Report assignment 1

Shiva Besharat Pour
shivabp@kth.se

Introduction

The purpose of the assignment was to implement a one-layer network that performs mini-batch gradient descent on a set of data to minimize the error of its prediction compared to the actual targets. In this report, I will present the results for the 4 different test cases I got from the task and my conclusions regarding the result. I will then evaluate my learnings and the way I performed the task.

Methods:

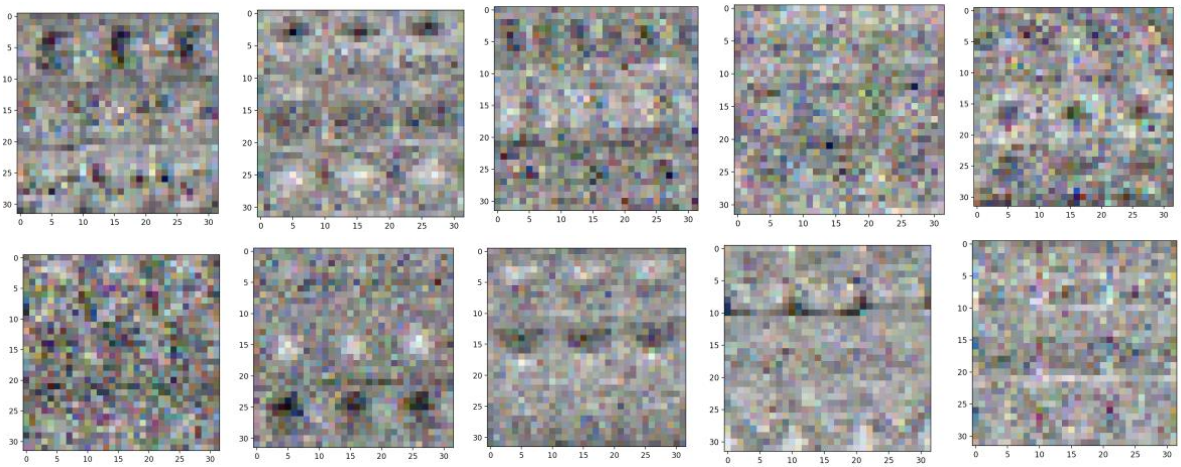
I used the equations stated in the assignment instructions and lecture 3 slides in order to implement the functionality of this one layer-network. I did my implementation in python.

Results:

Test case 1:

Lambda = 0 ETA = 0.1 n_epochs= 40 n_batch=100

Plots of learned weight matrix for each class (ordered from 1-10, left to right):



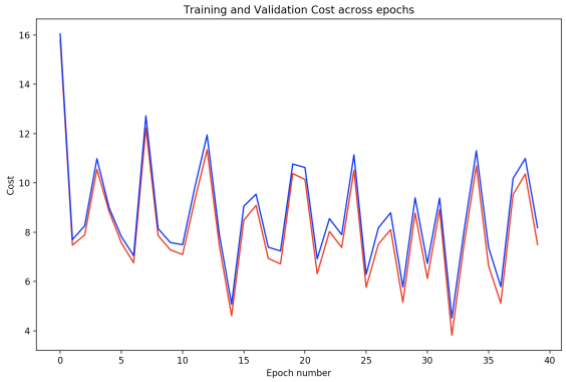


Figure 1:Total loss according to the cost function

Figure 2: Accuracy of prediction across epochs

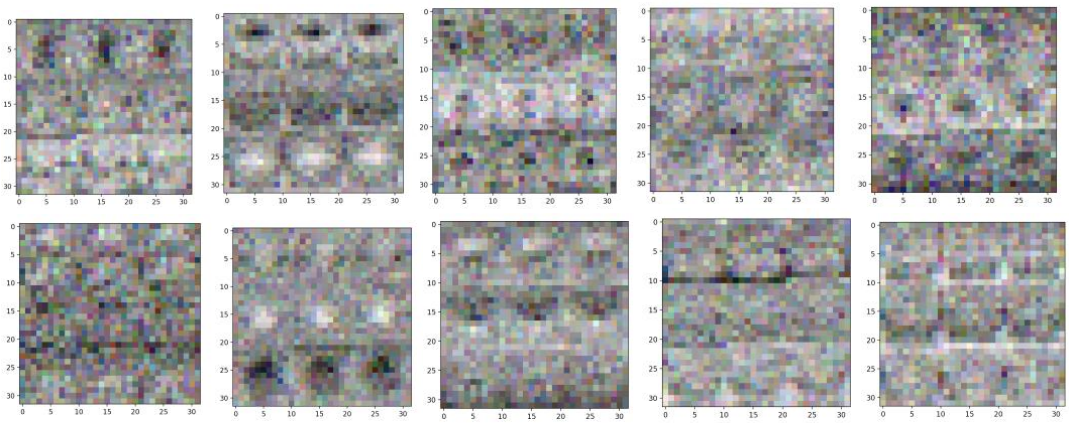
The red line is plotted according to results on training data and the blue line is plotted for the validation data. One would want the total loss to be constantly decreasing and the accuracy across epochs to be increasing. However, in the figures above, both accuracy and total loss are fluctuating which would mean that the parameters are not suited. In this case I would say that the learning rate of 0.1 is perhaps too big for the network to be able to find a stable global minimum. Thus, the network falls into local minima which may have caused the fluctuations in accuracy and loss. Since lambda was 0 in this test case, the regularization term was 0 and thereby the cost was dependent on the cross-entropy loss only. The network achieved accuracy of 25.71% on the test set.

```
Test accuracy:  25.71
```

Test case 2:

Lambda = 0 ETA = 0.01 n_epochs= 40 n_batch=100

Plots of learned weight matrix for each class (ordered from 1-10, left to right):



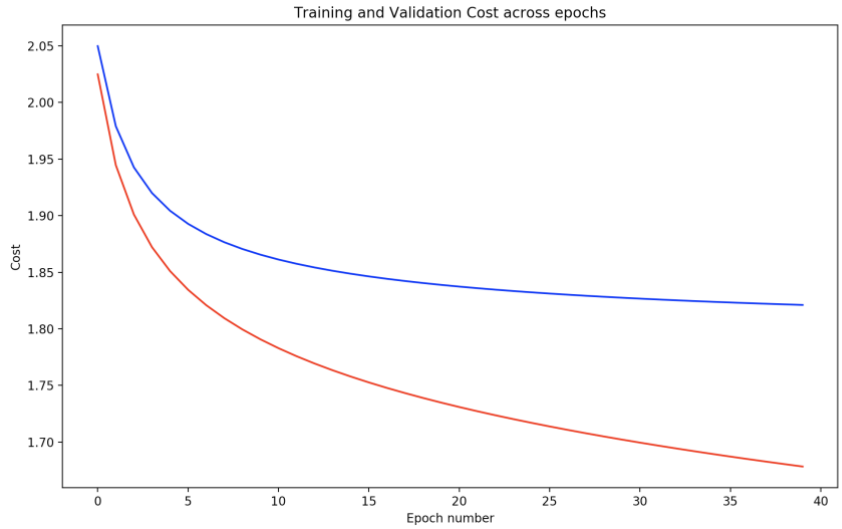


Figure 3:Total loss according to the cost function

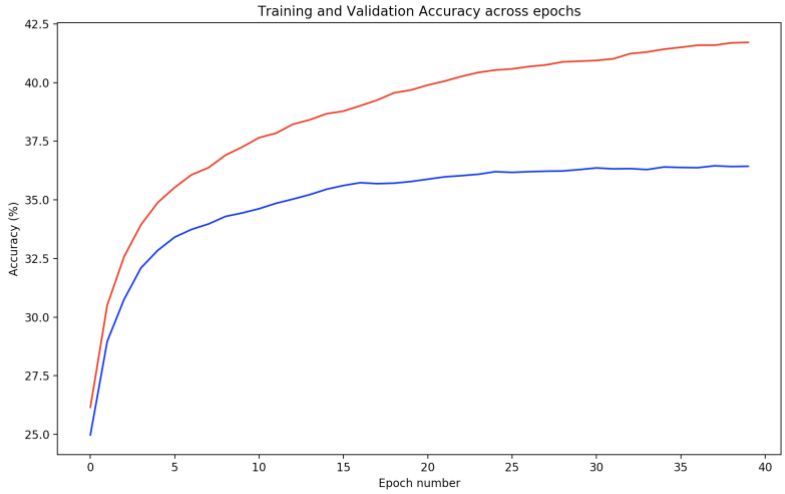


Figure 4: Accuracy of prediction across epochs

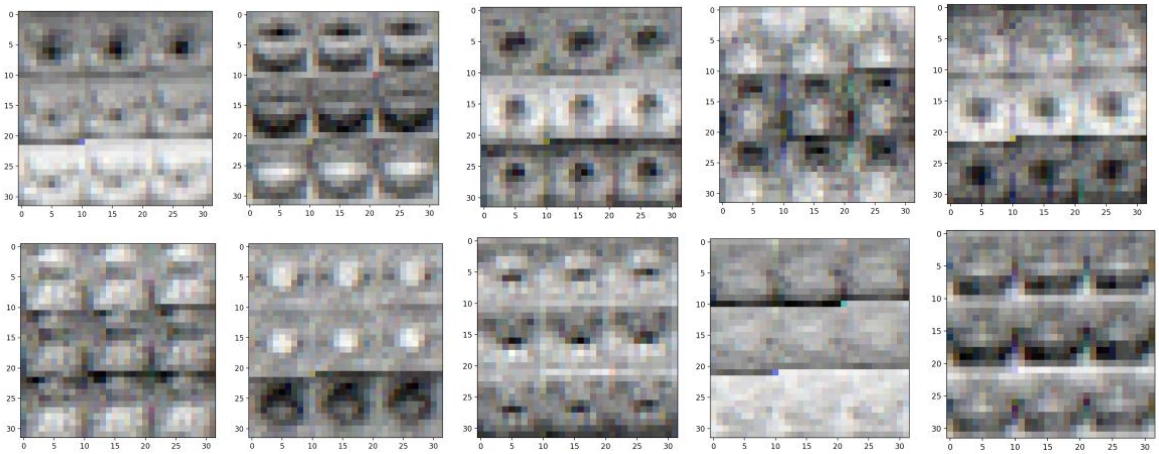
The red line is plotted according to results on training data and the blue line is plotted for the validation data. These figures confirm further that the learning rate in the previous test might have been too high because now having lowered the learning rate with all other parameters unchanged, accuracy is increasing whilst total loss is decreasing which is more desirable. Also, the accuracy and loss for training data are better relative to the validation data which is expected as network is of course more familiar with the training data than the validation data, however it still performs quite descent on the validation data. The network achieved accuracy of almost 37% on the test set which shows that the parameters were better tuned than they were in the previous case. Accuracy performance of the network on test set can be used to discuss how good network is on generalizing on unseen data and to make sure that the network has not overfit. Overfitting can be caused by having too complex network with too little data or too much training. I would say that one would perhaps want a test accuracy of at least 50% in real applications.

```
Test accuracy:  36.870000000000005
```

Test case 3:

Lambda = 0.1 ETA = 0.01 n_epochs= 40 n_batch=100

Plots of learned weight matrix for each class (ordered from 1-10, left to right):



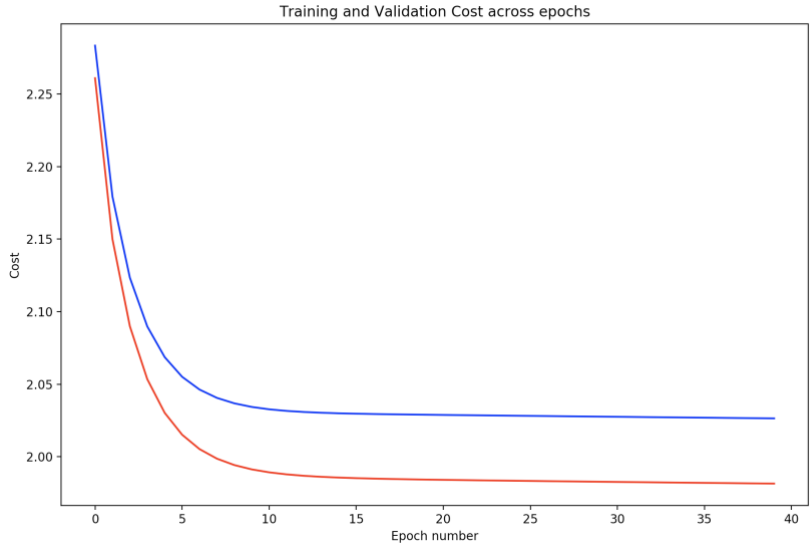


Figure 5:Total loss according to the cost function

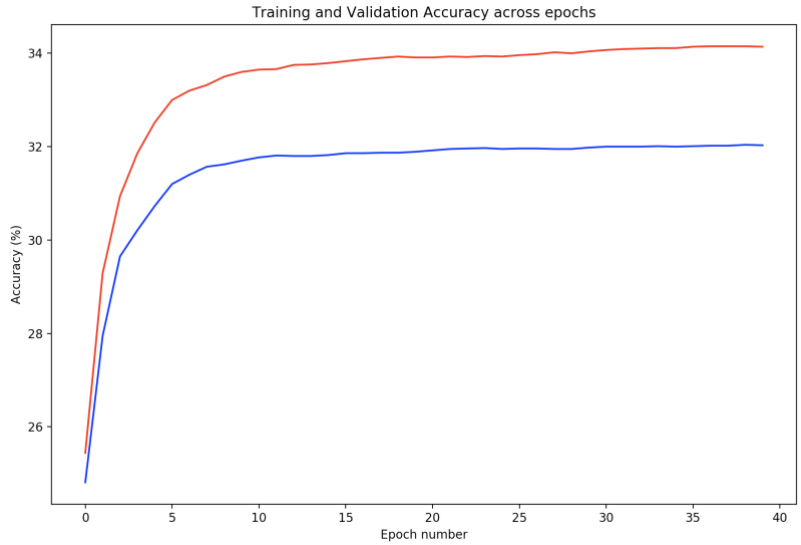


Figure 6:Accuracy of prediction across epochs

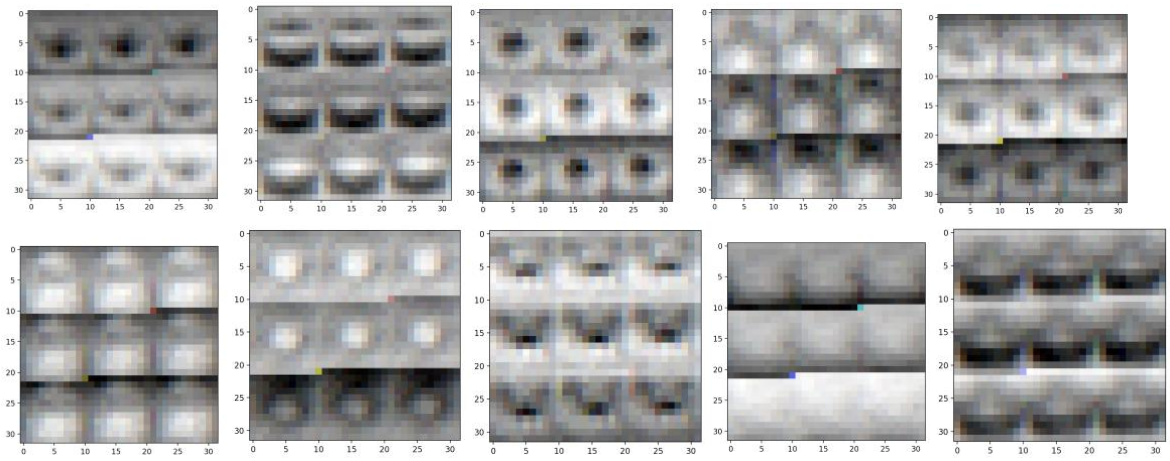
The red line is plotted according to results on training data and the blue line is plotted for the validation data. The difference between this test case and test case 2 is increased regularization parameter by introducing lambda 0.1. The network seems to have decreased its total loss as calculated by the cost function faster as well as the accuracy seems to be growing faster. The regularization parameter is usually used to avoid overfitting by avoiding large weights. This issue is known as bias-variance trade-off as high variance results in overfitting and regularization can be used to lower variance. However, this may result in higher bias error. The networked achieved an accuracy of 33.39% on the test set.

```
Test accuracy:  33.39
```

Test case 4:

Lambda = 1 ETA = 0.01 n_epochs= 40 n_batch=100

Plots of learned weight matrix for each class (ordered from 1-10, left to right):



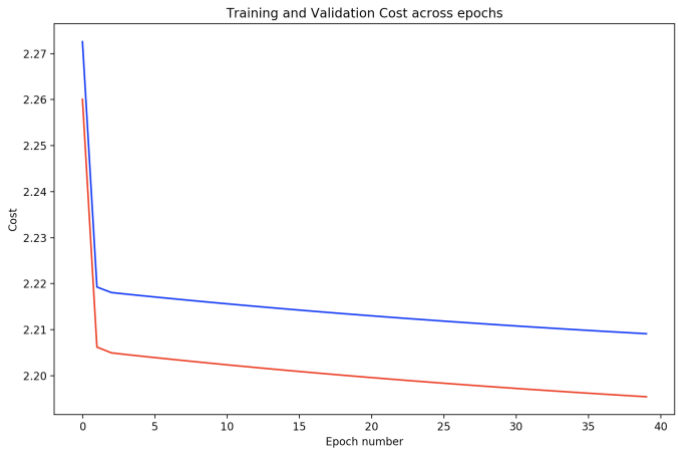


Figure 7:Total loss according to the cost function

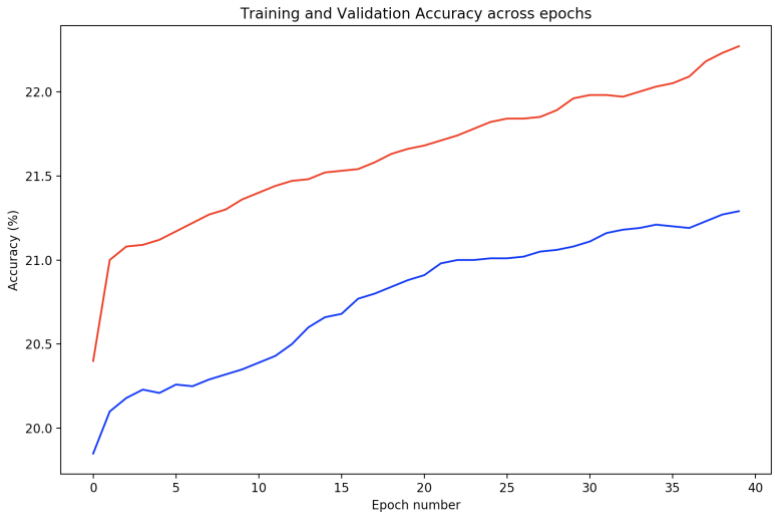


Figure 8:Accuracy of prediction across epochs

The red line is plotted according to results on training data and the blue line is plotted for the validation data. We can see that the cost function is more rapidly decreasing and accuracy is increasing in a more linear manner. However, the accuracy on the test set is 21.87% which may be due to too large regularization which may be too strong and introduce higher bias error.

```
Test accuracy:  21.87
```

Evaluation:

In order to check for correctness of the gradients that I had implemented based on the analytical method, I used the MATLAB code given for numerically calculating the gradients (I used the more efficient version). I then took 1 data point from the training data and used 20 dimensions of that data point as input to both the numerical and analytical gradient functions.

```
grad_w_Analytic , grad_b_Analytic  = computeGradAnalytic(X[:20 , 0:1], W[: , :20] , b , Y[: , 0:1]  )
grad_w_Numeric , grad_b_Numeric = computeGradNumeric(X[:20 , 0:1] , W[: , :20] , b , Y[: , 0:1] )
```

Figure 9: Choosing data point as input for gradient functions

I then implemented the following in order to automate the gradient evaluation:

```
def checkGradients( ):
    X, Y, y = readData("data_batch_1")
    W, b = initParams()
    grad_w_Analytic , grad_b_Analytic  = computeGradAnalytic(X[:20 , 0:1], W[: , :20] , b , Y[: , 0:1]  )
    grad_w_Numeric , grad_b_Numeric = computeGradNumeric(X[:20 , 0:1] , W[: , :20] , b , Y[: , 0:1] )
    print("gradW results:" )
    print('Sum of absolute differences is: ' , np.abs(grad_w_Analytic - grad_w_Numeric).sum())
    print("Analytic gradW:  Mean:   " ,np.abs(grad_w_Analytic).mean() , "   Min:    " ,np.abs(grad_w_Analytic).min() , "    Max:    " ,  np.abs(grad_w_Analytic).max(
    print("Numeric gradW:   Mean:   " ,np.abs(grad_w_Numeric).mean() , "   Min:    " ,np.abs(grad_w_Numeric).min() , "    Max:    " ,  np.abs(grad_w_Numeric).max()
    print("gradB results:" )
    print('Sum of absolute differences is: ' , np.abs(grad_b_Analytic - grad_b_Numeric).sum())
    print("Analytic gradb:  Mean:   " ,np.abs(grad_b_Analytic).mean() , "   Min:    " ,np.abs(grad_b_Analytic).min() , "    Max:    " ,  np.abs(grad_b_Analytic).max(
    print("Numeric gradb:   Mean:   " ,np.abs(grad_b_Numeric).mean() , "   Min:    " ,np.abs(grad_b_Numeric).min() , "    Max:    " ,  np.abs(grad_b_Numeric).max()
```

Figure 10: Gradient evaluation

Through the above implementation, I use both the numerical and analytical gradient functions to get their gradient values on the same data point with the same dimensions. I ran this test 4 times and the results were as follows:

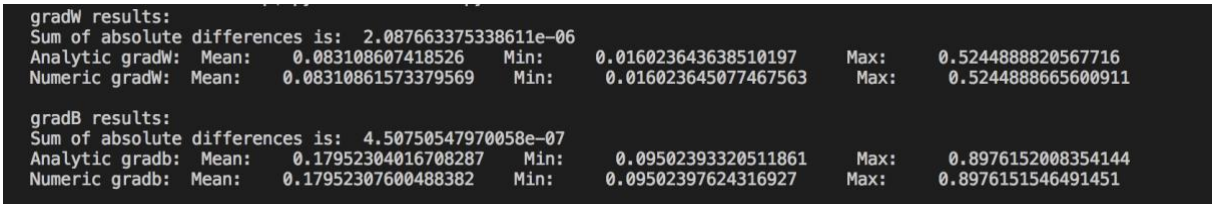


Figure 11: Gradient check results 1

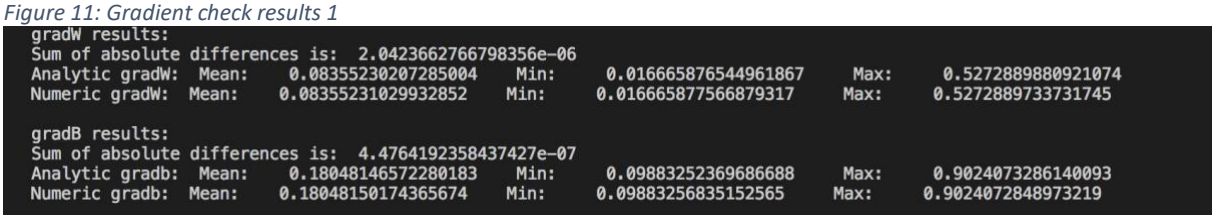


Figure 12: Gradient check results 2

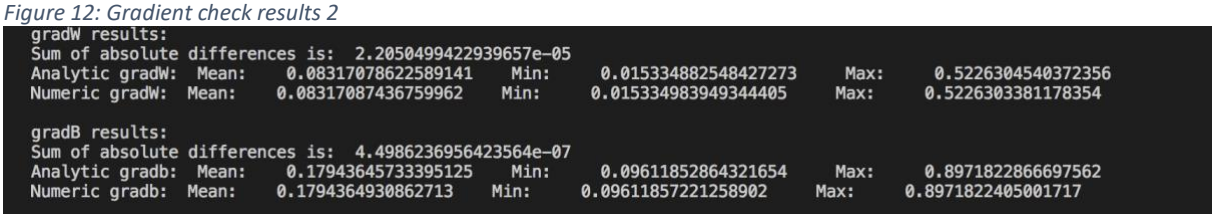


Figure 13: Gradient check results 3

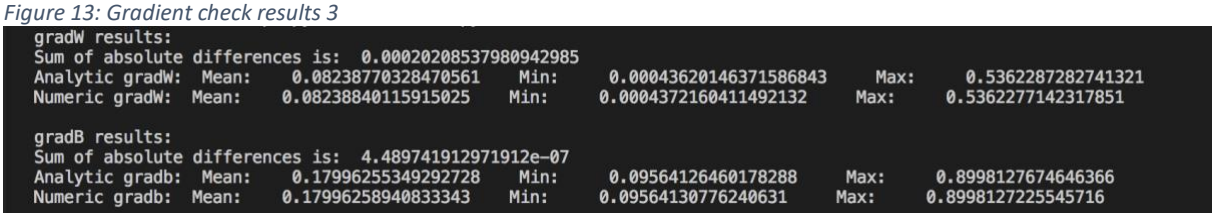


Figure 14: Gradient check results 2

The reason I did this test 4 times was for the purpose of double checking that the results are reliable. All 4 tests confirmed that the gradients as computed by the numerical gradient function have very little difference to the gradients as calculated by the analytical gradient function. Thus, the analytical gradient function which was used for the gradient descent training is reliable.

Conclusion:

Regularization is good to use to avoid network from overfitting by penalizing large weights. However, this comes at the price of bias-variance trade off as the bias error may increase in return.