CSC 578-701 Loop Quiz #3
Lavinia Wang #1473704

1. Add comments for the following code in the function backprop() in "NN578_network.py".

```
 # backward pass

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

In this line, we call function cost_derivative() for partical derivative, which we pass into two parameters: output_activations and y. For output_activations, we have created variable called activations, which is a list to store all the activations, sigmoid(z), layer by layer. [-1] means the last element in the list, which represent the output layer activation. What this line of code means in mathematical representation is $\frac{\partial C}{\partial a}$.

```
sigmoid_prime(zs[-1])
```

Function sigmoid_prime is computing the derivative of the sigmoid function, in mathematical representation, $\frac{\partial a}{\partial z}$. Zs is a list to store all the z vectors, layer by layer. [-1] means the last element in the list, which represents the output layer. As a result, delta = $\frac{\partial C}{\partial a} \cdot \frac{\partial a}{\partial z}$. Sigmoid_prime returns a number. So here is vector and scalar multiplication. That's why we use *.

```
nabla_b[-1] = delta
```

nabla_b is [np.zeros(b.shape) for b in self.biases], which represents a list of biases initialized to 0 for b's shape. What this code does is replace the last bias with delta.

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

nabla_w is [np.zeros(w.shape) for w in self.weights], which is a list of weights initialized to 0 for w's shape. Here delta and activations are both vectors, and they might look like this. [x,x,…,x]. For vector multiplication, we need to transpose one of the factors, here is the activations. For computing the weights for last layer, as it is going backward, we need two layers of neurons. So [-2] is the second to the last layer. Both delta and activations are vector so we use dot for multiplication.

CSC 578-701 Loop Quiz #3
Lavinia Wang #1473704

2. In the single-neuron discussion at the start of this section, I argued that the cross-entropy is small if σ(z)≈y for all training inputs. The argument relied on y being equal to either 0 or 1. This is usually true in classification problems, but for other problems (e.g., regression problems) y can sometimes take values intermediate between 0 and 1. Show that the cross-entropy is still minimized when σ(z)=y for all training inputs. When this is the case the cross-entropy has the value:

$$C = -\frac{1}{n}\sum_x [y \ln y + (1-y)\ln(1-y)]. \qquad (64)$$

Solution:

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

y	a	c	comment
0.5	0.1	1.20397	
0.5	0.2	0.91629	
0.5	0.3	0.78032	
0.5	0.4	0.71356	
0.5	0.5	0.69315	This is minimum
0.5	0.6	0.71356	
0.5	0.7	0.78032	
0.5	0.8	0.91629	
0.5	0.9	1.20397	
0.5	1.0	error	
y	a	c	comment
0.01	0.9	2.28061	
0.01	0.7	1.19550	
0.01	0.5	0.69315	
0.01	0.3	0.36515	
0.01	0.1	0.12733	
0.01	0.05	0.08073	
0.01	0.03	0.06522	
0.01	0.01	0.05600	This is minimum
0.01	0.009	0.05606	
0.01	0.005	0.05795	

CSC 578-701 Loop Quiz #3
Lavinia Wang #1473704

3. I first run the original code provided in the provided basic classification file. The original model performed on the test data has accuracy of 0.8747. Ankle boot is 93% confidence(correct) and sandal is 93% confidence which should be sneaker.

My first experiment is the change of activation function on hidden layer. In the original code, activation function was *relu* and I replaced it with *sigmoid*. Everything else remained the same. The accuracy on test data slightly changed to 0.8715. The accuracy predicting ankle boot dropped a little to 87% confidence and it's predicting sneaker with 56%, which is correct.

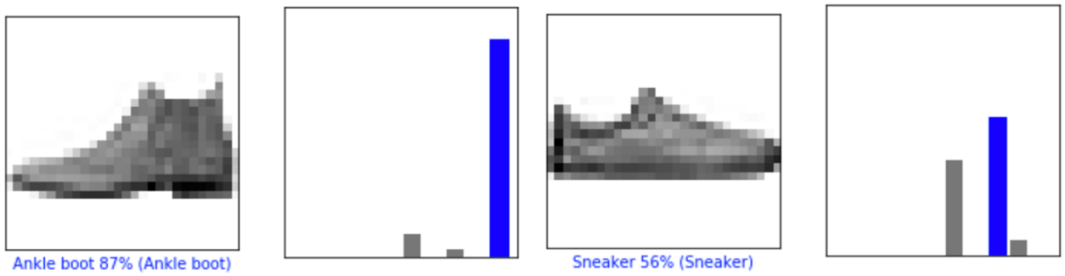

Figure 1 activation sigmoid

Then I changed this hidden layer activation function to *tanh* and everything else remained the same. This time, my accuracy of test data is 0.8738. The accuracy predicting ankle boot is 99% and 94% for sneaker, which is very ideal.

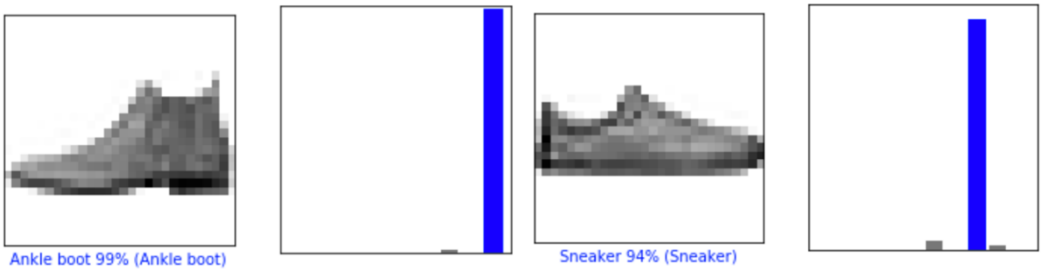

Figure 2 activation tanh

My second experiment is the change of optimizer. In the original code, optimizer was Adam and I replaced with Stochastic gradient descent optimizer with learning rate at 0.01 and momentum at 0.3. My accuracy of test data is 0.8449. The accuracy predicting ankle boot is 72% and 60% as sandal, which is wrong.

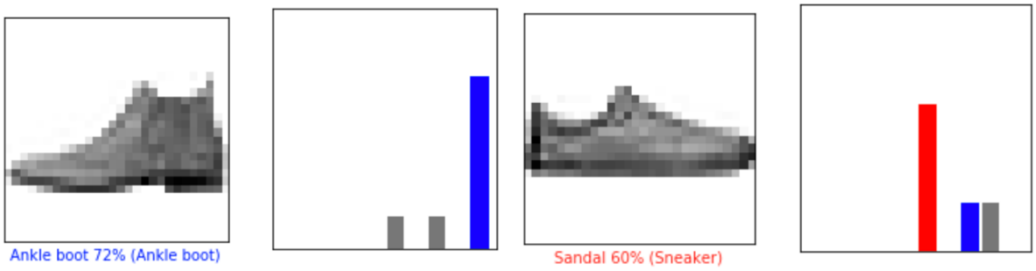

Figure 3 SGD optimizer learning rate at 0.01

This result made me wonder if the learning rate has a great impact on accuracy and its prediction. So I change the learning rate to 0.1 and this gives me an accuracy of 0.8704 on test

data. The accuracy predicting ankle boot is 94% and 45% for sneaker. The confidence for sneaker is low but the category is correct.

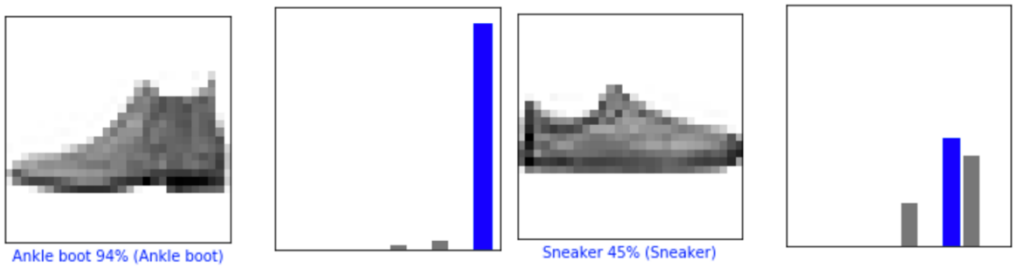

Figure 4 SGD optimizer learning rate at 0.1

My last experiment is the change of number of epochs. In the original code, there was 5 epochs. I changed activation function back to *relu* and optimizer to adam then increased epochs to 10. Now my accuracy on training data reached 0.9119 but only 0.876 on test data. It seems like I have an overfitting problem. I changed epochs to 15 to verify, and I have a larger gap between training data and test data. So 5 epochs is the maximum for training.

Based on the above experiments, I come up with a few assumptions: activation function on hidden layers may not have a huge impact on accuracy (i.e. 0.8747, 0.8715and 0.8738), but affect the classification outcome. Take the confidence level of sneaker as an example. *Relu* is highly confident about a wrong classification. *Sigmoid* is on the edge while *tanh* is highly confident about a correct classification. Learning rate also plays a vital role in SGD optimizer as the classification could change drastically from sandal to sneaker.