Lab 1. PyTorch and ANNs

Deadline: Friday May 14th, 11:59pm.

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://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/14lKBku4IenpGADHgFtxEef8082Y_ijnj/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/

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
    """
    if type(n) != int:
      print("Invalid input")
      return -1
    if n < 0:
      print("Invalid input")
      return -1
    if n == 0:
      return 0
    return n**3 + sum_of_cubes(n-1)
```

In [ ]:
```
sum_of_cubes(3)
```

Out[ ]:
```
36
```

In [ ]:
```
sum_of_cubes(1)
```

```
Out[ ]: 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

```
In [ ]: help(str.split)
```

```
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]
            """
            words = sentence.split()
            return [len(word) for word in words]
```

```
In [ ]: word_lengths("welcome to APS360!")
```

```
Out[ ]: [7, 2, 7]
```

```
In [ ]: word_lengths("machine learning is so cool")
```

```
Out[ ]: [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
            >>> word_lengths("hello world")
            True
            """
            l = word_lengths(sentence)
            return l.count(l[0]) == len(l)
```

```
In [ ]: all_same_length("all same length")
```

```
Out[ ]: False
```

```
In [ ]: all_same_length("hello world")
```

```
Out[ ]: 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.

```
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,)
```

Answer:

`<NumpyArray>.size` represents the number of elements in the `<NumpyArray>` .

`<NumpyArray>.shape` represents the dimensions of the `<NumpyArray>` .

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

In [ ]:
```
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(b), 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 = np.dot(matrix, vector)
```

In [ ]:
```
output2
```

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

Part (d) -- 1pt

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

In [ ]:
```
np.array_equal(output, output2)
```

Out[ ]:
```
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[ ]:
```
0.001956939697265625
```

In [ ]:
```
start_time = time.time()
output = []
for i in range(len(matrix)):
  row = 0
  for j in range(len(matrix[i])):
    row += matrix[i][j] * vector[j]
  output.append(row)
output = np.array(output)
end_time = time.time()
part_c_time = end_time - start_time

start_time = time.time()
output2 = np.dot(matrix, vector)
end_time = time.time()
part_d_time = end_time - start_time
```

In [ ]:
```
part_c_time
```

Out[ ]:
```
0.00025272369384765625
```

In [ ]:
```
part_d_time
```

Out[ ]:
```
7.05718994140625e-05
```

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) 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_qzpKQ47i9rVUIkl
```

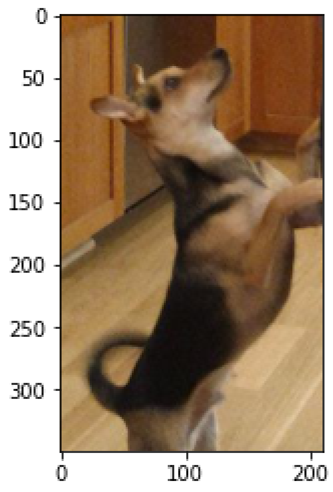
Part (b) -- 1pt

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 0x7fa135561110>
```

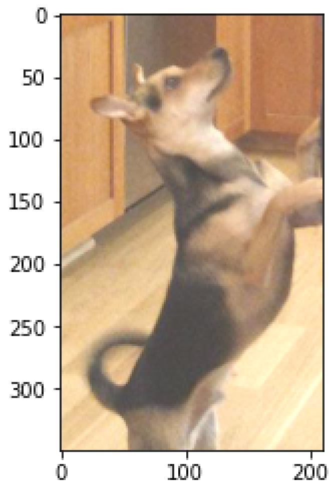


Part (c) -- 2pt

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

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7fa1354fad90>
```



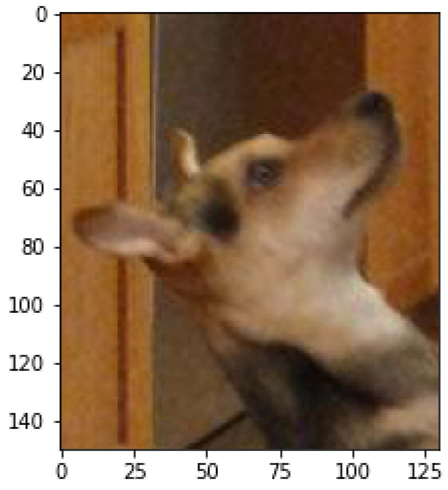
Part (d) -- 2pt

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 [ ]:
```
img_cropped = img[0:150, 20:150, 0:3]
plt.imshow(img_cropped)
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7fa1354655d0>
```



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

Part (c) -- 1pt

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

In [ ]:
```
img_torch.numel()
```

```
Out[ ]: 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 [ ]: img_torch.transpose(0,2).shape
```

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

Answer:

`img_torch.transpose(0,2)` computes the transpose of the `img_torch` tensor by swapping the input dimensions 0 and 2. This can be observed through differences in the `shape` of input and output tensors. The `shape` of `img_torch` is `[150, 130, 3]` and the `shape` of `img_torch.transpose(0,2)` is `[3, 130, 150]`, where dimensions 0 and 2 are indeed swapped after the transpose.

The expression returns the tranposed version of `img_torch` as a tensor.

The original variable `img_torch` is unchanged with the usage of this expression, but the resulting `img_torch.transpose(0,2)` tensor shares its underlying storage with `img_torch`. This means that if additional changes are made to the contents of `img_torch.transpose(0,2)`, the contents of `img_torch` will change accordingly.

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 [ ]: img_torch.unsqueeze(0).shape
```

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

```
In [ ]: img_torch[0, :, 0].shape
```

```
Out[ ]: torch.Size([130])
```

Answer:

`img_torch.unsqueeze(0)` inserts a dimension of size 1 to the `img_torch` tensor at the specified position of 0. This can be observed through differences in the `shape` of input and output tensors. The `shape` of `img_torch` is `[150, 130, 3]` and the `shape` of `img_torch.unsqueeze(0)` is `[1, 150, 130, 3]`, where a new dimension of size 1 is indeed inserted at the 0th index.

The expression returns the modified version of `img_torch` as a tensor.

The original variable `img_torch` is unchanged with the usage of this expression, but the resulting `img_torch.unsqueeze(0)` tensor shares its underlying storage with `img_torch`. This means that if additional changes are made to the contents of `img_torch.unsqueeze(0)`, the contents of `img_torch` will change accordingly.

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 [ ]:
```
for i in range(img_torch.shape[-1]):
  print(torch.max(img_torch[:, :, i]))
```

```
tensor(0.8941)
tensor(0.7882)
tensor(0.6745)
```

In [ ]:
```
torch.max(torch.max(img_torch, 0)[0], 0)[0]
```

Out[ ]:
```
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

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" task
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 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 [ ]:

```
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)

iterations = 30
for i in range(iterations):
  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 paramet
      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: 0.059
Test Accuracy: 0.9410000000000001
```

Answer:

As shown in the code above, by simply increasing the number of training iterations from showing each image only once to 30 times, the best accuracy on the training data is achieved. The resulting training accuracy is 100%.

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 [ ]:
```
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, 350)
        self.layer2 = nn.Linear(350, 180)
        self.layer3 = nn.Linear(180, 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" task
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.SGD(pigeon.parameters(), lr=0.005, momentum=0.9)
```

```
iterations = 30
for i in range(iterations):
  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 paramet
      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: 0.056
Test Accuracy: 0.944
```

Answer:

As shown in the code above, on top of increasing the number of training iterations from (a), adding an additional hidden layer and increasing the number of hidden nodes produced the best accuracy on the testing data. The resulting test accuracy is 94.4%.

Part (c) -- 4 pt

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

Answer:

While it is tempting to use the model hyperparameters from (b) since it produces a higher test accuracy, the hyperparameters from (a) should be utilized. This is due to the fact that the hyperparameters from (b) are tuned using the test set, which defeats the purpose of the training/testing dataset split. This also leads to overfitting the model to the test data.