HW4

December 1, 2020

1 CSE 252A Computer Vision I Fall 2020 - Assignment 4

1.1 Instructor: David Kriegman

1.1.1 Assignment Published On: Tuesday, December 1, 2020

1.1.2 Due On: Friday, December 11, 2020 11:59 pm

1.2 Instructions

- Review the academic integrity and collaboration policies on the course website.
- This assignment must be completed individually.
- This assignment contains theoretical and programming exercises. If you plan to submit handwritten answers for the theoretical exercises, please be sure that your writing is readable (illegible answers will not be given the benefit of the doubt!) and merge your handwritten solutions in problem order with the PDF that you create from this notebook. You can also write the answers within the notebook itself by creating Markdown cells.
- Programming aspects of this assignment must be completed using Python in this notebook.
- If you want to modify the skeleton code, you can do so. The existing code is merely meant to provide you with a framework for your solution.
- You may use Python packages for basic linear algebra (you can use 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, then ask the instructor and teaching assistants for clarification.
- You must submit to Gradescope:
  - (1) This notebook exported as a `.pdf` file (including any handwritten solutions scanned and merged into the PDF, if applicable).
  - (2) This notebook as an `.ipynb` file.
- You must select the pages associated with each problem on Gradescope (for your PDF submission).
- **Late policy:** Assignments submitted late will receive a 10% grade reduction for each day late (e.g. an assignment submitted an hour after the due date will receive a 10% penalty, an assignment submitted 10 hours after the due date will receive a 10% penalty, and an assignment submitted 28 hours after the due date will receive a 20% penalty). Assignments will not be accepted 72 hours after the due date. If you require an extension (for personal reasons only), you must request one as far in advance as possible. Extensions requested close to or after the due date will only be granted for clear emergencies or clearly unforeseeable circumstances.

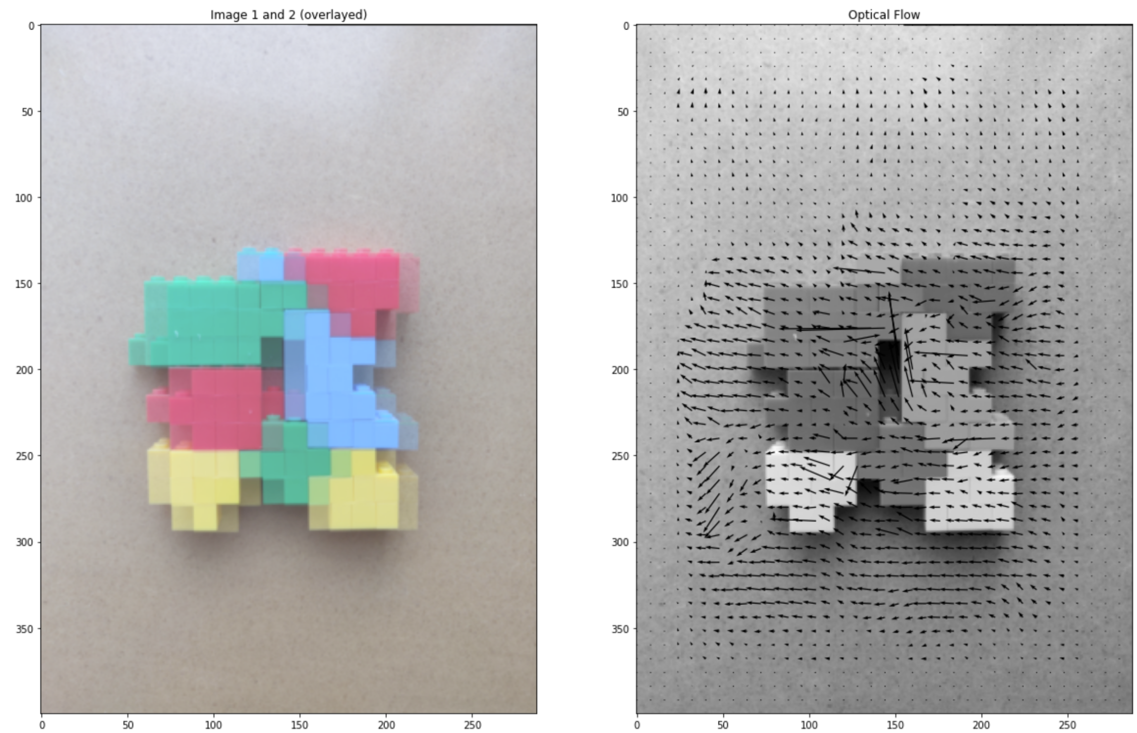
1.3 Problem 1: Optical Flow [14 pts]

In this problem, the multi-resolution Lucas-Kanade algorithm for estimating optical flow will be implemented, and the data needed for this problem can be found in the folder 'optical_flow_images'.

An example optical flow output is shown below - this is not a solution, just an example output.

Note 1: You can choose to implement a single scale version of Lucas-Kanade instead of multi-scale for partial credit in part 1.

Note 2: You are free to use either multi-scale or single-scale version to complete part 3 and part 4 without incurring any penalty.



1.3.1 Part 1: Multi-resolution Lucas-Kanade implementation [6 pts]

Implement the Lucas-Kanade method for estimating optical flow. The function 'LucasKanadeMultiScale' needs to be completed. You can use 'upsample_flow' and 'OpticalFlowRefine' as 2 building blocks in order to complete this.

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

from scipy.signal import convolve, convolve2d
# from tqdm import tqdm_notebook

def grayscale(img):
    '''
    Converts RGB image to Grayscale
```

```python
    '''
    gray=np.zeros((img.shape[0],img.shape[1]))
    gray=img[:,:,0]*0.2989+img[:,:,1]*0.5870+img[:,:,2]*0.1140
    return gray

def plot_optical_flow(img0,img1,U,V,titleStr, color=False):
    '''
    Plots optical flow given U,V and the images
    '''

    # Change t if required, affects the number of arrows
    # t should be between 1 and min(U.shape[0],U.shape[1])
    t=8

    # Subsample U and V to get visually pleasing output
    U1 = U[::t,::t]
    V1 = V[::t,::t]

    # Create meshgrid of subsampled coordinates
    r, c = img0.shape[0],img0.shape[1]
    cols,rows = np.meshgrid(np.linspace(0,c-1,c), np.linspace(0,r-1,r))
    cols = cols[::t,::t]
    rows = rows[::t,::t]

    # Plot optical flow
    plt.figure(figsize=(20,20))
    plt.subplot(121)
    plt.imshow(img0, alpha=0.5)
    plt.imshow(img1, alpha=0.5)
    plt.title('Overlayed Images')
    plt.subplot(122)
    if color:
        plt.imshow(img0)
    else:
        plt.imshow(grayscale(img0), cmap='gray')
    plt.quiver(cols,rows,U1,V1)
    plt.title(titleStr)
    plt.show()


images=[]
for i in range(1,5):
    images.append(plt.imread('optical_flow_images/im'+str(i)+'.png')[:,:288,:])
# each image after converting to gray scale is of size -> 400x288
```

[ ]:
```python
# you can use interpolate from scipy
# You can implement 'upsample_flow' and 'OpticalFlowRefine'
```

```
# as 2 building blocks in order to complete this.
import scipy.misc
from skimage.transform import resize
def upsample_flow(u_prev, v_prev):
    ''' You may implement this method to upsample optical flow from
    previous level
    u_prev, v_prev -> optical flow from prev level
    u, v -> upsampled optical flow to the current level
    '''
    if u_prev is None and v_prev is None:
        return u_prev, v_prev
    u = resize(u_prev,(u_prev.shape[0]*2,u_prev.shape[1]*2),order=1)
    v = resize(v_prev,(u_prev.shape[0]*2,u_prev.shape[1]*2),order=1)
    u = u*2;
    v = v*2;
    return u, v
```

[ ]:
```
def OpticalFlowRefine(im1,im2,window, u_prev=None, v_prev=None):
    '''
    Inputs: the two images at current level and window size
    u_prev, v_prev - previous levels optical flow
    Return u,v - optical flow at current level
    '''
    """ ==========
    YOUR CODE HERE
    ========== """
    return u, v
```

[ ]:
```
def LucasKanadeMultiScale(im1,im2,window, numLevels=2):
    '''
    Implement the multi-resolution Lucas kanade algorithm
    Inputs: the two images, window size and number of levels
    if numLevels = 1, then compute optical flow at only the given image level.
    Returns: u, v - the optical flow
    '''

    """ ==========
    YOUR CODE HERE
    ========== """
    # You can call OpticalFlowRefine iteratively
    return u, v
```

1.3.2 Part 2: Number of levels [2 pts]

Plot optical flow for the pair of images im1 and im2 for different number of levels mentioned below. Comment on the results and justify. (i) window size = 13, numLevels = 1 (ii) window size = 13, numLevels = 3 (iii) window size = 13, numLevels = 5 So, you are expected to provide 3 outputs here

Note: if numLevels = 1, then it means the optical flow is only computed at the image resolution i.e. no downsampling

[ ]:
```
# Example code to generate output
window=13
numLevels=1
U,V=LucasKanadeMultiScale(grayscale(images[0]),grayscale(images[1]),\
                          window,numLevels)
plot_optical_flow(images[0],images[1],U,V, \
                  'levels = ' + str(numLevels) + ', window = '+str(window))

numLevels=3
# Plot
U,V=LucasKanadeMultiScale(grayscale(images[0]),grayscale(images[1]),\
                          window,numLevels)
plot_optical_flow(images[0],images[1],U,V, \
                  'levels = ' + str(numLevels) + ', window = '+str(window))

numLevels=5
# Plot
U,V=LucasKanadeMultiScale(grayscale(images[0]),grayscale(images[1]),\
                          window,numLevels)
plot_optical_flow(images[0],images[1],U,V, \
                  'levels = ' + str(numLevels) + ', window = '+str(window))
```

Your Comments on the results of Part 2:

1.3.3 Part 3: Window size [3 pts]

Plot optical flow for the pair of images im1 and im2 for at least 3 different window sizes which leads to observable difference in the results. Comment on the effect of window size on results and justify. For this part fix the number of levels to be 3 for multi-scale Lucas-Kanade.

[ ]:
```
# Example code, change as required
numLevels=3

w1, w2, w3 = 15, 25, 35
for window in [w1, w2, w3]:
    U,V=LucasKanadeMultiScale(grayscale(images[0]),grayscale(images[1]),\
                              window,numLevels)
    plot_optical_flow(images[0],images[1], U,V, \
                      'levels = ' + str(numLevels) + ', window = '+str(window))
```

Your Comments on the results of Part 3:

1.3.4 Part 4: Other pairs [3 pts]

Find optical flow for the pairs (im1,im2), (im1,im3), (im1,im4) using one good window size and number of levels. Does the optical flow result seem consistent with visual inspection? Comment on the type of motion indicated by results and visual inspection and explain why they might be consistent or inconsistent.

[ ]:
```
# Your code here
# use one fixed window and numLevels for all pairs
# Example code to generate output
window=
numLevels=
U,V=LucasKanadeMultiScale(grayscale(images[0]),grayscale(images[1]),\
                              window,numLevels)
plot_optical_flow(images[0],images[1],U,V, \
                     'levels = ' + str(numLevels) + ', window = '+str(window) + ',␣
 ↪image 1 and image 2')


U,V=LucasKanadeMultiScale(grayscale(images[0]),grayscale(images[2]),\
                              window,numLevels)
plot_optical_flow(images[0],images[2],U,V, \
                     'levels = ' + str(numLevels) + ', window = '+str(window) + ',␣
 ↪image 1 and image 3')


U,V=LucasKanadeMultiScale(grayscale(images[0]),grayscale(images[3]),\
                              window,numLevels)
plot_optical_flow(images[0],images[3],U,V, \
                     'levels = ' + str(numLevels) + ', window = '+str(window) + ',␣
 ↪image 1 and image 4')
```

Your Comments on the results of Part 4:

1.4 Problem 2: RGB Optical Flow[10 pts]

In this problem, we extend optical flow to RGB images.

1.4.1 Part 1: Derivation [5 pts]

In lecture, we used brightness constancy constraint and taylor series expansion to come up with the optical flow equation

$$I_x u + I_y v + I_t = 0$$

and we derived the Lucas-Kanade least squares solution assuming same velocity over a patch as:

$$\begin{pmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = - \begin{pmatrix} \sum I_x I_t \\ \sum I_y I_t \end{pmatrix}$$

Derive similar equation for RGB images under a color constancy constraint by applying the brightness constancy constraint to each of the channels, and formulating a cost function over the thee channels.

Your derivation here:

1.4.2 Part 2: Computing Flow [4pts]

Complete the functions LucasKanadeMultiScaleRGB and OpticalFlowRefineRGB and run it for window size = 13 and number of levels = 3

Note: You can implement the single scale version without incurring any penalty.

[ ]:
```
def OpticalFlowRefineRGB(im1,im2,window, u_prev=None, v_prev=None):
    '''
    Inputs: the two RGB images at current level and window size
    u_prev, v_prev - previous levels optical flow
    Return u,v - optical flow at current level
    '''
    """ ==========
    YOUR CODE HERE
    ========== """
    return u, v
```

[ ]:
```
def LucasKanadeMultiScaleRGB(im1,im2,window, numLevels=2):
    '''
    Implement the multi-resolution Lucas kanade algorithm for RGB images
    Inputs: the two RGB images, window size and number of levels
    if numLevels = 1, then compute optical flow at only the given image level.
    Returns: u, v - the optical flow
    '''

    """ ==========
    YOUR CODE HERE
    ========== """
    # You can call OpticalFlowRefineRGB iteratively
    return u_prev, v_prev
```

[ ]:
```
# Your code here
# use one fixed window and numLevels for all pairs
# Example code to generate output
window=13
numLevels=3
U,V=LucasKanadeMultiScaleRGB(images[0][:,:,:3],images[1][:,:,:3],\
                          window,numLevels)
plot_optical_flow(images[0],images[1],U,V, \
                   'levels = ' + str(numLevels) + ', window = '+str(window) + ',␣
 ↪image 1 and image 2', color=True)
```

```
U,V=LucasKanadeMultiScale(grayscale(images[0]),grayscale(images[1]),\
                          window,numLevels)
plot_optical_flow(images[0],images[1],U,V, \
                  'levels = ' + str(numLevels) + ', window = '+str(window) + ',␣
 ↪image 1 and image 2 grayscale')



U,V=LucasKanadeMultiScaleRGB(images[0][:,:,:3],images[2][:,:,:3],\
                          window,numLevels)
plot_optical_flow(images[0],images[2],U,V, \
                  'levels = ' + str(numLevels) + ', window = '+str(window) + ',␣
 ↪image 1 and image 3', color=True)

U,V=LucasKanadeMultiScale(grayscale(images[0]),grayscale(images[2]),\
                          window,numLevels)
plot_optical_flow(images[0],images[2],U,V, \
                  'levels = ' + str(numLevels) + ', window = '+str(window) + ',␣
 ↪image 1 and image 3 grayscale')




U,V=LucasKanadeMultiScaleRGB(images[0][:,:,:3],images[3][:,:,:3],\
                          window,numLevels)
plot_optical_flow(images[0],images[3],U,V, \
                  'levels = ' + str(numLevels) + ', window = '+str(window) + ',␣
 ↪image 1 and image 4', color=True)

U,V=LucasKanadeMultiScale(grayscale(images[0]),grayscale(images[3]),\
                          window,numLevels)
plot_optical_flow(images[0],images[3],U,V, \
                  'levels = ' + str(numLevels) + ', window = '+str(window) + ',␣
 ↪image 1 and image 4 grayscale')
```

1.4.3 Part 3 (1pt):

Comment on differences in outputs of the RGB optical flow with the grayscale version

1.5 Problem 3: Machine Learning [12 pts]

In this problem, you will implement several machine learning solutions for computer vision problems.

1.5.1 Part 1: Initial setup [1 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. Furthermore, installing with GPU support is often more difficult to configure so it is suggested that you install the CPU only version. TA's will not provide any support related to GPU or CUDA.

Run the torch import statements below to verify your instalation.

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 training data.

[ ]:
```python
import torch.nn as nn
import torch.nn.functional as F
import torch
from torch.autograd import Variable

x = torch.rand(5, 3)
print(x)
```

[ ]:
```python
import os
import struct

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

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')))
```

[ ]:
```
# Understand the shapes of the each variable carying data
print(trainData.shape, trainLabels.shape)
print(testData.shape, testLabels.shape)
```

[ ]:
```
# display one image from each label
""" ==========
YOUR CODE HERE
========== """
```

Some helper functions are given below.

[ ]:
```
# a generator for batches of data
# yields data (batchsize, 28, 28) and labels (batchsize)
# if shuffle, it will load batches in a random order
from tqdm import tqdm
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␣
↪tqdm(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' %
      test(testData, testLabels, randomClassifier))
```

1.5.2 Part 2: Confusion Matrix [2 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 was classified as class j. Can you justify the accuracy given by the random classifier?

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

# 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):
    M=np.zeros((10,10))
    acc=0.0
    """ ==========
    YOUR CODE HERE
    ========== """
    return M, acc

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

```
M_ran,acc_ran = Confusion(testData, testLabels, randomClassifier)
VisualizeConfusion(M_ran)
```

Your Comments on the accuracy & confusion matrix of random classifier:

1.5.3 Part 3: K-Nearest Neighbors (KNN) [4 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 confusion matrix and accuracy for your KNN classifier trained on the entire train set. (should be ~97 %)
- After evaluating the classifier on the testset, based on the confusion matrix, mention the number that the number '2' is most often predicted to be, other than '2'.

[ ]:
```
from sklearn.neighbors import KNeighborsClassifier
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 with all the training examples (This may take a while)
knnClassifer = KNNClassifer()
knnClassifer.train(trainData, trainLabels)
```

```
[ ]: # display confusion matrix for your KNN classifier with all the training␣
      ↪examples
     # (This may take a while)
     """ ==========
     YOUR CODE HERE
     ========== """
```

Comments here on which number the number 2 is most often misclassified as

1.5.4 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=3 and 25 principal components). You should implement PCA yourself using svd (you may not use sklearn.decomposition.PCA or any other package that directly implements PCA transformations

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=3):
             # 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
             ========== """

         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)
     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
========== """
```

Comments here on PCA vs PCA KNN testing time and the reasoning

1.6 Problem 4: Deep learning [14 pts]

Below is some helper code to train your deep networks.

1.6.1 Part 1: Training with PyTorch [2 pts]

Below is some helper code to train your deep networks. Complete the train function for DNN below. You should write down the training operations in this function. That means, for a batch of data you have to initialize the gradients, forward propagate the data, compute error, do back propagation and finally update the parameters. 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 deep neural networks. It implements the training loop␣
↪(train_net).
# You will need to implement the "__init__()" function to define the networks
# structures and "forward()", to propagate your data, in the following problems.

import torch.nn.init
import torch.optim as optim
from torch.autograd import Variable
from torch.nn.parameter import Parameter
from tqdm import tqdm
from scipy.stats import truncnorm

class DNN(nn.Module):
    def __init__(self):
        super(DNN, self).__init__()
        pass

    def forward(self, x):
        raise NotImplementedError

    def train_net(self, trainData, trainLabels, epochs=1, batchSize=50):
        criterion = nn.CrossEntropyLoss()
        optimizer = torch.optim.Adam(self.parameters(), lr = 3e-4)

        for epoch in range(epochs):
            self.train()  # set netowrk in training mode
```

```
            for i, (data,labels) in enumerate(DataBatch(trainData, trainLabels,␣
↪batchSize, shuffle=True)):
                data = Variable(torch.FloatTensor(data))
                labels = Variable(torch.LongTensor(labels))
                optimizer.zero_grad()
                outputs = self.forward(data)
                loss = criterion(outputs,labels)
                loss.backward()
                optimizer.step()


            self.eval()  # set network in evaluation mode
            print ('Epoch:%d Accuracy: %f'%(epoch+1, test(testData, testLabels,␣
↪self)))

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

# 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)
```

[ ]:
```
# example linear classifier - input connected to output
# you can take this as an example to learn how to extend DNN class
class LinearClassifier(DNN):
    def __init__(self, in_features=28*28, classes=10):
        super(LinearClassifier, self).__init__()
        # in_features=28*28
        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

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'))))
```

[ ]:
```
# test the example linear classifier (note you should get around 90% accuracy
# for 10 epochs and batchsize 50)
linearClassifier = LinearClassifier()
linearClassifier.train_net(trainData, trainLabels, epochs=10)

print ('Linear classifier accuracy: %f'%test(testData, testLabels,␣
↪linearClassifier))
```

[ ]:
```
# display confusion matrix
""" ==========
YOUR CODE HERE
========== """
```

1.6.2 Part 2: Single Layer Perceptron [2 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.

[ ]:
```
# Plot filter weights corresponding to each class, you may have to reshape them␣
↪to make sense out of them
# linearClassifier.weight1.data will give you the first layer weights
""" ==========
YOUR CODE HERE
========== """
```

Comments on weights

1.6.3 Part 3: Multi Layer Perceptron (MLP) [5 pts]

Here you will implement an MLP. The MLP should 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 ~ 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?

[ ]:
```
class MLPClassifer(DNN):
    def __init__(self, in_features=28*28, classes=10, hidden=100):
        super(MLPClassifer, self).__init__()
        """ ==========
        YOUR CODE HERE
        ========== """



    def forward(self, x):
        """ ==========
        YOUR CODE HERE
        ========== """



mlpClassifer = MLPClassifer()
mlpClassifer.train_net(trainData, trainLabels, epochs=10, batchSize=50)
```

[ ]:
```
# Plot confusion matrix
""" ==========
YOUR CODE HERE
========== """
```

[ ]:
```
# Plot filter weights
""" ==========
YOUR CODE HERE
========== """
```

Comments on weights:

1.6.4 Part 4: Convolutional Neural Network (CNN) [5 pts]

Here you will implement a CNN with the following architecture:

- n=5
- ReLU(Conv(kernel_size=5x5, stride=2, output_features=n))
- ReLU(Conv(kernel_size=5x5, stride=2, output_features=n*2))
- ReLU(Linear(hidden units = 64))
- Linear(output_features=classes)

So, 2 convolutional layers, followed by 1 fully connected hidden layer and then the output layer

Display the confusion matrix and accuracy after training. You should get around ~ 98 % accuracy for 10 epochs and batch size 50. **Note: You are not allowed to use torch.nn.Conv2d() and torch.nn.Linear(), Using these will lead to deduction of points. Use the declared conv2d(), weight_variable() and bias_variable() functions.** Although, in practice, when you move forward after this class you will use torch.nn.Conv2d() which makes life easier and hides all the operations underneath.

[ ]:
```python
def conv2d(x, W, stride, bias=None):
    # x: input
    # W: weights (out, in, kH, kW)
    return F.conv2d(x, W, bias, stride=stride, padding=2)

# Defining a Convolutional Neural Network
class CNNClassifer(DNN):
    def __init__(self, classes=10, n=5):
        super(CNNClassifer, self).__init__()
        """ ==========
        YOUR CODE HERE
        ========== """
    def forward(self, x):
        """ ==========
        YOUR CODE HERE
        ========== """
cnnClassifer = CNNClassifer()
cnnClassifer.train_net(trainData, trainLabels, epochs=10)
```

[ ]:
```python
# Plot confusion matrix
""" ==========
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 significant increase in accuracy, but in this case since the problem is not too hard, the increase in accuracy is not 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 neural nets/ pytorch at https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
- You can play with a demo of neural network created by Daniel Smilkov and Shan Carter at https://playground.tensorflow.org/