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:

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
    """
    res = 0

    if(n>0 and type(n)== int):
        for i in range(n+1):
```

```
            res += i**3

        return res
    else:
        print("Invalid Input")
        return -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)
```

```
Help on method_descriptor:

split(self, /, sep=None, maxsplit=-1)
    Return a list of the words in the string, using sep as the delimiter string.

    sep
      The delimiter according which to split the string.
      None (the default value) means split according to any whitespace,
      and discard empty strings from the result.
    maxsplit
      Maximum number of splits to do.
      -1 (the default value) means no limit.
```

```
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]
    """

    splitSent = sentence.split(" ")
    print(splitSent)

    resLst = []
    for i in splitSent:
        resLst.append(len(i))

    return resLst
```

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
    """

    check = word_lengths(sentence)

    for i in check:
        for j in check:
            if i != j:
                return "They are not all the same Length"

    return "They are the all the same length"
```

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.])

# 2a.) <NumpyArray>.size represents the number of elements in the matrix
#      and <NumpyArray>.shape represent the size of a matrix (n x n)
```

```
matrix.size
```

```
12
```

```
matrix.shape
```

```
(3, 4)
```

```
vector.size
```

```
4
```

```
vector.shape
```

```
(4,)
```

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

```
def matrixMultiplier(matrix, vector):
  matMult = []
  for i in range(matrix.shape[0]):
      temp = 0
      for j in range(matrix.shape[1]):
        temp += matrix[i, j]*vector[j]
      matMult.append(temp)

  return matMult
```

```
output = None
```

```
output = matrixMultiplier(matrix, vector)
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.

```
print(output==output2)
```

```
[ True  True  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
for i in range(10000):
    99*99
    matrixMultiplier(matrix, vector)

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

# compute the difference -- my function version
diff = end_time - start_time
diff
```

```
0.08761787414550781
```

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

# place code to run here
for i in range(10000):
    99*99
    np.dot(matrix, vector)

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

# compute the difference -- numpy.dot versin
diff = end_time - start_time
diff
```

```
0.021400928497314453
```

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.

![alt text]

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')
```

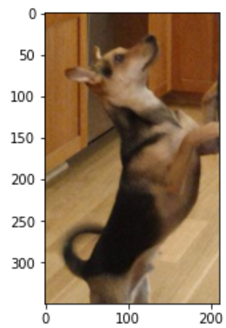
Part (b) -- 1pt

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 0x7f3e9dc0e970>
```

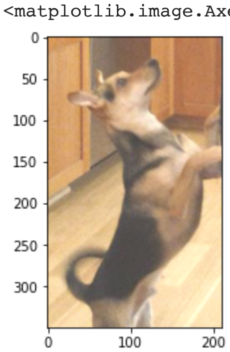


Part (c) -- 2pt

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
np.clip(img_add, 0, 1)
plt.imshow(img_add)
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow
<matplotlib.image.AxesImage at 0x7f3e9db94070>
```



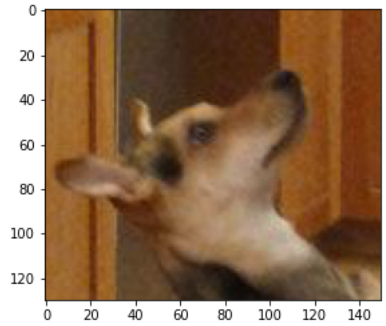
Part (d) -- 2pt

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=img[0:130, 20:170, 0:3]
plt.imshow(img_cropped)
```

```
<matplotlib.image.AxesImage at 0x7f3e9caedac0>
```



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`.

```
img_torch = torch.from_numpy(img_cropped)
```

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)+' floating-point numbers are stored in the tensor')
```

```
58500 floating-point numbers are stored in the tensor
```

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)

# this transposes the img_torch tensor by swapping the third dimension with the first dimentsion. The variable img_torch is update
# transposed tensor but its values are not changed, its dimensions are just arranged differently.
```

```
tensor([[[0.5961, 0.6078, 0.6235,  ..., 0.6078, 0.6039, 0.6157],
         [0.5804, 0.5882, 0.6000,  ..., 0.6196, 0.6196, 0.6275],
```

```
         [0.5765, 0.5725, 0.5843,  ..., 0.6314, 0.6353, 0.6392],
         ...,
         [0.4941, 0.4824, 0.4824,  ..., 0.7176, 0.7333, 0.7176],
         [0.4902, 0.4941, 0.5098,  ..., 0.7137, 0.7059, 0.7059],
         [0.5098, 0.5098, 0.5137,  ..., 0.7373, 0.7373, 0.7137]],

        [[0.3569, 0.3647, 0.3686,  ..., 0.3922, 0.3882, 0.4039],
         [0.3412, 0.3451, 0.3451,  ..., 0.4039, 0.4078, 0.4157],
         [0.3373, 0.3294, 0.3294,  ..., 0.4157, 0.4235, 0.4275],
         ...,
         [0.2549, 0.2431, 0.2431,  ..., 0.6000, 0.6157, 0.6078],
         [0.2471, 0.2510, 0.2667,  ..., 0.6157, 0.6039, 0.6118],
         [0.2667, 0.2667, 0.2706,  ..., 0.6392, 0.6353, 0.6196]],

        [[0.1333, 0.1412, 0.1490,  ..., 0.1725, 0.1647, 0.1686],
         [0.1176, 0.1216, 0.1255,  ..., 0.1843, 0.1725, 0.1725],
         [0.1137, 0.1059, 0.1020,  ..., 0.1922, 0.1882, 0.1843],
         ...,
         [0.0824, 0.0706, 0.0706,  ..., 0.4588, 0.4667, 0.4510],
         [0.0549, 0.0588, 0.0745,  ..., 0.4588, 0.4588, 0.4706],
         [0.0745, 0.0745, 0.0784,  ..., 0.4824, 0.4902, 0.4784]]])
```

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.

- This adds a new dimension to the tensor at position 0. The original variable remains unchanged

```
img_torch.unsqueeze(0)
# This adds a new dimension to the tensor at position 0. The original variable remains unchanged
```

```
tensor([[[[0.5961, 0.3569, 0.1333],
          [0.5804, 0.3412, 0.1176],
          [0.5765, 0.3373, 0.1137],
          ...,
          [0.4941, 0.2549, 0.0824],
          [0.4902, 0.2471, 0.0549],
          [0.5098, 0.2667, 0.0745]],

         [[0.6078, 0.3647, 0.1412],
          [0.5882, 0.3451, 0.1216],
          [0.5725, 0.3294, 0.1059],
          ...,
          [0.4824, 0.2431, 0.0706],
          [0.4941, 0.2510, 0.0588],
          [0.5098, 0.2667, 0.0745]],

         [[0.6235, 0.3686, 0.1490],
          [0.6000, 0.3451, 0.1255],
          [0.5843, 0.3294, 0.1020],
          ...,
          [0.4824, 0.2431, 0.0706],
          [0.5098, 0.2667, 0.0745],
          [0.5137, 0.2706, 0.0784]],

         ...,

         [[0.6078, 0.3922, 0.1725],
          [0.6196, 0.4039, 0.1843],
          [0.6314, 0.4157, 0.1922],
          ...,
          [0.7176, 0.6000, 0.4588],
          [0.7137, 0.6157, 0.4588],
          [0.7373, 0.6392, 0.4824]],

         [[0.6039, 0.3882, 0.1647],
          [0.6196, 0.4078, 0.1725],
          [0.6353, 0.4235, 0.1882],
          ...,
          [0.7333, 0.6157, 0.4667],
          [0.7059, 0.6039, 0.4588],
          [0.7373, 0.6353, 0.4902]],

         [[0.6157, 0.4039, 0.1686],
          [0.6275, 0.4157, 0.1725],
          [0.6392, 0.4275, 0.1843],
          ...,
          [0.7176, 0.6078, 0.4510],
          [0.7059, 0.6118, 0.4706],
          [0.7137, 0.6196, 0.4784]]]])
```

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]
print(max_vals)
```

```
tensor([0.8941, 0.7882, 0.6745])
```

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

# 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)
        self.layer2 = nn.Linear(30, 1)
    def forward(self, img):
        flattened = img.view(-1, 28 * 28)
        activation1 = self.layer1(flattened)
        activation1 = F.relu(activation1)
        activation2 = self.layer2(activation1)
        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]
mnist_val   = mnist_data[1000:2000]
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
    # update the parameters based on the loss
    loss = criterion(out, actual)      # step 3
    loss.backward()                    # step 4 (compute the updates for each parameter)
    optimizer.step()                   # step 4 (make the updates for each parameter)
```

```
        optimizer.zero_grad()              # a clean up step for PyTorch

    # 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):
            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))
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to
100%                                    9912422/9912422 [00:00<00:00,
                                        103231625.32it/s]
Extracting data/MNIST/raw/train-images-idx3-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to
100%                                    28881/28881 [00:00<00:00, 974762.57it/s]
Extracting data/MNIST/raw/train-labels-idx1-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to d
100%                                    1648877/1648877 [00:00<00:00,
                                        46227675.52it/s]
Extracting data/MNIST/raw/t10k-images-idx3-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to d
100%                                    4542/4542 [00:00<00:00, 98717.63it/s]
Extracting data/MNIST/raw/t10k-labels-idx1-ubyte.gz to data/MNIST/raw

Training Error Rate: 0.036
```

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?

- The changes made here were adding extra hidden layers, changing the number of iterations and changing the learning rate. I found that the more layers I added, the worse the accuracy got. For instance, I added 1 extra layer and the Test accuracy dropped from 92% to 89%. I then changed the number of iterations to 10 and the learning rate from 0.005 to 0.002. This resulted in a slightly beter performing Test Accuracy. Then two extra layers and decreasing the number of iterations to 5, and changing the learning rate 0.003 as seen in the code below increases the from Test Accuracy from 92% in the previous model to 94%.

```
#**************************************************
#   The change that was made was adding two extra layers to the ANN

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

# define a 2-layer artificial neural network
class Pigeon(nn.Module):
    def __init__(self):
        super(Pigeon, self).__init__()
```

```
        self.layer1 = nn.Linear(28 * 28, 80)
        self.layer2 = nn.Linear(80, 50)
        self.layer3 = nn.Linear(50, 30)
        self.layer4 = nn.Linear(30, 1)
    def forward(self, img):
        flattened = img.view(-1, 28 * 28)
        activation1 = self.layer1(flattened)
        activation2 = F.relu(self.layer2(activation1))
        activation3 = F.relu(self.layer3(activation2))
        activation4 = self.layer4(activation3)
        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]
mnist_val   = mnist_data[1000:2000]
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=0.003, momentum=0.9)

for i in range(5):
  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
      # update the parameters based on the loss
      loss = criterion(out, actual)      # step 3
      loss.backward()                    # step 4 (compute the updates for each parameter)
      optimizer.step()                   # step 4 (make the updates for each parameter)
      optimizer.zero_grad()              # a clean up step for PyTorch

# 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):
        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: 0.019
Training Accuracy: 0.981
Test Error Rate: 0.063
Test Accuracy: 0.937
```

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?

- The next change that was made was halfing the learning rate from 0.005 to 0.01 and increasing the number of iterations to 5. This increased the Test accuracy from 92% to 94% and decreased the Test error rate to 5.8%. The learning rate of 0.01 is the best fit as going lower than that or higher than that results in a Test Accuracy drop.

```
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

# 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)
        self.layer2 = nn.Linear(30, 1)
    def forward(self, img):
        flattened = img.view(-1, 28 * 28)
        activation1 = self.layer1(flattened)
        activation1 = F.silu(activation1)
        activation2 = self.layer2(activation1)
        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]
mnist_val   = mnist_data[1000:2000]
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=0.01, momentum=0.9)

for i in range(5):
  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
      # update the parameters based on the loss
      loss = criterion(out, actual)      # step 3
      loss.backward()                    # step 4 (compute the updates for each parameter)
      optimizer.step()                   # step 4 (make the updates for each parameter)
      optimizer.zero_grad()              # a clean up step for PyTorch

# 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):
        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: 0.012
Training Accuracy: 0.988
Test Error Rate: 0.058
Test Accuracy: 0.942
```

Part (c) -- 4 pt

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

```
* I would go with the the model b hyperperameters because they g
error rate and the highest Test Accuracy which reflects the more
applications
```

- I would go with the the model b hyperperameters because they give the lowest error rate and the highest Test Accuracy which reflects the more realistic applications

Google Colab Link: https://colab.research.google.com/drive/1BuBJdT0wS6gy4dtNbrJobhD_icsXLX28#scrollTo=9PQ-pRIlbzH8&uniqifier=1

✓ 8s completed at 1:39 AM