

▼ 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 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.

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/1DlxLOVpBaEoGYlauwpJi2Tj7cbpiwErz?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` is 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
    """
    return sum([i**3 for i in range(1, n+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 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)
```

Show hidden output

```
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()]
```

▼ 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
    """
    lengths = word_lengths(sentence)
    return len(set(lengths)) == 1
```

▼ 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
```

```
"""
```

```
Matrix size represents the number of elements in the matrix, which is 12.
```

```
"""
```

Show hidden output

```
matrix.shape
```

```
"""
```

```
Matrix shape represents the number of rows and columns (dimensions) in the matrix, whi
```

```
"""
```

Show hidden output

```
vector.size
```

```
"""
```

```
Vector size represents the number of elements in the vector, which is 4.
```

```
"""
```

Show hidden output

```
vector.shape
```

```
"""
```

```
Vector shape represents the number of rows and columns (dimensions) in the vector, whi
```

```
"""
```

Show hidden output

▼ 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 = np.zeros(matrix.shape[0])
```

```
for i in range(matrix.shape[0]):  
    for j in range(matrix.shape[1]):  
        output[i] += matrix[i][j] * vector[j]
```

```
output
```

Show hidden output

▼ 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(d), 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

Show hidden output

▼ Part (d) -- 1pt

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

```
print(output == output2)
```

Show hidden output

▼ 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

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

# place code to run here
for i in range(matrix.shape[0]):
    for j in range(matrix.shape[1]):
        output[i] += matrix[i][j] * vector[j]

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

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

Show hidden output

```
start_time2 = time.time()

# place code to run here
np.dot(matrix, vector)

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

# compute the difference
diff2 = end_time2 - start_time2
diff2
```

Show hidden output

▼ 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 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.

```
import urllib.request
from PIL import Image

url = 'https://drive.google.com/uc?export=view&id=1oaLVR2hr1_qzpKQ47i9rVUIklwbDcews'
response = urllib.request.urlopen(url)
img = np.array(Image.open(response))
img = img / 255.0
```

▼ 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)
```

Show hidden output

▼ 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)
```

Show hidden output

▼ 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[20:150, 30:180, :][:, :, :3]
plt.imshow(img_cropped)
```

<matplotlib.image.AxesImage at 0x7da96879cb20>



▼ 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 = 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
```

Show hidden output

▼ Part (c) -- 1pt

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

```
img_torch.numel()
```

Show hidden output

▼ 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.size()

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

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

Show hidden output

The transpose method swaps the dimensions of the tensor. For this case, the tensor has 3 dimensions, thus the first dimension (0) is swapped with the third dimension (2).

By checking the size of the tensors, we can observe that effectively the first and third dimensions are swapped.

▼ 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.size()

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

img_torch.unsqueeze(0).size()
```

Show hidden output

The unsqueeze method adds a dimension to the tensor. Image tensors are usually 3-dimensional, and by unsqueezing the tensor a fourth dimension of size 1 is added at the beginning of the tensor (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`.

```
max_ = torch.max(img_torch, 0) # Get the maximum value on dimension 0
max_ = torch.max(max_[0], 0) # Get the maximum value on dimension 0 and 1
```

```
max_[0] # Get the maximum value from the tuple

tensor([0.8941, 0.7882, 0.6745], dtype=torch.float64)
```

▼ 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

# 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=0.001, momentum=0.9)

for epoch in range(1):
    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 para
        optimizer.step()                  # step 4 (make the updates for each paramet
        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))

```

Show hidden output

```

# Modified to change the hyperparams faster
learning_rate = 0.005
train_epochs = 10
hidden_units = 100

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, hidden_units)
        self.layer2 = nn.Linear(hidden_units, 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=learning_rate, momentum=0.9)

for epoch in range(train_epochs):
    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 para
        optimizer.step() # step 4 (make the updates for each paramet
        optimizer.zero_grad() # a clean up step for PyTorch

# computing the error and accuracy on the training set
error = 0
for epoch in range(train_epochs):
    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: 0.058
Test Accuracy: 0.942

```

▼ 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?

Changed parameter	Value	Training Accuracy
Control (default)		0.923
Number of training iterations	3	0.968
Number of training iterations	5	0.977
Number of training iterations	10	0.997
Number of hidden units	50	0.928
Number of hidden units	100	0.923
Number of hidden units	150	0.925
Learning rate	0.003	0.966
Learning rate	0.005	0.964
Learning rate	0.01	0.961

All the other parameters are the same as the default.

We can observe that the most accurate training came from when iterations were at 10. This is because the more iterations we have, the more the model is trained and the more accurate it becomes. However, we can also observe that the accuracy is not always increasing with the number of iterations. This is because the model is overfitting the training data. There seems to be no significant change on the accuracy as we tweak the number of hidden units, and the same could be

said about the learning rate. We have to consider that this is a very simple model, with a small

▼ 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?

Changed parameter	Value	Testing Accuracy
Control (default)		0.887
Number of training iterations	3	0.913
Number of training iterations	5	0.918
Number of training iterations	10	0.927
Number of hidden units	50	0.888
Number of hidden units	100	0.892
Number of hidden units	150	0.887
Learning rate	0.003	0.91
Learning rate	0.005	0.921
Learning rate	0.01	0.918

All the other parameters are the same as the default.

10 iterations yielded the highest testing data accuracy. This means that even though we might think that the model is overfitting the training data, it is still able to generalize well to the testing data. This could be due to the small number of data we input as training and testing and the simplicity of the model itself. Even the number of hidden units and the learning rate did not have a significant impact on the accuracy of the testing data, as they keep a similar ratio to the training accuracy. One thing to note is that making the learning faster has better results on the testing data than the default values set on the Lab.

▼ Part (c) -- 4 pt

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

We can't really decide between the two, as both of them had better results when at 10 iterations. However, since the accuracy of training data has a bigger weight when determining the usefulness of a model, if we change the amount of iterations to 10, with a 100 hidden units and a learning rate of 0.005, we can achieve a testing accuracy of 0.942. This is the best result we could achieve with