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 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://colab.research.google.com/drive/1vi3OboZjb7vV-xC7SFtkUdjFerH3RKVd?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`.

```
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
    else:
      sum = 0
      for i in range (1, n+1):
        sum += i**3
    return sum

sum_of_cubes(2)
```

```
9
```

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

```
help(str.split)
```

Show hidden output

```
def word_lengths(sentence):
    """Return a list containing the length of each word in
    sentence.

    >>> word_lengths("welcome to APS360!")
    [7, 2, 7]
    >>> word_lengths("machine learning is so cool")
    [7, 8, 2, 2, 4]
    """
    if type(sentence) != str:
      print("Invalid input")
      return -1
    Wlenth = []
    for i in sentence.split():
        lens = len(i)  #i is word
        Wlenth.append(lens)
    return  Wlenth

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.

```
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
    """
    if type(sentence) != str:
      print("Invalid input")
      return -1
    combine = set(word_lengths(sentence))
    return len(combine) == 1

all_same_length("fdf 333")
```

```
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`.

```
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.

Double-click (or enter) to edit

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

(NumpyArray).size represents the total number of elements in the Numpy array

(NumpyArray).shape represents the dimensions of the array (ie. how many rows/columns)

```
matrix.size
```

```
12
```

For example, here matrix.size returned 12 because there are 12 elements in matrix (return the total number of elementz in matrix).

```
matrix.shape
```

```
(3, 4)
```

For example, here matrix.shape returned (3, 4) because the matrix is 4x3 a matrix (return the dimensions of matrix).

```
vector.size
```

```
4
```

For example, here vector.size returned 4 because there are 4 elements in vector

```
vector.shape
```

```
(4,)
```

And lastly, here vector.shape returned (4,) because vector is one-dimensional

Part (b) -- 1pt

Perform matrix multiplication `output = matrix x vector` by using for loops to iterate through the columns and rows. Do not use any builtin NumPy functions. Cast your output into a NumPy array, if it isn't one already.

Hint: be mindful of the dimension of output

```
output = None

def matrix_multiplication(matrix, vector):
  ret = []
  x = 0
  for i in range(0, matrix.shape[0]):
    ret.append(0)  #for index's sake  (to match); should it be 0. instead to match the format?
    for j in range(0, matrix.shape[1]):
      ret[i] += matrix[i, j] * vector[j]
```

```
    x = 0
  ret = np.array(ret)
  return ret
```

```
output = matrix_multiplication(matrix, vector)
print(output)
```

```
[ 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.

```
output2 = None
```

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

```
[ 4.  8. -3.]
```

Part (d) -- 1pt

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

```
if np.array_equal(output, output2):  # to avoid for loops in our code as mentioned in d; use build in fuctions instead
    print(True)
else:
    print(False)
```

```
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:

```
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
print(diff)
```

```
0.0005261898040771484
```

```
def measure_runtime(func, input1, input2):
    start_time = time.time()
    func(input1, input2)
    end_time = time.time()
    return (end_time - start_time)

a= measure_runtime(matrix_multiplication, matrix, vector)
measure_runtime(matrix_multiplication, matrix, vector)
```

```
1.7881393432617188e-05
```

```
b=measure_runtime(np.dot, matrix, vector)
measure_runtime(np.dot, matrix, vector)
```

```
1.2636184692382812e-05
```

```
if (b < a):
    print(True)
else:
    print(False)
```

```
True
```

Part 3. Images [6 pt]

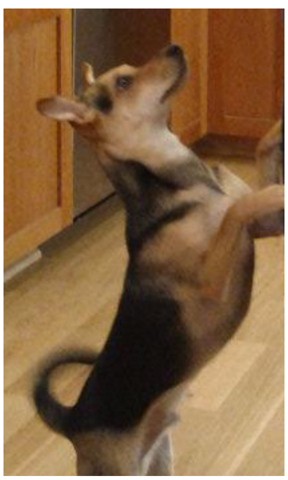
A picture or image can be represented as a NumPy array of "pixels", with dimensions H × W × C, where H is the height of the image, W is the width of the image, and C is the number of colour channels. Typically we will use an image with channels that give the the Red, Green, and Blue "level" of each pixel, 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.

```
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.

```
#img = plt.imread('https://drive.google.com/uc?export=view&id=1oaLVR2hr1_qzpKQ47i9rVUIklwbDcews')
import requests
from io import BytesIO

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

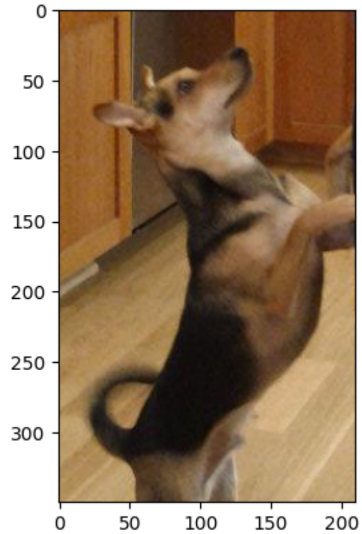
Part (b) -- 1pt

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

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

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

<matplotlib.image.AxesImage at 0x79208cceeb00>

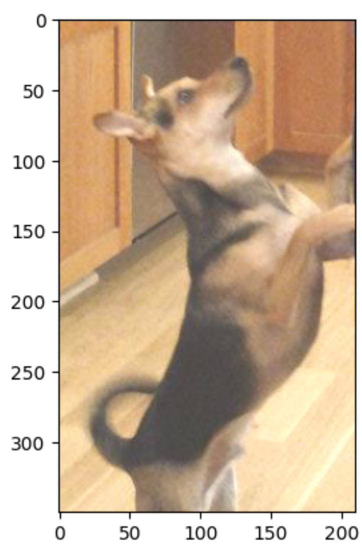


Part (c) -- 2pt

Modify the image by adding a constant value of 0.25 to each pixel in the `img` and store the result in the variable `img_add`. Note that, since the range for the pixels needs to be between [0, 1], you will also need to clip img_add to be in the range [0, 1] using `numpy.clip`. Clipping sets any value that is outside of the desired range to the closest endpoint. Display the image using `plt.imshow`.

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

<matplotlib.image.AxesImage at 0x79208bd1a1a0>



Part (d) -- 2pt

Crop the **original** image (`img` variable) to a 130 x 150 image including Mochi's face. Discard the alpha colour channel (i.e. resulting `img_cropped` should **only have RGB channels**)

Display the image.

```
img_cropped = img[20:150, 20:170, 0:3]
plt.imshow(img_cropped)
```

```
<matplotlib.image.AxesImage at 0x79208a2c23b0>
```

Part 4. Basics of PyTorch [6 pt]

PyTorch is a Python-based neural networks package. Along with tensorflow, PyTorch is currently one of the most popular machine learning libraries.

PyTorch, at its core, is similar to Numpy in a sense that they both try to make it easier to write codes for scientific computing achieve improved performance over vanilla Python by leveraging highly optimized C back-end. However, compare to Numpy, PyTorch offers much better GPU support and provides many high-level features for machine learning. Technically, Numpy can be used to perform almost every thing PyTorch does. However, Numpy would be a lot slower than PyTorch, especially with CUDA GPU, and it would take more effort to write machine learning related code compared to using PyTorch.

```
import torch
```

Part (a) -- 1 pt

Use the function `torch.from_numpy` to convert the numpy array `img_cropped` into a PyTorch tensor. Save the result in a variable called `img_torch`.

```
img_torch = torch.from_numpy(img_cropped)
torch.from_numpy(img_cropped)
```

```
tensor([[[0.6353, 0.4353, 0.2275],
         [0.6431, 0.4431, 0.2353],
         [0.6510, 0.4510, 0.2431],
         ...,
         [0.4627, 0.2157, 0.0471],
         [0.4784, 0.2235, 0.0667],
         [0.5059, 0.2510, 0.0941]],

        [[0.6392, 0.4392, 0.2314],
         [0.6392, 0.4353, 0.2392],
         [0.6353, 0.4314, 0.2353],
         ...,
         [0.4784, 0.2314, 0.0627],
         [0.5098, 0.2549, 0.0980],
         [0.5176, 0.2627, 0.1059]],

        [[0.6392, 0.4392, 0.2314],
         [0.6314, 0.4275, 0.2314],
         [0.6235, 0.4196, 0.2235],
         ...,
         [0.4941, 0.2471, 0.0784],
         [0.5137, 0.2588, 0.1020],
         [0.5098, 0.2549, 0.0980]],

        ...,

        [[0.5961, 0.3765, 0.1765],
```

```
         [0.5804, 0.3608, 0.1608],
         [0.5961, 0.3765, 0.1843],
         ...,
         [0.7529, 0.6118, 0.5255],
         [0.7333, 0.5647, 0.4275],
         [0.7059, 0.5373, 0.4000]],

        [[0.6078, 0.3882, 0.1882],
         [0.6000, 0.3804, 0.1804],
         [0.6078, 0.3882, 0.1961],
         ...,
         [0.7490, 0.6000, 0.5176],
         [0.6667, 0.4941, 0.3412],
         [0.6314, 0.4588, 0.3059]],

        [[0.6118, 0.3922, 0.1922],
         [0.6000, 0.3804, 0.1804],
         [0.6039, 0.3843, 0.1922],
         ...,
         [0.6667, 0.5176, 0.4353],
         [0.6118, 0.4353, 0.2745],
         [0.5882, 0.4118, 0.2510]]])
```

Part (b) -- 1pt

Use the method `<Tensor>.shape` to find the shape (dimension and size) of `img_torch`.

```
img_torch.shape
```

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

Part (c) -- 1pt

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

```
nums = 1
for i in img_torch.shape:
    nums *= i
print(nums)
```

```
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.

```
img_torch.transpose(0,2)
#print(img_torch.shape)
#print(img_torch.transpose(0,2).shape)
```

```
tensor([[[0.6353, 0.6392, 0.6392,  ..., 0.5961, 0.6078, 0.6118],
         [0.6431, 0.6392, 0.6314,  ..., 0.5804, 0.6000, 0.6000],
         [0.6510, 0.6353, 0.6235,  ..., 0.5961, 0.6078, 0.6039],
         ...,
         [0.4627, 0.4784, 0.4941,  ..., 0.7529, 0.7490, 0.6667],
         [0.4784, 0.5098, 0.5137,  ..., 0.7333, 0.6667, 0.6118],
         [0.5059, 0.5176, 0.5098,  ..., 0.7059, 0.6314, 0.5882]],

        [[0.4353, 0.4392, 0.4392,  ..., 0.3765, 0.3882, 0.3922],
         [0.4431, 0.4353, 0.4275,  ..., 0.3608, 0.3804, 0.3804],
         [0.4510, 0.4314, 0.4196,  ..., 0.3765, 0.3882, 0.3843],
         ...,
         [0.2157, 0.2314, 0.2471,  ..., 0.6118, 0.6000, 0.5176],
         [0.2235, 0.2549, 0.2588,  ..., 0.5647, 0.4941, 0.4353],
         [0.2510, 0.2627, 0.2549,  ..., 0.5373, 0.4588, 0.4118]],

        [[0.2275, 0.2314, 0.2314,  ..., 0.1765, 0.1882, 0.1922],
         [0.2353, 0.2392, 0.2314,  ..., 0.1608, 0.1804, 0.1804],
         [0.2431, 0.2353, 0.2235,  ..., 0.1843, 0.1961, 0.1922],
         ...,
         [0.0471, 0.0627, 0.0784,  ..., 0.5255, 0.5176, 0.4353],
         [0.0667, 0.0980, 0.1020,  ..., 0.4275, 0.3412, 0.2745],
         [0.0941, 0.1059, 0.0980,  ..., 0.4000, 0.3059, 0.2510]]])
```

The img_torch.transpose(0,2) swaps the img_torch's values along the first dimension (index 0) with the third dimension (index 2). This operation changes the shape of the tensor but does not change data within it. The operation instead of changing the original tensor img_torch, creates creates a new one with swapped dimensions. In another word, img_torch.transpose(0, 2) returns a transposed view of img_torch, not affecting its original shape and content. However, because it is merely a view of the original tensor, sharing the same data as the original tensor, when data changed in img_torch will reflect in img_torch.transpose(0, 2) and vice versa. But, just the operation img_torch.transpose(0,2) will not let img_torch vriable 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.

```
img_torch.unsqueeze(0)
#img_torch.unsqueeze(0).shape
```

```
tensor([[[[0.6353, 0.4353, 0.2275],
          [0.6431, 0.4431, 0.2353],
          [0.6510, 0.4510, 0.2431],
          ...,
          [0.4627, 0.2157, 0.0471],
          [0.4784, 0.2235, 0.0667],
          [0.5059, 0.2510, 0.0941]],

         [[0.6392, 0.4392, 0.2314],
          [0.6392, 0.4353, 0.2392],
          [0.6353, 0.4314, 0.2353],
          ...,
          [0.4784, 0.2314, 0.0627],
          [0.5098, 0.2549, 0.0980],
          [0.5176, 0.2627, 0.1059]],

         [[0.6392, 0.4392, 0.2314],
          [0.6314, 0.4275, 0.2314],
          [0.6235, 0.4196, 0.2235],
          ...,
          [0.4941, 0.2471, 0.0784],
          [0.5137, 0.2588, 0.1020],
          [0.5098, 0.2549, 0.0980]],

         ...,

         [[0.5961, 0.3765, 0.1765],
          [0.5804, 0.3608, 0.1608],
          [0.5961, 0.3765, 0.1843],
          ...,
          [0.7529, 0.6118, 0.5255],
          [0.7333, 0.5647, 0.4275],
          [0.7059, 0.5373, 0.4000]],

         [[0.6078, 0.3882, 0.1882],
          [0.6000, 0.3804, 0.1804],
          [0.6078, 0.3882, 0.1961],
          ...,
          [0.7490, 0.6000, 0.5176],
          [0.6667, 0.4941, 0.3412],
          [0.6314, 0.4588, 0.3059]],

         [[0.6118, 0.3922, 0.1922],
          [0.6000, 0.3804, 0.1804],
          [0.6039, 0.3843, 0.1922],
          ...,
          [0.6667, 0.5176, 0.4353],
          [0.6118, 0.4353, 0.2745],
          [0.5882, 0.4118, 0.2510]]]])
```

img_torch.unsqueeze(0) adds a new dimension of size 1 at index 0, changing the shape(size) of img_torch to [1, 130, 150, 3]. Just like part d img_torch remains unchanged and unsqueeze() creates a new tensor with the added dimension. In this case, at index 0.

Part (f) -- 1 pt

Find the maximum value of `img_torch` along each colour channel? Your output should be a one-dimensional PyTorch tensor with exactly three values.

Hint: lookup the function `torch.max`.

```
max_vals = torch.max(torch.max(img_torch, 1)[0], 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.

```
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=0.015, momentum=0.9)

itteration = 1
for i in range(itteration):
  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: 0.129
Training Accuracy: 0.871
Test Error Rate: 0.148
Test Accuracy: 0.852
```

number of training iterations

Trial #	Trainng Iterations	Training Error Rate	Training Accuracy	Test Error Rate	Test Accuracy
0	2	0.016	0.984	0.057	0.943
1	5	0.011	0.989	0.066	0.934
2	8	0.001	0.999	0.062	0.938
3	10	0.001	0.999	0.059	0.941

number of Hidden Units

Trial #	Hidden UNits	Training Error Rate	Training Accuracy	Test Error Rate	Test Accuracy
0	15	0.038	0.962	0.087	0.913
1	30	0.036	0.964	0.079	0.921
2	100	0.03	0.97	0.077	0.923
3	120	0.027	0.973	0.072	0.928

learning rate

Trial #	Learning Rate Parmeter	Training Error Rate	Training Accuracy	Test Error Rate	Test Accuracy
0	0.001	0.078	0.922	0.113	0.887
1	0.005	0.036	0.964	0.079	0.921
2	0.01	0.039	0.961	0.082	0.918
3	0.015	0.129	0.871	0.148	0.852

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?

The most significant changes occured with increasing the number of iterations. When the number of interation increases, the accuracy increases as well and it increased the most out of the three options I choose. In the table above, it seems that the training accuracy converge on 0.999 (seem in both 8 and 10 iterations) and it the highest out of all the test cases.

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?

Like part a, the best results occur with increasing the number of iterations. With 10 iterations the testing accuracy achieved becomes 0.941.

When adding more hidden units, it seems to also improve performances but the end result of increasing it up to 120, resulting in a test accuracy 0.928 is still less than interations tests.

For increasing learning rate to a too high value, the model exhibits issues. The updates to the model's weights after each iteration are too large. This causes the model to overshoot the minimum in the loss function. In addition, instead of gradually approaching a good solution, it jumps around, and fail to settle at the minimum. Test accuracy in fact, decreased.

Part (c) -- 4 pt

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

I should use the model hyperparameters from (b) because testing data accuracy weighs more than training data accuracy. When using the hyperparameters that perform well in the training dataset, it has the risk of overfitting the model to the training data. In such a case, the model would perform great on the known training dataset but still poorly on the unseen testing one, similar to real-world application data. In contrast, if the dataset performs well on the testing dataset, it will likely perform better in actual applications when dealing with unknown new data not used and seen in training.