Name: Hee Seung Hong

Student Number: 1005216650

Colab Link: https://colab.research.google.com/drive/1N8mrnGIjRc4BLjFqw1sUYfUtpfbV5oxK?usp=sharing (https://colab.research.google.com/drive/1N8mrnGIjRc4BLjFqw1sUYfUtpfbV5oxK?usp=sharing)

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.

In [38]:
```
##%%shell
##jupyter nbconvert --to html /content/Lab_1_PyTorch_and_ANNs.ipynb
```

Colab Link

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

Colab Link: https://colab.research.google.com/drive/1N8mrnGIjRc4BLjFqw1sUYfUtpfbV5oxK?usp=sharing (https://colab.research.google.com/drive/1N8mrnGIjRc4BLjFqw1sUYfUtpfbV5oxK?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 [39]:
```
def sum_of_cubes(n):
    sum = 0

    if n <= 0 or type(n) != int:
        print('Invalid input')
        return -1

    else:
        for i in range(1, n+1):
            sum += i*i*i
        return sum
```

In [40]:
```
sum_of_cubes(3)
```

Out[40]:
```
36
```

In [41]:
```
sum_of_cubes(3.5)
```

```
Invalid input
```

Out[41]:
```
-1
```

In [42]:
```
sum_of_cubes(0)
```

```
Invalid input
```

Out[42]: -1

In [43]:
```
sum_of_cubes(-1)
```

```
Invalid input
```

Out[43]: -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 are not 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)](https://docs.python.org/3.6/library/stdtypes.html#str.split)

In [44]:
```
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 [45]:
```
def word_lengths(sentence):
    words = sentence.split(' ') #list of words from sentence
    n = len(words) #length of the list words

    wordcount = [] #list of the length of words in sentence

    for i in range(0, n):
        wordcount.append(len(words[i]))

    return wordcount
```

In [46]:
```
word_lengths("welcome to APS360!")
```

Out[46]: [7, 2, 7]

In [47]:
```
word_lengths("machine learning is so cool")
```
Out[47]:
```
[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.

In [48]:
```
def all_same_length(sentence):
    lengths = word_lengths(sentence)
    temp = lengths[0]
    same_or_no = True

    for item in lengths:
        if temp != item:
            same_or_no = False
            break

    return same_or_no

    """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
    """
```

In [49]:
```
all_same_length("all same length")
```
Out[49]:
```
False
```

In [50]:
```
all_same_length("hello world")
```
Out[50]:
```
True
```

In [51]:
```
all_same_length("hello world apple white")
```
Out[51]:
```
True
```

In [52]:
```
all_same_length("hello world hi white")
```
Out[52]:
```
False
```

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 [53]:
```
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.

I think that `<NumpyArray>.size` returns the total number of elements in the NumpyArray

and `<NumpyArray>.shape` returns the dimensions of the NumpyArray.

In [54]:
```
matrix = np.array([[1., 2., 3., 0.5],
                   [4., 5., 0., 0.],
                   [-1., -2., 1., 1.]])
vector = np.array([2., 0., 1., -2.])
```

In [55]:
```
matrix.size
```

Out[55]:
```
12
```

In [56]:
```
matrix.shape
```

Out[56]:
```
(3, 4)
```

In [57]:
```
vector.size
```

Out[57]:
```
4
```

In [58]:
```
vector.shape
```

Out[58]:
```
(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 [59]:
```
output = np.array([0,0,0])
for i in range(0, len(matrix)):
    for j in range(0, len(vector)):
        output[i] += matrix[i][j] * vector[j]
```

In [60]:
```
output
```

Out[60]:
```
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.

In [61]:
```
output2 = np.dot(matrix, vector)
```

In [62]:
```
output2
```

Out[62]:
```
array([ 4.,  8., -3.])
```

Part (d) -- 1pt

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

In [63]:
```
is_equal = output == output2
arrays_equal = is_equal.all()
print(arrays_equal)
```

```
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 [64]:
```python
import time

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

# place code to run here
for i in range(10000):
    99*99

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

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

Out[64]:
```
0.00146484375
```

In [65]:
```python
import time

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

# Matrix by For Loop
output = np.array([0,0,0])
for i in range(0, len(matrix)):
    for j in range(0, len(vector)):
        output[i] += matrix[i][j] * vector[j]

# record the time after the For Loop matrix is run
end_time1 = time.time()

# Matrix by np.dot
output2 = np.dot(matrix, vector)

# record the time after the np.dot is run
end_time2 = time.time()

# compute the difference
time_forloop = end_time1 - start_time
time_npdot = end_time2 - end_time1
diff = time_npdot - time_forloop
diff #if diff is negative, it means that npdot was faster
```

Out[65]:
```
-0.0002396106719970703
```

Part 3. Images [6 pt]

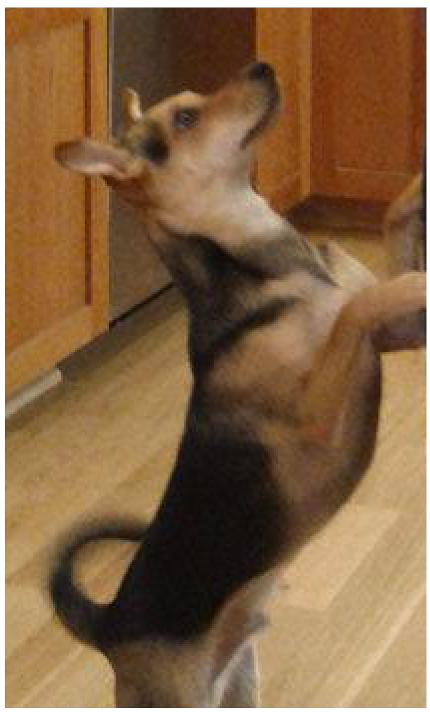
A picture or image can be represented as a NumPy array of “pixels”, with dimensions H × W × C, where **H is the height** of the image, **W is the width of the image**, and **C is the number of colour channels**. Typically we will use an image with channels that give the the Red, Green, and Blue “level” 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 [66]:
```
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 [67]:
```
img = plt.imread("https://drive.google.com/uc?export=view&id=1oaLVR2hr1_
qzpKQ47i9rVUIklwbDcews")
```

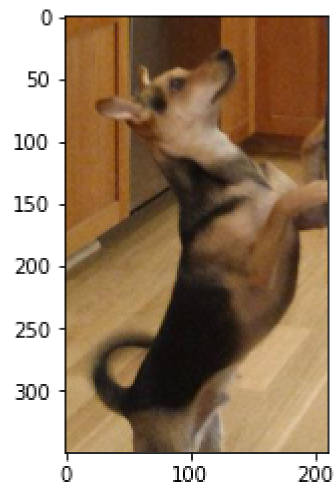
Part (b) -- 1pt

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 [68]:
```
plt.imshow(img)
```

Out[68]: <matplotlib.image.AxesImage at 0x7f186fdf9438>



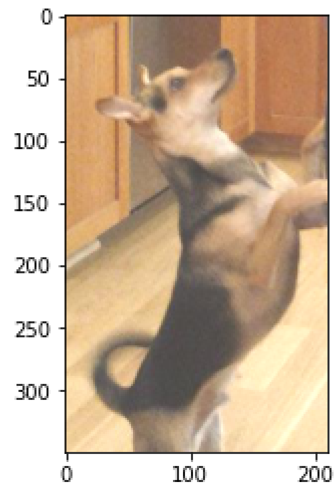
Part (c) -- 2pt

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 [69]:
```
img_add = np.clip(img + 0.25, 0, 1)

plt.imshow(img_add)
```

Out[69]: <matplotlib.image.AxesImage at 0x7f186c921da0>



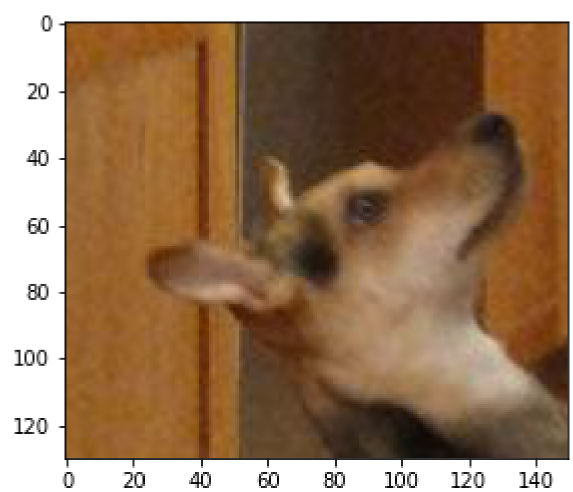
Part (d) -- 2pt

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 [70]:
```
img_cropped = img[0:130, 0:150, 0:3]
plt.imshow(img_cropped)
img_cropped.shape
```

Out[70]: (130, 150, 3)



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 [71]:
```
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 [72]:
```
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 [73]:
```
img_torch.shape
```

Out[73]:
```
torch.Size([130, 150, 3])
```

Part (c) -- 1pt

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

In [74]:
```
torch.numel(img_torch)
```

Out[74]:
```
58500
```

Number of floating-point numbers: 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.

img_torch.transpose(0,2) swaps the dimesions of dim0 and dim2 and transposes them. The expression returns a tensor that is transposed of img_torch. The original variable img_torch is not updated because img_torch.traspose(0,2) was not stored to img_torch.

In [75]:
```
img_torch
```

Out[75]:
```
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]]])
```

In [76]:
```
img_torch.transpose(0,2)
```

Out[76]:
```
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]]])
```

In [77]:
```
img_torch.shape
```

Out[77]:
```
torch.Size([130, 150, 3])
```

In [78]:
```
img_torch.transpose(0,2).shape
```

Out[78]:
```
torch.Size([3, 150, 130])
```

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.

img_torch.unsqueeze(0) adds a size 1 dimension to the particular position written (here it is position dim0). It returns a new tensor with a size 1 dimesion added at the position dim0 of img_torch. For example img_torch has the shape `([130,150,3])` whereas img_torch.unsqueeze(0) has the shape `([1,130,150,3])` . The original variable img_torch is not updated because img_torch.unsqueeze(0) was not stored to img_torch.

In [79]:
```
img_torch.unsqueeze(0)
```

Out[79]:
```
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]]]])
```

In [80]:
```
img_torch.unsqueeze(0).shape
```

Out[80]:
```
torch.Size([1, 130, 150, 3])
```

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`.

All of the Three codes below work, and give the same Maximum values of img_torch along each colour channel. I have tried torch.amax which was not from the hint, and also torch.max to be consistent with the given hint.

In [81]:
```
#Code1 using torch.amax
torch.amax(img_torch, dim=(0,1))
```

Out[81]:
```
tensor([0.8941, 0.7882, 0.6745])
```

In [82]:
```
#Code2
R = torch.max(img_torch[:,:,0])
G = torch.max(img_torch[:,:,1])
B = torch.max(img_torch[:,:,2])
RGB_Maxes = [R, G, B]
stacked_RGB_Maxes = torch.stack(RGB_Maxes)
```

In [83]:
```
#Code3
RGB_Max = torch.stack([torch.max(img_torch[:,:,0]), torch.max(img_torch
[:,:,1]), torch.max(img_torch[:,:,2])])
```

In [84]:
```
stacked_RGB_Maxes
```

Out[84]:
```
tensor([0.8941, 0.7882, 0.6745])
```

In [85]:
```
RGB_Max
```

Out[85]:
```
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

In [87]:
```
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.layer3 = nn.Linear(1500, 1)
        #self.layer4 = nn.Linear(100, 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)
        #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 recogni
tion" task
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.SGD(pigeon.parameters(), lr=0.005, momentum=0.9) #lr = 
learning rate

for ii in range(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.Float
Tensor)
        # 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 fo
r 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: 0.036
Training Accuracy: 0.964
Test Error Rate: 0.079
Test Accuracy: 0.921
```

Original Code

number of training iterations: 1

number of hidden units: 30

numbers of layers: 2

types of activation functions:

learning rate (lr): 0.005

Original Accuracy

Training Accuracy: 0.964

Test Accuracy: 0.921

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 the **number of hidden units** from 30 to 1500, the Training Accuracy was 0.984.

When increasing the **number of training iterations** from 1 to 9, the Training Accuracy was 0.999.

Both of them together made the Training Accuracy to 1.0. (Test Accuracy to 0.946)

Increasing **layers** to 3 of 1500 hidden units and decreasing the **learning rate** to 0.4 made the Test Accuracy 0.949.

In conclusion, the Trainging Accuracy was best when increasing the **number of training iterations** from 1 to 9, and increasing the **number of hidden units** from 30 to 1500, with 3 **layers**, and a **learning rate** of 0.004 made the Training Accuracy 1.0 (Test Accuracy 0.949).

In [ ]:

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?

When increasing the **number of training iterations** from 1 to 6, the Test Accuracy was 0.937. (6 was better than 7).

When increasing the **number of hidden units** from 30 to 20000, the Test Accuracy was 0.945.

Both of them together made the Test Accuracy to 0.951.

Adding **layers** of 30 hidden units made it worse.

Changing the **learning rate** decreased the Training Accuracy.

In conclusion, the Test Accuracy was best when increasing the **number of training iterations** to 6 and increasing the **number of hidden units** to 20000, which achieved an Test Accuracy of 0.951 (Training Accuracy of 1.0).

In [ ]:

Part (c) -- 4 pt

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

I should use the hyperparameters from (b) because for both (a) and (b), the Training Accuracy was 1.0, but (a) had a Test Accuracy of 0.949 and (b) had a Test Accuracy of 0.951. This means that (b) had better Test Accuracy than (a) while having the same Training Accuracy. Thus, I should use the hyperparameters from (b) because it definitely shows a better result when looking at both Training Accuracy and Test Accuracy. The model hyperparameters are **number of training iterations** of 6 and the **number of hidden units** of 20000 with no change in **number of layers** and the **learning rate**.

In [ ]:

```
```