

## Exercise 1: Upgrade Assignment 2 code to train & test k-layer networks

To make the code more dynamic I changed my loops and static variables/functions into objects, namely Network and Layer. A network was initialized with its training parameters and then layer was added to the network, with given activation function and nodes in and out.

```
myNetwork = Network();  
  
myNetwork.putLayer(LayerObject("RELU",3072,50))  
myNetwork.putLayer(LayerObject("SOFTMAX",50,10))  
cost, acc = myNetwork.feedForwardEvaluate(inputData, realLabel)  
print("Cost:",cost)  
print("Accuracy:",acc)  
myNetwork.calcGradients(inputData, realLabel)  
myNetwork.updateWeights()
```

This both made the code more readable and helped me test the layers more efficiently. I was assured that the functions were correctly implemented since I got the same cost and accuracy as in assignment 2. Now, more research on the gradients were needed. I did the same as before, computing only for the first 100 data points using lambda value of 0.

### 2 Layer

```
The relative error of First layer weights is 0.000176580025562  
The relative error of First layer bias is 0.000305356888981  
The relative error of Second layer weights is 6.07498545343e-08  
The relative error of Second layer bias is 7.02870709609e-06
```

### 3 Layer

```
The relative error of First layer weights is 0.000288290815218  
The relative error of First layer bias is 0.000287105670434  
The relative error of Second layer weights is 2.32598567545e-05  
The relative error of Second layer bias is 0.002274799225270  
The relative error of Third layer weights is 7.12311400764e-07  
The relative error of Third layer bias is 7.03361199912e-06
```


### 4 Layer

```
The relative error of First layer weights is 0.117655752889000  
The relative error of First layer bias is 0.001720435059170  
The relative error of Second layer weights is 0.008201179742860  
The relative error of Second layer bias is 0.022842117340000  
The relative error of Third layer weights is 0.000377128146086  
The relative error of Third layer bias is 2.46287689477e-07  
The relative error of Fourth layer weights is 8.85938941184e-07  
The relative error of Fourth layer bias is 7.03285172821e-06
```

Note: I used my original code to compute these gradients but for some reason, it seemed like they rose considerably with each layer added. I wasn't really sure why this was happening until I read that

*"The discrepancy between the analytic and the numerical gradients increases for the earlier layers as the gradient is back-propagated through the network."*

# discrepancy

/dis'krɛp(ə)nsi/ 

*noun*

an illogical or surprising lack of compatibility or similarity between two or more facts.

Which seemed to explain these huge, rising errors in my network. Since I managed to compute the same starting cost and accuracy with the old method network and my upgraded version, getting the same results in both cases (the same as in assignment 2) I built my theory on that my gradients were correctly calculated. Due note that the structure and the functions of my code didn't change much. I only created the batch normalization function which only added a little bit of value manipulation.

## Exercise 2: Can I train a 3-layer network?

After changing the structure of my code (and debugging a lot) I managed to get the exact same results using the same values and training for a few epochs

```
Info on network with lambda=8.248206928786062e-05 epochs=5 batch=500 eta=0.29189273799480775
Info on Training data: Accuracy: 50.28% and Cost: 1.42
Info on Validation data: Accuracy: 41.88% and Cost: 1.7
Info on Testing data: Accuracy: 41.84% and Cost: 1.66
```

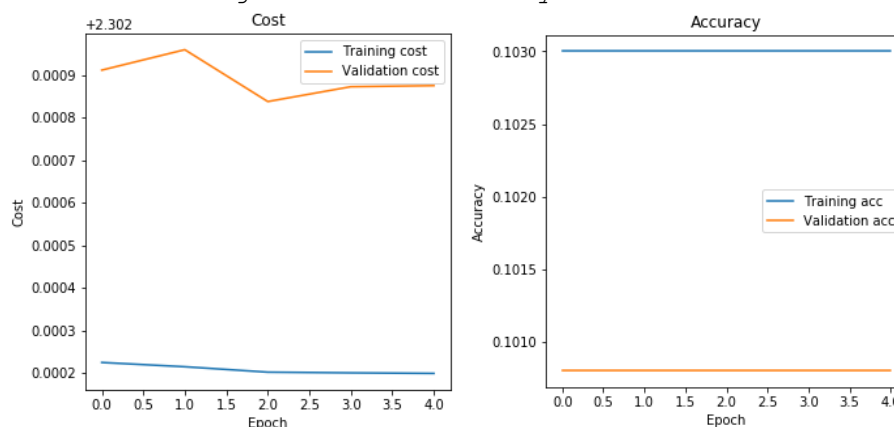
### Info on my old code

```
Network Eta:0.29189273799480775 epoch: 5 batch: 500 lambdaVal: 8.248206928786062e-05 momentum: 0.9
Info on Training data: Accuracy: 50.28% and Cost: 1.42
Info on Validation data: Accuracy: 41.88% and Cost: 1.7
Info on Testing data: Accuracy: 41.84% and Cost: 1.66
```

### Upgraded code, same results with two layer network

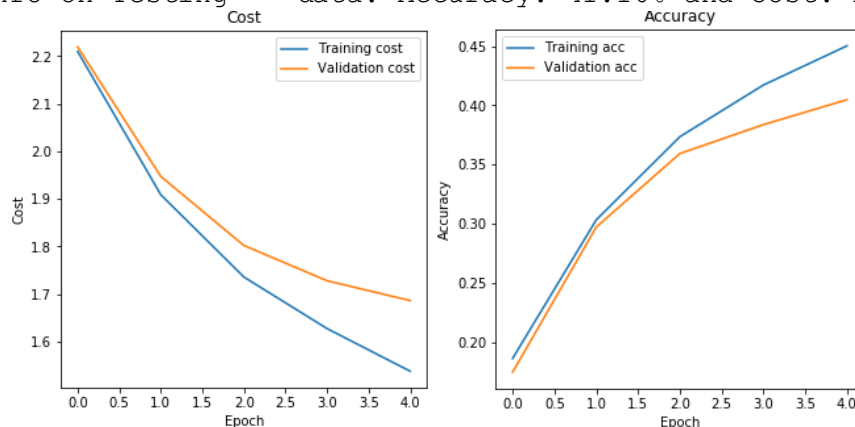
When I tried out my first initial training with an extra layer with 30 hidden nodes, this happened

```
Network Eta:0.15 epoch: 5 batch: 500 lambdaVal: 0 momentum: 0.90
Info on Training data: Accuracy: 10.30% and Cost: 2.30
Info on Validation data: Accuracy: 10.08% and Cost: 2.30
Info on Testing data: Accuracy: 10.00% and Cost: 2.30
```



Needless to say, the network didn't learn anything at all and didn't progress with its accuracy even though the cost lowered by a small number. Fortunately, I implemented He initialization in the former assignment and using it here solved my problems.

Network Eta:0.15 epoch: 5 batch: 500 lambdaVal: 0 momentum: 0.90  
 Info on Training data: Accuracy: 45.03% and Cost: 1.54  
 Info on Validation data: Accuracy: 40.48% and Cost: 1.69  
 Info on Testing data: Accuracy: 41.16% and Cost: 1.66



Nice to see how such a small difference can have a big impact. This means, my setup is trainable.

### Exercise 3: Implement batch normalization

To implement the batch normalization I had to modify my class structure a little bit. An extra parameter was introduced in the layer that specified if the output of the layer should be normalized before pushed to the activation function, looking like this.

```
basicNetwork = Network(eta=0.15, epoch=15, batch=500, lambdaVal=0, momentum=0.9);
basicNetwork.putLayer(LayerObject("RELU", 3072, 50, True))
basicNetwork.putLayer(LayerObject("RELU", 50, 30, True))
basicNetwork.putLayer(LayerObject("SOFTMAX", 30, 10, False))
```

I also had to keep track of the moving average mean and variance which was implemented in the *batchForward* function. It doesn't look pretty but it got the job done, since I had engineered my class architecture in a problematic way in the beginning. In hindsight, the batch normalization should have been implemented as its own layer. By comparing my current code and the ones from the former assignments, doing it this way would have saved me a lot of trouble.

### Passing assignment 3 criteria

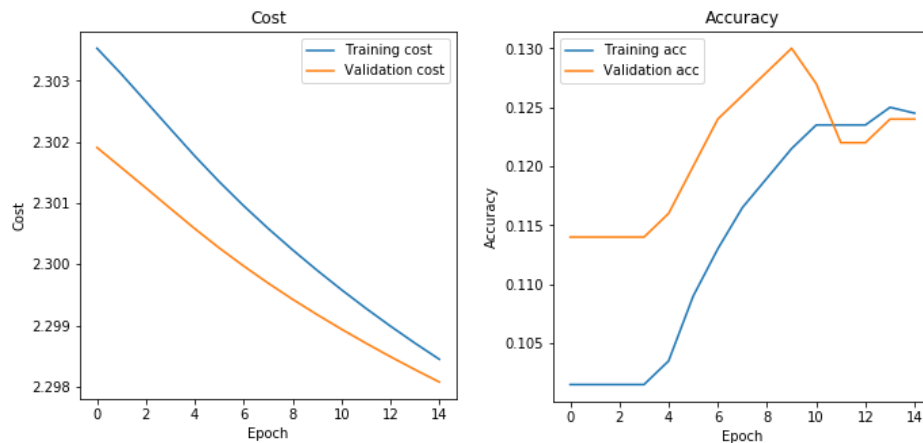
*State how you checked your analytic gradient computations (with some accompanying numbers) and whether you think that your gradient computations are bug free for your  $k$ -layer network with batch normalization.*

See Exercise 1 on top of the paper.

*Include graphs of the evolution of the loss function when you tried to train your 3-layer network without batch normalization and with batch normalization.*

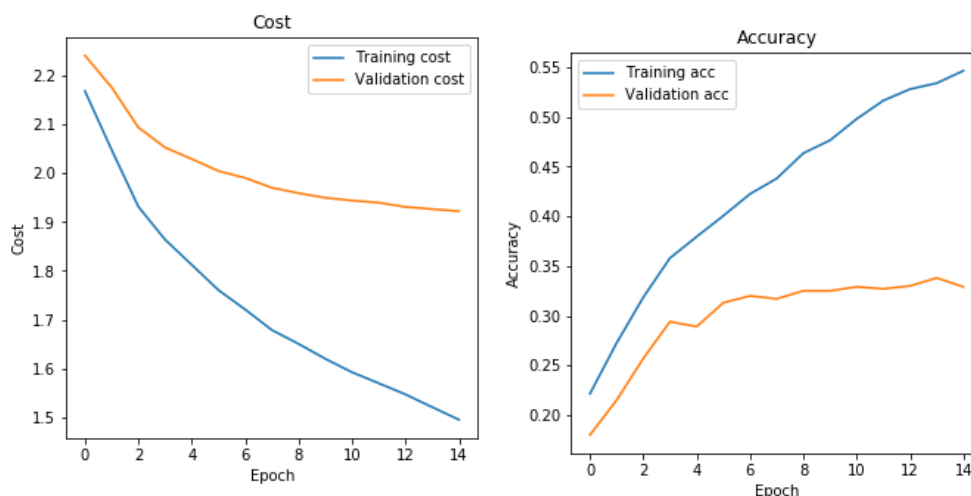
#### Without using batch normalization

Network Eta:0.01 epoch: 15 batch: 500 lambdaVal: 0 momentum: 0.9  
 Info on Training data: Accuracy: 12.45% and Cost: 2.30  
 Info on Validation data: Accuracy: 12.40% and Cost: 2.30  
 Info on Testing data: Accuracy: 12.58% and Cost: 2.30



### Using batch normalization

Network Eta:0.01 epoch: 15 batch: 500 lambdaVal: 0 momentum: 0.9  
 Info on Training data: Accuracy: 54.65% and Cost: 1.49  
 Info on Validation data: Accuracy: 32.90% and Cost: 1.92  
 Info on Testing data: Accuracy: 33.50% and Cost: 1.90



This indicates that the network learns way faster by implementing batch normalization and gets way better performance as well. Note that the non-batch normalization network actually fails to train. The network will now be tested for appropriate values and finally, trained on all the data to get proper results.

*State the range of the values you searched for  $\lambda$  and  $\eta$ , the number of epochs used for training during the fine search, and the hyper-parameter settings for your best performing 3-layernetwork you trained with batch normalization. Also state the test accuracy achieved by network.*

For the course search I limited my training set to only half of the batch to get quicker results with these initial values

COURSE'EPOCH = 5  
 COURSE'BATCH = 500  
 COURSE'MOMENTUM = 0.9  
 COURSE'DECAY = 0.95

To find appropriate  $\lambda$  and learning rate for the network, I created a small array with the following values to pick the best values to fine search for.

```
COURSE`LAMBDA` = [0.3, 0.15, 0.03, 0.015, 0.003, 0.0015, 0.0003, 0.00015, 0.00003]
COURSE`LEARNRATE` = [0.3, 0.15, 0.03, 0.015, 0.003, 0.0015, 0.0003, 0.00015, 0.00003]
```

After looping through all of these values, I got

```
Network Eta: 0.003 epoch: 5 batch: 500 lambdaVal: 0.003 momentum: 0.9
Testing dataset cost: 1.91005976498
Testing dataset accuracy: 36.93%
```

```
Network Eta: 0.015 epoch: 5 batch: 500 lambdaVal: 3e-05 momentum: 0.9
Testing dataset cost: 1.9243807677
Testing dataset accuracy: 36.61%
```

```
Network Eta: 0.015 epoch: 5 batch: 500 lambdaVal: 0.00015 momentum: 0.9
Testing dataset cost: 1.92595580684
Testing dataset accuracy: 36.56%
```

Thus for the fine search I narrowed my values down to these parameters by running 50 loops with random generated values for lambda and the learning rate, namely

```
FINE`LAMBDA`MIN = 3e-05
FINE`LAMBDA`MAX = 0.003
```

```
FINE`LEARNRATE`MIN = 0.003
FINE`LEARNRATE`MAX = 0.015
```

I also increased the dataset trained on by 50% (7500 data points) and the epochs to 10. My best three networks were the following

```
Network Eta: 0.01094904727469526 epoch: 10 batch: 500 lambdaVal:
0.00026372195718672053 momentum: 0.9
Testing dataset cost: 1.94
Testing dataset accuracy: 37.84%
```

```
Network Eta: 0.004689144729781953 epoch: 10 batch: 500 lambdaVal:
0.00028532908363981943 momentum: 0.9
Testing dataset cost: 1.76
Testing dataset accuracy: 38.12%
```

```
Network Eta: 0.010062184458341922 epoch: 10 batch: 500 lambdaVal:
0.0014561487876284364 momentum: 0.9
Testing dataset cost: 1.84
Testing dataset accuracy: 38.54%
```

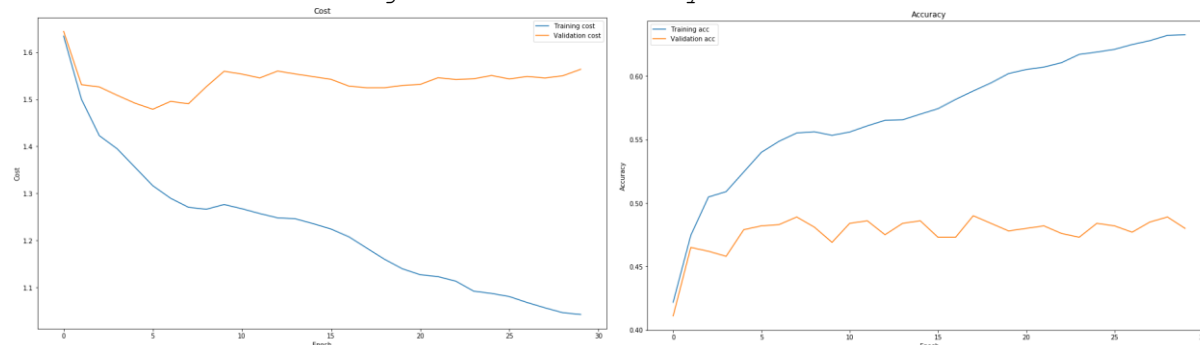
So finally training on my last, best network on the first batch with normalization

Info on network with  $\lambda=8.248206928786062e-05$  epochs=30 batch=500  $\eta=0.29189273799480775$  momentum: 0.9

Info on Training data: Accuracy: 63.27% and Cost: 1.04

Info on Validation data: Accuracy: 48.00% and Cost: 1.56

Info on Testing data: Accuracy: 48.34% and Cost: 1.58



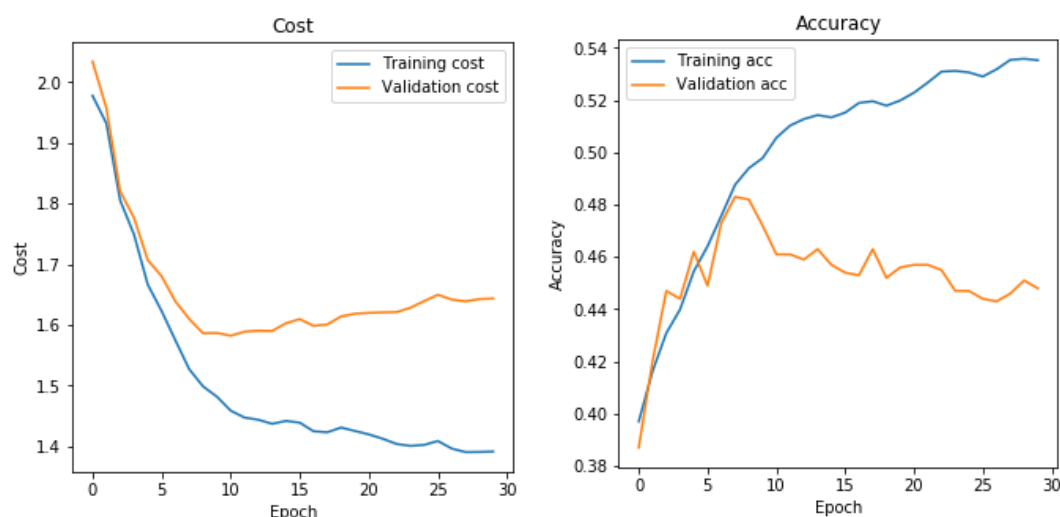
And finally, training on all of the batches to see how well it performs (and for science)

Network  $\eta=0.010062184458341922$  epoch: 30 batch: 500  $\lambda_{\text{Val}}=0.0014561487876284364$  momentum: 0.9

Info on Training data: Accuracy: 53.26% and Cost: 1.40

Info on Validation data: Accuracy: 53.10% and Cost: 1.39

Info on Testing data: Accuracy: 44.89% and Cost: 1.65



Now I must say I expected higher accuracy on this badboy but I guess we could use different methods from the optimization part to test that further or add more layers to it.

*Plot the training and validation loss for your 2-layer network with batch normalization with 3 different learning rates (small, medium, high) for 10 epochs and make the same plots for a 2-layer network with no batch normalization.*

Now for the last part I considered small and high learning rate as a percentage of my optimal learning rate. This means that for small learning rate, I divide my optimal one with 100 and multiply with 100 for the high one.

From our previous results, it seemed like batch normalization helped us train using higher learning rate and allowed us to reach an optimal solution faster. It seemed like it surely helped the two layer networks when using way to high or low learning rate. In the following, the general trend is that the batch normalized network is able to reach higher performances, often in a shorter amount of iterations. The results and plots that support my statement are located on the next 2 pages.

WITHOUT BATCH NORMALIZATION

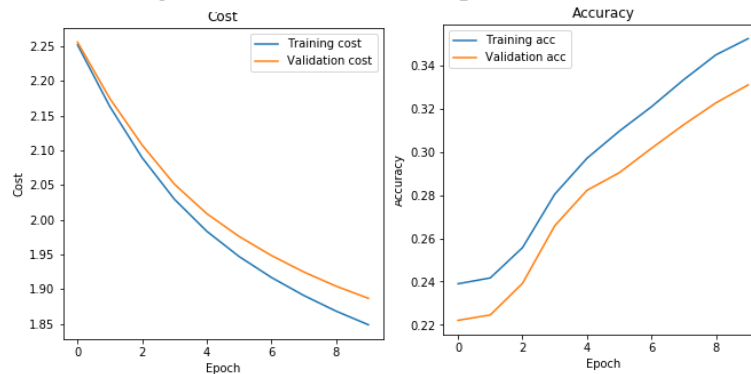
## NORMAL LEARNING RATE

Network Eta:0.010062184458341922 epoch: 10 batch: 500 lambdaVal: 0.0014  
561487876284364 momentum: 0.9

Info on Training data: Accuracy: 35.44% and Cost: 1.84

Info on Validation data: Accuracy: 35.00% and Cost: 1.90

Info on Testing data: Accuracy: 33.95% and Cost: 1.87



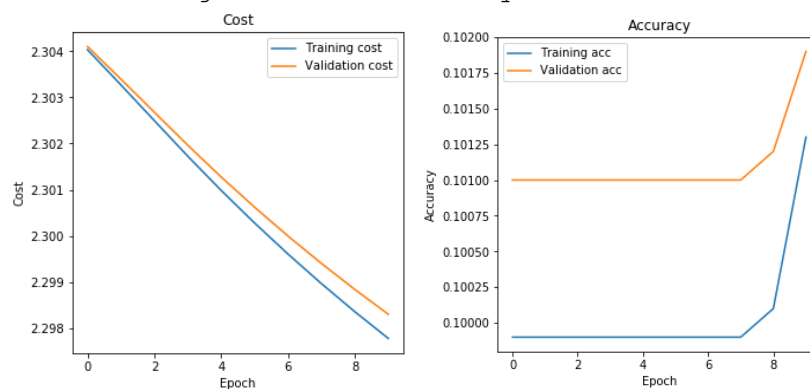
## SMALL LEARNING RATE

Network Eta:0.00010062184458341922 epoch: 10 batch: 500 lambdaVal: 0.00  
14561487876284364 momentum: 0.9

Info on Training data: Accuracy: 10.54% and Cost: 2.3

Info on Validation data: Accuracy: 11.00% and Cost: 2.3

Info on Testing data: Accuracy: 10.12% and Cost: 2.3



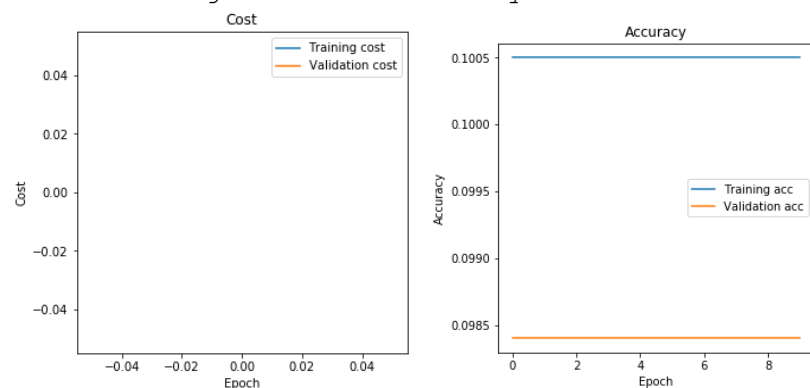
## HIGH LEARNING RATE

Network Eta:1.0062184458341923 epoch: 10 batch: 500 lambdaVal: 0.001456  
1487876284364 momentum: 0.9

Info on Training data: Accuracy: 10.1% and Cost: nan

Info on Validation data: Accuracy: 10.2% and Cost: nan

Info on Testing data: Accuracy: 10.0% and Cost: nan

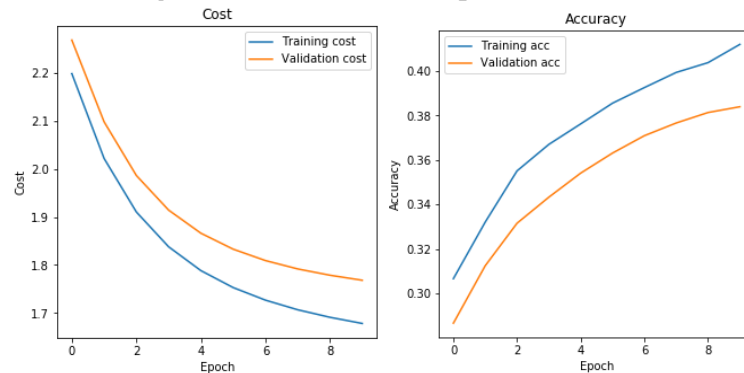


WITH BATCH NORMALIZATION

## NORMAL LEARNING RATE

Network Eta:0.010062184458341922 epoch: 10 batch: 500 lambdaVal: 0.0014  
561487876284364 momentum: 0.9

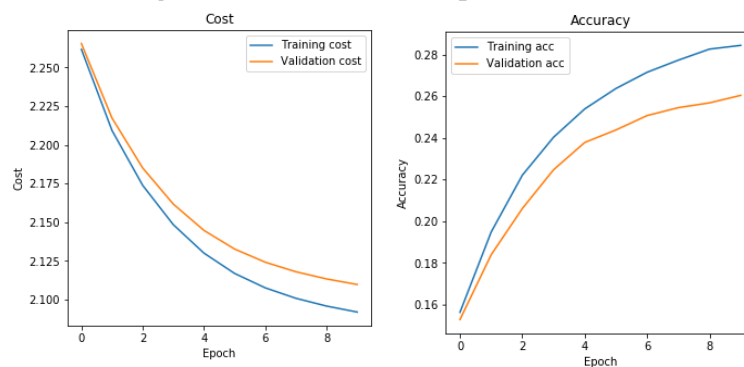
Info on Training data: Accuracy: 41.68% and Cost: 1.67  
Info on Validation data: Accuracy: 39.70% and Cost: 1.80  
Info on Testing data: Accuracy: 39.05% and Cost: 1.73



## SMALL LEARNING RATE

Network Eta:0.00010062184458341922 epoch: 10 batch: 500 lambdaVal: 0.00  
14561487876284364 momentum: 0.9

Info on Training data: Accuracy: 28.82% and Cost: 2.09  
Info on Validation data: Accuracy: 27.60% and Cost: 2.10  
Info on Testing data: Accuracy: 27.16% and Cost: 2.10



## HIGH LEARNING RATE

Network Eta:1.0062184458341923 epoch: 10 batch: 500 lambdaVal: 0.001456  
1487876284364 momentum: 0.9

Info on Training data: Accuracy: 46.40% and Cost: 1.61  
Info on Validation data: Accuracy: 39.90% and Cost: 1.83  
Info on Testing data: Accuracy: 39.50% and Cost: 1.78

