Report assignment 3

Shiva Besharat Pour
shivabp@kth.se

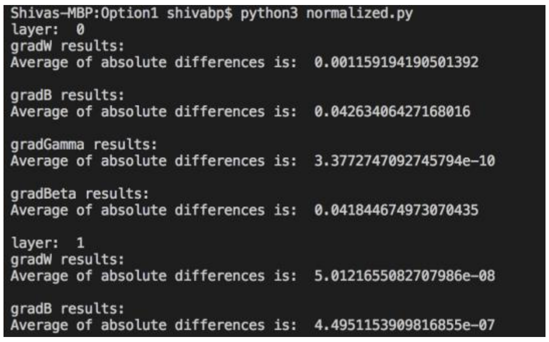
Introduction:

This assignment focused on implementing a k-layer network trained with mini-batch gradient descent. The training was to be batch normalized and the performance to be analysed.
NOTE: In all figures below, red line represents the training data and blue line represents the validation data.

Implementation and results:

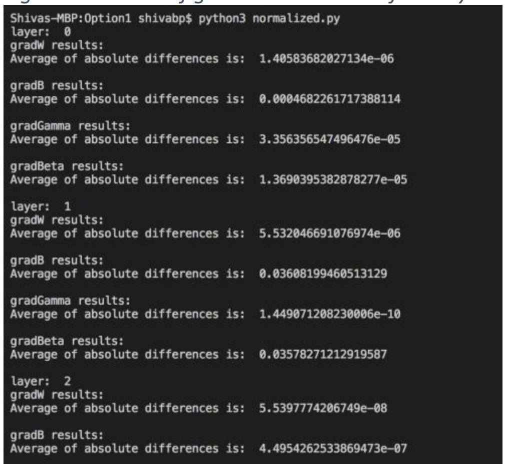
Gradient evaluation:

In order to check my analytic gradient computation for the batch normalized code, I first implemented the numerical gradients that we were given MATLAB code for in python. I then had to upgrade them so they can support gammas and betas as well.
I then computed the difference between the analytic and numerical gradients for both 2 layers and 3 layers networks to ensure myself of the reliability of my analytic gradients. The results are as follows:



```
Shivas-MBP:Option1 shivabp$ python3 normalized.py
layer:  0
gradW results:
Average of absolute differences is:  0.001159194190501392

gradB results:
Average of absolute differences is:  0.04263406427168016

gradGamma results:
Average of absolute differences is:  3.3772747092745794e-10

gradBeta results:
Average of absolute differences is:  0.041844674973070435

layer:  1
gradW results:
Average of absolute differences is:  5.0121655082707986e-08

gradB results:
Average of absolute differences is:  4.4951153909816855e-07
```

Figure 1: Results of gradient evaluation for 2 layers network



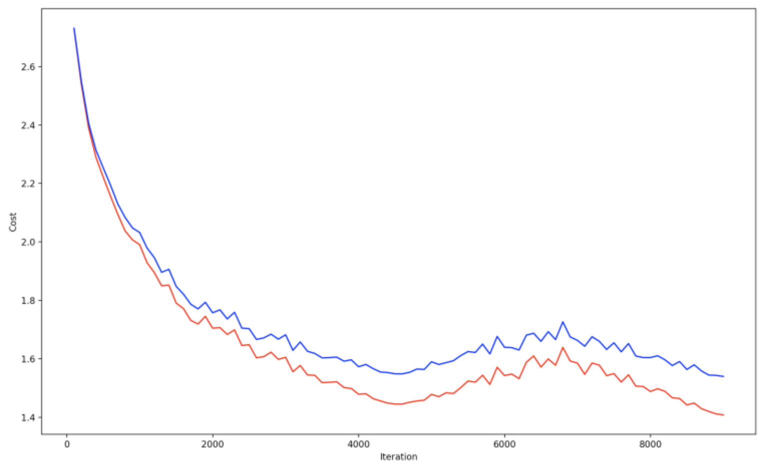
```
Shivas-MBP:Option1 shivabp$ python3 normalized.py
layer:  0
gradW results:
Average of absolute differences is:  1.40583682027134e-06

gradB results:
Average of absolute differences is:  0.0004682261717388114

gradGamma results:
Average of absolute differences is:  3.356356547496476e-05

gradBeta results:
Average of absolute differences is:  1.3690395382878277e-05

layer:  1
gradW results:
Average of absolute differences is:  5.532046691076974e-06

gradB results:
Average of absolute differences is:  0.03608199460513129

gradGamma results:
Average of absolute differences is:  1.449071208230006e-10

gradBeta results:
Average of absolute differences is:  0.03578271212919587

layer:  2
gradW results:
Average of absolute differences is:  5.5397774206749e-08

gradB results:
Average of absolute differences is:  4.4954262533869473e-07
```

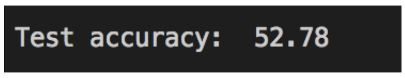
Figure 2: Results of gradient evaluation for 3 layers network

In general, the difference seems to be very small which confirms the reliability of the analytic gradients. Also, it can be noticed that in the 3-layer network, the difference is larger in earlier layers compared to the later ones. This is expected as mentioned in the instructions as the error is back-propagated.

Effects of batch normalization on 3-layer network:

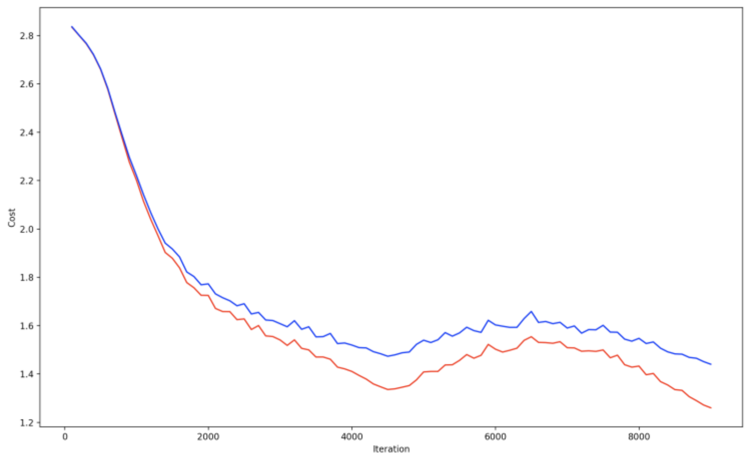
Without batch normalization:

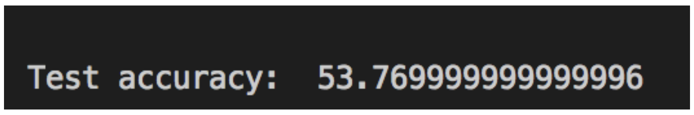




```
Test accuracy:  52.78
```

With batch normalization:

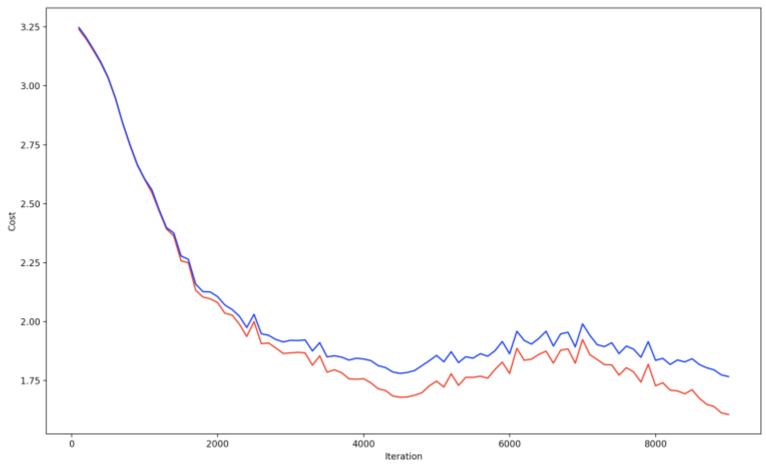


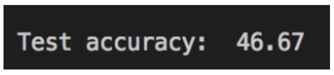


```
Test accuracy:  53.769999999999996
```

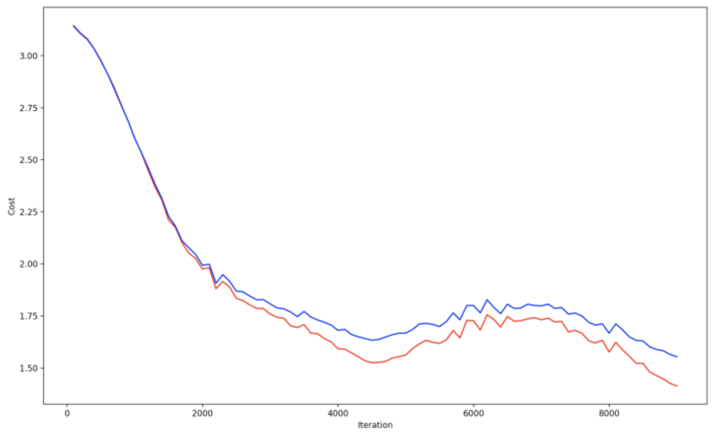
Effects of batch normalization on 9-layer network:

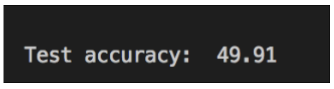
Without batch normalization:




```
Test accuracy:  46.67
```

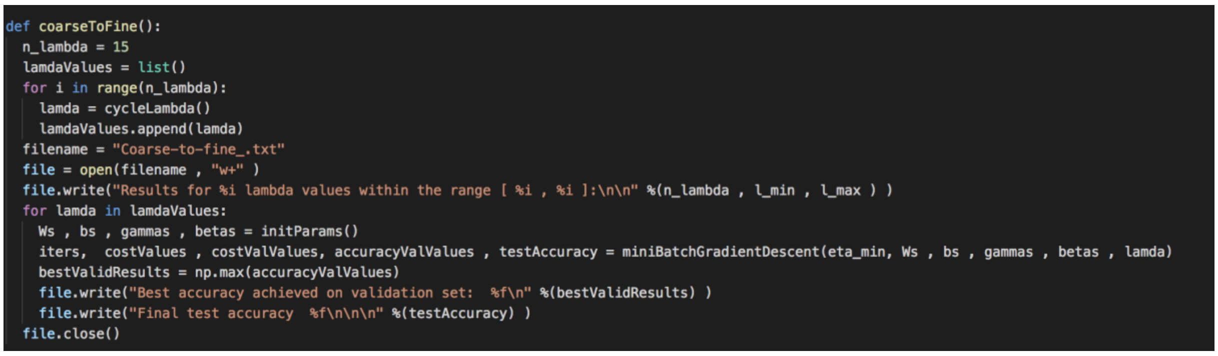
With batch normalization:




```
Test accuracy:  49.91
```

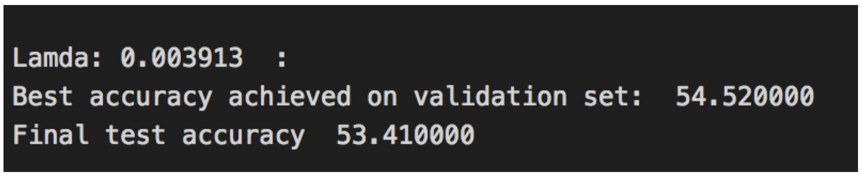
Lambda search:

In order to search for good lambda values, I implemented the coarse-to-fine function as given in assignment 2 as follows:



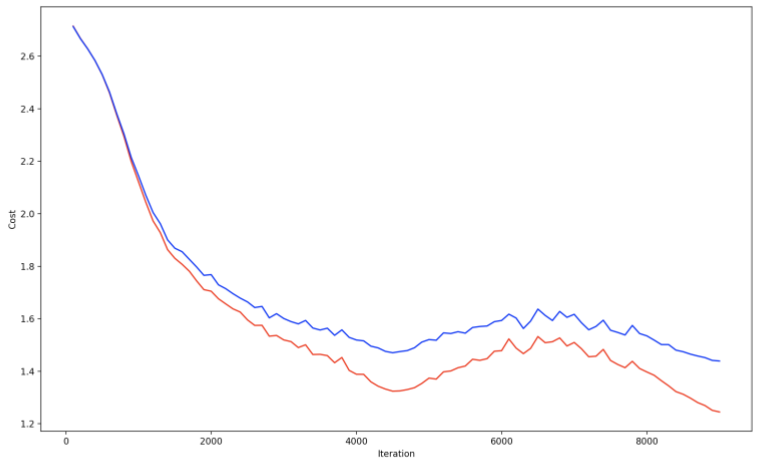
```
def coarseToFine():
  n_lambda = 15
  lamdaValues = list()
  for i in range(n_lambda):
    lamda = cycleLambda()
    lamdaValues.append(lamda)
  filename = "Coarse-to-fine_.txt"
  file = open(filename , "w+" )
  file.write("Results for %i lambda values within the range [ %i , %i ]:\n\n" %(n_lambda , l_min , l_max ) )
  for lamda in lamdaValues:
    Ws , bs , gammas , betas = initParams()
    iters,  costValues , costValValues, accuracyValValues , testAccuracy = miniBatchGradientDescent(eta_min, Ws , bs , gammas , betas , lamda)
    bestValidResults = np.max(accuracyValValues)
    file.write("Best accuracy achieved on validation set:  %f\n" %(bestValidResults) )
    file.write("Final test accuracy  %f\n\n\n" %(testAccuracy) )
  file.close()
```

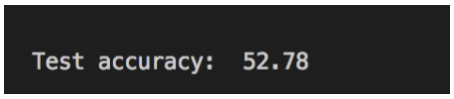
The range of values I search for was [-6, -2]. The results of the search are presented below:



```
Lamda: 0.003913  :
Best accuracy achieved on validation set:  54.520000
Final test accuracy  53.410000
```

Using lambda value that resulted in the highest accuracy from the search, in training resulted in the following:



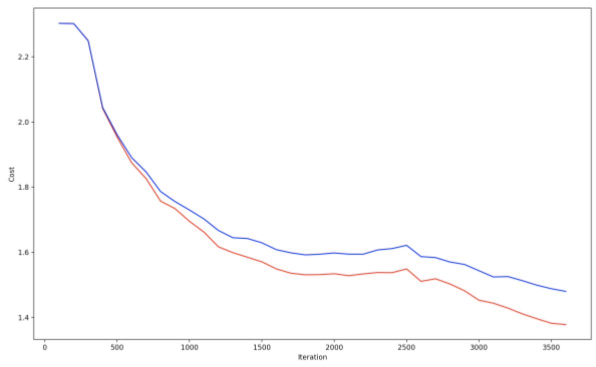


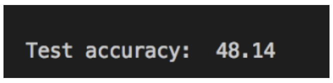
```
Test accuracy:  52.78
```

Not much improvement nor difference. Therefore, I think lambda 0.005 as suggested by the instructions resulted in better performance and I used that lambda value for the remainder of performance measures.

Sensitivity to initialization:

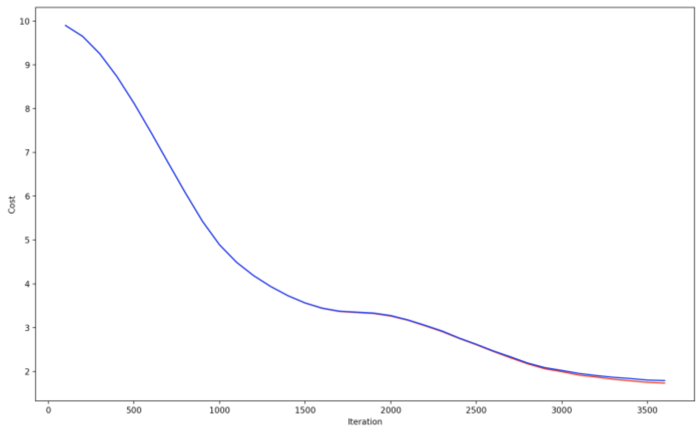
Without batch normalization:

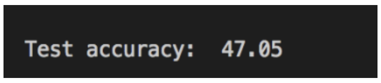




```
Test accuracy:  48.14
```

With batch normalization:





```
Test accuracy:  47.05
```

Batch normalization seems to be very sensitive to initialization as opposed to the network trained without batch normalization. As the figures show, the initial cost of network with batch normalization is very high compared to that of the network without batch normalization. Also, the test accuracy seems to be affected which as well implies that the generalization capabilities of a network trained with batch normalization are affected by the initialization values.