CSC 578 Homework #4
Serena Yang

1.

```
1: delta = self.cost_derivative(activations[-1], y) * \

2:    sigmoid_prime(zs[-1])

3: nabla_b[-1] = delta

4: nabla_w[-1] = np.dot(delta, activations[-2].transpose())
```

The forward pass creates a list of activations, where activations[i] contain the activation vectors for neurons in layer i. So activation[-1] is the last layer, and y is the desired output.
The first line computes the derivatives in the backpropagation for the gradient decent and outputs a vector with the same shape as our output layer. The second line computes the gradient of bias, and the third line computes the gradient of weights. In gradient decent, we need first to compute the cost function, multiply the cost function by the gradient to get derivatives for weights and bias, and finally continue this process in the backpropagation to update weight and bias for each epoch. These there line code is doing this process in the back pass.
We need to get a scalar for the gradient of weights, then multiply delta by the output from the activation function and do addition for all multiplications. That's why the first line uses *, but the third line uses np.dot().
The reason why .transpose() in the last line is needed it's because transpose is to flip the row and column, so the dot product of delta and activations is doable.

2.

For this problem, I changed the value of a, and computed the cross-entropy for three y equal to 0.2, 0.5 and 0.8 that I picked randomly between 0 to 1:
The cross-entropy formula: $-[y\ln a+(1-y)\ln(1-a)]$

First trial: y=0.2, the minimum value for cross-entropy is 0.5, which corresponds to a =0.2
Second trial: y=0.5, the minimum value for cross-entropy is 0.69 which corresponds to a =0.5
Third trial: y=0.8, the minimum value for cross-entropy is 0.5 which corresponds to a =0.8

0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	y
0.544805	0.500402	0.526135	0.591919	0.693147	0.835198	1.034513	1.332179	1.86314	0.2
1.203973	0.916291	0.780324	0.713558	0.693147	0.713558	0.780324	0.916291	1.203973	0.5
1.86314	1.332179	1.034513	0.835198	0.693147	0.591919	0.526135	0.500402	0.544805	0.8

When the a is closer to y, we will get the lowest amount and minimum value for cost function.

3.

The smaller the input x_j, the slower and more corresponding weight w_j will learn. However, x_j can not be 0 because x_j determines how will train the network. Weights from the cost function are calculated by partial derivative, and x_j is the constant for the cost function, so we cannot eliminate x_j term.

4.

I have run various model configurations and hyperparameter values and examined their effects on learning. I have experimented with the following parameters:

- Number of nodes in hidden layers
- Number of hidden layers
- Activation function for hidden layers
- Regularization methods (set through Keras Regularizer)

a. Number of nodes in hidden layers

I have experimented with 64, 128, 256, and 320 nodes in hidden layers. The results showed that 1) when lowering the number of nodes, the time of running the model decreased, but the training model's prediction or accuracy decreased. 2) When doubling the number of nodes increased the time, it also increased the accuracy of both the train and test. Therefore, I would choose the original number of nodes as there was no significant difference in accuracy and run time when decreasing and increasing the number of nodes in hidden layers.

Number of nodes in hidden layers = 64
Test accuracy: 0.8786
Test loss: 0.34776569106578825
****

Number of nodes in hidden layers = 128
Test accuracy: 0.8766
Test loss: 0.3548506158709526
****

Number of nodes in hidden layers = 256
Test accuracy: 0.8775
Test loss: 0.3447158704638481
****

Number of nodes in hidden layers = 320 Test accuracy: 0.8737
Test loss: 0.3506243628740311

b. Number of hidden layers

I have added one more hidden layer with 64 nodes on the original model. The results show that there is no significant difference in terms of accuracy and loss value on both train and test sets. But the time of running the model with two hidden layers is slightly longer than the one with only one hidden layer. Therefore, I suggest keeping only one hidden layer as there is no significant difference in their final output.

Hidden layer 1: 128 nodes; hidden layer 2: 64 nodes
Test accuracy: 0. 0.8813999891281128
Test loss: 0. 0.34036076068878174
****

c. Activation function for hidden layers

I have experimented with different activation functions ('relu', 'sigmoid', 'tanh', and 'hard sigmoid') and compared each activation function's results in classification accuracy, loss value, and time. The results show that changing the activation function to sigmoid and tanh would reduce the time and increase the accuracy of both the train and test set. However, switching to the hard_sigmoid activation function have not much effect on the model.

active_function = relu
Test accuracy: 0.8799
Test loss: 0.34621168494224547
****

active_function = sigmoid
Test accuracy: 0.8773
Test loss: 0.3418476326704025
****

active_function = tanh
Test accuracy: 0.8759
Test loss: 0.34962565730810163
****

active_function = hard_sigmoid
Test accuracy: 0.8782
Test loss: 0.34676927230358123
****

d. Regularization methods

I have experimented different regularization methods ('sgd', 'RMSprop', 'Adagrad', and

'Adamax') and compared the results from each regularization method regarding classification accuracy, loss value, and time. The results show that the performance of the model when changing the learning rate to Adamax was not better than Adam's (the original). In general, the performance of RMSprop is the best only except for the time perspective.

optimizer = adam
Test accuracy: 0.8779
Test loss: 0.3390931452035904
****

optimizer = sgd
Test accuracy: 0.8157
Test loss: 0.5119203433513642
****

optimizer = RMSprop
Test accuracy: 0.8678
Test loss: 0.36427381875514986
****

optimizer = Adagrad
Test accuracy: 0.782
Test loss: 0.6934799962043762
****

optimizer = Adamax
Test accuracy: 0.864
Test loss: 0.3842434607744217