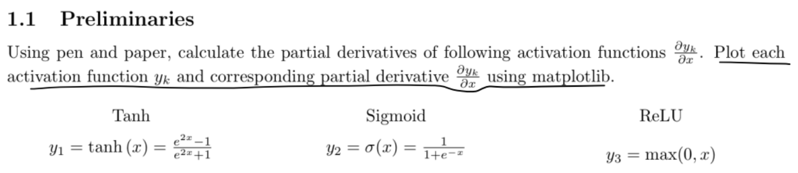
EE449 HW1

Defne Odabaşı-2443604

1. Basic Neural Network Construction and Training

1.1 Implementation

1.1.1 Calculation of partial derivatives by pen and paper



1.1 Preliminaries

Using pen and paper, calculate the partial derivatives of following activation functions $\frac{\partial y_k}{\partial x}$. Plot each activation function y_k and corresponding partial derivative $\frac{\partial y_k}{\partial x}$ using matplotlib.

Tanh	Sigmoid	ReLU
$y_1 = \tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$	$y_2 = \sigma(x) = \frac{1}{1+e^{-x}}$	$y_3 = \max(0, x)$

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

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

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

$(e^{2x}+1)$

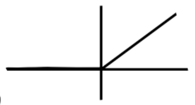
$$= 1 - (\tanh(x))^2$$

2) $y_2 = \sigma(x) = \frac{1}{1+e^{-x}}$

$$\frac{\partial y_2}{\partial x} = \frac{\partial(\sigma(x))}{\partial x} = (\cancel{-1})(1+e^{-x})^{-2}(e^{-x})(\cancel{-1}) = \frac{e^{-x}}{(1+e^{-x})^2}$$

$$= (1-\sigma(x))(\sigma(x))$$

3) $y_3 = \max(0, x)$



$$\frac{\partial y_3}{\partial x} = \begin{cases} 0, & x<0 \\ 1, & x>0 \end{cases}$$

1.1.2 Plot of each activation function using matplotlib

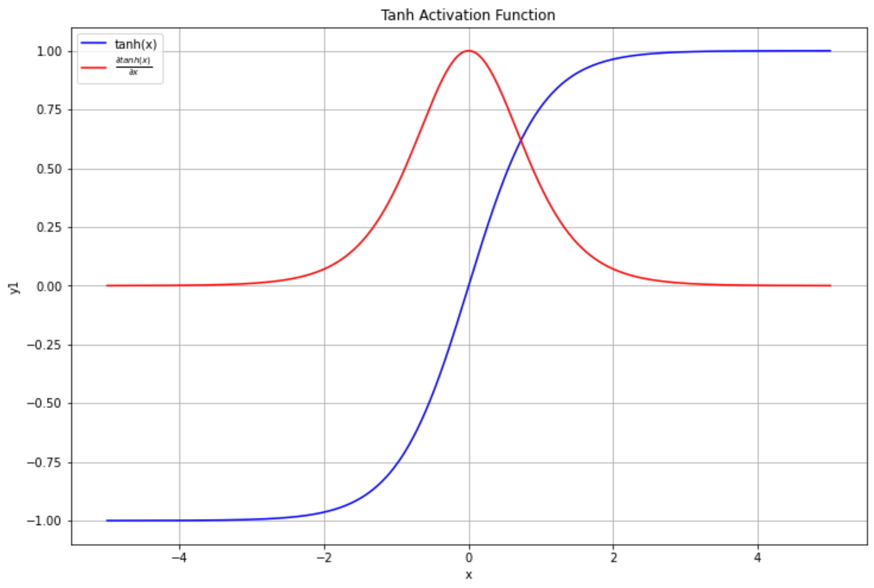


Figure 1 Tanh activation

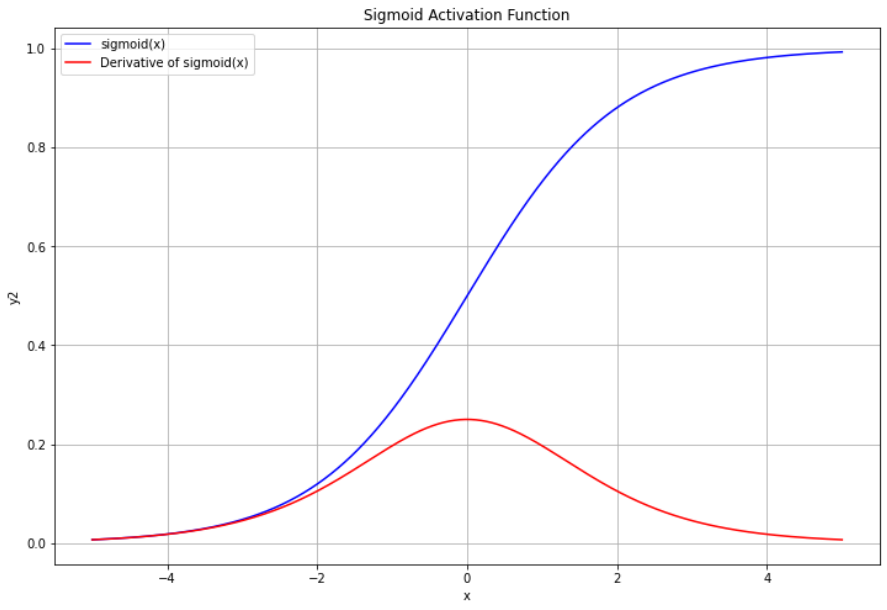


Figure 2 Sigmoid activation

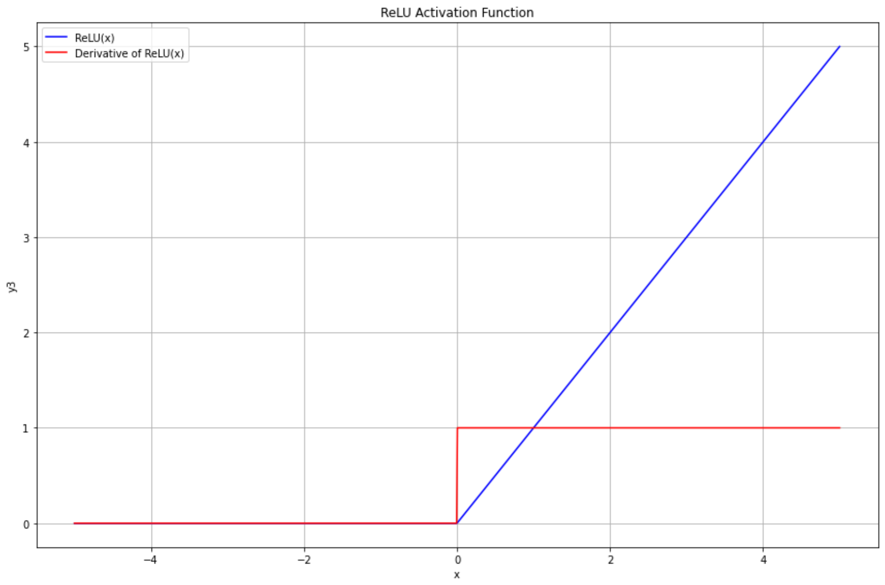


Figure 3 ReLU activation

1.2 Implementation

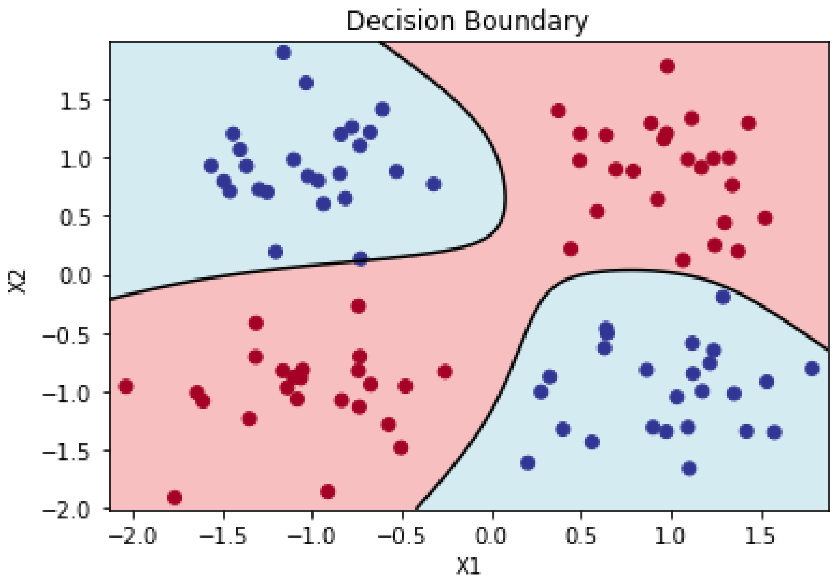


Figure 4 Training with Sigmoid activation

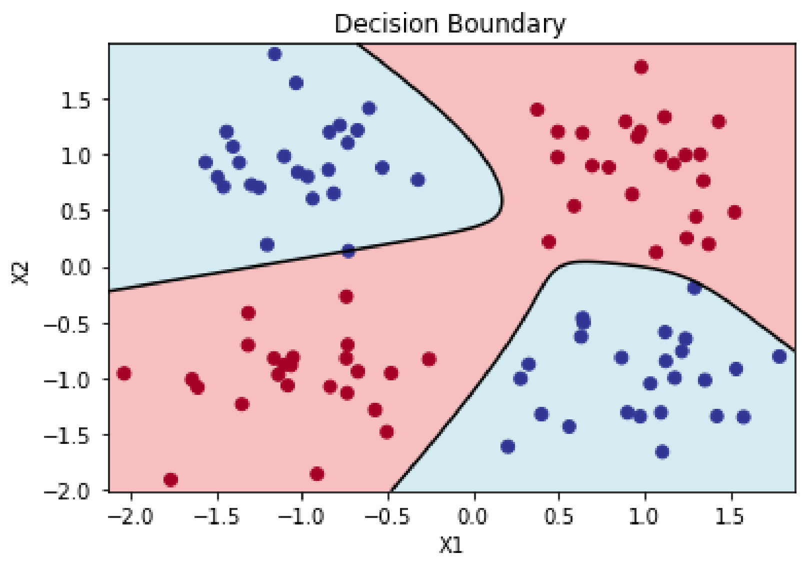


Figure 5 Decision Boundary for tanh activation

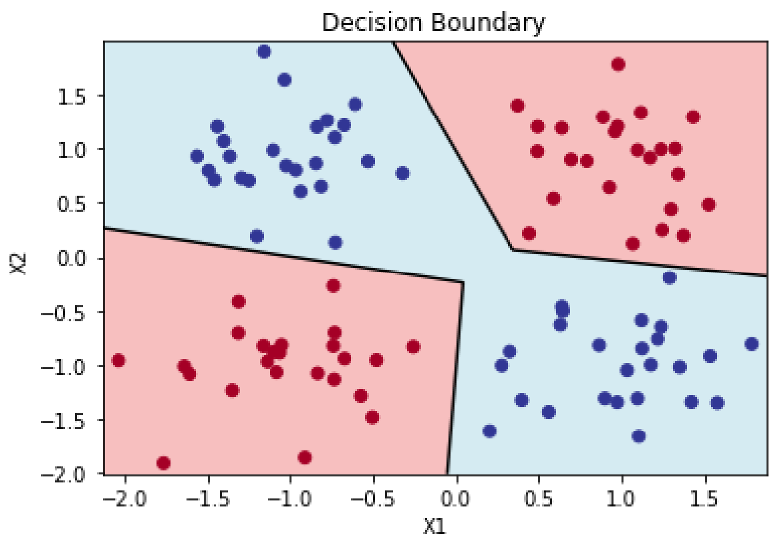


Figure 6 Decision Boundary for ReLU activation

1.3 Discussions

a. With and without considering your experimental results, what are the advantages and disadvantages of using each activation function? Which one is useful in which case?

Sigmoid Activation

The advantage of using sigmoid is that the sigmoid activation takes an input x and return the output in the interval (0,1). The most apparent drawback of using sigmoid activation function is that near the y = 1 and y= 0 areas the activation of neurons saturates either 0 or 1. Therefore, the derivative is very small for these areas. This results in with a very small gradient loss function which prevents the updating the weights and thus the entire learning process. Also, we can see that sigmoid function is not zero centered which is also not desirable since it always provide input grater than 0.

Tanh Activation
Similar to the sigmoid function neurons saturate at large negative and positive values and the derivative approaches to zero. However, unlike the sigmoid function, its outputs are zero-centered. Therefore, tanh activation can be more frequently used than sigmoid.

When mapping is required at the output, tanh can be more useful tan the sigmoid.

ReLU Activation
ReLU can accelerate the convergence of the gradient descent toward the minimum of the loss since it it linear and does not saturate for x>0. Also, the other activation functions are computationally expensive since they require a mapping. On the other hand, ReLU activation can be easily implemented with a very low computational power. The downside of using ReLU as the activation function is that for input values below zero, the neurons become ineffective. This makes no contribution to the training process. Therefore, when ReLU activation is used some neurons may not be activated at all.

For a problem where we want the neural network to predict values greater than 1, ReLU can be useful. However if we want a mapping at the output to (0,1) we should be map the output values using a sigmoid or tanh activation function. ReLU can be more generally used as an activation function in the hidden layers.

b. What is the XOR problem? Why do we need to use MLP instead of a single layer to solve it?

XOR problem gives the outputs of XOR logic gate given two inputs.
We need to use MLP instead of single layer to solve XOR since XOR is not a linearly separable problem. In single layer model a linear decision boundary is found which cannot effectively solve the XOR problem. However in Multi Layer neural model, neurons are capable of learning non-linear decision boundaries

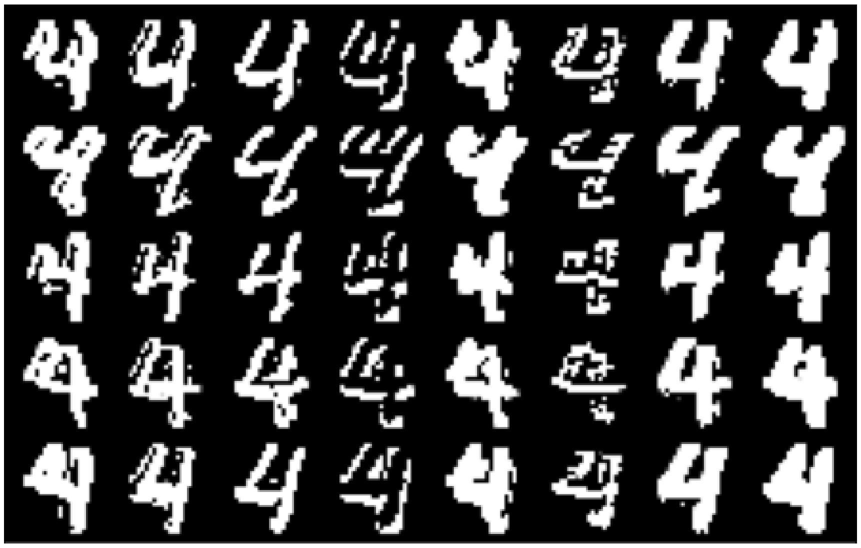
c. c. Run your training code a couple of times. Does your decision boundary change with each run or stay the same? Why?

Yes, when the code is runned for a couple of times the decision boundary changes when there is no seed provided. This is because we have random initialization of weights and inputs each time. Their distributions are different within the boundaries. Since we are training the MLP with with different weight initializations and inputs the decision boundary changes.

2. Implementing a Convolutional Layer with NumPy

2.1 Experimental Work

The output image of the my_conv2d function.



2.2 Discussion

a. Why are Convolutional Neural Networks important? Why are they used in image processing?

Convolutional neural networks are important since we can use these networks for feature extraction of the images. In image processing by learning these features we are able to train the whole network to learn and classify images based on these features. For example, when we are training a network with handwritten numbers from 0 to 9 with images, the network can learn the edges, lines and the connections between these lines to deduce a final output of that number. Therefore, this feature extraction property of CNN is very useful for the image processing.

b. What is a kernel of a Convolutional Layer? What do the sizes of a kernel correspond to?

Kernel can be thought as a weighted matrix, which slides over the input to produce a feature map. This feature map gives us information based on the kernel. The size of the kernel corresponds to the area in which the input data is considered. It is as if a window we observe the input for the output. When kernel size is smaller, we can detect finer details of the image and when the kernel size is larger we focus more on the bigger portion of the input.

c. Briefly explain the output image. What happened here?

The output of the my_conv2d function is based on the last digit of my school number, which is 4. In the resulting image, we can observe variations in the representation of the digit "4" across different rows. Within the same row, each column corresponds to the image convolved with a different kernel. When the same kernel is applied, similar features of the number become apparent. Conversely, using different kernels reveals diverse aspects of the image. As can be seen from the image, the fourth kernel demonstrates an ability to detect edges within the digit. With eight kernels in total, the output comprises eight columns, each offering distinct insights into the features of the input image.

d. Why are the numbers in the same column look alike to each other, even though they belong to different images?

The numbers in the same column look alike to each other since they undergo convolution with the same kernel. The kernel filters the image and extracts the specific features from the input image. Even though the same kernel is applied on different images, similar features are highlighted. This leads to similarities among the number in the same column.

e. Why are the numbers in the same row do not look alike to each other, even though they belong to same image?

Different kernels highlight different features of the input image, leading to variations in the output across different columns. In other words, when the same image is convolved with different kernels, different characteristics of that image are highlighted at the output. Therefore, we can see diverse characteristics of the same image, resulting in differences among the numbers in the same row.

f. What can be deduced about Convolutional Layers from your answers to Questions 4 and 5?

We can deduce that Convolutional Layers are extremely important in feature extraction since they detect various charactheristics of the input image by using different kernels. We can say that when the same kernel used in different input images, the outputs have similar characteristics since kernel may highlight one specific feature of the input. On the other hand different kernel on the same input image can show different features of that image.

3. Experimenting ANN Architectures

3.1 Experimental Work

Visualizations

In this question I cannot provide the full visualizations due to Google collab. Although I did not receive any error ,it took too much time so my files are removed. I wrote the val_accuracy cone wrong this is why the 3th and the 4th images are not visible. I couldn't run the code again due to time constraints.

However, I did work on this question for more than 2 days so I will try to answer the questions based on the plots I have. I have also provided the code. Here is the visualization of the parts that I could receive the results.

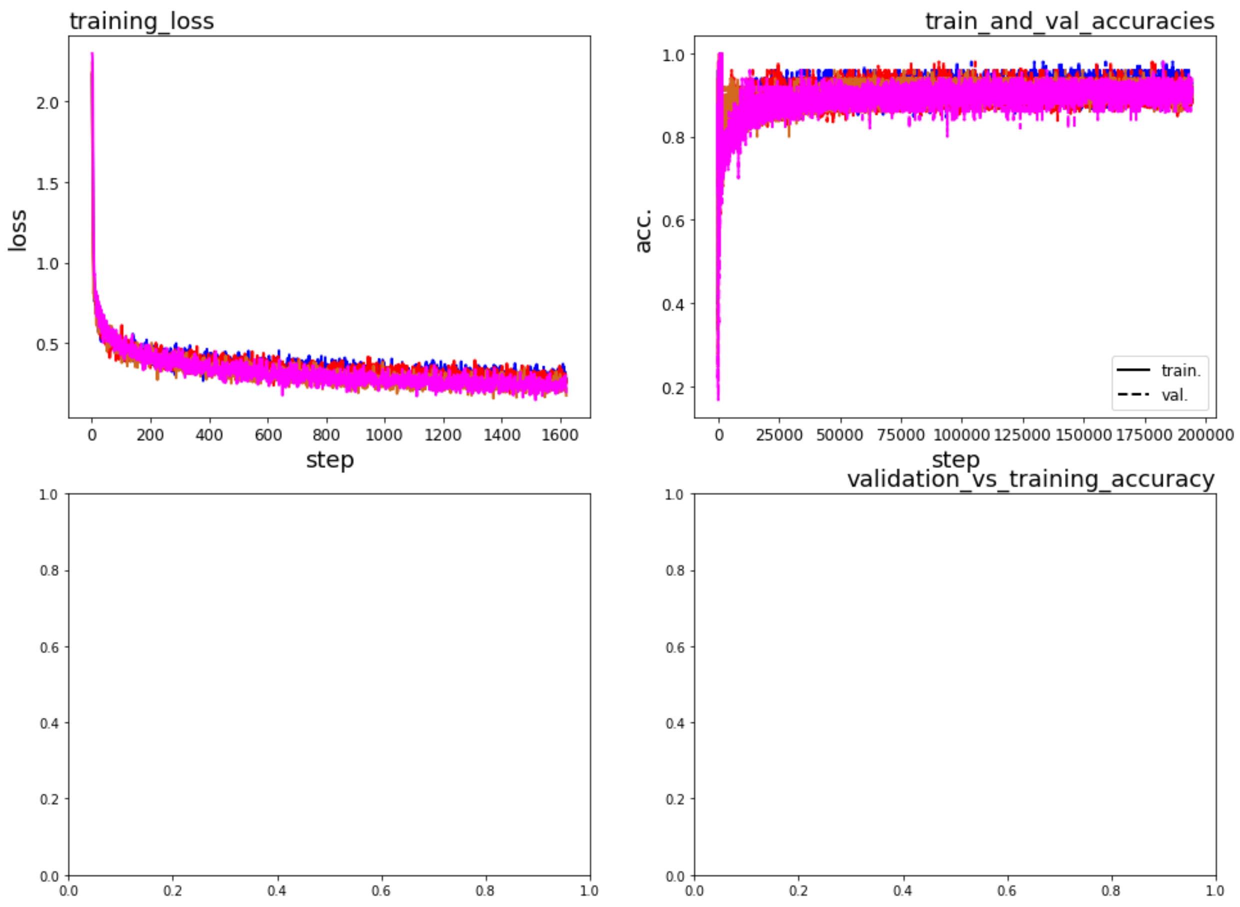


Figure 7 Loss and Accuracy Figures

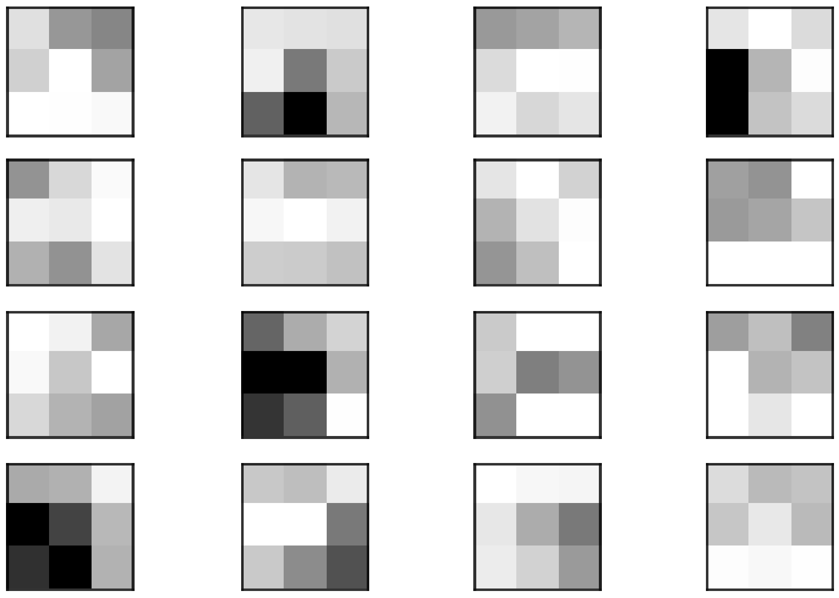


Figure 8 mlp_2 weights

3.2 Discussion

Compare the architectures by considering the performances, the number of parameters, architecture structures and the weight visualizations.

1. What is the generalization performance of a classifier?

The generalization performance of a classifier can be explained as its ability to predict new and unseen data. This means that if the classifier can apply what it has learned during the training process to the unseen data correctly, this indicates a good performance for the classifier. Therefore when we generalize the performance we would not only good performance on training data but also the test data.

2. Which plots are informative to inspect generalization performance?

Since the generalization performance is decided based on the consistency of the accuracy of test and training accuracy. I expect that the 'test_vs_training_accuracy' can be the most informative plot to inspect generalization performance.

3. Compare the generalization performance of the architectures.

Since I could not obtain all the plots for this part, I am not able to compare them using plots. However, from my research I may say that for a descent number of convolution layers combined with maxpool can increase the generalization performance. The balance of underfitting and overfitting is important so from the artitectures, mlp_1 and mlp_2 may underfit and cnn_5 may overfit. Therefore, I inspect that the best performance can be obtained from either cnn_3 or cnn_5

4. How does the number of parameters affect the classification and generalization performance?

As discussed above, the number of parameters may have an adverse effect on the generalization performance if overfitting occurs. Overfitting may occur when the model starts to 'memorize' rather than 'learn'. Therefore, I inspect that the excessive number of parameters as in cnn_5 can decrease the generalization performance. On the other hand, for very small number of parameters as in mlp_1, the model may not learn the features well so underfitting occurs.

5. How does the depth of the architecture affect the classification and generalization performance?

Yes, the depth is important for feature extraction. Therefore, we expect that as the depth increases the classification and generalization performance will increase until some level. However as I have stated before when the depth increases too much, overfitting may occur which results in with memorization. Therefore even if the training accuracy is very well, the test accuracy will not be that well.

6. Considering the visualizations of the weights, are they interpretable?

Unfortunately, I cannot visualize the weights. However, I know from my research that weights can be interpretable from their visualizations. I mean that, the weight function can highlight a certain feature of an image to be learned. Therefore, from the weight function we may interpret what it aims to characterize such as edges, the inside, curves etc.

7. Can you say whether the units are specialized to specific classes?
8. Weights of which architecture are more interpretable?

I think that cnn weights can be more interpretable due to their characterizations. When they are visualized, we can see more difference.

9. Considering the architecture, comment on the structures (how they are designed). Can you say that some architectures are akin to each other? Compare the performance of similarly structured architectures and architectures with different structures.

Yes some architectures are alike, they undergo similar convolution and pooling operations with different kernel sizes.

10. Which architecture would you pick for this classification task? Why?

I would pick either CNN-3 or CNN-4 due to their high performance with minimum loss. However due to the ease of performing mlp_2, I may have gained computational power and time.

Python Code:

```
# -*- coding: utf-8 -*-
"""
Created on Mon Mar 11 22:19:04 2024

@author: defne
"""


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


#%% Tanh Activation Function
#np.random.seed()


def tanh(x):
   return np.tanh(x)


def tanh_derivative(x):
   return 1-tanh(x)**2


x_values = np.linspace(-5, 5, 1000)


y_tanh = tanh(x_values)
y__tanh_derivative = tanh_derivative(x_values)


plt.figure(figsize=(12,8))
plt.plot(x_values, y_tanh, label='tanh(x)', color = 'b')

plt.plot(x_values, y__tanh_derivative, label=r'$\frac{\partial tanh(x)}{\partial x}$', color =
'red')

plt.title('Tanh Activation Function')

plt.xlabel('x')

plt.ylabel('y1')

plt.grid()

plt.legend()


plt.show()


#%% Sigmoid Activation Function


def sigmoid(x):

   return 1/(1+np.exp(-x))


def sigmoid_derivative(x):

   return sigmoid(x) * (1-sigmoid(x))


x_values = np.linspace(-5, 5, 1000)


y_sigmoid = sigmoid(x_values)

y_sigmoid_derivative = sigmoid_derivative(x_values)


# Plot the sigmoid activation function

plt.figure(figsize=(12, 8))

plt.plot(x_values, y_sigmoid, label='sigmoid(x)', color = 'b')
plt.plot(x_values, y_sigmoid_derivative, label='Derivative of sigmoid(x)', color = 'r')
plt.title('Sigmoid Activation Function')
plt.xlabel('x')
plt.ylabel('y2')
plt.legend()

plt.grid()

plt.show()

#%% ReLU Activation Function

def relu(x):
   return np.maximum(0,x)

def relu_derivative(x):
   return 1 * (x > 0)

x_values = np.linspace(-5, 5, 1000)

# Compute y values for ReLU and its derivative
y_relu = relu(x_values)
y_relu_derivative = relu_derivative(x_values)

plt.figure(figsize=(12, 8))
plt.plot(x_values, y_relu, label='ReLU(x)', color = 'b')
plt.plot(x_values, y_relu_derivative, label='Derivative of ReLU(x)', color = 'r')
plt.title('ReLU Activation Function')

plt.xlabel('x')

plt.ylabel('y3')

plt.legend()


plt.grid()

# Show the plots

plt.tight_layout()

plt.show()


#%% Part 1

#MLP with activation function sigmoid is provided

class MLP:


    def __init__(self, input_size, hidden_size, output_size):

        self.input_size = input_size

        self.hidden_size = hidden_size

        self.output_size = output_size


        # Initialize weights and biases

        # np.random.rand() returns a ndarray, shape of the returned array is input_size x
hidden_size

        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 1/(1+np.exp(-x))
  def sigmoid_derivative(self, x):
    return self.sigmoid(x)*(1-self.sigmoid(x))
  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 = self.sigmoid(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 network
    # Compute error
    output_error = (targets -
self.output)*(self.sigmoid_derivative(np.dot(self.hidden_output,
self.weights_hidden_output) + self.bias_output))
    hidden_error = self.sigmoid_derivative(np.dot(inputs, self.weights_input_hidden) +
self.bias_hidden)*np.dot(output_error, self.weights_hidden_output.T)

    # Compute gradients
    output_delta = learning_rate * np.dot(self.hidden_output.T, output_error)
    hidden_delta = learning_rate * np.dot(inputs.T, hidden_error)

    #Update weights and biases
    self.weights_hidden_output = self.weights_hidden_output + output_delta
    self.bias_output = self.bias_output + np.sum(output_error, axis=0) * learning_rate
    self.weights_input_hidden = self.weights_input_hidden + hidden_delta
    self.bias_hidden = self.bias_hidden + np.sum(hidden_error, axis=0) * learning_rate
```

```
#for activation function tanh:

# class MLP:


#    def __init__(self, input_size, hidden_size, output_size):

#       self.input_size = input_size

#       self.hidden_size = hidden_size

#       self.output_size = output_size


#       # Initialize weights and biases

#       # np.random.rand() returns a ndarray, shape of the returned array is input_size x
hidden_size

#       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.tanh(x)


#    def tanh_derivative(self,x):

#       return 1-(self.tanh(x))**2


#    def sigmoid(self, x):

#       return 1/(1+np.exp(-x))

#    def sigmoid_derivative(self, x):

#       return self.sigmoid(x)*(1-self.sigmoid(x))
#    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 = self.sigmoid(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 network
#        # Compute error
#        output_error = (targets -
self.output)*(self.sigmoid_derivative(np.dot(self.hidden_output,
self.weights_hidden_output) + self.bias_output))
#        hidden_error = self.tanh_derivative(np.dot(inputs, self.weights_input_hidden) +
self.bias_hidden)*np.dot(output_error, self.weights_hidden_output.T)

#        # Compute gradients
#        output_delta = learning_rate * np.dot(self.hidden_output.T, output_error)
#        hidden_delta = learning_rate * np.dot(inputs.T, hidden_error)

#        #Update weights and biases
#        self.weights_hidden_output = self.weights_hidden_output + output_delta
#        self.bias_output = self.bias_output + np.sum(output_error, axis=0, keepdims=True) *
learning_rate
#        self.weights_input_hidden = self.weights_input_hidden + hidden_delta
#        self.bias_hidden = self.bias_hidden + np.sum(hidden_error, axis=0, keepdims=True) *
learning_rate
```

```
# For activation function ReLU

# class MLP:


#    def __init__(self, input_size, hidden_size, output_size):

#        self.input_size = input_size

#        self.hidden_size = hidden_size

#        self.output_size = output_size


#        # Initialize weights and biases

#        # np.random.rand() returns a ndarray, shape of the returned array is input_size x
hidden_size

#        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 1 * (x > 0)


#    def sigmoid(self, x):

#        return 1/(1+np.exp(-x))

#    def sigmoid_derivative(self, x):

#        return self.sigmoid(x)*(1-self.sigmoid(x))
#    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 = self.sigmoid(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 network
#        # Compute error
#        output_error = (targets -
self.output)*(self.sigmoid_derivative(np.dot(self.hidden_output,
self.weights_hidden_output) + self.bias_output))
#        hidden_error = self.relu_derivative(np.dot(inputs, self.weights_input_hidden) +
self.bias_hidden)*np.dot(output_error, self.weights_hidden_output.T)

#        # Compute gradients
#        output_delta = learning_rate * np.dot(self.hidden_output.T, output_error)
#        hidden_delta = learning_rate * np.dot(inputs.T, hidden_error)

#        #Update weights and biases
#        self.weights_hidden_output = self.weights_hidden_output + output_delta
#        self.bias_output = self.bias_output + np.sum(output_error, axis=0, keepdims=True) *
learning_rate
#        self.weights_input_hidden = self.weights_input_hidden + hidden_delta
#        self.bias_hidden = self.bias_hidden + np.sum(hidden_error, axis=0, keepdims=True) *
learning_rate
```

```
#%%

x_train, y_train, x_val, y_val = part1CreateDataset(train_samples=1000, val_samples=100,
std=0.4)

input_size = 2
hidden_size = 4
output_size = 1
learning_rate = 0.001

nn = MLP(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)

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

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

```
part1PlotBoundary(x_val, y_val, nn)


#%% Part 2
import numpy as np
import torch
import torch.nn as nn
from utils import part2Plots


# inpute shape: [batch size, input_channels, input_height, input_width]
input = np.load('C:\\Users\\defne\\Desktop\\2023-
2024SpringSemester\\EE449\\HW1\\data\\samples_4.npy')


#kernel shape: [output_channels, input_channels, filter_height, filter width]
kernel = np.load('C:\\Users\\defne\\Desktop\\2023-
2024SpringSemester\\EE449\\HW1\\data\\kernel.npy')


#out = my_conv2d(input, kernel)


def my_conv2d(input_data, kernel):


    batch_size, input_channels, input_height, input_width = input_data.shape
    output_channels, input_channels, filter_height, filter_width = kernel.shape


    #Calculating the ouput dimensions
    output_height = input_height - filter_height + 1
    output_width = input_width - filter_width + 1
```

```
   #Initialize the output tensor
   output = np.zeros((batch_size, output_channels, output_height, output_width))


   #Performing the convolution operation
   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(np.multiply(input_data[b, ic, i:i+filter_height,
j:j+filter_width], kernel[oc, ic]))
   return output


output_my_conv2d = my_conv2d(input, kernel)


part2Plots(output_my_conv2d)


# # For the comparison with Conv2d library
# # Converting input data and kernel to PyTorch tensors
# input_tensor = torch.tensor(input, dtype=torch.float32)
# kernel_tensor = torch.tensor(kernel, dtype=torch.float32)


# # Create a Conv2d layer
# conv_layer = nn.Conv2d(in_channels=1,  # Input channels
#                  out_channels=8,  # Output channels
#                  kernel_size=(4, 4))  # Kernel size
# #Setting the kernel weights for the convolutional layer
# conv_layer.weight.data = kernel_tensor

# # Perform the convolution operation
# output = conv_layer(input_tensor)  # Convert input data to float

# # Print the shape of the output tensor
# part2Plots(output)

#%% Part 3

import torchvision

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

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

#%%

import torch

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

```
class FullyConnected(torch.nn.Module):
  def __init__(self, input_size, hidden_size, num_classes):
    super(FullyConnected, self).__init__()
    self.input_size = input_size
    self.fc1 = torch.nn.Linear(input_size, hidden_size) #fc1: fully connected layer1
    self.fc2 = torch.nn.Linear(hidden_size, num_classes) #fc2: fully connected layer2
    self.relu = torch.nn.ReLU()
  def forward(self, x):
    x = x.view(-1, self.input_size)
    hidden = self.fc1(x)
    relu = self.relu(hidden)
    output = self.fc2(relu)
    return output

# initialize your model
model_mlp = FullyConnected(784, 128, 10)

# get the parameters 784x128 layer as numpy array
params_784x128 = model_mlp.fc1.weight.data.numpy()

# create loss: use cross entropy loss
loss = torch.nn.CrossEntropyLoss()
# create optimizer
optimizer = torch.optim.SGD(model_mlp.parameters(), lr=0.01, momentum = 0.0)
# transfer your model to train mode
model_mlp.train()
# transfer your model to eval mode
```

```
model_mlp.eval()


#%%


#classes


class mlp_1(torch.nn.Module):
  def __init__(self):
    super(mlp_1, self).__init__()
    self.fc1 = torch.nn.Linear(784, 32)
    self.relu = torch.nn.ReLU()
    self.fc2 = torch.nn.Linear(32, 10)

  def forward(self, x):
    x = x.view(-1, 784)
    hidden = self.fc1(x)
    relu = self.relu(hidden)
    output = self.fc2(relu)
    return output

class mlp_2(torch.nn.Module):
  def __init__(self):
    super(mlp_2, self).__init__()
    self.fc1 = torch.nn.Linear(784, 32)
    self.relu = torch.nn.ReLU()
    self.fc2 = torch.nn.Linear(32, 64, bias=False)
    self.fc3 = torch.nn.Linear(64, 10)
  def forward(self,x):
    x = x.view(-1, 784)
    hidden = self.fc1(x)
    relu = self.relu(hidden)
    hidden_2 = self.fc2(relu)
    output = self.fc3(hidden_2)
    return output

class cnn_3(torch.nn.Module):
  def __init__(self):
    super(cnn_3, self).__init__()
    self.conv1 = torch.nn.Conv2d(1, 16, (3,3), stride=1, padding = 'valid')
    self.relu = torch.nn.ReLU()
    self.conv2 = torch.nn.Conv2d(16, 8, (5,5), stride=1, padding = 'valid')
    self.pool = torch.nn.MaxPool2d( 2, stride = 2)
    self.conv3 = torch.nn.Conv2d(8, 16, (7,7), stride=1, padding = 'valid')
    self.fc1 = torch.nn.Linear(16*2*2, 10)

  def forward(self,x):
    x = self.relu(self.conv1(x))
    x = self.relu(self.conv2(x))
    x = self.pool(x)
    x = self.pool(self.conv3(x))
    x = x.view(-1, 16*2*2)
    x = self.fc1(x)
    return x
```

```
class cnn_4(torch.nn.Module):
  def __init__(self):
    super(cnn_4, self).__init__()
    self.conv1 = torch.nn.Conv2d(1, 16, (3,3), stride=1, padding = 'valid')
    self.relu = torch.nn.ReLU()
    self.conv2 = torch.nn.Conv2d(16, 8, (3,3), stride=1, padding = 'valid')
    self.conv3 = torch.nn.Conv2d(8, 16, (5,5), stride=1, padding = 'valid')
    self.pool = torch.nn.MaxPool2d( 2, stride = 2)
    self.conv4 = torch.nn.Conv2d(16, 16, (5,5), stride=1, padding = 'valid')
    self.fc1 = torch.nn.Linear(16*3*3, 10)

  def forward(self,x):
    x = self.relu(self.conv1(x))
    x = self.relu(self.conv2(x))
    x = self.relu(self.conv3(x))
    x = self.pool(x)
    x = self.relu(self.conv4(x))
    x = self.pool(x)
    x = x.view(-1, 16*3*3)
    x = self.fc1(x)
    return x

class cnn_5(torch.nn.Module):
  def __init__(self):
    super(cnn_5, self).__init__()
    self.conv1 = torch.nn.Conv2d(1, 8, (3,3), stride=1, padding = 'valid')
```

```
        self.relu = torch.nn.ReLU()

        self.conv2 = torch.nn.Conv2d(8, 16, (3,3) , stride=1, padding = 'valid')

        self.conv3 = torch.nn.Conv2d(16, 8, (3,3), stride = 1, padding = 'valid')

        self.pool = torch.nn.MaxPool2d( 2, stride = 2)

        self.conv4 = torch.nn.Conv2d(16, 16, (3,3), stride = 1, padding = 'valid')

        self.fc1 = torch.nn.Linear(8*3*3, 10)


    def forward(self,x):

        x = self.relu(self.conv1(x))

        x = self.relu(self.conv2(x))

        x = self.relu(self.conv3(x))

        x = self.relu(self.conv2(x))

        x = self.pool(x)

        x = self.relu(self.conv4(x))

        x = self.relu(self.conv3(x))

        x = self.pool(x)

        x = x.view(-1, 8*3*3)

        x = self.fc1(x)

        return x




#%%

#Importing useful libraries

import torch

import numpy as np

import torch.optim as optim
import torch.nn.functional as F
from torchvision import datasets, transforms
import torchvision
from sklearn.model_selection import train_test_split
import pickle
from utils import part3Plots


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


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


# Stratified Sampling for train and val
train_idx, validation_idx = train_test_split(np.arange(len(train_data)),
                                    test_size=0.1,
                                    random_state=42,
                                    shuffle=True,
                                    stratify=train_data.train_labels)


# Subset dataset for train and val
train_dataset = torch.utils.data.Subset(train_data, train_idx)
validation_dataset = torch.utils.data.Subset(train_data, validation_idx)


# Dataloader for train and val
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=50, shuffle=True)
```

```
validation_loader = torch.utils.data.DataLoader(validation_dataset, batch_size=50,
shuffle=False)

test_loader = torch.utils.data.DataLoader(test_data, batch_size = 50, shuffle = False)


results = []


#%%

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


#model_names = [mlp_1, mlp_2, cnn_3, cnn_4, cnn_5]

model_names = [mlp_1]


for model_class in model_names:

  model = model_class().to(device)


  criterion = torch.nn.CrossEntropyLoss() #Cost function. Predicts the probability
distribution of each class

  optimizer = optim.Adam(model.parameters())

  epochs = 15


  train_losses = []

  train_accuracy = []


  valid_accuracy = []

  for e in range(epochs):

    model.train() #preparing the model for training

    running_loss = 0

    train_accur = 0
    correct = 0
    total = 0
    # train the model
    for steps, (images, labels) in enumerate(train_loader):
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad() #clear the gradients in order to avoid accumulation
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward() #perform a backward pass through the network to calculate the
gradients for model parameters
        optimizer.step() #take a step with the optimizer to update model parameters

        running_loss += loss.item()*images.size(0)

        if steps % 10 == 9: #recording every ten steps
            train_losses.append(running_loss/(10*images.size(0)))
            running_loss = 0

            _, predicted = torch.max(outputs.data, 1)
            correct = (predicted == labels).sum().item()
            train_accur = correct / labels.size(0)
            train_accuracy.append(train_accur)

            # Validation phase
            model.eval() #enters to validations part
```

```
      valid_loss = 0
      val_correct = 0
      val_total = 0
      for val_images, val_labels in validation_loader:
        val_images, val_labels = val_images.to(device), val_labels.to(device)

        #optimizer.zero_grad() #clear the gradients in order to avoid accumulation
        val_outputs = model(val_images)
        _, val_predicted = torch.max(val_outputs.data, 1)
        val_correct += (val_predicted == val_labels).sum().item()
        val_total += val_labels.size(0)
        val_accur = val_correct / val_labels.size(0)
        valid_accuracy.append(val_accur)
      model.train() #goes back to training

  model.eval() #testing evaluation part
  test_loss = 0
  test_correct = 0
  test_total = 0
  for images, labels in test_loader:
    images, labels = images.to(device), labels.to(device)

    optimizer.zero_grad() #clear the gradients in order to avoid accumulation
    outputs = model(images)
    loss = criterion(outputs, labels)
    test_loss += loss.item()*images.size(0)
    _, predicted = torch.max(outputs.data, 1)

    test_total += labels.size(0)

    test_correct += (predicted == labels).sum().item()


test_accuracy = test_correct / test_total


# if hasattr(model, 'conv1'):

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

# elif hasattr(model, 'fc1'):

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



results_dict = {

    'name' : str(model),

    'loss_curve' : train_losses,

    'train_acc_curve' : train_accuracy,

    'val_acc_curve' : valid_accuracy,

    'test_acc' : test_accuracy,

    'weights' : first_layer_weights

    }


results.append(results_dict)

#print(results_dict)

# Save results with pickle

model_name = model.__class__.__name__  # This will be 'mlp_1', 'mlp_2', etc.


filename = f'part3_{model_name}.pkl'
```

```
    with open(filename, 'wb') as f:
        pickle.dump(results_dict, f)


#%%


part3Plots(results, 'C:\\Users\\defne\\Desktop\\2023-2024SpringSemester\\EE449\\HW1',
'mlp_2')


import pickle
from utils import part3Plots, visualizeWeights


with open('C:\\Users\\defne\\Downloads\\part3_mlp_1.pkl', 'rb') as handle:
    a = pickle.load(handle)
with open('C:\\Users\\defne\\Downloads\\part3_mlp_2.pkl', 'rb') as handle:
    b = pickle.load(handle)
with open('C:\\Users\\defne\\Downloads\\part3_cnn_3.pkl', 'rb') as handle:
    c = pickle.load(handle)
with open('C:\\Users\\defne\\Downloads\\part3_cnn_4.pkl', 'rb') as handle:
    d = pickle.load(handle)
# with open('part3_cnn_5.pkl','rb') as handle:
#     e=pickle.load(handle)
part3Plots([a,b,c,d])
a_weights = a['weights']
b_weights = b['weights']
c_weights = c['weights']
d_weights = d['weights']
```

```
visualizeWeights(a_weights, 'C:\\Users\\defne\\Desktop\\2023-2024SpringSemester\\EE449\\HW1', 'a')

visualizeWeights(b_weights, 'C:\\Users\\defne\\Desktop\\2023-2024SpringSemester\\EE449\\HW1', 'b')

visualizeWeights(c_weights, 'C:\\Users\\defne\\Desktop\\2023-2024SpringSemester\\EE449\\HW1', 'c')

visualizeWeights(d_weights, 'C:\\Users\\defne\\Desktop\\2023-2024SpringSemester\\EE449\\HW1', 'd')
```