. The current vehicle type model is a GoogLeNet model trained on the task of classifying an input image as a car, van, bus, truck, or pickup truck. I used FGSM to adversarially perturb the GoogLeNet model to identify error cases. To make the model testing approach more useful in a software development context, I developed a method to include adversarial testing of models in a continuous integration (CI) framework. The method used a plugin for the Jenkins open source CI framework to visualize results. With the plugin, we can visualize the build logs for standard and adversarially trained classifiers. This method can help the software engineering team to analyze the performance of model. With this method, any kind of dataset can be used to generate adversarial examples, enabling us to detect model degradation and improve their expected error behavior under unexpected circumstances.

Keywords: Adversarial robustness, Adversarial examples, Convolutional neural networks, Image classification, Continuous integration, Software testing.

TABLE OF CONTENTS

CHAPTER	TITLE	PAGE
	Title Page	i
	Acknowledgments	ii
	Abstract	iii
	Table of Contents	iv
	List of Figures	v
	List of Tables	vi
1	Introduction	1
	1.1 Background	1
	1.2 Problem Statement	2
	1.3 Objectives	3
	1.4 Organization of the report	3
2	Literature Review	4
	2.1 Convolutional neural network models for image classification	4
	2.2 Overview of adversarial examples	4
	2.3 Optimization perspective on adversarial training	8
	2.4 Verification of ML models	10
3	Methodology	14
	3.1 System Overview	14
	3.2 Adversarial Examples	14
	3.3 Continuous Integration framework model	15
4	Experimental Results	18
	4.1 Standard classifier performance	18
	4.2 Adversarially trained classifier performance	18
	4.3 Normalization bounds	18
	4.4 Jenkins job visualizations	19
5	Conclusion and Future Scope	25
	5.1 Conclusion	25
	5.2 Future scope	25
6	References	26
7	Appendices	27

LIST OF FIGURES

FIGURE	TITLE	PAGE
2.1	Inception module concept	5
2.2	Classic CNN architecture: LeNet5.	5
2.3	AlexNet architecture.	5
2.4	Adversarial examples in AlexNet.	7
2.5	panda and gibbon example.	7
2.6	Illustration of NCov	11
2.7	Gradient ascent process to find difference-inducing inputs.	12
2.8	Occlusions, blackouts, and tiny rectangles.	13
3.1	Vehicle type dataset. (a) Pickup. (b) Bus. (c) Van. (d) Sedan. (e) Truck.	15
3.2	Overview of adversarial testing process.	16
3.3	Stages in a Jenkins Pipeline.	17
4.1	Plot for ϵ versus validation accuracy	19
4.2	FGSM attack visualizations on standard model	20
4.3	Plot for ϵ versus validation accuracy	21
4.4	FGSM attack visualization for adversarial model	22
4.5	Jenkins job visualization for standard model	23
4.6	Jenkins job visualization for adversarial model	24

LIST OF TABLES

TABLE	TITLE	PAGE
2.1	CNNs introduced in recent years with performance in ImageNet competition.	6

CHAPTER 1

INTRODUCTION

This chapter provides information about the background, problem statement, and objectives of the research study.

1.1 Background

Science and technology enhances human civilization in society. Today, advances in technology are providing more advanced solutions to traditional problems. Research on machine learning (ML) draws upon various fields. Today, ML is a rapidly-growing field in industry. Visual pattern recognition is one of the currently most popular applications of ML. For example, automatic image classification has recently surpassed human performance for some datasets. The best performance on image classification dataset challenges such as ImageNet, MNIST, and CIFAR achieve 90-95% accuracy in different modes. Convolutional neural networks (CNNs) are the most popular deep neural networks for image classification tasks. State-of-the-art CNN models achieve very high accuracy on image classification dataset challenges.

As the technology and available computational power increases, CNN models are advancing to handle very large scale real-world image classification tasks. For example, GoogLeNet is an advanced CNN model used for image classification tasks. GoogLeNet achieves high efficiency on large scale datasets. Many industry-oriented ML solutions now use the GoogLeNet model because of its simplicity, parallelism, computation, and memory efficiency. Besides GoogLeNet, other CNN models such as LeNet, AlexNet, ZF Net, VGGNet, and ResNet perform well on specific datasets. Real-world applications such as traffic-law enforcement require good image classification performance when reading licence plates and when preforming traffic violation processing.

As the scope of image classification tasks in real-world problems increases, researchers are considering privacy and security aspects of ML. This is because of the misbehaving nature of ML models in production environments. ML models are prone to biases in real-world data that was not present in the test or validation phase of model development. People from the software testing domain would like to treat ML models as white boxes, looking at the internals of the model architecture to identify the misbehavior of neural processing layers. In summary, it is necessary to test ML models using automated test processes and adversarial methods that simulates real-world attacks.

1.2 Problem Statement

Machine learning (ML) models' ability to learn from datasets has many benefits. The process of training a model's behavior on a specific training set has caveats, however. For example, models may misclassify real-world data that are not present in the training dataset. To deal with such cases, several black-box and white-box testing techniques have evolved to improve the security of ML models. One relevant technique is adversarial testing of models.

Since predictive robustness is a concern, it is important to measure the internals of a model. Test data manipulation and model architecture are equal concerns. In the past, few escalated and concerned about the real-world error-rate of models on a task. As the models' complexity increases, there is increasing possibility of adversarial loopholes on the same dataset. An adversarial example is an example that has been carefully constructed to be misclassified.

Tools for testing models' internal architectures and creating adversarial examples are scarce. We need to develop appropriate tools for testing and measuring convolutional neural network (CNN) models. Finding and fixing bugs is cheaper the earlier we detect them. Moreover, existing ML systems have complex architectures, so we are unlikely to detect error cases on our own. We would like to put bounds on the possible level of incorrect behavior on input data. As of yet, adoption of automatic testing and measuring methods is very low today.

Recently, however, researchers have found ways to explore ML models' behavior with respect to their internal architecture and generate inputs that are likely to cause errors by applying real-world constraints. These approaches fall into the category of adversarial testing of the models. Since the dangers of putting trained model into production seen as larger, as a case study, this research study will focus on testing CNN model that classify vehicle types (VTypes include buses, sedans, pickups, trucks, and vans etc.) based training data from the AIT VISION lab. This process will help improve the robustness of trained other models. In order to make the system more reliable, I will develop a method using continuous integration to automate this process.

1.3 Objectives

The main objective of this research study is to explore testing and measuring CNN models according to adversarial robustness and how such approaches can be linked with continuous integration tools.

As a case study, I test the existing vehicle type (VType) CNN classification model from the AIT VISION lab. The AIT VISION lab is an interesting hub for computer vision research. The current VType classifier obtains 86% accuracy on its test dataset. It is important to identify the error behaviors in this VType classifier and other datasets.

To achieve these main objectives, I perform the following specific steps:

- Train and test CNN models (GoogLeNet) using PyTorch framework.
- Identify methods to generate error-inducing inputs for classification models.
- Deploy a case study system to a continuous integration (CI) framework.
- Evaluate models by automatically measuring the robustness of CNN models.

By using automatic ML models testing in a CI environment and providing feedback to the training and model development phase, we should be able to improve ML models' robustness in the production environment.

1.4 Organization of the report

I organize the rest of the report as follows.

- In Chapter 2, I provide a literature review.
- In Chapter 3, I propose my methodology.
- In Chapter 4, I provide my results.
- Finally, in Chapter 5, I conclude.

CHAPTER 2

LITERATURE REVIEW

This chapter reviews previous work on testing and measurement of ML model performance.

2.1 Convolutional neural network models for image classification

Today, convolutional neural network (CNN) models are used for many high-end real-world applications of image classification. For example, consider the ImageNet dataset challenge introduced in 2010. This challenge was initially tackled with image analysis algorithms such as scale-invariant feature transform (SIFT (Lowe, 2004)) with low success rates. Earlier, LeCun et al. (1998) proposed a layered CNN model (see Figure 2.2). Krizhevsky et al. (2012) were inspired by LeCun’s architecture and proposed a CNN model named AlexNet, which achieved a top-5 error-rate of 15.3% compared to that of the previous SIFT-type models, which had a previous best accuracy of 26.2%. This drew a great deal of attention in the community. AlexNet is a relatively simple architecture with five layers of consecutive convolutional filters, max-pooling, and three fully-connected layers (see Figure 2.3). After AlexNet, different CNN models have achieved even higher accuracy. One such CNN model utilizes a combination of different “inception” modules. The inception model was much deeper than previously proposed architectures. Szegedy et al. (2015) propose a CNN model named GoogLeNet (see Figure 2.1) based on this architecture. GoogLeNet is a 22-layer CNN model with deep convolutional layers. As GoogLeNet is a combination of inception modules, each module is composed of 1x1, 3x3, and 5x5 convolution layers and a 3x3 max-pool layer to increase sparsity in the model and obtain different types of patterns. Feature maps are concatenated and analyzed by later inception modules. GoogLeNet obtained a 6.7% error rate in the 2014 ImageNet challenge, lower than the next best competitor, VGG16. The tiny difference was perhaps due to the presence of three large fully-connected layers in the VGG architecture. Table 2.1 compares the different CNN models and their top-5 error-rates year-wise according to ImageNet dataset challenges. In summary, there are now several very robust image recognition models, and they have been improving unanimously in accuracy year by year. Now, measuring performance against adversarial examples as opposed to randomly selected examples may be as important as overall accuracy on a training or test set.

2.2 Overview of adversarial examples

An adversarial example is a test input that is obtained by perturbing training data in such way that the model misclassifies the example at test time. In the case of image-based inputs, the perturbation is a carefully constructed vector of pixel changes. The input is usually pixel space (RGB), so the changes will be applied to integer representations of the pixels in the range 0 to 255. Many of the proposed attacks use a distance metric to quantify and bound the perturbation used to generated samples (Carlini & Wagner, 2017). L_p norms such as L_0 , L_2 , and L_∞ are the most widely-used distance metrics. The L_p distance is written as $\|\vec{x} - \vec{x}'\|_p$. Following are the common distance metrics in detail.

- L_0 distance corresponds to the total number of image pixels that differ in value between the two

Figure 2.1: Inception module concept. (a) Inception module, naïve version. (b) Inception module with dimension reductions. Reprinted from Szegedy et al. (2015)

Figure 2.2: Classic CNN architecture: LeNet5. Reprinted from LeCun et al. (1998).

Figure 2.3: AlexNet architecture. Reprinted from Krizhevsky et al. (2012).

Table 2.1: CNNs introduced in recent years with performance in the ImageNet competition.

CNN model	Developed by	Position	Top-5 error-rate	No. of params
LeNet	LeCun et al. (1998)	NA	NA	60 thousand
AlexNet	Krizhevsky et al. (2012)	1st	15.3%	60 million
ZFNet	Zeiler and Fergus (2014)	1st	14.8%	NA
GoogLeNet	Szegedy et al. (2015)	1st	6.67%	4 million
VGG Net	Simonyan and Zisserman (2015)	2nd	7.3%	138 million
ResNet	He et al. (2016)	1st	3.6%	NA

vectors.

- L_2 distance measures the Euclidean (root-mean-square) distance between the two images. This metric was used by Szegedy et al. (2014).
- L_∞ distance measures the maximum change to any of the pixels.

Szegedy et al. (2014) generate adversarial examples while utilizing the box-constrained L-BFGS optimization algorithm. This method aims at solving a box-constrained optimization problem to find distortions that do not perceptually modify the image bound on a box-shaped regulation of pixel space. The authors use the AlexNet model. They observe that the model is not robust to tiny pixel changes in the image (see Figure 2.4). If $\vec{x}$ is an input image, their method finds a different image $\vec{x}'$ similar to $\vec{x}$ under the L_2 distance that is misclassified by the model. The authors implement the following objective function for the box-constrained L-BFGS:

$$\begin{aligned} &\text{minimize } c \cdot \|\vec{x} - \vec{x}'\|_2^2 + \text{loss}_{F,l}(\vec{x}') \\ &\text{such that } x' \in [0, 1]^n \end{aligned}$$

where $\text{loss}_{F,l}$ is a function mapping an image to a positive real number.

In the meantime, Goodfellow et al. (2015) developed an attack method to identify adversarial examples for an existing model. See Figure 2.5 for more clarity. On the left is a panda that is not and modified recognized as a panda. The model assigns a 60% probability of this image being a panda. The middle image looks like noise to the human eye. But actually, it is a carefully-computed function of the parameters of the network. To cause the CNN model to make a mistake, we multiply the structured attack perturbation by a very small coefficient (e.g. $\epsilon = 0.007$) and add it to the original image (the panda in this case). We see that the CNN model has much more confidence in its incorrect prediction that the image on the right is a gibbon, than it had for the original being a panda. It believes that the image on the right is a gibbon with 99% probability (Goodfellow, Shlens, & Szegedy, 2015).

This method is a single-step process that generates an adversarial example by taking the sign of the gradient, which the authors name the fast gradient sign method (FGSM). Specifically, to minimize the loss function, this method uses the gradient of the loss function to determine which change in direction of the pixel inten-

Figure 2.4: Adversarial examples generated to be misclassified by AlexNet. Reprinted from Szegedy et al. (2014).

Figure 2.5: Adding a perturbation to a panda image that leads it to be identified as a gibbon by a CNN model. Reprinted from Goodfellow et al. (2015).

sity (increase or decrease) leads to misclassifying the image. Let $\vec{x}$ be an original input image, let $\vec{x}_{adv}$ be an adversarial image, J the cost function, and y_{orig} the original label for the input $\vec{x}$. The adversarial example $\vec{x}_{adv}$ is computed as

$$\vec{x}_{adv} = \vec{x} + \epsilon \cdot \text{sign} \left[\nabla_x J(\theta, \vec{x}, y_{orig}) \right]. \quad (\text{Equation 2.1})$$

The FGSM method has two key differences from the Szegedy et al. (2014) method: it uses a L_∞ distance metric, and it runs very fast. However, FGSM does not produce minimal adversarial perturbations.

The above discussed attacks were the early methods for the adversarial example generation process. They are the base for more attack methods. The general and basic process for resisting those examples, thus improving classifier robustness, is to train the model with adversarial examples in the first place. This feedback has not yet been fully automated.

2.3 Optimization perspective on adversarial training

In the previous section, I explained two basic approaches to generate adversarial examples. Box-constrained L-BFGS and FGSM are the two basic attack methods. These attack methods are attractive due to the fact that they only induce small pixel changes. One way to resist such attacks is to train the model with new synthetic data points. This approach is called adversarial training. In the traditional model training process, we minimize a loss function indicating the inaccuracy of the predictions on the training set. Madry et al. (2018) propose interesting approaches to adversarial training from the point of view of optimization.

Let $h_\theta : \mathcal{X} \rightarrow \mathbb{R}^k$ be a hypothesis function mapping from an input space X to the output space, which is the set of k -dimensional vectors, where k is the number of classes, and θ represents all the parameters of the model. Let $\ell : \mathbb{R}^k \times \mathbb{Z}_+ \rightarrow \mathbb{R}_+$ be a loss function mapping model predictions and true labels to a non-negative number. Let $x \in \mathcal{X}$ be an input and $y \in \mathbb{Z}$ be a true class (label). The most common loss used in ML is the cross entropy loss, which can be defined as

$$\ell(h_\theta(x), y) = \log \left(\sum_{j=1}^k \exp(h_\theta(x)_j) \right) - h_\theta(x)_y. \quad (\text{Equation 2.2})$$

Now, let us look at the empirical risk of a model. The risk of a model is its expected loss under the true distribution of test samples. Let $\mathcal{D}$ be the true distribution. Let d be a finite set of samples drawn i.i.d. (independent and identically distributed), i.e., $d = \{(x_i, y_i) \sim \mathcal{D}\}, i = 1, \dots, m$. The risk is given by

$$R(h_\theta) = \mathbb{E}_{(x,y) \sim \mathcal{D}} [\ell(h_\theta(x), y)]. \quad (\text{Equation 2.3})$$

In contrast, the *empirical risk* is

$$\hat{R}(h_\theta, d) = \frac{1}{|d|} \sum_{(x,y) \in d} \ell(h_\theta(x), y). \quad (\text{Equation 2.4})$$

As mentioned earlier, the traditional process of training a model is to minimize the empirical loss on a some training set from d (d_{train}), so the training process can be seen as

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \hat{R}(h_\theta, d_{\text{train}}). \quad (\text{Equation 2.5})$$

Again consider generating adversarial examples. Here, instead of minimizing the loss by optimizing network parameters in the traditional way defined above, we can see generation of an adversarial example as an adjustment to the input image in the allowable set of perturbation set Δ , where $\Delta = \{\delta : \|\delta\|_\infty \leq \epsilon\}$

$$\hat{x}^*, y^* = \underset{\hat{x}, y \in \Delta}{\operatorname{argmax}} \ell(h_\theta(\hat{x}), y), \quad (\text{Equation 2.6})$$

where $\hat{x}$ is an adversarial example for class y . We might then consider

$$\max_{\hat{x}, y} \ell(h_\theta(\hat{x}), y) \quad (\text{Equation 2.7})$$

as the *adversarial risk*. The *adversarial risk* is similar to the ordinary risk as an expectation over $\mathcal{D}$, with the maximization limited to a region around a sample point. Consider the $\Delta(x)$ to be the allowable set of perturbation δ for adversarial examples. Then the *adversarial risk* can be defined as

$$R_{\text{adv}}(h_\theta) = \mathbb{E}_{(x,y) \sim \mathcal{D}} \left[\max_{\delta \in \Delta(x)} \ell(h_\theta(x + \delta)), y \right]. \quad (\text{Equation 2.8})$$

The *empirical adversarial risk* is then given by

$$\hat{R}_{\text{adv}}(h_\theta, d) = \frac{1}{|d|} \sum_{(x,y) \in d} \max_{\delta \in \Delta(x)} \ell(h_\theta(x + \delta)), y). \quad (\text{Equation 2.9})$$

Training classifiers with adversarial examples may enable then to resist adversarial attacks. Madry et al. (2018) formulate adversarial robust training as the optimization problem

$$\min_{\theta} \hat{R}_{\text{adv}}(h_\theta, d_{\text{train}}) \equiv \min_{\theta} \frac{1}{|d_{\text{train}}|} \sum_{(x,y) \in d_{\text{train}}} \max_{\delta \in \Delta(x)} \ell(h_\theta(x + \delta)), y). \quad (\text{Equation 2.10})$$

The authors refer to this formulation as the min-max or robust optimization formulation. The problem can be solved, for example, by stochastic gradient descent (SGD) over θ . The gradient of the inner maximized expression is the key. Gradient descent on the empirical adversarial risk objective, function with a minimum number of 30 iterations, can be solved by the following steps:

- For each $x, y \in B$, where B is a mini-batch set, compute the adversarial perturbation (inner maximization problem)

$$\delta^*(s) = \operatorname{argmax}_{\delta \in \Delta(x)} \ell(h_\theta(x + \delta), y). \quad (\text{Equation 2.11})$$

- Next, compute the gradient of the empirical *adversarial risk* and then update θ (with α as a step size)

$$\theta \leftarrow \theta - \frac{\alpha}{|B|} \sum_{(x,y) \in B} \nabla_\theta \ell(h_\theta(x + \delta^*(x)), y). \quad (\text{Equation 2.12})$$

2.4 Verification of ML models

Adversarial examples show how sensitive a model's performance may be to a small change in the pixel space of an image. Several gradient search techniques have been proposed to challenge this sensitivity. DeepXplore (Pei, Cao, Yang, & Jana, 2017) is a new white-box approach to observe models' misbehaving nature on datasets automatically with full view of model's internal architecture. Gradient search gives guidance in verifying the model's performance.

Pei et al. (2017) define a *coverage* metric to find the activation of neurons in the network, specifically to find which layers cause different behavior in the model. This coverage metric is designed similarly to code coverage in the traditional software systems. Code coverage in traditional systems is a standard empirical metric for measuring what code is expected by a test suite. Let the neurons in a deep neural network (DNN) be represented by the set $N = \{n_1, n_2, \dots\}$, let test inputs be given by the set $T = \{x_1, x_2, \dots\}$, and let $out(n, \vec{x})$ be a function with a test input $\vec{x}$ that returns the output value of neuron n in the DNN. Let t represent a threshold for considering a neuron to be activated. For example, see Figure 2.6. The *neuron coverage* (with a threshold say, 0), the ratio between the number of neurons activated to the total number of neurons in the example, 0.625, is formulated as

$$NCov(T, \vec{x}) = \frac{|\{n \mid out(n, \vec{x}) > t\}|}{|N|}. \quad (\text{Equation 2.13})$$

In software testing, differential testing techniques have been widely applied to detect bugs in logic without manual specifications. In the same way, authors find that two or more models with similar functionality can be used as cross-referencing oracles to identify error cases automatically. For example, in the case of self-driving cars, suppose that for a particular input, one decides to turn left while others decides to turn right. This is the other main work of DeepXplore to find the must difference-inducing inputs, which can be found by comparing two or more similar models' output for the same task. The automatically generated test input pixels are in the range of 0 and 255. See Figure 2.8 for domain-specific constraints such as illumination conditions and occlusions as in the real physical world. Gradient-based search, is a guided search along the gradient aiming to find differences that lie between the decision boundaries of two similar DNN models (see Figure 2.7). To achieve above-mentioned outcome, the authors solve a joint optimization problem based on gradient search. Now, by keeping the input as variable and weight parameter as a constant, the gradient of the output of each neuron can be computed. Let $F_i(x)[c]$ and $F_j(x)[c]$ be two objective functions for different class outputs i and j . Let λ_2 be a hyperparameter to balance the objectives of the joint-optimization process. The combined objective is to minimize

Figure 2.6: A simplified neural network that takes an image as input and classifies it into two different classes: cars and faces. Reprinted from Pei et al. (2017).

$$J(\mathbf{x}) = (\sum_{i \neq j} F_i(\mathbf{x})[c] - \lambda_1 F_j(\mathbf{x})[c]) + \lambda_2 \cdot f_n(\mathbf{x}.) \quad (\text{Equation 2.14})$$

This is a new approach compared to prior literature. The adversarial example generation process deals with a model's robustness to specific perturbations while carefully computing *deltas* in the pixel space. DeepXplore, on the other hand, considers a model's internal architecture to identify its neurons' sensitivity to perturbations while generating them automatically. To make the best use of these technologies, we need an appropriate automated framework for ensuring model performance in the production environment is resistant to such white-box attacks.

Figure 2.7: Gradient ascent process to find difference-inducing inputs. Reprinted from Pei et al. (2017).

Figure 2.8: Multiple DNNs’ performance with domain-specific constraints: illumination conditions and occlusions by single and multiple tiny rectangles. First column shows images from the Udacity self driving car challenge dataset. The second column shows images have the MNIST dataset, and the third column shows images have the ImageNet dataset. Pei et al. (2017).

CHAPTER 3

METHODOLOGY

This chapter provides information about generating the adversarial examples and linkage of continuous integration framework.

3.1 System Overview

I generate adversarial example for the GoogLeNet image classification model and train them to resist such attacks. Similarly, I automate the testing and training process by developing a continuous integration (CI) framework where we can visualize model performance metrics. I use the PyTorch deep learning framework to perform the above-mentioned tasks.

3.2 Adversarial Examples

In the AIT VISION lab, the GoogLeNet model is used for vehicle type image classification. The model identifies buses, cars, vans, trucks, and pickups on a test dataset with a reported accuracy of 86%. Now, in order to fool the model (force it to identify a car instead of a van, for example), I apply the FGSM attack with different pixel space adversaries. We can find such adversaries by calculating the gradients of the loss function with respect to the image.

As discussed in Chapter 2, I use the fast gradient sign method (FGSM) to generate adversarial examples (see flow chart in Figure 3.2). They are generated with Algorithm 1.

Algorithm 1 should be included in the model training train loop. Below are the steps to generate adversarial examples automatically:

- Initiate *DataLoaders* with the PyTorch framework.
- Create mini-batches from the training dataset.
- Pass each mini-batch to Algorithm 1.
- Take the generated adversarial examples and include them in the original training set.
- Train the model as usual with respective parameters.

Figure 3.1: Vehicle type dataset. (a) Pickup. (b) Bus. (c) Van. (d) Sedan. (e) Truck.

Algorithm 1 FGSM Algorithm

Input: $\vec{x}$: original image, $\epsilon = 0.30$

Output: $\vec{x}_{adv}$: adversarial image

if ϵ is defined **then**

 Compute gradient of the loss function with respect to input ($\vec{x}$).

 Collect the element-wise sign of the data gradient.

 Create the perturbed image ($\vec{x}_{adv}$) by adjusting each pixel of the input image $\vec{x}$.

 Clip $\vec{x}_{adv}$ to the $[0, 1]$ valid range.

else

 Make no changes to the input image ($\vec{x}$).

end if

3.3 Continuous Integration framework model

Continuous Integration (CI) is a process to verify early integration of the tested tasks. This process allows us to identify errors as early as possible. I use the Jenkins open source software to develop a method to perform CI mechanism of adversarial robustness. Jenkins is an open-source automation server written in Java. It enables us to create custom plugins for particular tasks or environments. It can perform different build tests with respect to the build system. Tests must involve executing a shell script, archiving a build result, and running software tests.

In Figure 3.3, the stages in a Jenkins pipeline are defined. A typical pipeline has four stages: build, test, deploy, and release. All such pipeline stages are defined in a single file called the *Jenkinsfile*. In the build stage, the model is tested with adversarial perturbations. Next, the test stage follows by testing the updated model with new images. Then the model is deployed to servers and can be released to the production environment.

Below are the steps that I followed to build a Jenkins job.

Figure 3.2: Overview of adversarial testing process.

Figure 3.3: Stages in a Jenkins Pipeline.

- First, create a *free style project*.
- After the *workspace* is created, go to the *build* and select *shell execution*.
- Using the *shell*, give path to the local repository and run the Python script.
- Make sure *Plot* plugin is installed. Next, go to the *post build actions* and select *plot build data*.
- If the *shell* is executed, it will give a *csv* format file, which will be given as input to the *plot*. Finally, when the job is *built*, we can see the plots.

CHAPTER 4

EXPERIMENTAL RESULTS

This chapter discusses the final results obtained during my research study on training an image classifier to be adversarially robust.

4.1 Standard classifier performance

I trained a GoogLeNet image classifier on the AVL vehicle type dataset to classify images containing buses, pickups, sedans, trucks, and vans. The model achieve a validation accuracy of 90% on the validation dataset. For fast and direct experimentation process, I used the PyTorch framework.

The standard classifier has very low performance when it is attacked with the fast gradient sign method (FGSM). I attacked the validation dataset with a perturbation range $\epsilon = [0, 0.3]$ (pixel intensities are normalized to the $[0, 1]$ range) and steps in ϵ of 0.05 (see Figure 4.1). For every ϵ , the classifier is fooled with high confidence (see Figure 4.2).

4.2 Adversarially trained classifier performance

The process of attacking a classifier helps us understand its behavior. In order to resist successful attacks, we can train the model with batch-wise adversarial perturbations of the dataset. This process will in principle enable the model to defend such attacks.

By using the FGSM attack method, each batch in the dataset is subjected to a perturbation bound with a maximum magnitude of $\epsilon = 0.3$. These examples are included in the training batch, and the classifier is trained with several such batches over the entire training dataset. After this process I achieved a validation accuracy of 83%. Although the validation accuracy is slightly lower, when the adversarial-trained classifier is again attacked with the FGSM method, there is only a slight decrease in accuracy. See Figure 4.3 and 4.4 for accuracy visualizations under the attacks.

4.3 Normalization bounds

Many ImageNet-based models are trained with default normalization bounds such as, $mean = [0.485, 0.456, 0.406]$ and standard deviation, $std = [0.229, 0.224, 0.225]$. These scalings must be preserved during the adversarial input. The perturbation samples should scale correctly so that for $\epsilon = 0$, we should get the same validation accuracy observed during the training, otherwise we may observe very low accuracy.

Figure 4.1: Plot for perturbation bounds of $\epsilon = [0, 0.3]$ versus validation accuracy for a GoogLeNet model trained without adversarial examples.

4.4 Jenkins job visualizations

I use *plot* plugin to visualize FGSM attack performance on both standard and adversarially trained classifiers. Using such visualizations, a software engineering team can see the performance of the classifier and average change in accuracy over time. Figures 4.5 and 4.6 show screenshots of the Jenkins workspace enabled with FGSM attack performance plotting.

Figure 4.2: Visualization of the FGSM attack on random samples from the test dataset for standard trained model. Note that adversarial examples that visually resemble the original class are misclassified, indicating a poor model successful attack.

Figure 4.3: Plot of validation accuracy versus perturbation bound $\epsilon = [0, 0.3]$ for an adversarially-trained GoogLeNet model.

Figure 4.4: Visualization of the FGSM attack on random samples from the test dataset for adversarially trained model. Note that fewer of the successful adversarial examples visually resemble the original class.

Figure 4.5: Visualization of a Jenkins job showing performance against FGSM attacks over two builds for the standard classifier.

Figure 4.6: Visualization of a Jenkins job showing performance against FGSM attacks over two builds for the adversarially trained classifier

CHAPTER 5

CONCLUSION AND FUTURE SCOPE

This chapter discusses the conclusion and future scope of my research study.

5.1 Conclusion

From the obtained results, we can see that the process of attacking and adversarially training a classifier shows loopholes in the classifier. For the perturbation range $\epsilon = [0, 0.3]$, the accuracy quickly decreases for larger ϵ values. Such variations in the validation accuracy in the different perturbation ranges show that even if a adversarially training a classifier still has caveats. A CI framework can help us visualize such changes as a project goes on. Currently, this approach is used for laboratory testing with single adversarial attack inference.

5.2 Future scope

The adversarial attacks and training developed in this study can be further improved and scaled to larger deployments in production environments. Also, more state-of-the-art adversarial attack methods can be included, and such behaviour can be studied for various classification tasks, such as natural language processing, video and audio processing, and other tasks such as object detection.

REFERENCES

- Carlini, N., & Wagner, D. (2017). Towards evaluating the robustness of neural networks. In *2017 IEEE symposium on security and privacy (sp)* (pp. 39–57).
- Goodfellow, I., Shlens, J., & Szegedy, C. (2015). Explaining and harnessing adversarial examples. In *Proceedings of the international conference on learning representations (ICLR)*.
- He, K., Zhang, X., Ren, S., & Sun, J. (2016, June). Deep residual learning for image recognition. In *The IEEE conference on computer vision and pattern recognition (CVPR)*.
- Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th international conference on neural information processing systems - volume 1* (pp. 1097–1105). USA: Curran Associates Inc.
- LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278–2324.
- Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision (IJCV)*, 60(2), 91–110.
- Madry, A., Makelov, A., Schmidt, L., Tsipras, D., & Vladu, A. (2018). Towards deep learning models resistant to adversarial attacks. In *International conference on learning representations (ICLR)*.
- Pei, K., Cao, Y., Yang, J., & Jana, S. (2017). Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th symposium on operating systems principles (SOSP)* (pp. 1–18).
- Simonyan, K., & Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. *International Conference on Learning Representations (ICLR)*.
- Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., et al. (2015, June). Going deeper with convolutions. In *The IEEE conference on computer vision and pattern recognition (CVPR)*.
- Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., et al. (2014). Intriguing properties of neural networks. *International Conference on Learning Representations (ICLR)*.
- Zeiler, M. D., & Fergus, R. (2014). Visualizing and understanding convolutional networks. In *The european conference on computer vision (ECCV)* (pp. 818–833). Springer International Publishing.

APPENDIX A

Python code for FGSM attack

```
# FGSM attack script for a trained classifier.
# Used Nathan Inkawhich (2018) script, whcih was
# originally written for 1-channel models. I modified
# the same script to work with 3-channel models.
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, models, transforms
import numpy as np
import matplotlib.pyplot as plt
import os
import pandas as pd
epsilons = [0, .05, .1, .15, .2, .25, .3]
pretrained_model = "path_to_model_weights"
use_cuda=True
Net = models.googlenet(aux_logits=False, num_classes=5)
# Main variable for loading dataset batch-wise. Make sure to set
batch_size=1 in this case.
test_loader = torch.utils.data.DataLoader(
    datasets.ImageFolder(os.path.join('path_to_validation_dataset'
    ), transform=transforms.Compose([
        transforms.Resize(225),
        transforms.RandomCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
        0.225])
    ])),
    batch_size=1, shuffle=True, num_workers=4)
print("CUDA-Available:",torch.cuda.is_available())
device = torch.device("cuda:1" if (use_cuda and torch.cuda.is_available
    ()) else "cpu")
model = Net.to(device)
# Switch between devices
model.load_state_dict(torch.load(pretrained_model, map_location='cpu'))
# set model to evaluation stage
model.eval()
def fgsm_attack(image, epsilon, data_grad):
    # Collect the element-wise sign of the data gradient
    sign_data_grad = data_grad.sign()
    # Create the perturbed image by adjusting each pixel of the input
    image
    perturbed_image = image + epsilon*sign_data_grad
```

```

# Adding clipping to maintain [0,1] range, Check for different
    scaling factors
perturbed_image = torch.clamp(perturbed_image, -2.117, 2.64)
# Return the perturbed image
return perturbed_image
def test( model, device, test_loader, epsilon ):
    # Accuracy counter
    correct = 0
    adv_examples = []
    # Loop over all examples in test set
    for data, target in test_loader:
        # Send the data and label to the device
        data, target = data.to(device), target.to(device)
        # Set requires_grad attribute of tensor. Important for Attack.
        data.requires_grad = True
        # Forward pass the data through the model
        output = model(data)
        init_pred = output.max(1, keepdim=True)[1] # get the index of the
            max log-probability
        # If the initial prediction is wrong, dont bother attacking, just
            move on
        if init_pred.item() != target.item():
            continue
        # Calculate the loss
        loss = criterion(output, target)
        # Zero all existing gradients
        model.zero_grad()
        # Calculate gradients of model in backward pass
        loss.backward()
        # Collect datagrad
        data_grad = data.grad.data
        # Call FGSM Attack
        perturbed_data = fgsm_attack(data, epsilon, data_grad)
        # Re-classify the perturbed image
        output = model(perturbed_data)
        # Check for success
        final_pred = output.max(1, keepdim=True)[1] # get the index of
            the max log-probability
        if final_pred.item() == target.item():
            correct += 1
            # case for saving 0 epsilon examples
            if (epsilon == 0) and (len(adv_examples) < 5):
                adv_ex = perturbed_data.squeeze().detach().cpu().numpy()
                adv_examples.append( (init_pred.item(), final_pred.item(),
                    adv_ex) )
        else:
            # Save some adv examples for visualization later
            if len(adv_examples) < 5:

```

```

        adv_ex = perturbed_data.squeeze().detach().cpu().numpy()
        adv_examples.append( (init_pred.item(), final_pred.item(),
                               adv_ex) )
    # Calculate final accuracy for each epsilon
    final_acc = correct/float(len(test_loader))
    print("Epsilon:{}\tTest_Accuracy={} / {} = {}".format(epsilon, correct,
        len(test_loader), final_acc))
    # Return the accuracy and an adversarial example
    return final_acc, adv_examples
criterion = nn.CrossEntropyLoss()
accuracies = []
examples = []
# Run test for each epsilon
for eps in epsilons:
    acc, ex = test(model, device, test_loader, eps)
    accuracies.append(acc)
    examples.append(ex)
#Export acc, ex values to the csv for Jenkins Plot plugin.
pd.DataFrame(acc,ex).to_csv("results_acc.csv", header=None, index=None)
plt.figure(figsize=(5,5))
plt.plot(epsilons, accuracies, "*-")
plt.yticks(np.arange(0, 1.1, step=0.1))
plt.xticks(np.arange(0, .35, step=0.05))
plt.title("Accuracy_vs_Epsilon")
plt.xlabel("Epsilon")
plt.ylabel("Accuracy")
plt.show()
cnt = 0
plt.figure(figsize=(8,10))
for i in range(len(epsilons)):
    for j in range(len(examples[i])):
        cnt += 1
        plt.subplot(len(epsilons),len(examples[0]),cnt)
        plt.xticks([], [])
        plt.yticks([], [])
        if j == 0:
            plt.ylabel("Eps:{}".format(epsilons[i]), fontsize=14)
        orig,adv,ex = examples[i][j]
        map = {0: 'bus', 1: 'pickup', 2: 'sedan', 3: 'truck', 4: 'van'}
        plt.title("{}->{}".format(map[orig], map[adv]))
        ex = ex.transpose((1, 2, 0))
        plt.imshow(ex)

```