Assignment 5: Convolutional Neural Networks and Dimensionality Reduction

Part 1: Implement a convolutional neural network for classifying MNIST digits using PyTorch and the provided template CNN Template.py. Define your layers in the init () function, and define the connectivity between the layers in the forward() function of class Convnet(nn.Module). Use the configuration provided in the table below. For training, use a batch size of 32, and the Adam optimizer (torch.optim.Adam) with learning rate of 0.01 for 10 epochs.

Python file name on github: convolutional_neural_network.py

```
from __future__ import print_function
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms

import numpy as np
import torch.utils.data as utils

#for checking summary of model
from torchvision import models
from torchsummary import summary

#for plotting
import matplotlib.pyplot as plt

# The parts that you should complete are designated as TODO
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        # TODO: define the layers of the network
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, stride=1, padding=0)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=0)
        self.relu = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.dropout = nn.Dropout2d(p=0.25)
        self.flatten = nn.Flatten()
        self.linear1 = nn.Linear(9216, 128)
        self.dropout2 = nn.Dropout(p=0.5)
        self.linear2 = nn.Linear(128,10)

    def forward(self, x):
        # TODO: define the forward pass of the network using the layers you defined in constructor
        x = self.conv1(x)
        x = self.relu(x)
        x = self.conv2(x)
        x = self.relu(x)
        x = self.pool1(x)
        x = self.dropout(x)
        x = self.flatten(x)
        x = self.linear1(x)
        x = self.relu(x)
        x = self.dropout2(x)
        x = self.linear2(x)

        return x

def train(model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.cross_entropy(output, target)
        loss.backward()
        optimizer.step()
        #accuracy = 100. * batch_idx / len(train_loader)
        if batch_idx % 100 == 0: #Print loss every 100 batch
            print('Train Epoch: {} [{}/{}]\tLoss: {:.6f}'.format(
```

```
                epoch, batch_idx * len(data), len(train_loader.dataset),
                loss.item()))
    print('-'*20)
    print('Training Accuracy (not test accuracy):')
    accuracy = test(model, device, train_loader)
    print('-'*20)

    return accuracy

def test(model, device, test_loader):
    model.eval()
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            pred = output.argmax(dim=1, keepdim=True)  # get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()

    accuracy = 100. * correct / len(test_loader.dataset)
    print('\nTest set Accuracy: {}/{} ({:.0f}%)\n'.format(
        correct, len(test_loader.dataset),
        accuracy))
    print('-'*20)

    return accuracy


def main():
    print('-'*20)
    print('Start of CNN Script')
    torch.manual_seed(1)
    np.random.seed(1)
    # Training settings
    use_cuda = False # Switch to False if you only want to use your CPU
    learning_rate = 0.01
    NumEpochs = 10
    batch_size = 32

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

    train_X = np.load('../../Data/X_train.npy')
    train_Y = np.load('../../Data/y_train.npy')

    test_X = np.load('../../Data/X_test.npy')
    test_Y = np.load('../../Data/y_test.npy')

    train_X = train_X.reshape([-1,1,28,28]) # the data is flatten so we reshape it here to get to the original
dimensions of images
    test_X = test_X.reshape([-1,1,28,28])

    # transform to torch tensors
    tensor_x = torch.tensor(train_X, device=device)
    tensor_y = torch.tensor(train_Y, dtype=torch.long, device=device)

    test_tensor_x = torch.tensor(test_X, device=device)
    test_tensor_y = torch.tensor(test_Y, dtype=torch.long)

    train_dataset = utils.TensorDataset(tensor_x,tensor_y) # create your datset
    train_loader = utils.DataLoader(train_dataset, batch_size=batch_size) # create your dataloader

    test_dataset = utils.TensorDataset(test_tensor_x,test_tensor_y) # create your datset
    test_loader = utils.DataLoader(test_dataset) # create your dataloader if you get a error when loading test
data you can set a batch_size here as well like train_dataloader

    model = ConvNet().to(device)
    #print a summary of layer dimensions and parameters
    print('Model Summary:')
    summary(model, input_size=(1,28,28))
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    #create lists to store training and testing accuracy for plotting
    training_accuracy = []
    testing_accuracy = []

    for epoch in range(NumEpochs):
        train_acc = train(model, device, train_loader, optimizer, epoch)
        #append training accuracy list
        training_accuracy.append(train_acc)
        test_acc = test(model, device, test_loader)
        #append testing accuracy list
```

```
        testing_accuracy.append(test_acc)

    torch.save(model.state_dict(), "mnist_cnn.pt")

    #TODO: Plot train and test accuracy vs epoch
    plt.plot(list(range(1,11)), training_accuracy, label='Training Accuracy')
    plt.plot(list(range(1,11)), testing_accuracy, label='Testing Accuracy')
    plt.title('Training and Testing Accuracy\nfor Each Epoch')
    plt.xlabel('Number of Epochs')
    plt.ylabel('Accuracy (%)')
    plt.legend()
    plt.savefig('../Figures/cnn_accuracy.png')
    plt.show()
    plt.clf()

    print('End of CNN Script')
    print('-'*20)
    print('End of HW5 Scripts')


if __name__ == '__main__':
    main()


#Below was used to generate the predictions for the submission folder
#select all and run selection if you would like to reproduce
#digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
#model.eval()
#correct = 0
#predictions_vect = []
#with torch.no_grad():
#    for data, target in test_loader:
#        data, target = data.to(device), target.to(device)
#        output = model(data)
#        pred = output.argmax(dim=1, keepdim=True)  # get the index of the max log-probability
#        correct += pred.eq(target.view_as(pred)).sum().item()
#        _, predictions = torch.max(output, 1)
#        predictions_vect.append(digits[predictions])
#np.savetxt('../Predictions/predictions_MNIST.csv', predictions_vect, delimiter=',')
```

Part 2: Implement an algorithm that computes a sub-optimal *k* rank approximation of the original data matrix. This is done by calculating the approximation of the right singular matrix V. Try to get a low rank approximation of the baboon image.

Python file name on github: singular_value_decomposition

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
def SVD(A, s, k):
    # TODO: Calculate probabilities p_i
    n,m = A.shape
    #create a matrix to store probabilities in
    p_i_all = np.zeros(n)
    for i in range(n):
        #each pi should be a scalar, so p_i_all should be vector of length n (number of rows)
        p_i = (np.linalg.norm(A[i,:])*np.linalg.norm(A[i,:]))/(np.linalg.norm(A, ord='fro')*np.linalg.norm(A,
ord='fro'))
        p_i_all[i] = p_i

    # TODO: Construct S matrix of size s by m
    #create a matrix of zeros to store values
    S = np.zeros((s,m))
    #pick a random integer j with probability Pr(pick j)=p_j
    for i in range(s):
        #use random choice function to select integer and store in S
        j = np.random.choice(n, replace=False, p=p_i_all)
        S[i] = A[j,:]

    # TODO: Calculate SS^T
    sst = np.dot(S,np.matrix.transpose(S))

    # TODO: Compute SVD for SS^T
    w, s, vh = np.linalg.svd(sst)
```

```
    # TODO: Construct H matrix of size m by k
    # create matrix H to store data
    H = np.zeros((m,k))
    #loop over k rows and calculate h_t for each row
    for i in range(k):
        h_t =
(np.dot(np.matrix.transpose(S),vh[i,:]))/(np.linalg.norm((np.dot(np.matrix.transpose(S),vh[i,:]))))
        #store h_t in matrix H
        H[:,i] = h_t

    # Return matrix H and top-k singular values sigma
    return H, s[0:k]

def main():
    print('Start of SVD Script')
    im = Image.open("../../Data/baboon.tiff")
    A = np.array(im)
    H, sigma = SVD(A, 80, 60)
    k = 60

    # TO DO: Compute SVD for A and calculate optimal k-rank approximation for A.
    #compute the SVD for A
    U, S, V = np.linalg.svd(A)
    #calculate A_k using equation A_k=USV^T
    A_k = np.matmul(U[:,0:k],np.matmul(np.diag(S[0:k]),V[0:k,:]))

    # TO DO: Use H to compute sub-optimal k rank approximation for A
    #equation: A_hat = AHH^t
    A_k_hat = np.matmul((np.matmul(A,H)),np.transpose(H))

    # To DO: Generate plots for original image, optimal k-rank and sub-optimal k rank approximation
    #plot original matrix
    plt.imshow(A, cmap='gray', vmin=0, vmax=255)
    #plot title
    plt.title("Original Baboon Image")
    #save, show, and clear figure
    plt.savefig(('../Figures/baboon_original.png'))
    plt.show()
    plt.clf()

    #plot optimal k-rank approximation
    plt.imshow(A_k, cmap='gray', vmin=0, vmax=255)
    #title
    plt.title("Optimal K-Rank Approximation")
    #save, show, and clear figure
    plt.savefig(('../Figures/optimal_approx.png'))
    plt.show()
    plt.clf()

    #plotsub-optimal k rank approximation
    plt.imshow(A_k_hat, cmap='gray', vmin=0, vmax=255)
    #title
    plt.title("Sub-optimal K-Rank Approximation")
    #save, show, and clear figure
    plt.savefig(('../Figures/sub_optimal_approx.png'))
    plt.show()
    plt.clf()

    # TO DO: Calculate the error in terms of the Frobenius norm for both the optimal-k
    # rank produced from the SVD and for the k-rank approximation produced using
    # sub-optimal k-rank approximation for A using H.

    #calcualate error of optimal k-rank approximation, A_k, in terms of Frobenius norm
    A_k_fro = np.linalg.norm(A-A_k, ord='fro')
    #calcualate error of sub-optimal k-rank approximation, A_k_hat, in terms of Frobenius norm
    A_k_hat_fro = np.linalg.norm(A-A_k_hat, ord='fro')

    #subtract and print errors
    print('-'*20)
    print('Optimal K-Rank Approximation Error:')
    print(A_k_fro)
    print('')
    print('Sub-Optimal K-Rank Approximation Error:')
    print(A_k_hat_fro)
    print('End of SVD Script')
    print('-'*20)



if __name__ == "__main__":
    main()
```

Part 3: Implement PCA from scratch and run the algorithm on the MNIST dataset. Produce images of the top 5 eigenvectors and plot the datapoints on the first and second principal components.

Python file name on github: principal_component_analysis.py

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

print('-'*20)
print('Start of PCA Script')
#Load data
X = np.load('../../Data/X_train.npy')
Y = np.load('../../Data/y_train.npy')
#%% Plotting mean of the whole dataset
mean_all = np.mean(X, axis=0)
#reshape the image so it is 28x28
mean_all_reshape = mean_all.reshape(28,28)
#create an image of mean of all images in dataset
plt.imshow(mean_all_reshape)
#plot title
plt.title('Rendering of Mean of All\nDigit Images in Sample')
#save figure
plt.savefig(('../Figures/mean_of_digits.png'))
#show figure
plt.show()
#clear figure
plt.clf()


#%% Plotting each digit
#need to calculate the average of all of the same digits (which will give us 10 total digits ranging from 0-9)
#loop over 10 digits (0-9)
for i in range(10):
    #get the average vector for the particular digit
    digit_avgs = np.average(X[Y==i],0)
    #create a subplot
    plt.subplot(2, 5, i+1)
    #reshape the vector to 28x28
    current_digit = digit_avgs.reshape(28,28)
    #add the image plot to the subplot
    plt.imshow(current_digit)
    #remove axes
    plt.axis('off')
#arrange images nicely on subplot
#plt.subplots_adjust(wspace=0, hspace=0)
#plot title
plt.suptitle('Rendering of Average of Same\nDigit Images in Sample\n')
#save figure
plt.savefig(('../Figures/indivdual_digits.png'))
#show figure
plt.show()
#clear figure
plt.clf()

#%% Center the data (subtract the mean)
a, b = X.shape
#calculate mean of each rcolumn (each column represents a variable)
mean_X = np.mean(X, axis=0)
#subtract the mean from every entry in the matrix
mean_cent_X = np.zeros((a, b))
for i in range(a):
    mean_cent_X[i,:] = X[i,:] - mean_X


#%% Calculate Covariate Matrix
cov_mat = np.cov(mean_cent_X, rowvar=False, bias=True)

#%% Calculate eigen values and vectors
#np.linal.eigh sorts the eigenvectors and eigenvalues in reverse order
eigenvals, eigenvects = np.linalg.eigh(cov_mat)
#sort the eigenvalues in descending order
sorted_eigenvals = eigenvals[::-1]
#sorting the eigenvectors too
sorted_eigenvects = np.flip(eigenvects, axis=1)
```

```
#%% Plot eigen values
plt.plot(sorted_eigenvals)
plt.ylabel('Eigenvalue (value)')
plt.xlabel('Eigenvalue Number (count)')
plt.title('Eigenvalue Numbers and Respective Values')
plt.savefig('../Figures/eigenvalues.png')

#%% Plot 5 first eigen vectors
#loop over 5
for i in range(5):
    #create a subplot
    plt.subplot(1, 5, i+1)
    #reshape the current eigenvector
    current_eigenvector = sorted_eigenvects[:,i].reshape(28,28)
    #plot the reshaped eigenvector
    plt.imshow(current_eigenvector)
    #turn off axes
    plt.axis('off')
    #add title
    plt.title('PC'+str(i+1))
#plot title
#plt.suptitle('Plots of Top 5 Eigenvectors')
#save figure
plt.savefig(('../Figures/top_five_eigenvectors.png'))
#show figure
plt.show()
#clear figure
plt.clf()

#%% Project to two first bases
projection = np.matmul(X, sorted_eigenvects)

#%% Plotting the projected data as scatter plot
for i in range(10):
    #get the average vector for the particular digit
    plt.scatter(projection[Y==i,0], projection[Y==i,1], label=i)

#axis labels
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
#title
plt.title('Original Data Projected onto PC1 and PC2')
#add a legend
plt.legend(bbox_to_anchor=(1, 0.9))
#add tight layout to pad edges of plot
plt.tight_layout()
#save figure
plt.savefig(('../Figures/projected_data.png'))
#show figure
plt.show()
#clear figure
plt.clf()

print('End of PCA Script')
print('-'*20)
```