Assignment 1. Python and libraries

Deadline: May 15, 9pm.

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.

TAs: Andrew Jung

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

Welcome to the first lab of APS360! This lab is a warm up to get you used to the 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. Setup and use Google Colab.
2. Write basic, object-oriented Python code.
3. Be able to perform matrix operations using `numpy`.
4. Be able to plot using `matplotlib`.
5. Be able to load, process, and visualize image data.
6. Be able to perform basic PyTorch tensor operations.

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/1dNeiiyv59KSnu4L4NtjDhjEMYyCGX72e?usp=sharing

Part 0. Environment Setup; Readings

Please refer to Colab instructions https://colab.research.google.com/drive/1YKHHLSIG-B9Ez2-zf-YFxXTVgfC_Aqtt

If you want to use Jupyter Notebook locally, please refer to https://www.cs.toronto.edu/~lczhang/aps360_20191/files/install.pdf

Part 1. Python Basics [9 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) -- 3pt

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 or int(n) != n:
      return -1
    else:
      return (n*(n + 1)/ 2) ** 2

print(sum_of_cubes(2))
print(sum_of_cubes(-10))
print(sum_of_cubes(10.5))
print(sum_of_cubes(-10.5))
```

```
9.0
-1
-1
-1
```

Part (b) -- 3pt

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)
```

```
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]
    """
    out = []
    for word in sentence.split():
      out.append(len(word))

    return out
```

```
print(word_lengths("welcome to APS360!"),
word_lengths("machine learning is so cool"))
```

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

Part (c) -- 3pt

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
    """
    length = -1
    for word in sentence.split():
      if length == -1:
        length = len(word)
      elif length != len(word):
        return False
    return True

print(all_same_length("all same length"))
print(all_same_length("hello world"))
print(all_same_length(""))
print(all_same_length("o   h   w"))
print(all_same_length("hello      w"))
print(all_same_length("world"))
```

```
False
True
True
True
False
True
```

Part 2. NumPy Exercises [11 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) -- 2pt

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 = row * col
```

```
12
```

```
matrix.shape # Row and Columns respectively
```

```
(3, 4)
```

```
vector.size # Number of elements
```

```
4
```

```
vector.shape # Length of the array (1 - dimensional)
```

```
(4,)
```

Part (c) -- 3pt

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

```
# When we multiply a 3 * 4 matrix with a 4 * 1 column vector we get a 3 * 1 column vector

output = np.zeros(3)

for row_index in range(0, matrix.shape[0]):
  sum = 0
  for index in range(0, len(matrix[row_index])):
    sum += matrix[row_index][index] * vector[index]
  output[row_index] = sum
```

```
output
```

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

Part (d) -- 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 (e) -- 2pt

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

```
output == output2
```

```
array([ True,  True,  True])
```

Part (f) -- 3pt

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

You may find the below code snippit helpful:

For loops time consumption

```
import time

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

# place code to run here
for i in range(10000):
    for row_index in range(0, matrix.shape[0]):
```

```
      sum = 0
      for index in range(0, len(matrix[row_index])):
        sum += matrix[row_index][index] * vector[index]
      output[row_index] = sum

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

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

```
0.13895010948181152
```

np.dot time consumption

```
import time

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

# place code to run here
for i in range(10000):
    output2 = np.dot(matrix, vector)

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

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

```
0.020548582077026367
```

```
# np.dot is approximately 6 times faster than our 'for' loop (sad that I produce slow code ha
```

Part 3. Callable Objects [11 pt]

A *callable object* is any object that can be called like a function. In Python, any object whose class has a `__call__` method will be callable. For example, we can define an `AddBias` class that is initialized with a value `val`. When the object of the Adder class is called with `input`, it will return the sum of `val` and `input`:

```
class AddBias(object):       # this is a new class AddBias, which inherits from the class `ob
    def __init__(self, val): # this is the object constructor
```

```
        self.val = val
    def __call__(self, input):
        return self.val + input # `self` is like `this` in many languages


add4 = AddBias(4)
add4(3)
```

```
     7
```

```
# AddBias works with numpy arrays as well

add1 = AddBias(1)
add1(np.array([3,4,5]))
```

```
     array([4, 5, 6])
```

Part (a) -- 2pt

Create a callable object class `ElementwiseMultiply` that is initialized with `weight`, which is a numpy array (with 1-dimension). When called on `input` of **the same shape** as `weight`, the object will output an elementwise product of `input` and `weight`. For example, the 1st element in the output will be a product of the first element of `input` and first element of `weight`. If the `input` and `weight` have different shape, do not return anything.

```
class ElementwiseMultiply(object):
  def __init__(self, weight):
    self.weight = np.array(weight)
  def __call__(self, input):
    if self.weight.size == np.array(input).size:
      return np.multiply(self.weight, input)

obj1 = ElementwiseMultiply([1, 2, 3])
obj1([6, 5, 6])
```

```
     array([ 6, 10, 18])
```

Part (b) -- 4pt

Create a callable object class `LeakyRelu` that is initialized with `alpha`, which is a scalar value. When called on input `x`, which may be a NumPy array, the object will output:

- x if $x \geq 0$
- αx if $x < 0$

For example,

```
>>> leaky_relu = LeakyRelu(0.1)
>>> leaky_relu(1)
1
>>> leaky_relu(-1)
-0.1
>>> x = np.array([1, -1])
>>> leaky_relu(x)
np.array([1, -0.1])
```

To obtain full marks, do **not** use any for-loops to implement this class.

```
class LeakyRelu(object):
  def __init__(self, alpha):
    self.alpha = alpha
  def __call__(self, x):
    x = np.array(x)
    out = np.where(x < 0, x*self.alpha, x)
    if out.size == 1:
      return float(out)
    else:
      return out

leaky_relu = LeakyRelu(0.1)
leaky_relu([13,  13, -12, -37])
```

```
array([13. , 13. , -1.2, -3.7])
```

Part (c) -- 4pt

Create a callable object class `Compose` that is initialized with `layers`, which is a list of callable objects each taking in one argument when called. For example, `layers` can be something like `[add1, add4]` that we created above. Each `add1` and `add4` take in one argument. When `Compose` object is called on **a single argument**, the object will output a composition of object calls in `layers`, in the order given in `layers` (e.g. `add1` will be called first and then `add4` will be called after using the result from `add1` call)

```
class Compose(object):
  def __init__(self, layers):
    self.layers = layers
  def __call__(self, input):
    if len(self.layers) == 1:
      return self.layers(input)
    temp = self.layers[0](input)
```

```
    print(temp)
    for elem in range(1, len(self.layers)):
      temp = self.layers[elem](temp)
      print(temp)
    return temp
```

Part (d) -- 1pt

Run the below code and include the output in your report.

```
weight_1 = np.array([1, 2, 3, 4.])
weight_2 = np.array([-1, -2, -3, -4.])
bias_1 = 3.0
bias_2 = -2.0
alpha = 0.1

elem_mult_1 = ElementwiseMultiply(weight_1)
add_bias_1 = AddBias(bias_1)
leaky_relu = LeakyRelu(alpha)
elem_mult_2 = ElementwiseMultiply(weight_2)
add_bias_2 = AddBias(bias_2)
layers = Compose([elem_mult_1,
                  add_bias_1,
                  leaky_relu,
                  elem_mult_2,
                  add_bias_2,
                  leaky_relu])

input = np.array([10, 5, -5, -10.])
print("Input: ", input)

output = layers(input)
print("Output:", output)
```

```
Input:  [ 10.   5.  -5. -10.]
[ 10.  10. -15. -40.]
[ 13.  13. -12. -37.]
[13.  13.  -1.2 -3.7]
[-13.  -26.    3.6  14.8]
[-15.  -28.    1.6  12.8]
[-1.5 -2.8  1.6 12.8]
Output: [-1.5 -2.8  1.6 12.8]
```

Part 4. Images [7 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

```
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_qzpKQ47i9rVUIklwbDcew
```

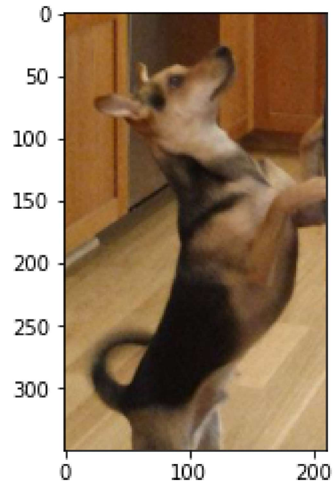
Part (b) -- 1pt

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 0x7f2514fdd6d0>
```

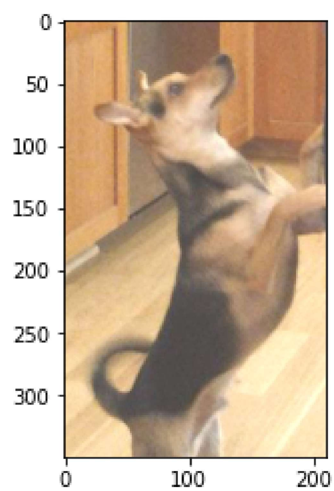


Part (c) -- 2pt

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

```
<matplotlib.image.AxesImage at 0x7f2513533550>
```



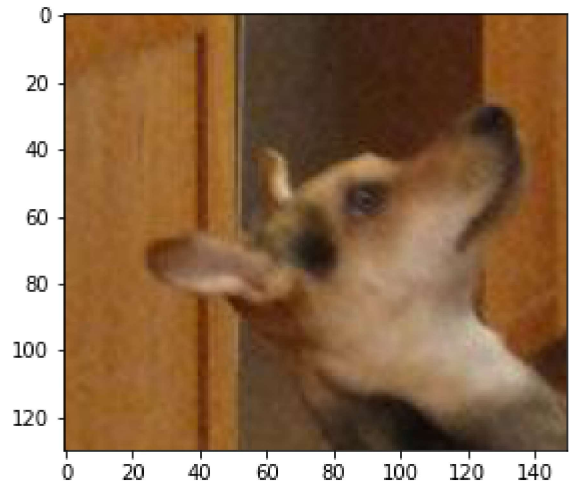
Part (d) -- 3pt

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 = np.array(img)[:130, :150, :3]
plt.imshow(img_cropped)
print(img_cropped.shape)
```

```
(130, 150, 3)
```



Part 5. Basics of PyTorch [12 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_cropped)
```

```
tensor([[[0.5882, 0.3725, 0.1490],
```

```
 [0.5765, 0.3608, 0.1373],
 [0.5569, 0.3412, 0.1176],
 ...,
 [0.5804, 0.3412, 0.1294],
 [0.6039, 0.3647, 0.1529],
 [0.6157, 0.3765, 0.1647]],

[[0.5412, 0.3216, 0.0902],
 [0.5647, 0.3451, 0.1137],
 [0.5961, 0.3765, 0.1451],
 ...,
 [0.5882, 0.3490, 0.1373],
 [0.6078, 0.3686, 0.1569],
 [0.6196, 0.3804, 0.1686]],

[[0.6157, 0.3765, 0.1529],
 [0.6196, 0.3843, 0.1490],
 [0.6196, 0.3843, 0.1412],
 ...,
 [0.5922, 0.3529, 0.1373],
 [0.6157, 0.3765, 0.1608],
 [0.6275, 0.3882, 0.1725]],

...,

[[0.6039, 0.3882, 0.1686],
 [0.6078, 0.3922, 0.1686],
 [0.6118, 0.3961, 0.1725],
 ...,
 [0.3804, 0.3098, 0.2157],
 [0.3765, 0.3059, 0.2118],
 [0.3765, 0.3098, 0.2078]],

[[0.5882, 0.3725, 0.1529],
 [0.6078, 0.3922, 0.1725],
 [0.6196, 0.4039, 0.1804],
 ...,
 [0.3882, 0.3176, 0.2314],
 [0.3804, 0.3098, 0.2157],
 [0.3804, 0.3098, 0.2157]],

[[0.5804, 0.3647, 0.1451],
 [0.6039, 0.3882, 0.1686],
 [0.6235, 0.4078, 0.1882],
 ...,
 [0.4196, 0.3373, 0.2549],
 [0.4039, 0.3216, 0.2392],
 [0.3961, 0.3137, 0.2314]]])
```

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`?

```
130*150*3
```

```
58500
```

Part (d) -- 3 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)
print(img_torch.transpose(0,2).shape)
print(img_torch.shape)
# It converts a x b x c into c x b x a
# it does not alter the original variable img_torch
```

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

Part (e) -- 3 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) # It adds an additional dimension to the tensor
img_torch.shape # It does not alter the original variable
```

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

Part (f) -- 3 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.max(img_torch[:, :, 0]), torch.max(img_torch[:, :, 1]), torch.max(img_torch[:, :, 2])]
```

```
[tensor(0.8941), tensor(0.7882), tensor(0.6745)]
```

The end!