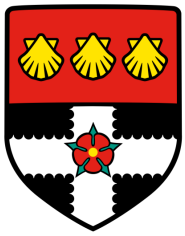

University of Reading

Department of Computer Science

Robustness Analysis Of Neural Networks

Yuqiao Chen

Supervisor: Dr Varun Ojha

A report submitted in partial fulfilment of the requirements of
the University of Reading for the degree of
Master of Science in *Data Science and Advanced Computing*

September 14, 2022

Declaration

I, Yuqiao Chen, of the Department of Computer Science, University of Reading, confirm that this is my own work and figures, tables, equations, code snippets, artworks, and illustrations in this report are original and have not been taken from any other person's work, except where the works of others have been explicitly acknowledged, quoted, and referenced. I understand that if failing to do so will be considered a case of plagiarism. Plagiarism is a form of academic misconduct and will be penalised accordingly.

I give consent to a copy of my report being shared with future students as an exemplar.

I give consent for my work to be made available more widely to members of UoR and public with interest in teaching, learning and research.

Yuqiao Chen
September 14, 2022

Abstract

More and more machine learning models are being deployed in production environments, with various types of application scenarios, and with them, more and more attacks against the models. This paper aims to compare common types of attacks and defences and test the models' robustness by comparing experimental results. These results are done by combining different attacks developed for different directions and defences developed for specific attacks. Throughout the experiments, the different attacks have good results for the original model, reducing the accuracy rate from about 70% to about 5%-10%. Because the defences are generally developed for specific attacks, the defence against a specific attack can re-increase the low accuracy rate, but the generalisation is very weak and has little effect against other types of attacks. In general, the robustness of a model is relative, and the depth and complexity of the model itself, the different types and strengths of the attacks it faces, and the defence methods implemented by the model itself all have a different impact on the results.

Keywords: Robustness, Attacks, Defences, Model Building, Tensorflow.

Project code repository:
https://gitlab.act.reading.ac.uk/vo836354/msc_final_project

Report's total word count: 10,009

Contents

List of Figures

List of Tables

List of Abbreviations

CNN	Convolutional Neural Networks
DenseNet	Densely Connected Convolutional Networks
ResNet	Residual Network
VGG	Very Deep Convolutional Networks
FSGM	Fast Gradient Sign Method
BIM	Basic Iterative Method
JSMA	Jacobian-based Saliency Maps Attacks
HSJ	HopSkipJump

Chapter 1

Introduction

1.1 Background

As technologies related to machine learning continue to mature, more and more instances of the model are being deployed in the cloud or on the client side for text(Xu et al., 2021), image(Balaji and Lavanya, 2019), video(Camastra and Vinciarelli, 2015), data analysis(Mahdavinejad et al., 2018) and more. As the number of users and usage rises, security issues come to the fore. Artificial intelligence security can be seen as a big issue because of its widespread civilian use. Artificial intelligence and machine learning are widely used in imaging, and there is no shortage of third parties that can attack and test the models deployed for different purposes, which can cause problems for the original deployer and reduce the correct recognition rate. Neural networks are increasingly used as classical models, and most emerging networks are based on neural networks with different modifications, somewhat modifying the structure and somewhat modifying the hyperparameters, to achieve the goal of high accuracy. Because of this, the tug-of-war between attacks and defences around neural networks is carried out. Studying and ensuring that neural network models remain active and reliably accurate in the midst of an attack has become a popular research topic in recent years.

1.2 Problem statement

The problem faced by this project is to test the robustness of the image detection model.

Traditional machine learning typically deploys models after the debugging and building phase of the model to improve its accuracy to a suitable value, and because of the wide range of attack types against the model, the model is rarely built with targeted attacks on the model in mind. One of the questions we will investigate is how well such primitive models perform in terms of prediction accuracy after being subjected to a targeted attack.

After the model has been attacked, defensive measures are inevitably needed to ensure the normal operation of the model. It is then necessary to develop or select a corresponding method to defend the model according to the type of attack suffered, and the performance after defence is what this project needs to test.

1.3 Aims

Our goal is to perform an analysis of the robustness of neural networks. It can be divided into three categories: the construction of the model, the performance of the model after being attacked and the performance of the model after implementing the defense measures.

The experiment will first study the construction of the model, that is, select the appropriate neural network model and the data set used in the test. After debugging the model and compiling and building it, the data will be input into the model for training, and the trained model will be saved for introduction in the later experiment.

Then to the stage where the model is attacked. We will find and choose suitable attack methods to test the model, use different sizes of data and adjust the hyperparameters of the attack algorithm to achieve a variety of comparisons, and finally calculate the accuracy of the model, that is, test the robustness of the model.

Finally, there is the defensive phase. To find a suitable defense algorithm to protect the model, and then test the accuracy of the model after the implementation of the defense algorithm to test the robustness of the model.

1.4 Solution approach

The project was divided into three phases in total. Model building, attack testing, defense testing.

Model building phase. We selected "cifar-10"(Krizhevsky et al., 2009) as the dataset for the whole project, which includes a total of 60000 32*32 RGB images with a total of 10 categories. In the selection of the model, we choose the classic convolutional neural network(Krizhevsky et al., 2012), VGG19(Simonyan and Zisserman, 2014), ResNET(He et al., 2016*a*) and DenseNet(Huang et al., 2017) as the test. Among them, VGG19, ResNET and DenseNet all use the transfer learning method, which only introduces the structure and "ImageNet" parameters, while changing the input layer and output layer. After all the selections are completed, the model suitable for this experiment is constructed, and then the "CIFAR-10" data set is put into the model for training, and finally the model file is saved.

In the model attack phase, the attack types can be divided into white-box attacks and black-box attacks. Due to the number of attacks on neural network methods is various and different methods, we in the experiments, the use more different types of attacks, use different attack algorithms to attack several kinds of model, we build and high granular attack test parameters for different parameters, which use a lot of workforces.

During the model defence phase. As the defence algorithms were all targeted and low in number, we then selected a few very representative defence algorithms to test.

1.5 Summary of contributions and achievements

The performance of neural networks with different structures was tested on the "Cifar-10" dataset. A number of different types of attack algorithms were tested and the comparison of different hyperparameters was tested for each algorithm. The robustness of the neural network was tested against different types of defence algorithms.

In the model building phase. The performance of neural networks with different architectures is tested on the "CIFAR-10" dataset. The convolutional neural network, VGG19, ResNet-50, ResNet V2 and DenseNet were introduced to test. The accuracy of these models on the "CIFAR-10" test set is about 60%, and the details can be seen in Figure 4.14.1. And store the trained model here.

A number of different types of attack algorithms were tested. They are black-box and white-box attacks, respectively, and the different hyperparameters of each algorithm are compared. White-box attacks include "FSGM"(Goodfellow et al., 2014), "I-FSGM"(Kurakin et al., 2018), "R-FSGM"(Papernot et al., 2017), "JSMA(Papernot et al., 2016)", "C&W(Carlini and

Wagner, 2017)", "DeepFool(Moosavi-Dezfooli et al., 2016)", and black-box attacks include "PixelAttack(Song et al., 2017)", "SquareAttack(Andriushchenko et al., 2020)", "GeoDA(Rahmati et al., 2020)", "HSJ(Chen et al., 2020)". They are very aggressive, can reduce the accuracy of the model about 60% significantly, the model is the lowest or even useless.

In the defence phase, the robustness of the neural network was tested against different types of defence algorithms. We tested the "Total Variance Minimization"(Guo et al., 2017) algorithm of the Preprocessor type, which was developed for FSGM(Goodfellow et al., 2014) and C&W(Carlini and Wagner, 2017), as well as the DeepFool(Moosavi-Dezfooli et al., 2016) attack algorithm, which in different models, all have good defence performance and can significantly improve the accuracy of the attacked model, but the effects are different. The "Reverse Sigmoid(Lee et al., 2019)" algorithm of the Postprocessor type is tested. The algorithm is to process and operate the built model, and the purpose is to prevent attackers or third parties from stealing the specific details of the model. After the application of the algorithm, it only plays a defence role under some attack algorithms and models. Moreover, unlike the "Total Variance Minimization" algorithm, the detection of this algorithm is not particularly intuitive. The "Madry's Protocol(Madry et al., 2017)" algorithm of Trainer type is tested in the project. The algorithm uses the original model's prediction label to retrain the model's principle and carries out the anti-resistance training of the model. However, the performance of the algorithm in this project is not good, and it needs more targeted development to achieve the originally expected effect. At the same time, the "SubsetScanningDetector(McFowland et al., 2013)" algorithm of Detector type was tested. The original goal of the algorithm was to detect adversarial data quickly. In this project, the algorithm performed well and could accurately detect data sets with adversarial images.

1.6 Organization of the report

The project as a whole is divided into 7 chapters.

The first chapter 1 introduces the background of the project, the problem to be tackled and the overall goal of the project. The chapter 2, reviews the foundations built in the previous papers and collates some of the attacks and defences against neural networks, we then continue to experiment on their basis. In chapter 3, the details of the specific implementation of the project are presented, introducing several neural network models and implementing the specific construction of a deep learning network for substantial comparison and testing of multiple types of attack and defence algorithms. The chapter 4, presents the overall results of the project and whether the expected goals were achieved. chapter 5 carries out a reflection on the project as a whole. chapter 6 carries out a summary and future work that can be built upon.

Chapter 2

Literature Review

2.1 Review of state-of-the-art

As machine learning continues to evolve and the number of domains covered grows, more and more attention is gathered on whether the models are robust enough.

Yuan et al. (2019) summarized the study of adversarial instances for DNNs. Strategies for creating adversarial examples were summarized, and a taxonomy based on these methods was proposed for these approaches. Under this taxonomy, they studied the application of adversarial examples and elaborated on counter-adversarial examples. Moreover, they discuss three main challenges and examples of potential solutions for adversarial examples.

Silva and Najafirad (2020) investigated strategies for implementing adversarial training algorithms to secure machine learning algorithms. They provide a taxonomy to classify adversarial attacks and defences, formulate robustness optimisation problems in a min-max setting and divide them into three subcategories. We survey the latest and most important results generated by adversarial examples and approaches to derive robustness formally certification by solving the optimisation problem exactly or using approximations to upper or lower bounds and discussing the most recent challenges faced by the algorithms.

By presenting three novel attack algorithms that are 100% successful on both distilled and undistilled neural networks, Carlini and Wagner (2017) show that defensive distillation does not greatly boost the resilience of the neural network. In general, the proposed assaults outperform earlier adversarial sample generation methods because they are suited to the three distance metrics utilised in the prior research. The effectiveness of their assault will serve as a baseline for further defence efforts to build neural networks that can withstand adversarial sampling.

2.2 relation between project and literature

These papers all focus on analysing the robustness of image recognition models. Inspired by these papers, in this project, we will combine some of the attacks and defences that appear in the papers and additionally add some emerging methods that are not mentioned in the papers to experiment with.

Building on Carlini and Wagner (2017) ideas, we can use them to find similar attack algorithms and perform tests of the robustness of the neural network model when testing defence algorithms where possible.

Furthermore, in these papers, there is basically nothing about how to defend the model, and we will add this section later in the project.

2.3 Critique of the review

For Yuan et al. (2019), their overall code is too academic and only designed to build complete data flows and outputs without generalization and reproducibility.

In Silva and Najafirad (2020) papers, most of them are theoretical presentations and conclusions, and no corresponding code implementations are provided.

2.4 Summary

Being inspired by these papers gave us the initial direction for the project. We will, however, test more robust attacks and defences against image recognition models in more detail in this thesis. And we will add a full data comparison and code implementation to these papers.

Chapter 3

Methodology

In this section we will divide the whole project into three parts. These are model building, targeted attacks and selective defence.

We used CIFAR-10 as the dataset for the whole project, the choice of test set may have varied slightly at different stages.

3.1 Model Build

In this sub-section we present the complete process of the models built for this project, the reasons why certain models have been chosen, the characteristics of the models and other information.

A total of five models were chosen to be built for the project, they are CNN(Krizhevsky et al., 2012), DenseNet(Huang et al., 2017), ResNet(He et al., 2016*a*), ResNet_V2(He et al., 2016*b*), VGG(Simonyan and Zisserman, 2014). Where ResNet_V2(He et al., 2016*b*) was added mid to late in the project to compare with ResNet, which was introduced because it did not perform as expected under certain attacks and defences.

All models are done with Tensorflow as the back-end. The implementation of several models also differs, with CNN being the model structure we have implemented by hand, while for several other models, we have used migration learning to complete the model implementation.

In the building blocks of the model, we use the TPU cluster on Google Colab for acceleration, so the implementation of the code can vary and may require slight modifications to the code if we want to reproduce the model structure.

Tensorflow2 is very easy to use with its three ways of defining models, the Sequential API, the Functional API and the Subclassing API. The methods used are explained in detail when a model is involved.

In the training phase, we use the "datasets.cifar10.load_data()" function that comes with Tensorflow to obtain the segmented cifar10 training and test sets.

The file for the model defence test is here.

3.1.1 CNN

For building CNNs, we use the Sequential API, which means building models with Keras. Building a model with Keras is as easy as "putting Lego blocks together". Apart from matching our perceived model, the models built this way are easy to debug.

CNN structure

Thanks to our use of the Sequential API, we can very intuitively get the structure of the CNN we are using from the code.

Code

```
def create_model():
 return tf.keras.Sequential(
      [tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
       tf.keras.layers.MaxPooling2D((2, 2)),
       tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
       tf.keras.layers.MaxPooling2D((2, 2)),
       tf.keras.layers.Conv2D(128, (3, 3), activation='relu'),
       tf.keras.layers.MaxPooling2D((2, 2)),
       tf.keras.layers.Flatten(),
       tf.keras.layers.Dense(64, activation='relu'),
       tf.keras.layers.Dense(10, activation='softmax')
       ])
```

We use tables to show the overall structure of the model:

Table

CNN
6 weight layers
input(32*32 RGB image)
conv2-32
maxpooling
conv2-64
maxpooling
conv2-128
maxpooling
Flatten
FC-64
FC-10
soft-max

The accuracy of the model under the CIFAR-10 test set was approximately 63%.

3.1.2 VGG19

At the Oxford's Visual Geometry Group, the group proposed VGG. The network was the focus of related work at ILSVRC 2014, with the primary goal being to show that a network's ultimate performance may be somewhat impacted by its depth. Only the network's depth distinguishes the two VGG structures, VGG16 and VGG19, which are essentially similar to one another.

This project uses the most common VGG19 as our experimental model. Because we use "transfer learning," we only refer to the structure of VGG19 while changing its input and fully connected layers and using the parameters of ImageNet.

This subchapter uses the keras constructed function, "tf.keras.applications.VGG19".

VGG19 structure

Below is code and a table to demonstrate the model structure of VGG19.

Code

```
def create_model():
  # Initialize the ResNet50 model, without fully-connected layers
  model = tf.keras.applications.vgg19.VGG19(include_top=False , weights='imagenet',
                                            input_shape=(32, 32, 3))
  # Frozen layer
  model.trainable = False

  inputs = keras.Input(shape=(32, 32, 3))

  x = model(inputs, training=False)
  # Convert features of shape `model.output_shape[1:]` to vectors
  x = keras.layers.GlobalAveragePooling2D()(x)

  # let's add a fully-connected layer
  #x = layers.Dense(64, activation='relu')(x)

  # A Dense classifier with a single unit (10 classification)
  outputs = layers.Dense(10)(x)
  model = keras.Model(inputs, outputs)

  return model
```

Table

ConvNet Configuration
19 layer weights

input size (32*32 RGB image)
conv3-64conv3-64
max pool
conv3-128conv3-128
max pool
conv3-256conv3-256conv3-256conv3-256
max pool
conv3-512conv3-512 conv3-512 conv3-512
max pool
conv3-512 conv3-512 conv3-512 conv3-512
GlobalAveragePooling2D
FC-10
soft-max

The accuracy of the model under the CIFAR-10 test set was approximately 58%.

3.1.3 DenseNet

DenseNet is a neural network created to address the problem of gradient loss when a neural network is too deep and too broad and is a neural network commonly used for testing.

Since we are using "transfer learning", we import the structure of VGG19, changing both its input and fully connected layers and using the parameters of "ImageNet".

This subchapter uses the keras constructed function, "tf.keras.applications.DenseNet121".

DenseNet structure

Below is code and a table to demonstrate the model structure of VGG19.

Code

```
def create_model():
    # Initialize the ResNet50 model, without fully-connected layers
    model = tf.keras.applications.densenet.DenseNet121(include_top=False,
        weights='imagenet', input_shape=(32, 32, 3))
    # Frozen layer
    model.trainable = False

    inputs = keras.Input(shape=(32, 32, 3))

    x = model(inputs, training=False)
```

```
        # Convert features of shape `model.output_shape[1:]` to vectors
        x = keras.layers.GlobalAveragePooling2D()(x)

         # A Dense classifier with a single unit (10 classification)
         outputs = layers.Dense(10)(x)

        model = keras.Model(inputs, outputs)

        return model
```

Table

DenseNet121 Configuration
19 layer weights

input size (32*32 RGB image)
DenseNet121
GlobalAveragePooling2D
FC-10
soft-max

The accuracy of the model under the CIFAR-10 test set was approximately 65%.

3.1.4 ResNet

In this sub-module, we refer to the ResNet-50 network, a very classical and widely utilised neural network.

ResNet has been developed for specific problem-solving. The first problem is vanishing/-exploding gradients, which can be solved by BN and better initialisation of the network; the second problem is degradation, i.e. when the network is saturated with layers, adding more layers leads to difficulties in optimisation and greater training and prediction errors. The authors solve the degradation problem by adding a residual structure.

Since we are using "transfer learning", we import the structure of ResNet, changing both its input and fully connected layers and using the parameters of "ImageNet".

This subchapter uses the keras constructed function, "tf.keras.applications.ResNet50".

Code

```
    def create_model():
        # Initialize the ResNet50 model, without fully-connected layers
        model = ResNet50(include_top=False , weights='imagenet',
            input_shape=(32, 32, 3), pooling='max')

        model.trainable = False

        inputs = keras.Input(shape=(32, 32, 3))

        x = model(inputs, training=False)
        # Convert features of shape `base_model.output_shape[1:]` to vectors
```

```
        x = keras.layers.Flatten()(x)


        # A Dense classifier with a single unit (binary classification)
        outputs = keras.layers.Dense(10)(x)
        model = keras.Model(inputs, outputs)

        return model
```

Table

ResNet-50 Configuration
19 layer weights

input size (32*32 RGB image)
ResNet-50
Flatten
FC-10
soft-max

The accuracy of the model under the CIFAR-10 test set was approximately 42%.

We guess that the low correctness of the ResNet-50 model is due to the use of the "imagenet" parameters and the freezing of the parameters of the original middle layer during training without performing the same operation as the "imagenet" on the Cifar-10 dataset.

3.1.5 ResNet_V2

The ResNet-V2 network is introduced in this sub-module, a very classical and widely utilised neural network. This model was introduced late in the project and was only used to form a control group with the low correct rate of ResNet-50.

Since we are using "transfer learning", we import the structure of ResNet, changing both its input and fully connected layers and using the parameters of "ImageNet".

This subchapter uses the keras constructed function, "tf.keras.applications.ResNet50V2".

Code

```
    def create_model():
        # Initialize the ResNet50 model, without fully-connected layers
        model = ResNet50V2(include_top=False , weights='imagenet',
            input_shape=(32, 32, 3), pooling='max')

        model.trainable = False

        inputs = keras.Input(shape=(32, 32, 3))

        x = model(inputs, training=False)
        # Convert features of shape `base_model.output_shape[1:]` to vectors
        x = keras.layers.Flatten()(x)

        x = keras.layers.Dense(128)(x)
```

```
    # A Dense classifier with a single unit (binary classification)
    outputs = keras.layers.Dense(10)(x)
    model = keras.Model(inputs, outputs)

    return model
```

Table

ResNet V2 Configuration
19 layer weights

input size (32*32 RGB image)
ResNet V2
Flatten
FC-128
FC-10
soft-max

The accuracy of the model under the CIFAR-10 test set was approximately 61%.

3.2 Model Attacks

We go into great depth on the many ways the model has been attacked in this sub-module. While some of the approaches make use of the Adversarial Robustness Toolbox(Nicolae et al., 2018), others are re-implemented algorithms.

Black Box Attacks and White Box Attacks are used in this research to concentrate on Evasion Attacks. The major testing tools in this project, which focuses on evasion attacks, are both black-box and white-box assaults.

We will test each model, in turn, using each approach in this subsection, gauging the model's accuracy after an assault and visualising the input data following perturbations.

3.2.1 FSGM (WhiteBox)

FGSM (fast gradient sign method)(Goodfellow et al., 2014) among image attack algorithms is very classical. This paper, published in ICLR 2015, uses gradients to generate attack noise, the core idea of which is shown in Figure 3.1. The model classifies it as a gibbon, although it still looks like a panda.

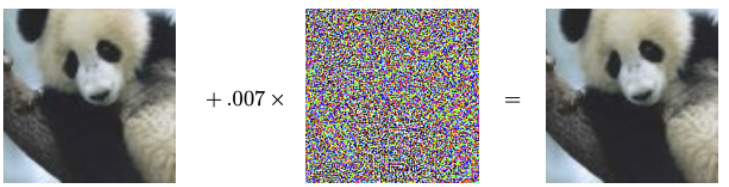

Figure 3.1: FSGM-adversarial-example

Formula

$$x' = x + \epsilon \cdot \text{sign}(\nabla_x J(x, y))$$

Figure 3.2: FSGM-Formula

The adversarial samples generated by FSGM can be considered the original input image, plus a coefficient multiplied by the value of the inverse maximum of the model gradient.

Code

```
loss_object = tf.keras.losses.CategoricalCrossentropy()


# Function to calculate adversary noise
def generate_adversary(image, label):
  image = tf.cast(image, tf.float32)

  with tf.GradientTape() as tape:
    tape.watch(image)
    prediction = model(image)[0]
    loss = loss_object(label, prediction)
  gradient = tape.gradient(loss, image)
  sign_grad = tf.sign(gradient)

  return sign_grad
```

FSGM was implemented independently for this project. With the help of TensorFlow's automatic differentiation, we can track the gradients of the model quickly and easily.

Compare

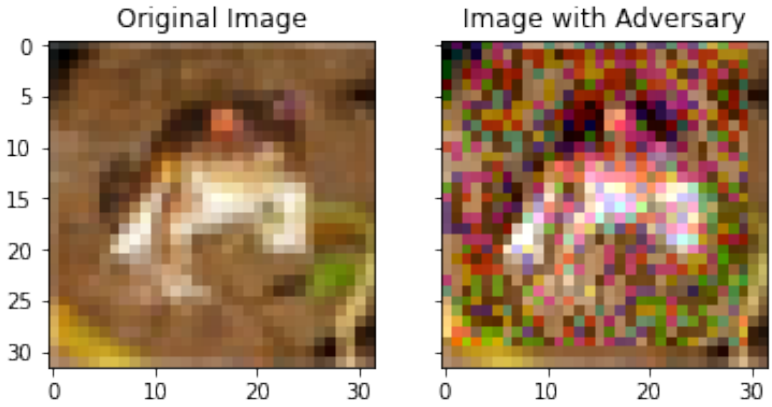

Figure 3.3: FSGM-Compare

The FSGM has only one parameter, epsilon, which can be adjusted to balance the differences in intensity and apparent distance of the generated confrontation image with the original image.

Although FSGM can generate adversarial images for deception models, the generated adversarial images may differ too much from the original ones, so how to weigh the proportion of perturbations is the main issue to be considered by FSGM.

The FSGM attack has a very good effect. In the CNN we constructed, epsilon = 0.01 cuts the accuracy of the model in half, and the higher the epsilon value, the stronger the attack, leaving the model with a minimum accuracy of only about 2%. this project have used different values to measure the effect of the attack in our tests, the details of which are here.

A comparison of the effectiveness of the attack algorithm on different models is here.

3.2.2 I-FSGM (WhiteBox)

BIM(Kurakin et al., 2018) is also known as I-FSGM. BIM is an extension and improvement of FGSM, not just one calculation like FSGM, BIM performs multiple small steps of FSGM.

Formula

$$\boldsymbol{X}_0^{adv} = \boldsymbol{X}, \quad \boldsymbol{X}_{N+1}^{adv} = Clip_{X,\epsilon}\left\{\boldsymbol{X}_N^{adv} + \alpha\,\mathrm{sign}\left(\nabla_X J(\boldsymbol{X}_N^{adv}, y_{true})\right)\right\}$$

Figure 3.4: FSGM-Formula

Code

```
def I_FSGM(image, label, alpha, epsilon):
    '''Performs I-FSGM attack on a given image and label
    Formula:
        X_0_adv = X
        X_N+1_adv = Clip(X_N_adv + alpha * sign(grad(X_N_adv, Y_true)))
        alpha selected from 1
        iterations to be min(epsilon+4, 1.25 * epsilon)

    Args:
        image(np.array):  input image, shape like (height, width, channels)
        label(int): label of the image, shape like (nb_classes, )
        alpha(float): alpha parameter for I-FSGM
        epsilon(int): epsilon parameter for I-FSGM

    Examples:
        >>> tmp = I_FSGM(train_images[0], train_labels[0], 0.1, 5)

    Returns:
        adv_image(np.array): adversarial image, shape like (height, width, channels)
    '''

    image = image.reshape(1,32,32,3)

    for i in range(int(min(epsilon+4, 1.25 * epsilon))):
```

```
        # Generate adversarial noise
        # Because the original image is already between [0, 1]
        # the resulting pixels are also divided by 255
        sign_grad = (generate_adversary(image, label) / 255).numpy()

        img_adv = (image + alpha * sign_grad).reshape(32,32,3)

        for j in range(len(img_adv)):
            for k in range(len(img_adv[j])):
                for l in range(len(img_adv[j][k])):

                    img_adv[j][k] = np.clip(img_adv[j][k] , image[0][j][k]
                        - epsilon/255, image[0][j][k]  + epsilon/255)

                    if img_adv[j][k][l] > 1:
                        img_adv[j][k][l] = 1
                    elif img_adv[j][k][l] < 0:
                        img_adv[j][k][l] = 0

        image = img_adv.reshape(1,32,32,3)

    return image.reshape(32,32,3)
```

There are two parameters that control the entire function. epsilon" controls the range of the calculation and "Alpha" controls the scale of each small step.

Compare

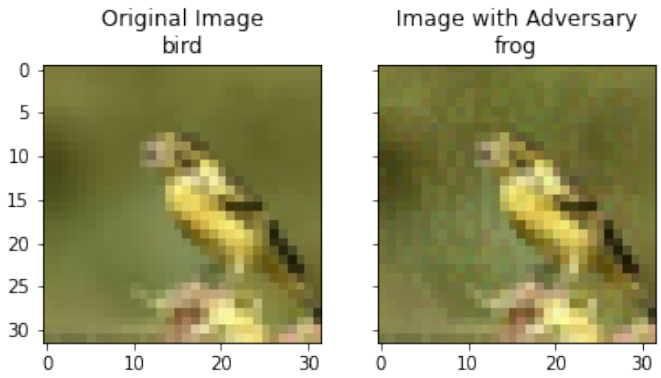

Figure 3.5: I-FSGM_Compare

Visually speaking, I-FSGM largely alleviates the problem of excessive disparity between the FSGM-generated adversarial image and the original image by multiple small steps.

The I-FSGM attack also has a very good effect. Since the I-FSGM is determined by two parameters together, this project have used different values to measure the effect of the attack in our tests, the details of which are here. After using I-FSGM as the attack algorithm, the accuracy of the model plummeted, with the lowest accuracy even approaching 0.

A comparison of the effectiveness of the attack algorithm on different models is here.

3.2.3 R-FSGM (WhiteBox)

The authors advise that before computing the loss of the first-order derivative with respect to the input in R-FGSM(Kurakin et al., 2018), random perturbations taken from a Gaussian distribution be included.

Formula

$$\boldsymbol{X}_0^{adv} = \boldsymbol{X}, \quad \boldsymbol{X}_{N+1}^{adv} = Clip_{X,\epsilon}\left\{\boldsymbol{X}_N^{adv} + \alpha\,\mathrm{sign}\left(\nabla_X J(\boldsymbol{X}_N^{adv}, y_{true})\right)\right\}$$

Figure 3.6: R-FSGM-Formula

α is a further parameter in the R-FGSM formulation that regulates the size of the random disturbances taken from the normal distribution.

R-FGSM(Papernot et al., 2017) was created to go beyond gradient masking defences, which is a crucial idea in adversarial machine learning. Gradient masking methods mask or obfuscate the gradient of the model, making it more difficult for an attacker to determine the precise gradient dL/dx.

Code

```
loss_object = tf.keras.losses.CategoricalCrossentropy()

def R_FSGM(input_image, input_image_label, epsilon, alpha, Parameters_I):
    '''Create the adversarial example using the R_FSGM attack

    Formula:
        x_adv = x' + (epsilon - alpha) * sign(grad(x', Ytrue)),
                    where x' = x + alpha * sample(N(0**d, I**d))

    Args:
        input_image(numpy.ndarray): the original image, shape (32, 32, 3)
        input_image_label(numpy.ndarray): the original label, shape (10, )
        epsilon(float): the maximum perturbation
        alpha(float): the step size
        Parameters_I(int): the parameters of the attack

    Returns:
        input_image_adv(tonser): the adversarial example, shape (32, 32, 3)

    '''

    input_image = tf.cast(input_image, tf.float32)
```

```
    sample_from_Gaussian = tf.get_static_value(tfp.distributions.Normal(loc=0.,
                            scale=Parameters_I**2).sample(1)[0])
    input_image = input_image + alpha * (sample_from_Gaussian/255)

    with tf.GradientTape() as tape:
        tape.watch(input_image)
        prediction = model(tf.reshape(input_image, [1, 32, 32, 3]))[0]
        loss = loss_object(tf.cast(input_image_label, tf.float32), prediction)
    gradient = tape.gradient(loss, input_image)
    sign_grad = tf.sign(gradient)

    input_image = input_image + (epsilon - alpha) * sign_grad

    input_image = tf.clip_by_value(input_image, 0, 1)

    return input_image
```

After configuring the environment, you can use this code directly to generate the corresponding R-FSGM counter samples.

Compare

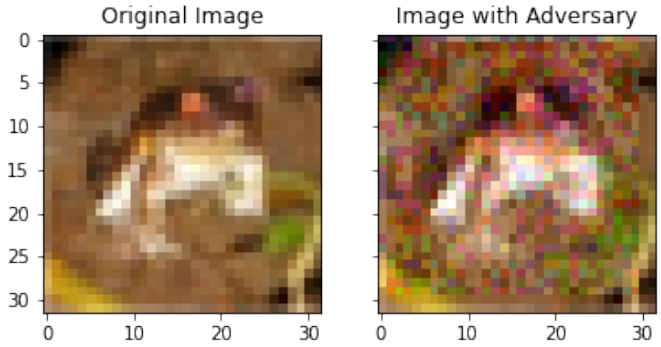

Figure 3.7: R-FSGM_Compare

A comparison of the effectiveness of the attack algorithm on different models is here.

3.2.4 DeepFool (WhiteBox)

Deepfool(Moosavi-Dezfooli et al., 2016) is an attack method based on hyperplane classification. It is well known that in binary classification problems, the hyperplane is the basis for classification, so to change the classification of sample x, the smallest perturbation is to move x to the hyperplane, and the distance from this sample to the hyperplane is the least costly place. The problem of multi-classification is similar. This gives a clearer picture of how far the

adversarial attack does just right and allows the robustness metric proposed by the authors to be used.

Formula

Algorithm 2 DeepFool: multi-class case

input: Image $\boldsymbol{x}$, classifier f.
output: Perturbation $\hat{\boldsymbol{r}}$.

Initialize $\boldsymbol{x}_0 \leftarrow \boldsymbol{x}$, $i \leftarrow 0$.
while $\hat{k}(\boldsymbol{x}_i) = \hat{k}(\boldsymbol{x}_0)$ **do**
  for $k \neq \hat{k}(\boldsymbol{x}_0)$ **do**
    $\boldsymbol{w}'_k \leftarrow \nabla f_k(\boldsymbol{x}_i) - \nabla f_{\hat{k}(\boldsymbol{x}_0)}(\boldsymbol{x}_i)$
    $f'_k \leftarrow f_k(\boldsymbol{x}_i) - f_{\hat{k}(\boldsymbol{x}_0)}(\boldsymbol{x}_i)$
  end for
  $\hat{l} \leftarrow \arg\min_{k \neq \hat{k}(\boldsymbol{x}_0)} \frac{|f'_k|}{\|\boldsymbol{w}'_k\|_2}$
  $\boldsymbol{r}_i \leftarrow \frac{|f'_{\hat{l}}|}{\|\boldsymbol{w}'_{\hat{l}}\|_2^2} \boldsymbol{w}'_{\hat{l}}$
  $\boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_i + \boldsymbol{r}_i$
  $i \leftarrow i + 1$
end while
return $\hat{\boldsymbol{r}} = \sum_i \boldsymbol{r}_i$

Figure 3.8: DeepFool-Formula

It should be noted that DeepFool's optimisation strategy is closely related to existing optimisation techniques. In the dichotomous case, it can be seen as Newton's iterative algorithm for finding the roots of a system of non-linear equations in the presence of uncertainty.

Code

```
def DeepFool(image, label, model, limit_times):

    label = label[0]

    image = tf.cast(image, tf.float32)
    image_ori = image

    store_list = []
    store_list.append(image_ori)
    i = 0

    while model(tf.reshape(store_list[i], [1, 32, 32, 3])).numpy().argmax()
            == model(tf.reshape(image_ori, [1, 32, 32, 3])).numpy().argmax()
            and i < limit_times:

        predict_vector = model(tf.reshape(store_list[i], [1, 32, 32, 3]))[0]
```

```
        gradient_ori = generate_gradient(store_list[i], label)
        predict_vector_ori = predict_vector[label]

        gradient_difference = []
        predict_vector_difference = []
        l_list = []


        # total 10 classification
        for j in range(10):
            predict_vector_difference.append(tf.math.abs(
                predict_vector[j] - predict_vector_ori))

            gradient = generate_gradient(store_list[i], j)
            gradient_difference.append(gradient - gradient_ori)

        for j in range(10):
            if j != label:
                tmp_1 = tf.math.abs(predict_vector_difference[j])
                tmp_2 = tf.math.reduce_euclidean_norm(gradient_difference[j])

                result = tf.math.divide(tmp_1, tmp_2)

                #print(f'list append {result}')

                l_list.append(result)
            else:
                l_list.append(1)

                #print(f'list append {0}')

        l = np.asarray(l_list).argmin()

        disturbance = tf.math.divide_no_nan(tf.math.abs(predict_vector_difference[l]),
                tf.math.square(tf.math.reduce_euclidean_norm(gradient_difference[l])))
        disturbance = tf.math.multiply_no_nan(disturbance, gradient_difference[l])

        store_list.append(tf.clip_by_value(store_list[i] + disturbance, 0, 1))
        i += 1

    true_disturbance = store_list[i] - image_ori

    return store_list[i], true_disturbance
    #return predict_vector_difference, gradient_difference
```

After configuring the environment, you can use this code directly to generate the corresponding DeepFool counter samples.

Compare

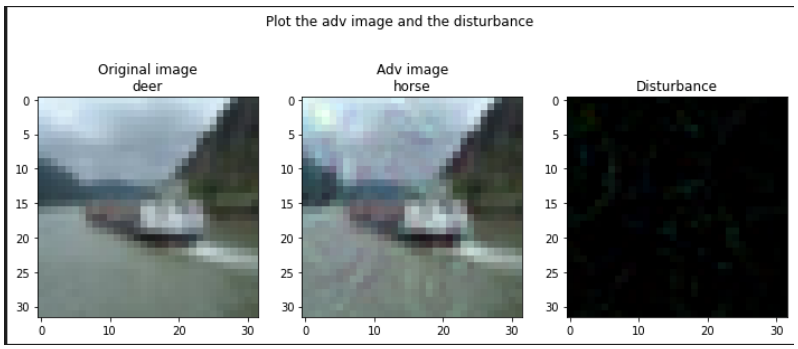

Figure 3.9: DeepFool_Compare

DeepFool is tested in detail later in the "Defense" module because each call to DeepFool uses a large amount of computational resources, and because this project lacks a GPU for computing, DeepFool can reduce the accuracy of the model by up to one-third with a small number of iterations. As the number of iterations increases, the congratulatory effect becomes stronger and stronger.

3.2.5 C&W (WhiteBox)

Instead of using the conventional cross-entropy loss, Carlini & Wagner(Carlini and Wagner, 2017) expand the L-BFGS(Szegedy et al., 2013) approach by changing the goal function. This paper successfully attacked the defensive distillation, which was better defended at the time, and exhaustively investigated the optimization methods of the objective function and objective function of the adversarial sample, which is extremely enlightening for the design of methods to generate the adversarial sample.

Formula

$$f(x') = \max(\max\{Z(x')_i : i \neq t\} - Z(x')_t, -\kappa)$$

Figure 3.10: C&W-Formula

Take note of the C&W attack's usage of a loss function. Take note of the notational shift, where f now denotes the classifier's loss function rather than the classifier itself. Here, t stands for the target misclassification label, $Z(x')$ represents the logarithm as it traverses the adversarial input (x'), and k is a constant that regulates the necessary confidence score. (Carlini and Wagner, 2017).

The goal of this objective function is to minimise the distance between the most probable category and the target category t. The optimisation process will end when the difference in logit between t and the runner-up category is at most $_k$ if t presently has the highest logit value. In other words, the confidence necessary for an adversarial example is controlled by $_k$. On the other hand, reducing "f" narrows the difference between the logits of the highest class and the target class if t does not have the highest logit. This decreases the confidence of the highest class or boosts the confidence of the target class. Finally, minimisation is the new objective of the optimisation problem.

Code

```
attack = CarliniL0Method(classifier=classifier)
x_test_adv = attack.generate(x=x_test_10)
```

Here we used the functions in the toolbox to generate an adversarial sample of C&W and used matplotlib to plot a comparison of the original image, the adversarial image and the perturbations.

Compare

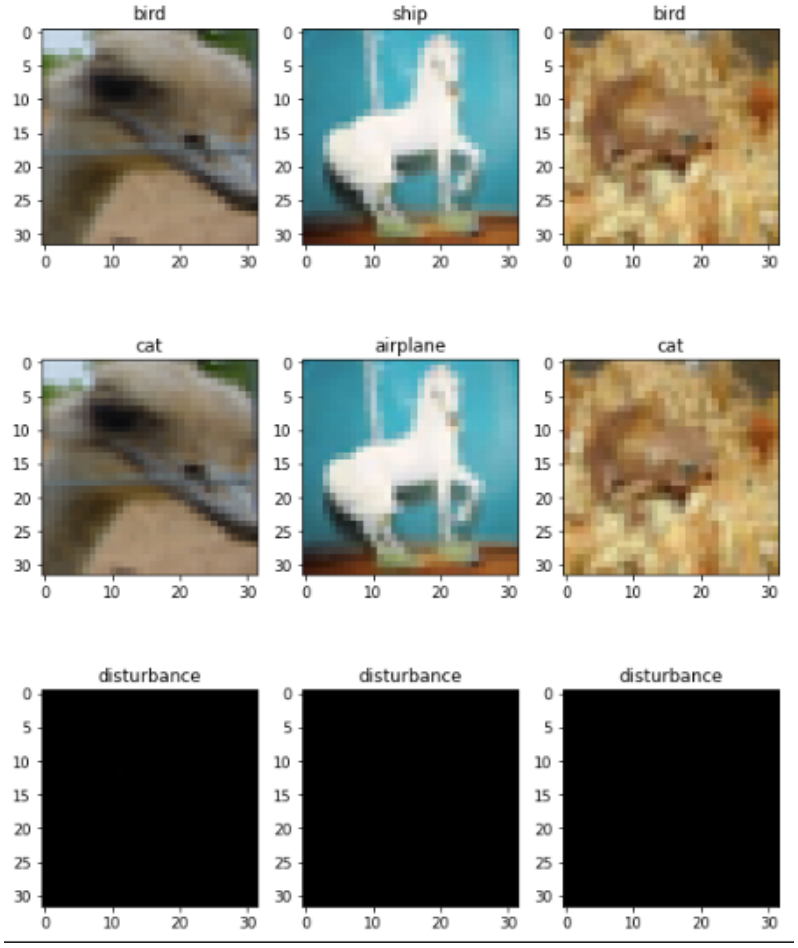

Figure 3.11: C&W_Compare

C&W has a perfect attack effect, and although accompanied by a large number of operations, it can easily make the model misclassify, and the differential perturbations, which are also visually subtle, are challenging to detect with the naked eye.

3.2.6 JSMA (WhiteBox)

JSMA(Papernot et al., 2016) primarily utilises the output category probability information from the model to backpropagate the relevant gradient information, in contrast to FGSM(Goodfellow et al., 2014), which uses the loss function gradient information from the model output. JSMA is referred to by the authors as the forward gradient.

By applying the forward gradient described above, we can determine how much each pixel point influences the model classification result. We can then utilise this knowledge to update the clean sample X. The adversarial sample that results may then be put into the designated category. The writers' macroscopic thought may be summed up thus.

The article presents the idea of a saliency map or the level of impact that various input factors have on the classification outcome.

Formula

Algorithm 1 Crafting adversarial samples

$\mathbf{X}$ is the benign sample, $\mathbf{Y}^*$ is the target network output, $\mathbf{F}$ is the function learned by the network during training, Υ is the maximum distortion, and θ is the change made to features. This algorithm is applied to a specific DNN in Algorithm 2.

Input: $\mathbf{X}$, $\mathbf{Y}^*$, $\mathbf{F}$, Υ, θ

$\mathbf{X}^* \leftarrow \mathbf{X}$
$\Gamma = \{1 \dots |\mathbf{X}|\}$
while $\mathbf{F}(\mathbf{X}^*) \neq \mathbf{Y}^*$ and $||\delta_{\mathbf{X}}|| < \Upsilon$ **do**
    Compute forward derivative $\nabla \mathbf{F}(\mathbf{X}^*)$
    $S = \texttt{saliency\_map}\,(\nabla \mathbf{F}(\mathbf{X}^*), \Gamma, \mathbf{Y}^*)$
    Modify $\mathbf{X}^*_{i_{max}}$ by θ s.t. $i_{max} = \arg\max_i S(\mathbf{X}, \mathbf{Y}^*)[i]$
    $\delta_{\mathbf{X}} \leftarrow \mathbf{X}^* - \mathbf{X}$
end while
return $\mathbf{X}^*$

Figure 3.12: JSMA-Formula1

Algorithm 2 Crafting adversarial samples for LeNet-5
$\mathbf{X}$ is the benign image, $\mathbf{Y}^*$ is the target network output, $\mathbf{F}$ is the function learned by the network during training, Υ is the maximum distortion, and θ is the change made to pixels.

Input: $\mathbf{X}$, $\mathbf{Y}^*$, $\mathbf{F}$, Υ, θ
$\mathbf{X}^* \leftarrow \mathbf{X}$
$\Gamma = \{1 \ldots |\mathbf{X}|\}$ ▷ search domain is all pixels
$\texttt{max\_iter} = \lfloor \frac{784 \cdot \Upsilon}{2 \cdot 100} \rfloor$
$s = \arg\max_j \mathbf{F}(\mathbf{X}^*)_j$ ▷ source class
$t = \arg\max_j \mathbf{Y}^*_j$ ▷ target class
while $s \neq t$ & $\texttt{iter} < \texttt{max\_iter}$ & $\Gamma \neq \emptyset$ **do**
  Compute forward derivative $\nabla \mathbf{F}(\mathbf{X}^*)$
  $p_1, p_2 = \texttt{saliency\_map}(\nabla \mathbf{F}(\mathbf{X}^*), \Gamma, \mathbf{Y}^*)$
  Modify p_1 and p_2 in $\mathbf{X}^*$ by θ
  Remove p_1 from Γ if $p_1 == 0$ or $p_1 == 1$
  Remove p_2 from Γ if $p_2 == 0$ or $p_2 == 1$
  $s = \arg\max_j \mathbf{F}(\mathbf{X}^*)_j$
  $\texttt{iter}++$
end while
return $\mathbf{X}^*$

Figure 3.13: JSMA-Formula2

Algorithm 3 Increasing pixel intensities saliency map
$\nabla \mathbf{F}(\mathbf{X})$ is the forward derivative, Γ the features still in the search space, and t the target class

Input: $\nabla \mathbf{F}(\mathbf{X})$, Γ, t
for each pair $(p, q) \in \Gamma$ **do**
  $\alpha = \sum_{i=p,q} \frac{\partial \mathbf{F}_t(\mathbf{X})}{\partial \mathbf{X}_i}$
  $\beta = \sum_{i=p,q} \sum_{j \neq t} \frac{\partial \mathbf{F}_j(\mathbf{X})}{\partial \mathbf{X}_i}$
  if $\alpha > 0$ and $\beta < 0$ and $-\alpha \times \beta >$ max **then**
    $p_1, p_2 \leftarrow p, q$
    $max \leftarrow -\alpha \times \beta$
  end if
end for
return p_1, p_2

Figure 3.14: JSMA-Formula3

Calculates the bias derivative of each output of the final layer of the model concerning the input, identifying the extent to which each input feature influences each output classification. (The derivatives are calculated similarly to backpropagation)

An adversarial saliency map is constructed to calculate which pixel position changes impact the target classification t. If the corresponding position derivative is positive, increasing the position pixel increases the target t score; if negative, it decreases. If the value of the derivative of the corresponding position is positive, increasing the pixel at that position will increase the

target t t score; if it is negative, it will decrease.

Using the saliency map to select the pixel positions that need to be changed, the positions with the largest values are selected, and then the pixel values are increased or decreased, corresponding to an increase in the output of target t. Iteration is performed until the attack is successful or the maximum damage threshold is reached.

Code

```
def JSMA(input_image,target_label,epochs=100,theta=.1,max_=1,min_=0):
    '''
    :param input_image: input image 32*32 RGB
    :param target_label: 0...9
    :param epochs: Maximum number of iterations
    :param theta: Perturbation factor, in the paper is 1
    :param max_: Maximum pixel boundary
    :param min_: Minimum pixel boundary
    :return: adv_image 32*32*3
    '''

    mask = np.ones_like(input_image)
    for epoch in range(epochs):
        input_image = tf.cast(input_image,tf.float32)

        s = model(tf.reshape(input_image, [1, 32, 32, 3]))[0]

        derivative = calculate_grad(input_image,target_label)

        predictions = np.squeeze(s)
        input_image = input_image.numpy()
        top_k = predictions.argsort()[-3:][::-1]
        predictions_id=top_k[0]

        if predictions_id == np.where(target_label == 1)[0][0]:

        idx,pix_sign = saliency_map(derivative,mask)
        ## apply perturbation
        input_image[idx]+=pix_sign * theta * (max_ - min_)
        ##Points that have reached their limit are no longer involved in the update
        if (input_image[idx]<=min_) or (input_image[idx]>=max_):
        ## print("idx={} over {}".format(idx,input_image[idx]))
            mask[idx]=0
            input_image[idx]=np.clip(input_image[idx], min_, max_)


    return input_image
```

The code in this section is a stand-alone implementation and can be used directly to generate adversarial images. The more specific code is here.

Compare

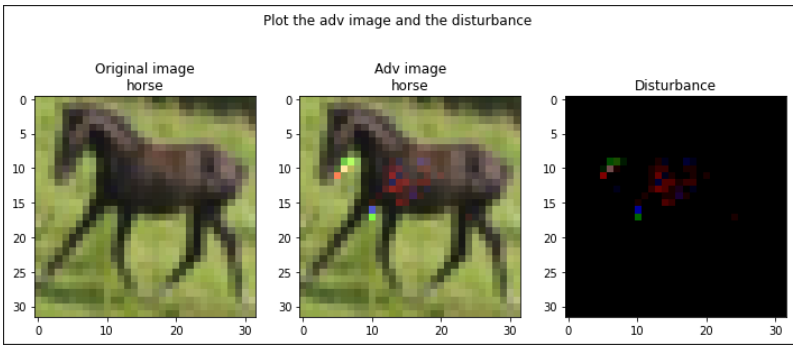

Figure 3.15: JSMA_Compare

This paper assumes that the network is bootstrap able, while the paper uses LeNet as a Baseline, arguing that a successful LeNet attack must be able to attack those deep models.

In contrast to earlier work, this paper starts with a sample and finds a suitable perturbation for the attack. Also, its attack method perturbs a smaller number of input features.

Generating the Jacobian matrix is costly, and the method is slower.

3.2.7 GeoDA (BlackBox)

GeoDA(Rahmati et al., 2020) is a very efficient and low computational black box attack algorithm. GeoDA was originally developed for CNNs with the goal of attacking the decision boundaries of CNNs.

Principle

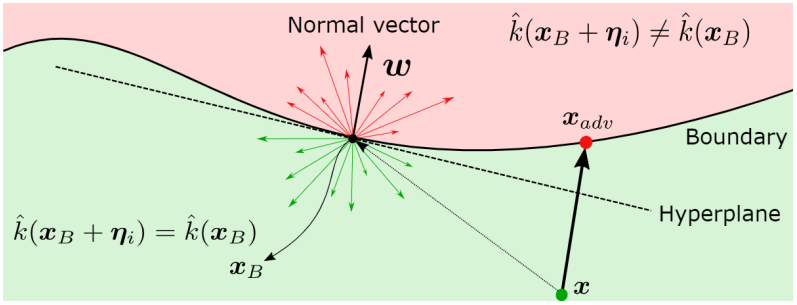

Figure 3.16: GeoDA-Principle

In their study, the authors provide a brand-new geometric framework for decision-based black-box assaults in which the adversary can only get their hands on the target model's top-1 label. It makes sense to look for minor adversarial perturbations near the data sample in the direction of the classifier decision border. We effectively estimate the decision boundary's normal vector using the decision boundary's low mean curvature near the data samples. Using this essential prior, the number of queries needed to trick the black box classifier may be drastically reduced.

Code

```
attack = GeoDA(estimator=classifier, batch_size=1024,
    norm=2, sub_dim=10, max_iter=4000, verbose=False)
```

In this project, the GeoDA algorithm from the ART toolbox is referenced. For testing, the previously previously trained model is first imported and transformed into the classifier in the ART toolbox, and after instantiating GeoDA, the corresponding adversarial samples can be generated.

All the test files are here, with separate attacks tested on several models.

Compare

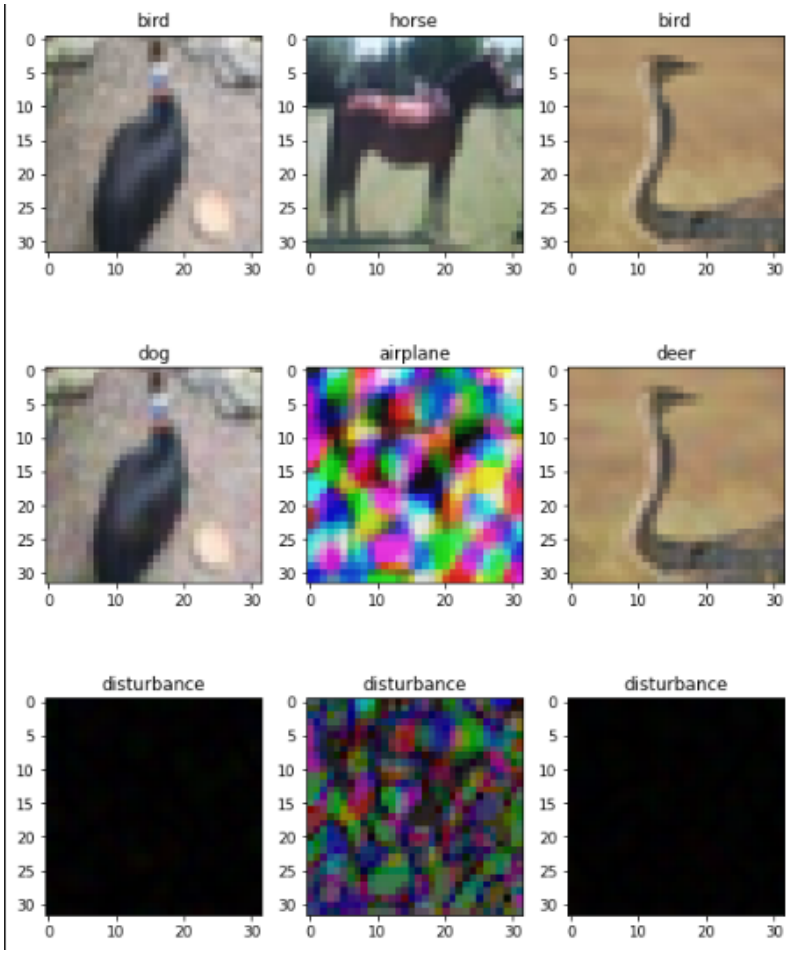

Figure 3.17: GeoDA_Compare

GeoDA is still very effective, but due to its principle, the same amount of arithmetic accompanies it. A random sample was selected in the experiments, and out of 10,000 test samples, 100 were randomly selected for testing. Although the attack algorithm was developed for CNNs, it also works excellent when experimenting on the models constructed in the test project, reducing a model with a minimum 70% fight rate to around 2%.

3.2.8 HopSkipJump (BlackBox)

HopSkipJump(Chen et al., 2020) is the name of an optimization framework's decision-based attack. Author of the paper provide a variety of novel methods for the construction of targeted and untargeted adversarial instances that are optimised for the least distance of "$L2 - distance$" or "$L\infty - distance$". The methods go through three stages throughout each iteration: estimating the gradient direction; step search using geometric steps; and boundary search using dichotomization. The methods themselves need repetition. The theoretical analysis of the optimisation framework and the calculation of the gradient direction are provided in addition to serving as a guide for choosing the hyperparameters. This research also provides inspiration for the critical steps of the suggested strategy. HopSkipJumpAttack is the name of the attack Algorithms.

Core Algorithm

Algorithm 1 Bin-Search

Require: Samples x', x, with a binary function ϕ, such that $\phi(x') = 1, \phi(x) = 0$, threshold θ, constraint ℓ_p.
Ensure: A sample x'' near the boundary.
  Set $\alpha_l = 0$ and $\alpha_u = 1$.
  while $|\alpha_l - \alpha_u| > \theta$ **do**
    Set $\alpha_m \leftarrow \frac{\alpha_l + \alpha_u}{2}$.
    if $\phi(\Pi_{x,\alpha_m}(x')) = 1$ **then**
      Set $\alpha_u \leftarrow \alpha_m$.
    else
      Set $\alpha_l \leftarrow \alpha_m$.
    end if
  end while
  Output $x'' = \Pi_{x,\alpha_u}(x')$.

Figure 3.18: HopSkipJump-Algorithm1

Algorithm 2 HopSkipJumpAttack

Require: Classifier C, a sample x, constraint ℓ_p, initial batch size B_0, iterations T.
Ensure: Perturbed image x_t.
Set θ (Equation (15)).
Initialize at $\tilde{x}_0$ with $\phi_{x^\star}(\tilde{x}_0) = 1$.
Compute $d_0 = \|\tilde{x}_0 - x^\star\|_p$.
for t in $1, 2, \ldots, T-1$ **do**
  (Boundary search)
  $x_t = \text{BIN-SEARCH}(\tilde{x}_{t-1}, x, \theta, \phi_{x^\star}, p)$
  (Gradient-direction estimation)
  Sample $B_t = B_0\sqrt{t}$ unit vectors $u_1, \ldots, u_{B_t}$.
  Set δ_t (Equation (15)).
  Compute $v_t(x_t, \delta_t)$ (Equation (12)).
  (Step size search)
  Initialize step size $\xi_t = \|x_t - x^\star\|_p/\sqrt{t}$.
  while $\phi_{x^\star}(x_t + \varepsilon_t v_t) = 0$ **do**
    $\xi_t \leftarrow \xi_t/2$.
  end while
  Set $\tilde{x}_t = x_t + \xi_t v_t$.
  Compute $d_t = \|\tilde{x}_t - x^\star\|_p$.
end for
Output $x_t = \text{BIN-SEARCH}(\tilde{x}_{t-1}, x, \theta, \phi_{x^\star}, p)$.

Figure 3.19: HopSkipJump-Algorithm2

The approach gives a mechanism to manage mistakes that stray from the border and presents a new, unbiased estimate of the direction of the gradient at the decision boundary based only on access to the model decisions.

A set of algorithms called HopSkipJumpAttack that have no hyperparameters, are query-efficient and include convergence analysis are constructed as part of the technique based on the suggested estimate and analysis.

Code

```
attack = HopSkipJump(classifier=classifier, targeted=False,
    norm=2, max_iter=20, max_eval=100, init_eval=10, verbose=False)

x_test_adv = attack.generate(x=x_test_10_ori)
```

In this project, the HopSkipJump algorithm from the ART toolbox is referenced. For testing, the previously previously trained model is first imported and transformed into the classifier in the ART toolbox, and after instantiating HopSkipJump, the corresponding adversarial samples can be generated.

All the test files are here, with separate attacks tested on several models.

Compare

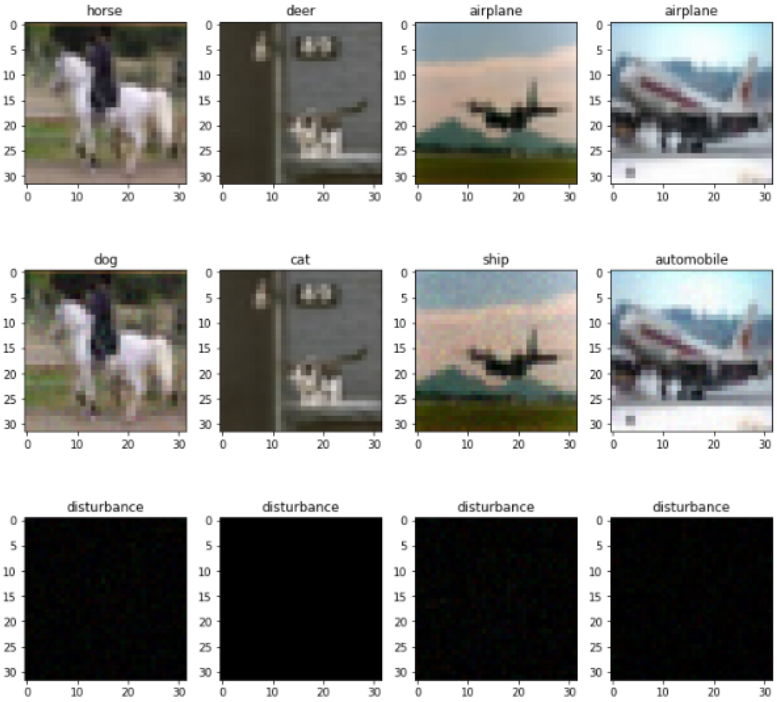

Figure 3.20: HopSkipJump-Compare

In the experiments of this project, HSJ had an excellent attack effect, reducing the accuracy of the tested model to about one-fifth. With single-threaded performance in the R7-5800U, about 2s generated one adversarial image.

3.2.9 PixelAttack (BlackBox)

PixelAttack(Song et al., 2017) was very novel at the time the paper was presented. The attack applies image watermarking techniques to the field of adversarial samples, making the implementation of this type of attack possible. The attack is a black-box attack and is based on a global random search algorithm for population genes. The attack only considers position and transparency for image watermarking and does not consider rotation.

Algorithm

Algorithm 1 PixelDefend

Input: Image $\mathbf{X}$, Defense parameter ϵ_{defend}, Pre-trained PixelCNN model p_{CNN}
Output: Purified Image $\mathbf{X}^*$

$\mathbf{X}^* \leftarrow \mathbf{X}$
for each row i **do**
  for each column j **do**
    for each channel k **do**
      $x \leftarrow \mathbf{X}[i, j, k]$
      Set feasible range $R \leftarrow [\max(x - \epsilon_{\text{defend}}, 0), \min(x + \epsilon_{\text{defend}}, 255)]$
      Compute the 256-way softmax $p_{\text{CNN}}(\mathbf{X}^*)$.
      Update $\mathbf{X}^*[i, j, k] \leftarrow \arg\max_{z \in R} p_{\text{CNN}}[i, j, k, z]$
    end for
  end for
end for

Figure 3.21: PixelAttack-Algorithm

Simply identifying adversarial images is frequently insufficient. In spite of such adversarial alterations, it is frequently crucial to be able to identify images correctly. In this section, we describe PixelDefend, a particular example of a novel family of defence techniques that dramatically improve the state-of-the-art performance of sophisticated attacks while also being effective against all other assaults.

Code

```
attack = PixelAttack(classifier, th=1, es=1,
    max_iter=20, targeted=False, verbose=False)

x_test_adv = attack.generate(x=x_test_10_ori)
```

In this project, the PixelAttack algorithm from the ART toolbox is referenced. For testing, the previously previously trained model is first imported and transformed into the classifier in the ART toolbox, and after instantiating PixelAttack, the corresponding adversarial samples can be generated.

All the test files are here, with separate attacks tested on several models.

Compare

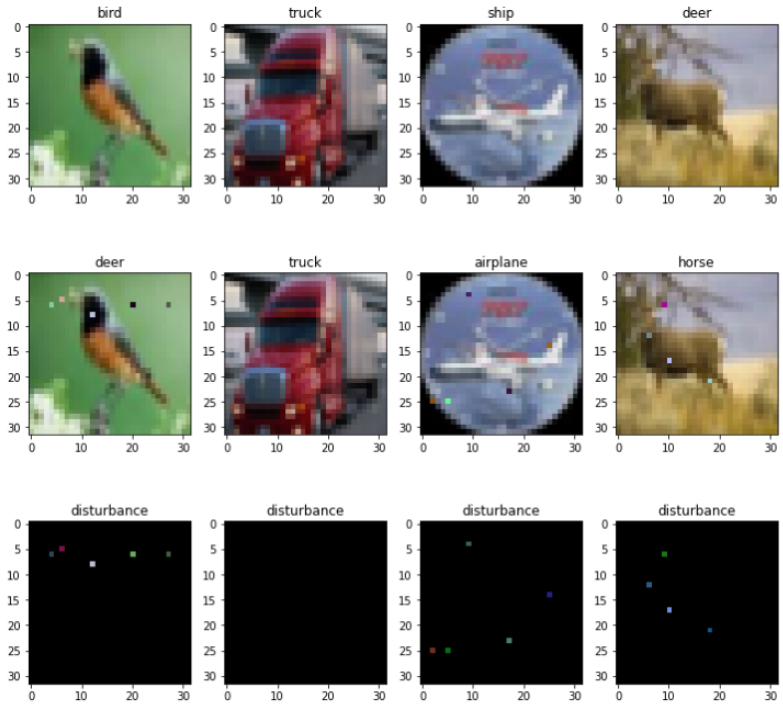

Figure 3.22: PixelAttack-Compare

True to its name, PixelAttack produces adversarial images that only differ from the original images in a few key pixels.

In the test of this section, 100 of the 10,000 test images are randomly selected for testing because the attack requires a lot of calculations. After being attacked, the accuracy of the model under test often drops significantly, dropping to the range of 50% to 20% of the original accuracy.

3.2.10 SquareAttack (BlackBox)

SquareAttack(Andriushchenko et al., 2020) solves some of the problems encountered with other attacks. While gradient-based white-box attacks are often susceptible to gradient obfuscation or masking, black-box attacks are less formal than PGD, but require more computational performance than white-box attacks, and are generally relatively less effective.

Square Attack is a score-based black-box attack that significantly reduces the number of queries under $L2$ and $L\infty$ constraints and is similar in performance to a white-box attack.

Algorithm

Algorithm 1: The Square Attack via random search

Input: classifier f, point $x \in \mathbb{R}^d$, image size w, number of color channels c, l_p-radius ϵ, label $y \in \{1, \dots, K\}$, number of iterations N

Output: approximate minimizer $\hat{x} \in \mathbb{R}^d$ of the problem stated in Eq. (1)

$\hat{x} \leftarrow init(x)$, $l^* \leftarrow L(f(x), y)$, $i \leftarrow 1$

while $i < N$ ***and*** $\hat{x}$ *is not adversarial* **do**

  $h^{(i)} \leftarrow$ side length of the square to modify (according to some schedule)

  $\delta \sim P(\epsilon, h^{(i)}, w, c, \hat{x}, x)$ (see Alg. 2 and 3 for the sampling distributions)

  $\hat{x}_{\text{new}} \leftarrow$ Project $\hat{x} + \delta$ onto $\{z \in \mathbb{R}^d : \|z - x\|_p \leq \epsilon\} \cap [0, 1]^d$

  $l_{\text{new}} \leftarrow L(f(\hat{x}_{\text{new}}), y)$

  if $l_{new} < l^*$ **then** $\hat{x} \leftarrow \hat{x}_{\text{new}}$, $l^* \leftarrow l_{\text{new}}$;

  $i \leftarrow i + 1$

end

Figure 3.23: SquareAttack-Algorithm

The fundamental principle of Square Attack's random search is to randomly choose one δ every iteration and then assess if doing so improves the objective function before deciding whether to update it. For various assaults, different sample distributions are recommended. The algorithm's basic concept is to choose one square point at a time and update it.

Algorithm 2: Sampling distribution P for l_∞-norm

Input: maximal norm ϵ, window size h, image size w, color channels c

Output: New update δ

$\delta \leftarrow$ array of zeros of size $w \times w \times c$

sample uniformly $r, s \in \{0, \dots, w-h\} \subset \mathbb{N}$

for $i = 1, \dots, c$ **do**

  $\rho \leftarrow Uniform(\{-2\epsilon, 2\epsilon\})$

  $\delta_{r+1:r+h,\ s+1:s+h,\ i} \leftarrow \rho \cdot \mathbb{1}_{h \times h}$

end

Figure 3.24: SquareAttack-Algorithm-$L\infty$

The initialisation is the random noise of $[-\epsilon, \epsilon]$.
The sampling method is the random noise of $[-2\epsilon, 2\epsilon]$.

Algorithm 3: Sampling distribution P for l_2-norm

Input: maximal norm ϵ, window size h, image size w, number of color channels c, current image $\hat{x}$, original image x

Output: New update δ

$\nu \leftarrow \hat{x} - x$

sample uniformly $r_1, s_1, r_2, s_2 \in \{0, \dots, w-h\}$

$W_1 := r_1 + 1 : r_1 + h, s_1 + 1 : s_1 + h$, $W_2 := r_2 + 1 : r_2 + h, s_2 + 1 : s_2 + h$

$\epsilon^2_{\text{unused}} \leftarrow \epsilon^2 - \|\nu\|_2^2$, $\eta^* \leftarrow \eta / \|\eta\|_2$ with η as in (2)

for $i = 1, \dots, c$ **do**

  $\rho \leftarrow Uniform(\{-1, 1\})$

  $\nu_{\text{temp}} \leftarrow \rho\eta^* + \nu_{W_1,i} / \|\nu_{W_1,i}\|_2$

  $\epsilon^i_{\text{avail}} \leftarrow \sqrt{\|\nu_{W_1 \cup W_2,i}\|_2^2 + \epsilon^2_{\text{unused}}/c}$

  $\nu_{W_2,i} \leftarrow 0$, $\nu_{W_1,i} \leftarrow (\nu_{\text{temp}} / \|\nu_{\text{temp}}\|_2)\epsilon^i_{\text{avail}}$

end

$\delta \leftarrow x + \nu - \hat{x}$

Figure 3.25: SquareAttack-Algorithm-$L2y$

The initialisation is multiple random squares sampled in the following manner.

Because it is noted that the antagonistic perturbation is more localised under the $L2$ constraint than $L\infty$, to mimic this feature, the η is updated with two centres, absolute and opposite, with the other parts having smaller absolute values but not zero.

Since the $L2$ attack does not have a similar $L\infty$ constraint, the second strategy is to use a 'move the mass' strategy, choosing two random squares, denoted vW_1 and vW_2, and using vW_2 to increase the perturbation of vW_1.

Code

```
attack = SquareAttack(estimator=classifier, norm=np.inf,
    max_iter=1000, eps=0.05, p_init=0.8, nb_restarts=1, verbose=False)

x_test_adv = attack.generate(x=x_test_10_ori)
```

In this project, the SquareAttack algorithm from the ART toolbox is referenced. For testing, the previously previously trained model is first imported and transformed into the classifier in the ART toolbox, and after instantiating SquareAttack, the corresponding adversarial samples can be generated.

All the test files are here, with separate attacks tested on several models.

Compare

Figure 3.26: SquareAttack-Compare

In the original paper, the authors prove the convergence of the method and why they chose a square, which was chosen because it is the same shape as the convolution kernel and maximizes the output value of the convolution kernel.

The l1 attack is somewhat simpler than l2, but both are relatively simple and efficient. Apart from the tuning, We wonder if there is also a reason why square reduces the search space and makes the attack more efficient.

SquareAttack reduces the model's accuracy to a minimum of 3%-5% in this chapter, which is far better than other attacks.

3.3 Model Defences

In this section, we introduce defence methods for models. Defence is usually developed after a model has been attacked by a specific method, and in this way, it can be seen that defence methods are targeted and not generalised, i.e. they are only defended against a specific attack method.

In this section, we test several very typical defence methods, each of which belongs to a different category, including preprocessors, postprocessors, and trainers.

3.3.1 Total Variance Minimization (Preprocessor)

Total Variance Minimization(Guo et al., 2017) was developed in the context of developing methods to effectively defend against non-targeted adversarial attacks. In the Guo et al. (2017)' s paper, they used FSGM and C&W as test attacks. In this project we have used not only FSGM(Goodfellow et al., 2014) but also I-FSGM(Kurakin et al., 2018) and DeepFool(Moosavi-Dezfooli et al., 2016) as attack tests.

Algorithm

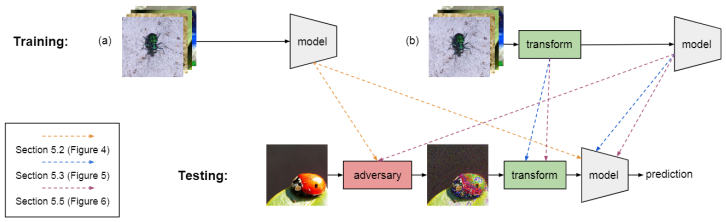

Figure 3.27: TotalVarianceMinimization-Algorithm

The overall use of Total Variance Minimisation is shown in the 3.27 diagram.

A description of the methods used by the author is in (Guo et al., 2017, chapter 4.2).

In this project, two different tests were carried out.

One is the original image is processed by Total Variance Minimisation and then put into the model for training. The resulting adversarial images are again subjected to Total Variance Minimisation when testing the defence effect and then put into the model for prediction.

The other is to train the model without any additional processing, only Total Variance Minimization is applied to the generated adversarial images and then the model is used to make predictions.

Code

```
preprocess  = TotalVarMin(clip_values=[0, 1])
preprocess.fit(x_test_adv)
x_test_def = preprocess(x_test_adv)[0]
predictions = classifier.predict(x_test_def)
accuracy = np.sum(np.argmax(predictions, axis=1)
    == np.argmax(y_test, axis=1)) / len(y_test)
```

In this project, the TVM algorithm from the ART toolbox is referenced. For testing, the previously previously trained model is first imported and transformed into the classifier in

the ART toolbox, and after instantiating TVM, the corresponding adversarial samples can be generated.

The pre-processed test results from the model training are here. The results of Total Variance Minimization on the adversarial images directly without pre-processing are here.

Compare

In this project, the test set of cifar10 was used to complete the validation of the defence effect.

Total Variance Minimisation was tested to be effective in defending against several of the attacks we used. The model that was trained after pre-processing had a very good defence effect, while the model that was not pre-processed had a less good defence effect.

3.3.2 Reverse Sigmoid (Postprocessor)

Reverse Sigmoid(Guo et al., 2017) is a post-processor type of defence method. It is a defence method developed to prevent a model built to be stolen by an attacker. Specifically, it is a defence method developed to protect the interests of companies such as cloud computing and cloud servers. By using Reverse Sigmoid, models deployed in the cloud can be effectively prevented from being stolen and probed by attackers for various details of the model parameters.(Tramèr et al., 2016)

In this section we also use this method to test whether it is effective against adversarial images.

Algorithm

To stop models from being stolen from cloud service APIs, Guo et al. (2017) suggest an extension layer that may be deployed to the majority of neural network classifiers. The layer introduces a modest, controlled disturbance to maximise the loss of stolen models while retaining accuracy. In other words, rather than attempting to identify assaults, Guo et al. (2017) utilise noise that hardly affects everyday users while nevertheless reducing and mitigating model theft attempts. The best defence should benefit the service user while giving the attacker significant advantages above and beyond the final label.

A specific explanation is in (Guo et al., 2017, chapter 3 Method).

Code

```
postprocessor = ReverseSigmoid(beta=1.0, gamma=0.1)
post_preds = postprocessor(preds=predictions)

accuracy = np.sum(np.argmax(post_preds, axis=1)
    == np.argmax(y_test_infunc, axis=1)) / len(y_test_infunc)
```

In this project, the ReverseSigmoid algorithm from the ART toolbox is referenced. For testing, the previously previously trained model is first imported and transformed into the classifier in the ART toolbox, and after instantiating ReverseSigmoid, the corresponding adversarial samples can be generated.

Compare

In this project, the test set of cifar10 was used to complete the validation of the defence effect.

Probably because ReverseSigmoid is not a defence method developed for confrontation, it did not receive good results in practical tests.

3.3.3 Madry's Protocol (Trainer)

Through the perspective of robust optimisation, the Madry et al. (2017) investigate the adversarial resilience of neural networks. This method offers a thorough and comprehensive overview of most earlier research on the issue. Its principled character makes it possible to identify dependable and, in a sense, universal techniques for training and attacking neural networks. The approaches specifically provide a security guarantee against any hostile assault. We can train the network using these techniques to be much more resistant to a variety of adversarial assaults. Additionally, they introduce the idea of security from first-order adversaries as a common and organic security guarantee.

Code

```
# trainer
adv_trainer = AdversarialTrainerMadryPGD(classifier=classifier, batch_size=4096)
adv_trainer.fit(x_train, y_train)

predictions_new = np.argmax(adv_trainer.trainer.get_classifier().predict(x_test), axi
```

In this project, the ReverseSigmoid algorithm from the ART toolbox is referenced. For testing, the previously previously trained model is first imported and transformed into the classifier in the ART toolbox, and after instantiating ReverseSigmoid, the corresponding adversarial samples can be generated.

Compare

In this project, the test set of cifar10 was used to complete the validation of the defence effect.

Probably because ReverseSigmoid is not a defence method developed for confrontation, it did not receive good results in practical tests.

3.3.4 SubsetScanningDetector (Detector)

This method can be considered not only as a detection method, but also as a defense method. "Fast Generalized Subset Scan for Anomalous Pattern Detection". (McFowland et al., 2013)

Algorithm

2.7 FGSS Algorithm

Inputs: test data set, training data set, $\alpha_{\max}$, r, Y.

1. Learn a Bayesian network (structure and parameters) from the training data set.

2. For each data record R_i and each attribute A_j, in both training and test data sets, compute the likelihood l_{ij} given the Bayesian network.
3. Compute the p-value range $p_{ij} = [p_{\min}(p_{ij}), p_{\max}(p_{ij})]$ corresponding to each likelihood l_{ij} in the test data set.
4. For each (non-duplicate) data record R_i in the test data set, define the local neighborhood S_i to consist of R_i and all other data records R_j where $d(R_i, R_j) \leq r$.
5. For each local neighborhood S_i, iterate the following steps Y times. Record the maximum value F^* of $F(S)$, and the corresponding subsets of records R^* and attributes A^* over all such iterations:
   (a) Initialize $A \leftarrow$ random subset of attributes.
   (b) Repeat until convergence:
      i. Maximize $F(S) = \max_{\alpha \leq \alpha_{\max}} F_\alpha(R \times A)$ over subsets of records $R \subseteq S_i$ in the local neighborhood, for the current subset of attributes A, and set $R \leftarrow \arg\max_{R \subseteq S_i} F(R \times A)$.
      ii. Maximize $F(S) = \max_{\alpha \leq \alpha_{\max}} F_\alpha(R \times A)$ over all subsets of attributes A, for the current subset of records R, and set $A \leftarrow \arg\max_{A \subseteq \{A_1 \ldots A_M\}} F(R \times A)$.
6. Output $S^* = R^* \times A^*$.
7. Optionally, perform randomization testing, and report the p-value of S^*.

Figure 3.28: TotalVarianceMinimization-Algorithm

Code

```
# Generate adversarial samples
x_test_adv = attacker.generate(x_test)

# Instantiate the detector
detector = SubsetScanningDetector(classifier, x_train, layer=1)

clean_scores, adv_scores, dpwr = detector.scan(clean, anom)
dpwr > 0.5
```

In this project we used the "Detector based on Fast Generalized Subset Scan" algorithm from the ART toolbox(Nicolae et al., 2018) and instantiated it for testing.

Compare

In this project the test set of cifar10 was used to test the effectiveness of adversarial image detection.

We tried to generate adversarial images using a variety of attacks and detected them using the "SubsetScanningDetector" method.

Chapter 4

Results

4.1 Model Building

In this section, we build several different neural networks to test the attacks and defences for the project, laying the groundwork.

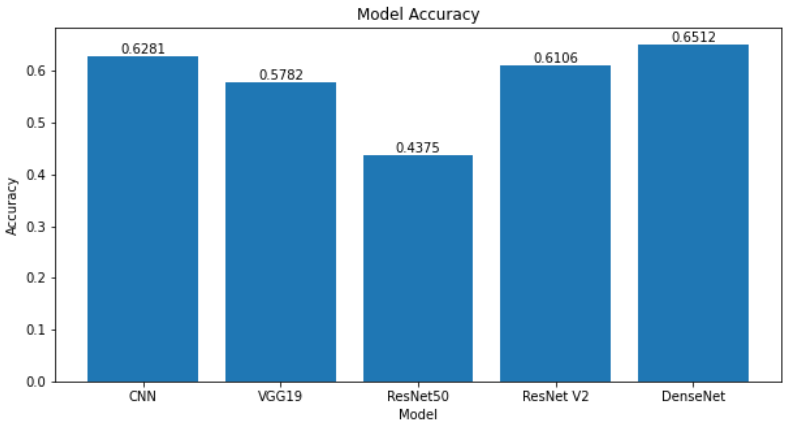

Figure 4.1: Accuracy of each models

We can visually see in the image4.1 the accuracy of each model, under the cifar-10 test set.

As none of the models are specifically optimised for robustness, it is not possible to identify whether the models are robust enough in this section, but we can guarantee that, under normal circumstances, the models have an accuracy rate of about 60% for the cifar-10 dataset, which is still relatively high.

4.2 Model Attack

In this section, both black-box and white-box attacks are notoriously effective, essentially reducing the accuracy of the model to less than 20% accuracy, and at best reducing it to close to 0% accuracy, making the model no longer able to function and work properly, and the model's robustness manifesting itself as an inability to resist the attack. However, relatively speaking, these attack algorithms still require a large number of operations to complete the attack.

Here we use the classic FSGM as an example in figure (4.2). to see the effect of FSGM attacks on individual models:

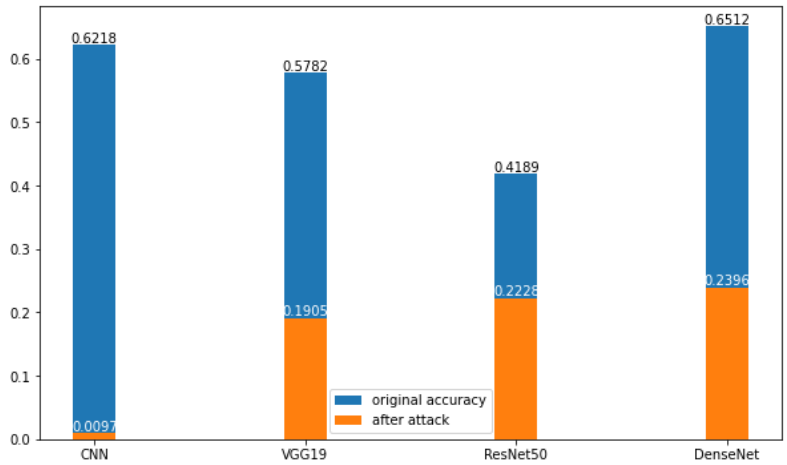

Figure 4.2: Accuracy of each models

4.3 Model Defence

In terms of defence, the number of defence algorithms is much smaller than the number of attack algorithms, as defence requires targeted development. With specific targeting of defence methods, specific attacks can be effectively defended against. However, when testing untargeted development of attack algorithms, it was found that the defence algorithms generally failed.

In the following, we use TotalVarianceMinimization as an example. TotalVarianceMinimization was developed for FSGM(Goodfellow et al., 2014), but in this example, we use both I-FSGM(Kurakin et al., 2018), an upgraded version of FSGM, and DeepFool(Moosavi-Dezfooli et al., 2016), a black box attack, to demonstrate it.

Table 4.1: Example of using TotalVarianceMinimization to defence

	FSGM		I-FSGM		DeepFool	
Model accuracy (%) (AF/DAF) (After Attack / Defence After Attack)	AF	DAF	AF	DAF	AF	DAF
CNN	38.26	56.26	11.44	14.54	17.81	44.69
VGG19	38.4	41.9	6.18	9.41	23.56	41.74
ResNet	34.02	35.67	6.36	15.36	21.08	35.38
ResNet V2	32.4	48.42	5.45	18.68	19.47	48.3
DenseNet121	32.67	48.43	4.77	17.79	23.84	48.64

Chapter 5

Discussion and Analysis

5.1 Significance of the findings

In the attack session, most of the attack algorithms considerably reduce the model's accuracy, and the model's robustness is very weak. With the introduction of partial defence algorithms, defence methods can achieve effective mitigation against specific attacks, but not perfect defence. For other attack algorithms that are not explicitly developed, the defence algorithms cannot effectively defend against these attacks.

5.2 Limitations

The experiments conducted in this thesis require a large number of hardware resources for computing, especially in the attack part, where white-box attacks require less computing compared to black-box attacks but, in general, require a large amount of computing power.

The limitation of this thesis is that it is not possible to find a defence algorithm that matches the number of attack algorithms, and it is not possible to find all defence algorithms that match the attack algorithms.

However, this thesis shows, by way of example, the robustness of different neural network models for different types of attack and defence algorithms on the cifar-10 dataset.

The only regret is that, due to the sheer volume of computation, it is only possible to present a comparison of the different parameters of the algorithms as far as possible without being able to present a super-detailed comparison of the subtle parameter variations that lead to different results.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This paper studies the robustness of neural networks, that is, the accuracy of neural networks after being attacked and the implementation of defence respectively.

Firstly, the basic convolutional neural network was constructed by TensorFlow, and several classical neural networks were introduced by using "transfer learning". The "CIFAR-10" dataset was used to train the neural network and saved as the "H5" format file for later experiments.

In the attack stage of the experiment, by instantiating a variety of different types of attacks and fine-tuning the subtle parameters of different attacks, the attack effect under different parameters is compared, that is, the accuracy of the test model after being attacked, so as to observe the robustness of the model.

In the defence stage of the experiment, several defence methods developed against the attack algorithms used in the paper are selected, and the test of the defence algorithm against other non-targeted attack algorithms is conducted on the basis of the basic defence to verify its effect. The robustness of the model is verified by testing the model's accuracy after the test.

In the experiment, we found that the model is extremely fragile without the development of robustness and almost loses its original accuracy after being attacked. After using the defence algorithm, it can only effectively alleviate the attack effect but not completely defend the attack.

6.2 Future work

This project has completed testing a large number of attack algorithms and some defence algorithms. In the future, a neural network model with strong robustness can be considered to test the robustness problem. Yun et al. (2019) have developed a deep learning model with high inverse detection ability for text, and similar models for images can be developed in the future.

At the same time, it is necessary to prepare equipment with large computing power to compare data under different parameters. Due to the limited equipment resources in this project, it is a pity that a high-fine-grained parameter comparison cannot be carried out in the later defence test.

Chapter 7

Reflection

The project is written in Python, with Jupiter Notebook(Kluyver et al., 2016) as the primary debugging tool, VS Code as the programming tool, TensorFlow(Abadi et al., 2015) as the backend for the neural network build and the use of a third-party tool, ART Toolbox(Nicolae et al., 2018), to complete the project. Latex was used to write the reports.

The project faced different problems at each stage.

At the beginning of the project, choosing a suitable deep learning framework was a top priority. Among the major platforms, we chose TensorFlow because it has a perfect adaptation to Google Colab's TPU after upgrading to 2.0 and incorporates Keras(Chollet et al., 2015), a front-end for building deep learning models extremely quickly, features that were very appealing to me because of my previous experience with Keras.

During the model building phase, I was faced with the problem of model selection. To make the experiments reproducible and generalisable, I chose to build the basic convolutional neural network and several special cases of neural network models. A large number of operations required even for a fast TPU took a lot of time, so I introduced "migration learning" and used the parameters of "ImageNet" to speed up the construction.

When experimenting with the attack part, I initially refactored several classical attack algorithms on my own, which worked well but were not fast enough to develop. After careful study, I introduced the ART toolbox(Nicolae et al., 2018), which implemented the algorithms well and simply instantiated them for direct use, which was twice the effort for the project. However, during the experiments, a lot of hyperparameter tuning and comparison of experimental results required a lot of computing, which Google Colab could not accommodate, so I moved the main computing to the RACC cluster at the University of Reading, which greatly eased the computing problem for the project.

When the experiment reached the defence part of the stage, as the defence required first testing the attack and then the defence, the computational effort for both calculations was huge, and the computational cost required increased exponentially. I took the step of reducing the granularity of hyperparameter tuning and moving to use only highly characterised parameters for the experiments. Future phases of the experiments can be filled in with time for this part.

References

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.
URL: *https://www.tensorflow.org/*

Andriushchenko, M., Croce, F., Flammarion, N. and Hein, M. (2020), Square attack: a query-efficient black-box adversarial attack via random search, *in* 'European Conference on Computer Vision', Springer, pp. 484–501.

Balaji, K. and Lavanya, K. (2019), Medical image analysis with deep neural networks, *in* 'Deep learning and parallel computing environment for bioengineering systems', Elsevier, pp. 75–97.

Camastra, F. and Vinciarelli, A. (2015), *Machine learning for audio, image and video analysis: theory and applications*, Springer.

Carlini, N. and Wagner, D. (2017), Towards evaluating the robustness of neural networks, *in* '2017 ieee symposium on security and privacy (sp)', Ieee, pp. 39–57.

Chen, J., Jordan, M. I. and Wainwright, M. J. (2020), Hopskipjumpattack: A query-efficient decision-based attack, *in* '2020 ieee symposium on security and privacy (sp)', IEEE, pp. 1277–1294.

Chollet, F. et al. (2015), 'Keras'.
URL: *https://github.com/fchollet/keras*

Goodfellow, I. J., Shlens, J. and Szegedy, C. (2014), 'Explaining and harnessing adversarial examples', *arXiv preprint arXiv:1412.6572* .

Guo, C., Rana, M., Cisse, M. and Van Der Maaten, L. (2017), 'Countering adversarial images using input transformations', *arXiv preprint arXiv:1711.00117* .

He, K., Zhang, X., Ren, S. and Sun, J. (2016*a*), Deep residual learning for image recognition, *in* 'Proceedings of the IEEE conference on computer vision and pattern recognition', pp. 770–778.

He, K., Zhang, X., Ren, S. and Sun, J. (2016*b*), Identity mappings in deep residual networks, *in* 'European conference on computer vision', Springer, pp. 630–645.

Huang, G., Liu, Z., Van Der Maaten, L. and Weinberger, K. Q. (2017), Densely connected convolutional networks, *in* 'Proceedings of the IEEE conference on computer vision and pattern recognition', pp. 4700–4708.

Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J., Grout, J., Corlay, S., Ivanov, P., Avila, D., Abdalla, S. and Willing, C. (2016), Jupyter notebooks – a publishing format for reproducible computational workflows, *in* F. Loizides and B. Schmidt, eds, 'Positioning and Power in Academic Publishing: Players, Agents and Agendas', IOS Press, pp. 87 – 90.

Krizhevsky, A., Hinton, G. et al. (2009), 'Learning multiple layers of features from tiny images'.

Krizhevsky, A., Sutskever, I. and Hinton, G. E. (2012), 'Imagenet classification with deep convolutional neural networks', *Advances in neural information processing systems* **25**.

Kurakin, A., Goodfellow, I. J. and Bengio, S. (2018), Adversarial examples in the physical world, *in* 'Artificial intelligence safety and security', Chapman and Hall/CRC, pp. 99–112.

Lee, T., Edwards, B., Molloy, I. and Su, D. (2019), Defending against neural network model stealing attacks using deceptive perturbations, *in* '2019 IEEE Security and Privacy Workshops (SPW)', IEEE, pp. 43–49.

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* .

Mahdavinejad, M. S., Rezvan, M., Barekatain, M., Adibi, P., Barnaghi, P. and Sheth, A. P. (2018), 'Machine learning for internet of things data analysis: A survey', *Digital Communications and Networks* **4**(3), 161–175.

McFowland, E., Speakman, S. and Neill, D. B. (2013), 'Fast generalized subset scan for anomalous pattern detection', *The Journal of Machine Learning Research* **14**(1), 1533–1561.

Moosavi-Dezfooli, S.-M., Fawzi, A. and Frossard, P. (2016), Deepfool: a simple and accurate method to fool deep neural networks, *in* 'Proceedings of the IEEE conference on computer vision and pattern recognition', pp. 2574–2582.

Nicolae, M.-I., Sinn, M., Tran, M. N., Buesser, B., Rawat, A., Wistuba, M., Zantedeschi, V., Baracaldo, N., Chen, B., Ludwig, H., Molloy, I. and Edwards, B. (2018), 'Adversarial robustness toolbox v1.2.0', *CoRR* **1807.01069**.
URL: *https://arxiv.org/pdf/1807.01069*

Papernot, N., McDaniel, P., Goodfellow, I., Jha, S., Celik, Z. B. and Swami, A. (2017), Practical black-box attacks against machine learning, *in* 'Proceedings of the 2017 ACM on Asia conference on computer and communications security', pp. 506–519.

Papernot, N., McDaniel, P., Jha, S., Fredrikson, M., Celik, Z. B. and Swami, A. (2016), The limitations of deep learning in adversarial settings, *in* '2016 IEEE European symposium on security and privacy (EuroS&P)', IEEE, pp. 372–387.

Rahmati, A., Moosavi-Dezfooli, S.-M., Frossard, P. and Dai, H. (2020), Geoda: a geometric framework for black-box adversarial attacks, *in* 'Proceedings of the IEEE/CVF conference on computer vision and pattern recognition', pp. 8446–8455.

Silva, S. H. and Najafirad, P. (2020), 'Opportunities and challenges in deep learning adversarial robustness: A survey', *arXiv preprint arXiv:2007.00753* .

Simonyan, K. and Zisserman, A. (2014), 'Very deep convolutional networks for large-scale image recognition', *arXiv preprint arXiv:1409.1556* .

Song, Y., Kim, T., Nowozin, S., Ermon, S. and Kushman, N. (2017), 'Pixeldefend: Leveraging generative models to understand and defend against adversarial examples', *arXiv preprint arXiv:1710.10766* .

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I. and Fergus, R. (2013), 'Intriguing properties of neural networks', *arXiv preprint arXiv:1312.6199* .

Tramèr, F., Zhang, F., Juels, A., Reiter, M. K. and Ristenpart, T. (2016), Stealing machine learning models via prediction {APIs}, *in* '25th USENIX security symposium (USENIX Security 16)', pp. 601–618.

Xu, J., Chen, J., You, S., Xiao, Z., Yang, Y. and Lu, J. (2021), 'Robustness of deep learning models on graphs: A survey', *AI Open* **2**, 69–78.

Yuan, X., He, P., Zhu, Q. and Li, X. (2019), 'Adversarial examples: Attacks and defenses for deep learning', *IEEE transactions on neural networks and learning systems* **30**(9), 2805–2824.

Yun, X., Huang, J., Wang, Y., Zang, T., Zhou, Y. and Zhang, Y. (2019), 'Khaos: An adversarial neural network dga with high anti-detection ability', *IEEE transactions on information forensics and security* **15**, 2225–2240.