1.

1.1

In these experiments, our function takes size of 1024 as an input and gives output size of 10. So that function form of this function will be 1024->10.

Binary classification:

$$\big(\mathrm{ylog(p)} + (1-\mathrm{y})\mathrm{log}(1-\mathrm{p})\big)\ if\ M = 2$$

Multiclass Classification:

$$= -\sum_{c=1}^{M}\big(y_{o,c} * \log\big(p_{o,c}\big)\big)\ \ if\ M > 2$$

In this experiment we are going to use Multiclass Classification since our output size has 10 which is 10>2.

In addition to this, the loss function penalizes the difference between predicted probabilities and accurate class labels. If the predicted probability is close to 1 for the correct class, the loss value will be close to 0. The loss value will be large if the predicted probability is close to 0 for the correct class. This logarithmic increase in loss value with increasing differences between predicted and actual probabilities ensures that the model is encouraged to make accurate predictions and minimize the dissimilarity between predicted and actual probabilities.

1.2

We knew that there are few equations from our notes.

$$\mathrm{W_{k+1}} = \mathrm{W_k} - \gamma * (\nabla\, \mathrm{L}\,(\mathrm{W} = @\mathrm{W_K}))$$

$$\nabla \mathrm{L}(\mathrm{W} = @\mathrm{W_K}) = (\mathrm{W_k} - \mathrm{W_{k+1}}) \,/\, \gamma$$

1.3

1.3.1 Batch size refers to a subset of the training data that is used to update the model's weights during each iteration of the training process. Epoch size on the other hand, refers to a complete pass through the entire training dataset during the training process. During an epoch, multiple batches are sequentially processed to update the weights of the ANN.

1.3.2

If the dataset has N samples and the batch size is B, the number of batches per epoch would be given by the formula:

Number of batches per epoch = N / B

This is because each batch contains B samples, and in order to process the entire dataset of N samples, N/B batches would need to be processed sequentially to complete one epoch of training.

1.3.3

Number of SGD iterations = (N/B) * E

This is because in each epoch, N/B batches are processed, and this process is repeated for E epochs.

1.4

1.4.1

- Number of parameters in connections from input layer to first hidden layer = D_{in} * H_1
- Number of parameters in connections between k_{th} and $(k-1)_{th}$ hidden layers = H_{k-1} * H_k
- Number of parameters in connections from output layer = D_{out}

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

1.4.2

It is like previous question, so that we can say that,

- Number of parameters in connections from input layer to first hidden layer = $(C_{in} * C_1 * H_{in} * W_{in})$
- Number of parameters in connections between k_{th} and $(k-1)_{th}$ hidden layers = $H_k * W_k$

$$\#\ of\ Parameters = (C_{in} * C_1 * H_{in} * W_{in}) + \sum_{k=1}^{k=K-1} (C_k * C_{k+1} * H_k * W_k) + \sum_{k=1}^{k=K} (C_k)$$

2

2.1

Figure 1: .png file of Conv code

2.2

2.2.1 Convolutional Neural Networks (CNNs) are important in image processing because they are designed to automatically learn localized features from images, use parameter sharing and spatial invariance to be computationally efficient, learn deep hierarchical representations for complex patterns, allow for transfer learning from pre-trained models, and have a wide range of applications in tasks such as image recognition, object detection, and medical imaging.

2.2.2 A kernel in a Convolutional Neural Network (CNN) is a small matrix of weights used for convolution operation. Its size determines the spatial extent it can capture from the input data and needs to match the number of channels in the input data.

2.2.3 In output picture, we can see that there are multiple types of ‘seven’ pictures.

2.2.4 The numbers in the same column of the MNIST dataset may look alike to each other, even though they belong to different images, due to consistency in writing style, similar structural features, and random sampling from the respective digit class. Machine learning algorithms, such as CNNs, can learn and generalize from these shared visual features for accurate digit classification.

2.2.5 The numbers in the same row of a MNIST dataset may not look alike to each other, even though they belong to the same image, due to natural variations in writing style and other factors. This lack of exact similarity highlights the need for machine learning algorithms to be robust to such variations for accurate digit recognition.

2.2.6 Convolutional Layers in a Convolutional Neural Network (CNN) capture shared visual features across images of the same class (numbers in the same column) and local variations within the same image (numbers in the same row). They use convolutional operations and filters to extract relevant features, allowing the network to learn and generalize from these features for accurate image recognition and classification.

3

3.1

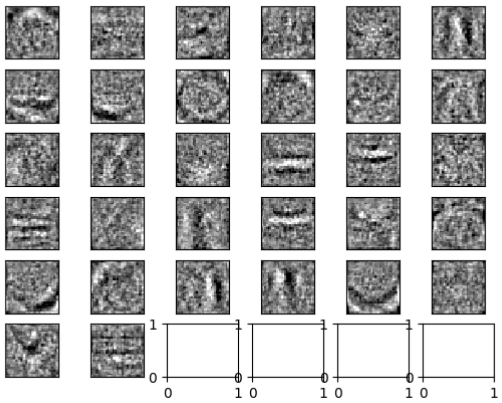

Figure 2 : Weights of MLP_1

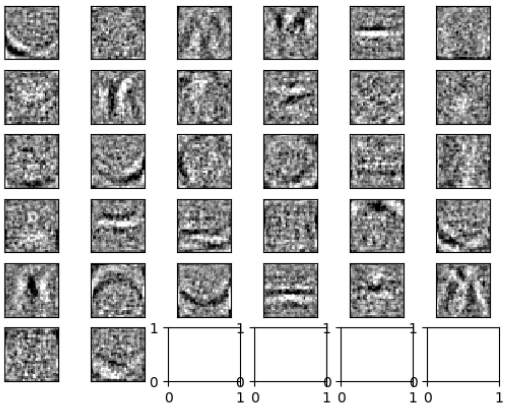

Figure 3 : Weights of MLP_2

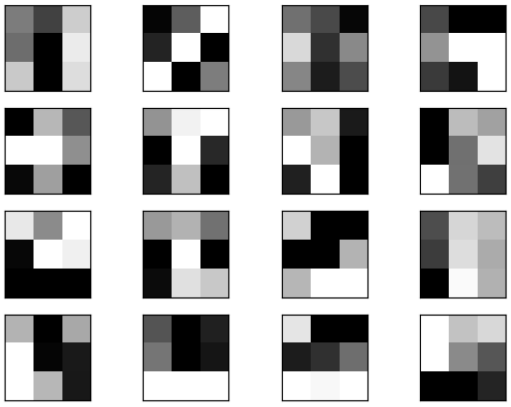

Figure 3 : Weights of CNN_3

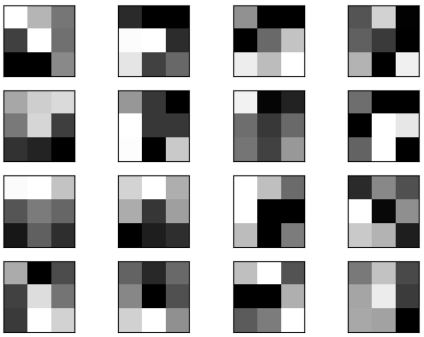

Figure 4 : Weights of CNN_4

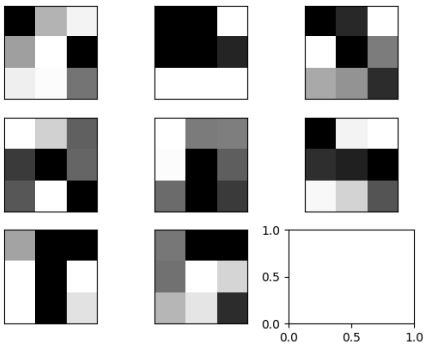

Figure 5 : Weights of CNN_5

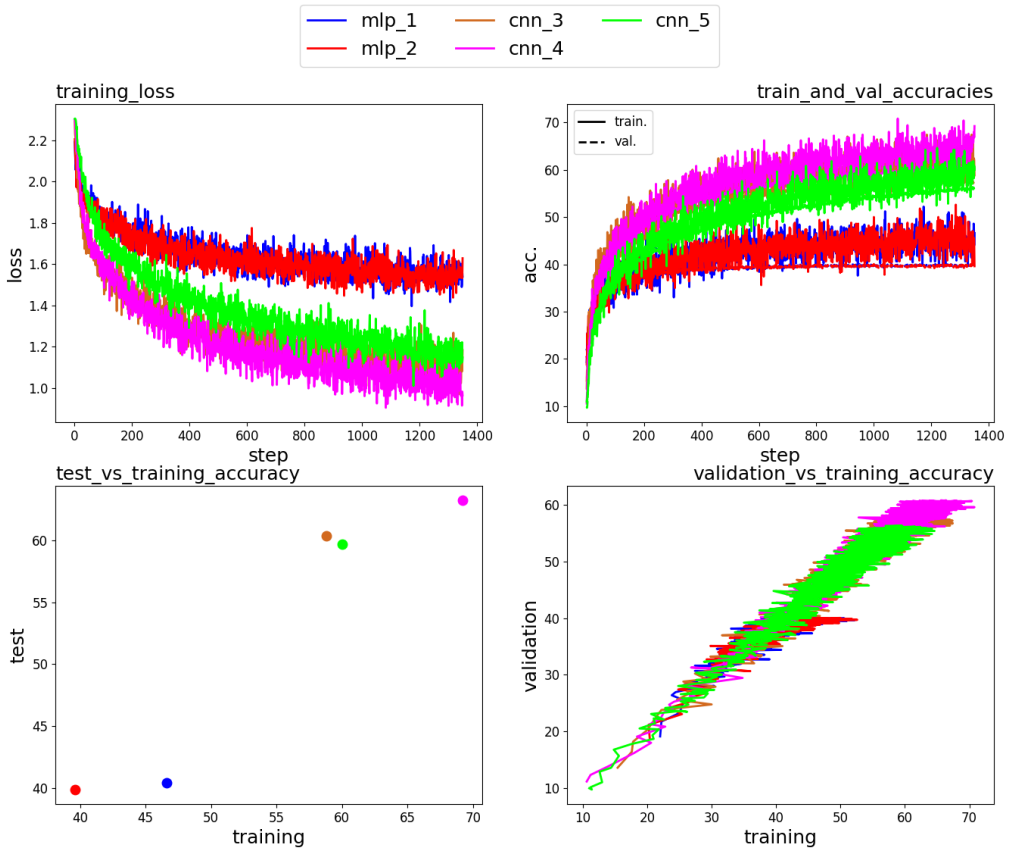

Figure 6 : Comparing of 5 Different Model

3.2

3.2.1 The generalization performance of a classifier refers to its ability to accurately classify new, unseen data beyond the training data it was trained on. It measures how well a classifier can generalize its learned knowledge to new data.

3.2.2 The generalization performance of a model is determined by the validation and test results and not by the accuracy achieved during training.

3.2.3 We can see that 'cnn_4' has the best test accuracy result; therefore, as we mentioned that generalization performance depends on validation and test result's previous part,

3.2.4 The number of parameters in a model affects its capacity and can impact its classification and generalization performance. Models with more parameters may have

higher capacity and better classification performance but can also be prone to overfitting and reduced generalization performance. Finding the right balance between model capacity and the number of parameters is crucial for achieving optimal performance. Regularization techniques and proper model selection, hyperparameter tuning, and evaluation on validation or test data are essential in determining the optimal number of parameters for a given task and dataset.

In our experiment, the order of parameter is *mlp_1>mlp_2>cnn_3>cnn_4>cnn_5*.

Cnn_5 seems the best model for generalization performance in the order above. However, we can observe that from Figure 6, It is sure that cnn_4 shows best performance.

3.2.5 The depth of a neural network architecture, which refers to the number of layers, can affect both the classification and generalization performance. Deeper architectures may improve classification performance by allowing the model to learn more complex features but may also increase the risk of overfitting. In addition, deeper architectures may be trained harder. Finding the proper depth involves a trade-off between model capacity and overfitting, and regularization techniques and valid model selection are essential for achieving optimal performance.

We can easily see that from the number of layers written in the manual.

So that in our experiment, the order of depth is *cnn_5> cnn_4> cnn_3 > mlp_2> mlp_1.*

Again, Cnn_5 seems the best model for generalization performance in the order above. However, we can observe that from Figure 6, It is sure that cnn_4 shows the best performance.

3.2.6 Interpretation is hard to do. However, if we observe figures 2 and 3, we can see some animals and some different object but they are not clear. Therefore, we can say that MLP visualizations are interpretable at some point while CNN ones absolutely are not.

3.2.7 The answer is generally no in the context of visualizations of weights in neural networks, as the weights in the first layer typically determine different features rather than specific classes. Visualizing the weights in deeper layers, such as the last layer, may correspond to specific classes, providing more interpretable information about the model's predictions. However, the interpretability of weight visualizations depends on the layer and task, with early layers capturing more general features and deeper layers capturing more class-specific information.

3.2.8 We mentioned that on part 3.2.6. whereas MLP model are interpretable at some point, but CNN models are absolutely not.

3.2.9 Let's examine CNN models first; CNN model depths are mentioned in part 3.2.5, and we observe from experimental results that cnn_4 performs the best despite having fewer parameters.

The difference between the test and training accuracy should be considered to assess generalization performance. If the testing accuracy is higher than the training accuracy, it may indicate overfitting, while if the testing accuracy is lower than the training accuracy, it may indicate underfitting. Therefore, a slight difference between test and training accuracies is desirable.

3.2.10 It seems CNN_5 has deeper layers and generalization performance; we observed it does not like this. I would pick CNN_4 due to the results. However, we do not know what will happen if we increase the epoch size. CNN_5 can beat up CNN_4; therefore, I will answer for only epoch size, which is 15. My choice would be CNN_4.

4

4.1

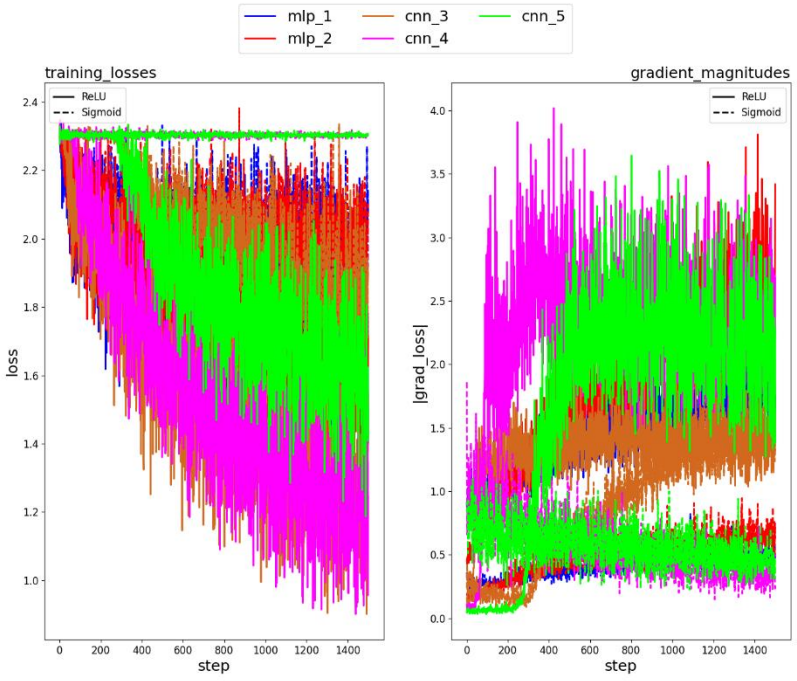

Figure 7 : Comparison of ReLu and Sigmoid functions for 5 Each models

To make more understandable I added two more plot.

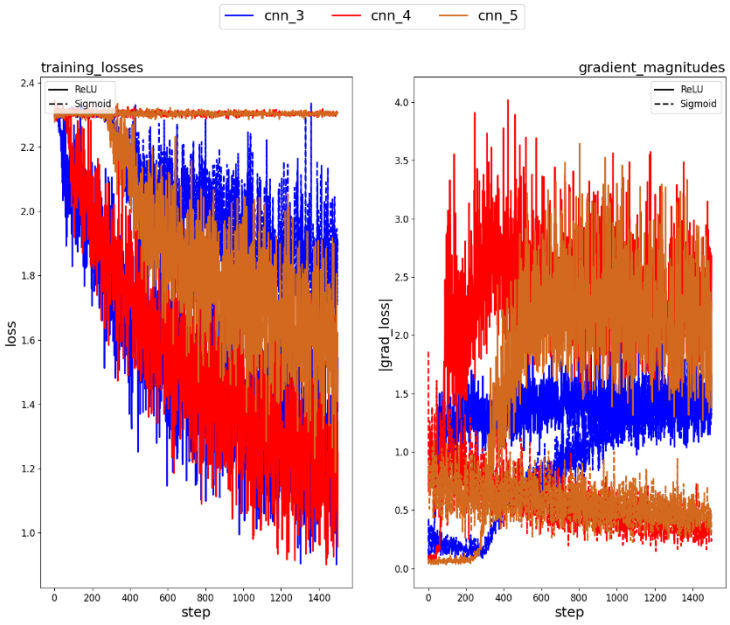

Figure 8 : Comparison of ReLu and Sigmoid functions for CNN models

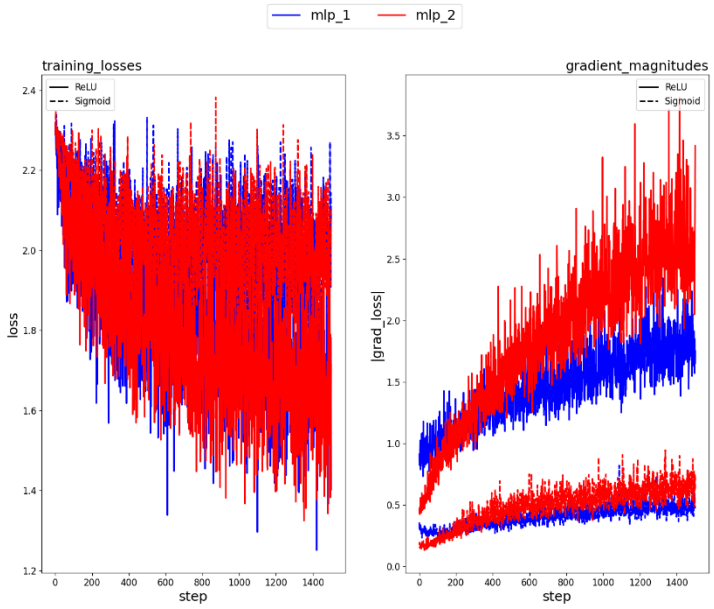

Figure 9 : Comparison of ReLu and Sigmoid functions for MLP models

4.2

4.2.1 It can be observed that when ReLu function is activated, Gradient Magnitudes becomes bigger than the Sigmoid function activated for both architectures.

When depth increases, Gradient of CNN models decrease. In this experiment we can observe this situation; however, MLP models do not show this behavior.

4.2.2 As the depth of a neural network increases, it becomes a more complex system with more steps, allowing it to learn more quickly and find optimal weight values. As we mentioned on previous part, when depth increases, Gradient of CNN models decrease. Additionally, in models that use the ReLU activation function, the output is in the range of 0 to infinity, whereas in models that use the sigmoid activation function, the output is in the range of 0 to 1. This difference in output range results in the model with ReLU having larger gradient values compared to a model with sigmoid, which can affect the gradient behavior during training.

4.2.3 Scaling inputs range of [-1.0, 1.0] is important for stable training, accurate calculations, and unbiased model predictions. Due to finding optima point is getting longer, gradient loss becomes larger.

5

5.1

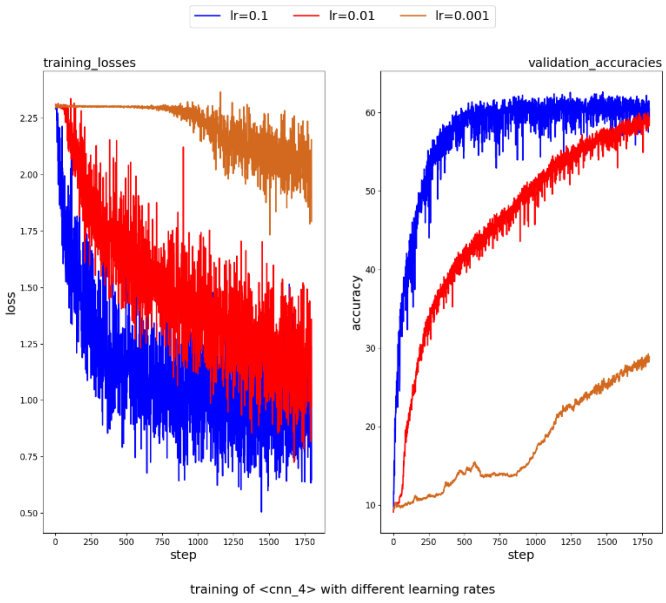

Figure 10: Training of CNN_4 with different LR's

This figure shows that when LR=0.1, the epoch size that doesn't increase point is $1000/90 \approx 11$.

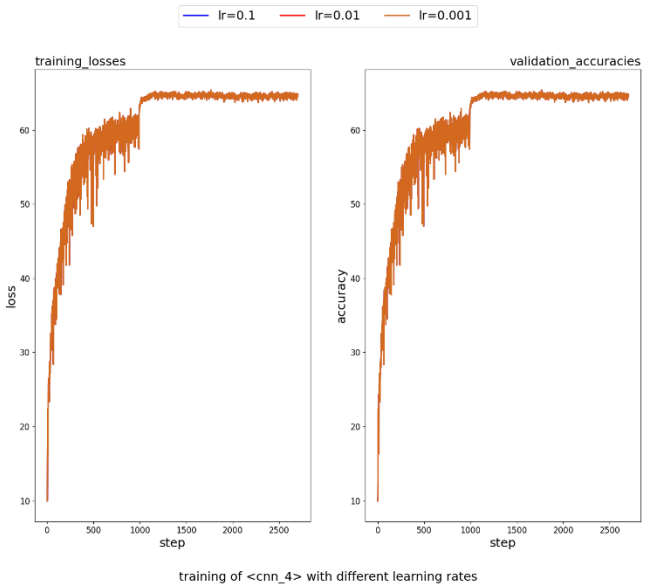

Figure 11: Training of CNN_4 with 11 epoch with LR=0.1 and after LR=0.01

This figure shows that when LR=0.01, the epoch size that doesn't increase point is $(2000 - 1000)/90 \approx 11$.

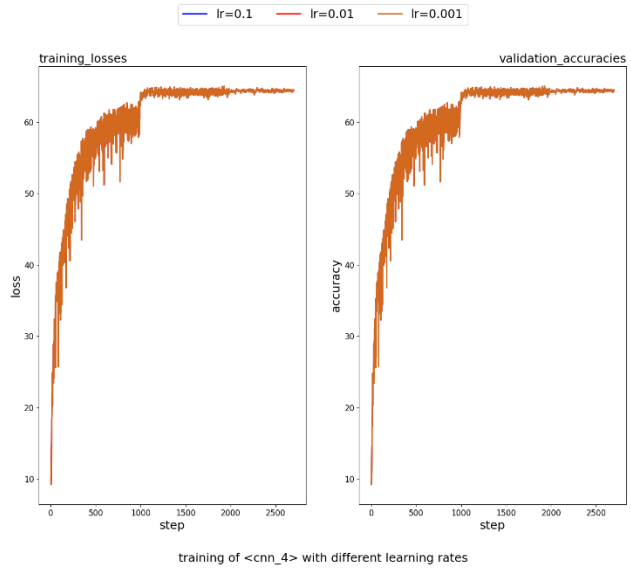

Figure 12: Training of CNN_4 with 11 epoch with LR=0.1 and 11 epoch with LR=0.01 and after LR=0.001

5.2

5.2.1 It is obvious that learning rates and convergence speed have positive correlation. If Learning rates increase, convergence speed increase.

5.2.2 The learning rate should be carefully chosen to balance fast convergence and stability. An adequately determined learning rate can help the model converge to a better point more quickly, leading to faster training and better performance. Training time will become bigger if the learning rate is too low, and even the model can get stuck in local optima or saddle points; if the learning rate is too high, the model can be unstable.

5.2.3 In this experiment we have three learning rates. If we observe the results, we can realize that 0.1 is unstable LR, 0.001 is slower LR. Therefore, it is obvious that an adequately determined learning rate is 0.01.

5.2.4 In both method accuracies are almost the same but SGD is faster.

Q2

```python
import os
import torch
from torchvision.utils import make_grid
from matplotlib import pyplot as plt
import utils
import numpy as np


def my_conv2d(input, kernel):
    batch_size, in_channels, in_height, in_width = input.shape
    out_channels, _, kernel_height, kernel_width = kernel.shape  # I wrote "_" since we had input_channels value from above

    out_height = in_height - kernel_height + 1
    out_width = in_width - kernel_width + 1

    # initialize output tensor
    out = torch.zeros(batch_size, out_channels, out_height, out_width)

    for b in range(batch_size):
        for c_out in range(out_channels):
            for i in range(out_height):
                for j in range(out_width):
                    # extract the input patch corresponding to the current output pixel
                    patch = input[b, :, i: i + kernel_height, j: j + kernel_width]

                    # apply the kernel to the patch and sum the result
                    out[b, c_out, i, j] = torch.sum(torch.from_numpy(patch) * kernel[c_out])

    return out


# input shape: [batch size, input_channels, input_height, input_width]
input = np.load("data\\samples_7.npy")
# input shape: [output_channels, input_channels, filter_height, filter width]
kernel = np.load("data\\kernel.npy")
out = my_conv2d(input, kernel)
utils.part2Plots(out, 64, "ResultQ2\\", "my_first_conv2D")
```

Script for ANN models

```python
import torchvision
import torch
import torch.nn as nn
import torchvision.transforms as transforms
from sklearn.model_selection import train_test_split

BATCH_SIZE = 50
EPOCH_SIZE = 15
TRAIN_SIZE = 10

class mlp_1(torch.nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(mlp_1, self).__init__()
        self.input_size = input_size
        self.fc1 = torch.nn.Linear(input_size, hidden_size)
        self.fc2 = torch.nn.Linear(hidden_size, num_classes)
        self.relu = torch.nn.ReLU()

    def forward(self, x):
        x = x.view(-1, self.input_size)
        hidden = self.fc1(x)
        relu = self.relu(hidden)
        output = self.fc2(relu)
        return output

class mlp_1_sigmoid(torch.nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(mlp_1_sigmoid, self).__init__()
        self.input_size = input_size
        self.fc1 = torch.nn.Linear(input_size, hidden_size)
        self.fc2 = torch.nn.Linear(hidden_size, num_classes)
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        x = x.view(-1, self.input_size)
        hidden = self.fc1(x)
        relu = self.sigmoid(hidden)
        output = self.fc2(relu)
        return output


class mlp_2(torch.nn.Module):
    def __init__(self, input_size, hidden_size, hidden_size2, num_classes):
        super(mlp_2, self).__init__()
        self.input_size = input_size
        self.fc1 = torch.nn.Linear(input_size, hidden_size)
        self.fc2 = torch.nn.Linear(hidden_size, hidden_size2, bias=False)
        self.fc3 = torch.nn.Linear(hidden_size2, num_classes)
        self.relu = torch.nn.ReLU()

    def forward(self, x):
        x = x.view(-1, self.input_size)
        hidden = self.fc1(x)
        relu = self.relu(hidden)
        hidden2 = self.fc2(relu)
        output = self.fc3(hidden2)
        return output


class mlp_2_sigmoid(torch.nn.Module):
```

```
    def __init__(self, input_size, hidden_size, hidden_size2, num_classes):
        super(mlp_2_sigmoid, self).__init__()
        self.input_size = input_size
        self.fc1 = torch.nn.Linear(input_size, hidden_size)
        self.fc2 = torch.nn.Linear(hidden_size, hidden_size2, bias=False)
        self.fc3 = torch.nn.Linear(hidden_size2, num_classes)
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        x = x.view(-1, self.input_size)
        hidden = self.fc1(x)
        sigmoid = self.sigmoid(hidden)
        hidden2 = self.fc2(sigmoid)
        output = self.fc3(hidden2)
        return output


class cnn_3(torch.nn.Module):


    def __init__(self, input_size, num_classes):
        super(cnn_3, self).__init__()
        self.input_size = input_size
        self.conv1 = torch.nn.Conv2d(in_channels=1, out_channels=16, kernel_size=(3, 3), stride=1, padding='valid')
        self.relu1 = torch.nn.ReLU()
        self.conv2 = torch.nn.Conv2d(in_channels=16, out_channels=8, kernel_size=(5, 5), stride=1, padding='valid')
        self.relu2 = torch.nn.ReLU()
        self.pool1 = torch.nn.MaxPool2d(kernel_size=(2, 2), stride=2, padding=0)
        self.conv3 = torch.nn.Conv2d(in_channels=8, out_channels=16, kernel_size=(7, 7), stride=1, padding='valid')
        self.pool2 = torch.nn.MaxPool2d(kernel_size=(2, 2), stride=2, padding=0)

        self.fc1 = torch.nn.Linear(in_features=16 * 3 * 3, out_features=num_classes)

    def forward(self, x):
        hidden1 = self.conv1(x)
        relu1 = self.relu1(hidden1)
        hidden2 = self.conv2(relu1)
        relu2 = self.relu2(hidden2)
        pool1 = self.pool1(relu2)
        hidden3 = self.conv3(pool1)
        pool2 = self.pool2(hidden3)
        # Reshaping linear input
        pool2 = pool2.view(BATCH_SIZE, 16 * 3 * 3)
        output = self.fc1(pool2)
        return output


class cnn_3_sigmoid(torch.nn.Module):


    def __init__(self, input_size, num_classes):
        super(cnn_3_sigmoid, self).__init__()
        self.input_size = input_size
        self.conv1 = torch.nn.Conv2d(in_channels=1, out_channels=16, kernel_size=(3, 3), stride=1, padding='valid')
        self.sigmoid1 = torch.nn.Sigmoid()
        self.conv2 = torch.nn.Conv2d(in_channels=16, out_channels=8, kernel_size=(5, 5), stride=1, padding='valid')
        self.sigmoid2 = torch.nn.Sigmoid()
        self.pool1 = torch.nn.MaxPool2d(kernel_size=(2, 2), stride=2, padding=0)
        self.conv3 = torch.nn.Conv2d(in_channels=8, out_channels=16, kernel_size=(7, 7), stride=1, padding='valid')
        self.pool2 = torch.nn.MaxPool2d(kernel_size=(2, 2), stride=2, padding=0)

        self.fc1 = torch.nn.Linear(in_features=16 * 3 * 3, out_features=num_classes)

    def forward(self, x):
        hidden1 = self.conv1(x)
        sigmoid1 = self.sigmoid1(hidden1)
        hidden2 = self.conv2(sigmoid1)
        sigmoid2 = self.sigmoid2(hidden2)
        pool1 = self.pool1(sigmoid2)
        hidden3 = self.conv3(pool1)
        pool2 = self.pool2(hidden3)
        # Reshaping linear input
        pool2 = pool2.view(BATCH_SIZE, 16 * 3 * 3)
        output = self.fc1(pool2)
        return output


class cnn_4_sigmoid(torch.nn.Module):


    def __init__(self, input_size, num_classes):
        super(cnn_4_sigmoid, self).__init__()
        self.input_size = input_size
        self.conv1 = torch.nn.Conv2d(in_channels=1, out_channels=16, kernel_size=(3, 3), stride=1, padding='valid')
        self.sigmoid1 = torch.nn.Sigmoid()
        self.conv2 = torch.nn.Conv2d(in_channels=16, out_channels=8, kernel_size=(3, 3), stride=1, padding='valid')
        self.sigmoid2 = torch.nn.Sigmoid()
        self.conv3 = torch.nn.Conv2d(in_channels=8, out_channels=16, kernel_size=(5, 5), stride=1, padding='valid')
        self.sigmoid3 = torch.nn.Sigmoid()
        self.pool1 = torch.nn.MaxPool2d(kernel_size=(2, 2), stride=2)
        self.conv4 = torch.nn.Conv2d(in_channels=16, out_channels=16, kernel_size=(5, 5), stride=1, padding='valid')
        self.sigmoid4 = torch.nn.Sigmoid()
        self.pool2 = torch.nn.MaxPool2d(kernel_size=(2, 2), stride=2)

        self.fc1 = torch.nn.Linear(in_features=16 * 4 * 4, out_features=num_classes)

    def forward(self, x):
        hidden1 = self.conv1(x)
        sigmoid1 = self.sigmoid1(hidden1)
        hidden2 = self.conv2(sigmoid1)
        sigmoid2 = self.sigmoid2(hidden2)
        hidden3 = self.conv3(sigmoid2)
```

```
        sigmoid3 = self.sigmoid3(hidden3)
        pool1 = self.pool1(sigmoid3)
        hidden4 = self.conv4(pool1)
        sigmoid4 = self.sigmoid4(hidden4)
        pool2 = self.pool2(sigmoid4)
        pool2 = pool2.view(50, 16 * 4 * 4)
        output = self.fc1(pool2)
        return output


class cnn_4(torch.nn.Module):


    def __init__(self, input_size, num_classes):
        super(cnn_4, self).__init__()
        self.input_size = input_size
        self.conv1 = torch.nn.Conv2d(in_channels=1, out_channels=16, kernel_size=(3, 3), stride=1, padding='valid')
        self.relu1 = torch.nn.ReLU()
        self.conv2 = torch.nn.Conv2d(in_channels=16, out_channels=8, kernel_size=(3, 3), stride=1, padding='valid')
        self.relu2 = torch.nn.ReLU()
        self.conv3 = torch.nn.Conv2d(in_channels=8, out_channels=16, kernel_size=(5, 5), stride=1, padding='valid')
        self.relu3 = torch.nn.ReLU()
        self.pool1 = torch.nn.MaxPool2d(kernel_size=(2, 2), stride=2)
        self.conv4 = torch.nn.Conv2d(in_channels=16, out_channels=16, kernel_size=(5, 5), stride=1, padding='valid')
        self.relu4 = torch.nn.ReLU()
        self.pool2 = torch.nn.MaxPool2d(kernel_size=(2, 2), stride=2)

        self.fc1 = torch.nn.Linear(in_features=16 * 4 * 4, out_features=num_classes)

    def forward(self, x):
        hidden1 = self.conv1(x)
        relu1 = self.relu1(hidden1)
        hidden2 = self.conv2(relu1)
        relu2 = self.relu2(hidden2)
        hidden3 = self.conv3(relu2)
        relu3 = self.relu3(hidden3)
        pool1 = self.pool1(relu3)
        hidden4 = self.conv4(pool1)
        relu4 = self.relu4(hidden4)
        pool2 = self.pool2(relu4)
        pool2 = pool2.view(50, 16 * 4 * 4)
        output = self.fc1(pool2)
        return output


class cnn_5(torch.nn.Module):


    def __init__(self, input_size, num_classes):
        super(cnn_5, self).__init__()
        self.input_size = input_size
        self.conv1 = torch.nn.Conv2d(in_channels=1, out_channels=8, kernel_size=(3, 3), stride=1, padding='valid')
        self.relu1 = torch.nn.ReLU()
        self.conv2 = torch.nn.Conv2d(in_channels=8, out_channels=16, kernel_size=(3, 3), stride=1, padding='valid')
        self.relu2 = torch.nn.ReLU()
        self.conv3 = torch.nn.Conv2d(in_channels=16, out_channels=8, kernel_size=(3, 3), stride=1, padding='valid')
        self.relu3 = torch.nn.ReLU()
        self.conv4 = torch.nn.Conv2d(in_channels=8, out_channels=16, kernel_size=(3, 3), stride=1, padding='valid')
        self.relu4 = torch.nn.ReLU()
        self.pool1 = torch.nn.MaxPool2d(kernel_size=(2, 2), stride=2, padding=0)
        self.conv5 = torch.nn.Conv2d(in_channels=16, out_channels=16, kernel_size=(3, 3), stride=1, padding='valid')
        self.relu5 = torch.nn.ReLU()
        self.conv6 = torch.nn.Conv2d(in_channels=16, out_channels=8, kernel_size=(3, 3), stride=1, padding='valid')
        self.relu6 = torch.nn.ReLU()
        self.pool2 = torch.nn.MaxPool2d(kernel_size=(2, 2), stride=2, padding=0)

        self.fc1 = torch.nn.Linear(in_features=8 * 4 * 4, out_features=num_classes)

    def forward(self, x):
        hidden1 = self.conv1(x)
        relu1 = self.relu1(hidden1)
        hidden2 = self.conv2(relu1)
        relu2 = self.relu2(hidden2)
        hidden3 = self.conv3(relu2)
        relu3 = self.relu3(hidden3)
        hidden4 = self.conv4(relu3)
        relu4 = self.relu4(hidden4)
        pool1 = self.pool1(relu4)
        hidden5 = self.conv5(pool1)
        relu5 = self.relu5(hidden5)
        hidden6 = self.conv6(relu5)
        relu6 = self.relu6(hidden6)
        pool2 = self.pool2(relu6)
        # Reshaping linear input
        pool2 = pool2.view(BATCH_SIZE, 8 * 4 * 4)
        output = self.fc1(pool2)
        return output


class cnn_5_sigmoid(torch.nn.Module):


    def __init__(self, input_size, num_classes):
        super(cnn_5_sigmoid, self).__init__()
        self.input_size = input_size
        self.conv1 = torch.nn.Conv2d(in_channels=1, out_channels=8, kernel_size=(3, 3), stride=1, padding='valid')
        self.sigmoid1 = torch.nn.Sigmoid()
        self.conv2 = torch.nn.Conv2d(in_channels=8, out_channels=16, kernel_size=(3, 3), stride=1, padding='valid')
        self.sigmoid2 = torch.nn.Sigmoid()
        self.conv3 = torch.nn.Conv2d(in_channels=16, out_channels=8, kernel_size=(3, 3), stride=1, padding='valid')
        self.sigmoid3 = torch.nn.Sigmoid()
        self.conv4 = torch.nn.Conv2d(in_channels=8, out_channels=16, kernel_size=(3, 3), stride=1, padding='valid')
```

```
        self.sigmoid4 = torch.nn.Sigmoid()
        self.pool1 = torch.nn.MaxPool2d(kernel_size=(2, 2), stride=2, padding=0)
        self.conv5 = torch.nn.Conv2d(in_channels=16, out_channels=16, kernel_size=(3, 3), stride=1, padding='valid')
        self.sigmoid5 = torch.nn.Sigmoid()
        self.conv6 = torch.nn.Conv2d(in_channels=16, out_channels=8, kernel_size=(3, 3), stride=1, padding='valid')
        self.sigmoid6 = torch.nn.Sigmoid()
        self.pool2 = torch.nn.MaxPool2d(kernel_size=(2, 2), stride=2, padding=0)

        self.fc1 = torch.nn.Linear(in_features=8 * 4 * 4, out_features=num_classes)

    def forward(self, x):
        hidden1 = self.conv1(x)
        sigmoid1 = self.sigmoid1(hidden1)
        hidden2 = self.conv2(sigmoid1)
        sigmoid2 = self.sigmoid2(hidden2)
        hidden3 = self.conv3(sigmoid2)
        sigmoid3 = self.sigmoid3(hidden3)
        hidden4 = self.conv4(sigmoid3)
        sigmoid4 = self.sigmoid4(hidden4)
        pool1 = self.pool1(sigmoid4)
        hidden5 = self.conv5(pool1)
        sigmoid5 = self.sigmoid5(hidden5)
        hidden6 = self.conv6(sigmoid5)
        sigmoid6 = self.sigmoid6(hidden6)
        pool2 = self.pool2(sigmoid6)
        # Reshaping linear input
        pool2 = pool2.view(BATCH_SIZE, 8 * 4 * 4)
        output = self.fc1(pool2)
        return output
```

Q3)Train,Validation,test

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
from sklearn.model_selection import train_test_split
import q3
import utils
import json
#Let cuda works
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

torch.cuda.empty_cache()
print(f"Device is : {device}")
# Define transforms for the dataset
transform = transforms.Compose([
    transforms.ToTensor(),  # convert PIL image to tensor
    # torchvision.transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),  # normalize to [-1, 1]
    transforms.Grayscale()
])

# Load the CIFAR10 dataset
train_data = torchvision.datasets.CIFAR10("./data", train=True, download=True, transform=transform)
train_set, val_set = train_test_split(train_data, test_size=0.1)

test_data = torchvision.datasets.CIFAR10("./data", train=False, download=True, transform=transform)
test_set = torch.utils.data.DataLoader(test_data, batch_size=50, shuffle=False)

# Create the data loaders
val_loader = torch.utils.data.DataLoader(val_set, batch_size=q3.BATCH_SIZE, shuffle=False)

#Once my code has worked, It finishes all models.
models = ['mlp_1', 'mlp_2', 'cnn_3', 'cnn_4', 'cnn_5']
for modelselected in models:

    print(f"Training is started for model {modelselected}")
    # Init Required Arrays
    training_loss_total = []
    training_accuracy_total = []
    validation_accuracy_total = []
    validation_loss_total = []
    maxPerformanceTask = 0
    for stepX in range(q3.TRAIN_SIZE):
        print(f"Step {stepX + 1} is started")
        #Init models
        if modelselected == 'mlp_1':
            model = q3.mlp_1(1024, 32, 10).to(device)
        elif modelselected == 'mlp_2':
            model = q3.mlp_2(1024, 32, 64, 10).to(device)
        elif modelselected == 'cnn_3':
            model = q3.cnn_3(1024, 10).to(device)
        elif modelselected == 'cnn_4':
            model = q3.cnn_4(1024, 10).to(device)
        elif modelselected == 'cnn_5':
            model = q3.cnn_5(1024, 10).to(device)
        else:
            print("Model name is not true.")
        #Init arrays for each model
```

```
        training_loss = []
        training_accuracy = []
        validation_accuracy = []
        validation_loss = []
        test_accuracy = []
        #Init optimizer
        optimizer = torch.optim.Adam(model.parameters())
        # Train the model
        for epoch in range(q3.EPOCH_SIZE):
            print(f"Step {stepX + 1} is started")
            model.train()
            train_total = 0
            train_correct = 0
            train_loss = 0
            #DataLoader
            train_loader = torch.utils.data.DataLoader(train_set, batch_size=q3.BATCH_SIZE, shuffle=True)

            for i, (images, labels) in enumerate(train_loader):
                images = images.to(device)
                labels = labels.to(device)

                output = model(images)
                loss = nn.CrossEntropyLoss()(output, labels.to(device))

                optimizer.zero_grad()
                loss.backward()
                optimizer.step()
                if (i + 1) % 10 == 0:
                    model.eval()
                    _, predicted = torch.max(output.data, 1)
                    train_total = labels.size(0)#take label size
                    train_correct = (predicted == labels).sum().item()#Take number of corrections
                    train_loss = loss.item()#Take number of loss
                    training_accuracy.append((train_correct / train_total) * 100)  # Save Training accuracy
each 10 step
                    training_loss.append(train_loss)  # Save Training loss each 10 step
                    val_correct = 0
                    val_size = 0
                    #Validation
                    for j, (inputs_val, labels_val) in enumerate(val_loader):
                        inputs_val, labels_val = inputs_val.to(device), labels_val.to(device)
                        val_outputs = model(inputs_val)
                        loss = nn.CrossEntropyLoss()(val_outputs, labels_val)
                        # val_loss += loss.item()
                        _, val_pred = val_outputs.max(1)
                        val_size += labels_val.size(0)
                        val_correct += val_pred.eq(labels_val).sum().item()
                    # validation_loss.append((val_loss / val_size) * 100)
                    validation_accuracy.append((val_correct / val_size) * 100)
            #After each model, add each data to required arrays
            validation_accuracy_total.append(validation_accuracy)
            training_loss_total.append(training_loss)
            training_accuracy_total.append(training_accuracy)
            # Arrays ends with '_total' has 5 members which are for each model
            # Evaluate on test set

        with torch.no_grad():
            model.eval()
            correct = 0
            total = 0
            for images, labels in test_set:
                #Doing same things for testing
                images = images.to(device)
                labels = labels.to(device)
                outputs = model(images)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
            test_acc = (correct / total) * 100
            test_accuracy.append(test_acc)
            #Choose maxPerformanceTask
            if test_acc > maxPerformanceTask:
                maxPerformanceTask = test_acc
                #to get weight_best switch model to 'CPU'
                model.to('cpu')
                #The reason of there are two condition is name difference of name of first layers.
                if modelselected == "mlp_1" or modelselected == "mlp_2":
                    weight_best = model.fc1.weight.data.numpy()
                elif modelselected == "cnn_3" or modelselected == "cnn_4" or modelselected == "cnn_5":
                    weight_best = model.conv1.weight.data.numpy()
                #After get weight_best, switch again
                model.to(device)
        print('Test Accuracy of the model on the test images: {} %'.format((correct / total) * 100))
    #After Train_Size is finished for each model, take average and save to Json file
    average_train_losses = [sum(sub_list) / len(sub_list) for sub_list in zip(*training_loss_total)]
    average_train_accu = [sum(sub_list) / len(sub_list) for sub_list in zip(*training_accuracy_total)]
    average_valid_accu = [sum(sub_list) / len(sub_list) for sub_list in zip(*validation_accuracy_total)]
```

```
    print(f"MaxPerformance: {maxPerformanceTask}")
    # Dictionary for json
    dictionary = {
        'name': modelselected,
        'loss_curve': average_train_losses,
        'train_acc_curve': average_train_accu,
        'val_acc_curve': average_valid_accu,
        'test_acc': maxPerformanceTask,
        'weights': weight_best.tolist(),
    }
    with open("ResultQ3/Json_files/Q3_" + modelselected + ".json", "w") as outfile:
        json.dump(dictionary, outfile)

    utils.visualizeWeights(weight_best, save_dir='ResultQ3/Weights', filename='input_weights_' +
modelselected)
```

Q3)plot

```
from utils import part3Plots

import json

models = ['mlp_1', 'mlp_2', 'cnn_3', 'cnn_4', 'cnn_5']  # Models that will be printed
results = list()
for model in models:
    f = open('ResultQ3/Json_files/Q3_' + model + '.json')

    #assemble of all data in results
    results.append(json.load(f))
    f.close()
#print results
part3Plots(results, save_dir=r'ResultQ3', filename='part3Plots')
```

Q4)Train

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
from sklearn.model_selection import train_test_split
import q3
import utils
import json
import numpy as np
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

torch.cuda.empty_cache()
print(f"Device is : {device}")
# Define transforms for the dataset
transform = transforms.Compose([
    transforms.ToTensor(),  # convert PIL image to tensor
    # torchvision.transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),  # normalize to [-1, 1]
    transforms.Grayscale()
])

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


# Create the data loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=q3.BATCH_SIZE, shuffle=True)

maxPerformanceTask = 0

#this array has size [5][2]. 5 represents types of models and 2 represents types of activation functions
models = (
    ("mlp_1", "mlp_1_sigmoid"), ("mlp_2", "mlp_2_sigmoid"), ("cnn_3", "cnn_3_sigmoid"), ("cnn_4", "cnn_4_sigmoid"),
    ("cnn_5", "cnn_5_sigmoid"))
#firstly we choose represents types of models
for BothModel in models:
    print(BothModel)
    # Init arrays for each model
    relu_loss_curve = []
    relu_grad_curve = []
    sigmoid_loss_curve = []
    sigmoid_grad_curve = []
    #then we choose types of activation functions
    for modelselected in BothModel:
        print(f"Training is started for model {modelselected}")
        for stepX in range(1):
            print(f"Step {stepX + 1} is started")
            if modelselected == 'mlp_1':
                model = q3.mlp_1(1024, 32, 10).to(device)
            elif modelselected == 'mlp_2':
                model = q3.mlp_2(1024, 32, 64, 10).to(device)
            elif modelselected == 'cnn_3':
                model = q3.cnn_3(1024, 10).to(device)
            elif modelselected == 'cnn_4':
                model = q3.cnn_4(1024, 10).to(device)
            elif modelselected == 'cnn_5':
                model = q3.cnn_5(1024, 10).to(device)
            elif modelselected == 'mlp_1_sigmoid':
                model = q3.mlp_1_sigmoid(1024, 32, 10).to(device)
            elif modelselected == 'mlp_2_sigmoid':
                model = q3.mlp_2_sigmoid(1024, 32, 64, 10).to(device)
```

```
            elif modelselected == 'cnn_3_sigmoid':
                model = q3.cnn_3_sigmoid(1024, 10).to(device)
            elif modelselected == 'cnn_4_sigmoid':
                model = q3.cnn_4_sigmoid(1024, 10).to(device)
            elif modelselected == 'cnn_5_sigmoid':
                model = q3.cnn_5_sigmoid(1024, 10).to(device)
            else:
                print("Model name is not true.")
            # Init optimizer
            optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.0)
            # Train the model
            for epoch in range(q3.EPOCH_SIZE):
                print(f"Epoch {epoch + 1}/15")
                model.train()
                train_total = 0
                train_correct = 0
                train_loader = torch.utils.data.DataLoader(train_data, batch_size=q3.BATCH_SIZE, shuffle=True)
                for i, (images, labels) in enumerate(train_loader):
                    images = images.to(device)
                    labels = labels.to(device)

                    model.to('cpu')
                    weightBefore = model.fc1.weight.data.numpy().flatten()
                    model.to(device)

                    # Forward pass
                    outputs = model(images)
                    loss = nn.CrossEntropyLoss()(outputs, labels.to(device))

                    # Backward and optimize
                    optimizer.zero_grad()
                    loss.backward()
                    optimizer.step()
                    running_loss = loss.item()

                    if (i + 1) % 10 == 0:
                        model.to('cpu')
                        weightAfter = model.fc1.weight.data.numpy().flatten()
                        model.to(device)
                        running_grad = float(np.linalg.norm(weightAfter - weightBefore) / 0.01)

                        if modelselected[-1] == 'd':  # That means it is sigmoid EX:mlp_1_sigmoid
                            sigmoid_loss_curve.append(running_loss)
                            sigmoid_grad_curve.append(running_grad)
                        else:  # That means it is RELU
                            relu_loss_curve.append(running_loss)
                            relu_grad_curve.append(running_grad)

                # Evaluate the model on the validation set
                # Evaluate on test set
    # Dictionary for json
    if modelselected[6] == 's':#In models array sigmoid models are always come after Relu so that when models with Sigmoid
funcitons, we can save data and run other model
        dictionary = {
            'name': modelselected[:5],  # It means mlp_2,cnn_5 since last model will be sigmoid but we want only its model
name.
            'relu_loss_curve': relu_loss_curve,
            'sigmoid_loss_curve': sigmoid_loss_curve,
            'relu_grad_curve': relu_grad_curve,
            'sigmoid_grad_curve': sigmoid_grad_curve,

        }
        # print(train_losses_total)
        with open("resultQ4/Q4_" + modelselected[:5] + ".json", "w") as outfile:
            json.dump(dictionary, outfile)
        print(f"Both Models which are {BothModel[0]} and {BothModel[1]} are finished.")
```

Q4 plot

```
from utils import part4Plots

import json

models_all = ["mlp_1", "mlp_2", "cnn_3", "cnn_4", "cnn_5"]
models_mlp = ['mlp_1', 'mlp_2']  # Models that will be plotted
models_cnn = ['cnn_3', 'cnn_4', 'cnn_5']
results = list()
#plot all models in single graph
for model in models_all:
    f = open("ResultQ4/Q4_" + model + ".json" )

    #assemble of all data in results
    results.append(json.load(f))
    f.close()
#print results
part4Plots(results, save_dir=r"ResultQ4", filename=f"part4Result_all")
results = list()
#plot all mlp models in single graph
for model in models_mlp:
    f = open("ResultQ4/Q4_" + model + ".json" )
    results.append(json.load(f))
    f.close()
part4Plots(results, save_dir=r"ResultQ4", filename=f"part4Result_mlp")
results = list()
#plot all cnn models in single graph
for model in models_cnn:
    f = open("ResultQ4/Q4_" + model + ".json" )
```

```
    results.append(json.load(f))
    f.close()
part4Plots(results, save_dir=r"ResultQ4", filename=f"part4Result_cnn")
```

Q5-1)

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
from sklearn.model_selection import train_test_split
import q3
import utils
import json

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

torch.cuda.empty_cache()
print(f"Device is : {device}")
# Define transforms for the dataset
transform = transforms.Compose([
    transforms.ToTensor(),  # convert PIL image to tensor
    # torchvision.transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),  # normalize to [-1, 1]
    transforms.Grayscale()
])

# Load the CIFAR10 dataset
train_data = torchvision.datasets.CIFAR10("./data", train=True, download=True, transform=transform)
train_set, val_set = train_test_split(train_data, test_size=0.1)

# Create the data loaders
val_loader = torch.utils.data.DataLoader(val_set, batch_size=q3.BATCH_SIZE, shuffle=True)

training_loss_total = []
validation_accuracy_total = []

models = 'cnn_4' # I choose cnn4
lrs = [0.1, 0.01, 0.001]  # Learning Rates
for LR in lrs:
    print(f"Training is started for model {models} and LR = {LR}")
    model = q3.cnn_4(1024, 10).to(device)
    # Init arrays
    training_loss = []
    validation_accuracy = []
    optimizer = torch.optim.SGD(model.parameters(), lr=LR, momentum=0.00)  # Setting LR
    # Train the model
    for epoch in range(q3.EPOCH_SIZE + 5):  # to make Epoch_size=20
        print(f"Epoch {epoch + 1}/20")
        model.train()
        train_total = 0
        train_correct = 0
        train_loader = torch.utils.data.DataLoader(train_set, batch_size=q3.BATCH_SIZE, shuffle=True)

        for i, (images, labels) in enumerate(train_loader):
            images = images.to(device)
            labels = labels.to(device)

            output = model(images)
            loss = nn.CrossEntropyLoss()(output, labels)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            if (i + 1) % 10 == 0:
                model.eval()
                running_loss = loss.item()
                val_generator = torch.utils.data.DataLoader(val_set, batch_size=q3.BATCH_SIZE, shuffle=False)
                val_total = 0
                val_correct = 0
                for j, (inputs_val, labels_val) in enumerate(val_loader):
                    inputs_val, labels_val = inputs_val.to(device), labels_val.to(device)
                    val_outputs = model(inputs_val)
                    loss = nn.CrossEntropyLoss()(val_outputs, labels_val)
                    _, val_pred = val_outputs.max(1)
                    val_total += labels_val.size(0)
                    val_correct += val_pred.eq(labels_val).sum().item()
                training_loss.append(running_loss)
                validation_accuracy.append((val_correct / val_total) * 100)
        # Evaluate the model on the validation set
        # Evaluate on test set
        print(f"Training Loss: {sum(training_loss)/len(training_loss)} and Validation Accuracy:
{sum(validation_accuracy)/len(validation_accuracy)} on Epoch : {epoch}")
    training_loss_total.append(training_loss)
    validation_accuracy_total.append(validation_accuracy)
dictonary = {
    'name': 'cnn_4',
    'loss_curve_1': training_loss_total[0],
    'loss_curve_01': training_loss_total[1],
    'loss_curve_001': training_loss_total[2],
    'val_acc_curve_1': validation_accuracy_total[0],
    'val_acc_curve_01': validation_accuracy_total[1],
    'val_acc_curve_001': validation_accuracy_total[2],
}
# Recording Results
with open("Q5_CNN_4.json", "w") as outfile:
    json.dump(dictonary, outfile)
```

Q5-2)

```python
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
from sklearn.model_selection import train_test_split
import q3
import utils
import json

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

torch.cuda.empty_cache()
print(f"Device is : {device}")
# Define transforms for the dataset
transform = transforms.Compose([
    transforms.ToTensor(),  # convert PIL image to tensor
    # torchvision.transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),  # normalize to [-1, 1]
    transforms.Grayscale()
])

# Load the CIFAR10 dataset
train_data = torchvision.datasets.CIFAR10("./data", train=True, download=True, transform=transform)
train_set, val_set = train_test_split(train_data, test_size=0.1)

# Create the data loaders
val_loader = torch.utils.data.DataLoader(val_set, batch_size=q3.BATCH_SIZE, shuffle=True)

# Create the model and optimizer

models = 'cnn_4'
lrs = [0.1, 0.01, 0.001]  # Learning Rates
model = q3.cnn_4(1024, 10).to(device)


validation_accuracy = []
optimizer = torch.optim.SGD(model.parameters(), lr=lrs[0], momentum=0.00)  # Setting LR
# Train the model
for epoch in range(q3.EPOCH_SIZE + 15):  # to make Epoch_size=30
    if(epoch == 11):#1000/90 = 11
        optimizer = torch.optim.SGD(model.parameters(), lr=lrs[1], momentum=0.00)  # Setting LR
        print("LR is changed to 0.01 from 0.1")
    print(f"Epoch {epoch + 1}/30")
    train_total = 0
    train_correct = 0
    train_loader = torch.utils.data.DataLoader(train_set, batch_size=q3.BATCH_SIZE, shuffle=True)

    for i, (images, labels) in enumerate(train_loader):
        model.train()
        images = images.to(device)
        labels = labels.to(device)

        output = model(images)
        loss = nn.CrossEntropyLoss()(output, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i + 1) % 10 == 0:
            model.eval()
            running_loss = loss.item()
            val_generator = torch.utils.data.DataLoader(val_set, batch_size=q3.BATCH_SIZE, shuffle=False)
            val_total = 0
            val_correct = 0
            for j, (inputs_val, labels_val) in enumerate(val_loader):
                inputs_val, labels_val = inputs_val.to(device), labels_val.to(device)
                val_outputs = model(inputs_val)
                loss = nn.CrossEntropyLoss()(val_outputs, labels_val)
                _, val_pred = val_outputs.max(1)
                val_total += labels_val.size(0)
                val_correct += val_pred.eq(labels_val).sum().item()

            validation_accuracy.append((val_correct / val_total) * 100)
    # Evaluate the model on the validation set
    # Evaluate on test set
    print(f"Validation Accuracy: {sum(validation_accuracy) / len(validation_accuracy)} on Epoch : {epoch}")
dictonary = {
    'name': 'cnn_4',
    'loss_curve_1': validation_accuracy,
    'loss_curve_01': validation_accuracy,
    'loss_curve_001': validation_accuracy,
    'val_acc_curve_1': validation_accuracy,
    'val_acc_curve_01': validation_accuracy,
    'val_acc_curve_001': validation_accuracy,
}
# Recording Results
with open("ResultQ5/Q5_cnn4_second.json", "w") as outfile:
    json.dump(dictonary, outfile)
```

Q5-3)

```python
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
from sklearn.model_selection import train_test_split
import q3
import utils
import json

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

torch.cuda.empty_cache()
print(f"Device is : {device}")
# Define transforms for the dataset
transform = transforms.Compose([
    transforms.ToTensor(),  # convert PIL image to tensor
    # torchvision.transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261)),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),  # normalize to [-1, 1]
    transforms.Grayscale()
])

# Load the CIFAR10 dataset
train_data = torchvision.datasets.CIFAR10("./data", train=True, download=True, transform=transform)
train_set, val_set = train_test_split(train_data, test_size=0.1)

# Create the data loaders
val_loader = torch.utils.data.DataLoader(val_set, batch_size=q3.BATCH_SIZE, shuffle=True)

models = 'cnn_4'
lrs = [0.1, 0.01, 0.001]  # Learning Rates
model = q3.cnn_4(1024, 10).to(device)


validation_accuracy = []
optimizer = torch.optim.SGD(model.parameters(), lr=lrs[0], momentum=0.00)  # Setting LR
# Train the model
for epoch in range(q3.EPOCH_SIZE + 15):  # to make Epoch_size=30
    if(epoch == 11):#1000/90 = 11
        optimizer = torch.optim.SGD(model.parameters(), lr=lrs[1], momentum=0.00)  # Setting LR
        print("LR is changed to 0.01 from 0.1")
    elif (epoch == 22):  # 1000/90 = 11
        optimizer = torch.optim.SGD(model.parameters(), lr=lrs[2], momentum=0.00)  # Setting LR
        print("LR is changed to 0.01 from 0.1")
    print(f"Epoch {epoch + 1}/30")
    train_total = 0
    train_correct = 0
    train_loader = torch.utils.data.DataLoader(train_set, batch_size=q3.BATCH_SIZE, shuffle=True)

    for i, (images, labels) in enumerate(train_loader):
        model.train()
        images = images.to(device)
        labels = labels.to(device)

        output = model(images)
        loss = nn.CrossEntropyLoss()(output, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i + 1) % 10 == 0:
            model.eval()
            running_loss = loss.item()
            val_generator = torch.utils.data.DataLoader(val_set, batch_size=q3.BATCH_SIZE, shuffle=False)
            val_total = 0
            val_correct = 0
            for j, (inputs_val, labels_val) in enumerate(val_loader):
                inputs_val, labels_val = inputs_val.to(device), labels_val.to(device)
                val_outputs = model(inputs_val)
                loss = nn.CrossEntropyLoss()(val_outputs, labels_val)
                _, val_pred = val_outputs.max(1)
                val_total += labels_val.size(0)
                val_correct += val_pred.eq(labels_val).sum().item()

            validation_accuracy.append((val_correct / val_total) * 100)
    # Evaluate the model on the validation set
    # Evaluate on test set
    print(f"Validation Accuracy: {sum(validation_accuracy) / len(validation_accuracy)} on Epoch : {epoch}")
dictonary = {
    'name': 'cnn_4',
    'loss_curve_1': validation_accuracy,
    'loss_curve_01': validation_accuracy,
    'loss_curve_001': validation_accuracy,
    'val_acc_curve_1': validation_accuracy,
    'val_acc_curve_01': validation_accuracy,
    'val_acc_curve_001': validation_accuracy,
}
# Recording Results
with open("ResultQ5/Q5_cnn4_third.json", "w") as outfile:
    json.dump(dictonary, outfile)
```

Q5-plot)

```
from utils import part5Plots
import json
f = open('ResultQ5/Q5_cnn4_third.json')
data1 = json.load(f)
f.close()

part5Plots(data1, save_dir="ResultQ5", filename="Q5_cnn4_third.png")
```