This week's assignment was about improving the network by introducing another layer to the network using ReLu activation in the hidden layer so the system could learn more complex features and higher correlations. We were also to implement and explore the effects of momentum and batch normalization and comment on how those methods could improve the learning speed of the network.

Batch Normalization

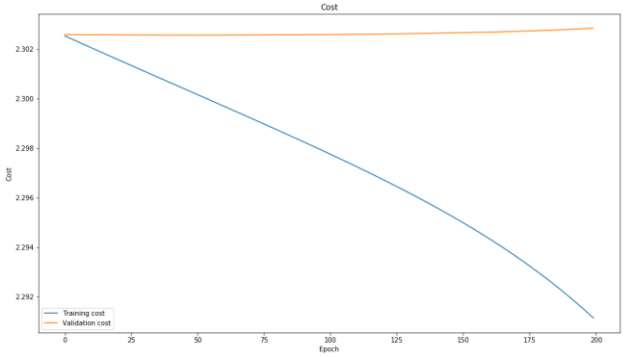
A little bit of preprocessing was needed on the data before the training. This was done by finding the mean if the training set and subtracting it from all data points. We can see the effects of this on the images below, but the ones to the right have been normalized having their colors a little bit more saturated/toned down.

i) State how you checked your analytic gradient computations and whether you think that your gradient computations were bug free. Give evidence for these conclusions.

Just like last time, we could use the formula given in lecture to compare the gradients computed from the backprop function and the numerical function. The function sums up the absolute values of the differences between both computations, divided by the sum of both their absolute values. The resulting number should be pretty low which they were (around the power of -7). As seen below, running the network for 200 epochs manages to overfit the training data as the cost is reduced for it, but not the validation data.

```
The relative error of First layer weights is 9.96426726888e-07
The relative error of First layer bias is 5.12623893905e-05
The relative error of Second layer weights is 5.30763719076e-08
The relative error of Second layer bias is 7.02932340418e-06
```

```
Network with lambda=0 epochs=200 batch=100 eta=0.005
```

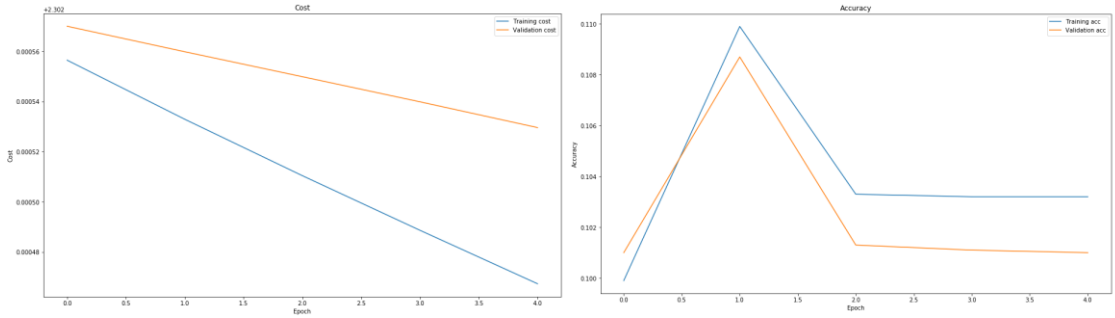



```
Info on Training   data: Accuracy: 16.00% and Cost: 2.29
Info on Validation data: Accuracy: 10.08% and Cost: 2.30
Info on Testing    data: Accuracy: 10.00% and Cost: 2.30
```

ii) Comment on how much faster the momentum term made your training.

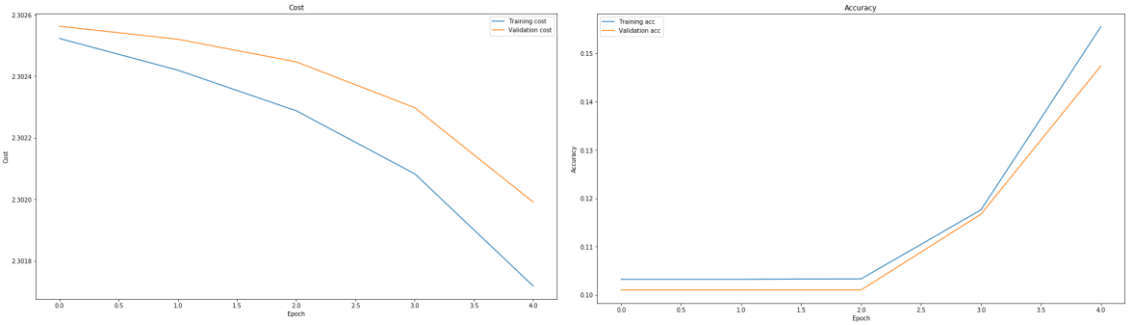
Training can be redundantly slow using the classical gradient descent method. A new method, momentum, was introduced to speed up the training process but what it does is that the weights are updated using both the gradients of the current run and the former one. By doing so the update is affected also by the past update times as a momentum value. To test out the momentum I tried running my training loop on 0.5, 0.9 and 0.99 momentum so I could see the effects for myself.

```
Network with lambda=1e-06 epochs=5 batch=500 eta=0.005 momentum=0.5
```



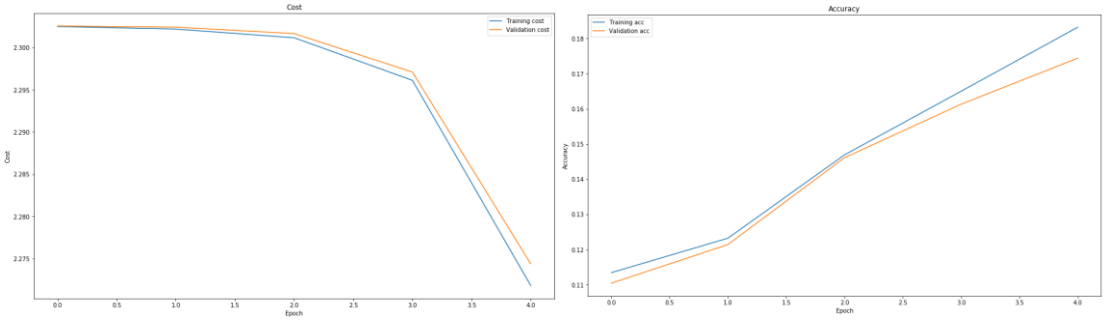

```
Info on Training   data: Accuracy: 10.32% and Cost: 2.3
Info on Validation data: Accuracy: 10.10% and Cost: 2.3
Info on Testing    data: Accuracy: 10.00% and Cost: 2.3
```

```
Network with lambda=1e-06 epochs=5 batch=500 eta=0.005 momentum=0.9
```




```
Info on Training   data: Accuracy: 15.55% and Cost: 2.3
Info on Validation data: Accuracy: 14.74% and Cost: 2.3
Info on Testing    data: Accuracy: 14.93% and Cost: 2.3
```

```
Network with lambda=1e-06 epochs=5 batch=500 eta=0.005 momentum=0.99
```




```
Info on Training   data: Accuracy: 18.32% and Cost: 2.27
Info on Validation data: Accuracy: 17.44% and Cost: 2.27
Info on Testing    data: Accuracy: 18.11% and Cost: 2.27
```

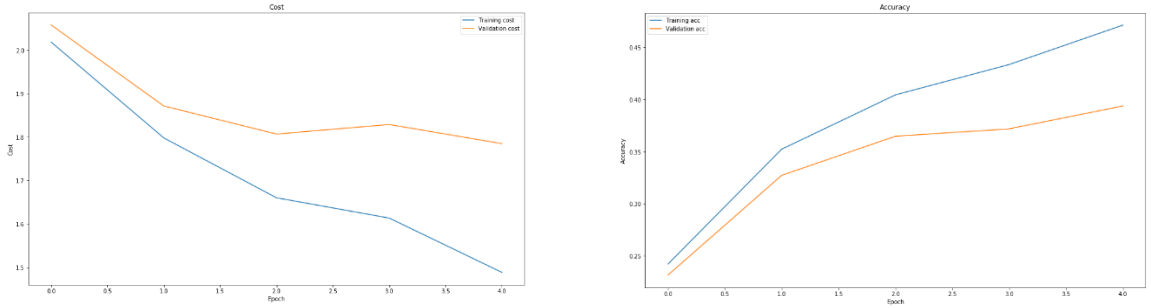
We can see from these plots that by using the momentum the cost drops significantly faster and the accuracy rises faster, indicating that momentum does really increase learning speed. That doesn't mean that we can pick the highest number possible for the momentum, since it might overshoot the local minima as it reaches it. Thus it is necessary to use some sort of weight decay with it.

iii) State the range of the values you searched for lambda and eta, the number of epochs used for training during the coarse search and the hyper-parameter settings for the 3 best performing networks you trained.

Find a reasonable range of values for the learning rate.

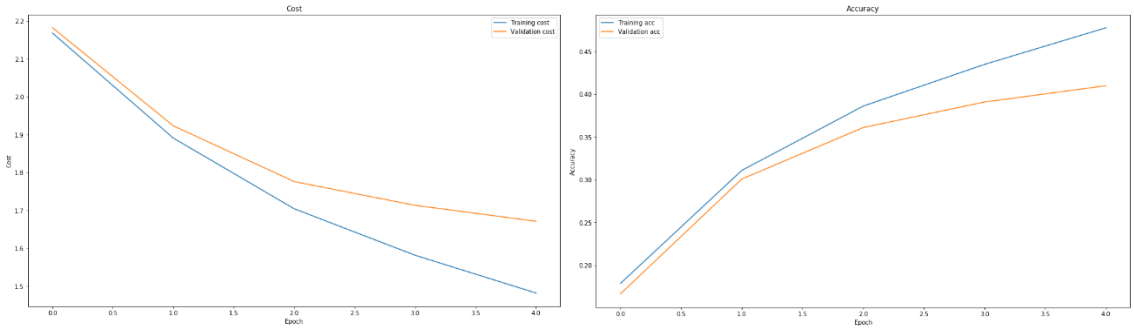
To search for the best learning rate for my stacked network I created a broad range of numbers ranging from 0.6 to 0.0001. To my surprise, the top three had the values of 0.6, 0.4 and 0.1. From my experience that learning rate is way too high, but maybe in this case it makes sense since we are speeding up the learning and decaying the learning rate at the same time. I used this information in my finer search for the best lambda and learning rate

```
Info on network with lambda=1e-06 epochs=5 batch=500 eta=0.6
```



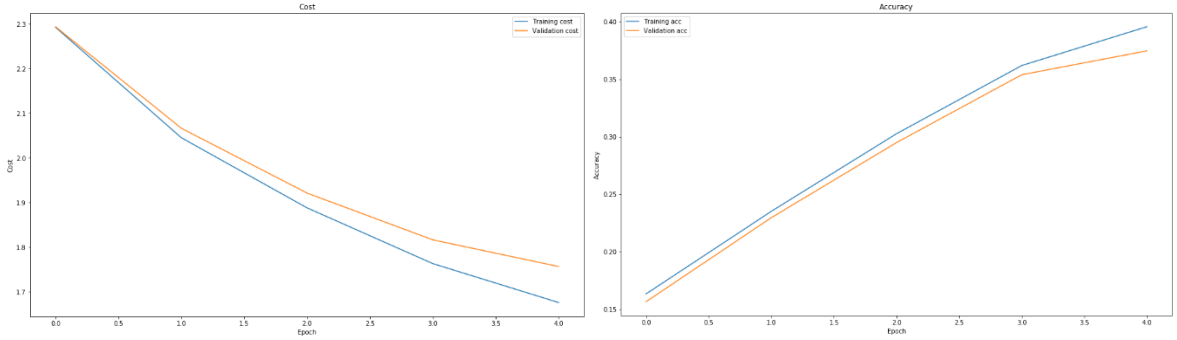

```
Info on Training   data: Accuracy: 45.16% and Cost: 1.60
Info on Validation data: Accuracy: 36.69% and Cost: 1.92
Info on Testing    data: Accuracy: 37.85% and Cost: 1.87
```

```
Info on network with lambda=1e-06 epochs=5 batch=500 eta=0.4
```




```
Info on Training   data: Accuracy: 47.14% and Cost: 1.49
Info on Validation data: Accuracy: 39.38% and Cost: 1.78
Info on Testing    data: Accuracy: 39.38% and Cost: 1.76
```

```
Info on network with lambda=1e-06 epochs=5 batch=500 eta=0.2
```

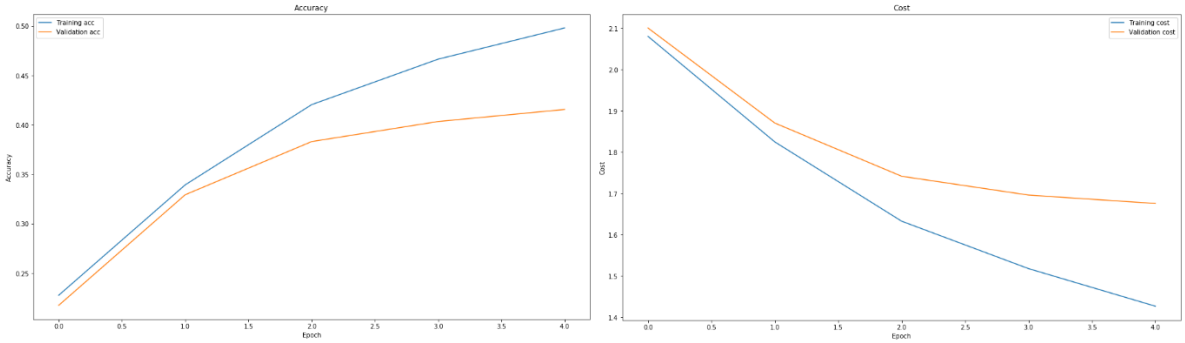



```
Info on Training   data: Accuracy: 47.77% and Cost: 1.48
Info on Validation data: Accuracy: 40.99% and Cost: 1.67
Info on Testing    data: Accuracy: 41.79% and Cost: 1.64
```

Coarse to fine random search to set lambda and eta.

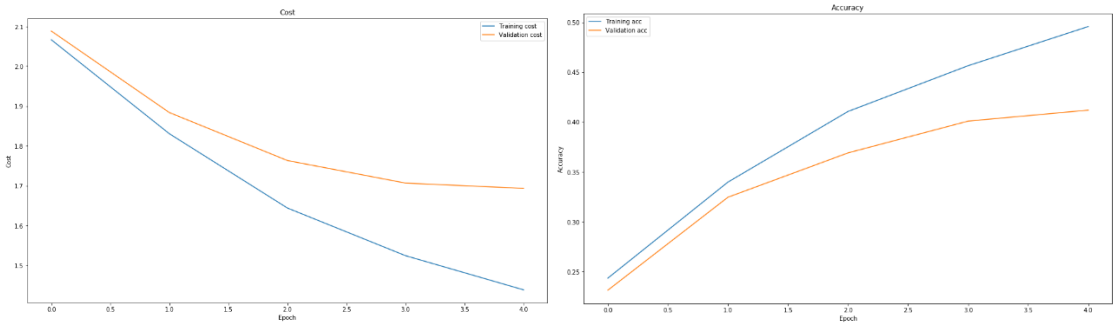
For the finer search I let my network run 80 times with random lambda and learning rate generated each time from my initial results. The eta ranged from 0.1 to 0.4 while the lambda ranged from 0.0001 to 0.000001. In the end, these following results gave me indication about my fine search parameters.

```
Info on network with epochs=5 batch=500
Lambda =5.998869935504525e-06
Eta    =0.255499147877516
```



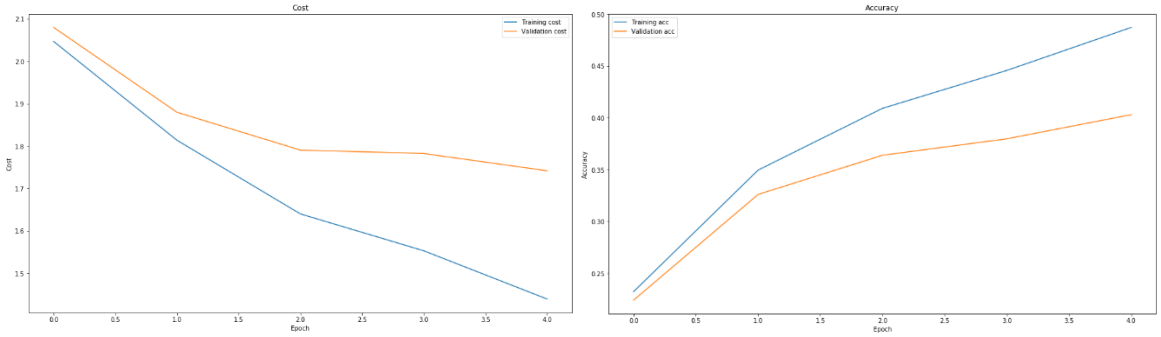

```
Info on Training   data: Accuracy: 49.81% and Cost: 1.43
Info on Validation data: Accuracy: 41.56% and Cost: 1.68
Info on Testing    data: Accuracy: 41.92% and Cost: 1.65
```

```
Info on network with epochs=5 batch=500
Lambda =8.248206928786062e-05
Eta    =0.29189273799480775
```




```
Info on Training   data: Accuracy: 50.00% and Cost: 1.43
Info on Validation data: Accuracy: 41.84% and Cost: 1.69
Info on Testing    data: Accuracy: 42.79% and Cost: 1.65
```

```
Info on network with epochs=5 batch=500
Lambda =9.423530463242987e-05
Eta    =0.37013743703824153
```



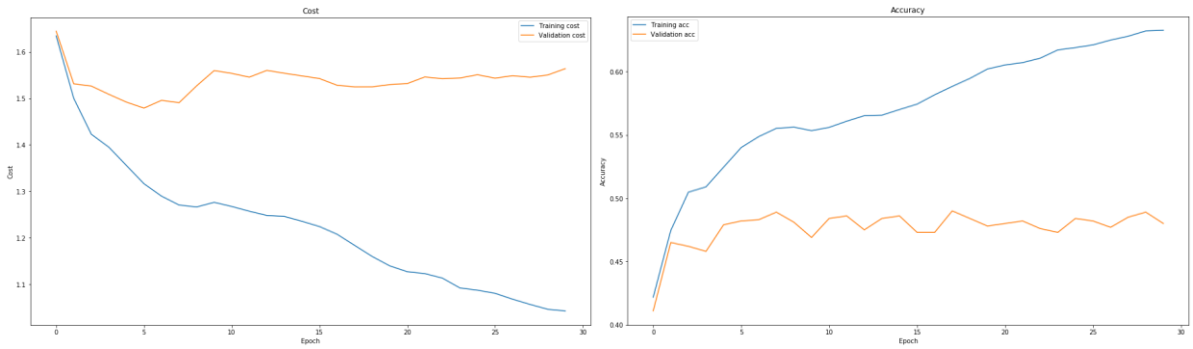

```
Info on Training   data: Accuracy: 50.25% and Cost: 1.43
Info on Validation data: Accuracy: 41.11% and Cost: 1.74
Info on Testing    data: Accuracy: 42.11% and Cost: 1.71
```

Now knowing the my best results lied somewhere around I restricted the eta between 0.25 and 33 and lambda around 8.24820e-05to 9.42353e-05 I decided to fine tune one more time, narrowing the range for a better result. Unfortunately, my results didn't get any better so I'll stick eta 0.29189273799480775 and lambda=8.248206928786062e-05.

v) For your best found hyper-parameter setting (according to performance on the validation set), train the network on all the training data (all the batch data), except for 1000 examples in a validation set, for 30 epochs. Plot the training and validation cost after each epoch of training and then report the learnt network's performance on the test data.

After picking my best parameters, I tried to give this a 30 epoch run resulting in these plots.

```
Info on network with epochs=30 batch=500
Lambda=8.248206928786062e-05
Eta=0.29189273799480775
```




```
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
```

As I managed to get my accuracy above 44% I'm still not quite happy with the results. Obviously the network over fits the training data as it gets accuracy close to 70% but is stuck at 50% for the validation and the testing data. What could have been done here was to use early stopping. Hopefully I can make a few improvements in the bonus part.