report

November 5, 2021

0.1 SPDS HW5

0.1.1 Name: Jeonghyun Moon

0.1.2 Student ID: 2021-35791

0.1.3 Problem 1

As an approach to this problem, I have used the code provided from Michael Nielson's book, which is provided in the following website: http://neuralnetworksanddeeplearning.com/chap1.html. Because the original code had eta instead of lr, I have updated areas in which it had lr to eta. Other than lr-eta pair, I have also updated xrange() to range() because xrange() is only in Python 2, and it was replaced with range() in Python 3. I will provide the section of the code that performs each bullent points asked in the questions.

The question asks where the evaluation of the feedforward computation and input label is performed. In order to compare the two values, the evaluate(self, test_data) is used, and in the main function of SGD, it can be found right after updating the mini batch occurs, which is the following area:

[ ]:
```
    def SGD(self, training_data, epochs, mini_batch_size, lr, momentum,␣
↪test_data=None):
        ...
        ...
            if test_data:
                print("Epoch {0}: {1} / {2}".format(
                    j, self.evaluate(test_data), n_test))
            else:
                print("Epoch {0} complete".format(j))
```

The above code will run if there is the test data available. It will compare tthe label and forward computed value for each epoch.

Then, the next question asks where the last layer's output's gradient is calculated. This happens inside the backprop(self,x,y) algorithm. After it computes the activation layer by layer and saving z vectors for each layer, it starts from the back. It will first measure the gradient or how fast the cast is chaning as function of the output activation. The second term sigmoid_prime(zs[-1])comes from sum of all nuerons k in output layer where we have sum of k of da_k^L / dz_j^L. Because it only depends on case when k = j, this simplifies to da_j^L / dz_j^L because $a_j^L = \sigma(z_j^L)$ resulting $\sigma'(z_j^L)$. The bias is the delta value itself, and the dot product of delta and previous layers activation function would provide the nabla weight of the final layer. The code is in the following section where it

computes the gradient/delta and obtain nabla for both bias and weight of the last layer:

[ ]:
```
    def backprop(self, x, y):
        ...
        ...
        # backward pass
        # start from the back
        # The first term self.cost_derivative(activations[-1], y) measures
            # how fast the cost is changing as function of the output activation
        # The second term sigmoid_prime(zs[-1]) comes from sum of all nuerons k␣
↪in output layer
            # where we have sum of k of da_k^L/ dz_j^L
            # because it only depends on case when k = j, this simplifies to
            # da_j^L/ dz_j^L
            # because a_j^L = sigma(z_j^L), it become sigmoid_prime(z_j^L)
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        # This looks complicated, but there is an simple explanation
            # since we know delta and activations, the transpose of weight␣
↪matrix is applied
            # because we can think it as moving error backward through network
            # giving idea of measure of error at output layer
            # (here he expects column vector so that is why transpose is␣
↪applied)
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
```

The next qusetion asks where all the other layer's nabla for bias and weight is updated. This is right after computing the nabla for bias and weight of the code.

[ ]:
```
    def backprop(self, x, y):
        ...
        ...
        # Note that the variable l in the loop below is used a little
        # differently to the notation in Chapter 2 of the book.  Here,
        # l = 1 means the last layer of neurons, l = 2 is the
        # second-last layer, and so on.  It's a renumbering of the
        # scheme in the book, used here to take advantage of the fact
        # that Python can use negative indices in lists.
        for l in range(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)
```

The code starts from 2 to number of layers using for loop. The code starts from 2, because we are

starting from second to last layer. The following line z = zs[-l] makes z to be the activation value obtained earler process of back propogation. Then, sp = sigmoid_prime(z) will obtain derivative of sigmoid function. This needs to be obtained because the error term in the error is the delta, which is product of transpose of the weight matrix and error of the last layer with performing Hadamard product (HP) with derivative of activation of sigmoid function. The equation can be written as follows:

$$\delta^l = ((w^{l+1})^T \delta^{l+1} HP \sigma'(z^l)$$

While the above equation looks complicated, Michael Nielson provides simplar explation. Since we know delta and activations, the transpose of weight matrix is applied because we can think it as moving error backward through network giving idea of measure of error at output layer. Recall that in the matrix multiplication for forward process, we have weights * input to produce outputs, and the size of the matrix really matters since the column of the first vector must match with row of the second vector, here number of columns for the weight must equal number of rows in input to obtain the correct size for output. Note the the backward pass is actually takes in error and updates t the weight matrix and bias. Because weight is in shape of derivative of activation function and transpose of earlier activation. The weight updates are considering this item so if the weight is not tranposed, the matrix multiplication cannot be performed.

The next question asks where the weight and bias itself is updated. Note the backpropogation is one that calcuates the error terms. The actual weight and bias is updated in the mini-back function in following section of the code:

[ ]:
```
    def update_mini_batch(self, mini_batch, lr):
    # TODO: Implement here
    #pass
    # I have used code from Michael Nielsen
        """Update the network's weights and biases by applying
        gradient descent using backpropagation to a single mini batch.
        The ``mini_batch`` is a list of tuples ``(x, y)``, and ``eta``
        is the learning rate."""
        # first initialize nabla values for b and w to be 0
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # for each mini batch
        for x, y in mini_batch:
            # apply backpropagation algorithm to compute the gradient of cost␣
↪function
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            # update the values according to back propogation
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
        # update weights and biases
        self.weights = [w-(lr/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(lr/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)]
```

First, we have to initial nabla b and w to be 0 because we do not know the values. Then, for each mini batch, we will perform the back propogation to obtain delta of bias and delta of b. Since updated nabla is sum of old and the delta. After the nabla values are obtained, we will use the equation to obtain updating the weights and bias which are done in the self.weights and self.biases updating in end of the end of the code (recall the updated weight is difference of old and changed value).

When running the problem, I have obtained the following result:

- Epoch 0: 3981 / 10000
- Epoch 1: 6037 / 10000
- Epoch 2: 6962 / 10000
- Epoch 3: 7381 / 10000
- Epoch 4: 7600 / 10000

0.1.4 Problem 2

Next step was doing fine tuning where if tuning is True, only the last layer's weight and bias is update. For this step, other than putting the tuning in functions, only change was made in the backpropogation where after the last layer is updated if tuning is true, it will return 0 for delta other than last layer, and if tuning is false, it will perform what was done in Problem 1. The code can be seen in the following section:

[ ]:
```
    def backprop(self, x, y, tuning):
        ...
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())

        # Because last layer is finished, if tuning will end here
        if tuning:
            return (nabla_b, nabla_w)
        else:
            # Note that the variable l in the loop below is used a little
            # differently to the notation in Chapter 2 of the book.  Here,
            # l = 1 means the last layer of neurons, l = 2 is the
            # second-last layer, and so on.  It's a renumbering of the
            # scheme in the book, used here to take advantage of the fact
            # that Python can use negative indices in lists.

            for l in range(2, self.num_layers):
                z = zs[-l]
                sp = sigmoid_prime(z)
                delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
                nabla_b[-l] = delta
                nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
            return (nabla_b, nabla_w)
```

The result for running the code is as follows:

- ==Full training==
  - Epoch 0: 3981 / 10000
  - Epoch 1: 6037 / 10000
  - Epoch 2: 6962 / 10000
- ==Fine-tuning==
  - Epoch 0: 7070 / 10000
  - Epoch 1: 7171 / 10000
  - Epoch 2: 7184 / 10000

Note that we can see that full training is working correctly, for we have same accurracy value. Note that for fine tuning, epoch 0 was very good, but after that there were slower improvement compared to original code.

0.1.5 Problem 3

For problem 3, we were supposed to do momentum SGD. This was done through adding the velocity value for both bias and weight in initialization and updated them in the minibatch as seen in the following code.

[ ]:
```
    def __init__(self, sizes):
        ...
        # Add velocities for weight and biases
        self.bias_velocities = [np.zeros((y, 1)) for y in sizes[1:]]
        self.weight_velocities = [np.zeros((y, x))
                        for x, y in zip(sizes[:-1], sizes[1:])]


    def update_mini_batch(self, mini_batch, lr, momentum):
        ...
        # Update weights and biases velocities
        self.weight_velocities = [momentum * v - (lr / len(mini_batch))
                                  for v in self.weight_velocities]
        self.bias_velocities = [momentum * v - (lr / len(mini_batch))
                                for v in self.bias_velocities]

        # update weights and biases
        self.weights = [w + nw * v
                        for w, nw, v in zip(self.weights, nabla_w, self.
↪weight_velocities)]
        self.biases = [b + nb * v
                       for b, nb, v in zip(self.biases, nabla_b, self.
↪bias_velocities)]
```

In the above code, as directed in the question, weight and bias velocities are updated first. Then, after the weight and bias velocities are updated, I have updated the weight and bias using the

equation provided. When running the code, I have obtained the following result:

- Epoch 0: 3979 / 10000
- Epoch 1: 6043 / 10000
- Epoch 2: 6964 / 10000
- Epoch 3: 7379 / 10000
- Epoch 4: 7599 / 10000