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: https://colab.research.google.com/drive/1e7RYK3rbwEzQN3YRo3G09Dcz0_aKYlkg#scrollTo=pi6bWs7jclem&uniqifier=1

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):
    """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
    """
```

Saved successfully!

```
    cubes = [i ** 3 for i in range(1, n+1)]
    return sum(cubes)
```

```
# part A tests
sum_of_cubes(-1)
sum_of_cubes("hi")
sum_of_cubes(3)
```

```
Invalid input
Invalid input
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 words in the string, using sep as the delimiter string.

    sep
      The delimiter according which to split the string.
      None (the default value) means split according to any whitespace,
      and discard empty strings from the result.
    maxsplit
      Maximum number of splits to do.
      -1 (the default value) means no limit.
```

```
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]
    """
    parts = sentence.split()
    return [len(part) for part in parts]
```

```
# part B tests
word_lengths("1 11 111 1111 11111")
```

```
[1, 2, 3, 4, 5]
```

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
    """
    if len(sentence) == 0:
        return False
```

Saved successfully!

```
r part in sentence.split()]
```

```
    num = parts_lengths[0]
    return all([len(part_length) == num for part_length in parts_lengths ])
```

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
```

```
12
```

```
matrix.shape
```

```
(3, 4)
```

```
vector.size
```

```
4
```

```
vector.shape
```

```
(4,)
```

The size property of a NumpyArray object is the amount of elements in that array, so for a matrix it is the product of the dimensions of the matrix, e.g., a 2D array with dimensions N (rows) and M (columns) would return $N \cdot M$. Similarly, calling it on a vector returns the number of elements in a vector (as it can be thought of as a $N \times 1$ matrix). The shape property simply returns the dimensions of the array so first number of rows, then columns, and so on.

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 = []

for i in range(len(matrix)):
    sum_temp = 0
    for j in range(len(matrix[0])):
        sum_temp += matrix[i][j] * vector[j]
    output.append(sum_temp)

output = np.array(output)
```

```
output
```

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

Part (c) -- 1pt

`= 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.

```
output == output2
```

```
array([ 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
```

```
0.004560232162475586
```

```
# Time it takes  for the vectorized operations
import time

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

# place code to run here
for _ in range(10000):
    output = []
    for i in range(len(matrix)):
        sum_temp = 0
        for j in range(len(matrix[0])):
            sum_temp += matrix[i][j] * vector[j]
        output.append(sum_temp)

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

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

```
0.20680785179138184
```

```
# Time it takes  for the vectorized operations
import time

code

```

Saved successfully!

```
# place code to run here
for _ in range(10000):
    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
```

```
0.028533458709716797
```

As seen from the timing examples, the numpy call was much faster (0.028533458709716797) than the python for loop (0.20680785179138184). This is due to python needing to interpret the native python method and numpy calling its C backend. Numpy also has support of SIMD instructions which are present on many modern CPUs.

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.

```
# reading directly from a URL is deprecated  - https://matplotlib.org/3.5.1/api/_as_gen/matplotlib.pyplot.imread.html
import urllib
file = urllib.request.urlopen("https://drive.google.com/uc?export=view&id=1oaLVR2hr1_qzpKQ47i9rVUIklwbDcews")
img = plt.imread(file)
```

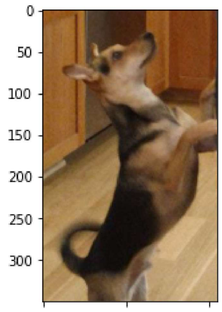
Part (b) -- 1pt

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.

Saved successfully!

```
<matplotlib.image.AxesImage at 0x7f6461890310>
```

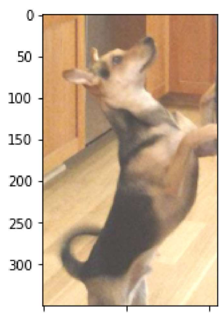


Part (c) -- 2pt

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

```
<matplotlib.image.AxesImage at 0x7f6461810ac0>
```



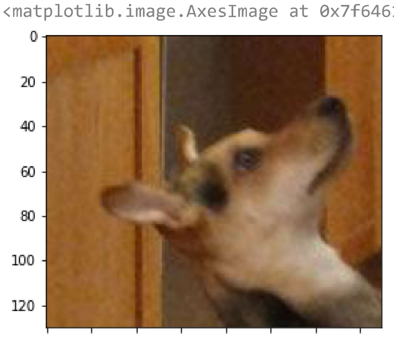
Part (d) -- 2pt

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[:130, :150, :3]
img_cropped.shape
plt.imshow(img_cropped)
```

```
<matplotlib.image.AxesImage at 0x7f64617ea790>
```



Part 4. Basics of PyTorch [6 pt]

Saved successfully!

orks package. Along with tensorflow, PyTorch is currently one of the most popular machine learning

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)
```

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`?

```
torch.numel(img_torch)
```

```
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)
img_torch.transpose(0, 2).shape
```

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

```
img_torch.shape
```

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

It returns a new tensor with the first and third dimensions swapped, which in this case means that the three colour channels become the first dimension and the rows of the image become the last dimension. The original `img_torch` variable is not changed.

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)
img_torch.unsqueeze(0).shape
```

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

From the pytorch docs (https://pytorch.org/docs/stable/generated/torch.unsqueeze.html): "Returns a new tensor with a dimension of size one inserted at the specified position". So here, it returns a new tensor (does not modify `img_torch`) with a new dimension of size 1 inserted at position 0

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_max = torch.max(img_torch[:, :, 0])
green_max = torch.max(img_torch[:, :, 1])
blue_max = torch.max(img_torch[:, :, 2])
red_max, green_max, blue_max
```

```
(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

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

# 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()

def model_loop(lr = 0.005, momentum=0.9, hidden_units = 30, train_iters = 10, activation = F.relu):
    # define a 2-layer artificial neural network
  class Pigeon(nn.Module):
      def __init__(self):
          super(Pigeon, self).__init__()
          self.layer1 = nn.Linear(28 * 28, hidden_units)
          self.layer2 = nn.Linear(hidden_units, 1)
      def forward(self, img):
          flattened = img.view(-1, 28 * 28)
          activation1 = self.layer1(flattened)
          activation1 = activation(activation1)
          activation2 = self.layer2(activation1)
          return activation2

  pigeon = Pigeon()
  # simplified training code to train `pigeon` on the "small digit recognition" task
  criterion = nn.BCEWithLogitsLoss()
  optimizer = optim.SGD(pigeon.parameters(), lr=lr, momentum=momentum)
  for epoch in range(train_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
                                           # step 4 (compute the updates for each parameter)
                                           # 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
  train_error = error/len(mnist_train)
  train_acc = 1-train_error
  print("Training Error Rate:", train_error)
  print("Training Accuracy:", train_acc)


  # 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
  test_error = error/len(mnist_val)
  test_acc = 1-test_error
  print("Test Error Rate:", test_error)
  print("Test Accuracy:", test_acc)

  return (train_error, train_acc, test_error, test_acc)

lrs = [0.005, 0.05]
momentums = [0.9, 0.5]
hidden_units = [30, 60]
train_iters = [5, 10]
activations = [F.relu, nn.SiLU()]

for lr in lrs:
  for momentum in momentums:
    for hidden_unit in hidden_units:
      for train_iter in train_iters:
        for activation in activations:
          print(f"Running pidgeon with hyperparams: lr ({lr}),"\
                f"momentum ({momentum}), hidden_units ({hidden_unit}),"\
                f"train_iters ({train_iter}), activation ({activation})")
          model_loop(lr = lr,
                     momentum= momentum,
                     hidden_units=hidden_unit,
                     train_iters=train_iter,
                     activation=activation)
```

```
Training Accuracy: 0.994
Test Error Rate: 0.069
Test Accuracy: 0.931
Running pidgeon with hyperparams: lr (0.05), momentum (0.5), hidden_units (30),  train_iters (10), activation (<function relu at 0x7f6
Training Error Rate: 0.003
Training Accuracy: 0.997
Test Error Rate: 0.057
Test Accuracy: 0.943
Running pidgeon with hyperparams: lr (0.05), momentum (0.5), hidden_units (30),  train_iters (10), activation (SiLU())
Training Error Rate: 0.0
Training Accuracy: 1.0
Test Error Rate: 0.061
Test Accuracy: 0.9390000000000001
Running pidgeon with hyperparams: lr (0.05), momentum (0.5), hidden_units (60),  train_iters (5), activation (<function relu at 0x7f64
Training Error Rate: 0.012
Training Accuracy: 0.988
Test Error Rate: 0.069
Test Accuracy: 0.931
Running pidgeon with hyperparams: lr (0.05), momentum (0.5), hidden_units (60),  train_iters (5), activation (SiLU())
Training Error Rate: 0.012
Training Accuracy: 0.988
```

Hyperparameters:

Learning Rates	Momentums	Epochs (Training Iterations)	Activations
0.005	0.9	30	ReLU
0.05	0.5	60	Swish (SiLU)

Note that all combinations of hyperparameter choices were tested.

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?

I was able to achieve a perfect training accuracy (1.0) on the training data with multiple combinations of hyperparameters:

- lr (0.005), momentum (0.9), hidden_units (30), train_iters (10), activation (SiLU)
- lr (0.005), momentum (0.9), hidden_units (60), train_iters (10), activation (SiLU)
- lr (0.005), momentum (0.9), hidden_units (60), train_iters (5), activation (ReLU)
- lr (0.005), momentum (0.9), hidden_units (60), train_iters (10), activation (ReLU)
- lr (0.05), momentum (0.5), hidden_units (30), train_iters (10), activation (SiLU)
- lr (0.05), momentum (0.5), hidden_units (60), train_iters (10), activation (SiLU)

So overall, having more epochs and model capacity (more hidden units per layer) with a lower learning rate and higher momentum yielded the perfect training accuracy. As an additional test (upon seeing lower performance of the higher learning rate), I was able to achieve perfect training accuracy on a larger learning rate if I decreased the momentum of the SGD optimizer (making the optimizer "bounce" less around the loss surface). The choice of activation function (Swish/SiLU) didn't seem to matter for this task.

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 test accuracy was 0.944 (94.4% accuracy) with the following choices of hyperparameters:

- lr (0.005), momentum (0.9), hidden_units (60), train_iters (10), activation (SiLU)
- lr (0.005), momentum (0.9), hidden_units (60), train_iters (10), activation (ReLU)
- lr (0.05), momentum (0.5), hidden_units (60), train_iters (10), activation (SiLU)
- lr (0.05), momentum (0.5), hidden_units (60), train_iters (10), activation (ReLU)

We can see that, genearlly, the hyperparameter choices that best maximize training accuracy also maximized test accuracy. That is, a model with more training epochs and more capacity with a lower learning rate and higher momentum yielded better test loss. Also if a higher learning rate is chosen, then a lower momentum yielded better test accuracy. The choice of activation function (Swish/SiLU) didn't seem to matter for this task. Finally, having more capacity here didn't seem to make the model overfit.

Part (c) -- 4 pt

Saved successfully!

you use, the ones from (a) or (b)?

We do not want to ultimately pick the hyperparameters that best minimize the training loss, instead we want to pick the hyperparameters that best minimize the test loss. This is because a really low training loss may indicate overfitting especially if the test loss is not comparable, e.g, if the test loss/evaluation metric is much worse than that of the training set. So the test set loss/accuracy is a proxy for out of sample performance. In this case, that would be when the trainig accuracy is much higher than the test accuracy. A good example of this the following hyperparameter combination: lr (0.05), momentum (0.5), hidden_units (60), train_iters (10), activation (ReLU). It did not result in a model that could be a perfect classifier, but it did achieve the best test accuracy I could get.

Saved successfully!