Lab 1. PyTorch and ANNs

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

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

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

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

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

You can also reference Python API documentations freely.

What to submit

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

Do not submit any other files produced by your code.

Include a link to your colab file in your submission.

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

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

Colab Link

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

Colab Link: https://colab.research.google.com/drive/1jtKRieX0HLWOvlMCkWf8BKwRSQ-QodC8#scrollTo=_TlLBrWBIGt0

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
    """
    if n < 0:
      print("Invalid Input")
      return -1
    else:
      sum = 0
      cube =0
      for i in range(1,n+1):
        cube = i**3
        sum += cube
      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]
    """

    words_array = sentence.split(" ")

    returning = [len(i) for i in words_array]

    return returning
```

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
    """

    workable = word_lengths(sentence)

    ref = workable[0]
    i = 0
    length = len(workable)
    for i in range(0,length):
      if workable[i] != ref:
        return False

    return True
#dont know if this is the right syntax lol c++ is biting me
```

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

#This call would return the amount of entries in the matrix, a single integer value that holds the value of the number of rows times the number of columns, n*m
```

```
12
```

```
matrix.shape

#This call would return the dimensions of the matrix, which would be useful for, lets say, applying a linear transformation to some n by 1 or 1 by m vector.

#matrix.size -> output: [3,5], an array of two numbers where one represents the amount of rows and one represents the amount of columns
```

```
(3, 4)
```

Double-click (or enter) to edit

```
vector.size

#This call, again, would function equivalently to the matrix call, as a vector is a 1xn or mx1 matrix, though it would just return the integer value n or m depending on
#whether it is a row or column vector, respectively.
```

```
4
```

```
vector.shape

#This call, again similarily to the matrix version, would give the dimensions of the vector, 1xn or mx1 matrix, depending on
#whether it is a row or column vector, respectively.

#vector.shape -> output: [n,1], an array of two numbers
```

```
(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 is 3x1

output=[0] * 3

rows = 3
size = 4

for i in range(0,rows):
```

```
  dot = 0
  for j in range(0,size):
    dot += matrix[i][j]*vector[j]
  output[i] = dot

print(output)
```

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

```
output
```

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

Part (c) -- 1pt

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

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

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

print(output2)
```

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

```
output2
```

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

Part (d) -- 1pt

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

```
if all(output) == all(output2):
  print("YIPPEE!!!")

else:
  print(">:(")
```

```
YIPPEE!!!
```

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

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

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

```
0.0006799697875976562
```

```
import time

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

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

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

# compute the difference
diff1 = end_time1 - start_time1

# record the time before running code
start_time2 = time.time()

# place code to run here
output=[0] * 3

rows = 3
size = 4

for i in range(0,rows):
  dot = 0
  for j in range(0,size):
    dot += matrix[i][j]*vector[j]
  output[i] = dot

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

# compute the difference
diff2 = end_time2 - start_time2
print(diff1)
print(diff2)

#about double the speed but seems like it should be faster
```

```
0.00031447410583496094
0.0002145767211914062
```

Part 3. Images [6 pt]

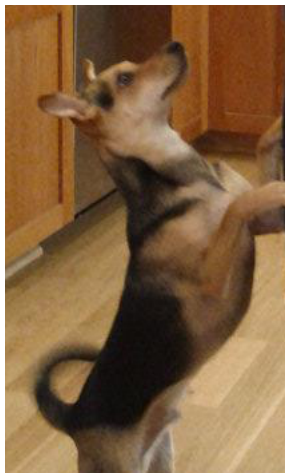
A picture or image can be represented as a NumPy array of “pixels”, with dimensions H × W × C, where H is the height of the image, W is the width of the image, and C is the number of colour channels. Typically we will use an image with channels that give the the Red, Green, and Blue “level” of each pixel,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

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

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

Part (a) -- 1 pt

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



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

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

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

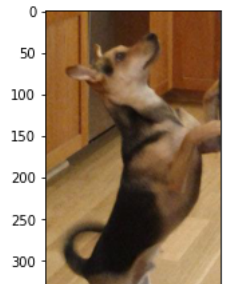
Part (b) -- 1pt

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

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

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

```
<matplotlib.image.AxesImage at 0x7f8b56df5ee0>
```



Part (c) -- 2pt

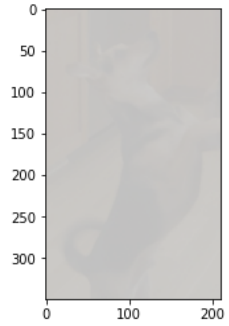
Modify the image by adding a constant value of 0.25 to each pixel in the `img` and store the result in the variable `img_add`. Note that, since the range for the pixels needs to be between [0, 1], you will also need to clip img_add to be in the range [0, 1] using `numpy.clip`. Clipping sets any value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

```
img_add = 0.25*img

np.clip(img_add, 0, 1)

plt.imshow(img_add)
```

```
<matplotlib.image.AxesImage at 0x7f8b56ede520>
```



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[20:150, 20:170]

img_last = img_cropped[:,:,:3]

plt.imshow(img_last)

#nice
```

```
<matplotlib.image.AxesImage at 0x7f8b56f08ac0>
```

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_last)
```

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`?

```
#when running the command:

img_torch

#we can see that there is in fact a floating point number used to represent
#each and every iteration of a pixel for the entire image, and since there are 3
#floats used to represent each pixel, alongside a 130x150 size tensor, we can conclude that
#there is 58500 floating point numbers.

     tensor([[[0.6353, 0.4353, 0.2275],
              [0.6431, 0.4431, 0.2353],
              [0.6510, 0.4510, 0.2431],
              ...,
              [0.4627, 0.2157, 0.0471],
              [0.4784, 0.2235, 0.0667],
              [0.5059, 0.2510, 0.0941]],

             [[0.6392, 0.4392, 0.2314],
              [0.6392, 0.4353, 0.2392],
              [0.6353, 0.4314, 0.2353],
              ...,
              [0.4784, 0.2314, 0.0627],
              [0.5098, 0.2549, 0.0980],
              [0.5176, 0.2627, 0.1059]],

             [[0.6392, 0.4392, 0.2314],
              [0.6314, 0.4275, 0.2314],
              [0.6235, 0.4196, 0.2235],
              ...,
              [0.4941, 0.2471, 0.0784],
              [0.5137, 0.2588, 0.1020],
              [0.5098, 0.2549, 0.0980]],

             ...,

             [[0.5961, 0.3765, 0.1765],
              [0.5804, 0.3608, 0.1608],
              [0.5961, 0.3765, 0.1843],
              ...,
              [0.7529, 0.6118, 0.5255],
              [0.7333, 0.5647, 0.4275],
              [0.7059, 0.5373, 0.4000]],

             [[0.6078, 0.3882, 0.1882],
              [0.6000, 0.3804, 0.1804],
              [0.6078, 0.3882, 0.1961],
              ...,
              [0.7490, 0.6000, 0.5176],
              [0.6667, 0.4941, 0.3412],
              [0.6314, 0.4588, 0.3059]],

             [[0.6118, 0.3922, 0.1922],
              [0.6000, 0.3804, 0.1804],
              [0.6039, 0.3843, 0.1922],
              ...,
```

```
               [0.6667, 0.5176, 0.4353],
               [0.6118, 0.4353, 0.2745],
               [0.5882, 0.4118, 0.2510]]])
```

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.

```
torch.transpose(img_torch, 0, 2)

#the expression returns a tranposed tensor, dim0 (0) and
#dim1 (2) are swapped.

#the original variable is not updated.

#is this even right?

     tensor([[[0.6353, 0.6392, 0.6392,  ..., 0.5961, 0.6078, 0.6118],
              [0.6431, 0.6392, 0.6314,  ..., 0.5804, 0.6000, 0.6000],
              [0.6510, 0.6353, 0.6235,  ..., 0.5961, 0.6078, 0.6039],
              ...,
              [0.4627, 0.4784, 0.4941,  ..., 0.7529, 0.7490, 0.6667],
              [0.4784, 0.5098, 0.5137,  ..., 0.7333, 0.6667, 0.6118],
              [0.5059, 0.5176, 0.5098,  ..., 0.7059, 0.6314, 0.5882]],

             [[0.4353, 0.4392, 0.4392,  ..., 0.3765, 0.3882, 0.3922],
              [0.4431, 0.4353, 0.4275,  ..., 0.3608, 0.3804, 0.3804],
              [0.4510, 0.4314, 0.4196,  ..., 0.3765, 0.3882, 0.3843],
              ...,
              [0.2157, 0.2314, 0.2471,  ..., 0.6118, 0.6000, 0.5176],
              [0.2235, 0.2549, 0.2588,  ..., 0.5647, 0.4941, 0.4353],
              [0.2510, 0.2627, 0.2549,  ..., 0.5373, 0.4588, 0.4118]],

             [[0.2275, 0.2314, 0.2314,  ..., 0.1765, 0.1882, 0.1922],
              [0.2353, 0.2392, 0.2314,  ..., 0.1608, 0.1804, 0.1804],
              [0.2431, 0.2353, 0.2235,  ..., 0.1843, 0.1961, 0.1922],
              ...,
              [0.0471, 0.0627, 0.0784,  ..., 0.5255, 0.5176, 0.4353],
              [0.0667, 0.0980, 0.1020,  ..., 0.4275, 0.3412, 0.2745],
              [0.0941, 0.1059, 0.0980,  ..., 0.4000, 0.3059, 0.2510]]])
```

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.

```
torch.unsqueeze(img_torch, 0)

#Torch.unsqueeze adds an additional dimesion to the tensor, in this case creating an
#additional layer on the axis 0

#The original variable does not update, as a new tensor is created.

     tensor([[[[0.6353, 0.4353, 0.2275],
               [0.6431, 0.4431, 0.2353],
               [0.6510, 0.4510, 0.2431],
               ...,
               [0.4627, 0.2157, 0.0471],
```

```
 [0.4784, 0.2235, 0.0667],
 [0.5059, 0.2510, 0.0941]],

[[0.6392, 0.4392, 0.2314],
 [0.6392, 0.4353, 0.2392],
 [0.6353, 0.4314, 0.2353],
 ...,
 [0.4784, 0.2314, 0.0627],
 [0.5098, 0.2549, 0.0980],
 [0.5176, 0.2627, 0.1059]],

[[0.6392, 0.4392, 0.2314],
 [0.6314, 0.4275, 0.2314],
 [0.6235, 0.4196, 0.2235],
 ...,
 [0.4941, 0.2471, 0.0784],
 [0.5137, 0.2588, 0.1020],
 [0.5098, 0.2549, 0.0980]],

...,

[[0.5961, 0.3765, 0.1765],
 [0.5804, 0.3608, 0.1608],
 [0.5961, 0.3765, 0.1843],
 ...,
 [0.7529, 0.6118, 0.5255],
 [0.7333, 0.5647, 0.4275],
 [0.7059, 0.5373, 0.4000]],

[[0.6078, 0.3882, 0.1882],
 [0.6000, 0.3804, 0.1804],
 [0.6078, 0.3882, 0.1961],
 ...,
 [0.7490, 0.6000, 0.5176],
 [0.6667, 0.4941, 0.3412],
 [0.6314, 0.4588, 0.3059]],

[[0.6118, 0.3922, 0.1922],
 [0.6000, 0.3804, 0.1804],
 [0.6039, 0.3843, 0.1922],
 ...,
 [0.6667, 0.5176, 0.4353],
 [0.6118, 0.4353, 0.2745],
 [0.5882, 0.4118, 0.2510]]])
```

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`.

```
torch.stack((torch.max(img_torch[::,::,0]), torch.max(img_torch[::,::,1]), torch.max(img_torch[::,::,2])),-1)
```

```
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))
```

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

```
#General tables included here

#Learning Rate
#+--+--+--+--+--+
#| 0.005->0.01 | Training Error Rate: 0.039, Training Accuracy: 0.961, Test Error Rate: 0.082, Test Accuracy: 0.918
#+--+--+--+--+--+
#| 0.01->0.1 | Training Error Rate: 0.312, Training Accuracy: 0.688, Test Error Rate: 0.297, Test Accuracy: 0.7030000000000001
#+--+--+--+--+--+
#| 0.1->0.0004 | Training Error Rate: 0.116, Training Accuracy: 0.884, Test Error Rate: 0.146, Test Accuracy: 0.854
#+--+--+--+--+--+

#Training Iterations
#+--+--+--+--+--+
#| 1000->2000 | Training Error Rate: 0.042, Training Accuracy: 0.958, Test Error Rate: 0.041, Test Accuracy: 0.959
#+--+--+--+--+--+
#| 2000->5000 | Training Error Rate: 0.0288, Training Accuracy: 0.9712, Test Error Rate: 0.047, Test Accuracy: 0.953
#+--+--+--+--+--+
#| 5000->100 | Training Error Rate: 0.22, Training Accuracy: 0.78, Test Error Rate: 0.208, Test Accuracy: 0.792
#+--+--+--+--+--+

#Hidden Units
#+--+--+--+--+--+
#| 30->45 | Training Error Rate: 0.034, Training Accuracy: 0.966, Test Error Rate: 0.071, Test Accuracy: 0.929
```

```
#| 30->45 | Training Error Rate: 0.034, Training Accuracy: 0.966, Test Error Rate: 0.071, Test Accuracy: 0.929
#+--+--+--+--+--+
#| 45->200 | Training Error Rate: 0.028, Training Accuracy: 0.972, Test Error Rate: 0.073, Test Accuracy: 0.927
#+--+--+--+--+--+
#| 200->10 | Training Error Rate: 0.047, Training Accuracy: 0.953, Test Error Rate: 0.104, Test Accuracy: 0.896
#+--+--+--+--+--+

#Of the above changes, the one that produced the highest training data accuracy was increasing the training iterations to 5000 instead of 1000. This makes sense, as increasing
#training iterations is one of the things that is guaranteed to increase training dataset accuracy, though it can lead to bias if too many iterations are performed. A close second
#was refining the training rate. Training Accuracy: 0.9712
```

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?

```
#Of the above changes, the one that produced the greatest accuracy on testing data was in fact, again, increasing training iterations, though this time to 2000.
#There seems to exist some sweet spot range where the iterations are large enough to serve as a good training set whilst not introducing too much bias. Test Accuracy: 0.959
```

Part (c) -- 4 pt

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

```
#I believe that the model should be able to score the highest accuracy on the training data, as this is what the model would be expected to do in a
#"real life" scenario, where it is expected to analyze some new dataset and make accurate conclusions. As such, I beliece that the model hyper parameters should be the ones from
#b), as we want don't want to introduce too much validation error. Therefore, the training sample set should be increase slightly, and it also seems that choosing a slightly lower
#learning rate would also serve to be beneficial in this regard.
```