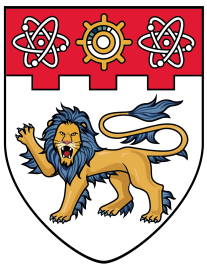
Adversarial attack defences for neural network

Singh Kirath

2022



NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Adversarial attack defences for Neural Network

Kirath Singh

U8122329J

Supervisor: Associate Professor Anupam Chattopadhyay

School of Computer Science and Engineering

Submitted in partial fulfilment of the degree of Bachelor of Engineering in Computer Science

2021/2022 Semester 2

Abstract

Since the advent of deep learning, we have been wielding them to solve intricate problems in the field of natural language processing, image processing, etc. Furthermore, we have been deploying complex deep learning models in real-time systems like autonomous vehicles, security cameras, etc purely based on their precision only to realize that these high precision models can be vulnerable to a variety of adversaries in the environment, that can hamper the overall robustness of our deep learning models.

The contemporary defense strategies in the market either cannot alleviate a variety of adversarial attacks primarily in a white box environment or do not have a standardized approach that can be applied to any form of the complex deep-learning models to make them inert from a variety of adversaries. Moreover, there is a need for standardized adversarial defense strategies for mitigating a variety of adversarial attacks to make our models more robust in a white box environment.

In this project, we make use of three different state-of-the-art deep-learning architectures trained on 2 benchmarking datasets – CIFAR-10 and CIFAR-100, to analyze the difference in the performance of these models in the absence of an adversary as well as in the presence of an adversary in a white-box environment. We primarily use two white box attack methodologies – Fast Gradient Sign Method (FGSM) and Projected Gradient Descent (PGD) to plant adversarial samples using epsilon values ranging from 0.1 to 0.8. Furthermore, we go one step further to devise a defense strategy – Defensive Distillation, that can be applied to a deep-learning architecture to deplete the overall efficacy of FGSM and PGD attacks.

Keywords: Deep Learning , natural language processing , image processing , Adversarial Attacks, Fast Gradient Sign Method (FGSM) , Projected Gradient Descent (PGD) , Defensive Distillation

Acknowledgement

I would like to express my sincere gratitude towards my project supervisor, Associate Professor Anupam Chattopadhyay for allowing me to work on a project that is related to adversarial attacks. The guidance and resources he provided me were incredibly substantial, in boosting my understanding in the field of Machine Learning within a short period of time.

I would also like to thank Ph.D. Candidate, Nandish Chattopadhyay for his constant guidance and input on the proposed framework. Also, I would like to thank him for his instantaneous responses and was readily available for any meetups to explain any uncertainty and resolving issues.

Lastly, I would like to thank my parents for their constant motivation, throughout the journey of this project. This project would not be possible without the support from the all the people mentioned above.

Table of Contents

Nomenclature

AI	Artificial Intelligence
CNN	Convolutional Neural Network
DNN	Deep Neural Network
FGSM	Fast Gradient Sign Method
PGD	Projected Gradient Descent
VGG-16	Visual Geometry Group-16
ResNet-50	Residual Neural Network-50

Lists of Figures

Listings

List of Tables

1 Introduction

Deep learning is a field of machine learning that attempts to imitate the functioning of the human brain with of the help of a network of layers. The neural-network layers have the ability to learn from a massive amount of data in order to make precise predictions on unseen data. Nowadays, deep learning is the driving force behind many of the streamlined application and services in the form of digital voice assistants, video captioning, etc.

Deep-learning has also gained popularity in the market owing to the ease at which machine learning engineers are able to push neural network models to production with the help of frameworks like tensorflow and pytorch. But, besides that, the main factor that has revolutionised the development of neural-network models is the vast availability of data. This pump in the availability of data has enabled humans to solve complex problems with deep-learning based architectures to such an extent that we now aim to automate most of the human-dependent task with the help of machine learning.

Deep-learning models are usually trained on a major portion of the available dataset while a minor portion of it is used to test the model during training. The true test, however, comes in the phase when the model is exposed to a real-world setting. Here the model primarily deals with data which it has not encountered before which makes it more liable to produce misclassifications or make predictions with a low confidence.

The main reason behind the failure of deep-learning models is due to the extreme non-linearity that the neural networks possess as well as the inability of the models to deal with data that has been, to some degree, perturbated knowingly by an adversary or perhaps unknowingly during the data collection phase.

Owing to the complexity of problems that the deep-learning models have to solve, it becomes a herculean task to make the model generalize to a wide variety of data. Additionally, it is an arduous task to mimic any outlandish data that the adversary or the environment might throw at the model when it is exposed.

These challenges make us question our over reliability on deep learning-based models to solve a plethora of complex problems. Furthermore, to date, we still do not possess a metric that can help us in verifying the reliability of our neural-network models in a real-world setting. The solution that we can devise should be along the lines of making our models robust enough to handle any form of adversaries that might be present in the real-world setting as well as any form of abnormalities that might hamper the performance of the model

1.1 Motivation for this project

With the growth in the usage of deep learning models to solve problems in real-life, the risk of malfunctioning of neural-network model is day-by-day increasing due to the presence of a plethora of adversaries. Furthermore, these adversaries are exploiting various aspects of a neural-network model, thereby compelling the model to under-perform. Hence, there is a need to scrutinize the impact of various adversaries on the performance of neural-network models. Furthermore, state-of-the-art defence strategies can be proposed in order to enable the neural-network models to overcome the impact of numerous adversarial attacks, thereby making them resilient from any sort of adversaries.

1.2 Objectives

The aim of this project is to examine how adversarial attacks are performed on neural-network models as well as the impact of those attacks on the performance of the models. Furthermore, with the available information regarding the adversarial attacks,we devise defence strategies to counteract the influence of those adversarial attacks on the model in order to make them more robust.

1.3 Scope

This project mainly focuses on identifying the difference in the number of misclassifications during the absence of any adversary as well as when an adversary plants adversarial samples, in a white-box environment. Furthermore, the defence strategies discussed in the project will be used to bring the rate of misclassification down in the presence of an adversary in a white box environment.

2 Literature Review

This section mainly focuses on understanding the various aspects of neural networks in the area of machine learning, various adversarial attacking methodologies on neural networks as well as past research on some of the defence mechanisms to mitigate any form of adversarial attacks.

2.1 Machine Learning

Machine learning is a field of Artificial Intelligence that is mainly used to formulate automated analytical models in order to identify patterns and learn from a given dataset and make precise predictions using the learnings gathered during the training phase. Machine learning models are trained using mainly three difference approaches namely – supervised learning, unsupervised learning and reinforcement learning respectively.

- **Supervised Learning** – Supervised machine learning is a subcategory of machine learning that utilises labelled datasets in order to train the model for making predictions. It mainly derives an inferred mathematical function from the training data in order make precise predictions [1]. Some of the popular supervised machine learning algorithms are decision trees [2], Support Vector Machines [3] and Naive Bayes [4].

- **Unsupervised Learning** – Unsupervised machine learning is a subcategory of machine learning that utilises unlabelled data to uncover the underlying patterns in the data-set. It is primarily used in pattern detection and descriptive modelling [5]. Additionally, these algorithms help in deriving meaningful insights from the data. Some of the popular unsupervised learning algorithms are k-means clustering [6], apriori algorithm [7] and k nearest neighbors [8].

- **Reinforcement Learning** - Reinforcement learning is the learning of a mapping from situations to actions so as to maximize a scalar reward or reinforcement signal. The learner is not told which action to take, as in most forms of machine learning, but instead must discover which actions yield the highest reward by trying them [9].

2.2 Neural Networks

Neural Networks are a subset of machine learning that are inspired from the network of neurons found in the brain. Neurons form the basis of computation for the neural networks and are connected in a closed network in order to process data[10]. The ability to learn depends on a neural network's weights, activation function, etc. The primary aim of the neural networks is to find a set of weights that could solve a complex problem with the highest precision. Weights of a neural network are trained using back-propagation which makes it feasible to update the weights in the inner layers of a deeply stacked neural network. For this project we will mainly be focusing on deep-learning models that are trained using a supervised learning approach.

- **Deep Neural Networks (DNN)** – DNNs are neural networks that essentially contain a stack of layers, interconnected to each other. DNNs make use of the plethora of layers in their architecture for extracting features from the dataset. The stacks of layers in these networks empower them to fit complex non-linear functions that are not feasible in traditional machine learning models[11]. Figure 2-1 shows the skeleton of a typical deep neural network.

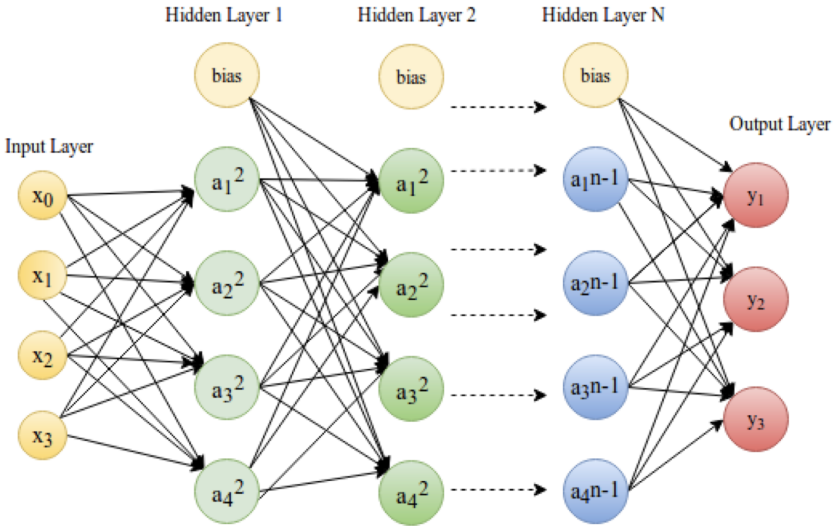


Figure 2-1: Deep Neural Network[12]

- **Convolutional Neural Networks (CNN)** – CNNs are neural network models that utilise a pre-processed input image and allocate weights and biases to certain aspects of the input image in order to distinguish it among a set of images in the data. CNNs outperform the traditional neural network models available in the market mainly because they are able to capture the spatial and temporal dependencies of an image using the available kernels[13]. Figure 2-2 is an illustration of a CNN architecture built to be trained on MNIST dataset. CNNs form a combined feature map with the help of pooling layers to decrement the dimensions of other feature maps while still maintaining the essential details of the input image as shown in Figure 2-3. Moreover, CNN based models are mainly utilised for feature extraction in image recognition tasks.

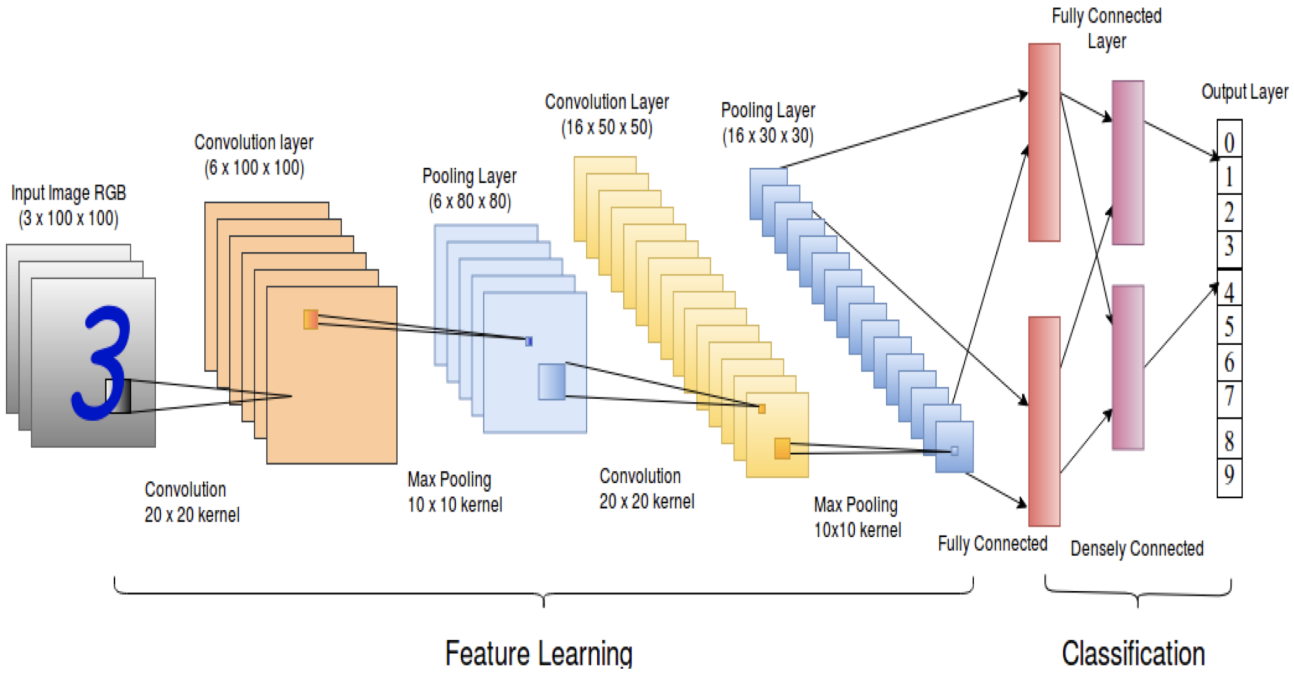


Figure 2-2: MNIST digit recognition using Convolutional Neural Networks[12]

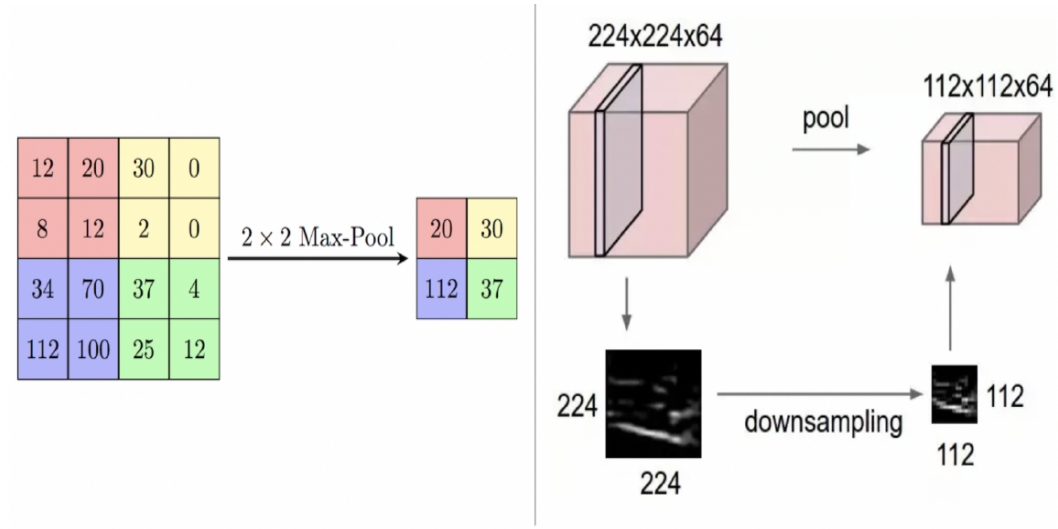


Figure 2-3: Max-pooling in CNN

2.3 Adversarial Threat Models

Apart from validating the precision of a neural network model on unseen data, the model's precision should also reflect its ability to be resilient to a plethora of adversarial attacks. The adversary can corrupt the training stage of the model's pipeline by inserting poisoned datapoints or it can observe the training of the model and make the model fail in the testing phase. This section explores the various adversarial threat models that can make a model fail.

2.3.1 Attack Scenarios

- **Evasion Attacks** - It is one of the widely used attack. It involves making use of adversarial insertions in order to manipulate a precise model. In this attack, the adversary perturbs the sample in the testing phase thereby forcing the model to produce imprecise results. Figure 2-4 shows an example of an Evasion attack found in real life.

Figure 2-4: Perturbation Made In Real Life vs Physically Created Perturbation[14]

- **Data Poisoning Attacks** - In this attack, an adversary attempts to hamper the performance of a model by inserting adversarial samples in the training dataset. This leads to the manipulation of decision boundaries thereby forcing the model to produce misclassifications.

2.3.2 Adversarial Capabilities Exploration

Adversaries tend to investigate the behaviour of the model to obtain as much information as they can which is directly linked to the Adversarial Threat Model. The greater the amount of information in the hands of the adversary, the greater is the threat it poses it to the model. Adversarial attacks are mainly conducted in two main phases of a model's life-cycle – training phase and testing phase.

- **Training Phase** - In a white box setting, the adversary mainly targets a model by tinkering with the training data in order to handicap the entire training process of the model. In order to perform adversarial attack in the training phase, the adversary can use mainly three different strategies -

    1. **Data Injection** - The adversary does not possess access to the training data and the underlying training algorithm; however, the adversary can append poisoned data samples in the training dataset to hamper the learnings of the model.

2. **Data Modification** - In this scenario, the adversary does not possess access to the underlying learning algorithm, however, it has complete access to the training dataset. Moreover, the adversary can corrupt the training samples in the training dataset to manipulate the training of the model.

3. **Logic Corruption** - In this scenario, the adversary has complete access of the model parameters. Hence, the adversary can fiddle around with the targeted model thereby completely altering it.

- **Testing Phase** - Most attacks on neural-network models occur in the testing phase. The adversary does not have access to the training phase but has the ability to make the model misclassify by altering the data fed to the model during its testing phase. The probability of success of an adversary primarily depends on the information it possesses about the model. Furthermore, the odds of success of an adversary in a white box setting will be usually higher as compared to an adversary in a black box setting.

1. **White-Box attack** – In this type of attack, the adversary possesses complete information about the model including its data distribution. Furthermore, the adversary can tinker with the parameters of the model. With the available information, the adversary is able to exploit the vulnerabilities by planting carefully crafted samples.

2. **Black-Box attack** – In this type of attack, the adversary does not possess any information about the model. However, the adversary uses the model as an oracle and uses a series of carefully crafted samples to detect vulnerabilities in the model using the output it generates. Black-Box attacks can be further divided into three kinds namely – Adaptive Black-Box attack, Non-Adaptive Black-Box attack, and Strict Black-Box attack. The difference between them depends on the varying level of access in the hands of the adversary.

2.3.3 Adversarial Samples

Adversarial samples are specifically generated samples that are designed to delude a machine learning model into producing erroneous results. For this project, we primarily emphasize on adversarial samples generated via evasion attacks as they are among the most widely used attacks on machine learning models.

2.4 Generation of Adversarial Samples

Adversarial samples generation falls under the category of Evasion attack because of their ability to make a classifier produce flawed results. Moreover, this attack primarily targets the testing phase where the model is exposed to the outside world for making predictions.

2.4.1 Testing-Deployment Phase

The adversary can craft adversarial samples that are similar to the samples found in the testing data-set of the models. Attacks in the testing phase can take place both in a white-box as well as black-box setting.

2.4.1.1 Adversarial Sample Generation in a White-Box Setting

In a paper in 2015, Papernot et al [15] proposed a generalised 2 step adversarial sample crafting mechanism which is the backbone of several white-box evasion attacking methodologies. The detailed mechanism is discussed below-

- **Stage One – Direction Sensitivity Estimation**: The aim of this step is to determine the dimensions of the input sample(X) that will produce the expected adversarial behaviour with the least amount of perturbation [15]. In order to achieve this, the adversary evaluates the sensitivity of the trained model(F) for every input feature and varying classes. This helps in identifying every direction in the data manifold around the sample. Figure 2-5 shows how each direction is identified in the data manifold around sample X, in order to build this knowledge. Several methods like fast gradient sign method[16] and forward derivative[17] make use of this mechanism.

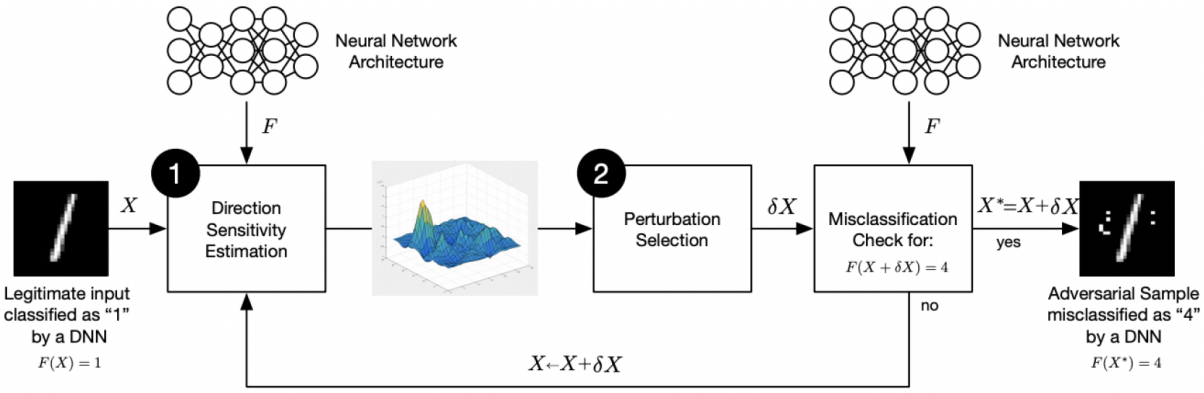


Figure 2-5: Adversarial Sample Crafting Mechanism for Evasion Attacks[15]

- **Stage Two – Perturbation Selection**: The aim of this step is to exploit the knowledge gathered in step one to find the requisite adversarial perturbation(δX) [15]. However, the total amount of perturbation used for generating an adversarial sample should be minimalist in order to ensure that the sample is indistinguishable via naked eye.The mechanism can be encapsulated using the expression given below -

$$X_* = x + \underset{\delta X}{arg\,min}\{||\delta X|| : F(X+\delta X) \neq F(X)\} \tag{2.1}$$

Some of the techniques of adversarial attack are –

- **Fast Gradient Sign Method (FGSM)** – This method is based on the two-stage mechanism discussed before. This method generates adversarial samples by adding perturbations having direction parallel to the gradient of the cost function with respect to the dataset used to train the neural-network model[12]. The perturbation is scaled using the epsilon value constrained using maximum normalization. Adversarial samples generated via this method have the ability to confuse a well-trained neural-network model. The concept is summarized via the an expression -

$$X_* = X + \varepsilon * sign\nabla_x J(\theta, X, y_{true}) \tag{2.2}$$

where

- X_* : Adversarial image (Output)
- X : Original Image
- y_{true} : Original input label
- ε : Multiplier for smaller perturbations
- θ : Model parameters
- J : Loss of model

The two variants of FGSM mechanism are -

1. **Target Class Method** [18] – This variant maximises the probability of a specific target class which results in incorrect allocation of the true class given an input sample. This variant can be summarized using the following expression -

$$X_* = X - \varepsilon * sign \nabla_x J(\theta, X, y_{target}) \tag{2.3}$$

where

- X_* : Adversarial image (Output)
- X : Original Image
- y_{target} : Targeted label
- ε : Multiplier for smaller perturbations
- θ : Model parameters
- J : Loss of model

2. **Basic Iterative Method** [18] – This variant is based on the fundamentals of Fast Gradient Sign Method (FGSM). This variant generates adversarial samples via small iterative steps. However, this method does not rely on the approximation of the model. Furthermore, an increase in the number of iterations lead to increase in the intensity of the adversarial samples. This variant can be summarized using the following expression -

$$X_*^0 = X; X_*^{(n+1)} = Clip_{X,e}\{X_*^{(n)} + \alpha * sign \nabla_x J(\theta, X_*^n, y_{true})\} \tag{2.4}$$

where

- α : Step size
- $Clip_{X,e}\{A\}$: Element-wise clipping of X

- **Projected Gradient Descent (PGD)** - It is an iterative attacking mechanism and can be regarded as an iterative version of Fast Gradient Sign Method (FGSM). The difference between the two mechanisms is that Projected Gradient Descent (PGD) has a greater number of iterations and appends additional randomization, thereby making Projected Gradient Descent more effective than Fast Gradient Sign Method (FGSM) [19].

2.4.1.2 Adversarial Sample Generation in a Black-Box Setting

As mentioned before, there are three main variants of Blacks-Box adversarial attack namely –

- **Adaptive Black-Box Attack** – In this variant, the adversary does not possess access to the training data but has the ability to use the model as an oracle in order to perform queries using the type of dataset used to train the model. This helps the adversary in selectively augmenting the data to form adversarial samples.

- **Non-Adaptive Black-Box Attack** – In this variant, the adversary has access to the dataset used to train a model. Furthermore, this allows the adversary to observe the decision boundaries of the model, thereby providing the adversary, the requisite information to generate adversarial samples for the targeted model.

- **Strict Black-Box Attack** – In this variant, the adversary has access to the dataset, which enables the adversary to train a local substitute model. Furthermore, if the local substitute models obtain a high-confidence training result, the adversary can make use of any white-box attacking methodology to generate adversarial samples.

The Strict Black-Box variant uses the property – Transferability Property of Neural Network which was presented by Papernot et al [20]. This property states that adversarial samples generated by training a model using an architecture, can also affect another model with a non-comparable architecture. Taking a model(F) and another model with a different architecture (F') as an example. An adversary wants to target model F but only has access to its dataset. Using a different model F', trained on the same dataset, the adversary can acquire information using the adversarial samples generated for the model F' and transfer it to the model F. This concept consists of two techniques –

- **Intra-Technique Transferability** – Models F and F' use the same machine learning approaches (ex. both use Neural Network (NN) or both use Random Forest)

- **Cross-Technique Transferability** – Models F and F' use different machine learning approaches (ex. Model F uses Neural Networks while on the other hand, Model F' uses Random Forest.

These techniques provide an adversary the requisite information to attack the model without the need to adapt similar model architecture, learning algorithm as well as hyperparameters in a black-box setting.

2.5 Adversarial Attack Defences and Countermeasures

Adversarial attacks have portrayed to target a plethora of modern state-of-the-art machine learning architectures due to several attacking vectors found during the overall lifecycle of the development of a machine learning model, starting from the training phase to the deployment phase. Some of the difficulties in protecting a model are -

- **Difficulties in developing theoretical models of the crafting process of an adversarial sample** – Currently, there is a lack of proper tools needed to define solutions to the convoluted optimization problems. Moreover, this makes it a herculean task to come up with any theoretical arguments that a specific defence can be used to mitigate a set of adversarial samples.
- **Proper Outputs provided by Machine Learning models for every possible input** – Even a minute adjustment results in inconsistency in the performance as well as the output generated by a model.

Majority of the defence strategies are not versatile enough to counter all the different types of adversarial attacks in the market. Furthermore, some defence strategies may result in the performance overhead of the model due to its complexity. Thereby, leading to a decrease in the overall precision. A few of the widely used defence strategies are discussed in the upcoming sub-sections.

2.5.1 Adversarial Training

Adversarial training involves training the model on adversarial samples in hopes of making it resilient to adversarial samples [21]. It is considered more of a brute force approach where the defender of the model attempts to generate as many adversarial samples a possible in order to use them as a part of the training dataset. Furthermore, augmentation can be done by combining both the original samples as well as the generated adversarial samples as shown by Kurakin et al. [18]. Goodfellow et [16] also presented a modified objective function in which the model is capable of learning from the adversarial samples. The main objective is to strengthen the model by ensuring that it classifies the same class for both the original sample as well as the perturbated sample. The downside of adversarial training is that it is only applicable if the adversarial samples are generated on the original model. Moreover, under a black-box setting, it is not adequate considering the adversary generates adversarial samples on a substitute model instead. In a study conducted by Papernot et al.[22], it was concluded that adversarial training could be evaded using a black-box attack with rates up to 71.25.

2.5.2 Defensive Distillation

Defensive Distillation involves transferring knowledge from a teacher neural network to a student neural network. The idea for his mechanism was proposed by Hinton et al.[23]. The defence strategy is described as follows. Assuming model (F) is trained with a dataset (X) into the target classes (Y), acquiring maximum accuracy. By extracting out the probability distribution over (Y) from the final softmax layer, the output is used to train another model (F') with the same architecture and same dataset (X). With the "soft" labels from the model (F), the new labels contain additional information about the membership of dataset (X) to the

different target classes. This acts as an additional filter for model (F'), giving it an element of randomness, strengthening the algorithm. The advantage of this defence strategy is that it is much more adaptable to unknown adversarial attacks as it is much more dynamic and depend less on human intervention. However, model (F') is bounded by the general rules of model (F). Reverse engineering can be done to discover exploits if given enough computing power to adversaries. Also, data poisoning attacks can affect distillation models by corrupting the initial training dataset (X). Under a black box setting, this defence approach can be evaded as presented by Carlini and Papernot et al.[24]. Recent advancements have allowed for substantial transferability of adversarial examples across neural network models.

2.5.3 Feature Squeezing

Feature Squeezing involves reducing the complexity of the input data. This reduces the sensitivity in the gradient direction, thereby minimizing the adversarial perturbations. This approach was proposed by Xu et al.[25]. It comprises of two operations on the sample image.

- Reduction of colour depth on a pixel level.
- Applying a smoothing filter on the samples images allows for multiple inputs to be mapped into a single value. This ensures that the model is resilient from any noise and adversarial attacks.

The only downside of this defence strategy is that it may affect the precision of the model due to its high complexity.

2.5.4 Blocking Transferability

Most of the defence mechanisms are susceptible to attacks under a black-box setting, specifically due to the transferability property of neural networks. A defence mechanism was devised by Hosseini et al.[26] with the help of a three step Null Labeling method in order to prevent the transferability of adversarial samples from one neural network model to another. This approach works by augmenting a new NULL label in the dataset and have the classifier to train based on the new label. This enables the classifier to reject instances of adversarial samples in the dataset by denoting them as NULL. This mechanism comprises of three main steps –

- **Initial training** – perform training with original dataset in order to form the decision boundaries for the model.
- **Compute NULL Probabilities** – Probabilities that belong to NULL class is computed using an expression -

$$P_{null} = f(\frac{||\delta X||_0}{||X||}) \tag{2.5}$$

  where

  - δX : Perturbation
  - $||\delta X||_0 \backsim U[1, N_{max}]$, where N_{max} refers to the minimum number for which the $f(\frac{||\delta X||_0}{||X||})$ = 1

- **Adversarial Training** – The original model is retrained on a combination of original samples as well as perturbated samples. The probabilities of the NULL labels is used to obtain the labels of the training data.

This defence mechanism is widely regarded as the most effective countermeasure for both white-box as well as black-box attacks without hampering the performance of the model.

3 Experimental Pre-Requisites

This section discusses the details of the experiment to be performed, chosen datasets for benchmarking as well as the chosen neural-network architectures on which the experimentation will be performed. Furthermore, it also discusses the aspects of the proposed defence mechanism.

3.1 Popular CNN Architectures

To decide on the CNN architectures for this experiment, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) project is used as the benchmarks for choosing the CNN architectures.

ILSVRC is based on the ImageNet project that utilizes an extensive visual database for visual object recognition software research. This project is organized as an annual software context, where software programs are developed and proposed to compete in the classification and detection of images, objects, and scenery. Below is the list of CNN that have come top in this contest from the year 2012 to 2015 -

- **LeNet**: Developed by Yann Lecun et al [27], consisting of 60,000 parameters.

- **AlexNet**: Developed by Alex Krizehvsky et al [28], which came 1st in the challenge in the year 2012. The model consists of 60,000,000 parameters and achieve a Top-5 error rate of 15.5

- **ZFNet**: Developed by Matthew Zeiler and Rob Fergus [29], which came 1st in the challenge in the year 2013. The model has no fixed parameters as it is dependent on the given convolutional layers. It achieved a Top-5 error rate of 14.8

- **Inception**: Developed by Google [30], which came 1st in the challenge in the year 2014. The model consists of 4,000,000 parameters and achieve a Top-5 error rate of 6.67

- **VGGNet**: Developed by K.Simonyan et al [31], which was the 2nd runner up in the challenge in the year 2014. It won other awards in this challenge and became widely used in fields of Machine Learning. The model consist of 138,000,000 parameters and achieved a Top-5 error rate of 7.3

- **ResNet**: Developed by Kaiming et al [32], which came 1st in the challenge in the year 2015. Like ZFNet, the model has no fixed parameters as it is dependent on the given convolutional layers, which in the case during the challenge, it utilized 152 layers. It managed to achieve a Top- 5 error-rate of 3.6

3.2 Chosen Architectures

3.2.1 VGG-16

VGG16 is a Convolutional Neural Network proposed by K.Simonyan and A.Zisserman from the University of Oxford in the paper ”Very Deep Convolutional Networks for Large-Scale Image Recognition” [31]. The model is based on AlexNet due to its similarities except it replace its filters with multiple (3 x 3) kernel-sized filters chronologically, allowing VGG16 to surpass AlexNet in performance. This model was submitted to the ”ImageNet Large Scale Visual Recognition Challenge” [33] back in the year 2014, and performed well compared to other proposed models, winning multiple awards. Soonafter,it gained recognition and is one of the conventional models used for machine learning experiments today.

There are different types of VGGNet models, typically, VGG13, VGG16, and VGG19. The difference between these variants is the total number of layers in the neural network. For this experiment, VGG16 is chosen as the model of choice. The model consists of twelve Convolutional layers, some of which are followed by maximum pooling layers, and then four fully connected layers and lastly, a 1000-way soft-max classifier. The architecture of VGG16 is shown in Fig 3-1 below.

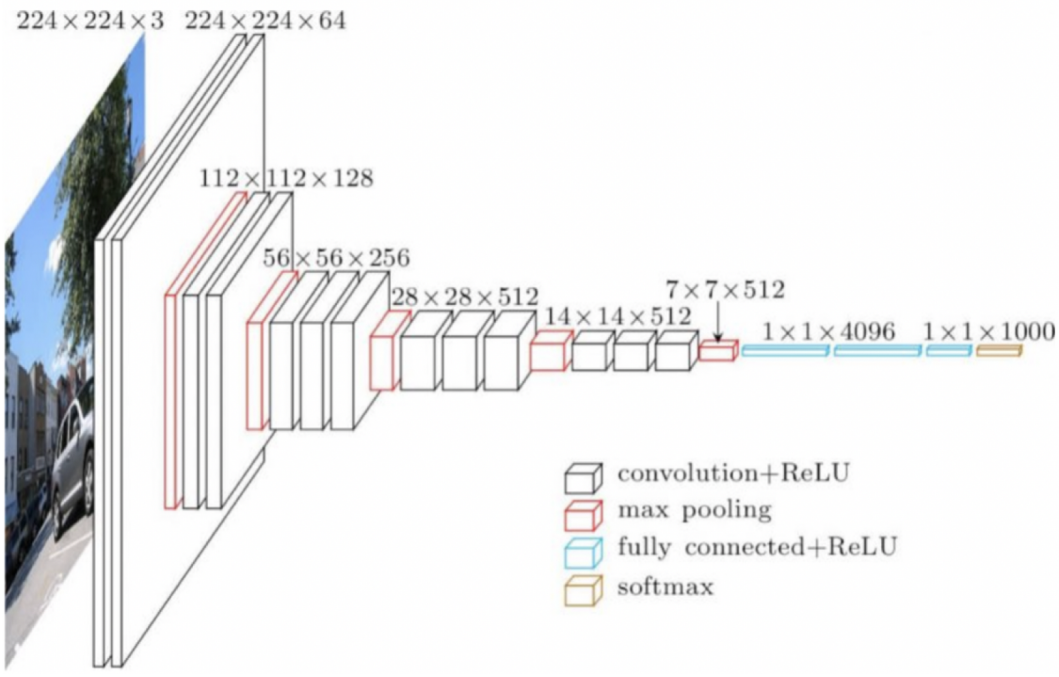


Figure 3-1: VGG-16 Architecture [34]

During the ILSVRC, K.Simonyan et al. initialize the model with a (224 x 224 x 3) tensor dimension input. The input image passes through a stack of convolutional layers, where each layer consists of (3 x 3) feature filters. It also utilizes a (1 x 1) convolution filter that acts as a linear transformation of input channels. The convolution stride is fixed to one pixel; spatial padding of a convolution layer input is such that the spatial resolution is preserved after each

convolution. It means that for each of the (3 x 3) convolution layers, the padding is one pixel.

Within the model, it has five max-pooling layers which perform spatial pooling. Max-pooling is performed over a (2 x 2) pixel filter, with two strides. Lastly, the output of the convolutional layer passes through three fully-connected layers. The first two layers consist of 4096 channels each. The third fully-connected layer performs a 1000-way classification for each class. This means that for each class input into the model, there are 1000 channels. The final layer of the model is the soft-max layer, which determines the probability of the output. This configuration of the fully connected layers applies to all networks that utilize the VGGNet architecture.

3.2.2 ResNet-50

ResNet stands for Residual Network, it was originally proposed by Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun in their paper titled "Deep Residual Learning for Image Recognition" [32], published in 2015. ResNet gained popularity after it attained the top position at the ILSVRC 2015 classification competition with an error rate of 3.57 percent. ResNet has several variants that are built upon the same underlying structure and the differentiating factor is the number of layers in a specific Residual Network based architecture.

Deep Neural Networks use a stack of layers in order complex complex problems. However, the increased number of stacked layers leads to a deterioration in the model's accuracies. ResNet is able to tackle this problem by making use of residual blocks using the concept of "skip connections". Skip connections alleviate the vanishing gradient issue by setting up an alternate shortcut for the gradient to pass through. Additionally, it enables the model to learn an identity function, thereby ensuring that the higher layers of the model do not perform any worse than the lower layers [35].

ResNet-50 consists of five stages each with a convolution and identity block. Each convolution block has 3 convolution layers and each identity block also has 3 convolution layers. The detailed architecture is discussed below -

- A convolution with a kernel size of 7 * 7 and 64 different kernels all with a stride of size 2 giving us 1 layer. Then there is a max pooling layer with also a stride size of 2.
- In the next convolution there is a 1 * 1,64 kernel following this a 3 * 3,64 kernel and at last a 1 * 1,256 kernel, Furthermore, three layers are repeated for a total 3 times, thereby giving us 9 layers in this step.
- Next there is a kernel of 1 * 1,128 after that a kernel of 3 * 3,128 and at last a kernel of 1 * 1,512 this step is repeated 4 time, thereby producing 12 layers in this step.
- After that there is a kernel of 1 * 1,256 and two more kernels with 3 * 3,256 and 1 * 1,1024 and this is repeated 6 time, producing a total of 18 layers.
- And then again a 1 * 1,512 kernel with two more of 3 * 3,512 and 1 * 1,2048 and this is repeated 3 times, producing a total of 9 layers.

- In the end, we have an average pooling layer, fully-connected layer consisting of 1000 nodes with a softmax function.

In the end we get a 1 + 9 + 12 + 18 + 9 + 1 = 50 layers Deep Convolution network. The architecture for ResNet-50 is shows in Fig 3-2 below -

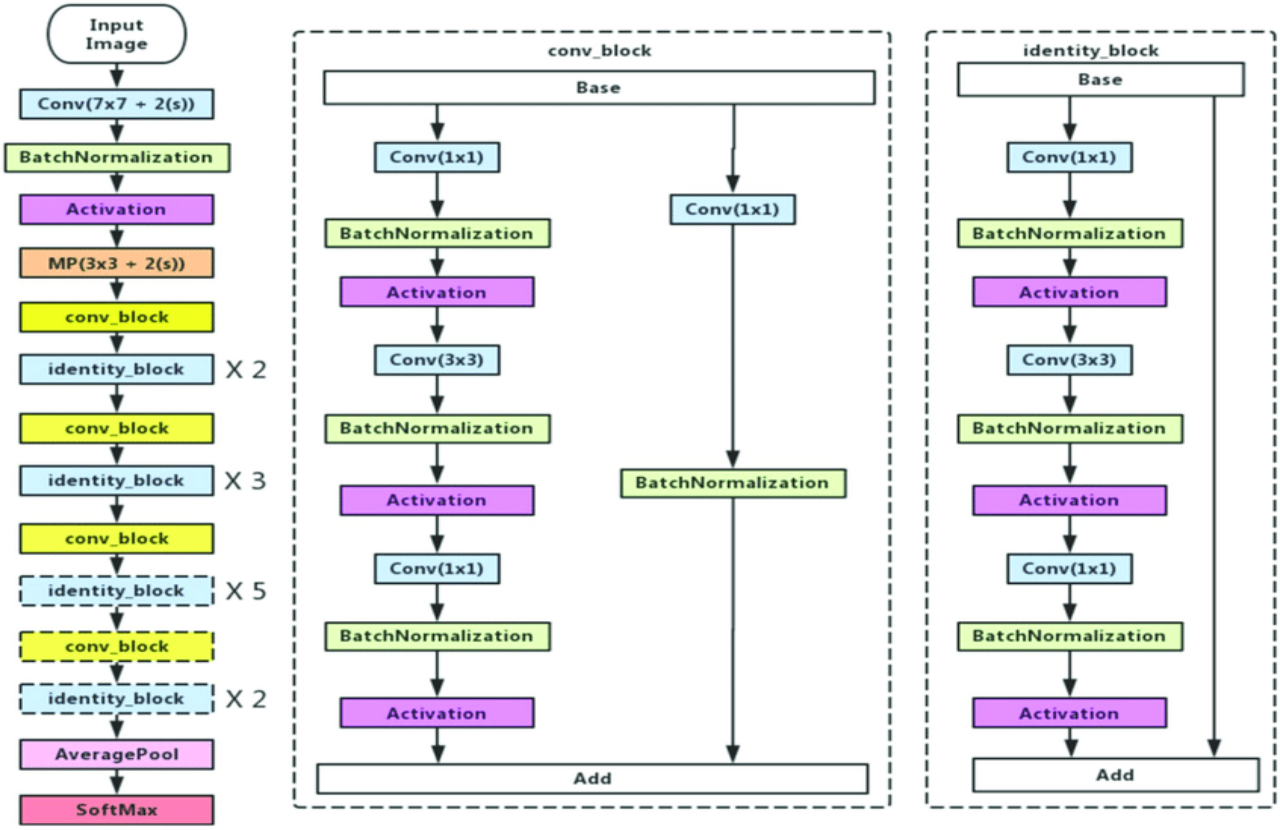


Figure 3-2: ResNet-50 Architecture [36]

3.2.3 Inception-V3

Inception is a deep convolutional neural network architecture which was first introduced by Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke and Andrew Rabinovich in their paper “Going Deeper with Convolutions” published in 2014 [30]. Inception network rose to fame after it won the ILSVRC in 2014 and made substantial improvement over ZFNET(winner in 2013) [29] , AlexNet(winner in 2012) [28] and also had a relatively low error rate compared to the other architectures.

Deeply stacked networks like VGG-16 are able to achieve remarkable results on the ImageNet dataset but deploying such architectures is computationally expensive, this was found by Hussam Qassim, David Feinzimer and Abhishek Verma in their paper ”Residual Squeeze VGG16” [37]. Furthermore, very deep convolution networks are highly susceptible

to overfitting as explained in the paper ”Reducing Overfitting in Deep Convolutional Neural Networks Using Redundancy Regularizer” [38].

Inception architecture is a 27 layer deep network and primarily makes use of “Inception Layer” [30], which is a combination of several layers – 1x1 convolutional layer, 3x3 convolutional layer and 5x5 convolutional layers with their output filter banks concatenated into a single output vector in order to form the input for the corresponding layers. Inception architecture is able to mitigate the overfitting issue by making use of 1x1 convolution for dimensionality reduction thereby leading to reduced computation. To improve convergence on this relatively deep network, the authors also introduced additional losses tied to the classification error of intermediate layers. This trick is only used for training, and the output of these layers is discarded during inference. Figure 3-3 shows a single Inception module. In this project, we make use of the Inception V3 variant of the original Inception architecture [39].

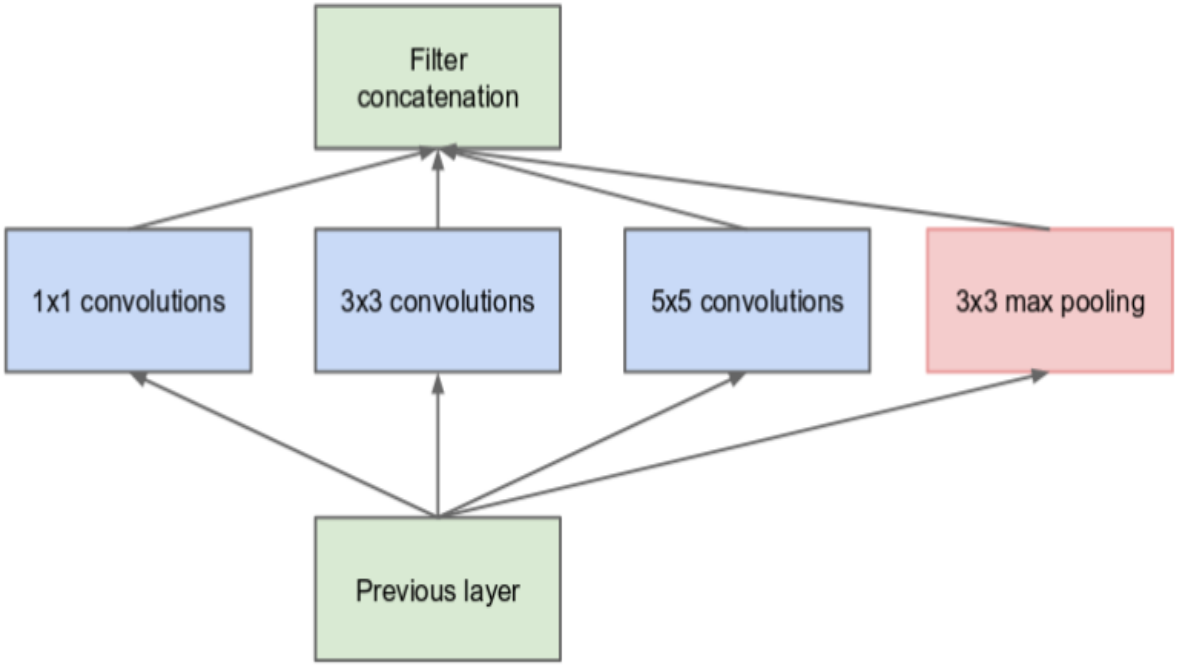


Figure 3-3: Inception Module [30]

3.3 Chosen Datasets

For this project we chose two benchmarking datasets mainly -

- **CIFAR-10 (Canadian Institute For Advance Research 10)**[40] - is a collection of images that represents natural-coloured images. It has a training set of 50,000 examples and a testing set of 10,000 examples. The images have a dimension size of (32 x 32), with pixel values ranging from 0 to 255 and possesses three channels (red, green and blue). It is widely used to evaluate image recognition capabilities of deep-learning models. Images of the dataset are taken under various light conditions, size and position, with many, occlude objects being partially present. Figure 3-4 shows some of the samples present in the CIFAR-10 dataset.

Figure 3-4: Samples from CIFAR-10 dataset[40]

- **CIFAR-100(Canadian Institute For Advance Research 100)**[41] - is a collection of images that represents natural-coloured images. It has a training set of 50,000 examples and a testing set of 10,000 examples. The images have a dimension size of (32 x 32), with pixel values ranging from 0 to 255 and possesses three channels (red, green and blue). It is widely used to evaluate image recognition capabilities of deep-learning models. Images of the dataset are taken under various light conditions, size and position, with many, occlude objects being partially present. It is an extension of the CIFAR-10 dataset which contains 10 different classes of images, while on the other hand, CIFAR-100 dataset contains 100 different classes of images. Figure 3-5 shows some of the samples present in the CIFAR-100 dataset.

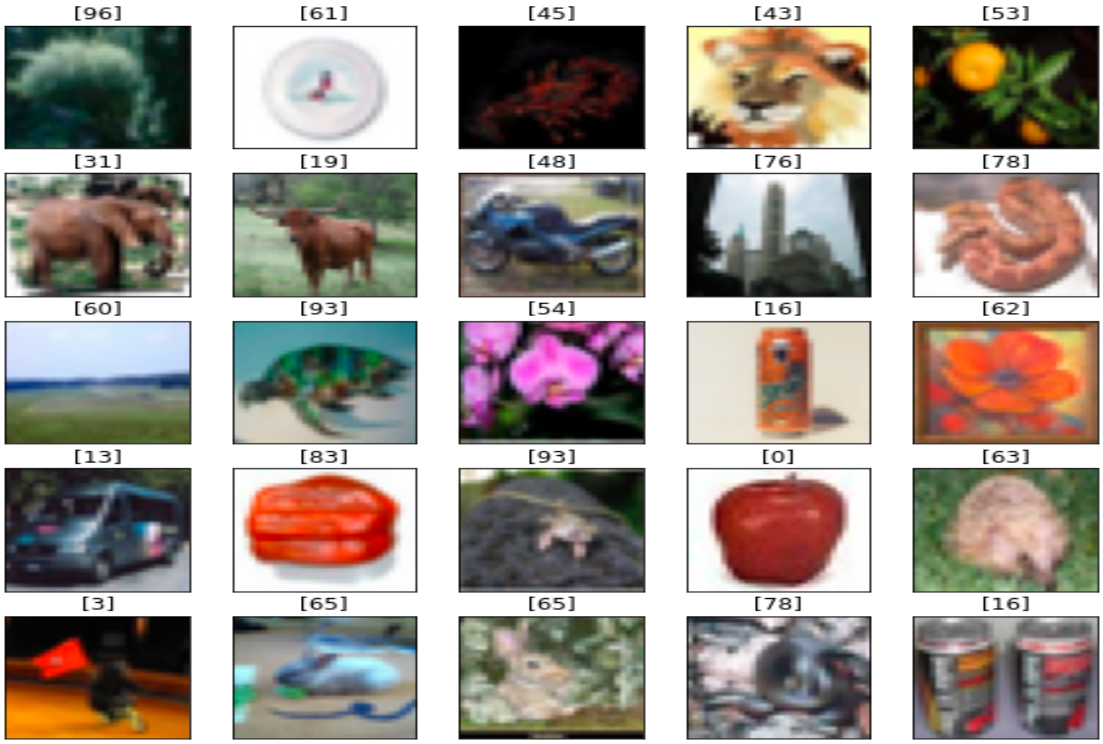


Figure 3-5: Samples from CIFAR-100 dataset[41]

These two datasets were chosen mainly because they require minimal amount of pre-processing and augmentation.

3.4 Chosen Attacks

In this section, we primarily discuss the adversarial attacks chosen for the experiment.

- **Untargeted Fast Gradient Sign Method Attack** – Untargeted FGSM attack also known as Basic Iterative FGSM Attack is planted in the testing phase of the experiment, in a white-box setting. As discussed in section 2.4.1.1, this method generates adversarial samples with the help of small iterative steps. Moreover, it does not rely on an approximation of the model. Additionally, it does not maximise the probability of a specific class as opposed to the other variant of FGSM attack (Target Class FGSM method) [16]. Figure 3-6 shows a adversarial crafted with the help of Untargeted FGSM attack.

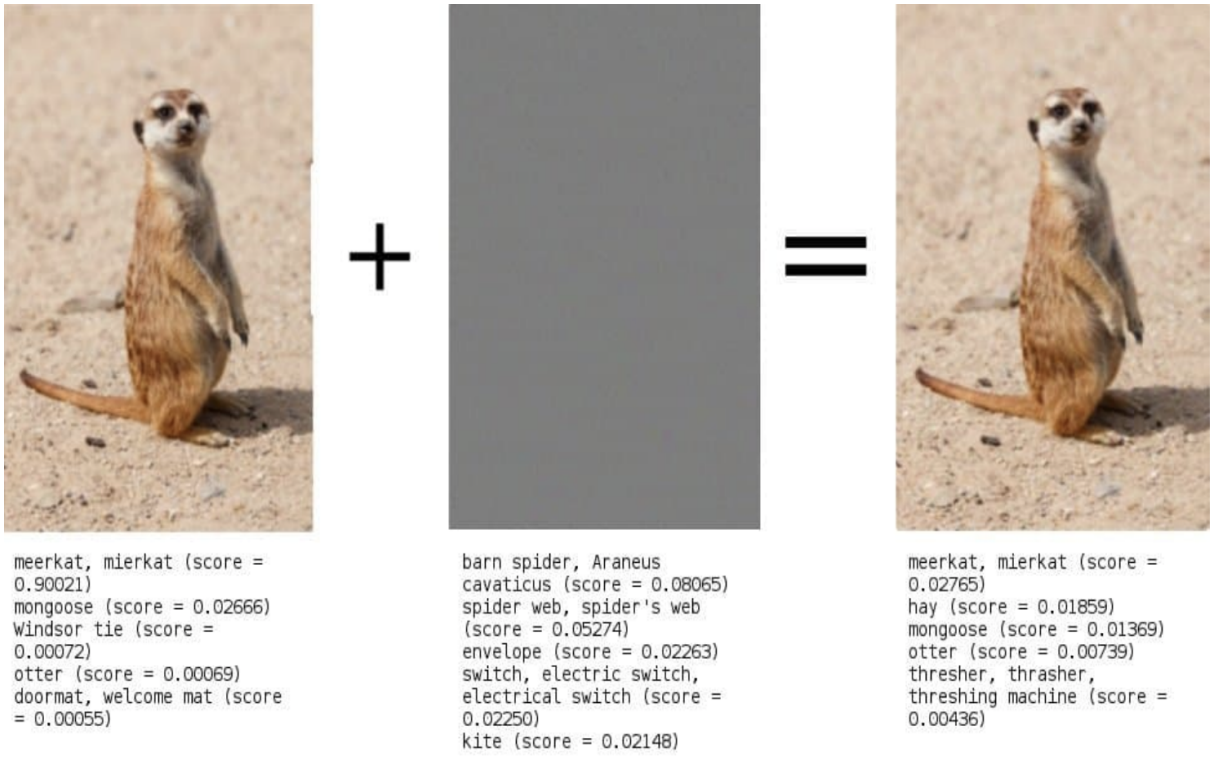


Figure 3-6: Adversarial Sample Crafted Using Untargeted FGSM attack [42]

- **Projected gradient Descent Attack** – PGD attack is an iterative attacking mechanism that exploits projected descent to iteratively craft adversarial samples in a white-box setting [19]. As discussed in section 2.4.1.1, this method performs one step of standard gradient descent and then clips all coordinates to be within the box.

3.5 Defense Methodology

In this section, we discuss the defence methodology that will be applied to our architectures in order to make them resilient from the chosen adversarial attacks.

For this project, we primarily use defensive distillation as a defence methodology. Defensive distillation was introduced by Nicholas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha and Ananthram Swami in their paper -"Distillation as a Defense to Adversarial Perturbations against Deep Neural Networks", published in 2015 [15]. Defensive distillation has been adapted from knowledge distillation, which involves transferring knowledge from a larger neural network to a small neural network as discussed in section 2.5.2 [43]. The main difference between defensive distillation and knowledge distillation is that we keep the exact neural network architecture to train the original as well as the distilled models

The defensive distillation procedure is discussed below –

1. The input of the defensive distillation training algorithm is a set of samples with their class labels. let X be a sample, Y (X) denotes its discrete label, also known as hard label. Y (X) is a one-hot encoded vector such that the only non-zero element corresponds to

the index of the correct class label in the vector.

2. Given this training dataset, we train a deep neural network F with a softmax output layer at temperature T, where F(X) is a probability vector over the class of all possible labels. The model F is also known as the teacher model. Using a high temperature value systematically reduces the neural network's sensitivity to small variations of its input when defensive distillation is applied. However, the temperature is decreased back to T=1 in order to make the probability predictions on unseen samples [15].

3. We build a new training dataset, by using samples of the form (X, F(X)) from the original training dataset. That is, instead of using hard label Y (X) for X, we use the soft-target F(X) encoding F's belief probabilities over the label class. Class probabilities help in encoding additional information regarding all the classes thereby enabling the model to deduce relative information from this extra entropy [15].

4. Using the new training dataset, we then train another neural network model, with the same neural network architecture as F and same temperature. This new model is known as the distilled model or the student model. This student model is further used to make predictions on unknown data samples. Figure 3-7 shows an overview of the defense mechanism based on transferring of knowledge contained in probability vectors with the help of distillation

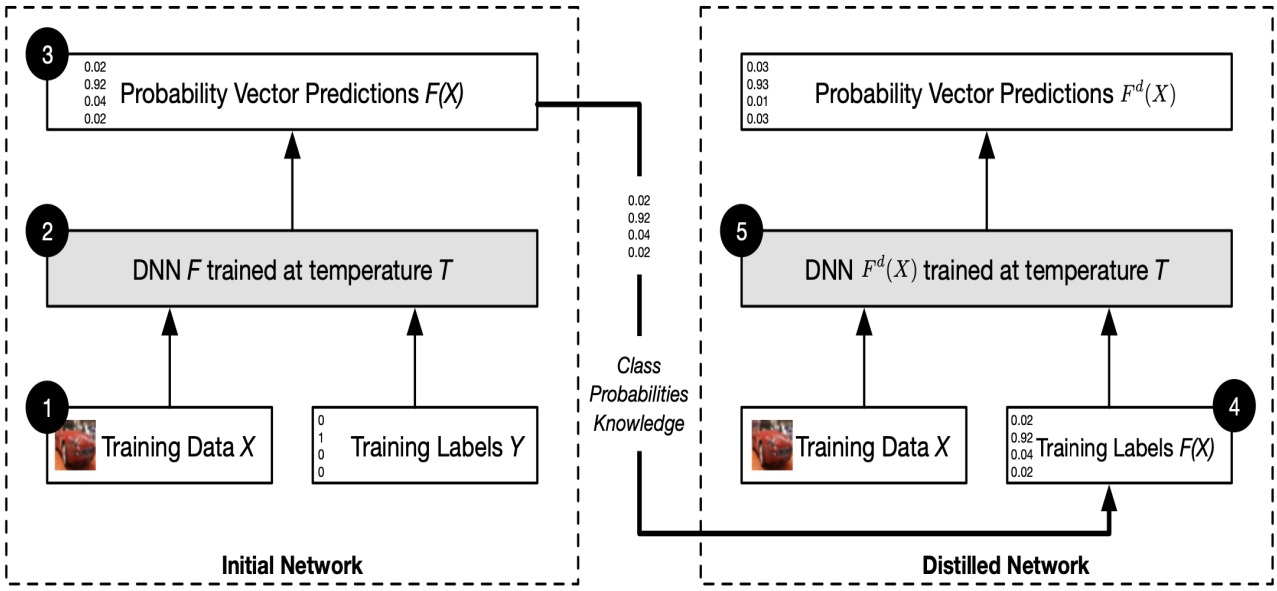


Figure 3-7: Defensive Distillation [15]

4 Experimental Methodology

This section primarily deals with -

- Implementation of the three different neural network architectures as discussed in section 3.2 .
- Details regarding the training of the architectures on the selected benchmarking datasets mentioned in section 3.3 .
- Implementation of the selected white-box adversarial attacking methodologies to craft adversarial samples, as discussed in section 3.4 .
- Implementation of an adversarial attack defence methodology as discussed in section 3.5.

4.1 Environment Setup and Libraries used

For this project, we primarily make use of Google Co-laboratory as the integrated development environment to develop and run .ipynb notebooks. In order to utilise the local GPU for running TensorFlow models, we make use of **jupyter_http_over_ws library** [44]. This library is used to generate a jupyter notebook instance token, which is then used to mount an instance of google colab over the jupyter notebook instance, thereby enabling us to run google colab locally. Running an instance of google colab locally provides us with the access to all the locally available GPU devices present in the system. Furthermore, google colab is preferred over jupyter notebook as the former takes care of all the dependencies of various libraries and prevents any rollbacks from occurring in the library versions.

Below is the list of libraries with their respective versions, used for this project -

- Python version: 3.6.9
- CUDA v10.0.130
- tensorflow==2.2.0
- Matplotlib version: 3.1.3
- scikit-image==0.17.2
- scikit-learn==0.24.2
- cleverhans
- jupyter-http-over-ws==0.0.8
- Keras==2.4.3
- Keras-Applications==1.0.8

- Keras-Preprocessing==1.1.2
- numpy==1.19.1
- pandas==1.1.1
- Pillow==7.2.0

4.2 Data Preprocessing

In order to pre-process our data before using it to train our models, we make use of Random Erasing Data Augmentation. This Augmentation Strategy was Proposed by Zhun Zhong, Liang Zheng, Guoliang Kang, Shaozi Li and Yi Yang in their paper "Random erasing Data Augmentation" [45]. Random erasing is a parameter learning free, easy to implement Data Augmentation process that can be integrated with a plethora of CNN-based architectures.

Random Erasing is conducted with a certain probability p. For an image in a batch, the probability of it undergoing the Random Erasing procedure is P. Random Erasing randomly selects a rectangular region in an image and erases its pixels with random values. This technique generates training samples with various levels of occlusion which helps in reducing the risk of over-fitting. Figure 4-1 shows an image before and after applying Random Erasing Data Augmentation technique.

(a) Original image **(b)** Random erasing image

Figure 4-1: Image before and after applying Random Erasing Augmentation

Figure 4-2 shows the detailed algorithm for the Random Erasing Data Augmentation procedure.

Algorithm 1: Random Erasing Procedure

Input : Input image I;
Image size W and H;
Area of image S;
Erasing probability p;
Erasing area ratio range s_l and s_h;
Erasing aspect ratio range r_1 and r_2.
Output: Erased image I^*.
Initialization: $p_1 \leftarrow$ Rand (0, 1).

if $p_1 \geq p$ **then**
  $I^* \leftarrow I$;
  return I^*.
else
  while *True* **do**
    $S_e \leftarrow$ Rand $(s_l, s_h) \times S$;
    $r_e \leftarrow$ Rand (r_1, r_2);
    $H_e \leftarrow \sqrt{S_e \times r_e}$, $W_e \leftarrow \sqrt{\frac{S_e}{r_e}}$;
    $x_e \leftarrow$ Rand $(0, W)$, $y_e \leftarrow$ Rand $(0, H)$;
    if $x_e + W_e \leq W$ *and* $y_e + H_e \leq H$ **then**
      $I_e \leftarrow (x_e, y_e, x_e + W_e, y_e + H_e)$;
      $I(I_e) \leftarrow$ Rand (0, 255);
      $I^* \leftarrow I$;
      return I^*.
    end
  end
end

Figure 4-2: Random Erasing Data Augmentation Algorithm [45]

For both the datasets (CIFAR-10 and CIFAR-100), we use 50,000 images for **training** and 10,000 images for **validation**. We use one-hot encoding to encode the integer representations for the class labels to form a sparse N dimensional vector where the class label index is set to 1 and the rest of the values are set to 0. Additionally we build a **test** dataset using 200 images from the validation test which will be used to compare model performances.

4.3 Building Neural Network Models

In this section, we primarily discuss the implementation of the three neural network architectures trained on two different datasets, selection of hyperparameters as well as the reason behind the selections.

4.3.1 VGG16

VGG16 architecture is implemented using keras and fine-tuned by making use of several dropouts as well as batch-normalization layers to mitigate over-fitting issues. During the implementation, the batch normalization layer is always placed after the convolution layer and before the activation layer as it enables the network to produce activations with the desired distribution [46].

An up-sampling layer is used to increase the image-size to a desired dimension before passing it through to the further layers. Additionally Stochastic Gradient Descent optimiser(SGD) is used as an optimizer instead of Adam because the former has a ”smaller escaping time and tends to converge to flatter minima whose local basins have larger Radon measure, explaining its better generalization performance” as stated by Pan Zhou, Jiashi Feng, Chao Ma, Caiming Xiong, Steven HOI and Weinan E in their paper - ”Towards Theoretically Understanding Why SGD Generalizes Better Than ADAM in Deep Learning” [47]. Listing A.1 shows the function which is used to define and compile the VGG16 architecture.

The model uses categorical crossentropy as the loss function. Additionally a model checkpoint is used to save the best model (in .h5 format) based on the validation accuracy.

- **CIFAR-10**

  Table 4-1 shows the list of hyperparameters used for fitting the VGG-16 model, mentioned earlier, on CIFAR-10 dataset.

Table 4-1: Hyperparameters of VGG-16 architecture trained on CIFAR-10 dataset.

Hyperparameter	**Value**
Learning rate	0.1
Learning rate decay	1e-6
Weight decay	0.0005
Momentum	0.9
Nesterov	True
Batch size	128
Total Number of Classes	10
Epochs	200

- **CIFAR-100**

  Table 4-2 shows the list of hyperparameters used for fitting the VGG-16 model, mentioned earlier, on CIFAR-100 dataset.

Table 4-2: Hyperparameters of VGG16 architecture trained on CIFAR-100 dataset.

Hyperparameter	**Value**
Learning rate	0.1
Learning rate decay	1e-6
Weight decay	0.0005
Momentum	0.9
Nesterov	True
Batch size	32
Total Number of Classes	100
Epochs	200

4.3.2 ResNet-50

In order to implement the ResNet-50 architecture, transfer learning is used. Transfer learning is a term used to denote the process of reusing a model, trained on a specific problem, to train a model based on another related problem. Transfer Learning not only speeds up the training process but also leads to a more precise model [48]. We make use of the keras.application library in order to load a Renset-50 network that has been trained on the imagenet dataset. Since, the input shape required for this model is (224,224,3), we add an upsampling layer to bump the dimensions of the images to the requisite sizes.

ResNet-50 network has several batch-normalization layers and using pre-trained weights can cause issues with the batch normalization, if the target dataset on which the model is supposed to be trained, is different from the one which was originally used to train the model. Hence the batch normalization layers of the model are set as trainable in order to allow the weight updates during training. The model is fine-tuned further by using a set of sequential layers. Additionally the optimizer selected for this architecture is adam and the loss function used is categorical crossentropy.

The Listing 4.1 shows the function which is used to define and compile the ResNet-50 architecture

```
1 def Resnet50(num_classes , channel=3):
2
3   resnet_model = ResNet50(weights='imagenet', include_top=False, input_shape=(224,
        224, 3))
4
5   for layer in resnet_model.layers:
6     if isinstance(layer, BatchNormalization):
7       layer.trainable = True
8     else:
```

```
        layer.trainable = False

    model = Sequential()
    model.add(UpSampling2D((7,7)))
    model.add(resnet_model)
    model.add(GlobalAveragePooling2D())
    model.add(Dense(256, activation='relu'))
    model.add(Dropout(.25))
    model.add(BatchNormalization())

    model.add(Dense(num_classes, activation='softmax'))
    model.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
    return model
```

Listing 4.1: Function used to define ResNet-50 architecture

- **CIFAR-10**

  Table 4-3 shows the list of hyperparameters used for fitting the ResNet-50 model, mentioned earlier, on CIFAR-10 dataset.

Table 4-3: Hyperparameters of ResNet-50 architecture trained on CIFAR-10 dataset.

Hyperparameter	**Value**
Learning rate	0.001
beta_1	0.9
beta_2	0.999
amsgrad	False
epsilon	1e-07
Batch size	128
Total Number of Classes	10
Epochs	50

- **CIFAR-100**

  Table 4-4 shows the list of hyperparameters used for fitting the ResNet-50 model, mentioned earlier, on CIFAR-100 dataset.

Table 4-4: Hyperparameters of ResNet-50 architecture trained on CIFAR-100 dataset.

Hyperparameter	**Value**
Learning rate	0.001
beta_1	0.9
beta_2	0.999
amsgrad	False
epsilon	1e-07
Batch size	64
Total Number of Classes	100
Epochs	70

4.3.3 Inception-V3

This architecture also makes use of transfer learning [48]. We make use of the keras.application library in order to load Inception-V3 network that has been trained on the imagenet dataset. Since, the input shape required for this model is (224,224,3), we add an upsampling layer to bump the dimensions of the images to the requisite sizes.

Inception v3 network has several batch-normalization layers and using pre-trained weights can cause issues with the batch normalization, if the target dataset on which the model is supposed to be trained, is different from the one which was originally used to train the model.Hence the batch normalization layers of the model are set as trainable in order to allow the weight updates during training. The model is fine-tuned further by using a set of sequential layers. Additionally the optimizer selected for this architecture is adam and the loss function used is categorical crossentropy.

The Listing 4.2 shows the function which is used to define and compile the Inception-V3 architecture

```
def Inception(num_classes , channel=3):
  inception_model = InceptionV3(weights='imagenet', include_top=False,
      input_shape=(224, 224, 3))

  for layer in inception_model.layers:
    if isinstance(layer, BatchNormalization):
      layer.trainable = True
    else:
      layer.trainable = False
  model = Sequential()
  model.add(UpSampling2D((7,7)))
  model.add(inception_model)
```

```
model.add(GlobalAveragePooling2D())
model.add(Dense(1024, activation='relu'))
model.add(Dropout(.3))
model.add(Dense(512, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(256, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(num_classes, activation='softmax'))
model.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
return model
```

Listing 4.2: Function used to define Inception-V3 architecture

- **CIFAR-10**

  Table 4-5 shows the list of hyperparameters used for fitting the Inception-V3 model, mentioned earlier, on CIFAR-10 dataset.

Table 4-5: Hyperparameters of Inception-V3 architecture trained on CIFAR-10 dataset.

Hyperparameter	**Value**
Learning rate	0.001
beta_1	0.9
beta_2	0.999
amsgrad	False
epsilon	1e-07
Batch size	64
Total Number of Classes	10
Epochs	40

- **CIFAR-100**

  Table 4-6 shows the list of hyperparameters used for fitting the Inception-V3 model, mentioned earlier, on CIFAR-100 dataset.

Table 4-6: Hyperparameters of Inception-V3 architecture trained on CIFAR-100 dataset.

Hyperparameter	**Value**
Learning rate	0.001
beta_1	0.9
beta_2	0.999
amsgrad	False
epsilon	1e-07
Batch size	64
Total Number of Classes	100
Epochs	200

4.4 Application of Adversarial Attack

This section primarily focuses on the implementation of the selected white-box adversarial attacking techniques - Fast Gradient Sign Method (FGSM) and Projected Gradient Descent (PGD).

We make use of the cleverhans library developed by Ian Goodfellow and Nicholas Papernot [49]. Cleverhans is a python library used to benchmark machine learning models' vulnerability to adversarial samples. The name **'CleverHans'** is a reference to a presentation by Bob Sturm titled "Clever Hans, Clever Algorithms: Are Your Machine Learnings Learning What You Think?" and the corresponding publication, "A Simple Method to Determine if a Music Information Retrieval System is a 'Horse'." [50] Clever Hans was a horse that appeared to have learned to answer arithmetic questions, but had in fact only learned to read social cues that enabled him to give the correct answer. In controlled settings where he could not see people's faces or receive other feedback, he was unable to answer the same questions. The story of Clever Hans is a metaphor for machine learning systems that may achieve very high accuracy on a test set drawn from the same distribution as the training data, but that do not actually understand the underlying task and perform poorly on other inputs.

In order to craft a sample using FGSM, we need to set the following parameters -

- model - Refers to the output of trained_model.get_logits
- image - Refers to the actual image vector
- epsilon - Refers to the input variation parameter. For this experiment we set the epsilon values ranging from 0.1 to 0.8
- ord - This is an optional parameter and refers to the order of norm. It is set to np.inf by default for this experiment.
- targeted - This parameter is used to select if the attack should be a targeted variant of FGSM or an untargeted one. For this experiment, we craft FGSM samples only using the untargeted variant.

Listing 4.3 shows the implementation of crafting an untargeted FGSM sample using cleverhans.

```
fgsm_sample = fast_gradient_method(model, image, epsilon, np.inf, targeted=False)
```

Listing 4.3: Crafting an untargeted FGSM sample

In order to craft a sample using PGD, we need to set the following parameters -

- model - Refers to the output of trained_model.get_logits
- image - Refers to the actual image vector
- epsilon - Refers to the input variation parameter. For this experiment we set the epsilon values ranging from 0.1 to 0.8
- eps_iter - It is an optional parameter that refers to the step size for each iteration. It is set to 0.05 by default for this experiment.
- nb_iter - It is an optional parameter that refers to the number of attack iterations. It is set to 40 by default for this experiment.
- ord - This is an optional parameter and refers to the order of norm. It is set to np.inf by default for this experiment.

Listing 4.4 shows the implementation of crafting a PGD sample using cleverhans.

```
pgd_sample = projected_gradient_descent(model, image, epsilon, 0.01, 40, np.inf)
```

Listing 4.4: Crafting an untargeted FGSM sample

For this project, we generate adversarial samples using PGD and untargeted FGSM methods on 200 images extracted from the validation dataset. We vary the epsilon values from 0.1 to 0.8 and observe the effect on the total number of misclassifications by a model for a given dataset and attack.

4.5 Application of Defence Mechanism

This section primarily focuses on the implementation of defensive distillation as a defence mechanism to make the neural network architectures resilient from the adversarial samples generated using untargeted FGSM and PGD attacks.

4.5.1 Defensive Distillation implementation

In order to implement defensive distillation, we make use of the boiler plate code for knowledge distillation mechanism [51] as defensive distillation is originally derived from knowledge distillation, the only difference between the two is that the former uses the same architecture as the student and the teacher models and uses a temperature parameter. Listing A.2 shows the implementation of the Distillation class that helps in transferring the knowledge from the teacher model to the student model.

4.5.1.1 Distillation code walkthrough

The custom Distiller() class, overrides the Model methods train_step, test_step, and compile(). In order to use the distiller class, the following things are required:

- A trained teacher model
- A student model to train
- A student loss function on the difference between student predictions and ground-truth
- A distillation loss function, along with a temperature, on the difference between the soft student predictions and the soft teacher labels
- An alpha factor to weight the student and distillation loss
- An optimizer for the student and (optional) metrics to evaluate performance

The list below discusses the parameters requisiste for compiling the distiller -

- optimizer - refers to the keras optimizer for the student weights
- metrics - refers to the keras metrics for evaluation
- student_loss_fn - refers to the loss function of difference between student predictions and ground-truth
- distillation_loss_fn - refers to the loss function of difference between soft student predictions and soft teacher predictions
- alpha - refers to the weight to student_loss_fn and 1-alpha to distillation_loss_fn
- temperature - refers to the Temperature parameter used for softening probability distributions. Larger temperature gives softer distributions.

In the train_step method, a forward pass is performed on both the teacher and student models, loss is calculated with weighting of the student_loss and distillation_loss by alpha and 1-alpha, respectively, furthermore, the backward pass is executed. Only the student weights are updated, and therefore only the gradients for the student weights are calculated. In the test_step method, the student model is evaluated on the provided dataset.

4.5.1.2 Teacher model

Teacher model refers to the original model, which is trained and fixed, and helps in transferring the knowledge acquired during the training phase, to the corresponding student model with the help of knowledge distillation. For defensive distillation however, we add a lambda layer after the model_logits in order to soften the logits values using the temperature parameter.

4.5.1.3 Student model

After fixing the teacher model, the knowledge attained by the teacher model is imparted to the student model with the help of knowledge distillation. The Distiller assists in transferring the knowledge from the teacher to the student model. Listing 4.5 shows the code snippet for the distillation process where the knowledge from the teacher model is transferred to the student model using the requisite parameters.

```
distiller = Distiller(student=student, teacher=teacher)
distiller.compile(
    optimizer=keras.optimizers.Adam(),
    metrics=[keras.metrics.SparseCategoricalAccuracy()],
    student_loss_fn=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    distillation_loss_fn=keras.losses.KLDivergence(),
    alpha=0.1,
    temperature=10,
)

```

Listing 4.5: Distillation process

4.5.2 Applying Defensive Distillation on the architectures

In this section, we primarily discuss the process of applying defensive distillation on the same architectures as discussed previously. All the teacher architectures make use of a temperature parameter which is set to 200 for this experiment and alpha

4.5.2.1 VGG-16

We use the same VGG-16 architecture as discussed in section 4.3.1 for the student as well as the teacher model.

- **CIFAR-10** - This section shows the hyperparameters of the teacher and student VGG-16 architectures for CIFAR-10 dataset.

Table 4-7: Hyperparameters of teacher VGG-16 architecture trained on CIFAR-10 dataset.

Hyperparameter	**Value**
Learning rate	0.1
Learning rate decay	1e-6
Weight decay	0.0005
Momentum	0.9
Nesterov	True
Batch size	128
Total Number of Classes	10
Epochs	125
Temperature	200

Table 4-8: Hyperparameters of student VGG-16 architecture trained on CIFAR-10 dataset.

Hyperparameter	**Value**
Learning rate	0.1
Learning rate decay	1e-6
Weight decay	0.0005
Momentum	0.9
Nesterov	True
Batch size	128
Total Number of Classes	10
Epochs	150
Temperature	200
alpha	0.2

- **CIFAR-100** - This section shows the hyperparameters of the teacher and student VGG-16 architectures for CIFAR-100 dataset.

Table 4-9: Hyperparameters of teacher VGG-16 architecture trained on CIFAR-100 dataset.

Hyperparameter	**Value**
Learning rate	0.1
Learning rate decay	1e-6
Weight decay	0.0005
Momentum	0.9
Nesterov	True
Batch size	32
Total Number of Classes	100
Epochs	250
Temperature	200

Table 4-10: Hyperparameters of student VGG-16 architecture trained on CIFAR-100 dataset.

Hyperparameter	Value
Learning rate	0.1
Learning rate decay	1e-6
Weight decay	0.0005
Momentum	0.9
Nesterov	True
Batch size	32
Total Number of Classes	100
Epochs	200
Temperature	200
alpha	0.2

4.5.2.2 ResNet-50

We use the same ResNet-50 architecture as discussed in section 4.3.2 for the student as well as the teacher model.

- **CIFAR-10** - This section shows the hyperparameters of the teacher and student ResNet-50 architectures for CIFAR-10 dataset.

Table 4-11: Hyperparameters of teacher ResNet-50 architecture trained on CIFAR-10 dataset.

Hyperparameter	Value
Learning rate	0.001
beta_1	0.9
beta_2	0.999
amsgrad	False
epsilon	1e-07
Batch size	128
Total Number of Classes	10
Epochs	40
Temperature	200

Table 4-12: Hyperparameters of student ResNet-50 architecture trained on CIFAR-10 dataset.

Hyperparameter	**Value**
Learning rate	0.001
beta_1	0.9
beta_2	0.999
amsgrad	False
epsilon	1e-07
Batch size	128
Total Number of Classes	10
Epochs	40
Temperature	200
alpha	0.2

- **CIFAR-100** - This section shows the hyperparameters of the teacher and student ResNet-50 architectures for CIFAR-100 dataset.

Table 4-13: Hyperparameters of teacher ResNet-50 architecture trained on CIFAR-100 dataset.

Hyperparameter	**Value**
Learning rate	0.001
beta_1	0.9
beta_2	0.999
amsgrad	False
epsilon	1e-07
Batch size	64
Total Number of Classes	100
Epochs	50
Temperature	200

Table 4-14: Hyperparameters of student ResNet-50 architecture trained on CIFAR-100 dataset.

Hyperparameter	Value
Learning rate	0.001
beta_1	0.9
beta_2	0.999
amsgrad	False
epsilon	1e-07
Batch size	64
Total Number of Classes	100
Epochs	50
Temperature	200
alpha	0.2

4.5.2.3 Inception-V3

We use the same Inception-V3 architecture as discussed in section 4.3.3.

- **CIFAR-10** - This section shows the hyperparameters of the teacher and student Inception-V3 architectures for CIFAR-10.

Table 4-15: Hyperparameters of teacher Inception-V3 trained on CIFAR-10 dataset

Hyperparameter	Value
Learning rate	0.001
beta_1	0.9
beta_2	0.999
amsgrad	False
epsilon	1e-07
Batch size	128
Total Number of Classes	10
Epochs	50
Temperature	200

Table 4-16: Hyperparameters of student Inception-V3 architecture trained on CIFAR-10 dataset.

Hyperparameter	**Value**
Learning rate	0.001
beta_1	0.9
beta_2	0.999
amsgrad	False
epsilon	1e-07
Batch size	128
Total Number of Classes	10
Epochs	50
Temperature	200
alpha	0.2

- **CIFAR-100** - This section shows the hyperparameters of the teacher and student Inception-V3 architectures for CIFAR-100 dataset.

Table 4-17: Hyperparameters of teacher Inception-V3 architecture trained on CIFAR-100

Hyperparameter	**Value**
Learning rate	0.001
beta_1	0.9
beta_2	0.999
amsgrad	False
epsilon	1e-07
Batch size	64
Total Number of Classes	100
Epochs	50
Temperature	200

Table 4-18: Hyperparameters of student Inception-V3 architecture trained on CIFAR-100

Hyperparameter	**Value**
Learning rate	0.001
beta_1	0.9
beta_2	0.999
amsgrad	False
epsilon	1e-07
Batch size	64
Total Number of Classes	100
Epochs	50
Temperature	200
alpha	0.2

5 Experiment Results

This section primarily discusses the compilation of results obtained on training the models, applying adversarial attacks and adding a defence mechanism. We make use of the test dataset in section 4.2, to generate adversarial samples using PGD and FGSM attacks respectively,using varying epsilon values. Additionally, we make use of defensive distillation as a defence mechanism to make the model resilient from FGSM and PGD attacks and perform predictions on the generated adversarial samples. Test accuracy is calculated as - **(total number of correctly classified test images / total number of test images)**

5.1 VGG-16

5.1.1 CIFAR-10

- **Model Training**
  - **Training Accuracy** - 98.15%
  - **Validation Accuracy** - 89.83%
  - **Test Accuracy** - 87.00%

  Figures 5-1 and 5-2 show the accuracy and loss curves for VGG-16 model trained on CIFAR-10 dataset.

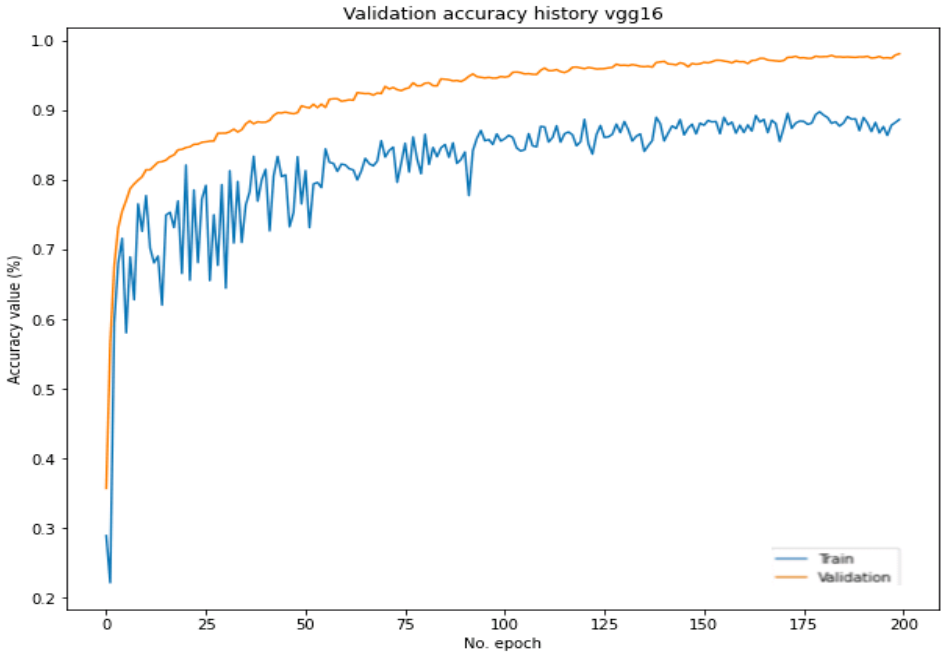


Figure 5-1: Accuracy Curve for VGG-16 model, trained on CIFAR-10 dataset

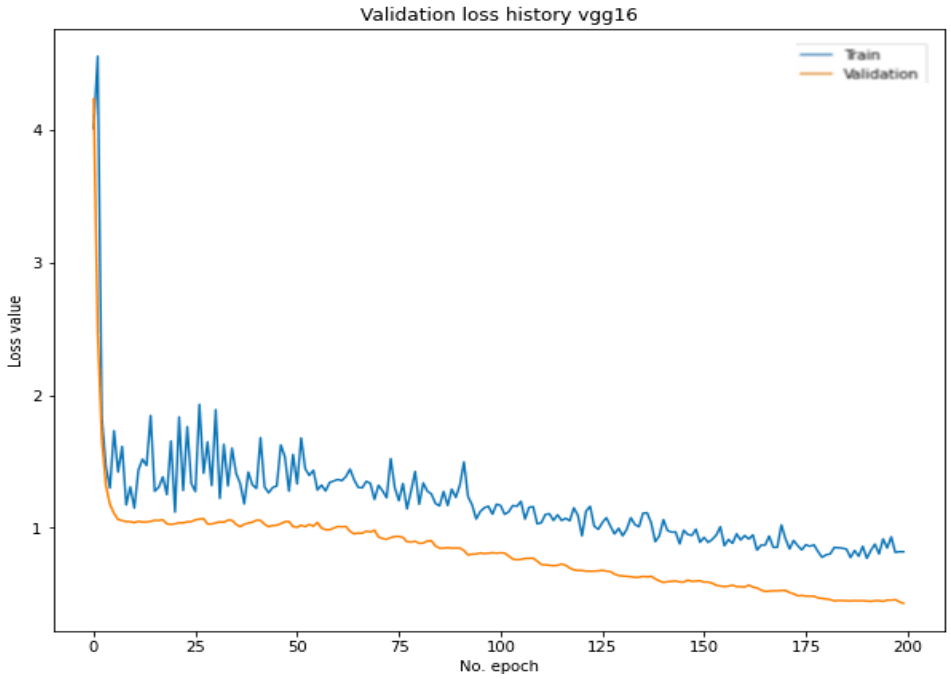


Figure 5-2: Loss Curve for VGG-16 model, trained on CIFAR-10 dataset

- **Applying FGSM and PGD attacks**

  Table 5-1 shows the accuracy of the VGG-16 model on adversarial CIFAR-10 samples generated using FGSM and PGD attacks respectively, with varying epsilon values. There is a constant drop in the test accuracy with an increase in epsilon value for both FGSM and PGD attacks. Additionally, the decrease in test accuracy stalls at the 0.4 epsilon mark for adversarial samples crafted using PGD attack, similar trend was observed by Amirreza Shaeiri, Rozhin Nobahari and Mohammad Hossein Rohban in their paper - ”Towards Deep Learning Models Resistant to Large Perturbations” [52].

  On an average there was a 9.25% drop in test accuracy for the VGG-16 model on adversarial CIFAR-10 samples generated using FGSM attack, with epsilon values ranging from 0.10 to 0.80. On the other hand, there was a 5.44% drop in test accuracy on adversarial CIFAR-10 samples generated using PGD attack, with epsilon values ranging from 0.10 to 0.80.

Table 5-1: Accuracy on FGSM samples and PGD samples using varying epsilon values

Epsilon value	Accuracy on FGSM samples	Accuracy on PGD samples
0.10	86.00%	87.00%
0.20	84.50%	83.50%
0.30	83.50%	82.00%
0.40	80.00%	80.00%
0.50	78.00%	80.00%
0.60	75.00%	80.00%
0.70	69.50%	80.00%
0.80	65.50%	80.00%

- **Applying Defensive Distillation**

  Table 5-2 shows the accuracy for the VGG-16 model on adversarial CIFAR-10 samples generated using FGSM and PGD attacks respectively, with varying epsilon values, after applying defensive distillation on the model. As observed previously, there is a constant drop in test accuracy with an increase in epsilon value for both FGSM and PGD attacks. Additionally, the decrease in test accuracy stalls at an epsilon value of 0.4 for adversarial samples crafted using PGD attack.

  On an average, there was a 8.56% drop in test accuracy for the VGG-16 model,protected via defensive distillation, on adversarial CIFAR-10 samples generated using FGSM attack, with epsilon values ranging from 0.10 to 0.80. On the other hand, there was a 6.00% drop in test accuracy on adversarial CIFAR-10 samples generated using PGD attack, with epsilon values ranging from 0.10 to 0.80. Moreover, defensive distillation lead to an increase in the model's accuracy by 0.69% against adversarial samples crafted using FGSM attack, as compared to the original model. On the flip side, it lead to a decrease in the model's accuracy by 0.56% against samples crafted using PGD attack.

  A key point to observe is that, after applying defensive distillation, the model had a better test accuracy for adversarial samples crafted using lower epsilon values (0.1 to 0.4), for both PGD and FGSM attacks.

Table 5-2: Accuracy on FGSM and PGD samples after applying Defensive Distillation

Epsilon value	Accuracy on FGSM samples	Accuracy on PGD samples
0.10	87.00%	87.00%
0.20	86.00%	86.00%
0.30	82.50%	82.50%
0.40	80.00%	78.50%
0.50	77.50%	78.50%
0.60	75.00%	78.50%
0.70	72.00%	78.50%
0.80	67.50%	78.50%

5.1.2 CIFAR-100

- **Model Training**
  - **Training Accuracy** - 84.96%
  - **Validation Accuracy** - 60.32%
  - **Test Accuracy** - 66.50%

Figures 5-3 and 5-4 show the accuracy and loss curves for VGG-16 model on CIFAR-100.

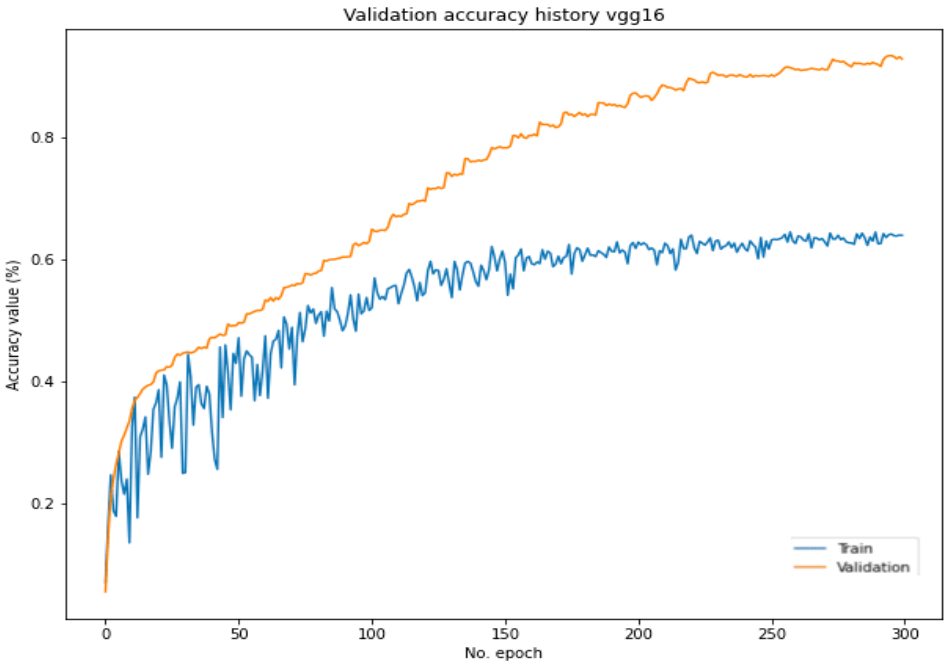


Figure 5-3: Accuracy Curve for VGG-16 model, trained on CIFAR-100 dataset

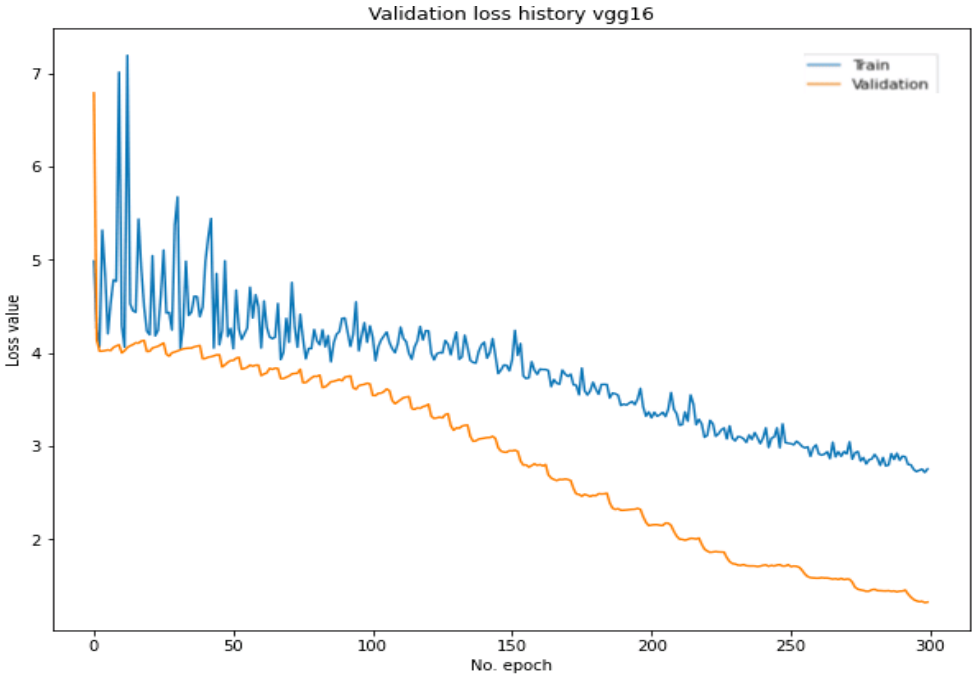


Figure 5-4: Loss Curve for VGG-16 model, trained on CIFAR-100 dataset

- **Applying FGSM and PGD attacks**

  Table 5-3 shows the accuracy of the VGG-16 model on adversarial CIFAR-100 samples generated using FGSM and PGD attacks respectively, with varying epsilon values. There is a constant drop in the test accuracy with an increase in epsilon value for both FGSM and PGD attacks,except for the epsilon value of 0.20. Additionally, the decrease in test accuracy stalls at the 0.4 epsilon mark for adversarial samples crafted using PGD attack.

  On an average, there was a 7.13% drop in test accuracy for the VGG-16 model on adversarial CIFAR-100 samples generated using FGSM attack, with epsilon values ranging from 0.10 to 0.80. On the other hand, there was a 2.94% drop in test accuracy on adversarial CIFAR-10 samples generated using PGD attack, with epsilon values ranging from 0.10 to 0.80.

Table 5-3: Accuracy on FGSM samples and PGD samples using varying epsilon values

Epsilon value	Accuracy on FGSM samples	Accuracy on PGD samples
0.10	64.50%	64.50%
0.20	66.00%	67.00%
0.30	65.00%	64.50%
0.40	64.00%	62.50%
0.50	60.00%	62.50%
0.60	55.50%	62.50%
0.70	51.00%	62.50%
0.80	49.00%	62.50%

- **Applying Defensive Distillation**

  Table 5-4 shows the accuracy for the VGG-16 model on adversarial CIFAR-100 samples generated using FGSM and PGD attacks respectively, with varying epsilon values, after applying defensive distillation on the model. As it can be observed, there is an increase in test accuracy for both FGSM samples and PGD samples, as compared to the original model. However, there is a constant drop in test accuracy with an increase in epsilon value for both FGSM and PGD attacks. Furthermore, the decrease in test accuracy stalls at an epsilon value of 0.4 for adversarial samples crafted using PGD attack.

  On an average, there was a 0.50% drop in test accuracy for the VGG-16 model,protected via defensive distillation, on adversarial CIFAR-100 samples generated using FGSM attack, with epsilon values ranging from 0.10 to 0.80. On the other hand, there was a 4.06% increase in test accuracy on adversarial CIFAR-100 samples generated using PGD attack, with epsilon values ranging from 0.10 to 0.80. Moreover, defensive distillation lead to an increase in the model's accuracy by 6.63% against adversarial samples crafted using FGSM attack, as compared to the original model. Additionally, it lead to an increase in the model's accuracy by 7.00%, against samples crafted using PGD attack.

  After applying defensive distillation, the model had an overall better test accuracy for adversarial samples crafted, for both PGD and FGSM attacks. Additionally, defensive distillation led to an exhorbitant increase in model accuracy for FGSM and PGD attacks, for lower epsilon values (0.1 to 0.4).

Table 5-4: Accuracy on FGSM and PGD samples after applying Defensive Distillation

Epsilon value	Accuracy on FGSM samples	Accuracy on PGD samples
0.10	76.00%	76.00%
0.20	73.50%	73.00%
0.30	70.00%	70.50%
0.40	69.00%	69.00%
0.50	67.00%	69.00%
0.60	63.50%	69.00%
0.70	57.00%	69.00%
0.80	54.50%	69.00%

5.2 ResNet-50

5.2.1 CIFAR-10

- **Model training**
  - **Training Accuracy** - 97.58%
  - **Validation Accuracy** - 95.03%
  - **Test Accuracy** - 95.00%

Figures 5-5 and 5-6 show the accuracy and loss curves for ResNet-50 model trained on CIFAR-10 dataset.

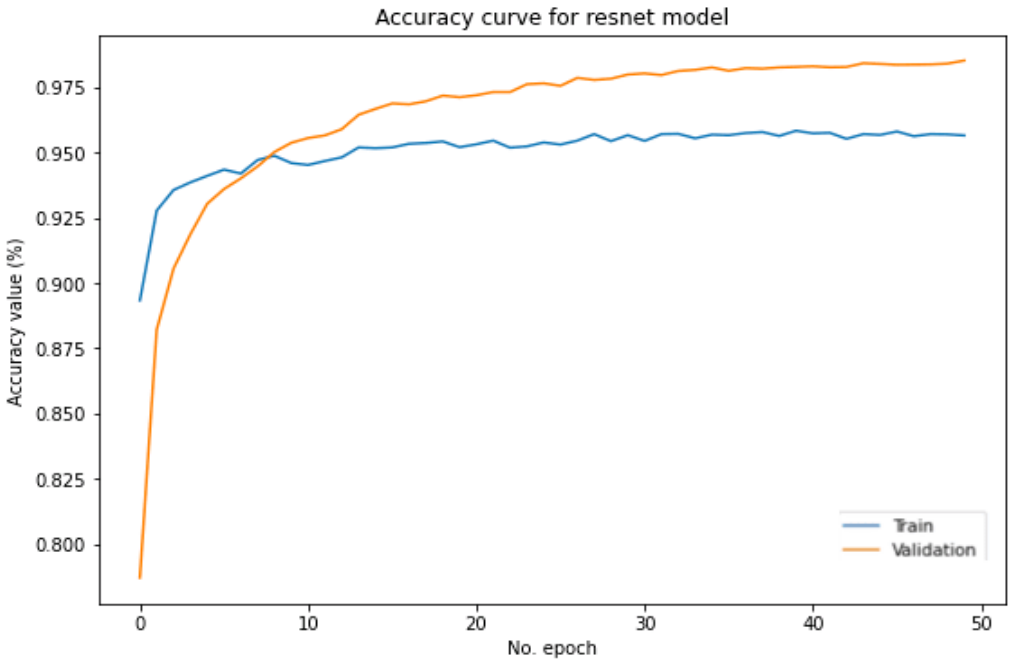


Figure 5-5: Accuracy Curve for ResNet-50 model, trained on CIFAR-10 dataset

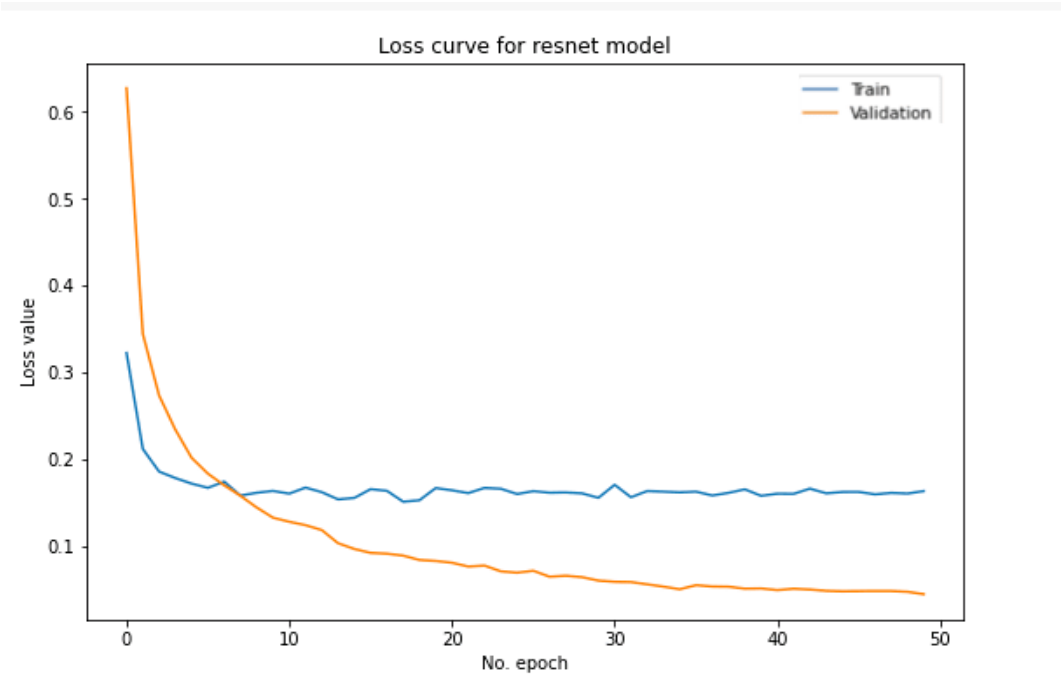


Figure 5-6: Loss Curve for ResNet-50 model, trained on CIFAR-10 dataset

- **Applying FGSM and PGD attacks**

  Table 5-5 shows the accuracy of the ResNet-50 model on adversarial CIFAR-10 samples generated using FGSM and PGD attacks respectively, with varying epsilon values. There is a constant drop in the test accuracy with an increase in epsilon value for both FGSM and PGD attacks. Additionally, the decrease in test accuracy stalls at the 0.4 epsilon mark for adversarial samples crafted using PGD attack.

  On an average there was a 26.63% drop in test accuracy for the ResNet-50 model on adversarial CIFAR-10 samples generated using FGSM attack. On the other hand, there was a 31.69% drop in test accuracy on adversarial CIFAR-10 samples generated using PGD attack.

Table 5-5: Accuracy on FGSM samples and PGD samples using varying epsilon values

Epsilon value	Accuracy on FGSM samples	Accuracy on PGD samples
0.10	91.50%	89.50%
0.20	83.50%	79.00%
0.30	78.00%	63.00%
0.40	69.50%	55.00%
0.50	63.50%	55.00%
0.60	60.50%	55.00%
0.70	51.50%	55.00%
0.80	49.00%	55.00%

- **Applying Defensive Distillation**

  Table 5-6 shows the accuracy for the ResNet-50 model on adversarial CIFAR-10 samples generated using FGSM and PGD attacks respectively, with varying epsilon values, after applying defensive distillation on the model. As observed previously, there is a constant drop in test accuracy with an increase in epsilon value for both FGSM and PGD attacks. Additionally, the decrease in test accuracy stalls at an epsilon value of 0.4 for adversarial samples crafted using PGD attack.

  On an average, there was a 21.31% drop in test accuracy for the ResNet-50 model, protected via defensive distillation, on adversarial CIFAR-10 samples generated using FGSM attack, with epsilon values ranging from 0.10 to 0.80. On the other hand, there was a 27.44% drop in test accuracy on adversarial CIFAR-10 samples generated using PGD attack. Moreover, defensive distillation lead to an increase in the model's accuracy by 5.32% against adversarial samples crafted using FGSM attack, as compared to the original model. Additionally, it lead to an increase in the model's accuracy by 4.25% against samples crafted using PGD attack.

Table 5-6: Accuracy on FGSM and PGD samples after applying Defensive Distillation

Epsilon value	Accuracy on FGSM samples	Accuracy on PGD samples
0.10	92.50%	90.50%
0.20	86.00%	79.50%
0.30	79.50%	68.00%
0.40	73.00%	60.50%
0.50	69.50%	60.50%
0.60	66.00%	60.50%
0.70	62.50%	60.50%
0.80	60.50%	60.50%

5.2.2 CIFAR-100

- **Model training**
  - **Training Accuracy** - 93.32%
  - **Validation Accuracy** - 80.01%
  - **Test Accuracy** - 85.00%

Figures 5-7 and 5-8 show the accuracy and loss curves for ResNet-50 model trained on CIFAR-100 dataset.

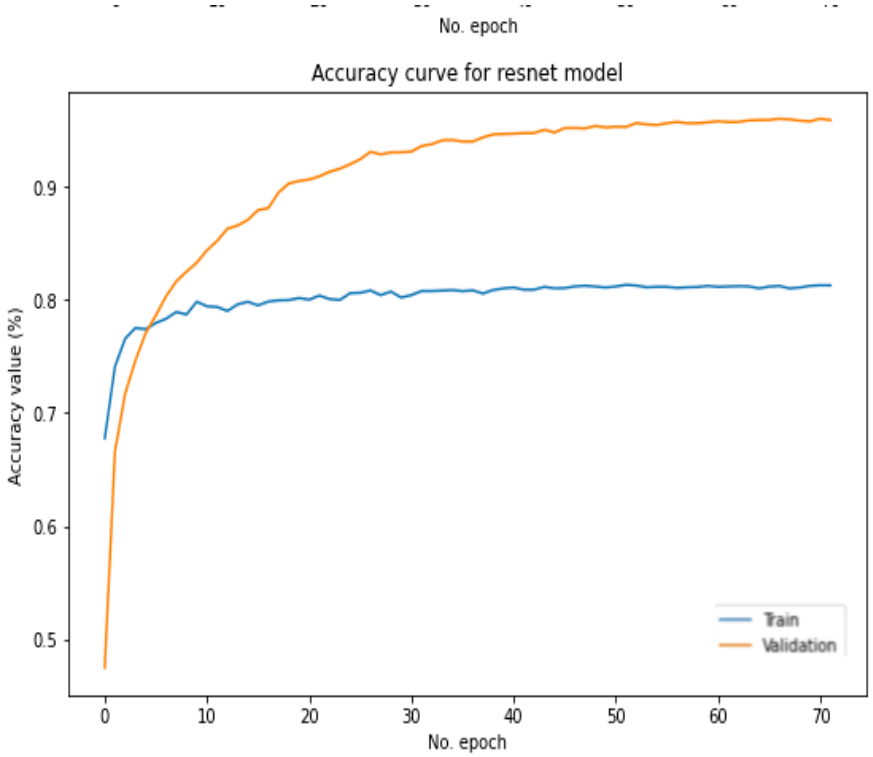


Figure 5-7: Accuracy Curve for ResNet-50 model, trained on CIFAR-100 dataset

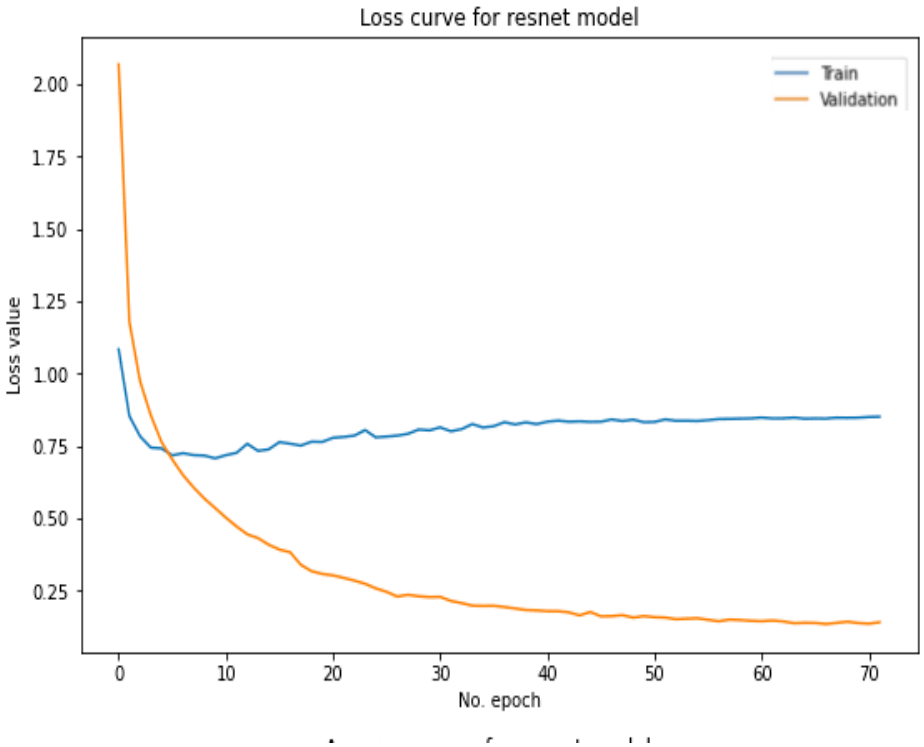


Figure 5-8: Loss Curve for ResNet-50 model, trained on CIFAR-100 dataset

- **Applying FGSM and PGD attacks**

  Table 5-7 shows the accuracy of the ResNet-50 model on adversarial CIFAR-100 samples generated using FGSM and PGD attacks respectively, with varying epsilon values. There is a constant drop in the test accuracy with an increase in epsilon value for both FGSM and PGD attacks. Additionally, the decrease in test accuracy stalls at the 0.4 epsilon mark for adversarial samples crafted using PGD attack.

  On an average there was a 37.44% drop in test accuracy for the ResNet-50 model on adversarial CIFAR-100 samples generated using FGSM attack. On the other hand, there was a 44.50% drop in test accuracy on adversarial CIFAR-100 samples generated using PGD attack.

Table 5-7: Accuracy on FGSM samples and PGD samples using varying epsilon values

Epsilon value	Accuracy on FGSM samples	Accuracy on PGD samples
0.10	77.00%	74.00%
0.20	63.50%	54.50%
0.30	53.00%	40.50%
0.40	46.00%	31.00%
0.50	40.00%	31.00%
0.60	37.50%	31.00%
0.70	33.50%	31.00%
0.80	30.00%	31.00%

- **Applying Defensive Distillation**

  Table 5-8 shows the accuracy for the ResNet-50 model on adversarial CIFAR-100 samples generated using FGSM and PGD attacks respectively, with varying epsilon values, after applying defensive distillation on the model. As observed previously, there is a constant drop in test accuracy with an increase in epsilon value for both FGSM and PGD attacks. Additionally, the decrease in test accuracy stalls at an epsilon value of 0.4 for adversarial samples crafted using PGD attack.

  On an average, there was a 30.50% drop in test accuracy for the ResNet-50 model, protected via defensive distillation, on adversarial CIFAR-100 samples generated using FGSM attack, with epsilon values ranging from 0.10 to 0.80. On the other hand, there was a 36.56% drop in test accuracy on adversarial CIFAR-100 samples generated using PGD attack. Moreover, defensive distillation lead to an increase in the model's accuracy by 6.89% against adversarial samples crafted using FGSM attack, as compared to the original model. Additionally, it lead to an increase in the model's accuracy by 7.94% against samples crafted using PGD attack.

Table 5-8: Accuracy on FGSM and PGD samples after applying Defensive Distillation

Epsilon value	Accuracy on FGSM samples	Accuracy on PGD samples
0.10	78.50%	76.00%
0.20	68.50%	57.00%
0.30	58.50%	47.00%
0.40	52.00%	41.50%
0.50	49.00%	41.50%
0.60	46.00%	41.50%
0.70	42.50%	41.50%
0.80	41.00%	41.50%

5.3 Inception V3

5.3.1 CIFAR-10

- **Model Training**
  - **Training Accuracy** - 92.17%
  - **Validation Accuracy** - 92.13%
  - **Test Accuracy** - 92.50%

Figures 5-9 and 5-10 show the accuracy and loss curves for Inception V3 model trained on CIFAR-10 dataset.

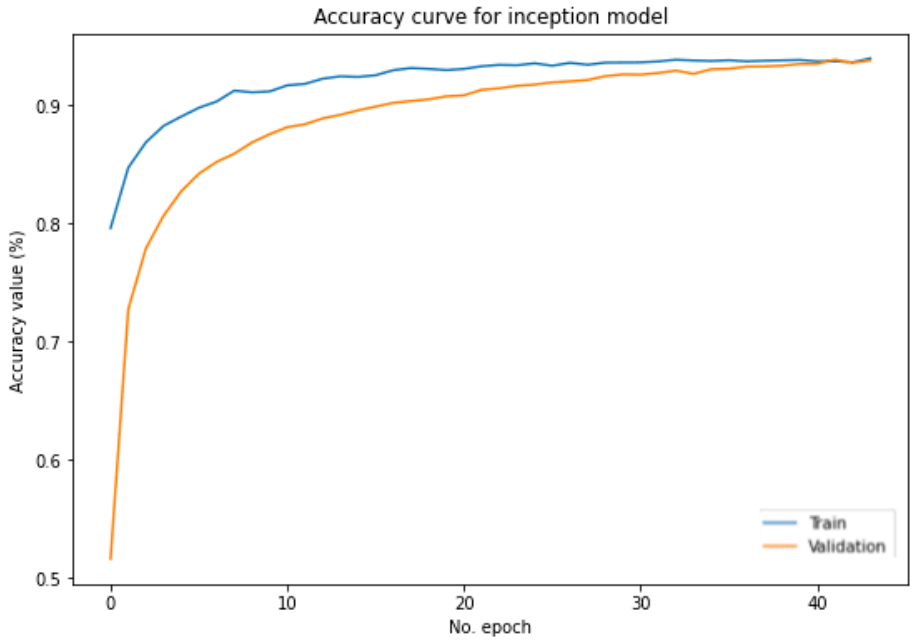


Figure 5-9: Accuracy Curve for Inception V3 model, trained on CIFAR-10 dataset

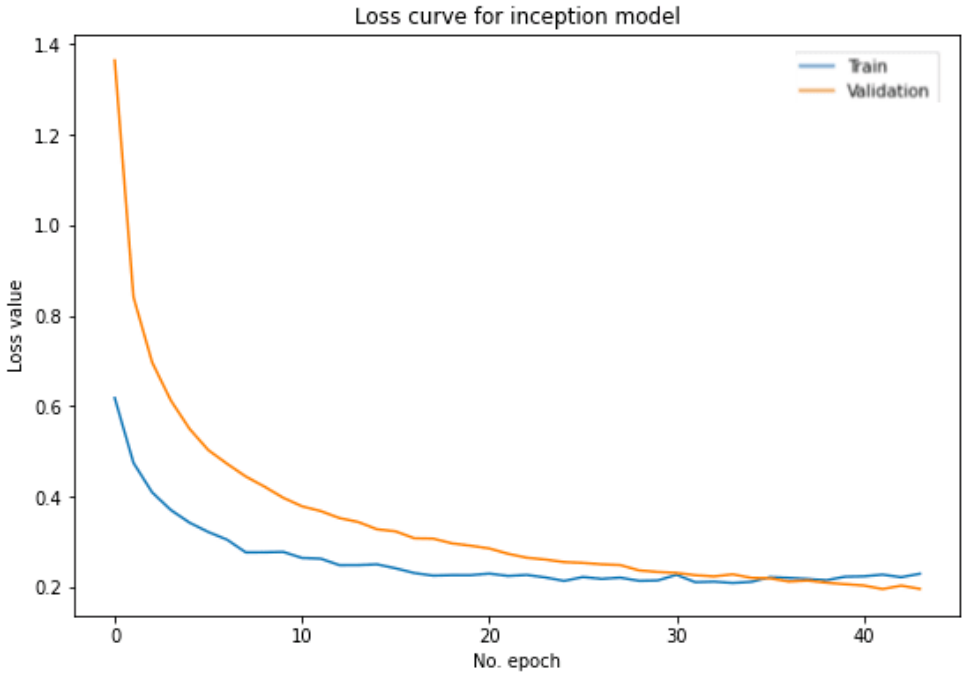


Figure 5-10: Loss Curve for Inception V3 model, trained on CIFAR-10 dataset

- **Applying FGSM and PGD attacks**

  Table 5-9 shows the accuracy of the Inception V3 model on adversarial CIFAR-10 samples generated using FGSM and PGD attacks respectively, with varying epsilon values. There is a constant drop in the test accuracy with an increase in epsilon value for both FGSM and PGD attacks. Additionally, the decrease in test accuracy stalls at the 0.4 epsilon mark for adversarial samples crafted using PGD attack.

  On an average there was a 29.06% drop in test accuracy for the Inception V3 model on adversarial CIFAR-10 samples generated using FGSM attack. On the other hand, there was a 42.75% drop in test accuracy on adversarial CIFAR-10 samples generated using PGD attack.

Table 5-9: Accuracy on FGSM samples and PGD samples using varying epsilon values

Epsilon value	Accuracy on FGSM samples	Accuracy on PGD samples
0.10	83.50%	78.00%
0.20	75.50%	61.00%
0.30	68.50%	46.50%
0.40	64.00%	42.50%
0.50	60.00%	42.50%
0.60	55.00%	42.50%
0.70	52.00%	42.50%
0.80	49.00%	42.50%

- **Applying Defensive Distillation**

  Table 5-10 shows the accuracy for the Inception V3 model on adversarial CIFAR-10 samples generated using FGSM and PGD attacks respectively, with varying epsilon values, after applying defensive distillation on the model. As observed previously, there is a constant drop in test accuracy with an increase in epsilon value for both FGSM and PGD attacks. Additionally, the decrease in test accuracy stalls at an epsilon value of 0.4 for adversarial samples crafted using PGD attack.

  On an average, there was a 18.00% drop in test accuracy for the Inception V3 model, protected via defensive distillation, on adversarial CIFAR-10 samples generated using FGSM attack. On the other hand, there was a 23.25% drop in test accuracy on adversarial CIFAR-10 samples generated using PGD attack. Moreover, defensive distillation lead to an increase in the model's accuracy by 11.06% against adversarial samples crafted using FGSM attack, as compared to the original model. Additionally, it lead to an increase in the model's accuracy by 19.50% against samples crafted using PGD attack.

Table 5-10: Accuracy on FGSM and PGD samples after applying Defensive Distillation

Epsilon value	Accuracy on FGSM samples	Accuracy on PGD samples
0.10	92.50%	91.00%
0.20	86.50%	79.50%
0.30	80.50%	68.50%
0.40	76.50%	63.00%
0.50	70.50%	63.00%
0.60	66.00%	63.00%
0.70	62.50%	63.00%
0.80	61.00%	63.00%

5.3.2 CIFAR-100

- **Model Training**
  - **Training Accuracy** - 92.82%
  - **Validation Accuracy** - 78.04%
  - **Test Accuracy** - 80.50%

Figures 5-11 and 5-12 show the accuracy and loss curves for Inception V3 model trained on CIFAR-100 dataset.

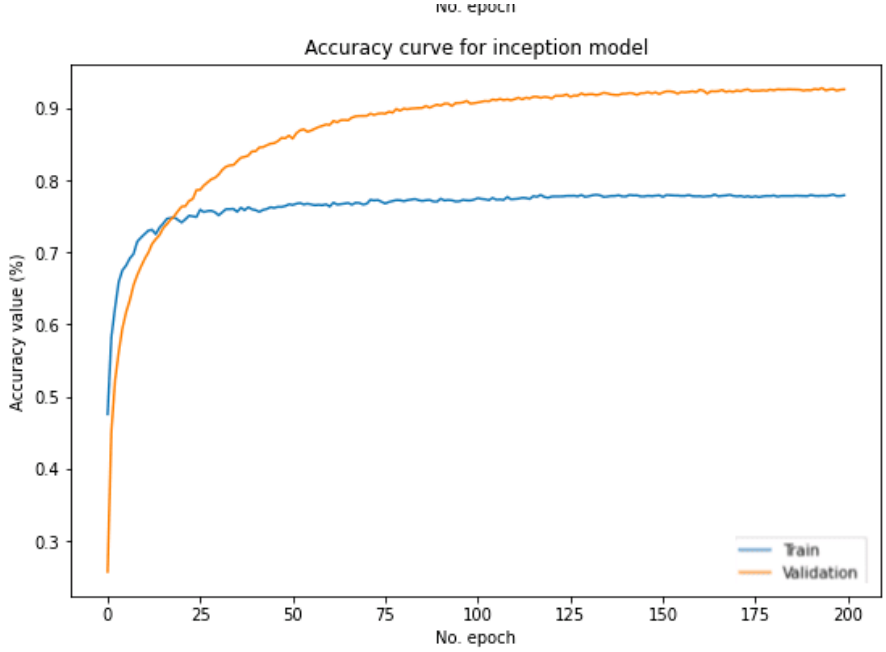


Figure 5-11: Accuracy Curve for Inception V3 model, trained on CIFAR-100 dataset

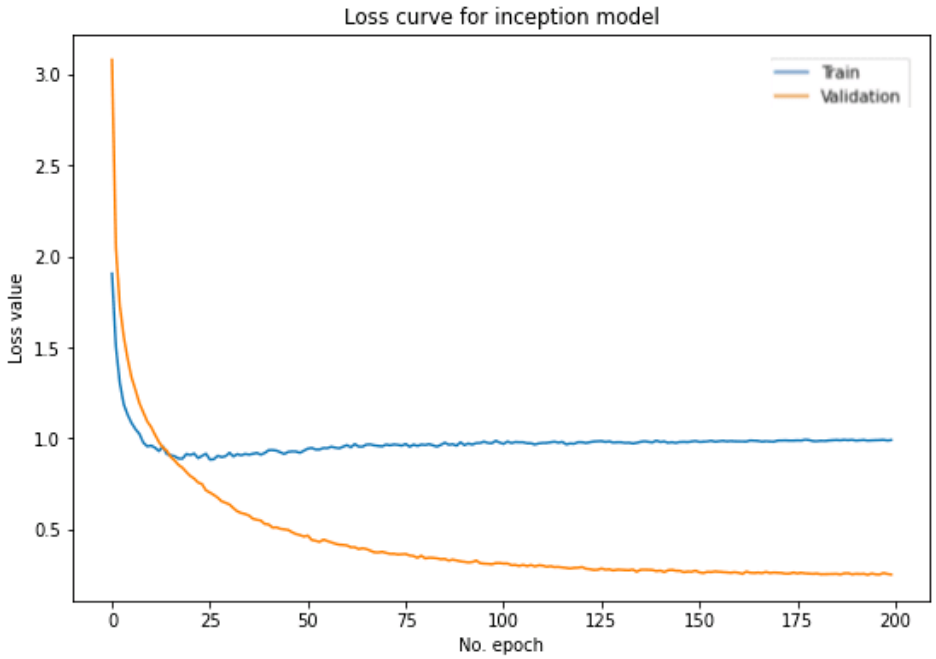


Figure 5-12: Loss Curve for Inception V3 model, trained on CIFAR-100 dataset

- **Applying FGSM and PGD attacks**

  Table 5-11 shows the accuracy of the Inception V3 model on adversarial CIFAR-100 samples generated using FGSM and PGD attacks respectively, with varying epsilon values. There is a constant drop in the test accuracy with an increase in epsilon value for both FGSM and PGD attacks. Additionally, the decrease in test accuracy stalls at the 0.4 epsilon mark for adversarial samples crafted using PGD attack.

  On an average there was a 33.25% drop in test accuracy for the Inception V3 model on adversarial CIFAR-100 samples generated using FGSM attack. On the other hand, there was a 44.25% drop in test accuracy on adversarial CIFAR-10 samples generated using PGD attack.

Table 5-11: Accuracy on FGSM samples and PGD samples using varying epsilon values

Epsilon value	Accuracy on FGSM samples	Accuracy on PGD samples
0.10	70.00%	66.00%
0.20	58.00%	46.50%
0.30	51.50%	35.00%
0.40	47.50%	28.50%
0.50	44.50%	28.50%
0.60	39.00%	28.50%
0.70	35.00%	28.50%
0.80	32.50%	28.50%

- **Applying Defensive Distillation**

  Table 5-12 shows the accuracy for the Inception V3 model on adversarial CIFAR-100 samples generated using FGSM and PGD attacks respectively, with varying epsilon values, after applying defensive distillation on the model. As observed previously, there is a constant drop in test accuracy with an increase in epsilon value for both FGSM and PGD attacks. Additionally, the decrease in test accuracy stalls at an epsilon value of 0.4 for adversarial samples crafted using PGD attack.

  On an average, there was a 27.56% drop in test accuracy for the Inception V3 model, protected via defensive distillation, on adversarial CIFAR-100 samples generated using FGSM attack. On the other hand, there was a 32.94% drop in test accuracy on adversarial CIFAR-100 samples generated using PGD attack. Moreover, defensive distillation lead to an increase in the model's accuracy by 5.69% against adversarial samples crafted using FGSM attack, as compared to the original model. Additionally, it lead to an increase in the model's accuracy by 11.31% against samples crafted using PGD attack.

Table 5-12: Accuracy on FGSM samples and PGD samples after applying Defensive Distillation

Epsilon value	Accuracy on FGSM samples	Accuracy on PGD samples
0.10	75.50%	74.00%
0.20	64.00%	56.00%
0.30	56.50%	45.50%
0.40	51.00%	41.00%
0.50	47.00%	41.00%
0.60	44.00%	41.00%
0.70	43.00%	41.00%
0.80	42.50%	41.00%

5.4 Key Observations

- Increase in epsilon values leads to a decrease in model accuracy.
- The model accuracy does not change for samples crafted using PGD attacks, beyond an epsilon value of 0.4. Similar trend was discussed in the paper -"Towards deep learning models resistant to large perturbations" [52].
- Models trained using defensive distillation exhibit higher accuracy in classifying FGSM and PGD samples with lower epsilon values (0.1 to 0.2).
- Models trained using transfer learning are more vulnerable to samples crafted using FGSM and PGD attacks, as observed in section 5.3 and 5.4 . A similar trend was reported by Biprodip Pal, Debashis Gupta, Md. Rashed-Al-Mahfuz, Salem A. Alyami and Mohammad Ali Moni, in their article - "Vulnerability in Deep Transfer Learning Models to Adversarial Fast Gradient Sign Attack for COVID-19 Prediction from Chest Radiography Images" [53].
- Defensive distillation had a greater impact in improving the performance of models based on transfer learning (ResNet-50 and Inception V3) than a traditional neural network model (VGG-16). But the downside of defensive distillation is that is requires additional training time, considering the fact that in this method, two models are getting trained (teacher and student). Moreover, there is a trade-off between training time and model accuracy when choosing defensive distillation as a defence mechanism.

6 Conclusion and Future Works

6.1 Conclusion

To reiterate, this project aims to understand the impact of adversarial attacks on neural network architectures, primarily in a white-box setting. Additionally, it aims to devise a robust defence mechanism, in order to enable the neural network architectures in mitigating adversarial attacks in a white-box setting.

The test bench for this experiment included three state-of-the-art neural network architectures namely - VGG-16, Resnet-50 and Inception V3. These models were trained on two benchmarking datasets - CIFAR-10 and CIFAR-100. The experiment had two main adversarial attack methodologies which were used to generate perturbated samples - Fast Gradient Sign Method(FGSM) and Projected Gradient Descent(PGD), implemented using Cleverhans library.

The experiment showed that adversarial samples crafted using PGD attack, having epsilon value beyond 0.4, had the same test accuracy, irrespective of the model. The experiment also showed that models trained using transfer learning were more vulnerable to PGD and FGSM attacks as compared to the traditional neural network models.

A defensive mechanism was implemented, in the form of Defensive Distillation, to make the neural network architectures more resilient to FGSM and PGD attacks. The experiment concluded that Defensive Distillation was effective in somewhat mitigating the risk of adversarial attacks (primarily PGD and FGSM attacks). However, Defensive Distillation had a greater impact in boosting the performance of models built using transfer learning (Inception V3 and ResNet-50).

6.2 Future Works

In this section, we primarily discuss the areas of research that can be explored as an extension to this project.

6.2.1 Additional White-Box Adversarial Attacks

Apart from using the two mainstream adversarial attacks - FGSM and PGD, other adversarial attacks like Jacobian based method [15] can be implemented to study the overall impact of adversarial attacks on the performance of a neural network architecture.

6.2.2 Testing On New Datasets

Alongside the standard benchmarking datasets like CIFAR-10 and CIFAR-100, other real-world datasets can be used to further testify the findings of this project. The only downside of using real-world datasets is the added debt of performing additional data pre-processing and augmentation steps.

6.2.3 Additional Neural Network Architectures

Having seen the effect of adverarial attacks on three primary state-of-the-art neural network architectures - VGG-16, Inception V3 and ResNet-50, similar experimentation can be performed on relatively less-complex architectures to observe the effect of such attacks as well as the degree of improvement brought about by defensive distillation.

6.2.4 Experimentation on the Temperature Parameter

In this project, we primarily set the temperature parameter to 200, furthermore, a similar experimentation can performed where defensive distillation is applied using varying values of the temperature parameter. This will help in determining the impact of the temperature parameter in the overall efficacy of defensive distillation.

References

[1] S. B. Kotsiantis, I. Zaharakis, P. Pintelas, *et al.*, "Supervised machine learning: A review of classification techniques," *Emerging artificial intelligence applications in computer engineering*, vol. 160, no. 1, pp. 3–24, 2007.

[2] (), [Online]. Available: https://hunch.net/~coms-4771/quinlan.pdf.

[3] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "A training algorithm for optimal margin classifiers," in *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, pp. 144–152.

[4] D. Berrar, "Bayes' theorem and naive bayes classifier," *Encyclopedia of Bioinformatics and Computational Biology: ABC of Bioinformatics*, vol. 403, 2018.

[5] R. Gentleman and V. J. Carey, "Unsupervised machine learning," in *Bioconductor case studies*, Springer, 2008, pp. 137–157.

[6] J. A. Hartigan and M. A. Wong, "Algorithm AS 136: A k-means clustering algorithm," *Journal of the royal statistical society. series c (applied statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[7] J. Dongre, G. L. Prajapati, and S. Tokekar, "The role of apriori algorithm for finding the association rules in data mining," in *2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, IEEE, 2014, pp. 657–660.

[8] Z. Zhang, "Introduction to machine learning: K-nearest neighbors," *Annals of translational medicine*, vol. 4, no. 11, 2016.

[9] R. S. Sutton, "Introduction: The challenge of reinforcement learning," in *Reinforcement Learning*, R. S. Sutton, Ed. Boston, MA: Springer US, 1992, pp. 1–3, ISBN: 978-1-4615-3618-5. DOI: 10.1007/978-1-4615-3618-5_1. [Online]. Available: https://doi.org/10.1007/978-1-4615-3618-5_1.

[10] B. Mehlig, "Machine learning with neural networks," Oct. 2021. DOI: 10.1017/9781108860604. [Online]. Available: http://dx.doi.org/10.1017/9781108860604.

[11] P. Sharma and A. Singh, "Era of deep neural networks: A review," in *2017 8th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 2017, pp. 1–5. DOI: 10.1109/ICCCNT.2017.8203938.

[12] A. Chakraborty, M. Alam, V. Dey, A. Chattopadhyay, and D. Mukhopadhyay, *Adversarial attacks and defences: A survey*, 2018. arXiv: 1810.00069 [cs.LG].

[13] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *2017 International Conference on Engineering and Technology (ICET)*, 2017, pp. 1–6. DOI: 10.1109/ICEngTechnol.2017.8308186.

[14] I. Evtimov, K. Eykholt, E. Fernandes, *et al.*, "Robust physical-world attacks on machine learning models," *CoRR*, vol. abs/1707.08945, 2017. arXiv: 1707.08945. [Online]. Available: http://arxiv.org/abs/1707.08945.

[15] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, *Distillation as a defense to adversarial perturbations against deep neural networks*, 2016. arXiv: 1511.04508 [cs.CR].

[16] I. J. Goodfellow, J. Shlens, and C. Szegedy, *Explaining and harnessing adversarial examples*, 2015. arXiv: 1412.6572 [stat.ML].

[17] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, *The limitations of deep learning in adversarial settings*, 2015. arXiv: 1511.07528 [cs.CR].

[18] A. Kurakin, I. J. Goodfellow, and S. Bengio, "Adversarial machine learning at scale," *CoRR*, vol. abs/1611.01236, 2016. arXiv: 1611.01236. [Online]. Available: http://arxiv.org/abs/1611.01236.

[19] L. Wang, C. Zhang, Z. Luo, *et al.*, "Progressive defense against adversarial attacks for deep learning as a service in internet of things," *CoRR*, vol. abs/2010.11143, 2020. arXiv: 2010.11143. [Online]. Available: https://arxiv.org/abs/2010.11143.

[20] N. Papernot, P. D. McDaniel, and I. J. Goodfellow, "Transferability in machine learning: From phenomena to black-box attacks using adversarial samples," *CoRR*, vol. abs/1605.07277, 2016. arXiv: 1605.07277. [Online]. Available: http://arxiv.org/abs/1605.07277.

[21] U. Shaham, Y. Yamada, and S. Negahban, "Understanding adversarial training: Increasing local stability of supervised models through robust optimization," *Neurocomputing*, vol. 307, pp. 195–204, Sep. 2018, ISSN: 0925-2312. DOI: 10.1016/j.neucom.2018.04.027. [Online]. Available: http://dx.doi.org/10.1016/j.neucom.2018.04.027.

[22] N. Papernot, P. D. McDaniel, I. J. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against deep learning systems using adversarial examples," *CoRR*, vol. abs/1602.02697, 2016. arXiv: 1602.02697. [Online]. Available: http://arxiv.org/abs/1602.02697.

[23] G. Hinton, O. Vinyals, and J. Dean, *Distilling the knowledge in a neural network*, 2015. arXiv: 1503.02531 [stat.ML].

[24] N. Carlini and D. A. Wagner, "Towards evaluating the robustness of neural networks," *CoRR*, vol. abs/1608.04644, 2016. arXiv: 1608.04644. [Online]. Available: http://arxiv.org/abs/1608.04644.

[25] W. Xu, D. Evans, and Y. Qi, "Feature squeezing: Detecting adversarial examples in deep neural networks," *CoRR*, vol. abs/1704.01155, 2017. arXiv: 1704.01155. [Online]. Available: http://arxiv.org/abs/1704.01155.

[26] H. Hosseini, Y. Chen, S. Kannan, B. Zhang, and R. Poovendran, *Blocking transferability of adversarial examples in black-box learning systems*, 2017. arXiv: 1703.04318 [cs.LG].

[27] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. DOI: 10.1109/5.726791.

[28] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25, Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[29] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," *CoRR*, vol. abs/1311.2901, 2013. arXiv: 1311.2901. [Online]. Available: http://arxiv.org/abs/1311.2901.

[30] C. Szegedy, W. Liu, Y. Jia, *et al.*, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014. arXiv: 1409.4842. [Online]. Available: http://arxiv.org/abs/1409.4842.

[31] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, 2015. arXiv: 1409.1556 [cs.CV].

[32] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. arXiv: 1512.03385. [Online]. Available: http://arxiv.org/abs/1512.03385.

[33] O. Russakovsky, J. Deng, H. Su, *et al.*, "Imagenet large scale visual recognition challenge," *CoRR*, vol. abs/1409.0575, 2014. arXiv: 1409.0575. [Online]. Available: http://arxiv.org/abs/1409.0575.

[34] X. Zhang, J. Zou, K. He, and J. Sun, "Accelerating very deep convolutional networks for classification and detection," *CoRR*, vol. abs/1505.06798, 2015. arXiv: 1505.06798. [Online]. Available: http://arxiv.org/abs/1505.06798.

[35] D. Wu, Y. Wang, S. Xia, J. Bailey, and X. Ma, "Skip connections matter: On the transferability of adversarial examples generated with resnets," *CoRR*, vol. abs/2002.05990, 2020. arXiv: 2002.05990. [Online]. Available: https://arxiv.org/abs/2002.05990.

[36] Q. Ji, J. Huang, W. He, and Y. Sun, "Optimized deep convolutional neural networks for identification of macular diseases from optical coherence tomography images," *Algorithms*, vol. 12, no. 3, 2019, ISSN: 1999-4893. DOI: 10.3390/a12030051. [Online]. Available: https://www.mdpi.com/1999-4893/12/3/51.

[37] H. Qassim, D. Feinzimer, and A. Verma, "Residual squeeze VGG16," *CoRR*, vol. abs/1705.03004, 2017. arXiv: 1705.03004. [Online]. Available: http://arxiv.org/abs/1705.03004.

[38] B. Wu, Z. Liu, Z. Yuan, G. Sun, and C. Wu, "Reducing overfitting in deep convolutional neural networks using redundancy regularizer," in *ICANN*, 2017.

[39] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," *CoRR*, vol. abs/1512.00567, 2015. arXiv: 1512.00567. [Online]. Available: http://arxiv.org/abs/1512.00567.

[40] A. Krizhevsky, V. Nair, and G. Hinton, "CIFAR-10 (canadian institute for advanced research)," [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html.

[41] ——, "CIFAR-100 (canadian institute for advanced research)," [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html.

[42] R. Awasthi. ""breaking deep learning with adversarial examples using tensorflow." Windows Phone Central, Ed. (), [Online]. Available: https://cv-tricks.com/how-to/breaking-deep-learning-with-adversarial-examples-using-tensorflow/.

[43] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge distillation: A survey," *CoRR*, vol. abs/2006.05525, 2020. arXiv: 2006.05525. [Online]. Available: https://arxiv.org/abs/2006.05525.

[44] (), [Online]. Available: https://research.google.com/colaboratory/local-runtimes.html.

[45] Z. Zhong, L. Zheng, G. Kang, S. Li, and Y. Yang, *Random erasing data augmentation*, 2017. arXiv: 1708.04896 [cs.CV].

[46] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. arXiv: 1502.03167. [Online]. Available: http://arxiv.org/abs/1502.03167.

[47] P. Zhou, J. Feng, C. Ma, C. Xiong, S. C. H. Hoi, and W. E, "Towards theoretically understanding why SGD generalizes better than ADAM in deep learning," *CoRR*, vol. abs/2010.05627, 2020. arXiv: 2010.05627. [Online]. Available: https://arxiv.org/abs/2010.05627.

[48] F. Zhuang, Z. Qi, K. Duan, *et al.*, "A comprehensive survey on transfer learning," *CoRR*, vol. abs/1911.02685, 2019. arXiv: 1911.02685. [Online]. Available: http://arxiv.org/abs/1911.02685.

[49] N. Papernot, F. Faghri, N. Carlini, *et al.*, "Technical report on the cleverhans v2.1.0 adversarial examples library," *arXiv preprint arXiv:1610.00768*, 2018.

[50] B. L. Sturm, "A simple method to determine if a music information retrieval system is a "horse"," *IEEE Transactions on Multimedia*, vol. 16, no. 6, pp. 1636–1644, 2014. DOI: 10.1109/TMM.2014.2330697.

[51] K. Borup. (), [Online]. Available: https://keras.io/examples/vision/knowledge_distillation/.

[52] A. Shaeiri, R. Nobahari, and M. H. Rohban, "Towards deep learning models resistant to large perturbations," *arXiv preprint arXiv:2003.13370*, 2020.

[53] B. Pal, D. Gupta, M. Rashed-Al-Mahfuz, S. A. Alyami, and M. A. Moni, "Vulnerability in deep transfer learning models to adversarial fast gradient sign attack for covid-19 prediction from chest radiography images," *Applied Sciences*, vol. 11, no. 9, p. 4233, 2021.

A Appendix Tables and Figures

```
def VGG16(num_classes , channel=3):
  model = Sequential()
  weight_decay = 0.0005
  learning_rate = 0.1
  lr_decay = 1e-6
  lr_drop = 20

  image_shape = (160, 160, 3)
  model.add(UpSampling2D((5,5)))

  model.add(Conv2D(64, (3, 3),
      padding='same',input_shape=image_shape,kernel_regularizer=regularizers.l2(weight_decay)))
  model.add(BatchNormalization())
  model.add(Activation('relu'))

  model.add(Conv2D(64, (3, 3),
      padding='same',kernel_regularizer=regularizers.l2(weight_decay)))
  model.add(BatchNormalization())
  model.add(Activation('relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Dropout(0.25))

  model.add(Conv2D(128, (3, 3),
      padding='same',kernel_regularizer=regularizers.l2(weight_decay)))
  model.add(BatchNormalization())
  model.add(Activation('relu'))

  model.add(Conv2D(128, (3, 3),
      padding='same',kernel_regularizer=regularizers.l2(weight_decay)))
  model.add(BatchNormalization())
  model.add(Activation('relu'))

  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Dropout(0.25))

  model.add(Conv2D(256, (3, 3),
      padding='same',kernel_regularizer=regularizers.l2(weight_decay)))
  model.add(BatchNormalization())
  model.add(Activation('relu'))

  model.add(Conv2D(256, (3, 3),
      padding='same',kernel_regularizer=regularizers.l2(weight_decay)))
  model.add(BatchNormalization())
  model.add(Activation('relu'))

  model.add(Conv2D(256, (3, 3),
      padding='same',kernel_regularizer=regularizers.l2(weight_decay)))
  model.add(BatchNormalization())
  model.add(Activation('relu'))

  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Dropout(0.25))

  model.add(Conv2D(512, (3, 3),
      padding='same',kernel_regularizer=regularizers.l2(weight_decay)))
  model.add(BatchNormalization())
  model.add(Activation('relu'))
```

```
50
51  model.add(Conv2D(512, (3, 3),
        padding='same',kernel_regularizer=regularizers.l2(weight_decay)))
52  model.add(BatchNormalization())
53  model.add(Activation('relu'))
54
55  model.add(Conv2D(512, (3, 3),
        padding='same',kernel_regularizer=regularizers.l2(weight_decay)))
56  model.add(BatchNormalization())
57  model.add(Activation('relu'))
58
59  model.add(MaxPooling2D(pool_size=(2, 2)))
60  model.add(Dropout(0.25))
61
62  model.add(Conv2D(512, (3, 3),
        padding='same',kernel_regularizer=regularizers.l2(weight_decay)))
63  model.add(BatchNormalization())
64  model.add(Activation('relu'))
65
66  model.add(Conv2D(512, (3, 3),
        padding='same',kernel_regularizer=regularizers.l2(weight_decay)))
67  model.add(BatchNormalization())
68  model.add(Activation('relu'))
69
70  model.add(Conv2D(512, (3, 3),
        padding='same',kernel_regularizer=regularizers.l2(weight_decay)))
71  model.add(BatchNormalization())
72  model.add(Activation('relu'))
73
74  model.add(MaxPooling2D(pool_size=(2, 2)))
75  model.add(Dropout(0.25))
76
77  model.add(Flatten())
78  model.add(Dense(512,kernel_regularizer=regularizers.l2(weight_decay)))
79  model.add(BatchNormalization())
80  model.add(Activation('relu'))
81
82  model.add(Dropout(0.25))
83  model.add(Dense(num_classes))
84  model.add(BatchNormalization())
85  model.add(Activation('softmax'))
86  sgd = keras.optimizers.SGD(lr=learning_rate, decay=lr_decay, momentum=0.9,
        nesterov=True)
87  model.compile(optimizer=sgd, loss=keras.losses.categorical_crossentropy,
        metrics=['accuracy'])
88
89  return model
```

Listing A.1: Function used to define VGG16 architecture

```python
class Distiller(keras.Model):
    def __init__(self, student, teacher):
        super(Distiller, self).__init__()
        self.teacher = teacher
        self.student = student

    def compile(
        self,
        optimizer,
        metrics,
        student_loss_fn,
        distillation_loss_fn,
        alpha=0.2,
        temperature=20,
    ):

        super(Distiller, self).compile(optimizer=optimizer, metrics=metrics)
        self.student_loss_fn = student_loss_fn
        self.distillation_loss_fn = distillation_loss_fn
        self.alpha = alpha
        self.temperature = temperature

    def train_step(self, data):
        # Unpack data
        x, y = data

        # Forward pass of teacher
        teacher_predictions = self.teacher(x, training=False)

        with tf.GradientTape() as tape:
            # Forward pass of student
            student_predictions = self.student(x, training=True)

            # Compute losses
            student_loss = self.student_loss_fn(y, student_predictions)
            distillation_loss = self.distillation_loss_fn(
                tf.nn.softmax(teacher_predictions / self.temperature, axis=1),
                tf.nn.softmax(student_predictions / self.temperature, axis=1),
            )
            loss = self.alpha * student_loss + (1 - self.alpha) * distillation_loss

        # Compute gradients
        trainable_vars = self.student.trainable_variables
        gradients = tape.gradient(loss, trainable_vars)

        # Update weights
        self.optimizer.apply_gradients(zip(gradients, trainable_vars))

        # Update the metrics configured in `compile()`.
        self.compiled_metrics.update_state(y, student_predictions)

        # Return a dict of performance
        results = {m.name: m.result() for m in self.metrics}
        results.update(
            {"student_loss": student_loss, "distillation_loss": distillation_loss}
        )
        return results

    def test_step(self, data):
        # Unpack the data
        x, y = data

        # Compute predictions
```

```
        y_prediction = self.student(x, training=False)

        # Calculate the loss
        student_loss = self.student_loss_fn(y, y_prediction)

        # Update the metrics.
        self.compiled_metrics.update_state(y, y_prediction)

        # Return a dict of performance
        results = {m.name: m.result() for m in self.metrics}
        results.update({"student_loss": student_loss})
        return results

    def call(self, inputs, *args, **kwargs):
        return self.student(inputs)
```

Listing A.2: Distillation process implementation

B Appendix Source of Information

B.1 Github Repositories

The following repositories were used for the experiment -

- **CleverHans** - Library used in benchmarking a Machine Learning system's vulnerability by planting adversarial attacks.
- **Knowledge Distillation** - Link to the boiler-plate source code for knowledge distillation implementation using keras.
- **Project Source Code** - Link to the complete source code for this project