Sharable Link:
https://colab.research.google.com/drive/1lj6hFHkqw_SilH1cAqMjQk1Q5P7N31Dd?usp=sharing

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):

    sum = 0

    if type(n) != int or n < 0:
      return -1

    for i in range(n+1):
      sum += i**3
    return sum

sum_of_cubes(3)
```

```
36
```

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 substrings in the string, using sep as the
separator string.

      sep
```

```
        The separator used to split the string.

        When set to None (the default value), will split on any
whitespace
        character (including \\n \\r \\t \\f and spaces) and will
discard
        empty strings from the result.
      maxsplit
        Maximum number of splits (starting from the left).
        -1 (the default value) means no limit.

    Note, str.split() is mainly useful for data that has been
intentionally
    delimited.  With natural text that includes punctuation, consider
using
    the regular expression module.


def word_lengths(sentence):

    sentence = sentence.split()

    arr = []

    for i in range(len(sentence)):
      arr.append(len(sentence[i]))

    return arr

word_lengths("Hello my name is")

[5, 2, 4, 2]
```

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."""

    arr = word_lengths(sentence)

    for i in range(len(arr)):
      if arr[i] != arr[0]:
        return False

    return True
```

```
all_same_length("mama miaa")

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
import time
```

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
"""number of elements included in the matrix (array of arrays)"""

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call
last)
<ipython-input-47-51ac9e5fef4c> in <cell line: 1>()
----> 1 matrix.size

NameError: name 'matrix' is not defined

matrix.shape
"""the dimensions of the matrix, (a,b) where a and b are number of
elements in rows and columns respectively"""

vector.size
"""number of elements, or length, of the vector"""

vector.shape
"""similar to the dimensions of the matrix, except only one row is
present"""
```

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 = np.array([])

start_time = time.time()

row, col = matrix.shape

for i in range(row):
  sum = 0
  for j in range(col):
    sum += matrix[i][j]*vector[j]
  output = np.append(output, sum)

end_time = time.time()
diff = end_time - start_time

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.

```
output2 = np.array([])

start_time2 = time.time()
output2 = np.dot(matrix, vector)

end_time2 = time.time()
diff2 = end_time2 - start_time2

print(output2)
```

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

Part (d) -- 1pt

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

```
print(output == output2)

[ True  True  True]
```

Part (e) -- 1pt

Show that using `np.dot` is faster than using your code from part (c).

You may find the below code snippit helpful:

```
import time

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

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

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

# compute the difference
diff = end_time - start_time
diff

print(diff)
print(diff2)

0.0003826618194580078
0.0001575946807861328
```

Part 3. Images [6 pt]

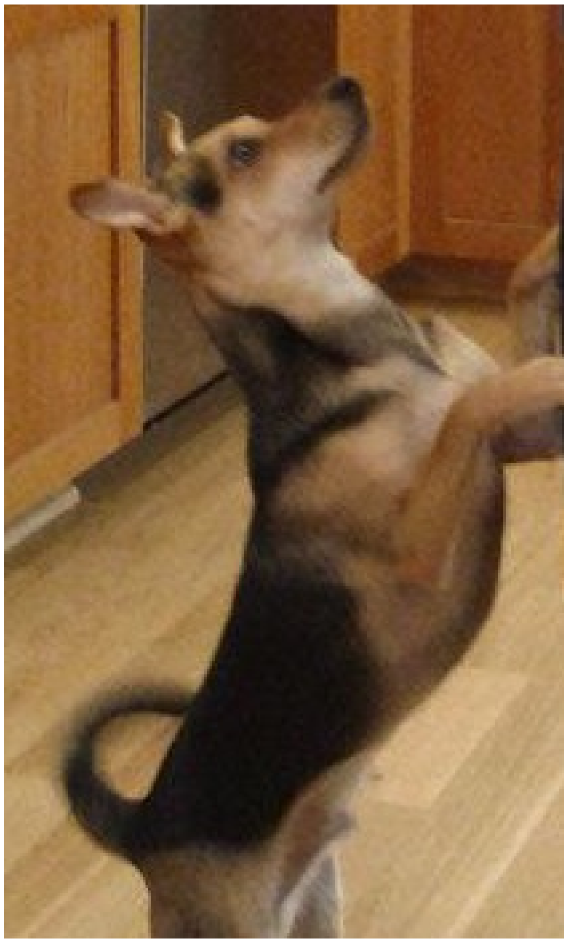
A picture or image can be represented as a NumPy array of "pixels", with dimensions H × W × C, where H is the height of the image, W is the width of the image, and C is the number of colour channels. Typically we will use an image with channels that give the the Red, Green, and Blue "level" of each pixel,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

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.

```
"""plt.imread function returned error with the provided URL"""
f = urllib.request.urlopen('https://drive.google.com/uc?export=view&id=1oaLVR2hr1_qzpKQ47i9rVUIklwbDcews')
img = Image.open(f)
```

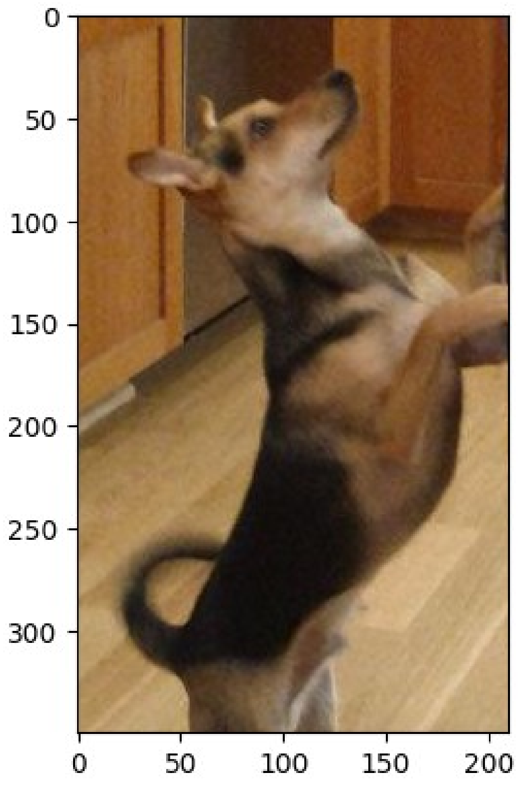
Part (b) -- 1pt

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 = np.array(img)
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x7886c292d2d0>
```



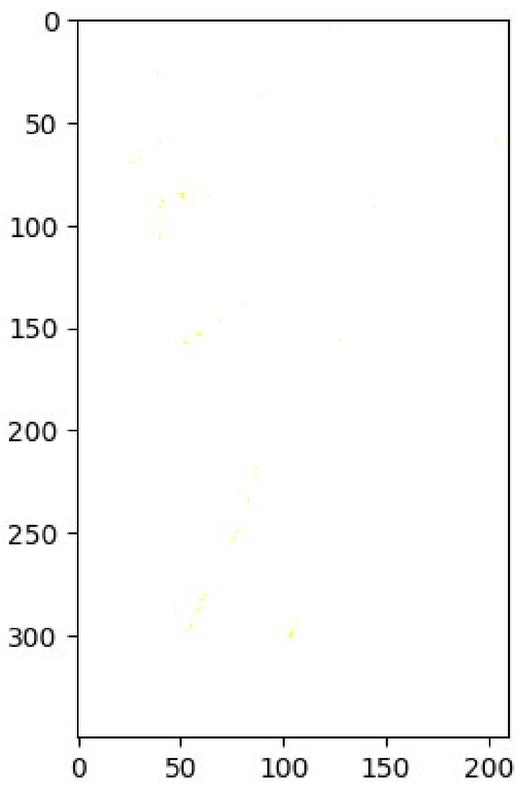
Part (c) -- 2pt

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

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

```
WARNING:matplotlib.image:Clipping input data to the valid range for
imshow with RGB data ([0..1] for floats or [0..255] for integers).

<matplotlib.image.AxesImage at 0x7886c080ee00>
```



Part (d) -- 2pt

Crop the **original** image (`img` variable) to a 130 x 150 image including Mochi's face. Discard the alpha colour channel (i.e. resulting `img_cropped` should **only have RGB channels**)

Display the image.

```
img_cropped = img[0:150, 20:150]
img_cropped = np.delete(img_cropped, 3, 2)

img_cropped
```

```
array([[[152,  91,  34],
        [148,  87,  30],
        [147,  86,  29],
        ...,
        [148,  87,  33],
        [154,  93,  39],
        [157,  96,  42]],

       [[155,  93,  36],
        [150,  88,  31],
        [146,  84,  27],
        ...,
        [150,  89,  35],
        [155,  94,  40],
        [158,  97,  43]],

       [[159,  94,  38],
        [153,  88,  32],
        [149,  84,  26],
        ...,
        [151,  90,  35],
        [157,  96,  41],
        [160,  99,  44]],

       ...,

       [[152,  96,  45],
        [148,  92,  41],
        [152,  96,  47],
        ...,
        [128,  98,  70],
        [126,  96,  68],
        [128,  97,  68]],

       [[155,  99,  48],
        [153,  97,  46],
        [155,  99,  50],
        ...,
        [131, 101,  73],
        [134, 103,  74],
        [137, 106,  77]],

       [[156, 100,  49],
        [153,  97,  46],
        [154,  98,  49],
        ...,
        [138, 108,  80],
        [137, 106,  77],
        [139, 108,  79]]], dtype=uint8)
```

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.

```python
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`.

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

img_torch
```

```
tensor([[[152,  91,  34],
         [148,  87,  30],
         [147,  86,  29],
         ...,
         [148,  87,  33],
         [154,  93,  39],
         [157,  96,  42]],

        [[155,  93,  36],
         [150,  88,  31],
         [146,  84,  27],
         ...,
         [150,  89,  35],
         [155,  94,  40],
         [158,  97,  43]],

        [[159,  94,  38],
         [153,  88,  32],
         [149,  84,  26],
         ...,
         [151,  90,  35],
         [157,  96,  41],
         [160,  99,  44]],

        ...,

        [[152,  96,  45],
         [148,  92,  41],
         [152,  96,  47],
         ...,
         [128,  98,  70],
         [126,  96,  68],
         [128,  97,  68]],

        [[155,  99,  48],
         [153,  97,  46],
         [155,  99,  50],
         ...,
         [131, 101,  73],
         [134, 103,  74],
         [137, 106,  77]],

        [[156, 100,  49],
         [153,  97,  46],
         [154,  98,  49],
         ...,
         [138, 108,  80],
         [137, 106,  77],
         [139, 108,  79]]], dtype=torch.uint8)
```

Part (b) -- 1pt

Use the method `<Tensor>.shape` to find the shape (dimension and size) of `img_torch`.

```
img_torch.shape
```

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

Part (c) -- 1pt

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

```
img_torch.numel()
```

```
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)
"""the function swaps axes. In this case, the rows and columns of
img_torch are swapped, i.e. element [i][j] becomes [j][i]"""
```

```
tensor([[[152, 155, 159,  ..., 152, 155, 156],
         [148, 150, 153,  ..., 148, 153, 153],
         [147, 146, 149,  ..., 152, 155, 154],
         ...,
         [148, 150, 151,  ..., 128, 131, 138],
         [154, 155, 157,  ..., 126, 134, 137],
         [157, 158, 160,  ..., 128, 137, 139]],

        [[ 91,  93,  94,  ...,  96,  99, 100],
         [ 87,  88,  88,  ...,  92,  97,  97],
         [ 86,  84,  84,  ...,  96,  99,  98],
         ...,
         [ 87,  89,  90,  ...,  98, 101, 108],
         [ 93,  94,  96,  ...,  96, 103, 106],
         [ 96,  97,  99,  ...,  97, 106, 108]],

        [[ 34,  36,  38,  ...,  45,  48,  49],
         [ 30,  31,  32,  ...,  41,  46,  46],
         [ 29,  27,  26,  ...,  47,  50,  49],
         ...,
         [ 33,  35,  35,  ...,  70,  73,  80],
         [ 39,  40,  41,  ...,  68,  74,  77],
         [ 42,  43,  44,  ...,  68,  77,  79]]], dtype=torch.uint8)
```

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)
"""the function add new axes. In this case, 0 axes are added,
therefore img_torch is not updated"""

tensor([[[[152,  91,  34],
          [148,  87,  30],
          [147,  86,  29],
          ...,
          [148,  87,  33],
          [154,  93,  39],
          [157,  96,  42]],

         [[155,  93,  36],
          [150,  88,  31],
          [146,  84,  27],
          ...,
          [150,  89,  35],
          [155,  94,  40],
          [158,  97,  43]],

         [[159,  94,  38],
          [153,  88,  32],
          [149,  84,  26],
          ...,
          [151,  90,  35],
          [157,  96,  41],
          [160,  99,  44]],

         ...,

         [[152,  96,  45],
          [148,  92,  41],
          [152,  96,  47],
          ...,
          [128,  98,  70],
          [126,  96,  68],
          [128,  97,  68]],

         [[155,  99,  48],
          [153,  97,  46],
          [155,  99,  50],
          ...,
          [131, 101,  73],
          [134, 103,  74],
          [137, 106,  77]],

         [[156, 100,  49],
          [153,  97,  46],
          [154,  98,  49],
          ...,
          [138, 108,  80],
          [137, 106,  77],
          [139, 108,  79]]]], dtype=torch.uint8)
```

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`.

```
red_val, _ = torch.max(img_torch[0], dim=0)
grn_val, _ = torch.max(img_torch[1], dim=0)
blu_val, _ = torch.max(img_torch[2], dim=0)

max_values = torch.tensor([red_val[0], grn_val[0], blu_val[0]])
max_values

tensor([177, 176, 176], dtype=torch.uint8)
```

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): #define a class of neural network called
"Pigeon"
    def __init__(self):
        super(Pigeon, self).__init__()
        self.layer1 = nn.Linear(28 * 28, 30) # input layer
        self.layer2 = nn.Linear(30, 1)

    def forward(self, img): # defines the forward pass
        flattened = img.view(-1, 28 * 28) # flattens the image
        activation1 = self.layer1(flattened) # passing the flattened
image thru the first layer
        activation1 = F.leaky_relu(activation1) # activation function
        activation2 = self.layer2(activation1)
        return activation2

pigeon = Pigeon() # create an instance of the Pigeon neural network

# load the data
mnist_data = datasets.MNIST('data', train=True, download=True) # load
in the MNIST data set
mnist_data = list(mnist_data) # convert the dataset into a list
mnist_train = mnist_data[:1000] # training set
mnist_val   = mnist_data[1000:2000] # validation set
img_to_tensor = transforms.ToTensor() # defines a transformation to
convert image to tensors


# simplified training code to train `pigeon` on the "small digit
recognition" task
criterion = nn.BCEWithLogitsLoss() # loss function (binary cross-
entropy with logits)
optimizer = optim.SGD(pigeon.parameters(), lr=0.005, momentum=0.9) #
defines the optimizer with learning rate and momentum

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: 0.037
Training Accuracy: 0.963
Test Error Rate: 0.079
Test Accuracy: 0.921
```

TESTING RESULTS:

ANN Variables	Parameter 1	Parameter 2	Parameter 3	Parameter 4	Training Accuracy	Testing Accuracy
Number of Iterations	100	1000	4000	8000	(0.780, 0.964, 0.966, 0.976)	(0.770, 0.921, 0.960, 0.965)
Learning Rate	0.001	0.005	0.01	0.1	(0.922, 0.964, 0.961, 0.688)	(0.887, 0.921, 0.918, 0.703)

ANN Variables	Parameter 1	Parameter 2	Parameter 3	Parameter 4	Training Accuracy	Testing Accuracy
Number of Layers	1	2	3	10	(0.312, 0.964, 0.955, 0.688)	(0.297, 0.921, 0.921, 0.703)
Number of Hidden Units at 2 Layers	10	30	100	1000	(0.953, 0.964, 0.970, 0.982)	(0.896, 0.921, 0.923, 0.935)
Activation Function	Sigmoid	ReLU	Leaky ReLU	Tanh Re	(0.927, 0.964, 0.963, 0.96)	(0.883, 0.921, 0.921, 0.906)

Note: Each variable is tested with the other variables being set to default (given values)

Note2: For number of iterations, the size of testing set is set to match the training set. For example, if the training size is 8000, the testing size is also 8000

Note3: The number of layers is changed without changing the number of hidden units (30). For example, 3 hidden layers each with 30 hidden units each summing to a total of 90. The activation function is set to ReLU for all layers

Observations:

- Increasing the number of iterations has a positive, but plateauing effect on both accuracies
- Increasing learning rate is similar to increasing the number of hidden layer, where there is an initial spike in both accuracies, followed by continued decline in both
- Increasing the number of hidden units has a positive and plateauing effect on both accuracies, similar to the number of iterations
- For activation functions, ReLU ranks best in terms of accuracy, followed by Leaky ReLU, Tanh, and Sigmoid

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?

```
"""
* Increased accuracy on training data is produced by increasing the
number of
iterations and number of hidden units as much as possible.

* The best training accuracy produced is 98.2% accuracy, achieved by
increasing
the number of hidden units to 1000. This is followed by a 97.6%
accuracy
```

```
achieved by increasing the number of training iterations to 8000
"""
```

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?

```
"""
* Similar to training data, increased accuracy on testing data is
produced by
increasing the number of iterations and number of hidden units as much
as possible.

* The best testing accuracy produced is 96.5% accuracy, achieved by
increasing
the number of training iterations to 8000. This is followed by a 93.5%
accuracy
achieved by increasing the number of hidden units to 1000
"""
```

Part (c) -- 4 pt

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

```
"""
The hyperparameters from part (b) should be used. This is because a
high accuracy
based on training data may be a result of overfitting, and may not
reflect the
"true" performance of the NN when fed with real world data. In
contrast, testing
is designed to simulate real world circumstances, and a high accuracy
based on
the testing dataset (even though lower than the training accuracy)
would better
reflect the capacity of the NN when fed unpredictable datasets.
"""
```