Enhancing Neural Network Robustness using Hybrid Adversarial Training

Rajarshi Nandi
201791225

Supervised by Luisa Cutillo & Mike Croucher

Submitted in accordance with the requirements for the
module MATH5872M: Dissertation in Data Science and Analitics
as part of the degree of

Master of Science in Data Science and Analystics

The University of Leeds, School of Mathematics

September 2024

The candidate confirms that the work submitted is his/her own and that appropriate credit has been given where reference has been made to the work of others.

UNIVERSITY OF LEEDS

School of Mathematics

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

Academic integrity statement

I am aware that the University defines plagiarism as presenting someone else's work, in whole or in part, as your own. Work means any intellectual output, and typically includes text, data, images, sound or performance.

I promise that in the attached submission I have not presented anyone else's work, in whole or in part, as my own and I have not colluded with others in the preparation of this work. Where I have taken advantage of the work of others, I have given full acknowledgement. I have not resubmitted my own work or part thereof without specific written permission to do so from the University staff concerned when any of this work has been or is being submitted for marks or credits even if in a different module or for a different qualification or completed prior to entry to the University. I have read and understood the University's published rules on plagiarism and also any more detailed rules specified at School or module level. I know that if I commit plagiarism I can be expelled from the University and that it is my responsibility to be aware of the University's regulations on plagiarism and their importance.

I re-confirm my consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to monitor breaches of regulations, to verify whether my work contains plagiarised material, and for quality assurance purposes. I confirm that I have declared all mitigating circumstances that may be relevant to the assessment of this piece of work and that I wish to have taken into account. I am aware of the University's policy on mitigation and the School's procedures for the submission of statements and evidence of mitigation. I am aware of the penalties imposed for the late submission of coursework.

Name _____

Student ID 201791225

Abstract

This dissertation explores a different approach to improving the robustness of neural networks against adversarial attacks, specifically focusing on the Fast Gradient Sign Method (FGSM). The research was conducted on a High Performance Computing Cluster at the University of Leeds using MATLAB. Three experiments were carried out, each building upon the insights gained from the previous ones. In the first experiment, a ResNet-18 architecture was trained from scratch on CIFAR-10 data using normal, unperturbed images. The second experiment trained the same architecture using only adversarial data generated with respect to itself. Finally, the third experiment balanced the training, using half normal images training and half adversarial training. The results show that the hybrid training approach of the third experiment was able to balance validation accuracy on normal data with improved robustness against adversarial attacks. This technique of mixing normal and adversarial data during training represents a key contribution of this work. In addition to measuring performance on normal and adversarial validation sets, the study also employed MATLAB's GradCAM tool to gain deeper insights into the decision-making process of the trained models. This multimodal analysis provides a comprehensive evaluation of the proposed approaches for enhancing neural network robustness.

Acknowledgments

I would like to express my deepest gratitude to my supervisors, Luisa Cutillo and Mike Croucher, whose invaluable guidance, insightful feedback, and unwavering support have been essential to the completion of this dissertation. Your expertise and encouragement throughout the research process have significantly shaped the direction and quality of my work. Thank you for your continued support.

This work was undertaken on ARC4, part of the High Performance Computing facilities at the University of Leeds, UK.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Neural networks, especially deep neural networks (DNNs), have transformed areas like image recognition, natural language processing, and autonomous driving due to their capacity to learn from large datasets. These models consist of layers of interconnected neurons, where each connection has a weight. During training, these weights are adjusted using optimization algorithms such as stochastic gradient descent to minimize prediction errors (LeCun et al., 2015). Convolutional neural networks (CNNs) excel in image classification tasks (Krizhevsky et al., 2012), while recurrent neural networks (RNNs) perform well in sequence data processing like language translation (Sutskever et al., 2014).

Despite their success, neural networks are often fragile and susceptible to small perturbations in input data. This lack of robustness is particularly concerning in critical applications such as healthcare, communication, and autonomous vehicles. For instance, a self-driving car must accurately interpret its environment despite variations in lighting or weather. Any misinterpretation could lead to catastrophic outcomes (Eykholt et al., 2018). Therefore, ensuring neural networks perform reliably under such perturbations is crucial for their safe deployment.

However it was observed that adversarial robustness may come at the cost of reduced accuracy on clean, unperturbed data (Tsipras D. et al., 2019). This dissertation aims to develop three image classification models using three different training procedures and analyze the performance of the models on normal and perturbed data.

1.1 Objective

The main aim of this dissertation is to train a convolutional neural network with residual (skip) connections based on ResNet-18 architecture on CIFAR 10 dataset using three techniques:

- 100% training on normal data to benchmark the network,
- 100% training on adversarial data to analyze the effect of adversarial training on the accuracy of normal data,

- 50% training on normal data & 50% training on adversarial data to balance the drop in accuracy of normal data with the improvement in accuracy of adversarial data.

The three models are then tested against both normal data and progressively stronger adversarial data. Here adversarial data is referred to any set of images which have been attacked using Fast Sign Gradient Method (FGSM) which has been explained later.

1.2 Overview

This introductory chapter demonstrates the primary objective of this dissertation and provides an overview of the work. Chapter 2 delves into the literature review which covers various types of adversarial attacks along with an effective defenses against the most common type of white box attack (Fast Gradient Sign Method) i.e. adversarial training. It also provides two real world example where neural network's vulnerability towards adversarial attacks can have serious consequences. Chapter 3 describes the data used, model architecture, experiment setup and training & testing methods used to perform the experiments. Chapter 4 focuses on the analysis of the experiment results along with the explainability of the model's decision making. Chapter 5 concludes the entire dissertation along with possible future works.

Chapter 2

Literature Review

This chapter provides an overview of adversarial examples along with two most widely used attack methodologies used in this dissertation and their mathematical foundation. A very effective defense against adversarial example has also been discussed in this chapter. Two real world examples of adversarial attack has also been discussed in this section to highlight the significance of robust neural network models.

2.1 What are Adversarial Examples?

Adversarial examples are inputs to neural networks that have been intentionally perturbed to cause the network to make a mistake (Szegedy et al., 2014). These perturbations are often imperceptible to humans but can significantly alter the network's output. For example, a slight noise addition to an image of an airplane can cause a ResNet-18 architecture trained on 100% normal data to misclassify it as a ship while both the images are indistinguishable to naked human eye as evident from Figure 2.1 below. Understanding and addressing these vulnerabilities is essential for developing robust AI systems.

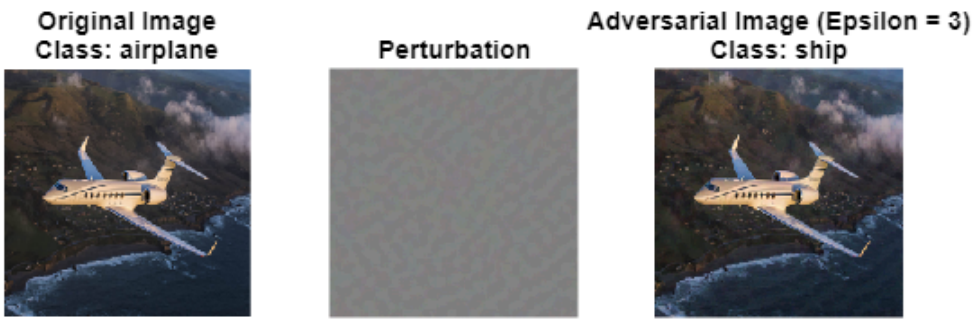

Figure 2.1: ResNet-18 trained on 100% normal data misclassify an airplane for a ship when a perturbation of 3 (pixel range: 0-255) is introduced in the image

The concept of adversarial examples was first highlighted by Szegedy et al. (2014). It was found that adding imperceptible perturbations to an image could cause a state-of-the-art neural network to misclassify it. This discovery led to extensive research into understanding and mitigating these vulnerabilities. Goodfellow et al., (2015) introduced the Fast Gradient Sign Method (FGSM), a simple yet powerful technique to generate adversarial examples efficiently, sparking further exploration into neural network weaknesses.

2.2 Mathematical Foundation of Adversarial Attacks

The fundamental principle of adversarial attacks involves finding a minimal perturbation to an input that induces misclassification by a neural network. This can be formulated as an optimization problem. Given an input x with true label y, the objective is to find a perturbation δ such that the perturbed input $x' = x + \delta$ is misclassified by the model f. Mathematically, this can be expressed as:

$$\begin{aligned} &\underset{\delta}{\text{minimize}} \quad \|\delta\|_p \\ &\text{subject to} \quad f(x+\delta) \neq y \end{aligned} \tag{2.1}$$

Here, $\|\delta\|_p$ denotes the p-norm of δ, which is minimized to ensure the perturbation remains imperceptible. The constraint ensures misclassification of the perturbed input.

For targeted attacks, where the aim is to induce the model to predict a specific incorrect label t, the formulation becomes:

$$\begin{aligned} &\underset{\delta}{\text{minimize}} \quad \|\delta\|_p \\ &\text{subject to} \quad f(x+\delta) = t \end{aligned} \tag{2.2}$$

2.3 Types of Adversarial Attacks

There are three main types of adversarial attacks: white-box attacks, black-box attacks, and gray-box attacks. In white-box attacks, the adversary has complete knowledge of the target model, including its architecture, parameters, and training data, allowing precise crafting of adversarial examples (Liang, H. et al., 2022). In black-box attacks, the adversary has no knowledge of the target model's internals but can query the model and observe outputs, making the attack more challenging yet still effective. Black-box attacks often rely on transferability, where adversarial examples generated for one model are used against another (Papernot, N. et al., 2017). Gray-box attacks assume partial knowledge of the target model, such as the architecture but not the exact parameters (Papernot, N. et al., 2016).

2.4 Common Adversarial White-Box Attacks

This dissertation focuses specifically on adversarial white-box attack and defense against it. The FGSM and PGD attack presented below are the two most widely implemented attack techniques utilized in this dissertation to test the robustness of neural network to adversarial examples.

2.4.1 Fast Gradient Sign Method (FGSM)

The Fast Gradient Sign Method (FGSM), introduced by Goodfellow et al., (2015), is defined by the equation:

$$x' = x + \epsilon \cdot \text{sign}(\nabla_x J(x, y)) \tag{2.3}$$

In this formulation, $x \in \mathbb{R}^n$ represents the original input, while $x' \in \mathbb{R}^n$ is the resulting adversarial example. The true label of x is denoted by $y \in \{1, ..., K\}$, where K is the number of classes. The scalar $\epsilon \in \mathbb{R}^+$ controls the magnitude of the perturbation. $J : \mathbb{R}^n \times \{1, ..., K\} \to \mathbb{R}$ is the loss function used to train the neural network, and $\nabla_x J(x, y) \in \mathbb{R}^n$ is its gradient with respect to x. Finally, sign : $\mathbb{R}^n \to \{-1, 1\}^n$ is the element-wise sign function. FGSM operates by perturbing the input in the direction that maximizes the loss, constrained by the L_∞-norm bound ϵ.

One of the key advantages of FGSM is its computational efficiency. Since it requires only a single forward and backward pass through the network, it is relatively fast to compute compared to more complex adversarial attack methods. This makes FGSM suitable for scenarios where quick evaluations or large-scale adversarial testing are needed. Additionally, FGSM can effectively generate adversarial examples with minimal computational overhead, making it a practical choice for initial vulnerability assessments of models.

However, FGSM has notable limitations. As a single-step attack, it may fail to find the optimal adversarial example, especially when the model has been adversarially trained or when more sophisticated defenses are in place. This can lead to suboptimal adversarial perturbations that do not fully exploit the model's weaknesses.

2.4.2 Projected Gradient Descent (PGD)

Projected Gradient Descent (PGD), proposed by Madry, A. et al., (2018), is an iterative variant of FGSM defined by the equation:

$$x^{(t+1)} = \Pi_S(x^{(t)} + \alpha \cdot \text{sign}(\nabla_x J(x^{(t)}, y))) \tag{2.4}$$

Here, $x^{(t)} \in \mathbb{R}^n$ is the perturbed input at iteration t, and $x^{(t+1)} \in \mathbb{R}^n$ is the perturbed input at the next iteration. As in FGSM, $y \in \{1, ..., K\}$ is the true label of the original input. The step size for each iteration is controlled by $\alpha \in \mathbb{R}^+$. The loss function J and its gradient $\nabla_x J(x^{(t)}, y)$ are defined as in FGSM. $S \subset \mathbb{R}^n$ represents the set of allowed perturbations, and

$\Pi_S : \mathbb{R}^n \rightarrow S$ is the projection operator onto this set. PGD works by iteratively applying FGSM-like updates, projecting the result back onto the set of allowed perturbations S after each step. Typically, S is defined as $\{x' : \|x' - x\|_\infty \leq \epsilon\}$, where x is the original input and ϵ is the maximum allowed perturbation.

The iterative nature of PGD allows it to generate stronger and more precise adversarial examples than FGSM, making it one of the most powerful first-order attack methods. By repeatedly adjusting the input and refining the perturbation through multiple iterations, PGD is capable of exploiting vulnerabilities in models that single-step methods, like FGSM, may overlook. The projection step ensures that the adversarial example remains within the allowed perturbation boundaries, making PGD highly effective against models trained with defenses such as adversarial training.

However, the increased complexity of PGD comes at the cost of computational efficiency. Since it requires multiple forward and backward passes through the network, it is significantly more computationally expensive than FGSM. This makes PGD less suitable for real-time or large-scale adversarial testing. Additionally, while PGD generates stronger adversarial examples, the iterative process can lead to more noticeable perturbations, particularly at higher ϵ values, which may make the adversarial examples detectable by humans.

While FGSM attack is simple and fast, PGD is more complex and powerful. Each has its strengths and weaknesses, and the choice of which to use depends on the specific requirements of the task at hand, including computational resources, desired attack strength, and the nature of the target model.

2.5 Defenses against Adversarial Attacks

While various defense techniques against adversarial white-box attacks exist, such as DeepDefense (Yan Z. et al., 2018), TRADES (Zhang H. et al., 2019), and JPEG Compression (Dziugaite G. et al., 2016), Projected Gradient Descent (PGD) adversarial training (Tramèr F et al., 2018) demonstrates exceptional effectiveness. PGD-AT consistently exhibits enhanced robustness across diverse perturbation budgets and attack strengths (ϵ). Its training on worst-case adversarial examples confers a generalized resilience, outperforming other methods in maintaining effectiveness across varying attack parameters and norms (Y. Dong et al., 2020).

While PGD-AT is computationally expensive and it was also found by Wong E. et al., (2020) that FGSM-AT can significantly improve the robustness of models towards much powerful adversarial attack when the initialization of the adversarial noise to be added to adversarial training data is set to ”random” instead of ”zero”. Hence, this dissertation focuses on training respective models on FGSM with initialization set to ”random”.

2.5.1 FGSM Adversarial Training

Fast Gradient Sign Method (FGSM) adversarial training is a technique used to enhance the robustness of machine learning models against adversarial attacks. Developed by Goodfellow et al., (2015), this approach involves augmenting the training data with adversarial examples generated using the FGSM algorithm. By exposing the model to these perturbed inputs during training, it learns to maintain accurate predictions even in the presence of small, intentionally crafted disturbances (Madry, A. et al., (2018)).

Adversarial training using the Fast Gradient Sign Method (FGSM) offers a key advantage in terms of computational efficiency. FGSM requires only a single forward and backward pass through the neural network to generate adversarial examples, making it significantly faster than iterative methods like Projected Gradient Descent (PGD) (Goodfellow et al., , 2015). This efficiency makes FGSM particularly suitable for large-scale applications or scenarios where computational resources are limited. Additionally, FGSM is easy to implement, requiring minimal changes to standard training pipelines.

However, this method may lead to overfitting to specific perturbation types and potentially reduce model performance on clean data. (Tsipras D. et al., 2019) highlighted that adversarial robustness may come at the cost of reduced accuracy on clean, unperturbed data. The trade-off exists because robust models, trained to resist adversarial attacks, learn fundamentally different feature representations compared to standard models, which can cause underperformance in natural settings. In an attempt to mitigate this trade-off, this dissertation aims to strike a balance by training the model on both clean and adversarial data, seeking to maintain competitive performance on both types of inputs.

2.6 Significance of Robust Neural Networks

Here adversarial attacks have been demonstrated in real-world scenarios, including medical imaging systems and image segmentation models utilized in self-driving cars, highlighting vulnerabilities in critical AI applications.

2.6.1 Medical Imaging System

Simple adversarial attacks like the Fast Gradient Sign Method (FGSM) can expose severe vulnerabilities in advanced medical image segmentation models. The authors demonstrate that even state-of-the-art U-Net variants trained on MRI datasets can be easily fooled by subtle perturbations to input images, causing dramatic changes in the predictions. This susceptibility to attack could have severe consequences in clinical settings, where doctors rely on these AI systems to make critical decisions. For instance, a maliciously altered tumor segmentation could lead to misdiagnosis, improper treatment planning, or even fatal outcomes in case of false negatives i.e. if portions of tumors are overlooked (Wang et al., 2024).

The ease with which these sophisticated models can be compromised raises serious concerns about the security and reliability of AI-assisted medical imaging systems. As the authors note, most clinicians are not trained to detect such subtle adversarial manipulations, making this vulnerability particularly dangerous. Moreover, the finding that medical imaging models may be more susceptible to these attacks than other types of AI systems (Ma et al., 2021) further underscores the urgent need for robust defense mechanisms and rigorous security protocols before deploying these technologies in real-world healthcare environments.

2.6.2 Vision Models in Self Driving Vehicles

Adversarial attacks pose a significant threat to the safety and reliability of self-driving vehicles that rely on semantic segmentation models for visual perception. As demonstrated in Figure 2.2 (MathWorks.com et al., 2024) using a Deeplab v3+ network trained on the CamVid data set with weights initialized from a pretrained ResNet-18 network, imperceptible perturbations to input images can cause dramatic misclassifications of critical objects like pedestrians, cyclists, and road features. In this specific case shown, an epsilon value of just 5 in a basic iterative method attack was sufficient to fool the model into completely misclassifying a bicyclist as road, building, and pavement - a potentially catastrophic error if translated to a real-world driving scenario.

Even state-of-the-art architectures like Deeplab v3+ are susceptible, highlighting the pressing need for more robust approaches. In autonomous driving applications, where split-second visual decisions can mean the difference between safety and disaster, defending against such attacks is crucial.

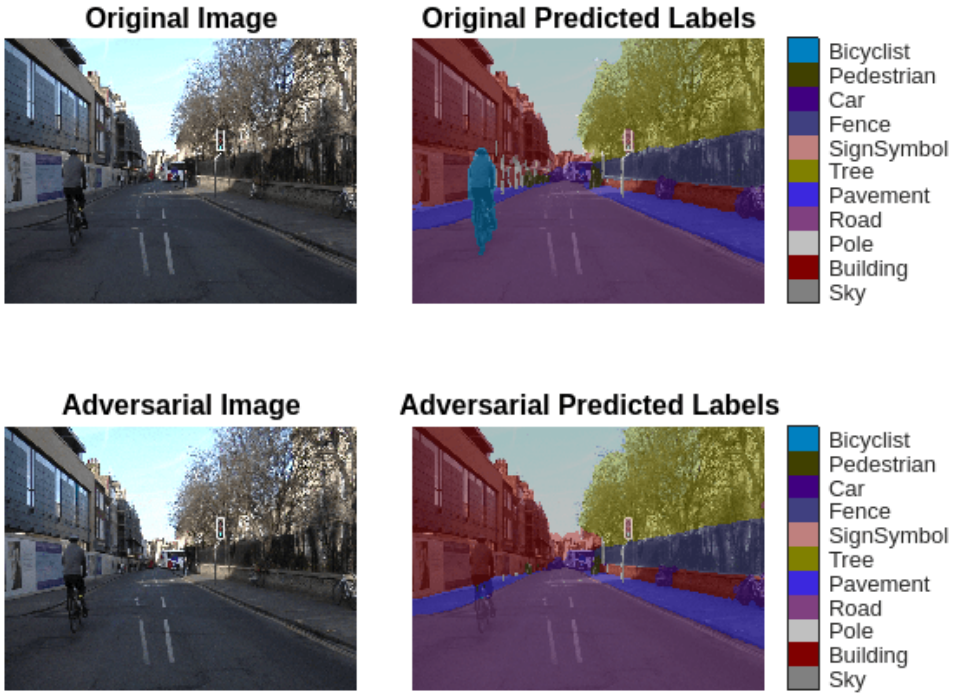

Figure 2.2: Deeplab v3+ network misclassifies a bicyclist when attacked with Basic Iterative Method of adversarial attack with maximum perturbation 8, step size of 2 (pixel range: 0-255) and in 10 iterations (MathWorks.com et al., 2024).

Chapter 3

Methodology

This chapter provides a detailed account of the methodology employed in this work to evaluate the robustness of neural networks against adversarial attacks. It begins with an overview of the project, followed by an explanation of the naming conventions adopted across the various experiments. The chapter then outlines the resources utilized and offers a comprehensive description of the dataset and model architecture. Subsequent sections detail the processes of data loading and preprocessing, as well as the training methodologies and testing procedures applied—both under standard conditions and in the presence of adversarial perturbations. The methodology is presented with the intent of ensuring clarity and reproducibility of the experimental process.

3.1 Project Overview

This dissertation investigates an approach to enhancing the robustness of neural networks against adversarial attacks while maintaining decent accuracy on normal data, with a particular focus on the Fast Gradient Sign Method (FGSM) attack. A visualization of the project has been provided in Figure 3.1 for clarity. The experiments were conducted on a High Performance Computing Cluster at the University of Leeds using MATLAB. The primary objective was to train an 18-layer deep residual neural network (ResNet-18) on the CIFAR-10 dataset using three distinct training methodologies, each aimed at understanding and improving the network's robustness against adversarial attacks while keeping respectable accuracy on normal data:

1. Experiment 1: Benchmark Training on Normal Data
   The first experiment involved training the ResNet-18 model from scratch using the CIFAR-10 dataset. The training was performed solely on normal, unperturbed images to establish a benchmark for the network's performance. This experiment serves as a control to compare the effects of adversarial training in subsequent experiments. All the files related to this experiment are suffixed with "v1" (e.g., resnet_18_v1.m, nrml_validation_test_v1.m).

2. Experiment 2: Training on Adversarial Data
   In the second experiment, the ResNet-18 model was trained exclusively on adversarial

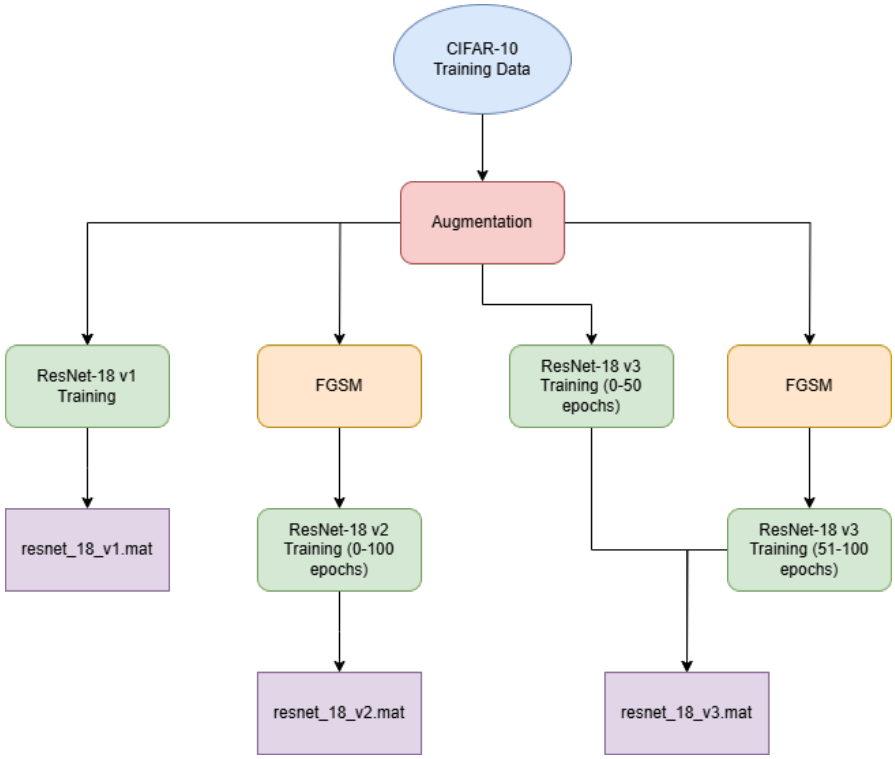

Figure 3.1: Visual representation of the entire project workflow.

data generated using the FGSM technique. The adversarial data was crafted specifically against the model being trained, providing insights into how adversarial training impacts the model's accuracy on both adversarial and normal data. All the files related to this experiment are suffixed with "v2" (e.g., resnet_18_v2.m, FGSM_validation_test_v2.m).

3. Experiment 3: Hybrid Training with Normal and Adversarial Data
   The third experiment introduced a hybrid training approach, where the ResNet-18 model was trained on a balanced mix of 50% normal images and 50% adversarial images. This approach aimed to mitigate the accuracy drop on normal data observed in adversarial training while improving the model's robustness against adversarial attacks. All the files related to this experiment are suffixed with "v3" (e.g., resnet_18_v3.mat, nrml_validation_test_v3.m).

Upon completing the training in each experiment, the models were tested on both normal validation data and progressively stronger adversarial data to evaluate their performance and robustness.

3.2 Resources Utilized

The experiments detailed in this dissertation were executed using the computational resources provided by the ARC4 Cluster, part of the High Performance Computing (HPC) facility at the University of Leeds. All models were trained on a single NVIDIA Tesla V100 GPU accelerator,

which is equipped with 32GB of memory. The training process for each model, regardless of the adversarial training intensity (i.e., the value of ϵ in Equation 2.3), typically required approximately six hours to complete.

The models were developed and trained using MATLAB R2022a, leveraging the MATLAB Deep Learning Toolbox to manually design the ResNet-18 architecture and to employ verification tools such as Gradient-weighted Class Activation Mapping (Grad-CAM) (R. R. Selvaraju et al., 2017). Additionally, the MATLAB Parallel Computing Toolbox was utilized to fully exploit the capabilities of the GPU, thereby accelerating the training process. The combination of these software tools and HPC resources was crucial in efficiently conducting the experiments, enabling the exploration of various training methodologies and their effects on model robustness.

3.3 Data Description

The experiments conducted in this dissertation utilize the CIFAR-10 dataset, a well-established benchmark dataset in the field of computer vision, particularly in the training and evaluation of deep learning models. The CIFAR-10 dataset is a labeled subset of the larger 80 million tiny images dataset, which was collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. This dataset is widely used for image classification tasks due to its manageable size and the diversity of its content.

The CIFAR-10 dataset consists of 60,000 32x32 color images, distributed across 10 mutually exclusive classes. Each class contains 6,000 images, with the dataset divided into 50,000 training images and 10,000 test images. The 10 classes represented in the dataset are: *airplane*, *automobile*, *bird*, *cat*, *deer*, *dog*, *frog*, *horse*, *ship*, and *truck*. The classes are designed to be completely distinct, with no overlap between categories; for instance, the "automobile" class includes sedans and SUVs, while the "truck" class encompasses only larger trucks, excluding pickup trucks.

The dataset is structured into five training batches and one test batch, each containing 10,000 images. The test batch is composed of 1,000 randomly selected images from each class, ensuring an equal representation across all categories. The training batches collectively contain 5,000 images per class.

In its stored format, each batch file contains a dictionary with two primary components: *data* and *labels*. The *data* array is a 10,000x3,072 numeric array of unsigned 8-bit integers, where each row corresponds to a 32x32 color image with each pixel ranging from 0 to 255 in value. The image data is stored in row-major order, with the first 1,024 entries representing the red channel, the next 1,024 the green channel, and the final 1,024 the blue channel. The *labels* array consists of 10,000 integers ranging from 0 to 9, where each integer corresponds to the class label of the respective image in the *data* array.

Additionally, the dataset includes a file named *batches.meta*, which contains a dictionary

with a single key, *label_names*. This key maps to a list of the 10 class names, providing a meaningful interpretation of the numeric labels used in the dataset.

The CIFAR-10 dataset serves as the foundation for the training and evaluation of the models in this study. Given the nature of the experiments, which include adversarial training and testing, the CIFAR-10 dataset was particularly advantageous due to its well-defined class boundaries and the small size of the images, which facilitate efficient computation. Source

3.4 Model Description

The neural network model employed in this research is based on the ResNet-18 architecture, a deep convolutional neural network specifically designed to address the vanishing gradient problem commonly encountered in deep networks. The ResNet-18 architecture achieves this through the use of residual connections, which allow the network to learn identity mappings more effectively, thereby improving the flow of gradients during back-propagation. A visualization of the network structure along with residual block has been provided in the Figure 3.2.

3.4.1 Architecture

The ResNet-18 model used in the experiments with it's 11.1 million learnable parameters, consists of the following layers:

- **Input Layer:** The input layer accepts images of size $224 \times 224 \times 3$, corresponding to the height, width, and RGB color channels of the input images. The input data is normalized using the `zscore` normalization method with a mean of 0 and a standard deviation of 1.
- **Convolutional and Pooling Layers:** The network begins with a convolutional layer consisting of 64 filters, each of size 7×7, with a stride of 2 and padding of 3. This is followed by a batch normalization layer and a ReLU activation function. The resulting feature map is then passed through a max-pooling layer with a filter size of 3×3, a stride of 2, and padding of 1.
- **Residual Blocks:** The core of the ResNet-18 architecture is composed of four residual blocks, each containing two convolutional layers with 3×3 filters. The first set of residual blocks in each stage maintains the same number of filters as the previous stage, while the subsequent blocks double the number of filters to increase the depth of the network. Each residual block includes a shortcut connection (an identity mapping) that bypasses the convolutional layers, and the output of this shortcut connection is added to the output of the convolutional layers. The residual blocks are as follows:
  - **Stage 1:** Two residual blocks, each with 64 filters.
  - **Stage 2:** Two residual blocks, each with 128 filters.

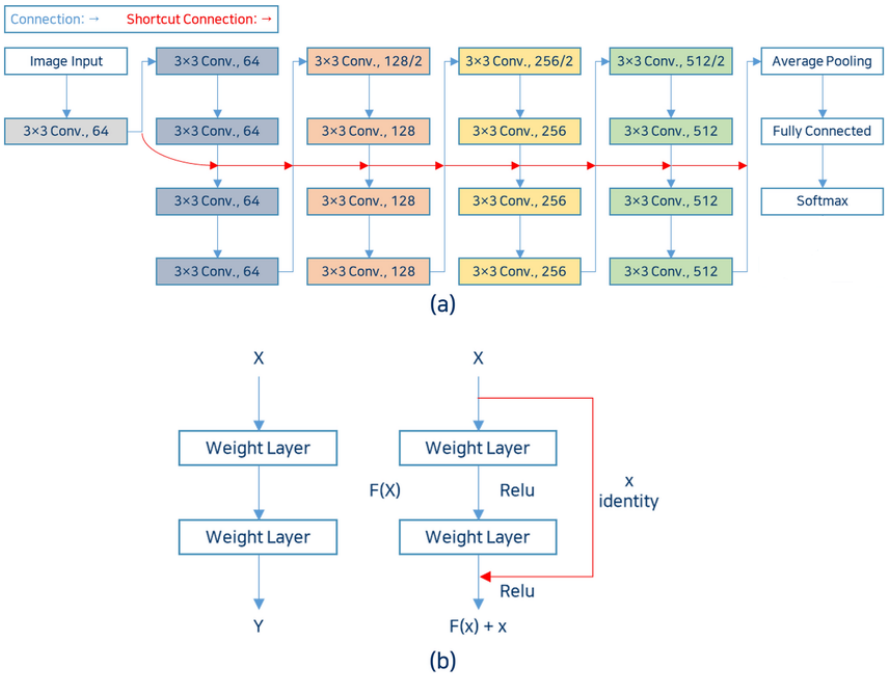

Figure 3.2: ResNet-18 architecture: (a) Neural network structure of ResNet-18 (b) Residual learning: building block. (Kim J. et al., 2022)

  - **Stage 3:** Two residual blocks, each with 256 filters.
  - **Stage 4:** Two residual blocks, each with 512 filters.

  Each residual block is followed by batch normalization for improving the training of the model by normalizing the inputs to each layer and Rectified Linear Unit (ReLU) activation layers to output the input directly if it is positive and zero otherwise, introducing non-linearity into the model while being computationally efficient.

- **Global Average Pooling Layer:** The output of the final residual block is passed through a global average pooling layer, which reduces the spatial dimensions of the feature map, resulting in a single feature vector for each image.

- **Fully Connected Layer and Softmax:** The feature vector is fed into a fully connected layer with 10 output units, corresponding to the 10 classes of the CIFAR-10 dataset. The final classification probabilities are generated by a softmax layer, which produces a probability distribution over the 10 classes.

- **Classification Layer:** The final classification layer has been removed due to lack of support of the same on MATLAB R2024a. This has been done to make the model compatible on MATLAB R2024a.

3.4.2 Initialization, Model Loss and Optimizer

The ResNet-18 model was trained using the following initialization, loss function and optimizer:

- **Weight Initialization:** The model's weights were initialized using the default initialization method in MATLAB's Deep Learning Toolbox. Source
- **Loss Function:** The loss function used in the model is the cross-entropy loss, which measures the difference between the predicted probability distribution and the true labels. This loss function is commonly used for classification tasks as it encourages the model to output high probabilities for the correct class while penalizing incorrect predictions. Source
- **Optimizer:** The Stochastic Gradient Descent with Momentum (SGDM) optimizer was employed to update the model's weights by combining the gradient of the current mini-batch with a fraction of the previous update during training, which helps in accelerating convergence and avoiding local minima. Source

This ResNet-18 architecture, with its residual connections, forms the foundation of the experiments conducted in this dissertation. The design choices were made to ensure that the model training is computationally less expensive while maintaining decent accuracy.

NOTE: The plot of the entire model architecture along with all of it's layers and learnable parameters can be analyzed in MATLAB by placing any of the models (resnet_18_v1.mat, resnet_18_v2.mat or resnet_18_v3.mat) in the current directory and using the following code snippet:

```
% Load the model
load("resnet_18_v1.mat", "net");

% Analyze the network
analyzeNetwork(net);
```

3.5 Data Downloading and Loading

The data used in the experiments was obtained from the CIFAR-10 dataset, a popular benchmark in the field of computer vision. To ensure reproducibility and ease of use, the data downloading and loading procedures were automated through custom MATLAB functions. This section details the process of downloading the dataset and loading it into memory for training and testing the models.

3.5.1 Data Downloading

The first step in the data preparation process is to download the CIFAR-10 dataset from its official source if it is not already present in the specified directory. This is achieved using the `downloadCIFARData` function:

```
downloadCIFARData(datadir);
```

The `downloadCIFARData` function is defined as follows:

```
function downloadCIFARData(destination)

    url = 'https://www.cs.toronto.edu/~kriz/cifar-10-matlab.tar.gz';
    unpackedData = fullfile(destination,'cifar-10-batches-mat');

    if ~exist(unpackedData,'dir')
        fprintf('Downloading CIFAR-10 dataset (175 MB). This can take a while
    ...');
        untar(url,destination);
        fprintf('done.\n\n');
    end

end
```

This function first checks if the CIFAR-10 dataset has already been downloaded and unpacked in the specified `destination` directory. The path to the unpacked data is constructed using the `fullfile` function, which combines the destination directory with the folder name `'cifar-10-batches-mat'`.

If the dataset is not found in the destination directory (checked using the `exist` function), the dataset is downloaded from the URL specified by the `url` variable. The `untar` function is then used to extract the contents of the downloaded archive to the destination directory. The user is informed of the download progress via messages printed to the MATLAB command window using the `fprintf` function. This process ensures that the dataset is only downloaded once, saving both time and bandwidth. Source

3.5.2 Data Loading

Once the dataset is available locally, the next step is to load the data into MATLAB for use in training and testing the neural network models. This is done using the `loadCIFARData` function:

```
[XTrain,TTrain,XValidation,TValidation] = loadCIFARData(datadir);
```

The `loadCIFARData` function is defined as follows:

```
function [XTrain,YTrain,XTest,YTest] = loadCIFARData(location)

```

```
    location = fullfile(location,'cifar-10-batches-mat');

    [XTrain1,YTrain1] = loadBatchAsFourDimensionalArray(location,'
    data_batch_1.mat');
    [XTrain2,YTrain2] = loadBatchAsFourDimensionalArray(location,'
    data_batch_2.mat');
    [XTrain3,YTrain3] = loadBatchAsFourDimensionalArray(location,'
    data_batch_3.mat');
    [XTrain4,YTrain4] = loadBatchAsFourDimensionalArray(location,'
    data_batch_4.mat');
    [XTrain5,YTrain5] = loadBatchAsFourDimensionalArray(location,'
    data_batch_5.mat');
    XTrain = cat(4,XTrain1,XTrain2,XTrain3,XTrain4,XTrain5);
    YTrain = [YTrain1;YTrain2;YTrain3;YTrain4;YTrain5];

    [XTest,YTest] = loadBatchAsFourDimensionalArray(location,'test_batch.mat
    ');
end
```

This function is responsible for loading the CIFAR-10 data from the specified directory into MATLAB variables. The function first constructs the full path to the dataset directory using the `fullfile` function.

The CIFAR-10 dataset is divided into five training batches and one test batch. The function `loadBatchAsFourDimensionalArray` is called for each training batch to load the image data and labels. These batches are then concatenated along the fourth dimension using the `cat` function, resulting in a 4D array `XTrain` that contains all the training images, and a corresponding label array `YTrain`.

The test batch is loaded separately into the arrays `XTest` and `YTest`. These arrays contain the validation images and labels, respectively. Source

3.5.3 Batch Loading and Label Conversion

The `loadBatchAsFourDimensionalArray` function is used within `loadCIFARData` to load each batch of data:

```
function [XBatch,YBatch] = loadBatchAsFourDimensionalArray(location,
    batchFileName)
    s = load(fullfile(location,batchFileName));
    XBatch = s.data';
    XBatch = reshape(XBatch,32,32,3,[]);
    XBatch = permute(XBatch,[2 1 3 4]);
    YBatch = convertLabelsToCategorical(location,s.labels);
end
```

This function loads a batch of CIFAR-10 data from a specified `batchFileName`. The data is stored in a structure `s`, and the image data is extracted into `XBatch` by transposing the data array. The `reshape` function is then used to reshape the data into a 32x32 image with 3

color channels (RGB), for all images in the batch. The `permute` function is used to rearrange the dimensions so that the data is in the correct format for input into the neural network.

The labels are converted into categorical format using the `convertLabelsToCategorical` function:

```
function categoricalLabels = convertLabelsToCategorical(location,
    integerLabels)
    s = load(fullfile(location,'batches.meta.mat'));
    categoricalLabels = categorical(integerLabels,0:9,s.label_names);
end
```

This function converts the numeric labels from the dataset into categorical labels, making them more interpretable and easier to use in MATLAB. The function loads the label names from the `batches.meta.mat` file, which contains a list of label names corresponding to the integer labels. The `categorical` function then maps the integer labels to their corresponding class names.

Overall, these functions work together to efficiently download, load, and prepare the CIFAR-10 dataset for use in training and testing the models in the experiments. Source

3.6 Data Preprocessing

Before training the model, it is essential to preprocess the data to ensure that it is in the correct format and augmented to improve the model's generalization ability. The data preprocessing involves resizing the input images, augmenting the data to introduce variability, and creating a data store for efficient loading during training.

3.6.1 Resizing the Images

Since the CIFAR-10 images are originally 32×32 pixels, they need to be resized to match the input size required by the ResNet-18 architecture, which expects images of size $224 \times 224 \times 3$ (height, width, and RGB channels). The following line defines the target image size:

```
imageSize = [224 224 3];
```

This line sets the variable `imageSize` to a 3-element vector that specifies the dimensions of the resized images.

3.6.2 Data Augmentation

Data augmentation is applied to introduce variability in the training data, which helps the model generalize better and reduces overfitting. The augmentation techniques applied here include random horizontal reflections and random translations along the x and y axes. The range for the translations is set using the `pixelRange` variable:

```
pixelRange = [-4 4];
```

This defines the range of translation in both the x and y directions, allowing the images to be shifted randomly by up to 4 pixels in either direction.

The augmentation process is defined using the `imageDataAugmenter` function:

```
imageAugmenter = imageDataAugmenter( ...
    RandXReflection=true, ...
    RandXTranslation=pixelRange, ...
    RandYTranslation=pixelRange);
```

This creates an `imageDataAugmenter` object, which specifies the augmentation techniques to be applied. The parameter `RandXReflection=true` enables random horizontal flipping of images, while `RandXTranslation` and `RandYTranslation` specify the range for random translations along the x and y axes, respectively, as defined by the `pixelRange` variable.

3.6.3 Creating an Augmented Image Datastore

To efficiently load and augment the data during training, an augmented image datastore is created using the `augmentedImageDatastore` function:

```
augimdsTrain = augmentedImageDatastore(imageSize,XTrain,TTrain, ...
    DataAugmentation=imageAugmenter);
```

This line creates the variable `augimdsTrain`, which is an augmented image datastore. The datastore resizes the training images `XTrain` to the target size `imageSize` and applies the augmentation defined by `imageAugmenter`. Additionally, it associates the images with their corresponding labels `TTrain`. The `augmentedImageDatastore` function allows the training data to be loaded in mini-batches during training, improving memory efficiency and reducing the overall training time.

3.7 Training

The training process of the ResNet-18 models in the experiments follows a similar approach, with variations in data used for training across the three experiments. This section describes the training configuration, data preparation for mini-batch, training loop and forward & backward pass common across Experiment 1, 2 & 3, followed by a comparison of the approaches used in all the experiments.

Training Configuration

The training configuration includes setting the number of epochs, mini-batch size, and learning rate:

```
numEpochs = 100;
miniBatchSize = 128;
learnRate = 0.01;
```

Here, all the models are trained for 100 epochs, meaning that the entire training dataset is passed through the models 100 times. The mini-batch size is set to 128, which means that the dataset is divided into batches of 128 images for each forward and backward pass during training. The learning rate is set to 0.01, which controls the step size for updating the models' parameters during training. A higher learning rate could cause the models to converge faster but risks overshooting the optimal parameters, whereas a lower learning rate provides more fine-grained updates at the cost of longer training time.

Data Preparation for Mini-Batch Training

The mini-batch queue is created for all the models using the `minibatchqueue` function, which prepares the data for training by creating mini-batches from the augmented image datastore:

```
mbq = minibatchqueue(augimdsTrain, ...
    MiniBatchSize=miniBatchSize,...
    MiniBatchFcn=@preprocessMiniBatch,...
    MiniBatchFormat=["SSCB",""]);
```

This line sets up a data pipeline that retrieves mini-batches from the `augimdsTrain` datastore (which was created during data preprocessing). The function `preprocessMiniBatch` is applied to each mini-batch to ensure that the data is in the correct format. The `MiniBatchFormat` option specifies that the input data should be in the format `["SSCB"]`, where "S" represents the spatial dimensions, "C" represents the channels (RGB), and "B" represents the mini-batch dimension.

Training Loop

All the training loop consists of an outer loop that iterates over the epochs and an inner loop that processes each mini-batch of data. The outer loop begins with the following lines:

```
epoch = 0;
while epoch < numEpochs
    epoch = epoch + 1;
    shuffle(mbq);
```

The training starts with epoch 0 and continues until the specified number of epochs is reached. At the beginning of each epoch, the mini-batch queue is shuffled to ensure that the data is presented to the model in a different order each time, helping to prevent overfitting.

Within each epoch, the inner loop processes the mini-batches:

```
while hasdata(mbq)
    iteration = iteration + 1;
    [X,T] = next(mbq);
```

The `hasdata` function checks whether there is more data to process, and the `next` function retrieves the next mini-batch of images `X` and corresponding labels `T`.

Forward and Backward Passes

For each mini-batch, the models perform forward and backward passes to compute the loss and update the models' parameters. The data is converted into a `gpuArray` if any GPU is available:

```
if canUseGPU
    X = gpuArray(X);
    T = gpuArray(T);
end
```

Next, the `dlfeval` function is used to evaluate the model loss and gradients:

```
[loss,gradients,state] = dlfeval(@modelLoss,net,X,T);
net.State = state;
```

The function `modelLoss` computes the cross-entropy loss between the predicted labels and the true labels. The gradients of the loss with respect to the model parameters are calculated using the `dlgradient` function, and the model state is updated accordingly.

3.7.1 Experiment 1: Training on Normal Data

In Experiment 1, the ResNet-18 model is trained on the CIFAR-10 dataset consisting of unperturbed, normal images. The training process involves several key steps, which are outlined and explained below.

Updating the Model Parameters

The model parameters are updated using Stochastic Gradient Descent with Momentum (SGDM), as implemented in the following line:

```
[net,velocity] = sgdmupdate(net,gradients,velocity,learnRate);
```

This step updates the model weights based on the computed gradients, with momentum used to help accelerate convergence and avoid local minima. The learning rate of 0.01 controls the step size for each parameter update.

3.7.2 Experiment 2: Training on Adversarial Data

In Experiment 2, the ResNet-18 model is trained exclusively on adversarially perturbed data generated using the Fast Gradient Sign Method (FGSM). This experiment aims to assess the effect of adversarial training on the model's robustness by exposing it to adversarial examples generated with respect to itself throughout the training process. This implies the model used to generate the adversarial examples is the same as the training model. The training procedure in Experiment 2 shares similarities with Experiment 1 but introduces additional steps to generate adversarial examples on-the-fly during training.

Adversarial Perturbations

Adversarial perturbations are introduced using the Fast Gradient Sign Method (FGSM) with the following parameters:

```
epsilon = 2;
numIter = 1;
initialization = "random";
alpha = epsilon;
```

The parameter `epsilon` controls the strength of the adversarial perturbation. In this case, `epsilon = 2`, meaning the perturbations applied to the images are of relatively low magnitude, precisely 0.7% of the maximum pixel value. The `numIter` parameter specifies the number of iterations to apply the perturbations (set to 1 in this case), and `initialization = "random"` indicates that the initial perturbation is randomly generated. The `alpha` parameter, which controls the step size for each iteration, is set equal to `epsilon`.

Training Loop with Adversarial Data

The training loop proceeds similarly to Experiment 1, but with the addition of adversarial perturbations applied to the mini-batch data. The outer loop iterates over the epochs, shuffling the mini-batch queue at the start of each epoch. The key difference in this experiment lies in the application of adversarial perturbations to the input data. This is accomplished using the `basicIterativeMethod` function, which modifies the input data using the FGSM approach:

```
X = basicIterativeMethod(netFGSM,X,T,alpha,epsilon, ...
    numIter,initialization);
```

Here, the `basicIterativeMethod` function generates adversarial examples by iteratively perturbing the input data based on the model's gradient with respect to the input. The magnitude of the perturbation is controlled by `epsilon` and `alpha`, while the direction of the perturbation is guided by the gradient of the loss function. The input data is modified in the direction that maximizes the model's loss, making it more difficult for the model to classify correctly.

Updating the Model Parameters

The model parameters are updated using the SGDM optimizer, as in Experiment 1:

```
[netFGSM,velocity] = sgdmupdate(netFGSM,gradients,velocity,learnRate);
```

3.7.3 Experiment 3: Hybrid Training with Normal and Adversarial Data

In Experiment 3, the ResNet-18 model is trained using a hybrid approach that combines normal and adversarial data. The key distinction in this experiment is that during the first 50 epochs, the model is trained solely on normal data, as in Experiment 1. However, from epoch 51 onward, adversarial perturbations are applied to the input data, similar to the approach in Experiment 2 with parameters for generating adversarial perturbation identical to Experiment 2. This method aims to balance the model's accuracy on clean data with its robustness to adversarial attacks.

Training Loop: Normal Data First, Adversarial Data Later

The training loop follows the same structure as the previous experiments, iterating over epochs and mini-batches. The key difference in this experiment is that adversarial perturbations are only applied after epoch 50:

```
if epoch > 50
    % Apply adversarial perturbations to the data.
    X = basicIterativeMethod(netMX,X,T,alpha,epsilon, ...
        numIter,initialization);
end
```

During the first 50 epochs, the model is trained on clean images without any perturbations, similar to Experiment 1. From epoch 51 onward, adversarial examples are generated on-the-fly using the `basicIterativeMethod` function with the same parameters as in Experiment 2. This hybrid training approach allows the model to first learn robust features from clean data and then improve its adversarial robustness in the latter half of training.

Updating the Model Parameters

The model parameters are updated using the SGDM optimizer, following the same procedure as in the previous experiments:

```
[netMX,velocity] = sgdmupdate(netMX,gradients,velocity,learnRate);
```

The final models are saved as `resnet_18_v1.mat`, `resnet_18_v2.mat`, and `resnet_18_v3.mat`, respectively.

3.7.4 Key Functions for Training

Several key functions are used across the experiments to compute loss, apply adversarial perturbations, and preprocess data into small batches. These functions are described below.

preprocessMiniBatch Function

The `preprocessMiniBatch` function prepares each mini-batch of data for input into the network. It concatenates the input data and labels and ensures that the data is in the correct format for training:

```
function [X,T] = preprocessMiniBatch(XCell,TCell)

% Concatenate.
X = cat(4,XCell{1:end});

X = single(X);

% Extract label data from the cell and concatenate.
T = cat(2,TCell{1:end});

% One-hot encode labels.
T = onehotencode(T,1);

end
```

This function concatenates the images in the mini-batch along the fourth dimension (representing different samples in the batch) and converts them to single precision format. The labels are also concatenated and one-hot encoded to match the expected output format of the neural network. (Source)

modelGradientsInput Function

The `modelGradientsInput` function is used to compute the gradient of the loss with respect to the input data, which is essential for generating adversarial perturbations:

```
function gradient = modelGradientsInput(net,X,T)

T = squeeze(T);
T = dlarray(T,'CB');

[YPred] = forward(net,X);

loss = crossentropy(YPred,T);
gradient = dlgradient(loss,X);

end
```

This function first squeezes and formats the true labels `T` to match the dimensions expected by the network. It then computes the predictions `YPred` by performing a forward pass through the network. The cross-entropy loss is calculated between the predictions and the true labels, and the gradient of the loss with respect to the input data `X` is computed using `dlgradient`. This gradient is used in the adversarial training process to perturb the input data in the direction that maximizes the model's loss, thus generating adversarial examples. (Source)

modelLoss Function

The `modelLoss` function computes the cross-entropy loss between the predicted output and the true labels for a given mini-batch:

```
function [loss,gradients,state] = modelLoss(net,X,T)

[YPred,state] = forward(net,X);

loss = crossentropy(YPred,T);
gradients = dlgradient(loss,net.Learnables);

loss = double(loss);

end
```

In this function, the input data `X` is passed through the network `net` using the `forward` function, which returns the predictions `YPred` and the updated network state. The cross-entropy loss is then computed between the predicted labels `YPred` and the true labels `T`. The gradients of the loss with respect to the learnable parameters of the network are calculated using `dlgradient`, which are later used to update the network weights. The loss is converted to a double data type for easier logging and interpretation. (Source)

basicIterativeMethod Function

The `basicIterativeMethod` function is responsible for generating adversarial examples during training by iteratively applying small perturbations to the input data:

```
function XAdv = basicIterativeMethod(net,X,T,alpha,epsilon,numIter,
    initialization)

% Initialize the perturbation.
if initialization == "zero"
    delta = zeros(size(X),like=X);
else
    delta = epsilon*(2*rand(size(X),like=X) - 1);
end

for i = 1:numIter

    % Apply adversarial perturbations to the data.
    gradient = dlfeval(@modelGradientsInput,net,X+delta,T);
    delta = delta + alpha*sign(gradient);
    delta(delta > epsilon) = epsilon;
    delta(delta < -epsilon) = -epsilon;
end

XAdv = X + delta;

end
```

This function initializes the adversarial perturbation `delta` either as a zero matrix or a random matrix (depending on the initialization strategy). It then iteratively adjusts the perturbation by calculating the gradient of the loss with respect to the input data (using `modelGradientsInput`) and updating `delta` in the direction that increases the loss. The perturbation is bounded by `epsilon`, which limits the magnitude of the adversarial change to the input. The resulting adversarial example `XAdv` is the perturbed version of the original input data `X`. (Source)

3.7.5 Comparison of Methodologies

The training methodologies of the three experiments differ in terms of data exposure:

- **Experiment 1** focuses on optimizing the model for clean data only. It typically achieves high accuracy on normal data but lacks robustness against adversarial attacks.
- **Experiment 2** trains the model entirely on adversarial data, which improves its adversarial robustness at the cost of reduced accuracy on clean images.
- **Experiment 3** adopts a hybrid strategy, combining clean and adversarial data. The first 50 epochs focus on clean data, and the last 50 epochs introduce adversarial examples. This approach aims to achieve a balance between accuracy on clean images and robustness to adversarial attacks.

The hybrid approach of Experiment 3 represents a middle ground, attempting to retain the

benefits of both clean and adversarial training, whereas Experiment 1 and Experiment 2 represent extremes focused either solely on accuracy or solely on robustness.

3.8 Testing

After training the models in each experiment, two distinct testing procedures are employed to assess their performance. The first involves testing on normal, unperturbed validation data, providing a baseline for the model's classification accuracy under standard conditions. The second testing procedure evaluates the models' robustness against FGSM attacks, progressively applying stronger perturbations to the validation data and measuring how well the models can resist these FGSM adversarial examples. Finally, the third procedure tests the robustness of the models against a much stronger PGD adversarial attack. All the testing procedures follow a similar structure but differ in the type of input data provided to the model.

3.8.1 Testing on Normal Data

The first step in the testing process is to assess the models on clean validation data. The CIFAR-10 validation data, which has 10000 equally distributed classes of images, is loaded and resized to match the input dimensions of the model using `augmentedImageDatastore`, then it is divided into a mini-batch queues of 512 randomly selected validation images. This allows for faster inference and better memory management, especially when dealing with large datasets.

Once the model is loaded, predictions are generated for the 10000 validation images present in `test_batch.mat`. The output from the model, which consists of predicted class labels, is compared with the true labels of the validation images to calculate the model's average accuracy on clean data. The results are then logged into a text file for analysis and comparison with the results from adversarial testing.

This procedure provides a clear measure of the model's baseline performance, helping to evaluate how accurately it classifies normal images. The testing on normal data does not involve any adversarial perturbations, allowing for an untainted view of the model's ability to perform under ideal conditions.

3.8.2 Testing on FGSM Data

The second testing procedure assesses the robustness of the models on the CIFAR-10 validation set composed of 10000 images divided into mini-batch queues of 512 images by applying adversarial perturbations to the validation images. Adversarial examples are generated using the Fast Gradient Sign Method (FGSM), which perturbs the images in a direction that maximizes the model's prediction error. The strength of the perturbation is controlled by a parameter `epsilon` which starts from 2 and goes all the way up to 20, making the images more difficult to classify correctly.

To perform this test, the validation images are first passed through the network, with perturbations added to the inputs on-the-fly using `basicIterativeMethod` function and keeping `numIterAdv` to 1. The perturbed images are then classified by the model, and the predictions are compared with the true labels to compute accuracy. By varying the value of ϵ, the strength of the adversarial attack is progressively increased, providing insight into how well the model can maintain accuracy as the attack becomes stronger.

Lower accuracy with increasing ϵ values typically indicates the model's vulnerability to stronger adversarial attacks, whereas models that maintain higher accuracy despite larger perturbations are considered more robust.

3.8.3 Testing on PGD Data

The third testing procedure examines the robustness of the models by applying adversarial perturbations to the validation images generated using the Projected Gradient Descent (PGD), an iterative version of FGSM and is much stronger than the later. The entire validation set for CIFAR-10 consisting of 10000 equally distributed classes of images are divided into mini-batch queues of 512 images for efficient testing. The strength of the perturbation controlled by the value of ϵ is kept at 8 and the step size α is kept the same as ϵ.

Testing on PGD data follows a similar procedure to that of testing on FGSM data except that the perturbations added to the inputs of the model using `basicIterativeMethod` function is generated by setting the `numIterAdv` to 30.

This test measure the robustness of the models in extreme adversarial condition, where the adversarial PGD data may not be subtle and can be noticed by naked eyes. But the objective of this test is to test the extreme robustness of the models.

Chapter 4

Results and Analysis

This chapter presents a detailed comparison of the three experiments conducted in this dissertation, focusing on the models' performance across different testing scenarios. The chapter begins with an analysis of the validation accuracies on normal, unperturbed data to assess the baseline performance of each model. This is followed by an in-depth evaluation of the models under adversarial conditions using both the Fast Gradient Sign Method (FGSM) and Projected Gradient Descent (PGD) attacks. Finally, a visual interpretation of the models' decision-making processes is provided through Grad-CAM outputs, highlighting the key features each model focused on during classification. These analyses offer insights into how the different training methodologies impact the model's ability to generalize on clean data while improving robustness against adversarial attacks.

4.1 Comparative Analysis of Validation Results on Normal Data

The validation performance of the three models on normal data reveals distinct trends, highlighting the trade-offs between training methods in terms of accuracy. The results are summarized in Figure 4.1, which illustrates the validation accuracy on normal unperturbed data achieved by each model.

In Experiment 1 (v1), where the model was trained solely on normal, unperturbed data, the validation accuracy reached 91.58%. This high accuracy indicates that the model was well-optimized for clean data, as there were no adversarial examples during training. This model serves as a benchmark for how well the model architecture can perform under ideal conditions.

In contrast, Experiment 2 (v2), which employed adversarial training throughout, achieved a validation accuracy of 85.81%. The reduction in accuracy compared to Experiment 1 is expected, as adversarial training shifts the model's focus toward robustness against adversarial attacks, potentially sacrificing some performance on clean data. This suggests that while the model gained resilience to adversarial examples, its ability to generalize on normal data was slightly compromised.

Experiment 3 (v3), which employed a hybrid training approach using both normal and ad-

versarial data, resulted in a validation accuracy of 88.75%. This intermediate result reflects a balance between the two training methods. The model's performance on normal data improved compared to Experiment 2, though it did not reach the same level of accuracy as Experiment 1. This suggests that the hybrid approach helps mitigate the performance loss caused by adversarial training, while still improving the model's robustness.

The overall comparison shows that while adversarial training (Experiment 2) results in lower accuracy on clean data, a hybrid approach (Experiment 3) provides a compromise by improving accuracy on unperturbed data relative to purely adversarial training, while still retaining some of the benefits of adversarial robustness.

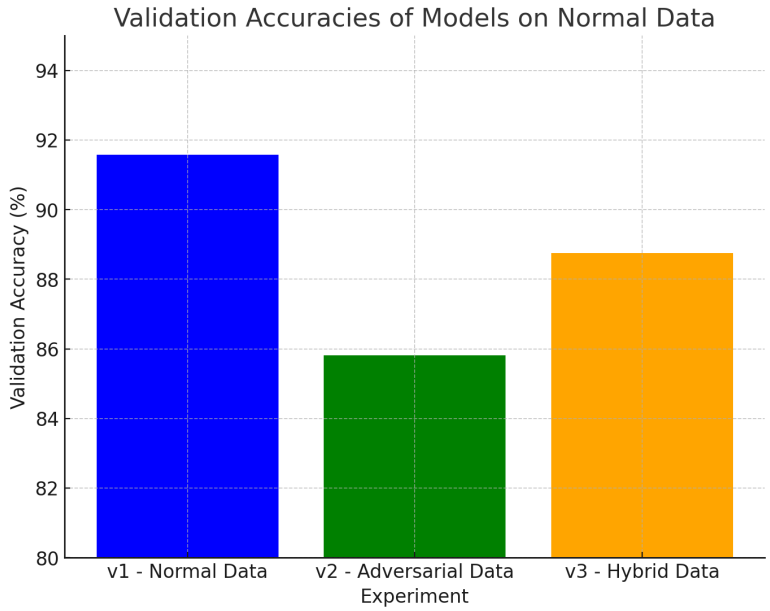

Figure 4.1: Validation accuracy of models on normal data for Experiments 1, 2, and 3.

4.2 FGSM Accuracy of Experiment 1 (v1)

In Experiment 1, where the model was trained exclusively on normal, unperturbed data, its performance under adversarial attacks drops significantly as the strength of the adversarial attack increases. Figure 4.2 illustrates this relationship, where the model's validation accuracy is plotted against increasing values of ϵ, which controls the strength of the adversarial perturbations.

The model starts with a validation accuracy of 61.31% at $\epsilon = 2$, but as ϵ increases, the accuracy steadily declines, reaching only 16.84% at $\epsilon = 20$. This significant drop demonstrates the model's vulnerability to adversarial attacks, as it was not exposed to adversarial examples during training.

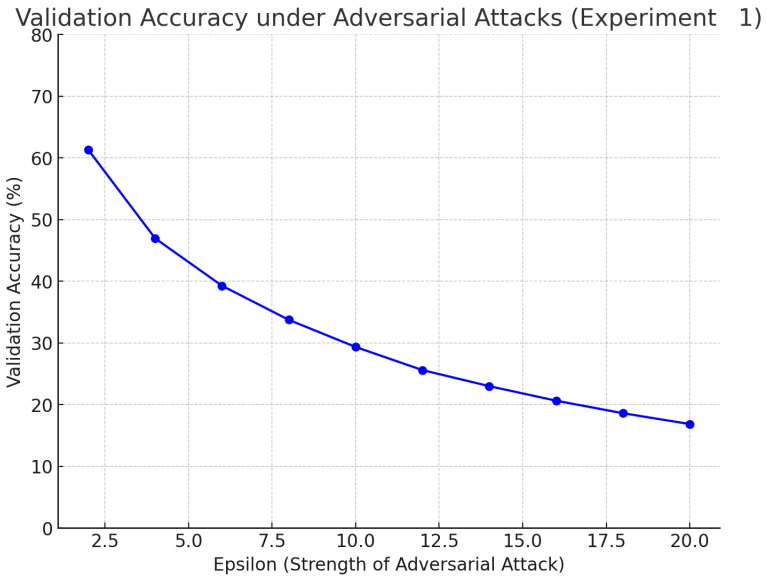

Figure 4.2: Validation accuracy of Experiment 1 (model v1) under progressively stronger FGSM attack.

4.3 FGSM Accuracy of Experiment 2 (v2)

Experiment v2 trained the model exclusively using adversarial data. As expected, this approach yielded improved robustness against adversarial attacks, as seen in Figure 4.3. The model maintains a higher level of accuracy across all ϵ values compared to v1.

Starting at 76.00% accuracy with $\epsilon = 2$, the model's performance decreases more gradually, reaching 46.01% at $\epsilon = 20$. This demonstrates the effectiveness of adversarial training in mitigating the impact of adversarial attacks, making the model more robust to perturbations.

4.4 FGSM Accuracy of Experiment 3 (v3)

Experiment 3 employed a hybrid training strategy, combining normal and adversarial training data. The results, displayed in Figure 4.5, show that this approach provides a balance between performance on normal data and robustness against adversarial attacks.

With an initial accuracy of 77.30% at $\epsilon = 2$, the model's performance remains relatively strong, gradually decreasing to 43.78% at $\epsilon = 20$. This result is slightly better than Experiment 2, suggesting that the hybrid training approach helps in achieving robustness while retaining some of the accuracy benefits seen in normal training.

4.5 Comparative Analysis of Validation Results on FGSM Data

A comparative analysis of the three experiments reveals the differences in how each training approach responds to progressively stronger adversarial attacks. Figure 4.4 provides a clear

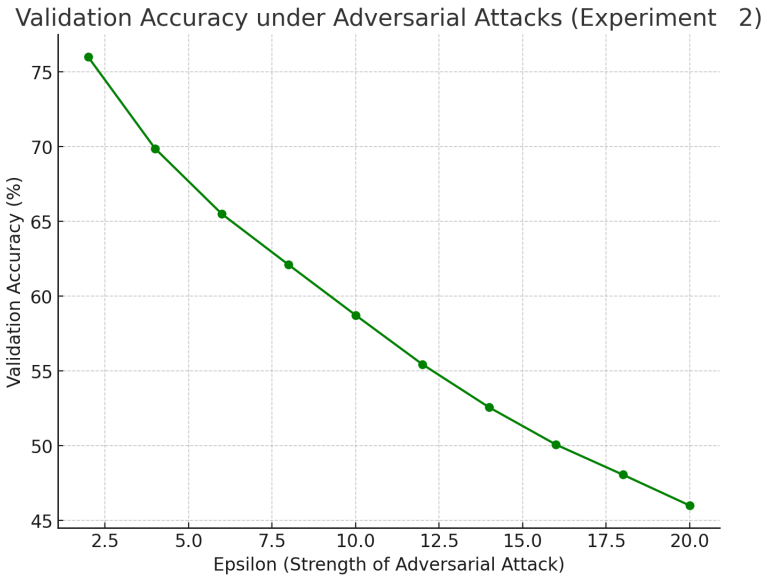

Figure 4.3: Validation accuracy of Experiment 2 (model v2) under progressively stronger FGSM attack.

comparison of the validation accuracy for v1, v2, and v3 across varying levels of ϵ.

Experiment 1, trained solely on normal data, exhibits the sharpest decline in accuracy as ϵ increases, reflecting its vulnerability to adversarial perturbations. In contrast, Experiment 2, trained entirely on adversarial data, maintains higher accuracy under attack, demonstrating its improved robustness. Experiment 3, with its hybrid approach, strikes a balance between the two, offering relatively high adversarial accuracy following the accuracy curve of Experiment 2 while still maintaining better result on clean data.

Table 4.1 further summarizes the validation accuracies for each model across the different ϵ values, providing a clearer understanding of how each training methodology impacts the model's robustness.

Epsilon	**v1 - Normal Training (%)**	**v2 - Adversarial Training (%)**	**v3 - Hybrid Training (%)**
2	61.31	76.00	77.30
4	46.96	69.88	70.45
6	39.27	65.50	65.17
8	33.74	62.10	61.09
10	29.34	58.74	57.30
12	25.59	55.45	53.77
14	22.99	52.57	50.88
16	20.63	50.08	48.20
18	18.6	48.07	46.00
20	16.84	46.01	43.78

Table 4.1: Comparative validation accuracy of all the models under progressively stronger FGSM attack.

The comparative analysis highlights that while adversarial training (v2) leads to enhanced

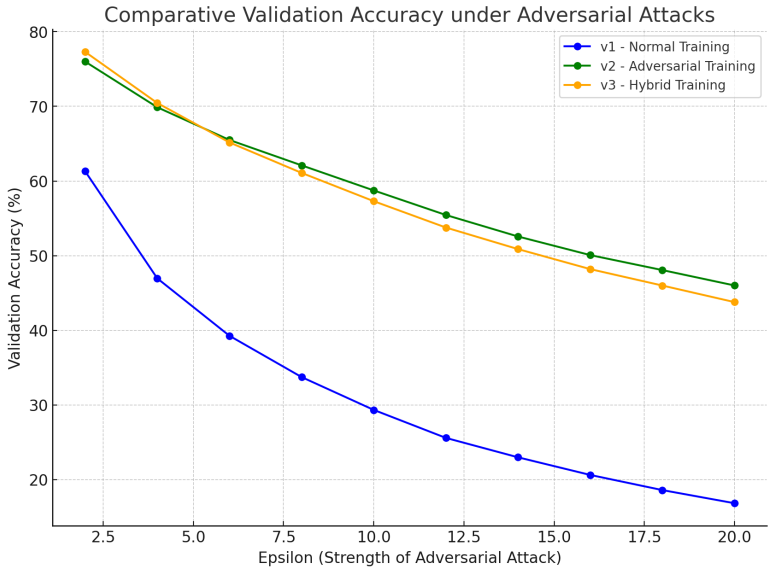

Figure 4.4: Validation accuracy of all the models under progressively stronger FGSM attack.

robustness, the hybrid approach (v3) provides a balance, maintaining competitive accuracy on both normal and adversarial data.

4.6 Comparative Analysis of Validation Results on PGD Data

In addition to the Fast Gradient Sign Method (FGSM), the robustness of the three models was also tested against a stronger adversarial attack: the Projected Gradient Descent (PGD) attack. PGD is widely regarded as one of the most powerful first-order adversarial attacks, and the results obtained from this test provide valuable insights into the models' ability to withstand adversarial perturbations.

The PGD attack results demonstrate stark differences between the three models, as shown in Figure 4.6. Model v1 (Experiment 1), which was trained solely on normal data, shows the weakest robustness to PGD attacks, achieving only 0.57% accuracy. This result highlights the vulnerability of models that are not trained with adversarial data, as even slight perturbations can drastically reduce their performance.

On the other hand, model v2 (Experiment 2), which was trained using adversarial data, exhibits significantly better robustness with an accuracy of 38.75%. This result reflects the effectiveness of adversarial training in preparing the model to resist stronger attacks like PGD. However, even with adversarial training, the model's accuracy is still far from ideal, demonstrating that the PGD attack remains a formidable challenge.

Model v3 (Experiment 3), trained with a hybrid approach of both normal and adversarial data, also shows notable robustness to PGD attacks, achieving 35.66% accuracy. While this is slightly lower than the accuracy of model v2, it still represents a substantial improvement over

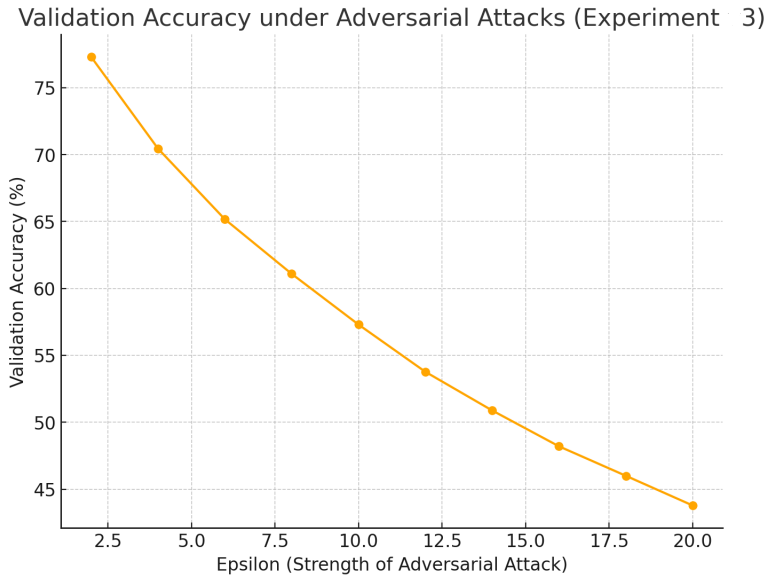

Figure 4.5: Validation accuracy of Experiment 3 (model v3) under progressively stronger FGSM attack.

model v1, suggesting that the hybrid approach offers a balance between clean accuracy and adversarial robustness, even against more powerful attacks like PGD.

As observed in this test, while normal training (model v1) leaves models highly susceptible to such attacks, adversarially trained models (model v2) and hybrid models (model v3) demonstrate much stronger resilience, even though they still experience performance degradation under such adversarial conditions. The comparison emphasizes that while adversarial training and hybrid approaches significantly enhance robustness, the PGD attack remains a challenging benchmark for evaluating the security of neural networks.

4.7 Grad-CAM Output Analysis

The Grad-CAM technique leverages the gradients of the classification score with respect to the final convolutional feature map to highlight the areas of an input image that have the greatest influence on the classification decision. Regions with larger gradients indicate where the model's prediction is most sensitive to the image data, reflecting the areas that contribute most to the final classification score. (Source)

Figure 4.7 demonstrates the Grad-CAM output for model v1 (resnet_18_v1.mat) highlighting that the model focuses primarily on the central and rear portions of the vehicle when making its classification decision. This concentration of activation suggests that these areas contain the most critical features for the model's prediction. However, the heatmap indicates a lack of attention to the front portion of the car, potentially signifying that the model might be overlooking some informative features in this region.

The output for model v2 (resnet_18_v2.mat) in Figure 4.8 reveals that the model places

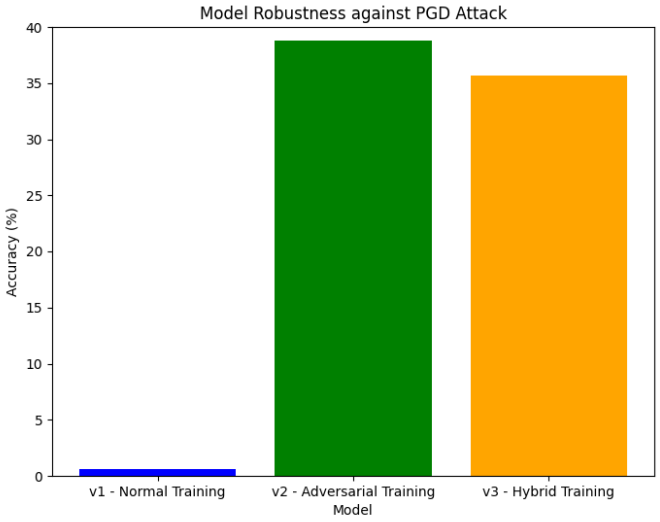

Figure 4.6: Comparison of robustness of all the models against PGD attack.

significant attention on the central body of the vehicle, particularly around the door and lower sections. Compared to model v1, the focus is slightly broader, but still concentrated on key structural features. This broader distribution of activation may indicate a better generalization, as the model seems to capture a wider set of relevant features while still prioritizing the core aspects of the vehicle.

In Figure 4.9 model v3 (resnet_18_v3.mat) shows a relatively balanced focus, covering both the central body of the vehicle and extending toward the roof, wheel and lower parts of the car. This wider spread of attention compared to model v1 and v2 indicates that the hybrid training approach in Experiment 3 has allowed the model to capture a more diverse set of features from the image, while still concentrating on the most relevant areas.

Figure 4.7: Grad-CAM output for model v1 (resnet_18_v1.mat).

Figure 4.8: Grad-CAM output for model v2 (resnet_18_v2.mat).

Figure 4.9: Grad-CAM output for model v3 (resnet_18_v3.mat).

Chapter 5

Discussion and Future Work

This chapter provides a comprehensive discussion of the results obtained from the experiments conducted in this dissertation. It reflects on the effectiveness of various training approaches in enhancing the robustness of neural networks against adversarial attacks, particularly focusing on the balance between clean data accuracy and adversarial robustness. Furthermore, this chapter outlines potential directions for future research, including the exploration of progressively stronger adversarial training strategies, aimed at improving the model's resilience to more sophisticated attacks, while maintaining performance on clean data. The insights and suggestions provided in this chapter offer a roadmap for refining adversarial defense mechanisms in neural network models.

5.1 Conclusion

This dissertation explored different training strategies for improving the robustness of neural networks against adversarial attacks, specifically focusing on the Fast Gradient Sign Method (FGSM) and Projected Gradient Descent (PGD) attacks. Through three distinct experiments—training on normal data, adversarial data, and a hybrid of both—the models were evaluated on their performance with clean data and progressively stronger adversarial attacks. The results demonstrated that while adversarial training significantly improves a model's robustness to attacks, it comes at the cost of decreased performance on normal data. However, the hybrid training approach strikes a meaningful balance, achieving competitive accuracy on both clean and adversarial data.

The comparative analysis across the experiments highlights that models trained exclusively on clean data are highly vulnerable to adversarial attacks, while adversarial training effectively increases robustness. The hybrid approach, as demonstrated in Experiment 3, offers a middle ground where the model generalizes better to both types of input hence achieving the primary goal of balancing accuracy on both normal and adversarial data.

5.2 Future Work

In this dissertation, a hybrid approach to training a neural network on both clean and adversarial data has shown promising results, balancing accuracy on clean data with robustness against adversarial attacks. However, there is potential for further refinement and exploration of this approach. One avenue for future research involves training a model using a progressively stronger adversarial training method, where the model is exposed to both clean data and adversarial data generated with varying levels of perturbation strength. Instead of relying on a fixed adversarial strength, the Fast Gradient Sign Method (FGSM) could be applied with progressively increasing ϵ values during training.

The primary goal of this approach would be to maintain high accuracy on clean data while simultaneously enhancing the model's robustness to stronger adversarial attacks such as PGD. By progressively increasing the adversarial perturbations during training, the model would be incrementally challenged, enabling it to adapt to more aggressive attacks over time. This method could prevent the model from overfitting to adversarial examples of a specific strength and might yield a model that generalizes better to both weaker and stronger adversarial attacks. Future work could also investigate optimizing the ratio of clean to adversarial data at each stage of training to ensure that the model retains its performance on clean inputs while becoming progressively more resilient to stronger adversarial attacks.

Bibliography

LeCun, Y., Bengio, Y. & Hinton, G. (2015), *Deep learning*, Nature 521(7553), 436–444. Source

Krizhevsky, A., Sutskever, I. & Hinton, G. E. (2012), *Imagenet classification with deep convolutional neural networks*, in Advances in neural information processing systems, pp. 1097–1105. Source

Eykholt, K., Evtimov, I., Fernandes, E., Li, B., Rahmati, A., Xia, C., Prakash, A., Kohno, T. & Song, D. (2018), *Robust physical-world attacks on deep learning visual classification*, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Source

Sutskever, I., Vinyals, O. & Le, Q. V. (2014), *Sequence to sequence learning with neural networks*, in Advances in neural information processing systems, pp. 3104–3112. Source

Kurakin, Alexey, Ian Goodfellow, & Samy Bengio. (2017), *Adversarial Examples in the Physical World.* Source.

Wang, Z. and Xu, L., 2024. *Susceptibility of Adversarial Attack on Medical Image Segmentation Models.* arXiv preprint arXiv:2401.11224 Source.

Ma, X., Niu, Y., Gu, L., Wang, Y., Zhao, Y., Bailey, J. & Lu, F., 2021. *Understanding adversarial attacks on deep learning based medical image analysis systems.* Pattern Recognition, 110, p.107332. Source

MathWorks.com, 2024 *Generate Adversarial Examples for Semantic Segmentation.* Source

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I. & Fergus, R. (2014), *Intriguing properties of neural networks*, arXiv preprint arXiv:1312.6199. Source

Goodfellow, I. J., Shlens, J. & Szegedy, C. (2015), *Explaining and harnessing adversarial examples*, in International Conference on Learning Representations (ICLR). Source

Papernot, N., McDaniel, P. & Goodfellow, I. (2016), *Transferability in machine learning: from phenomena to black-box attacks using adversarial samples.* Source

Liang, H.; He, E.; Zhao, Y.; Jia, Z.; Li, H. *Adversarial Attack and Defense: A Survey.* Electronics 2022, 11, 1283. Source

Papernot, N., McDaniel, P. & Goodfellow, I. (2017), *Practical black-box attacks against machine learning.* Source

Madry, A., Makelov, A., Schmidt, L., Tsipras, D. & Vladu, A. (2018), *Towards deep learning models resistant to adversarial attacks*, in International Conference on Learning Representations (ICLR).

Carlini, N. & Wagner, D. (2017), *Towards evaluating the robustness of neural networks*, 2017 IEEE Symposium on Security and Privacy (SP). Source

Ziang Yan, Yiwen Guo & Changshui Zhang, *Deep defense: Training dnns with improved adversarial robustness*, Advances in Neural Information Processing Systems (NeurIPS), 2018. Source

Gintare Karolina Dziugaite, Zoubin Ghahramani and Daniel M Roy, *A study of the effect of jpg compression on adversarial images*, 2016. Source

Hongyang Zhang, Yaodong Yu, Jiantao Jiao, Eric P Xing, Laurent El Ghaoui and Michael-I Jordan, *Theoretically principled trade-off between robustness and accuracy*, International Conference on Machine Learning (ICML), 2019. Source

Y. Dong et al., *Benchmarking Adversarial Robustness on Image Classification*, 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Seattle, WA, USA, 2020, pp. 318-328, doi: 10.1109/CVPR42600.2020.00040. Source

Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Dan Boneh and Patrick McDaniel, *Ensemble adversarial training: Attacks and defenses*, International Conference on Learning Representations (ICLR), 2018. Source

Tsipras, D., Santurkar, S., Engstrom, L., Turner, A. and Madry, A., 2018. Robustness may be at odds with accuracy. arXiv preprint arXiv:1805.12152. Source

Wong E., Rice L., Kolter J., 2020. *FAST IS BETTER THAN FREE: REVISITING ADVERSARIAL TRAINING.* ICLR, 2020. Source

R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh and D. Batra, *Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization*, 2017 IEEE International Conference on Computer Vision (ICCV), Venice, Italy, 2017, pp. 618-626, doi: 10.1109/ICCV.2017.74. Source

Kim, Jisoo & Park, Chul-Min & Kim, Sung & Cho, Angela. (2022). Convolutional neural network-based classification of cervical intraepithelial neoplasias using colposcopic image segmentation for acetowhite epithelium. Scientific Reports. 12. 10.1038/s41598-022-21692-5. Source

B. T. Polyak. *Some methods of speeding up the convergence of iteration methods.* USSR Computational Mathematics and Mathematical Physics, 1964. Source

Chapter 6

Appendix

6.1 Experiment 1

Code to train model v1 of Experiment 1 (resnet_18_v1.mat):

```
datadir = "/home/home02/mm23rn/resnet_18_v1";
downloadCIFARData(datadir);

[XTrain,TTrain,~,~] = loadCIFARData(datadir);

imageSize = [224 224 3];
pixelRange = [-4 4];

imageAugmenter = imageDataAugmenter( ...
    RandXReflection=true, ...
    RandXTranslation=pixelRange, ...
    RandYTranslation=pixelRange);

augimdsTrain = augmentedImageDatastore(imageSize,XTrain,TTrain, ...
    DataAugmentation=imageAugmenter);

% Define the network layers without the classification and softmax layers
lgraph = layerGraph();

tempLayers = [
    imageInputLayer([224 224 3],"Name","data","Normalization","zscore",...
    "Mean", 0, "StandardDeviation", 1)
    convolution2dLayer([7 7],64,"Name","conv1","BiasLearnRateFactor",0,"
    Padding",[3 3 3 3],"Stride",[2 2])
    batchNormalizationLayer("Name","bn_conv1")
    reluLayer("Name","conv1_relu")
    maxPooling2dLayer([3 3],"Name","pool1","Padding",[1 1 1 1],"Stride",[2
    2])];
lgraph = addLayers(lgraph,tempLayers);

```

```
tempLayers = [
    convolution2dLayer([3 3],64,"Name","res2a_branch2a","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn2a_branch2a")
    reluLayer("Name","res2a_branch2a_relu")
    convolution2dLayer([3 3],64,"Name","res2a_branch2b","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn2a_branch2b")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    additionLayer(2,"Name","res2a")
    reluLayer("Name","res2a_relu")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    convolution2dLayer([3 3],64,"Name","res2b_branch2a","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn2b_branch2a")
    reluLayer("Name","res2b_branch2a_relu")
    convolution2dLayer([3 3],64,"Name","res2b_branch2b","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn2b_branch2b")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    additionLayer(2,"Name","res2b")
    reluLayer("Name","res2b_relu")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    convolution2dLayer([1 1],128,"Name","res3a_branch1","BiasLearnRateFactor
    ",0,"Stride",[2 2])
    batchNormalizationLayer("Name","bn3a_branch1")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    convolution2dLayer([3 3],128,"Name","res3a_branch2a","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1],"Stride",[2 2])
    batchNormalizationLayer("Name","bn3a_branch2a")
    reluLayer("Name","res3a_branch2a_relu")
    convolution2dLayer([3 3],128,"Name","res3a_branch2b","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn3a_branch2b")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    additionLayer(2,"Name","res3a")
    reluLayer("Name","res3a_relu")];
```

```
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    convolution2dLayer([3 3],128,"Name","res3b_branch2a","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn3b_branch2a")
    reluLayer("Name","res3b_branch2a_relu")
    convolution2dLayer([3 3],128,"Name","res3b_branch2b","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn3b_branch2b")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    additionLayer(2,"Name","res3b")
    reluLayer("Name","res3b_relu")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    convolution2dLayer([1 1],256,"Name","res4a_branch1","BiasLearnRateFactor
    ",0,"Stride",[2 2])
    batchNormalizationLayer("Name","bn4a_branch1")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    convolution2dLayer([3 3],256,"Name","res4a_branch2a","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1],"Stride",[2 2])
    batchNormalizationLayer("Name","bn4a_branch2a")
    reluLayer("Name","res4a_branch2a_relu")
    convolution2dLayer([3 3],256,"Name","res4a_branch2b","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn4a_branch2b")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    additionLayer(2,"Name","res4a")
    reluLayer("Name","res4a_relu")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    convolution2dLayer([3 3],256,"Name","res4b_branch2a","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn4b_branch2a")
    reluLayer("Name","res4b_branch2a_relu")
    convolution2dLayer([3 3],256,"Name","res4b_branch2b","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn4b_branch2b")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
```

```
    additionLayer(2,"Name","res4b")
    reluLayer("Name","res4b_relu")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    convolution2dLayer([1 1],512,"Name","res5a_branch1","BiasLearnRateFactor
    ",0,"Stride",[2 2])
    batchNormalizationLayer("Name","bn5a_branch1")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    convolution2dLayer([3 3],512,"Name","res5a_branch2a","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1],"Stride",[2 2])
    batchNormalizationLayer("Name","bn5a_branch2a")
    reluLayer("Name","res5a_branch2a_relu")
    convolution2dLayer([3 3],512,"Name","res5a_branch2b","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn5a_branch2b")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    additionLayer(2,"Name","res5a")
    reluLayer("Name","res5a_relu")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    convolution2dLayer([3 3],512,"Name","res5b_branch2a","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn5b_branch2a")
    reluLayer("Name","res5b_branch2a_relu")
    convolution2dLayer([3 3],512,"Name","res5b_branch2b","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn5b_branch2b")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    additionLayer(2,"Name","res5b")
    reluLayer("Name","res5b_relu")
    globalAveragePooling2dLayer("Name","pool5")
    fullyConnectedLayer(10,"Name","fc10")
    softmaxLayer("Name","prob")];
lgraph = addLayers(lgraph,tempLayers);

% clean up helper variable
clear tempLayers;

lgraph = connectLayers(lgraph,"pool1","res2a_branch2a");
lgraph = connectLayers(lgraph,"pool1","res2a/in2");
lgraph = connectLayers(lgraph,"bn2a_branch2b","res2a/in1");
```

```
lgraph = connectLayers(lgraph,"res2a_relu","res2b_branch2a");
lgraph = connectLayers(lgraph,"res2a_relu","res2b/in2");
lgraph = connectLayers(lgraph,"bn2b_branch2b","res2b/in1");
lgraph = connectLayers(lgraph,"res2b_relu","res3a_branch1");
lgraph = connectLayers(lgraph,"res2b_relu","res3a_branch2a");
lgraph = connectLayers(lgraph,"bn3a_branch1","res3a/in2");
lgraph = connectLayers(lgraph,"bn3a_branch2b","res3a/in1");
lgraph = connectLayers(lgraph,"res3a_relu","res3b_branch2a");
lgraph = connectLayers(lgraph,"res3a_relu","res3b/in2");
lgraph = connectLayers(lgraph,"bn3b_branch2b","res3b/in1");
lgraph = connectLayers(lgraph,"res3b_relu","res4a_branch1");
lgraph = connectLayers(lgraph,"res3b_relu","res4a_branch2a");
lgraph = connectLayers(lgraph,"bn4a_branch2b","res4a/in1");
lgraph = connectLayers(lgraph,"bn4a_branch1","res4a/in2");
lgraph = connectLayers(lgraph,"res4a_relu","res4b_branch2a");
lgraph = connectLayers(lgraph,"res4a_relu","res4b/in2");
lgraph = connectLayers(lgraph,"bn4b_branch2b","res4b/in1");
lgraph = connectLayers(lgraph,"res4b_relu","res5a_branch1");
lgraph = connectLayers(lgraph,"res4b_relu","res5a_branch2a");
lgraph = connectLayers(lgraph,"bn5a_branch1","res5a/in2");
lgraph = connectLayers(lgraph,"bn5a_branch2b","res5a/in1");
lgraph = connectLayers(lgraph,"res5a_relu","res5b_branch2a");
lgraph = connectLayers(lgraph,"res5a_relu","res5b/in2");
lgraph = connectLayers(lgraph,"bn5b_branch2b","res5b/in1");

% Initialize dlnetwork
net = dlnetwork(lgraph);
net = initialize(net);

numEpochs = 100;
miniBatchSize = 128;
learnRate = 0.01;

mbq = minibatchqueue(augimdsTrain, ...
    MiniBatchSize=miniBatchSize,...
    MiniBatchFcn=@preprocessMiniBatch,...
    MiniBatchFormat=["SSCB",""]);

% Open a text file for writing metrics
fileID = fopen('metrics_log_resnet_18_v1.txt', 'w');
fprintf(fileID, 'Epoch\tIteration\tLoss\n'); % Header for the log file

velocity = [];
epoch = 0;
iteration = 0;

% Loop over epochs.
while epoch < numEpochs
    epoch = epoch + 1;
```

```
206
207     % Shuffle data.
208     shuffle(mbq)
209
210     % Loop over mini-batches.
211     while hasdata(mbq)
212         iteration = iteration + 1;
213
214         % Read mini-batch of data.
215         [X,T] = next(mbq);
216
217         % If training on a GPU, then convert data to gpuArray.
218         if canUseGPU
219             X = gpuArray(X);
220             T = gpuArray(T);
221         end
222
223         % Evaluate the model loss, gradients, and state.
224         [loss,gradients,state] = dlfeval(@modelLoss,net,X,T);
225         net.State = state;
226
227         % Update the network parameters using the SGDM optimizer.
228         [net,velocity] = sgdmupdate(net,gradients,velocity,learnRate);
229
230         % Write the metrics to the text file
231         fprintf(fileID, '%d\t%d\t%.4f\n', epoch, iteration, loss);
232     end
233 end
234
235 save("resnet_18_v1.mat", "net");
236
237 fclose(fileID);
238
239 quit;
240
241 %--------------------------SUPPORTING FUNCTIONS
        ---------------------------------------
242
243 function [loss,gradients,state] = modelLoss(net,X,T)
244
245 [YPred,state] = forward(net,X);
246
247 loss = crossentropy(YPred,T);
248 gradients = dlgradient(loss,net.Learnables);
249
250 loss = double(loss);
251
252 end
253
```

```
function [X,T] = preprocessMiniBatch(XCell,TCell)

% Concatenate.
X = cat(4,XCell{1:end});

X = single(X);

% Extract label data from the cell and concatenate.
T = cat(2,TCell{1:end});

% One-hot encode labels.
T = onehotencode(T,1);

end

function downloadCIFARData(destination)

url = 'https://www.cs.toronto.edu/~kriz/cifar-10-matlab.tar.gz';

unpackedData = fullfile(destination,'cifar-10-batches-mat');
if ~exist(unpackedData,'dir')
    fprintf('Downloading CIFAR-10 dataset (175 MB). This can take a while
    ...');
    untar(url,destination);
    fprintf('done.\n\n');
end

end

function [XTrain,YTrain,XTest,YTest] = loadCIFARData(location)

location = fullfile(location,'cifar-10-batches-mat');

[XTrain1,YTrain1] = loadBatchAsFourDimensionalArray(location,'data_batch_1.
    mat');
[XTrain2,YTrain2] = loadBatchAsFourDimensionalArray(location,'data_batch_2.
    mat');
[XTrain3,YTrain3] = loadBatchAsFourDimensionalArray(location,'data_batch_3.
    mat');
[XTrain4,YTrain4] = loadBatchAsFourDimensionalArray(location,'data_batch_4.
    mat');
[XTrain5,YTrain5] = loadBatchAsFourDimensionalArray(location,'data_batch_5.
    mat');
XTrain = cat(4,XTrain1,XTrain2,XTrain3,XTrain4,XTrain5);
YTrain = [YTrain1;YTrain2;YTrain3;YTrain4;YTrain5];

[XTest,YTest] = loadBatchAsFourDimensionalArray(location,'test_batch.mat');
end

```

```
function [XBatch,YBatch] = loadBatchAsFourDimensionalArray(location,
    batchFileName)
s = load(fullfile(location,batchFileName));
XBatch = s.data';
XBatch = reshape(XBatch,32,32,3,[]);
XBatch = permute(XBatch,[2 1 3 4]);
YBatch = convertLabelsToCategorical(location,s.labels);
end

function categoricalLabels = convertLabelsToCategorical(location,
    integerLabels)
s = load(fullfile(location,'batches.meta.mat'));
categoricalLabels = categorical(integerLabels,0:9,s.label_names);
end
```

Code to test model v1 of Experiment 1 (resnet_18_v1.mat) on normal data:

```
datadir = "/home/home02/mm23rn/resnet_18_v1";
downloadCIFARData(datadir);

[~,~,XValidation,TValidation] = loadCIFARData(datadir);

classes = categories(TValidation);

imageSize = [224 224 3];

augimdsVal = augmentedImageDatastore(imageSize,XValidation,TValidation);

load("resnet_18_v1.mat", "net");

miniBatchSize = 512;

mbqVal = minibatchqueue(augimdsVal, ...
    MiniBatchSize=miniBatchSize,...
    MiniBatchFcn=@preprocessMiniBatch,...
    MiniBatchFormat=["SSCB",""]);

YPred = modelPredictions(net,mbqVal,classes);
acc = mean(YPred == TValidation);

fileID = fopen('nrml_acc_resnet_18_v1.txt', 'w');

val_acc = "Validation accuracy: " + acc*100 + "%";

fprintf(fileID, val_acc);

quit;

%-------------------------SUPPORTING FUNCTIONS
    --------------------------------------
```

```
33
34 function [X,T] = preprocessMiniBatch(XCell,TCell)
35
36 % Concatenate.
37 X = cat(4,XCell{1:end});
38
39 X = single(X);
40
41 % Extract label data from the cell and concatenate.
42 T = cat(2,TCell{1:end});
43
44 % One-hot encode labels.
45 T = onehotencode(T,1);
46
47 end
48
49 function predictions = modelPredictions(net,mbq,classes)
50
51 predictions = [];
52
53 while hasdata(mbq)
54
55     XTest = next(mbq);
56     YPred = predict(net,XTest);
57
58     YPred = onehotdecode(YPred,classes,1)';
59
60     predictions = [predictions; YPred];
61 end
62
63 end
64
65 function downloadCIFARData(destination)
66
67 url = 'https://www.cs.toronto.edu/~kriz/cifar-10-matlab.tar.gz';
68
69 unpackedData = fullfile(destination,'cifar-10-batches-mat');
70 if ~exist(unpackedData,'dir')
71     fprintf('Downloading CIFAR-10 dataset (175 MB). This can take a while
    ...');
72     untar(url,destination);
73     fprintf('done.\n\n');
74 end
75
76 end
77
78 function [XTrain,YTrain,XTest,YTest] = loadCIFARData(location)
79
80 location = fullfile(location,'cifar-10-batches-mat');
```

```
81
82 [XTrain1,YTrain1] = loadBatchAsFourDimensionalArray(location,'data_batch_1.
       mat');
83 [XTrain2,YTrain2] = loadBatchAsFourDimensionalArray(location,'data_batch_2.
       mat');
84 [XTrain3,YTrain3] = loadBatchAsFourDimensionalArray(location,'data_batch_3.
       mat');
85 [XTrain4,YTrain4] = loadBatchAsFourDimensionalArray(location,'data_batch_4.
       mat');
86 [XTrain5,YTrain5] = loadBatchAsFourDimensionalArray(location,'data_batch_5.
       mat');
87 XTrain = cat(4,XTrain1,XTrain2,XTrain3,XTrain4,XTrain5);
88 YTrain = [YTrain1;YTrain2;YTrain3;YTrain4;YTrain5];
89
90 [XTest,YTest] = loadBatchAsFourDimensionalArray(location,'test_batch.mat');
91 end
92
93 function [XBatch,YBatch] = loadBatchAsFourDimensionalArray(location,
       batchFileName)
94 s = load(fullfile(location,batchFileName));
95 XBatch = s.data';
96 XBatch = reshape(XBatch,32,32,3,[]);
97 XBatch = permute(XBatch,[2 1 3 4]);
98 YBatch = convertLabelsToCategorical(location,s.labels);
99 end
100
101 function categoricalLabels = convertLabelsToCategorical(location,
       integerLabels)
102 s = load(fullfile(location,'batches.meta.mat'));
103 categoricalLabels = categorical(integerLabels,0:9,s.label_names);
104 end
```

Code to test model v1 of Experiment 1 (resnet_18_v1.mat) on FGSM data:

```
1 datadir = "/home/home02/mm23rn/resnet_18_v1";
2 downloadCIFARData(datadir);
3
4 [~,~,XValidation,TValidation] = loadCIFARData(datadir);
5
6 classes = categories(TValidation);
7
8 imageSize = [224 224 3];
9
10 augimdsVal = augmentedImageDatastore(imageSize,XValidation,TValidation);
11
12 load("resnet_18_v1.mat", "net");
13
14 miniBatchSize = 512;
15
16 mbqVal = minibatchqueue(augimdsVal, ...
```

```
    MiniBatchSize=miniBatchSize,...
    MiniBatchFcn=@preprocessMiniBatch,...
    MiniBatchFormat=["SSCB",""]);

epsilon = 2;
numAdvIter = 1;
alpha = epsilon;

[~,YPredAdv] = adversarialExamples(net,mbqVal,epsilon,alpha,numAdvIter,
    classes);
accAdv = mean(YPredAdv == TValidation);

fileID = fopen('adv_acc_resnet_18_v1.txt', 'w');

val_acc = "Validation accuracy (FGSM attack): " + accAdv*100;

fprintf(fileID, val_acc);

quit;

%--------------------------SUPPORTING FUNCTIONS
    ---------------------------------------

function gradient = modelGradientsInput(net,X,T)

T = squeeze(T);
T = dlarray(T,'CB');

[YPred] = forward(net,X);

loss = crossentropy(YPred,T);
gradient = dlgradient(loss,X);

end

function [X,T] = preprocessMiniBatch(XCell,TCell)

% Concatenate.
X = cat(4,XCell{1:end});

X = single(X);

% Extract label data from the cell and concatenate.
T = cat(2,TCell{1:end});

% One-hot encode labels.
T = onehotencode(T,1);

end
```

```
64
65 function XAdv = basicIterativeMethod(net,X,T,alpha,epsilon,numIter,
       initialization)
66
67 % Initialize the perturbation.
68 if initialization == "zero"
69     delta = zeros(size(X),like=X);
70 else
71     delta = epsilon*(2*rand(size(X),like=X) - 1);
72 end
73
74 for i = 1:numIter
75
76     % Apply adversarial perturbations to the data.
77     gradient = dlfeval(@modelGradientsInput,net,X+delta,T);
78     delta = delta + alpha*sign(gradient);
79     delta(delta > epsilon) = epsilon;
80     delta(delta < -epsilon) = -epsilon;
81 end
82
83 XAdv = X + delta;
84
85 end
86
87 function [XAdv,predictions] = adversarialExamples(net,mbq,epsilon,alpha,
       numIter,classes)
88
89 XAdv = {};
90 predictions = [];
91 iteration = 0;
92
93 % Generate adversarial images for each mini-batch.
94 while hasdata(mbq)
95
96     iteration = iteration +1;
97     [X,T] = next(mbq);
98
99     initialization = "zero";
100
101    % Generate adversarial images.
102    XAdvMBQ = basicIterativeMethod(net,X,T,alpha,epsilon, ...
103        numIter,initialization);
104
105    % Predict the class of the adversarial images.
106    YPred = predict(net,XAdvMBQ);
107    YPred = onehotdecode(YPred,classes,1)';
108
109    XAdv{iteration} = XAdvMBQ;
110    predictions = [predictions; YPred];
```

```
end

% Concatenate.
XAdv = cat(4,XAdv{:});

end

function downloadCIFARData(destination)

url = 'https://www.cs.toronto.edu/~kriz/cifar-10-matlab.tar.gz';

unpackedData = fullfile(destination,'cifar-10-batches-mat');
if ~exist(unpackedData,'dir')
    fprintf('Downloading CIFAR-10 dataset (175 MB). This can take a while ...');
    untar(url,destination);
    fprintf('done.\n\n');
end

end

function [XTrain,YTrain,XTest,YTest] = loadCIFARData(location)

location = fullfile(location,'cifar-10-batches-mat');

[XTrain1,YTrain1] = loadBatchAsFourDimensionalArray(location,'data_batch_1.mat');
[XTrain2,YTrain2] = loadBatchAsFourDimensionalArray(location,'data_batch_2.mat');
[XTrain3,YTrain3] = loadBatchAsFourDimensionalArray(location,'data_batch_3.mat');
[XTrain4,YTrain4] = loadBatchAsFourDimensionalArray(location,'data_batch_4.mat');
[XTrain5,YTrain5] = loadBatchAsFourDimensionalArray(location,'data_batch_5.mat');
XTrain = cat(4,XTrain1,XTrain2,XTrain3,XTrain4,XTrain5);
YTrain = [YTrain1;YTrain2;YTrain3;YTrain4;YTrain5];

[XTest,YTest] = loadBatchAsFourDimensionalArray(location,'test_batch.mat');
end

function [XBatch,YBatch] = loadBatchAsFourDimensionalArray(location,batchFileName)
s = load(fullfile(location,batchFileName));
XBatch = s.data';
XBatch = reshape(XBatch,32,32,3,[]);
XBatch = permute(XBatch,[2 1 3 4]);
YBatch = convertLabelsToCategorical(location,s.labels);
end
```

```

function categoricalLabels = convertLabelsToCategorical(location,
    integerLabels)
s = load(fullfile(location,'batches.meta.mat'));
categoricalLabels = categorical(integerLabels,0:9,s.label_names);
end
```

Code to test model v1 of Experiment 1 (resnet_18_v1.mat) on PGD data:

```
datadir = "/home/home02/mm23rn/resnet_18_v1";
downloadCIFARData(datadir);

[~,~,XValidation,TValidation] = loadCIFARData(datadir);

classes = categories(TValidation);

imageSize = [224 224 3];

augimdsVal = augmentedImageDatastore(imageSize,XValidation,TValidation);

load("resnet_18_v1.mat", "net");

miniBatchSize = 512;

mbqVal = minibatchqueue(augimdsVal, ...
    MiniBatchSize=miniBatchSize,...
    MiniBatchFcn=@preprocessMiniBatch,...
    MiniBatchFormat=["SSCB",""]);

epsilon = 8;
numAdvIter = 30;
alpha = epsilon;

[~,YPredAdv] = adversarialExamples(net,mbqVal,epsilon,alpha,numAdvIter,
    classes);
accAdv = mean(YPredAdv == TValidation);

fileID = fopen('pgd_acc_resnet_18_v1.txt', 'w');

fprintf(fileID, "PGD(Iteration = 30) Validation Report for ResNet-18 V1\n\
    nAlpha = Epsilon\n\n");

fprintf(fileID, "Epsilon: %f\t\t\t\t\t\t\t Validation accuracy (PGD): %f\n",
    epsilon, accAdv*100);

quit;

%-------------------------SUPPORTING FUNCTIONS
    ---------------------------------------

```

```
function gradient = modelGradientsInput(net,X,T)

T = squeeze(T);
T = dlarray(T,'CB');

[YPred] = forward(net,X);

loss = crossentropy(YPred,T);
gradient = dlgradient(loss,X);

end

function [X,T] = preprocessMiniBatch(XCell,TCell)

% Concatenate.
X = cat(4,XCell{1:end});

X = single(X);

% Extract label data from the cell and concatenate.
T = cat(2,TCell{1:end});

% One-hot encode labels.
T = onehotencode(T,1);

end

function XAdv = basicIterativeMethod(net,X,T,alpha,epsilon,numIter,
    initialization)

% Initialize the perturbation.
if initialization == "zero"
    delta = zeros(size(X),like=X);
else
    delta = epsilon*(2*rand(size(X),like=X) - 1);
end

for i = 1:numIter

    % Apply adversarial perturbations to the data.
    gradient = dlfeval(@modelGradientsInput,net,X+delta,T);
    delta = delta + alpha*sign(gradient);
    delta(delta > epsilon) = epsilon;
    delta(delta < -epsilon) = -epsilon;
end

XAdv = X + delta;

end
```

```
86
87 function [XAdv,predictions] = adversarialExamples(net,mbq,epsilon,alpha,
       numIter,classes)
88
89 XAdv = {};
90 predictions = [];
91 iteration = 0;
92
93 % Generate adversarial images for each mini-batch.
94 while hasdata(mbq)
95
96     iteration = iteration +1;
97     [X,T] = next(mbq);
98
99     initialization = "zero";
100
101    % Ustilize GPU if available
102     if canUseGPU
103         X = gpuArray(X);
104         T = gpuArray(T);
105     end
106
107    % Generate adversarial images.
108    XAdvMBQ = basicIterativeMethod(net,X,T,alpha,epsilon, ...
109        numIter,initialization);
110
111    % Predict the class of the adversarial images.
112    YPred = predict(net,XAdvMBQ);
113    YPred = onehotdecode(YPred,classes,1)';
114
115    XAdv{iteration} = XAdvMBQ;
116    predictions = [predictions; YPred];
117 end
118
119 % Concatenate.
120 XAdv = cat(4,XAdv{:});
121
122 end
123
124 function downloadCIFARData(destination)
125
126 url = 'https://www.cs.toronto.edu/~kriz/cifar-10-matlab.tar.gz';
127
128 unpackedData = fullfile(destination,'cifar-10-batches-mat');
129 if ~exist(unpackedData,'dir')
130     fprintf('Downloading CIFAR-10 dataset (175 MB). This can take a while
       ...');
131     untar(url,destination);
132     fprintf('done.\n\n');
```

```
end

end

function [XTrain,YTrain,XTest,YTest] = loadCIFARData(location)

location = fullfile(location,'cifar-10-batches-mat');

[XTrain1,YTrain1] = loadBatchAsFourDimensionalArray(location,'data_batch_1.
    mat');
[XTrain2,YTrain2] = loadBatchAsFourDimensionalArray(location,'data_batch_2.
    mat');
[XTrain3,YTrain3] = loadBatchAsFourDimensionalArray(location,'data_batch_3.
    mat');
[XTrain4,YTrain4] = loadBatchAsFourDimensionalArray(location,'data_batch_4.
    mat');
[XTrain5,YTrain5] = loadBatchAsFourDimensionalArray(location,'data_batch_5.
    mat');
XTrain = cat(4,XTrain1,XTrain2,XTrain3,XTrain4,XTrain5);
YTrain = [YTrain1;YTrain2;YTrain3;YTrain4;YTrain5];

[XTest,YTest] = loadBatchAsFourDimensionalArray(location,'test_batch.mat');
end

function [XBatch,YBatch] = loadBatchAsFourDimensionalArray(location,
    batchFileName)
s = load(fullfile(location,batchFileName));
XBatch = s.data';
XBatch = reshape(XBatch,32,32,3,[]);
XBatch = permute(XBatch,[2 1 3 4]);
YBatch = convertLabelsToCategorical(location,s.labels);
end

function categoricalLabels = convertLabelsToCategorical(location,
    integerLabels)
s = load(fullfile(location,'batches.meta.mat'));
categoricalLabels = categorical(integerLabels,0:9,s.label_names);
end
```

6.2 Experiment 2

Code to train model v2 of Experiment 2 (resnet_18_v2.mat):

```
datadir = "/home/home02/mm23rn/resnet_18_v1";
downloadCIFARData(datadir);

[XTrain,TTrain,~,~] = loadCIFARData(datadir);

```

```
imageSize = [224 224 3];
pixelRange = [-4 4];

imageAugmenter = imageDataAugmenter( ...
    RandXReflection=true, ...
    RandXTranslation=pixelRange, ...
    RandYTranslation=pixelRange);

augimdsTrain = augmentedImageDatastore(imageSize,XTrain,TTrain, ...
    DataAugmentation=imageAugmenter);

% Define the network layers without the classification and softmax layers
lgraph = layerGraph();

tempLayers = [
    imageInputLayer([224 224 3],"Name","data","Normalization","zscore",...
    "Mean", 0, "StandardDeviation", 1)
    convolution2dLayer([7 7],64,"Name","conv1","BiasLearnRateFactor",0,"
    Padding",[3 3 3 3],"Stride",[2 2])
    batchNormalizationLayer("Name","bn_conv1")
    reluLayer("Name","conv1_relu")
    maxPooling2dLayer([3 3],"Name","pool1","Padding",[1 1 1 1],"Stride",[2
    2])];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    convolution2dLayer([3 3],64,"Name","res2a_branch2a","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn2a_branch2a")
    reluLayer("Name","res2a_branch2a_relu")
    convolution2dLayer([3 3],64,"Name","res2a_branch2b","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn2a_branch2b")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    additionLayer(2,"Name","res2a")
    reluLayer("Name","res2a_relu")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    convolution2dLayer([3 3],64,"Name","res2b_branch2a","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn2b_branch2a")
    reluLayer("Name","res2b_branch2a_relu")
    convolution2dLayer([3 3],64,"Name","res2b_branch2b","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn2b_branch2b")];
lgraph = addLayers(lgraph,tempLayers);
```

```
49
50 tempLayers = [
51     additionLayer(2,"Name","res2b")
52     reluLayer("Name","res2b_relu")];
53 lgraph = addLayers(lgraph,tempLayers);
54
55 tempLayers = [
56     convolution2dLayer([1 1],128,"Name","res3a_branch1","BiasLearnRateFactor
       ",0,"Stride",[2 2])
57     batchNormalizationLayer("Name","bn3a_branch1")];
58 lgraph = addLayers(lgraph,tempLayers);
59
60 tempLayers = [
61     convolution2dLayer([3 3],128,"Name","res3a_branch2a","BiasLearnRateFactor
       ",0,"Padding",[1 1 1 1],"Stride",[2 2])
62     batchNormalizationLayer("Name","bn3a_branch2a")
63     reluLayer("Name","res3a_branch2a_relu")
64     convolution2dLayer([3 3],128,"Name","res3a_branch2b","BiasLearnRateFactor
       ",0,"Padding",[1 1 1 1])
65     batchNormalizationLayer("Name","bn3a_branch2b")];
66 lgraph = addLayers(lgraph,tempLayers);
67
68 tempLayers = [
69     additionLayer(2,"Name","res3a")
70     reluLayer("Name","res3a_relu")];
71 lgraph = addLayers(lgraph,tempLayers);
72
73 tempLayers = [
74     convolution2dLayer([3 3],128,"Name","res3b_branch2a","BiasLearnRateFactor
       ",0,"Padding",[1 1 1 1])
75     batchNormalizationLayer("Name","bn3b_branch2a")
76     reluLayer("Name","res3b_branch2a_relu")
77     convolution2dLayer([3 3],128,"Name","res3b_branch2b","BiasLearnRateFactor
       ",0,"Padding",[1 1 1 1])
78     batchNormalizationLayer("Name","bn3b_branch2b")];
79 lgraph = addLayers(lgraph,tempLayers);
80
81 tempLayers = [
82     additionLayer(2,"Name","res3b")
83     reluLayer("Name","res3b_relu")];
84 lgraph = addLayers(lgraph,tempLayers);
85
86 tempLayers = [
87     convolution2dLayer([1 1],256,"Name","res4a_branch1","BiasLearnRateFactor
       ",0,"Stride",[2 2])
88     batchNormalizationLayer("Name","bn4a_branch1")];
89 lgraph = addLayers(lgraph,tempLayers);
90
91 tempLayers = [
```

```
    convolution2dLayer([3 3],256,"Name","res4a_branch2a","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1],"Stride",[2 2])
    batchNormalizationLayer("Name","bn4a_branch2a")
    reluLayer("Name","res4a_branch2a_relu")
    convolution2dLayer([3 3],256,"Name","res4a_branch2b","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn4a_branch2b")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    additionLayer(2,"Name","res4a")
    reluLayer("Name","res4a_relu")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    convolution2dLayer([3 3],256,"Name","res4b_branch2a","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn4b_branch2a")
    reluLayer("Name","res4b_branch2a_relu")
    convolution2dLayer([3 3],256,"Name","res4b_branch2b","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn4b_branch2b")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    additionLayer(2,"Name","res4b")
    reluLayer("Name","res4b_relu")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    convolution2dLayer([1 1],512,"Name","res5a_branch1","BiasLearnRateFactor
    ",0,"Stride",[2 2])
    batchNormalizationLayer("Name","bn5a_branch1")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    convolution2dLayer([3 3],512,"Name","res5a_branch2a","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1],"Stride",[2 2])
    batchNormalizationLayer("Name","bn5a_branch2a")
    reluLayer("Name","res5a_branch2a_relu")
    convolution2dLayer([3 3],512,"Name","res5a_branch2b","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn5a_branch2b")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    additionLayer(2,"Name","res5a")
    reluLayer("Name","res5a_relu")];
lgraph = addLayers(lgraph,tempLayers);
```

```
134
135 tempLayers = [
136     convolution2dLayer([3 3],512,"Name","res5b_branch2a","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
137     batchNormalizationLayer("Name","bn5b_branch2a")
138     reluLayer("Name","res5b_branch2a_relu")
139     convolution2dLayer([3 3],512,"Name","res5b_branch2b","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
140     batchNormalizationLayer("Name","bn5b_branch2b")];
141 lgraph = addLayers(lgraph,tempLayers);
142
143 tempLayers = [
144     additionLayer(2,"Name","res5b")
145     reluLayer("Name","res5b_relu")
146     globalAveragePooling2dLayer("Name","pool5")
147     fullyConnectedLayer(10,"Name","fc10")
148     softmaxLayer("Name","prob")];
149 lgraph = addLayers(lgraph,tempLayers);
150
151 % clean up helper variable
152 clear tempLayers;
153
154 lgraph = connectLayers(lgraph,"pool1","res2a_branch2a");
155 lgraph = connectLayers(lgraph,"pool1","res2a/in2");
156 lgraph = connectLayers(lgraph,"bn2a_branch2b","res2a/in1");
157 lgraph = connectLayers(lgraph,"res2a_relu","res2b_branch2a");
158 lgraph = connectLayers(lgraph,"res2a_relu","res2b/in2");
159 lgraph = connectLayers(lgraph,"bn2b_branch2b","res2b/in1");
160 lgraph = connectLayers(lgraph,"res2b_relu","res3a_branch1");
161 lgraph = connectLayers(lgraph,"res2b_relu","res3a_branch2a");
162 lgraph = connectLayers(lgraph,"bn3a_branch1","res3a/in2");
163 lgraph = connectLayers(lgraph,"bn3a_branch2b","res3a/in1");
164 lgraph = connectLayers(lgraph,"res3a_relu","res3b_branch2a");
165 lgraph = connectLayers(lgraph,"res3a_relu","res3b/in2");
166 lgraph = connectLayers(lgraph,"bn3b_branch2b","res3b/in1");
167 lgraph = connectLayers(lgraph,"res3b_relu","res4a_branch1");
168 lgraph = connectLayers(lgraph,"res3b_relu","res4a_branch2a");
169 lgraph = connectLayers(lgraph,"bn4a_branch2b","res4a/in1");
170 lgraph = connectLayers(lgraph,"bn4a_branch1","res4a/in2");
171 lgraph = connectLayers(lgraph,"res4a_relu","res4b_branch2a");
172 lgraph = connectLayers(lgraph,"res4a_relu","res4b/in2");
173 lgraph = connectLayers(lgraph,"bn4b_branch2b","res4b/in1");
174 lgraph = connectLayers(lgraph,"res4b_relu","res5a_branch1");
175 lgraph = connectLayers(lgraph,"res4b_relu","res5a_branch2a");
176 lgraph = connectLayers(lgraph,"bn5a_branch1","res5a/in2");
177 lgraph = connectLayers(lgraph,"bn5a_branch2b","res5a/in1");
178 lgraph = connectLayers(lgraph,"res5a_relu","res5b_branch2a");
179 lgraph = connectLayers(lgraph,"res5a_relu","res5b/in2");
180 lgraph = connectLayers(lgraph,"bn5b_branch2b","res5b/in1");
```

```
181
182 % Initialize dlnetwork
183 netFGSM = dlnetwork(lgraph);
184 netFGSM = initialize(netFGSM);
185
186 numEpochs = 100;
187 miniBatchSize = 128;
188 learnRate = 0.01;
189
190 epsilon = 2;
191 numIter = 1;
192 initialization = "random";
193 alpha = epsilon;
194
195 mbq = minibatchqueue(augimdsTrain, ...
196     MiniBatchSize=miniBatchSize,...
197     MiniBatchFcn=@preprocessMiniBatch,...
198     MiniBatchFormat=["SSCB",""]);
199
200 % Open a text file for writing metrics
201 fileID = fopen('metrics_log_resnet_18_v2.txt', 'w');
202 fprintf(fileID, 'Epoch\tIteration\tLoss\n'); % Header for the log file
203
204 velocity = [];
205 epoch = 0;
206 iteration = 0;
207
208 % Loop over epochs.
209 while epoch < numEpochs
210     epoch = epoch + 1;
211
212     % Shuffle data.
213     shuffle(mbq)
214
215     % Loop over mini-batches.
216     while hasdata(mbq)
217         iteration = iteration + 1;
218
219         % Read mini-batch of data.
220         [X,T] = next(mbq);
221
222         % If training on a GPU, then convert data to gpuArray.
223         if canUseGPU
224             X = gpuArray(X);
225             T = gpuArray(T);
226         end
227
228         % Apply adversarial perturbations to the data.
229         X = basicIterativeMethod(netFGSM,X,T,alpha,epsilon, ...
```

```
            numIter,initialization);

        % Evaluate the model loss, gradients, and state.
        [loss,gradients,state] = dlfeval(@modelLoss,netFGSM,X,T);
        netFGSM.State = state;

        % Update the network parameters using the SGDM optimizer.
        [netFGSM,velocity] = sgdmupdate(netFGSM,gradients,velocity,learnRate)
    ;

        % Write the metrics to the text file
        fprintf(fileID, '%d\t%d\t%.4f\n', epoch, iteration, loss);
    end
end

save("resnet_18_v2.mat", "netFGSM");

fclose(fileID);

quit;

%-------------------------SUPPORTING FUNCTIONS
    --------------------------------------

function [loss,gradients,state] = modelLoss(net,X,T)

[YPred,state] = forward(net,X);

loss = crossentropy(YPred,T);
gradients = dlgradient(loss,net.Learnables);

loss = double(loss);

end

function gradient = modelGradientsInput(net,X,T)

T = squeeze(T);
T = dlarray(T,'CB');

[YPred] = forward(net,X);

loss = crossentropy(YPred,T);
gradient = dlgradient(loss,X);

end

function [X,T] = preprocessMiniBatch(XCell,TCell)

```

```
% Concatenate.
X = cat(4,XCell{1:end});

X = single(X);

% Extract label data from the cell and concatenate.
T = cat(2,TCell{1:end});

% One-hot encode labels.
T = onehotencode(T,1);

end

function XAdv = basicIterativeMethod(net,X,T,alpha,epsilon,numIter,
    initialization)

% Initialize the perturbation.
if initialization == "zero"
    delta = zeros(size(X),like=X);
else
    delta = epsilon*(2*rand(size(X),like=X) - 1);
end

for i = 1:numIter

    % Apply adversarial perturbations to the data.
    gradient = dlfeval(@modelGradientsInput,net,X+delta,T);
    delta = delta + alpha*sign(gradient);
    delta(delta > epsilon) = epsilon;
    delta(delta < -epsilon) = -epsilon;
end

XAdv = X + delta;

end

function downloadCIFARData(destination)

url = 'https://www.cs.toronto.edu/~kriz/cifar-10-matlab.tar.gz';

unpackedData = fullfile(destination,'cifar-10-batches-mat');
if ~exist(unpackedData,'dir')
    fprintf('Downloading CIFAR-10 dataset (175 MB). This can take a while
    ...');
    untar(url,destination);
    fprintf('done.\n\n');
end

end
```

```
function [XTrain,YTrain,XTest,YTest] = loadCIFARData(location)

location = fullfile(location,'cifar-10-batches-mat');

[XTrain1,YTrain1] = loadBatchAsFourDimensionalArray(location,'data_batch_1.
    mat');
[XTrain2,YTrain2] = loadBatchAsFourDimensionalArray(location,'data_batch_2.
    mat');
[XTrain3,YTrain3] = loadBatchAsFourDimensionalArray(location,'data_batch_3.
    mat');
[XTrain4,YTrain4] = loadBatchAsFourDimensionalArray(location,'data_batch_4.
    mat');
[XTrain5,YTrain5] = loadBatchAsFourDimensionalArray(location,'data_batch_5.
    mat');
XTrain = cat(4,XTrain1,XTrain2,XTrain3,XTrain4,XTrain5);
YTrain = [YTrain1;YTrain2;YTrain3;YTrain4;YTrain5];

[XTest,YTest] = loadBatchAsFourDimensionalArray(location,'test_batch.mat');
end

function [XBatch,YBatch] = loadBatchAsFourDimensionalArray(location,
    batchFileName)
s = load(fullfile(location,batchFileName));
XBatch = s.data';
XBatch = reshape(XBatch,32,32,3,[]);
XBatch = permute(XBatch,[2 1 3 4]);
YBatch = convertLabelsToCategorical(location,s.labels);
end

function categoricalLabels = convertLabelsToCategorical(location,
    integerLabels)
s = load(fullfile(location,'batches.meta.mat'));
categoricalLabels = categorical(integerLabels,0:9,s.label_names);
end
```

Code to test model v2 of Experiment 2 (resnet_18_v2.mat) on normal data:

```
datadir = "/home/home02/mm23rn/resnet_18_v1";
downloadCIFARData(datadir);

[~,~,XValidation,TValidation] = loadCIFARData(datadir);

classes = categories(TValidation);

imageSize = [224 224 3];

augimdsVal = augmentedImageDatastore(imageSize,XValidation,TValidation);

load("resnet_18_v2.mat", "netFGSM");
```

```
13
14 miniBatchSize = 512;
15
16 mbqVal = minibatchqueue(augimdsVal, ...
17     MiniBatchSize=miniBatchSize,...
18     MiniBatchFcn=@preprocessMiniBatch,...
19     MiniBatchFormat=["SSCB",""]);
20
21 YPred = modelPredictions(netFGSM,mbqVal,classes);
22 acc = mean(YPred == TValidation);
23
24 fileID = fopen('nrml_acc_resnet_18_v2.txt', 'w');
25
26 val_acc = "Validation accuracy: " + acc*100 + "%";
27
28 fprintf(fileID, val_acc);
29
30 quit;
31
32 %--------------------------SUPPORTING FUNCTIONS
       ---------------------------------------
33
34 function [X,T] = preprocessMiniBatch(XCell,TCell)
35
36 % Concatenate.
37 X = cat(4,XCell{1:end});
38
39 X = single(X);
40
41 % Extract label data from the cell and concatenate.
42 T = cat(2,TCell{1:end});
43
44 % One-hot encode labels.
45 T = onehotencode(T,1);
46
47 end
48
49 function predictions = modelPredictions(net,mbq,classes)
50
51 predictions = [];
52
53 while hasdata(mbq)
54
55     XTest = next(mbq);
56     YPred = predict(net,XTest);
57
58     YPred = onehotdecode(YPred,classes,1)';
59
60     predictions = [predictions; YPred];
```

```
end

end

function downloadCIFARData(destination)

url = 'https://www.cs.toronto.edu/~kriz/cifar-10-matlab.tar.gz';

unpackedData = fullfile(destination,'cifar-10-batches-mat');
if ~exist(unpackedData,'dir')
    fprintf('Downloading CIFAR-10 dataset (175 MB). This can take a while
    ...');
    untar(url,destination);
    fprintf('done.\n\n');
end

end

function [XTrain,YTrain,XTest,YTest] = loadCIFARData(location)

location = fullfile(location,'cifar-10-batches-mat');

[XTrain1,YTrain1] = loadBatchAsFourDimensionalArray(location,'data_batch_1.
    mat');
[XTrain2,YTrain2] = loadBatchAsFourDimensionalArray(location,'data_batch_2.
    mat');
[XTrain3,YTrain3] = loadBatchAsFourDimensionalArray(location,'data_batch_3.
    mat');
[XTrain4,YTrain4] = loadBatchAsFourDimensionalArray(location,'data_batch_4.
    mat');
[XTrain5,YTrain5] = loadBatchAsFourDimensionalArray(location,'data_batch_5.
    mat');
XTrain = cat(4,XTrain1,XTrain2,XTrain3,XTrain4,XTrain5);
YTrain = [YTrain1;YTrain2;YTrain3;YTrain4;YTrain5];

[XTest,YTest] = loadBatchAsFourDimensionalArray(location,'test_batch.mat');
end

function [XBatch,YBatch] = loadBatchAsFourDimensionalArray(location,
    batchFileName)
s = load(fullfile(location,batchFileName));
XBatch = s.data';
XBatch = reshape(XBatch,32,32,3,[]);
XBatch = permute(XBatch,[2 1 3 4]);
YBatch = convertLabelsToCategorical(location,s.labels);
end

function categoricalLabels = convertLabelsToCategorical(location,
    integerLabels)
```

```
s = load(fullfile(location,'batches.meta.mat'));
categoricalLabels = categorical(integerLabels,0:9,s.label_names);
end
```

Code to test model v2 of Experiment 2 (resnet_18_v2.mat) on FGSM data:

```
datadir = "/home/home02/mm23rn/resnet_18_v1";
downloadCIFARData(datadir);

[~,~,XValidation,TValidation] = loadCIFARData(datadir);

classes = categories(TValidation);

imageSize = [224 224 3];

augimdsVal = augmentedImageDatastore(imageSize,XValidation,TValidation);

load("resnet_18_v2.mat", "netFGSM");

miniBatchSize = 512;

mbqVal = minibatchqueue(augimdsVal, ...
    MiniBatchSize=miniBatchSize,...
    MiniBatchFcn=@preprocessMiniBatch,...
    MiniBatchFormat=["SSCB",""]);

epsilon = 2;
numAdvIter = 1;
alpha = epsilon;

[~,YPredAdv] = adversarialExamples(netFGSM,mbqVal,epsilon,alpha,numAdvIter,
    classes);
accAdv = mean(YPredAdv == TValidation);

fileID = fopen('adv_acc_resnet_18_v2.txt', 'w');

val_acc = "Validation accuracy (FGSM attack): " + accAdv*100;

fprintf(fileID, val_acc);

quit;

%-------------------------SUPPORTING FUNCTIONS
    ---------------------------------------

function gradient = modelGradientsInput(net,X,T)

T = squeeze(T);
T = dlarray(T,'CB');

```

```
[YPred] = forward(net,X);

loss = crossentropy(YPred,T);
gradient = dlgradient(loss,X);

end

function [X,T] = preprocessMiniBatch(XCell,TCell)

% Concatenate.
X = cat(4,XCell{1:end});

X = single(X);

% Extract label data from the cell and concatenate.
T = cat(2,TCell{1:end});

% One-hot encode labels.
T = onehotencode(T,1);

end

function XAdv = basicIterativeMethod(net,X,T,alpha,epsilon,numIter,
    initialization)

% Initialize the perturbation.
if initialization == "zero"
    delta = zeros(size(X),like=X);
else
    delta = epsilon*(2*rand(size(X),like=X) - 1);
end

for i = 1:numIter

    % Apply adversarial perturbations to the data.
    gradient = dlfeval(@modelGradientsInput,net,X+delta,T);
    delta = delta + alpha*sign(gradient);
    delta(delta > epsilon) = epsilon;
    delta(delta < -epsilon) = -epsilon;
end

XAdv = X + delta;

end

function [XAdv,predictions] = adversarialExamples(net,mbq,epsilon,alpha,
    numIter,classes)

XAdv = {};
```

```
predictions = [];
iteration = 0;

% Generate adversarial images for each mini-batch.
while hasdata(mbq)

    iteration = iteration +1;
    [X,T] = next(mbq);

    initialization = "zero";

    % Generate adversarial images.
    XAdvMBQ = basicIterativeMethod(net,X,T,alpha,epsilon, ...
        numIter,initialization);

    % Predict the class of the adversarial images.
    YPred = predict(net,XAdvMBQ);
    YPred = onehotdecode(YPred,classes,1)';

    XAdv{iteration} = XAdvMBQ;
    predictions = [predictions; YPred];
end

% Concatenate.
XAdv = cat(4,XAdv{:});

end

function downloadCIFARData(destination)

url = 'https://www.cs.toronto.edu/~kriz/cifar-10-matlab.tar.gz';

unpackedData = fullfile(destination,'cifar-10-batches-mat');
if ~exist(unpackedData,'dir')
    fprintf('Downloading CIFAR-10 dataset (175 MB). This can take a while
    ...');
    untar(url,destination);
    fprintf('done.\n\n');
end

end

function [XTrain,YTrain,XTest,YTest] = loadCIFARData(location)

location = fullfile(location,'cifar-10-batches-mat');

[XTrain1,YTrain1] = loadBatchAsFourDimensionalArray(location,'data_batch_1.
    mat');
[XTrain2,YTrain2] = loadBatchAsFourDimensionalArray(location,'data_batch_2.
```

```
    mat');
[XTrain3,YTrain3] = loadBatchAsFourDimensionalArray(location,'data_batch_3.
    mat');
[XTrain4,YTrain4] = loadBatchAsFourDimensionalArray(location,'data_batch_4.
    mat');
[XTrain5,YTrain5] = loadBatchAsFourDimensionalArray(location,'data_batch_5.
    mat');
XTrain = cat(4,XTrain1,XTrain2,XTrain3,XTrain4,XTrain5);
YTrain = [YTrain1;YTrain2;YTrain3;YTrain4;YTrain5];

[XTest,YTest] = loadBatchAsFourDimensionalArray(location,'test_batch.mat');
end

function [XBatch,YBatch] = loadBatchAsFourDimensionalArray(location,
    batchFileName)
s = load(fullfile(location,batchFileName));
XBatch = s.data';
XBatch = reshape(XBatch,32,32,3,[]);
XBatch = permute(XBatch,[2 1 3 4]);
YBatch = convertLabelsToCategorical(location,s.labels);
end

function categoricalLabels = convertLabelsToCategorical(location,
    integerLabels)
s = load(fullfile(location,'batches.meta.mat'));
categoricalLabels = categorical(integerLabels,0:9,s.label_names);
end
```

Code to test model v2 of Experiment 2 (resnet_18_v2.mat) on PGD data:

```
datadir = "/home/home02/mm23rn/resnet_18_v1";
downloadCIFARData(datadir);

[~,~,XValidation,TValidation] = loadCIFARData(datadir);

classes = categories(TValidation);

imageSize = [224 224 3];

augimdsVal = augmentedImageDatastore(imageSize,XValidation,TValidation);

load("resnet_18_v2.mat", "netFGSM");

miniBatchSize = 512;

mbqVal = minibatchqueue(augimdsVal, ...
    MiniBatchSize=miniBatchSize,...
    MiniBatchFcn=@preprocessMiniBatch,...
    MiniBatchFormat=["SSCB",""]);

```

```
epsilon = 8;
numAdvIter = 30;
alpha = epsilon;

[~,YPredAdv] = adversarialExamples(netFGSM,mbqVal,epsilon,alpha,numAdvIter,
    classes);
accAdv = mean(YPredAdv == TValidation);

fileID = fopen('pgd_acc_resnet_18_v2.txt', 'w');

fprintf(fileID, "PGD(Iteration = 30) Validation Report for ResNet-18 V2\n\
    nAlpha = Epsilon\n\n");

fprintf(fileID, "Epsilon: %f\t\t\t\t\t\t\t Validation accuracy (PGD): %f\n",
    epsilon, accAdv*100);

quit;

%--------------------------SUPPORTING FUNCTIONS
    ----------------------------------------

function gradient = modelGradientsInput(net,X,T)

T = squeeze(T);
T = dlarray(T,'CB');

[YPred] = forward(net,X);

loss = crossentropy(YPred,T);
gradient = dlgradient(loss,X);

end

function [X,T] = preprocessMiniBatch(XCell,TCell)

% Concatenate.
X = cat(4,XCell{1:end});

X = single(X);

% Extract label data from the cell and concatenate.
T = cat(2,TCell{1:end});

% One-hot encode labels.
T = onehotencode(T,1);

end

function XAdv = basicIterativeMethod(net,X,T,alpha,epsilon,numIter,
```

```
    initialization)

% Initialize the perturbation.
if initialization == "zero"
    delta = zeros(size(X),like=X);
else
    delta = epsilon*(2*rand(size(X),like=X) - 1);
end

for i = 1:numIter

    % Apply adversarial perturbations to the data.
    gradient = dlfeval(@modelGradientsInput,net,X+delta,T);
    delta = delta + alpha*sign(gradient);
    delta(delta > epsilon) = epsilon;
    delta(delta < -epsilon) = -epsilon;
end

XAdv = X + delta;

end

function [XAdv,predictions] = adversarialExamples(net,mbq,epsilon,alpha,
    numIter,classes)

XAdv = {};
predictions = [];
iteration = 0;

% Generate adversarial images for each mini-batch.
while hasdata(mbq)

    iteration = iteration +1;
    [X,T] = next(mbq);

    initialization = "zero";

    % Ustilize GPU if available
     if canUseGPU
         X = gpuArray(X);
         T = gpuArray(T);
     end

    % Generate adversarial images.
    XAdvMBQ = basicIterativeMethod(net,X,T,alpha,epsilon, ...
        numIter,initialization);

    % Predict the class of the adversarial images.
    YPred = predict(net,XAdvMBQ);
```

```
    YPred = onehotdecode(YPred,classes,1)';

    XAdv{iteration} = XAdvMBQ;
    predictions = [predictions; YPred];
end

% Concatenate.
XAdv = cat(4,XAdv{:});

end

function downloadCIFARData(destination)

url = 'https://www.cs.toronto.edu/~kriz/cifar-10-matlab.tar.gz';

unpackedData = fullfile(destination,'cifar-10-batches-mat');
if ~exist(unpackedData,'dir')
    fprintf('Downloading CIFAR-10 dataset (175 MB). This can take a while
    ...');
    untar(url,destination);
    fprintf('done.\n\n');
end

end

function [XTrain,YTrain,XTest,YTest] = loadCIFARData(location)

location = fullfile(location,'cifar-10-batches-mat');

[XTrain1,YTrain1] = loadBatchAsFourDimensionalArray(location,'data_batch_1.
    mat');
[XTrain2,YTrain2] = loadBatchAsFourDimensionalArray(location,'data_batch_2.
    mat');
[XTrain3,YTrain3] = loadBatchAsFourDimensionalArray(location,'data_batch_3.
    mat');
[XTrain4,YTrain4] = loadBatchAsFourDimensionalArray(location,'data_batch_4.
    mat');
[XTrain5,YTrain5] = loadBatchAsFourDimensionalArray(location,'data_batch_5.
    mat');
XTrain = cat(4,XTrain1,XTrain2,XTrain3,XTrain4,XTrain5);
YTrain = [YTrain1;YTrain2;YTrain3;YTrain4;YTrain5];

[XTest,YTest] = loadBatchAsFourDimensionalArray(location,'test_batch.mat');
end

function [XBatch,YBatch] = loadBatchAsFourDimensionalArray(location,
    batchFileName)
s = load(fullfile(location,batchFileName));
XBatch = s.data';
```

```
XBatch = reshape(XBatch,32,32,3,[]);
XBatch = permute(XBatch,[2 1 3 4]);
YBatch = convertLabelsToCategorical(location,s.labels);
end

function categoricalLabels = convertLabelsToCategorical(location,
    integerLabels)
s = load(fullfile(location,'batches.meta.mat'));
categoricalLabels = categorical(integerLabels,0:9,s.label_names);
end
```

6.3 Experiment 3

Code to train model v3 of Experiment 3 (resnet_18_v3.mat):

```
datadir = "/home/home02/mm23rn/resnet_18_v1";
downloadCIFARData(datadir);

[XTrain,TTrain,~,~] = loadCIFARData(datadir);

imageSize = [224 224 3];
pixelRange = [-4 4];

imageAugmenter = imageDataAugmenter( ...
    RandXReflection=true, ...
    RandXTranslation=pixelRange, ...
    RandYTranslation=pixelRange);

augimdsTrain = augmentedImageDatastore(imageSize,XTrain,TTrain, ...
    DataAugmentation=imageAugmenter);

% Define the network layers without the classification and softmax layers
lgraph = layerGraph();

tempLayers = [
    imageInputLayer([224 224 3],"Name","data","Normalization","zscore",...
    "Mean", 0, "StandardDeviation", 1)
    convolution2dLayer([7 7],64,"Name","conv1","BiasLearnRateFactor",0,"
    Padding",[3 3 3 3],"Stride",[2 2])
    batchNormalizationLayer("Name","bn_conv1")
    reluLayer("Name","conv1_relu")
    maxPooling2dLayer([3 3],"Name","pool1","Padding",[1 1 1 1],"Stride",[2
    2])];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    convolution2dLayer([3 3],64,"Name","res2a_branch2a","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
```

```
    batchNormalizationLayer("Name","bn2a_branch2a")
    reluLayer("Name","res2a_branch2a_relu")
    convolution2dLayer([3 3],64,"Name","res2a_branch2b","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn2a_branch2b")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    additionLayer(2,"Name","res2a")
    reluLayer("Name","res2a_relu")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    convolution2dLayer([3 3],64,"Name","res2b_branch2a","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn2b_branch2a")
    reluLayer("Name","res2b_branch2a_relu")
    convolution2dLayer([3 3],64,"Name","res2b_branch2b","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn2b_branch2b")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    additionLayer(2,"Name","res2b")
    reluLayer("Name","res2b_relu")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    convolution2dLayer([1 1],128,"Name","res3a_branch1","BiasLearnRateFactor
    ",0,"Stride",[2 2])
    batchNormalizationLayer("Name","bn3a_branch1")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    convolution2dLayer([3 3],128,"Name","res3a_branch2a","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1],"Stride",[2 2])
    batchNormalizationLayer("Name","bn3a_branch2a")
    reluLayer("Name","res3a_branch2a_relu")
    convolution2dLayer([3 3],128,"Name","res3a_branch2b","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn3a_branch2b")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    additionLayer(2,"Name","res3a")
    reluLayer("Name","res3a_relu")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
```

```
    convolution2dLayer([3 3],128,"Name","res3b_branch2a","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn3b_branch2a")
    reluLayer("Name","res3b_branch2a_relu")
    convolution2dLayer([3 3],128,"Name","res3b_branch2b","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn3b_branch2b")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    additionLayer(2,"Name","res3b")
    reluLayer("Name","res3b_relu")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    convolution2dLayer([1 1],256,"Name","res4a_branch1","BiasLearnRateFactor
    ",0,"Stride",[2 2])
    batchNormalizationLayer("Name","bn4a_branch1")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    convolution2dLayer([3 3],256,"Name","res4a_branch2a","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1],"Stride",[2 2])
    batchNormalizationLayer("Name","bn4a_branch2a")
    reluLayer("Name","res4a_branch2a_relu")
    convolution2dLayer([3 3],256,"Name","res4a_branch2b","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn4a_branch2b")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    additionLayer(2,"Name","res4a")
    reluLayer("Name","res4a_relu")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    convolution2dLayer([3 3],256,"Name","res4b_branch2a","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn4b_branch2a")
    reluLayer("Name","res4b_branch2a_relu")
    convolution2dLayer([3 3],256,"Name","res4b_branch2b","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
    batchNormalizationLayer("Name","bn4b_branch2b")];
lgraph = addLayers(lgraph,tempLayers);

tempLayers = [
    additionLayer(2,"Name","res4b")
    reluLayer("Name","res4b_relu")];
lgraph = addLayers(lgraph,tempLayers);
```

```
116
117 tempLayers = [
118     convolution2dLayer([1 1],512,"Name","res5a_branch1","BiasLearnRateFactor
    ",0,"Stride",[2 2])
119     batchNormalizationLayer("Name","bn5a_branch1")];
120 lgraph = addLayers(lgraph,tempLayers);
121
122 tempLayers = [
123     convolution2dLayer([3 3],512,"Name","res5a_branch2a","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1],"Stride",[2 2])
124     batchNormalizationLayer("Name","bn5a_branch2a")
125     reluLayer("Name","res5a_branch2a_relu")
126     convolution2dLayer([3 3],512,"Name","res5a_branch2b","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
127     batchNormalizationLayer("Name","bn5a_branch2b")];
128 lgraph = addLayers(lgraph,tempLayers);
129
130 tempLayers = [
131     additionLayer(2,"Name","res5a")
132     reluLayer("Name","res5a_relu")];
133 lgraph = addLayers(lgraph,tempLayers);
134
135 tempLayers = [
136     convolution2dLayer([3 3],512,"Name","res5b_branch2a","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
137     batchNormalizationLayer("Name","bn5b_branch2a")
138     reluLayer("Name","res5b_branch2a_relu")
139     convolution2dLayer([3 3],512,"Name","res5b_branch2b","BiasLearnRateFactor
    ",0,"Padding",[1 1 1 1])
140     batchNormalizationLayer("Name","bn5b_branch2b")];
141 lgraph = addLayers(lgraph,tempLayers);
142
143 tempLayers = [
144     additionLayer(2,"Name","res5b")
145     reluLayer("Name","res5b_relu")
146     globalAveragePooling2dLayer("Name","pool5")
147     fullyConnectedLayer(10,"Name","fc10")
148     softmaxLayer("Name","prob")];
149 lgraph = addLayers(lgraph,tempLayers);
150
151 % clean up helper variable
152 clear tempLayers;
153
154 lgraph = connectLayers(lgraph,"pool1","res2a_branch2a");
155 lgraph = connectLayers(lgraph,"pool1","res2a/in2");
156 lgraph = connectLayers(lgraph,"bn2a_branch2b","res2a/in1");
157 lgraph = connectLayers(lgraph,"res2a_relu","res2b_branch2a");
158 lgraph = connectLayers(lgraph,"res2a_relu","res2b/in2");
159 lgraph = connectLayers(lgraph,"bn2b_branch2b","res2b/in1");
```

```
lgraph = connectLayers(lgraph,"res2b_relu","res3a_branch1");
lgraph = connectLayers(lgraph,"res2b_relu","res3a_branch2a");
lgraph = connectLayers(lgraph,"bn3a_branch1","res3a/in2");
lgraph = connectLayers(lgraph,"bn3a_branch2b","res3a/in1");
lgraph = connectLayers(lgraph,"res3a_relu","res3b_branch2a");
lgraph = connectLayers(lgraph,"res3a_relu","res3b/in2");
lgraph = connectLayers(lgraph,"bn3b_branch2b","res3b/in1");
lgraph = connectLayers(lgraph,"res3b_relu","res4a_branch1");
lgraph = connectLayers(lgraph,"res3b_relu","res4a_branch2a");
lgraph = connectLayers(lgraph,"bn4a_branch2b","res4a/in1");
lgraph = connectLayers(lgraph,"bn4a_branch1","res4a/in2");
lgraph = connectLayers(lgraph,"res4a_relu","res4b_branch2a");
lgraph = connectLayers(lgraph,"res4a_relu","res4b/in2");
lgraph = connectLayers(lgraph,"bn4b_branch2b","res4b/in1");
lgraph = connectLayers(lgraph,"res4b_relu","res5a_branch1");
lgraph = connectLayers(lgraph,"res4b_relu","res5a_branch2a");
lgraph = connectLayers(lgraph,"bn5a_branch1","res5a/in2");
lgraph = connectLayers(lgraph,"bn5a_branch2b","res5a/in1");
lgraph = connectLayers(lgraph,"res5a_relu","res5b_branch2a");
lgraph = connectLayers(lgraph,"res5a_relu","res5b/in2");
lgraph = connectLayers(lgraph,"bn5b_branch2b","res5b/in1");

% Initialize dlnetwork
netMX = dlnetwork(lgraph);
netMX = initialize(netMX);

numEpochs = 100;
miniBatchSize = 128;
learnRate = 0.01;

epsilon = 2;
numIter = 1;
initialization = "random";
alpha = epsilon;

mbq = minibatchqueue(augimdsTrain, ...
    MiniBatchSize=miniBatchSize,...
    MiniBatchFcn=@preprocessMiniBatch,...
    MiniBatchFormat=["SSCB",""]);

% Open a text file for writing metrics
fileID = fopen('metrics_log_resnet_18_v3.txt', 'w');
fprintf(fileID, 'Epoch\tIteration\tLoss\n'); % Header for the log file

velocity = [];
epoch = 0;
iteration = 0;

% Loop over epochs.
```

```
while epoch < numEpochs
    epoch = epoch + 1;

    % Shuffle data.
    shuffle(mbq)

    % Loop over mini-batches.
    while hasdata(mbq)
        iteration = iteration + 1;

        % Read mini-batch of data.
        [X,T] = next(mbq);

        % If training on a GPU, then convert data to gpuArray.
        if canUseGPU
            X = gpuArray(X);
            T = gpuArray(T);
        end


        if epoch > 50
            % Apply adversarial perturbations to the data.
            X = basicIterativeMethod(netMX,X,T,alpha,epsilon, ...
                numIter,initialization);
        end

        % Evaluate the model loss, gradients, and state.
        [loss,gradients,state] = dlfeval(@modelLoss,netMX,X,T);
        netMX.State = state;

        % Update the network parameters using the SGDM optimizer.
        [netMX,velocity] = sgdmupdate(netMX,gradients,velocity,learnRate);

        % Write the metrics to the text file
        fprintf(fileID, '%d\t%d\t%.4f\n', epoch, iteration, loss);
    end
end

save("resnet_18_v3.mat", "netMX");

fclose(fileID);

quit;

%-------------------------SUPPORTING FUNCTIONS
    ----------------------------------------

function [loss,gradients,state] = modelLoss(net,X,T)

```

```
[YPred,state] = forward(net,X);

loss = crossentropy(YPred,T);
gradients = dlgradient(loss,net.Learnables);

loss = double(loss);

end

function gradient = modelGradientsInput(net,X,T)

T = squeeze(T);
T = dlarray(T,'CB');

[YPred] = forward(net,X);

loss = crossentropy(YPred,T);
gradient = dlgradient(loss,X);

end

function [X,T] = preprocessMiniBatch(XCell,TCell)

% Concatenate.
X = cat(4,XCell{1:end});

X = single(X);

% Extract label data from the cell and concatenate.
T = cat(2,TCell{1:end});

% One-hot encode labels.
T = onehotencode(T,1);

end

function XAdv = basicIterativeMethod(net,X,T,alpha,epsilon,numIter,
    initialization)

% Initialize the perturbation.
if initialization == "zero"
    delta = zeros(size(X),like=X);
else
    delta = epsilon*(2*rand(size(X),like=X) - 1);
end

for i = 1:numIter

    % Apply adversarial perturbations to the data.
```

```
    gradient = dlfeval(@modelGradientsInput,net,X+delta,T);
    delta = delta + alpha*sign(gradient);
    delta(delta > epsilon) = epsilon;
    delta(delta < -epsilon) = -epsilon;
end

XAdv = X + delta;

end

function downloadCIFARData(destination)

url = 'https://www.cs.toronto.edu/~kriz/cifar-10-matlab.tar.gz';

unpackedData = fullfile(destination,'cifar-10-batches-mat');
if ~exist(unpackedData,'dir')
    fprintf('Downloading CIFAR-10 dataset (175 MB). This can take a while
    ...');
    untar(url,destination);
    fprintf('done.\n\n');
end

end

function [XTrain,YTrain,XTest,YTest] = loadCIFARData(location)

location = fullfile(location,'cifar-10-batches-mat');

[XTrain1,YTrain1] = loadBatchAsFourDimensionalArray(location,'data_batch_1.
    mat');
[XTrain2,YTrain2] = loadBatchAsFourDimensionalArray(location,'data_batch_2.
    mat');
[XTrain3,YTrain3] = loadBatchAsFourDimensionalArray(location,'data_batch_3.
    mat');
[XTrain4,YTrain4] = loadBatchAsFourDimensionalArray(location,'data_batch_4.
    mat');
[XTrain5,YTrain5] = loadBatchAsFourDimensionalArray(location,'data_batch_5.
    mat');
XTrain = cat(4,XTrain1,XTrain2,XTrain3,XTrain4,XTrain5);
YTrain = [YTrain1;YTrain2;YTrain3;YTrain4;YTrain5];

[XTest,YTest] = loadBatchAsFourDimensionalArray(location,'test_batch.mat');
end

function [XBatch,YBatch] = loadBatchAsFourDimensionalArray(location,
    batchFileName)
s = load(fullfile(location,batchFileName));
XBatch = s.data';
XBatch = reshape(XBatch,32,32,3,[]);
```

```
XBatch = permute(XBatch,[2 1 3 4]);
YBatch = convertLabelsToCategorical(location,s.labels);
end

function categoricalLabels = convertLabelsToCategorical(location,
    integerLabels)
s = load(fullfile(location,'batches.meta.mat'));
categoricalLabels = categorical(integerLabels,0:9,s.label_names);
end
```

Code to test model v3 of Experiment 3 (resnet_18_v3.mat) on normal data:

```
datadir = "/home/home02/mm23rn/resnet_18_v1";
downloadCIFARData(datadir);

[~,~,XValidation,TValidation] = loadCIFARData(datadir);

classes = categories(TValidation);

imageSize = [224 224 3];

augimdsVal = augmentedImageDatastore(imageSize,XValidation,TValidation);

load("resnet_18_v3.mat", "netMX");

miniBatchSize = 512;

mbqVal = minibatchqueue(augimdsVal, ...
    MiniBatchSize=miniBatchSize,...
    MiniBatchFcn=@preprocessMiniBatch,...
    MiniBatchFormat=["SSCB",""]);

YPred = modelPredictions(netMX,mbqVal,classes);
acc = mean(YPred == TValidation);

fileID = fopen('nrml_acc_resnet_18_v3.txt', 'w');

val_acc = "Validation accuracy: " + acc*100 + "%";

fprintf(fileID, val_acc);

quit;

%--------------------------SUPPORTING FUNCTIONS
    ----------------------------------------

function [X,T] = preprocessMiniBatch(XCell,TCell)

% Concatenate.
X = cat(4,XCell{1:end});
```

```
38
39 X = single(X);
40
41 % Extract label data from the cell and concatenate.
42 T = cat(2,TCell{1:end});
43
44 % One-hot encode labels.
45 T = onehotencode(T,1);
46
47 end
48
49 function predictions = modelPredictions(net,mbq,classes)
50
51 predictions = [];
52
53 while hasdata(mbq)
54
55     XTest = next(mbq);
56     YPred = predict(net,XTest);
57
58     YPred = onehotdecode(YPred,classes,1)';
59
60     predictions = [predictions; YPred];
61 end
62
63 end
64
65 function downloadCIFARData(destination)
66
67 url = 'https://www.cs.toronto.edu/~kriz/cifar-10-matlab.tar.gz';
68
69 unpackedData = fullfile(destination,'cifar-10-batches-mat');
70 if ~exist(unpackedData,'dir')
71     fprintf('Downloading CIFAR-10 dataset (175 MB). This can take a while
    ...');
72     untar(url,destination);
73     fprintf('done.\n\n');
74 end
75
76 end
77
78 function [XTrain,YTrain,XTest,YTest] = loadCIFARData(location)
79
80 location = fullfile(location,'cifar-10-batches-mat');
81
82 [XTrain1,YTrain1] = loadBatchAsFourDimensionalArray(location,'data_batch_1.
    mat');
83 [XTrain2,YTrain2] = loadBatchAsFourDimensionalArray(location,'data_batch_2.
    mat');
```

```
[XTrain3,YTrain3] = loadBatchAsFourDimensionalArray(location,'data_batch_3.
    mat');
[XTrain4,YTrain4] = loadBatchAsFourDimensionalArray(location,'data_batch_4.
    mat');
[XTrain5,YTrain5] = loadBatchAsFourDimensionalArray(location,'data_batch_5.
    mat');
XTrain = cat(4,XTrain1,XTrain2,XTrain3,XTrain4,XTrain5);
YTrain = [YTrain1;YTrain2;YTrain3;YTrain4;YTrain5];

[XTest,YTest] = loadBatchAsFourDimensionalArray(location,'test_batch.mat');
end

function [XBatch,YBatch] = loadBatchAsFourDimensionalArray(location,
    batchFileName)
s = load(fullfile(location,batchFileName));
XBatch = s.data';
XBatch = reshape(XBatch,32,32,3,[]);
XBatch = permute(XBatch,[2 1 3 4]);
YBatch = convertLabelsToCategorical(location,s.labels);
end

function categoricalLabels = convertLabelsToCategorical(location,
    integerLabels)
s = load(fullfile(location,'batches.meta.mat'));
categoricalLabels = categorical(integerLabels,0:9,s.label_names);
end
```

Code to test model v3 of Experiment 3 (resnet_18_v3.mat) on FGSM data:

```
datadir = "/home/home02/mm23rn/resnet_18_v1";
downloadCIFARData(datadir);

[~,~,XValidation,TValidation] = loadCIFARData(datadir);

classes = categories(TValidation);

imageSize = [224 224 3];

augimdsVal = augmentedImageDatastore(imageSize,XValidation,TValidation);

load("resnet_18_v3.mat", "netMX");

miniBatchSize = 512;

mbqVal = minibatchqueue(augimdsVal, ...
    MiniBatchSize=miniBatchSize,...
    MiniBatchFcn=@preprocessMiniBatch,...
    MiniBatchFormat=["SSCB",""]);

epsilon = 2;
```

```
numAdvIter = 1;
alpha = epsilon;

[~,YPredAdv] = adversarialExamples(netMX,mbqVal,epsilon,alpha,numAdvIter,
    classes);
accAdv = mean(YPredAdv == TValidation);

fileID = fopen('adv_acc_resnet_18_v3.txt', 'w');

val_acc = "Validation accuracy (FGSM attack): " + accAdv*100 + "%";

fprintf(fileID, val_acc);

quit;

%--------------------------SUPPORTING FUNCTIONS
    ----------------------------------------

function gradient = modelGradientsInput(net,X,T)

T = squeeze(T);
T = dlarray(T,'CB');

[YPred] = forward(net,X);

loss = crossentropy(YPred,T);
gradient = dlgradient(loss,X);

end

function [X,T] = preprocessMiniBatch(XCell,TCell)

% Concatenate.
X = cat(4,XCell{1:end});

X = single(X);

% Extract label data from the cell and concatenate.
T = cat(2,TCell{1:end});

% One-hot encode labels.
T = onehotencode(T,1);

end

function XAdv = basicIterativeMethod(net,X,T,alpha,epsilon,numIter,
    initialization)

% Initialize the perturbation.
```

```
if initialization == "zero"
    delta = zeros(size(X),like=X);
else
    delta = epsilon*(2*rand(size(X),like=X) - 1);
end

for i = 1:numIter

    % Apply adversarial perturbations to the data.
    gradient = dlfeval(@modelGradientsInput,net,X+delta,T);
    delta = delta + alpha*sign(gradient);
    delta(delta > epsilon) = epsilon;
    delta(delta < -epsilon) = -epsilon;
end

XAdv = X + delta;

end

function [XAdv,predictions] = adversarialExamples(net,mbq,epsilon,alpha,
    numIter,classes)

XAdv = {};
predictions = [];
iteration = 0;

% Generate adversarial images for each mini-batch.
while hasdata(mbq)

    iteration = iteration +1;
    [X,T] = next(mbq);

    initialization = "zero";

    % Generate adversarial images.
    XAdvMBQ = basicIterativeMethod(net,X,T,alpha,epsilon, ...
        numIter,initialization);

    % Predict the class of the adversarial images.
    YPred = predict(net,XAdvMBQ);
    YPred = onehotdecode(YPred,classes,1)';

    XAdv{iteration} = XAdvMBQ;
    predictions = [predictions; YPred];
end

% Concatenate.
XAdv = cat(4,XAdv{:});

```

```
116 end
117
118 function downloadCIFARData(destination)
119
120 url = 'https://www.cs.toronto.edu/~kriz/cifar-10-matlab.tar.gz';
121
122 unpackedData = fullfile(destination,'cifar-10-batches-mat');
123 if ~exist(unpackedData,'dir')
124     fprintf('Downloading CIFAR-10 dataset (175 MB). This can take a while
    ...');
125     untar(url,destination);
126     fprintf('done.\n\n');
127 end
128
129 end
130
131 function [XTrain,YTrain,XTest,YTest] = loadCIFARData(location)
132
133 location = fullfile(location,'cifar-10-batches-mat');
134
135 [XTrain1,YTrain1] = loadBatchAsFourDimensionalArray(location,'data_batch_1.
    mat');
136 [XTrain2,YTrain2] = loadBatchAsFourDimensionalArray(location,'data_batch_2.
    mat');
137 [XTrain3,YTrain3] = loadBatchAsFourDimensionalArray(location,'data_batch_3.
    mat');
138 [XTrain4,YTrain4] = loadBatchAsFourDimensionalArray(location,'data_batch_4.
    mat');
139 [XTrain5,YTrain5] = loadBatchAsFourDimensionalArray(location,'data_batch_5.
    mat');
140 XTrain = cat(4,XTrain1,XTrain2,XTrain3,XTrain4,XTrain5);
141 YTrain = [YTrain1;YTrain2;YTrain3;YTrain4;YTrain5];
142
143 [XTest,YTest] = loadBatchAsFourDimensionalArray(location,'test_batch.mat');
144 end
145
146 function [XBatch,YBatch] = loadBatchAsFourDimensionalArray(location,
    batchFileName)
147 s = load(fullfile(location,batchFileName));
148 XBatch = s.data';
149 XBatch = reshape(XBatch,32,32,3,[]);
150 XBatch = permute(XBatch,[2 1 3 4]);
151 YBatch = convertLabelsToCategorical(location,s.labels);
152 end
153
154 function categoricalLabels = convertLabelsToCategorical(location,
    integerLabels)
155 s = load(fullfile(location,'batches.meta.mat'));
156 categoricalLabels = categorical(integerLabels,0:9,s.label_names);
```

```
end
```

Code to test model v3 of Experiment 3 (resnet_18_v3.mat) on PGD data:

```
datadir = "/home/home02/mm23rn/resnet_18_v1";
downloadCIFARData(datadir);

[~,~,XValidation,TValidation] = loadCIFARData(datadir);

classes = categories(TValidation);

imageSize = [224 224 3];

augimdsVal = augmentedImageDatastore(imageSize,XValidation,TValidation);

load("resnet_18_v3.mat", "netMX");

miniBatchSize = 512;

mbqVal = minibatchqueue(augimdsVal, ...
    MiniBatchSize=miniBatchSize,...
    MiniBatchFcn=@preprocessMiniBatch,...
    MiniBatchFormat=["SSCB",""]);

epsilon = 8;
numAdvIter = 30;
alpha = epsilon;

[~,YPredAdv] = adversarialExamples(netMX,mbqVal,epsilon,alpha,numAdvIter,
    classes);
accAdv = mean(YPredAdv == TValidation);

fileID = fopen('pgd_acc_resnet_18_v3.txt', 'w');

fprintf(fileID, "PGD(Iteration = 30) Validation Report for ResNet-18 V3\n\
    nAlpha = Epsilon\n\n");

fprintf(fileID, "Epsilon: %f\t\t\t\t\t\t\t Validation accuracy (PGD): %f\n",
    epsilon, accAdv*100);

quit;

%-------------------------SUPPORTING FUNCTIONS
    ---------------------------------------

function gradient = modelGradientsInput(net,X,T)

T = squeeze(T);
T = dlarray(T,'CB');

```

```
[YPred] = forward(net,X);

loss = crossentropy(YPred,T);
gradient = dlgradient(loss,X);

end

function [X,T] = preprocessMiniBatch(XCell,TCell)

% Concatenate.
X = cat(4,XCell{1:end});

X = single(X);

% Extract label data from the cell and concatenate.
T = cat(2,TCell{1:end});

% One-hot encode labels.
T = onehotencode(T,1);

end

function XAdv = basicIterativeMethod(net,X,T,alpha,epsilon,numIter,
    initialization)

% Initialize the perturbation.
if initialization == "zero"
    delta = zeros(size(X),like=X);
else
    delta = epsilon*(2*rand(size(X),like=X) - 1);
end

for i = 1:numIter

    % Apply adversarial perturbations to the data.
    gradient = dlfeval(@modelGradientsInput,net,X+delta,T);
    delta = delta + alpha*sign(gradient);
    delta(delta > epsilon) = epsilon;
    delta(delta < -epsilon) = -epsilon;
end

XAdv = X + delta;

end

function [XAdv,predictions] = adversarialExamples(net,mbq,epsilon,alpha,
    numIter,classes)

XAdv = {};
```

```
predictions = [];
iteration = 0;

% Generate adversarial images for each mini-batch.
while hasdata(mbq)

    iteration = iteration +1;
    [X,T] = next(mbq);

    initialization = "zero";

    % Ustilize GPU if available
     if canUseGPU
         X = gpuArray(X);
         T = gpuArray(T);
     end

    % Generate adversarial images.
    XAdvMBQ = basicIterativeMethod(net,X,T,alpha,epsilon, ...
        numIter,initialization);

    % Predict the class of the adversarial images.
    YPred = predict(net,XAdvMBQ);
    YPred = onehotdecode(YPred,classes,1)';

    XAdv{iteration} = XAdvMBQ;
    predictions = [predictions; YPred];
end

% Concatenate.
XAdv = cat(4,XAdv{:});

end

function downloadCIFARData(destination)

url = 'https://www.cs.toronto.edu/~kriz/cifar-10-matlab.tar.gz';

unpackedData = fullfile(destination,'cifar-10-batches-mat');
if ~exist(unpackedData,'dir')
    fprintf('Downloading CIFAR-10 dataset (175 MB). This can take a while
    ...');
    untar(url,destination);
    fprintf('done.\n\n');
end

end

function [XTrain,YTrain,XTest,YTest] = loadCIFARData(location)
```

```
138
139 location = fullfile(location,'cifar-10-batches-mat');
140
141 [XTrain1,YTrain1] = loadBatchAsFourDimensionalArray(location,'data_batch_1.
    mat');
142 [XTrain2,YTrain2] = loadBatchAsFourDimensionalArray(location,'data_batch_2.
    mat');
143 [XTrain3,YTrain3] = loadBatchAsFourDimensionalArray(location,'data_batch_3.
    mat');
144 [XTrain4,YTrain4] = loadBatchAsFourDimensionalArray(location,'data_batch_4.
    mat');
145 [XTrain5,YTrain5] = loadBatchAsFourDimensionalArray(location,'data_batch_5.
    mat');
146 XTrain = cat(4,XTrain1,XTrain2,XTrain3,XTrain4,XTrain5);
147 YTrain = [YTrain1;YTrain2;YTrain3;YTrain4;YTrain5];
148
149 [XTest,YTest] = loadBatchAsFourDimensionalArray(location,'test_batch.mat');
150 end
151
152 function [XBatch,YBatch] = loadBatchAsFourDimensionalArray(location,
    batchFileName)
153 s = load(fullfile(location,batchFileName));
154 XBatch = s.data';
155 XBatch = reshape(XBatch,32,32,3,[]);
156 XBatch = permute(XBatch,[2 1 3 4]);
157 YBatch = convertLabelsToCategorical(location,s.labels);
158 end
159
160 function categoricalLabels = convertLabelsToCategorical(location,
    integerLabels)
161 s = load(fullfile(location,'batches.meta.mat'));
162 categoricalLabels = categorical(integerLabels,0:9,s.label_names);
163 end
```

6.4 GitHub

All the MATLAB scripts used to perform the experiments detailed in this dissertation, along with instructions on how to run them, can be found in the following GitHub repository: [GitHub Repository Link]. Link: https://github.com/rajo69/Enhancing-Neural-Network-Robustness-using-Hybrid-Adversarial-Training.git