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://drive.google.com/file/d/1Nc_BaNZ9wdNuFGpDJUxf6FKqCTiB69p8/view?usp=sharing (https://drive.google.com/file/d/1Nc_BaNZ9wdNuFGpDJUxf6FKqCTiB69p8/view?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 [1]: def sum_of_cubes(n):
            """Return the sum (1^3 + 2^3 + 3^3 + ... + n^3)

            Precondition: n > 0, type(n) == int

            >>> sum_of_cubes(3)
            36
            >>> sum_of_cubes(1)
            1
            """

            if (type(n) != int) or (n < 0):
              print("Invalid input")
              return -1

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

        print("sum_of_cubes(\"string\") =", sum_of_cubes("string"))
        print()
        print("sum_of_cubes(-2) =", sum_of_cubes(-2))
        print()
        print("sum_of_cubes(3) =", sum_of_cubes(3))

        Invalid input
        sum_of_cubes("string") = -1

        Invalid input
        sum_of_cubes(-2) = -1

        sum_of_cubes(3) = 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 (https://docs.python.org/3.6/library/stdtypes.html#str.split)

```
In [2]: 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 [3]: 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(part) for part in sentence.split()]

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

```
word_lengths("welcome to APS360!") = [7, 2, 7]
word_lengths("machine learning is so cool") = [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 [4]: 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
            """

            return len(set(word_lengths(sentence))) == 1

        print("all_same_length(\"all same length\") =", all_same_length("all same length"))
        print("all_same_length(\"hello world\") =", all_same_length("hello world"))
```

```
all_same_length("all same length") = False
all_same_length("hello world") = 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 [5]: 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.

```
In [6]: matrix = np.array([[1., 2., 3., 0.5],
                           [4., 5., 0., 0.],
                           [-1., -2., 1., 1.]])
        vector = np.array([2., 0., 1., -2.])
```

```
In [7]: # This counts the total number of elements in the matrix.
        matrix.size
```

```
Out[7]: 12
```

```
In [8]: # This shows the dimensions of the matrix, i.e. how many row and column vectors
        # there are. The product of these numbers will equal matrix.size!
        matrix.shape
```

```
Out[8]: (3, 4)
```

```
In [9]: # This counts the total number of elements in the vector.
        vector.size
```

```
Out[9]: 4
```

```
In [10]: # This also returns the dimensions of the vector, but since a vector is always
         # defined as a 1-dimensional matrix, the dimensions in which the vector is
         # restricted in will have an empty entry in vector.shape.
         vector.shape
```

```
Out[10]: (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 [11]: output = np.array([
           sum([
             matrix[i][j] * vector[j]

             for j in range(matrix.shape[1])
           ])

           for i in range(matrix.shape[0])
         ])
```

```
In [12]: output
```

```
Out[12]: 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 [13]: output2 = np.dot(matrix, vector)
```

```
In [14]: output2
```

```
Out[14]: array([ 4.,  8., -3.])
```

Part (d) -- 1pt

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

```
In [15]: all(np.equal(output, output2))
```

```
Out[15]: 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 [16]: 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[16]: 0.002379179000854492
```

```
In [17]:  # record the time before running code
          start_time = time.time()

          # place code to run here
          for i in range(10000):
              output = np.array([
                sum([
                    matrix[i][j] * vector[j]

                    for j in range(matrix.shape[1])
                ])

                for i in range(matrix.shape[0])
              ])

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

          # compute the difference
          diff1 = end_time - start_time

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

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

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

          # compute the difference
          diff2 = end_time - start_time

          print("Not using np.dot() =", diff1)
          print("Using np.dot() =", diff2)
```

```
Not using np.dot() = 0.11407232284545898
Using np.dot() = 0.011280298233032227
```

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 [18]:  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 [19]:  img = plt.imread("https://drive.google.com/uc?export=view&id=1oaLVR2hr1_qzpKQ47i9rVUIklwbDcews"
          )
```
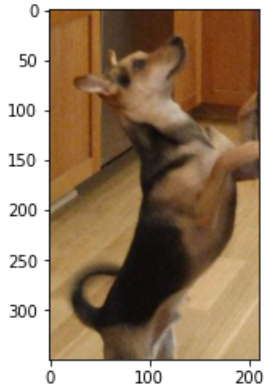
Part (b) -- 1pt

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 [20]: plt.imshow(img)
```

Out[20]: <matplotlib.image.AxesImage at 0x7f17b4eb1e48>

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 [21]: img_add = img + 0.25
         img_add = np.clip(img_add, 0, 1)
         plt.imshow(img_add)
```

Out[21]: <matplotlib.image.AxesImage at 0x7f17b35c65f8>
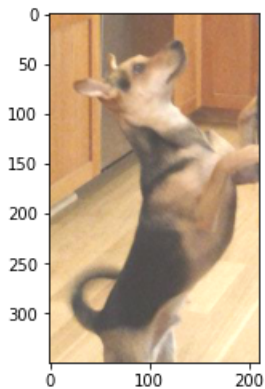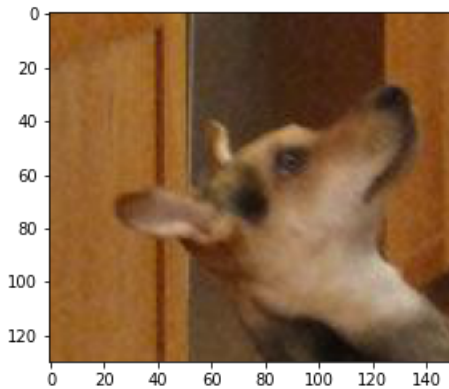

Part (d) -- 2pt

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 [22]:   img_cropped = img[:130, :150, :3]
           plt.imshow(img_cropped)
```

Out[22]:   <matplotlib.image.AxesImage at 0x7f17b35af048>

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 [23]:   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 [24]:   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 [25]:   img_torch.shape
```

Out[25]:   torch.Size([130, 150, 3])

Part (c) -- 1pt

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

```
In [26]:  # There are 58, 479 floating-point numbers in img_torch, assuming that numbers
          # such as 1.0 are assumed to be non floating-point numbers.

          # Otherwise, if a floating-point number is defined as being of torch.float32
          # type, then there are 58, 500 floating-point numbers in img_torch.
          img_torch.numel() - torch.sum((img_torch == img_torch.int()))

Out[26]:  tensor(58479)
```

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.

```
In [27]:  # The PyTorch transpose function swaps dimensions in a tensor. img_torch
          # represents a 130x150 matrix where each element is an RGB tuple, i.e. the RGB
          # information is grouped by the position of the pixels. #After transposing, it
          # will change from a shape of [130, 150, 3] to a shape of [3, 150, 130]. In this
          # new tensor, instead of the RGB information being grouped by the position of
          # the pixels, the position of the pixels are now grouped by the RGB information!
          # Furthermore, within the RGB groupings, the pixels positions are transposed
          # (flipped along the diagonal), and the matrices are 150x130 instead of 130x150.
          # The original variable img_torch is not updated; the transpose function
          # returns a brand new tensor.
          img_torch.transpose(0, 2)

Out[27]:  tensor([[[0.5882, 0.5412, 0.6157,  ..., 0.6039, 0.5882, 0.5804],
                  [0.5765, 0.5647, 0.6196,  ..., 0.6078, 0.6078, 0.6039],
                  [0.5569, 0.5961, 0.6196,  ..., 0.6118, 0.6196, 0.6235],
                  ...,
                  [0.5804, 0.5882, 0.5922,  ..., 0.3804, 0.3882, 0.4196],
                  [0.6039, 0.6078, 0.6157,  ..., 0.3765, 0.3804, 0.4039],
                  [0.6157, 0.6196, 0.6275,  ..., 0.3765, 0.3804, 0.3961]],

                 [[0.3725, 0.3216, 0.3765,  ..., 0.3882, 0.3725, 0.3647],
                  [0.3608, 0.3451, 0.3843,  ..., 0.3922, 0.3922, 0.3882],
                  [0.3412, 0.3765, 0.3843,  ..., 0.3961, 0.4039, 0.4078],
                  ...,
                  [0.3412, 0.3490, 0.3529,  ..., 0.3098, 0.3176, 0.3373],
                  [0.3647, 0.3686, 0.3765,  ..., 0.3059, 0.3098, 0.3216],
                  [0.3765, 0.3804, 0.3882,  ..., 0.3098, 0.3098, 0.3137]],

                 [[0.1490, 0.0902, 0.1529,  ..., 0.1686, 0.1529, 0.1451],
                  [0.1373, 0.1137, 0.1490,  ..., 0.1686, 0.1725, 0.1686],
                  [0.1176, 0.1451, 0.1412,  ..., 0.1725, 0.1804, 0.1882],
                  ...,
                  [0.1294, 0.1373, 0.1373,  ..., 0.2157, 0.2314, 0.2549],
                  [0.1529, 0.1569, 0.1608,  ..., 0.2118, 0.2157, 0.2392],
                  [0.1647, 0.1686, 0.1725,  ..., 0.2078, 0.2157, 0.2314]]])
```

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.

```
In [28]:  # The PyTorch unsqueeze function adds a new dimension of 1 at the specified
          # position in the shape of the tensor. Before unsqueezing at 0, the shape of
          # img_torch is [130, 150, 3], but after unsqueezing at 0, the shape of img_torch
          # becomes [1, 130, 150, 3]. The original variable img_torch is not updated; the
          # unsqueeze function returns a brand new tensor.
          img_torch.unsqueeze(0)

Out[28]:  tensor([[[[0.5882, 0.3725, 0.1490],
                    [0.5765, 0.3608, 0.1373],
                    [0.5569, 0.3412, 0.1176],
                    ...,
                    [0.5804, 0.3412, 0.1294],
                    [0.6039, 0.3647, 0.1529],
                    [0.6157, 0.3765, 0.1647]],

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

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

                   ...,

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

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

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

Part (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 [29]:  # The max values along each colour channel of img_torch are 0.8941, 0.7882, and
          # 0.6745 for the red, green and blue respectively.
          torch.max(torch.flatten(img_torch.transpose(0, 2), 1), 1)
```

Out[29]: torch.return_types.max(values=tensor([0.8941, 0.7882, 0.6745]), indices=tensor([16475, 16475,
         16475]))

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

```python
In [30]: 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=0.005, 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 parameter)
               optimizer.step()                     # 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
         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
```

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?

```
In [31]:  # Increasing the number of epochs or iterations increased the training accuracy
          # by the most. Increasing from 1 to 10 increased the accuracy by 3.5%, for the
          # highest accuracy of 99.9%.

          # Increasing the number of hidden units only marginally increased the training
          # accuracy; increasing from 30 to 200 only increased the accuracy of 0.8%.

          # I could not make any improvements by adding an additional hidden layer, or
          # changing the type of activation function or learning rate.
```

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?

```
In [32]:  # Increasing the number of epochs or iterations increased the test accuracy by
          # the most. Increasing from 1 to 10 increased the accuracy by 2%, for the
          # highest accuracy of 94.1%.

          # Increasing the number of hidden units only marginally increased the test
          # accuracy; increasing from 30 to 200 only increased the accuracy of 0.6%.

          # I could not make any improvements by adding an additional hidden layer, or
          # changing the type of activation function or learning rate.
```

Part (c) -- 4 pt

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

```
In [33]:  # The model hyperparameters from b) should be used. The training set should only
          # be used to tune the weights, or the "learnable" parameters. The testing set
          # (or validation set) should be used to tune the hyperparameters. The testing
          # set is used to measure the accuracy of the model with respect to unobserved
          # data, and raising this accuracy is far more important than raising the
          # accuracy against the training set.
```