Homework 3 Report

Modified Network Code

Mai Ngo

DSC 578 Neural Network and Deep Learning - SEC 701

Dr. Noriko Tomuro

October 8, 2023

Table of Contents

Section 1 - Complete

The calculation from my evaluateEpoch function were matched with Check2 given output. I did not start the assignment right away with coding. First I went through neural network concept, then thoroughly break down the original code and rename variable to help me understand required modifications better which took me 2 days. For example, you can see I named the function evaluateAccuracy on original code and evaluateEpoch on modified code.

What I did was breaking down provided code within the for loop: **'testResults = [(np.argmax(self.feedForward(instanceInput)), np.argmax(targetOutput)) for (instanceInput, targetOutput) in testData]'**.

Understand that it originally was looking at indices of the target and actual output, while for modified code, I will need to look at the values of each output itself. Then just manually input formulas to calculate each loss types: MSE, CE, and LL. Furthermore, the original testResults of each epoch was stored as a list with each element as a tuple contains indices. I know this will be too extensive for 4 different kinds of losses (considered Accuracy and count can be done together), so I just set counters and keep adding the output accordingly for each observation (this was a standard approach in DSC 478). And I have the output stored as a dictionary for each epoch to make the code more efficient when the function being called in SGD( ).

Initially I did have the output stored as a list, then output retrieval print in SGD( ) was way too redundant. I did not have difficult time in making this modification at all given thoroughly understand original code.

Return output from modified evaluateEpoch function:

```
return {'Count': epochPred, 'Accuracy': epochPred/testNum, 'MSE': epochMSE / testNum,
        'CE': epochCE / testNum, 'LL': epochLL / testNum}
```

Section 2 - Complete

The printed output from my SGD function were matched with Check2 given output. For subpart a, this was pretty much printing the output from evaluateEpoch function. As I mentioned earlier, having output as a list originally made my modified code very long and redundant, not to mention list indices in print statement. With dictionary form, I just can call the keys itself which is already named as corresponding type of losses. I also use f-string because it is shorter and more efficient.

As for printing with and without test data, that was just based on the if-else statement. If **'not testData'**, then I just need to call function evaluationEpoch for training, and vice versa. It was not that difficult for me, since I 100% understand original code and know that when SGD( ) calls evaluateEpoch( ), it should pass the entire training and/or testing data.

For subpart b, the original code just keeps printing output from evaluate function. While for Check2, it also wants to print some specific evaluate output. Looking at Check2 code,

'**print ('** train: {}'.format(trainRes[-1]))**
print (' valid: {}'.format(testRes[-1]))'**

I can tell the accumulated output needs to be stored as lists. So originally, I have SGD( ) returns as 2 lists (train and test), the Check2 code output still goes through correctly. Since the lists just keep appending for every epoch, so even without testData, it will return as the empty initiate list.

But given nested list requirement, I was not sure how that works, so I tried on simple code like **x1, x2 = ['x1'], ['x2']]** and it works, so just simple change on the return statement. I actually did not know I can retrieve elements from a list in this way prior to this assignment.

Return output from SGD function:

return [summary_trainRes, summary_testRes if testData else []]

Section 3 – Complete

The printed output from my SGD function were matched with Check2 given output. The early stopping extension requirement was very straightforward from what you explained in class. This was not hard at all since I already know exactly how my evaluation outputs are being stored, and which data structure being used.

```
#Early exit if applies.
if trainRes['Accuracy'] == 1.0:
    break
elif (epoch >=3 and trainRes['MSE'] >= max (summary_trainRes[epoch-1]['MSE'],
                                            summary_trainRes[epoch-2]['MSE'],
                                            summary_trainRes[epoch-3]['MSE'])):
    break
```

My **'trainRes'** is the output from evaluateEpoch function, which is already in dictionary form. Since I have dictionary keys as the loss function names itself, it was very convenient to code. The requirement is after each epoch output, I need to check the iterated training Accuracy and MSE evaluations, hence this will be outside the for loop. **'summary_trainRes'** is the list accumulated all epoch evaluation output. I just need to look at the prior 3 elements **('epoch'** is the order -th aka. currently iterated) and gives specific key as **'MSE'.**

Also at this point, I noticed **'init_acts_shape'** was not being appended for each activation layer. This actually took me a while to figured it out. Since you put this in backprop()function in the original code:

'## nt: DO NOT REMOVE THIS LINE!!

self.init_acts_shape = [act.shape for act in activations]'

I started to see where I can make the appending update in this backprop() function. Eventually I found out that it is within the activation output for loop. This was not easy, only when I start printing out 'activation' and activations', understand it shapes then I was able to code 'self.init_acts_shape' update.

```
#Store initial activation shape.
self.init_acts_shape = [X.shape]

zList = []  #List to store all the z vectors, layer by layer.
for i, (b, w) in enumerate(zip(self.biases, self.weights)):
    z = np.dot(w, activations[i]) + b
    zList.append(z)
    activation = sigmoid(z)
    activations[i + 1] = activation
    self.init_acts_shape.append(activation.shape)  #Append for each activation.
```

Section 4 – Complete

Given the printed output matched with Check2 given output. I was able to correctly implement backprop function requirement. The original code initiate **'activations'** as a list containing only initial X aka. instance input, so start out with one element. The requirement says 'for instance if the network size were [4, 20, 3], you create a list containing three column vectors -- Numpy arrays whose shapes are (4,1), (20,1) and (3,1) respectively', this gives me a hint that I can code **'activations'** as a replica of **'self.networkSize'**. I just use the same code for generating **'nabla_b'**. Then set the first element of **'activations'** list as first instance input X.

```
#Forward pass: start with initial instanceInput X
#Activations list contains the activations of each layer progressed through the network.
activations = [np.zeros((layerSize, 1)) for layerSize in self.networkSize]  #Replica of network size.
activations[0] = X
```

'Then during the forward-propagation, activation values of each layer are copied/assigned into the respective array (overwriting the zero's).' This was very tricky for me, because at first I was seeing each element in **'activations'** list as a neuron itself. So, I overthink the construct of backprop function and made it super complicated. Then when printing out the shape of **'activations'** and **'activation'**, then I know variable **'activation'** is a neuron. Then **'activations[i + 1] = activation'** is the code to overwriting initial zero value.

This modification took me a lot of time, because I was getting confusing between **'activation'** and **'activations'** list – should have name it better as 'activationLayer_list'. I was also seeing the two for loops item iteration 'feed forward' and 'backward pass' the same way initially. But one was calculating each neuron's activation output individually, and the other makes update for the entire layer.

Section 5 – Complete

Given the printed output matched with Check2 given output. I was able to correctly implement backprop function requirement. This modification took me a very long time to figure out. Because ironically, I was not able to understand why the original code was 'not efficient'. I started by breaking down the original code. For each new *baby* 'nabla_b', 'nabla_w' created from an (X, Y) instance in a mini batch, it creates new accumulated 'nabla_b', 'nabla_w' by summing with the existing one. While the requirement asks to keep appending *baby* 'nabla_b', 'nabla_w' to an existing 'nabla_b', 'nabla_w' accumulator.

Basically, the modified code should only have **ONE 'nabla_b', 'nabla_w'** at all times, aka. the syntax 'nabla_b = ' and 'nabla_w = ' should only be coded 1 time. So that was my logic to approach for this modification. Then I was looking at the *for loop* for each instance, 'backprop_nabla_b, backprop_nabla_w = self.backprop(X, Y)' yields *baby* 'nabla_b', 'nabla_w'. How can I append it interactively for EACH NEURON?

What I did was troubleshoot by printing multiple mid-way outputs from functions backprop and update_miniBatch. Then, I found out 'nabla_b', and 'nabla_w' returns from backprop function as lists with each element as a layer (each element is an array with number of rows = number of neurons).

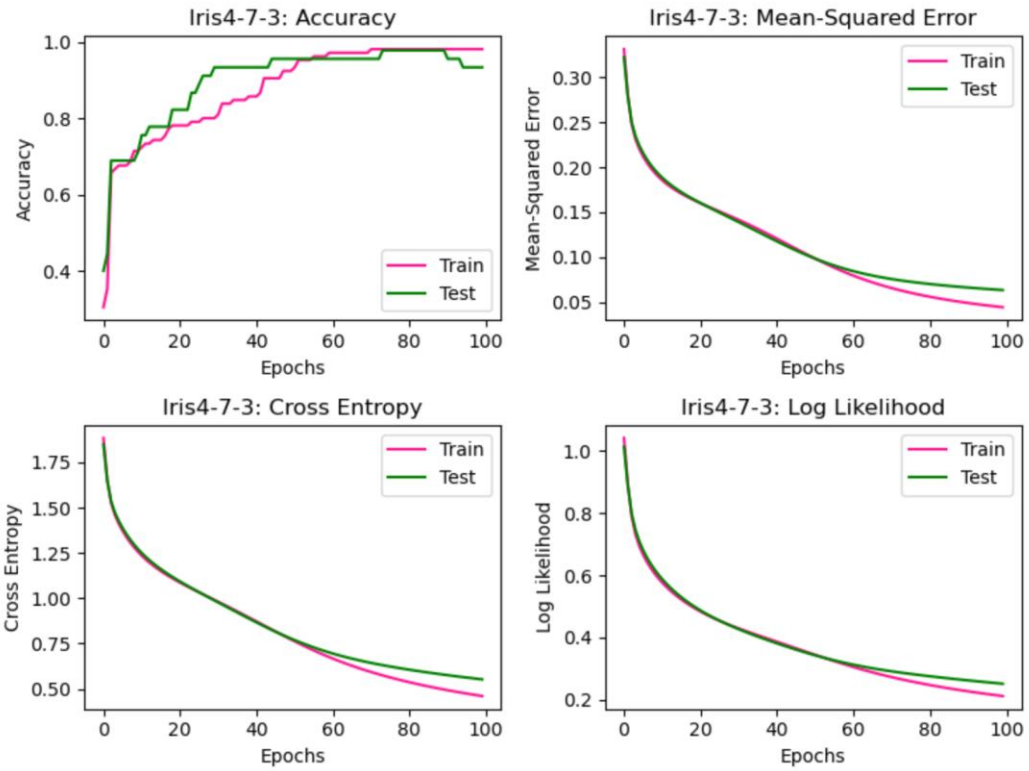
Then go back to the update_miniBatch function, I would need to create an inner for loop to make required update. For every layer iteration, it would update ALL neurons' biases and connected weights within that layer. Then I started to look at which variable length I can use for this inner iteration, and it was the **length of 'nabla_b'**.

```
def update_miniBatch(self, miniBatch, eta):
    '''Update the network's weights and biases to a single mini batch.
    miniBatch: a list of tuples (X, Y), eta: learning rate.'''

    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]

    for X, Y in miniBatch:
        backprop_nabla_b, backprop_nabla_w = self.backprop(X, Y)
        for i in range(len(nabla_b)): #Iterate over eaxch layer.
            nabla_b[i] += backprop_nabla_b[i]
            nabla_w[i] += backprop_nabla_w[i]
    self.weights = [w - (eta / len(miniBatch)) * nw for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b - (eta / len(miniBatch)) * nb for b, nb in zip(self.biases, nabla_b)]

    return nabla_b, nabla_w
```

Section 6 – Complete

I started out with learning rate eta = 0.5, it resulted early stopping around 20 epochs. Then I moved down to eta = 0.4 and still got early stopping around 40 epochs. Then with eta = 0.3, the graphs start to look somewhat similar to provided sample graphs, curves are smoother and ran through 100 epochs. Eta = 0.2 yields interesting results but not consistence; MSE and CE performs well, but LL and Accuracy are somewhat fluctuated.

Given graphs here are using eta = 0.315, so eta 0.3 should yields consistent results on the performances of both train and test sets. From the graphs, Accuracy increases over time, at epoch 100, trainset yields 98.10% and test set yields 93.33%. Mean-Squared Error, Cross Entropy and Log Likelihood are decreasing over time and both train and test sets. Around epoch 60th forward, errors on testing data are slightly higher than training data but not significant. The two lines are almost overlap indicates there is no underfitting/overfitting of the data using eta approximately 0.3. The neural network is performing well on unseen data, correctly classifying the three Iris species.



Section 7 – Complete

Since I have explained thoroughly how I approached every modification requirement in each above section, I will be reflecting my experience in doing this homework.

It took me around 5 days to complete the assignment, and honestly the modification coding was not that difficult once I thoroughly understand the concept of neural network (NN) and the original code. First, I went through the last quiz with simplified NN and compiled a sheet that map out variable names; for example: what is an epoch, minibatch or activation output. Calculation process/formula for both forward passing and backward pass.

Secondly, then I started go through the original code. Actually understand which function does what and rename the variable to my own understanding. I also comment further on original code exactly which line of code does what, how each function producing/retrieving outputs and stored by which data type/structure. This took me around 2.5 days. Thirdly, this is when I actually read the assignment sheet and everything/required modifications make sense. I also look at Check1 and Check2 codes to see the data structure of the SGD function output. For example, list to store accumulated training and testing results of each epoch. Making modifications only takes me 1.5 days, modifications 4^{th} and 5^{th} took me most of the time.

Overall, I enjoyed doing this assignment. I guess by choosing not to jump straight to coding, it was not as stressful for me in completing this homework.