Lab 1. PyTorch and ANNs

Deadline: Monday, Jan 25, 5:00pm.

Total: 30 Points

Late Penalty: There is a penalty-free grace period of one hour past the deadline. Any work that is submitted between 1 hour and 24 hours past the deadline will receive a 20% grade deduction. No other late work is accepted. Quercus submission time will be used, not your local computer time. You can submit your labs as many times as you want before the deadline, so please submit often and early.

Grading TA: Justin Beland, Ali Khodadadi

This lab is based on assignments developed by Jonathan Rose, Harris Chan, Lisa Zhang, and Sinisa Colic.

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://docs.scipy.org/doc/numpy/reference/)
- https://pytorch.org/docs/stable/torch.html (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://drive.google.com/file/d/1Kz8-jV3ICCpL4H2cyAXRZ3HhJkHO8XO0/view?usp=sharing (https://drive.google.com/file/d/1Kz8-jV3ICCpL4H2cyAXRZ3HhJkHO8XO0/view?usp=sharing)

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/ (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`.

In [ ]:
```
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
    """
    #If the type of input is not an integer or is less than 1, return "Invalid
Input" error.
    if type(n) is not int or n <= 0:
      print("Invalid input")
      return -1

    ans = 0

    #Calculate the sum of cubes in the while loop iteratively
    while n > 0:
      ans += n ** 3
      n -= 1

    print(ans)
    return ans

#Checking if the function works
sum_of_cubes("3")
sum_of_cubes(3)
sum_of_cubes(6)
sum_of_cubes(-1)
```

```
Invalid input
36
441
Invalid input
```

Out[ ]:
```
-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 (https://docs.python.org/3.6/library/stdtypes.html#str.split)

In [ ]:
```
help(str.split)
```

```
Help on method_descriptor:

split(...)
    S.split(sep=None, maxsplit=-1) -> list of strings

    Return a list of the words in S, using sep as the
    delimiter string.  If maxsplit is given, at most maxsplit
    splits are done. If sep is not specified or is None, any
    whitespace string is a separator and empty strings are
    removed from the result.
```

In [ ]:
```
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]
    """
    #Generate a list of strings, which are the words of the input sentence
    myList = sentence.split()
    answer = []

    #Count the length of the words in the list iteratively
    for word in myList:
      answer.append(len(word))

    return answer

#Checking if the function works
word_lengths("welcome to APS360!")
```

Out[ ]:
```
[7, 2, 7]
```

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.

In [ ]:
```
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
    >>> all_same_length("hello world")
    True
    """
    array = word_lengths(sentence)

    #If any of the words in the list is not the same length as the first word
 in the array, then return False
    for length in array:
      if length != array[0]:
        return False

    return True

#Checking if the function works
all_same_length("hello world")
```

Out[ ]:
```
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`.

In [ ]:
```
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.

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

In [ ]:
```
#<NumpyArray>.size returns the total number of elements in the array
matrix.size
```

Out[ ]:
```
12
```

In [ ]:
```
#<NumpyArray>.shape returns the dimensions of the array in (rows, columns) for
mat
matrix.shape
```

Out[ ]:
```
(3, 4)
```

In [ ]:
```
#<NumpyArray>.size returns the total number of elements in the array
vector.size
```

Out[ ]:
```
4
```

In [ ]:
```
#<NumpyArray>.shape returns the dimensions of the array in (rows, columns) for
mat
vector.shape
```

Out[ ]:
```
(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

In [ ]:
```
output = None
```

In [ ]:
```
output = []
#Performing matrix multiplication iteratively with nested for loops
for x in range(matrix.shape[0]):
  sum = 0
  for y in range(matrix.shape[1]):
    sum += matrix[x, y] * vector[y]
  output.append(sum)
output = np.array(output)
print (output)
```

```
[ 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.

In [ ]:
```
output2 = None
```

In [ ]:
```
#Performing matrix multiplication using np.dot() function
output2 = np.dot(matrix, vector)
print (output2)
```

```
[ 4.  8. -3.]
```

Part (d) -- 1pt

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

In [ ]:
```
#Checking whether all elements of output match all elements of output2
print (output == output2)
#if all elements return True, that means that they match, else they are incons
istent
```

```
[ 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:

In [ ]:
```
import time

#Diff1 will hold the time it takes to perform matrix multiplication
#where output = matrix x vector by using for loops to iterate through the colu
mns and rows

#Record the time before running code
start_time1 = time.time()

#Place code to run here
output = []
for x in range(matrix.shape[0]):
  sum = 0
  for y in range(matrix.shape[1]):
    sum += matrix[x, y] * vector[y]
  output.append(sum)
output = np.array(output)
print (output)

#Record the time after the code is run
end_time1 = time.time()

#Compute the difference
diff1 = end_time1 - start_time1

#Diff2 will hold the time it takes to perform matrix multiplication using nump
y.dot() function
# record the time before running code
start_time2 = time.time()

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

#Record the time after the code is run
end_time2 = time.time()

#Print the differences in time
#If diff1 - diff2 > 0, then numpy.dot() is faster
#Else if diff1 - diff2 < 0, then the original matrix multiplication is faster
#Compute the difference
diff2 = end_time2 - start_time2

#As we can see diff1 - diff2 > 0, meaning numpy.dot() is faster
print (diff1 - diff2)
```

```
[ 4.  8. -3.]
[ 4.  8. -3.]
0.00021910667419433594
```

Part 3. Images [6 pt]

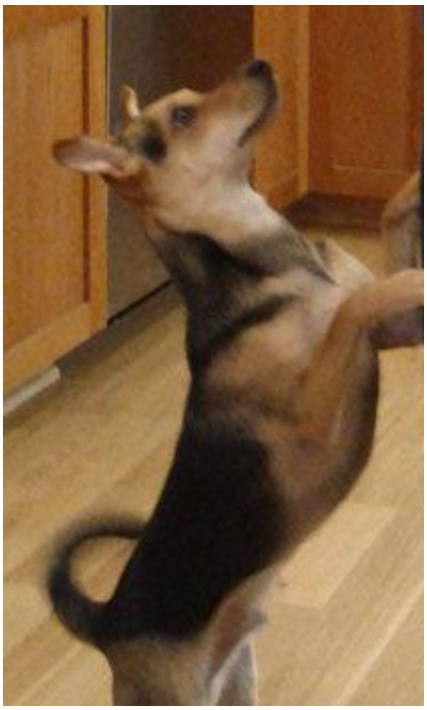
A picture or image can be represented as a NumPy array of "pixels", with dimensions H × W × C, where H is the height of the image, W is the width of the image, and C is the number of colour channels. Typically we will use an image with channels that give the the Red, Green, and Blue "level" of each pixel,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

In [ ]:
```
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 (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.

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

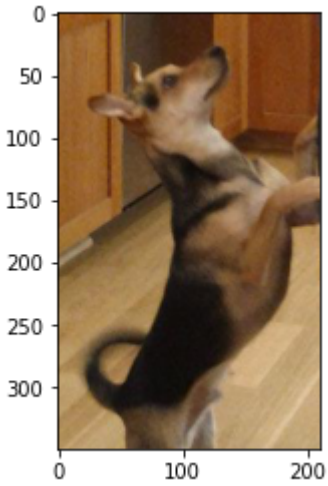
Part (b) -- 1pt

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.

In [ ]:
```
plt.imshow(img)
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7f4421cb2e48>
```

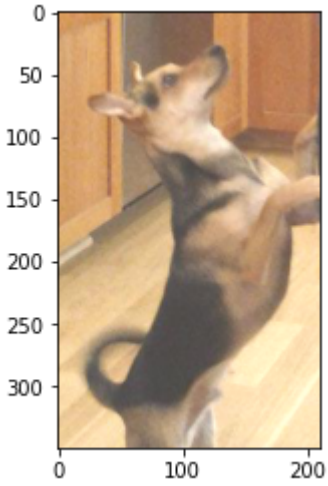


Part (c) -- 2pt

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`.

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

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7f4421d1f588>
```



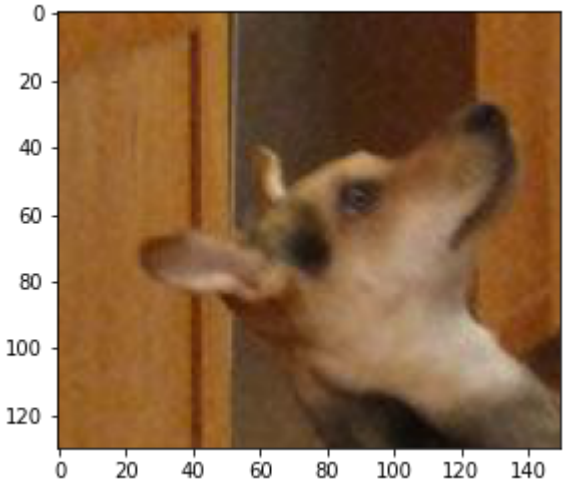
Part (d) -- 2pt

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.

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

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7f441e91dcf8>
```



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.

In [ ]:
```
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` .

In [ ]:
```
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` .

In [ ]:
```
img_torch.shape
```

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

Part (c) -- 1pt

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

In [ ]:
```
#Calculating the floating-point number count iteratively in the tensor img_tor
ch
floatPointNumCount = 1
for x in list(img_torch.shape):
  floatPointNumCount *= x
floatPointNumCount
```

Out[ ]:
```
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.

In [ ]:
```
#Original img_torch
print(img_torch)

#Transposed img_torch
print(img_torch.transpose(0,2))

#Orignal img_torch stays the same
print(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]]])
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]],
```

```
                [[0.3725, 0.3216, 0.3765,  ..., 0.3882, 0.3725, 0.3647],
                 [0.3608, 0.3451, 0.3843,  ..., 0.3922, 0.3922, 0.3882],
                 [0.3412, 0.3765, 0.3843,  ..., 0.3961, 0.4039, 0.4078],
                 ...,
                 [0.3412, 0.3490, 0.3529,  ..., 0.3098, 0.3176, 0.3373],
                 [0.3647, 0.3686, 0.3765,  ..., 0.3059, 0.3098, 0.3216],
                 [0.3765, 0.3804, 0.3882,  ..., 0.3098, 0.3098, 0.3137]],

                [[0.1490, 0.0902, 0.1529,  ..., 0.1686, 0.1529, 0.1451],
                 [0.1373, 0.1137, 0.1490,  ..., 0.1686, 0.1725, 0.1686],
                 [0.1176, 0.1451, 0.1412,  ..., 0.1725, 0.1804, 0.1882],
                 ...,
                 [0.1294, 0.1373, 0.1373,  ..., 0.2157, 0.2314, 0.2549],
                 [0.1529, 0.1569, 0.1608,  ..., 0.2118, 0.2157, 0.2392],
                 [0.1647, 0.1686, 0.1725,  ..., 0.2078, 0.2157, 0.2314]]])
        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]]])
```

Interpretation:
This function: img_torch.transpose(0,2) is the transpose function that transposes the tensor given to it based on the dimension give. In this case, the y-axis (0-dimension) and the z-axis(2-dimension) are interchange (transposed). Then the function returns the transposed tensor. The original tensor is not replaced with the transposed tensor. The original tensor stays the same.

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.

In [ ]:
```
#Original img_torch
print(img_torch)

#Using the unsqueeze function
print(img_torch.unsqueeze(0))

#Checking img_torch
print(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]]])
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, 0.3725, 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]]])
```

Interpretation:
The function:img_torch.unsqueeze(0) returns a new tensor with a dimension of size one inserted at specified position 0. As we notices above, that the unsqueeze function returns a new function and doesn't change the orginal img_torch tensor.

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` .

In [ ]:
```
maxVal = torch.max(img_torch, 1)
maxVal[0][0]
```

Out[ ]:
```
tensor([0.6941, 0.4941, 0.3490])
```

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

In [ ]:
```
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__()
        #Controlling the number of hidden units here
        self.layer1 = nn.Linear(28 * 28, 30)
        self.layer2 = nn.Linear(30, 1)
        #Controlling the number of layers
        #self.layer3 = nn.Linear(100, 1)
        #self.layer4 = nn.Linear(50, 1)
    def forward(self, img):
        flattened = img.view(-1, 28 * 28)
        activation1 = self.layer1(flattened)
        activation1 = F.relu(activation1)

        #Controlling the type of activation function
        #activation1 = torch.tanh(activation1)
        #activation1 = torch.relu(activation1)

        activation2 = self.layer2(activation1)

        #Controlling the number of layers
        #activation2 = F.relu(activation2)
        #activation3 = self.layer3(activation2)
        #activation3 = F.relu(activation3)
        #activation4 = self.layer4(activation3)
        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" t
ask
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.SGD(pigeon.parameters(), lr=0.005, momentum=0.9)

#Controlling the learning rate
#optimizer = optim.SGD(pigeon.parameters(), lr=0.1, momentum=0.9)

#Controlling the number of training iterations
```

```
for x in range(0,1):
  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 pa
rameter)
      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: 0.036
Training Accuracy: 0.964
Test Error Rate: 0.079
Test Accuracy: 0.921
```

Changing parameters results

Original Data:
Training Error Rate: 0.036
Training Accuracy: 0.964
Test Error Rate: 0.079
Test Accuracy: 0.921

Number of training iterations = 2:
Training Error Rate: 0.016
Training Accuracy: 0.984
Test Error Rate: 0.057
Test Accuracy: 0.943

Number of training iterations = 10:
Training Error Rate: 0.001
Training Accuracy: 0.999
Test Error Rate: 0.059
Test Accuracy: 0.9410000000000001

Number of hidden units = 10:
Training Error Rate: 0.047
Training Accuracy: 0.953
Test Error Rate: 0.104
Test Accuracy: 0.896

Number of hidden units = 100:
Training Error Rate: 0.03
Training Accuracy: 0.97
Test Error Rate: 0.077
Test Accuracy: 0.923

Number of layers = 1 [Hidden units:->1]:
Training Error Rate: 0.052
Training Accuracy: 0.948
Test Error Rate: 0.114
Test Accuracy: 0.886

Number of layers = 3 [Hidden units:->200->100->1]:
Training Error Rate: 0.035
Training Accuracy: 0.965
Test Error Rate: 0.082
Test Accuracy: 0.918

Number of layers = 4 [Hidden units:->200->100->50->1]:
Training Error Rate: 0.047
Training Accuracy: 0.953
Test Error Rate: 0.091
Test Accuracy: 0.909

Type of activation function = torch.relu:
Training Error Rate: 0.036
Training Accuracy: 0.964
Test Error Rate: 0.079
Test Accuracy: 0.921

Type of activation function = torch.tanh:
Training Error Rate: 0.04
Training Accuracy: 0.96
Test Error Rate: 0.094
Test Accuracy: 0.906

Learning Rate = 0.001:
Training Error Rate: 0.078
Training Accuracy: 0.922
Test Error Rate: 0.113
Test Accuracy: 0.887

Learning Rate = 0.01:
Training Error Rate: 0.039
Training Accuracy: 0.961
Test Error Rate: 0.082
Test Accuracy: 0.918

Learning Rate = 0.1:
Training Error Rate: 0.312
Training Accuracy: 0.688
Test Error Rate: 0.297
Test Accuracy: 0.7030000000000001

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?

As witnessed in the results listed above, that increasing the number of training iterations improved the accuracy of the training data the most. With 10 training iterations, I was able to achieve a training accuracy of 0.999. Increasing the number of hidden units also increrased the accuracy of the training data. With a 100 hidden units, I was able to acheive a training accuracy of 0.97.
Combining both of these changes can perhaps increase accuracy on training data even more.

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?

As noted in the results above, increasing the number of training iterations to 2, increased the accuracy of the testing data the most. With 2 training iterations, I was able to achieve a testing accuracy of 0.943.
Increasing the number of hidden units to 100 also helped increase the accuracy of testing data to 0.923.
It seems that increasing the value of some parameters to a certain extent helps improve the accuracy of testing data by a little amount. However, if we increase this amount by too much, then the accuracy of the testing data decreases.
To find the highest testing data accuracy, we must fine tune some parameters by a little amount in combination. Doing so would result in the greatest testing data accuracy.

Part (c) -- 4 pt

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

Evidently, we should use the model hyperparameters from part (b) as those to show how well the model is performing. This is because we can easily improve the training data accuracy by increasing the number of iterations of training data. However, a high training data accuracy often means that our model has been overfitted. Here, the model is extremely specialized to the training data, and is not general enough for new data or testing data.
On the other hand, a higher testing data accuracy is a much better indicator of a better model. This is because the model is able to accept new data that it has never seen before, and provide the most accurate responses. The model is able to interpret new data and give the best predictions and provide a generalized trend that is correctly fitted.
In essence, the model hyperparameters we should use, are the ones from (b) that lead us to the best testing data accuracy.

In [ ]:
```
%%shell
jupyter nbconvert --to html /content/Lab_1_PyTorch_and_ANNs.ipynb
```

```
[NbConvertApp] Converting notebook /content/Lab_1_PyTorch_and_ANNs.ipynb to h
tml
[NbConvertApp] Writing 592125 bytes to /content/Lab_1_PyTorch_and_ANNs.html
```

Out[ ]: