Ahmet Akman 2442366
Middle East Technical University
Electrical and Electronics Engineering

April 7, 2024

HOMEWORK 1 — Report

1 Question 1

Question 1.1 - Preliminaries

The partial derivative calculation steps for Tanh, Sigmoid, and ReLU activation functions are shown in Figure 1.

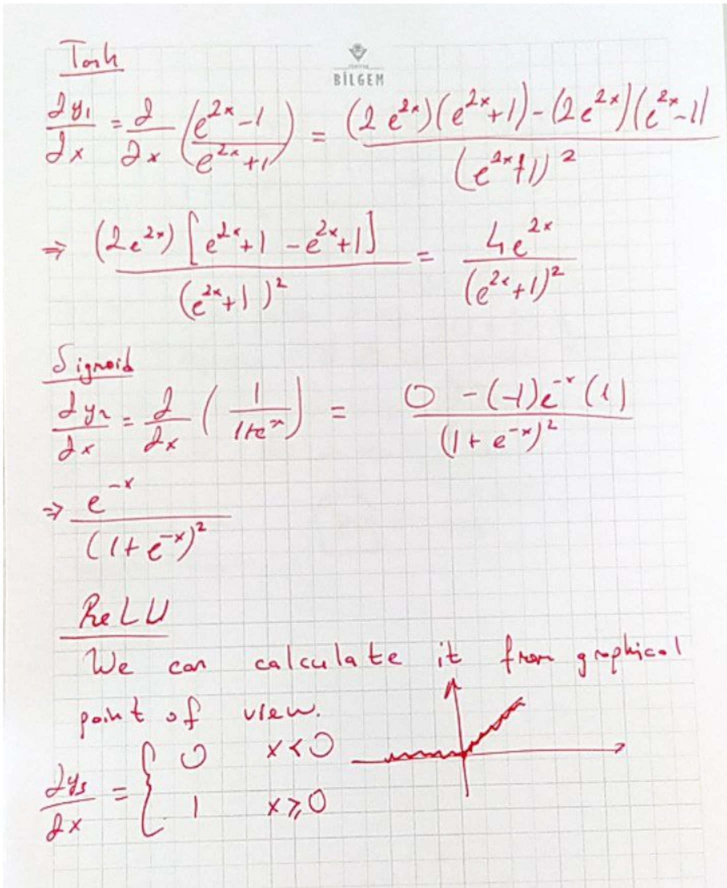


Figure 1: Partial derivative calculation steps for Tanh, Sigmoid and ReLU activation functions.

Figure 2 illustrates the activation functions' response between -2 and 2. The plots are obtained using the matplotlib library as instructed.

Figure 3 indicates the gradients of those functions in the same range. The gradients are calculated using the partial derivatives derived in Figure 1.

Question 1.2

In this part, MLP with one hidden layer is implemented utilizing the given code template. The input-output pairs are fetched from the XOR data provided in utils.py. Note that for all networks, the learning

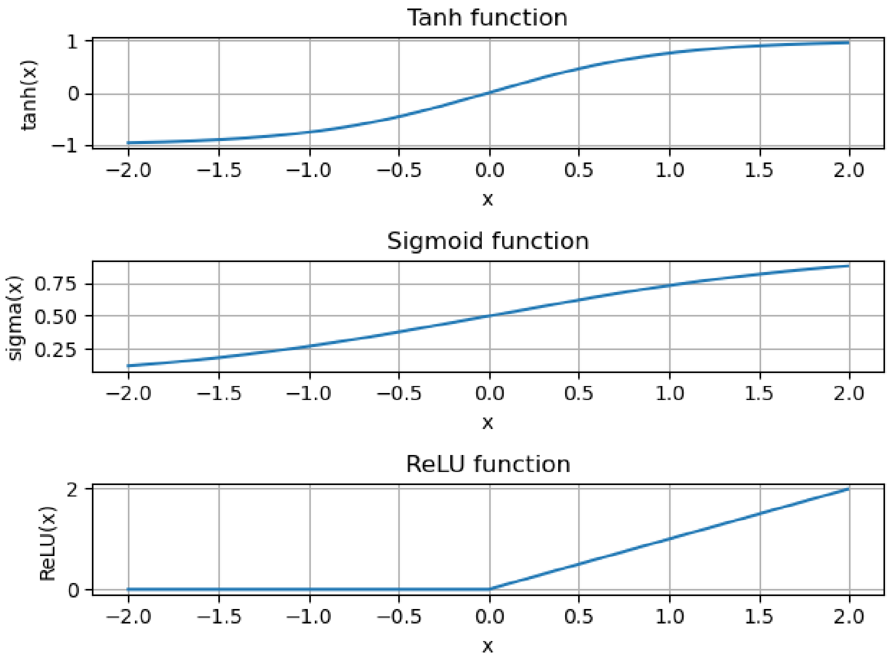


Figure 2: Activation functions plot.

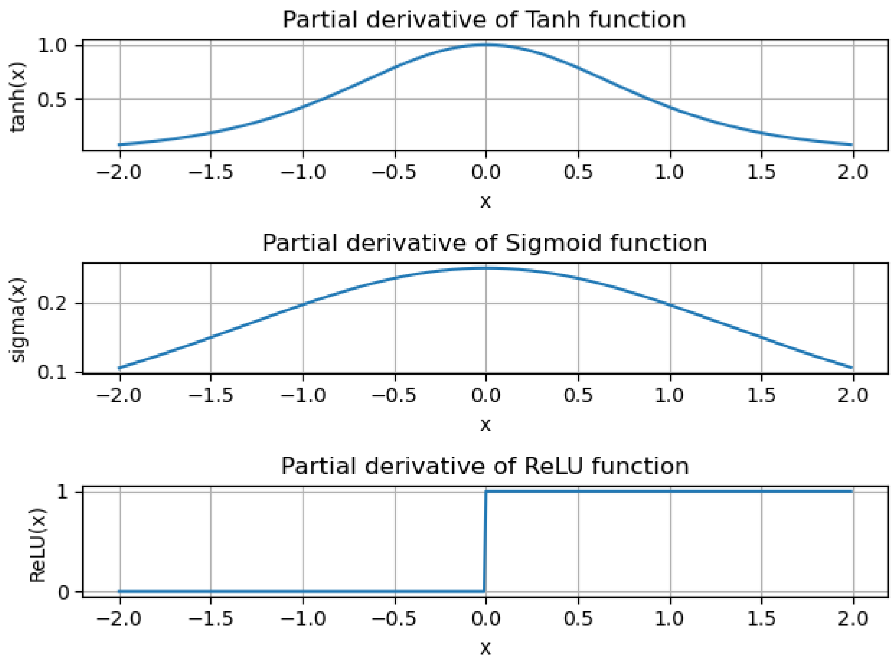


Figure 3: Gradients of the activation functions plot.

rate is fixed at 0.00001, and seed is utilized. Figure 4 shows the decision boundary for the sigmoid-activated network.

Similarly, Figure 5 is the decision boundary for the tanh-activated network, and Figure 6 is the decision boundary for the ReLU-activated network.

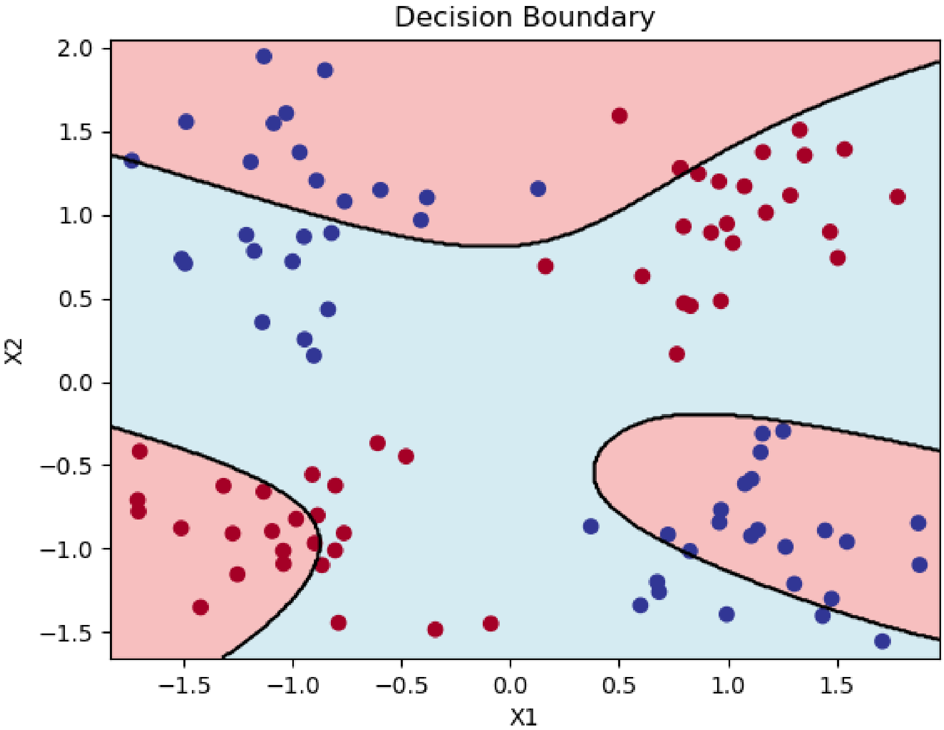


Figure 4: Sigmoid activated XOR problem output.

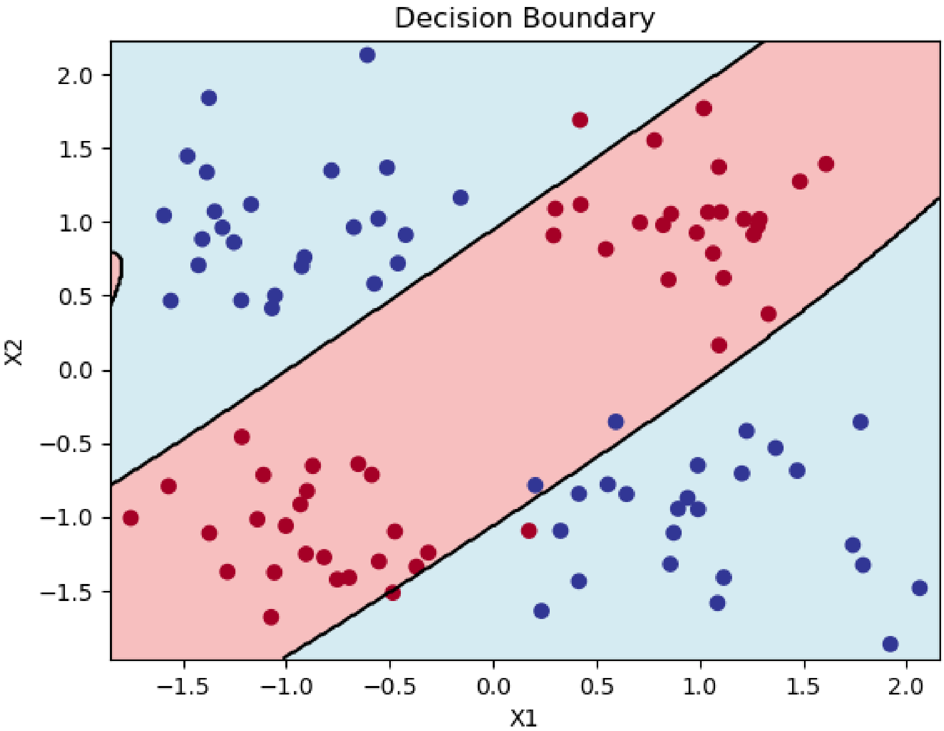


Figure 5: Tanh activated XOR problem output.

Question 1.3 - Discussions

1. All of those activation functions provide a smooth transition from one state to another. The advantage of Tanh and Sigmoid is they are limited in the range of -1 to 1 and 0 to 1, respectively. This property

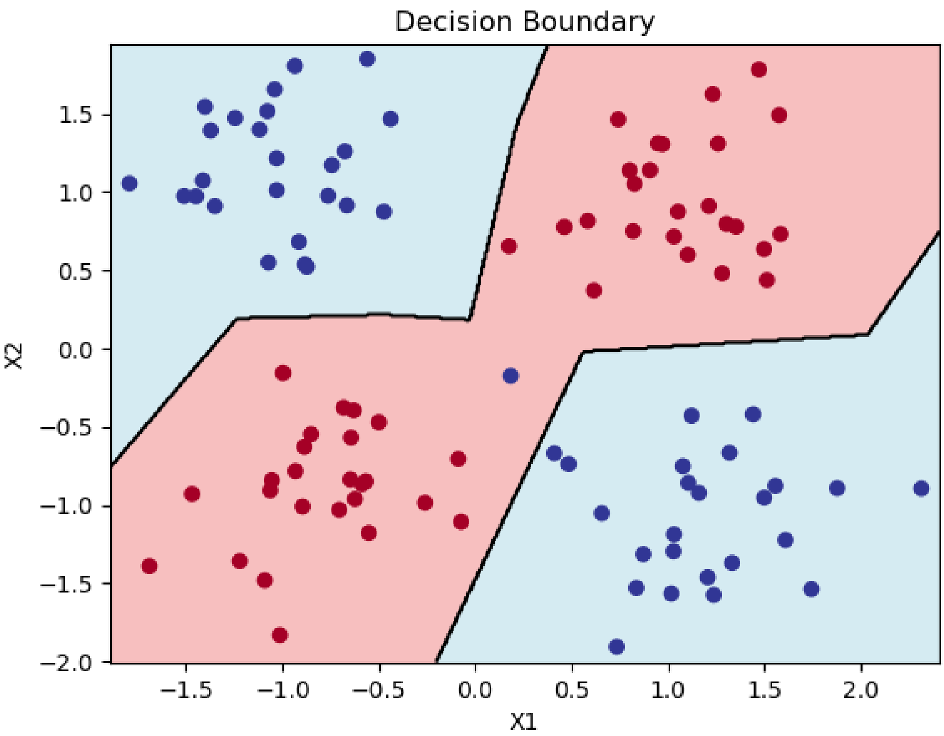


Figure 6: ReLU activated XOR problem output.

can be beneficial in some cases. However, the ReLU function is not limited in the range. It is also computationally cheaper than the other two. The disadvantage of ReLU is that it is not smooth at the origin. This can cause some problems in the optimization process. Another advantage of ReLU is negative gradients are zero. This may be helpful in some cases.

2. XOR problem is an input decision problem where inputs have to be different than each other to obtain 1. Since the output is always with respect to the state of two variables, the decision boundary is not linear. Therefore, a single-layer perceptron can not be solved since, in one step, two case evaluations can not be done. Yet, by adding a hidden layer, the problem is solved. The activation functions are used to introduce non-linearity to the network. The decision boundary of the XOR problem is shown in Figures 4, 5 and 6. As can be seen from the figures, the MLPs with activation functions can solve the XOR problem to some extent.

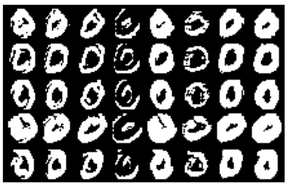
3. The boundaries change in each run since the initial weights and the data points are randomly generated. Therefore, the decision boundaries are dependent on the training process, where initial randomness leads to different outputs.

2 Question 2

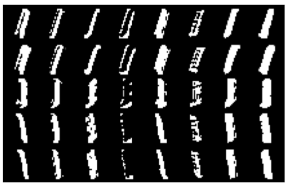
A convolution operator is implemented using a set of nested loops. The outputs are checked on a separate code so that they are identical to the output of torch.conv2d. Using the provided inputs, outputs provided in Figure 7 are obtained.

Question 2.2 - Discussions

1. Two-dimensional convolution operation provides filtering in two dimensions via a kernel. The kernel is applied to the input image in a sliding window fashion. The output is obtained by element-wise

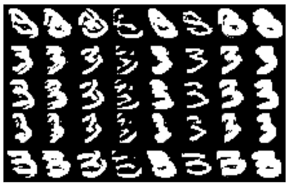


(a) Output 0



(b) Output 1

(c) Output 2



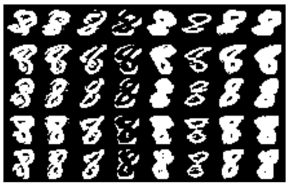
(d) Output 3

(e) Output 4

(f) Output 5

(g) Output 6

(h) Output 7



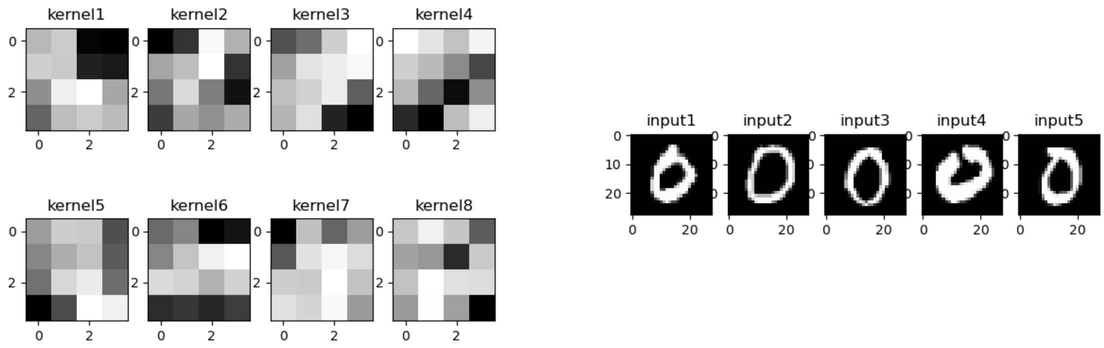
(i) Output 8

Figure 7: Convolution output

multiplication of the kernel and the input image. The kernel is then shifted by a stride, and the process is repeated. That is, two-dimensional convolution operations with learnable kernel entries are commonly used in image processing to extract features from the image. The kernel is learned during the training process. Those feature maps encode necessary information about distinct clues in the image. For example, in the case of object recognition, a specific kernel can be trained to detect bicycle rims. The kernel of a convolution layer corresponds to the filter function in one dimension. It is used to suppress or enhance certain information in the input image.

2. The sizes of the kernel correspond to the size of the filter. Therefore, the size list can be explained as follows (batch size, input channels, output channels, filter height, filter width). Batch size is the number of input sets in a batch if batching is used; we have not utilized it here. Input channels are the number of depth dimensions in the image; for example, in RGB frames, it is three. This might be different for different sensor inputs and representations. The output channel number determines the depth dimension of the output tensor. The filter height and width are the dimensions of the filter.

3. To understand what exactly happened here, let us plot the kernel and input for the number zero. Figure 8 illustrates the plot.



(a) Kernels

(b) Input set for number zero.

Figure 8: Kernel and input for number zero.

When we have a look at what the output image represents, on each row, an input image is convolved with a kernel. Different kernels are applied to the input image. The output image is the result of those convolutions. Figure 8 shows the input and kernel pairs in order.

4. Each convolution kernel embeds a certain feature of the image. That is, if we have a look at Figure 8.a, we can see that kernel 4 enhances the contours on the image, whereas kernel 6 enhances the filled part of those contours. This is the reason why we see similarly formatted outputs in the same column.

5. Similarly, since we "highlight" different properties on the image without filters, we see different patterns on the output side, even if the input is the same.

6. So, we can interpret from 4 and 5 that the output of the convolution layer is the result of the feature extraction process. The output is the result of the convolution of the input image with the learned filters. The output is the feature map that encodes the information about the input image. By post-processing those feature maps, we can obtain the necessary interpretations of the input image.

3 Question 3

Question 3.1

The implementations related to each architecture are completed as instructed. The code is given in the appendix. As a result, the plot shown in Figure 9 is obtained. Also, the first layer weights recorded are plotted and shown in Figure 10.

Question 3.2 - Discussions

1. The generalization performance of a classifier is the ability of the classifier to perform well on unseen data. That is, the classifier should be able to generalize the patterns in the training data to the test data. The overfitting phenomenon is the case where the model fits the training data too much and loses its generalization.

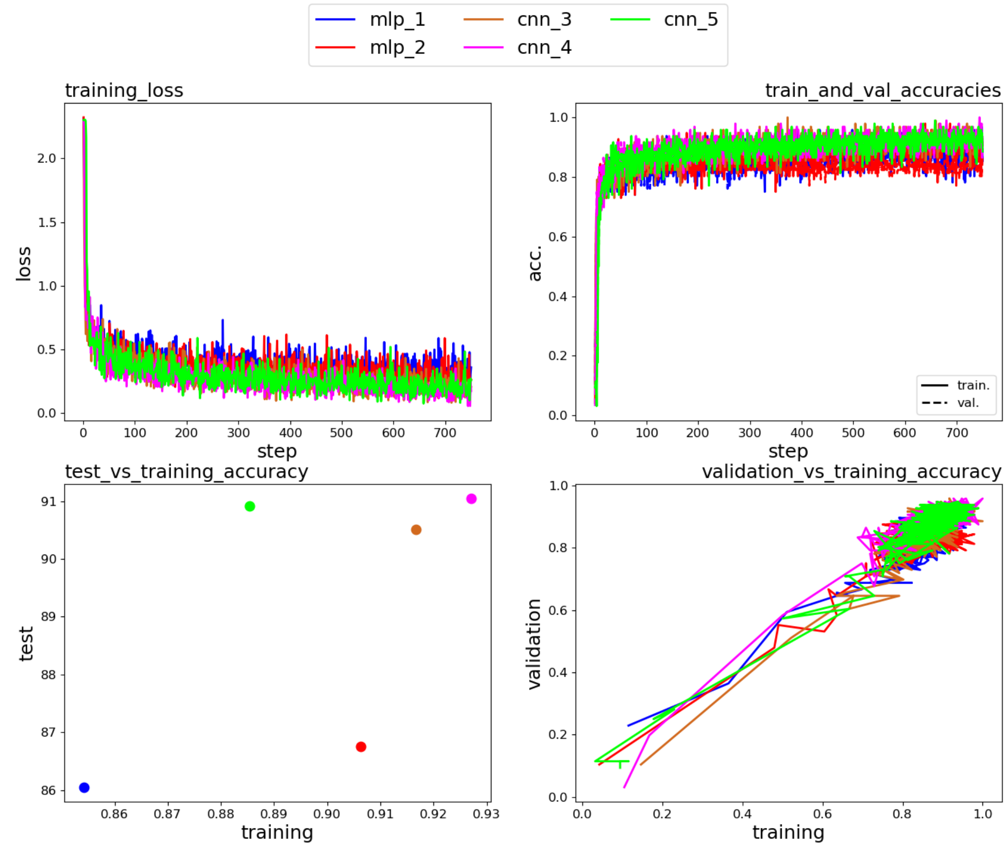


Figure 9: Benchmark of five different architectures.

2. The plots, test vs. training accuracy and validation vs. training accuracy are the most informative in this sense.

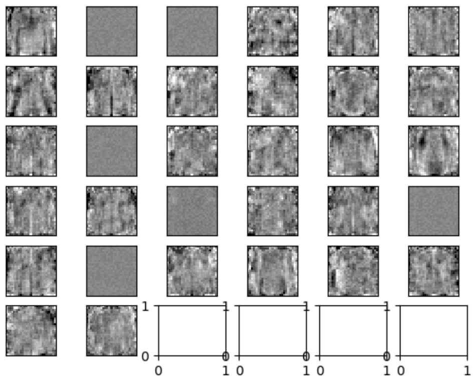
3. Somehow, the validation vs training accuracy plot is hard to read out. Therefore, deductions are made from test vs training accuracy plots. First of all, one can see right away that the plot that CNN-based models generalized better, whereas MLP-based ones have testing accuracy lower than training accuracy. On the other hand, amongst CNN-based ones, cnn4 has the best accuracy in terms of both training and test accuracy. However, it can be deduced that cnn5 has the best generalization performance since it has quite high test accuracy compared to its training accuracy. The test scores are even better than the training scores.

4.

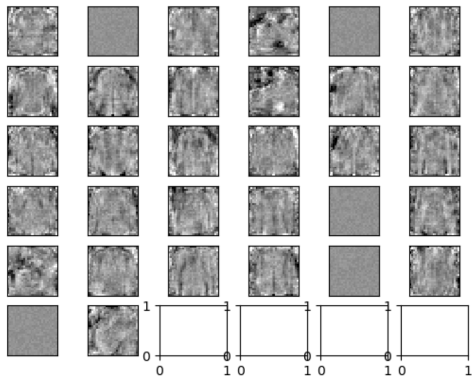
Architecture	**Trainable Params**
mlp 1	25,450
mlp 2	27,818
cnn 3	22,626
cnn 4	14,962
cnn 5	10,986

Table 1: Trainable Parameters of Different Architectures

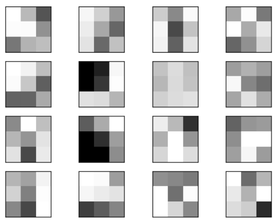
As the number of parameters increases, the number of points to fit the data increases. Therefore, the model can fit the data better in the most basic setting. The number of parameters for each architecture is given in Figure 1. So, classification performance increases as the number of parameters increases to a



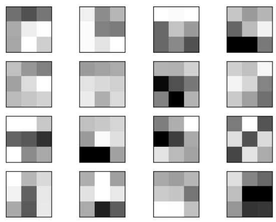
(a) mlp 1



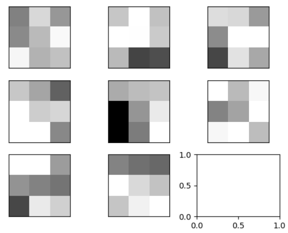
(b) mlp 2



(c) cnn 3



(d) cnn 4



(e) cnn 5

Figure 10: Weights of the first layers.

certain point, as we can observe from multi-layer perceptron training. From cnn3 to cnn4, both training and test accuracy increases, but a number of parameters are lower in cnn4. However, from cnn4 to cnn5, the test accuracy is almost the same, whereas the training accuracy decreases considerably. Therefore, it can be said that using only a number of parameters to describe generalization and training capabilities is not the best idea.

5. As the depth of the network increases, e.g., from mlp1 cnn5 network depth increases, in general, both classification performance increases up to a certain level and then decreases. For generalization, we can say that it always increases according to the data we have right now in those examples.

6. The mlp weights are not interpretable. However, one may interpret what kind of filtering is done by the CNN weights by looking at the plots shown in Figure 10. The first layers, in general, encode the most basic features like cırbers, edges, and so on.

7. The limited experience of the report writer does not allow them to extract the information on whether the filters are specific to the cases by looking at them. However, principally, certain CNN filters are trained to detect certain features in the input image. Again, the first layers encode the most simplistic features like cırbers and edges. Therefore, the filters are specific to the classes to distinguish them better.

8. Similarly, it is hard to distinguish between the filters by looking at the weights. The most interpretable ones are from the cnn4 architecture. The filters are more clear in this case.

9. The mlp1 and mlp2 are the simplest architectures that are similar to each other. The mlp2 introduces one more layer. The cnn3, cnn4, and cnn5 are the convolutional neural network architectures. The cnn3 has three layers with three different spatial kernels. The cnn5 has six convolutional layers with fixed spatial dimensions, but it is the deepest of all five of the architectures. As done in the previous sections, mlp2 performs better than mlp1 by increasing the number of parameters. Cnn4 performs better than cnn3 while having similar number parameters and having higher depth. Cnn5 has a significantly low number of parameters, so it underfits the training set but generalizes quite nicely.

10. I would pick cnn4 since it has the best test accuracy and generalization performance. It seems it can fit the data well and does not overfit in this setting.

4 Question 4

Question 4.1

As instructed in the homework documentation, the implementation for both ReLU and sigmoid activate models is done. Necessary statistics are recorded. As a result, the plot shown in Figure 11 is obtained. Note that the utils.py had to be modified (alpha value edited) to have a proper distinction. Also, Figure 12 illustrates the individual results for each architecture.

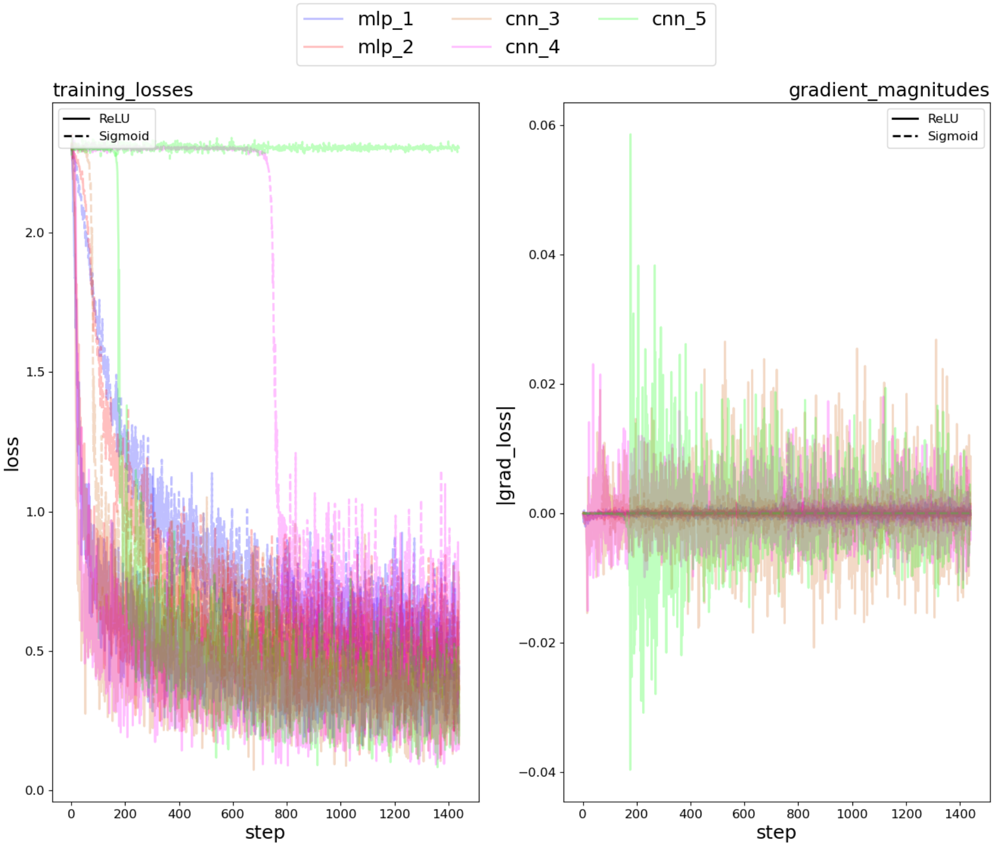
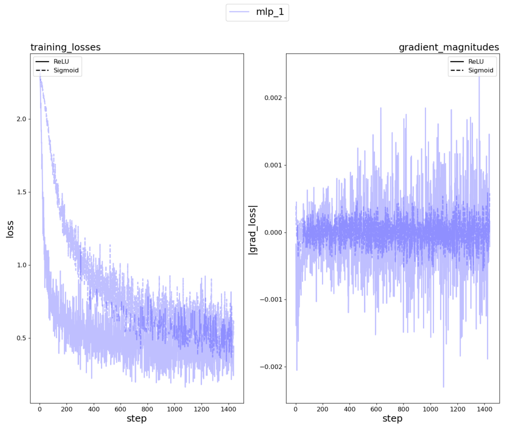


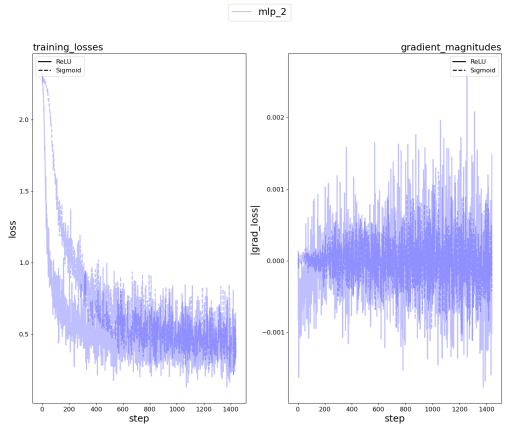
Figure 11: Benchmark of five different architectures trained using ReLU and Sigmoid activation functions.

Question 4.2 - Discussions

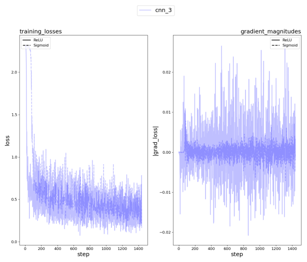
1. – 2. Gradient behavior is similar in each architecture if we focus on ReLU-activated ones. The values fluctuate around the 0 line as expected. However, the sigmoid-activated ones exhibit different behaviors. The gradients are quite small. Also, as the depth increases, the gradients for sigmoid-activated ones get smaller and quite close to 0. This is due to the vanishing gradient problem. The gradients are smaller and smaller as the depth increases in each layer pass. Furthermore, for the cnn5 case, it is almost always 0. It can be said that using the sigmoid activation function instead of ReLU is not the best idea for deep networks.



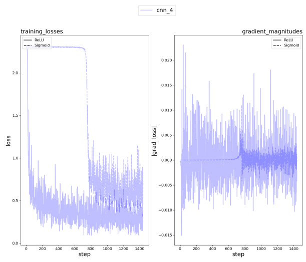
(a) mlp 1



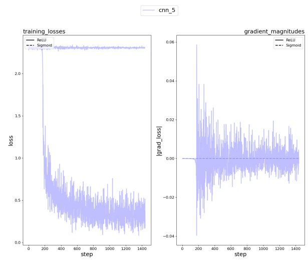
(b) mlp 2



(c) cnn 3



(d) cnn 4



(e) cnn 5

Figure 12: ReLU vs Sigmoid for each architecture.

3. In part 1.2. case, the network was only one layer, but the effect was similar, where the gradient values were smaller in the activated one. However, it is important to note that in order to obtain similar performance in both the ReLU and Sigmoid one, the learning rate needs to be adjusted where, overall, the weight update step yields similar scales. In this report, to have a fair comparison, the learning rate was fixed to a small value where ReLU did not blow up the gradients.

4. As pointed out in the previous paragraph, this time, Sigmoid could have performed better and avoided vanishing the gradients. On the other hand, depending on the rates, ReLU could have exploded and improperly updated the weights.

5 Question 5

This time, the validation set is selected as ten percent of the training set.

Question 5.1

I picked the cnn3 architecture for learning rate experimentation. The learning rates are selected as 0.1, 0.01, 0.001, 0.0001 and 0.00001. The results are shown in Figure 13.
Then scheduling is applied to the learning rate. Figure 14,15,16 illustrates the scheduling applied cases. The learning rate is decreased by a factor of 0.1 at each selected epoch.
It should be indicated that an inconsistent behavior for dropping the learning rate only for one case is observed. For example, when the exact same setup is run on different CUDA setups, the results are quite different, yet this also can be observed from the initial part of the b). This is repeated multiple times in different computation units. This was assumed to be an unsolved problem related to random

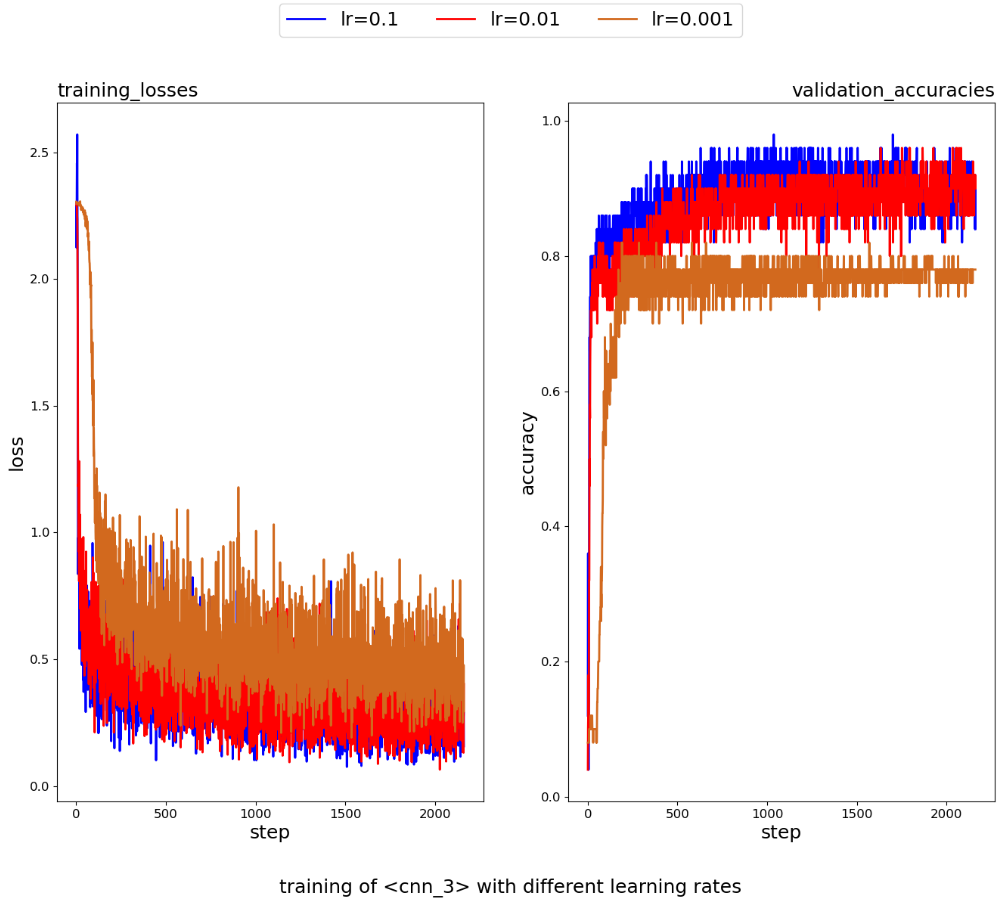


Figure 13: Different learning rate settings on cnn3.

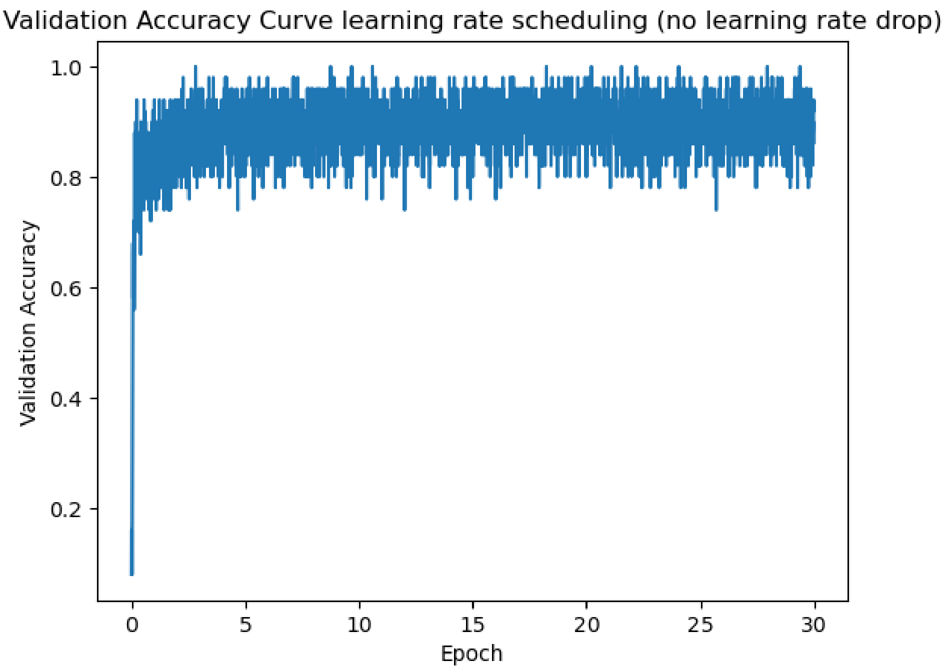


Figure 14: No scheduling

initializations and seeds.

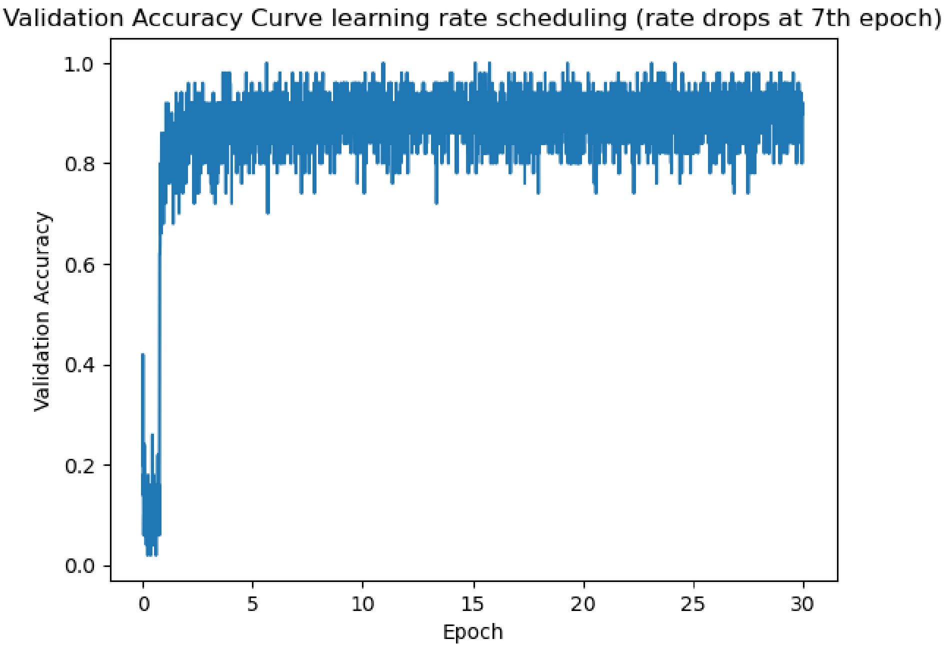


Figure 15: LR dropped at 7th epch

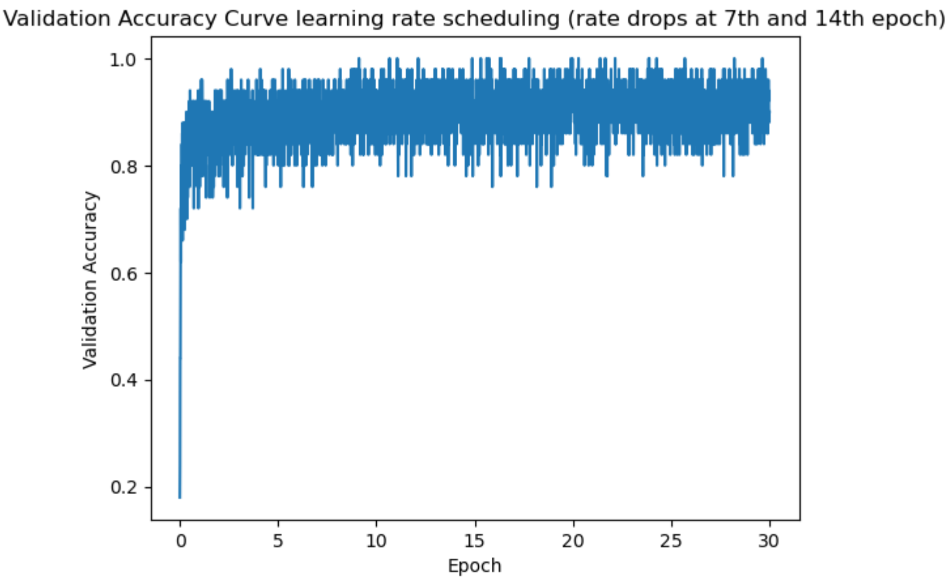


Figure 16: LR dropped at 7th and 15th epoch

Question 5.2 - Discussions

1. As can be observed from Figure 13, as the learning rate increases, the convergence speed also increases.

2. There are two points to consider when it comes to convergence to a better point. First, if the learning rate is too small, the weights may stuck at a suboptimal local minima. Also, even if the system convergence around the same minima, a small learning rate may lead to slow convergence. On the other hand, if the learning rate is too high, the weights may oscillate around the optimal point and may not go beyond a certain level. Therefore, the learning rate should be selected carefully. So, in order to find the best learning rate, one can start with a high learning rate and decrease it gradually.

3. In this case, the learning scheduling attempt worked out in terms of going to a better point. The oscillations around a convergence level are overcome by decreasing the learning rate.

4. When we compare these results with the one in question 3, where we trained the network with an ADAM optimizer, we can see that SGD yields a slightly better point. However, ADAM seems to converge faster than SGD. Yet, in this case, these two differences are quite subtle.

Appendix

The code set used throughout this homework is provided as follows. The code is written using VSCode and jupyter notebook infrastructure. The code is run on several different computational units that is why there are generic CUDA and CPU selection expressions. The code is converted from jupyter notebook to python script using the jupyter nbconvert command. At the end checkers used for q2 and a3 are also provided.

```
# %% [markdown]
# # Homework 1 Ahmet Akman
# This notebook consists the efforts led by homework 1 and done by Ahmet Akman in the
    scope of EE449 Course

# %% [markdown]
# ## Imports

# %%
import numpy as np
# import torch
import matplotlib.pyplot as plt
# import sklearn

# %% [markdown]
# ## Basic Neural Network Construction and Training
#

# %% [markdown]
# ### 1.1
#
#
#

# %%
x = np.arange(-2, 2, 0.01, dtype = float)
tanh_result = (np.exp(2*x)-1) / (np.exp(2*x)+1)
sigmoid_result =  1 / (np.exp(-x)+1)
ReLU_result = np.maximum(0,x)


fig, ax = plt.subplots(3)
ax[0].plot(x,tanh_result)
ax[0].set_title("Tanh function")
ax[0].set_xlabel("x")
ax[0].set_ylabel("tanh(x)")
ax[0].grid()

ax[1].plot(x,sigmoid_result)
ax[1].set_title("Sigmoid function")
ax[1].set_xlabel("x")
ax[1].set_ylabel("sigma(x)")
ax[1].grid()

ax[2].plot(x,ReLU_result)
```

```python
ax[2].set_title("ReLU function")
ax[2].set_xlabel("x")
ax[2].set_ylabel("ReLU(x)")
ax[2].grid()
fig.tight_layout()


# %%
p_tanh_result = 1- (np.exp(x)-np.exp(-x))**2 / (np.exp(x) + np.exp(-x))**2
p_sigmoid_result =  np.exp(-x) / (np.exp(-x)+1)**2
p_ReLU_result = x >= 0


fig, ax = plt.subplots(3)
ax[0].plot(x, p_tanh_result)
ax[0].set_title("Partial derivative of Tanh function")
ax[0].set_xlabel("x")
ax[0].set_ylabel("tanh(x)")
ax[0].grid()

ax[1].plot(x, p_sigmoid_result)
ax[1].set_title("Partial derivative of Sigmoid function")
ax[1].set_xlabel("x")
ax[1].set_ylabel("sigma(x)")
ax[1].grid()

ax[2].plot(x, p_ReLU_result)
ax[2].set_title("Partial derivative of ReLU function")
ax[2].set_xlabel("x")
ax[2].set_ylabel("ReLU(x)")
ax[2].grid()
fig.tight_layout()

# %% [markdown]
# ### 1.2
# - Sigmoid activated MLP.
#

# %%
np.random.seed(1234)
learning__rate_for_all_activation_functions = 0.00001

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

        # Initialize weights and biases
        self.weights_input_hidden = np.random.randn(self.input_size, self.hidden_size)
        self.bias_hidden = np.zeros((1, self.hidden_size))
        self.weights_hidden_output = np.random.randn(self.hidden_size, self.output_size)
        self.bias_output = np.zeros((1, self.output_size))

    def sigmoid(self, x):
        return np.exp(x) / (np.exp(x)+1)
    def sigmoid_derivative(self, x):
        return np.exp(-x) / (np.exp(-x)+1)**2

```

```
    def forward(self, inputs):
        # Forward pass through the network
        self.hidden_output =  self.sigmoid(np.dot(inputs, self.weights_input_hidden) + self.bias_hidden)
        self.output =  np.round(np.dot(self.hidden_output, self.weights_hidden_output) + self.bias_output)
        return self.output

    def backward(self, inputs, targets, learning_rate):
        # Backward pass through the network
        # Compute error
        output_error = targets - self.output
        hidden_error = output_error * self.sigmoid_derivative(self.hidden_output)
        # Compute gradients
        output_delta = output_error * self.sigmoid_derivative(self.output)
        hidden_delta = hidden_error * self.sigmoid_derivative(self.hidden_output)
        # Update weights and biases
        self.weights_hidden_output = self.weights_hidden_output + learning_rate * np.dot(self.hidden_output.T, output_delta)
        self.bias_output = self.bias_output + learning_rate * np.sum(output_delta, axis=0, keepdims=True)
        self.weights_input_hidden = self.weights_input_hidden + learning_rate * np.dot(inputs.T, hidden_delta)
        self.bias_hidden = self.bias_hidden + learning_rate * np.sum(hidden_delta, axis=0, keepdims=True)

from utils import part1CreateDataset, part1PlotBoundary
x_train, y_train, x_val, y_val = part1CreateDataset(train_samples=1000, val_samples=100, std=0.4)

# Define neural network parameters
input_size = 2
hidden_size = 4
output_size = 1
learning_rate = learning__rate_for_all_activation_functions
# Create neural network
nn = MLP_sigmoid(input_size, hidden_size, output_size)
# Train the neural network
for epoch in range(10000):
    # Forward propagation
    output = nn.forward(x_train)
    # Backpropagation
    nn.backward(x_train, y_train, learning_rate)

    # Print the loss (MSE) every 1000 epochs
    if epoch % 1000 == 0:
        loss = np.mean((y_train - output)**2)
        print(f'Epoch {epoch}: Loss = {loss}')


# Test the trained neural network
y_predict = nn.forward(x_val)
print(f'{np.mean(y_predict==y_val)*100} % of test examples classified correctly.')


part1PlotBoundary(x_val, y_val, nn)

# %% [markdown]
# - Tanh activated MLP.
```

```python

# %%
class MLP_tanh:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # Initialize weights and biases
        self.weights_input_hidden = np.random.randn(self.input_size, self.hidden_size)
        self.bias_hidden = np.zeros((1, self.hidden_size))
        self.weights_hidden_output = np.random.randn(self.hidden_size, self.output_size)
        self.bias_output = np.zeros((1, self.output_size))

    def tanh(self, x):
        return (np.exp(2*x)-1) / (np.exp(2*x)+1)
    def tanh_derivative(self, x):
        return 1- (np.exp(x)-np.exp(-x))**2 / (np.exp(x) + np.exp(-x))**2

    def forward(self, inputs):
        # Forward pass through the network
        self.hidden_output =  self.tanh(np.dot(inputs, self.weights_input_hidden) + self.bias_hidden)
        self.output =  np.round(np.dot(self.hidden_output, self.weights_hidden_output) + self.bias_output)
        return self.output

    def backward(self, inputs, targets, learning_rate):
        # Backward pass through the network
        # Compute error
        output_error = targets - self.output
        hidden_error = output_error * self.tanh_derivative(self.hidden_output)
        # Compute gradients
        output_delta = output_error * self.tanh_derivative(self.output)
        hidden_delta = hidden_error * self.tanh_derivative(self.hidden_output)
        # Update weights and biases
        self.weights_hidden_output = self.weights_hidden_output + learning_rate * np.dot(self.hidden_output.T, output_delta)
        self.bias_output = self.bias_output + learning_rate * np.sum(output_delta, axis=0, keepdims=True)
        self.weights_input_hidden = self.weights_input_hidden + learning_rate * np.dot(inputs.T, hidden_delta)
        self.bias_hidden = self.bias_hidden + learning_rate * np.sum(hidden_delta, axis=0, keepdims=True)

from utils import part1CreateDataset, part1PlotBoundary
x_train, y_train, x_val, y_val = part1CreateDataset(train_samples=1000, val_samples=100, std=0.4)

# Define neural network parameters
input_size = 2
hidden_size = 4
output_size = 1
learning_rate = learning__rate_for_all_activation_functions
# Create neural network
nn = MLP_tanh(input_size, hidden_size, output_size)
# Train the neural network
for epoch in range(10000):
    # Forward propagation
```

```python
    output = nn.forward(x_train)
    # Backpropagation
    nn.backward(x_train, y_train, learning_rate)

    # Print the loss (MSE) every 1000 epochs
    if epoch % 1000 == 0:
        loss = np.mean((y_train - output)**2)
        print(f'Epoch {epoch}: Loss = {loss}')


# Test the trained neural network
y_predict = nn.forward(x_val)
print(f'{np.mean(y_predict==y_val)*100} % of test examples classified correctly.')


part1PlotBoundary(x_val, y_val, nn)

# %% [markdown]
# - ReLU Activated MLP
#

# %%
class MLP_ReLU:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # Initialize weights and biases
        self.weights_input_hidden = np.random.randn(self.input_size, self.hidden_size)
        self.bias_hidden = np.zeros((1, self.hidden_size))
        self.weights_hidden_output = np.random.randn(self.hidden_size, self.output_size)
        self.bias_output = np.zeros((1, self.output_size))

    def ReLU(self, x):
        return np.maximum(0,x)
    def ReLU_derivative(self, x):
        return x >= 0

    def forward(self, inputs):
        # Forward pass through the network
        self.hidden_output =  self.ReLU(np.dot(inputs, self.weights_input_hidden) + self.bias_hidden)
        self.output =  np.round(np.dot(self.hidden_output, self.weights_hidden_output) + self.bias_output)
        return self.output

    def backward(self, inputs, targets, learning_rate):
        # Backward pass through the network
        # Compute error
        output_error = targets - self.output
        hidden_error = output_error * self.ReLU_derivative(self.hidden_output)
        # Compute gradients
        output_delta = output_error * self.ReLU_derivative(self.output)
        hidden_delta = hidden_error * self.ReLU_derivative(self.hidden_output)
        # Update weights and biases
        self.weights_hidden_output = self.weights_hidden_output + learning_rate * np.dot(self.hidden_output.T, output_delta)
```

```
        self.bias_output = self.bias_output + learning_rate * np.sum(output_delta, axis=0, keepdims=True)
        self.weights_input_hidden = self.weights_input_hidden + learning_rate * np.dot(inputs.T, hidden_delta)
        self.bias_hidden = self.bias_hidden + learning_rate * np.sum(hidden_delta, axis=0, keepdims=True)

from utils import part1CreateDataset, part1PlotBoundary
x_train, y_train, x_val, y_val = part1CreateDataset(train_samples=1000, val_samples=100, std=0.4)

# Define neural network parameters
input_size = 2
hidden_size = 4
output_size = 1
learning_rate = learning__rate_for_all_activation_functions
# Create neural network
nn = MLP_ReLU(input_size, hidden_size, output_size)
# Train the neural network
for epoch in range(10000):
    # Forward propagation
    output = nn.forward(x_train)
    # Backpropagation
    nn.backward(x_train, y_train, learning_rate)

    # Print the loss (MSE) every 1000 epochs
    if epoch % 1000 == 0:
        loss = np.mean((y_train - output)**2)
        print(f'Epoch {epoch}: Loss = {loss}')


# Test the trained neural network
y_predict = nn.forward(x_val)
print(f'{np.mean(y_predict==y_val)*100} % of test examples classified correctly.')


part1PlotBoundary(x_val, y_val, nn)

# %% [markdown]
# ### 1.3
#

# %% [markdown]
# ## Implementing a Convolutional Layer with NumPy

# %%


# implement a 2D convolutional layer using numpy
def my_conv2d(input, kernel):
    # input shape: [batch size, input_channels, input_height, input_width]
    # kernel shape: [output_channels, input_channels, filter_height, filter width]
    batch_size, input_channels, input_height, input_width = input.shape
    output_channels, input_channels, filter_height, filter_width = kernel.shape
    # output shape: [batch size, output_channels, output_height, output_width]
    output_height = input_height - filter_height + 1
    output_width = input_width - filter_width + 1
    output = np.zeros((batch_size, output_channels, output_height, output_width))
    for b in range(batch_size):
```

```
        for oc in range(output_channels):
            for ic in range(input_channels):
                for i in range(output_height):
                    for j in range(output_width):
                        output[b, oc, i, j] = np.sum(input[b, ic, i:i+filter_height, j:j+filter_width] * kernel[oc, ic])
    return output

# input shape: [batch size, input_channels, input_height, input_width]
kernel=np.load('data/kernel.npy')

for i in range(10):
    input=np.load('data/samples_{}.npy'.format(i))
    # input shape: [output_channels, input_channels, filter_height, filter width]
    out = my_conv2d(input, kernel)
    out_check = torch.conv2d(torch.tensor(input).float(), torch.tensor(kernel).float())
    np.save('outputs/out_{}.npy'.format(i), out)

    from utils import part2Plots
    part2Plots(out = out, save_dir='outputs', filename='out_{}'.format(i) )
    part2Plots(out = out_check, save_dir='outputs', filename='out_{}_check'.format(i) )

# %% [markdown]
# ## 2.2
#

# %%
%reset -f

# %% [markdown]
# ## Experimenting ANN Architectures

# %%
# Load fashion MNIST dataset
import torchvision

#training set
train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=True, transform= torchvision.transforms.ToTensor(), shuffle=True)

#test set
test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=True, transform= torchvision.transforms.ToTensor())


# %%
import torch
# divide training data into training and validation sets of 0.8 and 0.2 respectively

train_data, val_data = torch.utils.data.random_split(train_data, [0.8, 0.2])

train_generator = torch.utils.data.DataLoader(train_data, batch_size=96, shuffle=True)
val_generator = torch.utils.data.DataLoader(val_data, batch_size=96, shuffle=False)
test_generator = torch.utils.data.DataLoader(test_data, batch_size=96, shuffle=False)


# %% [markdown]
# ### mlp_1 case
# mlp_1 corresponds to FC-32, ReLU + FC10
```

```python

# %%

from tqdm import tqdm

# example mlp classifier
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.FC = torch.nn.Linear(input_size, hidden_size)
        self.prediction_layer = 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.FC(x)
        relu = self.relu(hidden)
        output = self.prediction_layer(relu)
        return output

# initialize your model
model_mlp_1 = mlp_1(784,32,10)
if torch.cuda.is_available():
    device = torch.device("cuda:0")
    print("CUDA to be used")
else:
    device = "cpu"
    print("CPU to be used.")
model_mlp_1.to(device)
# create loss: use cross entropy loss
loss = torch.nn.CrossEntropyLoss()
# create optimizer
optimizer = torch.optim.Adam(model_mlp_1.parameters(), lr=0.001)
# transfer your model to train mode
model_mlp_1.train()


mlp_1_dict = {"name":"mlp_1", "loss_curve": [],  "train_acc_curve": [], "val_acc_curve":
    [], "test_acc": 0, 'weights': []}



# train the model and save the training loss and validation loss for every 10 batches
    using model eval mode.s
for epoch in tqdm(range(15)):
    for batch, (x, y) in enumerate(train_generator):
        x = x.to(device)
        y = y.to(device)
        # forward pass
        output = model_mlp_1(x)
        # compute loss
        loss_val = loss(output, y)
        # zero gradients
        optimizer.zero_grad()
        # backward pass
        loss_val.backward()
        # optimize
        optimizer.step()
        # print loss
```

```
        if batch % 10 == 0:
            model_mlp_1.eval()
            #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item()))
            mlp_1_dict['loss_curve'].append(loss_val.item())
            training_accuracy = torch.mean((torch.argmax(output, dim=1) == y).float())
            mlp_1_dict['train_acc_curve'].append(training_accuracy.item())

            # validation loss
            validation_accuracy_per_batch = torch.tensor([])
            with torch.no_grad():
                for val_x, val_y in val_generator:
                    val_x = val_x.to(device)
                    val_y = val_y.to(device)

                    val_output = model_mlp_1(val_x)
                    #val_loss = loss(val_output, val_y)
                    val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).float())
                    try:
                        validation_accuracy_per_batcuda = torch.cat((validation_accuracy_per_batch ,val_accuracy))
                    except:
                        validation_accuracy_per_batch = val_accuracy
            #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean()))
            mlp_1_dict['val_acc_curve'].append(validation_accuracy_per_batch.mean().item())
            model_mlp_1.train()



# test the model
correct = 0
total = 0
for x, y in test_generator:
    x = x.to(device)
    y = y.to(device)

    output = model_mlp_1(x)
    _, predicted = torch.max(output, 1)
    total += y.size(0)
    correct += (predicted == y).sum().item()
print('Test Accuracy: {} %'.format(100 * correct / total))
mlp_1_dict['test_acc'] = 100 * correct / total
# save the model as a pty file
torch.save(model_mlp_1.state_dict(), 'q3_models/model_mlp_1.pty')
print("model saved as 'q3_models/model_mlp_1.pty'")
# get the parameters 784x32 layer as numpy array
weights_first_layer = model_mlp_1.FC.weight.data.cpu().numpy()
mlp_1_dict['weights'] = weights_first_layer


# %%
# save mlp_1_dict as a pickle file
import pickle
with open('q3_models/mlp_1_dict.pkl', 'wb') as f:
    pickle.dump(mlp_1_dict, f)
```

```

# %% [markdown]
# ### mlp_2 case
#

# %%
%reset -f

# Load fashion MNIST dataset
import torchvision

from tqdm import tqdm

#training set
train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False,
    transform= torchvision.transforms.ToTensor(), shuffle=True)

#test set
test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False,
    transform= torchvision.transforms.ToTensor())


import torch
# divide training data into training and validation sets of 0.8 and 0.2 respectively

train_data, val_data = torch.utils.data.random_split(train_data, [0.8, 0.2])

train_generator = torch.utils.data.DataLoader(train_data, batch_size=96, shuffle=True)
val_generator = torch.utils.data.DataLoader(val_data, batch_size=96, shuffle=False)
test_generator = torch.utils.data.DataLoader(test_data, batch_size=96, shuffle=False)





# example mlp classifier
class mlp_2(torch.nn.Module):
    def __init__(self, input_size, hidden_size_1, hidden_size_2, num_classes):
        super(mlp_2, self).__init__()
        self.input_size = input_size
        self.FC1 = torch.nn.Linear(input_size, hidden_size_1)
        self.FC2 = torch.nn.Linear(hidden_size_1, hidden_size_2, bias=False)
        self.prediction_layer = torch.nn.Linear(hidden_size_2, num_classes)
        self.relu = torch.nn.ReLU()
    def forward(self, x):
        x = x.view(-1, self.input_size)
        hidden1 = self.FC1(x)
        relu = self.relu(hidden1)
        hidden2 = self.FC2(relu)
        output = self.prediction_layer(hidden2)
        return output

# initialize your model
model_mlp_2 = mlp_2(784 ,32 ,64 ,10)

if torch.cuda.is_available():
    device = torch.device("cuda:0")
    print("CUDA to be used")
else:
```

```
    device = "cpu"
    print("CPU to be used.")
model_mlp_2.to(device)

# create loss: use cross entropy loss
loss = torch.nn.CrossEntropyLoss()
# create optimizer
optimizer = torch.optim.Adam(model_mlp_2.parameters(), lr=0.001)
# transfer your model to train mode
model_mlp_2.train()


mlp_2_dict = {"name":"mlp_2", "loss_curve": [],  "train_acc_curve": [], "val_acc_curve":
    [], "test_acc": 0, 'weights': []}



# train the model and save the training loss and validation loss for every 10 batches
    using model eval mode.s
for epoch in tqdm(range(15)):
    for batch, (x, y) in enumerate(train_generator):
        x = x.to(device)
        y = y.to(device)

        # forward pass
        output = model_mlp_2(x)
        # compute loss
        loss_val = loss(output, y)
        # zero gradients
        optimizer.zero_grad()
        # backward pass
        loss_val.backward()
        # optimize
        optimizer.step()
        # print loss
        if batch % 10 == 0:
            model_mlp_2.eval()
            #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
    )
            mlp_2_dict['loss_curve'].append(loss_val.item())
            training_accuracy = torch.mean((torch.argmax(output, dim=1) == y).float())
            mlp_2_dict['train_acc_curve'].append(training_accuracy.item())

            # validation loss
            validation_accuracy_per_batch = torch.tensor([])
            with torch.no_grad():
                for val_x, val_y in val_generator:
                    val_x = val_x.to(device)
                    val_y = val_y.to(device)

                    val_output = model_mlp_2(val_x)
                    #val_loss = loss(val_output, val_y)
                    val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).
    float())
                    try:
                        validation_accuracy_per_batch = torch.cat((
    validation_accuracy_per_batch ,val_accuracy))
                    except:
                        validation_accuracy_per_batch = val_accuracy
```

```python
            #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean())
    )
            mlp_2_dict['val_acc_curve'].append(validation_accuracy_per_batch.mean().item
    ())
            model_mlp_2.train()



# test the model
correct = 0
total = 0
for x, y in test_generator:
    x = x.to(device)
    y = y.to(device)

    output = model_mlp_2(x)
    _, predicted = torch.max(output, 1)
    total += y.size(0)
    correct += (predicted == y).sum().item()
print('Test Accuracy: {} %'.format(100 * correct / total))
mlp_2_dict['test_acc'] = 100 * correct / total
# save the model as a pty file
torch.save(model_mlp_2.state_dict(), 'q3_models/model_mlp_2.pty')
print("model saved as 'q3_models/model_mlp_2.pty'")
# get the parameters 784x32 layer as numpy array
weights_first_layer = model_mlp_2.FC1.weight.data.cpu().numpy()
mlp_2_dict['weights'] = weights_first_layer


# save mlp_2_dict as a pickle file
import pickle
with open('q3_models/mlp_2_dict.pkl', 'wb') as f:
    pickle.dump(mlp_2_dict, f)

# %% [markdown]
# ### CNN3
# Where Conv - 3x3x16 Relu
# Conv - 5x5x8 Relu, MaxPool 2x2
# Conv - 7x7x16 MaxPool 2x2
# Fully connected layer of 10.

# %%
%reset -f

# Load fashion MNIST dataset
import torchvision

from tqdm import tqdm

#training set
train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False,
     transform= torchvision.transforms.ToTensor(), shuffle=True)

#test set
test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False,
     transform= torchvision.transforms.ToTensor())


import torch
```

```python
# divide training data into training and validation sets of 0.8 and 0.2 respectively

train_data, val_data = torch.utils.data.random_split(train_data, [0.8, 0.2])

train_generator = torch.utils.data.DataLoader(train_data, batch_size=96, shuffle=True)
val_generator = torch.utils.data.DataLoader(val_data, batch_size=96, shuffle=False)
test_generator = torch.utils.data.DataLoader(test_data, batch_size=96, shuffle=False)





# example cnn_3 classifier
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(1 ,16 ,3, stride=1, padding=1)
        self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
        self.Conv2 = torch.nn.Conv2d(16, 8, 5,  stride=1, padding=1)
        self.Conv3 = torch.nn.Conv2d(8, 16, 7, stride=1, padding=1)
        self.prediction_layer = torch.nn.Linear(1296, num_classes)
        self.relu = torch.nn.ReLU()
    def forward(self, x):
        x = x.view(-1, 1, 28, 28)
        hidden1 = self.Conv1(x)
        relu1 = self.relu(hidden1)
        hidden2 = self.Conv2(relu1)
        relu2 = self.relu(hidden2)
        pool = self.MaxPool(relu2)
        hidden3 = self.Conv3(pool)
        flattened = hidden3.view(96, -1)
        output = self.prediction_layer(flattened)
        return output

# initialize your model
model_cnn_3 = cnn_3(784 ,10)
if torch.cuda.is_available():
    device = torch.device("cuda:0")
    print("CUDA to be used")
else:
    device = "cpu"
    print("CPU to be used.")
model_cnn_3.to(device)

# create loss: use cross entropy loss
loss = torch.nn.CrossEntropyLoss()
# create optimizer
optimizer = torch.optim.Adam(model_cnn_3.parameters(), lr=0.001)
# transfer your model to train mode
model_cnn_3.train()


cnn_3_dict = {"name":"cnn_3", "loss_curve": [],  "train_acc_curve": [], "val_acc_curve":
    [], "test_acc": 0, 'weights': []}
```

```
# train the model and save the training loss and validation loss for every 10 batches
    using model eval mode.s
for epoch in tqdm(range(15)):
    for batch, (x, y) in enumerate(train_generator):

        x = x.to(device)
        y = y.to(device)

        # forward pass
        output = model_cnn_3(x)
        # compute loss
        loss_val = loss(output, y)
        # zero gradients
        optimizer.zero_grad()
        # backward pass
        loss_val.backward()
        # optimize
        optimizer.step()
        # print loss
        if batch % 10 == 0:
            model_cnn_3.eval()
            #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
    )
            cnn_3_dict['loss_curve'].append(loss_val.item())
            training_accuracy = torch.mean((torch.argmax(output, dim=1) == y).float())
            cnn_3_dict['train_acc_curve'].append(training_accuracy.item())

            # validation loss
            validation_accuracy_per_batch = torch.tensor([])
            with torch.no_grad():
                for val_x, val_y in val_generator:

                    val_x = val_x.to(device)
                    val_y = val_y.to(device)

                    val_output = model_cnn_3(val_x)
                    #val_loss = loss(val_output, val_y)
                    val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).
    float())
                    try:
                        validation_accuracy_per_batch = torch.cat((
    validation_accuracy_per_batch ,val_accuracy))
                    except:
                        validation_accuracy_per_batch = val_accuracy
            #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean())
    )
            cnn_3_dict['val_acc_curve'].append(validation_accuracy_per_batch.mean().item
    ())
            model_cnn_3.train()



# %%


# test the model
correct = 0
total = 0
for x, y in test_generator:
```

```
    if y.size(dim = 0) < 96:
        break
    x = x.to(device)
    y = y.to(device)

    output = model_cnn_3(x)
    _, predicted = torch.max(output, 1)
    total += y.size(0)
    correct += (predicted == y).sum().item()
print('Test Accuracy: {} %'.format(100 * correct / total))
cnn_3_dict['test_acc'] = 100 * correct / total
# save the model as a pty file
torch.save(model_cnn_3.state_dict(), 'q3_models/model_cnn_3.pty')
print("model saved as 'q3_models/model_cnn_3.pty'")
# get the parameters 784x32 layer as numpy array
weights_first_layer = model_cnn_3.Conv1.weight.data.cpu().numpy()
cnn_3_dict['weights'] = weights_first_layer

# save cnn_3 as a pickle file
import pickle
with open('q3_models/cnn_3_dict.pkl', 'wb') as f:
    pickle.dump(cnn_3_dict, f)

# %% [markdown]
# ### CNN 4
#

# %%
%reset -f

# Load fashion MNIST dataset
import torchvision

from tqdm import tqdm

#training set
train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False,
     transform= torchvision.transforms.ToTensor(), shuffle=True)

#test set
test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False,
     transform= torchvision.transforms.ToTensor())


import torch
# divide training data into training and validation sets of 0.8 and 0.2 respectively

train_data, val_data = torch.utils.data.random_split(train_data, [0.8, 0.2])

train_generator = torch.utils.data.DataLoader(train_data, batch_size=96, shuffle=True)
val_generator = torch.utils.data.DataLoader(val_data, batch_size=96, shuffle=False)
test_generator = torch.utils.data.DataLoader(test_data, batch_size=96, shuffle=False)





# example cnn_4 classifier
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(1 ,16 ,3, stride=1, padding=1)
        self.Conv2 = torch.nn.Conv2d(16, 8, 3,  stride=1, padding=1)
        self.Conv3 = torch.nn.Conv2d(8, 16, 5, stride=1, padding=1)
        self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
        self.Conv4 = torch.nn.Conv2d(16, 16, 5,  stride=1, padding=1)
        self.prediction_layer = torch.nn.Linear(400, num_classes)
        self.relu = torch.nn.ReLU()
    def forward(self, x):
        x = x.view(-1, 1, 28, 28)
        hidden1 = self.Conv1(x)
        relu1 = self.relu(hidden1)
        hidden2 = self.Conv2(relu1)
        relu2 = self.relu(hidden2)
        hidden3 = self.Conv3(relu2)
        relu3 = self.relu(hidden3)
        pool1 = self.MaxPool(relu3)
        hidden4 = self.Conv4(pool1)
        pool2 = self.MaxPool(hidden4)
        flattened = pool2.view(96, -1)
        output = self.prediction_layer(flattened)
        return output

# initialize your model
model_cnn_4 = cnn_4(784 ,10)

if torch.cuda.is_available():
    device = torch.device("cuda:0")
    print("CUDA to be used")
else:
    device = "cpu"
    print("CPU to be used.")
model_cnn_4.to(device)


# create loss: use cross entropy loss
loss = torch.nn.CrossEntropyLoss()
# create optimizer
optimizer = torch.optim.Adam(model_cnn_4.parameters(), lr=0.001)
# transfer your model to train mode
model_cnn_4.train()


cnn_4_dict = {"name":"cnn_4", "loss_curve": [],  "train_acc_curve": [], "val_acc_curve":
    [], "test_acc": 0, 'weights': []}



# train the model and save the training loss and validation loss for every 10 batches
    using model eval mode.s
for epoch in tqdm(range(15)):
    for batch, (x, y) in enumerate(train_generator):

        x = x.to(device)
        y = y.to(device)

        # forward pass
```

```
        output = model_cnn_4(x)
        # compute loss
        loss_val = loss(output, y)
        # zero gradients
        optimizer.zero_grad()
        # backward pass
        loss_val.backward()
        # optimize
        optimizer.step()
        # print loss
        if batch % 10 == 0:
            model_cnn_4.eval()
            #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
    )
            cnn_4_dict['loss_curve'].append(loss_val.item())
            training_accuracy = torch.mean((torch.argmax(output, dim=1) == y).float())
            cnn_4_dict['train_acc_curve'].append(training_accuracy.item())

            # validation loss
            validation_accuracy_per_batch = torch.tensor([])
            with torch.no_grad():
                for val_x, val_y in val_generator:

                    val_x = val_x.to(device)
                    val_y = val_y.to(device)

                    val_output = model_cnn_4(val_x)
                    #val_loss = loss(val_output, val_y)
                    val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).
    float())
                    try:
                        validation_accuracy_per_batch = torch.cat((
    validation_accuracy_per_batch ,val_accuracy))
                    except:
                        validation_accuracy_per_batch = val_accuracy
            #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean())
    )
            cnn_4_dict['val_acc_curve'].append(validation_accuracy_per_batch.mean().item
    ())
            model_cnn_4.train()



# test the model
correct = 0
total = 0
for x, y in test_generator:
    if y.size(dim = 0) < 96:
        break
    x = x.to(device)
    y = y.to(device)

    output = model_cnn_4(x)
    _, predicted = torch.max(output, 1)
    total += y.size(0)
    correct += (predicted == y).sum().item()
print('Test Accuracy: {} %'.format(100 * correct / total))
cnn_4_dict['test_acc'] = 100 * correct / total
# save the model as a pty file
```

```
torch.save(model_cnn_4.state_dict(), 'q3_models/model_cnn_4.pty')
print("model saved as 'q3_models/model_cnn_4.pty'")
# get the parameters 784x32 layer as numpy array
weights_first_layer = model_cnn_4.Conv1.weight.data.cpu().numpy()
cnn_4_dict['weights'] = weights_first_layer

# save cnn_4 as a pickle file
import pickle
with open('q3_models/cnn_4_dict.pkl', 'wb') as f:
    pickle.dump(cnn_4_dict, f)

# %% [markdown]
# ### CNN 5
#

# %%
%reset -f

# Load fashion MNIST dataset
import torchvision

from tqdm import tqdm

#training set
train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False,
     transform= torchvision.transforms.ToTensor(), shuffle=True)

#test set
test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False,
     transform= torchvision.transforms.ToTensor())


import torch
# divide training data into training and validation sets of 0.8 and 0.2 respectively

train_data, val_data = torch.utils.data.random_split(train_data, [0.8, 0.2])

train_generator = torch.utils.data.DataLoader(train_data, batch_size=96, shuffle=True)
val_generator = torch.utils.data.DataLoader(val_data, batch_size=96, shuffle=False)
test_generator = torch.utils.data.DataLoader(test_data, batch_size=96, shuffle=False)





# example cnn_5 classifier
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(1 ,8 ,3, stride=1, padding=1)
        self.Conv2 = torch.nn.Conv2d(8, 16, 3,  stride=1, padding=1)
        self.Conv3 = torch.nn.Conv2d(16, 8, 3, stride=1, padding=1)
        self.Conv4 = torch.nn.Conv2d(8, 16, 3,  stride=1, padding=1)
        self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
        self.Conv5 = torch.nn.Conv2d(16, 16, 3,  stride=1, padding=1)
        self.Conv6 = torch.nn.Conv2d(16, 8, 3,  stride=1, padding=1)
        self.prediction_layer = torch.nn.Linear(392, num_classes)
        self.relu = torch.nn.ReLU()
```

```python
    def forward(self, x):
        x = x.view(-1, 1, 28, 28)
        hidden1 = self.Conv1(x)
        relu1 = self.relu(hidden1)
        hidden2 = self.Conv2(relu1)
        relu2 = self.relu(hidden2)
        hidden3 = self.Conv3(relu2)
        relu3 = self.relu(hidden3)
        hidden4 = self.Conv4(relu3)
        relu4 = self.relu(hidden4)
        pool1 = self.MaxPool(relu4)
        hidden5 = self.Conv5(pool1)
        relu5 = self.relu(hidden5)
        hidden6 = self.Conv6(relu5)
        relu6 = self.relu(hidden6)
        pool2 = self.MaxPool(relu6)
        flattened = pool2.view(96, -1)
        output = self.prediction_layer(flattened)
        return output

# initialize your model
model_cnn_5 = cnn_5(784 ,10)

if torch.cuda.is_available():
    device = torch.device("cuda:0")
    print("CUDA to be used")
else:
    device = "cpu"
    print("CPU to be used.")
model_cnn_5.to(device)


# create loss: use cross entropy loss
loss = torch.nn.CrossEntropyLoss()
# create optimizer
optimizer = torch.optim.Adam(model_cnn_5.parameters(), lr=0.001)
# transfer your model to train mode
model_cnn_5.train()


cnn_5_dict = {"name":"cnn_5", "loss_curve": [],  "train_acc_curve": [], "val_acc_curve":
    [], "test_acc": 0, 'weights': []}



# train the model and save the training loss and validation loss for every 10 batches
    using model eval mode.s
for epoch in tqdm(range(15)):
    for batch, (x, y) in enumerate(train_generator):

        x = x.to(device)
        y = y.to(device)

        # forward pass
        output = model_cnn_5(x)
        # compute loss
        loss_val = loss(output, y)
        # zero gradients
        optimizer.zero_grad()
```

```
        # backward pass
        loss_val.backward()
        # optimize
        optimizer.step()
        # print loss
        if batch % 10 == 0:
            model_cnn_5.eval()
            #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item()))
            cnn_5_dict['loss_curve'].append(loss_val.item())
            training_accuracy = torch.mean((torch.argmax(output, dim=1) == y).float())
            cnn_5_dict['train_acc_curve'].append(training_accuracy.item())

            # validation loss
            validation_accuracy_per_batch = torch.tensor([])
            with torch.no_grad():
                for val_x, val_y in val_generator:
                    val_x = val_x.to(device)
                    val_y = val_y.to(device)


                    val_output = model_cnn_5(val_x)
                    #val_loss = loss(val_output, val_y)
                    val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).float())
                    try:
                        validation_accuracy_per_batch = torch.cat((validation_accuracy_per_batch ,val_accuracy))
                    except:
                        validation_accuracy_per_batch = val_accuracy
            #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean()))
            cnn_5_dict['val_acc_curve'].append(validation_accuracy_per_batch.mean().item())
            model_cnn_5.train()



# test the model
correct = 0
total = 0
for x, y in test_generator:
    if y.size(dim = 0) < 96:
        break
    x = x.to(device)
    y = y.to(device)

    output = model_cnn_5(x)
    _, predicted = torch.max(output, 1)
    total += y.size(0)
    correct += (predicted == y).sum().item()
print('Test Accuracy: {} %'.format(100 * correct / total))
cnn_5_dict['test_acc'] = 100 * correct / total
# save the model as a pty file
torch.save(model_cnn_5.state_dict(), 'q3_models/model_cnn_5.pty')
print("model saved as 'q3_models/model_cnn_5.pty'")
# get the parameters 784x32 layer as numpy array
weights_first_layer = model_cnn_5.Conv1.weight.data.cpu().numpy()
cnn_5_dict['weights'] = weights_first_layer
```

```python

# save cnn_5 as a pickle file
import pickle
with open('q3_models/cnn_5_dict.pkl', 'wb') as f:
    pickle.dump(cnn_5_dict, f)

# %%
%reset -f

import pickle

results = []
with open('q3_models/mlp_1_dict.pkl', 'rb') as f:
    data = pickle.load(f)

results.append(data)
print(data.keys())

with open('q3_models/mlp_2_dict.pkl', 'rb') as f:
    data = pickle.load(f)
results.append(data)
print(data.keys())


with open('q3_models/cnn_3_dict.pkl', 'rb') as f:
    data = pickle.load(f)
results.append(data)
print(data.keys())
with open('q3_models/cnn_4_dict.pkl', 'rb') as f:
    data = pickle.load(f)
results.append(data)
print(data.keys())

with open('q3_models/cnn_5_dict.pkl', 'rb') as f:
    data = pickle.load(f)
results.append(data)
print(data.keys())



from utils import part3Plots, visualizeWeights
# To plot the curves
part3Plots(results, save_dir='q3_models', filename='q3_results')

# To plot the weights
for i in range(len(results)):
    weights = results[i]['weights']
    visualizeWeights(weights, save_dir='q3_models', filename='weights_'+results[i]['name'
    ])

# %% [markdown]
# ## Experimenting Activation Functions

# %% [markdown]
# ### This part will consist of several activation function comparison trainings.
# First element will be the training of mlp_1 with SGD and ReLU
#

# %%
```

```python
%reset -f

# Load fashion MNIST dataset
import torchvision

from tqdm import tqdm

#training set
train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False, transform= torchvision.transforms.ToTensor())

#test set
test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False, transform= torchvision.transforms.ToTensor())


import torch
# divide training data into training and validation sets of 0.8 and 0.2 respectively

train_data, val_data = torch.utils.data.random_split(train_data, [0.8, 0.2])

train_generator = torch.utils.data.DataLoader(train_data, batch_size=50, shuffle=True)
val_generator = torch.utils.data.DataLoader(val_data, batch_size=50, shuffle=False)
test_generator = torch.utils.data.DataLoader(test_data, batch_size=50, shuffle=False)




# ReLU mlp classifier
class mlp_1_ReLU(torch.nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(mlp_1_ReLU, self).__init__()
        self.input_size = input_size
        self.FC = torch.nn.Linear(input_size, hidden_size)
        self.prediction_layer = 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.FC(x)
        relu = self.relu(hidden)
        output = self.prediction_layer(relu)
        return output

# Sigmoid mlp classifier
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.FC = torch.nn.Linear(input_size, hidden_size)
        self.prediction_layer = 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.FC(x)
        sigmoid = self.sigmoid(hidden)
        output = self.prediction_layer(sigmoid)
        return output
```

```python
# initialize the relu model
model_mlp_1_relu = mlp_1_ReLU(784,32,10)
# initialize the sigmoid model
model_mlp_1_sigmoid = mlp_1_Sigmoid(784,32,10)


if torch.cuda.is_available():
    device = torch.device("cuda:0")
    print("CUDA to be used")
else:
    device = "cpu"
    print("CPU to be used.")

model_mlp_1_relu.to(device)
model_mlp_1_sigmoid.to(device)

# create loss: use cross entropy loss
loss = torch.nn.CrossEntropyLoss()

# create optimizer
optimizer_relu = torch.optim.SGD(model_mlp_1_relu.parameters(), lr=0.01)
optimizer_sigmoid = torch.optim.SGD(model_mlp_1_sigmoid.parameters(), lr=0.01)


# transfer your model to train mode
model_mlp_1_relu.train()
model_mlp_1_sigmoid.train()

mlp_1_dict = {"name":"mlp_1", "relu_loss_curve": [],  "sigmoid_loss_curve": [], "relu_grad_curve": [], "sigmoid_grad_curve": []}



# train the model and save the training loss and validation loss for every 10 batches using model eval mode.s
for epoch in tqdm(range(15)):
    for batch, (x, y) in enumerate(train_generator):
        x = x.to(device)
        y = y.to(device)
        # forward pass
        output = model_mlp_1_relu(x)
        # compute loss
        loss_val = loss(output, y)
        # zero gradients
        optimizer_relu.zero_grad()
        # backward pass
        loss_val.backward()
        # optimize
        optimizer_relu.step()
        # print loss
        if batch % 10 == 0:
            model_mlp_1_relu.eval()
            #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item()))
            mlp_1_dict['relu_loss_curve'].append(loss_val.item())
            mlp_1_dict['relu_grad_curve'].append(model_mlp_1_relu.FC.weight.grad.mean().item())
```

```
            model_mlp_1_relu.train()




# save the model as a pty file
torch.save(model_mlp_1_relu.state_dict(), 'q4_models/model_mlp_1_relu.pty')
print("model saved as 'q4_models/model_mlp_1_relu.pty'")

# train the model and save the training loss and validation loss for every 10 batches using model eval mode.s
for epoch in tqdm(range(15)):
    for batch, (x, y) in enumerate(train_generator):
        x = x.to(device)
        y = y.to(device)
        # forward pass
        output = model_mlp_1_sigmoid(x)
        # compute loss
        loss_val = loss(output, y)
        # zero gradients
        optimizer_sigmoid.zero_grad()
        # backward pass
        loss_val.backward()
        # optimize
        optimizer_sigmoid.step()
        # print loss
        if batch % 10 == 0:
            model_mlp_1_sigmoid.eval()
            #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item()))
            mlp_1_dict['sigmoid_loss_curve'].append(loss_val.item())
            mlp_1_dict['sigmoid_grad_curve'].append(model_mlp_1_sigmoid.FC.weight.grad.mean().item())

            model_mlp_1_sigmoid.train()




# save the model as a pty file
torch.save(model_mlp_1_sigmoid.state_dict(), 'q4_models/model_mlp_1_sigmoid.pty')
print("model saved as 'q4_models/model_mlp_1_sigmoid.pty'")



# save mlp_1_dict as a pickle file
import pickle
with open('q4_models/part4_mlp_1_dict.pkl', 'wb') as f:
    pickle.dump(mlp_1_dict, f)

# %% [markdown]
# ### MLP2

# %%
%reset -f

# Load fashion MNIST dataset
import torchvision
```

```python

from tqdm import tqdm

#training set
train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False,
    transform= torchvision.transforms.ToTensor())

#test set
test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False,
    transform= torchvision.transforms.ToTensor())


import torch
# divide training data into training and validation sets of 0.8 and 0.2 respectively

train_data, val_data = torch.utils.data.random_split(train_data, [0.8, 0.2])

train_generator = torch.utils.data.DataLoader(train_data, batch_size=50, shuffle=True)
val_generator = torch.utils.data.DataLoader(val_data, batch_size=50, shuffle=False)
test_generator = torch.utils.data.DataLoader(test_data, batch_size=50, shuffle=False)




# ReLU mlp classifier
class mlp_2_ReLU(torch.nn.Module):
    def __init__(self, input_size, hidden_size_1, hidden_size_2, num_classes):
        super(mlp_2_ReLU, self).__init__()
        self.input_size = input_size
        self.FC1 = torch.nn.Linear(input_size, hidden_size_1)
        self.FC2 = torch.nn.Linear(hidden_size_1, hidden_size_2, bias=False)
        self.prediction_layer = torch.nn.Linear(hidden_size_2, num_classes)
        self.relu = torch.nn.ReLU()
    def forward(self, x):
        x = x.view(-1, self.input_size)
        hidden1 = self.FC1(x)
        relu = self.relu(hidden1)
        hidden2 = self.FC2(relu)
        output = self.prediction_layer(hidden2)
        return output


# Sigmoid mlp classifier
class mlp_2_Sigmoid(torch.nn.Module):
    def __init__(self, input_size, hidden_size_1, hidden_size_2, num_classes):
        super(mlp_2_Sigmoid, self).__init__()
        self.input_size = input_size
        self.FC1 = torch.nn.Linear(input_size, hidden_size_1)
        self.FC2 = torch.nn.Linear(hidden_size_1, hidden_size_2, bias=False)
        self.prediction_layer = torch.nn.Linear(hidden_size_2, num_classes)
        self.sigmoid = torch.nn.Sigmoid()
    def forward(self, x):
        x = x.view(-1, self.input_size)
        hidden1 = self.FC1(x)
        sigmoid = self.sigmoid(hidden1)
        hidden2 = self.FC2(sigmoid)
        output = self.prediction_layer(hidden2)
        return output

```

```python


# initialize the relu model
model_mlp_2_relu = mlp_2_ReLU(784 ,32 ,64 ,10)
# initialize the sigmoid model
model_mlp_2_sigmoid = mlp_2_Sigmoid(784 ,32 ,64 ,10)


if torch.cuda.is_available():
    device = torch.device("cuda:0")
    print("CUDA to be used")
else:
    device = "cpu"
    print("CPU to be used.")

model_mlp_2_relu.to(device)
model_mlp_2_sigmoid.to(device)

# create loss: use cross entropy loss
loss = torch.nn.CrossEntropyLoss()

# create optimizer
optimizer_relu = torch.optim.SGD(model_mlp_2_relu.parameters(), lr=0.01)
optimizer_sigmoid = torch.optim.SGD(model_mlp_2_sigmoid.parameters(), lr=0.01)


# transfer your model to train mode
model_mlp_2_relu.train()
model_mlp_2_sigmoid.train()

mlp_2_dict = {"name":"mlp_2", "relu_loss_curve": [],  "sigmoid_loss_curve": [], "relu_grad_curve": [], "sigmoid_grad_curve": []}



# train the model and save the training loss and validation loss for every 10 batches using model eval mode.s
for epoch in tqdm(range(15)):
    for batch, (x, y) in enumerate(train_generator):
        x = x.to(device)
        y = y.to(device)
        # forward pass
        output = model_mlp_2_relu(x)
        # compute loss
        loss_val = loss(output, y)
        # zero gradients
        optimizer_relu.zero_grad()
        # backward pass
        loss_val.backward()
        # optimize
        optimizer_relu.step()
        # print loss
        if batch % 10 == 0:
            model_mlp_2_relu.eval()
            #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item()))
            mlp_2_dict['relu_loss_curve'].append(loss_val.item())
            mlp_2_dict['relu_grad_curve'].append(model_mlp_2_relu.FC1.weight.grad.mean().item())
```

```
            model_mlp_2_relu.train()




# save the model as a pty file
torch.save(model_mlp_2_relu.state_dict(), 'q4_models/model_mlp_2_relu.pty')
print("model saved as 'q4_models/model_mlp_2_relu.pty'")

# train the model and save the training loss and validation loss for every 10 batches
    using model eval mode.s
for epoch in tqdm(range(15)):
    for batch, (x, y) in enumerate(train_generator):
        x = x.to(device)
        y = y.to(device)
        # forward pass
        output = model_mlp_2_sigmoid(x)
        # compute loss
        loss_val = loss(output, y)
        # zero gradients
        optimizer_sigmoid.zero_grad()
        # backward pass
        loss_val.backward()
        # optimize
        optimizer_sigmoid.step()
        # print loss
        if batch % 10 == 0:
            model_mlp_2_sigmoid.eval()
            #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
    )
            mlp_2_dict['sigmoid_loss_curve'].append(loss_val.item())
            mlp_2_dict['sigmoid_grad_curve'].append(model_mlp_2_sigmoid.FC1.weight.grad.
    mean().item())

            model_mlp_2_sigmoid.train()




# save the model as a pty file
torch.save(model_mlp_2_sigmoid.state_dict(), 'q4_models/model_mlp_2_sigmoid.pty')
print("model saved as 'q4_models/model_mlp_2_sigmoid.pty'")



# save mlp_2_dict as a pickle file
import pickle
with open('q4_models/part4_mlp_2_dict.pkl', 'wb') as f:
    pickle.dump(mlp_2_dict, f)

# %% [markdown]
# ### CNN3 Activation Function Experiment

# %%
%reset -f

# Load fashion MNIST dataset
import torchvision
```

```python

from tqdm import tqdm

#training set
train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False,
    transform= torchvision.transforms.ToTensor())

#test set
test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False,
    transform= torchvision.transforms.ToTensor())


import torch
# divide training data into training and validation sets of 0.8 and 0.2 respectively

batch_size = 50


train_data, val_data = torch.utils.data.random_split(train_data, [0.8, 0.2])

train_generator = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=
    True)
val_generator = torch.utils.data.DataLoader(val_data, batch_size=batch_size, shuffle=
    False)
test_generator = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=
    False)



# cnn_3 classifier
class cnn_3_relu(torch.nn.Module):
    def __init__(self, input_size, num_classes):
        super(cnn_3_relu, self).__init__()
        self.input_size = input_size
        self.Conv1 = torch.nn.Conv2d(1 ,16 ,3, stride=1, padding=1)
        self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
        self.Conv2 = torch.nn.Conv2d(16, 8, 5,  stride=1, padding=1)
        self.Conv3 = torch.nn.Conv2d(8, 16, 7, stride=1, padding=1)
        self.prediction_layer = torch.nn.Linear(1296, num_classes)
        self.relu = torch.nn.ReLU()
    def forward(self, x):
        x = x.view(-1, 1, 28, 28)
        hidden1 = self.Conv1(x)
        relu1 = self.relu(hidden1)
        hidden2 = self.Conv2(relu1)
        relu2 = self.relu(hidden2)
        pool = self.MaxPool(relu2)
        hidden3 = self.Conv3(pool)
        flattened = hidden3.view(batch_size, -1)
        output = self.prediction_layer(flattened)
        return output

# sigmoid cnn_3 classifier
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(1 ,16 ,3, stride=1, padding=1)
        self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
```

```
        self.Conv2 = torch.nn.Conv2d(16, 8, 5,  stride=1, padding=1)
        self.Conv3 = torch.nn.Conv2d(8, 16, 7, stride=1, padding=1)
        self.prediction_layer = torch.nn.Linear(1296, num_classes)
        self.sigmoid = torch.nn.Sigmoid()
    def forward(self, x):
        x = x.view(-1, 1, 28, 28)
        hidden1 = self.Conv1(x)
        sigmoid1 = self.sigmoid(hidden1)
        hidden2 = self.Conv2(sigmoid1)
        sigmoid2 = self.sigmoid(hidden2)
        pool = self.MaxPool(sigmoid2)
        hidden3 = self.Conv3(pool)
        flattened = hidden3.view(batch_size, -1)
        output = self.prediction_layer(flattened)
        return output


# initialize cnn_3 relu model
model_cnn_3_relu = cnn_3_relu(784 ,10)
# initialize cnn_3 sigmoid model
model_cnn_3_sigmoid = cnn_3_sigmoid(784 ,10)


if torch.cuda.is_available():
    device = torch.device("cuda:0")
    print("CUDA to be used")
else:
    device = "cpu"
    print("CPU to be used.")

model_cnn_3_relu.to(device)
model_cnn_3_sigmoid.to(device)

# create loss: use cross entropy loss
loss = torch.nn.CrossEntropyLoss()

# create optimizer
optimizer_relu = torch.optim.SGD(model_cnn_3_relu.parameters(), lr=0.01)
optimizer_sigmoid = torch.optim.SGD(model_cnn_3_sigmoid.parameters(), lr=0.01)


# transfer your model to train mode
model_cnn_3_relu.train()
model_cnn_3_sigmoid.train()

cnn_3_dict = {"name":"cnn_3", "relu_loss_curve": [],  "sigmoid_loss_curve": [], "
    relu_grad_curve": [], "sigmoid_grad_curve": []}



# train the model and save the training loss and validation loss for every 10 batches
    using model eval mode.s
for epoch in tqdm(range(15)):
    for batch, (x, y) in enumerate(train_generator):
        x = x.to(device)
        y = y.to(device)
        # forward pass
        output = model_cnn_3_relu(x)
        # compute loss
```

```python
        loss_val = loss(output, y)
        # zero gradients
        optimizer_relu.zero_grad()
        # backward pass
        loss_val.backward()
        # optimize
        optimizer_relu.step()
        # print loss
        if batch % 10 == 0:
            model_cnn_3_relu.eval()
            #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
    )
            cnn_3_dict['relu_loss_curve'].append(loss_val.item())
            cnn_3_dict['relu_grad_curve'].append(model_cnn_3_relu.Conv1.weight.grad.mean
    ().item())

            model_cnn_3_relu.train()




# save the model as a pty file
torch.save(model_cnn_3_relu.state_dict(), 'q4_models/model_cnn_3_relu.pty')
print("model saved as 'q4_models/model_cnn_3_relu.pty'")

# train the model and save the training loss and validation loss for every 10 batches
    using model eval mode.s
for epoch in tqdm(range(15)):
    for batch, (x, y) in enumerate(train_generator):
        x = x.to(device)
        y = y.to(device)
        # forward pass
        output = model_cnn_3_sigmoid(x)
        # compute loss
        loss_val = loss(output, y)
        # zero gradients
        optimizer_sigmoid.zero_grad()
        # backward pass
        loss_val.backward()
        # optimize
        optimizer_sigmoid.step()
        # print loss
        if batch % 10 == 0:
            model_cnn_3_sigmoid.eval()
            #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
    )
            cnn_3_dict['sigmoid_loss_curve'].append(loss_val.item())
            cnn_3_dict['sigmoid_grad_curve'].append(model_cnn_3_sigmoid.Conv1.weight.grad
    .mean().item())

            model_cnn_3_sigmoid.train()




# save the model as a pty file
torch.save(model_cnn_3_sigmoid.state_dict(), 'q4_models/model_cnn_3_sigmoid.pty')
print("model saved as 'q4_models/model_cnn_3_sigmoid.pty'")

```

```


# save cnn_3_dict as a pickle file
import pickle
with open('q4_models/part4_cnn_3_dict.pkl', 'wb') as f:
    pickle.dump(cnn_3_dict, f)

# %% [markdown]
# ### CNN4 Activation function experiment

# %%
%reset -f

# Load fashion MNIST dataset
import torchvision

from tqdm import tqdm

#training set
train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False,
     transform= torchvision.transforms.ToTensor())

#test set
test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False,
     transform= torchvision.transforms.ToTensor())

batch_size = 50

import torch
# divide training data into training and validation sets of 0.8 and 0.2 respectively

train_data, val_data = torch.utils.data.random_split(train_data, [0.8, 0.2])

train_generator = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=
    True)
val_generator = torch.utils.data.DataLoader(val_data, batch_size=batch_size, shuffle=
    False)
test_generator = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=
    False)

# relu cnn_4 classifier
class cnn_4_relu(torch.nn.Module):
    def __init__(self, input_size, num_classes):
        super(cnn_4_relu, self).__init__()
        self.input_size = input_size
        self.Conv1 = torch.nn.Conv2d(1 ,16 ,3, stride=1, padding=1)
        self.Conv2 = torch.nn.Conv2d(16, 8, 3,  stride=1, padding=1)
        self.Conv3 = torch.nn.Conv2d(8, 16, 5, stride=1, padding=1)
        self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
        self.Conv4 = torch.nn.Conv2d(16, 16, 5,  stride=1, padding=1)
        self.prediction_layer = torch.nn.Linear(400, num_classes)
        self.relu = torch.nn.ReLU()
    def forward(self, x):
        x = x.view(-1, 1, 28, 28)
        hidden1 = self.Conv1(x)
        relu1 = self.relu(hidden1)
        hidden2 = self.Conv2(relu1)
        relu2 = self.relu(hidden2)
        hidden3 = self.Conv3(relu2)
```

```
        relu3 = self.relu(hidden3)
        pool1 = self.MaxPool(relu3)
        hidden4 = self.Conv4(pool1)
        pool2 = self.MaxPool(hidden4)
        flattened = pool2.view(batch_size, -1)
        output = self.prediction_layer(flattened)
        return output


# sigmoid cnn_4 classifier
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(1 ,16 ,3, stride=1, padding=1)
        self.Conv2 = torch.nn.Conv2d(16, 8, 3,  stride=1, padding=1)
        self.Conv3 = torch.nn.Conv2d(8, 16, 5, stride=1, padding=1)
        self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
        self.Conv4 = torch.nn.Conv2d(16, 16, 5,  stride=1, padding=1)
        self.prediction_layer = torch.nn.Linear(400, num_classes)
        self.sigmoid = torch.nn.Sigmoid()
    def forward(self, x):
        x = x.view(-1, 1, 28, 28)
        hidden1 = self.Conv1(x)
        sigmoid1 = self.sigmoid(hidden1)
        hidden2 = self.Conv2(sigmoid1)
        sigmoid2 = self.sigmoid(hidden2)
        hidden3 = self.Conv3(sigmoid2)
        sigmoid3 = self.sigmoid(hidden3)
        pool1 = self.MaxPool(sigmoid3)
        hidden4 = self.Conv4(pool1)
        pool2 = self.MaxPool(hidden4)
        flattened = pool2.view(batch_size, -1)
        output = self.prediction_layer(flattened)
        return output




# initialize cnn_4 relu model
model_cnn_4_relu = cnn_4_relu(784 ,10)
# initialize cnn_4 sigmoid model
model_cnn_4_sigmoid = cnn_4_sigmoid(784 ,10)


if torch.cuda.is_available():
    device = torch.device("cuda:0")
    print("CUDA to be used")
else:
    device = "cpu"
    print("CPU to be used.")

model_cnn_4_relu.to(device)
model_cnn_4_sigmoid.to(device)

# create loss: use cross entropy loss
loss = torch.nn.CrossEntropyLoss()

# create optimizer
```

```python
optimizer_relu = torch.optim.SGD(model_cnn_4_relu.parameters(), lr=0.01)
optimizer_sigmoid = torch.optim.SGD(model_cnn_4_sigmoid.parameters(), lr=0.01)


# transfer your model to train mode
model_cnn_4_relu.train()
model_cnn_4_sigmoid.train()

cnn_4_dict = {"name":"cnn_4", "relu_loss_curve": [],  "sigmoid_loss_curve": [], "
    relu_grad_curve": [], "sigmoid_grad_curve": []}



# train the model and save the training loss and validation loss for every 10 batches
    using model eval mode.s
for epoch in tqdm(range(15)):
    for batch, (x, y) in enumerate(train_generator):
        x = x.to(device)
        y = y.to(device)
        # forward pass
        output = model_cnn_4_relu(x)
        # compute loss
        loss_val = loss(output, y)
        # zero gradients
        optimizer_relu.zero_grad()
        # backward pass
        loss_val.backward()
        # optimize
        optimizer_relu.step()
        # print loss
        if batch % 10 == 0:
            model_cnn_4_relu.eval()
            #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
    )
            cnn_4_dict['relu_loss_curve'].append(loss_val.item())
            cnn_4_dict['relu_grad_curve'].append(model_cnn_4_relu.Conv1.weight.grad.mean
    ().item())

            model_cnn_4_relu.train()




# save the model as a pty file
torch.save(model_cnn_4_relu.state_dict(), 'q4_models/model_cnn_4_relu.pty')
print("model saved as 'q4_models/model_cnn_4_relu.pty'")

# train the model and save the training loss and validation loss for every 10 batches
    using model eval mode.s
for epoch in tqdm(range(15)):
    for batch, (x, y) in enumerate(train_generator):
        x = x.to(device)
        y = y.to(device)
        # forward pass
        output = model_cnn_4_sigmoid(x)
        # compute loss
        loss_val = loss(output, y)
        # zero gradients
        optimizer_sigmoid.zero_grad()
```

```
        # backward pass
        loss_val.backward()
        # optimize
        optimizer_sigmoid.step()
        # print loss
        if batch % 10 == 0:
            model_cnn_4_sigmoid.eval()
            #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
    )
            cnn_4_dict['sigmoid_loss_curve'].append(loss_val.item())
            cnn_4_dict['sigmoid_grad_curve'].append(model_cnn_4_sigmoid.Conv1.weight.grad
    .mean().item())

            model_cnn_4_sigmoid.train()




# save the model as a pty file
torch.save(model_cnn_4_sigmoid.state_dict(), 'q4_models/model_cnn_4_sigmoid.pty')
print("model saved as 'q4_models/model_cnn_4_sigmoid.pty'")



# save cnn_4_dict as a pickle file
import pickle
with open('q4_models/part4_cnn_4_dict.pkl', 'wb') as f:
    pickle.dump(cnn_4_dict, f)

# %% [markdown]
# ### CNN5 Activation Function Experiment

# %%
%reset -f

# Load fashion MNIST dataset
import torchvision

from tqdm import tqdm



#training set
train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False,
     transform= torchvision.transforms.ToTensor())

#test set
test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False,
     transform= torchvision.transforms.ToTensor())


batch_size = 50
import torch
# divide training data into training and validation sets of 0.8 and 0.2 respectively

train_data, val_data = torch.utils.data.random_split(train_data, [0.8, 0.2])

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

```python
val_generator = torch.utils.data.DataLoader(val_data, batch_size=batch_size, shuffle=
    False)
test_generator = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=
    False)

# relu cnn_5 classifier
class cnn_5_relu(torch.nn.Module):
    def __init__(self, input_size, num_classes):
        super(cnn_5_relu, self).__init__()
        self.input_size = input_size
        self.Conv1 = torch.nn.Conv2d(1 ,8 ,3, stride=1, padding=1)
        self.Conv2 = torch.nn.Conv2d(8, 16, 3,  stride=1, padding=1)
        self.Conv3 = torch.nn.Conv2d(16, 8, 3, stride=1, padding=1)
        self.Conv4 = torch.nn.Conv2d(8, 16, 3,  stride=1, padding=1)
        self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
        self.Conv5 = torch.nn.Conv2d(16, 16, 3,  stride=1, padding=1)
        self.Conv6 = torch.nn.Conv2d(16, 8, 3,  stride=1, padding=1)
        self.prediction_layer = torch.nn.Linear(392, num_classes)
        self.relu = torch.nn.ReLU()
    def forward(self, x):
        x = x.view(-1, 1, 28, 28)
        hidden1 = self.Conv1(x)
        relu1 = self.relu(hidden1)
        hidden2 = self.Conv2(relu1)
        relu2 = self.relu(hidden2)
        hidden3 = self.Conv3(relu2)
        relu3 = self.relu(hidden3)
        hidden4 = self.Conv4(relu3)
        relu4 = self.relu(hidden4)
        pool1 = self.MaxPool(relu4)
        hidden5 = self.Conv5(pool1)
        relu5 = self.relu(hidden5)
        hidden6 = self.Conv6(relu5)
        relu6 = self.relu(hidden6)
        pool2 = self.MaxPool(relu6)
        flattened = pool2.view(batch_size, -1)
        output = self.prediction_layer(flattened)
        return output

# sigmoid cnn_5 classifier
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(1 ,8 ,3, stride=1, padding=1)
        self.Conv2 = torch.nn.Conv2d(8, 16, 3,  stride=1, padding=1)
        self.Conv3 = torch.nn.Conv2d(16, 8, 3, stride=1, padding=1)
        self.Conv4 = torch.nn.Conv2d(8, 16, 3,  stride=1, padding=1)
        self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
        self.Conv5 = torch.nn.Conv2d(16, 16, 3,  stride=1, padding=1)
        self.Conv6 = torch.nn.Conv2d(16, 8, 3,  stride=1, padding=1)
        self.prediction_layer = torch.nn.Linear(392, num_classes)
        self.sigmoid = torch.nn.Sigmoid()
    def forward(self, x):
        x = x.view(-1, 1, 28, 28)
        hidden1 = self.Conv1(x)
        sigmoid1 = self.sigmoid(hidden1)
        hidden2 = self.Conv2(sigmoid1)
        sigmoid2 = self.sigmoid(hidden2)
```

```
        hidden3 = self.Conv3(sigmoid2)
        sigmoid3 = self.sigmoid(hidden3)
        hidden4 = self.Conv4(sigmoid3)
        sigmoid4 = self.sigmoid(hidden4)
        pool1 = self.MaxPool(sigmoid4)
        hidden5 = self.Conv5(pool1)
        sigmoid5 = self.sigmoid(hidden5)
        hidden6 = self.Conv6(sigmoid5)
        sigmoid6 = self.sigmoid(hidden6)
        pool2 = self.MaxPool(sigmoid6)
        flattened = pool2.view(batch_size, -1)
        output = self.prediction_layer(flattened)
        return output



# initialize cnn_5 relu model
model_cnn_5_relu = cnn_5_relu(784 ,10)
# initialize cnn_5 sigmoid model
model_cnn_5_sigmoid = cnn_5_sigmoid(784 ,10)


if torch.cuda.is_available():
    device = torch.device("cuda:0")
    print("CUDA to be used")
else:
    device = "cpu"
    print("CPU to be used.")

model_cnn_5_relu.to(device)
model_cnn_5_sigmoid.to(device)

# create loss: use cross entropy loss
loss = torch.nn.CrossEntropyLoss()

# create optimizer
optimizer_relu = torch.optim.SGD(model_cnn_5_relu.parameters(), lr=0.01)
optimizer_sigmoid = torch.optim.SGD(model_cnn_5_sigmoid.parameters(), lr=0.01)


# transfer your model to train mode
model_cnn_5_relu.train()
model_cnn_5_sigmoid.train()

cnn_5_dict = {"name":"cnn_5", "relu_loss_curve": [],  "sigmoid_loss_curve": [], "
    relu_grad_curve": [], "sigmoid_grad_curve": []}



# train the model and save the training loss and validation loss for every 10 batches
    using model eval mode.s
for epoch in tqdm(range(15)):
    for batch, (x, y) in enumerate(train_generator):
        x = x.to(device)
        y = y.to(device)
        # forward pass
        output = model_cnn_5_relu(x)
        # compute loss
        loss_val = loss(output, y)
```

```
        # zero gradients
        optimizer_relu.zero_grad()
        # backward pass
        loss_val.backward()
        # optimize
        optimizer_relu.step()
        # print loss
        if batch % 10 == 0:
            model_cnn_5_relu.eval()
            #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item()))
            cnn_5_dict['relu_loss_curve'].append(loss_val.item())
            cnn_5_dict['relu_grad_curve'].append(model_cnn_5_relu.Conv1.weight.grad.mean().item())

            model_cnn_5_relu.train()




# save the model as a pty file
torch.save(model_cnn_5_relu.state_dict(), 'q4_models/model_cnn_5_relu.pty')
print("model saved as 'q4_models/model_cnn_5_relu.pty'")

# train the model and save the training loss and validation loss for every 10 batches using model eval mode.s
for epoch in tqdm(range(15)):
    for batch, (x, y) in enumerate(train_generator):
        x = x.to(device)
        y = y.to(device)
        # forward pass
        output = model_cnn_5_sigmoid(x)
        # compute loss
        loss_val = loss(output, y)
        # zero gradients
        optimizer_sigmoid.zero_grad()
        # backward pass
        loss_val.backward()
        # optimize
        optimizer_sigmoid.step()
        # print loss
        if batch % 10 == 0:
            model_cnn_5_sigmoid.eval()
            #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item()))
            cnn_5_dict['sigmoid_loss_curve'].append(loss_val.item())
            cnn_5_dict['sigmoid_grad_curve'].append(model_cnn_5_sigmoid.Conv1.weight.grad.mean().item())

            model_cnn_5_sigmoid.train()




# save the model as a pty file
torch.save(model_cnn_5_sigmoid.state_dict(), 'q4_models/model_cnn_5_sigmoid.pty')
print("model saved as 'q4_models/model_cnn_5_sigmoid.pty'")
```

```

# save cnn_5_dict as a pickle file
import pickle
with open('q4_models/part4_cnn_5_dict.pkl', 'wb') as f:
    pickle.dump(cnn_5_dict, f)

# %% [markdown]
# ## Part 4 plotting part.

# %%
%reset -f
from utils import part4Plots
import pickle

# load the pickle files
with open('q4_models/part4_mlp_1_dict.pkl', 'rb') as f:
    mlp_1_dict = pickle.load(f)

with open('q4_models/part4_mlp_2_dict.pkl', 'rb') as f:
    mlp_2_dict = pickle.load(f)

with open('q4_models/part4_cnn_3_dict.pkl', 'rb') as f:
    cnn_3_dict = pickle.load(f)

with open('q4_models/part4_cnn_4_dict.pkl', 'rb') as f:
    cnn_4_dict = pickle.load(f)

with open('q4_models/part4_cnn_5_dict.pkl', 'rb') as f:
    cnn_5_dict = pickle.load(f)
results = [mlp_1_dict]

part4Plots(results, "q4_models", "part4_alpha_mlp_1")

results = [mlp_2_dict]

part4Plots(results, "q4_models", "part4_alpha_mlp_2")

results = [cnn_3_dict]

part4Plots(results, "q4_models", "part4_alpha_cnn_3")

results = [cnn_4_dict]

part4Plots(results, "q4_models", "part4_alpha_cnn_4")

results = [cnn_5_dict]

part4Plots(results, "q4_models", "part4_alpha_cnn_5")


# %% [markdown]
# ## Experimenting Learning Rate
#
#  My favorite architecture is CNN3.
#
#

# %%
%reset -f
```

```python

# Load fashion MNIST dataset
import torchvision

from tqdm import tqdm

#training set
train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False, transform= torchvision.transforms.ToTensor())

#test set
test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False, transform= torchvision.transforms.ToTensor())

batch_size = 50
import torch
# divide training data into training and validation sets of 0.9 and 0.1 respectively

train_data, val_data = torch.utils.data.random_split(train_data, [0.9, 0.1])

train_generator = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True)
val_generator = torch.utils.data.DataLoader(val_data, batch_size=batch_size, shuffle=False)
test_generator = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=False)



# cnn_3 classifier
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(1 ,16 ,3, stride=1, padding=1)
        self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
        self.Conv2 = torch.nn.Conv2d(16, 8, 5,  stride=1, padding=1)
        self.Conv3 = torch.nn.Conv2d(8, 16, 7, stride=1, padding=1)
        self.prediction_layer = torch.nn.Linear(1296, num_classes)
        self.relu = torch.nn.ReLU()
    def forward(self, x):
        x = x.view(-1, 1, 28, 28)
        hidden1 = self.Conv1(x)
        relu1 = self.relu(hidden1)
        hidden2 = self.Conv2(relu1)
        relu2 = self.relu(hidden2)
        pool = self.MaxPool(relu2)
        hidden3 = self.Conv3(pool)
        flattened = hidden3.view(batch_size, -1)
        output = self.prediction_layer(flattened)
        return output


# initialize cnn_5 relu model
model_cnn_3 = cnn_3(784 ,10)


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

```python
    print("CUDA to be used")
else:
    device = "cpu"
    print("CPU to be used.")

model_cnn_3.to(device)


# create loss: use cross entropy loss
loss = torch.nn.CrossEntropyLoss()

# create optimizer
optimizer = torch.optim.SGD(model_cnn_3.parameters(), lr=0.1)


# transfer your model to train mode
model_cnn_3.train()


cnn_3_dict = {"name":"cnn_3", "loss_curve_1": [],  "loss_curve_01": [], "loss_curve_001":
     [], "val_acc_curve_1": [], "val_acc_curve_01": [], "val_acc_curve_001": []}



# train the model and save the training loss and validation loss for every 10 batches
    using model eval mode.s
for epoch in tqdm(range(20)):
    for batch, (x, y) in enumerate(train_generator):

        x = x.to(device)
        y = y.to(device)

        # forward pass
        output = model_cnn_3(x)
        # compute loss
        loss_val = loss(output, y)
        # zero gradients
        optimizer.zero_grad()
        # backward pass
        loss_val.backward()
        # optimize
        optimizer.step()
        # print loss
        if batch % 10 == 0:
            model_cnn_3.eval()
            #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
    )
            cnn_3_dict['loss_curve_1'].append(loss_val.item())

            # validation loss
            validation_accuracy_per_batch = torch.tensor([])
            with torch.no_grad():
                for val_x, val_y in val_generator:

                    val_x = val_x.to(device)
                    val_y = val_y.to(device)

                    val_output = model_cnn_3(val_x)
                    #val_loss = loss(val_output, val_y)
```

```
                    val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).float())
                    try:
                        validation_accuracy_per_batch = torch.cat((validation_accuracy_per_batch ,val_accuracy))
                    except:
                        validation_accuracy_per_batch = val_accuracy
            #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean()))
            cnn_3_dict['val_acc_curve_1'].append(validation_accuracy_per_batch.mean().item())
            model_cnn_3.train()




# initialize cnn_5 relu model
model_cnn_3 = cnn_3(784 ,10)

model_cnn_3.to(device)


# create loss: use cross entropy loss
loss = torch.nn.CrossEntropyLoss()

# create optimizer
optimizer = torch.optim.SGD(model_cnn_3.parameters(), lr=0.01)


# transfer your model to train mode
model_cnn_3.train()

# train the model and save the training loss and validation loss for every 10 batches using model eval mode.s
for epoch in tqdm(range(20)):
    for batch, (x, y) in enumerate(train_generator):

        x = x.to(device)
        y = y.to(device)

        # forward pass
        output = model_cnn_3(x)
        # compute loss
        loss_val = loss(output, y)
        # zero gradients
        optimizer.zero_grad()
        # backward pass
        loss_val.backward()
        # optimize
        optimizer.step()
        # print loss
        if batch % 10 == 0:
            model_cnn_3.eval()
            #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item()))
            cnn_3_dict['loss_curve_01'].append(loss_val.item())

            # validation loss
            validation_accuracy_per_batch = torch.tensor([])
```

```
            with torch.no_grad():
                for val_x, val_y in val_generator:

                    val_x = val_x.to(device)
                    val_y = val_y.to(device)

                    val_output = model_cnn_3(val_x)
                    #val_loss = loss(val_output, val_y)
                    val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).float())
                    try:
                        validation_accuracy_per_batch = torch.cat((validation_accuracy_per_batch ,val_accuracy))
                    except:
                        validation_accuracy_per_batch = val_accuracy
            #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean()))
            cnn_3_dict['val_acc_curve_01'].append(validation_accuracy_per_batch.mean().item())
            model_cnn_3.train()




# initialize cnn_5 relu model
model_cnn_3 = cnn_3(784 ,10)

model_cnn_3.to(device)


# create loss: use cross entropy loss
loss = torch.nn.CrossEntropyLoss()

# create optimizer
optimizer = torch.optim.SGD(model_cnn_3.parameters(), lr=0.001)


# transfer your model to train mode
model_cnn_3.train()

# train the model and save the training loss and validation loss for every 10 batches using model eval mode.s
for epoch in tqdm(range(20)):
    for batch, (x, y) in enumerate(train_generator):

        x = x.to(device)
        y = y.to(device)

        # forward pass
        output = model_cnn_3(x)
        # compute loss
        loss_val = loss(output, y)
        # zero gradients
        optimizer.zero_grad()
        # backward pass
        loss_val.backward()
        # optimize
        optimizer.step()
        # print loss
```

```
        if batch % 10 == 0:
            model_cnn_3.eval()
            #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item()))
            cnn_3_dict['loss_curve_001'].append(loss_val.item())

            # validation loss
            validation_accuracy_per_batch = torch.tensor([])
            with torch.no_grad():
                for val_x, val_y in val_generator:

                    val_x = val_x.to(device)
                    val_y = val_y.to(device)

                    val_output = model_cnn_3(val_x)
                    #val_loss = loss(val_output, val_y)
                    val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).float())
                    try:
                        validation_accuracy_per_batch = torch.cat((validation_accuracy_per_batch ,val_accuracy))
                    except:
                        validation_accuracy_per_batch = val_accuracy
            #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean()))
            cnn_3_dict['val_acc_curve_001'].append(validation_accuracy_per_batch.mean().item())
            model_cnn_3.train()



# save cnn_3_dict as a pickle file
import pickle
with open('q5_models/part5_cnn_3_dict.pkl', 'wb') as f:
    pickle.dump(cnn_3_dict, f)

# %%
from utils import part5Plots
import pickle

# load the pickle files
with open('q5_models/part5_cnn_3_dict.pkl', 'rb') as f:
    cnn_3_dict = pickle.load(f)

part5Plots(cnn_3_dict, "q5_models")
print(len(cnn_3_dict['val_acc_curve_1']))

# %%
%reset -f

# Load fashion MNIST dataset
import torchvision

from tqdm import tqdm

#training set
train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False, transform= torchvision.transforms.ToTensor())

```

```python
#test set
test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False, transform= torchvision.transforms.ToTensor())

batch_size = 50
import torch
# divide training data into training and validation sets of 0.9 and 0.1 respectively

train_data, val_data = torch.utils.data.random_split(train_data, [0.9, 0.1])

train_generator = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True)
val_generator = torch.utils.data.DataLoader(val_data, batch_size=batch_size, shuffle=True)
test_generator = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=False)



# cnn_3 classifier
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(1 ,16 ,3, stride=1, padding=1)
        self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
        self.Conv2 = torch.nn.Conv2d(16, 8, 5,  stride=1, padding=1)
        self.Conv3 = torch.nn.Conv2d(8, 16, 7, stride=1, padding=1)
        self.prediction_layer = torch.nn.Linear(1296, num_classes)
        self.relu = torch.nn.ReLU()
    def forward(self, x):
        x = x.view(-1, 1, 28, 28)
        hidden1 = self.Conv1(x)
        relu1 = self.relu(hidden1)
        hidden2 = self.Conv2(relu1)
        relu2 = self.relu(hidden2)
        pool = self.MaxPool(relu2)
        hidden3 = self.Conv3(pool)
        flattened = hidden3.view(batch_size, -1)
        output = self.prediction_layer(flattened)
        return output


# initialize cnn_5 relu model
model_cnn_3 = cnn_3(784 ,10)


if torch.cuda.is_available():
    device = torch.device("cuda:0")
    print("CUDA to be used")
else:
    device = "cpu"
    print("CPU to be used.")

model_cnn_3.to(device)


# create loss: use cross entropy loss
loss = torch.nn.CrossEntropyLoss()
```

```python

# create optimizer
optimizer = torch.optim.SGD(model_cnn_3.parameters(), lr=0.1)


# transfer your model to train mode
model_cnn_3.train()


cnn_3_dict = {"name":"cnn_3_scheduled", "loss_curve": [],  "val_acc_curve": []}



# train the model and save the training loss and validation loss for every 10 batches using model eval mode.s
for epoch in tqdm(range(30)):
    #decrease learning rate to 0.01 after 7 epochs
    if epoch == 8:
        optimizer = torch.optim.SGD(model_cnn_3.parameters(), lr=0.01)
    for batch, (x, y) in enumerate(train_generator):

        x = x.to(device)
        y = y.to(device)

        # forward pass
        output = model_cnn_3(x)
        # compute loss
        loss_val = loss(output, y)
        # zero gradients
        optimizer.zero_grad()
        # backward pass
        loss_val.backward()
        # optimize
        optimizer.step()
        # print loss
        if batch % 10 == 0:
            model_cnn_3.eval()
            #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item()))
            cnn_3_dict['loss_curve'].append(loss_val.item())

            # validation loss
            validation_accuracy_per_batch = torch.tensor([])
            with torch.no_grad():
                for val_x, val_y in val_generator:

                    val_x = val_x.to(device)
                    val_y = val_y.to(device)

                    val_output = model_cnn_3(val_x)
                    #val_loss = loss(val_output, val_y)
                    val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).float())
                    try:
                        validation_accuracy_per_batch = torch.cat((validation_accuracy_per_batch ,val_accuracy))
                    except:
                        validation_accuracy_per_batch = val_accuracy
```

```
            #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean())
    )
            cnn_3_dict['val_acc_curve'].append(validation_accuracy_per_batch.mean().item
    ())
            model_cnn_3.train()

    # #decrease learning rate to 0.001 after 15 epochs
    if epoch == 18:
        optimizer = torch.optim.SGD(model_cnn_3.parameters(), lr=0.001)

# plot validation accuracy curve where the x axis is epoch
import matplotlib.pyplot as plt
import numpy as np
x_axis = np.linspace(0, 30, len(cnn_3_dict['val_acc_curve']))
plt.plot(x_axis , cnn_3_dict['val_acc_curve'])
plt.title("Validation Accuracy Curve learning rate scheduling (rate drops at 7th and 14th
     epoch)")
plt.xlabel("Epoch")
plt.ylabel("Validation Accuracy")
plt.show()
plt.savefig("q5_models/part5_cnn_3_scheduled_3.png")

# %%


%reset -f

# Load fashion MNIST dataset
import torchvision

from tqdm import tqdm

#training set
train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False,
     transform= torchvision.transforms.ToTensor())

#test set
test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False,
     transform= torchvision.transforms.ToTensor())

batch_size = 50
import torch
# divide training data into training and validation sets of 0.9 and 0.1 respectively

train_data, val_data = torch.utils.data.random_split(train_data, [0.9, 0.1])

train_generator = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=
    True)
val_generator = torch.utils.data.DataLoader(val_data, batch_size=batch_size, shuffle=True
    )
test_generator = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=
    False)



# cnn_3 classifier
class cnn_3(torch.nn.Module):
    def __init__(self, input_size, num_classes):
        super(cnn_3, self).__init__()
```

```python
        self.input_size = input_size
        self.Conv1 = torch.nn.Conv2d(1 ,16 ,3, stride=1, padding=1)
        self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
        self.Conv2 = torch.nn.Conv2d(16, 8, 5,  stride=1, padding=1)
        self.Conv3 = torch.nn.Conv2d(8, 16, 7, stride=1, padding=1)
        self.prediction_layer = torch.nn.Linear(1296, num_classes)
        self.relu = torch.nn.ReLU()
    def forward(self, x):
        x = x.view(-1, 1, 28, 28)
        hidden1 = self.Conv1(x)
        relu1 = self.relu(hidden1)
        hidden2 = self.Conv2(relu1)
        relu2 = self.relu(hidden2)
        pool = self.MaxPool(relu2)
        hidden3 = self.Conv3(pool)
        flattened = hidden3.view(batch_size, -1)
        output = self.prediction_layer(flattened)
        return output


# initialize cnn_5 relu model
model_cnn_3 = cnn_3(784 ,10)


if torch.cuda.is_available():
    device = torch.device("cuda:0")
    print("CUDA to be used")
else:
    device = "cpu"
    print("CPU to be used.")

model_cnn_3.to(device)


# create loss: use cross entropy loss
loss = torch.nn.CrossEntropyLoss()

# create optimizer
optimizer = torch.optim.SGD(model_cnn_3.parameters(), lr=0.1)


# transfer your model to train mode
model_cnn_3.train()


cnn_3_dict = {"name":"cnn_3_scheduled", "loss_curve": [],  "val_acc_curve": []}



# train the model and save the training loss and validation loss for every 10 batches
    using model eval mode.s
for epoch in tqdm(range(7)):
    # #decrease learning rate to 0.01 after 7 epochs
    # if epoch == 8:
    #     optimizer = torch.optim.SGD(model_cnn_3.parameters(), lr=0.01)
    for batch, (x, y) in enumerate(train_generator):

        x = x.to(device)
        y = y.to(device)
```

```

        # forward pass
        output = model_cnn_3(x)
        # compute loss
        loss_val = loss(output, y)
        # zero gradients
        optimizer.zero_grad()
        # backward pass
        loss_val.backward()
        # optimize
        optimizer.step()
        # print loss
        if batch % 10 == 0:
            model_cnn_3.eval()
            #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
    )
            cnn_3_dict['loss_curve'].append(loss_val.item())

            # validation loss
            validation_accuracy_per_batch = torch.tensor([])
            with torch.no_grad():
                for val_x, val_y in val_generator:

                    val_x = val_x.to(device)
                    val_y = val_y.to(device)

                    val_output = model_cnn_3(val_x)
                    #val_loss = loss(val_output, val_y)
                    val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).
    float())
                    try:
                        validation_accuracy_per_batch = torch.cat((
    validation_accuracy_per_batch ,val_accuracy))
                    except:
                        validation_accuracy_per_batch = val_accuracy
            #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean())
    )
            cnn_3_dict['val_acc_curve'].append(validation_accuracy_per_batch.mean().item
    ())
            model_cnn_3.train()

for g in optimizer.param_groups:
    g['lr'] = 0.001

# transfer your model to train mode
model_cnn_3.train()

# train the model and save the training loss and validation loss for every 10 batches
    using model eval mode.s
for epoch in tqdm(range(23)):
    for batch, (x, y) in enumerate(train_generator):

        x = x.to(device)
        y = y.to(device)

        # forward pass
        output = model_cnn_3(x)
        # compute loss
        loss_val = loss(output, y)
```

```
        # zero gradients
        optimizer.zero_grad()
        # backward pass
        loss_val.backward()
        # optimize
        optimizer.step()
        # print loss
        if batch % 10 == 0:
            model_cnn_3.eval()
            #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item()))
            cnn_3_dict['loss_curve'].append(loss_val.item())

            # validation loss
            validation_accuracy_per_batch = torch.tensor([])
            with torch.no_grad():
                for val_x, val_y in val_generator:

                    val_x = val_x.to(device)
                    val_y = val_y.to(device)

                    val_output = model_cnn_3(val_x)
                    #val_loss = loss(val_output, val_y)
                    val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).float())
                    try:
                        validation_accuracy_per_batch = torch.cat((validation_accuracy_per_batch ,val_accuracy))
                    except:
                        validation_accuracy_per_batch = val_accuracy
            #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean()))
            cnn_3_dict['val_acc_curve'].append(validation_accuracy_per_batch.mean().item())
            model_cnn_3.train()



# plot validation accuracy curve where the x axis is epoch
import matplotlib.pyplot as plt
import numpy as np
x_axis = np.linspace(0, 30, len(cnn_3_dict['val_acc_curve']))
plt.plot(x_axis , cnn_3_dict['val_acc_curve'])
plt.title("Validation Accuracy Curve learning rate scheduling (rate drops at 7th epoch)")
plt.xlabel("Epoch")
plt.ylabel("Validation Accuracy")
plt.show()
plt.savefig("q5_models/part5_cnn_3_scheduled_2.png")

# %%
%reset -f

# Load fashion MNIST dataset
import torchvision

from tqdm import tqdm

#training set
```

```python
train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False,
     transform= torchvision.transforms.ToTensor())

#test set
test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False,
     transform= torchvision.transforms.ToTensor())

batch_size = 50
import torch
# divide training data into training and validation sets of 0.9 and 0.1 respectively

train_data, val_data = torch.utils.data.random_split(train_data, [0.9, 0.1])

train_generator = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=
    True)
val_generator = torch.utils.data.DataLoader(val_data, batch_size=batch_size, shuffle=True
    )
test_generator = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=
    False)



# cnn_3 classifier
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(1 ,16 ,3, stride=1, padding=1)
        self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
        self.Conv2 = torch.nn.Conv2d(16, 8, 5,  stride=1, padding=1)
        self.Conv3 = torch.nn.Conv2d(8, 16, 7, stride=1, padding=1)
        self.prediction_layer = torch.nn.Linear(1296, num_classes)
        self.relu = torch.nn.ReLU()
    def forward(self, x):
        x = x.view(-1, 1, 28, 28)
        hidden1 = self.Conv1(x)
        relu1 = self.relu(hidden1)
        hidden2 = self.Conv2(relu1)
        relu2 = self.relu(hidden2)
        pool = self.MaxPool(relu2)
        hidden3 = self.Conv3(pool)
        flattened = hidden3.view(batch_size, -1)
        output = self.prediction_layer(flattened)
        return output


# initialize cnn_5 relu model
model_cnn_3 = cnn_3(784 ,10)


if torch.cuda.is_available():
    device = torch.device("cuda:0")
    print("CUDA to be used")
else:
    device = "cpu"
    print("CPU to be used.")

model_cnn_3.to(device)

```

```python

# create loss: use cross entropy loss
loss = torch.nn.CrossEntropyLoss()

# create optimizer
optimizer = torch.optim.SGD(model_cnn_3.parameters(), lr=0.1)


# transfer your model to train mode
model_cnn_3.train()


cnn_3_dict = {"name":"cnn_3_scheduled", "loss_curve": [],  "val_acc_curve": []}



# train the model and save the training loss and validation loss for every 10 batches using model eval mode.s
for epoch in tqdm(range(30)):
    #decrease learning rate to 0.01 after 7 epochs
    # if epoch == 8:
    #     optimizer = torch.optim.SGD(model_cnn_3.parameters(), lr=0.01)
    for batch, (x, y) in enumerate(train_generator):

        x = x.to(device)
        y = y.to(device)

        # forward pass
        output = model_cnn_3(x)
        # compute loss
        loss_val = loss(output, y)
        # zero gradients
        optimizer.zero_grad()
        # backward pass
        loss_val.backward()
        # optimize
        optimizer.step()
        # print loss
        if batch % 10 == 0:
            model_cnn_3.eval()
            #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item()))
            cnn_3_dict['loss_curve'].append(loss_val.item())

            # validation loss
            validation_accuracy_per_batch = torch.tensor([])
            with torch.no_grad():
                for val_x, val_y in val_generator:

                    val_x = val_x.to(device)
                    val_y = val_y.to(device)

                    val_output = model_cnn_3(val_x)
                    #val_loss = loss(val_output, val_y)
                    val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).float())
                    try:
                        validation_accuracy_per_batch = torch.cat((validation_accuracy_per_batch ,val_accuracy))
```

```
                    except:
                        validation_accuracy_per_batch = val_accuracy
            #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean()))
            cnn_3_dict['val_acc_curve'].append(validation_accuracy_per_batch.mean().item())
            model_cnn_3.train()

    # # #decrease learning rate to 0.001 after 15 epochs
    # if epoch == 18:
    #     optimizer = torch.optim.SGD(model_cnn_3.parameters(), lr=0.001)

# plot validation accuracy curve where the x axis is epoch
import matplotlib.pyplot as plt
import numpy as np
x_axis = np.linspace(0, 30, len(cnn_3_dict['val_acc_curve']))
plt.plot(x_axis , cnn_3_dict['val_acc_curve'])
plt.title("Validation Accuracy Curve learning rate scheduling (no learning rate drop)")
plt.xlabel("Epoch")
plt.ylabel("Validation Accuracy")
plt.show()
plt.savefig("q5_models/part5_cnn_3_scheduled_1.png")
```

```
import numpy as np
import matplotlib.pyplot as plt


# load the kernel from
kernel=np.load('hw1/data/kernel.npy')

input=np.load('hw1/data/samples_0.npy')

# plot kernels on the same figure there are eight kernels in total
plt.figure()
for i in range(8):
    plt.subplot(2,4,i+1)
    plt.imshow(kernel[i,0,:,:],cmap='gray')
    plt.title('kernel'+str(i+1))

plt.show()

# plot input

plt.figure()
for i in range(5):
    plt.subplot(1,5,i+1)
    plt.imshow(input[i,0,:,:],cmap='gray')
    plt.title('input'+str(i+1))

plt.show()
```

```
import torchinfo

import torch

# example mlp classifier
class mlp_1(torch.nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(mlp_1, self).__init__()
```

```python
        self.input_size = input_size
        self.FC = torch.nn.Linear(input_size, hidden_size)
        self.prediction_layer = 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.FC(x)
        relu = self.relu(hidden)
        output = self.prediction_layer(relu)
        return output

# initialize your model
model_mlp_1 = mlp_1(784,32,10)


torchinfo.summary(model_mlp_1, input_size=(96, 784))

# Total params: 25,450
# Trainable params: 25,450





# example mlp classifier
class mlp_2(torch.nn.Module):
    def __init__(self, input_size, hidden_size_1, hidden_size_2, num_classes):
        super(mlp_2, self).__init__()
        self.input_size = input_size
        self.FC1 = torch.nn.Linear(input_size, hidden_size_1)
        self.FC2 = torch.nn.Linear(hidden_size_1, hidden_size_2, bias=False)
        self.prediction_layer = torch.nn.Linear(hidden_size_2, num_classes)
        self.relu = torch.nn.ReLU()
    def forward(self, x):
        x = x.view(-1, self.input_size)
        hidden1 = self.FC1(x)
        relu = self.relu(hidden1)
        hidden2 = self.FC2(relu)
        output = self.prediction_layer(hidden2)
        return output

# initialize your model
model_mlp_2 = mlp_2(784 ,32 ,64 ,10)

torchinfo.summary(model_mlp_2, input_size=(96, 784))


# Total params: 27,818
# Trainable params: 27,818


# example cnn_3 classifier
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(1 ,16 ,3, stride=1, padding=1)
        self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
        self.Conv2 = torch.nn.Conv2d(16, 8, 5,  stride=1, padding=1)
```

```
        self.Conv3 = torch.nn.Conv2d(8, 16, 7, stride=1, padding=1)
        self.prediction_layer = torch.nn.Linear(1296, num_classes)
        self.relu = torch.nn.ReLU()
    def forward(self, x):
        x = x.view(-1, 1, 28, 28)
        hidden1 = self.Conv1(x)
        relu1 = self.relu(hidden1)
        hidden2 = self.Conv2(relu1)
        relu2 = self.relu(hidden2)
        pool = self.MaxPool(relu2)
        hidden3 = self.Conv3(pool)
        flattened = hidden3.view(96, -1)
        output = self.prediction_layer(flattened)
        return output

# initialize your model
model_cnn_3 = cnn_3(784 ,10)

torchinfo.summary(model_cnn_3, input_size=(96, 784))

# Total params: 22,626
# Trainable params: 22,626


# example cnn_4 classifier
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(1 ,16 ,3, stride=1, padding=1)
        self.Conv2 = torch.nn.Conv2d(16, 8, 3,  stride=1, padding=1)
        self.Conv3 = torch.nn.Conv2d(8, 16, 5, stride=1, padding=1)
        self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
        self.Conv4 = torch.nn.Conv2d(16, 16, 5,  stride=1, padding=1)
        self.prediction_layer = torch.nn.Linear(400, num_classes)
        self.relu = torch.nn.ReLU()
    def forward(self, x):
        x = x.view(-1, 1, 28, 28)
        hidden1 = self.Conv1(x)
        relu1 = self.relu(hidden1)
        hidden2 = self.Conv2(relu1)
        relu2 = self.relu(hidden2)
        hidden3 = self.Conv3(relu2)
        relu3 = self.relu(hidden3)
        pool1 = self.MaxPool(relu3)
        hidden4 = self.Conv4(pool1)
        pool2 = self.MaxPool(hidden4)
        flattened = pool2.view(96, -1)
        output = self.prediction_layer(flattened)
        return output

# initialize your model
model_cnn_4 = cnn_4(784 ,10)

torchinfo.summary(model_cnn_4, input_size=(96, 784))

# Total params: 14,962
# Trainable params: 14,962

```

```python
# example cnn_5 classifier
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(1 ,8 ,3, stride=1, padding=1)
        self.Conv2 = torch.nn.Conv2d(8, 16, 3,  stride=1, padding=1)
        self.Conv3 = torch.nn.Conv2d(16, 8, 3, stride=1, padding=1)
        self.Conv4 = torch.nn.Conv2d(8, 16, 3,  stride=1, padding=1)
        self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
        self.Conv5 = torch.nn.Conv2d(16, 16, 3,  stride=1, padding=1)
        self.Conv6 = torch.nn.Conv2d(16, 8, 3,  stride=1, padding=1)
        self.prediction_layer = torch.nn.Linear(392, num_classes)
        self.relu = torch.nn.ReLU()
    def forward(self, x):
        x = x.view(-1, 1, 28, 28)
        hidden1 = self.Conv1(x)
        relu1 = self.relu(hidden1)
        hidden2 = self.Conv2(relu1)
        relu2 = self.relu(hidden2)
        hidden3 = self.Conv3(relu2)
        relu3 = self.relu(hidden3)
        hidden4 = self.Conv4(relu3)
        relu4 = self.relu(hidden4)
        pool1 = self.MaxPool(relu4)
        hidden5 = self.Conv5(pool1)
        relu5 = self.relu(hidden5)
        hidden6 = self.Conv6(relu5)
        relu6 = self.relu(hidden6)
        pool2 = self.MaxPool(relu6)
        flattened = pool2.view(96, -1)
        output = self.prediction_layer(flattened)
        return output

# initialize your model
model_cnn_5 = cnn_5(784 ,10)


torchinfo.summary(model_cnn_5, input_size=(96, 784))
# Total params: 10,986
# Trainable params: 10,986
```

Submitted by Ahmet Akman 2442366 on April 7, 2024.