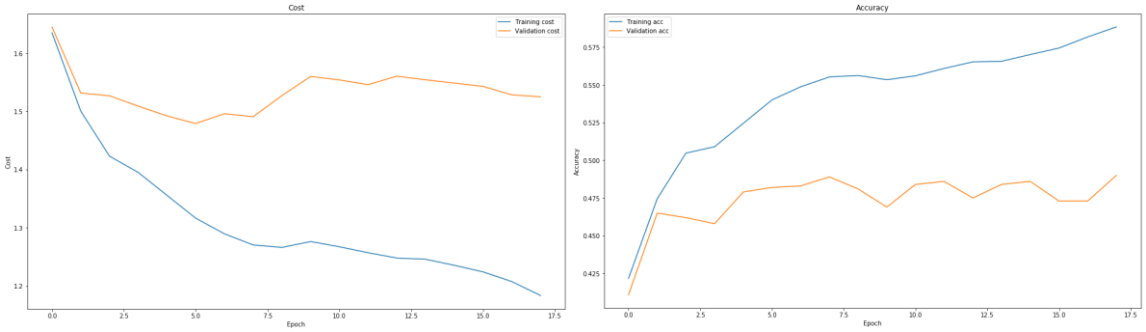
# 1. Optimize the performance of the network.

Tested all the optimizations on the same best parameters as I used in basic assignment. To see the effects of these optimizations I only trained on the first 10 epochs for demonstration with these values for the parameters.

```
batch=500 decay=0.95 momentum=0.9
Lambda=8.248206928786062e-05
Eta=0.29189273799480775
```

*(a) Use all the available data from training, train for more update steps and use your validation set to make sure you don't overfit or keep a record of the best model before you begin to overfit.*

Using the information from the former report (without optimization) I knew where the network got overfitted to the training data. Thus I stopped earlier training on all the data, at epoch 18



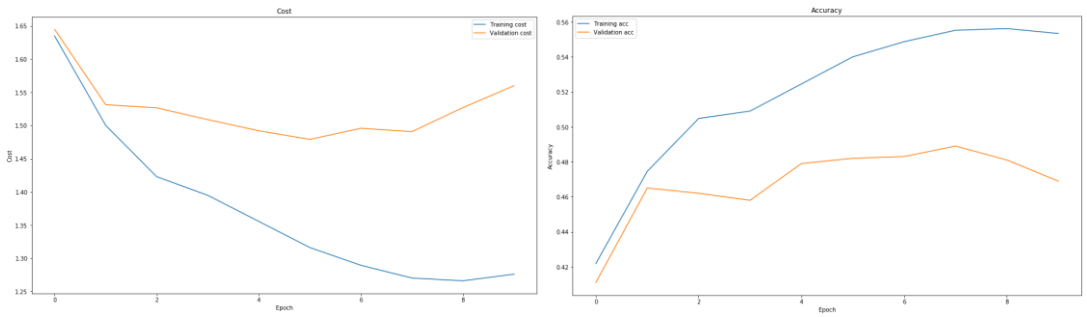

```
Info on Training   data: Accuracy: 58.83% and Cost: 1.18
Info on Validation data: Accuracy: 49.00% and Cost: 1.52
Info on Testing    data: Accuracy: 48.06% and Cost: 1.56
```

This is nearly the same result as training for 30 epochs, so early stopping saves us some computation time!

*(b) Switch to He initialization and see the effect it has on training.*

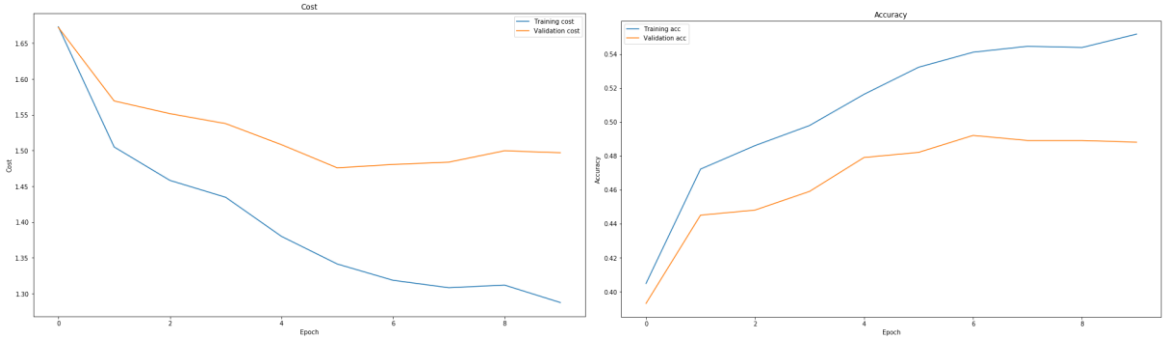
To check out the effects of this type of initialization (note: different from last assignment's Xavier init) I ran the network with the same parameters that I found from the fine search. Needless to say, here are my findings.

Without HE initialization, using variance 0.001




```
Info on Training   data: Accuracy: 55.33% and Cost: 1.28
Info on Validation data: Accuracy: 46.90% and Cost: 1.56
Info on Testing    data: Accuracy: 47.17% and Cost: 1.54
```

Using the HE initialization



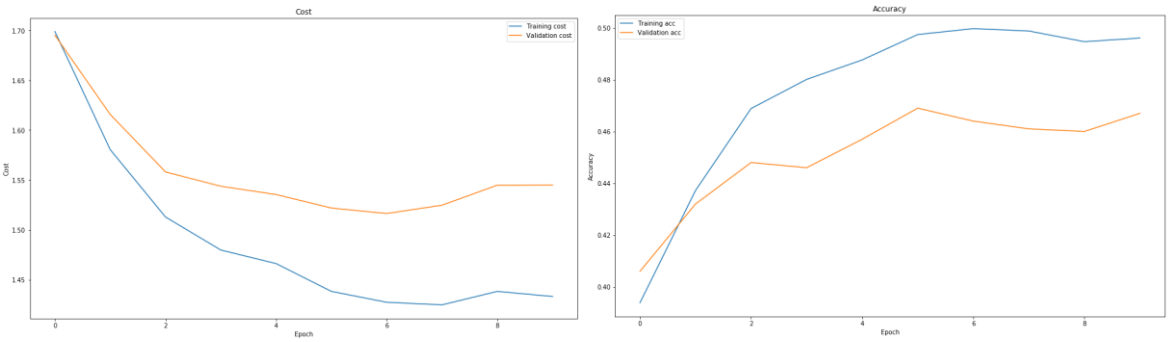

```
Info on Training   data: Accuracy: 55.17% and Cost: 1.29
Info on Validation data: Accuracy: 48.80% and Cost: 1.50
Info on Testing    data: Accuracy: 47.04% and Cost: 1.54
```

Using the HE initialization was a big improvement. We can see that correctness of the validation data follows the training data a little better and the cost of the validation doesn't rise as high as before. Picking a variance for the randomization surely helps the network.

*(c) You could also explore whether having more hidden nodes improves the final classification rate. One would expect that with more hidden nodes then the amount of regularization would have to increase.*

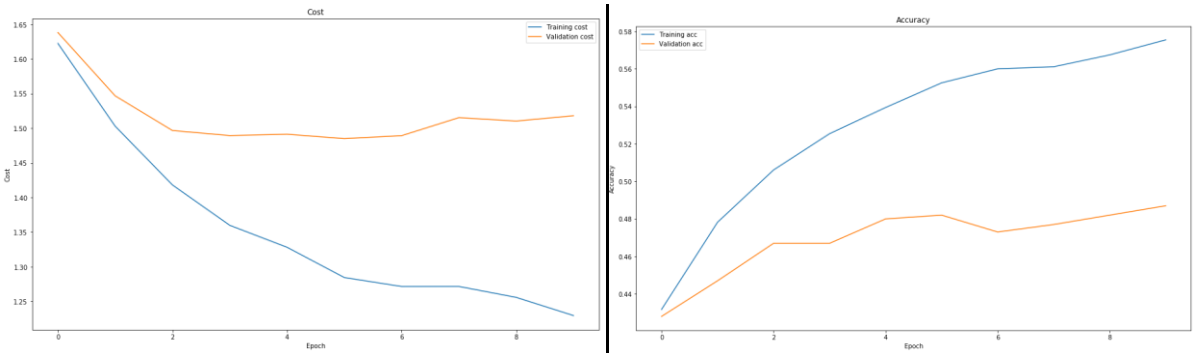
This required a simple for loop for testing out different sizes of the hidden layer. Unfortunately, there is no great theory on how many nodes you should use. What we have to keep in mind is the bias-variance dilemma to find that sweet spot where their respective error doesn't rise at all-time high. Since I have no knowledge of how much I should increase the regularization in comparison with the nodes, I decided to increase it by percentage of added nodes. Thus, going from 50 to 100 nodes would double my lambda value. Here are my results

```
25 nodes
```



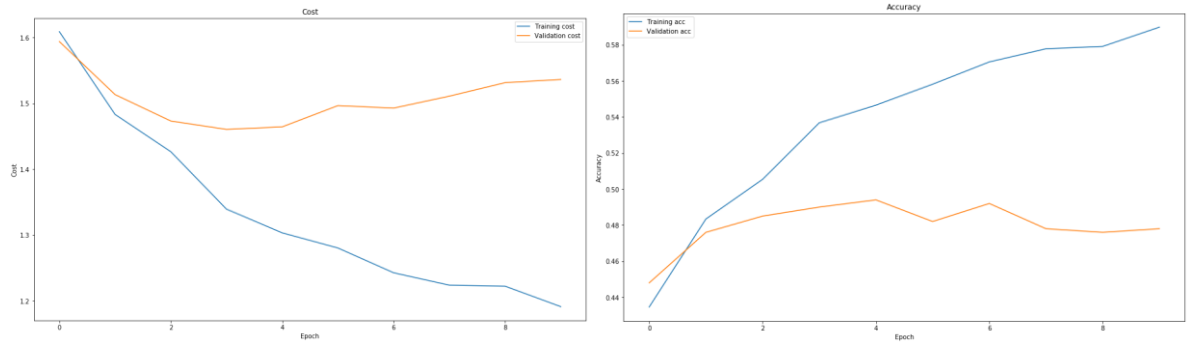

```
Info on Training   data: Accuracy: 49.61% and Cost: 1.43
Info on Validation data: Accuracy: 46.70% and Cost: 1.54
Info on Testing    data: Accuracy: 44.37% and Cost: 1.61
```

75 nodes




```
Info on Training   data: Accuracy: 57.54% and Cost: 1.23
Info on Validation data: Accuracy: 48.70% and Cost: 1.52
Info on Testing    data: Accuracy: 48.70% and Cost: 1.55
```

100 nodes




```
Info on Training   data: Accuracy: 58.98% and Cost: 1.19
Info on Validation data: Accuracy: 47.80% and Cost: 1.54
Info on Testing    data: Accuracy: 49.43% and Cost: 1.53
```

I kept my tests for 100 nodes since it took such a long time computing the 10 epochs for these values. I also decided to stop since seeing the plots, one can tell that the accuracy of the training data is increasing but the validation isn't. Increasing the nodes did give a small boost, with 75 giving the cleanest result, 48.70% on both validation and the test.

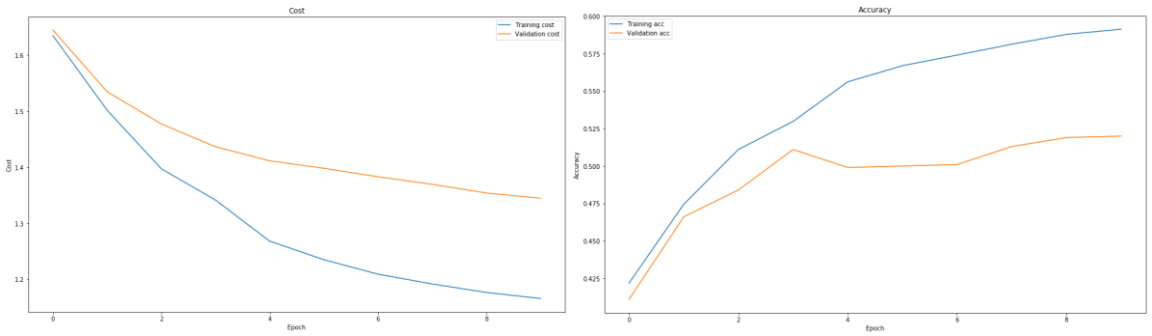
*(e) Play around with different approaches to anneal the learning rate. For example you can keep the learning rate fixed over several epochs then decay it by a factor of 10 after n epochs.*
I decided to test out Exponential Decay for this week's assignment.

$$\eta = \eta_0 e^{-kt}$$

Or in my code eta =`0.29189273799480775*np.exp(-1*k*epoch)`

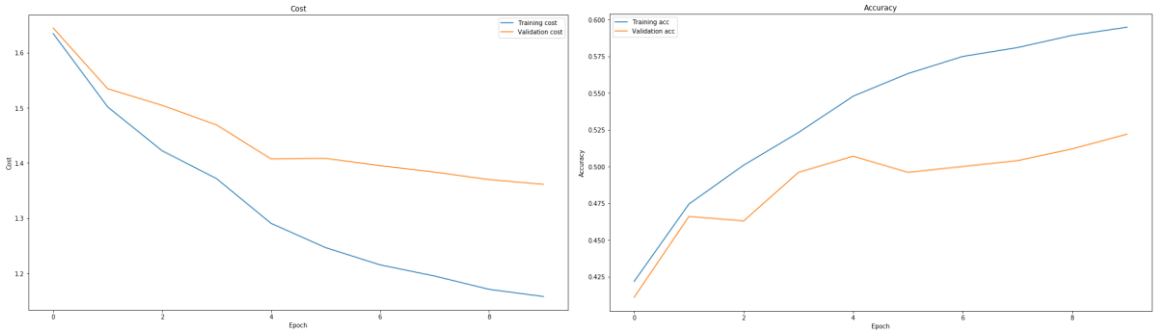
Since there isn't much info on how much the learning rate should be lowered by, I followed a graph online on how fast the values get lowered. These are the results.

k = 0.4



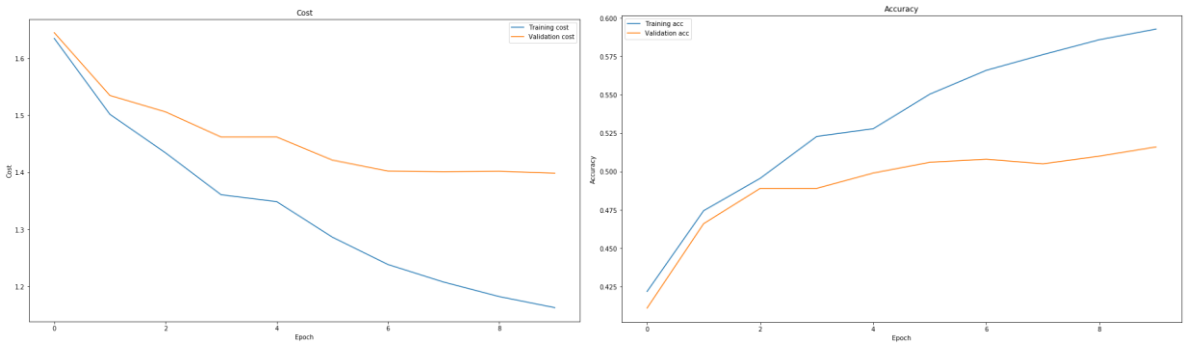

```
Info on Training   data: Accuracy: 59.13% and Cost: 1.17
Info on Validation data: Accuracy: 52.00% and Cost: 1.34
Info on Testing    data: Accuracy: 51.57% and Cost: 1.39
```

k = 0.3




```
Info on Training   data: Accuracy: 59.48% and Cost: 1.16
Info on Validation data: Accuracy: 52.20% and Cost: 1.36
Info on Testing    data: Accuracy: 51.12% and Cost: 1.40
```

k = 0.2




```
Info on Training   data: Accuracy: 59.28% and Cost: 1.16
Info on Validation data: Accuracy: 51.60% and Cost: 1.40
Info on Testing    data: Accuracy: 50.39% and Cost: 1.43
```

As one can see, controlling the learning rate decaying with a more advanced feature clearly did improve my accuracy. K as 0.3 gave me my best result with the lowest average cost on all data sets. According the lectures, this is how the cost plot should look like, even though the curve could be a little steeper.

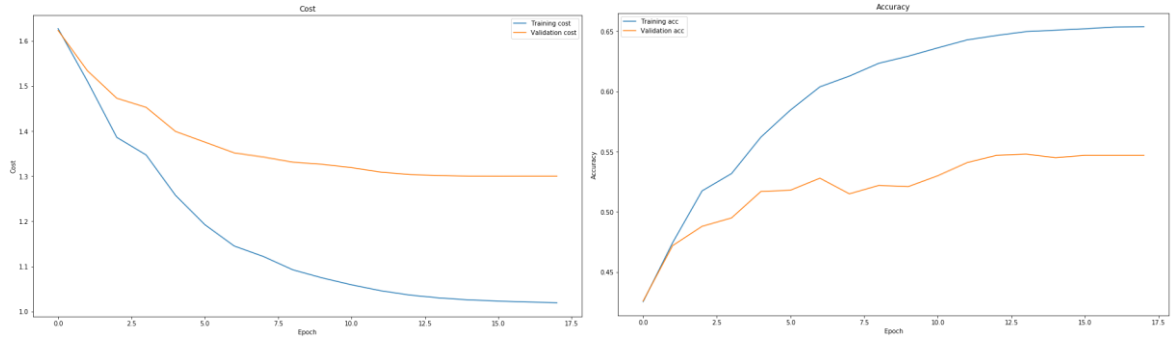
**Final version**

Putting all of those best 4 optimizations values together, that is, using all data, exponential decay of 0.31, HE initialization and using 80 nodes in the hidden layer gave me these results.

| Epoch | Training Accuracy | Validation Accuracy |
|---|---|---|
| 0 | 0.42 | 0.426 |
| 1 | 0.47 | 0.472 |
| 2 | 0.51 | 0.488 |
| 3 | 0.53 | 0.495 |
| 4 | 0.56 | 0.517 |
| 5 | 0.58 | 0.518 |
| 6 | 0.60 | 0.528 |
| 7 | 0.61 | 0.515 |
| 8 | 0.62 | 0.522 |
| 9 | 0.62 | 0.521 |
| 10 | 0.63 | 0.53 |
| 11 | 0.64 | 0.541 |
| 12 | 0.64 | 0.547 |
| 13 | 0.64 | 0.548 |
| 14 | 0.65 | 0.545 |
| 15 | 0.65 | 0.547 |
| 16 | 0.65 | 0.547 |
| 17 | 0.65 | 0.547 |
| 18 | 0.65 | 0.548 |
| 19 | 0.65 | 0.546 |
| 20 | 0.65 | 0.546 |
| 21 | 0.65 | 0.546 |
| 21 | 0.65 | 0.544 |

From these plots it seemed like my network should early stop at 18 epochs so that's what I did, ending with these plots.

```
Info on network with epochs=18 batch=500 exponential decay=0.31 80
hidden nodes and He init
Lambda=0.00013197131086057699
Eta=0.29189273799480775
```




```
Info on Training   data: Accuracy: 65.38% and Cost: 1.02
Info on Validation data: Accuracy: 54.70% and Cost: 1.30
Info on Testing    data: Accuracy: 52.66% and Cost: 1.37
```

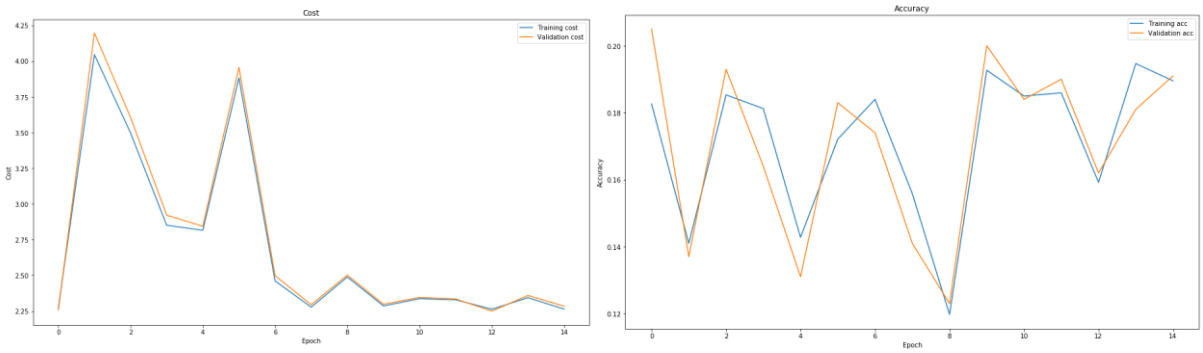
I'm unsure why I ended up with 54.70% in this run while I ended up in 54.80% in the former one. For this run, I hit 54.80% accuracy at 13 epochs. I should add the early stopping as a code and keep a record of the best weights.

Note: Here would be a perfect time to test out different combinations of all of these values with coarse and fine search. I leave that up to my free time since time is of the essence.

## 2. Train network using a different activation to ReLu

I wanted to try out LeakyRelu first but I wasn't sure that it would count as 2 extra points since it is so similar, or at least pretty similar. Thus I implemented ArcTan (can be found in the code as TanhForward, calculateGradientTanh and trainTanh) to see the effects. These are my outcomes.

```
Info on network with epochs=15 batch=500 momentum 0.9 decay 0.95
Lambda=8.248206928786062e-05
Eta=0.29189273799480775
```




```
Info on Training   data: Accuracy: 18.95% and Cost: 2.26
Info on Validation data: Accuracy: 19.10% and Cost: 2.28
Info on Testing    data: Accuracy: 18.68% and Cost: 2.27
```

The network manages to learn by hard start and gradually reduce the average cost but it doesn't performs nearly as well as using the ReLU. This might be optimized if the weight initialization and learning rate would be done differently.