MEng Project Report

Towards Robust Traffic Signs Classifier in Autonomous Vehicles

Qiaoyu Li

8799712

Supervisor: Dr. Ali Khanafer

Department of Mechanical Engineering

University of Ottawa

Master of Engineering

Faculty of Engineering November 2017

Abstract

Traffic signs classification systems based on Convolutional Neural Networks (CNNs) have shown vulnerability towards adversarial attacks that inject small-magnitude, unnoticeable perturbations to the signs. This project presents a robust traffic signs classifier that is able to guard against such attacks. Before building our model, we discuss various existing adversarial algorithms and defense methods based on adversarial training. Our model is based on LeNet and is trained on the German Traffic Sign Recognition Benchmark dataset. Based on the adversarial training results in our experiments, we increase the capacity and number of units of the original model to increase robustness. This model is tested on both pure and adversarial sets and shows better results than the original network. Although the new model does not completely eliminate the impact of adversarial attacks, it provides a potential direction for building robust CNNs.

Table of contents

List of figures

List of tables

1 Introduction

1.1 Problem Description

Autonomous driving has attracted much attention in both traditional car manufactures and IT companies in recent years. Companies such as Google, Uber, Audi, BMW have already tested their self-driving cars on the road. Traffic signs detection is an essential components of the vision capability in autonomous vehicles. A precise detection system is the basis of self-driving development. Similar to most image classification systems, traffic signs classification are also largely dependent on convoluntional neural networks (CNNs). This algorithm has gained plenty of traction due to its enormous success in image processing tasks.

However, CNNs have been proven to be vulnerable to adversarial attacks [29, 7, 21, 23, 20, 19]. An image that is perturbed by small-magnitude noise signal could be misclassified by the CNN, with high confidence. These perturbed images are called *adversarial examples*. This “cheap" attack may cause severe accidents if an attacker can cause the car to ignore a stop sign by simply drawing on the sign with a marker and causing it to be recognized as a speed limit sign [5]. This realization should arouse more attention to the security of autonomous cars before they appear on the road. Currently, although various attacks have been constructed, it is still not clear whether a common CNN is able to defend against these attacks hyperparameters tuning or structure alternation. Thus, it is important to test the defense ability of CNNs and improve the robustness of current CNNs by modifying structure and hyperparameters.

1.2 Contributions

In this work, we provide insights on existing methods for crafting adversarial examples and defending against them. We adopt a 3-stage CNN architecture as the base model for our experiments. Through adversarial training, we construct a model with improved robustness against adversarial attacks. Further, we analyze the impact of varying the structure and hyperparameters of the base model on the robustness and propose a more robust network. Finally, we identify potential future research directions.

1.3 Structure of the Report

The report is organized as follows:

- Chapter 2 describes the background and related works. A brief introduction of neural networks and CNNs is given. Existing attack approaches and training strategies are introduced in this chapter.
- Chapter 3 presents the methodology and the design of our traffic signs classifier in detail.
- Chapter 4 describes the experiments on the adversarial examples dataset using different training methods and model settings. The analysis and comparisons of the results are also shown. In addition, a more robust network is proposed according to our experiments.
- Chapter 5 gives the conclusion of this report and future work.

2 Literature Review

In this chapter, we will provide a brief introduction of general neural networks before we explain CNNs which are widely used for image classification. Next, we will describe the methods of generating adversarial examples and state-of-the-art defense methods against these examples.

2.1 Neural Networks

Neural networks have been studied since 1950s and are widely used to implement deep learning. A neural network is a hierarchical model inspired by the structure of the brain, which consists of neurons. It originates from a perceptron where neurons are used to weight up the inputs and the summation of the weighted values is compared to a given threshold to produce a binary value (shown in Fig 2.1). The mathematical expression of the perceptron is given in (2.1).

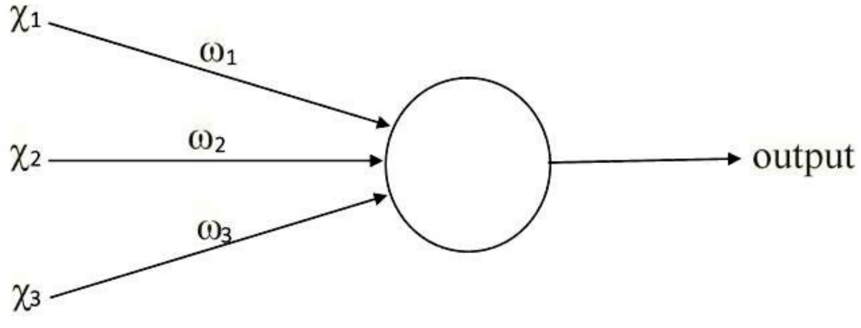


Fig. 2.1 Structure of perceptron

$$output = \begin{cases} 0, & if \sum \omega_j X_j > threshold \\ 1, & if \sum \omega_j X_j \leq threshold \end{cases} \tag{2.1}$$

where ω_j is the weight assigned to input X_j.

Neural networks typically contain multiple layers and each layer includes several percetrons as shown in Fig. 2.2. The first layer is the input layer, the last layer is called the output layer, and those in between are referred to as the hidden layers. Information passes through the connections of neurons between adjacent layers, where neurons are fully connected in general. This means every pair of neurons have a connection between any two adjacent layers.

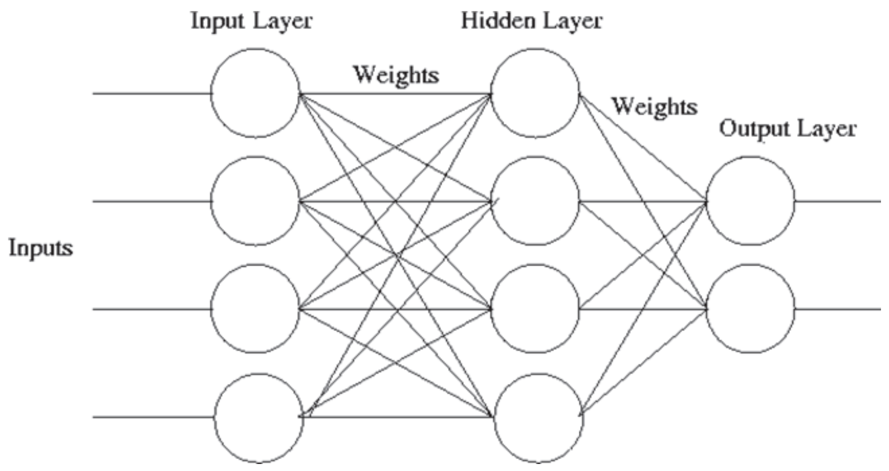


Fig. 2.2 Structure of a 3-layer fully connected neural network. This figure only shows one hidden layer. In general, there are many hidden layers.

By integrating non-linear perceptrons and fully connected layers, a neural network is able to theoretically represent a highly non-linear function. Thus, the neural networks have a strong modeling capacity for complex data [16]. The remaining task is to obtain proper parameters (weights) to fit the given data. This process is called *training* where a neural network learns from the errors between the predictions and true values by updating its weights. Errors are defined through a cost function. In an effective training, the errors will gradually decrease and the network will provide an accurate approximation of the data.

2.2 Convolutional Neural Networks

The CNNs, first proposed by LeCun [17], are powerful neural networks for processing image data which can be regarded as a 2D grid of pixels. The most significant contribution of CNNs is the introduction of the convolutional operation and weight sharing. The convolutional operation replaces the full connection of the neural network with local connections, and the weight of the connections between the different layers is shared [2]. This allows the neural network to be utilized in computer vision tasks with large image datasets. Since CNNs were introduced, many improved CNNs such as AlexNet [14], GoogleNet [28] have shown remarkable achievements on processing many large datasets such as the ImageNet [4] which contains more than ten million images.

The basic structure of a CNN consists of the input layer, the convolutional layer, the pooling layer, the activation layer, and the fully connected layer. Below, we explain these elements in more detail.

Input layer

The input layer is responsible for pre-processing the images in the given dataset so that these images can be suitable for feature extraction. In general, the pre-processing includes multiple operations such as resizing, augmentation, and enhancement. In the image classification, the input layer processes the raw pixel values of the input images. For example, an image containing pixel values in the range of $[0,255]$ can be processed to the range of $[0,1]$.

Convolution layer

The convolution layer is mainly used to extract the spatial relationships between input features. These features are detected by iterating a small matrix of numbers, called *kernel* or *filter*, over grid-like images from the input layer. The distance that the kernel slides over the input during each iteration is referred to as a *stride*. In each iteration, the sum of multiplying the parameters of the kernel and the pixel values of the corresponding input that the kernel overlaps yields an element of the output matrix. The kernel slides over the input according to the stride. When the entire region of the input has been iterated, the output matrix is completely populated. We refer to this output matrix as the *feature map* [6]. Fig. 2.3 displays an example of this process.

It is worth mentioning that the same kernel weights are used on each slice of an input to obtain the features during the process. As a result, the number of parameters is significantly reduced compared with a fully connected layer where a unique weight is assigned to every input. This idea, referred to as *parameter sharing*, is the most essential idea of CNNs.

Pooling layer

After extracting the features from the convolutional layer, the pooling layer is used to reduce the feature dimentionality and the number of parameters in the network. This helps to reduce the degree of overfitting and increase the ability of the network to detect the same objects in different orientations, i.e., it enables the CNN to *generalize* to examples not seen in training.

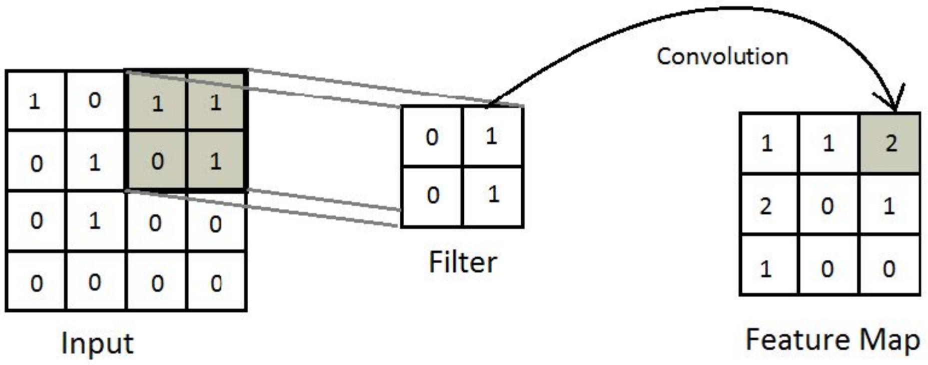


Fig. 2.3 Convolutional layer example with kernel size 2×2 and stride 1. The elements in the input are referred to as *pixel values* or *input features*; the elements in the filter are defined as *weights* which are updated by training; the numbers of kernels are named as *depth* of the kernel which leads to the same depth for the feature map; the elements in feature map are called *extracted features* which represent features that distinguish the image from other images in other classes.

There are various types of pooling such as max-pooling[32] and mean-pooling. Max-pooling is the most popular approach and we will describe it below [6].

Similar to the convolutional layer, a max-pooling layer includes a stride and a pool kernel. The stride is used in the same way as in the convolutional layer, whereas the pool kernel does not contain any weights but is an operation that returns the maximum value of the input the kernel overlaps. Once the entire input has been processed, the matrix having the same depth but smaller width and height as the input will be regarded as the output. Fig. 2.4 describes an example of this process.

Activation layer

The complex features of images necessitate the use of nonlinear factors to represent images, as opposed to linear modeling. This is done at the activation layer. A commonly used activation function is the Rectified Linear Unit (ReLU) [12] which is defined in (2.2).

$$f(z) = max(z, 0) \tag{2.2}$$

where z represents an input value. It performs a threshold operation on each input where any value less than zero is set to zero.

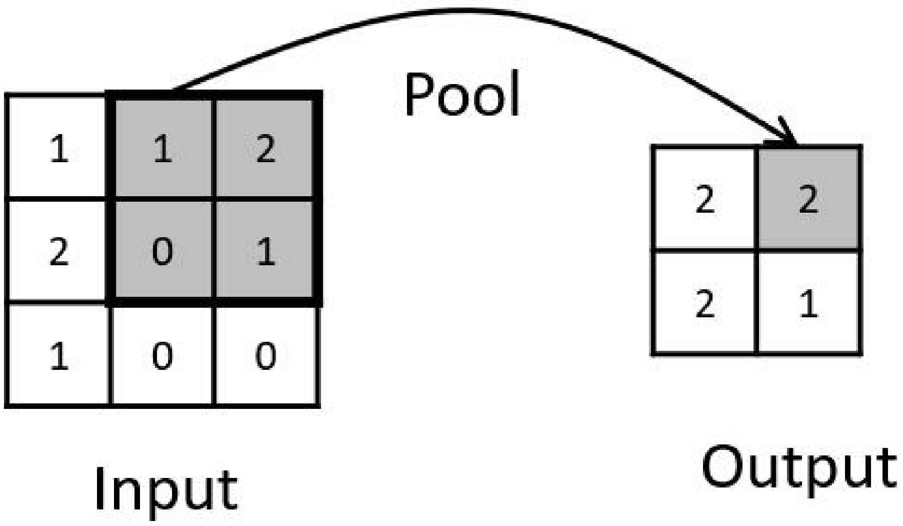


Fig. 2.4 Pooling layer example with kernel size 2×2 and stride 1

Fully connected layer

The fully connected layer is also referred to as inner product layer which is equivalent to a classifier in the network. It is used to calculate the probability of each image corresponding to a certain output category. The softmax function, defined in (2.3), is applied in this work:

$$softmax(z)_i = \frac{exp(z_i)}{\sum_j exp(z_j)} \tag{2.3}$$

where z is a j-dimensional vector. Each real value of the entries of z is transformed to a relative value in the range $[0, 1]$, and the sum of all these relative values in the vector is 1. As a result, $softmax(z)$ can be used as a probability for classification purposes.

2.3 Adversarial Attack

Although many CNNs have shown excellent ability at image classification, they still can be fooled by some perturbed images and cause misclassification. Furthermore, Szegedy *et al.* [29] discussed how small the perturbation needs to be to cause a misclassification. They discovered that in a typical image classification network, even a slightly changed image can make the classifier generate completely different labels for the perturbed image, without an apparent difference between these two images to the human eye. This perturbation precess is referred to as an *adversarial attack*, and the perturbed images as *adversarial examples*.

A variety of methods have been proposed to generate adversarial examples. These methods try to find a small perturbations that are capable of fooling the model and make it output the wrong label. Attacks attempting to find perturbations that cause misclassification to any class are called *untargeted attack*, whereas those seeking to find a perturbation that produces misclassification to a specific class are called *targeted attack*.

The first algorithm for finding adversarial examples was proposed by Szegedy *et al.* [29]. This algorithm, called *Box-constrained L-BFGS*, employs an optimization procedure that minimizes the loss function of an adversarial input for a target class. The method has shown effective attack on a variety of classifiers with different architectures. However, it requires impractical time to repeat the iteration to find adversarial examples with a small perturbation. Hence, this algorithm is not an suitable for performing adversarial training. However, this experiment suggests that the classifiers based on CNNs have not learned the true underlying concepts and primary features of images, even if they have shown excellent performance on some dataset.

To solve the problem of the computational complexity, Goodfellow *et al.* [7] proposed a linear method to generate adversarial examples, which is called *Fast Gradient Sign Method* (FGSM). The perturbation of the original images is crafted by the sign of the gradient of a network's loss with respect to the input sample to that network, which can be obtained through the backpropagation algorithm. This perturbation proposed in this method is generated using the following formula:

$$\tilde{x} = x + \varepsilon sign(\bigtriangledown_x J(\theta, x, y)) \tag{2.4}$$

where x is the original image, $\tilde{x}$ represents adversarial example, ε is a small factor, θ is the parameters of the model, y is the label of the image, and J denotes the loss function. Since FGSM only changes the input slightly in the direction of that sign, adversarial examples can be generated with much lower computational complexity. This attack is not aimed at any specific class, so the class of the perturbed image could be any other one—it is an untargeted attack. It achieves an error rate of 99.9% on the MNIST dataset[15] and 87.15% on CIFAR-10[13], which is an outstanding result for such a simple, cheap algorithm. An example of the perturbed image is displayed in Fig. 2.5.

Another attack method called *DeepFool* proposed by Moosavi-Dezfooli *et. al* [20] is an iterative algorithm which attempts to generate adversarial examples that fool the image classifiers. It is essentially an iterative linearization of the classifiers, which causes minimum perturbation to the original images and sufficiently changes the output labels. This approach also performs an untargeted attack due to its optimization algorithm. Although it has larger computational complexity than FGSM, the perturbation computed by DeepFool is much smaller than that crafted by FGSM, which means that the difference between the adversarial example and its corresponding original image is harder to distinguish.

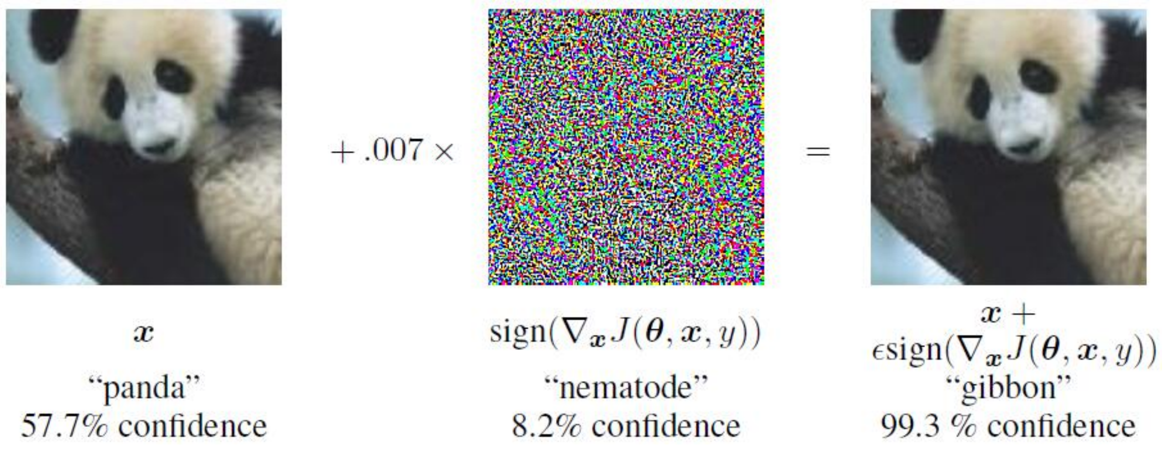

$+ .007 \times$
$=$
$\boldsymbol{x}$
"panda"
57.7% confidence
$\mathrm{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"nematode"
8.2% confidence
$\boldsymbol{x} + \epsilon\mathrm{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"gibbon"
99.3 % confidence

Fig. 2.5 An example of FGSM [7]

Evtimov *et al.* [5] proposed an intriguing attack which generates perturbations to real traffic signs, and they obtained a robust result regardless of a the changing viewpoint caused by distances, lighting, camera angles, and occlusion of the sign. When computing the perturbation according to an optimization-based method, called *Robust Physical Perturbation* (RP2), they introduced a perturbation mask matrix whose dimensions are the same as the size of input to the classifier and a non-printability score which is used to improve the printability of the adversarial perturbation. Furthermore, RP2 is suitable for both untargeted and targeted attack. Some examples from RP2 are plotted in Fig. 2.6.

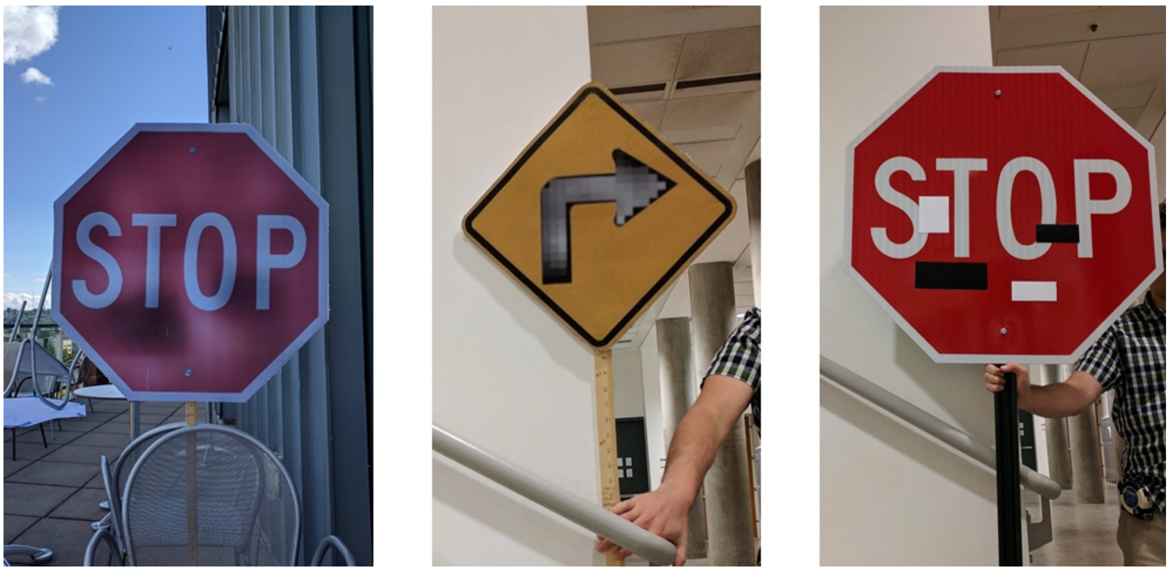


Fig. 2.6 Some examples from RP2 [5]

2.4 Defending against Adversarial Examples

A number of methods have been proposed to prevent neural networks being fooled by adversarial examples. In general, these methods can be split into two categories. First one is the sample filtering, where the network is able to detect the adversarial examples and then pre-process them. Such methods attempt to built a firewall to defend all examples that should not belong to the test dataset of the network. For example, an image contains a pixel value of 300, but the maximum pixel value in this dataset is 255. This kind of detection and filtering may fence some primitive attacks such as the example above, but may not stop more advanced attacks. Many researchers such as Papernot *et. al* [24] and Gu *et. al* [8] have investigated in this area. This work, however, will not discuss more about this topic and will focus on the second category instead.

The second approach is *adversarial training* where the original network is trained to be robust towards the adversarial examples by perturbing the training dataset. This approach attracts most attention of researchers who aim to creating resilient networks. The process entails crafting adversarial examples for a model, and then training the model on those examples with right labels so that the model may learn to recognize the images correctly. Introducing this data provides the relevant information to the model to fit the true classifier better.

However, the training method still needs to be took consideration. Many experiments have been performed to attempt to obtain an effective way for adversarial training. Szegedy *et. al* [29] suggested to apply a random adversarial training subset of which the data is continuously updated by newly crafted adversarial examples and then mix it with the original training dataset all the time. Thus, the network was trained in an alternating style, in which adversarial examples are maintained and updated for each layer separately and then added to the original training dataset. Some regularization methods such as weight decay [3] and dropout [10, 26] are used during the adversarial training to prevent the overfitting. Furthermore, they found that adversarial examples for the higher layers is obviously more important than those on the input or lower layers.

In the work of Goodfellow *et. al* [7], a modified adversarial objective function (shown in Formula 2.5) was proposed as an effective training method.

$$\tilde{J}(\theta,x,y) = \alpha J(\theta,x,y) + (1-\alpha)J(\theta,x+\varepsilon sign(\bigtriangledown_x J(\theta,x,y)) \tag{2.5}$$

This approach means that the adversarial examples are updated by the algorithm continuously, and make them attack the latest version of the model. As a result, the network can be robust to the FGSM attack. It shows that with adversarial training the error rate became to 17.9%

on adversarial examples, while an error rate of 89.4% occurred without adversarial training. Larger model with more units per layer and early stopping are also applied to the training experiments for regularization. Moreover, Goodfellow *et. al* [7] presented a new perspective of the adversarial training, where training on the perturbed dataset was regarded as one of the regularization methods, or rather data augmentation methods. Unlike other data augmentation with transformations such as translations, flipping that might probably occur in the test set, this form of data augmentation provides examples that should not appear in natural test set but help the classifier to discover the true decision function. Using this method to train the network, they reduced the error rate on pure images from 0.94% without adversarial training to 0.84% with adversarial training.

Moosavi-Dezfooli et. al[20] proposed a new evaluation method to calculate the robustness of a classifier f to adversarial attack. The average robustness $\hat{\rho}_{adv}(f)$ is defined by

$$\hat{\rho}_{adv}(f) = \frac{1}{|\mathscr{D}|} \sum_{x \in \mathscr{D}} \frac{\| \hat{r}(x) \|_2}{\| x \|_2} \tag{2.6}$$

where x is an image, $\hat{r}(x)$ denotes the estimated minimum perturbation obtained using DeepFool [20], and $\mathscr{D}$ represents the test set. After crafting the adversarial examples, they improved the robustness of the network by performing 5 additional epochs training on the adversarial examples with a 50% reduced learning rate. The dataset remained the same through all 5 extra training rather than updated as Goodfellow *et.al* [7] did. It can be seen (in Fig. 2.7) that the robustness of the model to adversarial attack increased significantly after only one extra epoch. They also showed that the accuracy of the original network is improved by the fine-tuning with the DeepFool [20]. Conversely, they argued that the same fine-tuning method with FGSM [7] resulted to a decline of the test accuracy, of which they provided a possible explanation that the FGMS generates overly perturbed images that are impossible to occur in the test set.

Another intriguing work achieved by Madry *et. al* [19] presented an optimization view on adversarial attack and defense, which can be described by

$$\min_{\theta} \rho(\theta), \quad where\ \rho(\theta) = \mathbb{E}_{(x,y)\sim D}[\max_{\delta \in S} L(\theta, x+\delta, y)] \tag{2.7}$$

where D is the data distribution over images x and corresponding labels y. θ represents the set of model parameters. δ means the perturbation applied on the images and S is the set of allowed perturbations. Then $L(x, y, \theta)$ is defined as the loss function. This formula is composed of an *inner maximization* problem and an *outer minimization* problem. The inner maximization part is used to obtain a possible perturbation on the given image set that achieves the highest loss, while the outer minimization problem aims to find proper model

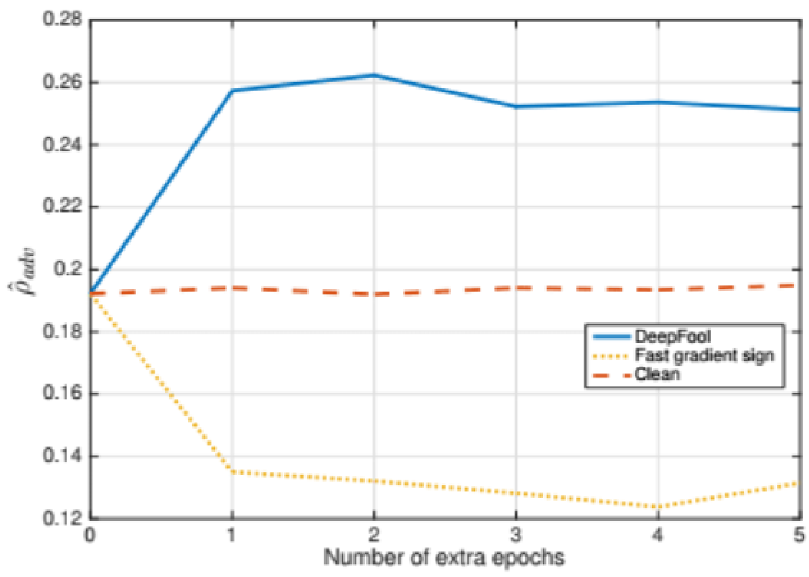


Fig. 2.7 Effect of adversarial training by DeelFool [20]

parameters that the adversarial examples give the minimum impact on the network. This perspective provided a unifying description that generalize much prior work on adversarial training. It showed a specific goal for an ideal robust image classifier, where the model parameters θ generate a vanishing risk that is able to perturb the network. And they claimed that the *projected gradient descent* (PGD) is a "universal" adversary among first-order approaches, i.e., provides the maximum attack towards the network.

$$x^{t+1} = \prod_{x+S}(x^t + \alpha sign(\bigtriangledown_x L(\theta, x, y))) \tag{2.8}$$

After applying this algorithm to craft the perturbation, they replaced the original input images by their corresponding adversarial examples and then trained the network normally on these perturbed input to compute the gradient of the loss function. In addition, they mentioned that the increasing capacity of the network helps the model fits the adversarial examples better and thus improves the robustness.

3 Original Traffic Signs Classifier Design and Implementation

In this chapter, the details of the CNN used in this project are described. This includes the traffic signs dataset, the architecture of the CNN model, the implementation process and tools used to conduct the experiment.

3.1 Data Description and Preparation

The *German Traffic Sign Recognition Benchmark* (GTSRB) [27] dataset is used for this experiment.This dataset provides various traffic signs with correct labels, so it should make this experiment reliable compared with other datasets.This dataset also presents plenty of difficult challenges due to real-world uncertainty such as lighting conditions, motion-blur and physical damage. Moreover, it is a small dataset but includes significantly different classes, which makes it easy for training and modification due to the lack of compute power, as we do not have GPUs.

The GTSRB dataset is divided into two subsets namely *training dataset* and *test dataset*. The training dataset consists of 39209 images while the test dataset of 12630 images. Some images are randomly chosen from the dataset and shown in Fig. 3.1. All of these images are pickled and have been resized to $32 \times 32 \times 3$ array of pixel intensities, represented values between 0 and 255 in RGB color space. There are 43 classes with varying frequencies as shown in Fig. 3.2.

As can be seen, the images from the dataset are colorful and extremely unbalanced. some classes are represented significantly better than others, i.e., some classes only have 200 samples, which is not enough for most of the models in training. As a result, it is necessary to process the dataset before experiment.

First, converting the colored images to grayscale ones is performed, that is only a single channel used in this experiment. As mentioned in the paper of Pierre Sermanet and LeCun [25], there is no clear difference between using color channels and gray channel,

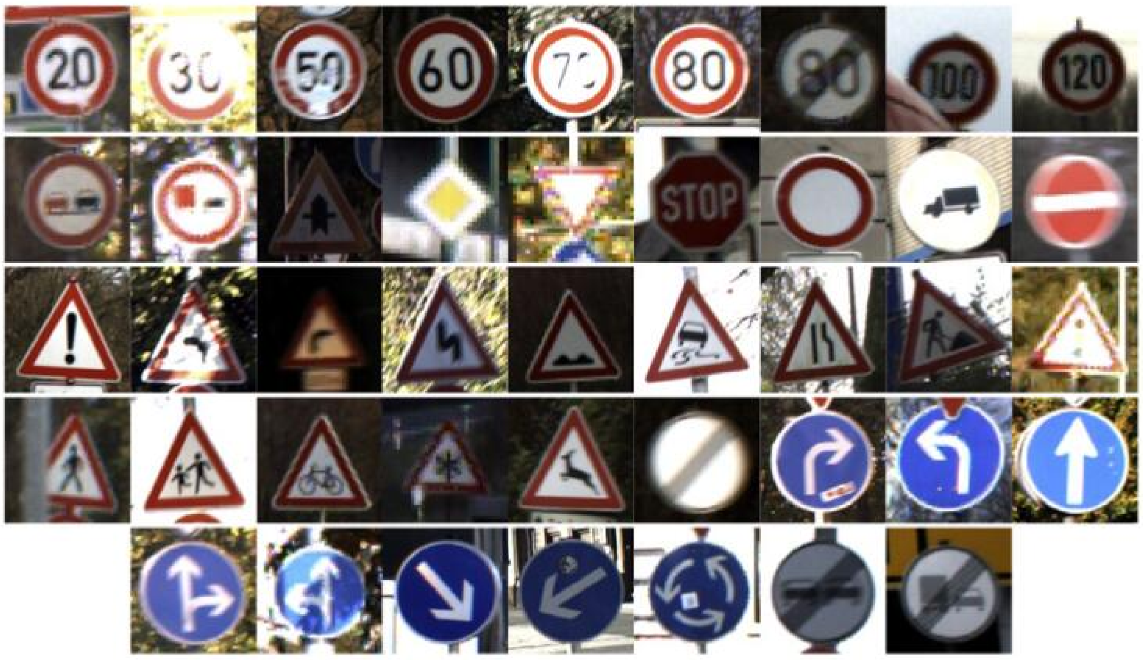


Fig. 3.1 Random representatives of the 43 traffic sign classes in the GTSRB dataset

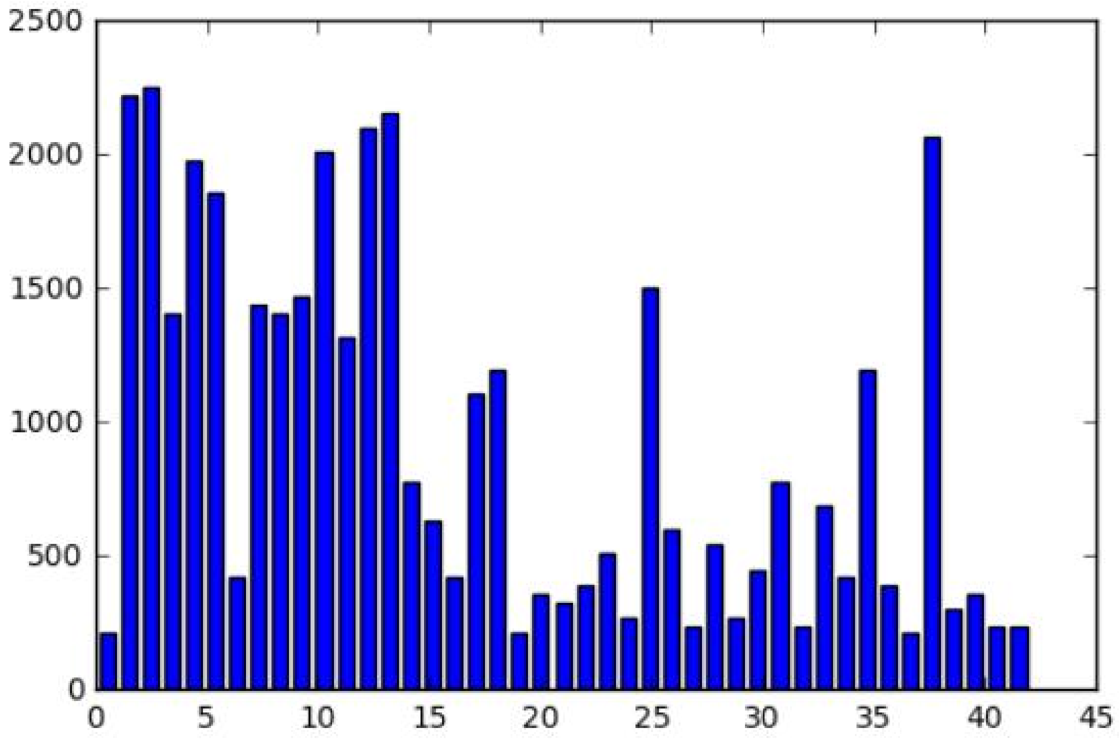
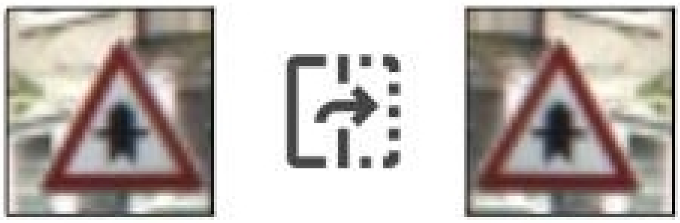


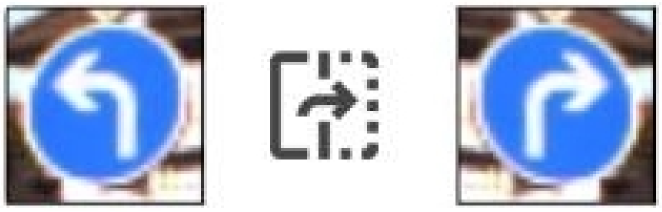
Fig. 3.2 Relative class frequencies in the dataset. Each number in the horizontal axis represents a class

and the training time of grayscale images is significantly less than color ones. Next, scaling of pixel values, which ranges from 0 to 255 currently, to $[0,1]$, representing labels in a one-hot encoding, and shuffling the dataset are completed in sequence. Finally, augmentation of the images is achieved by flipping some capable classes. For instance, some signs are horizontally and/or vertically symmetrical (like Right-of-way at the Next Intersection in Fig. 3.3a). These images can be augmented by simply flipping, thus allowing us to double the original dataset. Some other signs compose a kind of interchangeable pairs (like Turn Left Ahead in Fig. 3.3b). The amount of these images pairs can be increased after flipping and assigning to the paired classes. Other augmentation methods, including *random translate, random scale, random warp* and *random brightness* are applied to the original dataset as well. The augmentation is only used for the classes that are consisted of less than 800 images. The relative class frequencies after the augmentation is shown in Fig. 3.4. Fig. 3.5 displays some random images from the processed dataset.

During training, the training dataset is split into validation set and training set with 25% splitting ratio, i.e. training set includes 37132 images and validation set 12378 images.



(a) Flipping of Right-of-way at the Next Intersection



(b) Flipping of Turn Left Ahead

Fig. 3.3 Examples of flipping

3.2 Model Architecture

The ConvNets (Convoluntional Networks) from the paper [25] has been modified and used as the main model for the following experiments in this work. This architecture was

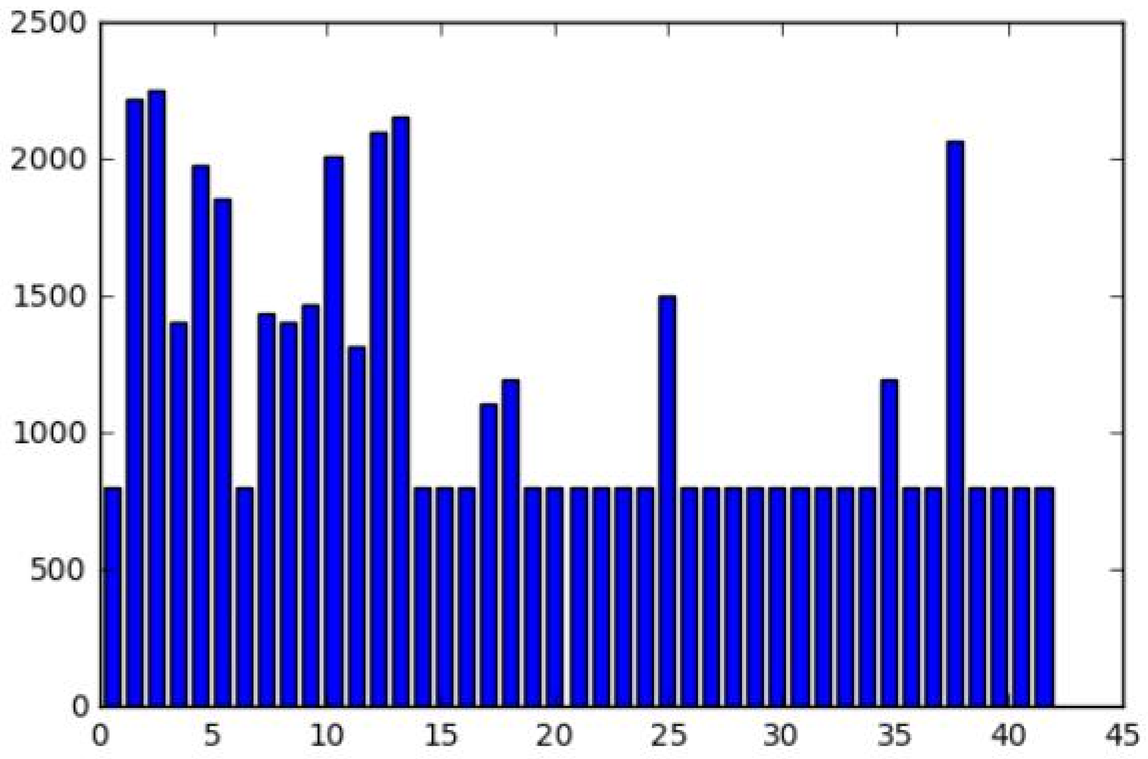


Fig. 3.4 Relative class frequencies in the dataset after augmentation

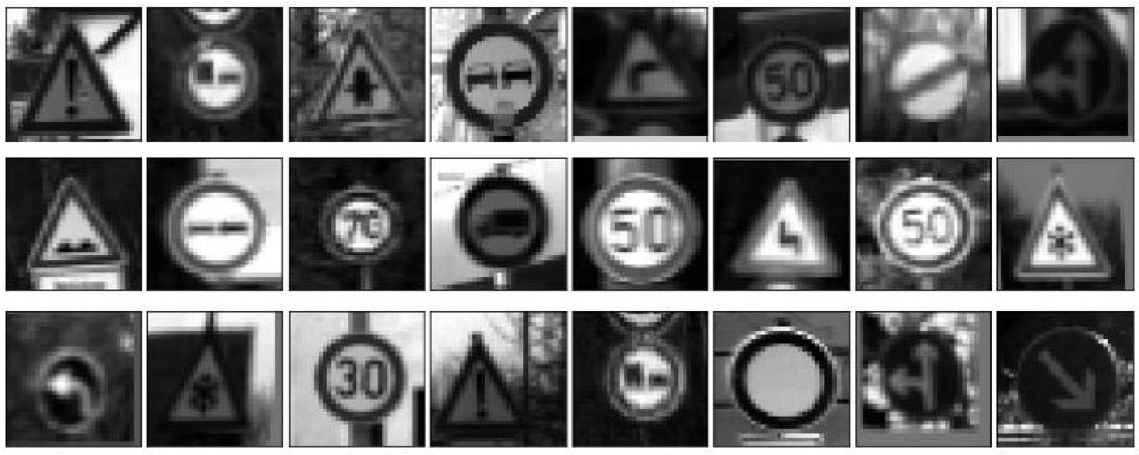


Fig. 3.5 Random samples from processed dataset

first proposed by Yann LeCun in 1998 [18] since then it has been studied and improved by many researchers. So, it is easier to compare the results from this work with other experiments. Furthermore, since the LeNet is the basis of many recent CNNs architecture such as GoogLeNet [28], training on this network should provide primary insight into more complicated modern networks.

The original ConvNets consists of two stages, each of which is composed of a convolutional filter layer, a non-linear activation layer, and a spatial feature pooling layer, followed by a classifier and gives the prediction of the input. Furthermore, the output of a stage is not only fed into the subsequent stage, but also forwarded into classifier after proportionally subsampled by addition max-pooling [25]. This network reached 98.97% accuracy after training with color images in GTSRB and 99.17% with grayscale images. The architecture of the ConvNets is displayed in Fig. 3.6.

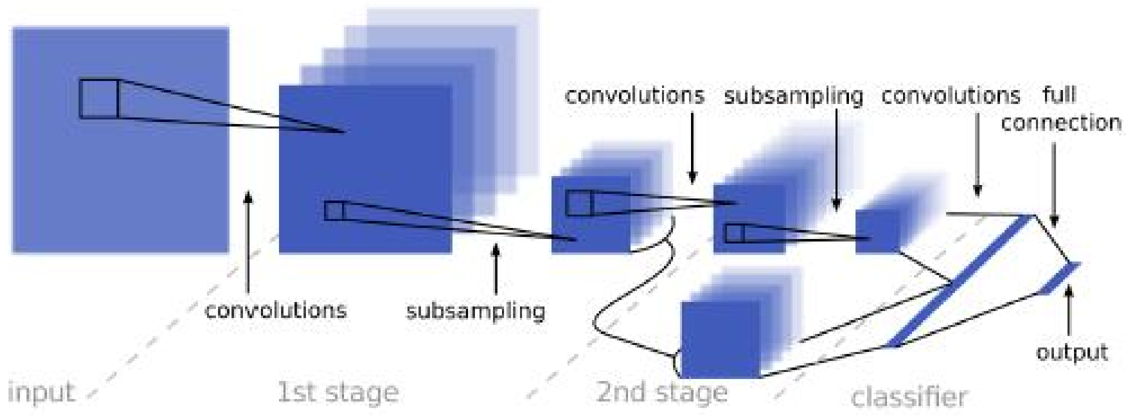


Fig. 3.6 A 2-stage ConvNet architecture by LeCun [25]

In this work, a three-stage convolutional network is proposed based on ConvNets and is used as the main model for signs classification in this report. Similar to the ConvNets, each stage in this 3-stage network includes a convolutional layer, a non-linear transform layer, and a pooling layer. Along the lines of these three stages, a classifier is added at the end. The architecture is shown in Fig. 3.7. Table 3.1 shows the parameters of this model.

3.3 Implementation of 3-stage ConvNet

After the structure of model, regularization and hyperparameters also need to be considered for implementation of 3-stage ConvNet.

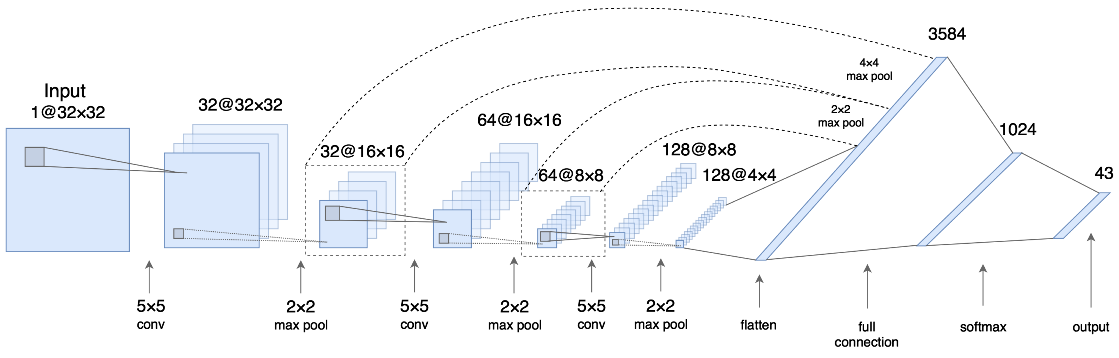


Fig. 3.7 A 3-stage convolutional network based on ConvNet

Table 3.1 Parameters of the original 3-stage CNN

Parameter	**Value**
Kernel Size of First Convolutional Layer	5×5
Kernel Depth of First Convolutional Layer	32
Kernel Size of Second Convolutional Layer	5×5
Kernel Depth of Second Convolutional Layer	64
Kernel Size of Third Convolutional Layer	5×5
Kernel Depth of Third Convolutional Layer	128
Size of Fully Connected Layer	1024
Size of Classifier	43
Stride Size	(1,1,1,1)
Kernel Size of Pooling Layer	2×2

3.3.1 Regulation

The following regularization methods are used in this work to minimize overfitting.

- **Dropout** [10, 26]. Dropout is implemented by randomly omitting some hidden units from the network with a certain probability during training. It shows that dropout dramatically improves the performance of neural networks in many application domains such as image classification, object recognition and speech recognition. In general, dropout is usually applied only on fully connected layers. In this work, however, it is extended to all stages with small probability because it is noticed that the performance of the network is improved slightly.

- **L2 Regularization** [6]. L2 regularization is a widely used method to reduce the effect of overfitting. By reducing the weight by a factor of λ, the weight value tent to decay during training.

- **Early Stopping** [6]. The network will be stopped when the error on the validation set has not improved for some training epochs instead of when it reaches a local minimum. The model parameters are stored each time the error on the validation set improves. When the training process reaches the patience of a pre-set epochs, these parameters are returned rather than the last ones.

3.3.2 Hyperparameters

The hyperparameters used in this work for the primary training are described in Table 3.2. Due to the adversarial training, it is necessary to adjust one or some of the hyperparameters in the following experiments to obtain a robust network.

Table 3.2 Hyperparameters for basic training

Parameter	**Value**
Activation Unit	ReLu
Batch Size	256
Weight Initializer	Xavier
Bias Initializer	Zeros
Learning Rate Decay	Disable
Learning Rate	0.0001
Optimizer	Adam Optimizer
Dropout on First Stage	10%
Dropout on Second Stage	20%
Dropout on Third Stage	30%
Dropout on Classifier	50%
L2 Regulation	0.0001
Early Stopping	100
Stride Size	(1,1,1,1)
Loss	Cross Entropy

3.4 Software and Instrument

In this work, the model code was implemented in Python 3.5 in a Jupyter notebook. The deeplearning framework used is **Tensorflow** [1] along with other packages such as Numpy [30], Matplotlib [11] ,and scikit-iamge [31]. The training is performed on a Linux virtual machine (Ubuntu 16.04) on Google Cloud Platform.

3.5 Result

Based on previously described setting, the model stops after 524 epochs by the early stopping and achieves validation loss of 0.0048, validation accuracy of 99.90%, test loss of 0.1871, and test accuracy of 96.86%.

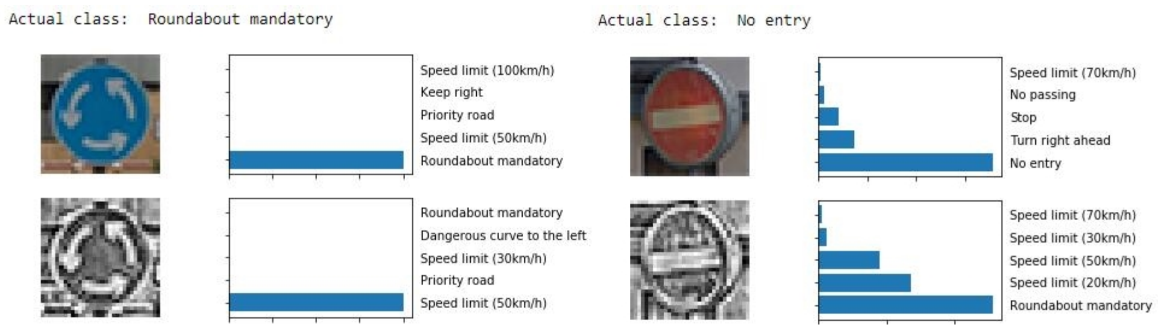
4 Adversarial Training Experiments and Results

Based on the previously mentioned CNN, adversarial training experiments are performed and presented in this chapter. First, the choice of the attack algorithm and the perturbed images are analyzed. Following this, the adversarial training is conducted on the basis of the original network and the perturbed dataset. Next, several improvements, including adding an extra stage, increasing the kernel size and changing other hyperparameters, are performed on the original model to increase the robustness followed by adversarial training experiments. Finally, a robust model architecture is constructed according to the prior trials.

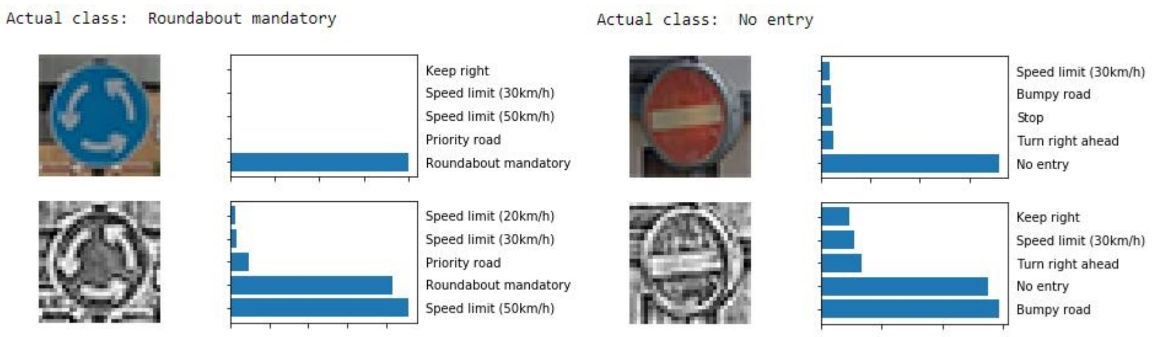
4.1 Attack Algorithm

A number of algorithms have been developed for generating adversarial examples, as mentioned in section 2.3. Among these algorithms, FGSM from Goodfellow *et. al* [7] and DeepFool from Moosavi-Dezfooli *et. al* [20] are selected as candidates for further training. FGSM is the first practical attack algorithm that is effective and straightforward. And the code is available on CleverHans [22] which is an adversarial example library for constructing attacks, building defenses, and benchmarking. Therefore, it should provide a heuristic perspective to learn the principle of the attack and access more complex attack algorithms. DeepFool is chosen due to a two main reasons. First, it provides an untargeted attack which crafts more powerful unlimited perturbations that are more likely to cause misclassification. This is necessary because it allows a robust network to resist perturbations from any type of attack. Second, DeepFool is an iterative algorithm that stops iterating when the classifier outputs an incorrect label for the given input, which means that the perturbations added on the image are the smallest one. As a result, this method is capable of fine-tuning the original network by adversarial training without large overfitting. The implementation code is also available on Github but it is implemented by Pytorch (a python package for Deep Neural Networks) instead of TensorFlow. So the reconstruction of the code is necessary at first.

After adding perturbations on the pure test set by these two algorithms, we observe the test accuracy drops to 7.69% for FGSM attack and 3.11% for DeepFool attack respectively. Compared with the test accuracy of 96.86% on pure set, the accuracy on the adversarial set is influenced dramatically by these two attacks. To dig deeper into the difference between these two algorithms, some adversarial examples from different attacks are displayed in Fig. 4.1.



(a) Examples of FGSM attack



(b) Examples of DeepFool attack

Fig. 4.1 Comparison between FGSM and DeepFool attack

Fig 4.1a shows two examples of the FGSM attack. In the figure, the above colorful traffic sign is the original image while the grey one is the perturbed image. To the right of these signs, the bar charts represent the top five possibility for the images predicted by the classifier. Fig 4.1b displays the same two traffic signs but attacked by DeepFool. Comparing the actual effect of these two attack methods, we can conclude that FGSM provides a random attack and that the perturbed image label has little correlation with the previous label, whereas DeepFool attacks the model with a small perturbation which changes the previous label from top one probability to the second place. Therefore, DeepFool will be selected as the main attack method for the following experiments. It crafts more powerful attacks on the given image set and should help to generate more robust network.

4.2 Experiments

In this section, the adversarial training on the original work is performed using different training methods. As mentioned in Section 2.4, a variety of adversarial training approaches have been applied on the network after crafting perturbed dataset to obtain robust classifiers. Among them, several approaches including training for extra epochs, on fully perturbed dataset, and on partially perturbed dataset are chosen in this work during the experiments. Following this, two improvements, modifying the hyperparameters and adding a stage, are achieved and the model is trained again to obtain the responses to different changes.

4.2.1 Experiment 1: Increasing Model Capacity

As suggested by Madry *et. al* [19], the capacity of network is crucial for robustness and the ability to defense against strong attacks. So the extension of the number of stages is first considered, from 3 stages of the original model to 4 stages. The parameters of the modified structure are listed in the Table 4.1. And other training parameters remain the same as the original training (see Table 3.2).

Table 4.1 Parameters of the CNN in Experiment 1

Parameter	**Value**
Kernel Size of First Convolutional Layer	5×5
Kernel Depth of First Convolutional Layer	16
Dropout on First Stage	10%
Kernel Size of Second Convolutional Layer	5×5
Kernel Depth of Second Convolutional Layer	32
Dropout on Second Stage	20%
Kernel Size of Third Convolutional Layer	5×5
Kernel Depth of Third Convolutional Layer	64
Dropout on Third Stage	30%
Kernel Size of Fourth Convolutional Layer	3×3
Kernel Depth of Fourth Convolutional Layer	128
Dropout on Fourth Stage	40%
Size of Fully Connected Layer	1024
Size of Classifier	43
Dropout on Classifier	50%
Stride Size	(1,1,1,1)
Kernel Size of Pooling Layer	2×2

4.2.2 Experiment 2: Changing Hyperparameters

As mentioned by Goodfellow *et. al* [7], although using more weights per layer could cause slight overfitting without adversarial training, it is able to help the model extract more features of the adversarial examples so that achieve better performance on adversarial training. Therefore, larger kernels are applied in this experiment. In order to prevent overfitting, larger rate of dropout is also used. The improved parameters are shown in Table 4.2 and other training parameters that are not listed keep unchanged (see Table 3.2).

Table 4.2 Parameters of the CNN in Experiment 2

Parameter	**Value**
Kernel Size of First Convolutional Layer	7×7
Kernel Depth of First Convolutional Layer	32
Dropout on First Stage	20%
Kernel Size of Second Convolutional Layer	7×7
Kernel Depth of Second Convolutional Layer	64
Dropout on Second Stage	30%
Kernel Size of Third Convolutional Layer	5×5
Kernel Depth of Third Convolutional Layer	128
Dropout on Third Stage	40%
Size of Fully Connected Layer	1024
Size of Classifier	43
Dropout on Classifier	60%
Stride Size	(1,1,1,1)
Kernel Size of Pooling Layer	2×2

4.3 Results

In this section, the results of the adversarial training on original network and analysis towards these results are shown. Then the outputs of Experiment 1 and 2 followed by the respective comparison with the original results are described in the next two subsections.

4.3.1 Results of Adversarial Training on Original Model

Table 4.3 displays the adversarial training results of original network. At first, 5 extra epochs training on the fully perturbed dataset is completed as suggested by Moosavi-Dezfooli *et. al* [20]. It can be seen that for test on adversarial set, the accuracy has increased significantly from 3.94% to 46.33%. On the other hand, however, the accuracy for test on pure set falls

Table 4.3 Adversarial training results of original network

Training Type	Validation		Test with Pure Set		Test with Adversarial Set	
	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy
Original Training	0.0048	99.90%	0.19	96.86%	8.27	3.94%
5 Extra Epochs	1.32	64.25%	1.09	78.21%	1.98	46.33%
Fully Perturbed Set	2.08	69.63%	4.93	48.17%	1.48	51.63%
50% Perturbed Set	0.69	82.33%	0.68	89.07%	2.46	45.27%
30% Perturbed Set	0.45	88.24%	0.47	92.06%	2.36	45.73%
10% Perturbed Set	0.18	95.48%	0.37	93.08%	2.27	42.94%

from 96.86% to 78.21%, which is unacceptable for a classifier. A similar trend appears to the fully perturbed set training as well. The reason is that the training set has been highly perturbed and the feature of pure images can not be extracted from this set. Furthermore, the test accuracy of perturbed dataset is also not satisfied since the structure of the model is so simple that the proper features of the adversarial images can not be obtained.

To solve this problem, partially perturbed sets are introduced with different perturbed ratio 50%, 30% and 10%. With the decrease of the perturbed ratio, the accuracy on pure set increases slightly from 89.07% to 93.08% while the accuracy on adversarial set shows an opposite tendency. This is reasonable because smaller perturbed ratio makes the dataset contain more pure images so that the model performs better on pure set but worse on adversarial set.

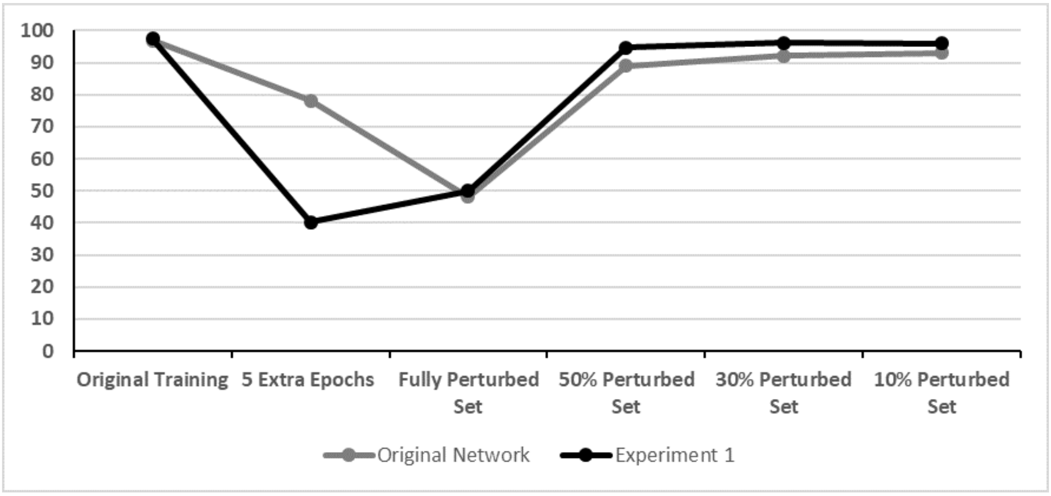
According to the results, we can conclude that this 3-stage is unable to defense the Deepfool attack due to the low accuracy of classifying adversarial examples even trained on fully perturbed set.

4.3.2 Results of Experiment 1

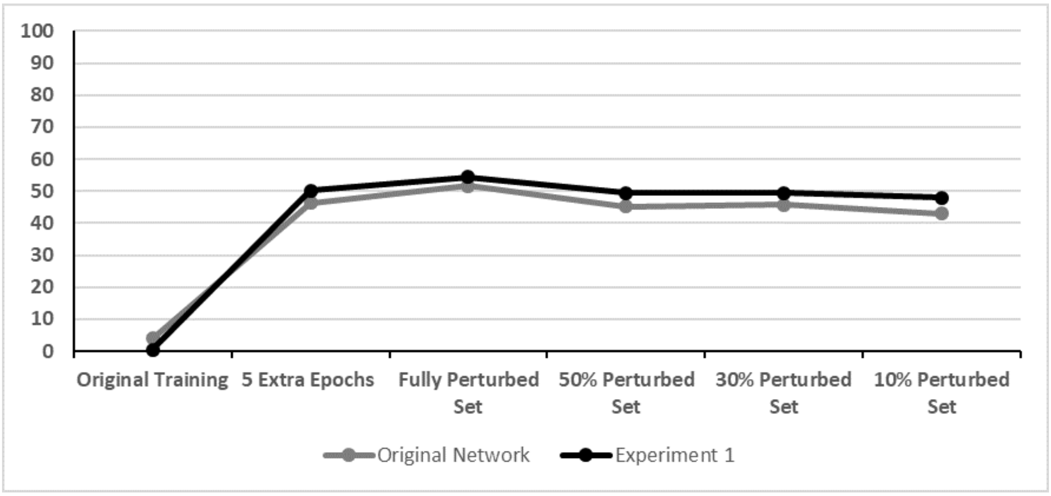
The results of Experiment 1 show in Table 4.4. Fig 4.2 illustrates the comparison between the original training and Experiment 1. From Fig. 4.2a, we can see that after 5 epochs training on the fully perturbed set, the modified model shows much lower accuracy than original one when tested on pure set. This means more weights in the 4-stage model have shifted towards adversarial data within 5 epochs and it should learn faster. Combining with Fig. 4.2b, we can conclude that the improved model performs slightly better both tested on pure set and adversarial set. The feature extraction ability of the model has enhanced by introducing additional stage. But under the condition of an acceptable accuracy on pure set, the accuracy

Table 4.4 Adversarial training results of Experiment 1

Training Type	Validation		Test with Pure Set		Test with Adversarial Set	
	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy
Original Training	0.0034	99.93%	0.12	97.51%	4.72	0.54%
5 Extra Epochs	1.42	61.61%	7.67	40.23%	2.19	50.23%
Fully Perturbed Set	1.96	67.82%	4.27	50.03%	1.88	54.39%
50% Perturbed Set	0.59	84.02%	0.26	94.76%	2.27	49.52%
30% Perturbed Set	0.38	90.10%	0.18	96.11%	2.15	49.41%
10% Perturbed Set	0.15	95.77%	0.16	95.95%	2.04	47.97%



(a) Comparison between the results of original training and Experiment 1 on pure set



(b) Comparison between the results of original training and Experiment 1 on adversarial set

Fig. 4.2 Comparison between original training and Experiment 1

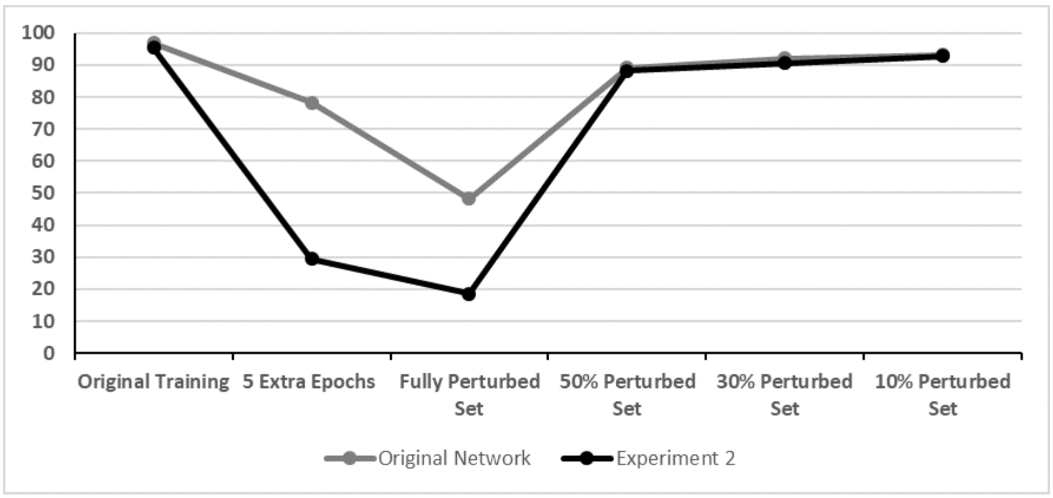
on the adversarial set is only 49.41% (trained on 30% perturbed set), which is not satisfying to defense a possible attack.

4.3.3 Results of Experiment 2

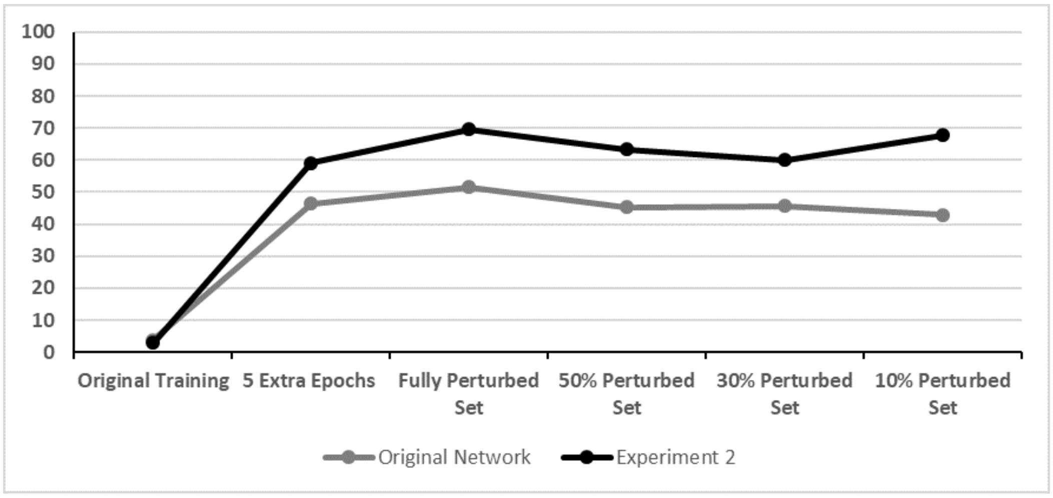
The results of Experiment 2 are shown in Table 4.5. The comparison with the original training results is displayed in Fig. 4.3. In Fig. 4.3a, it is clear that the accuracy of training on fully perturbed set in Experiment 2, either 5 epochs or fully training, is significantly smaller than that on original training. When tested on adversarial set (shown in Fig. 4.3b), the model in Experiment 2 promotes the accuracy dramatically. It means increasing the amount of units in each layers is an effective approach to relieve the attack. This is reasonable since using more units in the network helps the classifier extract more features of the images. On the other hand, however, because the model utilizes more features, the perturbation in the partially perturbed set exerts more influence on the classifier, which leads to a slight decline of test accuracy on pure set.

Table 4.5 Adversarial training results of Experiment 2

Training Type	Validation		Test with Pure Set		Test with Adversarial Set	
	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy
Original Training	0.011	99.71%	0.23	95.29%	3.25	2.96%
5 Extra Epochs	1.61	56.16%	6.49	29.41%	1.53	59.13%
Fully Perturbed Set	0.90	76.70%	94.95	18.46%	1.08	69.54%
50% Perturbed Set	0.58	85.11%	0.87	88.22%	1.42	63.28%
30% Perturbed Set	0.47	89.72%	0.71	90.57%	1.65	60.06%
10% Perturbed Set	0.17	95.38%	0.58	92.73%	1.83	57.78%



(a) Comparison between the results of original training and Experiment 2 on pure set



(b) Comparison between the results of original training and Experiment 2 on adversarial set

Fig. 4.3 Comparison between original training and Experiment 2

4.4 Building a More Robust Model

Based on Experiment 1 and 2, we can conclude that both network capacity and number of units have large impact on the adversarial resistance of the resulting models. But neither of them produces a reliable network that performs better than the original one. A model combining the modifications from both Experiment 1 and 2 is thus assumed to be able to balance the advantages of the two models and yield larger robustness. The parameters of this combining model are listed in Table 4.6 and other training parameters are the same as the original training (shown in Table 3.2).

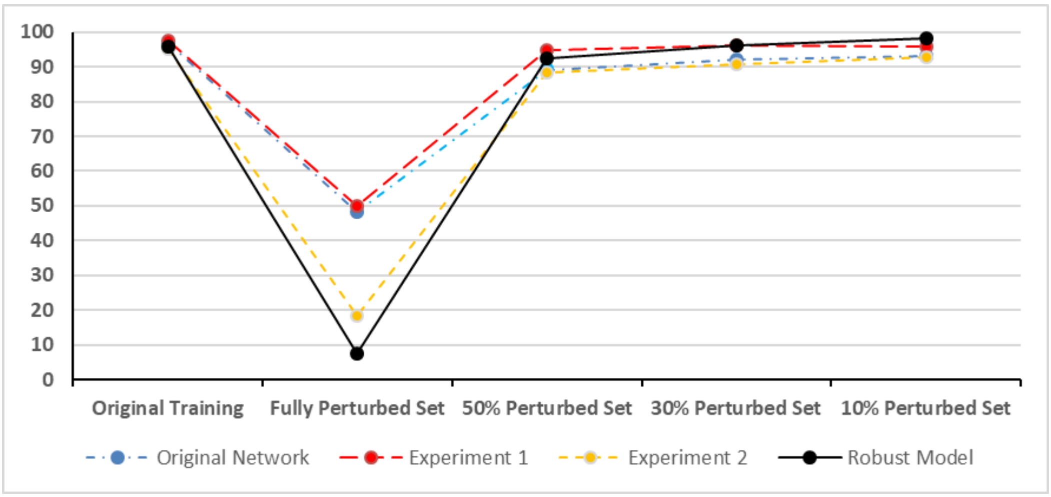
Table 4.6 Parameters of the robust model

Parameter	**Value**
Kernel Size of First Convolutional Layer	7×7
Kernel Depth of First Convolutional Layer	16
Dropout on First Stage	20%
Kernel Size of Second Convolutional Layer	7×7
Kernel Depth of Second Convolutional Layer	32
Dropout on Second Stage	30%
Kernel Size of Third Convolutional Layer	5×5
Kernel Depth of Third Convolutional Layer	64
Dropout on Third Stage	40%
Kernel Size of Fourth Convolutional Layer	3×3
Kernel Depth of Fourth Convolutional Layer	128
Dropout on Fourth Stage	50%
Size of Fully Connected Layer	1024
Size of Classifier	43
Dropout on Classifier	60%
Stride Size	(1,1,1,1)
Kernel Size of Pooling Layer	2×2

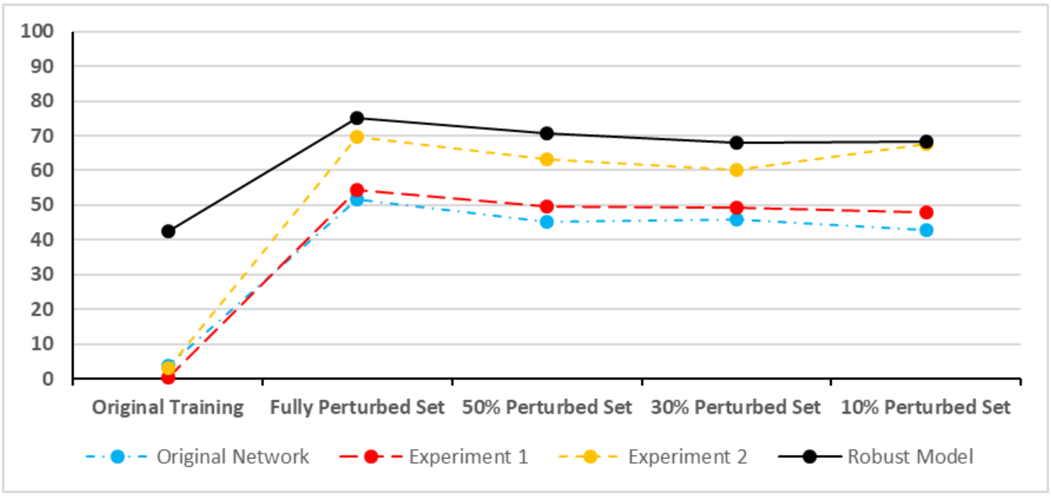
After training with different dataset, tests on different sets are achieved to check the robustness of the new model and the test results are displayed in Table 4.7. To compare the results with the previous models, we plot Fig. 4.4. From the results of test on pure set (shown in Fig. 4.4a), we can notice that the accuracy obtained by training on 10% perturbed set reaches the highest value of 98.28%, which is even larger than the original training of 95.94%. As mentioned by Goodfellow *et. al* [7], this phenomenon can be explained as regularization by adversarial training. The robust model should overfit slightly on this dataset and the adversarial training on small ratio perturbed set helps to relieve this flaw. In Fig. 4.4b, new model achieves largest accuracy on all training methods which indicates

Table 4.7 Test results of robust model

Training Type	Validation		Test with Pure Set		Test with Adversarial Set	
	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy
Original Training	0.0076	99.81%	0.22	95.94%	3.16	42.42%
Fully Perturbed Set	1.53	80.24%	1755.38	7.58%	0.86	75.26%
50% Perturbed Set	0.62	83.11%	0.46	92.55%	1.17	70.71%
30% Perturbed Set	0.31	90.26%	0.29	96.18%	1.31	68.09%
10% Perturbed Set	0.16	96.76%	0.09	98.28%	1.49	68.42%



(a) Comparison between the results of robust and previous model on pure set



(b) Comparison between the results of robust and previous model on adversarial set

Fig. 4.4 Comparison between robust model and previous models

that the defense ability of this model is better than other previously proposed models as we assumed before. Another intriguing point is that when tested on adversarial set, the accuracy gained by original training is much larger than any other models but almost no apparent differences are observed from the losses (8.27, 4.72, 3.25 and 3.16 respectively). A possible reason could be the perturbations added on the images are so small that do not cause large distinction between original data and adversarial data. As a result, the losses are still small but accuracy drops sharply. This also verifies that the new model is more robust than other models.

5 Conclusion

This work presents a modified CNN based on LeNet [25] that aims to build a robust classifier for traffic signs in autonomous vehicles. This is motivated by solving the contradiction between the vulnerability of CNNs and the safety demand of self-driving cars. First, we studied existing attack algorithms and a number of training approaches that attempt to defend against adversarial attacks. We found that existing approaches are not efficient for defending advanced adversarial attacks. Further, we analyzed the basic structure of LeNet and improved it to fit our experiment. We proposed a 3-stage LeNet where regulation methods such as dropout, early stopping and L2 regularization are applied during training on traffic signs data from GTSRB. Before training, image processing and data augmentation were completed to achieve a balanced image set. On the basis of the original training, we attacked the network with DeepFool [20] by adding perturbations to the traffic signs. Next, we performed adversarial training using this perturbed dataset by different training methods, such as 5-extra epochs training and partially perturbed dataset training. We observed that with an increase of perturbed ratio, the accuracy tested on pure images decreased, while the accuracy tested on the perturbed set does not show a clear improvement. Therefore, we assume that this original model is not able to guard against the DeepFool adversarial attack and an improved model is needed. We modified the model in two ways: we enhance the model capacity by adding an extra stage, and we increased the number of units of each stage by utilizing larger kernels. After achieving the same training on these two modified model, we saw that using larger kernels improve the accuracy tested on an adversarial set dramatically, whereas a 4-stage network is capable of boosting the robustness. As a result, a model that combines these two modifications was proposed and was conjectured to be a more robust one. The test results proved our assumption that this new network achieved 98.28% accuracy when tested on a pure set and 70.71% accuracy on an adversarial set.

Although the proposed network alone is not sufficient to eliminate the impact of attacks on a system, it does show a possible direction to a robust network that can resist any slight enough perturbations that can not be recognized by human. We have shown that model capacity and the number of units have large influence on robustness of a CNN. In the future, we will try deeper CNNs such as GoogleNet [28] and ResNet [9]. And we need to consider the correlation between the number of units and the overfitting of the network. As we conclude from the experiment, with the increasing in number of units, the network starts to overfit slightly. If we continue to increase the number of units, will the degree of overfitting keep enlarging? This relationship is really significant for the robustness of a CNN and still needs to work on.

References

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[2] Bebis, G. and Georgiopoulos, M. (1994). Feed-forward neural networks. *IEEE Potentials*, 13(4):27–31.

[3] Bos, S. and Chug, E. (1996). Using weight decay to optimize the generalization ability of a perceptron. In *Neural Networks, 1996., IEEE International Conference on*, volume 1, pages 241–246 vol.1.

[4] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.

[5] Evtimov, I., Eykholt, K., Fernandes, E., Kohno, T., Li, B., Prakash, A., Rahmati, A., and Song, D. (2017). Robust physical-world attacks on machine learning models. *CoRR*, abs/1707.08945.

[6] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[7] Goodfellow, I. J., Shlens, J., and Szegedy, C. (2014). Explaining and Harnessing Adversarial Examples. *ArXiv e-prints*.

[8] Gu, S. and Rigazio, L. (2014). Towards deep neural network architectures robust to adversarial examples. *CoRR*, abs/1412.5068.

[9] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition. *CoRR*, abs/1512.03385.

[10] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580.

[11] Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95.

[12] Jarrett, K., Kavukcuoglu, K., Ranzato, M., and LeCun, Y. (2009). What is the best multi-stage architecture for object recognition? In *2009 IEEE 12th International Conference on Computer Vision*, pages 2146–2153.

[13] Krizhevsky, A. (2009). Learning multiple layers of features from tiny images.

[14] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

[15] LeCun, Y. (1998). The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/.*

[16] LeCun, Y., Bengio, Y., et al. (1995). Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995.

[17] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1(4):541–551.

[18] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

[19] Madry, A., Makelov, A., Schmidt, L., Tsipras, D., and Vladu, A. (2017). Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083.*

[20] Moosavi-Dezfooli, S., Fawzi, A., and Frossard, P. (2015). Deepfool: a simple and accurate method to fool deep neural networks. *CoRR*, abs/1511.04599.

[21] Nguyen, A., Yosinski, J., and Clune, J. (2015). Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 427–436.

[22] Nicolas Papernot, Nicholas Carlini, I. G. R. F. F. F. A. M. K. H. Y.-L. J. A. K. R. S. A. G. Y.-C. L. (2017). cleverhans v2.0.0: an adversarial machine learning library. *arXiv preprint arXiv:1610.00768.*

[23] Papernot, N., McDaniel, P., Jha, S., Fredrikson, M., Celik, Z. B., and Swami, A. (2016). The limitations of deep learning in adversarial settings. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pages 372–387. IEEE.

[24] Papernot, N., McDaniel, P. D., Wu, X., Jha, S., and Swami, A. (2015). Distillation as a defense to adversarial perturbations against deep neural networks. *CoRR*, abs/1511.04508.

[25] Sermanet, P. and LeCun, Y. (2011). Traffic sign recognition with multi-scale convolutional networks. In *The 2011 International Joint Conference on Neural Networks*, pages 2809–2813.

[26] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958.

[27] Stallkamp, J., Schlipsing, M., Salmen, J., and Igel, C. (2011). The German Traffic Sign Recognition Benchmark: A multi-class classification competition. In *IEEE International Joint Conference on Neural Networks*, pages 1453–1460.

[28] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9.

[29] Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I. J., and Fergus, R. (2013). Intriguing properties of neural networks. *CoRR*, abs/1312.6199.

[30] van der Walt, S., Colbert, S. C., and Varoquaux, G. (2011). The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30.

[31] van der Walt, S., Schönberger, J. L., Nunez-Iglesias, J., Boulogne, F., Warner, J. D., Yager, N., Gouillart, E., and Yu, T. a. (2014). scikit-image: image processing in python. *PeerJ*, 2:e453.

[32] Zhou, Y. T. and Chellappa, R. (1988). Computation of optical flow using a neural network. In *IEEE 1988 International Conference on Neural Networks*, pages 71–78 vol.2.

Appendix A Related Code

A.1 Image Processing

Feature scaling

```
from sklearn.utils import shuffle
from skimage import exposure
import warnings
num_classes = 43

def preprocess_dataset(X, y = None, shuffle=True):
    """
    Performs feature scaling, one-hot encoding of labels
       and shuffles the data if labels are provided.
    Parameters
    ----------
    X: ndarray
       Dataset array containing feature examples.
    y: ndarray, optional, defaults to None
       Dataset labels in index form.
    Returns
    -------
    A tuple of X and y.
    """
    print("Preprocessing dataset with {} examples:".format(
       X.shape[0]))

    #Convert to grayscale, e.g. single channel Y
    X = 0.299 * X[:, :, :, 0] + 0.587 * X[:, :, :, 1] +
       0.114 * X[:, :, :, 2]
    #Scale features to be in [0, 1]
    X = (X / 255.).astype(np.float32)

    for i in range(X.shape[0]):
        with warnings.catch_warnings():
            warnings.simplefilter("ignore")
            X[i] = exposure.equalize_adapthist(X[i])
        print_progress(i + 1, X.shape[0])

    if y is not None:
        # Convert to one-hot encoding. Convert back with y
           = y.nonzero()[1]
        y = np.eye(num_classes)[y]
        if shuffle:
            X, y = shuffle(X, y)


    # Add a single grayscale channel
    X = X.reshape(X.shape + (1,))
    return X, y
```

Image flipping

```
def flip_extend(X, y):
    """
    Extends existing images dataset by flipping images of
       some classes. As some images would still belong to
       same class after flipping we extend such classes
       with flipped images. Images of other would toggle
       between two classes when flipped, so for those we
       extend existing datasets as well.
     Parameters
    ----------
    X: ndarray
       Dataset array containing feature examples.
    y: ndarray, optional, defaults to None
       Dataset labels in index form.
    Returns
    -------
    A tuple of X and y.
    """
    # Classes of signs that, when flipped horizontally,
       should still be classified as the same class
       self_flippable_horizontally = np.array([11, 12, 13,
       15, 17, 18, 22, 26, 30, 35])
    # Classes of signs that, when flipped vertically,
       should still be classified as the same class
       self_flippable_vertically = np.array([1, 5, 12, 15,
       17])
    # Classes of signs that, when flipped horizontally and
       then vertically, should still be classified as the
       same class self_flippable_both = np.array([32, 40])
    # Classes of signs that, when flipped horizontally,
       would still be meaningful, but should be classified
       as some other class
    cross_flippable = np.array([
        [19, 20],
        [33, 34],
        [36, 37],
        [38, 39],
        [20, 19],
        [34, 33],
        [37, 36],
        [39, 38],
    ])
    num_classes = 43

    X_extended = np.empty([0, X.shape[1], X.shape[2], X.
       shape[3]], dtype = X.dtype)
    y_extended = np.empty([0], dtype = y.dtype)
    for c in range(num_classes):
        # First copy existing data for this class
        X_extended = np.append(X_extended, X[y == c], axis
           = 0)
        # If we can flip images of this class horizontally
           and they would still belong to said class...
        if c in self_flippable_horizontally:
            # ...Copy their flipped versions into extended
               array.
            X_extended = np.append(X_extended, X[y == c][:,
                :, ::-1, :], axis = 0)
        # If we can flip images of this class horizontally
           and they would belong to other class...
        if c in cross_flippable[:, 0]:
            # ...Copy flipped images of that other class to
                the extended array.
            flip_class = cross_flippable[cross_flippable[:,
                0] == c][0][1]
            X_extended = np.append(X_extended, X[y ==
               flip_class][:, :, ::-1, :], axis = 0)
        # Fill labels for added images set to current class
           .
        y_extended = np.append(y_extended, np.full((
           X_extended.shape[0] - y_extended.shape[0]), c,
           dtype = int))

        # If we can flip images of this class vertically
           and they would still belong to said class...
        if c in self_flippable_vertically:
            # ...Copy their flipped versions into extended
               array.
            X_extended = np.append(X_extended, X_extended[
               y_extended == c][:, ::-1, :, :], axis = 0)
        # Fill labels for added images set to current class
           .
        y_extended = np.append(y_extended, np.full((
           X_extended.shape[0] - y_extended.shape[0]), c,
           dtype = int))

        # If we can flip images of this class horizontally
           AND vertically and they would still belong to
           said class...
        if c in self_flippable_both:
            # ...Copy their flipped versions into extended
               array.
            X_extended = np.append(X_extended, X_extended[
               y_extended == c][:, ::-1, ::-1, :], axis =
               0)
        # Fill labels for added images set to current class
           .
        y_extended = np.append(y_extended, np.full((
           X_extended.shape[0] - y_extended.shape[0]), c,
           dtype = int))

    return (X_extended, y_extended)
```

A.2 Training

Model Pass for a 4-layer network

```
def fully_connected(input, size):
    """
    Performs a single fully connected layer pass, e.g.
       returns input * weights + bias.
    """
    weights = tf.get_variable( 'weights',
        shape = [input.get_shape()[1], size],
        initializer = tf.contrib.layers.xavier_initializer
           ()
      )
    biases = tf.get_variable( 'biases',
        shape = [size],
        initializer = tf.constant_initializer(0.0)
      )
    return tf.matmul(input, weights) + biases

def fully_connected_relu(input, size):
    return tf.nn.relu(fully_connected(input, size))

def conv_relu(input, kernel_size, depth):
    """
    Performs a single convolution layer pass.
    """
    weights = tf.get_variable( 'weights',
        shape = [kernel_size, kernel_size, input.get_shape
           ()[3], depth],
        initializer = tf.contrib.layers.xavier_initializer
           ()
      )
    biases = tf.get_variable( 'biases',
        shape = [depth],
        initializer = tf.constant_initializer(0.0)
      )
    conv = tf.nn.conv2d(input, weights,
        strides = [1, 1, 1, 1], padding = 'SAME')
    return tf.nn.relu(conv + biases)

def pool(input, size):
    """
    Performs a max pooling layer pass.
    """
    return tf.nn.max_pool(
        input,
        ksize = [1, size, size, 1],
        strides = [1, size, size, 1],
        padding = 'SAME'
    )
```

```
def model_pass(input, params, is_training):
    """
    Performs a full model pass.
    Parameters
    ----------
    input: Tensor
           NumPy array containing a batch of examples.
    params: Parameters
            Structure (namedtuple) containing model
               parameters.
    is_training: Tensor of type tf.bool
                 Flag indicating if we are training or not
                    (e.g. whether to use dropout).
    Returns
    -------
    Tensor with predicted logits.
    """

    # Convolutions
    with tf.variable_scope('conv1'):
        conv1 = conv_relu(input, kernel_size = params.
           conv1_k, depth = params.conv1_d)
    with tf.variable_scope('pool1'):
        pool1 = pool(conv1, size = 2)
        pool1 = tf.cond(is_training, lambda: tf.nn.dropout(
           pool1, keep_prob = params.conv1_p), lambda:
           pool1)
    with tf.variable_scope('conv2'):
        conv2 = conv_relu(pool1, kernel_size = params.
           conv2_k, depth = params.conv2_d)
    with tf.variable_scope('pool2'):
        pool2 = pool(conv2, size = 2)
        pool2 = tf.cond(is_training, lambda: tf.nn.dropout(
           pool2, keep_prob = params.conv2_p), lambda:
           pool2)
    with tf.variable_scope('conv3'):
        conv3 = conv_relu(pool2, kernel_size = params.
           conv3_k, depth = params.conv3_d)
    with tf.variable_scope('pool3'):
        pool3 = pool(conv3, size = 2)
        pool3 = tf.cond(is_training, lambda: tf.nn.dropout(
           pool3, keep_prob = params.conv3_p), lambda:
           pool3)
    with tf.variable_scope('conv4'):
        conv4=conv_relu(pool3, kernel_size = params.conv4_k
           , depth = params.conv4_d)
    with tf.variable_scope('pool4'):
        pool4 = pool(conv4, size = 2)
        pool4 = tf.cond(is_training, lambda: tf.nn.dropout(
           pool4, keep_prob = params.conv4_p), lambda:
           pool4)

    # Fully connected
    # 1st stage output
    pool1 = pool(pool1, size = 8)
    shape = pool1.get_shape().as_list()
    pool1 = tf.reshape(pool1, [-1, shape[1] * shape[2] *
       shape[3]])

    # 2nd stage output
    pool2 = pool(pool2, size = 4)
    shape = pool2.get_shape().as_list()
    pool2 = tf.reshape(pool2, [-1, shape[1] * shape[2] *
       shape[3]])

    # 3rd stage output
    pool3 = pool(pool3, size = 2)
    shape = pool3.get_shape().as_list()
    pool3 = tf.reshape(pool3, [-1, shape[1] * shape[2] *
       shape[3]])

    # 4th stage output
    shape = pool4.get_shape().as_list()
    pool4 = tf.reshape(pool4, [-1, shape[1] * shape[2] *
       shape[3]])

    flattened = tf.concat([pool1, pool2, pool3,pool4],1)

    with tf.variable_scope('fc4'):
        fc4 = fully_connected_relu(flattened, size = params
           .fc4_size)
        fc4 = tf.cond(is_training, lambda: tf.nn.dropout(
           fc4, keep_prob = params.fc4_p), lambda: fc4)
    with tf.variable_scope('out'):
        logits = fully_connected(fc4, size = params.
           num_classes)
    return logits
```

Train the model

```
def train_model(params, X_train, y_train, X_valid, y_valid,
    X_test, y_test):
    """
    Performs model training based on provided training
       dataset according to provided parameters, and then
       evaluates trained model with testing dataset. Part
       of the training dataset may be used for validation
       during training if specified in model parameters.

    Parameters
    ----------
    params: Parameters
           Structure containing model parameters.
    X_train: Training dataset.
    y_train: Training dataset labels.
    X_valid: Validation dataset.
    y_valid: Validation dataset labels.
    X_test: Testing dataset.
    y_test: Testing dataset labels.
    logger_config: Logger configuration, containing Dropbox
        and Telegram settings for notifications and cloud
        logs backup.
    """

    # Initialisation routines: generate variable scope,
        create logger, note start time.
    paths = Paths(params)
    log = ModelCloudLog(
        os.path.join(paths.root_path, "logs"),
    )
    start = time.time()
    model_variable_scope = paths.var_scope

    log.log_parameters(params, y_train.shape[0], y_valid.
        shape[0], y_test.shape[0])

    # Build the graph
    graph = tf.Graph()
    with graph.as_default():
        # Input data. For the training data, we use a
            placeholder that will be fed at run time with a
            training minibatch.
        tf_x_batch = tf.placeholder(tf.float32, shape = (
            None, params.image_size[0], params.image_size
            [1], 1))
        tf_y_batch = tf.placeholder(tf.float32, shape = (
            None, params.num_classes))
        is_training = tf.placeholder(tf.bool)
        current_epoch = tf.Variable(0, trainable=False)  #
            count the number of epochs

        # Model parameters.
        if params.learning_rate_decay:
            learning_rate = tf.train.exponential_decay(
                params.learning_rate, current_epoch,
                decay_steps = params.max_epochs, decay_rate
                = 0.01)
        else:
            learning_rate = params.learning_rate

        # Training computation.
        with tf.variable_scope(model_variable_scope):
            logits = model_pass(tf_x_batch, params,
                is_training)
            if params.l2_reg_enabled:
                with tf.variable_scope('fc4', reuse = True)
                    :
                    l2_loss = tf.nn.l2_loss(tf.get_variable
                        ('weights'))
            else:
                l2_loss = 0

        predictions = tf.nn.softmax(logits)
        softmax_cross_entropy = tf.nn.
            softmax_cross_entropy_with_logits(logits=logits,
             labels=tf_y_batch)
        loss = tf.reduce_mean(softmax_cross_entropy) +
            params.l2_lambda * l2_loss

        # Optimizer.
        optimizer = tf.train.AdamOptimizer(
            learning_rate = learning_rate
        ).minimize(loss)

    with tf.Session(graph = graph) as session:
        session.run(tf.global_variables_initializer())

        # A routine for evaluating current model parameters
        def get_accuracy_and_loss_in_batches(X, y):
            p = []
            sce = []
            batch_iterator = BatchIterator(batch_size =
               128)
            for x_batch, y_batch in batch_iterator(X, y):
                [p_batch, sce_batch] = session.run([
                   predictions, softmax_cross_entropy],
                   feed_dict = {
                        tf_x_batch : x_batch,
                        tf_y_batch : y_batch,
                        is_training : False
                    }
                )
                p.extend(p_batch)
                sce.extend(sce_batch)
            p = np.array(p)
            sce = np.array(sce)
            accuracy = 100.0 * np.sum(np.argmax(p, 1) == np
               .argmax(y, 1)) / p.shape[0]
            loss = np.mean(sce)
            return (accuracy, loss)

        # If we chose to keep training previously trained
           model, restore session.
        if params.resume_training:
            try:
                tf.train.Saver().restore(session, paths.
                   model_path)
            except Exception as e:
                log("Failed restoring previously trained
                   model: file does not exist.")
                pass

        saver = tf.train.Saver()
        early_stopping = EarlyStopping(tf.train.Saver(),
           session, patience = params.
           early_stopping_patience, minimize = True)
        train_loss_history = np.empty([0], dtype = np.
           float32)
        train_accuracy_history = np.empty([0], dtype = np.
           float32)
        valid_loss_history = np.empty([0], dtype = np.
           float32)
        valid_accuracy_history = np.empty([0], dtype = np.
           float32)
        if params.max_epochs > 0:
            log("================= TRAINING
               =================")
        else:
            log("================== TESTING
               =================")
        log(" Timestamp: " + get_time_hhmmss())
        log.sync()

        for epoch in range(params.max_epochs):
            current_epoch = epoch
            # Train on whole randomised dataset in batches
            batch_iterator = BatchIterator(batch_size =
               params.batch_size, shuffle = True)
            for x_batch, y_batch in batch_iterator(X_train,
                y_train):
                session.run([optimizer], feed_dict = {
                        tf_x_batch : x_batch,
                        tf_y_batch : y_batch,
                        is_training : True
                    }
                )

            # If another significant epoch ended, we log
               our losses.
            if (epoch % params.log_epoch == 0):
                # Get validation data predictions and log
                   validation loss:
```

```
                valid_accuracy, valid_loss =
                   get_accuracy_and_loss_in_batches(X_valid
                   , y_valid)

                # Get training data predictions and log
                   training loss:
                train_accuracy, train_loss =
                   get_accuracy_and_loss_in_batches(X_train
                   , y_train)

                if (epoch % params.print_epoch == 0):
                    log("-------------- EPOCH %4d/%d
                       --------------" % (epoch, params.
                       max_epochs))
                    log("     Train loss: %.8f, accuracy:
                       %.2f%%" % (train_loss,
                       train_accuracy))
                    log("Validation loss: %.8f, accuracy:
                       %.2f%%" % (valid_loss,
                       valid_accuracy))
                    log("      Best loss: %.8f at epoch %d"
                        % (early_stopping.
                       best_monitored_value, early_stopping
                       .best_monitored_epoch))
                    log("   Elapsed time: " +
                       get_time_hhmmss(start))
                    log("      Timestamp: " +
                       get_time_hhmmss())
                    log.sync()
            else:
                valid_loss = 0.
                valid_accuracy = 0.
                train_loss = 0.
                train_accuracy = 0.
            valid_loss_history = np.append(
               valid_loss_history, [valid_loss])
            valid_accuracy_history = np.append(
               valid_accuracy_history, [valid_accuracy])
            train_loss_history = np.append(
               train_loss_history, [train_loss])
            train_accuracy_history = np.append(
               train_accuracy_history, [train_accuracy])

            if params.early_stopping_enabled:
                # Get validation data predictions and log
                   validation loss:
                if valid_loss == 0:
                    _, valid_loss =
                       get_accuracy_and_loss_in_batches(
                       X_valid, y_valid)
                if early_stopping(valid_loss, epoch):
                    log("Early stopping.\nBest monitored
                       loss was {:.8f} at epoch {}.".format
                       (
                        early_stopping.best_monitored_value
                           , early_stopping.
                           best_monitored_epoch
                    ))
                    break

        # Evaluate on test dataset.
        test_accuracy, test_loss =
           get_accuracy_and_loss_in_batches(X_test, y_test)
        valid_accuracy, valid_loss =
           get_accuracy_and_loss_in_batches(X_valid,
           y_valid)
        log("=============================================="
           )
        log(" Valid loss: %.8f, accuracy = %.2f%%)" % (
           valid_loss, valid_accuracy))
        log(" Test loss: %.8f, accuracy = %.2f%%)" % (
           test_loss, test_accuracy))
        log(" Total time: " + get_time_hhmmss(start))
        log("  Timestamp: " + get_time_hhmmss())

        # Save model weights for future use.
        saved_model_path = saver.save(session, paths.
           model_path)
        log("Model file: " + saved_model_path)
        np.savez(paths.train_history_path,
           train_loss_history = train_loss_history,
           train_accuracy_history = train_accuracy_history,
            valid_loss_history = valid_loss_history,
           valid_accuracy_history = valid_accuracy_history)
        log("Train history file: " + paths.
           train_history_path)
        log.sync(notify=True, message="Finished training
           with *%.2f%%* accuracy on the testing set (loss
           = *%.6f*)." % (test_accuracy, test_loss))

        plot_learning_curves(params)
        log.add_plot(notify=True, caption="Learning curves"
           )

        pyplot.show()
```

A.3 Adversarial Attack

Deepfool attack

```
def deepfool_attack(sess, x, predictions, logits, grads,
   sample, nb_candidate, overshoot, max_iter, clip_min,
   clip_max, feed=None):
    """
    TensorFlow implementation of DeepFool.
    Paper link: see https://arxiv.org/pdf/1511.04599.pdf
    param sess: TF session
    param x: The input placeholder
    param predictions: The model's sorted symbolic output
       of logits, only the top nb_candidate classes are
       contained
    param logits: The model's unnormalized output tensor (
       the input to the softmax layer)
    param grads: Symbolic gradients of the top nb_candidate
        classes, procuded from gradient_graph
    param sample: Numpy array with sample input
    param nb_candidate: The number of classes to test
       against, i.e., deepfool only consider nb_candidate
       classes when attacking(thus accelerate speed). The
       nb_candidate classes are chosen according to the
       prediction confidence during implementation.
    param overshoot: A termination criterion to prevent
       vanishing updates
    param max_iter: Maximum number of iteration for
       DeepFool
    param clip_min: Minimum value for components of the
       example returned
    param clip_max: Maximum value for components of the
       example returned
    return: Adversarial examples
    """
    import copy

    adv_x = copy.copy(sample)
    # Initialize the loop variables
    iteration = 0
    current = model_argmax(sess, x, logits, adv_x, feed)
    if current.shape == ():
        current = np.array([current])
    w = np.squeeze(np.zeros(sample.shape[1:]))  # same
       shape as original image
    r_tot = np.zeros(sample.shape)
    original = current  # use original label as the
       reference

    while (np.any(current == original) and iteration <
       max_iter):
        gradients = sess.run(grads, feed_dict={x: adv_x})
        predictions_val = sess.run(predictions, feed_dict={
           x: adv_x})
        for idx in range(sample.shape[0]):
            pert = np.inf
            if current[idx] != original[idx]:
                continue
            for k in range(1, nb_candidate):
                w_k = gradients[idx, k,...] - gradients[idx
                   , 0,...]
                f_k = predictions_val[idx, k] -
                   predictions_val[idx, 0]
                # adding value 0.00001 to prevent f_k = 0
                pert_k = (abs(f_k) + 0.00001) / np.linalg.
                   norm(w_k.flatten())
                if pert_k < pert:
                    pert = pert_k
                    w = w_k
            r_i = pert*w/np.linalg.norm(w)
            r_tot[idx, ...] = r_tot[idx, ...] + r_i

        adv_x = np.clip(r_tot + sample, clip_min, clip_max)
        current = model_argmax(sess, x, logits, adv_x, feed
           )
        if current.shape == ():
            current = np.array([current])
        # Update loop variables
        iteration = iteration + 1

    adv_x = np.clip((1+overshoot)*r_tot + sample, clip_min,
        clip_max)
    return adv_x
```

Generating adversarial set

```
import pickle

train_processed_dataset_file="data/test_processed.p"
adv_image_dataset_file="data/adv_image_test.p"
X_adv,y_adv=load_pickled_data(train_processed_dataset_file
   ,['features','labels'])

parameters = Parameters(
    # Data parameters
    num_classes = 43,
    image_size = (32, 32),
    # Training parameters
    batch_size = 256,
    max_epochs = 1000,
    log_epoch = 1,
    print_epoch = 1,
    # Optimisations
    learning_rate_decay = False,
    learning_rate = 0.0001,
    l2_reg_enabled = True,
    l2_lambda = 0.0001,
    early_stopping_enabled = True,
    early_stopping_patience = 100,
    resume_training = True,
    # Layers architecture
    conv1_k = 7, conv1_d = 16, conv1_p = 0.9,
    conv2_k = 7, conv2_d = 32, conv2_p = 0.8,
    conv3_k = 5, conv3_d = 64, conv3_p = 0.7,
    conv4_k = 3, conv4_d = 128, conv4_p = 0.6,
    fc4_size = 1024, fc4_p = 0.5,
    adv_training=None,
    perturbed_ratio= None
)
```

```
paths = Paths(parameters)

# Build the graph
graph = tf.Graph()
with graph.as_default():
    # Input data. For the training data, we use a
       placeholder that will be fed at run time with a
       training minibatch.
    tf_x = tf.placeholder(tf.float32, shape = (None,
       parameters.image_size[0], parameters.image_size[1],
       1))
    is_training = tf.constant(False)

    with tf.variable_scope(paths.var_scope):
        adv_logits=model_pass(tf_x, parameters, is_training
           )
        adv_preds = tf.reshape(tf.nn.top_k(adv_logits, k
           =10)[0],[-1, 10])
        adv_grads = tf.stack(jacobian_graph(adv_preds, tf_x
           , 10), axis=1)

    with tf.Session(graph=graph) as sess:
        sess.run(tf.global_variables_initializer())
        tf.train.Saver().restore(sess, paths.model_path)
        batch_iterator=BatchIterator(batch_size=128)
        f=0
        for X_batch,y_batch in batch_iterator(X_adv,y_adv):
            f=f+1
            if f%10==0:
                print(f)
            adv_image_batch=deepfool_attack(sess, tf_x,
               adv_preds, adv_logits, adv_grads, X_batch,
               nb_candidate=10,overshoot=0.02, max_iter=50,
                clip_min=0, clip_max=1, feed=None)
            if f==1:
                adv_image=adv_image_batch
            else:
                adv_image=np.concatenate((adv_image,
                   adv_image_batch),axis=0)

print(adv_image.shape)
pickle.dump({
        "features" : adv_image,
        "labels" : y_adv
    }, open(adv_image_dataset_file, "wb" ) )
print("Adversarial images dataset saved in",
   adv_image_dataset_file)
```