

# EE449 HW1\*

\*Note: This homework exaggerated and used LaTeX, I don't know why am I doing this.

Ömer Takkin

No.: 2516987

Electric and Electronic Engineer (EEE)

Middle East Technical University (METU)

Ankara, Turkey

omer.takkin@gmail.com

**Abstract**—This project explores the foundational principles of artificial neural networks by examining key components such as network architecture, training processes, backpropagation algorithms, and activation functions. The project aims to deepen understanding of how neural networks learn to represent complex, non-linear relationships through a combination of theoretical study and practical experimentation. Emphasis is placed on investigating the dynamics of weight adjustment via backpropagation, the impact of various activation functions on convergence behavior, and the overall structure of feed-forward network models. The insights gained lay a robust groundwork for advancing into more complex deep learning architectures and optimizing model performance in real-world applications.

**Index Terms**—METU, EEE, EE449, HW1, Neural Network, Help

$$\begin{aligned} &= \frac{(e^{2x} + 1) \frac{d}{dx}(e^{2x} - 1) - (e^{2x} - 1) \frac{d}{dx}(e^{2x} + 1)}{(e^{2x} + 1)^2} \\ &= \frac{(e^{2x} + 1) \cdot 2e^{2x} - (e^{2x} - 1) \cdot 2e^{2x}}{(e^{2x} + 1)^2} \\ &= \frac{2e^{4x} + 2e^{2x} - 2e^{4x} + 2e^{2x}}{(e^{2x} + 1)^2} = \frac{4e^{2x}}{(e^{2x} + 1)^2} \\ &= \frac{(e^{2x} + 1)^2 - (e^{2x} - 1)^2}{(e^{2x} + 1)^2} = 1 - \frac{(e^{2x} - 1)^2}{(e^{2x} + 1)^2} \\ &= 1 - \tanh^2(x) \end{aligned}$$

$$\therefore \frac{d}{dx} \tanh(x) = 1 - \tanh^2(x)$$

## I. BASIC NEURAL NETWORK CONSTRUCTION AND TRAINING

### A. Preliminary

In this section of the assignment, our objective is to analyze various activation functions along with their corresponding derivatives. Furthermore, we will explore the behavior of these functions within neural networks in the subsequent chapter.

#### 1) Tanh Function:

$$y_1 = \tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

Now we calculate the derivative of the tanh(x) function.

$$\frac{d}{dx} \tanh(x) = \frac{d}{dx} \left( \frac{e^{2x} - 1}{e^{2x} + 1} \right)$$

#### 2) Sigmoid Function:

$$y_1 = \text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

Now we calculate the derivative of the sigmoid(x) function.

$$\frac{d}{dx} \text{sigmoid}(x) = \frac{d}{dx} \left( \frac{1}{1 + e^{-x}} \right)$$

$$= \frac{(e^{-x})}{(1 + e^{-x})^2}$$

$$= \frac{1}{1 + e^{-x}} \cdot \frac{e^{-x}}{1 + e^{-x}}$$

$$= \frac{1}{1 + e^{-x}} \cdot \left( 1 - \frac{1}{1 + e^{-x}} \right)$$

$$= \text{sigmoid}(x) \cdot (1 - \text{sigmoid}(x))$$

$$\therefore \frac{d}{dx} \text{sigmoid}(x) = \text{sigmoid}(x) \cdot (1 - \text{sigmoid}(x))$$

### 3) ReLU Function:

$$y_3 = \max(0, x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$

$$\frac{d}{dx} \max(0, x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases}$$

## II. IMPLEMENTATION

This section of the assignment aims to implement an activation function within the provided neural network structure. The neural network is designed to model the XOR logic gate using a feedforward architecture, comprising one input layer, one hidden layer, and one output layer.

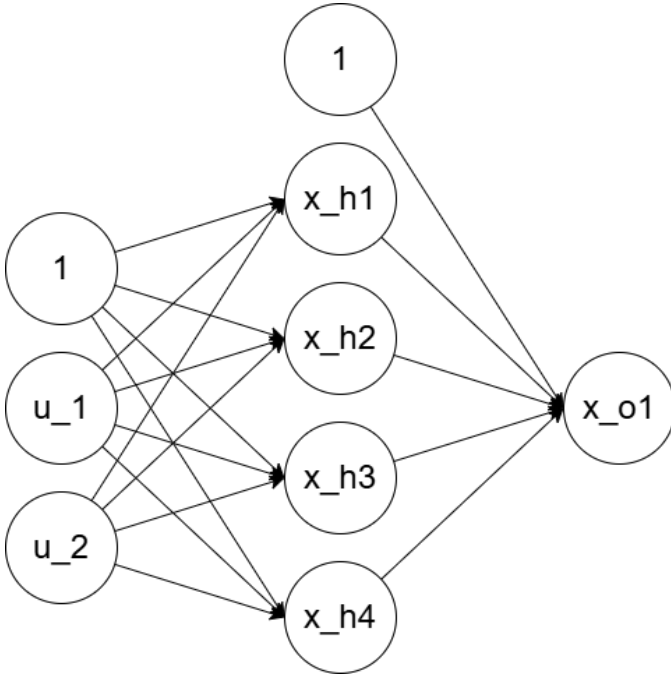


Fig. 1. Structure of neural network that is used.

### A. Parameter Definitions

#### 1) Input Layer Matrices:

$$\mathbf{U}^k = [u_1^k \ u_2^k]^T$$

$$\mathbf{X}_u^k = [\mathbf{U}^1 \ \mathbf{U}^2 \ \dots \ \mathbf{U}^k]^T$$

$$\mathbf{X}_{ubias}^k = [1 \ \mathbf{U}^1 \ \mathbf{U}^2 \ \dots \ \mathbf{U}^k]^T$$

#### 2) Hidden Layer Matrices:

$$\mathbf{X}_h^k = [x_{h1}^k \ x_{h2}^k \ x_{h3}^k \ x_{h4}^k]^T$$

$$\mathbf{X}_h^k = [\mathbf{X}_h^1 \ \mathbf{X}_h^2 \ \dots \ \mathbf{X}_h^k]^T$$

$$\mathbf{X}_{hbias}^k = [1 \ \mathbf{X}_h^1 \ \mathbf{X}_h^2 \ \dots \ \mathbf{X}_h^k]^T$$

### 3) Output Layer Matrices:

$$\mathbf{X}_o^k = [x_{o1}^1 \ x_{o1}^2 \ \dots \ x_{o1}^k]$$

### 4) Weight Matrices:

$\mathbf{W}_L$  : (2x4) matrix input to hidden layer

$\mathbf{W}_{Lbias}$  : (3x4) matrix input to hidden layer with bias

$\mathbf{W}_o$  : (1x4) matrix hidden to output layer

$\mathbf{W}_{obias}$  : (1x5) matrix hidden to output layer with bias

### 5) Activation Function:

$f()$  : Activation function

$f'(\cdot)$  : Derivative of activation function

### 6) Error and Delta Matrices:

$\mathbf{e}_o$  : output layer error

$\delta_o$  : output layer delta error

$\mathbf{e}_h$  : hidden layer error

$\delta_h$  : hidden layer delta error

### 7) Target Output:

$$\mathbf{Y}^k = [y^1 \ y^2 \ \dots \ y^k]$$

### 8) Target Output:

$\mu$  : Learning rate

### B. Forward Propagation

$$\mathbf{X}_{ubias}^k = [1 \ \mathbf{X}_u^k]$$

$$\mathbf{X}_h^k = f(\mathbf{X}_{ubias}^k \times \mathbf{W}_{Lbias})$$

$$\mathbf{X}_{hbias}^k = [1 \ \mathbf{X}_h^k]$$

$$\mathbf{X}_o^k = f(\mathbf{X}_{hbias}^k \times \mathbf{W}_{obias})$$

### C. Back Propagation

$$e_o = 2 \frac{(\mathbf{X}_o^k - \mathbf{Y}_o)}{\text{size}(\mathbf{Y}_o)}$$

$$\delta_o = e_o \odot f'(\mathbf{X}_o^k)$$

$$e_h = e_o \times \mathbf{W}_L$$

$$\delta_h = e_h \odot f'(\mathbf{X}_h^k)$$

$$\mathbf{W}_{Lbias}(n+1) = \mathbf{W}_{Lbias}(n) - \mu \cdot ((\mathbf{X}_{ubias}^k)^T \times \delta_h)$$

$$\mathbf{W}_{obias}(n+1) = \mathbf{W}_{obias}(n) - \mu \cdot ((\mathbf{X}_{hbias}^k)^T \times \delta_o)$$

## D. Results

Detailed analysis of the advantages and disadvantages of the sigmoid, tanh, and ReLU activation functions are given.

The sigmoid function produces outputs in the interval (0,1), which can be interpreted as probabilities. Moreover, it is smooth and differentiable which aids in gradient-based optimization. But for very high and low input values it saturates which can slow our training process. Output is always positive because it has a non-zero center, which leads to inefficient weight updates. The decision boundary of the neural network which uses the sigmoid function is shown in figure 2. Sigmoid functions are usually used when a network needs to predict probabilities.

Tanh function produces outputs in the interval (-1,1), which helps the centering data which causes the faster convergence during training. Also compared to the sigmoid it has a sharper derivative function which causes faster learning. But like sigmoid, tanh also suffers from saturation for large positive or negative inputs, which can slow down learning due to vanishing gradients in deep networks. The decision boundary of the neural network which uses the sigmoid function is shown in figure 3. Tanh function is usually used in hidden layers.

ReLU function is computationally simple and causes faster learning. Also, it is a non-saturating form for positive inputs. However, some neurons can become inactive if they consistently output zero, which causes the network to become unresponsive. Moreover, the unbounded nature of ReLU for positive values can sometimes lead to issues during training, especially with high learning rates. The decision boundary of the neural network which uses the sigmoid function is shown in Figure 4. The ReLU function is usually used in deep learning architectures and hidden layers.

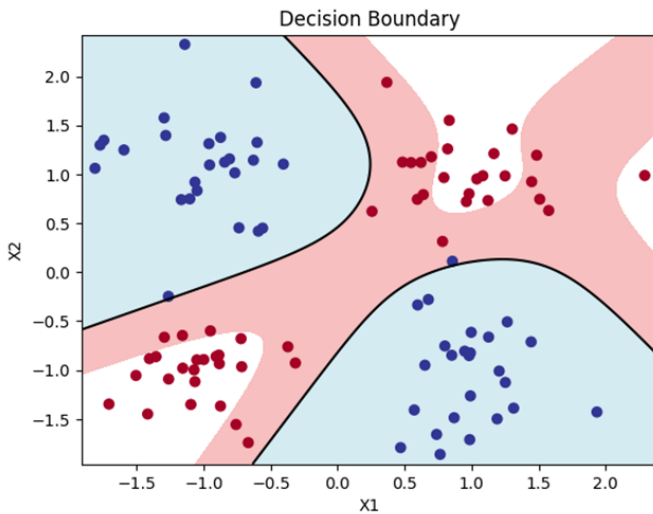


Fig. 2. Activation Function: Tanh, Epoch: 50000, Accuracy: 0.99

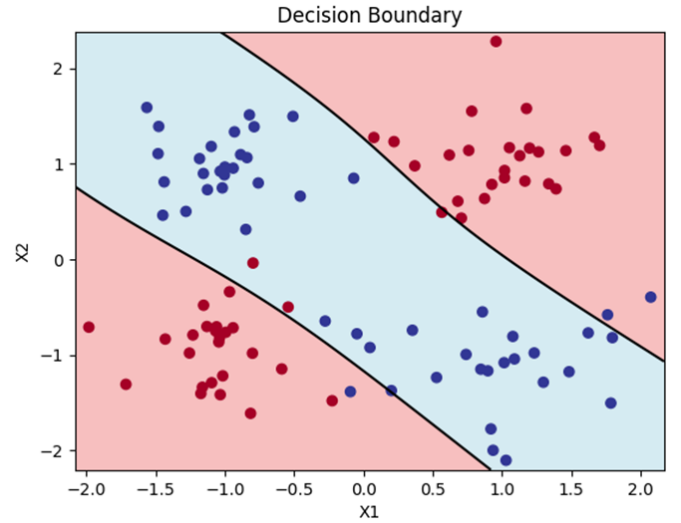


Fig. 3. Activation Function: Sigmoid, Epoch: 50000, Accuracy: 0.94

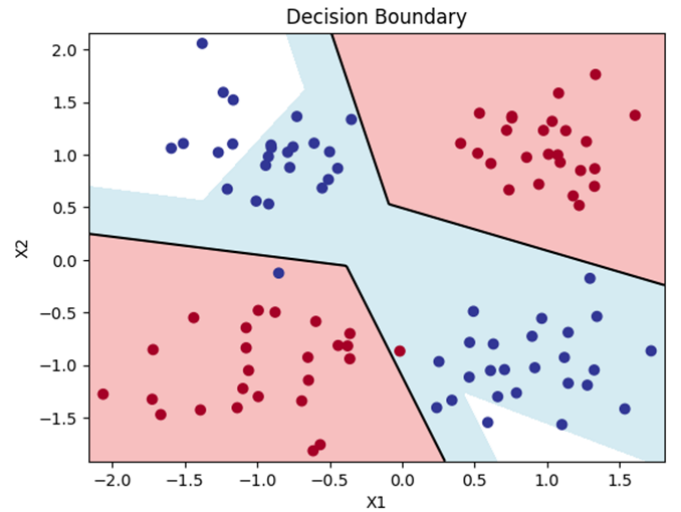


Fig. 4. Activation Function: ReLU, Epoch: 50000, Accuracy: 0.98

## E. XOR Problem

The XOR function is a binary function whose output is illustrated in Figure 5. The primary challenge with the XOR problem is that it is not linearly separable; no single line can effectively distinguish between true and false outputs.

In a two-layer neural network, only one decision boundary can be drawn, limiting the network's ability to separate the data effectively. This results in the network being able to classify points only on one side or the other of the boundary.

However, by introducing a third layer, the network's complexity increases, enabling it to define multiple decision boundaries. This allows the network to classify points not only above and below a single line but also between multiple boundaries. Consequently, this added complexity improves the network's ability to achieve an optimal decision region for the XOR problem.

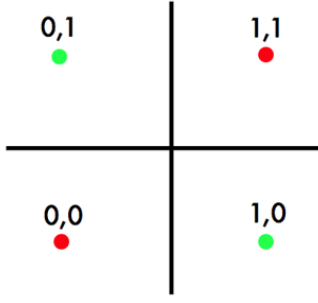


Fig. 5. XOR Problem

#### F. Behaviour of Decision Boundary

Our decision boundary is changing every time we train our network. There can be possible causes for that.

- Each time we train the model, it goes to a different local minimum point
- Our model goes to the same minimum point each time we run, but the distance of reaching the minimum point changes.

These differences are shown in Figure 6. Our model is run three times without changing any value.

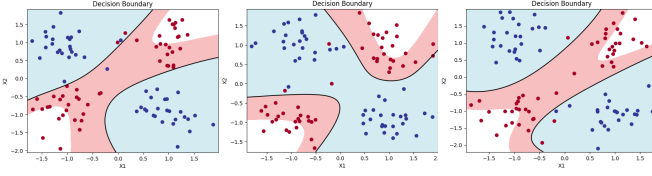


Fig. 6. XOR Problem

### III. IMPLEMENTING A CONVOLUTIONAL LAYER WITH NUMPY

#### A. Preliminary

Shape of input matrix and kernel matrix are given.

$$\text{Input Shape} = \text{BatchSize} \times \text{InputChannels} \\ \times \text{InputHeight} \times \text{InputWidth}$$

$$\text{KernelShape} = \text{OutputChannels} \times \text{InputChannels} \\ \times \text{KernelHeight} \times \text{KernelWidth}$$

First, we want to calculate output shape. We know we will use convolution. In linear convolution output must be as follows.

$$\begin{aligned} \text{OutputHeight} &= \text{InputHeight} - \text{KernelHeight} + 1 \\ \text{OutputWidth} &= \text{InputWidth} - \text{KernelWidth} + 1 \end{aligned}$$

1) *Calculations:* And for every batch we use convolution as shown here:

$$\text{Output}(b, oc, i, j) = \sum_{ic, m, n} \text{Input}(b, ic, i + m, j + n) \\ \times \text{Kernel}(oc, ic, m, n)$$

At the we results are given in figure 7:



Fig. 7. Output of 2D convolution

2) *Results:* Convolution Neural Networks are important because they are specifically designed for grid-like topologies such as images. Unlike traditional fully connected networks, Convolution Neural Networks provide partial structure, which reduces the number of trainable parameters. This makes them efficient tools for extracting features like edges, patterns, textures, etc.

A kernel (also called a filter) is a small matrix used to apply convolution operations on the input data. It slides over the input image to extract features. Kernel size determines the receptive field of our operation. The depth of the kernel should match the number of input channels.

The output image is the result of applying a convolution filter. Each pixel is calculated from some region at the input to extract important visual patterns. Depending on the filter weights, some features may become more pronounced while others are suppressed.

In Figure 7, numbers in the same column are similar, even though they belong to different images. This phenomenon occurs because convolutional filters are shared across the input. Filters are trained to detect similar patterns across multiple images. As a result, when the same feature appears in different images, the filter will produce similar activation values, especially in the early layers where basic features are captured.

The numbers in the same row different, even though they belong to the same image. Each neuron in a row typically corresponds to a different filter. Since each filter extracts a distinct feature, their outputs vary significantly. This diversity is crucial for CNNs to recognize complex patterns by combining multiple feature maps.

Convolutional layers specialize in detecting shared patterns across images. Also, convolutional layers extract diverse features through multiple filters. This combination of shared filter weights and diverse feature extraction enables CNNs to generalize well across different datasets and efficiently capture complex visual structures.

#### IV. EXPERIMENTING ANN ARCHITECTURES

In this part, we will experiment with several ANN architectures for classification tasks via PyTorch. The dataset we will work on is the CIFAR-10 dataset. It is composed of  $32 \times 32$  RGB images of 10 classes which are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. We will aim to find differences between the models that we are training.

##### A. Models

There are five different models we will train. The first two of them are just fully connected neural networks. The last three of them include convolutional neural networks and fully connected neural networks.

- 'mlp-1' : [FC-32, ReLU] + PredictionLayer
- 'mlp-2' : [FC-32, ReLU, FC-64 (no bias)]  
+ PredictionLayer
- 'cnn-3' : [Conv- $3 \times 3 \times 16$ , ReLU, Conv- $5 \times 5 \times 8$ , ReLU,  
MaxPool- $2 \times 2$ , Conv- $7 \times 7 \times 16$ , MaxPool- $2 \times 2$ ]  
+ PredictionLayer
- 'cnn-4' : [Conv- $3 \times 3 \times 16$ , ReLU, Conv- $3 \times 3 \times 8$ , ReLU,  
Conv- $5 \times 5 \times 16$ , ReLU, MaxPool- $2 \times 2$ ,  
Conv- $5 \times 5 \times 16$ , ReLU, MaxPool- $2 \times 2$ ]  
+ PredictionLayer
- 'cnn-5' : [Conv- $3 \times 3 \times 8$ , ReLU, Conv- $3 \times 3 \times 16$ , ReLU,  
Conv- $3 \times 3 \times 8$ , ReLU, Conv- $3 \times 3 \times 16$ , ReLU,  
MaxPool- $2 \times 2$ , Conv- $3 \times 3 \times 16$ ,  
ReLU, Conv- $3 \times 3 \times 8$ , ReLU, MaxPool- $2 \times 2$ ]  
+ PredictionLayer

##### B. Functions

Algorithms designed for train and evaluate models—one for fully connected networks and one for convolutional neural networks. Both functions start by moving the model to the target device and applying He initialization to the appropriate layers, which is particularly effective for ReLU activations, ensuring that weights are set with Kaiming uniform initialization while biases are reset to zero. They utilize cross-entropy loss along with the Adam optimizer for multi-class classification tasks. During training, a small noise term is added to the input images as an adversarial training strategy, which helps improve the model's robustness against minor input perturbations.

In both functions, the entire test dataset is preloaded before training begins to allow for a quick and efficient evaluation of validation performance during training. Training is conducted over a specified number of epochs, and performance metrics such as loss and accuracy are recorded at regular intervals. After each epoch, the model's performance on the preloaded

test data is evaluated to monitor its generalization capability. At the end of the training process, the final test accuracy is computed, and the weights from the first layer of the model are extracted and returned along with the training and validation curves, providing valuable insights into the model's learning dynamics and performance.

##### C. Training

First of all, we configure the device for running PyTorch models by checking if CUDA is available and then sets the device accordingly. It prints out the device being used and establishes key hyperparameters such as the number of training epochs, batch size, and learning rate. Later, we define a series of transformations to be applied to the dataset, which include converting images to tensors and normalizing them with specified mean and standard deviation values for each channel. This normalization is important for ensuring that the input features are on a comparable scale, which can lead to more stable training.

After setting up the transformations, the CIFAR10 dataset is loaded with the specified transformations applied to the images. The dataset is then split into training and testing subsets, with 90% of the data allocated for training and the remaining 10% for testing. Finally, DataLoaders are created for both the training and test datasets, with the training DataLoader set to shuffle the data to ensure randomness during training. This setup enables efficient and structured loading of data in batches for model training and evaluation.

##### D. Results

1) *What is the generalization performance of a classifier?:* The generalization performance of a classifier refers to its ability to accurately predict outcomes on new, unseen data rather than just performing well on the training dataset. It is an important measurement of a model's effectiveness and robustness. It indicates how well the model can apply what it has learned to real-world situations. When a classifier generalizes well, it means that it has captured the underlying patterns in the data rather than memorizing the training examples, which helps prevent issues like overfitting.

2) *Which plots are informative to inspect generalization performance?:* The plot that shows the accuracies of training and validation is informative about generalization performance. We should compare the difference between training and validation accuracies. If training accuracy is much higher than validation accuracy that means that our model is overfitted.

3) *Compare the generalization performance of the architectures.:* Let's analyze models one by one.

When we analyze mlp-1 plot which is given in figure 8, we see that both our validation and training accuracy plot cannot go to higher values. This means our model is underfitted, which means that its complexity is not enough. The number of input parameters of mlp-1 is:

$$3 \times 32 \times 32 = 3072$$

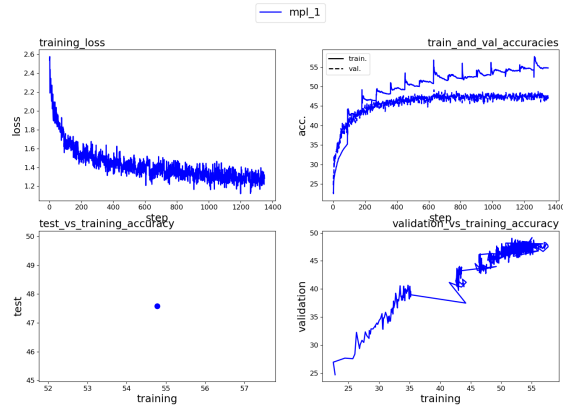


Fig. 8. Results of mlp-1 for part 3

These 3072 neurons go to 32 neurons at hidden layer and there are 32 bias wights:

$$3072 \times 32 + 32 = 98336$$

These 32 neurons go to 10 neurons at the output layer and there are 10 bias weights:

$$32 \times 10 + 10 = 330$$

Then the total number of parameters is:

$$98336 + 330 = 98666$$

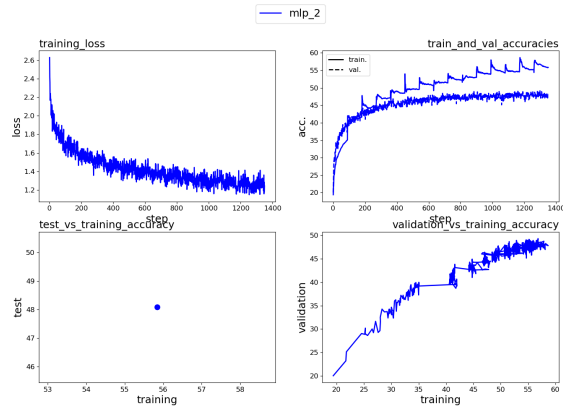


Fig. 9. Results of mlp-2 for part 3

When we analyze mlp-2 plot which is given in figure 9, we see that both our validation and training accuracy plot cannot go to higher values then mlp-1. This means our model is underfitted too. This means, we increased the complexity of our model, we could not increase the accuracy of our model. Now our number of parameters is 101034.

The current approach has shifted from simply increasing the number of parameters to employing a convolutional neural network that is better suited to capturing the local features of the input. Analysis of the cnn-3 plot, as illustrated in

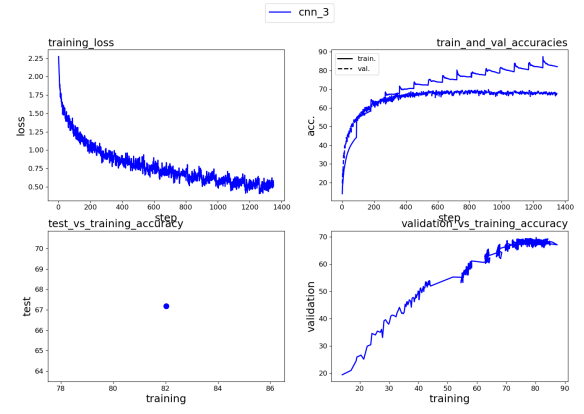


Fig. 10. Results of cnn-3 for part 3

Figure 10, reveals a significant improvement in both training and validation accuracy. In addition, in this model, we have 76194 parameters, which is less than mlp-1 and mlp-2 models but we acquire better performance. However, despite this improvement, the substantial gap between the training and validation accuracy curves indicates that the model is over-fitting. Although the training accuracy continues to increase, the validation accuracy does not exhibit a corresponding improvement, suggesting that the model is memorizing the training data rather than learning generalized representations.

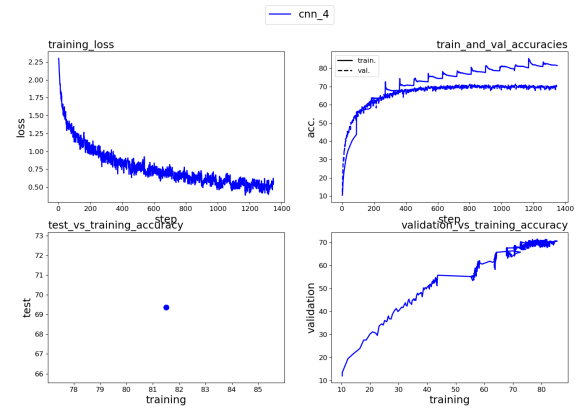


Fig. 11. Results of cnn-4 for part 3

In the CNN architectures between cnn-3 and cnn-4, the key difference lies in the design of the convolutional layers. In CNN-3, smaller filters are used more extensively, ensuring that the receptive field remains comparable to that of larger filters while increasing the overall complexity of the model. Specifically, CNN-3 comprises a sequence of [Conv-3×3×16, ReLU, Conv-5×5×8, ReLU, MaxPool-2×2, Conv-7×7×16, MaxPool-2×2] followed by a prediction layer. In contrast, CNN-4 modifies this configuration by employing [Conv-3×3×16, ReLU, Conv-3×3×8, ReLU, Conv-5×5×16, ReLU, MaxPool-2×2, Conv-5×5×16, ReLU, MaxPool-2×2] before the prediction layer. Although both networks achieve similar

receptive fields, CNN-4 introduces an additional convolutional layer and uses a combination of both  $3 \times 3$  and  $5 \times 5$  filters, thereby increasing the model's complexity and its capacity to capture finer-grained features. In addition our number of parameters is increased to 77490.

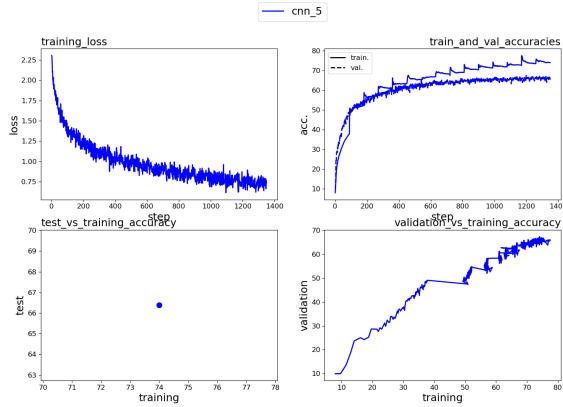


Fig. 12. Results of cnn-5 for part 3

CNN-3 and CNN-4 differ primarily in their convolutional layer configurations and filter sizes. CNN-3 employs a structure that begins with a  $3 \times 3 \times 16$  convolution, followed by a  $5 \times 5 \times 8$  layer, max pooling, and then a  $7 \times 7 \times 16$  convolution before a second pooling stage and the prediction layer. In contrast, CNN-4 incorporates additional convolutional layers—starting with a  $3 \times 3 \times 16$  layer followed by a  $3 \times 3 \times 8$ , then a  $5 \times 5 \times 16$  layer, max pooling, and another  $5 \times 5 \times 16$  layer with subsequent pooling—thus increasing the model's complexity and feature extraction capability. Although CNN-4 achieves higher training accuracy due to its increased capacity, this comes at the cost of a larger gap between training and validation accuracies, indicating a tendency toward overfitting. In comparison, CNN-5, which exclusively uses smaller  $3 \times 3$  filters arranged in a deeper architecture, achieves the best validation performance while still showing improvements in training accuracy. This suggests that CNN-5 is effectively learning generalized representations rather than merely memorizing the training data. In addition, our number of parameters is reduced to 40682. CNN-5 has significantly fewer parameters than CNN-3 and CNN-4 due to its deeper but narrower architecture, reducing the number of trainable parameters while still maintaining performance improvements.

4) *How does the number of parameters affect the classification and generalization performance?*: The number of parameters in a model directly influences its capacity to learn complex patterns from the data. A model with a larger number of parameters can potentially capture more intricate features and nuances, which may improve classification performance on the training data. However, if the number of parameters is excessively high relative to the amount of training data or without proper regularization, the model may overfit—memorizing the training examples rather than learning generalizable patterns. This overfitting leads to a significant gap between

training and validation accuracy, where the model performs well on seen data but poorly on unseen data. Conversely, models with too few parameters may underfit, failing to capture sufficient complexity to distinguish between classes. Therefore, striking the right balance in model complexity is crucial for achieving good generalization performance while maintaining strong classification accuracy.

5) *How does the depth of the architecture affect the classification and generalization performance?*: The depth of a neural network architecture significantly impacts both classification and generalization performance. A deeper network allows for hierarchical feature extraction, where lower layers learn simple features such as edges and textures, while higher layers capture more abstract patterns relevant to classification. This can improve classification accuracy, as the network becomes capable of modeling complex relationships in the data.

However, increasing depth also introduces challenges. If the network is too deep without proper optimization techniques such as batch normalization or residual connections, it may suffer from vanishing or exploding gradients, making training inefficient. Additionally, deeper architectures are more prone to overfitting, especially if the dataset is small, as they have more parameters to memorize the training data rather than generalizing to new samples. On the other hand, if the depth is too shallow, the model may underfit, failing to learn sufficiently complex representations needed for accurate classification. Therefore, an optimal depth must be chosen based on the dataset size, regularization techniques, and computational constraints to ensure good generalization performance.

6) *Considering the visualizations of the weights, are they interpretable?*: The interpretability of weight visualizations depends on the network's depth. In CNNs, early layer weights are often interpretable, resembling edge detectors or color filters. As depth increases, filters become more abstract, making them harder to interpret. Mid-level layers may capture textures or object parts, while deep layers encode complex, less intuitive patterns. Fully connected layers are even less interpretable due to dense mappings. While weight visualizations provide insights into feature learning, their clarity diminishes with depth, requiring techniques like activation maximization or attention maps for better understanding.

7) *Can you say whether the units are specialized to specific classes?*: Yes, in deep neural networks, particularly in CNNs, some units can become specialized to specific classes. In earlier layers, units detect general features like edges and textures, which are shared across classes. However, in deeper layers, neurons may respond strongly to specific patterns or object parts associated with particular classes. This specialization can be observed through techniques. However, not all units are strictly class-specific, as some may contribute to multiple classes by recognizing common patterns.

8) *Which architecture would you pick for this classification task? Why?*: I would select CNN-5 structure due to its performance even if it has half of number of parameters than other CNNs.



## EXPERIMENTING ACTIVATION FUNCTIONS

The gradient behavior varies across different architectures, with the ReLU function consistently exhibiting a larger gradient magnitude compared to the sigmoid function. This results in more substantial updates for ReLU-based models, leading to a more significant decrease in training loss. As the depth of the architecture increases, models utilizing ReLU demonstrate a sharper decline in training loss, whereas those using the sigmoid activation function struggle to reduce the loss effectively, often stagnating around a training loss of 2. In contrast, ReLU-based CNN models consistently achieve a training loss below 1.

This difference arises because the ReLU function avoids the vanishing gradient problem that affects sigmoid activations. Sigmoid functions tend to saturate in deeper networks, causing gradients to diminish, which hampers effective weight updates. Conversely, ReLU maintains non-zero gradients for positive values, enabling more efficient training, especially in deep architectures.

In Part 1.2, different activation functions did not yield significantly different results because the XOR neural network was relatively simple. However, in deeper CNN architectures, activation function choice plays a crucial role in training efficiency and convergence.

If inputs were in the range  $[0, 255]$  instead of  $[0.0, 1.0]$ , the network might struggle with optimization due to larger input magnitudes, potentially leading to instability or inefficient weight updates. Normalization helps stabilize training by keeping gradients in a manageable range, preventing numerical instability, and improving convergence rates.

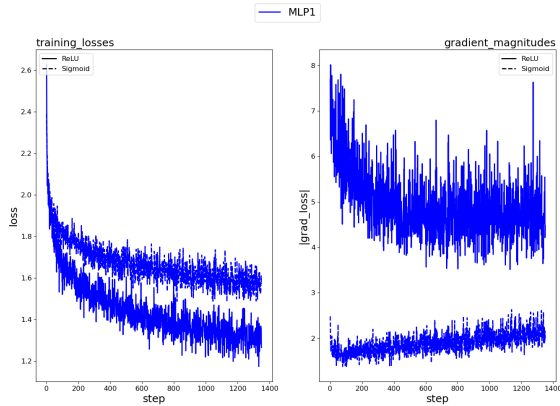


Fig. 13. Results of mlp-1 for part 4

## EXPERIMENTING LEARNING RATE

The learning rate plays a critical role in both the convergence speed and the final performance of the model. A higher learning rate allows the model to converge faster by taking larger steps in the optimization process, but it also increases the risk of overshooting the optimal solution or causing instability. Conversely, a lower learning rate results in

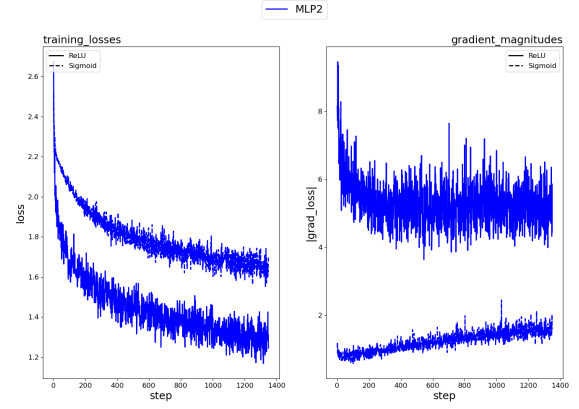


Fig. 14. Results of mlp-2 for part 4

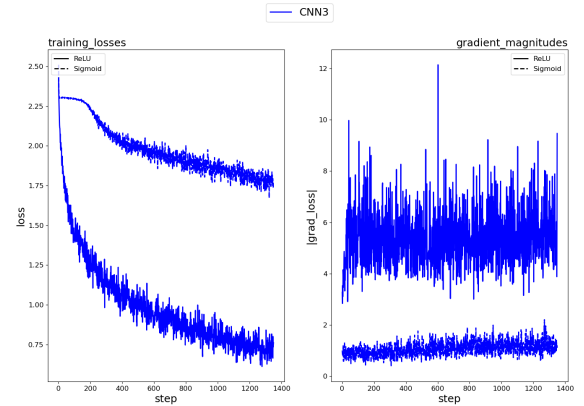


Fig. 15. Results of cnn-3 for part 4

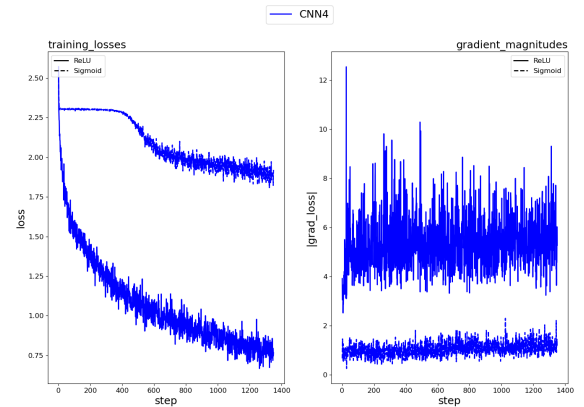


Fig. 16. Results of cnn-4 for part 4



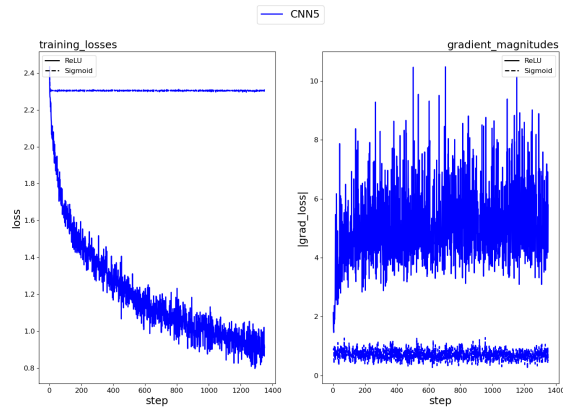


Fig. 17. Results of cnn-5 for part 4

slower convergence but improves stability, allowing the model to fine-tune its parameters more effectively.

The choice of learning rate also affects whether the model converges to a good solution. If the learning rate is too high, the optimization process may never settle at an optimal point, leading to poor generalization. The model may take too long to converge or get stuck in local minima if it is too low.

The scheduled learning rate method, if designed correctly, helps balance fast initial convergence with fine-tuning in later stages. By reducing the learning rate over time, the model avoids large updates that could destabilize training and allows for a more precise search for the optimal parameters.

Comparing the scheduled learning rate method with Adam, Adam generally performs well because it adaptively adjusts the learning rate for each parameter, leading to faster and more stable convergence. However, in some cases, a well-tuned scheduled learning rate can achieve similar or even better accuracy by gradually refining the model's parameters. Adam often reaches a good solution faster, but scheduled learning rates may provide better final accuracy by allowing more controlled fine-tuning.

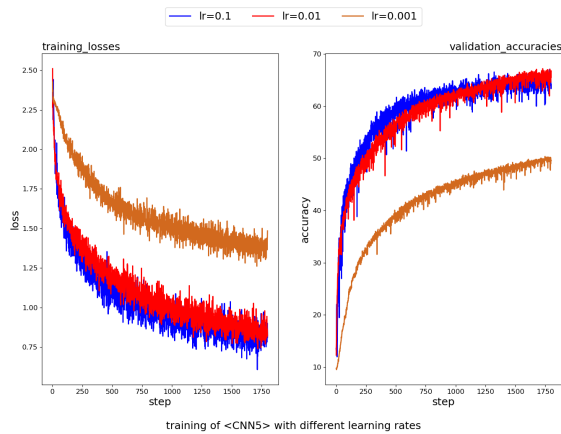


Fig. 18. Results of cnn-5 for part 5

## APPENDIX

### A. Part 1

#### 1) part1.py:

```
1 import sys
2 import os
3 sys.path.append(os.path.abspath('_given'))
4
5 import numpy as np
6 from matplotlib import pyplot as plt
7 from utils import part1CreateDataset, part1PlotBoundary
8
9
10 class MLP:
11     def __init__(self, input_size, hidden_size, output_size):
12         self.input_size = input_size
13         self.hidden_size = hidden_size
14         self.output_size = output_size
15
16         # Initialize weights and biases
17         self.weights_input_hidden = np.random.randn(self.input_size + 1, self.hidden_size) # 3x4 matrix
18         self.bias_hidden = np.zeros((1, self.hidden_size)) # 1x4
19
20         self.weights_hidden_output = np.random.randn(self.hidden_size + 1, self.output_size) # 5x1 matrix
21         self.bias_output = np.zeros((1, self.output_size)) # 1x1
22
23
24     def f_activation(self, x):
25         return (np.exp(2*x)-1)/(np.exp(2*x)+1) # Tanh
26         #return 1 / (1 + np.exp(-x)) # Sigmoid
27         #return np.maximum(0, x) # ReLU
28
29     def f_activation_derivative(self, x):
30         return (1 - np.square(x)) # Tanh
31         #return x * (1 - x) # Sigmoid
32         #return (x>0).astype(float) # ReLU
33
34     def forward(self, inputs):
35         # Forward pass through the network
36         inputs = np.hstack((np.ones((inputs.shape[0], 1)), inputs)) # Kx3 matrix, first column is 1
37         self.hidden_output = self.f_activation(np.matmul(inputs, self.weights_input_hidden)) # Kx4
38
39         self.hidden_output_w_bias = np.hstack((np.ones((self.hidden_output.shape[0], 1)), self.hidden_output))
40         # Kx5 matrix, first column is 1
41         self.output = self.f_activation(np.matmul(self.hidden_output_w_bias, self.weights_hidden_output)) # Kx1
42         return self.output
43
44     def backward(self, inputs, targets, learning_rate):
45         # Backward pass through the network
46
47         # Compute output layer error and its gradient
48         output_error = (self.output - targets) * 2.0 / targets.size # Kx1
49         output_delta = output_error * self.f_activation_derivative(self.output) * 2.0 # Kx1
50
51         # Compute hidden layer error and its gradient
52         hidden_error = np.matmul(output_error, self.weights_hidden_output[1:].T) # Kx4
53         hidden_delta = hidden_error * self.f_activation_derivative(self.hidden_output) # Kx4
54
55         inputs = np.hstack((np.ones((inputs.shape[0], 1)), inputs)) # Kx3 matrix, first column is 1
56
57         # Update weights and biases
58         self.weights_hidden_output -= learning_rate * np.matmul(self.hidden_output_w_bias.T, output_delta) # 5x1
59         self.weights_input_hidden -= learning_rate * np.matmul(inputs.T, hidden_delta) # 3x4 matrix
60
61
62 # Generate the dataset
63 x_train, y_train, x_val, y_val = part1CreateDataset(train_samples=1000, val_samples=100, std=0.4)
64
65 # Define neural network parameters
66 input_size = 2
67 hidden_size = 4
```

```

68 | output_size = 1
69 | learning_rate = 0.001
70 |
71 | # Create neural network
72 | nn = MLP(input_size, hidden_size, output_size)
73 |
74 | # Train the neural network
75 | for epoch in range(50000):
76 |
77 |     # Forward propagation
78 |     output = nn.forward(x_train)
79 |
80 |     # Backpropagation
81 |     nn.backward(x_train, y_train, learning_rate)
82 |
83 |     # Print the loss (MSE) every 1000 epochs
84 |     if epoch % 1000 == 0:
85 |         # Compute predictions for the entire validation set
86 |         val_predictions = nn.forward(x_val)
87 |         loss = 0.5 * np.mean(np.square(y_val - val_predictions))
88 |         print(f'Epoch_{epoch}: Loss_{loss}')
89 |
90 |
91 | # Test the trained neural network
92 | val_predictions = nn.forward(x_val)
93 |
94 | y_predict = ((val_predictions > 0.5).astype(int)).reshape(-1,1)
95 | accuracy = np.mean((y_predict == y_val).astype(int))
96 | print(f'{accuracy*100}% of test examples classified correctly.')
97 |
98 | part1PlotBoundary(x_val, y_val, nn)

```

## B. Part 2

### 1) part2.py:

```
1 import numpy as np
2 import os
3
4 # Assuming samples_7.npy is in /content/drive/MyDrive/EE449/HW1
5 file_path = os.path.join('_given\data\data', 'samples_7.npy')
6
7 # input shape: [batch size, input_channels, input_height, input_width]
8 input = np.load(file_path)
9 # input shape: [output_channels, input_channels, filter_height, filter width]
10 # Assuming kernel.npy is also in /content/drive/MyDrive/EE449/HW1
11 kernel_path = os.path.join('_given\data\data', 'kernel.npy')
12 kernel = np.load(kernel_path) # Load kernel.npy from the specified path
13
14 import sys
15 sys.path.append('_given')
16 from utils import part2Plots
17
18
19 def my_conv2d(input, kernel):
20     """
21     Performs a 2D convolution (forward propagation) with no padding and stride 1.
22
23     Args:
24         input (np.ndarray): Input data of shape (batch_size, in_channels, in_height, in_width).
25         kernel (np.ndarray): Convolutional kernel of shape (out_channels, in_channels, k_height, k_width).
26
27     Returns:
28         np.ndarray: The result of the convolution, of shape (batch_size, out_channels, out_height,
29                     out_width),
30                     where:
31                         out_height = in_height - kernel_height + 1,
32                         out_width = in_width - kernel_width + 1.
33     """
34     batch_size, in_channels, in_height, in_width = input.shape
35     out_channels, kernel_in_channels, kernel_height, kernel_width = kernel.shape
36
37     # Check if the input channels and kernel channels match
38     if in_channels != kernel_in_channels:
39         print("error")
40         raise ValueError("The number of input channels must match the kernel's input channels.")
41
42     # Compute output dimensions
43     out_height = in_height - kernel_height + 1
44     out_width = in_width - kernel_width + 1
45
46     # Initialize the output tensor with zeros
47     output = np.zeros((batch_size, out_channels, out_height, out_width))
48
49     # Iterate over the batch, output channels, and spatial locations to apply the convolution filter
50     for b in range(batch_size):
51         for oc in range(out_channels):
52             for i in range(out_height):
53                 for j in range(out_width):
54                     # Extract the current patch from the input
55                     patch = input[b, :, i:i+kernel_height, j:j+kernel_width]
56                     # Perform elementwise multiplication and sum over the channel and kernel dimensions
57                     output[b, oc, i, j] = np.sum(patch * kernel[oc, :, :, :])
58
59     print("Convolution is done!")
60     return output
61
62 out = my_conv2d(input, kernel)
63 part2Plots(out)
```

### C. Part 3

#### 1) part3.py:

```
1 import torch
2 import torchvision
3 import torchvision.transforms as transforms
4 from models import MLP_1, MLP_2, CNN_3, CNN_4, CNN_5
5 from functions import train_and_evaluate, CNN_train_and_evaluate, convert_to_serializable
6 import json
7
8 # Device configuration
9 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
10 print("You are using device:" , device)
11
12 # Hyper-parameters
13 num_epochs = 15
14 batch_size = 50
15 learning_rate = 0.001
16
17 #####
18 # Transformations
19 #####
20
21 transform = transforms.Compose([
22     transforms.ToTensor(),
23     transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),
24 ])
25
26 # Load the dataset (assuming it's a custom dataset with 100,000 images)
27 full_dataset = torchvision.datasets.CIFAR10(root='./data',
28                                             train=True,
29                                             download=True,
30                                             transform=transform)
31
32 # Compute split sizes
33 train_size = int(0.9 * len(full_dataset)) # 90% training
34 test_size = len(full_dataset) - train_size # 10% testing
35
36 # Split dataset
37 train_dataset, test_dataset = torch.utils.data.random_split(full_dataset, [train_size, test_size])
38
39 # Create DataLoaders
40 train_loader = torch.utils.data.DataLoader(train_dataset,
41                                           batch_size=batch_size,
42                                           shuffle=True)
43
44 test_loader = torch.utils.data.DataLoader(test_dataset,
45                                           batch_size=batch_size,
46                                           shuffle=False)
47
48
49 # Instantiate the model
50 model_MLP1 = MLP_1()
51 model_MLP2 = MLP_2()
52 model_CNN3 = CNN_3()
53 model_CNN4 = CNN_4()
54 model_CNN5 = CNN_5()
55
56 print("Training of MLP1 is starting")
57 result_MLP1 = train_and_evaluate('mpl_1',
58                                 model_MLP1,
59                                 train_loader,
60                                 test_loader,
61                                 learning_rate,
62                                 num_epochs)
63
64 print("MLP_1 Training and evaluation finished")
65 PATH = './part3/results/MLP_1.pth'
66 torch.save(model_MLP1.state_dict(), PATH)
67
68 result_serializable_MLP1 = convert_to_serializable(result_MLP1)
69 # Save to JSON file
70 with open('./part3/results/result_MLP1.json', 'w') as f:
71     json.dump(result_serializable_MLP1, f, indent=4)
72
```

```

73 print("Tranining_of_MLP2_is_starting")
74 result_MLP2 = train_and_evaluate('mlp_2',
75                                 model_MLP2,
76                                 train_loader,
77                                 test_loader,
78                                 learning_rate,
79                                 num_epochs)
80 print("MLP_2_Training_and_evaluation_finished")
81 PATH = './part3/results/MLP_2.pth'
82 torch.save(model_MLP2.state_dict(), PATH)
83
84 result_serializable_MLP2 = convert_to_serializable(result_MLP2)
85 # Save to JSON file
86 with open('./part3/results/result_MLP2.json', 'w') as f:
87     json.dump(result_serializable_MLP2, f, indent=4)
88
89
90 print("Tranining_of_CNN3_is_starting")
91 result_CNN3 = CNN_train_and_evaluate('cnn_3',
92                                     model_CNN3,
93                                     train_loader,
94                                     test_loader,
95                                     learning_rate,
96                                     num_epochs,
97                                     device)
98 print("CNN_3_Training_and_evaluation_finished")
99 PATH = './part3/results/CNN_3.pth'
100 torch.save(model_CNN3.state_dict(), PATH)
101
102 result_serializable_CNN3 = convert_to_serializable(result_CNN3)
103 # Save to JSON file
104 with open('./part3/results/result_CNN3.json', 'w') as f:
105     json.dump(result_serializable_CNN3, f, indent=4)
106
107
108 print("Tranining_of_CNN4_is_starting")
109 result_CNN4 = CNN_train_and_evaluate('cnn_4',
110                                     model_CNN4,
111                                     train_loader,
112                                     test_loader,
113                                     learning_rate ,
114                                     num_epochs,
115                                     device)
116 print("CNN_4_Training_and_evaluation_finished")
117 PATH = './part3/results/CNN_4.pth'
118 torch.save(model_CNN4.state_dict(), PATH)
119
120 result_serializable_CNN4 = convert_to_serializable(result_CNN4)
121 # Save to JSON file
122 with open('./part3/results/result_CNN4.json', 'w') as f:
123     json.dump(result_serializable_CNN4, f, indent=4)
124
125
126 print("Tranining_of_CNN5_is_starting")
127 result_CNN5 = CNN_train_and_evaluate('cnn_5',
128                                     model_CNN5,
129                                     train_loader,
130                                     test_loader,
131                                     learning_rate,
132                                     num_epochs,
133                                     device)
134 print("CNN_5_Training_and_evaluation_finished")
135 PATH = './part3/results/CNN_5.pth'
136 torch.save(model_CNN5.state_dict(), PATH)
137
138 result_serializable_CNN5 = convert_to_serializable(result_CNN5)
139 # Save to JSON file
140 with open('./part3/results/result_CNN5.json', 'w') as f:
141     json.dump(result_serializable_CNN5, f, indent=4)

```

## 2) functions.py:

```

1 import torch
2 import numpy as np
3

```

```

4 def train_and_evaluate(model_name, in_model, in_train_loader, in_test_loader, learning_rate, num_epochs=15,
5     device='cuda'):
6     model = in_model.to(device)
7
8     # He initialization, good for ReLU
9     def init_weights(m):
10         if isinstance(m, torch.nn.Linear):
11             torch.nn.init.kaiming_uniform_(m.weight)
12             if m.bias is not None:
13                 m.bias.data.fill_(0.0)
14
15     # Apply He initialization to all applicable layers
16     model.apply(init_weights)
17
18     criterion = torch.nn.CrossEntropyLoss()
19     optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
20
21     training_loss = []
22     training_accuracy = []
23     validation_accuracy = []
24
25     # Preload and flatten test data
26     test_images, test_labels = [], []
27     for images, labels in in_test_loader:
28         test_images.append(images.reshape(-1, 3*32*32).to(device))
29         test_labels.append(labels.to(device))
30     test_images = torch.cat(test_images, dim=0)
31     test_labels = torch.cat(test_labels, dim=0)
32     test_batch_size = in_test_loader.batch_size # Use the same batch size as the test loader
33
34     n_total_steps = len(in_train_loader)
35     for epoch in range(num_epochs):
36         running_loss = 0.0
37         correct = 0
38         total = 0
39         model.train()
40
41         for i, (images, labels) in enumerate(in_train_loader):
42             images = images.reshape(-1, 3*32*32).to(device)
43             images = images + 0.007 * torch.randn_like(images) # Adversarial Training
44             labels = labels.to(device)
45
46             # Forward pass
47             outputs = model(images)
48             loss = criterion(outputs, labels)
49
50             # Backward pass and optimize
51             optimizer.zero_grad()
52             loss.backward()
53             optimizer.step()
54
55             running_loss += loss.item()
56             _, predicted = torch.max(outputs.data, 1)
57             total += labels.size(0)
58             correct += (predicted == labels).sum().item()
59
60         # Log training metrics every 10 steps
61         if (i + 1) % 10 == 0:
62             training_loss.append(running_loss / 10)
63             training_accuracy.append(100 * correct / total)
64             running_loss = 0.0
65
66         # Calculate validation accuracy using preloaded data
67         model.eval()
68         val_correct = 0
69         val_total = 0
70         with torch.no_grad():
71             for i_batch in range(0, len(test_images), test_batch_size):
72                 batch_images = test_images[i_batch:i_batch+test_batch_size]
73                 batch_labels = test_labels[i_batch:i_batch+test_batch_size]
74                 outputs = model(batch_images)
75                 _, predicted = torch.max(outputs.data, 1)
76                 val_total += batch_labels.size(0)

```



```

77         val_correct += (predicted == batch_labels).sum().item()
78         val_acc = 100.0 * val_correct / val_total
79         validation_accuracy.append(val_acc)
80         model.train()
81
82     # Print epoch statistics
83     epoch_train_acc = 100 * correct / total
84     print(f'[{epoch+1}]_Training_Accuracy:_{epoch_train_acc:.2f}%,_Validation_Accuracy:_{val_acc:.2f}%')
85
86 # Final evaluation using preloaded test data
87 model.eval()
88 n_correct = 0
89 with torch.no_grad():
90     for i_batch in range(0, len(test_images), test_batch_size):
91         batch_images = test_images[i_batch:i_batch+test_batch_size]
92         batch_labels = test_labels[i_batch:i_batch+test_batch_size]
93         outputs = model(batch_images)
94         _, predicted = torch.max(outputs.data, 1)
95         n_correct += (predicted == batch_labels).sum().item()
96 acc = 100.0 * n_correct / len(test_labels)
97 print(f'Accuracy_of_the_model:_{acc:.2f}%')
98
99 first_layer_weights = model.fc1.weight.data.cpu().numpy()
100
101 return {
102     'name': model_name,
103     'loss_curve': training_loss,
104     'train_acc_curve': training_accuracy,
105     'val_acc_curve': validation_accuracy,
106     'test_acc': acc,
107     'weights': first_layer_weights
108 }
109
110 def CNN_train_and_evaluate(model_name, in_model, in_train_loader, in_test_loader, learning_rate, num_epochs
    =15, device='cuda'):
111     model = in_model.to(device)
112
113     # He initialization, good for ReLU
114     def init_weights(m):
115         if isinstance(m, torch.nn.Linear):
116             torch.nn.init.kaiming_uniform_(m.weight)
117             if m.bias is not None:
118                 m.bias.data.fill_(0.0)
119         elif isinstance(m, torch.nn.Conv2d):
120             torch.nn.init.kaiming_uniform_(m.weight)
121             if m.bias is not None:
122                 m.bias.data.fill_(0.0)
123
124     # Apply He initialization to all applicable layers
125     model.apply(init_weights)
126
127     criterion = torch.nn.CrossEntropyLoss()
128     optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
129
130     training_loss = []
131     training_accuracy = []
132     validation_accuracy = []
133
134     # Preload all test data once
135     test_images, test_labels = [], []
136     for images, labels in in_test_loader:
137         test_images.append(images.to(device))
138         test_labels.append(labels.to(device))
139     test_images = torch.cat(test_images)
140     test_labels = torch.cat(test_labels)
141     test_batch_size = in_test_loader.batch_size # Preserve original batch size
142
143     for epoch in range(num_epochs):
144         running_loss = 0.0
145         correct = 0
146         total = 0
147         model.train()
148
149         for i, (images, labels) in enumerate(in_train_loader):

```

```

150     images, labels = images.to(device), labels.to(device)
151     images = images + 0.007 * torch.randn_like(images) # Adversarial Training
152
153     # Forward + backward
154     outputs = model(images)
155     loss = criterion(outputs, labels)
156     loss.backward()
157     optimizer.step()
158     optimizer.zero_grad()
159
160     # Update metrics
161     running_loss += loss.item()
162     _, predicted = torch.max(outputs, 1)
163     total += labels.size(0)
164     correct += (predicted == labels).sum().item()
165
166     # Validation every 10 batches
167     if (i + 1) % 10 == 0:
168         # Store training metrics
169         training_loss.append(running_loss / 10)
170         training_accuracy.append(100 * correct / total)
171         running_loss = 0.0
172
173         # Fast validation using preloaded data
174         model.eval()
175         val_correct = 0
176         with torch.no_grad():
177             # Process in original batch sizes
178             for batch_start in range(0, len(test_images), test_batch_size):
179                 batch_images = test_images[batch_start:batch_start+test_batch_size]
180                 batch_labels = test_labels[batch_start:batch_start+test_batch_size]
181                 outputs = model(batch_images)
182                 _, predicted = torch.max(outputs, 1)
183                 val_correct += (predicted == batch_labels).sum().item()
184
185         val_acc = 100.0 * val_correct / len(test_labels)
186         validation_accuracy.append(val_acc)
187         model.train()
188
189     # Epoch statistics
190     epoch_acc = 100 * correct / total
191     print(f'[{epoch+1}]_Train_Acc:_{epoch_acc:.2f}%,_Val_Acc:_{val_acc:.2f}%')
192
193 # Final evaluation with preloaded data
194 model.eval()
195 final_correct = 0
196 with torch.no_grad():
197     for batch_start in range(0, len(test_images), test_batch_size):
198         batch_images = test_images[batch_start:batch_start+test_batch_size]
199         batch_labels = test_labels[batch_start:batch_start+test_batch_size]
200         outputs = model(batch_images)
201         final_correct += (torch.max(outputs, 1)[1] == batch_labels).sum().item()
202
203 acc = 100.0 * final_correct / len(test_labels)
204 print(f'Final_Accuracy:_{acc:.2f}%')
205
206 first_layer_weights = model.conv1.weight.data.cpu().numpy()
207
208 return {
209     'name': model_name,
210     'loss_curve': training_loss,
211     'train_acc_curve': training_accuracy,
212     'val_acc_curve': validation_accuracy,
213     'test_acc': acc,
214     'weights': first_layer_weights
215 }
216
217
218 def convert_to_serializable(obj):
219     if isinstance(obj, torch.Tensor):
220         return obj.tolist() # Convert tensors to lists
221     if isinstance(obj, np.ndarray):
222         return obj.tolist() # Convert NumPy arrays to lists
223     if isinstance(obj, dict):

```

```

224         return {k: convert_to_serializable(v) for k, v in obj.items()} # Recursively convert dicts
225     if isinstance(obj, list):
226         return [convert_to_serializable(i) for i in obj] # Recursively convert lists
227     return obj # Return the object as is if it's already serializable

```

### 3) models.py:

```

1 import torch
2 import torch.nn.functional as F
3
4
5 class MLP_1(torch.nn.Module):
6     def __init__(self):
7         super(MLP_1, self).__init__()
8         self.fc1 = torch.nn.Linear(3 * 32 * 32, 32)
9         self.relu = torch.nn.ReLU()
10        self.fc2 = torch.nn.Linear(32, 10)
11
12    def forward(self, x):
13        out = self.fc1(x)
14        out = self.relu(out)
15        out = self.fc2(out)
16        return out
17
18
19 class MLP_2(torch.nn.Module):
20     def __init__(self):
21         super(MLP_2, self).__init__()
22         self.fc1 = torch.nn.Linear(3 * 32 * 32, 32)
23         self.relu = torch.nn.ReLU()
24         self.fc2 = torch.nn.Linear(32, 64, bias=False)
25         self.fc3 = torch.nn.Linear(64, 10)
26
27    def forward(self, x):
28        out = self.fc1(x)
29        out = self.relu(out)
30        out = self.fc2(out)
31        out = self.relu(out)
32        out = self.fc3(out)
33        return out
34
35
36 class CNN_3(torch.nn.Module):
37     def __init__(self):
38         super().__init__()
39         self.conv1 = torch.nn.Conv2d(3, 16, 3, padding=1) # Conv-3x3x16
40         self.conv2 = torch.nn.Conv2d(16, 8, 5, padding=2) # Conv-5x5x8
41         self.conv3 = torch.nn.Conv2d(8, 16, 7, padding=3) # Conv-7x7x16
42         self.pool = torch.nn.MaxPool2d(2, 2)
43         self.fc1 = torch.nn.Linear(16 * 8 * 8, 64)
44         self.fc2 = torch.nn.Linear(64, 10)
45
46    def forward(self, x):
47        # Input: N, 3, 32, 32
48        x = F.relu(self.conv1(x)) # -> N, 16, 32, 32
49        x = F.relu(self.conv2(x)) # -> N, 8, 32, 32
50        x = self.pool(x) # -> N, 8, 16, 16
51        x = F.relu(self.conv3(x)) # -> N, 16, 16, 16
52        x = self.pool(x) # -> N, 16, 8, 8
53        x = torch.flatten(x, 1) # -> N, 16 * 8 * 8 = 1024
54        x = F.relu(self.fc1(x)) # -> N, 64
55        x = self.fc2(x) # -> N, 10
56        return x
57
58 class CNN_4(torch.nn.Module):
59     def __init__(self):
60         super().__init__()
61         self.conv1 = torch.nn.Conv2d(3, 16, 3, padding=1) # Conv-3x3x16
62         self.conv2 = torch.nn.Conv2d(16, 8, 3, padding=1) # Conv-3x3x8
63         self.conv3 = torch.nn.Conv2d(8, 16, 5, padding=2) # Conv-5x5x16
64         self.conv4 = torch.nn.Conv2d(16, 16, 5, padding=2) # Conv-5x5x16
65         self.pool = torch.nn.MaxPool2d(2, 2)
66         self.fc1 = torch.nn.Linear(16 * 8 * 8, 64)
67         self.fc2 = torch.nn.Linear(64, 10)
68

```

```

69 def forward(self, x):
70     # Input: N, 3, 32, 32
71     x = F.relu(self.conv1(x)) # -> N, 16, 32, 32
72     x = F.relu(self.conv2(x)) # -> N, 8, 32, 32
73     x = F.relu(self.conv3(x)) # -> N, 16, 32, 32
74     x = self.pool(x) # -> N, 16, 16, 16
75     x = F.relu(self.conv4(x)) # -> N, 16, 16, 16
76     x = self.pool(x) # -> N, 16, 8, 8
77
78     x = torch.flatten(x, 1) # -> N, 16 * 8 * 8 = 1024
79     x = F.relu(self.fc1(x)) # -> N, 64
80     x = self.fc2(x) # -> N, 10
81     return x
82
83
84 class CNN_5(torch.nn.Module):
85     def __init__(self):
86         super().__init__()
87         self.conv1 = torch.nn.Conv2d(3, 8, 3, padding=1) # Conv-3x3x8
88         self.conv2 = torch.nn.Conv2d(8, 16, 3, padding=1) # Conv-3x3x16
89         self.conv3 = torch.nn.Conv2d(16, 8, 3, padding=1) # Conv-3x3x8
90         self.conv4 = torch.nn.Conv2d(8, 16, 3, padding=1) # Conv-3x3x16
91         self.conv5 = torch.nn.Conv2d(16, 16, 3, padding=1) # Conv-3x3x16
92         self.conv6 = torch.nn.Conv2d(16, 8, 3, padding=1) # Conv-3x3x8
93         self.pool = torch.nn.MaxPool2d(2, 2)
94         self.fc1 = torch.nn.Linear(8 * 8 * 8, 64)
95         self.fc2 = torch.nn.Linear(64, 10)
96
97     def forward(self, x):
98         # Input: N, 3, 32, 32
99         x = F.relu(self.conv1(x)) # -> N, 8, 32, 32
100        x = F.relu(self.conv2(x)) # -> N, 16, 32, 32
101        x = F.relu(self.conv3(x)) # -> N, 8, 32, 32
102        x = F.relu(self.conv4(x)) # -> N, 16, 32, 32
103        x = self.pool(x) # -> N, 16, 16, 16
104        x = F.relu(self.conv5(x)) # -> N, 16, 16, 16
105        x = F.relu(self.conv6(x)) # -> N, 8, 16, 16
106        x = self.pool(x) # -> N, 8, 8, 8
107
108        x = torch.flatten(x, 1) # -> N, 8 * 8 * 8 = 512
109        x = F.relu(self.fc1(x)) # -> N, 64
110        x = self.fc2(x) # -> N, 10
111        return x

```

#### 4) results.py:

```

1 import sys
2 import os
3 import json
4 import numpy as np
5
6 sys.path.append(os.path.abspath('_given'))
7
8 from utils import part3Plots, visualizeWeights
9
10
11 with open('part3/results/result_MLP1.json', 'r') as file:
12     results_MLP1 = json.load(file)
13 with open('part3/results/result_MLP2.json', 'r') as file:
14     results_MLP2 = json.load(file)
15 with open('part3/results/result_CNN3.json', 'r') as file:
16     results_CNN3 = json.load(file)
17 with open('part3/results/result_CNN4.json', 'r') as file:
18     results_CNN4 = json.load(file)
19 with open('part3/results/result_CNN5.json', 'r') as file:
20     results_CNN5 = json.load(file)
21
22 visualizeWeights(np.array(results_MLP1['weights']), save_dir="part3/results")
23 visualizeWeights(np.array(results_MLP2['weights']), save_dir="part3/results")
24 visualizeWeights(np.array(results_CNN3['weights']), save_dir="part3/results")
25 visualizeWeights(np.array(results_CNN4['weights']), save_dir="part3/results")
26 visualizeWeights(np.array(results_CNN5['weights']), save_dir="part3/results")
27
28
29

```

```
30 part3Plots([results_MLP1])
31 part3Plots([results_MLP2])
32 part3Plots([results_CNN3])
33 part3Plots([results_CNN4])
34 part3Plots([results_CNN5])
```



```

72         num_epochs)
73 print("MLP_1_ReLU_Training_and_evaluation_finished")
74
75 print("Tranining_of_MLP_1_Sigmoid_is_starting")
76 results_model_MLP1_Sigmoid = train_and_evaluate('mpl_1',
77         model_MLP1_Sigmoid,
78         train_loader,
79         test_loader,
80         learning_rate,
81         num_epochs)
82 print("MLP_1_Sigmoid_Training_and_evaluation_finished")
83
84 results_model_MLP1 = {
85     'name': 'MLP1',
86     'relu_loss_curve': results_model_MLP1_ReLU['loss_curve'],
87     'sigmoid_loss_curve': results_model_MLP1_Sigmoid['loss_curve'],
88     'relu_grad_curve': results_model_MLP1_ReLU['grad_curve'],
89     'sigmoid_grad_curve': results_model_MLP1_Sigmoid['grad_curve']
90 }
91
92 result_serializable_MLP1 = convert_to_serializable(results_model_MLP1)
93 # Save to JSON file
94 with open('./part4/results/result_MLP1.json', 'w') as f:
95     json.dump(result_serializable_MLP1, f, indent=4)
96
97
98
99 #####
100 #     MLP 2
101 #####
102
103 print("Tranining_of_MLP_2_ReLU_is_starting")
104 results_model_MLP2_ReLU = train_and_evaluate('mpl_2',
105         model_MLP2_ReLU,
106         train_loader,
107         test_loader,
108         learning_rate,
109         num_epochs)
110 print("MLP_2_ReLU_Training_and_evaluation_finished")
111
112 print("Tranining_of_MLP_2_Sigmoid_is_starting")
113 results_model_MLP2_Sigmoid = train_and_evaluate('mpl_2',
114         model_MLP2_Sigmoid,
115         train_loader,
116         test_loader,
117         learning_rate,
118         num_epochs)
119 print("MLP_2_Sigmoid_Training_and_evaluation_finished")
120
121 results_model_MLP2 = {
122     'name': 'MLP2',
123     'relu_loss_curve': results_model_MLP2_ReLU['loss_curve'],
124     'sigmoid_loss_curve': results_model_MLP2_Sigmoid['loss_curve'],
125     'relu_grad_curve': results_model_MLP2_ReLU['grad_curve'],
126     'sigmoid_grad_curve': results_model_MLP2_Sigmoid['grad_curve']
127 }
128
129 result_serializable_MLP2 = convert_to_serializable(results_model_MLP2)
130 # Save to JSON file
131 with open('./part4/results/result_MLP2.json', 'w') as f:
132     json.dump(result_serializable_MLP2, f, indent=4)
133
134
135
136
137 #####
138 #     CNN 3
139 #####
140
141 print("Tranining_of_CNN_3_ReLU_is_starting")
142 results_model_CNN3_ReLU = CNN_train_and_evaluate('cnn_3',
143         model_CNN3_ReLU,
144         train_loader,
145         test_loader,

```



```

146         learning_rate,
147         num_epochs)
148 print("CNN_3_ReLU_Training_and_evaluation_finished")
149
150 print("Tranining_of_CNN_3_Sigmoid_is_starting")
151 results_model_CNN3_Sigmoid = CNN_train_and_evaluate('cnn_3',
152         model_CNN3_Sigmoid,
153         train_loader,
154         test_loader,
155         learning_rate,
156         num_epochs)
157 print("CNN_3_Sigmoid_Training_and_evaluation_finished")
158
159 results_model_CNN3 = {
160     'name': 'CNN3',
161     'relu_loss_curve': results_model_CNN3_ReLU['loss_curve'],
162     'sigmoid_loss_curve': results_model_CNN3_Sigmoid['loss_curve'],
163     'relu_grad_curve': results_model_CNN3_ReLU['grad_curve'],
164     'sigmoid_grad_curve': results_model_CNN3_Sigmoid['grad_curve']
165 }
166
167 result_serializable_CNN3 = convert_to_serializable(results_model_CNN3)
168 # Save to JSON file
169 with open('./part4/results/result_CNN3.json', 'w') as f:
170     json.dump(result_serializable_CNN3, f, indent=4)
171
172
173
174
175 #####
176 # CNN 4
177 #####
178
179 print("Tranining_of_CNN_4_ReLU_is_starting")
180 results_model_CNN4_ReLU = CNN_train_and_evaluate('cnn_4',
181         model_CNN4_ReLU,
182         train_loader,
183         test_loader,
184         learning_rate,
185         num_epochs)
186 print("CNN_4_ReLU_Training_and_evaluation_finished")
187
188 print("Tranining_of_CNN_4_Sigmoid_is_starting")
189 results_model_CNN4_Sigmoid = CNN_train_and_evaluate('cnn_4',
190         model_CNN4_Sigmoid,
191         train_loader,
192         test_loader,
193         learning_rate,
194         num_epochs)
195 print("CNN_4_Sigmoid_Training_and_evaluation_finished")
196
197 results_model_CNN4 = {
198     'name': 'CNN4',
199     'relu_loss_curve': results_model_CNN4_ReLU['loss_curve'],
200     'sigmoid_loss_curve': results_model_CNN4_Sigmoid['loss_curve'],
201     'relu_grad_curve': results_model_CNN4_ReLU['grad_curve'],
202     'sigmoid_grad_curve': results_model_CNN4_Sigmoid['grad_curve']
203 }
204
205 result_serializable_CNN4 = convert_to_serializable(results_model_CNN4)
206 # Save to JSON file
207 with open('./part4/results/result_CNN4.json', 'w') as f:
208     json.dump(result_serializable_CNN4, f, indent=4)
209
210
211
212 #####
213 # CNN 5
214 #####
215
216 print("Tranining_of_CNN_5_ReLU_is_starting")
217 results_model_CNN5_ReLU = CNN_train_and_evaluate('cnn_5',
218         model_CNN5_ReLU,
219         train_loader,

```

```

220         test_loader,
221         learning_rate,
222         num_epochs)
223 print("CNN_5_ReLU_Training_and_evaluation_finished")
224
225 print("Tranining_of_CNN_5_Sigmoid_is_starting")
226 results_model_CNN5_Sigmoid = CNN_train_and_evaluate('cnn_5',
227             model_CNN5_Sigmoid,
228             train_loader,
229             test_loader,
230             learning_rate,
231             num_epochs)
232 print("CNN_5_Sigmoid_Training_and_evaluation_finished")
233
234 results_model_CNN5 = {
235     'name': 'CNN5',
236     'relu_loss_curve': results_model_CNN5_ReLU['loss_curve'],
237     'sigmoid_loss_curve': results_model_CNN5_Sigmoid['loss_curve'],
238     'relu_grad_curve': results_model_CNN5_ReLU['grad_curve'],
239     'sigmoid_grad_curve': results_model_CNN5_Sigmoid['grad_curve']
240 }
241
242 result_serializable_CNN5 = convert_to_serializable(results_model_CNN5)
243 # Save to JSON file
244 with open('./part4/results/result_CNN5.json', 'w') as f:
245     json.dump(result_serializable_CNN5, f, indent=4)

```

## 2) functions.py:

```

1 import torch
2 import numpy as np
3
4 def train_and_evaluate(model_name, in_model, in_train_loader, in_test_loader, learning_rate, num_epochs=15,
5     device='cuda'):
6     model = in_model.to(device)
7
8     # He initialization, good for ReLU
9     def init_weights(m):
10         if isinstance(m, torch.nn.Linear):
11             torch.nn.init.kaiming_uniform_(m.weight)
12             if m.bias is not None:
13                 m.bias.data.fill_(0.0)
14
15     model.apply(init_weights)
16
17     criterion = torch.nn.CrossEntropyLoss()
18     optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0.0)
19
20     training_loss = []
21     grad_curve = []
22
23     n_total_steps = len(in_train_loader)
24     for epoch in range(num_epochs):
25         running_loss = 0.0
26         model.train()
27
28         for i, (images, labels) in enumerate(in_train_loader):
29             images = images.reshape(-1, 3*32*32).to(device)
30             images = images + 0.007 * torch.randn_like(images) # Adversarial Training
31             labels = labels.to(device)
32
33             # Forward pass
34             outputs = model(images)
35             loss = criterion(outputs, labels)
36
37             # Backward pass
38             optimizer.zero_grad()
39             loss.backward()
40
41             # Compute gradient norm
42             total_grad_norm = 0.0
43             for param in model.parameters():
44                 if param.grad is not None:
45                     total_grad_norm += param.grad.norm().item() ** 2
46             total_grad_norm = total_grad_norm ** 0.5 # Square root of sum of squares

```

```

46     optimizer.step()
47
48     running_loss += loss.item()
49
50     # Log loss and gradient every 10 steps
51     if (i + 1) % 10 == 0:
52         training_loss.append(running_loss / 10)
53         grad_curve.append(total_grad_norm)
54         running_loss = 0.0
55
56     print(f"Epoch_{(epoch+1)}/{(num_epochs)}_completed.")
57
58     return {
59         'loss_curve': training_loss,
60         'grad_curve': grad_curve
61     }
62
63
64
65 def CNN_train_and_evaluate(model_name, in_model, in_train_loader, in_test_loader, learning_rate, num_epochs
66                             =15, device='cuda'):
67     model = in_model.to(device)
68
69     # He initialization for applicable layers
70     def init_weights(m):
71         if isinstance(m, torch.nn.Linear) or isinstance(m, torch.nn.Conv2d):
72             torch.nn.init.kaiming_uniform_(m.weight)
73             if m.bias is not None:
74                 m.bias.data.fill_(0.0)
75     model.apply(init_weights)
76
77     criterion = torch.nn.CrossEntropyLoss()
78     optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0.0)
79
80     training_loss = []
81     grad_curve = []
82
83     for epoch in range(num_epochs):
84         running_loss = 0.0
85         model.train()
86
87         for i, (images, labels) in enumerate(in_train_loader):
88             images, labels = images.to(device), labels.to(device)
89             images = images + 0.007 * torch.randn_like(images) # Adversarial Training
90
91             # Forward pass
92             outputs = model(images)
93             loss = criterion(outputs, labels)
94
95             # Backward pass and gradient computation
96             optimizer.zero_grad()
97             loss.backward()
98
99             # Compute gradient norm over all parameters
100             total_grad_norm = 0.0
101             for param in model.parameters():
102                 if param.grad is not None:
103                     total_grad_norm += param.grad.norm().item() ** 2
104             total_grad_norm = total_grad_norm ** 0.5
105
106             optimizer.step()
107
108             running_loss += loss.item()
109
110             # Record metrics every 10 batches
111             if (i + 1) % 10 == 0:
112                 avg_loss = running_loss / 10
113                 training_loss.append(avg_loss)
114                 grad_curve.append(total_grad_norm)
115                 running_loss = 0.0
116
117             print(f"Epoch_{(epoch+1)}/{(num_epochs)}_completed.")
118
119     # Return only the loss and gradient curves

```

```

119     return {
120         'loss_curve': training_loss,
121         'grad_curve': grad_curve
122     }
123
124
125 def convert_to_serializable(obj):
126     if isinstance(obj, torch.Tensor):
127         return obj.tolist() # Convert tensors to lists
128     if isinstance(obj, np.ndarray):
129         return obj.tolist() # Convert NumPy arrays to lists
130     if isinstance(obj, dict):
131         return {k: convert_to_serializable(v) for k, v in obj.items()} # Recursively convert dicts
132     if isinstance(obj, list):
133         return [convert_to_serializable(i) for i in obj] # Recursively convert lists
134     return obj # Return the object as is if it's already serializable

```

### 3) models.py:

```

1 import torch
2 import torch.nn.functional as F
3
4
5 class MLP_1_ReLU(torch.nn.Module):
6     def __init__(self):
7         super(MLP_1_ReLU, self).__init__()
8         self.fc1 = torch.nn.Linear(3 * 32 * 32, 32)
9         self.relu = torch.nn.ReLU()
10        self.fc2 = torch.nn.Linear(32, 10)
11
12    def forward(self, x):
13        out = self.fc1(x)
14        out = self.relu(out)
15        out = self.fc2(out)
16        return out
17
18 class MLP_1_Sigmoid(torch.nn.Module):
19     def __init__(self):
20         super(MLP_1_Sigmoid, self).__init__()
21         self.fc1 = torch.nn.Linear(3 * 32 * 32, 32)
22         self.sigmoid = torch.nn.Sigmoid()
23         self.fc2 = torch.nn.Linear(32, 10)
24
25    def forward(self, x):
26        out = self.fc1(x)
27        out = self.sigmoid(out)
28        out = self.fc2(out)
29        return out
30
31
32 class MLP_2_ReLU(torch.nn.Module):
33     def __init__(self):
34         super(MLP_2_ReLU, self).__init__()
35         self.fc1 = torch.nn.Linear(3 * 32 * 32, 32)
36         self.relu = torch.nn.ReLU()
37         self.fc2 = torch.nn.Linear(32, 64, bias=False)
38         self.fc3 = torch.nn.Linear(64, 10)
39
40    def forward(self, x):
41        out = self.fc1(x)
42        out = self.relu(out)
43        out = self.fc2(out)
44        out = self.relu(out)
45        out = self.fc3(out)
46        return out
47
48 class MLP_2_Sigmoid(torch.nn.Module):
49     def __init__(self):
50         super(MLP_2_Sigmoid, self).__init__()
51         self.fc1 = torch.nn.Linear(3 * 32 * 32, 32)
52         self.sigmoid = torch.nn.Sigmoid()
53         self.fc2 = torch.nn.Linear(32, 64, bias=False)
54         self.fc3 = torch.nn.Linear(64, 10)
55
56    def forward(self, x):

```

```

57     out = self.fc1(x)
58     out = self.sigmoid(out)
59     out = self.fc2(out)
60     out = self.sigmoid(out)
61     out = self.fc3(out)
62     return out
63
64
65 class CNN_3_ReLU(torch.nn.Module):
66     def __init__(self):
67         super().__init__()
68         self.conv1 = torch.nn.Conv2d(3, 16, 3, padding=1) # Conv-3x3x16
69         self.conv2 = torch.nn.Conv2d(16, 8, 5, padding=2) # Conv-5x5x8
70         self.conv3 = torch.nn.Conv2d(8, 16, 7, padding=3) # Conv-7x7x16
71         self.pool = torch.nn.MaxPool2d(2, 2)
72         self.fc1 = torch.nn.Linear(16 * 8 * 8, 64)
73         self.fc2 = torch.nn.Linear(64, 10)
74
75     def forward(self, x):
76         # Input: N, 3, 32, 32
77         x = F.relu(self.conv1(x)) # -> N, 16, 32, 32
78         x = F.relu(self.conv2(x)) # -> N, 8, 32, 32
79         x = self.pool(x) # -> N, 8, 16, 16
80         x = F.relu(self.conv3(x)) # -> N, 16, 16, 16
81         x = self.pool(x) # -> N, 16, 8, 8
82         x = torch.flatten(x, 1) # -> N, 16 * 8 * 8 = 1024
83         x = F.relu(self.fc1(x)) # -> N, 64
84         x = self.fc2(x) # -> N, 10
85         return x
86
87 class CNN_3_Sigmoid(torch.nn.Module):
88     def __init__(self):
89         super().__init__()
90         self.conv1 = torch.nn.Conv2d(3, 16, 3, padding=1) # Conv-3x3x16
91         self.conv2 = torch.nn.Conv2d(16, 8, 5, padding=2) # Conv-5x5x8
92         self.conv3 = torch.nn.Conv2d(8, 16, 7, padding=3) # Conv-7x7x16
93         self.pool = torch.nn.MaxPool2d(2, 2)
94         self.fc1 = torch.nn.Linear(16 * 8 * 8, 64)
95         self.fc2 = torch.nn.Linear(64, 10)
96
97     def forward(self, x):
98         # Input: N, 3, 32, 32
99         x = F.sigmoid(self.conv1(x)) # -> N, 16, 32, 32
100        x = F.sigmoid(self.conv2(x)) # -> N, 8, 32, 32
101        x = self.pool(x) # -> N, 8, 16, 16
102        x = F.sigmoid(self.conv3(x)) # -> N, 16, 16, 16
103        x = self.pool(x) # -> N, 16, 8, 8
104        x = torch.flatten(x, 1) # -> N, 16 * 8 * 8 = 1024
105        x = F.sigmoid(self.fc1(x)) # -> N, 64
106        x = self.fc2(x) # -> N, 10
107        return x
108
109 class CNN_4_ReLU(torch.nn.Module):
110     def __init__(self):
111         super().__init__()
112         self.conv1 = torch.nn.Conv2d(3, 16, 3, padding=1) # Conv-3x3x16
113         self.conv2 = torch.nn.Conv2d(16, 8, 3, padding=1) # Conv-3x3x8
114         self.conv3 = torch.nn.Conv2d(8, 16, 5, padding=2) # Conv-5x5x16
115         self.conv4 = torch.nn.Conv2d(16, 16, 5, padding=2) # Conv-5x5x16
116         self.pool = torch.nn.MaxPool2d(2, 2)
117         self.fc1 = torch.nn.Linear(16 * 8 * 8, 64)
118         self.fc2 = torch.nn.Linear(64, 10)
119
120     def forward(self, x):
121         # Input: N, 3, 32, 32
122         x = F.relu(self.conv1(x)) # -> N, 16, 32, 32
123         x = F.relu(self.conv2(x)) # -> N, 8, 32, 32
124         x = F.relu(self.conv3(x)) # -> N, 16, 32, 32
125         x = self.pool(x) # -> N, 16, 16, 16
126         x = F.relu(self.conv4(x)) # -> N, 16, 16, 16
127         x = self.pool(x) # -> N, 16, 8, 8
128
129         x = torch.flatten(x, 1) # -> N, 16 * 8 * 8 = 1024
130         x = F.relu(self.fc1(x)) # -> N, 64

```

```

131     x = self.fc2(x)                # -> N, 10
132     return x
133
134 class CNN_4_Sigmoid(torch.nn.Module):
135     def __init__(self):
136         super().__init__()
137         self.conv1 = torch.nn.Conv2d(3, 16, 3, padding=1)    # Conv-3x3x16
138         self.conv2 = torch.nn.Conv2d(16, 8, 3, padding=1)    # Conv-3x3x8
139         self.conv3 = torch.nn.Conv2d(8, 16, 5, padding=2)    # Conv-5x5x16
140         self.conv4 = torch.nn.Conv2d(16, 16, 5, padding=2)    # Conv-5x5x16
141         self.pool = torch.nn.MaxPool2d(2, 2)
142         self.fc1 = torch.nn.Linear(16 * 8 * 8, 64)
143         self.fc2 = torch.nn.Linear(64, 10)
144
145     def forward(self, x):
146         # Input: N, 3, 32, 32
147         x = F.sigmoid(self.conv1(x))    # -> N, 16, 32, 32
148         x = F.sigmoid(self.conv2(x))    # -> N, 8, 32, 32
149         x = F.sigmoid(self.conv3(x))    # -> N, 16, 32, 32
150         x = self.pool(x)                # -> N, 16, 16, 16
151         x = F.sigmoid(self.conv4(x))    # -> N, 16, 16, 16
152         x = self.pool(x)                # -> N, 16, 8, 8
153
154         x = torch.flatten(x, 1)         # -> N, 16 * 8 * 8 = 1024
155         x = F.sigmoid(self.fc1(x))       # -> N, 64
156         x = self.fc2(x)                 # -> N, 10
157         return x
158
159 class CNN_5_ReLU(torch.nn.Module):
160     def __init__(self):
161         super().__init__()
162         self.conv1 = torch.nn.Conv2d(3, 8, 3, padding=1)    # Conv-3x3x8
163         self.conv2 = torch.nn.Conv2d(8, 16, 3, padding=1)    # Conv-3x3x16
164         self.conv3 = torch.nn.Conv2d(16, 8, 3, padding=1)    # Conv-3x3x8
165         self.conv4 = torch.nn.Conv2d(8, 16, 3, padding=1)    # Conv-3x3x16
166         self.conv5 = torch.nn.Conv2d(16, 16, 3, padding=1)    # Conv-3x3x16
167         self.conv6 = torch.nn.Conv2d(16, 8, 3, padding=1)    # Conv-3x3x8
168         self.pool = torch.nn.MaxPool2d(2, 2)
169         self.fc1 = torch.nn.Linear(8 * 8 * 8, 64)
170         self.fc2 = torch.nn.Linear(64, 10)
171
172     def forward(self, x):
173         # Input: N, 3, 32, 32
174         x = F.relu(self.conv1(x))        # -> N, 8, 32, 32
175         x = F.relu(self.conv2(x))        # -> N, 16, 32, 32
176         x = F.relu(self.conv3(x))        # -> N, 8, 32, 32
177         x = F.relu(self.conv4(x))        # -> N, 16, 32, 32
178         x = self.pool(x)                 # -> N, 16, 16, 16
179         x = F.relu(self.conv5(x))        # -> N, 16, 16, 16
180         x = F.relu(self.conv6(x))        # -> N, 8, 16, 16
181         x = self.pool(x)                 # -> N, 8, 8, 8
182
183         x = torch.flatten(x, 1)          # -> N, 8 * 8 * 8 = 512
184         x = F.relu(self.fc1(x))           # -> N, 64
185         x = self.fc2(x)                  # -> N, 10
186         return x
187
188 class CNN_5_Sigmoid(torch.nn.Module):
189     def __init__(self):
190         super().__init__()
191         self.conv1 = torch.nn.Conv2d(3, 8, 3, padding=1)    # Conv-3x3x8
192         self.conv2 = torch.nn.Conv2d(8, 16, 3, padding=1)    # Conv-3x3x16
193         self.conv3 = torch.nn.Conv2d(16, 8, 3, padding=1)    # Conv-3x3x8
194         self.conv4 = torch.nn.Conv2d(8, 16, 3, padding=1)    # Conv-3x3x16
195         self.conv5 = torch.nn.Conv2d(16, 16, 3, padding=1)    # Conv-3x3x16
196         self.conv6 = torch.nn.Conv2d(16, 8, 3, padding=1)    # Conv-3x3x8
197         self.pool = torch.nn.MaxPool2d(2, 2)
198         self.fc1 = torch.nn.Linear(8 * 8 * 8, 64)
199         self.fc2 = torch.nn.Linear(64, 10)
200
201     def forward(self, x):
202         # Input: N, 3, 32, 32
203         x = F.sigmoid(self.conv1(x))     # -> N, 8, 32, 32
204         x = F.sigmoid(self.conv2(x))     # -> N, 16, 32, 32

```

```

205     x = F.sigmoid(self.conv3(x))    # -> N, 8, 32, 32
206     x = F.sigmoid(self.conv4(x))    # -> N, 16, 32, 32
207     x = self.pool(x)                # -> N, 16, 16, 16
208     x = F.sigmoid(self.conv5(x))    # -> N, 16, 16, 16
209     x = F.sigmoid(self.conv6(x))    # -> N, 8, 16, 16
210     x = self.pool(x)                # -> N, 8, 8, 8
211
212     x = torch.flatten(x, 1)          # -> N, 8 * 8 * 8 = 512
213     x = F.sigmoid(self.fc1(x))       # -> N, 64
214     x = self.fc2(x)                  # -> N, 10
215     return x

```

#### 4) results.py:

```

1  import sys
2  import os
3  import json
4  import numpy as np
5
6  sys.path.append(os.path.abspath('_given'))
7
8  from utils import part4Plots
9
10
11  with open('part4/results/result_MLP1.json', 'r') as file:
12      results_MLP1 = json.load(file)
13  with open('part4/results/result_MLP2.json', 'r') as file:
14      results_MLP2 = json.load(file)
15  with open('part4/results/result_CNN3.json', 'r') as file:
16      results_CNN3 = json.load(file)
17  with open('part4/results/result_CNN4.json', 'r') as file:
18      results_CNN4 = json.load(file)
19  with open('part4/results/result_CNN5.json', 'r') as file:
20      results_CNN5 = json.load(file)
21
22
23  part4Plots([results_MLP1])
24  part4Plots([results_MLP2])
25  part4Plots([results_CNN3])
26  part4Plots([results_CNN4])
27  part4Plots([results_CNN5])

```



1) *part5.py*:  $\Gamma$

```

73                                     True,          # Reinitialize Weights
74                                     device)
75 print("CNN_5_Training_and_evaluation_finished_with_lr=_0.1")
76
77 print("Tranining_of_CNN5_is_starting_with_lr=_0.01")
78 result_CNN5_01 = CNN_train_and_evaluate('cnn_5',
79                                     model_CNN5,
80                                     train_loader,
81                                     test_loader,
82                                     0.01,          # Learning Rate
83                                     20,             # Number of epochs
84                                     100,            # Target Validation
85                                     True,          # Reinitialize Weights
86                                     device)
87 print("CNN_5_Training_and_evaluation_finishedwith_lr=_0.01")
88
89 print("Tranining_of_CNN5_is_starting_with_lr=_0.001")
90 result_CNN5_001 = CNN_train_and_evaluate('cnn_5',
91                                     model_CNN5,
92                                     train_loader,
93                                     test_loader,
94                                     0.001,         # Learning Rate
95                                     20,             # Number of epochs
96                                     100,            # Target Validation
97                                     True,          # Reinitialize Weights
98                                     device)
99 print("CNN_5_Training_and_evaluation_finished_with_lr=_0.001")
100
101 results_model_CNN5 = {
102     'name': 'CNN5',
103     'loss_curve_1': result_CNN5_1['loss_curve'],
104     'loss_curve_01': result_CNN5_01['loss_curve'],
105     'loss_curve_001': result_CNN5_001['loss_curve'],
106     'val_acc_curve_1': result_CNN5_1['validation_accuracy'],
107     'val_acc_curve_01': result_CNN5_01['validation_accuracy'],
108     'val_acc_curve_001': result_CNN5_001['validation_accuracy']
109 }
110
111 result_serializable_CNN5 = convert_to_serializable(results_model_CNN5)
112 # Save to JSON file
113 with open('./part5/results/result_CNN5_diff_lr.json', 'w') as f:
114     json.dump(result_serializable_CNN5, f, indent=4)
115
116
117
118
119
120
121 #####
122 # Scheduling Learning First Try
123 #####
124
125 print("Tranining_of_Scheduling_CNN5_is_starting_with_lr=_0.1")
126 result_CNN5_sch_part1 = CNN_train_and_evaluate('cnn_5',
127                                     model_CNN5_scheduled,
128                                     train_loader,
129                                     test_loader,
130                                     0.1,           # Learning Rate
131                                     30,             # Number of Epochs
132                                     65,             # Target Validancy
133                                     True,          # Reinitialize Weights
134                                     device)
135 print("CNN_5_Training_and_evaluation_finished_with_lr=_0.1,_without_achieved_desired_validancy")
136
137 print("Tranining_of_CNN5_is_starting_with_lr=_0.01")
138 result_CNN5_sch_part2 = CNN_train_and_evaluate('cnn_5',
139                                     model_CNN5_scheduled,
140                                     train_loader,
141                                     test_loader,
142                                     0.01,          # Learning Rate
143                                     30,             # Number of Epochs
144                                     70,             # Target Validancy
145                                     False,         # Reinitialize Weights
146                                     device)

```

```

147 print("CNN_5_Training_and_evaluation_finished_with_lr=_0.01,_without_achieved_desired_validancy")
148
149 print("Tranining_of_CNN5_is_starting_with_lr=_0.001")
150 result_CNN5_sch_part2 = CNN_train_and_evaluate('cnn_5',
151                                             model_CNN5_scheduled,
152                                             train_loader,
153                                             test_loader,
154                                             0.001,          # Learning Rate
155                                             30,             # Number of Epochs
156                                             100,            # Target Validancy
157                                             False,         # Reinitialize Weightss
158                                             device)
159 print("CNN_5_Training_and_evaluation_finishedwith_lr=_0.001")
160
161
162 results_model_CNN5_sch = []
163 for result in [result_CNN5_sch_part1, result_CNN5_sch_part2]:
164     results_model_CNN5_sch += result['validation_accuracy']
165
166 result_serializable_CNN5_sch = convert_to_serializable(results_model_CNN5_sch)
167 # Save to JSON file
168 with open('./part5/results/result_CNN5_sch_1.json', 'w') as f:
169     json.dump(result_serializable_CNN5_sch, f, indent=4)

```

## 2) functions.py:

```

1 import torch
2 import numpy as np
3
4
5 def CNN_train_and_evaluate(model_name, in_model, in_train_loader, in_test_loader, learning_rate, num_epochs
6                             =15, target_val_accuracy=None, reinitialize=True ,device='cuda'):
7     model = in_model.to(device)
8
9     if reinitialize:
10         # He initialization for ReLU activations for applicable layers
11         def init_weights(m):
12             if isinstance(m, (torch.nn.Linear, torch.nn.Conv2d)):
13                 torch.nn.init.kaiming_uniform_(m.weight)
14                 if m.bias is not None:
15                     m.bias.data.fill_(0.0)
16         model.apply(init_weights)
17
18     criterion = torch.nn.CrossEntropyLoss()
19     optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0.0)
20
21     loss_curve = []
22     validation_accuracy = []
23
24     # Preload all test data once
25     test_images, test_labels = [], []
26     for images, labels in in_test_loader:
27         test_images.append(images.to(device))
28         test_labels.append(labels.to(device))
29     test_images = torch.cat(test_images)
30     test_labels = torch.cat(test_labels)
31     test_batch_size = in_test_loader.batch_size # Preserve original batch size
32
33     early_stop = False # Flag to check early stopping condition
34
35     for epoch in range(num_epochs):
36         running_loss = 0.0
37         model.train()
38
39         for i, (images, labels) in enumerate(in_train_loader):
40             images, labels = images.to(device), labels.to(device)
41             images = images + 0.007 * torch.randn_like(images) # Adversarial Training: adding noise to images
42
43             # Forward + backward
44             outputs = model(images)
45             loss = criterion(outputs, labels)
46             loss.backward()
47             optimizer.step()
48             optimizer.zero_grad()

```

```

49     running_loss += loss.item()
50
51     # Validation every 10 batches
52     if (i + 1) % 10 == 0:
53         avg_loss = running_loss / 10
54         loss_curve.append(avg_loss)
55         running_loss = 0.0
56
57     model.eval()
58     val_correct = 0
59     with torch.no_grad():
60         for batch_start in range(0, len(test_images), test_batch_size):
61             batch_images = test_images[batch_start:batch_start+test_batch_size]
62             batch_labels = test_labels[batch_start:batch_start+test_batch_size]
63             outputs = model(batch_images)
64             _, predicted = torch.max(outputs, 1)
65             val_correct += (predicted == batch_labels).sum().item()
66         val_acc = 100.0 * val_correct / len(test_labels)
67         validation_accuracy.append(val_acc)
68
69     # Check if early stopping criterion is met
70     if target_val_accuracy is not None and val_acc >= target_val_accuracy:
71         print(f"Early_stopping_triggered:_Validation_accuracy_{val_acc:.2f}%_reached_target_of_{target_val_accuracy}%")
72         early_stop = True
73         break # Exit inner loop
74
75     model.train()
76
77     print(f'[{epoch+1}/{num_epochs}]_Epoch_completed._Last_Validation_Accuracy:_{val_acc:.2f}%')
78
79     if early_stop:
80         break # Exit outer loop if early stopping was triggered
81
82 # Return only the loss curve and validation accuracy data
83 return {
84     'loss_curve': loss_curve,
85     'validation_accuracy': validation_accuracy
86 }
87
88
89
90 def convert_to_serializable(obj):
91     if isinstance(obj, torch.Tensor):
92         return obj.tolist() # Convert tensors to lists
93     if isinstance(obj, np.ndarray):
94         return obj.tolist() # Convert NumPy arrays to lists
95     if isinstance(obj, dict):
96         return {k: convert_to_serializable(v) for k, v in obj.items()} # Recursively convert dicts
97     if isinstance(obj, list):
98         return [convert_to_serializable(i) for i in obj] # Recursively convert lists
99     return obj # Return the object as is if it's already serializable

```

### 3) models.py:

```

1 import torch
2 import torch.nn.functional as F
3
4
5 class CNN_4(torch.nn.Module):
6     def __init__(self):
7         super().__init__()
8         self.conv1 = torch.nn.Conv2d(3, 16, 3, padding=1) # Conv-3x3x16
9         self.conv2 = torch.nn.Conv2d(16, 8, 3, padding=1) # Conv-3x3x8
10        self.conv3 = torch.nn.Conv2d(8, 16, 5, padding=2) # Conv-5x5x16
11        self.conv4 = torch.nn.Conv2d(16, 16, 5, padding=2) # Conv-5x5x16
12        self.pool = torch.nn.MaxPool2d(2, 2)
13        self.fc1 = torch.nn.Linear(16 * 8 * 8, 64)
14        self.fc2 = torch.nn.Linear(64, 10)
15
16    def forward(self, x):
17        # Input: N, 3, 32, 32
18        x = F.relu(self.conv1(x)) # -> N, 16, 32, 32
19        x = F.relu(self.conv2(x)) # -> N, 8, 32, 32
20        x = F.relu(self.conv3(x)) # -> N, 16, 32, 32

```

```

21     x = self.pool(x)                # -> N, 16, 16, 16
22     x = F.relu(self.conv4(x))       # -> N, 16, 16, 16
23     x = self.pool(x)                # -> N, 16, 8, 8
24
25     x = torch.flatten(x, 1)         # -> N, 16 * 8 * 8 = 1024
26     x = F.relu(self.fc1(x))         # -> N, 64
27     x = self.fc2(x)                 # -> N, 10
28     return x
29
30
31 class CNN_5(torch.nn.Module):
32     def __init__(self):
33         super().__init__()
34         self.conv1 = torch.nn.Conv2d(3, 8, 3, padding=1)    # Conv-3x3x8
35         self.conv2 = torch.nn.Conv2d(8, 16, 3, padding=1)   # Conv-3x3x16
36         self.conv3 = torch.nn.Conv2d(16, 8, 3, padding=1)   # Conv-3x3x8
37         self.conv4 = torch.nn.Conv2d(8, 16, 3, padding=1)   # Conv-3x3x16
38         self.conv5 = torch.nn.Conv2d(16, 16, 3, padding=1)  # Conv-3x3x16
39         self.conv6 = torch.nn.Conv2d(16, 8, 3, padding=1)   # Conv-3x3x8
40         self.pool = torch.nn.MaxPool2d(2, 2)
41         self.fc1 = torch.nn.Linear(8 * 8 * 8, 64)
42         self.fc2 = torch.nn.Linear(64, 10)
43
44     def forward(self, x):
45         # Input: N, 3, 32, 32
46         x = F.relu(self.conv1(x))    # -> N, 8, 32, 32
47         x = F.relu(self.conv2(x))    # -> N, 16, 32, 32
48         x = F.relu(self.conv3(x))    # -> N, 8, 32, 32
49         x = F.relu(self.conv4(x))    # -> N, 16, 32, 32
50         x = self.pool(x)              # -> N, 16, 16, 16
51         x = F.relu(self.conv5(x))    # -> N, 16, 16, 16
52         x = F.relu(self.conv6(x))    # -> N, 8, 16, 16
53         x = self.pool(x)              # -> N, 8, 8, 8
54
55         x = torch.flatten(x, 1)       # -> N, 8 * 8 * 8 = 512
56         x = F.relu(self.fc1(x))       # -> N, 64
57         x = self.fc2(x)               # -> N, 10
58         return x

```

#### 4) results.py:

```

1  import sys
2  import os
3  import json
4  import numpy as np
5
6  sys.path.append(os.path.abspath('_given'))
7
8  from utils import part5Plots
9
10
11  with open('part5/results/result_CNN5_diff_lr.json', 'r') as file:
12      results_CNN5 = json.load(file)
13
14
15
16
17  part5Plots([results_CNN5])

```