CSE152 HW2

October 22, 2019

```
[ ]: import numpy as np
     import matplotlib.pyplot as plt
     from torch import nn
     import torch
     from torchvision.datasets import MNIST
     import torchvision.transforms as transforms
     from sklearn.metrics import accuracy_score
     %matplotlib inline
```

For this homework you will be using `pytorch` and `torchvision` library for neural networks and datasets. You can install them with `pip install torch torchvision`.

1 Question 1 Principal Component Analysis

This problem will guide you through the principal component analysis. You will be using a classical dataset, the MNIST hand written digit dataset.

```
[ ]: # Load the MNIST dataset
     mnist = MNIST('.', download=True)
     data = mnist.train_data.numpy()
     labels = mnist.train_labels.numpy()
     print('shapes:', data.shape, labels.shape)
     plt.imshow(data[0])
     print('label:', labels[0])
```

1.1 Question 1.1 Familiarize yourself with the data [5pt]

For this task, you will be using the torchvision package that provides the MNIST dataset. For each digit class(0-9), plot 1 image from the class and store those 10 images for each digit class in the array `digit_images`.

```
[ ]: digit_images = np.zeros([10, 28, 28])
     ### YOUR CODE HERE
     ### END OF CODE
```

1.2 Question 1.2 PCA

The following questions will guide you through the PCA algorithm.

1.2.1 Question 1.2.1 Centering the data [5pt]

For each image, flatten it to a 1-D vector. To perform PCA on the dataset, we first move the data points so they have 0 mean on each dimension. Store the centered data in variable `data_centered` and the mean of each dimension in variable `data_mean`.

[ ]:
```
data_centered = None
data_mean = None
### YOUR CODE HERE
### END OF CODE
```

1.2.2 Question 1.2.2 Compute the covariance matrix of the data [5pt]

You need to store the covariance matrix of the data in variable `data_covmat`. You may **not** use numpy.cov

[ ]:
```
data_covmat = None
### YOUR CODE HERE
### END OF CODE
```

1.2.3 Question 1.2.3 Compute the eigenvalues of the covariance matrix [5pt]

You need to store the eigenvalues of the covariance matrix in variable `covmat_eig`, sorted in descending order. Then you need to plot the eigenvalues with `plt.plot`. You can use any numpy function.

[ ]:
```
covmat_eig = None
### YOUR CODE HERE
### END OF CODE
```

1.2.4 Question 1.2.4 Project data onto the first 2 principal components [5pt]

Now you need to project the centered data on the 2D space formed by the eigenvectors corresponding to the 2 largest eigenvalues. Create a 2D scatter plot where you need to assign a unique color to each digit class.

[ ]:
```
### YOUR CODE HERE
### END OF CODE
```

1.2.5 Question 1.2.5 Unproject data back to high dimensions [10pt]

For this question, you need to project the 10 images you plotted in **1.1** on the first 2 principal components, and then unproject the "compressed" 2-D representations back to the original space. Plot the "compressed" digit (the reconstructed digit). Do they look similar to the original images?

[ ]:
```
### YOUR CODE HERE
### END OF CODE
```

1.2.6 Question 1.2.6 Choose a better low dimension space. [5pt]

Do the previous problem with more dimensions (e.g. 3, 5, 10, 20, 50, 100). You only need to show results for one of them. Answer the following questinos. How many dimensions are required to represent the digits reasonably well? How are your results related to **question 1.2.3**?

[ ]:
```
### YOUR CODE HERE
### END OF CODE
```

(Your explanation)

1.3 Question 1.3 Harris Corner and PCA [10pt]

Recall Harris corner detector algorithm: 1. Compute x and y derivatives (I_x, I_y) of an image 2. Compute products of derivatives (I_x^2, I_y^2, I_{xy}) at each pixel 3. Compute matrix M at each pixel, where

$$M(x_0, y_0) = \sum_{x,y} w(x - x_0, y - y_0) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

Here, we set weight $w(x, y)$ to be a box filter of size 3×3 (the box is placed centered at (x_0, y_0)).

In this problem, you need to show that Harris Corner detector is really just principal component analysis in the gradient space. Your explanation should answer the following quesions. 1. As we know, PCA is performed on data points. What are the data points in Harris corner detector when we think of it as a PCA? 2. What is the covariance matrix used in Harris corner detector and why it is a covariance matrix? 3. What are the principal components in Harris corner detector? 4. Briefly explain how principal components imply "cornerness".

(Your proof here)

2 Question 2 KNN, Softmax Regression

[ ]:
```
train_dataset = MNIST(root='.', train=True, transform=transforms.ToTensor,␣
↪download=True)
test_dataset = MNIST('.', train=False, transform=transforms.ToTensor())
train_X = train_dataset.data.numpy()  # training data, uint8 type to reduce␣
↪memory and comparison cost
```

```
train_y = train_dataset.targets.numpy()  # training label
test_X = test_dataset.data.numpy() # testing data, uint8 to reduce memory and␣
↪comparison cost
test_y = test_dataset.targets.numpy()  # testing label
```

[ ]:
```
train_X = train_X.reshape((train_X.shape[0], -1))
test_X = test_X.reshape((test_X.shape[0], -1))
```

2.1 Question 2.1 K-Nearest Neighbor [10pt]

In this problem you will be implementing the KNN classifier. Fill in the functions in the starter code below. You are are allowed to use `scipy.spatial.KDTree` and `scipy.stats.mode` (in case of a tie, pick any one). Please avoid `sklearn.neighbors.KDTree` as it appears extremely slow. You are **not** allowed to use a library KNN function that directly solves the problem.

If you do not know what a KD-tree is, please read the documentation for `scipy.spatial.KDTree` to understand how you can use it.

Note: if you run into memory issues or neighbor queries run for more than 10 minutes, you are allowed to reduce the data size, and explain what you have done to the training data.

[ ]:
```
from scipy.spatial import KDTree
from scipy.stats import mode
```

[ ]:
```
class KNNClassifier:
    def __init__(self, num_neighbors):
        """
        construct the classifier
        Args:
            num_centers: number of neighbors
        """
        ### YOU CODE HERE
        ### END OF CODE

    def fit(self, X, y):
        """
        train KNN classifier
        Args:
            X: training data, numpy array with shape (Nxk) where N is number of␣
↪data points, k is number of features
            y: training labels, numpy array with shape (N)
        """
        ### YOU CODE HERE
        ### END OF CODE
        return self

    def predict(self, X):
```

```
        """
        predict labels
        Args:
            X: testing data, numpy array with shape (Mxk) where M is number of␣
↪data points, k is number of features
        Return:
            y: predicted labels, numpy array with shape (N)
        """
        pred = None
        ### YOU CODE HERE
        ### END OF CODE
        return pred
```

[ ]:
```
from sklearn.metrics import accuracy_score
knn = KNNClassifier(3).fit(train_X, train_y)
pred_y = knn.predict(test_X)
print('KNN accuracy:', accuracy_score(test_y, pred_y))
```

2.2 Question 2.2 Softmax Regression

In this problm, you will be implementing the softmax regression(multi-class logistic regression). Here is a brief recap of several important concepts. In the following explanation, I will use x for data vector, y' for ground truth label, and y for predicted label.

Suppose we have a problem where we need to classify data points into m classes.

1. Softmax function S normalize a vector to have sum 1. (it turns any vector into a probability distribution)
$$S(x) = [\frac{e^{x_1}}{\sum_{j=1}^{m} e^{x_j}}, \frac{e^{x_2}}{\sum_{j=1}^{m} e^{x_j}}, ..., \frac{e^{x_m}}{\sum_{j=1}^{m} e^{x_j}}]$$

2. Cross entropy loss J is the multiclass logistic regression loss.
$$J(y', y) = -\sum_{i=1}^{m} y'_i \log y_i$$
where y' is the one-hot ground truth label and y is the predicted label distribution.

3. Softmax regression is the following optimization problem.
$$\min_{W,b} \sum_{(X,y') \in \{\text{training set}\}} J(y', S(Wx + b))$$
where W has shape $(m \times k)$ where k is the number of features in a data point; b is a m dimensional vector.

4. This objective is optimized with gradient descent. Let
$$L = \sum_{(x,y') \in \{\text{training set}\}} J(y', S(Wx + b))$$
Update W and b with $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$.

2.2.1 Question 2.2.1 Compute the gradients [10pt]

In this question, you need to do the following: 1. Compute the gradient $\frac{\partial J}{\partial y}$. i.e. compute

$$\frac{\partial J}{\partial y_i}$$

Express it in terms of y_i' and y_i. 2. Let $u = Wx + b$, $y_i = S_i(u_j)$ Compute

$$\frac{\partial y_i}{\partial u_j}$$

Express it in terms of y_i, y_j and δ_{ij}, where

$$\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

3. Compute

$$\frac{\partial J}{\partial W_{jk}} \text{ and } \frac{\partial J}{\partial b_j}$$

Express them in terms of y_j, y_j', x_k. Explain your results in an intuitive way. Hint: the results should have a very simple form that makes sense intuitively. 4. Compute

$$\frac{\partial J}{\partial W}$$

in the matrix form. It should be a matrix with the same shape as W, and entry jk is $\frac{\partial J}{\partial W_{jk}}$. Similarly, compute

$$\frac{\partial J}{\partial b}$$

(Your proof here)

2.2.2 Question 2.2.2 Stochastic Gradient Descent [10pt]

In gradient descent algorithm, we update W and b with $\partial L/\partial W$ and $\partial L/\partial b$. However, this requires the gradient w.r.t. the whole dataset. Computing such gradient is very slow. Instead, we can update the weights with per-data gradient. This is known as the SGD algorithm, which runs much faster. You need to take the following steps. 1. Implement softmax function S. We need to take special care in this function since e^x tends to overflow easily. However, we observe that $S(x) = S(x - m)$ for any constant vector m. We can stabilize softmax using $S(x) = S(x - \max(x))$. 2. Implement function `J`(loss) and `dJ`(loss gradient). Note: `J` is not required to run the algorithm, but you may want to implement it for debug purposes. 3. Implement the SGD algorithm. 4. Run the algorithm for 20 epochs (each epoch iterates the whole data set once) with learning rate `1e-3` and report accuracy on test set. You may use `sklearn.metrics.accuracy_score`. You need to achieve accuracy $> 90\%$. You are allowed to experiment with different epoch numbers and learning rates (even learning rate decay) to achieve this accuracy, but they are not required.

You may use print (or progress bar packages) to track the training progress since it might take several minutes.

[ ]:
```
train_dataset = MNIST(root='.', train=True, transform=transforms.ToTensor(),␣
↪download=True)
test_dataset = MNIST('.', train=False, transform=transforms.ToTensor())
train_X = train_dataset.data.numpy() / 255.  # normalize data to 0-1
train_y = train_dataset.targets.numpy()  # training label
test_X = test_dataset.data.numpy() / 255.  # normalize data to 0-1
test_y = test_dataset.targets.numpy()  # testing label
train_X = train_X.reshape((train_X.shape[0], -1))  # flatten the image
test_X = test_X.reshape((test_X.shape[0], -1))  # flatten the image
```

[ ]:
```
def softmax(x):
    """
    softmax function
    Args:
        x: a 1-d numpy array
    Return:
        results of softmax(x)
    """
    ### YOUR CODE HERE
    ### END OF CODE

def J(W, b, y_true, x):
    """
    Softmax Loss function
    Args:
        W: weights (num_classes x num_features)
        b: bias (num_classes)
        y_true: ground truth 1-hot label (num_classes)
        x: input data
    Return:
        J(y', y)
    """
    ### YOUR CODE HERE
    ### END OF CODE

def dJ(W, b, y_true, x):
    """
    Softmax Loss gradient
    Args:
        W: weights (num_classes x num_features)
        b: bias (num_features)
        y_true: ground truth 1-hot label (num_classes)
        x: input data (num_features)
    Return:
        (dW, db): gradient w.r.t. W and b
    """
    ### YOUR CODE HERE
```

```
    ### END OF CODE
```

[ ]:
```
from tqdm import tqdm_notebook
```

[ ]:
```
def SGD(f, df, Xs, ys, n_classes=10, lr=1e-3, max_epoch=20):
    """
    Args:
        f: function to optimize
        df: the gradient of the function
        Xs: input data, numpy array with shape (num_data x num_features)
        ys: true label, numpy array with shape (num_data x num_classes)
        lr: learning rate
        max_epoch: maximum epochs to run SGD
    Return:
        optimal weights and biases
    """
    N, m = Xs.shape
    W = np.random.rand(n_classes, m) - 0.5  # you do not need to change random␣
 ↪initialization
    b = np.random.rand(n_classes) - 0.5

    ### YOUR CODE HERE
    ### END OF CODE
    return W, b
```

[ ]:
```
train_y_onehot = np.zeros((train_y.shape[0], 10))
train_y_onehot[np.arange(len(train_y)), train_y] = 1
W, b = SGD(J, dJ, train_X, train_y_onehot, 10, max_epoch=20)
accuracy_score(test_y, np.argmax(test_X @ W.T + b, axis=1))
```

3 Question 3 Convolutional Neural Networks

This question requires you to use the PyTorch framework for neural network training. You will not need GPU to train the networks for this problem.

The following is a code sample for training a simple multi-layer perceptron neural network using PyTorch. Running it should give you about 98% testing accuracy.

Since network training takes long, I recommend installing the tqdm package for progress tracking.

[ ]:
```
from tqdm import tqdm_notebook
```

[ ]:
```
train_dataset = MNIST(root='.', train=True, transform=transforms.ToTensor(),␣
 ↪download=True)
test_dataset = MNIST('.', train=False, transform=transforms.ToTensor())
```

[ ]:
```
class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        """init function builds the required layers"""
        super(MLP, self).__init__()  # This line is always required
        # Hidden layer
        self.layer1 = nn.Linear(input_size, hidden_size)
        # activation
        self.relu = nn.ReLU()
        # output layer
        self.layer2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        """forward function describes how input tensor is transformed to output␣
↪tensor"""
        # flatten the input from (Nx1x28x28) to (Nx784)
        torch.flatten(x, 1)
        x = self.layer1(x)
        x = self.relu(x)
        x = self.layer2(x)
        # Note we do not need softmax layer, since this layer is included in␣
↪the CrossEntropyLoss provided by torch
        return x
```

[ ]:
```
model = MLP(784, 1024, 10)
model
```

[ ]:
```
opts = {
    'lr': 5e-4,
    'epochs': 5,
    'batch_size': 64
}
```

[ ]:
```
optimizer = torch.optim.Adam(model.parameters(), opts['lr'])  # Adam is a much␣
↪better optimizer compared to SGD
criterion = torch.nn.CrossEntropyLoss()  # loss function
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,␣
↪batch_size=opts['batch_size'], shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset,␣
↪batch_size=opts['batch_size'], shuffle=True)
```

[ ]:
```
for epoch in range(opts['epochs']):
    train_loss = []
    for i, (data, labels) in tqdm_notebook(enumerate(train_loader),␣
↪total=len(train_loader)):
        # reshape data
        data = data.reshape([-1, 784])
        # pass data through network
```

```
        outputs = model(data)
        loss = criterion(outputs, labels)
        optimizer.zero_grad()  # Important! Otherwise the optimizer will␣
↪accumulate gradients from previous runs!
        loss.backward()
        optimizer.step()
        train_loss.append(loss.item())
    test_loss = []
    test_accuracy = []
    for i, (data, labels) in enumerate(test_loader):
        # reshape data
        data = data.reshape([-1, 784])
        # pass data through network
        outputs = model(data)
        _, predicted = torch.max(outputs.data, 1)
        loss = criterion(outputs, labels)
        test_loss.append(loss.item())
        test_accuracy.append((predicted == labels).sum().item() / predicted.
↪size(0))
    print('epoch: {}, train loss: {}, test loss: {}, test accuracy: {}'.
↪format(epoch, np.mean(train_loss), np.mean(test_loss), np.
↪mean(test_accuracy)))
```

3.1 Question 3.1 Implementing CNN [15pt]

You need to implement a convolutional neural network for the same task as above. You may find the PyTorch documentation helpful. https://pytorch.org/docs/stable/nn.html

We provide a working network structure below. You can adjust the network size and training options for better performance, but a correct implementation of the provided network should give you the required accuracy. For convolutional layers, (conv MxM, N) means the layer has kernel size M by M and N output channels; for pooling layers, (maxpool MxM) means doing max pooling with kernel size M by M.

(conv 5x5, 32) -> (relu) -> (maxpool 2x2) -> (conv 5x5, 64) -> (relu) -> (maxpool 2x2) -> (flatten) -> (linear 10) -> (output)

For full score, you need to achieve 99% testing accuracy. Also, plot the hand-written digits that your network got wrong.

[ ]:
```
class CNN(nn.Module):
    def __init__(self, input_size, num_classes):
        """
        init convolution and activation layers
        Args:
            input_size: (1,28,28)
            num_classes: 10
```

```
        """
        super(CNN, self).__init__()
        ### YOUR CODE HERE
        ### END OF CODE


    def forward(self, x):
        """
        forward function describes how input tensor is transformed to output␣
↪tensor
        Args:
            x: (Nx1x28x28) tensor
        """
        ### YOUR CODE HERE
        ### END OF CODE
        return x
```

[ ]:
```
model = CNN((1, 28, 28), 10)
model
```

[ ]:
```
### You may (and should) change these
opts = {
    'lr': 1e-3,
    'epochs': 20,
    'batch_size': 64
}

### if you cannot get 99% with SGD, Adam optimizer can help you
optimizer = torch.optim.Adam(model.parameters(), opts['lr'])
```

[ ]:
```
criterion = torch.nn.CrossEntropyLoss()  # loss function
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,␣
↪batch_size=opts['batch_size'], shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset,␣
↪batch_size=opts['batch_size'], shuffle=True)
```

[ ]:
```
for epoch in range(opts['epochs']):
    train_loss = []
    for i, (data, labels) in tqdm_notebook(enumerate(train_loader),␣
↪total=len(train_loader)):
        # pass data through network
        outputs = model(data)
        loss = criterion(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        train_loss.append(loss.item())
```

```
    test_loss = []
    test_accuracy = []
    for i, (data, labels) in enumerate(test_loader):
        # pass data through network
        outputs = model(data)
        _, predicted = torch.max(outputs.data, 1)
        loss = criterion(outputs, labels)
        test_loss.append(loss.item())
        test_accuracy.append((predicted == labels).sum().item() / predicted.
↪size(0))
    print('epoch: {}, train loss: {}, test loss: {}, test accuracy: {}'.
↪format(epoch, np.mean(train_loss), np.mean(test_loss), np.
↪mean(test_accuracy)))
```

Don't forget plotting the digits that the network got wrong.

3.2 Question 3.2 Kernel weights visualization [5pt]

For this question, you need to visualize the kernel weights for your first convolutional layer. Suppose you have 5x5 kernels with 32 output channels. You will plot 32 5x5 images.

hint: You might need to look at PyTorch documentation (or play with the PyTorch model) to figure out how to get the weights.

[ ]:
```
### YOUR CODE HERE
### END OF CODE
```