

## ▼ Lab 1. PyTorch and ANNs

**Deadline:** Thursday, Oct 1, 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:** Justin Beland

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 plagiarism 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 configurations

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.  
**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):

    if (n > 0) and (type(n) == int) :
        sum = 0
        for i in range(0, n, 1):
            sum += i*i*i
        return sum

    else:
        print("invalid input")
        return -1

sum_of_cubes(3)
sum_of_cubes(4)
```

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 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>

```
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):
    if type(sentence) != str:
        return -1
    array = []
    sentence_broken = sentence.split()
    for i in sentence_broken:
        array += [len(i)]
    return array
```

### ▼ 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):
    length = set(word_lengths(sentence))
    if len(length) > 1:
        return False
    else:
        return True
```

### ▼ Part 2. NumPy Exercises [5 pt]

In this part of the assignment, you'll be manipulating arrays using 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
# returns the number of elements in the matrix (i.e. there are 12 elements in matrix)

12

matrix.shape
# returns the dimensions of the matrix (i.e matrix is 4x3 matrix)

(3, 4)

vector.size
# returns the number of elements in vector

4

vector.shape
# returns the dimension of vector; only (4,) because it's one-dimensional

(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 = None
def matrix_multiply(matrix, vector):
    array = []
    x = 0
    for i in range(0, matrix.shape[0]):
        for j in range(0, matrix.shape[1]):
            x = x + matrix[i][j]*vector[j]
        array += [x]
        x = 0
    ret = np.array(array)
    return ret

output = matrix_multiply(matrix, vector)

```

### ▼ 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 = None
def matrix_multiply2(matrix, vector):
    output2 = np.dot(matrix, vector)
    return output2

output2 = matrix_multiply2(matrix, vector)

```

### ▼ Part (d) -- 1pt

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

```

def test_if_same(output1, output2):
    comparison = output1 == output2
    equal = comparison.all()
    return equal

test_if_same(output, output2)

True

```

### ▼ Part (e) -- 1pt

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

You may find the below code snippet helpful:

```
import time

# 1 -- For Loop

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

# place code to run here
matrix_multiply(matrix, vector)

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

# compute the difference
diff = end_time - start_time

# 2 -- NumPy

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

# place code to run here
matrix_multiply2(matrix, vector)

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

# compute the difference
diff2 = end_time2 - start_time2

if (diff2 < diff):
    print("True -- NumPy is faster")
else:
    print("False -- For loop is faster")

    True -- NumPy is faster
```

### ▼ Part 3. Images [6 pt]

A picture or image can be represented as a NumPy array of “pixels”, with dimensions  $H \times W \times C$ , 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

```
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.

```
img = None
img = plt.imread("https://drive.google.com/uc?export=view&id=1oaLVR2hr1_qzpKQ47i9rVUIklwbDcews")
```

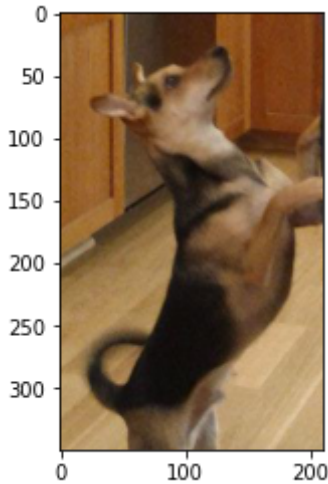
### ▼ Part (b) -- 1pt

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 0x7fdce910ae80>



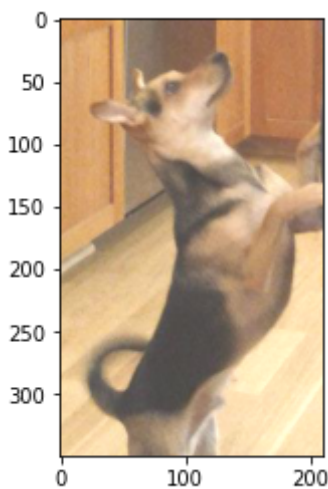
### ▼ Part (c) -- 2pt

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 = None
img_add = img + 0.25
np.clip(img_add, 0, 1)
plt.imshow(img_add)
```

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 0x7fdce906b828>



### ▼ Part (d) -- 2pt

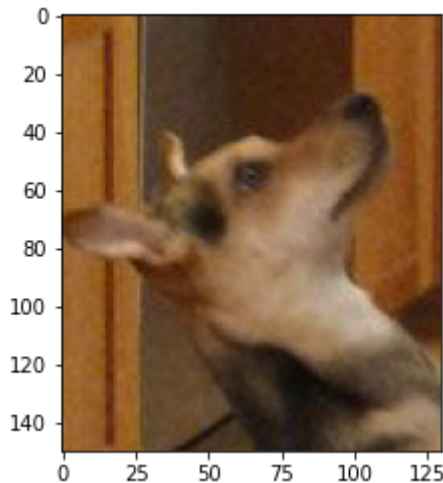


Crop the **original** image ( `img` variable) to a 130 x 150 image including Mochi's face. Discard the alpha colour channel (i.e. resulting `img_cropped` should **only have RGB channels**)

Display the image.

```
img_cropped = img[0:150,25:155]
plt.imshow(img_cropped)
```

<matplotlib.image.AxesImage at 0x7fdce9046b38>



#### ▼ 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 = None
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`.

```
img_torch.shape

torch.Size([150, 130, 4])
```

### ▼ Part (c) -- 1pt

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

```
# there are 3 floating point numbers
```

### ▼ 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.shape
img_torch.transpose(0,2).shape
img_torch.shape

# the original is not updates as we call img_torch.shape twice and the results are the

torch.Size([150, 130, 4])
```

### ▼ 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).shape
img_torch.shape

# unsqueeze adds a new dimension infront of the index indicated. in this case it is ac

torch.Size([1, 150, 130, 4])
torch.Size([150, 130, 4])
```

## ▼ 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`.

```
a = torch.max(img_torch[:, :, 0])
b = torch.max(img_torch[:, :, 1])
c = torch.max(img_torch[:, :, 2])
max_vals = [a, b, c]
max_vals

[tensor(0.8941), tensor(0.7882), tensor(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)
        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=0.004, momentum=0.9)
optimizer = torch.optim.Adam(pigeon.parameters(), lr = 0.005) #<----- ADAM OPTIMIZER

iters=20
for i in range(iters):
    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: 0.0

```
Training Accuracy: 1.0  
Test Error Rate: 0.052  
Test Accuracy: 0.948
```

## 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?**

The most significant changes occurred with the number of iterations. The following results were obtained from the training accuracy:

- iterations = 03 ---> accuracy = 0.986
- iterations = 05 ---> accuracy = 0.989
- iterations = 10 ---> accuracy = 0.999
- iterations = 15 ---> accuracy = 0.999

We see from the above that with the increasing iterations the training accuracy increases. It appears to converge on 0.999 with 10 iterations, so we conclude that this is the optimal iteration number.

Testing with other parameter changes, namely the learning rate and type of activation functions did not show to change the results much.

Thus, increasing the iterations to 10 delivered the best results with an accuracy of 0.999

## 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 the above, the best results occurred with increasing the number of iterations. With 10 iterations the testing accuracy achieved was 0.948.

Trying to change with the learning rate, both increasing and decreasing the current rate of 0.005 decreased the testing results. Also after trying with the Adam optimizer, there was still no improvement with the testing results.

## Part (c) -- 4 pt

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

The results from (b) should definitely be used over (a). It is important to make sure to not overfit the data. This could be a concern in (a) as the testing accuracy was nearing 1.

More importance should be placed on the testing accuracy as it is better if the model works on previously unseen data over the training data.

