Homework 5 Report

Modified Network Code

Mai Ngo

DSC 578 Neural Network and Deep Learning - SEC 701

Dr. Noriko Tomuro

October 29, 2023

Table of Contents

Cost Functions - Complete

The printed output from my Check 2 were matched with given output.

Cross Entropy – Derivative:

This implementation was not that difficult to make. Since I have done the derivative of Cross Entropy Cost in Homework 4, I just following this formula:

$$\frac{\partial C}{\partial a} = \frac{1}{n} * \sum_x \left(\frac{\text{acutalOutput} - \text{targetOutput}}{\text{acutalOutput} * (1 - \text{acutalOutput})} \right)$$

= '**return** (actualOutput-targetOutput)/(actualOutput*(1-actualOutput))'

LogLikelihood – Cost formula:

For LogLikelihood (LL), I use my formula from Homework 3. This is the equivalent to the formula from Optimizations PowerPoint (PP) slides 21:

$$C = \frac{1}{n} * \sum_x -targetOutput * ln(actualOutput)$$

='**return** np.sum(np.nan_to_num(-targetOutput*np.log(actualOutput)))'

LogLikelihood – Derivative:

The derivative formula of LL was very interesting to write. This is my derivative in 'pen and paper'. Given this, when targetOutput = 0, the LL derivation will be 0. Otherwise, if targetOutput = 1, the derivative of that neuron with respective to activation output is '1/activationOutput':

$$\frac{\partial C}{\partial a} = \frac{1}{n} * \sum_x \frac{\partial}{\partial a}\{-ln(a_y^L\} = \frac{1}{n} * \sum_x \frac{\partial}{\partial a}\{-targetOutput * ln(actualOutput)\}$$

$$= \frac{1}{n} * \sum_x \frac{\partial}{\partial a}\left\{- targetOutput * \frac{1}{\text{actualOutput}}\right\}$$

Equivalently, follow slides 14 formula:

$$\frac{\partial C}{\partial a} = \frac{1}{n} * \sum_x - 1 \,/\, (actualOutput[targetOutput\_index])$$

The only challenging part here is to remind myself, target/actual outputs are represented by column vectors, which I need to do some sort of index iteration for this derivative code.

```
def derivative(actualOutput, targetOutput):
    '''Return the first derivative of the function with respect to 
actual output.'''

    #Inititate zeros array with same shape as actualOutput.
    LL_dev = np.zeros_like(actualOutput)

    #Get the index of the target category.
    targetOutput_index = np.argmax(targetOutput)

    #The derivative values for all other components stay at 0, only 
activationOutput with index corresponding to targetOutput index 
recalculated as '-1/activationOutput'.
    LL_dev[targetOutput_index] = -1/actualOutput[targetOutput_index]

    return LL_dev
```

Activation Hidden - Complete

The printed output from my Check 2 were matched with given output.

Softmax – Activation formula:

Likewise, this implementation was also not that difficult to make. The formula for each of the activation function is already given the Optimizations PP. For Softmax, I follow Optimizations PP slide 9th:

$$\mathrm{softmax}(z) = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}$$

=`return expZ/expZ.sum()` given `expZ = np.exp(z)`

Tanh – Activation and Derivative formulas:

For Tanh both formula and derivative, I follow PP slide 6th:

$$\mathrm{Tanh}(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

= `return (np.exp(z) - np.exp(-z))/(np.exp(z) + np.exp(-z))`

Tanh'(z) $= 1 - \tanh(z)^2$ = `return 1 - (cls.fn(z)**2)`

ReLU – Activation and Derivative formulas:

For ReLU, I follow PP slide 7th:

$\mathrm{ReLU}(z) = e^{+} = \max(0, z)$ = `return np.maximum(0, z)`

$\mathbf{ReLU'(z)} = \begin{cases} 1 \text{ if } z > 0 \\ 0 \text{ if } z < 0 \end{cases}$ = `return np.where(z > 0, 1, 0)`

LeakyReLU – Activation and Derivative formulas:

For LeakyReLU, I follow PP slide 8th:

$\mathrm{LeakyReLU}(z) = \max(\alpha * z, z)$ given user choice α value

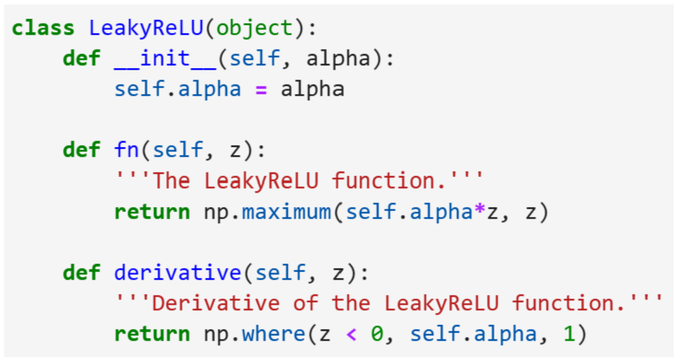
= `return np.maximum(self.alpha*z, z)`

$\mathbf{LeakyReLU'(z)} = \begin{cases} \text{alpha if } z < 0 \\ \alpha \text{ if } z > 0 \end{cases}$ = `return np.where(z < 0, self.alpha, 1)`

Breaking down LeakyReLU derivative,

- ✓ Given α = 0.3, if z = 0.7, i.e., > 0 then we will take z value for LeakyReLU activation output. The derivative would then just be the identity function of z itself, which is 1.
- ✓ Given α = 0.3, if z = -0.7 then we will take α ∗ z value for LeakyReLU activation output. The derivative would then respect to α , which is α.

Furthermore, I also code alpha as an attribute of class LeakyReLU, so input would be: ‘actHidden=LeakyReLU(alpha=0.3)’



```python
class LeakyReLU(object):
    def __init__(self, alpha):
        self.alpha = alpha

    def fn(self, z):
        '''The LeakyReLU function.'''
        return np.maximum(self.alpha*z, z)

    def derivative(self, z):
        '''Derivative of the LeakyReLU function.'''
        return np.where(z < 0, self.alpha, 1)
```

Activation Output – Complete

The printed output from my Check 2 were matched with given output. First I handle the activation output special case, if Tanh is passed as an **'activationOutput'**, print error message and apply Sigmoid instead. The code is very transparent as it is:

```
def set_modelParameters(self, cost=CrossEntropyCost,
                        actHidden=Sigmoid, actOutput=None):
     '''Set model parameters.'''
        …
        #Change based on request.
     if self.actOutput == Tanh:
          print('Error: Tanh cannot be used for output layer.
     Changing to Sigmoid!')
          self.actOutput = Sigmoid
```

Then I handle the feed forward process, which applied to the final output layer. In the backprop function, initially I was thinking to have all 'activation' iterated in the same loop, using if-else statement to be separated hidden versus output layer. However, later on I need to breakdown just the final layer outside the loop to debug for Dropout implementation. Finally, I decided to keep the code in this way. Personally, I think the code look more efficient, clean and easier to understand. Thus, my final code in function backprop()::

```
def backprop(self, X, Y):
       '''Calculate the gradients with respect to the network's
parameters (weights and biases). Return a tuple (nabla_b, nabla_w).'''
     …
     for b, w in zip(self.biases[:-1], self.weights[:-1]):
          z = np.dot(w, activation)+b
          zList.append(z)
          activation = (self.actHidden).fn(z)
          …
          activations.append(activation)

#After iteration to all hidden layers. At this point, 'activation'
would be at the second to last layer. I.e., final hidden layer.
#For output layer. Using self.actOutput.

b = self.biases[-1]
w = self.weights[-1]
z = np.dot(w, activation) + b
zList.append(z)
```

```
activation = (self.actOutput).fn(z)
activations.append(activation)
```

In addition, I also make same change on function feedforward, pretty much the same syntax and logic:

```
def feedForward(self, activationOutput):
       '''Return network activation output. '''

for b, w in zip(self.biases[:-1], self.weights[:-1]):
       activationOutput = (self.actHidden).fn(np.dot(w,
activationOutput) + b)
b = self.biases[-1]
w = self.weights[-1]
activationOutput = (self.actOutput).fn(np.dot(w,activationOutput) + b)
return activationOutput
```

I also want to point out that the given Softmax derivative and backward pass delta calculation in starter code certainly help me identify where to implementation the code, also less work to do. So, thank you!

Regularizations – Complete

The printed output from my Check 2 were matched with given outputs. Firstly, thanks to the discussion post, I was able to code regularization given three scenarios: L1, L2, and no regularization as default. In addition, you also give a very helpful hint, which I think it saves a lot of time which is the modifications will be made:

1. **During training, when weights are adjusted at the end of a mini-bath – the function update_miniBatch().**

```
def update_miniBatch(self, minibatch, eta, n):
       '''Update the network's weights and biases to a single mini batch. '''
…
#'L1' Regularization.
if self.regularization == 'L1':
      self.weights = [w - (eta * (self.lmbda/n) * np.sign(w)) -
((eta/len(miniBatch)) * nw) for w, nw in zip(self.weights, nabla_w)]

#'L2' Regularization.
elif self.regularization == 'L2':
      #Formula given.
      self.weights = [(1-eta * (self.lmbda/n)) * w-(eta/len(miniBatch))
* nw for w, nw in zip(self.weights, nabla_w)]

#No regularization used, weight update formula from Homework 3.
else:
      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)]
```

Breaking down 'L1' regularization formula from Optimizations PowerPoint slides 28:

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} * \frac{\lambda}{n} * sgn(w) = w - \eta\frac{\partial C_0}{\partial w} - \frac{\eta * \lambda}{n} * sgn(w)$$

$$= w - \left(\eta * \frac{\lambda}{n} * sgn(w)\right) - \left(\eta\frac{\partial C_0}{\partial w}\right)$$

```
= '[w - (eta * (self.lmbda/n) * np.sign(w)) -((eta/len(miniBatch)) *
nw) for w, nw in zip(self.weights, nabla_w)]'
```

In addition, I was also debating between using np.sign and a custom sign function (which I included in .ipynb file). But it yields the same output, so I stay with np.sign.

2. **During evaluation, when the cost is computed – the function totalCost().**

```
def totalCost(self, data):
        '''Return the total cost for input dataset.'''
…
penaltyValue = 0.0

#'L1' Regularization.
if self.regularization == 'L1':
     penaltyValue += (self.lmbda/len(data)) *
np.sum([np.sum(np.abs(w)) for w in self.weights])

#'L2' Regularization.
elif self.regularization == 'L2':
     penaltyValue += 0.5*(self.lmbda/len(data)) *
sum(np.linalg.norm(w)**2 for w in self.weights)

#No regularization used, penaltyValue remains as 0.0.
cost += penaltyValue
return cost
```

Breaking down 'L1' regularization formula from Optimizations PowerPoint slides 28:

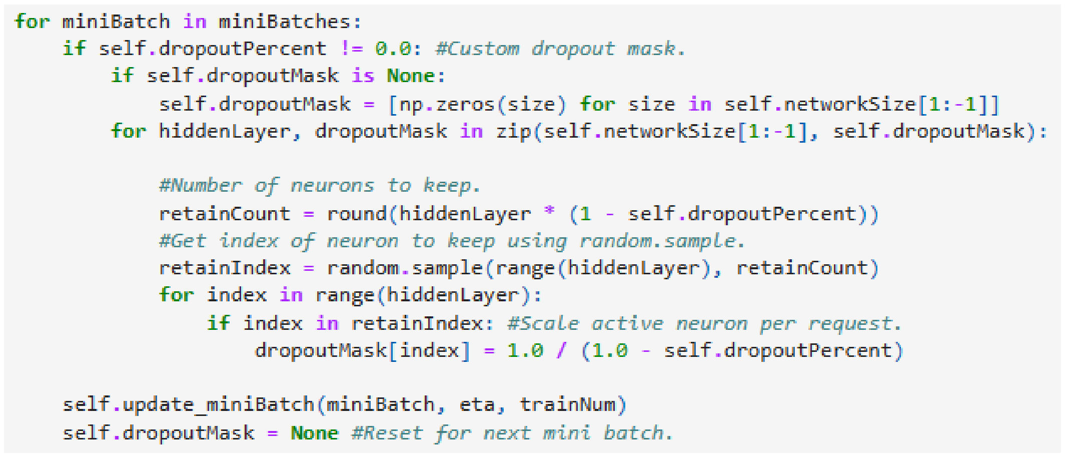
$C = C_0 + \frac{\lambda}{n} * \sum_w |w|$ = `'(self.lmbda/len(data)) * np.sum([np.sum(np.abs(w)) for w in self.weights])'`

This formula took me a bit to understand it fully. I originally have the code wrong which use sum of absolute value of 'np.linalg.norm'. Later on, I learned that it should be sum of the absolute values with respect to an individual weight array in self.weights, **'np.sum(np.abs(w)'**. More detail, it takes the absolute value of individual weight element in each weight array **'np.abs(w)'**. Then finally sum all weight arrays' sum of absolute values **'np.sum([np.sum(np.abs(w))]'**. And the scaling would be without '0.5' based on PP formula.

Dropout – Complete

The printed output from my Check 2 were not matching with given outputs. As I have mentioned with you in the email, also from the discussion board with other students. I believe none of us were able to obtain the same output as given text documents. And I am pretty confident in my Dropout implementation.

The process of doing dropout implementation was very intensive. My initial dropout mask code was incorrect because I did not apply scaling properly. Then followed your email, I was able to fix that problem, which turned out I was scaling values of inactive neurons, this should be done vice versa. My 'dropoutMask' is a list with each element (as numpy array) is a dropout mask for each hidden layer. The length of each dropout mask is equal to number of neurons in respective layer, i.e., element of network size. When I got that figured out, I just code accordingly.



```
for miniBatch in miniBatches:
    if self.dropoutPercent != 0.0: #Custom dropout mask.
        if self.dropoutMask is None:
            self.dropoutMask = [np.zeros(size) for size in self.networkSize[1:-1]]
        for hiddenLayer, dropoutMask in zip(self.networkSize[1:-1], self.dropoutMask):

            #Number of neurons to keep.
            retainCount = round(hiddenLayer * (1 - self.dropoutPercent))
            #Get index of neuron to keep using random.sample.
            retainIndex = random.sample(range(hiddenLayer), retainCount)
            for index in range(hiddenLayer):
                if index in retainIndex: #Scale active neuron per request.
                    dropoutMask[index] = 1.0 / (1.0 - self.dropoutPercent)

    self.update_miniBatch(miniBatch, eta, trainNum)
    self.dropoutMask = None #Reset for next mini batch.
```

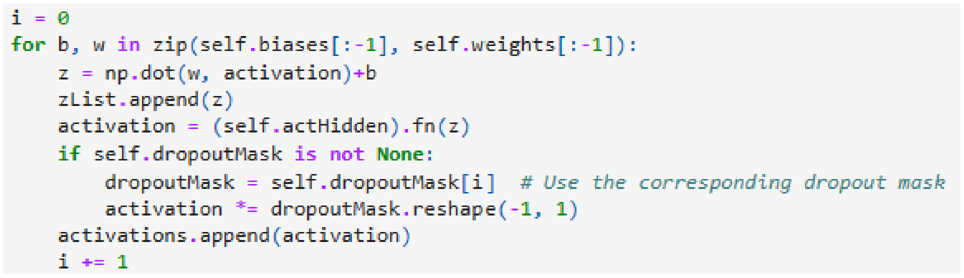
For each mini batch, the code will check if **'dropoutPercent'** is being input by user, i.e., **'self.dropoutPercent != 0.0'**. If it is, then I initiate dropoutMask as zeros arrays using networkSize. Within each dropoutMask of respective hidden layer, I obtain number of neurons to keep, and get index of neurons to keep using random.sample. After that, I iterate through each element in individual dropoutMask and scale active neuron per request **'1.0 / (1.0 - self.dropoutPercent)'**. Finally, after each miniBatch completion, I **reset 'dropoutMask = None'** for the next one. Given request, use the same dropout masks during one mini-batch.

In addition, I was not really understanding the idea of ‘Scale the output values of the layer’ initially. Later on, this is my understanding which helps me a lot in doing dropout implementation thoroughly; since dropout is for hidden layers, I need to boost the activations of chosen active neurons to potentially come close to the same final layer output, as if all neurons are active.

For both feed forward and backward pass, basically it is just element wise matrix multiplication. The only thing I need to tackle is shapes of hidden layer’s ‘activation’ and ‘delta’ versus shape of ‘dropoutMask’. What I did was having step-by-step print statements to troubleshoot, which I reshape ‘dropoutMask’ to a single column (-1, 1). Index iteration also plays an important role here, the 2nd layer with first ‘dropoutMask’ element in feed forward; but in backward pass, 1st hidden layer in loop iteration using last element of ‘dropoutMask’

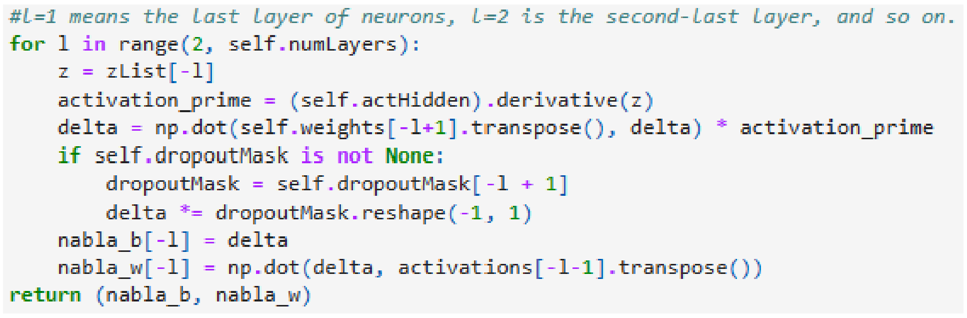
`def backprop(self, X, Y):`

#Feed forward.



```
i = 0
for b, w in zip(self.biases[:-1], self.weights[:-1]):
    z = np.dot(w, activation)+b
    zList.append(z)
    activation = (self.actHidden).fn(z)
    if self.dropoutMask is not None:
        dropoutMask = self.dropoutMask[i]  # Use the corresponding dropout mask
        activation *= dropoutMask.reshape(-1, 1)
    activations.append(activation)
    i += 1
```

#Backward pass.



```
#l=1 means the last layer of neurons, l=2 is the second-last layer, and so on.
for l in range(2, self.numLayers):
    z = zList[-l]
    activation_prime = (self.actHidden).derivative(z)
    delta = np.dot(self.weights[-l+1].transpose(), delta) * activation_prime
    if self.dropoutMask is not None:
        dropoutMask = self.dropoutMask[-l + 1]
        delta *= dropoutMask.reshape(-1, 1)
    nabla_b[-l] = delta
    nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
return (nabla_b, nabla_w)
```

Reflection – Complete

Similar to Homework 3, I started the assignment with going through the original code and re-name the variables to my own understanding. Since I already comprehend completely what was going on in the last homework, I was not spending a lot of time in this step.

The first three implementations, cost function, activation hidden, and activation output are not that difficult to make. Certainly, it still takes me a day to figured out everything correctly in these steps in order to move forward. Some formula, I have to do written down in pen and paper to type the code correctly. However, the last two implementations are very hard and time consuming, required a lot of manual works like pen-and-paper formula break down. I also have to debug each line of code, specifically shape and data type of each variable I am dealing with. On top of that, just to visualize the connections and where to put the implementations before coding also took me sometimes. In addition, attributes ‘dropoutPercent’, ‘dropoutMask’, ‘regularization’, and ‘lmbda’ are also initiated in the constructor as Network class attributes, so that for experiment 1 to 7, ‘dropoutPercent’ value will still be passed in as 0.0 when function ‘set_compileParameters’ is not being called.

Overall, I spend around five full days to do this homework, Regularizations and Dropout took me most of the time. Even though my Experiment 8 to 10 outputs are not similar, I’m pretty confident in my code and logic for dropout implementation. Thus, satisfied with where I am at with this assignment, and thoroughly understand everything.