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://docs.scipy.org/doc/numpy/reference/)
- https://pytorch.org/docs/stable/torch.html (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/1GBb8XtvP2lYFiOREJN-lkirKvt9PvpxA?usp=sharing (https://colab.research.google.com/drive/1GBb8XtvP2lYFiOREJN-lkirKvt9PvpxA?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/ (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 [ ]:

```
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
    """
    #check for invalid input, as n must >0 and is integer
    if n<=0 or (isinstance(n, int))==False:
      return -1

    sum_toReturn = 0
    for i in range(1,n+1):
      sum_toReturn += pow(i,3)

    return sum_toReturn

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 (https://docs.python.org/3.6/library/stdtypes.html#str.split)

In [ ]:

```
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.
```

In [ ]:

```
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]
    """
    #convert the sentence to the word list
    word_lst = sentence.split(" ")

    word_length = [0]*len(word_lst)
    for i in range(len(word_lst)):
      word_length[i] = len(word_lst[i])

    return word_length

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 [ ]:

```
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
    >>> all_same_length("hello world")
    True
    """
    #convert the sentence to the word list
    word_lst = sentence.split(" ")

    word_length = len(word_lst[0])
    for i in range(1,len(word_lst)):
      #not in the same length
      if word_length != len(word_lst[i]):
        return False

    #in the same length
    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 [ ]:

```
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:

1. In my opinion, the `<NumpyArray>.size` will return the number of individual elements inside the ndarrays.

   Or we can say that it will first reshape the ndarray to an one-dimension array, then return the size of the 1d array, which will also be the number of element inside the ndarray.
2. The `<NumpyArray>.shape` will return a tuple which has the length of its dimension. Each number inside the tuple will be size of each dimension.

   For example, the matrix is a 2d array, which contains three 1d arrays inside. Each 1d array contains 4 elements. So, the shape of the matrix is (3,4).

   Also, to be noticed is that, a tuple with one element has a trailing comma. So, the shape of vector is (4,), which has 4 elements in the 1d array and a trailing comma existing in the tuple.

In [ ]:

```
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 [ ]:

```
output = None
```

In [ ]:

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

#cast the list to ndarray
output = np.array(output)
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 [ ]:

```
output2 = None
```

In [ ]:

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

Out[ ]:

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

Part (d) -- 1pt

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

In [ ]:

```
if (output == output2).all:
  print(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:

In [ ]:

```
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
```

Out[ ]:

```
0.0010972023010253906
```

In [ ]:

```
#partB
start_time = time.time()

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

end_time = time.time()
B = end_time - start_time

#partC
start_time = time.time()

output2 = np.dot(matrix, vector)

end_time = time.time()
C = end_time - start_time

if C<B:
  print("Time for C (" + str(C) + ") is shorter than time for B (" + str(B)+ ")"
)
```

```
Time for C (4.100799560546875e-05) is shorter than time for B (0.000
15234947204589844)
```

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 [ ]:

```
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.

In [ ]:

```
img = plt.imread("https://drive.google.com/uc?export=view&id=1oaLVR2hr1_qzpKQ47i9rVUIklwbDcews")
```

Part (b) -- 1pt

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

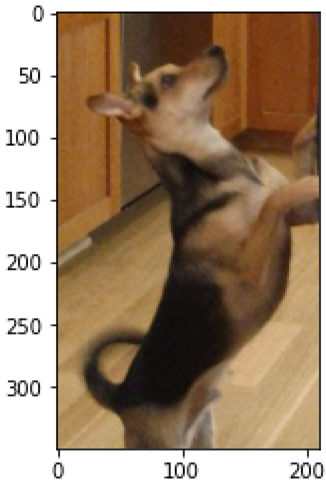
This function will also show the coordinate system used to identify pixels. The origin is at the top left corner, and the first dimension indicates the Y (row) direction, and the second dimension indicates the X (column) dimension.

In [ ]:

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

Out[ ]:

```
<matplotlib.image.AxesImage at 0x7f2b0ac42b38>
```



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` .

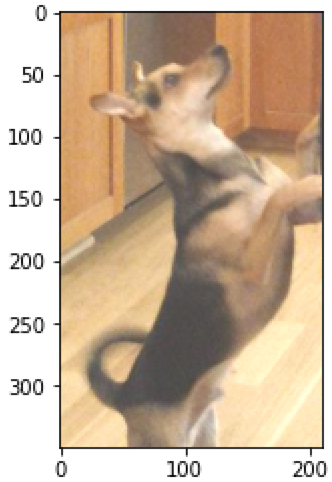
In [ ]:

```
#Modify the image by adding a constant value of 0.25 to each pixel
img_add = np.array(img)
img_add += 0.25

#you will also need to clip img_add to be in the range [0, 1] using numpy.clip
np.clip(img_add, a_min=0, a_max=1, out=img_add)
plt.imshow(img_add)
```

Out[ ]:

```
<matplotlib.image.AxesImage at 0x7f2b0ad9dc88>
```



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**)

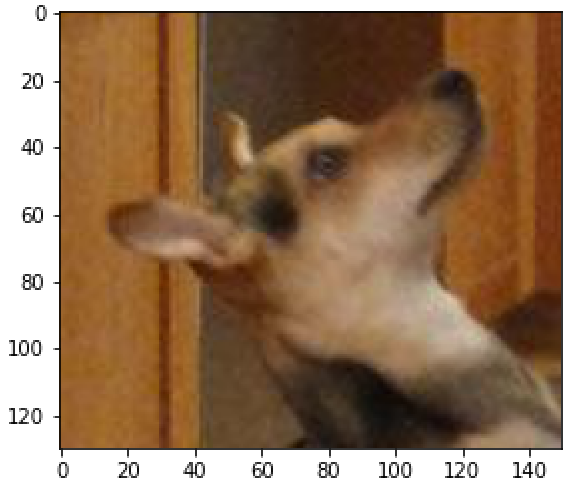
Display the image.

In [ ]:

```python
#the first dimension indicates the Y (row) direction, and the second dimension indicates the X (column) dimension.
img_cropped = img[10:140, 10:160, 0:3]    #red[:,:,0], green[:,:,1], blue[:,:,2]
plt.imshow(img_cropped)
```

Out[ ]:

```
<matplotlib.image.AxesImage at 0x7f2b0b6ccf60>
```



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 [ ]:

```python
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 [ ]:

```python
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 [ ]:

```
img_torch.shape
```

Out[ ]:

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

Part (c) -- 1pt

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

In [ ]:

```
num = 1;
for i in range(0, len(img_torch.shape)):
  num *= img_torch.shape[i]
print(num)
```

```
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:

As we can see from the output below, the code `img_torch.transpose(0,2)` is for transposing the given tensor array and it will print out or generate the transpose nd-array of the original nd-array. The original variable `img_torch` has not been updated.

The original img_torch has the size of [130,150,3] , after calling the transpose(0,2) function, the new nd-array becomes [3,150,130] with the original remains unchanged.

In [ ]:

```
print("--------------------------------------original nd-array------------------
--")
print(img_torch)
print(img_torch.shape)
print(id(img_torch))
print("--------------------------------------transposed nd-array----------------
----")
print(img_torch.transpose(0,2))
print(img_torch.transpose(0,2).shape)
print(id(img_torch.transpose(0,2)))
print("--------------------------------------check if the orignal nd-array is up
dated--------------------")
print(img_torch)
print(img_torch.shape)
print(id(img_torch))
```

```
--------------------------------------original nd-array-------------
-------
tensor([[[0.5529, 0.3412, 0.0588],
         [0.5569, 0.3412, 0.0784],
         [0.5725, 0.3529, 0.1137],
         ...,
         [0.5098, 0.2431, 0.0980],
         [0.5373, 0.2627, 0.1216],
         [0.5216, 0.2471, 0.1059]],

        [[0.5529, 0.3412, 0.0588],
         [0.5647, 0.3490, 0.0863],
         [0.5804, 0.3647, 0.1137],
         ...,
         [0.5020, 0.2353, 0.0902],
         [0.5216, 0.2471, 0.1059],
         [0.5059, 0.2314, 0.0902]],

        [[0.5451, 0.3451, 0.1216],
         [0.5608, 0.3608, 0.1451],
         [0.5725, 0.3686, 0.1725],
         ...,
         [0.4941, 0.2431, 0.0941],
         [0.4745, 0.2235, 0.0745],
         [0.4745, 0.2235, 0.0745]],

        ...,

        [[0.5922, 0.3725, 0.1804],
         [0.5961, 0.3765, 0.1922],
         [0.6000, 0.3804, 0.1961],
         ...,
         [0.5490, 0.4745, 0.4196],
         [0.5176, 0.4392, 0.3961],
         [0.4980, 0.4118, 0.3686]],

        [[0.6000, 0.3804, 0.1961],
         [0.6000, 0.3804, 0.1961],
         [0.5961, 0.3725, 0.1961],
         ...,
         [0.5765, 0.5059, 0.4510],
         [0.5725, 0.4941, 0.4510],
         [0.5686, 0.4824, 0.4392]],

        [[0.6039, 0.3804, 0.2039],
         [0.6000, 0.3765, 0.2000],
         [0.5922, 0.3686, 0.1922],
         ...,
         [0.6118, 0.5412, 0.4863],
         [0.6078, 0.5294, 0.4863],
         [0.6078, 0.5176, 0.4863]]])
torch.Size([130, 150, 3])
139822872695432
--------------------------------------transposed nd-array-----------
---------
tensor([[[0.5529, 0.5529, 0.5451,  ..., 0.5922, 0.6000, 0.6039],
         [0.5569, 0.5647, 0.5608,  ..., 0.5961, 0.6000, 0.6000],
         [0.5725, 0.5804, 0.5725,  ..., 0.6000, 0.5961, 0.5922],
         ...,
         [0.5098, 0.5020, 0.4941,  ..., 0.5490, 0.5765, 0.6118],
         [0.5373, 0.5216, 0.4745,  ..., 0.5176, 0.5725, 0.6078],
```

```
         [0.5216, 0.5059, 0.4745,  ..., 0.4980, 0.5686, 0.6078]],

        [[0.3412, 0.3412, 0.3451,  ..., 0.3725, 0.3804, 0.3804],
         [0.3412, 0.3490, 0.3608,  ..., 0.3765, 0.3804, 0.3765],
         [0.3529, 0.3647, 0.3686,  ..., 0.3804, 0.3725, 0.3686],
         ...,
         [0.2431, 0.2353, 0.2431,  ..., 0.4745, 0.5059, 0.5412],
         [0.2627, 0.2471, 0.2235,  ..., 0.4392, 0.4941, 0.5294],
         [0.2471, 0.2314, 0.2235,  ..., 0.4118, 0.4824, 0.5176]],

        [[0.0588, 0.0588, 0.1216,  ..., 0.1804, 0.1961, 0.2039],
         [0.0784, 0.0863, 0.1451,  ..., 0.1922, 0.1961, 0.2000],
         [0.1137, 0.1137, 0.1725,  ..., 0.1961, 0.1961, 0.1922],
         ...,
         [0.0980, 0.0902, 0.0941,  ..., 0.4196, 0.4510, 0.4863],
         [0.1216, 0.1059, 0.0745,  ..., 0.3961, 0.4510, 0.4863],
         [0.1059, 0.0902, 0.0745,  ..., 0.3686, 0.4392, 0.4863]]])
torch.Size([3, 150, 130])
139822841126488
--------------------------------------check if the orignal nd-array
is updated--------------------
tensor([[[0.5529, 0.3412, 0.0588],
         [0.5569, 0.3412, 0.0784],
         [0.5725, 0.3529, 0.1137],
         ...,
         [0.5098, 0.2431, 0.0980],
         [0.5373, 0.2627, 0.1216],
         [0.5216, 0.2471, 0.1059]],

        [[0.5529, 0.3412, 0.0588],
         [0.5647, 0.3490, 0.0863],
         [0.5804, 0.3647, 0.1137],
         ...,
         [0.5020, 0.2353, 0.0902],
         [0.5216, 0.2471, 0.1059],
         [0.5059, 0.2314, 0.0902]],

        [[0.5451, 0.3451, 0.1216],
         [0.5608, 0.3608, 0.1451],
         [0.5725, 0.3686, 0.1725],
         ...,
         [0.4941, 0.2431, 0.0941],
         [0.4745, 0.2235, 0.0745],
         [0.4745, 0.2235, 0.0745]],

        ...,

        [[0.5922, 0.3725, 0.1804],
         [0.5961, 0.3765, 0.1922],
         [0.6000, 0.3804, 0.1961],
         ...,
         [0.5490, 0.4745, 0.4196],
         [0.5176, 0.4392, 0.3961],
         [0.4980, 0.4118, 0.3686]],

        [[0.6000, 0.3804, 0.1961],
         [0.6000, 0.3804, 0.1961],
         [0.5961, 0.3725, 0.1961],
         ...,
         [0.5765, 0.5059, 0.4510],
         [0.5725, 0.4941, 0.4510],
```

```
          [0.5686, 0.4824, 0.4392]],

         [[0.6039, 0.3804, 0.2039],
          [0.6000, 0.3765, 0.2000],
          [0.5922, 0.3686, 0.1922],
          ...,
          [0.6118, 0.5412, 0.4863],
          [0.6078, 0.5294, 0.4863],
          [0.6078, 0.5176, 0.4863]]]])
torch.Size([130, 150, 3])
139822872695432
```

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:

As we can see from the output below, the code `img_torch.unsqueeze(0)` is for adding one dimension the given tensor array and it will print out or generate the (n+1)d-array of the original nd-array. The original variable `img_torch` has not been updated.

The original img_torch has the size of [130,150,3] , after calling the unsqueeze(0) function, the new nd-array becomes [1,130,150,3] with the original remains unchanged.

In [ ]:

```
print("--------------------------------------original nd-array--------------------")
print(img_torch)
print(img_torch.shape)
print(id(img_torch))
print("--------------------------------------new (n+1)d-array--------------------")
print(img_torch.unsqueeze(0))
print(img_torch.unsqueeze(0).shape)
print(id(img_torch.unsqueeze(0)))
print("--------------------------------------check if the orignal nd-array is updated--------------------")
print(img_torch)
print(img_torch.shape)
print(id(img_torch))
```

```
--------------------------------------original nd-array--------------
-------
tensor([[[0.5529, 0.3412, 0.0588],
         [0.5569, 0.3412, 0.0784],
         [0.5725, 0.3529, 0.1137],
         ...,
         [0.5098, 0.2431, 0.0980],
         [0.5373, 0.2627, 0.1216],
         [0.5216, 0.2471, 0.1059]],

        [[0.5529, 0.3412, 0.0588],
         [0.5647, 0.3490, 0.0863],
         [0.5804, 0.3647, 0.1137],
         ...,
         [0.5020, 0.2353, 0.0902],
         [0.5216, 0.2471, 0.1059],
         [0.5059, 0.2314, 0.0902]],

        [[0.5451, 0.3451, 0.1216],
         [0.5608, 0.3608, 0.1451],
         [0.5725, 0.3686, 0.1725],
         ...,
         [0.4941, 0.2431, 0.0941],
         [0.4745, 0.2235, 0.0745],
         [0.4745, 0.2235, 0.0745]],

        ...,

        [[0.5922, 0.3725, 0.1804],
         [0.5961, 0.3765, 0.1922],
         [0.6000, 0.3804, 0.1961],
         ...,
         [0.5490, 0.4745, 0.4196],
         [0.5176, 0.4392, 0.3961],
         [0.4980, 0.4118, 0.3686]],

        [[0.6000, 0.3804, 0.1961],
         [0.6000, 0.3804, 0.1961],
         [0.5961, 0.3725, 0.1961],
         ...,
         [0.5765, 0.5059, 0.4510],
         [0.5725, 0.4941, 0.4510],
         [0.5686, 0.4824, 0.4392]],

        [[0.6039, 0.3804, 0.2039],
         [0.6000, 0.3765, 0.2000],
         [0.5922, 0.3686, 0.1922],
         ...,
         [0.6118, 0.5412, 0.4863],
         [0.6078, 0.5294, 0.4863],
         [0.6078, 0.5176, 0.4863]]])
torch.Size([130, 150, 3])
139822872695432
--------------------------------------new (n+1)d-array--------------
------
tensor([[[[0.5529, 0.3412, 0.0588],
          [0.5569, 0.3412, 0.0784],
          [0.5725, 0.3529, 0.1137],
          ...,
          [0.5098, 0.2431, 0.0980],
          [0.5373, 0.2627, 0.1216],
```

```
          [0.5216, 0.2471, 0.1059]],

         [[0.5529, 0.3412, 0.0588],
          [0.5647, 0.3490, 0.0863],
          [0.5804, 0.3647, 0.1137],
          ...,
          [0.5020, 0.2353, 0.0902],
          [0.5216, 0.2471, 0.1059],
          [0.5059, 0.2314, 0.0902]],

         [[0.5451, 0.3451, 0.1216],
          [0.5608, 0.3608, 0.1451],
          [0.5725, 0.3686, 0.1725],
          ...,
          [0.4941, 0.2431, 0.0941],
          [0.4745, 0.2235, 0.0745],
          [0.4745, 0.2235, 0.0745]],

         ...,

         [[0.5922, 0.3725, 0.1804],
          [0.5961, 0.3765, 0.1922],
          [0.6000, 0.3804, 0.1961],
          ...,
          [0.5490, 0.4745, 0.4196],
          [0.5176, 0.4392, 0.3961],
          [0.4980, 0.4118, 0.3686]],

         [[0.6000, 0.3804, 0.1961],
          [0.6000, 0.3804, 0.1961],
          [0.5961, 0.3725, 0.1961],
          ...,
          [0.5765, 0.5059, 0.4510],
          [0.5725, 0.4941, 0.4510],
          [0.5686, 0.4824, 0.4392]],

         [[0.6039, 0.3804, 0.2039],
          [0.6000, 0.3765, 0.2000],
          [0.5922, 0.3686, 0.1922],
          ...,
          [0.6118, 0.5412, 0.4863],
          [0.6078, 0.5294, 0.4863],
          [0.6078, 0.5176, 0.4863]]]])
torch.Size([1, 130, 150, 3])
139822884090144
--------------------------------------check if the orignal nd-array
is updated--------------------
tensor([[[[0.5529, 0.3412, 0.0588],
         [0.5569, 0.3412, 0.0784],
         [0.5725, 0.3529, 0.1137],
         ...,
         [0.5098, 0.2431, 0.0980],
         [0.5373, 0.2627, 0.1216],
         [0.5216, 0.2471, 0.1059]],

        [[0.5529, 0.3412, 0.0588],
         [0.5647, 0.3490, 0.0863],
         [0.5804, 0.3647, 0.1137],
         ...,
         [0.5020, 0.2353, 0.0902],
         [0.5216, 0.2471, 0.1059],
```

```
         [0.5059, 0.2314, 0.0902]],

        [[0.5451, 0.3451, 0.1216],
         [0.5608, 0.3608, 0.1451],
         [0.5725, 0.3686, 0.1725],
         ...,
         [0.4941, 0.2431, 0.0941],
         [0.4745, 0.2235, 0.0745],
         [0.4745, 0.2235, 0.0745]],

        ...,

        [[0.5922, 0.3725, 0.1804],
         [0.5961, 0.3765, 0.1922],
         [0.6000, 0.3804, 0.1961],
         ...,
         [0.5490, 0.4745, 0.4196],
         [0.5176, 0.4392, 0.3961],
         [0.4980, 0.4118, 0.3686]],

        [[0.6000, 0.3804, 0.1961],
         [0.6000, 0.3804, 0.1961],
         [0.5961, 0.3725, 0.1961],
         ...,
         [0.5765, 0.5059, 0.4510],
         [0.5725, 0.4941, 0.4510],
         [0.5686, 0.4824, 0.4392]],

        [[0.6039, 0.3804, 0.2039],
         [0.6000, 0.3765, 0.2000],
         [0.5922, 0.3686, 0.1922],
         ...,
         [0.6118, 0.5412, 0.4863],
         [0.6078, 0.5294, 0.4863],
         [0.6078, 0.5176, 0.4863]]])
torch.Size([130, 150, 3])
139822872695432
```

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 [ ]:

```
#Returns the maximum value of all elements in the input tensor.
max_lst = [0]*3
max_lst[0] = torch.max(img_torch[:,:,0]) #R
max_lst[1] = torch.max(img_torch[:,:,1]) #G
max_lst[2] = torch.max(img_torch[:,:,2]) #B
print(max_lst)
```

```
[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

Original sample code:

- Training Accuracy: 0.964
- Test Accuracy: 0.921

In [ ]:

```
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
sk
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
rameter)
    optimizer.step()                   # step 4 (make the updates for each param
eter)
    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: 0.036
Training Accuracy: 0.964
Test Error Rate: 0.079
Test Accuracy: 0.921
```

Trail 1: number of training iterations

- I tried by increasing the number of training iterations to 10.
- Training Accuracy: 0.999
- Test Accuracy: 0.9410000000000001

In [ ]:

```
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
sk
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.SGD(pigeon.parameters(), lr=0.005, momentum=0.9)

for i in range(10):
    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 eac
h parameter)
        optimizer.step()                   # step 4 (make the updates for each p
arameter)
        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: 0.001
Training Accuracy: 0.999
Test Error Rate: 0.059
Test Accuracy: 0.9410000000000001
```

Trail 2: number of hidden units

- I tried by substuting the original hidden units from 30 to 100 on the first layer.
- Training Accuracy: 0.97
- Test Accuracy: 0.923

In [ ]:

```
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, 100)
        self.layer2 = nn.Linear(100, 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
sk
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
rameter)
    optimizer.step()                   # step 4 (make the updates for each param
eter)
    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: 0.03
Training Accuracy: 0.97
Test Error Rate: 0.077
Test Accuracy: 0.923
```

Trail 3: numbers of layers

- I tried by adding one layer to the model, which becomes 3 layers.
- Training Accuracy: 0.959
- Test Accuracy: 0.9

If the model is more complex with more training data sets, by increasing the number of layers will have positive effects on the accuracy. However, for simple models or for a model that has sufficient amount of layers, if we applied more layers to it, it will decrease the accuracy.

In [ ]:

```
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, 15)
        self.layer3 = nn.Linear(15, 1)
    def forward(self, img):
        flattened = img.view(-1, 28 * 28)
        activation1 = self.layer1(flattened)
        activation1 = F.relu(activation1)
        activation2 = self.layer2(activation1)
        activation2 = F.relu(activation2)
        activation3 = self.layer3(activation2)
        return activation3

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
sk
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
rameter)
    optimizer.step()                   # step 4 (make the updates for each param
eter)
    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: 0.041
Training Accuracy: 0.959
Test Error Rate: 0.1
Test Accuracy: 0.9
```

Trail 4: types of activation functions

- I tried by changing the activation function function from relu() to leaky_relu(), which results the highest traning and testing accuracy.

tanh():

- Training Accuracy: 0.96
- Test Accuracy: 0.906

sigmoid():

- Training Accuracy: 0.927
- Test Accuracy: 0.883

softmax():

- Training Accuracy: 0.688
- Test Accuracy: 0.7030000000000001

leaky_relu():

- Training Accuracy: 0.963
- Test Accuracy: 0.921

In [ ]:

```
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.leaky_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
sk
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
rameter)
    optimizer.step()                   # step 4 (make the updates for each param
eter)
    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: 0.037
Training Accuracy: 0.963
Test Error Rate: 0.079
Test Accuracy: 0.921
```

Trail 5: learning rate

- I tried by adjusting the lr value from 0.005 to a number in range [0.001 : 0.0015 : 0.0005] and find out when lr = 0.0045, both traning accuracy and test accuracy are high compare to the number within these trails.
- Training Accuracy: 0.967
- Test Accuracy: 0.915

In [ ]:

```
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
sk
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.SGD(pigeon.parameters(), lr=0.0045, 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
rameter)
    optimizer.step()                   # step 4 (make the updates for each param
eter)
    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: 0.033
Training Accuracy: 0.967
Test Error Rate: 0.085
Test Accuracy: 0.915
```

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:

Based on the trails and the results above, we generate the highest reference number from each trail to be:

- original samle - Training accuracy: 0.964
- training iterations - Training accuracy: 0.999
- hidden units - Training accuracy: 0.97
- layers - Training accuracy: 0.959
- activation functions - Training accuracy: 0.963
- learning rate - Training accuracy: 0.967

As we can see, there is a significant effect/improve on training accuracy when we increases the training iterations, which achieves 0.999.

Also, by adjusing the learning rate and adding hidden units, these two methods also increase the training accuracy by a small amount, which are 0.967 and 0.97 respectively.

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:

Based on the trails and the results above, we generate the highest reference number from each trail to be:

- original samle - Test Accuracy: 0.921
- training iterations - Test Accuracy: 0.9410000000000001
- hidden units - Test Accuracy: 0.923
- layers - Test Accuracy: 0.9
- activation functions - Test Accuracy: 0.921
- learning rate - Test Accuracy: 0.915

As we can see from the result above, when we increase the training iterations and the number of hidden units on the layers will result in the improvement on testing accuracy, which are 0.941 and 0.923 respectively.

Part (c) -- 4 pt

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

Answer:

I will use the model hyperparameters from part (b), since we are interested in how well the model can used to predict the unknown-result data sets, which is reflected by the testing accuracy. So, having a better accuracy among testing data sets reflects on the model can predict the unseen data sets result/label the better. We want to train the machine to learn from the training data sets to predict for the unknown-result onces. So, by having a better testing accuracy is more important.

To be noticed is that, the training accuracy is also important, but having a very high training accuracy may also reflect that the model is over-fitting to the training datas. So, by both comparing the training accuracy and testing accuracy, in this case, trainig accuracy might be very high but results in a lower testing accuracy compares to others. This is the case we do not want to be in. So, we should choose model hyperparameters from part b as better testing accuracy for better prediction on results for unseen data sets .