Lab 1. PyTorch and ANNs

Deadline: Monday, Jan 25, 5:00pm.

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, Ali Khodadadi

This lab is based on assignments developed by Jonathan Rose, Harris Chan, Lisa Zhang, and Sinisa Colic.

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/1ChhvQcIbRV9w4R2-CeWnLpKWaSs97knP?usp=sharing

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`.

In [158…

```
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 (not (type(n) is int and n > 0)):
      print("Invalid input")
      return -1

    return sum(pow(i, 3) for i in range(1, n + 1))

print(sum_of_cubes(3))
print(sum_of_cubes(1))
```

```
36
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

In [ ]:

```
help(str.split)
```

In [159…

```
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]
    """

    return [ len(word) for word in sentence.split() ]

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

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

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.

In [161…

```
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
    """

    lengths = word_lengths(sentence)

    for length in lengths:
      if length != lengths[0]:
        return False

    return True
```

```
print(all_same_length("all same length"))
print(all_same_length("hello world"))
```

```
False
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` .

In [106…

```
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.

ANSWER

`<NumpyArray>.size` : Number of elements in array

`<NumpyArray>.shape` : Dimensions of array in form (# of rows, # of cols)

In [107…

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

In [ ]:

```
matrix.size
```

Out[ ]: 12

In [ ]:

```
matrix.shape
```

Out[ ]: (3, 4)

In [ ]:

```
vector.size
```

Out[ ]: 4

In [ ]:

```
vector.shape
```

Out[ ]: (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

In [190…

```
def matrix_multiply(left, right):
  dim_row = left.shape[0]
  dim_col = 1 if len(right.shape) == 1 else right.shape[1]

  output = [ 0 for i in range(dim_row)]

  # in left matrix, go row by row
  for row in range(dim_row):
    for col in range(dim_col): # num of cols in right matrix
      for row_right in range(right.shape[0]): # rows in right matrix
        output[row] += left[row][row_right] * right[row_right]

  return np.array(output)

output = matrix_multiply(matrix, vector)
```

In [ ]:

```
output
```

Out[ ]:

```
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.

In [204…

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

In [205…

```
output2
```

Out[205…

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

Part (d) -- 1pt

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

In [213…

```
print(output.shape)
print(output2.shape)

print(f"Outputs match: {(output == output2).all()}")
```

```
(3,)
(3,)
Outputs match: 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:

In [218…

```python
import time

def get_runtime(func):
  start_time = time.time()
  func()                          # run function
  end_time = time.time()
  return end_time - start_time   # elapsed time

def my_version():
  matrix_multiply(matrix, vector)

def numpy_version():
  np.dot(matrix, vector)

my_time = get_runtime(my_version)
numpty_time = get_runtime(numpy_version)

f"{'Numpy' if numpty_time < my_time else 'Mine'} is faster."
```

Out[218…

```
'Numpy is faster.'
```

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.

In [7]:

```python
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.

In [9]:

```
img = plt.imread("https://drive.google.com/uc?export=view&id=1oaLVR2hr1_qzpKQ47i
```

Part (b) -- 1pt

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

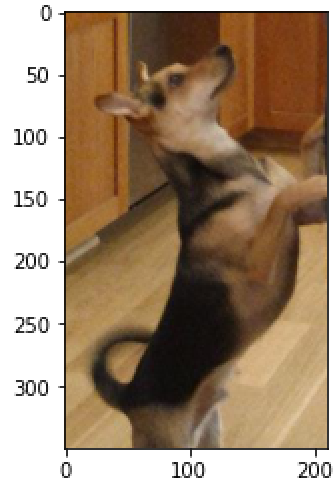
This function will also show the coordinate system used to identify pixels. The origin is at the top left corner, and the first dimension indicates the Y (row) direction, and the second dimension indicates the X (column) dimension.

In [ ]:

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

Out[ ]:

```
<matplotlib.image.AxesImage at 0x7fc896e86240>
```



Part (c) -- 2pt

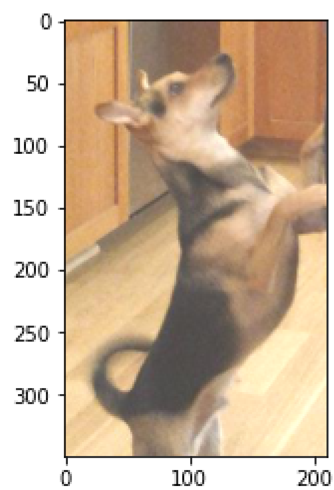
Modify the image by adding a constant value of 0.25 to each pixel in the `img` and store the result in the variable `img_add`. Note that, since the range for the pixels needs to be between [0, 1], you will also need to clip img_add to be in the range [0, 1] using `numpy.clip`. Clipping sets any value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

In [100…

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

plt.imshow(img_add)
```

Out[100… <matplotlib.image.AxesImage at 0x7f14b018c5c0>



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.

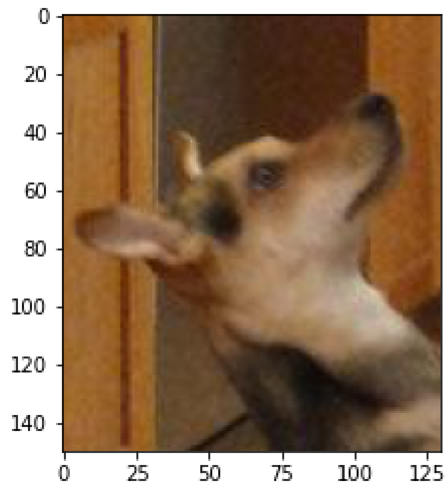
In [210…

```
width, height = 130, 150
```

```
x, y = 20, 0

img_cropped = img[y : y + height, x : x + width, :3]
plt.imshow(img_cropped)
```

Out[210…
```
<matplotlib.image.AxesImage at 0x7f14a6a1edd8>
```



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.

In [19]:
```
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`.

In [110…
```
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`.

In [104…
```
img_torch.shape
```

Out[104…
```
torch.Size([150, 130, 3])
```

Part (c) -- 1pt

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

In [170…
```
count = 0
rows, cols, rgb = img_torch.shape

for i in range(rows):
  for j in range(cols):
    for k in range(rgb):
      if (img_torch[i][j][k].is_floating_point):
        count += 1

count
```

Out[170…
```
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.

`ANSWER:`

It returns a tensor that has transposed img_torch by swapping the dimensions at 0 with 2 (the dimensions go from `[150, 130, 3]` in `img_torch` to `[3, 130, 150]` once transposed).

This does not transpose the original variable `img_torch` . However, the documentations states both tensors share the same underlying data which means if the value in one tensor is updated, it will also update the other.

In [182…
```
updated_img_torch = img_torch.transpose(0, 2)

print(img_torch.shape)
print(updated_img_torch.shape)
```

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

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.

`Answer`

According to the documentation, `img_torch.unsqueeze(0)` returns a new tensor with a dimension of size one inserted at position 0. Using `.shape` , the dimensions of `img_torch` are `[150, 130, 3]` and the new tensor's dimensions are `[1, 150, 130, 3]` .

While both the tensors share the same underlying data, `unsqueeze` doesn't update `img_torch` with the new dimension. The rest of the dimensions however will be affected if there are future changes to either tensor.

In [183…

```
updated_img_torch = img_torch.unsqueeze(0)

print(img_torch.shape)
print(updated_img_torch.shape)
```

```
torch.Size([150, 130, 3])
torch.Size([1, 150, 130, 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`.

In [181…

```
rows, cols, rgb = img_torch.shape
ans = [0, 0, 0]

for i in range(rgb):
  ans[i] = torch.max(img_torch[:, :, i])

torch.from_numpy(np.array(ans))
```

Out[181…

```
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

In [99]:

```
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" ta
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 pa
    optimizer.step()                   # step 4 (make the updates for each param
    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: 0.006
Training Accuracy: 0.994
Test Error Rate: 0.075
Test Accuracy: 0.925
```

In [ ]:

My Observations

Here are my results, with each parameter having been individually changed.

Parameter Changed	Value	Accuracy on Traning Data	Accuracy on Testing Data
Training Iterations	1 (original)	0.964	0.921
	2	0.984	0.943
	5	0.989	0.9339999999999999
	10	0.999	0.9410000000000001
	20	0.999	0.942
Hidden Units (HU)	30 (original)	0.964	0.921
	50	0.967	0.926
	100	0.97	0.923
	1000	0.982	0.935
Layers	2 (original)	0.964	0.921
	3 (60 HU > 30 HU > 1 output)	0.953	0.903
	4 (90 HU > 60 HU > 30 HU > 1 output)	0.947	0.915
Type of Activation Function	relu	0.964	0.921
	hardtanh	0.953	0.894
	elu	0.956	0.902
Learning Rate	0.005 (original)	0.964	0.921
	0.01	0.961	0.918
	0.1	0.688	0.7030000000000001
	1	0.6890000000000001	0.702

General conclusions

Increasing these increased accuracy:

- training iterations
- hidden units

Increasing these decreased accuracy:

- layers
- learning rate

Best of the three activation functions I tried was the original `relu` .

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?

`ANSWER`

The best accuracy I found on the training data was 99.9%, achieved at 10 training iterations (refer to table above).

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?

`ANSWER`

The best accuracy I found on the testing data was 94.3%, achieved at 2 training iterations (refer to table above).

Part (c) -- 4 pt

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

`ANSWER`

Part a.

Explanation:

The best accuracy in both the training and test data was achieved with changing the # of iterations.

Parameter Changed	Value	Accuracy on Traning Data	Accuracy on Testing Data
Training Iterations	1 (original)	0.964	0.921
	2	0.984	`0.943`
	5	0.989	0.9339999999999999
	10	`0.999`	0.9410000000000001
	20	`0.999`	0.942

Since I changed one parameter at a time, these observations show that though more iterations increases accuracy on the training data, it doesn't directly correlate on the testing data.

At first, this seemed that it was a sign that the model was "too" good on the test data with it's `99.9%` accuracy and thus, overfitted.

However, the overfitting/bias would actually come if we took the results from part b). We should only take the insight from part a) to tweak the parameters, and only use the test data when the model ready to take on the "real world" with unseen data. With each new test data, the slight variations in accuracy are to be expected.