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
    """
    # Check if n is negative or n isn't an integer
    if n < 0 or type(n) is not int:
      # Print invalid input and output -1
      print("Invalid input")
      return -1

    # Initialize sum variable
    sum = 0

    # Iterate through all values of n
    while n:
      # Find n^3 and add it to the sum
      sum += n ** 3
      # Decrease n by 1 each time
      n -= 1

    print(sum)
    return sum
```

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]
    """
    # Initialize a list that will store the lengths of the words
    lengths = []
    # Split the sentence
    sentence = sentence.split()

    # Go through the split sentence
    for x in sentence:
      # Add the length of each word to the list
      lengths.append(len(x))

    return lengths
```

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
    """
    # Call the word_lengths function
    lengths = word_lengths(sentence)

    # Go through the lengths list
    for x in lengths:
      # Check if the current length is loop is not equal to the first length
      if x != lengths[0]:
        # Return false in this case because they aren't equal
        return False

    # Return true if it successfully finishes the loop
    return 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
# <NumpyArray>.size represents the multiplication of the number of columns by the number of r
# it can also be interpreted as the total number of entries in the numpy array
```

```
12
```

```
matrix.shape
# <NumpyArray>.shape represents the dimensions in the array (num of rows, num of columns)
```

```
(3, 4)
```

```
vector.size
# <NumpyArray>.size represents the multiplication of the number of columns by the number of r
# it can also be interpreted as the total number of entries in the numpy array
```

```
4
```

```
vector.shape
# <NumpyArray>.shape represents the dimensions in the array (num of rows, num of columns)
```

```
(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

```
output = None
```

```
# Initialize output as a list to store the result
output = []
# Use nested for loop to get values in matrix
for x in range(matrix.shape[0]):
  sum = 0
  for y in range(matrix.shape[1]):
    # Perform dot product
    sum += matrix[x, y] * vector[y]
  output.append(sum)
# Cast output to a numpy array
output = np.array(output)
output
```

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

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 = None


# Use np.dot
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.

```
# Use if statement and print "They match!" if outputs match
if output.all() == output2.all():
  print("They match!")
else:
  print("They don't match :(")
```

```
They match!
```

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:

```
# ----- Time for np.dot implementation -----

import time

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

output2 = np.dot(matrix, vector)

# record the time after the code is run
```

```
end_time = time.time()

# compute the difference
time_npdot = end_time - start_time
time_npdot
```

```
8.749961853027344e-05
```

```
# ----- Time for manual implementation -----

import time

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

# Initialize output as a list to store the result
output = []
# Use nested for loop to get values in matrix
for x in range(matrix.shape[0]):
  sum = 0
  for y in range(matrix.shape[1]):
    # Perform dot product
    sum += matrix[x, y] * vector[y]
  output.append(sum)
# Cast output to a numpy array
output = np.array(output)

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

# compute the difference
time_manual = end_time - start_time
time_manual
```

```
0.0002567768096923828
```

```
# ---- Comparison ----
difference = time_npdot - time_manual
print(difference)

# As we see, since the difference is negative, that means that time_manual > time_npdot
# which proves that numpy.dot() is faster
```

```
-0.00016927719116210938
```

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 = None
# Use plt.imread
img = plt.imread("https://drive.google.com/uc?export=view&id=1oaLVR2hr1_qzpKQ47i9rVUIklwbDcew
img
```

```
array([[[0.5882353 , 0.37254903, 0.14901961, 1.        ],
        [0.5764706 , 0.36078432, 0.13725491, 1.        ],
        [0.5568628 , 0.34117648, 0.11764706, 1.        ],
        ...,
        [0.40784314, 0.22352941, 0.16078432, 1.        ],
```

```
         [0.37254903, 0.22352941, 0.17254902, 1.        ],
         [0.30980393, 0.20392157, 0.16078432, 1.        ]],

        [[0.5411765 , 0.32156864, 0.09019608, 1.        ],
         [0.5647059 , 0.34509805, 0.11372549, 1.        ],
         [0.59607846, 0.3764706 , 0.14509805, 1.        ],
         ...,
         [0.4117647 , 0.22352941, 0.16862746, 1.        ],
         [0.3882353 , 0.23921569, 0.19607843, 1.        ],
         [0.31764707, 0.21176471, 0.1764706 , 1.        ]],

        [[0.6156863 , 0.3764706 , 0.15294118, 1.        ],
         [0.61960787, 0.38431373, 0.14901961, 1.        ],
         [0.61960787, 0.38431373, 0.14117648, 1.        ],
         ...,
         [0.4117647 , 0.22352941, 0.1764706 , 1.        ],
         [0.39607844, 0.24705882, 0.21176471, 1.        ],
         [0.32156864, 0.21568628, 0.1882353 , 1.        ]],

        ...,

        [[0.70980394, 0.5764706 , 0.3882353 , 1.        ],
         [0.7058824 , 0.57254905, 0.38431373, 1.        ],
         [0.69803923, 0.5686275 , 0.36862746, 1.        ],
         ...,
         [0.7411765 , 0.64705884, 0.4745098 , 1.        ],
         [0.74509805, 0.64705884, 0.4862745 , 1.        ],
         [0.77254903, 0.6745098 , 0.5137255 , 1.        ]],

        [[0.72156864, 0.5882353 , 0.4       , 1.        ],
         [0.7176471 , 0.58431375, 0.39607844, 1.        ],
         [0.7176471 , 0.58431375, 0.39607844, 1.        ],
         ...,
         [0.7411765 , 0.6392157 , 0.4392157 , 1.        ],
         [0.75686276, 0.654902  , 0.4627451 , 1.        ],
         [0.7764706 , 0.6745098 , 0.48235294, 1.        ]],

        [[0.7137255 , 0.5803922 , 0.39215687, 1.        ],
         [0.7137255 , 0.5803922 , 0.39215687, 1.        ],
         [0.7176471 , 0.58431375, 0.39607844, 1.        ],
         ...,
         [0.75686276, 0.654902  , 0.45490196, 1.        ],
         [0.76862746, 0.6666667 , 0.4745098 , 1.        ],
         [0.77254903, 0.67058825, 0.47843137, 1.        ]]], dtype=float32)
```

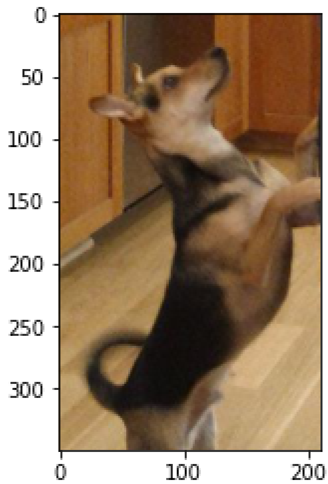
Part (b) -- 1pt

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.

```
img_show = plt.imshow(img)
img_show
```

```
<matplotlib.image.AxesImage at 0x7fb3492587d0>
```

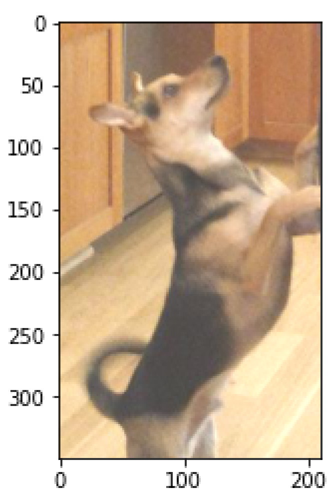


Part (c) -- 2pt

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

```
<matplotlib.image.AxesImage at 0x7fb343f34b10>
```

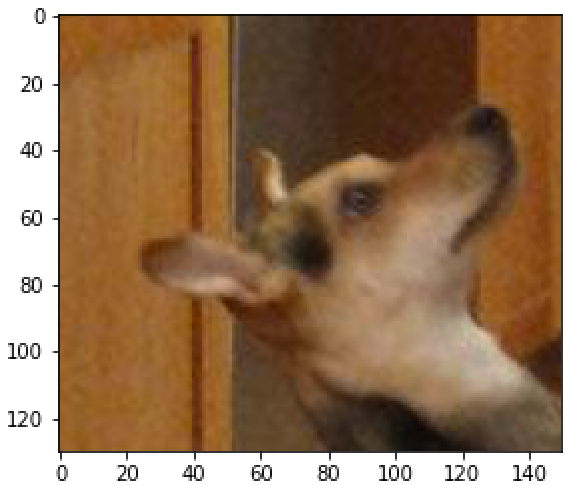


Part (d) -- 2pt

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**)

```
img_cropped = img[:130, :150, :3]
plt.imshow(img_cropped)
```

```
<matplotlib.image.AxesImage at 0x7fb3436a0a50>
```



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)
img_torch
```

```
tensor([[[0.5882, 0.3725, 0.1490],
         [0.5765, 0.3608, 0.1373],
         [0.5569, 0.3412, 0.1176],
         ...,
         [0.5804, 0.3412, 0.1294],
         [0.6039, 0.3647, 0.1529],
         [0.6157, 0.3765, 0.1647]],

        [[0.5412, 0.3216, 0.0902],
         [0.5647, 0.3451, 0.1137],
         [0.5961, 0.3765, 0.1451],
         ...,
         [0.5882, 0.3490, 0.1373],
         [0.6078, 0.3686, 0.1569],
         [0.6196, 0.3804, 0.1686]],

        [[0.6157, 0.3765, 0.1529],
         [0.6196, 0.3843, 0.1490],
         [0.6196, 0.3843, 0.1412],
         ...,
         [0.5922, 0.3529, 0.1373],
         [0.6157, 0.3765, 0.1608],
         [0.6275, 0.3882, 0.1725]],

        ...,

        [[0.6039, 0.3882, 0.1686],
         [0.6078, 0.3922, 0.1686],
         [0.6118, 0.3961, 0.1725],
         ...,
         [0.3804, 0.3098, 0.2157],
         [0.3765, 0.3059, 0.2118],
         [0.3765, 0.3098, 0.2078]],

        [[0.5882, 0.3725, 0.1529],
         [0.6078, 0.3922, 0.1725],
         [0.6196, 0.4039, 0.1804],
         ...,
         [0.3882, 0.3176, 0.2314],
         [0.3804, 0.3098, 0.2157],
         [0.3804, 0.3098, 0.2157]],

        [[0.5804, 0.3647, 0.1451],
         [0.6039, 0.3882, 0.1686],
         [0.6235, 0.4078, 0.1882],
         ...,
         [0.4196, 0.3373, 0.2549],
         [0.4039, 0.3216, 0.2392],
         [0.3961, 0.3137, 0.2314]]])
```

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`?

```
# Find the number of floating-point numbers stored in the tensor by multiplying the dimension
floating_point_nums = 1
for i in list(img_torch.shape):
  floating_point_nums = floating_point_nums * i

floating_point_nums
```

```
58500
```

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.

```
# Print the before for img_torch, the transpose implementation and the after img_torch to see
print(img_torch)
print(img_torch.transpose(0,2))
print(img_torch.transpose(0,2).shape)
print(img_torch)
print(img_torch.shape)
```

```
tensor([[[0.5882, 0.3725, 0.1490],
         [0.5765, 0.3608, 0.1373],
         [0.5569, 0.3412, 0.1176],
         ...,
         [0.5804, 0.3412, 0.1294],
         [0.6039, 0.3647, 0.1529],
         [0.6157, 0.3765, 0.1647]],

        [[0.5412, 0.3216, 0.0902],
         [0.5647, 0.3451, 0.1137],
         [0.5961, 0.3765, 0.1451],
         ...,
         [0.5882, 0.3490, 0.1373],
         [0.6078, 0.3686, 0.1569],
         [0.6196, 0.3804, 0.1686]],

        [[0.6157, 0.3765, 0.1529],
         [0.6196, 0.3843, 0.1490],
```

```
          [0.6196, 0.3843, 0.1412],
          ...,
          [0.5922, 0.3529, 0.1373],
          [0.6157, 0.3765, 0.1608],
          [0.6275, 0.3882, 0.1725]],

         ...,

         [[0.6039, 0.3882, 0.1686],
          [0.6078, 0.3922, 0.1686],
          [0.6118, 0.3961, 0.1725],
          ...,
          [0.3804, 0.3098, 0.2157],
          [0.3765, 0.3059, 0.2118],
          [0.3765, 0.3098, 0.2078]],

         [[0.5882, 0.3725, 0.1529],
          [0.6078, 0.3922, 0.1725],
          [0.6196, 0.4039, 0.1804],
          ...,
          [0.3882, 0.3176, 0.2314],
          [0.3804, 0.3098, 0.2157],
          [0.3804, 0.3098, 0.2157]],

         [[0.5804, 0.3647, 0.1451],
          [0.6039, 0.3882, 0.1686],
          [0.6235, 0.4078, 0.1882],
          ...,
          [0.4196, 0.3373, 0.2549],
          [0.4039, 0.3216, 0.2392],
          [0.3961, 0.3137, 0.2314]]])
tensor([[[0.5882, 0.5412, 0.6157,  ..., 0.6039, 0.5882, 0.5804],
          [0.5765, 0.5647, 0.6196,  ..., 0.6078, 0.6078, 0.6039],
          [0.5569, 0.5961, 0.6196,  ..., 0.6118, 0.6196, 0.6235],
          ...,
          [0.5804, 0.5882, 0.5922,  ..., 0.3804, 0.3882, 0.4196],
          [0.6039, 0.6078, 0.6157,  ..., 0.3765, 0.3804, 0.4039],
          [0.6157, 0.6196, 0.6275,  ..., 0.3765, 0.3804, 0.3961]],
```

Explanation: The original tensor stays the same even after the transpose is applied. Its value is not updated. The code img_torch.transpose(0,2) transposes the original tensor based on the dimensions passed in. Thus, in this context, the values of the first dimension of the tensor are swapped with the values of the third dimension of the tensor.

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.

```
# Print the before for img_torch, the unsqueeze implementation and the after img_torch to see
print(img_torch)
print(img_torch.unsqueeze(0))
print(img_torch.unsqueeze(0).shape)
print(img_torch)
print(img_torch.shape)
```

```
          [[0.5804, 0.3647, 0.1451],
           [0.6039, 0.3882, 0.1686],
           [0.6235, 0.4078, 0.1882],
           ...,
           [0.4196, 0.3373, 0.2549],
           [0.4039, 0.3216, 0.2392],
           [0.3961, 0.3137, 0.2314]]]])
torch.Size([1, 130, 150, 3])
tensor([[[0.5882, 0.3725, 0.1490],
         [0.5765, 0.3608, 0.1373],
         [0.5569, 0.3412, 0.1176],
         ...,
         [0.5804, 0.3412, 0.1294],
         [0.6039, 0.3647, 0.1529],
         [0.6157, 0.3765, 0.1647]],

        [[0.5412, 0.3216, 0.0902],
         [0.5647, 0.3451, 0.1137],
         [0.5961, 0.3765, 0.1451],
         ...,
         [0.5882, 0.3490, 0.1373],
         [0.6078, 0.3686, 0.1569],
         [0.6196, 0.3804, 0.1686]],

        [[0.6157, 0.3765, 0.1529],
         [0.6196, 0.3843, 0.1490],
         [0.6196, 0.3843, 0.1412],
         ...,
         [0.5922, 0.3529, 0.1373],
         [0.6157, 0.3765, 0.1608],
         [0.6275, 0.3882, 0.1725]],

        ...,

        [[0.6039, 0.3882, 0.1686],
         [0.6078, 0.3922, 0.1686],
         [0.6118, 0.3961, 0.1725],
         ...,
         [0.3804, 0.3098, 0.2157],
         [0.3765, 0.3059, 0.2118],
         [0.3765, 0.3098, 0.2078]],

        [[0.5882, 0.3725, 0.1529],
         [0.6078, 0.3922, 0.1725],
         [0.6196, 0.4039, 0.1804],
         ...,
         [0.3882, 0.3176, 0.2314],
         [0.3804, 0.3098, 0.2157],
         [0.3804, 0.3098, 0.2157]],
```

```
        [[0.5804, 0.3647, 0.1451],
         [0.6039, 0.3882, 0.1686],
         [0.6235, 0.4078, 0.1882],
         ...,
         [0.4196, 0.3373, 0.2549],
         [0.4039, 0.3216, 0.2392],
         [0.3961, 0.3137, 0.2314]]])
torch.Size([130, 150, 3])
```

Explanation: The code img_torch.unsqueeze(0) adds a new dimension to the tensor and places it before the other dimensions. The original tensor is not updated after doing unsqueeze(0).

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

```
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)
        self.layer1 = nn.Linear(28 * 28, 128)
        self.layer2 = nn.Linear(128, 16)
        self.layer3 = nn.Linear(16, 1)
    def forward(self, img):
        flattened = img.view(-1, 28 * 28)
        activation1 = self.layer1(flattened)
        activation1 = F.relu(activation1)
        activation2 = self.layer2(activation1)
        activation2 = F.relu(activation2)
        activation3 = self.layer3(activation2)
        return activation3

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)

# Adding more training iterations
for epoch 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))
```

```
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 data/MNIST/ra
100%                                9912422/9912422 [00:00<00:00, 10418477.32it/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 data/MNIST/ra
100%                                28881/28881 [00:00<00:00, 1060055.25it/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 data/MNIST/raw
100%                                1648877/1648877 [00:00<00:00, 7487022.88it/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 data/MNIST/raw
100%                                4542/4542 [00:00<00:00, 171082.31it/s]
Extracting data/MNIST/raw/t10k-labels-idx1-ubyte.gz to data/MNIST/raw

Training Error Rate: 0.005
Training Accuracy: 0.995
Test Error Rate: 0.055
Test Accuracy: 0.945
```

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?

Increasing the number of training iterations, also known as epochs, resulted in the best accuracy on training data. The accuracy I was able to achieve with 6 epochs was 99.9% on the training data. The accuracy on the training data stayed consistent with 99.9% even after I increased the number of epochs gradually from 6 to 15. One more change I made was changing the learning rate to 0.003 from the initial rate of 0.005. I was then able to achieve a training data accuracy of 100%.

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?

I used the same number of epochs as part a. The testing data accuracy then was 93.4%. One way I improved the testing data accuracy was reducing the number of epochs to 5 from the 6 I had in part a; I kept the learning rate the same (0.003) and then got a testing data accuracy of 93.9%. To further improve this, I added another layer to the ANN. This addition of a hidden layer improved my testing data accuracy to 94.5% but the trade-off was that my training data accuracy is now 99.5%.

Part (c) -- 4 pt

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

I would use the hyperparameters from part (b). This is because we get more accurate results for test data, which is more applicable as we can better predict new data that currently isn't at our disposal. Furthermore, a higher training data accuracy is sometimes indicative of overfitting the data which doesn't give good insight on predictions for test data. Thus, due to the real-life applicability of predicting new data, I would choose the hyperparamaters from part (b) as it gives us the highest accuracy for test data.

✓ 7s completed at 6:47 PM