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**

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.

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

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

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

$$= \frac{2e^{4x}+2e^{2x}-2e^{4x}+2e^{2x}}{(e^{2x}+1)^2} = \frac{4e^{2x}}{(e^{2x}+1)^2}$$

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

$$= 1 - \tanh^2(x)$$

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

2) Sigmoid Function:

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

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

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

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

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

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

$$= sigmoid(x) \cdot (1 - sigmoid(x))$$

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

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.

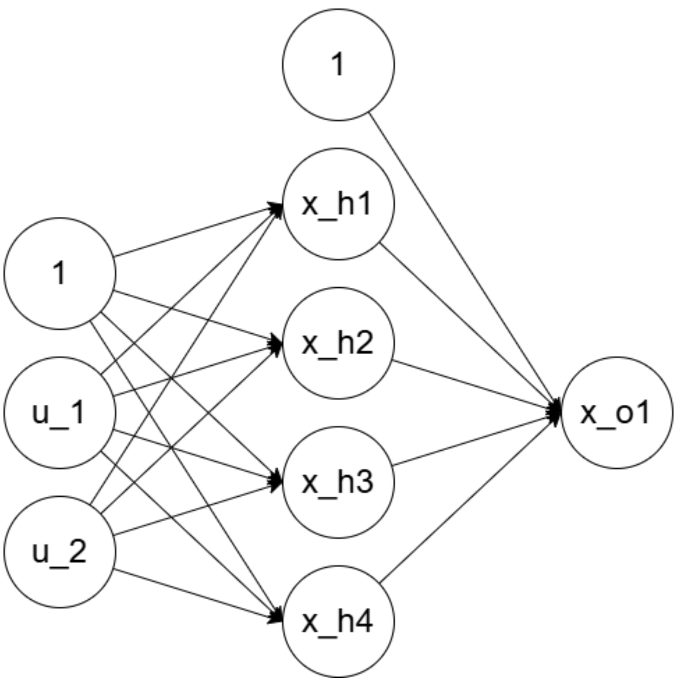

Fig. 1. Structure of neural network that is used.

A. Parameter Definitions

1) Input Layer Matrices:

$$\mathbf{U}^k = \begin{bmatrix} u_1^k & u_2^k \end{bmatrix}^T$$

$$\mathbf{X}_u^k = \begin{bmatrix} \mathbf{U}^1 & \mathbf{U}^2 & \cdots & \mathbf{U}^k \end{bmatrix}^T$$
$$\mathbf{X}_{u_{bias}}^k = \begin{bmatrix} \mathbf{1} & \mathbf{U}^1 & \mathbf{U}^2 & \cdots & \mathbf{U}^k \end{bmatrix}^T$$

2) Hidden Layer Matrices:

$$\mathbf{X}_h^k = \begin{bmatrix} x_{h1}^k & x_{h2}^k & x_{h3}^k & x_{h4}^k \end{bmatrix}^T$$

$$\mathbf{X}_h^k = \begin{bmatrix} \mathbf{X}_h^1 & \mathbf{X}_h^2 & \cdots & \mathbf{X}_h^k \end{bmatrix}^T$$
$$\mathbf{X}_{h_{bias}}^k = \begin{bmatrix} \mathbf{1} & \mathbf{X}_h^1 & \mathbf{X}_h^2 & \cdots & \mathbf{X}_h^k \end{bmatrix}^T$$

3) Output Layer Matrices:

$$\mathbf{X}_o^k = \begin{bmatrix} x_{o1}^1 & x_{o1}^2 & \cdots & x_{o1}^k \end{bmatrix}$$

4) Weight Matrices:

$\mathbf{W}_L$: (2x4) matrix input to hidden layer
$\mathbf{W}_{L_{bias}}$: (3x4) matrix input to hidden layer with bias
$\mathbf{W}_o$: (1x4) matrix hidden to output layer
$\mathbf{W}_{o_{bias}}$: (1x5) matrix hidden to output layer with bias

5) Activation Function:

$f()$: Activation function
$f^{'}(\cdot)$: Derivetive of activation function

6) Error and Delta Matrices:

$\mathbf{e}_o$: output layer error
δ_o : output layer delta error
$\mathbf{e}_h$: hidden layer error
δ_h : hidden layer delta error

7) Target Output:

$$\mathbf{Y}^k = \begin{bmatrix} y^1 & y^2 & \cdots & y^k \end{bmatrix}$$

8) Target Output:

μ : Learning rate

B. Forward Propagation

$$\mathbf{X}_{u_{bias}}^k = \begin{bmatrix} 1 & \mathbf{X}_u^k \end{bmatrix}$$

$$\mathbf{X}_h^k = f\left(\mathbf{X}_{u_{bias}}^k \times \mathbf{W}_{L_{bias}}\right)$$

$$\mathbf{X}_{h_{bias}}^k = \begin{bmatrix} 1 & \mathbf{X}_h^k \end{bmatrix}$$

$$\mathbf{X}_o^k = f\left(\mathbf{X}_u^k \times \mathbf{W}_{o_{bias}}\right)$$

C. Back Propagation

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

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

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

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

$$\mathbf{W}_{L_{bias}}(n+1) = \mathbf{W}_{L_{bias}}(n) - \mu \cdot \left(\left(\mathbf{X}_{u_{bias}}^k\right)^T \times \delta_h\right)$$

$$\mathbf{W}_{o_{bias}}(n+1) = \mathbf{W}_{o_{bias}}(n) - \mu \cdot \left(\left(\mathbf{X}_{h_{bias}}^k\right)^T \times \delta_o\right)$$

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.

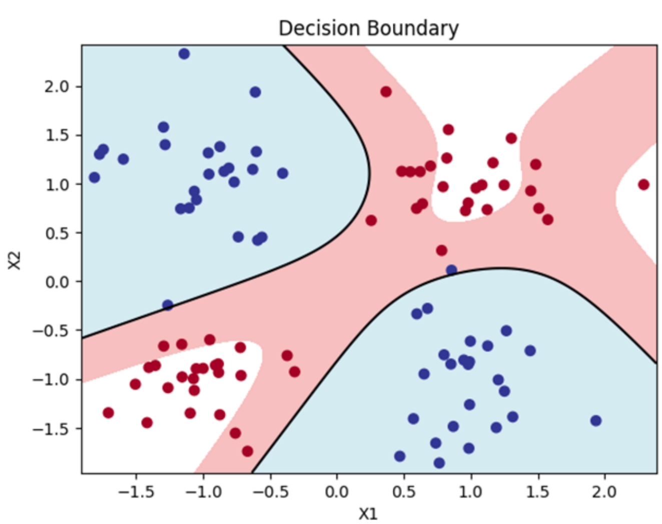

Fig. 2. Activation Function: Tanh, Epoch: 50000, Accuracy: 0.99

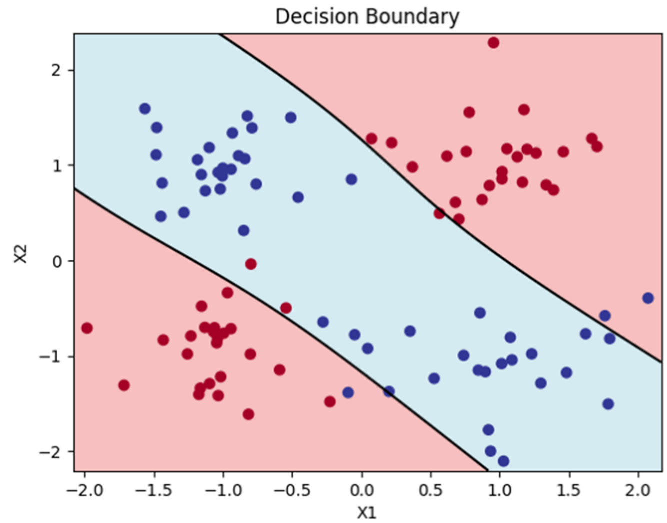

Fig. 3. Activation Function: Sigmoid, Epoch: 50000, Accuracy: 0.94

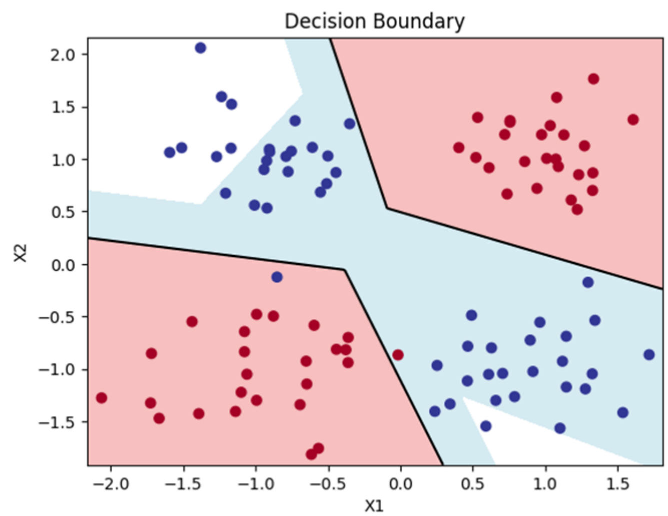

Fig. 4. Activation Function: ReLU, Epoch: 50000, Accuracy: 0.98

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.

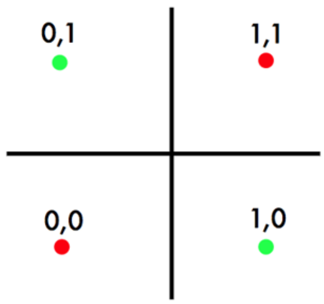

Fig. 5. XOR Problem

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.

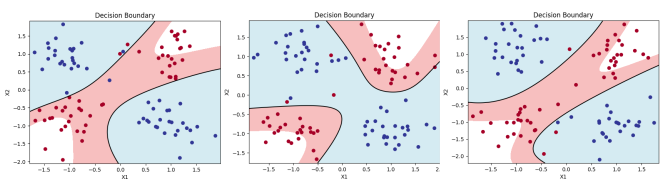

Fig. 6. XOR Problem

III. Implementing a Convolutional Layer with NumPy

A. Preliminary

Shape of input matrix and kernel matrix are given.

$$\begin{aligned}\text{Input Shape} &= \text{BatchSize} \times \text{InputChannels}\\ &\times \text{InputHeight} \times \text{InputWidth}\end{aligned}$$

$$\begin{aligned}\text{KernelShape} &= \text{OutputChannels} \times \text{InputChannels}\\ &\times \text{KernelHeight} \times \text{KernelWidth}\end{aligned}$$

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:

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

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 32×32 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-3×3×16, ReLU, Conv-5×5×8, ReLU,
MaxPool-2×2, Conv-7×7×16, MaxPool-2×2]
+ PredictionLayer

'cnn-4' : [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]
+ PredictionLayer

'cnn-5' : [Conv-3×3×8, ReLU, Conv-3×3×16, ReLU,
Conv-3×3×8, ReLU, Conv-3×3×16, ReLU,
MaxPool-2×2, Conv-3×3×16,
ReLU, Conv-3×3×8, ReLU, MaxPool-2×2]
+ PredictionLayer

B. Functions

Algorihms 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 mpl-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:

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

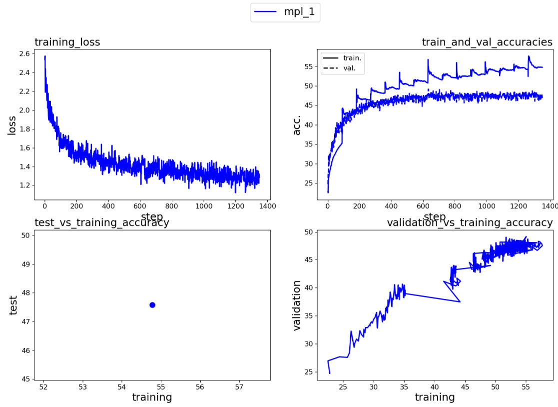

Fig. 8. Results of mlp-1 for part 3

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:

$$98336 + 330 = 98666$$

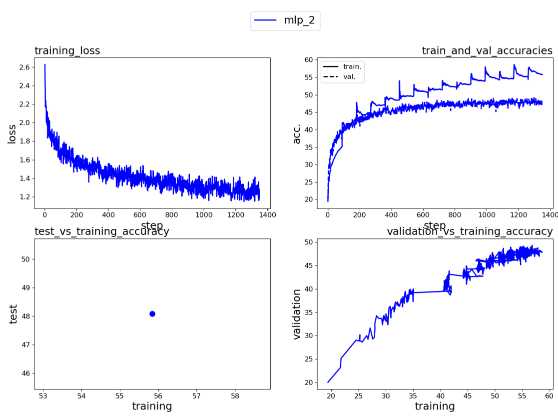

Fig. 9. Results of mlp-2 for part 3

When we analyze mpl-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

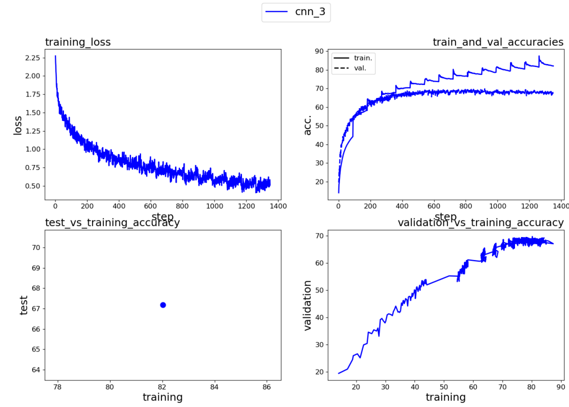

Fig. 10. Results of cnn-3 for part 3

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 overfitting. 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.

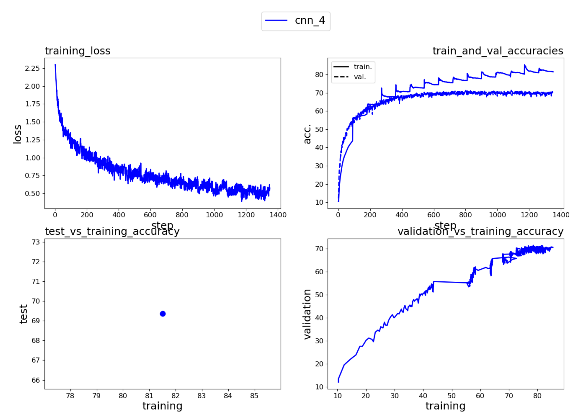

Fig. 11. Results of cnn-4 for part 3

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 3×3 and 5×5 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.

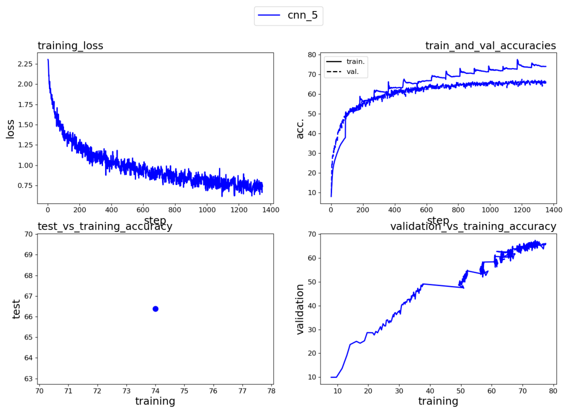

Fig. 12. Results of cnn-5 for part 3

CNN-3 and CNN-4 differ primarily in their convolutional layer configurations and filter sizes. CNN-3 employs a structure that begins with a 3×3×16 convolution, followed by a 5×5×8 layer, max pooling, and then a 7×7×16 convolution before a second pooling stage and the prediction layer. In contrast, CNN-4 incorporates additional convolutional layers—starting with a 3×3×16 layer followed by a 3×3×8, then a 5×5×16 layer, max pooling, and another 5×5×16 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 3×3 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.

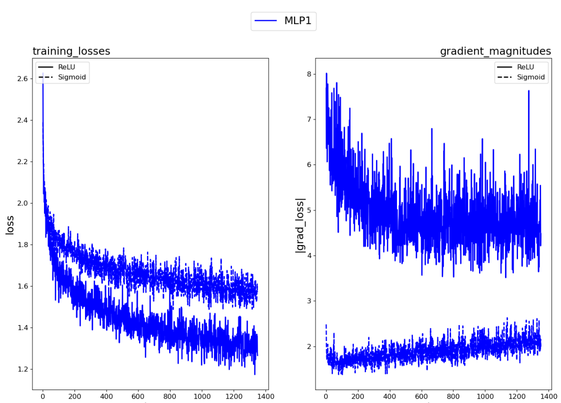

Fig. 13. Results of mlp-1 for part 4

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

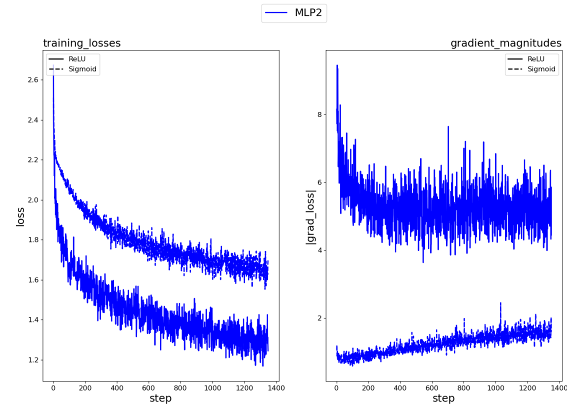

Fig. 14. Results of mlp-2 for part 4

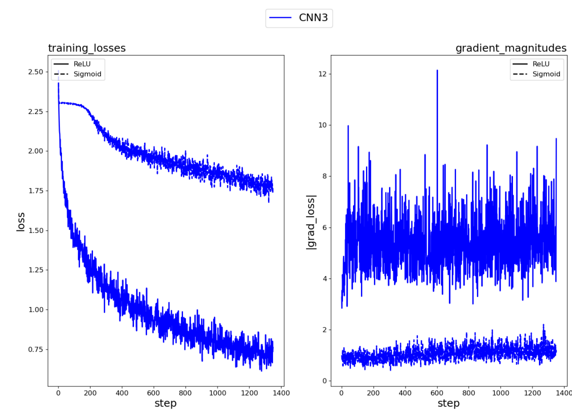

Fig. 15. Results of cnn-3 for part 4

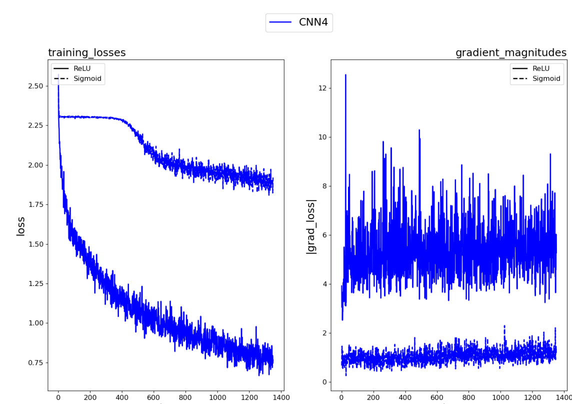

Fig. 16. Results of cnn-4 for part 4

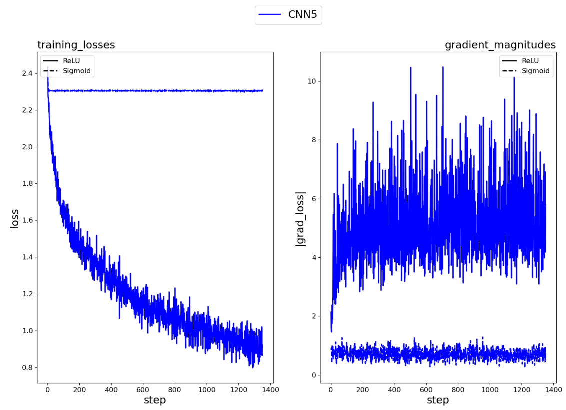

Fig. 17. Results of cnn-5 for part 4

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.

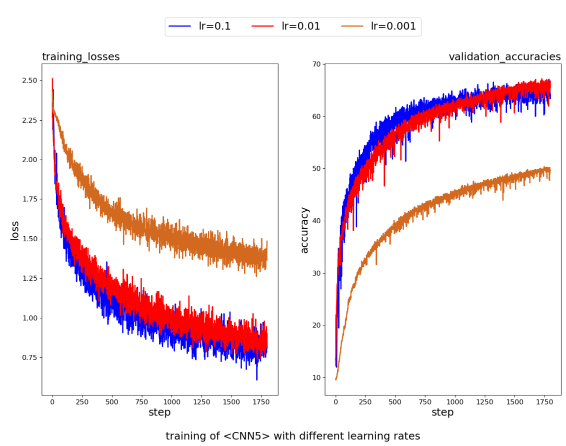

Fig. 18. Results of cnn-5 for part 5

APPENDIX

A. Part 1

1) part1.py:

```
import sys
import os
sys.path.append(os.path.abspath('_given'))

import numpy as np
from matplotlib import pyplot as plt
from utils import part1CreateDataset, part1PlotBoundary


class MLP:
  def __init__(self, input_size, hidden_size, output_size):
    self.input_size = input_size
    self.hidden_size = hidden_size
    self.output_size = output_size

    # Initialize weights and biases
    self.weights_input_hidden = np.random.randn(self.input_size + 1, self.hidden_size) # 3x4 matrix
    #self.bias_hidden = np.zeros((1 , self.hidden_size)) # 1x4

    self.weights_hidden_output = np.random.randn(self.hidden_size + 1, self.output_size) # 5x1 matrix
    #self.bias_output = np.zeros((1 , self.output_size)) # 1x1


  def f_activation(self, x):
    return (np.exp(2*x)-1)/(np.exp(2*x)+1)  # Tanh
    #return 1 / (1 + np.exp(-x))              # Sigmoid
    #return np.maximum(0, x)                   # ReLU

  def f_activation_derivative(self, x):
    return (1 - np.square(x))                           # Tanh
    #return x * (1 - x)                         # Sigmoid
    #return (x>0).astype(float)                 # ReLU

  def forward(self, inputs):
    # Forward pass through the network
    inputs = np.hstack((np.ones((inputs.shape[0], 1)), inputs)) # Kx3 matrix, first column is 1
    self.hidden_output =  self.f_activation(np.matmul( inputs , self.weights_input_hidden)) # Kx4

    self.hidden_output_w_bias = np.hstack((np.ones((self.hidden_output.shape[0], 1)), self.hidden_output))
        # Kx5 matrix, first column is 1
    self.output = self.f_activation(np.matmul(self.hidden_output_w_bias , self.weights_hidden_output)) #
        Kx1
    return self.output


  def backward(self, inputs, targets, learning_rate):
    # Backward pass through the network

    # Compute output layer error and its gradient
    output_error = (self.output - targets) * 2.0 / targets.size  # Kx1
    output_delta = output_error * self.f_activation_derivative(self.output) * 2.0 # Kx1

    # Compute hidden layer error and its gradient
    hidden_error = np.matmul(output_error , self.weights_hidden_output[1:].T) # Kx4
    hidden_delta = hidden_error * self.f_activation_derivative(self.hidden_output) # Kx4

    inputs = np.hstack((np.ones((inputs.shape[0], 1)), inputs)) # Kx3 matrix, first column is 1

    # Update weights and biases
    self.weights_hidden_output -= learning_rate * np.matmul( self.hidden_output_w_bias.T , output_delta) #
        5x1
    self.weights_input_hidden -= learning_rate * np.matmul( inputs.T , hidden_delta) # 3x4 matrix


# Generate the dataset
x_train, y_train, x_val, y_val = part1CreateDataset(train_samples=1000, val_samples=100, std=0.4)

# Define neural network parameters
input_size = 2
hidden_size = 4
```

```python
output_size = 1
learning_rate = 0.001

# Create neural network
nn = MLP(input_size, hidden_size, output_size)

# Train the neural network
for epoch in range(50000):

  # Forward propagation
  output = nn.forward(x_train)

  # Backpropagation
  nn.backward(x_train, y_train, learning_rate)

  # Print the loss (MSE) every 1000 epochs
  if epoch % 1000 == 0:
    # Compute predictions for the entire validation set
    val_predictions = nn.forward(x_val)
    loss = 0.5 * np.mean(np.square(y_val - val_predictions))
    print(f'Epoch {epoch}: Loss = {loss}')


# Test the trained neural network
val_predictions = nn.forward(x_val)

y_predict = ((val_predictions > 0.5).astype(int)).reshape(-1,1)
accuracy = np.mean((y_predict == y_val).astype(int))
print(f'{accuracy*100}% of test examples classified correctly.')

part1PlotBoundary(x_val, y_val, nn)
```

B. Part 2

1) part2.py:

```python
import numpy as np
import os

# Assuming samples_7.npy is in /content/drive/MyDrive/EE449/HW1
file_path = os.path.join('_given\data\data', 'samples_7.npy')

# input shape: [batch size, input_channels, input_height, input_width]
input = np.load(file_path)
# input shape: [output_channels, input_channels, filter_height, filter width]
# Assuming kernel.npy is also in /content/drive/MyDrive/EE449/HW1
kernel_path = os.path.join('_given\data\data', 'kernel.npy')
kernel = np.load(kernel_path) # Load kernel.npy from the specified path

import sys
sys.path.append('_given')
from utils import part2Plots


def my_conv2d(input, kernel):
    """
    Performs a 2D convolution (forward propagation) with no padding and stride 1.

    Args:
        input (np.ndarray): Input data of shape (batch_size, in_channels, in_height, in_width).
        kernel (np.ndarray): Convolutional kernel of shape (out_channels, in_channels, k_height, k_width).

    Returns:
        np.ndarray: The result of the convolution, of shape (batch_size, out_channels, out_height, out_width),
                    where:
                        out_height = in_height - kernel_height + 1,
                        out_width  = in_width - kernel_width + 1.
    """
    batch_size, in_channels, in_height, in_width = input.shape
    out_channels, kernel_in_channels, kernel_height, kernel_width = kernel.shape

    # Check if the input channels and kernel channels match
    if in_channels != kernel_in_channels:
        print("error")
        raise ValueError("The number of input channels must match the kernel's input channels.")

    # Compute output dimensions
    out_height = in_height - kernel_height + 1
    out_width = in_width - kernel_width + 1

    # Initialize the output tensor with zeros
    output = np.zeros((batch_size, out_channels, out_height, out_width))

    # Iterate over the batch, output channels, and spatial locations to apply the convolution filter
    for b in range(batch_size):
        for oc in range(out_channels):
            for i in range(out_height):
                for j in range(out_width):
                    # Extract the current patch from the input
                    patch = input[b, :, i:i+kernel_height, j:j+kernel_width]
                    # Perform elementwise multiplication and sum over the channel and kernel dimensions
                    output[b, oc, i, j] = np.sum(patch * kernel[oc, :, :, :])

    print("Convolution is done!")
    return output


out = my_conv2d(input, kernel)
part2Plots(out)
```

C. Part 3

1) part3.py:

```
import torch
import torchvision
import torchvision.transforms as transforms
from models import MLP_1,MLP_2,CNN_3,CNN_4,CNN_5
from functions import train_and_evaluate, CNN_train_and_evaluate,convert_to_serializable
import json

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print("You are using device:" , device)

# Hyper-parameters
num_epochs = 15
batch_size = 50
learning_rate = 0.001

#################################################################################
# Transformations
#################################################################################

transform = transforms.Compose([
  transforms.ToTensor(),
  transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),
])

# Load the dataset (assuming it's a custom dataset with 100,000 images)
full_dataset = torchvision.datasets.CIFAR10(root='./data',
                                            train=True,
                                            download=True,
                                            transform=transform)

# Compute split sizes
train_size = int(0.9 * len(full_dataset))   # 90% training
test_size = len(full_dataset) - train_size  # 10% testing

# Split dataset
train_dataset, test_dataset = torch.utils.data.random_split(full_dataset, [train_size, test_size])

# Create DataLoaders
train_loader = torch.utils.data.DataLoader(train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)


# Instantiate the model
model_MLP1 = MLP_1()
model_MLP2 = MLP_2()
model_CNN3 = CNN_3()
model_CNN4 = CNN_4()
model_CNN5 = CNN_5()

print("Tranining of MLP1 is starting")
result_MLP1 = train_and_evaluate('mpl_1',
                                 model_MLP1,
                                 train_loader,
                                 test_loader,
                                 learning_rate,
                                 num_epochs)
print("MLP_1 Training and evaluation finished")
PATH = './part3/results/MLP_1.pth'
torch.save(model_MLP1.state_dict(), PATH)

result_serializable_MLP1 = convert_to_serializable(result_MLP1)
# Save to JSON file
with open('./part3/results/result_MLP1.json', 'w') as f:
    json.dump(result_serializable_MLP1, f, indent=4)
```

```python
print("Tranining of MLP2 is starting")
result_MLP2 = train_and_evaluate('mlp_2',
                                 model_MLP2,
                                 train_loader,
                                 test_loader,
                                 learning_rate,
                                 num_epochs)
print("MLP_2 Training and evaluation finished")
PATH = './part3/results/MLP_2.pth'
torch.save(model_MLP2.state_dict(), PATH)

result_serializable_MLP2 = convert_to_serializable(result_MLP2)
# Save to JSON file
with open('./part3/results/result_MLP2.json', 'w') as f:
    json.dump(result_serializable_MLP2, f, indent=4)


print("Tranining of CNN3 is starting")
result_CNN3 = CNN_train_and_evaluate('cnn_3',
                                     model_CNN3,
                                     train_loader,
                                     test_loader,
                                     learning_rate,
                                     num_epochs,
                                     device)
print("CNN_3 Training and evaluation finished")
PATH = './part3/results/CNN_3.pth'
torch.save(model_CNN3.state_dict(), PATH)

result_serializable_CNN3 = convert_to_serializable(result_CNN3)
# Save to JSON file
with open('./part3/results/result_CNN3.json', 'w') as f:
    json.dump(result_serializable_CNN3, f, indent=4)


print("Tranining of CNN4 is starting")
result_CNN4 = CNN_train_and_evaluate('cnn_4',
                                     model_CNN4,
                                     train_loader,
                                     test_loader,
                                     learning_rate ,
                                     num_epochs,
                                     device)
print("CNN_4 Training and evaluation finished")
PATH = './part3/results/CNN_4.pth'
torch.save(model_CNN4.state_dict(), PATH)

result_serializable_CNN4 = convert_to_serializable(result_CNN4)
# Save to JSON file
with open('./part3/results/result_CNN4.json', 'w') as f:
    json.dump(result_serializable_CNN4, f, indent=4)


print("Tranining of CNN5 is starting")
result_CNN5 = CNN_train_and_evaluate('cnn_5',
                                     model_CNN5,
                                     train_loader,
                                     test_loader,
                                     learning_rate,
                                     num_epochs,
                                     device)
print("CNN_5 Training and evaluation finished")
PATH = './part3/results/CNN_5.pth'
torch.save(model_CNN5.state_dict(), PATH)

result_serializable_CNN5 = convert_to_serializable(result_CNN5)
# Save to JSON file
with open('./part3/results/result_CNN5.json', 'w') as f:
    json.dump(result_serializable_CNN5, f, indent=4)
```

2) functions.py:

```python
import torch
import numpy as np

```

```python
def train_and_evaluate(model_name, in_model, in_train_loader, in_test_loader, learning_rate, num_epochs=15,
     device='cuda'):
  model = in_model.to(device)

  # He initialization, good for ReLU
  def init_weights(m):
    if isinstance(m, torch.nn.Linear):
      torch.nn.init.kaiming_uniform_(m.weight)
      if m.bias is not None:
        m.bias.data.fill_(0.0)

  # Apply He initialization to all applicable layers
  model.apply(init_weights)


  criterion = torch.nn.CrossEntropyLoss()
  optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

  training_loss = []
  training_accuracy = []
  validation_accuracy = []

  # Preload and flatten test data
  test_images, test_labels = [], []
  for images, labels in in_test_loader:
    test_images.append(images.reshape(-1, 3*32*32).to(device))
    test_labels.append(labels.to(device))
  test_images = torch.cat(test_images, dim=0)
  test_labels = torch.cat(test_labels, dim=0)
  test_batch_size = in_test_loader.batch_size  # Use the same batch size as the test loader

  n_total_steps = len(in_train_loader)
  for epoch in range(num_epochs):
    running_loss = 0.0
    correct = 0
    total = 0
    model.train()

    for i, (images, labels) in enumerate(in_train_loader):
      images = images.reshape(-1, 3*32*32).to(device)
      images = images + 0.007 * torch.randn_like(images) # Adversarial Training
      labels = labels.to(device)

      # Forward pass
      outputs = model(images)
      loss = criterion(outputs, labels)

      # Backward pass and optimize
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()

      running_loss += loss.item()
      _, predicted = torch.max(outputs.data, 1)
      total += labels.size(0)
      correct += (predicted == labels).sum().item()

      # Log training metrics every 10 steps
      if (i + 1) % 10 == 0:
        training_loss.append(running_loss / 10)
        training_accuracy.append(100 * correct / total)
        running_loss = 0.0

        # Calculate validation accuracy using preloaded data
        model.eval()
        val_correct = 0
        val_total = 0
        with torch.no_grad():
          for i_batch in range(0, len(test_images), test_batch_size):
            batch_images = test_images[i_batch:i_batch+test_batch_size]
            batch_labels = test_labels[i_batch:i_batch+test_batch_size]
            outputs = model(batch_images)
            _, predicted = torch.max(outputs.data, 1)
            val_total += batch_labels.size(0)
```

```
            val_correct += (predicted == batch_labels).sum().item()
          val_acc = 100.0 * val_correct / val_total
        validation_accuracy.append(val_acc)
        model.train()

    # Print epoch statistics
    epoch_train_acc = 100 * correct / total
    print(f'[{epoch + 1}] Training Accuracy: {epoch_train_acc:.2f}%, Validation Accuracy: {val_acc:.2f}%')

  # Final evaluation using preloaded test data
  model.eval()
  n_correct = 0
  with torch.no_grad():
    for i_batch in range(0, len(test_images), test_batch_size):
      batch_images = test_images[i_batch:i_batch+test_batch_size]
      batch_labels = test_labels[i_batch:i_batch+test_batch_size]
      outputs = model(batch_images)
      _, predicted = torch.max(outputs.data, 1)
      n_correct += (predicted == batch_labels).sum().item()
  acc = 100.0 * n_correct / len(test_labels)
  print(f'Accuracy of the model: {acc:.2f} %')

  first_layer_weights = model.fc1.weight.data.cpu().numpy()

  return {
    'name': model_name,
    'loss_curve': training_loss,
    'train_acc_curve': training_accuracy,
    'val_acc_curve': validation_accuracy,
    'test_acc': acc,
    'weights': first_layer_weights
  }

def CNN_train_and_evaluate(model_name, in_model, in_train_loader, in_test_loader, learning_rate, num_epochs=15, device='cuda'):
  model = in_model.to(device)

  # He initialization, good for ReLU
  def init_weights(m):
    if isinstance(m, torch.nn.Linear):
      torch.nn.init.kaiming_uniform_(m.weight)
      if m.bias is not None:
        m.bias.data.fill_(0.0)
      elif isinstance(m, torch.nn.Conv2d):
        torch.nn.init.kaiming_uniform_(m.weight)
        if m.bias is not None:
          m.bias.data.fill_(0.0)

    # Apply He initialization to all applicable layers
    model.apply(init_weights)

  criterion = torch.nn.CrossEntropyLoss()
  optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

  training_loss = []
  training_accuracy = []
  validation_accuracy = []

  # Preload all test data once
  test_images, test_labels = [], []
  for images, labels in in_test_loader:
    test_images.append(images.to(device))
    test_labels.append(labels.to(device))
  test_images = torch.cat(test_images)
  test_labels = torch.cat(test_labels)
  test_batch_size = in_test_loader.batch_size  # Preserve original batch size

  for epoch in range(num_epochs):
    running_loss = 0.0
    correct = 0
    total = 0
    model.train()

    for i, (images, labels) in enumerate(in_train_loader):
```

```python
      images, labels = images.to(device), labels.to(device)
      images = images + 0.007 * torch.randn_like(images) # Adversarial Training

      # Forward + backward
      outputs = model(images)
      loss = criterion(outputs, labels)
      loss.backward()
      optimizer.step()
      optimizer.zero_grad()

      # Update metrics
      running_loss += loss.item()
      _, predicted = torch.max(outputs, 1)
      total += labels.size(0)
      correct += (predicted == labels).sum().item()

      # Validation every 10 batches
      if (i + 1) % 10 == 0:
        # Store training metrics
        training_loss.append(running_loss / 10)
        training_accuracy.append(100 * correct / total)
        running_loss = 0.0

        # Fast validation using preloaded data
        model.eval()
        val_correct = 0
        with torch.no_grad():
          # Process in original batch sizes
          for batch_start in range(0, len(test_images), test_batch_size):
            batch_images = test_images[batch_start:batch_start+test_batch_size]
            batch_labels = test_labels[batch_start:batch_start+test_batch_size]
            outputs = model(batch_images)
            _, predicted = torch.max(outputs, 1)
            val_correct += (predicted == batch_labels).sum().item()

        val_acc = 100.0 * val_correct / len(test_labels)
        validation_accuracy.append(val_acc)
        model.train()

    # Epoch statistics
    epoch_acc = 100 * correct / total
    print(f'[{epoch+1}] Train Acc: {epoch_acc:.2f}%, Val Acc: {val_acc:.2f}%')

  # Final evaluation with preloaded data
  model.eval()
  final_correct = 0
  with torch.no_grad():
    for batch_start in range(0, len(test_images), test_batch_size):
      batch_images = test_images[batch_start:batch_start+test_batch_size]
      batch_labels = test_labels[batch_start:batch_start+test_batch_size]
      outputs = model(batch_images)
      final_correct += (torch.max(outputs, 1)[1] == batch_labels).sum().item()

  acc = 100.0 * final_correct / len(test_labels)
  print(f'Final Accuracy: {acc:.2f}%')

  first_layer_weights = model.conv1.weight.data.cpu().numpy()

  return {
    'name': model_name,
    'loss_curve': training_loss,
    'train_acc_curve': training_accuracy,
    'val_acc_curve': validation_accuracy,
    'test_acc': acc,
    'weights': first_layer_weights
  }


def convert_to_serializable(obj):
    if isinstance(obj, torch.Tensor):
        return obj.tolist()  # Convert tensors to lists
    if isinstance(obj, np.ndarray):
        return obj.tolist()  # Convert NumPy arrays to lists
    if isinstance(obj, dict):
```

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

3) models.py:

```python
import torch
import torch.nn.functional as F


class MLP_1(torch.nn.Module):
  def __init__(self):
    super(MLP_1, self).__init__()
    self.fc1 = torch.nn.Linear(3 * 32 * 32, 32)
    self.relu = torch.nn.ReLU()
    self.fc2 = torch.nn.Linear(32, 10)

  def forward(self, x):
    out = self.fc1(x)
    out = self.relu(out)
    out = self.fc2(out)
    return out


class MLP_2(torch.nn.Module):
  def __init__(self):
    super(MLP_2, self).__init__()
    self.fc1 = torch.nn.Linear(3 * 32 * 32, 32)
    self.relu = torch.nn.ReLU()
    self.fc2 = torch.nn.Linear(32, 64, bias=False)
    self.fc3 = torch.nn.Linear(64, 10)

  def forward(self, x):
    out = self.fc1(x)
    out = self.relu(out)
    out = self.fc2(out)
    out = self.relu(out)
    out = self.fc3(out)
    return out


class CNN_3(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.conv1 = torch.nn.Conv2d(3, 16, 3, padding=1)  # Conv-3x3x16
    self.conv2 = torch.nn.Conv2d(16, 8, 5, padding=2)  # Conv-5x5x8
    self.conv3 = torch.nn.Conv2d(8, 16, 7, padding=3)  # Conv-7x7x16
    self.pool = torch.nn.MaxPool2d(2, 2)
    self.fc1 = torch.nn.Linear(16 * 8 * 8, 64)
    self.fc2 = torch.nn.Linear(64, 10)

  def forward(self, x):
    # Input: N, 3, 32, 32
    x = F.relu(self.conv1(x))   # -> N, 16, 32, 32
    x = F.relu(self.conv2(x))   # -> N, 8, 32, 32
    x = self.pool(x)            # -> N, 8, 16, 16
    x = F.relu(self.conv3(x))   # -> N, 16, 16, 16
    x = self.pool(x)            # -> N, 16, 8, 8
    x = torch.flatten(x, 1)     # -> N, 16 * 8 * 8 = 1024
    x = F.relu(self.fc1(x))     # -> N, 64
    x = self.fc2(x)             # -> N, 10
    return x

class CNN_4(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.conv1 = torch.nn.Conv2d(3, 16, 3, padding=1)   # Conv-3x3x16
    self.conv2 = torch.nn.Conv2d(16, 8, 3, padding=1)  # Conv-3x3x8
    self.conv3 = torch.nn.Conv2d(8, 16, 5, padding=2)  # Conv-5x5x16
    self.conv4 = torch.nn.Conv2d(16, 16, 5, padding=2)  # Conv-5x5x16
    self.pool = torch.nn.MaxPool2d(2, 2)
    self.fc1 = torch.nn.Linear(16 * 8 * 8, 64)
    self.fc2 = torch.nn.Linear(64, 10)

```

```python
  def forward(self, x):
    # Input: N, 3, 32, 32
    x = F.relu(self.conv1(x))   # -> N, 16, 32, 32
    x = F.relu(self.conv2(x))   # -> N, 8, 32, 32
    x = F.relu(self.conv3(x))   # -> N, 16, 32, 32
    x = self.pool(x)            # -> N, 16, 16, 16
    x = F.relu(self.conv4(x))   # -> N, 16, 16, 16
    x = self.pool(x)           # -> N, 16, 8, 8

    x = torch.flatten(x, 1)     # -> N, 16 * 8 * 8 = 1024
    x = F.relu(self.fc1(x))     # -> N, 64
    x = self.fc2(x)             # -> N, 10
    return x


class CNN_5(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.conv1 = torch.nn.Conv2d(3, 8, 3, padding=1)   # Conv-3x3x8
    self.conv2 = torch.nn.Conv2d(8, 16, 3, padding=1)  # Conv-3x3x16
    self.conv3 = torch.nn.Conv2d(16, 8, 3, padding=1)  # Conv-3x3x8
    self.conv4 = torch.nn.Conv2d(8, 16, 3, padding=1)  # Conv-3x3x16
    self.conv5 = torch.nn.Conv2d(16, 16, 3, padding=1)  # Conv-3x3x16
    self.conv6 = torch.nn.Conv2d(16, 8, 3, padding=1)  # Conv-3x3x8
    self.pool = torch.nn.MaxPool2d(2, 2)
    self.fc1 = torch.nn.Linear(8 * 8 * 8, 64)
    self.fc2 = torch.nn.Linear(64, 10)

  def forward(self, x):
    # Input: N, 3, 32, 32
    x = F.relu(self.conv1(x))   # -> N, 8, 32, 32
    x = F.relu(self.conv2(x))   # -> N, 16, 32, 32
    x = F.relu(self.conv3(x))   # -> N, 8, 32, 32
    x = F.relu(self.conv4(x))   # -> N, 16, 32, 32
    x = self.pool(x)            # -> N, 16, 16, 16
    x = F.relu(self.conv5(x))   # -> N, 16, 16, 16
    x = F.relu(self.conv6(x))   # -> N, 8, 16, 16
    x = self.pool(x)            # -> N, 8, 8, 8

    x = torch.flatten(x, 1)     # -> N, 8 * 8 * 8 = 512
    x = F.relu(self.fc1(x))     # -> N, 64
    x = self.fc2(x)             # -> N, 10
    return x
```

4) results.py:

```python
import sys
import os
import json
import numpy as np

sys.path.append(os.path.abspath('_given'))

from utils import part3Plots, visualizeWeights


with open('part3/results/result_MLP1.json', 'r') as file:
    results_MLP1 = json.load(file)
with open('part3/results/result_MLP2.json', 'r') as file:
    results_MLP2 = json.load(file)
with open('part3/results/result_CNN3.json', 'r') as file:
    results_CNN3 = json.load(file)
with open('part3/results/result_CNN4.json', 'r') as file:
    results_CNN4 = json.load(file)
with open('part3/results/result_CNN5.json', 'r') as file:
    results_CNN5 = json.load(file)

visualizeWeights(np.array(results_MLP1['weights']), save_dir="part3/results")
visualizeWeights(np.array(results_MLP2['weights']), save_dir="part3/results")
visualizeWeights(np.array(results_CNN3['weights']), save_dir="part3/results")
visualizeWeights(np.array(results_CNN4['weights']), save_dir="part3/results")
visualizeWeights(np.array(results_CNN5['weights']), save_dir="part3/results")
```

```
part3Plots([results_MLP1])
part3Plots([results_MLP2])
part3Plots([results_CNN3])
part3Plots([results_CNN4])
part3Plots([results_CNN5])
```

D. Part 4

1) part4.py:

```python
import torch
import torchvision
import torchvision.transforms as transforms
from models import MLP_1_ReLU,MLP_1_Sigmoid,MLP_2_ReLU,MLP_2_Sigmoid,CNN_3_ReLU,CNN_3_Sigmoid,CNN_4_ReLU,
    CNN_4_Sigmoid,CNN_5_ReLU,CNN_5_Sigmoid
from functions import train_and_evaluate, CNN_train_and_evaluate,convert_to_serializable
import json

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print("You are using device:" , device)

# Hyper-parameters
num_epochs = 15
batch_size = 50
learning_rate = 0.01

#################################################################################
# Transformations
#################################################################################

transform = transforms.Compose([
  transforms.ToTensor(),
  transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),
])

# Load the dataset (assuming it's a custom dataset with 100,000 images)
full_dataset = torchvision.datasets.CIFAR10(root='./data',
                                            train=True,
                                            download=True,
                                            transform=transform)

# Compute split sizes
train_size = int(0.9 * len(full_dataset))   # 90% training
test_size = len(full_dataset) - train_size  # 10% testing

# Split dataset
train_dataset, test_dataset = torch.utils.data.random_split(full_dataset, [train_size, test_size])

# Create DataLoaders
train_loader = torch.utils.data.DataLoader(train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)


# Instantiate the model
model_MLP1_ReLU = MLP_1_ReLU()
model_MLP2_ReLU = MLP_2_ReLU()
model_CNN3_ReLU = CNN_3_ReLU()
model_CNN4_ReLU = CNN_4_ReLU()
model_CNN5_ReLU = CNN_5_ReLU()

model_MLP1_Sigmoid = MLP_1_Sigmoid()
model_MLP2_Sigmoid = MLP_2_Sigmoid()
model_CNN3_Sigmoid = CNN_3_Sigmoid()
model_CNN4_Sigmoid = CNN_4_Sigmoid()
model_CNN5_Sigmoid = CNN_5_Sigmoid()

####################################################################
#    MLP 1
####################################################################

print("Tranining of MLP_1_ReLU is starting")
results_model_MLP1_ReLU = train_and_evaluate('mpl_1',
                                 model_MLP1_ReLU,
                                 train_loader,
                                 test_loader,
                                 learning_rate,
```

```
                                  num_epochs)
print("MLP_1_ReLU_Training_and_evaluation_finished")

print("Tranining_of_MLP_1_Sigmoid_is_starting")
results_model_MLP1_Sigmoid = train_and_evaluate('mpl_1',
                                  model_MLP1_Sigmoid,
                                  train_loader,
                                  test_loader,
                                  learning_rate,
                                  num_epochs)
print("MLP_1_Sigmoid_Training_and_evaluation_finished")

results_model_MLP1 = {
    'name': 'MLP1',
    'relu_loss_curve': results_model_MLP1_ReLU['loss_curve'],
    'sigmoid_loss_curve': results_model_MLP1_Sigmoid['loss_curve'],
    'relu_grad_curve': results_model_MLP1_ReLU['grad_curve'],
    'sigmoid_grad_curve': results_model_MLP1_Sigmoid['grad_curve']
}

result_serializable_MLP1 = convert_to_serializable(results_model_MLP1)
# Save to JSON file
with open('./part4/results/result_MLP1.json', 'w') as f:
    json.dump(result_serializable_MLP1, f, indent=4)



######################################################################
#    MLP 2
######################################################################

print("Tranining_of_MLP_2_ReLU_is_starting")
results_model_MLP2_ReLU = train_and_evaluate('mpl_2',
                                  model_MLP2_ReLU,
                                  train_loader,
                                  test_loader,
                                  learning_rate,
                                  num_epochs)
print("MLP_2_ReLU_Training_and_evaluation_finished")

print("Tranining_of_MLP_2_Sigmoid_is_starting")
results_model_MLP2_Sigmoid = train_and_evaluate('mpl_2',
                                  model_MLP2_Sigmoid,
                                  train_loader,
                                  test_loader,
                                  learning_rate,
                                  num_epochs)
print("MLP_2_Sigmoid_Training_and_evaluation_finished")

results_model_MLP2 = {
    'name': 'MLP2',
    'relu_loss_curve': results_model_MLP2_ReLU['loss_curve'],
    'sigmoid_loss_curve': results_model_MLP2_Sigmoid['loss_curve'],
    'relu_grad_curve': results_model_MLP2_ReLU['grad_curve'],
    'sigmoid_grad_curve': results_model_MLP2_Sigmoid['grad_curve']
}

result_serializable_MLP2 = convert_to_serializable(results_model_MLP2)
# Save to JSON file
with open('./part4/results/result_MLP2.json', 'w') as f:
    json.dump(result_serializable_MLP2, f, indent=4)




######################################################################
#    CNN 3
######################################################################

print("Tranining_of_CNN_3_ReLU_is_starting")
results_model_CNN3_ReLU = CNN_train_and_evaluate('cnn_3',
                                  model_CNN3_ReLU,
                                  train_loader,
                                  test_loader,
```

```
                                  learning_rate,
                                  num_epochs)
print("CNN 3 ReLU Training and evaluation finished")

print("Tranining of CNN 3 Sigmoid is starting")
results_model_CNN3_Sigmoid = CNN_train_and_evaluate('cnn_3',
                                  model_CNN3_Sigmoid,
                                  train_loader,
                                  test_loader,
                                  learning_rate,
                                  num_epochs)
print("CNN 3 Sigmoid Training and evaluation finished")

results_model_CNN3 = {
    'name': 'CNN3',
    'relu_loss_curve': results_model_CNN3_ReLU['loss_curve'],
    'sigmoid_loss_curve': results_model_CNN3_Sigmoid['loss_curve'],
    'relu_grad_curve': results_model_CNN3_ReLU['grad_curve'],
    'sigmoid_grad_curve': results_model_CNN3_Sigmoid['grad_curve']
}

result_serializable_CNN3 = convert_to_serializable(results_model_CNN3)
# Save to JSON file
with open('./part4/results/result_CNN3.json', 'w') as f:
    json.dump(result_serializable_CNN3, f, indent=4)




####################################################################
#    CNN 4
####################################################################

print("Tranining of CNN 4 ReLU is starting")
results_model_CNN4_ReLU = CNN_train_and_evaluate('cnn_4',
                                  model_CNN4_ReLU,
                                  train_loader,
                                  test_loader,
                                  learning_rate,
                                  num_epochs)
print("CNN 4 ReLU Training and evaluation finished")

print("Tranining of CNN 4 Sigmoid is starting")
results_model_CNN4_Sigmoid = CNN_train_and_evaluate('cnn_4',
                                  model_CNN4_Sigmoid,
                                  train_loader,
                                  test_loader,
                                  learning_rate,
                                  num_epochs)
print("CNN 4 Sigmoid Training and evaluation finished")

results_model_CNN4 = {
    'name': 'CNN4',
    'relu_loss_curve': results_model_CNN4_ReLU['loss_curve'],
    'sigmoid_loss_curve': results_model_CNN4_Sigmoid['loss_curve'],
    'relu_grad_curve': results_model_CNN4_ReLU['grad_curve'],
    'sigmoid_grad_curve': results_model_CNN4_Sigmoid['grad_curve']
}

result_serializable_CNN4 = convert_to_serializable(results_model_CNN4)
# Save to JSON file
with open('./part4/results/result_CNN4.json', 'w') as f:
    json.dump(result_serializable_CNN4, f, indent=4)



####################################################################
#    CNN 5
####################################################################

print("Tranining of CNN 5 ReLU is starting")
results_model_CNN5_ReLU = CNN_train_and_evaluate('cnn_5',
                                  model_CNN5_ReLU,
                                  train_loader,
```

```
                                  test_loader,
                                  learning_rate,
                                  num_epochs)
print("CNN_5_ReLU Training and evaluation finished")

print("Tranining of CNN_5_Sigmoid is starting")
results_model_CNN5_Sigmoid = CNN_train_and_evaluate('cnn_5',
                                  model_CNN5_Sigmoid,
                                  train_loader,
                                  test_loader,
                                  learning_rate,
                                  num_epochs)
print("CNN_5_Sigmoid Training and evaluation finished")

results_model_CNN5 = {
    'name': 'CNN5',
    'relu_loss_curve': results_model_CNN5_ReLU['loss_curve'],
    'sigmoid_loss_curve': results_model_CNN5_Sigmoid['loss_curve'],
    'relu_grad_curve': results_model_CNN5_ReLU['grad_curve'],
    'sigmoid_grad_curve': results_model_CNN5_Sigmoid['grad_curve']
}

result_serializable_CNN5 = convert_to_serializable(results_model_CNN5)
# Save to JSON file
with open('./part4/results/result_CNN5.json', 'w') as f:
    json.dump(result_serializable_CNN5, f, indent=4)
```

2) functions.py:

```
import torch
import numpy as np

def train_and_evaluate(model_name, in_model, in_train_loader, in_test_loader, learning_rate, num_epochs=15, device='cuda'):
  model = in_model.to(device)

  # He initialization, good for ReLU
  def init_weights(m):
    if isinstance(m, torch.nn.Linear):
      torch.nn.init.kaiming_uniform_(m.weight)
      if m.bias is not None:
        m.bias.data.fill_(0.0)

  model.apply(init_weights)

  criterion = torch.nn.CrossEntropyLoss()
  optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0.0)

  training_loss = []
  grad_curve = []

  n_total_steps = len(in_train_loader)
  for epoch in range(num_epochs):
    running_loss = 0.0
    model.train()

    for i, (images, labels) in enumerate(in_train_loader):
      images = images.reshape(-1, 3*32*32).to(device)
      images = images + 0.007 * torch.randn_like(images)  # Adversarial Training
      labels = labels.to(device)

      # Forward pass
      outputs = model(images)
      loss = criterion(outputs, labels)

      # Backward pass
      optimizer.zero_grad()
      loss.backward()

      # Compute gradient norm
      total_grad_norm = 0.0
      for param in model.parameters():
        if param.grad is not None:
          total_grad_norm += param.grad.norm().item() ** 2
      total_grad_norm = total_grad_norm ** 0.5  # Square root of sum of squares
```

```
            optimizer.step()

            running_loss += loss.item()

            # Log loss and gradient every 10 steps
            if (i + 1) % 10 == 0:
                training_loss.append(running_loss / 10)
                grad_curve.append(total_grad_norm)
                running_loss = 0.0

        print(f"Epoch [{epoch+1}/{num_epochs}] completed.")

    return {
        'loss_curve': training_loss,
        'grad_curve': grad_curve
    }


def CNN_train_and_evaluate(model_name, in_model, in_train_loader, in_test_loader, learning_rate, num_epochs
    =15, device='cuda'):
    model = in_model.to(device)

    # He initialization for applicable layers
    def init_weights(m):
        if isinstance(m, torch.nn.Linear) or isinstance(m, torch.nn.Conv2d):
            torch.nn.init.kaiming_uniform_(m.weight)
            if m.bias is not None:
                m.bias.data.fill_(0.0)
    model.apply(init_weights)

    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0.0)

    training_loss = []
    grad_curve = []

    for epoch in range(num_epochs):
        running_loss = 0.0
        model.train()

        for i, (images, labels) in enumerate(in_train_loader):
            images, labels = images.to(device), labels.to(device)
            images = images + 0.007 * torch.randn_like(images)  # Adversarial Training

            # Forward pass
            outputs = model(images)
            loss = criterion(outputs, labels)

            # Backward pass and gradient computation
            optimizer.zero_grad()
            loss.backward()

            # Compute gradient norm over all parameters
            total_grad_norm = 0.0
            for param in model.parameters():
                if param.grad is not None:
                    total_grad_norm += param.grad.norm().item() ** 2
            total_grad_norm = total_grad_norm ** 0.5

            optimizer.step()

            running_loss += loss.item()

            # Record metrics every 10 batches
            if (i + 1) % 10 == 0:
                avg_loss = running_loss / 10
                training_loss.append(avg_loss)
                grad_curve.append(total_grad_norm)
                running_loss = 0.0

        print(f"Epoch [{epoch+1}/{num_epochs}] completed.")

    # Return only the loss and gradient curves
```

```python
  return {
    'loss_curve': training_loss,
    'grad_curve': grad_curve
  }


def convert_to_serializable(obj):
    if isinstance(obj, torch.Tensor):
        return obj.tolist()  # Convert tensors to lists
    if isinstance(obj, np.ndarray):
        return obj.tolist()  # Convert NumPy arrays to lists
    if isinstance(obj, dict):
        return {k: convert_to_serializable(v) for k, v in obj.items()}  # Recursively convert dicts
    if isinstance(obj, list):
        return [convert_to_serializable(i) for i in obj]  # Recursively convert lists
    return obj  # Return the object as is if it's already serializable
```

3) models.py:

```python
import torch
import torch.nn.functional as F


class MLP_1_ReLU(torch.nn.Module):
  def __init__(self):
    super(MLP_1_ReLU, self).__init__()
    self.fc1 = torch.nn.Linear(3 * 32 * 32, 32)
    self.relu = torch.nn.ReLU()
    self.fc2 = torch.nn.Linear(32, 10)

  def forward(self, x):
    out = self.fc1(x)
    out = self.relu(out)
    out = self.fc2(out)
    return out

class MLP_1_Sigmoid(torch.nn.Module):
  def __init__(self):
    super(MLP_1_Sigmoid, self).__init__()
    self.fc1 = torch.nn.Linear(3 * 32 * 32, 32)
    self.sigmoid = torch.nn.Sigmoid()
    self.fc2 = torch.nn.Linear(32, 10)

  def forward(self, x):
    out = self.fc1(x)
    out = self.sigmoid(out)
    out = self.fc2(out)
    return out


class MLP_2_ReLU(torch.nn.Module):
  def __init__(self):
    super(MLP_2_ReLU, self).__init__()
    self.fc1 = torch.nn.Linear(3 * 32 * 32, 32)
    self.relu = torch.nn.ReLU()
    self.fc2 = torch.nn.Linear(32, 64, bias=False)
    self.fc3 = torch.nn.Linear(64, 10)

  def forward(self, x):
    out = self.fc1(x)
    out = self.relu(out)
    out = self.fc2(out)
    out = self.relu(out)
    out = self.fc3(out)
    return out

class MLP_2_Sigmoid(torch.nn.Module):
  def __init__(self):
    super(MLP_2_Sigmoid, self).__init__()
    self.fc1 = torch.nn.Linear(3 * 32 * 32, 32)
    self.sigmoid = torch.nn.Sigmoid()
    self.fc2 = torch.nn.Linear(32, 64, bias=False)
    self.fc3 = torch.nn.Linear(64, 10)

  def forward(self, x):
```

```python
    out = self.fc1(x)
    out = self.sigmoid(out)
    out = self.fc2(out)
    out = self.sigmoid(out)
    out = self.fc3(out)
    return out


class CNN_3_ReLU(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.conv1 = torch.nn.Conv2d(3, 16, 3, padding=1)  # Conv-3x3x16
    self.conv2 = torch.nn.Conv2d(16, 8, 5, padding=2)  # Conv-5x5x8
    self.conv3 = torch.nn.Conv2d(8, 16, 7, padding=3)  # Conv-7x7x16
    self.pool = torch.nn.MaxPool2d(2, 2)
    self.fc1 = torch.nn.Linear(16 * 8 * 8, 64)
    self.fc2 = torch.nn.Linear(64, 10)

  def forward(self, x):
    # Input: N, 3, 32, 32
    x = F.relu(self.conv1(x))   # -> N, 16, 32, 32
    x = F.relu(self.conv2(x))   # -> N, 8, 32, 32
    x = self.pool(x)            # -> N, 8, 16, 16
    x = F.relu(self.conv3(x))   # -> N, 16, 16, 16
    x = self.pool(x)            # -> N, 16, 8, 8
    x = torch.flatten(x, 1)     # -> N, 16 * 8 * 8 = 1024
    x = F.relu(self.fc1(x))     # -> N, 64
    x = self.fc2(x)             # -> N, 10
    return x

class CNN_3_Sigmoid(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.conv1 = torch.nn.Conv2d(3, 16, 3, padding=1)  # Conv-3x3x16
    self.conv2 = torch.nn.Conv2d(16, 8, 5, padding=2)  # Conv-5x5x8
    self.conv3 = torch.nn.Conv2d(8, 16, 7, padding=3)  # Conv-7x7x16
    self.pool = torch.nn.MaxPool2d(2, 2)
    self.fc1 = torch.nn.Linear(16 * 8 * 8, 64)
    self.fc2 = torch.nn.Linear(64, 10)

  def forward(self, x):
    # Input: N, 3, 32, 32
    x = F.sigmoid(self.conv1(x))   # -> N, 16, 32, 32
    x = F.sigmoid(self.conv2(x))   # -> N, 8, 32, 32
    x = self.pool(x)            # -> N, 8, 16, 16
    x = F.sigmoid(self.conv3(x))   # -> N, 16, 16, 16
    x = self.pool(x)            # -> N, 16, 8, 8
    x = torch.flatten(x, 1)     # -> N, 16 * 8 * 8 = 1024
    x = F.sigmoid(self.fc1(x))     # -> N, 64
    x = self.fc2(x)             # -> N, 10
    return x

class CNN_4_ReLU(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.conv1 = torch.nn.Conv2d(3, 16, 3, padding=1)   # Conv-3x3x16
    self.conv2 = torch.nn.Conv2d(16, 8, 3, padding=1)  # Conv-3x3x8
    self.conv3 = torch.nn.Conv2d(8, 16, 5, padding=2)  # Conv-5x5x16
    self.conv4 = torch.nn.Conv2d(16, 16, 5, padding=2)  # Conv-5x5x16
    self.pool = torch.nn.MaxPool2d(2, 2)
    self.fc1 = torch.nn.Linear(16 * 8 * 8, 64)
    self.fc2 = torch.nn.Linear(64, 10)

  def forward(self, x):
    # Input: N, 3, 32, 32
    x = F.relu(self.conv1(x))   # -> N, 16, 32, 32
    x = F.relu(self.conv2(x))   # -> N, 8, 32, 32
    x = F.relu(self.conv3(x))   # -> N, 16, 32, 32
    x = self.pool(x)            # -> N, 16, 16, 16
    x = F.relu(self.conv4(x))   # -> N, 16, 16, 16
    x = self.pool(x)           # -> N, 16, 8, 8

    x = torch.flatten(x, 1)     # -> N, 16 * 8 * 8 = 1024
    x = F.relu(self.fc1(x))     # -> N, 64
```

```
    x = self.fc2(x)             # -> N, 10
    return x

class CNN_4_Sigmoid(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.conv1 = torch.nn.Conv2d(3, 16, 3, padding=1)   # Conv-3x3x16
    self.conv2 = torch.nn.Conv2d(16, 8, 3, padding=1)  # Conv-3x3x8
    self.conv3 = torch.nn.Conv2d(8, 16, 5, padding=2)  # Conv-5x5x16
    self.conv4 = torch.nn.Conv2d(16, 16, 5, padding=2)  # Conv-5x5x16
    self.pool = torch.nn.MaxPool2d(2, 2)
    self.fc1 = torch.nn.Linear(16 * 8 * 8, 64)
    self.fc2 = torch.nn.Linear(64, 10)

  def forward(self, x):
    # Input: N, 3, 32, 32
    x = F.sigmoid(self.conv1(x))   # -> N, 16, 32, 32
    x = F.sigmoid(self.conv2(x))   # -> N, 8, 32, 32
    x = F.sigmoid(self.conv3(x))   # -> N, 16, 32, 32
    x = self.pool(x)            # -> N, 16, 16, 16
    x = F.sigmoid(self.conv4(x))   # -> N, 16, 16, 16
    x = self.pool(x)           # -> N, 16, 8, 8

    x = torch.flatten(x, 1)     # -> N, 16 * 8 * 8 = 1024
    x = F.sigmoid(self.fc1(x))     # -> N, 64
    x = self.fc2(x)             # -> N, 10
    return x

class CNN_5_ReLU(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.conv1 = torch.nn.Conv2d(3, 8, 3, padding=1)   # Conv-3x3x8
    self.conv2 = torch.nn.Conv2d(8, 16, 3, padding=1)  # Conv-3x3x16
    self.conv3 = torch.nn.Conv2d(16, 8, 3, padding=1)  # Conv-3x3x8
    self.conv4 = torch.nn.Conv2d(8, 16, 3, padding=1)  # Conv-3x3x16
    self.conv5 = torch.nn.Conv2d(16, 16, 3, padding=1)  # Conv-3x3x16
    self.conv6 = torch.nn.Conv2d(16, 8, 3, padding=1)  # Conv-3x3x8
    self.pool = torch.nn.MaxPool2d(2, 2)
    self.fc1 = torch.nn.Linear(8 * 8 * 8, 64)
    self.fc2 = torch.nn.Linear(64, 10)

  def forward(self, x):
    # Input: N, 3, 32, 32
    x = F.relu(self.conv1(x))   # -> N, 8, 32, 32
    x = F.relu(self.conv2(x))   # -> N, 16, 32, 32
    x = F.relu(self.conv3(x))   # -> N, 8, 32, 32
    x = F.relu(self.conv4(x))   # -> N, 16, 32, 32
    x = self.pool(x)            # -> N, 16, 16, 16
    x = F.relu(self.conv5(x))   # -> N, 16, 16, 16
    x = F.relu(self.conv6(x))   # -> N, 8, 16, 16
    x = self.pool(x)            # -> N, 8, 8, 8

    x = torch.flatten(x, 1)     # -> N, 8 * 8 * 8 = 512
    x = F.relu(self.fc1(x))     # -> N, 64
    x = self.fc2(x)             # -> N, 10
    return x

class CNN_5_Sigmoid(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.conv1 = torch.nn.Conv2d(3, 8, 3, padding=1)   # Conv-3x3x8
    self.conv2 = torch.nn.Conv2d(8, 16, 3, padding=1)  # Conv-3x3x16
    self.conv3 = torch.nn.Conv2d(16, 8, 3, padding=1)  # Conv-3x3x8
    self.conv4 = torch.nn.Conv2d(8, 16, 3, padding=1)  # Conv-3x3x16
    self.conv5 = torch.nn.Conv2d(16, 16, 3, padding=1)  # Conv-3x3x16
    self.conv6 = torch.nn.Conv2d(16, 8, 3, padding=1)  # Conv-3x3x8
    self.pool = torch.nn.MaxPool2d(2, 2)
    self.fc1 = torch.nn.Linear(8 * 8 * 8, 64)
    self.fc2 = torch.nn.Linear(64, 10)

  def forward(self, x):
    # Input: N, 3, 32, 32
    x = F.sigmoid(self.conv1(x))   # -> N, 8, 32, 32
    x = F.sigmoid(self.conv2(x))   # -> N, 16, 32, 32
```

```
    x = F.sigmoid(self.conv3(x))    # -> N, 8, 32, 32
    x = F.sigmoid(self.conv4(x))    # -> N, 16, 32, 32
    x = self.pool(x)              # -> N, 16, 16, 16
    x = F.sigmoid(self.conv5(x))    # -> N, 16, 16, 16
    x = F.sigmoid(self.conv6(x))    # -> N, 8, 16, 16
    x = self.pool(x)              # -> N, 8, 8, 8

    x = torch.flatten(x, 1)       # -> N, 8 * 8 * 8 = 512
    x = F.sigmoid(self.fc1(x))      # -> N, 64
    x = self.fc2(x)               # -> N, 10
    return x
```

4) results.py:

```
import sys
import os
import json
import numpy as np

sys.path.append(os.path.abspath('_given'))

from utils import part4Plots


with open('part4/results/result_MLP1.json', 'r') as file:
    results_MLP1 = json.load(file)
with open('part4/results/result_MLP2.json', 'r') as file:
    results_MLP2 = json.load(file)
with open('part4/results/result_CNN3.json', 'r') as file:
    results_CNN3 = json.load(file)
with open('part4/results/result_CNN4.json', 'r') as file:
    results_CNN4 = json.load(file)
with open('part4/results/result_CNN5.json', 'r') as file:
    results_CNN5 = json.load(file)


part4Plots([results_MLP1])
part4Plots([results_MLP2])
part4Plots([results_CNN3])
part4Plots([results_CNN4])
part4Plots([results_CNN5])
```

E. Part 5

1) part5.py:

```
import torch
import torchvision
import torchvision.transforms as transforms
from models import CNN_5, CNN_4
from functions import CNN_train_and_evaluate,convert_to_serializable
import json

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print("You are using device:" , device)

# Hyper-parameters
num_epochs = 20
batch_size = 50



################################################################################
# Transformations
################################################################################

transform = transforms.Compose([
  transforms.ToTensor(),
  transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),
])

# Load the dataset (assuming it's a custom dataset with 100,000 images)
full_dataset = torchvision.datasets.CIFAR10(root='./data',
                                            train=True,
                                            download=True,
                                            transform=transform)

# Compute split sizes
train_size = int(0.9 * len(full_dataset))   # 90% training
test_size = len(full_dataset) - train_size  # 10% testing

# Split dataset
train_dataset, test_dataset = torch.utils.data.random_split(full_dataset, [train_size, test_size])

# Create DataLoaders
train_loader = torch.utils.data.DataLoader(train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)




################################################################################
# Instantiate the model
################################################################################

model_CNN5 = CNN_5()
model_CNN5_scheduled = CNN_5()




################################################################################
# Testing D fferent Learning Rates
################################################################################
print("Tranining of CNN5 is starting with lr = 0.1")
result_CNN5_1 = CNN_train_and_evaluate('cnn_5',
                                     model_CNN5,
                                     train_loader,
                                     test_loader,
                                     0.1,         # Learning Rate
                                     20,          # Number of epochs
                                     100,         # Target Validation
```

```
                                     True,         # Reinitialize Weights
                                     device)
print("CNN_5␣Training␣and␣evaluation␣finished␣with␣lr␣=␣0.1")

print("Tranining␣of␣CNN5␣is␣starting␣with␣lr␣=␣0.01")
result_CNN5_01 = CNN_train_and_evaluate('cnn_5',
                                     model_CNN5,
                                     train_loader,
                                     test_loader,
                                     0.01,         # Learning Rate
                                     20,           # Number of epochs
                                     100,          # Target Validation
                                     True,         # Reinitialize Weights
                                     device)
print("CNN_5␣Training␣and␣evaluation␣finishedwith␣lr␣=␣0.01")

print("Tranining␣of␣CNN5␣is␣starting␣with␣lr␣=␣0.001")
result_CNN5_001 = CNN_train_and_evaluate('cnn_5',
                                     model_CNN5,
                                     train_loader,
                                     test_loader,
                                     0.001,        # Learning Rate
                                     20,           # Number of epochs
                                     100,          # Target Validation
                                     True,         # Reinitialize Weights
                                     device)
print("CNN_5␣Training␣and␣evaluation␣finished␣with␣lr␣=␣0.001")

results_model_CNN5 = {
    'name': 'CNN5',
    'loss_curve_1': result_CNN5_1['loss_curve'],
    'loss_curve_01': result_CNN5_01['loss_curve'],
    'loss_curve_001': result_CNN5_001['loss_curve'],
    'val_acc_curve_1': result_CNN5_1['validation_accuracy'],
    'val_acc_curve_01': result_CNN5_01['validation_accuracy'],
    'val_acc_curve_001': result_CNN5_001['validation_accuracy']
}

result_serializable_CNN5 = convert_to_serializable(results_model_CNN5)
# Save to JSON file
with open('./part5/results/result_CNN5_diff_lr.json', 'w') as f:
    json.dump(result_serializable_CNN5, f, indent=4)






##################################################################################
# Scheduling Learning First Try
##################################################################################

print("Tranining␣of␣Scheduling␣CNN5␣is␣starting␣with␣lr␣=␣0.1")
result_CNN5_sch_part1 = CNN_train_and_evaluate('cnn_5',
                                     model_CNN5_scheduled,
                                     train_loader,
                                     test_loader,
                                     0.1,          # Learning Rate
                                     30,           # Number of Epochs
                                     65,           # Target Validancy
                                     True,         # Reinitialize Weights
                                     device)
print("CNN_5␣Training␣and␣evaluation␣finished␣with␣lr␣=␣0.1,␣without␣achieved␣desired␣validancy")

print("Tranining␣of␣CNN5␣is␣starting␣with␣lr␣=␣0.01")
result_CNN5_sch_part2 = CNN_train_and_evaluate('cnn_5',
                                     model_CNN5_scheduled,
                                     train_loader,
                                     test_loader,
                                     0.01,         # Learning Rate
                                     30,           # Number of Epochs
                                     70,          # Target Validancy
                                     False,        # Reinitialize Weights
                                     device)
```

```python
print("CNN_5 Training and evaluation finished with lr = 0.01, without achieved desired validancy")

print("Tranining of CNN5 is starting with lr = 0.001")
result_CNN5_sch_part2 = CNN_train_and_evaluate('cnn_5',
                                     model_CNN5_scheduled,
                                     train_loader,
                                     test_loader,
                                     0.001,        # Learning Rate
                                     30,          # Number of Epochs
                                     100,         # Target Validancy
                                     False,       # Reinitialize Weightss
                                     device)
print("CNN_5 Training and evaluation finishedwith lr = 0.001")


results_model_CNN5_sch = []
for result in [result_CNN5_sch_part1, result_CNN5_sch_part2]:
  results_model_CNN5_sch += result['validation_accuracy']

result_serializable_CNN5_sch = convert_to_serializable(results_model_CNN5_sch)
# Save to JSON file
with open('./part5/results/result_CNN5_sch_1.json', 'w') as f:
    json.dump(result_serializable_CNN5_sch, f, indent=4)
```

2) functions.py:

```python
import torch
import numpy as np


def CNN_train_and_evaluate(model_name, in_model, in_train_loader, in_test_loader, learning_rate, num_epochs
    =15, target_val_accuracy=None, reinitialize=True ,device='cuda'):
  model = in_model.to(device)


  if reinitialize:
    # He initialization for ReLU activations for applicable layers
    def init_weights(m):
      if isinstance(m, (torch.nn.Linear, torch.nn.Conv2d)):
        torch.nn.init.kaiming_uniform_(m.weight)
        if m.bias is not None:
          m.bias.data.fill_(0.0)
    model.apply(init_weights)

  criterion = torch.nn.CrossEntropyLoss()
  optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0.0)

  loss_curve = []
  validation_accuracy = []

  # Preload all test data once
  test_images, test_labels = [], []
  for images, labels in in_test_loader:
    test_images.append(images.to(device))
    test_labels.append(labels.to(device))
  test_images = torch.cat(test_images)
  test_labels = torch.cat(test_labels)
  test_batch_size = in_test_loader.batch_size  # Preserve original batch size

  early_stop = False  # Flag to check early stopping condition

  for epoch in range(num_epochs):
    running_loss = 0.0
    model.train()

    for i, (images, labels) in enumerate(in_train_loader):
      images, labels = images.to(device), labels.to(device)
      images = images + 0.007 * torch.randn_like(images)  # Adversarial Training: adding noise to images

      # Forward + backward
      outputs = model(images)
      loss = criterion(outputs, labels)
      loss.backward()
      optimizer.step()
      optimizer.zero_grad()
```

```
      running_loss += loss.item()

      # Validation every 10 batches
      if (i + 1) % 10 == 0:
        avg_loss = running_loss / 10
        loss_curve.append(avg_loss)
        running_loss = 0.0

        model.eval()
        val_correct = 0
        with torch.no_grad():
          for batch_start in range(0, len(test_images), test_batch_size):
            batch_images = test_images[batch_start:batch_start+test_batch_size]
            batch_labels = test_labels[batch_start:batch_start+test_batch_size]
            outputs = model(batch_images)
            _, predicted = torch.max(outputs, 1)
            val_correct += (predicted == batch_labels).sum().item()
        val_acc = 100.0 * val_correct / len(test_labels)
        validation_accuracy.append(val_acc)

        # Check if early stopping criterion is met
        if target_val_accuracy is not None and val_acc >= target_val_accuracy:
          print(f"Early stopping triggered: Validation accuracy {val_acc:.2f}% reached target of {target_val_accuracy}%")
          early_stop = True
          break  # Exit inner loop

        model.train()

    print(f'[{epoch+1}/{num_epochs}] Epoch completed. Last Validation Accuracy: {val_acc:.2f}%')

    if early_stop:
      break  # Exit outer loop if early stopping was triggered

  # Return only the loss curve and validation accuracy data
  return {
    'loss_curve': loss_curve,
    'validation_accuracy': validation_accuracy
  }


def convert_to_serializable(obj):
    if isinstance(obj, torch.Tensor):
        return obj.tolist()  # Convert tensors to lists
    if isinstance(obj, np.ndarray):
        return obj.tolist()  # Convert NumPy arrays to lists
    if isinstance(obj, dict):
        return {k: convert_to_serializable(v) for k, v in obj.items()}  # Recursively convert dicts
    if isinstance(obj, list):
        return [convert_to_serializable(i) for i in obj]  # Recursively convert lists
    return obj  # Return the object as is if it's already serializable
```

3) *models.py:*

```
import torch
import torch.nn.functional as F


class CNN_4(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.conv1 = torch.nn.Conv2d(3, 16, 3, padding=1)   # Conv-3x3x16
    self.conv2 = torch.nn.Conv2d(16, 8, 3, padding=1)  # Conv-3x3x8
    self.conv3 = torch.nn.Conv2d(8, 16, 5, padding=2)  # Conv-5x5x16
    self.conv4 = torch.nn.Conv2d(16, 16, 5, padding=2)  # Conv-5x5x16
    self.pool = torch.nn.MaxPool2d(2, 2)
    self.fc1 = torch.nn.Linear(16 * 8 * 8, 64)
    self.fc2 = torch.nn.Linear(64, 10)

  def forward(self, x):
    # Input: N, 3, 32, 32
    x = F.relu(self.conv1(x))   # -> N, 16, 32, 32
    x = F.relu(self.conv2(x))   # -> N, 8, 32, 32
    x = F.relu(self.conv3(x))   # -> N, 16, 32, 32
```

```
    x = self.pool(x)              # -> N, 16, 16, 16
    x = F.relu(self.conv4(x))    # -> N, 16, 16, 16
    x = self.pool(x)             # -> N, 16, 8, 8

    x = torch.flatten(x, 1)      # -> N, 16 * 8 * 8 = 1024
    x = F.relu(self.fc1(x))      # -> N, 64
    x = self.fc2(x)              # -> N, 10
    return x


class CNN_5(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.conv1 = torch.nn.Conv2d(3, 8, 3, padding=1)   # Conv-3x3x8
    self.conv2 = torch.nn.Conv2d(8, 16, 3, padding=1)  # Conv-3x3x16
    self.conv3 = torch.nn.Conv2d(16, 8, 3, padding=1)  # Conv-3x3x8
    self.conv4 = torch.nn.Conv2d(8, 16, 3, padding=1)  # Conv-3x3x16
    self.conv5 = torch.nn.Conv2d(16, 16, 3, padding=1)  # Conv-3x3x16
    self.conv6 = torch.nn.Conv2d(16, 8, 3, padding=1)  # Conv-3x3x8
    self.pool = torch.nn.MaxPool2d(2, 2)
    self.fc1 = torch.nn.Linear(8 * 8 * 8, 64)
    self.fc2 = torch.nn.Linear(64, 10)

  def forward(self, x):
    # Input: N, 3, 32, 32
    x = F.relu(self.conv1(x))    # -> N, 8, 32, 32
    x = F.relu(self.conv2(x))    # -> N, 16, 32, 32
    x = F.relu(self.conv3(x))    # -> N, 8, 32, 32
    x = F.relu(self.conv4(x))    # -> N, 16, 32, 32
    x = self.pool(x)             # -> N, 16, 16, 16
    x = F.relu(self.conv5(x))    # -> N, 16, 16, 16
    x = F.relu(self.conv6(x))    # -> N, 8, 16, 16
    x = self.pool(x)             # -> N, 8, 8, 8

    x = torch.flatten(x, 1)      # -> N, 8 * 8 * 8 = 512
    x = F.relu(self.fc1(x))      # -> N, 64
    x = self.fc2(x)              # -> N, 10
    return x
```

4) results.py:

```
import sys
import os
import json
import numpy as np

sys.path.append(os.path.abspath('_given'))

from utils import part5Plots


with open('part5/results/result_CNN5_diff_lr.json', 'r') as file:
    results_CNN5 = json.load(file)




part5Plots([results_CNN5])
```