Lab 1. PyTorch and ANNs

Deadline: Thursday, January 23, 11:59pm.

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: Kevin Course

This lab is partially based on an assignment developed by Prof. Jonathan Rose and Harris Chan.

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.

With Colab, you can export a PDF file using the menu option `File -> Print` and save as PDF file.

Colab Link

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

Colab Link: https://colab.research.google.com/drive/15AT4Qz4YkrfIgdpM3gDd6gtK8mVWFgO9

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`.

```
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

    sum = 0
    for i in range(1, n + 1):
      sum += i**3
    return sum

#testing for correctness
print(sum_of_cubes(3))
print(sum_of_cubes(1))
print(sum_of_cubes(-10))
```

```
36
1
Invalid input
-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

```
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.
```

```
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]
    """
    length_of_words = []

    #in the case of an empty string, assume that there is only 1 word and it has length 0
    if len(sentence) == 0:
      length_of_words.append(0)
      return length_of_words

    words = sentence.split()

    for word in words:
      length_of_words.append(len(word))

    return length_of_words

#testing for correctness
```

```
#testing for correctness
oneTest = word_lengths("welcome to APS360!")
print(oneTest)

anotherTest = word_lengths("machine learning is so cool")
print(anotherTest)

oneMoreTest = word_lengths("")
print(oneMoreTest)

```

```
[7, 2, 7]
[7, 8, 2, 2, 4]
[0]
```

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
    """
    #in the case of an empty string, assume that all words have the same length and return True

    length_of_words = word_lengths(sentence)
    first_length = length_of_words[0]

    for length in length_of_words:
      if length != first_length:
        return False
    return True

#function to print out value of bool
def print_bool(TF):
  if TF:
    print("True")
  else:
    print("False")

#testing for correctness
firstTest = all_same_length("all same length")
print_bool(firstTest)

secondTest = all_same_length("hello world")
print_bool(secondTest)

thirdTest = all_same_length("")
print_bool(thirdTest)
```

```
False
True
True
```

Part 2. NumPy Exercises [5 pt]

In this part of the assignment, you'll be manipulating arrays usign NumPy. Normally, we use the shorter name `np` to represent the package `numpy`.

```
import numpy as np
```

Part (a) -- 1pt

The below variables `matrix` and `vector` are numpy arrays. Explain what you think `<NumpyArray>.size` and `<NumpyArray>.shape` represent.

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

```
matrix.size
# determines the number of elements in the array
```

```
12
```

```
matrix.shape
# determines the number of rows and columns, output is represented as a tuple (number of rows, number of columns)
```

```
(3, 4)
```

```
vector.size
# determines the number of elements in the vector
```

```
4
```

```
vector.shape
# determines the number of rows in a row vector or the number of columns in a column vector
# but vectors are one-dimensional in NumPy
```

```
(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

```
matrix_dim = matrix.shape #(3,4)
vector_dim = vector.shape #(4,)


output = np.array([0., 0., 0.]) # 3x4 array * 4x1 vector = 3x1 array
sum = 0

if vector_dim[0] == matrix_dim[1]:
  for row in range (0, matrix_dim[0]):
    for col in range(0, matrix_dim[1]):
      output[row] += vector[col] * matrix[row][col]

```

```
output
```

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

Part (c) -- 1pt

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

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

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

```
output2
```

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

Part (d) -- 1pt

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

```
if np.array_equal(output, output2):
    print("Outputs match")
else:
    print("Outputs don't match")
```

```
Outputs match
```

Part (e) -- 1pt

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

You may find the below code snippit helpful:

```
import time
start_time_my_function = time.time()

matrix_dim = matrix.shape #(3,4)
vector_dim = vector.shape #(4,)

output = np.array([0., 0., 0.])
sum = 0

if vector_dim[0] == matrix_dim[1]:
  for row in range (0, matrix_dim[0]):
    for col in range(0, matrix_dim[1]):
      output[row] += vector[col] * matrix[row][col]

end_time_my_function = time.time()

print("Runtime of function in part (b): {}".format(end_time_my_function - start_time_my_function))

start_time_numpy = time.time()
outputX = np.dot(matrix, vector)
end_time_numpy = time.time()

print("Runtime of NumPy function .dot: {}".format(end_time_numpy - start_time_numpy))

# compute the difference
#diff = abs((end_time_my_function - start_time_my_function) - (end_time_numpy - start_time_numpy))
#print("np.dot is {} seconds faster than the function in part (b)".format(diff))
```

```
Runtime of function in part (b): 0.0002918243408203125
Runtime of NumPy function .dot: 8.58306884765625e-05
```

Part 3. Images [6 pt]

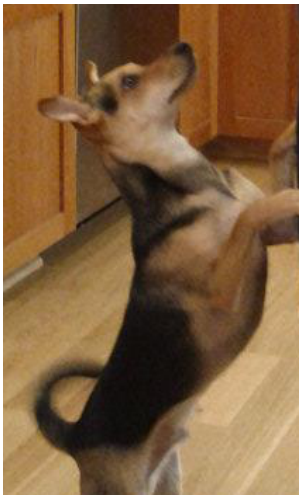
A picture or image can be represented as a NumPy array of “pixels”, with dimensions H × W × C, where H is the height of the image, W is the width of the image, and C is the number of colour channels. Typically we will use an image with channels that give the the Red, Green, and Blue “level” of each pixel,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

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

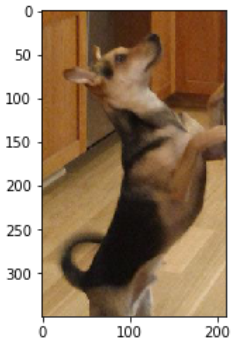
Part (b) -- 1pt

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.

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

```
<matplotlib.image.AxesImage at 0x7fc971833fd0>
```

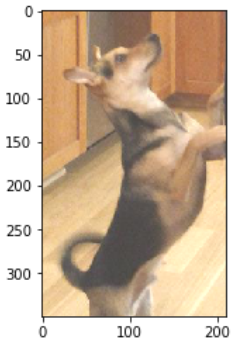


Part (c) -- 2pt

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`.

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

```
<matplotlib.image.AxesImage at 0x7fc96ff469b0>
```



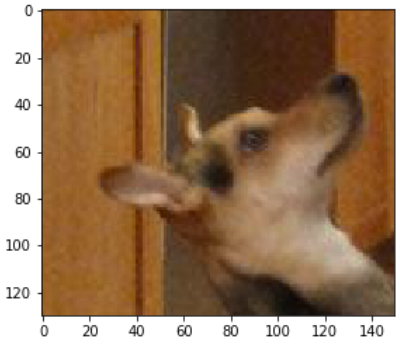
Part (d) -- 2pt

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.

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

```
<matplotlib.image.AxesImage at 0x7fc96ff2c5c0>
```



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.

```
1 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`.

```
1 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`.

```
1 img_torch.shape
```

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

Part (c) -- 1pt

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

```
1 img_torch.shape[0]*img_torch.shape[1]*img_torch.shape[2]
```

```
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.

```
1 #img_torch.transpose(0,2) transposes the rows and columns of img_torch by swapping the dimensions 0 and 2
2 #it returns a tensor that is the transpose of img_torch
3 #img_torch is not updated
4
5 print("dimensions of img_torch before transpose: {}".format(img_torch.shape))
6 print("dimensions of operation transposed img_torch: {}".format(img_torch.transpose(0,2).shape))
7 print("dimensions of img_torch after transpose: {}".format(img_torch.shape))
```

```
dimensions of img_torch before transpose: torch.Size([130, 150, 3])
dimensions of operation transposed img_torch: torch.Size([3, 150, 130])
dimensions of img_torch after transpose: torch.Size([130, 150, 3])
```

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.

```
1 #img_torch.unsqueeze(0) will return a new tensor with a dimension inserted where specified in the function call
2 #it returns a new tensor with an additional dimension
3 #img_torch is not updated
4
5 print("dimensions of img_torch before unsqueeze: {}".format(img_torch.shape))
6 print("dimensions of operation img_torch.unsqueeze: {}".format(img_torch.unsqueeze(0).shape))
7 print("dimensions of img_torch after unsqueeze: {}".format(img_torch.shape))
8
```

```
dimensions of img_torch before unsqueeze: torch.Size([130, 150, 3])
dimensions of operation img_torch.unsqueeze: torch.Size([1, 130, 150, 3])
dimensions of img_torch after unsqueeze: torch.Size([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`.

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

```
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

```
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) # 28*28 node -> 30 node

        # change the number of hidden units
        #self.layer1 = nn.Linear(28 * 28, 3)
        #self.layer2 = nn.Linear(3, 1)

        # add another layer
        #self.layer3 = nn.Linear(30,30) #intermediate 30 node input -> 30 node output layer
        #self.layer4 = nn.Linear(30,30)
        #self.layer5 = nn.Linear(30,30)
        #self.layer6 = nn.Linear(30,30)
        #self.layer7 = nn.Linear(30,30)
        #self.layer8 = nn.Linear(30,30)

        self.layer2 = nn.Linear(30, 1) # 30 node -> 1 node

    def forward(self, img):
        flattened = img.view(-1, 28 * 28)
        activation1 = self.layer1(flattened)
        activation1 = F.relu(activation1)

        #change the activation function
        #activation1 = F.sigmoid(activation1)
        #activation1 = F.tanh(activation1)

        # add another layer
        #activation3 = self.layer3(activation1) # after adding intermediate layer between input and hidden layer
        #activation4 = self.layer4(activation1)
        #activation5 = self.layer5(activation1)
        #activation6 = self.layer6(activation1)
        #activation7 = self.layer7(activation1)
        #activation8 = self.layer8(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)

# change learning rate
#optimizer = optim.SGD(pigeon.parameters(), lr=0.01, momentum=0.9)

# change the number of training iterations
#iterations = 2

#for iter in range(iterations) :
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?

```
"""
A list of hyperparameters changed and the resulting training and testing accuracies is included in the Appendix after part (c)

The highest accuracy in the training data occurred in the test cases when:
- the number of training iterations increased from 1 to 50
    - Training Accuracy: 1.0
- the number of hidden units increased from 30 to 60 and 300
    - Training Accuracy: 0.969 and Training Accuracy: 0.977, respectively
- the number of layers increased from 2 to 5
    - Training Accuracy: 0.965

Increasing the above three hyperparameters allows the NN to become more familiar with the training dataset because it has more
passes over the data, allowing more opportunities to find patterns and other trends in the data. However, allowing many passes
over the training data will also cause the NN to overfit to the noise and other irregularities present in the training dataset,
which may not be present in the testing dataset or other unseen datasets. Overfitting may be why the Training Accuracy decreased
when the number of layers used increased from 5 to 8 (0.965 -> 0.954)
"""
```

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?

```
"""
The highest accuracy in the testing data occurred in the test cases when:
- the number of training iterations is 2
    - Test Accuracy: 0.943
- the number of hidden units is 300
    - Test Accuracy: 0.929
- the number of layers is 5
    - Test Accuracy: 0.9299999999999999

If increasing a NN hyperparameter increased the accuracy of the NN on the training data, then it is reasonable to expect that
such changes would also result in improved accuracy on the testing data. This is the case when the number of hidden units is
set to 300 and the number of layers is set to 5.

```

```
However, it is interesting to note that the test accuracy is greatest when the number of training iterations is set to 2, not
when it is set to 50 as in part (a). This implies that the NN experienced overfitting when the number of iterations was 50, and
it began adapting to the noise that was present in the training dataset, and trying to fit this noise onto the testing dataset.
Because the testing dataset lacked the noise present in the training dataset, it was difficult for the NN to apply this pattern
recognition to something that doesn't exist.
"""
```

Part (c) -- 4 pt

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

```
"""
To achieve the highest accuracy on an unseen data set, we should:
- set the number of training iterations to 2 (part (b))
- set the number of hidden units to 300 (part (a))
- set the number of layers to 5 (part (a))
- keep all other hyperparameters the same as in the original model

In choosing the values of these hyperparameters, it is important to balance too few passes over the data against too many passes.
With too few passes, the NN does not have an opportunity to detect patterns, learn of their frequencies, and predict their
occurrence in future samples. However with too many passes, the NN will become overfitted to the training data, and it will
incorporate irregularities and noise present in the training data into its predictions on future samples where the same noise
may not exist.

It is also important to balance the accuracy of the NN and its performance. Although the test accuracy is higher when the number
of hidden units is set to 300, the accuracy only improved by 0.008 when compared to the accuracy of the original model that used
30 hidden units. For an accuracy improvement of less than 1%, it can be argued that the slight improvement in accuracy is not
a good justification for increasing the runtime.
"""
```

Appendix: Changing hyperparameters for Part 5

```
"""
I changed one parameter at a time
****************************************
            Original
number of training iterations: 1
number of hidden units: 30
numbers of layers: 2
types of activation functions: ReLU
learning rate: 0.005
number of training points: 1000
number of validation points: 1000

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

****************************************
            Changed number of training iterations
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: 0.058
Test Accuracy: 0.942

number of training iterations: 50

Training Error Rate: 0.0
Training Accuracy: 1.0
Test Error Rate: 0.059
```

```
Test Accuracy: 0.9410000000000001

****************************************
            Changed number of hidden units
number of hidden units: 3

Training Error Rate: 0.103
Training Accuracy: 0.897
Test Error Rate: 0.143
Test Accuracy: 0.857

number of hidden units: 60

Training Error Rate: 0.031
Training Accuracy: 0.969
Test Error Rate: 0.08
Test Accuracy: 0.92

number of hidden units: 300

Training Error Rate: 0.023
Training Accuracy: 0.977
Test Error Rate: 0.071
Test Accuracy: 0.929

****************************************
            Changed number of layers
numbers of layers: 3

Training Error Rate: 0.043
Training Accuracy: 0.957
Test Error Rate: 0.081
Test Accuracy: 0.919

numbers of layers: 5

Training Error Rate: 0.035
Training Accuracy: 0.965
Test Error Rate: 0.07
Test Accuracy: 0.9299999999999999

numbers of layers: 8

Training Error Rate: 0.046
Training Accuracy: 0.954
Test Error Rate: 0.086
Test Accuracy: 0.914

****************************************
            Changed activation function
type of activation function: sigmoid

Training Error Rate: 0.073
Training Accuracy: 0.927
Test Error Rate: 0.117
Test Accuracy: 0.883

type of activation function: tanh

Training Error Rate: 0.04
Training Accuracy: 0.96
Test Error Rate: 0.094
Test Accuracy: 0.906

type of activation function: softmax

Training Error Rate: 0.312
Training Accuracy: 0.688
Test Error Rate: 0.297
Test Accuracy: 0.7030000000000001

type of activation function: softsign

Training Error Rate: 0.053
```

```
Training Error Rate: 0.053
Training Accuracy: 0.947
Test Error Rate: 0.108
Test Accuracy: 0.892

****************************************
          Changed learning rate
learning rate: 0.00001

Training Error Rate: 0.321
Training Accuracy: 0.679
Test Error Rate: 0.311
Test Accuracy: 0.6890000000000001

learning rate: 0.0001

Training Error Rate: 0.312
Training Accuracy: 0.688
Test Error Rate: 0.297
Test Accuracy: 0.7030000000000001

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

****************************************
          Changed number of points in training and validation datasets
number of training points: 500
number of validation points: 1500

Training Error Rate: 0.06
Training Accuracy: 0.94
Test Error Rate: 0.11266666666666666
Test Accuracy: 0.8873333333333333

number of training points: 1500
number of validation points: 500

Training Error Rate: 0.05
Training Accuracy: 0.95
Test Error Rate: 0.074
Test Accuracy: 0.926

number of training points: 1800
number of validation points: 200

Training Error Rate: 0.055
Training Accuracy: 0.945
Test Error Rate: 0.085
Test Accuracy: 0.915
"""
```