

# 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:

## Part 1. Python Basics [3 pt]

The purpose of this section is to get you used to the basics of Python, including working with functions, numbers, lists, and strings.

Note that we **will** be checking your code for clarity and efficiency.

If you have trouble with this part of the assignment, please review <http://cs231n.github.io/python-numpy-tutorial/>

### Part (a) -- 1pt

Write a function `sum_of_cubes` that computes the sum of cubes up to `n`. If the input to `sum_of_cubes` invalid (e.g. negative or non-integer `n`), the function should print out `"Invalid input"` and return `-1`.

```
In [1]: import numpy as np
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(n < 0 or (not isinstance(n, int))):
    return -1
sum = 0
for i in range(n):
    sum += np.pow(i, 3)
return sum

```

```

In [2]: print(sum_of_cubes(3.5))
print(sum_of_cubes(3))
print(sum_of_cubes(4))
print(sum_of_cubes(-88))

```

```

-1
9
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 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>

```

In [3]: help(str.split)

```

Help on method\_descriptor:

```

split(self, /, sep=None, maxsplit=-1) unbound builtins.str method
    Return a list of the substrings in the string, using sep as the separator string.

```

```

sep
    The separator used to split the string.

    When set to None (the default value), will split on any whitespace
    character (including \n \r \t \f and spaces) and will discard
    empty strings from the result.
maxsplit
    Maximum number of splits.
    -1 (the default value) means no limit.

```

Splitting starts at the front of the string and works to the end.

Note, `str.split()` is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

```

In [4]: 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]
        """
        words = sentence.split(' ')
        lengths = []
        for word in words:
            lengths.append(len(word))
        return lengths

```

```

In [5]: 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 [6]: 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
        """
        words = sentence.split(' ')
        n = len(words)
        if n <= 1:
            return False
        else:
            last = len(words[0])
            for i in range(1, n):
                current = len(words[i])
                if current != last:
                    return False
            last = current
            return True
```

```
In [7]: print(all_same_length('hello world'))
        print(all_same_length('hellooo world'))
        print(all_same_length('hello world!'))
        print(all_same_length('hello!!'))
```

```
True
False
False
False
```

## 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`.

```
In [8]: 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 [9]: matrix = np.array([[1., 2., 3., 0.5],
                           [4., 5., 0., 0.],
                           [-1., -2., 1., 1.]])
        vector = np.array([2., 0., 1., -2.])
```

```
In [10]: matrix.size
```

```
Out[10]: 12
```

```
In [11]: matrix.shape
```

```
Out[11]: (3, 4)
```

```
In [12]: vector.size
```

```
Out[12]: 4
```

```
In [13]: vector.shape
```

```
Out[13]: (4,)
```

My Answer for Part (a): `<NumpyArray>.size` represents the total number of elements in the array/matrix, while `<NumpyArray>.shape` represents the number of row and column of the matrix.

## 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 [14]: n, m = matrix.shape

output = np.empty(n, dtype=float)

for i in range(n):
    sum = np.float64(0.0)
    for j in range(m):
        sum += np.multiply(matrix[i, j], vector[j])
    output[i] = sum
```

```
In [15]: output
```

```
Out[15]: 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 [16]: output2 = np.dot(matrix, vector)
```

```
In [17]: output2
```

```
Out[17]: array([ 4.,  8., -3.])
```

## Part (d) -- 1pt

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

```
In [18]: print(output == output2)
```

```
[ True  True  True]
```

## Part (e) -- 1pt

Show that using `np.dot` is faster than using your code from part (c).

You may find the below code snippet helpful:

```
In [19]: import time
import numpy as np

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

def for_method():
    n, m = matrix.shape
```

```

output = np.empty(n, dtype=float)
for i in range(n):
    sum = np.float64(0.0)
    for j in range(m):
        sum += np.multiply(matrix[i, j], vector[j])
    output[i] = sum

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

# place code to run here
for temp in range(100000):
    for_method()

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

# compute the difference
diff_for_method = end_time - start_time

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

# place code to run here
for temp in range(100000):
    output2 = np.dot(matrix, vector)

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

# compute the difference
diff_np_method = end_time - start_time

print(diff_for_method)
print(diff_np_method)

```

```

1.0323460102081299
0.0495600700378418

```

## 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.

```
In [20]: import matplotlib.pyplot as plt
```

### Part (a) -- 1 pt

This is a photograph of a dog whose name is Mochi.

 alt text

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 [21]: img = plt.imread("part3.png")
```

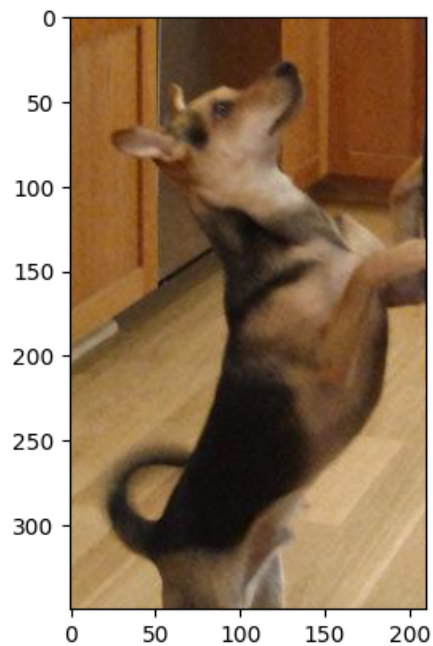
### Part (b) -- 1pt

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

```
Out[22]: <matplotlib.image.AxesImage at 0x1d13d272540>
```

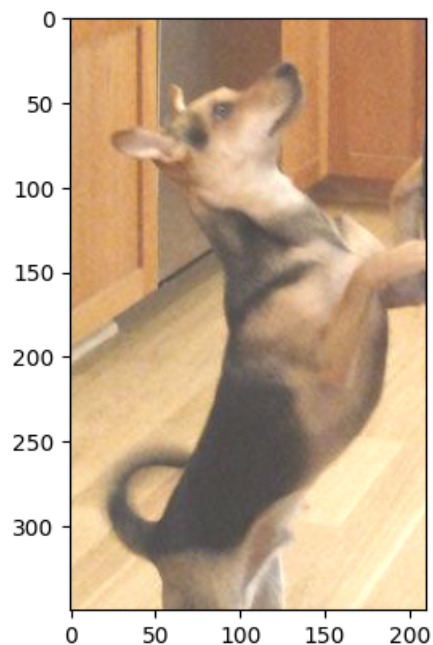


### Part (c) -- 2pt

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

```
Out[23]: <matplotlib.image.AxesImage at 0x1d13e563e90>
```



### 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 [24]: # img_cropped = np.zeros(130, 150)
x = 8
y = 9
x_plus_width = x + 130
y_plus_height = y + 150

img_cropped = img[x:x_plus_width, y:y_plus_height]
temp = img_cropped[:, :, :3]
img_cropped = temp

plt.imshow(img_cropped)
```

Out[24]: <matplotlib.image.AxesImage at 0x1d13e5bbe90>



## 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 [25]: 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 [26]: 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 [27]: img_torch_size = img_torch.shape
img_torch_size
```

```
Out[27]: torch.Size([130, 150, 3])
```

## Part (c) -- 1pt

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

```
In [28]: img_torch_size[0] * img_torch_size[1] * img_torch_size[2]
```

```
Out[28]: 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.

```
In [29]: transposed = img_torch.transpose(0,2)
transposed_shape = temp.shape
print(transposed)
(transposed_shape)
```

```
tensor([[ [0.5804, 0.5725, 0.5608, ..., 0.5882, 0.5882, 0.5882],
          [0.5765, 0.5647, 0.5529, ..., 0.5961, 0.5922, 0.5922],
          [0.5804, 0.5647, 0.5569, ..., 0.6000, 0.5961, 0.5961],
          ...,
          [0.4627, 0.4745, 0.4824, ..., 0.4627, 0.5412, 0.5608],
          [0.4863, 0.5020, 0.5098, ..., 0.4314, 0.5059, 0.5490],
          [0.5098, 0.5333, 0.5373, ..., 0.3922, 0.4510, 0.5176]],

         [[0.3569, 0.3529, 0.3529, ..., 0.3725, 0.3686, 0.3686],
          [0.3529, 0.3412, 0.3412, ..., 0.3804, 0.3725, 0.3725],
          [0.3529, 0.3412, 0.3412, ..., 0.3804, 0.3765, 0.3765],
          ...,
          [0.2078, 0.2196, 0.2275, ..., 0.3961, 0.4745, 0.4941],
          [0.2196, 0.2353, 0.2431, ..., 0.3569, 0.4314, 0.4745],
          [0.2353, 0.2588, 0.2627, ..., 0.3059, 0.3725, 0.4392]],

         [[0.0784, 0.0627, 0.0549, ..., 0.1608, 0.1686, 0.1765],
          [0.0824, 0.0627, 0.0588, ..., 0.1686, 0.1725, 0.1804],
          [0.1020, 0.0784, 0.0784, ..., 0.1804, 0.1843, 0.1922],
          ...,
          [0.0588, 0.0706, 0.0784, ..., 0.3333, 0.4118, 0.4314],
          [0.0745, 0.0902, 0.0980, ..., 0.3020, 0.3765, 0.4196],
          [0.0941, 0.1176, 0.1216, ..., 0.2549, 0.3294, 0.3961]]])
```

```
Out[29]: (130, 150, 3)
```

```
In [30]: img_torch.shape
```

```
Out[30]: torch.Size([130, 150, 3])
```

My answer: It will return a `[3, 150, 130]` array, the original variable `img_torch` is not updated.

## 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 [31]: unsqueezed = img_torch.unsqueeze(0)
unsqueezed_shape = unsqueezed.shape
print(unsqueezed)
print(unsqueezed_shape)
```



```

tensor([[[[0.5804, 0.3569, 0.0784],
          [0.5765, 0.3529, 0.0824],
          [0.5804, 0.3529, 0.1020],
          ...,
          [0.4627, 0.2078, 0.0588],
          [0.4863, 0.2196, 0.0745],
          [0.5098, 0.2353, 0.0941]],

         [[0.5725, 0.3529, 0.0627],
          [0.5647, 0.3412, 0.0627],
          [0.5647, 0.3412, 0.0784],
          ...,
          [0.4745, 0.2196, 0.0706],
          [0.5020, 0.2353, 0.0902],
          [0.5333, 0.2588, 0.1176]],

         [[0.5608, 0.3529, 0.0549],
          [0.5529, 0.3412, 0.0588],
          [0.5569, 0.3412, 0.0784],
          ...,
          [0.4824, 0.2275, 0.0784],
          [0.5098, 0.2431, 0.0980],
          [0.5373, 0.2627, 0.1216]],

         ...,

         [[0.5882, 0.3725, 0.1608],
          [0.5961, 0.3804, 0.1686],
          [0.6000, 0.3804, 0.1804],
          ...,
          [0.4627, 0.3961, 0.3333],
          [0.4314, 0.3569, 0.3020],
          [0.3922, 0.3059, 0.2549]],

         [[0.5882, 0.3686, 0.1686],
          [0.5922, 0.3725, 0.1725],
          [0.5961, 0.3765, 0.1843],
          ...,
          [0.5412, 0.4745, 0.4118],
          [0.5059, 0.4314, 0.3765],
          [0.4510, 0.3725, 0.3294]],

         [[0.5882, 0.3686, 0.1765],
          [0.5922, 0.3725, 0.1804],
          [0.5961, 0.3765, 0.1922],
          ...,
          [0.5608, 0.4941, 0.4314],
          [0.5490, 0.4745, 0.4196],
          [0.5176, 0.4392, 0.3961]]]])])
torch.Size([1, 130, 150, 3])

```

In [32]: `img_torch.shape`

Out[32]: `torch.Size([130, 150, 3])`

My answer: it returns a `[1, 130, 150, 3]` array, it added a dimension, which might be batch size 1, represent this is picture 1, it still don't change the shape of original `img_torch`.

## 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 [33]: max_vals = img_torch.max(dim=0)[0].max(dim=0)[0]
         print(max_vals)

```

```

tensor([0.8941, 0.7882, 0.6745])

```

## Part 5. Training an ANN [10 pt]

The sample code provided below is a 2-layer ANN trained on the MNIST dataset to identify digits less than 3 or greater than and equal to 3. Modify the code by changing any of the following and observe how the accuracy and error are affected:

- number of training iterations
- number of hidden units
- numbers of layers
- types of activation functions
- learning rate

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.

```
In [34]: 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, 120)
        self.layer2 = nn.Linear(120, 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 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 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: 0.0  
 Training Accuracy: 1.0  
 Test Error Rate: 0.057  
 Test Accuracy: 0.943

## My Experiment 1

I tried to modify this line

```
optimizer = optim.SGD(pigeon.parameters(), lr=0.005, momentum=0.9)
```

I'll change the variable `lr`, here's my table

Learning Rate	Training Accuracy	Test Accuracy
0.001	0.922	0.887
0.002	0.956	0.897
0.005	0.964	0.921
0.01	0.961	0.918
0.1	0.688	0.703

This means `lr = 0.005` might be a good value, higher and lower values might have worse result.

## My Experiment 2

I also tried to modify these lines

```
self.layer1 = nn.Linear(28 * 28, 30)
self.layer2 = nn.Linear(30, 1)
```

I'll change these `30`, which is the number of hidden units here's my table

The # of Hidden Units	Training Accuracy	Test Accuracy
10	0.953	0.896
30	0.964	0.921
60	0.969	0.920
120	0.973	0.928
240	0.972	0.928

This means higher value will get better accuracy.

## My Experiment 3

I also tried to add this line

```
for i in range(5):
    before
```

```
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
```

I'll change the number of training iterations, and here's the result

Training Iterations	Training Accuracy	Test Accuracy
1	0.964	0.921
3	0.986	0.931
5	0.989	0.934
10	0.999	0.941

This means higher Training Iterations will get better accuracy.

## Combine EX 3 and EX 2

I set Training Iterations to 10, and Hidden Unit to 120, and I got

```
Training Error Rate: 0.0
Training Accuracy: 1.0
Test Error Rate: 0.057
Test Accuracy: 0.943
```

This is the best result I got.

### 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?

My answer: Both Training Iterations and Hidden Units will result in better training accuracy. The maximum training accuracy I got is 1.0 when setting.

### 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?

My Answer: Both Training Iterations and Hidden Units will result in better testing accuracy. The maximum testing accuracy I got is 0.943.

### Part (c) -- 4 pt

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

My answer: From part (b), because I train this model to predict the data it has never seen before, rather than memorize data.