**hR**

1. Add comments for the following code in the function backprop() in "NN578\_network.py".

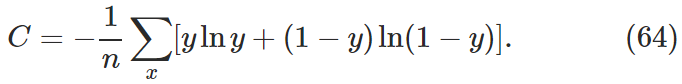
# backward pass  
 delta = self.cost\_derivative(activations[-1], y) \* \   
 sigmoid\_prime(zs[-1])   
 nabla\_b[-1] = delta  
 nabla\_w[-1] = np.dot(delta, activations[-2].transpose())

* Add comments for each line.
* Explain in detail what the line is doing, **meaning-wise**.  Explain what it is trying to accomplish and why, and how (by utilizing various variables in the network object).  Do NOT just what is happening syntactically (for example "a is multiplied by b").
* Include in your explanation on these points (REQUIRED):
  + Why the first line is using \* but the third line is using np.dot().
  + Why '.transpose()' in the last line is needed.

The backward propagation provides a way of computing the gradient cost function and using [-1] as the index for the last layer to count backwards.

* delta = self.cost\_derivative(activations[-1], y)\*sigmoid\_prime(zs[-1]) #Output layer Error where is computed here the gradient of the activation is multiplied with derivative of the sigmoid output.
  + The output layer is where the derivate of the error function and derivate of the sigmoid function is happening here. The sigmoid prime is applied to the last layers.
* nabla\_b[-1] = delta #Change in biases is taken by the delta backwards on the last layer and its on the biases.
  + Biases does not have any variables so it goes into the change directly
* nabla\_w[-1] = np.dot(delta, activations[-2].transpose()) #Change in the weights happen where you are computing the previous layers error where you are propagating the error
* multiplication is used because the values are scalar and the np.dot() is used with transpose() for multiplication because vector with matrix is being multiplied here

1. An [**Exercise**](http://neuralnetworksanddeeplearning.com/chap3.html#exercises_824189) on Cross-entropy cost function in NNDL Chapter 3. Specifically, the 2nd question only:

In the single-neuron discussion at the start of this section, I argued that the cross-entropy is small if (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 **(z) = y** for all training inputs. When this is the case the cross-entropy has the value:  


* For this problem, you can assume y as a single neuron (in the output layer) or a vector (i.e., the whole output layer).
* 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 compute the Cross-entropy value using that y and several varying 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.

Lets look at several values to prove that it is such a case when y=a then it is the minimal value of the cross-entropy. This shown below in the table when we use this expression to solve for Cross-entropy we get these values.

CE= -y\*np.log(a)-(1-y)\*np.log(1-a))

|  |  |  |
| --- | --- | --- |
| **y** | **a** | **CE** |
| 0.2 | 0.1 | 0.236606 |
|  | 0.2 | 0.217322 |
|  | 0.3 | 0.228497 |
|  | 0.4 | 0.257067 |
|  | 0.5 | 0.30103 |
|  | 0.6 | 0.362722 |
|  | 0.7 | 0.449283 |
|  | 0.8 | 0.578558 |
|  | 0.9 | 0.809151 |
| 0.5 | 0.1 | 0.522879 |
|  | 0.2 | 0.39794 |
|  | 0.3 | 0.33889 |
|  | 0.4 | 0.309894 |
|  | 0.5 | 0.30103 |
|  | 0.6 | 0.309894 |
|  | 0.7 | 0.33889 |
|  | 0.8 | 0.39794 |
|  | 0.9 | 0.522879 |
| 0.9 | 0.1 | 0.904576 |
|  | 0.2 | 0.638764 |
|  | 0.3 | 0.486081 |
|  | 0.4 | 0.380331 |
|  | 0.5 | 0.30103 |
|  | 0.6 | 0.239458 |
|  | 0.7 | 0.1917 |
|  | 0.8 | 0.157116 |
|  | 0.9 | 0.141182 |

Hence, proving that when a **= y** for all training inputs cross-entropy is minimized

, CE is minimized

1. A [**Problem**](http://neuralnetworksanddeeplearning.com/chap3.html#problem_712707) in NNDL Chapter 3,

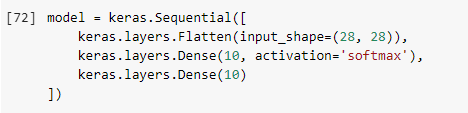
We've discussed at length the learning slowdown that can occur when output neurons saturate, in networks using the quadratic cost to train. Another factor that may inhibit learning is the presence of the xj term in Equation (61). Because of this term, when an input xj is near to zero, the corresponding weight wj will learn slowly. Explain why it is not possible to eliminate the xj term through a clever choice of cost function.

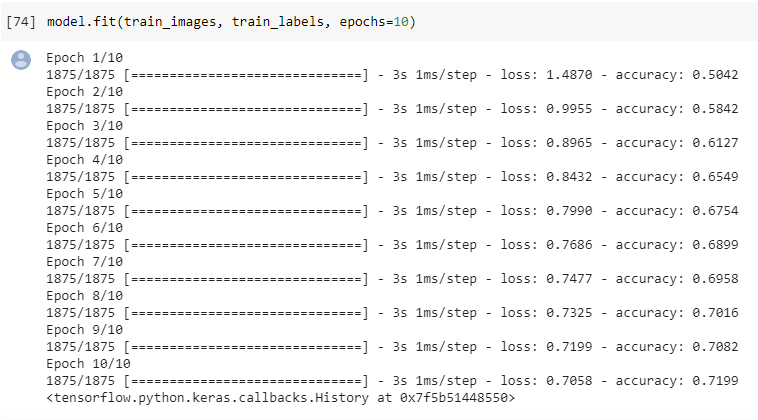
Whenever a partial derivate is taken using any of the activation function for the neural network. The cost function with respect to weight and the derivate of the weighted input always has an xj because of the chain rule. It is not possible to remove this xj due to the constraint that cost must be a function of the output activations. xj is the activation of the j node in the hidden layer.

4.

**Experimenting with hidden layers. I expect as I reduce the number of the hidden layers the accuracy drops but it calculates faster.**

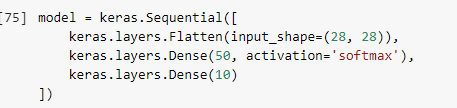
While choosing 10 for hidden layer with the activation at softmax as mentioned in the preliminary

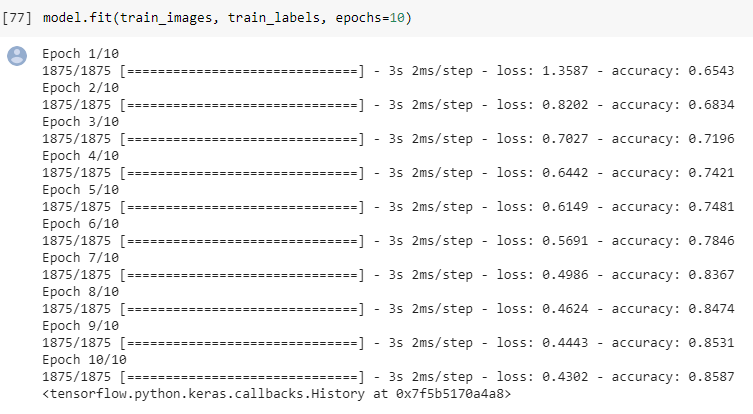




The model reaches about 72% accuracy.

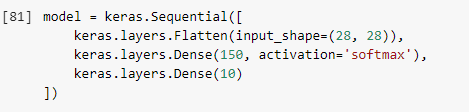
Modifying this number to 50 hidden layer should increase the accuracy and decrease the time as well.

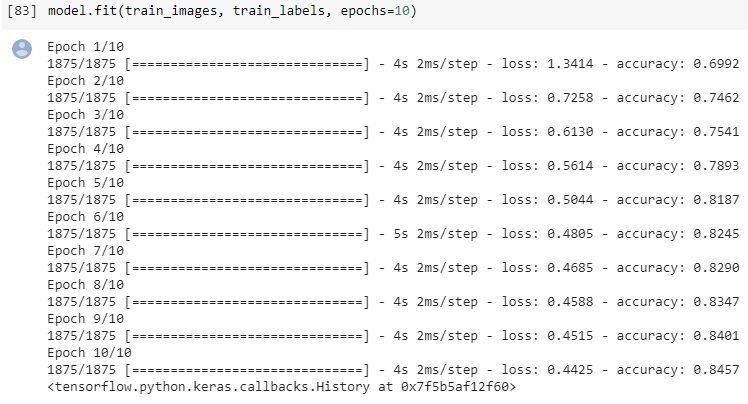




Accuracy goes up to 85% and the step size increased to 2m/s

Moving the hidden layers to 150 should fairy improved accuracy but also increase the stepsize time.





Observing from the increase in the hidden layers to 150 didn’t allow the model to get the accuracy to higher but decreased from the previous 50 hidden layers. It does seem like it requires more time to learn for many hidden layers.

The next experiment is to decrease the hidden layers to get optimum result. The choice is 200 hidden layers.



It seems like with the just 10 epochs it is not just enough to see the accuracy improved. It is hard to establish what is the right amount of hidden layers and it functions very randomly. But Can make an estimated guess.

**Experimentation on different activation functions of hidden layers**

|  |  |
| --- | --- |
| Activation | Max Accuracy at 10 Epocs on 128 hidden layers |
| Sigmoid | 0.9067 |
| Relu | 0.9106 |
| Tanh | 0.9112 |
| softmax | 0.8410 |

Tanh performed good at the different experiments.

**Performing experimentation on batch size. The expectation is that as the batch size increases the accuracy should increase. The configuration of tanh is used for activation and 50 hidden layers and 10 epocs**

|  |  |
| --- | --- |
| Batch\_size | Accuracy |
| 20 | 0.905 |
| 25 | 0.9037 |
| 30 | 0.9026 |
| 35 | 0.9024 |
| 40 | 0.9112 |

Looking at the results I see that at size of 40 the accuracy is at 91% which is good. But it did decrease from 30 batch size to 35 batch size.

**Extermination on Regularizes to see how they effect the model differently. Using the similar configuration from the last experiment with having 40 as batch size and 10 epochs with tanh activation I expect to see shorter timings for**

|  |  |  |
| --- | --- | --- |
| L1 | L2 | Accuracy |
| 0.01 | 0.01 | 0.8375 |
| 0.02 | 0.02 | 0.8271 |
| 0.03 | 0.03 | 0.8224 |
| 0.04 | 0.04 | 0.8228 |

Looking at these values. It seems as the regularization increases in values the accuracy also decreases and even the time increased for the last data experiment.