Lab 1. PyTorch and ANNs

This lab is a warm up to get you used to the PyTorch programming environment used in the course, and also to help you review and renew your knowledge of Python and relevant Python libraries. The lab must be done individually. Please recall that the University of Toronto plagarism rules apply.

By the end of this lab, you should be able to:

1. Be able to perform basic PyTorch tensor operations.
2. Be able to load data into PyTorch
3. Be able to configure an Artificial Neural Network (ANN) using PyTorch
4. Be able to train ANNs using PyTorch
5. Be able to evaluate different ANN configuations

You will need to use numpy and PyTorch documentations for this assignment:

- https://docs.scipy.org/doc/numpy/reference/
- https://pytorch.org/docs/stable/torch.html

You can also reference Python API documentations freely.

What to submit

Submit a PDF file containing all your code, outputs, and write-up from parts 1-5. You can produce a PDF of your Google Colab file by going to `File -> Print` and then save as PDF. The Colab instructions has more information.

Do not submit any other files produced by your code.

Include a link to your colab file in your submission.

Please use Google Colab to complete this assignment. If you want to use Jupyter Notebook, please complete the assignment and upload your Jupyter Notebook file to Google Colab for submission.

Adjust the scaling to ensure that the text is not cutoff at the margins.

Colab Link

Submit make sure to include a link to your colab file here

Colab Link: https://colab.research.google.com/drive/1Mf42A34Smw1QgW2Iu5BAcwHLzW8dZQDC

Part 1. Python Basics [3 pt]

The purpose of this section is to get you used to the basics of Python, including working with functions, numbers, lists, and strings.

Note that we **will** be checking your code for clarity and efficiency.

If you have trouble with this part of the assignment, please review http://cs231n.github.io/python-numpy-tutorial/

Part (a) -- 1pt

Write a function `sum_of_cubes` that computes the sum of cubes up to `n`. If the input to `sum_of_cubes` invalid (e.g. negative or non-integer `n`), the function should print out `"Invalid input"` and return `-1`.

```
def sum_of_cubes(n):
    """Return the sum (1^3 + 2^3 + 3^3 + ... + n^3)

    Precondition: n > 0, type(n) == int

    >>> sum_of_cubes(3)
    36
    >>> sum_of_cubes(1)
    1
    """

    s = 0
    if n<0 or type(n)!=int: # check for negative number of non-integer n
      print("Invalid input")
```

```
    return -1
  else:
    for i in range(1,n+1): # loop through to get sum of cubes
      s = s + i**3
    return s

print(sum_of_cubes(1.0))
```

```
Invalid input
-1
```

Part (b) -- 1pt

Write a function `word_lengths` that takes a sentence (string), computes the length of each word in that sentence, and returns the length of each word in a list. You can assume that words are always separated by a space character `" "`.

Hint: recall the `str.split` function in Python. If you arenot sure how this function works, try typing `help(str.split)` into a Python shell, or check out https://docs.python.org/3.6/library/stdtypes.html#str.split

```
help(str.split)
```

Show hidden output

```
def word_lengths(sentence):
    """Return a list containing the length of each word in
    sentence.

    >>> word_lengths("welcome to APS360!")
    [7, 2, 7]
    >>> word_lengths("machine learning is so cool")
    [7, 8, 2, 2, 4]
    """

    word = [] # create a new list to store lengths of each word
    s = sentence.strip(" ")
    l = s.split(" ") # split the string and store the words in a list
    for i in l:
      word.append(len(i)) # add length of each word to the new list 'word'
    return word

print(word_lengths("    machine learning is so cool"))
```

```
[7, 8, 2, 2, 4]
```

Part (c) -- 1pt

Write a function `all_same_length` that takes a sentence (string), and checks whether every word in the string is the same length. You should call the function `word_lengths` in the body of this new function.

```
def all_same_length(sentence):
    """Return True if every word in sentence has the same
    length, and False otherwise.

    >>> all_same_length("all same length")
    False
    >>> word_lengths("hello world")
    True
    """
    # Assuming there is no space before or after??
    s = sentence.strip(" ")
    l = word_lengths(s) # call 'word_lengths' function
    for i in range(1,len(l)): # loop from the second word to the last word
      if l[i] != l[0]: # if the length of the first word is not equal to any of the word lengths
        return False # return False
    return True

print(all_same_length("   world asdff awer1 wergt"))
```

```
True
```

Part 2. NumPy Exercises [5 pt]

In this part of the assignment, you'll be manipulating arrays usign NumPy. Normally, we use the shorter name `np` to represent the package `numpy`.

```
import numpy as np
```

Part (a) -- 1pt

The below variables `matrix` and `vector` are numpy arrays. Explain what you think `<NumpyArray>.size` and `<NumpyArray>.shape` represent.

```
matrix = np.array([[1., 2., 3., 0.5],
                   [4., 5., 0., 0.],
                   [-1., -2., 1., 1.]])
vector = np.array([2., 0., 1., -2.])
```

```
matrix.size
```

Show hidden output

```
matrix.shape
```

Show hidden output

```
vector.size
```

Show hidden output

```
vector.shape
```

Show hidden output

```
# <NumpyArray>.size: it returns the total number of elements in the numpy array

# <NumpyArray>.shape: it returns a tuple of the dimensions of the array
# for a matrix, it will return the number of rows followed by the number of columns: (n,m)
# for a single 1D numpy array, it acts like a column vector, and it will return the number of rows, and the colume part
# will be blank: (n,)
```

Part (b) -- 1pt

Perform matrix multiplication `output = matrix x vector` by using for loops to iterate through the columns and rows. Do not use any builtin NumPy functions. Cast your output into a NumPy array, if it isn't one already.

Hint: be mindful of the dimension of output

```
output = []
s = 0
for i in range(matrix.shape[0]): # loop through the rows
  for j in range(matrix.shape[1]): # loop through the columns
    s += matrix[i][j] * vector[j] # get the element through matrix multiplication
  output.append(s) # add the element to the result
  s = 0 # resume the process by setting the sum back to 0
```

```
output
```

```
[4.0, 8.0, -3.0]
```

Part (c) -- 1pt

Perform matrix multiplication `output2 = matrix x vector` by using the function `numpy.dot`.

We will never actually write code as in part(c), not only because `numpy.dot` is more concise and easier to read/write, but also performance-wise `numpy.dot` is much faster (it is written in C and highly optimized). In general, we will avoid for loops in our code.

```
output2 = np.dot(matrix, vector)
```

```
output2
```

```
array([ 4.,  8., -3.])
```

Part (d) -- 1pt

As a way to test for consistency, show that the two outputs match.

```
if np.array_equal(output, output2):
    print(True)
else:
    print(False)
```

```
True
```

Part (e) -- 1pt

Show that using `np.dot` is faster than using your code from part (c).

You may find the below code snippit helpful:

```
import time

# record the time before running code
start_time = time.time()

# place code to run here
output = []
s = 0
for i in range(matrix.shape[0]): # loop through the rows
  for j in range(matrix.shape[1]): # loop through the columns
    s += matrix[i][j] * vector[j] # get the element through matrix multiplication
  output.append(s) # add the element to the result
  s = 0 # resume the process by setting the sum back to 0

# record the time after the code is run
end_time = time.time()

# compute the difference
diff = end_time - start_time
diff # total time using for loops = 0.000275
```

```
0.00022292137145996094
```

```
import time

# record the time before running code
start_time = time.time()

# place code to run here
output2 = np.dot(matrix, vector)

# record the time after the code is run
end_time = time.time()

# compute the difference
diff = end_time - start_time
diff # total time using np.dot = 0.000184
```

```
0.0014181137084960938
```

Part 3. Images [6 pt]

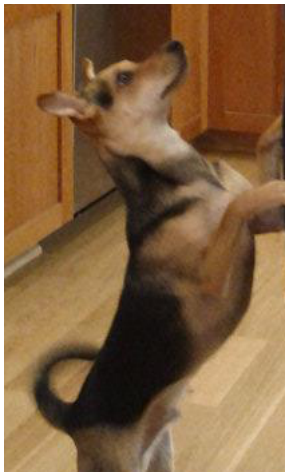
A picture or image can be represented as a NumPy array of “pixels”, with dimensions H × W × C, where H is the height of the image, W is the width of the image, and C is the number of colour channels. Typically we will use an image with channels that give the the Red, Green, and Blue “level” of each pixel, which is referred to with the short form RGB.

You will write Python code to load an image, and perform several array manipulations to the image and visualize their effects.

```
import matplotlib.pyplot as plt
```

Part (a) -- 1 pt

This is a photograph of a dog whose name is Mochi.



Load the image from its url (https://drive.google.com/uc?export=view&id=1oaLVR2hr1_qzpKQ47i9rVUIklwbDcews) into the variable `img` using the `plt.imread` function.

Hint: You can enter the URL directly into the `plt.imread` function as a Python string.

```
img = plt.imread("https://drive.google.com/uc?export=view&id=1oaLVR2hr1_qzpKQ47i9rVUIklwbDcews")
```

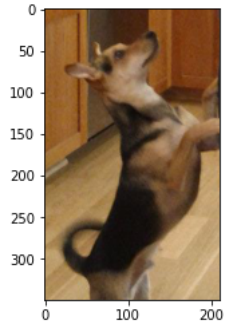
Part (b) -- 1pt

Use the function `plt.imshow` to visualize `img`.

This function will also show the coordinate system used to identify pixels. The origin is at the top left corner, and the first dimension indicates the Y (row) direction, and the second dimension indicates the X (column) dimension.

```
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x7f43a449a310>
```

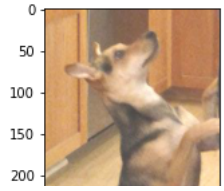


Part (c) -- 2pt

Modify the image by adding a constant value of 0.25 to each pixel in the `img` and store the result in the variable `img_add`. Note that, since the range for the pixels needs to be between [0, 1], you will also need to clip img_add to be in the range [0, 1] using `numpy.clip`. Clipping sets any value that is outside of the desired range to the closest endpoint. Display the image using `plt.imshow`.

```
img_add = img+0.25
new_image = np.clip(img_add,0,1) # min = 0, max = 1
plt.imshow(new_image)
```

```
<matplotlib.image.AxesImage at 0x7f43a43b8520>
```



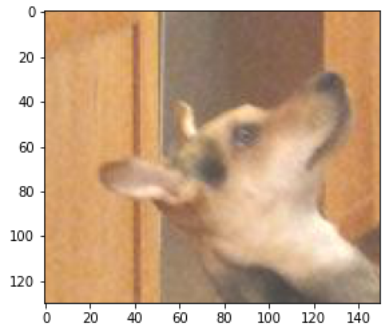
Part (d) -- 2pt

Crop the **original** image (`img` variable) to a 130 x 150 image including Mochi's face. Discard the alpha colour channel (i.e. resulting `img_cropped` should **only have RGB channels**)

Display the image.

```
img_cropped = new_image[0:130, 0:150] # crop the image
img_cropped = np.delete(img_cropped, 3,2) # delete alpha colour channel
plt.imshow(img_cropped)
```

```
<matplotlib.image.AxesImage at 0x7f43a4390ca0>
```



Part 4. Basics of PyTorch [6 pt]

PyTorch is a Python-based neural networks package. Along with tensorflow, PyTorch is currently one of the most popular machine learning libraries.

PyTorch, at its core, is similar to Numpy in a sense that they both try to make it easier to write codes for scientific computing achieve improved performance over vanilla Python by leveraging highly optimized C back-end. However, compare to Numpy, PyTorch offers much better GPU support and provides many high-level features for machine learning. Technically, Numpy can be used to perform almost every thing PyTorch does. However, Numpy would be a lot slower than PyTorch, especially with CUDA GPU, and it would take more effort to write machine learning related code compared to using PyTorch.

```
import torch
```

Part (a) -- 1 pt

Use the function `torch.from_numpy` to convert the numpy array `img_cropped` into a PyTorch tensor. Save the result in a variable called `img_torch`.

```
print(img_cropped)
print('\n')
img_torch = torch.from_numpy(img_cropped) # creating a PyTorch tensor called 'image_torch'
# A tensor is similar to an numpy array, just a generic n-dimensional array to be used for arbitrary numeric computation.
print(img_torch)
```

```
          [0.8069, 0.5912, 0.3676],
          ...,
          [0.8304, 0.5912, 0.3794],
          [0.8539, 0.6147, 0.4029],
          [0.8657, 0.6265, 0.4147]],

         [[0.7912, 0.5716, 0.3402],
          [0.8147, 0.5951, 0.3637],
          [0.8461, 0.6265, 0.3951],
          ...,
          [0.8382, 0.5990, 0.3873],
          [0.8578, 0.6186, 0.4069],
          [0.8696, 0.6304, 0.4186]],

         [[0.8657, 0.6265, 0.4029],
          [0.8696, 0.6343, 0.3990],
          [0.8696, 0.6343, 0.3912],
          ...,
          [0.8422, 0.6029, 0.3873],
          [0.8657, 0.6265, 0.4108],
          [0.8775, 0.6382, 0.4225]],

         ...,

         [[0.8539, 0.6382, 0.4186],
          [0.8578, 0.6422, 0.4186],
          [0.8618, 0.6461, 0.4225],
          ...,
          [0.6304, 0.5598, 0.4657],
          [0.6265, 0.5559, 0.4618],
          [0.6265, 0.5598, 0.4578]],

         [[0.8382, 0.6225, 0.4029],
          [0.8578, 0.6422, 0.4225],
          [0.8696, 0.6539, 0.4304],
          ...,
          [0.6382, 0.5676, 0.4814],
          [0.6304, 0.5598, 0.4657],
          [0.6304, 0.5598, 0.4657]],

         [[0.8304, 0.6147, 0.3951],
          [0.8539, 0.6382, 0.4186],
          [0.8735, 0.6578, 0.4382],
          ...,
          [0.6696, 0.5873, 0.5049],
          [0.6539, 0.5716, 0.4892],
          [0.6461, 0.5637, 0.4814]]])
```

Part (b) -- 1pt

Use the method `<Tensor>.shape` to find the shape (dimension and size) of `img_torch`.

```
img_torch.shape
```

```
torch.Size([130, 150, 3])
```

Part (c) -- 1pt

How many floating-point numbers are stored in the tensor `img_torch`?

```
print(str(130*150*3) + " numbers are stored.")
```

```
58500 numbers are stored.
```

Part (d) -- 1 pt

What does the code `img_torch.transpose(0,2)` do? What does the expression return? Is the original variable `img_torch` updated? Explain.

```
img_torch.transpose(0,2)

# it swaps the values in first dimension with values in the third dimension
```

```
tensor([[[0.8382, 0.6225, 0.3990],
         [0.8265, 0.6108, 0.3873],
         [0.8069, 0.5912, 0.3676],
         ...,
         [0.8304, 0.5912, 0.3794],
```

```
          [0.8539, 0.6147, 0.4029],
          [0.8657, 0.6265, 0.4147]],

         [[0.7912, 0.5716, 0.3402],
          [0.8147, 0.5951, 0.3637],
          [0.8461, 0.6265, 0.3951],
          ...,
          [0.8382, 0.5990, 0.3873],
          [0.8578, 0.6186, 0.4069],
          [0.8696, 0.6304, 0.4186]],

         [[0.8657, 0.6265, 0.4029],
          [0.8696, 0.6343, 0.3990],
          [0.8696, 0.6343, 0.3912],
          ...,
          [0.8422, 0.6029, 0.3873],
          [0.8657, 0.6265, 0.4108],
          [0.8775, 0.6382, 0.4225]],

         ...,

         [[0.8539, 0.6382, 0.4186],
          [0.8578, 0.6422, 0.4186],
          [0.8618, 0.6461, 0.4225],
          ...,
          [0.6304, 0.5598, 0.4657],
          [0.6265, 0.5559, 0.4618],
          [0.6265, 0.5598, 0.4578]],

         [[0.8382, 0.6225, 0.4029],
          [0.8578, 0.6422, 0.4225],
          [0.8696, 0.6539, 0.4304],
          ...,
          [0.6382, 0.5676, 0.4814],
          [0.6304, 0.5598, 0.4657],
          [0.6304, 0.5598, 0.4657]],

         [[0.8304, 0.6147, 0.3951],
          [0.8539, 0.6382, 0.4186],
          [0.8735, 0.6578, 0.4382],
          ...,
          [0.6696, 0.5873, 0.5049],
          [0.6539, 0.5716, 0.4892],
          [0.6461, 0.5637, 0.4814]]])
tensor([[[0.8382, 0.7912, 0.8657,  ..., 0.8539, 0.8382, 0.8304],
         [0.8265, 0.8147, 0.8696,  ..., 0.8578, 0.8578, 0.8539],
         [0.8069, 0.8461, 0.8696,  ..., 0.8618, 0.8696, 0.8735],
         ...,
         [0.8304, 0.8382, 0.8422,  ..., 0.6304, 0.6382, 0.6696],
         [0.8539, 0.8578, 0.8657,  ..., 0.6265, 0.6304, 0.6539],
         [0.8657, 0.8696, 0.8775,  ..., 0.6265, 0.6304, 0.6461]],

         [[0.6225, 0.5716, 0.6265,  ..., 0.6382, 0.6225, 0.6147],
```

Part (e) -- 1 pt

What does the code `img_torch.unsqueeze(0)` do? What does the expression return? Is the original variable `img_torch` updated? Explain.

```
# It returns a new tensor with a dimension of size one inserted into the original tensor at the very first location.
# The original variable is not updated.
```

Part (f) -- 1 pt

Find the maximum value of `img_torch` along each colour channel? Your output should be a one-dimensional PyTorch tensor with exactly three values.

Hint: lookup the function `torch.max`.

```
max_vals = torch.max(torch.max(img_torch, 1)[0], 0)[0]
max_vals
```

```
tensor([1.0000, 1.0000, 0.9245])
```

Part 5. Training an ANN [10 pt]

The sample code provided below is a 2-layer ANN trained on the MNIST dataset to identify digits less than 3 or greater than and equal to 3. Modify the code by changing any of the following and observe how the accuracy and error are affected:

- number of training iterations
- number of hidden units
- numbers of layers
- types of activation functions
- learning rate

Please select at least three different options from the list above. For each option, please select two to three different parameters and provide a table.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms
import matplotlib.pyplot as plt # for plotting
import torch.optim as optim

torch.manual_seed(1) # set the random seed to make sure the variation of the result is because of the changes doing in the model,

# define a 2-layer artificial neural network
class Pigeon(nn.Module):
    def __init__(self):
        super(Pigeon, self).__init__()
        self.layer1 = nn.Linear(28 * 28, 30) # first liniear hidden layer with 30 neurons each 28*28
        self.layer2 = nn.Linear(30, 1) # match 30 neurons with the output of 1 neuron because it is binary classification
    def forward(self, img): # receives input and makes prediction (forward pass)
        flattened = img.view(-1, 28 * 28) # flatten the images to a vector
        activation1 = self.layer1(flattened) # pass the vector to hidden layer one
        activation1 = F.relu(activation1) # neurons recieve ReLU activation function
        activation2 = self.layer2(activation1) # don't need to pass it to sigmond function because we are using BCE loss function
        return activation2

pigeon = Pigeon()

# load the data
mnist_data = datasets.MNIST('data', train=True, download=True)
mnist_data = list(mnist_data)
mnist_train = mnist_data[:1000] #the first 1000 samples are for training
mnist_val   = mnist_data[1000:2000] #next 1000 samples are for validation
img_to_tensor = transforms.ToTensor()


# simplified training code to train `pigeon` on the "small digit recognition" task
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.SGD(pigeon.parameters(), lr=0.005, momentum=0.9)

for (image, label) in mnist_train:
    # actual ground truth: is the digit less than 3?
    actual = torch.tensor(label < 3).reshape([1,1]).type(torch.FloatTensor)
    # pigeon prediction
    out = pigeon(img_to_tensor(image)) # step 1-2: convert the image to tensor and pass it to our NN
    # update the parameters based on the loss
    loss = criterion(out, actual)      # step 3 (compute loss)
    loss.backward()                    # step 4 (compute the gradients for each weight and bias)
    optimizer.step()                   # step 4 (make the updates for each weight and bias)
    optimizer.zero_grad()              # a clean up gradient

# computing the error and accuracy on the training set
error = 0
for (image, label) in mnist_train:
    prob = torch.sigmoid(pigeon(img_to_tensor(image)))
    if (prob < 0.5 and label < 3) or (prob >= 0.5 and label >= 3): # prob<0.5 means that the label belongs to class 0
        error += 1
print("Training Error Rate:", error/len(mnist_train))
print("Training Accuracy:", 1 - error/len(mnist_train))


# computing the error and accuracy on a test set
error = 0
for (image, label) in mnist_val:
    prob = torch.sigmoid(pigeon(img_to_tensor(image)))
    if (prob < 0.5 and label < 3) or (prob >= 0.5 and label >= 3):
        error += 1
```

```
print("Test Error Rate:", error/len(mnist_val))
print("Test Accuracy:", 1 - error/len(mnist_val))
```

```
Training Error Rate: 0.036
Training Accuracy: 0.964
Test Error Rate: 0.079
Test Accuracy: 0.921
```

```
# Without changing:

# Training Error Rate: 0.036
# Training Accuracy: 0.964
# Test Error Rate: 0.079
# Test Accuracy: 0.921
```

```
!pip install tabulate
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: tabulate in /usr/local/lib/python3.8/dist-packages (0.8.10)
```

```
from tabulate import tabulate
```

```
# Changing learning rate, number of iterations and number of hidden layers

#create data
data = [["0.005", "20", "3", 0.944],
        ["0.003", "10", "3", 0.946],
        ["0.003", "5", "2", 0.937],
        ["0.002", "3", "2", 0.935]]

#define header names
col_names = ["learning rate", "number of iterations", "number of hidden layers", "Test Accuracy"]

#display table
print(tabulate(data, headers=col_names))
```

```
  learning rate    number of iterations    number of hidden layers    Test Accuracy
---------------  ----------------------  -------------------------  ---------------
          0.005                      20                          3            0.944
          0.003                      10                          3            0.946
          0.003                       5                          2            0.937
          0.002                       3                          2            0.935
```

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms
import matplotlib.pyplot as plt # for plotting
import torch.optim as optim

torch.manual_seed(1) # set the random seed to make sure the variation of the result is because of the changes doing in the model,

# define a 2-layer artificial neural network
class Pigeon(nn.Module):
    def __init__(self):
        super(Pigeon, self).__init__()
        self.layer1 = nn.Linear(28 * 28, 120) # first liniear hidden layer with 30 neurons each 28*28
        self.layer2 = nn.Linear(120,50)
        self.layer3 = nn.Linear(50,20)
        self.layer4 = nn.Linear(20, 1) # match 30 neurons with the output of 1 neuron because it is binary classification
    def forward(self, img): # receives input and makes prediction (forward pass)
        flattened = img.view(-1, 28 * 28) # flatten the images to a vector
        activation1 = self.layer1(flattened) # pass the vector to hidden layer one
        activation1 = F.relu(activation1) # neurons recieve ReLU activation function
        activation2 = self.layer2(activation1) # pass the vector to hidden layer one
        activation2 = F.relu(activation2) # neurons recieve ReLU activation function
        activation3 = self.layer3(activation2) # pass the vector to hidden layer one
        activation4 = F.relu(activation3)
        activation4 = self.layer4(activation4) # don't need to pass it to sigmond function because we are using BCE loss function
        return activation4

pigeon = Pigeon()

# load the data
```

```
mnist_data = datasets.MNIST('data', train=True, download=True)
mnist_data = list(mnist_data)
mnist_train = mnist_data[:1000] #the first 1000 samples are for training
mnist_val   = mnist_data[1000:2000] #next 1000 samples are for validation
img_to_tensor = transforms.ToTensor()


# simplified training code to train `pigeon` on the "small digit recognition" task
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.SGD(pigeon.parameters(), lr=0.005, momentum=0.9)

for epoch in range(10):
  for (image, label) in mnist_train:
      # actual ground truth: is the digit less than 3?
      actual = torch.tensor(label < 3).reshape([1,1]).type(torch.FloatTensor)
      # pigeon prediction
      out = pigeon(img_to_tensor(image)) # step 1-2: convert the image to tensor and pass it to our NN
      # update the parameters based on the loss
      loss = criterion(out, actual)      # step 3 (compute loss)
      loss.backward()                    # step 4 (compute the gradients for each weight and bias)
      optimizer.step()                   # step 4 (make the updates for each weight and bias)
      optimizer.zero_grad()              # a clean up gradient

# computing the error and accuracy on the training set
error = 0
for (image, label) in mnist_train:
    prob = torch.sigmoid(pigeon(img_to_tensor(image)))
    if (prob < 0.5 and label < 3) or (prob >= 0.5 and label >= 3): # prob<0.5 means that the label belongs to class 0
        error += 1
print("Training Error Rate:", error/len(mnist_train))
print("Training Accuracy:", 1 - error/len(mnist_train))


# computing the error and accuracy on a test set
error = 0
for (image, label) in mnist_val:
    prob = torch.sigmoid(pigeon(img_to_tensor(image)))
    if (prob < 0.5 and label < 3) or (prob >= 0.5 and label >= 3):
        error += 1
print("Test Error Rate:", error/len(mnist_val))
print("Test Accuracy:", 1 - error/len(mnist_val))
```

```
Training Error Rate: 0.014
Training Accuracy: 0.986
Test Error Rate: 0.07
Test Accuracy: 0.9299999999999999
```

Part (a) -- 3 pt

Comment on which of the above changes resulted in the best accuracy on training data? What accuracy were you able to achieve?

```
# When increasing number of layers/hidden units of the network and increasing the number of training iterations to
# a relatviey large number like 20, as well as decreaing the learning rate a little bit,
# resulted in a 100% accuracy on training data.
```

Part (b) -- 3 pt

Comment on which of the above changes resulted in the best accuracy on testing data? What accuracy were you able to achieve?

```
# Similarly, increasing number of layers/hidden units of the network and increasing the numbero of training iterations
# but only to 10 will result in a higher accuracy on the testing data with 94.6% accuracy.
```

Part (c) -- 4 pt

Which model hyperparameters should you use, the ones from (a) or (b)?

```
# I would use the hyperparameters from part (b) because performance on the test set better reflects
# real world performance, while performance on the training set may just be a result of overfitting. If the
# number of interations is too much, it would just result in the algorithm memorizing the training data,
# not actually learning and improving the prediction. Therefore, it is better to have a realtively lower
# iterations, which will result in a higher testing accuracy.
```

✓ 0s completed at 9:33 PM