CSC 578 HW 4

Name: **Milin Desai**

1 Add comments for the following code snippet

from the function backprop() in NN578_network.py.

```
# backward pass
delta = self.cost_derivative(activations[-1], y) * \
    sigmoid_prime(zs[-1])
# delta function we used for finding the error in the output layer. We are also using the
#activation and sigmoid to make a consistent interface which can be useful in the cost
#functions. Also here activation[-1] is the last layer.
#In addition, delta will give us the last output layer. Also one thing to note is that we have single
input

nabla_b[-1] = delta #giving value of delta
nabla_w[-1] = np.dot(delta, activations[-2].transpose())
#here activation[-2] is the one layer before the last layer so to get the shape of the output layer
we need to do dot product and also transpose of the second last layer of the output layer which
will give us shape of the [p, q]
```

- Why .transpose() in the last line is needed.

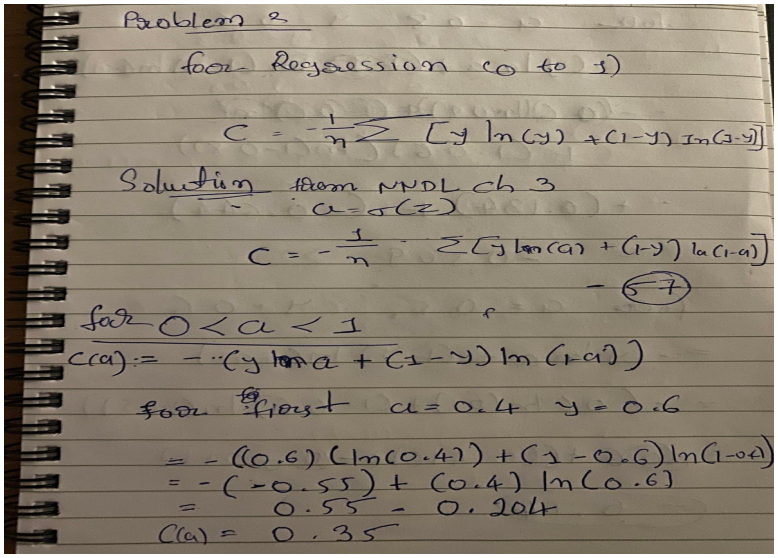
  Ans: Because, it won't have the same rows and columns in the network architecture so to avoid unexpected errors we will need .transpose() in the last. However, the matrix shape of the delta will be [P,1] also output of the activation[-2] will have similar shape [Q,1] so we need to transpose it so that we can get [1,Q]. Now dot product of the last line we will obtain exactly desirable output in the form of [P,Q]

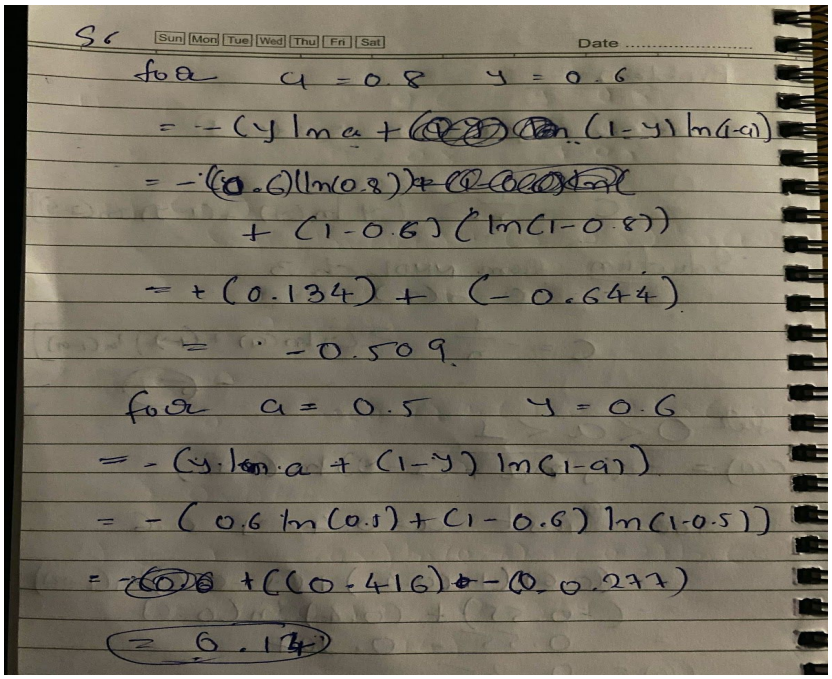
- Why the first line is using * but the third line is using np.dot().

  Ans: Main reason, we can use " * " in the case of matrices but when there are two vectors it is advisable to use dot product. In addition, first line is While when we are using " * " we will receive the output shape of the matrix [P, 1] something like this but for perfect shape like [P, Q] we will require the shape of weights to be like [1,Q] so that we can get perfect shape. So to obtain perfect matrix size it is important to use dot product.

2 An Exercise on Cross-entropy cost function in NNDL 3.

- If you assume the former, you can do a rigorous proof by using calculus and minimizing the derivative of the function. But if you are not comfortable with calculus, **you can pick at least three values for *y* (between 0 and 1), and for each value of *y*, you should compute the Cross-entropy value using that *y* and several varying values of *a* (e.g. 0.1, 0.2, 0.3,.. 0.9).**
- If you assume the latter, you can do a formal proof by calculus as well (although the derivative function will have several variables), but I recommend using Information Theory to prove the formula will minimize when *y* and *a* are equal.





Sun Mon Tue Wed Thu Fri Sat Date .....

Similarly for ① $a = 0.2$ $y = 0.5$
② $a = 0.6$ $y = 0.5$
③ $a = 0.8$ $y = 0.5$

① $C(a) = -(0.5 \ln(0.2) + (1-0.5)\ln(1-0.2)$
$= 0.804 - 0.111$
$= 0.69$

② $C(a) = -(0.5 \ln(0.8) + (1-0.5)\ln(1-0.6)$
$= 0.255 - 0.458$
$= -0.203$

③ $C(a) = -(0.5 \ln(0.8) + (1-0.5)\ln(1-0.8)$
$= 0.111 - 0.804$
$= 0.693$

Sun Mon Tue Wed Thu Fri Sat Date .....

1) $a = 0.6$ $y = 0.8$
$a = 0.7$ $y = 0.8$
$a = 0.8$ $y = 0.8$

1) $C(a) = -(0.8)\ln(0.6) + (1-0.8)\ln(1-0.6)$
$= 0.408 - 0.183$
$= 0.225$

2) $C(a) = -(0.8)\ln(0.7) + (1-0.8)\ln(1-0.7)$
$= +0.285 - 0.241$
$= 0.044$

3) $C(a) = -(0.8)\ln(0.8) + (0.2)\ln(0.2)$
$= +0.18 - 0.329$
$= -0.14$

Sun Mon Tue Wed Thu Fri Sat Date .....

OR

Entropy

$$C(a) = -\frac{1}{n}\sum_{x}[y\ln(y) + (1-y)\ln(1-y)]$$

$$C(a) = -(y\ln a + (1-y)\ln(1-a))$$

$$C'(a) = -\frac{y}{a} + \frac{1-y}{1-a}$$

$$0 = -\frac{y}{a} + \frac{1-y}{1-a} \quad (\because C'(a) = 0)$$

$$\frac{y}{a} = \frac{1-y}{1-a}$$

$$y(1-a) = a(1-y)$$

$$y - ay = a - ay$$

$$a = y$$

Second derivation of the function

$$C'' = \frac{y}{a^2} + \frac{1-y}{(1-a)^2}$$

∴ So this can not be less than or equal to zero

$$C'' \geq 0$$

So $a = y$

3 A Problem in NNDL Ch. 3 (updated on 10/17 with correct link)

Explain why it is not possible to eliminate the xj term through a clever choice of cost function.

Ans: From the equation (61), we can say that in cost function, the value of a and y are almost the same in that case if we eliminate the xj then there is a chance of getting almost zero divided by n so chances of getting will be zero. So it is not possible to eliminate xj term from the cost function so even if you choose any xj last activation. So it is advisable to not to neglect otherwise we will receive same output for all.

$$\frac{\partial C}{\partial w_j} = \frac{1}{n} \sum_x x_j(\sigma(z) - y)$$

4 TensorFlow/Keras tutorial:

From the document we can infer that this is the data of apparel and we sort of explore the data in the start 60k images used to train models and 10k for evaluation.
We preprocessed the data, made it in a correct format and then started building a model.Then we set up the

Initial Model:

In, Initial model we have selected the node value is 128, didn't make any major changes still output were as below. We got the optimal model and output is as below,I have got 93% accuracy. On the test model we achieved accuracy of 88% with loss of 0.35 so that we can say that this model have learning fast.

```
model.fit(train_images, train_labels, epochs=10)
```

```
Epoch 1/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2294 - accuracy: 0.9127
Epoch 2/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2240 - accuracy: 0.9158
Epoch 3/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2131 - accuracy: 0.9201
Epoch 4/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2094 - accuracy: 0.9215
Epoch 5/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2028 - accuracy: 0.9233
Epoch 6/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1960 - accuracy: 0.9255
Epoch 7/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1914 - accuracy: 0.9287
Epoch 8/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1885 - accuracy: 0.9298
Epoch 9/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1821 - accuracy: 0.9316
Epoch 10/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1769 - accuracy: 0.9332
<tensorflow.python.keras.callbacks.History at 0x7f9bad533e48>
```

```
test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)

print('\nTest accuracy:', test_acc)
```

```
313/313 - 0s - loss: 0.3522 - accuracy: 0.8851

Test accuracy: 0.8851000070571899
```

- **Number of nodes in hidden layers:**

  In the Initial model we did modification and added a number of nodes in the hidden layers we added 96 with the activation form as relu. So you can see the below how accuracy and losses changed.

  I have applied different values but for this value model was optimal with accuracy of 88% and lossed 34%which is slightly less than the initial model. For losses it is good but for accuracy initial model wins.

  In the model fit I have achieved accuracy of the 90% which is little less than the previous model.

```
model.fit(train_images, train_labels, epochs=10)
```

```
Epoch 1/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.5030 - accuracy: 0.8228
Epoch 2/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3795 - accuracy: 0.8648
Epoch 3/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3425 - accuracy: 0.8757
Epoch 4/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3181 - accuracy: 0.8838
Epoch 5/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3012 - accuracy: 0.8895
Epoch 6/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2893 - accuracy: 0.8931
Epoch 7/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2726 - accuracy: 0.8989
Epoch 8/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2614 - accuracy: 0.9030
Epoch 9/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2543 - accuracy: 0.9051
Epoch 10/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2459 - accuracy: 0.9077
<tensorflow.python.keras.callbacks.History at 0x7f9badd80978>
```

```
test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)

print('\nTest accuracy:', test_acc)
```

```
313/313 - 0s - loss: 0.3436 - accuracy: 0.8810

Test accuracy: 0.8809999823570251
```

- **Number of hidden layers**

In the initial model, I haven't changes anything in the input layer which flatten the image and convert in to the 2-D graph I have added three layer including one existing hidden layer I have same activation form ReLu for this as well with that I was not able to get any huge change in accuracy it was the same as the last modification was done.

From the below snippet you can see the accuracy of the model after making this experiment in this parameters we are getting 90% not any big change.

Where in the testing model losses were same as the above around 34% Although test accuracy reduced. Maybe because the model is learning slowly. However I was failed to observe overfitting problem in the data

And in above all examples we have calculated accuracy and loose with reference to the training set. Just quick note that I have used different nodes for the different layers to make the mode more efficient.

```
model.fit(train_images, train_labels, epochs=10)
```

```
Epoch 1/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.5190 - accuracy: 0.8178
Epoch 2/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3772 - accuracy: 0.8624
Epoch 3/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3407 - accuracy: 0.8749
Epoch 4/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3197 - accuracy: 0.8819
Epoch 5/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3020 - accuracy: 0.8893
Epoch 6/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2867 - accuracy: 0.8937
Epoch 7/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2755 - accuracy: 0.8973
Epoch 8/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2676 - accuracy: 0.9003
Epoch 9/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2576 - accuracy: 0.9033
Epoch 10/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2494 - accuracy: 0.9061
<tensorflow.python.keras.callbacks.History at 0x7f9bab398780>
```

```
test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)

print('\nTest accuracy:', test_acc)
```

```
313/313 - 0s - loss: 0.3453 - accuracy: 0.8785

Test accuracy: 0.8784999847412109
```

- **Activation function for hidden layers**

Here, I have used a different activation form. Rather than moving forward with our normal Relu activation I tried to use tanh. Softmax is also another popular activation form but I tried but my accuracy dropped down so I choose to move forward with tanh.

```
[62] model = tf.keras.Sequential([
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(96, activation='tanh'),
        tf.keras.layers.Dense(42, activation='tanh'),
        tf.keras.layers.Dense(32, activation='tanh'),
        tf.keras.layers.Dense(10)
    ])
```

```
model.fit(train_images, train_labels, epochs=10)
```

```
Epoch 1/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4880 - accuracy: 0.8274
Epoch 2/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3666 - accuracy: 0.8663
Epoch 3/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3356 - accuracy: 0.8752
Epoch 4/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3140 - accuracy: 0.8850
Epoch 5/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3012 - accuracy: 0.8888
Epoch 6/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2882 - accuracy: 0.8932
Epoch 7/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2761 - accuracy: 0.8979
Epoch 8/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2677 - accuracy: 0.9015
Epoch 9/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2602 - accuracy: 0.9023
Epoch 10/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2526 - accuracy: 0.9048
<tensorflow.python.keras.callbacks.History at 0x7f9bb7fcf3c8>
```

```
test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)

print('\nTest accuracy:', test_acc)
```

```
313/313 - 0s - loss: 0.3430 - accuracy: 0.8774

Test accuracy: 0.8773999810218811
```

- **Learning rate**
  Learning rate, I have tried with 0.01 and I got maximum result I've tried couple other values also but I received the highest output which I have shown below.

```
opt = tf.keras.optimizers.Adam(learning_rate=0.01)
model.compile(optimizer=opt,
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

```
model.fit(train_images, train_labels, epochs=10)
```

```
Epoch 1/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.7312 - accuracy: 0.7304
Epoch 2/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.6848 - accuracy: 0.7527
Epoch 3/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.6484 - accuracy: 0.7669
Epoch 4/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.7144 - accuracy: 0.7415
Epoch 5/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.6968 - accuracy: 0.7398
Epoch 6/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.6917 - accuracy: 0.7396
Epoch 7/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.6829 - accuracy: 0.7488
Epoch 8/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.6824 - accuracy: 0.7592
Epoch 9/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.6646 - accuracy: 0.7658
Epoch 10/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.6787 - accuracy: 0.7528
<tensorflow.python.keras.callbacks.History at 0x7f9bb8373fd0>
```

```
test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)

print('\nTest accuracy:', test_acc)
```

```
313/313 - 0s - loss: 0.6696 - accuracy: 0.7700

Test accuracy: 0.7699999809265137
```