Deniz Karakay - 2443307 Date: 21/04/2023

EE449- Homework 1

Training Artificial Neural Network

1. Basic Concepts

1.1. Which Function?

An ANN classifier that is trained using cross-entropy loss estimates the likelihood of the input belonging to each class. The cross-entropy loss means the difference between the probability distribution that is estimated by ANN and the actual distribution of the labels. This loss is used to support the model's decision to give the true label a higher probability than the false label. The basic goal of a classification assignment is to minimize the cross-entropy loss, which enables the ANN to learn to approximate the real posterior probability distribution of the input given the output classes. The formula for cross entropy loss can be examined in *Figure 1.*

$$Loss\ (y, \hat{y}) = -\sum_{k}^{N} y^{(k)} \times \log \hat{y}^{(k)}$$

Figure 1. The Cross Entropy Loss formula

N shows the number of classes, namely the output classes. The $y^{(k)}$ becomes 0 or 1 shows whether the class label k is correct for classification. The $\hat{y}^{(k)}$ is the predicted probability distribution for k.
Let's consider a simple example of binary classification, where the goal is to classify an input image as either containing a cat or not containing a cat.

Suppose the ANN predicts a probability of 0.85 for an input image which actually contains a cat. The true probability distribution for this image is [0, 1] since the image contains a cat. The cross-entropy loss for this prediction is:

$$L\ (y, \hat{y}) = -\sum_{k}^{N} y^{(k)} \times \log \hat{y}^{(k)} = -(0 \log(0.85) + 1 \log(0.15)) = 0.82$$

Similarly, suppose the ANN predicts a probability of 0.3 for an input image which does not contain a cat. The true probability distribution for this image is [1, 0] since the image does not contain a cat. The cross-entropy loss for this prediction is:

$$L\,(y,\hat{y}) = -\sum_{k}^{N} y^{(k)} \times \log \hat{y}^{(k)} = -(1\log(0.3) + 0\log(0.7)) = 0.52$$

The ANN modifies its parameters during training to reduce the average cross-entropy loss across all training cases. As a result, the input photos are correctly classified as binary images when the ANN learns to anticipate the true probability distribution of the labels given the input.

1.2. Gradient Computation

We know that:

$$\omega_{k+1}(t) = \omega_k(t) - \gamma\,\nabla e\big(\omega_k(t)\big)$$

If we utilize the variable defined in the question, we have:

$$\omega_{k+1} = \omega_k - \gamma\,\nabla L_{w=w_k}$$

If we leave the gradient of the loss at one side, we have:

$$\nabla L_{w=w_k} = \frac{1}{\gamma} \times (\omega_k - \omega_{k+1})$$

1.3. Some Training Parameters and Basic Parameter Calculations

1.3.1. Batch and Epoch

Batch and epoch are crucial notions that describe how the training data is processed in the context of training a Multilayer Perceptron (MLP).

A batch is a portion of training data that is processed collectively by the training algorithm during a single iteration. The training algorithm calculates the gradients of the loss function with respect to the MLP weights and updates the weights as necessary in a one batch.

On the other hand, an epoch shows a complete traversal of the entire training dataset. Each example in the training dataset is processed by the MLP once during a single epoch, and the weights are updated using the gradients calculated from the complete dataset.

1.3.2.

By dividing the total amount of samples in the dataset by the batch size, e can get the number of batches per epoch.

$$\#\ of\ batches\ per\ epoch = N/B$$

1.3.3.

$$\#\ of\ batches\ per\ epoch\ \times\ \#\ of\ epochs$$

$$= \frac{N}{B} \times E$$

1.4. Computing Number of Parameters of ANN Classifiers

1.4.1.

We need to count the number of weights and biases to compute the number of parameters.

The first hidden layer has H_1 units that take the input D_{in}. Therefore, the number of biases is H_1 and total the number of weights for 1st layer is $D_{in}\ \times H_1$

For the k-th hidden layer, we have H_k units that take the outputs from previous hidden layer. Therefore, we need to multiply the previous layer's bias with this bias in order to find the number of weights for this layer: $H_{(k-1)}\ \times H_k$

Similarly, the output layer has D_{out} units that take the outputs from the last hidden layer. Therefore, the number of weights for this layer is $H_k\ \times D_{out}$,

$$\#\ of\ params = D_{in} \times H_1 + \sum_{k=1}^{k=K-1} (H_k \times\ H_{k+1}) + \sum_{k=1}^{k=K} (H_k) + D_{out} + H_k\ \times D_{out}$$

1.4.2.

We need to count the number of weights and biases to compute the number of parameters.

$$Input\ Layer\ -1st\ Hid\ Layer\ = (C_{in} \times W_{in}\ \times H_{in}\ \times C_1) + C_1$$

$$1st\ Hid\ Layer - 2nd\ Hid\ Layer\ = (C_1 \times W_1\ \times H_1\ \times C_2) + C_2$$

$$K-1th\ Hid\ Layer - Kth\ Hid\ Layer\ = (C_{K-1} \times W_{K-1}\ \times H_{K-1} \times C_K) + C_K$$

$$\#\ of\ params = (C_{in} \times H_{in} \times C_1 \times W_{in} + \sum_{k=1}^{k=K-1} (C_K \times C_{K+1} \times H_K \times W_K) + \sum_{k=1}^{k=K} (C_k)$$

2. Implementing a Convolutional Layer with NumPy

2.1. Experimental Work

Since my school ID ends with 7, I utilized *samples_7.npy* to test my convolutional layer.

Figure 2. Result of my_conv2d function

Figure 3. Result of conv2d function of Tensorflow

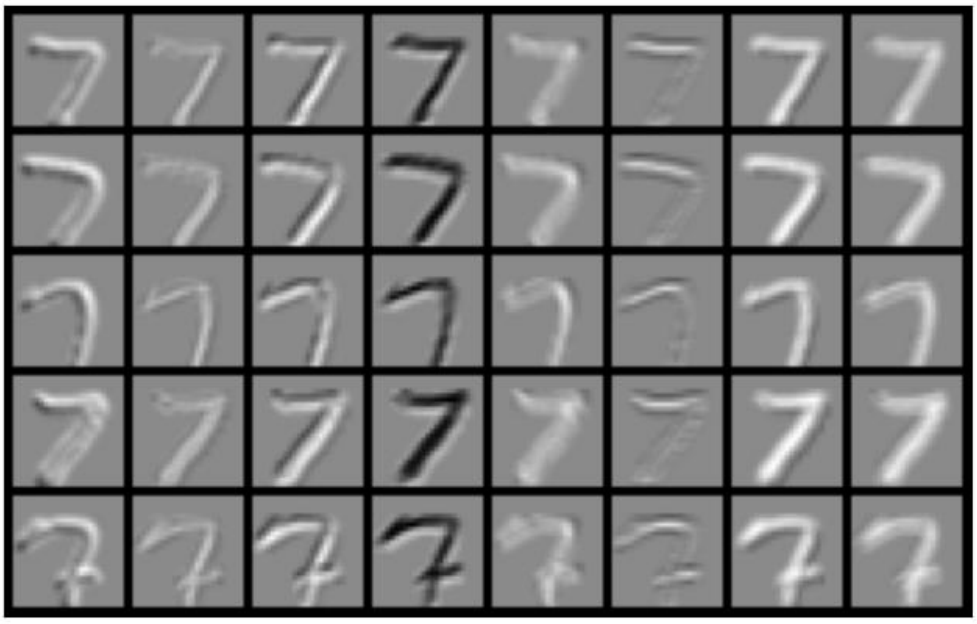

Figure 4. Normalized result of my_conv2d function

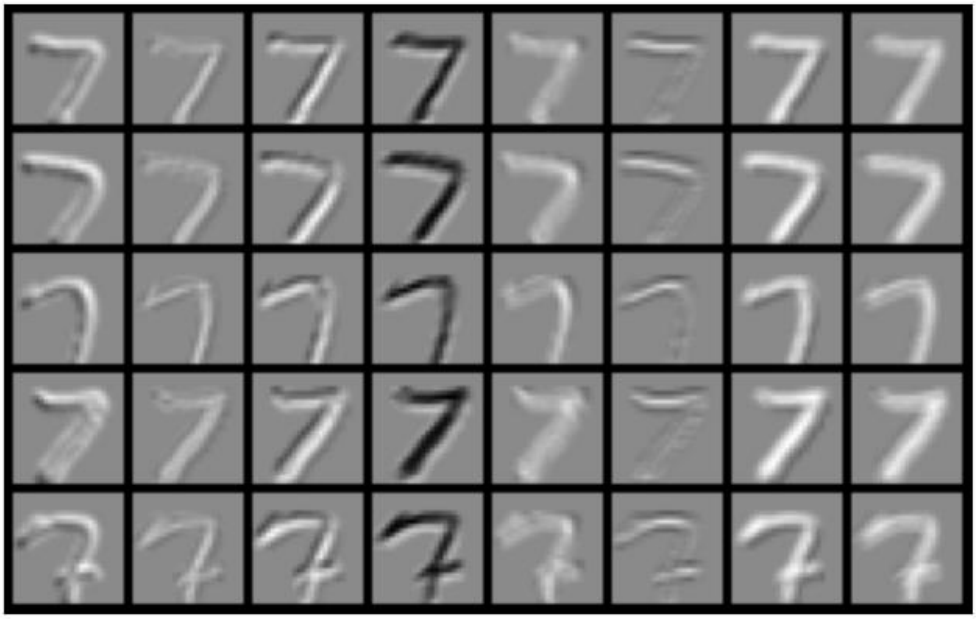

Figure 5. Normalized result of conv2d function of Tensorflow

2.2. Discussions

2.2.1.

The ability of convolutional layers to recognize patterns and characteristics in an image makes convolutional neural networks (CNNs) vital for image processing. These layers apply filters to the image, bringing out details and minimizing the quantity of data that needs to be processed. As a result, CNNs are particularly beneficial for applications like segmentation, classification, and image recognition. CNNs are an effective tool in computer vision applications because they are capable of learning and extract information from images without human interaction.

2.2.2.

The kernel of a CNN is a weight matrix that is applied to a tiny section of an input picture during processing. The kernel's size represents the input region's spatial extent, and the weights of the kernel control how the region's attributes are merged to provide an output value. Smaller kernels catch finer details whereas larger kernels record more general patterns, allowing for tighter control over the level of detail captured during feature recognition. However, due to a rise in parameters, larger kernels also raise the computational expense of training the model.

2.2.3.

By examining the output image of a CNN, we can observe different results of a handwritten number 7. Specifically, in *Figures 2 and 3*, we can see labeled white parts that represent the characteristics of the number. This indicates that the CNN's convolutional layer is successful in identifying and extracting the defining features of the number 7 in the white regions, while the black regions represent the background. This ability of CNNs to recognize and extract important features

effectively in images allows us to clearly observe the numbers, especially 7 for my case.

2.2.4.

We can observe the numbers in the same column are similar although they are from different images. This happened due to the fact that they share common features that allows the CNN network to learn and distinguish. During this training process, the CNN learns to identify and extract features from the input images, which are utilized to classify the number images into their categories.

2.2.5. Even though the number images in the same row of the output in *Figure 2-5* belong to the same image, they have different features that are extracted and emphasized by the filters. This allows the CNN to provide a more detailed and comprehensive representation of the image by highlighting different aspects of it.

2.2.6. We can deduct from Q4 and Q5 that Convolutional layers enable CNNs to accurately identify and classify images based on their features and patterns while also providing a more in-depth understanding of the image by highlighting different aspects of it, allowing us to conclude that they are a powerful tool in image processing and computer vision.

3. Experimenting ANN Architectures

3.1. Experimental Work

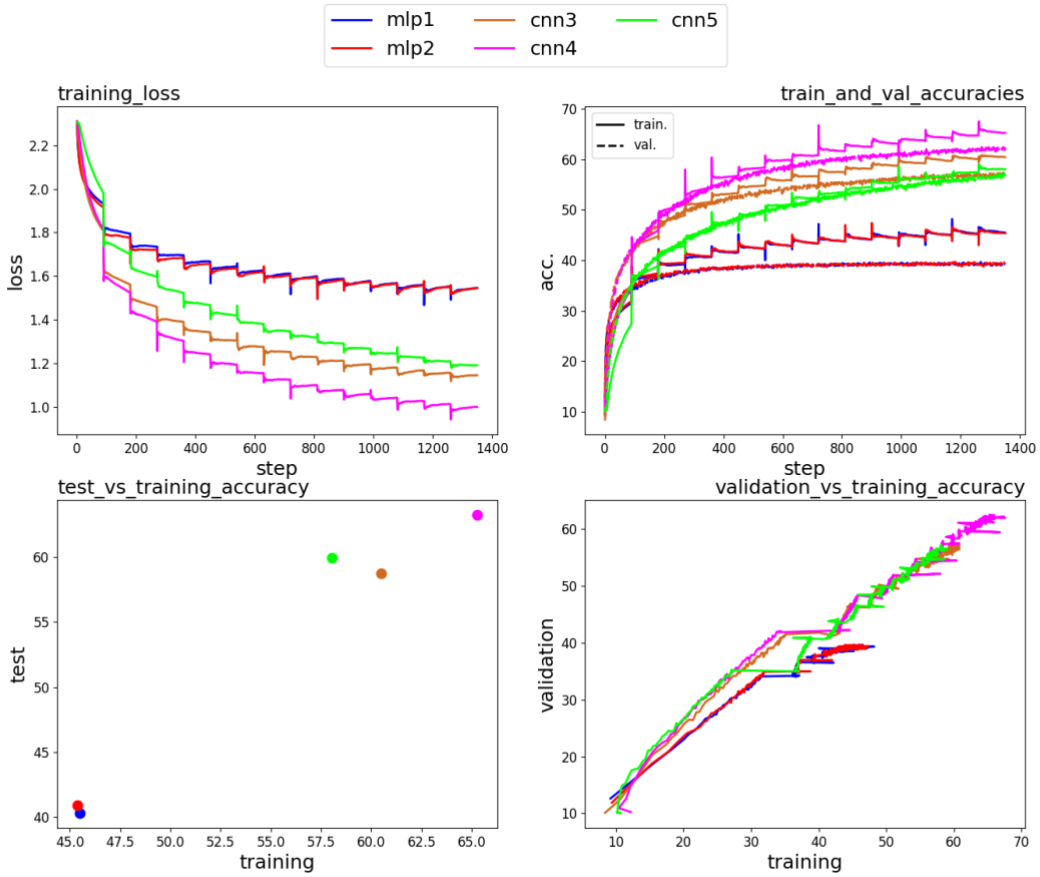

Figure 6. Result of the trainings of MLP1, MLP2, CNN3, CNN4, CNN5

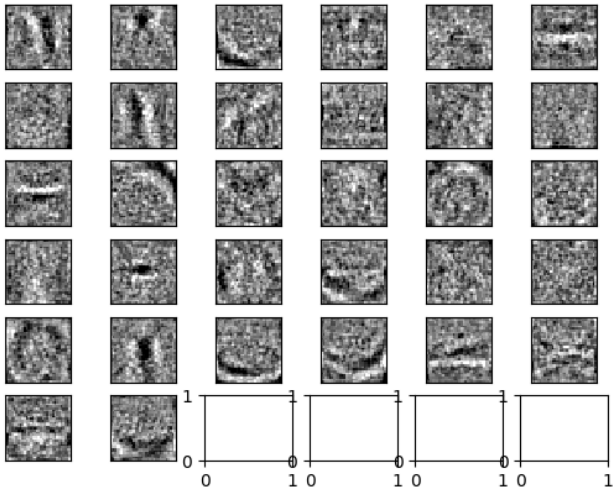

Figure 7. Results of visualizeWeights for MLP1

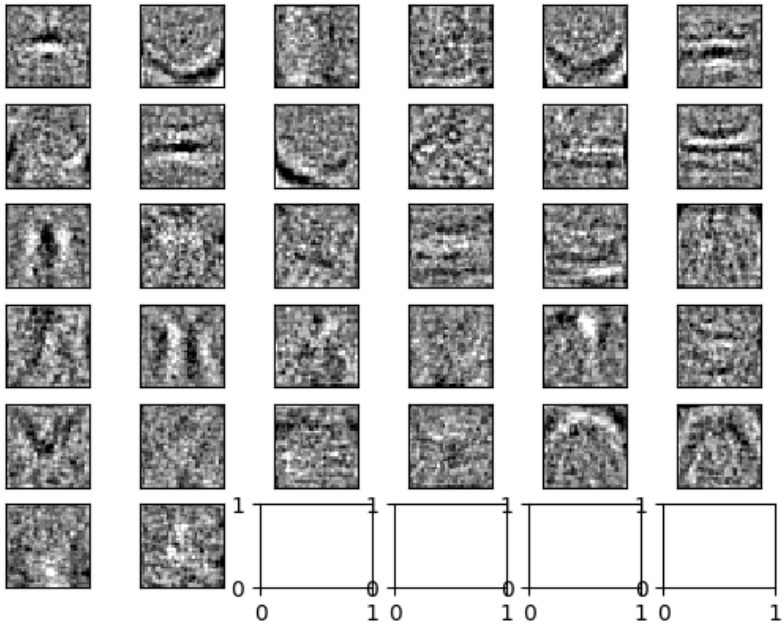

Figure 8. Results of visualizeWeights for MLP2

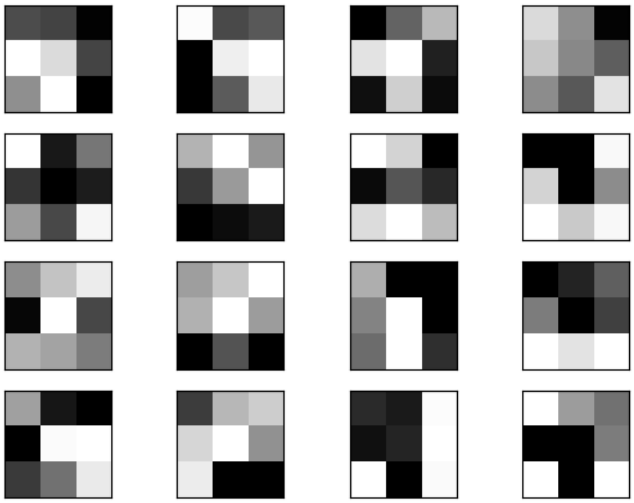

Figure 9. Results of visualizeWeights for CNN3

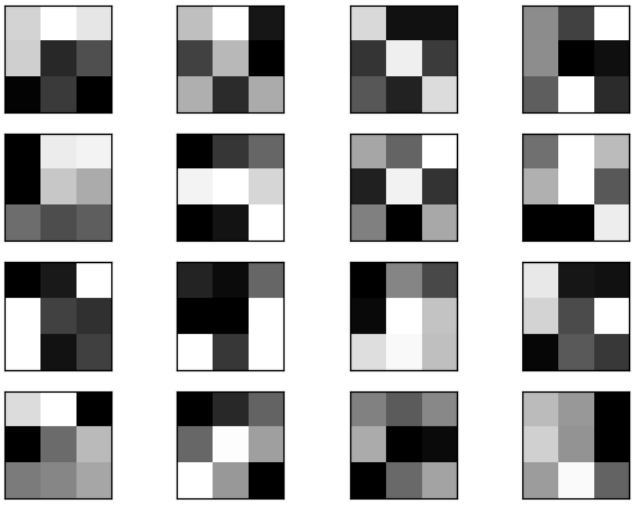

Figure 10. Results of visualizeWeights for CNN4

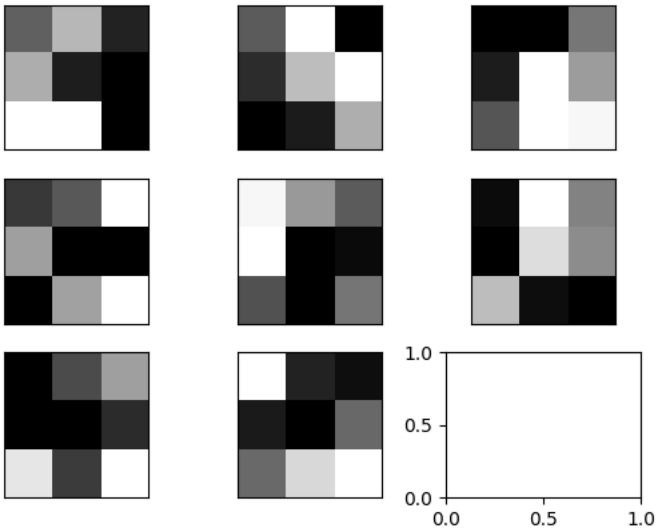

Figure 11. Results of visualizeWeights for CNN5

3.2. Discussions

3.2.1.

The generalization performance of a classifier describes the level to which the classifier can correctly predict the class labels of new, untrained data. In other words, it assesses how well the classifier can apply previously discovered patterns and guidelines to fresh data. The generalization performance can be evaluated using various metrics such as accuracy, precision, recall and can be measured using techniques like cross-validation, hold-out validation, and test set evaluation.

3.2.2.

As you can see from *Figure 6*, we have 4 different plots namely training loss, training and validation accuracies, test vs training accuracy and validation vs training accuracy. All plots are informative for our evaluation. However, especially training and validation accuracies and test vs training accuracies shows the generalization performance. A classifier is trained on a collection of labeled data throughout the training phase, and the training accuracy quantifies how well the model can anticipate the proper class labels on this training data.

A model may have memorized the training data and be overfitting, therefore great training accuracy alone does not ensure strong generalization performance. Use of a validation set to assess the model's performance on data that was not used for training is a typical method for preventing overfitting. How well the model generalizes to new data is determined by the validation accuracy.

3.2.3.

As it can be seen from *Figure 6,* CNNs are better architectures than MLPs for our task. This superiority is evident in all of the plots. In terms of test versus training accuracy, it is desirable to have higher values on both sides of the plot, indicating that the model is learning. As it can be seen, CNN4 is the best architectures among all. CNN5 and CNN3 have almost similar results. On the other hand, MLP1 and MLP2 have similar but, the poorest results in comparison to the other models.

MLPs are not designed to handle high-dimensional picture data, they perform worse than CNNs at image recognition. Due to the individual treatment of each input feature by MLPs, these models are unable to accurately capture the spatial relationships between pixels, which are essential for image identification tasks. Convolutional layers, on the other hand, are used by CNNs and are better at extracting relevant features from high-dimensional image data because they can recognize local patterns and hierarchical structures in images.

3.2.4.

In general, a model's ability to understand complicated patterns in the data can be increased with additional parameters, which can enhance classification performance on the training data. On the other hand, having too many parameters can also result in overfitting which the model memorizes the training data and harms the model's ability to generalize well to new, untested data.

If we compare the number of parameters for our models, we can see that;

MLP1 < MLP2 < CNN3 < CNN4 < CNN5

3.2.5.

We saw that adding new layers to CNN increase the number of parameters. However, the best result can be obtained in CNN4, which symbolizes of having more parameters lead overfitting for CNN5. Therefore, in general increasing the number of parameters is a good idea till a specific point where the model starts to memorize rather than learning.

3.2.6.

I think the weights for MLPs (in *Figure 7 and 8*) are interpretable since the first layer of a MLP is linked with inputs. For CNNs, it is hard to understand anything as it can be examined in *Figure 9 – 11*.

3.2.7.

In my opinion, it is difficult to identify any weight images in *Figures 7-11* that correspond to a specific class, so I don't think they are specialized for any particular class. This is likely because we are looking at the weights of the first layer, which are

simple and general features, rather than the more detailed and class-specific features that are present in the weights of the last layer. Therefore, if we examine the weights of the last layer, we may observe more class-oriented feature images.

3.2.8.

I believe the weights of MLPs are more interpretable compared to CNNs.

3.2.9.

As I explained in *3.2.3,* the CNNs are better architectures than MLPs for our task. As it can be seen in *Figure 6*, CNN4 is the best architectures among all. Based on the analysis of the plots, CNN4 can be considered the best performing model. This is because CNN4 has the highest validation accuracy and the best test versus training accuracy, which is indicated by higher values for both accuracies at the same time. CNN5 and CNN3 have almost similar results on all plots. On the other hand, MLP1 and MLP2 have alike but the less successful results not only on validation accuracies but also have larger training loss in which we aim to decrease the training loss.

3.2.10.

CNN4 is the top-performing architecture among all models. This conclusion is based on the plots analysis that indicates CNN4 has the highest validation accuracy and the best test versus training accuracy, as evidenced by higher values for both accuracies at the same time.

4. Experimenting Activation Functions

4.1. Experimental Work

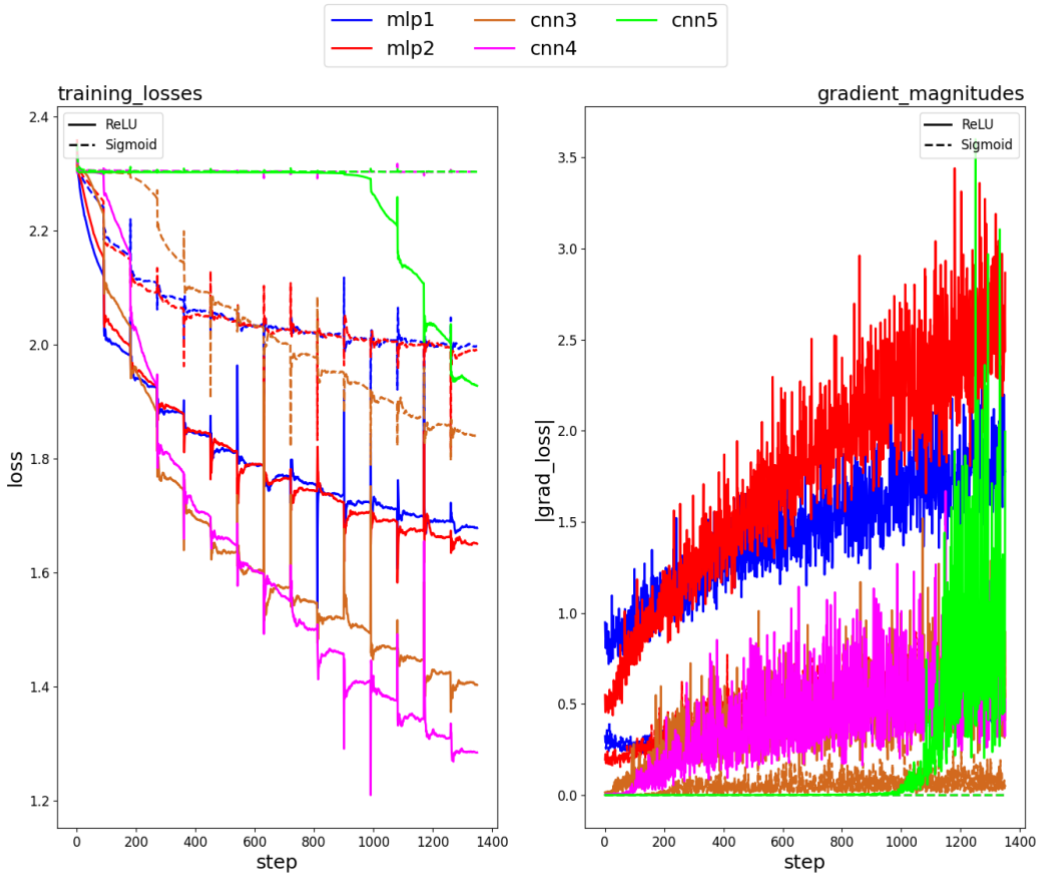

Figure 12. Result of training losses and gradient magnitudes using ReLU and Sigmoid activation functions

4.2. Discussions

4.2.1.

ReLU is typically more computationally effective than Sigmoid and leads to a higher rate of convergence during training. The "dying ReLU" issue, in which a neuron might indefinitely go dormant if it receives negative inputs, can also affect ReLU. While Sigmoid can avoid the "dying" problem because to its smoother gradient, it can be computationally expensive and cause training to converge more slowly.

The vanishing gradient problem and gradient propagation difficulties can be brought on by deepening a CNN. This is because it can be challenging to update the weights and improve the network in deep networks because the gradients might

shrink dramatically as they pass through successive layers.

4.2.2.

The gradients may get smaller and more challenging to spread throughout the network as we increase the depth of a CNN. This is because the weight matrices and activation functions at each layer can repeatedly double the gradients, resulting in a condition referred to as the vanishing gradient problem.

It can be challenging for the network to learn and improve its performance when the gradients are so small that the adjustments to the network weights are also so modest. Due to this, training may progress slowly or even come to a total stop.

4.2.3.

In neural network models like CNNs and MLPs with ReLU activation functions, using inputs in the range [0, 255] instead of [0.0, 1.0] might result in issues with excessive gradients and numerical instability during training. The performance of the model can be improved by normalizing the input data to the range [0.0, 1.0], while sigmoid activation functions can still function well with inputs in the [0, 255] range.

5. Experimenting Learning Rate

5.1. Experimental Work

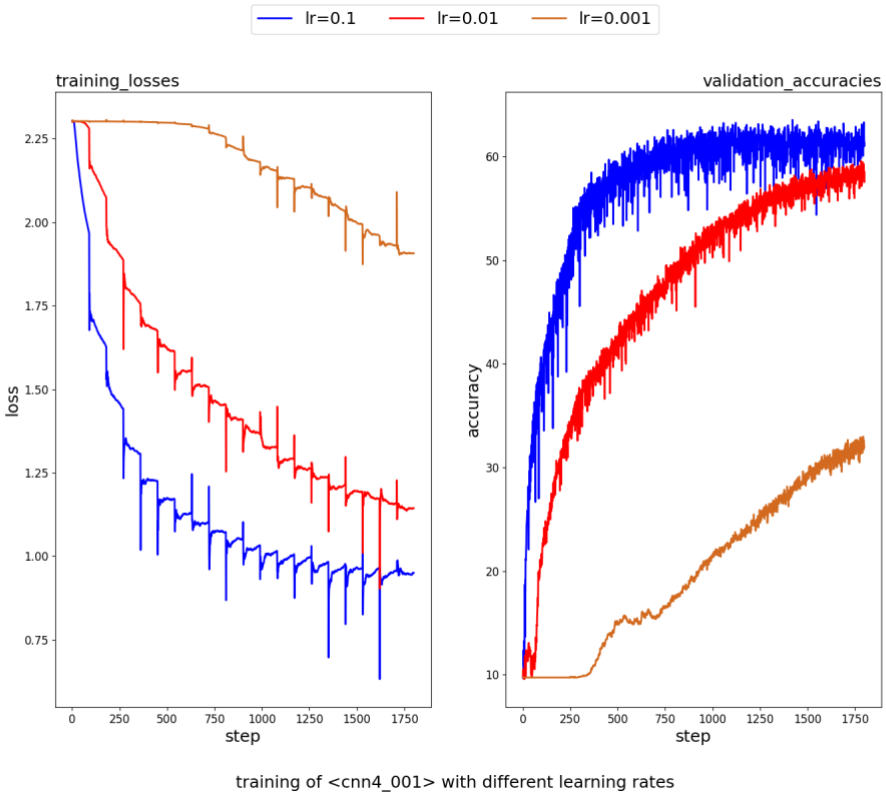

Figure 13. Results for CNN4 for learning rates: 0.1, 0.01, 0.001

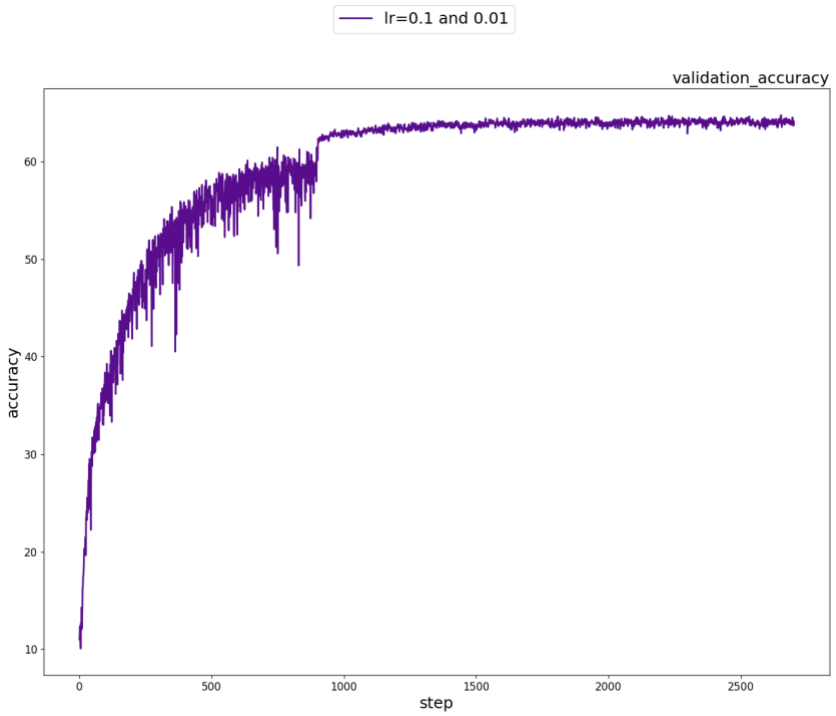

Figure 14. Validation accuracy of CNN4 for learning rates 0.1 and 0.01 combined

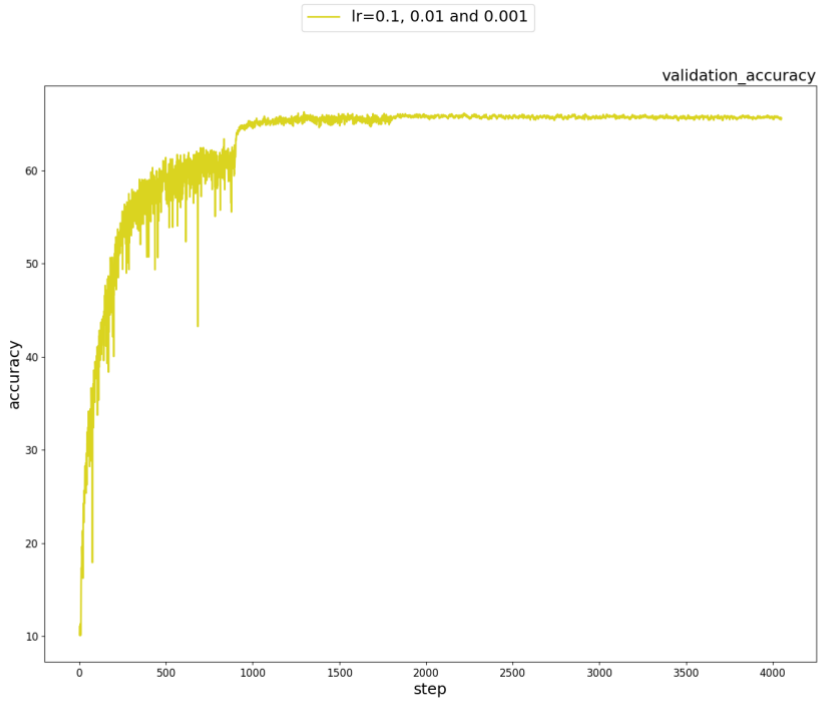

Figure 15. Validation accuracy of CNN4 for learning rates 0.1, 0.01, 0.001 combined

```
Training [1/1], 508.5757 s, Test Accuracy: 64.9200
```

Figure 16. Test accuracy after utilizing scheduled learning rate method

5.2. Discussions

5.2.1.

The convergence speed directly changes with learning rates. In general, a higher learning rate causes the optimization process to converge more quickly, but it can also cause the program to diverge and overshoot the ideal weights. In contrast, a slower learning rate can cause convergence, but it can also produce a final solution that is more accurate. The details can be seen in *Figure 13*.

5.2.2.

A suitable learning rate may assist in the model's convergence and improve accuracy. The model can converge more quickly in general with a greater learning rate, but we may also need to be more careful to prevent divergence by tuning. Although it could take more iterations to get decent results, a lower learning rate can be more stable and can aid the model's convergence to a better minimum.

Starting with a moderate learning rate and modifying it as needed based on the model's performance throughout training is the standard procedure to have a better convergence point. The convergence of the model can also be enhanced by using strategies like learning rate schedules, which gradually lower the learning rate. The learning rate can also be automatically adjusted during training by advanced optimization algorithms like Adam, which frequently leads to faster convergence and higher performance.

5.2.3.

We applied a method called scheduled learning rate to improve the convergence point, which had a positive impact on our results. The outcome can be seen in *Figures 14-16*. *Figure 14* shows a plot where we used a combination of learning rates 0.1 and 0.01, and we switched from 0.1 to 0.01 at Epoch 10, as the validation accuracy started to converge at around 60%. After changing the learning rate, the test accuracy increased to approximately 64%. In *Figure 15*, we added a checkpoint at Epoch 20 to change the learning rate from 0.01 to 0.001. Finally, in *Figure 16*, we achieved a test accuracy of about 64.92%.

5.2.4.

It seems that the scheduled learning rate method has a slightly higher accuracy than Adam, with the scheduled version achieving 64.92% accuracy compared to Adam's 63.25% accuracy. For the convergence time, it is obvious that scheduled learning rate has a better performance. Although we need to train three times to determine the better version, Adam based model took more than scheduled learning rate version.

Appendix I

question_2.py

```
#   This file contains the code for Question 2

import numpy as np
import torch
from utils.utils import part2Plots


def my_conv2d(input_tensor, kernel):
    """
    :param input_tensor: (batch_size, in_channels, in_height, in_width)
    :param kernel: (out_channels, in_channels, kernel_height, kernel_width)
    :return:
    """

    batch_size, i_channels, i_height, i_width = input_tensor.shape
    o_channels_kernel, i_channels_kernel, f_height, f_width = kernel.shape

    print("Input:")
    print(f"Batch size: {batch_size}")
    print(f"Input channels: {i_channels}")
    print(f"Input height: {i_height}")
    print(f"Input width: {i_width}")

    print("-------------------")
    print("Kernel:")
    print(f"Output channels: {o_channels_kernel}")
    print(f"Input channels: {i_channels_kernel}")
    print(f"Filter height: {f_height}")
    print(f"Filter width: {f_width}")
    print("-------------------")

    # Calculate output dimensions
    o_height = i_height - f_height + 1
    o_width = i_width - f_width + 1

    output = np.zeros((batch_size, o_channels_kernel, o_height, o_width))

    print("Output shape: ", output.shape)

    # 2D convolution
    for b in range(batch_size):  # batch
        for x in range(o_height):  # height
            for y in range(o_width):  # width
                for o in range(o_channels_kernel):  # output channels
                    output[b, o, x, y] = np.sum(
                        input_tensor[b, :, x:x + f_height, y:y + f_width, ] *
kernel[o, :, :, :]
                    )

    return output
```

```
# Pick based on my Student ID
input_sample = np.load('../utils/samples_7.npy')
kernel = np.load('../utils/kernel.npy')

# Perform 2D convolution with my_conv2d
my_output = my_conv2d(input_sample, kernel)

# Normalize input
my_output_normalized = (my_output - np.min(my_output)) / (np.max(my_output) -
np.min(my_output))

# Plot results
part2Plots(out=my_output, save_dir='results/', filename='q2_result')
part2Plots(out=my_output_normalized, save_dir='results/',
filename='q2_result_normalized')

# Compare with PyTorch's conv2d function
input_tensor = torch.from_numpy(input_sample).float()
kernel_tensor = torch.from_numpy(kernel).float()
output_torch = torch.conv2d(input_tensor, kernel_tensor, stride=1, padding=0)
output_torch_normalized = (output_torch - torch.min(output_torch)) /
(torch.max(output_torch) - torch.min(output_torch))

# Plot results
part2Plots(out=output_torch.detach().numpy(), save_dir='results/',
filename='q2_result_torch')
part2Plots(out=output_torch_normalized.detach().numpy(), save_dir='results/',
filename='q2_result_normalized_torch')
```

Appendix II

question_3.py

```python
# Created by Deniz Karakay at 17.04.2023
# Filename: question_3.py

import torch
import torch.nn as nn
import torchvision
from torchvision import transforms
import time
from torch.utils.data import SubsetRandomSampler
from sklearn.model_selection import train_test_split
import numpy as np
import pickle
from Q3 import models as my_models
from utils.utils import visualizeWeights

# Hyper-parameters
EPOCH_SIZE = 15
BATCH_SIZE = 50
TRAIN_COUNT = 10

# I tested and saw that CPU is faster than GPU on M1 Pro for MLPs

# CPU
# device = torch.device("cpu")

# MPS for GPU support on M1
device = torch.device("mps")

print(f"Using device: {device}...")

init_time = time.time()

# Transformations
transform = transforms.Compose([
    torchvision.transforms.ToTensor(),
    transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5)),
    torchvision.transforms.Grayscale()
])

# Load data
train_data = torchvision.datasets.CIFAR10('data/', train=True, download=True,
transform=transform)

# Print image size and label of first image in dataset (should be 32x32 and
6)
img, label = train_data[0]
print("Sample image size: ", img.size())

# Split data into training and validation
train_data, valid_data = train_test_split(train_data, test_size=0.1,
random_state=42)

# Load test data
```

```
test_data = torchvision.datasets.CIFAR10('data/', train=False, download=True,
transform=transform)

print(f"Training data size: {len(train_data)}")
print(f"Validation data size: {len(valid_data)}")
print(f"Test data size: {len(test_data)}")

valid_generator = torch.utils.data.DataLoader(valid_data,
batch_size=BATCH_SIZE, shuffle=False)

my_models_list = ['cnn4', ]

for model_name in my_models_list:
    if model_name.startswith('mlp'):
        continue

    print(f"Training {model_name}...")

    model_init_time = time.time()
    best_performance = 0
    best_weights = None

    train_acc_history_total = []
    train_loss_history_total = []
    valid_acc_history_total = []
    test_acc_history_total = []

    for tc in range(TRAIN_COUNT):

        if model_name == 'mlp1':
            model = my_models.MLP1(1024, 32, 10).to(device)
        elif model_name == 'mlp2':
            model = my_models.MLP2(1024, 32, 64, 10).to(device)
        elif model_name == 'cnn3':
            model = my_models.CNN3().to(device)
        elif model_name == 'cnn4':
            model = my_models.CNN4().to(device)
        elif model_name == 'cnn5':
            model = my_models.CNN5().to(device)

        # Initialize model
        criterion = nn.CrossEntropyLoss()
        optimizer = torch.optim.Adam(model.parameters())

        # Initialize lists for training and validation accuracy and loss
        train_acc_history = []
        train_loss_history = []
        valid_acc_history = []
        test_acc_history = []
        train_generator = torch.utils.data.DataLoader(train_data,
batch_size=BATCH_SIZE, shuffle=True)

        for epoch in range(EPOCH_SIZE):
            # Train the model
            start_time = time.time()
            total_step = len(train_generator)
```

```
            train_loss = 0
            train_acc = 0

            for i, data in enumerate(train_generator):
                model.train()
                inputs, labels = data
                train_inputs, train_labels = inputs.to(device),
labels.to(device)

                # Zero the parameter gradients
                optimizer.zero_grad()

                # Forward pass
                train_outputs = model(train_inputs)

                # Define loss function
                loss = criterion(train_outputs, train_labels)

                loss.backward()
                optimizer.step()

                model.eval()

                # Compute the loss
                train_loss += loss.item()

                # Compute the accuracy
                pred = train_outputs.argmax(dim=1, keepdim=True)

                # Track the statistics for training accuracy
                train_total = train_labels.size(0)
                train_correct =
pred.eq(train_labels.view_as(pred)).sum().item()
                train_acc += (train_correct / train_total) * 100

                if i % 10 == 0:

                    # Take average of train loss and accuracy for 10 steps
                    train_acc_history.append(train_acc / (i + 1))
                    train_loss_history.append(train_loss / (i + 1))

                    valid_correct = 0
                    valid_total = 0

                    with torch.no_grad():
                        for data in valid_generator:
                            inputs, labels = data
                            valid_inputs, valid_labels = inputs.to(device),
labels.to(device)

                            # Compute the outputs and predictions
                            valid_outputs = model(valid_inputs)
                            _, valid_predicted =
torch.max(valid_outputs.data, 1)

                            # Track the statistics
                            valid_total += valid_labels.size(0)
```

```
                            valid_correct += (valid_predicted ==
valid_labels).sum().item()

                        # Take average of validation accuracy
                        valid_acc = (valid_correct / valid_total) * 100
                        valid_acc_history.append(valid_acc)

            epoch_time = time.time() - start_time
            print(
                f"Epoch [{epoch + 1}/{EPOCH_SIZE}], Epoch Time:
{epoch_time:.4f} s, Train Loss: {train_loss_history[-1]:.4f}, "
                f"Train Accuracy: {train_acc_history[-1]:.3f}, Validation
Accuracy: {valid_acc_history[-1]:.3f}")

        train_acc_history_total.append(train_acc_history)
        train_loss_history_total.append(train_loss_history)
        valid_acc_history_total.append(valid_acc_history)

        # Evaluate the model on test set
        test_correct = 0
        test_total = 0
        with torch.no_grad():
            test_generator = torch.utils.data.DataLoader(test_data,
batch_size=BATCH_SIZE, shuffle=False)
            model.eval()
            for data in test_generator:
                inputs, labels = data
                test_inputs, test_labels = inputs.to(device),
labels.to(device)

                # Compute the outputs and predictions
                test_outputs = model(test_inputs)
                _, test_predicted = torch.max(test_outputs.data, 1)

                # Track the statistics
                test_total += test_labels.size(0)
                test_correct += (test_predicted == test_labels).sum().item()

                # Compute the accuracy
                test_acc = (test_correct / test_total) * 100

            # Save best weights
            if test_acc > best_performance:
                best_performance = test_acc
                best_weights = model.first.weight.data.cpu().numpy()

            # Save test accuracy
            test_acc_history.append(test_acc)

        took_time = time.time() - model_init_time
        print(f"Training [{tc + 1}/{TRAIN_COUNT}], {took_time:.4f} s, Test
Accuracy: {best_performance:.4f}")

        test_acc_history_total.append(test_acc_history)

    took_time = time.time() - model_init_time
    print(f"Best performance for {model_name}: {best_performance:.4f}, took
```

```
{took_time:.4f} s")

    # Average training loss
    t = np.array(train_loss_history_total)
    average_train_loss = np.mean(t, axis=0)

    # Average training accuracy
    t = np.array(train_acc_history_total)
    average_train_acc = np.mean(t, axis=0)

    # Average validation accuracy
    t = np.array(valid_acc_history_total)
    average_valid_acc = np.mean(t, axis=0)

    best_test_acc = np.max(test_acc_history_total)
    print(f"Best test accuracy: {best_test_acc:.4f}")

    # Save the results
    result_dict = {
        'name': model_name,
        'loss_curve': average_train_loss,
        'train_acc_curve': average_train_acc,
        'val_acc_curve': average_valid_acc,
        'test_acc': best_test_acc,
        'weights': best_weights
    }

    # Save the results to a file
    filename = 'results/question_3_' + model_name.replace(' ', '_') + '.pkl'
    with open(filename, 'wb') as f:
        pickle.dump(result_dict, f)

    took_time = time.time() - model_init_time
    print(f"All process for {model_name}: {took_time:.4f} s")

    visualizeWeights(best_weights, save_dir='results/',
                     filename='question_3_weights_' + model_name.replace(' ',
'_'))

took_time = time.time() - init_time
print(f"All process: {took_time:.4f} s")
```

Appendix III

question_4.py

```python
# Created by Deniz Karakay at 18.04.2023
# Filename: question_4.py

import torch
import torch.nn as nn
import torchvision
from torchvision import transforms
import time
from torch.utils.data import SubsetRandomSampler
from sklearn.model_selection import train_test_split
import numpy as np
import pickle
from Q4 import models as my_models
from utils.utils import visualizeWeights

# Hyper-parameters
EPOCH_SIZE = 15
BATCH_SIZE = 50
TRAIN_COUNT = 1

# I tested and saw that CPU is faster than GPU on M1 Pro for MLPs

# CPU
# device = torch.device("cpu")

# MPS for GPU support on M1
device = torch.device("mps")

print(f"Using device: {device}...")

init_time = time.time()

# Transformations
transform = transforms.Compose([
    torchvision.transforms.ToTensor(),
    transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5)),
    torchvision.transforms.Grayscale()
])

# Load data
train_data = torchvision.datasets.CIFAR10('../Q3/data/', train=True,
download=True, transform=transform)

# Print image size and label of first image in dataset (should be 32x32 and
6)
img, label = train_data[0]
print("Sample image size: ", img.size())

# Split data into training and validation
train_data, valid_data = train_test_split(train_data, test_size=0.1,
random_state=42)

# Load test data
```

```
test_data = torchvision.datasets.CIFAR10('../Q3/data', train=False,
download=True, transform=transform)

print(f"Training data size: {len(train_data)}")
print(f"Validation data size: {len(valid_data)}")
print(f"Test data size: {len(test_data)}")

valid_generator = torch.utils.data.DataLoader(valid_data,
batch_size=BATCH_SIZE, shuffle=False)

# Sigmoid and ReLU based models
my_models_list = ['mlp1', 'mlp1s', 'mlp2', 'mlp2s', 'cnn3', 'cnn3s', 'cnn4',
'cnn4s', 'cnn5', 'cnn5s']

relu_loss_history = []
relu_grad_history = []
sigmoid_loss_history = []
sigmoid_grad_history = []

for model_name in my_models_list:
    if model_name.startswith('mlp'):
        continue

    print(f"Training {model_name}...")

    model_init_time = time.time()
    best_performance = 0
    best_weights = None

    for tc in range(TRAIN_COUNT):

        if model_name == 'mlp1':
            model = my_models.MLP1(1024, 32, 10).to(device)
        elif model_name == 'mlp1s':
            model = my_models.MLP1S(1024, 32, 10).to(device)
        elif model_name == 'mlp2':
            model = my_models.MLP2(1024, 32, 64, 10).to(device)
        elif model_name == 'mlp2s':
            model = my_models.MLP2S(1024, 32, 64, 10).to(device)
        elif model_name == 'cnn3':
            model = my_models.CNN3().to(device)
        elif model_name == 'cnn3s':
            model = my_models.CNN3S().to(device)
        elif model_name == 'cnn4':
            model = my_models.CNN4().to(device)
        elif model_name == 'cnn4s':
            model = my_models.CNN4S().to(device)
        elif model_name == 'cnn5':
            model = my_models.CNN5().to(device)
        elif model_name == 'cnn5s':
            model = my_models.CNN5S().to(device)

        # Initialize model
        criterion = nn.CrossEntropyLoss()
        optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0)

        train_generator = torch.utils.data.DataLoader(train_data,
```

```
batch_size=BATCH_SIZE, shuffle=True)

        for epoch in range(EPOCH_SIZE):
            # Train the model
            start_time = time.time()
            total_step = len(train_generator)

            train_loss = 0
            train_acc = 0

            for i, data in enumerate(train_generator):
                model.train()
                inputs, labels = data
                train_inputs, train_labels = inputs.to(device),
labels.to(device)

                # Zero the parameter gradients
                optimizer.zero_grad()

                # Forward pass
                train_outputs = model(train_inputs)

                # Define loss
                loss = criterion(train_outputs, train_labels)

                # Backward
                loss.backward()

                # Update the parameters
                optimizer.step()

                model.eval()

                # Compute the loss
                train_loss += loss.item()

                if i % 10 == 0:

                    # Move model to CPU to compute the gradient
                    model.to('cpu')
                    # Get the gradient of the first layer
                    weight = model.first.weight.grad
                    # To indicate the gradient of the first layer better on
the plot
                    train_grad = np.linalg.norm(weight)
                    # Move model back to GPU
                    model.to(device)

                    # Sigmoid based models are saved in a different list
                    if model_name.endswith('s'):
                        sigmoid_loss_history.append(train_loss / (i + 1))
                        sigmoid_grad_history.append(train_grad)

                    # ReLU based models are saved in a different list
                    else:
                        relu_loss_history.append(train_loss / (i + 1))
                        relu_grad_history.append(train_grad)
```

```
            epoch_time = time.time() - start_time
            print(
                f"Epoch [{epoch + 1}/{EPOCH_SIZE}], Epoch Time:
{epoch_time:.4f} s  Gradient: {train_grad:.4f}  Loss: {train_loss /
total_step:.4f}")

        took_time = time.time() - model_init_time
        print(f"Training [{tc + 1}/{TRAIN_COUNT}], {took_time:.4f} s, Test
Accuracy: {best_performance:.4f}")

    if model_name.endswith('s'):
        # Save the results
        result_dict = {
            'name': model_name.replace('s', ''),
            'relu_loss_curve': relu_loss_history,
            'relu_grad_curve': relu_grad_history,
            'sigmoid_loss_curve': sigmoid_loss_history,
            'sigmoid_grad_curve': sigmoid_grad_history,
        }

        relu_loss_history = []
        relu_grad_history = []
        sigmoid_loss_history = []
        sigmoid_grad_history = []

        # Save the results to a file
        filename = 'results/question_4_' + model_name.replace('s', '') +
'.pkl'
        with open(filename, 'wb') as f:
            pickle.dump(result_dict, f)

        took_time = time.time() - model_init_time
        print(f"All process for {model_name}: {took_time:.4f} s")

took_time = time.time() - init_time
print(f"All process: {took_time:.4f} s")
```

Appendix IV

question_5.py

```python
# Created by Deniz Karakay at 19.04.2023
# Filename: question_5.py

import torch
import torch.nn as nn
import torchvision
from torchvision import transforms
import time
from torch.utils.data import SubsetRandomSampler
from sklearn.model_selection import train_test_split
import pickle
import models as my_models

# Hyper-parameters
EPOCH_SIZE = 20
BATCH_SIZE = 50
TRAIN_COUNT = 1

# I tested and saw that CPU is faster than GPU on M1 Pro for MLPs

# CPU
# device = torch.device("cpu")

# MPS for GPU support on M1
device = torch.device("mps")

print(f"Using device: {device}...")

init_time = time.time()

# Transformations
transform = transforms.Compose([
    torchvision.transforms.ToTensor(),
    transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5)),
    torchvision.transforms.Grayscale()
])

# Load data
train_data = torchvision.datasets.CIFAR10('../Q3/data/', train=True,
download=True, transform=transform)

# Print image size and label of first image in dataset (should be 32x32 and
6)
img, label = train_data[0]
print("Sample image size: ", img.size())

# Split data into training and validation
train_data, valid_data = train_test_split(train_data, test_size=0.1,
random_state=42)

# Load test data
test_data = torchvision.datasets.CIFAR10('../Q3/data', train=False,
download=True, transform=transform)
```

```
print(f"Training data size: {len(train_data)}")
print(f"Validation data size: {len(valid_data)}")
print(f"Test data size: {len(test_data)}")

valid_generator = torch.utils.data.DataLoader(valid_data,
batch_size=BATCH_SIZE, shuffle=False)

my_models_list = ['cnn4_1', 'cnn4_01', 'cnn4_001']

train_acc_history_total = []
train_loss_history_total = []
valid_acc_history_total = []

# Train models and save results to lists for plotting later
for model_name in my_models_list:
    if model_name.startswith('mlp'):
        continue

    print(f"Training {model_name}...")

    model_init_time = time.time()
    best_performance = 0
    best_weights = None

    for tc in range(TRAIN_COUNT):

        # Initialize model for learning rate = 0.1
        if model_name == 'cnn4_1':
            lr = 0.1
            model = my_models.CNN4().to(device)

        # Initialize model for learning rate = 0.01
        elif model_name == 'cnn4_01':
            lr = 0.01
            model = my_models.CNN4().to(device)

        # Initialize model for learning rate = 0.001
        elif model_name == 'cnn4_001':
            lr = 0.001
            model = my_models.CNN4().to(device)

        # Initialize model
        criterion = nn.CrossEntropyLoss()
        optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=0)

        train_acc_history = []
        train_loss_history = []
        valid_acc_history = []
        test_acc_history = []

        train_generator = torch.utils.data.DataLoader(train_data,
batch_size=BATCH_SIZE, shuffle=True)

        for epoch in range(EPOCH_SIZE):

            # Train the model
```

```
            start_time = time.time()
            total_step = len(train_generator)

            train_loss = 0
            train_acc = 0

            for i, data in enumerate(train_generator):
                model.train()
                inputs, labels = data
                train_inputs, train_labels = inputs.to(device),
labels.to(device)

                # Zero the parameter gradients
                optimizer.zero_grad()

                # Forward + backward + optimize
                train_outputs = model(train_inputs)
                loss = criterion(train_outputs, train_labels)

                loss.backward()
                optimizer.step()

                model.eval()

                # Compute the loss
                train_loss += loss.item()

                # Compute the accuracy
                pred = train_outputs.argmax(dim=1, keepdim=True)
                train_total = train_labels.size(0)
                train_correct =
pred.eq(train_labels.view_as(pred)).sum().item()
                train_acc += (train_correct / train_total) * 100

                # Print statistics every 10 batches
                if i % 10 == 0:

                    # Take average of train loss and accuracy
                    train_acc_history.append(train_acc / (i + 1))
                    train_loss_history.append(train_loss / (i + 1))

                    valid_correct = 0
                    valid_total = 0

                    # Compute the validation accuracy
                    with torch.no_grad():
                        for data in valid_generator:
                            inputs, labels = data
                            valid_inputs, valid_labels = inputs.to(device),
labels.to(device)

                            # Compute the outputs and predictions
                            valid_outputs = model(valid_inputs)
                            _, valid_predicted =
torch.max(valid_outputs.data, 1)

                            # Track the statistics
```

```
                            valid_total += valid_labels.size(0)
                            valid_correct += (valid_predicted ==
valid_labels).sum().item()

                        valid_acc = (valid_correct / valid_total) * 100
                        valid_acc_history.append(valid_acc)

            epoch_time = time.time() - start_time
            print(
                f"Epoch [{epoch + 1}/{EPOCH_SIZE}], Epoch Time:
{epoch_time:.4f} s, Train Loss: {train_loss_history[-1]:.4f}, "
                f"Train Accuracy: {train_acc_history[-1]:.3f}, Validation
Accuracy: {valid_acc_history[-1]:.3f}")

        train_acc_history_total.append(train_acc_history)
        train_loss_history_total.append(train_loss_history)
        valid_acc_history_total.append(valid_acc_history)
        took_time = time.time() - model_init_time
        print(f"Training [{tc + 1}/{TRAIN_COUNT}], {took_time:.4f} s, Test
Accuracy: {best_performance:.4f}")

    if model_name.endswith('001'):
        # Save the results
        result_dict = {
            'name': model_name.replace('s', ''),
            'loss_curve_1': train_loss_history_total[0],
            'loss_curve_01': train_loss_history_total[1],
            'loss_curve_001': train_loss_history_total[2],
            'val_acc_curve_1': valid_acc_history_total[0],
            'val_acc_curve_01': valid_acc_history_total[1],
            'val_acc_curve_001': valid_acc_history_total[2],
        }

        # Save the results to a file
        filename = 'results/question_5_' + model_name + '.pkl'
        with open(filename, 'wb') as f:
            pickle.dump(result_dict, f)

        took_time = time.time() - model_init_time
        print(f"All process for {model_name}: {took_time:.4f} s")

took_time = time.time() - init_time
print(f"All process: {took_time:.4f} s")
```

Appendix V

question_5_part_2.py

```python
# Created by Deniz Karakay at 20.04.2023
# Filename: question_5_part2.py


import torch
import torch.nn as nn
import torchvision
from torchvision import transforms
import time
from torch.utils.data import SubsetRandomSampler
from sklearn.model_selection import train_test_split
import pickle
import models as my_models

# Hyper-parameters
EPOCH_SIZE = 45
BATCH_SIZE = 50
TRAIN_COUNT = 1

# I tested and saw that CPU is faster than GPU on M1 Pro for MLPs

# CPU
# device = torch.device("cpu")

# MPS for GPU support on M1
device = torch.device("mps")

print(f"Using device: {device}...")

init_time = time.time()

# Transformations
transform = transforms.Compose([
    torchvision.transforms.ToTensor(),
    transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5)),
    torchvision.transforms.Grayscale()
])

# Load data
train_data = torchvision.datasets.CIFAR10('../Q3/data/', train=True,
download=True, transform=transform)

# Print image size and label of first image in dataset (should be 32x32 and
6)
img, label = train_data[0]
print("Sample image size: ", img.size())

# Split data into training and validation
train_data, valid_data = train_test_split(train_data, test_size=0.1,
random_state=42)

# Load test data
test_data = torchvision.datasets.CIFAR10('../Q3/data', train=False,
```

```
download=True, transform=transform)

print(f"Training data size: {len(train_data)}")
print(f"Validation data size: {len(valid_data)}")
print(f"Test data size: {len(test_data)}")

valid_generator = torch.utils.data.DataLoader(valid_data,
batch_size=BATCH_SIZE, shuffle=False)

my_models_list = ['cnn4_1']

train_acc_history_total = []
train_loss_history_total = []
valid_acc_history_total = []

model_name = my_models_list[0]
print(f"Training {model_name}...")

model_init_time = time.time()
best_performance = 0
best_weights = None

for tc in range(TRAIN_COUNT):

    # Initialize model for learning rate = 0.1
    if model_name == 'cnn4_1':
        lr = 0.1
        model = my_models.CNN4().to(device)

    # Initialize model for learning rate = 0.01
    elif model_name == 'cnn4_01':
        lr = 0.01
        model = my_models.CNN4().to(device)

    # Initialize model for learning rate = 0.001
    elif model_name == 'cnn4_001':
        lr = 0.001
        model = my_models.CNN4().to(device)

    # Initialize model
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=0)

    train_acc_history = []
    train_loss_history = []
    valid_acc_history = []
    test_acc_history = []

    train_generator = torch.utils.data.DataLoader(train_data,
batch_size=BATCH_SIZE, shuffle=True)

    for epoch in range(EPOCH_SIZE):
        # Train the model
        start_time = time.time()
        total_step = len(train_generator)

        train_loss = 0
```

```
        train_acc = 0

        if epoch == 10:
            lr = 0.01
            criterion = nn.CrossEntropyLoss()
            optimizer = torch.optim.SGD(model.parameters(), lr=lr,
momentum=0)
        if epoch == 20:
            lr = 0.001
            criterion = nn.CrossEntropyLoss()
            optimizer = torch.optim.SGD(model.parameters(), lr=lr,
momentum=0)
        for i, data in enumerate(train_generator):
            model.train()
            inputs, labels = data
            train_inputs, train_labels = inputs.to(device), labels.to(device)

            # Zero the parameter gradients
            optimizer.zero_grad()

            # Forward + backward + optimize
            train_outputs = model(train_inputs)
            loss = criterion(train_outputs, train_labels)

            loss.backward()
            optimizer.step()

            model.eval()

            # Compute the loss
            train_loss += loss.item()

            # Compute the accuracy
            pred = train_outputs.argmax(dim=1, keepdim=True)
            train_total = train_labels.size(0)
            train_correct = pred.eq(train_labels.view_as(pred)).sum().item()
            train_acc += (train_correct / train_total) * 100

            if i % 10 == 0:

                # Take average of train loss and accuracy
                train_acc_history.append(train_acc / (i + 1))
                train_loss_history.append(train_loss / (i + 1))

                valid_correct = 0
                valid_total = 0
                with torch.no_grad():
                    for data in valid_generator:
                        inputs, labels = data
                        valid_inputs, valid_labels = inputs.to(device),
labels.to(device)

                        # Compute the outputs and predictions
                        valid_outputs = model(valid_inputs)
                        _, valid_predicted = torch.max(valid_outputs.data, 1)

                        # Track the statistics
```

```
                        valid_total += valid_labels.size(0)
                        valid_correct += (valid_predicted ==
valid_labels).sum().item()

                    valid_acc = (valid_correct / valid_total) * 100
                    valid_acc_history.append(valid_acc)

        epoch_time = time.time() - start_time
        print(
            f"Epoch [{epoch + 1}/{EPOCH_SIZE}], Epoch Time: {epoch_time:.4f}
s, Train Loss: {train_loss_history[-1]:.4f}, "
            f"Train Accuracy: {train_acc_history[-1]:.3f}, Validation
Accuracy: {valid_acc_history[-1]:.3f}")
        print(f"Learning rate: {lr}")

        train_acc_history_total.append(train_acc_history)
        train_loss_history_total.append(train_loss_history)
        valid_acc_history_total.append(valid_acc_history)
        took_time = time.time() - model_init_time
        print(f"Training [{tc + 1}/{TRAIN_COUNT}], {took_time:.4f} s, Test
Accuracy: {best_performance:.4f}")

        # Evaluate the model on test set
        test_correct = 0
        test_total = 0
        with torch.no_grad():
            test_generator = torch.utils.data.DataLoader(test_data,
batch_size=BATCH_SIZE, shuffle=False)
            model.eval()
            for data in test_generator:
                inputs, labels = data
                test_inputs, test_labels = inputs.to(device),
labels.to(device)

                # Compute the outputs and predictions
                test_outputs = model(test_inputs)
                _, test_predicted = torch.max(test_outputs.data, 1)

                # Track the statistics
                test_total += test_labels.size(0)
                test_correct += (test_predicted == test_labels).sum().item()
                test_acc = (test_correct / test_total) * 100

            # Save best weights
            if test_acc > best_performance:
                best_performance = test_acc
                best_weights = model.first.weight.data.cpu().numpy()

            # Save test accuracy
            test_acc_history.append(test_acc)

        took_time = time.time() - model_init_time
        print(f"Training [{tc + 1}/{TRAIN_COUNT}], {took_time:.4f} s, Test
Accuracy: {best_performance:.4f}")

        print(best_performance)
```

```
result_dict = {
    'name': model_name,
    'val_acc_curve': valid_acc_history_total[0],
}

# Save the results to a file
filename = 'results/question_5_part2_final_cnn4' + '.pkl'
with open(filename, 'wb') as f:
    pickle.dump(result_dict, f)

took_time = time.time() - model_init_time
print(f"All process for {model_name}: {took_time:.4f} s")

took_time = time.time() - init_time
print(f"All process: {took_time:.4f} s")
print(f"All process for {model_name}: {took_time:.4f} s")
```

Appendix VI

question_5_part_2.py

```
# Created by Deniz Karakay at 20.04.2023
# Filename: visualize_valid_final.py

import pickle
import numpy as np
from matplotlib import pyplot as plt
import os


# Custom part5Plots function
def custom_plot_for_part_2(result, save_dir='', filename='', show_plot=True):
    if isinstance(result, (list, tuple)):
        result = result[0]

    color_list = ['#dad420', ]
    style_list = ['-', '--']

    num_curves = 1

    plot_args = [{'c': color_list[k],
                  'linestyle': style_list[0],
                  'linewidth': 2} for k in range(num_curves)]

    key_suffixes = ['1', ]

    font_size = 18

    fig, axes = plt.subplots(1, figsize=(16, 12))

    fig.suptitle('training of cnn4 with three different learning rates: 0.1,
0.01, 0.001',
                 fontsize=font_size, y=0.025)

    # training loss and validation accuracy
    axes.set_title('validation_accuracy', loc='right', fontsize=font_size)
    for key_suffix, plot_args in zip(key_suffixes, plot_args):
        acc_curve = result['val_acc_curve']
        label = 'lr=0.1, 0.01 and 0.001'

        axes.plot(np.arange(1, len(acc_curve) + 1),
                  acc_curve, label=label, **plot_args)
        axes.set_xlabel(xlabel='step', fontsize=font_size)
        axes.set_ylabel(ylabel='accuracy', fontsize=font_size)
        axes.tick_params(labelsize=12)

    # global legend
    lines = axes.get_lines()
    fig.legend(labels=[line._label for line in lines],
               ncol=3, loc="upper center", fontsize=font_size,
               handles=lines)

    if show_plot:
        plt.show()
```

```
    fig.savefig(os.path.join(save_dir, filename + '.png'))


filename = 'results/question_5_part2_final_cnn4.pkl'
with open(filename, 'rb') as f:
    loaded_dict = pickle.load(f)
    print(loaded_dict.keys())

    custom_plot_for_part_2(loaded_dict, save_dir='',
filename='question_5_plots_part2_final')
```