HW4

March 5, 2020

1 CSE 152: Intro to Computer Vision - Winter 2020 Assignment 4

1.1 Instructor: David Kriegman

1.1.1 Assignment published on Thursday, March 5, 2020

1.1.2 Due on Wednesday, March 18, 2020 at 11:59pm

1.2 Instructions

- This assignment must be completed individually. Review the academic integrity and collaboration policies on the course website.
- All solutions should be written in this notebook. Show your work for written questions.
- If you want to modify the skeleton code, you may do so. It has been merely been provided as a framework for your solution.
- You may use Python packages for basic linear algebra (e.g. NumPy or SciPy for basic operations), but you may not use packages that directly solve the problem. If you are unsure about using a specific package or function, ask the instructor and/or teaching assistants for clarification.
- You must submit this notebook exported as a PDF. You must also submit this notebook as an `.ipynb` file. Submit both files (`.pdf` and `.ipynb`) on Gradescope. **You must mark the PDF pages associated with each question in Gradescope. If you fail to do so, we may dock points.**
- It is highly recommended that you begin working on this assignment early.
- **Late policy:** a penalty of 10% per day after the due date.

1.3 Problem 1: Machine Learning [15 pts]

In this problem, you will implement K-Nearest Neighbors (KNN) algorithm for computer vision problems.

1.3.1 Part 1: Data preparation [1 pts]

Download the MNIST data from http://yann.lecun.com/exdb/mnist/.

Download the 4 zipped files, extract them into one folder, and change the variable 'path' in the code below. (Code taken from https://gist.github.com/akesling/5358964)

Plot one random example image corresponding to each label from the training data.

[ ]:
```python
import os
import struct
import numpy as np

# Change path as required
path = "./mnist_data/"

def read(dataset="training", datatype='images'):
    """
    Python function for importing the MNIST data set.  It returns an iterator
    of 2-tuples with the first element being the label and the second element
    being a numpy.uint8 2D array of pixel data for the given image.
    """

    if dataset is "training":
        fname_img = os.path.join(path, 'train-images.idx3-ubyte')
        fname_lbl = os.path.join(path, 'train-labels.idx1-ubyte')
    elif dataset is "testing":
        fname_img = os.path.join(path, 't10k-images.idx3-ubyte')
        fname_lbl = os.path.join(path, 't10k-labels.idx1-ubyte')

    # Load everything in some numpy arrays
    with open(fname_lbl, 'rb') as flbl:
        magic, num = struct.unpack(">II", flbl.read(8))
        lbl = np.fromfile(flbl, dtype=np.int8)

    with open(fname_img, 'rb') as fimg:
        magic, num, rows, cols = struct.unpack(">IIII", fimg.read(16))
        img = np.fromfile(fimg, dtype=np.uint8).reshape(len(lbl), rows, cols)

    if(datatype=='images'):
        get_data = lambda idx: img[idx]
    elif(datatype=='labels'):
        get_data = lambda idx: lbl[idx]

    # Create an iterator which returns each image in turn
    for i in range(len(lbl)):
        yield get_data(i)

trainData=np.array(list(read('training','images')))
trainLabels=np.array(list(read('training','labels')))
testData=np.array(list(read('testing','images')))
testLabels=np.array(list(read('testing','labels')))
```

Some helper functions are given below.

[ ]:
```python
# a generator for batches of data
# yields data (batchsize, 3, 32, 32) and labels (batchsize)
# if shuffle, it will load batches in a random order
import matplotlib.pyplot as plt
def DataBatch(data, label, batchsize, shuffle=True):
    n = data.shape[0]
    if shuffle:
        index = np.random.permutation(n)
    else:
        index = np.arange(n)
    for i in range(int(np.ceil(n/batchsize))):
        inds = index[i*batchsize : min(n,(i+1)*batchsize)]
        yield data[inds], label[inds]

# tests the accuracy of a classifier
def test(testData, testLabels, classifier):
    batchsize=50
    correct=0.
    for data,label in DataBatch(testData,testLabels,batchsize,shuffle=False):
        prediction = classifier(data)
        correct += np.sum(prediction==label)
    return correct/testData.shape[0]*100

# a sample classifier
# given an input it outputs a random class
class RandomClassifier():
    def __init__(self, classes=10):
        self.classes=classes
    def __call__(self, x):
        return np.random.randint(self.classes, size=x.shape[0])

randomClassifier = RandomClassifier()
print('Random classifier accuracy: %f \n' %
      test(testData, testLabels, randomClassifier))

print('Plot random training images for each class:')
count = 0
check = np.zeros(10)
imgs = np.zeros((10, 28, 28))
for img, lb in DataBatch(trainData, trainLabels, 1, shuffle=True):
    img = np.squeeze(img)
    if check[lb] == 1:
        continue
    else:
        check[lb] += 1
        count += 1
        imgs[lb,:,:] = img
    if count == 10:
        break
```

```
fig, ax = plt.subplots(nrows=2, ncols=5)
i = 0
for row in ax:
    for col in row:
        col.imshow(imgs[i,:,:])
        i += 1

plt.show()
```

1.3.2 Part 2: Confusion Matrix [3 pts]

Here you will implement a function that computes the confusion matrix for a classifier. The matrix (M) should be nxn where n is the number of classes. Entry M[i,j] should contain the fraction of images of class i that were classified as class j.

[ ]:
```
# Using the tqdm module to visualize run time is suggested
from tqdm import tqdm
import time

# It would be a good idea to return the accuracy, along with the confusion
# matrix, since both can be calculated in one iteration over test data, to
# save time
def Confusion(testData, testLabels, classifier):
    '''
    Your code here
    '''
    M = np.zeros((10,10))

    return M, accuracy

def VisualizeConfusion(M):
    plt.figure(figsize=(14, 6))
    plt.imshow(M)
    plt.show()
    print(np.round(M,2))

M, _ = Confusion(testData, testLabels, randomClassifier)
VisualizeConfusion(M)
```

1.3.3 Part 3: K-Nearest Neighbors (KNN) [6 pts]

- Here you will implement a simple knn classifier. The distance metric is Euclidean in pixel space. k refers to the number of neighbors involved in voting on the class, and should be 3. You are allowed to use sklearn.neighbors.KNeighborsClassifier.
- Display the confusion matrix and accuracy for your KNN classifier trained on the entire training dataset. (should be ~97%)

- After evaluating the classifier on the test set, based on the confusion matrix, mention the number that the number '7' is most often predicted to be, other than '7'. Write your comment below.

[ ]:
```
from sklearn.neighbors import KNeighborsClassifier
import numpy as np
import matplotlib.pyplot as plt
class KNNClassifer():
    def __init__(self, k=3):
        # k is the number of neighbors involved in voting
        '''
        your code here
        '''

    def train(self, trainData, trainLabels):
        '''
        your code here
        '''

    def __call__(self, x):
        # this method should take a batch of images
        # and return a batch of predictions
        '''
        your code here
        '''

# test your classifier with only the first 100 training examples (use this
# while debugging)
# note you should get ~ 65 % accuracy
knnClassiferX = KNNClassifer()
knnClassiferX.train(trainData[:100], trainLabels[:100])
print ('KNN classifier accuracy: %f'%test(testData, testLabels, knnClassiferX))
```

[ ]:
```
# test your classifier trained with all the training examples (This may take a␣
 ↪while)
knnClassifer = KNNClassifer()
knnClassifer.train(trainData[:-1], trainLabels[:-1])

# display confusion matrix and testing accuracy for your KNN classifier trained␣
 ↪with all the training examples
'''
your code here
'''
```

1.3.4 Comment:

Your comment here.

3.1 : Testing performance with different number of Neighbors Plot the overall accuracy of the classifier for k = [1,3,5,7,15]. Comment on your results. Choose the value of k that you feel is the best and use this value for the next question.

[ ]:
```
'''
Your code to plot the accuracy
for a variable number of nearest neighbours.
'''
```

1.3.5 Part 4: Principal Component Analysis (PCA) K-Nearest Neighbors (KNN) [5 pts]

Here you will implement a simple KNN classifer in PCA space (for k= the value you found best, and a variable number of principal components). You are allowed to use sklearn.decomposition.PCA or any other package that directly implements PCA transformations. For more information refer https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

Is the testing time for PCA KNN classifier more or less than that for KNN classifier? Comment on why it differs if it does.

[ ]:
```
class PCAKNNClassifer():
    def __init__(self, components=25, k= CHANGEME):
        # components = number of principal components
        # k is the number of neighbors involved in voting
        """ ==========
        YOUR CODE HERE
        ========== """

    def train(self, trainData, trainLabels):
        """ ==========
        YOUR CODE HERE
        ========== """
        '''
        For optimal results make sure your training data has 0 mean
        by computing the mean of the training vectors and subtracting
        the mean vector from each of the samples. (Very similar to the␣
↪procedure
        you followed in the template matching problem. (HW1 Q 3.3))
        '''

    def __call__(self, x):
        # this method should take a batch of images
        # and return a batch of predictions
        """ ==========
        YOUR CODE HERE
        ========== """
        '''
        If you have centered your training data by subtracting the mean
        make sure you subtract the same mean from your test data as well.
```

```
      '''



# test your classifier with only the first 100 training examples (use this
# while debugging)
pcaknnClassiferX = PCAKNNClassifer()
pcaknnClassiferX.train(trainData[:100], trainLabels[:100])
print ('KNN classifier accuracy: %f'%test(testData, testLabels,␣
 ↪pcaknnClassiferX))
```

[ ]:
```
# test your classifier with all the training examples
pcaknnClassifer = PCAKNNClassifer()
pcaknnClassifer.train(trainData, trainLabels)
```

[ ]:
```
# display confusion matrix for your PCA KNN classifier with all the training␣
 ↪examples
""" ==========
YOUR CODE HERE
========== """
```

4.1 : Testing performance with different number of PCA components. Plot the overall classification accuracy and display the confusion matrix for k = value chosen by you, and principal components = [1 , 10 , 25, 50]. For Principal components = 25, output the sensitivity and specificity of the classifier for digit 0. (Please look up the terms sensitivity and specificity for details on their formulas.) Comment on your results for sensitivity and specificity by mentioning the meaning of these terms. Suppose the classifier is used to detect 0's but it turns out that processing 0's is very expensive and hence a digit should be classified as 0 only if we are very sure. Which term (sensitivity or specificity) should we focus on optimizing?
Reference : https://en.wikipedia.org/wiki/Sensitivity_and_specificity

[ ]:
```
'''
Your code here for the variable number of principal components
and plotting the confusion matrix, and overall accuracy.
MAKE SURE ALL YOUR PLOTS HAVE TITLES AND/OR MARKDOWN CELLS CLEARLY
INDICATING WHAT EACH FIGURE CORRESPONDS TO.
'''
```

Comments: Your comments

1.4 Problem 2: Deep Learning [25 pts]

1.4.1 Part 1: Initial setup [0 pts]

Follow the directions on https://pytorch.org/get-started/locally/ to install PyTorch on your computer.

Note: You will not need GPU support for this assignment so don't worry if you don't have one. In any case, installing with GPU support is often more difficult to configure, so it is suggested that you install the CPU-only version regardless.

To ensure that PyTorch was installed correctly, we will now verify the installation by running some sample PyTorch code. Here we construct a randomly initialized tensor.

[ ]:
```
from __future__ import print_function
import torch
x = torch.rand(5, 3)
print(x)
```

1.4.2 Part 2: Training with PyTorch [3 pts]

Below is some helper code to train your deep networks. Complete the train function for PTClassifier below. You should write down the training operations in this function. This function will be used in the following questions with different networks. You can look at https://pytorch.org/tutorials/beginner/pytorch_with_examples.html for reference.

[ ]:
```
# base class for your PyTorch networks. It implements the training loop
# (__init__), (train) and prediction(__call__)  for you.
# You will need to complete the (train) function to define the training␣
 ↪operations
# structures in the following problems.
import torch.nn as nn
from torch.nn.parameter import Parameter
import torch.nn.functional as F
import torch.nn.init
import torch.optim as optim
from torch.autograd import Variable
from tqdm import tqdm
from scipy.stats import truncnorm

class PTClassifier():
    def __init__(self, net):
        self.net = net()

    def train(self, trainData, trainLabels, testData, testLabels, epochs=1,␣
 ↪batchsize=50):
        criterion = nn.CrossEntropyLoss()
        learning_rate=3e-4
        optimizer = optim.Adam(self.net.parameters(),lr=learning_rate)
        for epoch in range(epochs):
            for i, (data,label) in enumerate(DataBatch(trainData, trainLabels,␣
 ↪batchsize, shuffle=True)):
                inputs = Variable(torch.FloatTensor(data))
                targets = Variable(torch.LongTensor(label))

                # YOUR CODE HERE
                # Train the model using the optimizer and the batch data
            print ('Epoch:%d Accuracy: %f'%(epoch+1, test(testData, testLabels,␣
↪self)))

    def __call__(self, x):
        inputs = Variable(torch.FloatTensor(x))
        prediction = self.net(inputs)
        return np.argmax(prediction.data.cpu().numpy(), 1)

    def get_first_layer_weights(self):
        return self.net.weight1.data.cpu().numpy()

# helper function to get weight variable
def weight_variable(shape):
    initial = torch.Tensor(truncnorm.rvs(-1/0.01, 1/0.01, scale=0.01,␣
↪size=shape))
    return Parameter(initial, requires_grad=True)

# helper function to get bias variable
def bias_variable(shape):
    initial = torch.Tensor(np.ones(shape)*0.1)
    return Parameter(initial, requires_grad=True)

# Define Single Layer Perceptron network
class SLP(nn.Module):
    def __init__(self, in_features=28*28, classes=10):
        super(SLP, self).__init__()
        # model variables
        self.weight1 = weight_variable((classes, in_features))
        self.bias1 = bias_variable((classes))

    def forward(self, x):
        # linear operation
        y_pred = torch.addmm(self.bias1, x.view(list(x.size())[0], -1), self.
↪weight1.t())
        return y_pred


# test the example linear classifier (note you should get around 92% accuracy
# for 10 epochs and batchsize 50)
trainData=np.array(list(read('training','images')))
trainData=np.float32(np.expand_dims(trainData,-1))/255
trainData=trainData.transpose((0,3,1,2))
trainLabels=np.int32(np.array(list(read('training','labels'))))

testData=np.array(list(read('testing','images')))
testData=np.float32(np.expand_dims(testData,-1))/255
testData=testData.transpose((0,3,1,2))
```

```
testLabels=np.int32(np.array(list(read('testing','labels'))))

linearClassifier = PTClassifier(SLP)
linearClassifier.train(trainData, trainLabels, testData, testLabels, epochs=10)
print ('Linear classifier accuracy: %f'%test(testData, testLabels,␣
 ↪linearClassifier))
```

1.4.3 Part 3: Single Layer Perceptron [3 pts]

The simple linear classifier implemented in the cell already performs quite well. Plot the filter weights corresponding to each output class (weights, not biases) as images. (Normalize weights to lie between 0 and 1 and use color maps like 'inferno' or 'plasma' for good results). Comment on what the weights look like and why that may be so.

[ ]:
```
# Your code here.
```

1.4.4 Comment

Your comment here.

1.4.5 Part 4: Multi Layer Perceptron (MLP) [7 pts]

Here you will implement an MLP. The MLP shoud consist of 2 layers (matrix multiplication and bias offset) that map to the following feature dimensions:

- 28x28 -> hidden (100)
- hidden -> classes
- The hidden layer should be followed with a ReLU nonlinearity. The final layer should not have a nonlinearity applied as we desire the raw logits output.
- The final output of the computation graph should be stored in self.y as that will be used in the training.

Display the confusion matrix and accuracy after training. Note: You should get around 97% accuracy for 10 epochs and batch size 50.

Plot the filter weights corresponding to the mapping from the inputs to the first 10 hidden layer outputs (out of 100). Do the weights look similar to the weights plotted in the previous problem? Why or why not?

[ ]:
```
# Define Multi Layer Perceptron network
class MLP(nn.Module):
    def __init__(self, in_features=28*28, hidden=100, classes=10):
        super(MLP, self).__init__()
        '''
        your code here
        '''


    def forward(self, x):
        '''
        your code here
        '''

mlpClassifer = PTClassifier(MLP)
mlpClassifer.train(trainData, trainLabels, testData, testLabels, epochs=10)
```

[ ]:
```
# Confusion Matrix and Accuracy
'''
your code here
'''
```

[ ]:
```
# plot filter weights
'''
your code here
'''
```

1.4.6 Comment

Your comment here.

1.4.7 Part 5: Convolutional Neural Network (CNN) [12 pts]

Here we will implement the classic LeNET network with the following architecture:

- a convolutional layer connecting the input image to 6 feature maps with 5 Œ 5 convolutions and followed by ReLU and maxpooling,
- a convolutional layer connecting the 6 input channels to 16 output channels with 5 Œ 5 convolutions and followed by ReLU and maxpooling
- a fully-connected layer connecting 16 feature maps to 120 output units and followed by ReLU,
- a fully-connected layer connecting 120 inputs to 84 output units and followed by ReLU,
- a final linear layer connecting 84 inputs to 10 linear outputs (one for each of our digits).

Display the confusion matrix and accuracy after training. You should get around 98% accuracy for 10 epochs and batch size 100.

** START EARLY. RUNNING THIS CODE WILL TAKE ABOUT AN HOUR **

[ ]:
```
def conv2d(x, W, stride):
    # x: input
    # W: weights (out, in, kH, kW)
    return F.conv2d(x, W, stride=stride, padding=1)

# Define Convolutional Neural Network
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        '''
        your code here
        '''
    def forward(self, x):
        '''
        your code here
        '''
    #Determine the number of features in a batch of tensors
    def num_flat_features(self, x):
        size = x.size()[1:]
        return np.prod(size)

cnnClassifer = PTClassifier(LeNet)
cnnClassifer.train(trainData, trainLabels, testData, testLabels, epochs=10)
```

[ ]:
```
# Confusion Matrix and Accuracy
'''
your code here
'''
```

- Note that the MLP/ConvNet approaches lead to an accuracy a little higher than the K-NN approach.
- In general, neural net approaches lead to a significant increase in accuracy, but in this case the problem is not too hard, so the increase in accuracy will not be very high.
- However, this is still quite significant considering the fact that the ConvNets we've used are relatively simple while the accuracy achieved using K-NN is with a search over 60,000 training images for every test image.
- You can look at the performance of various machine learning methods on this problem at http://yann.lecun.com/exdb/mnist/
- You can learn more about PyTorch at https://pytorch.org/tutorials/index.html
- You can find another image classifier training example at https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#sphx-glr-beginner-blitz-cifar10-tutorial-py
- You can play with a demo of neural network created by Daniel Smilkov and Shan Carter at https://playground.tensorflow.org/

[ ]:
```

```