Report Assignment 2:

Shiva Besharat Pour
shivabp@kth.se

Introduction:

The purpose of the assignment was to implement a 2-layer feed forward network trained by mini batch gradient descent. The implementation focuses on methods to tune parameters such as the regularization coefficient, lambda, and the learning rate, eta, in order to improve the accuracy of the network predictions on the test set.

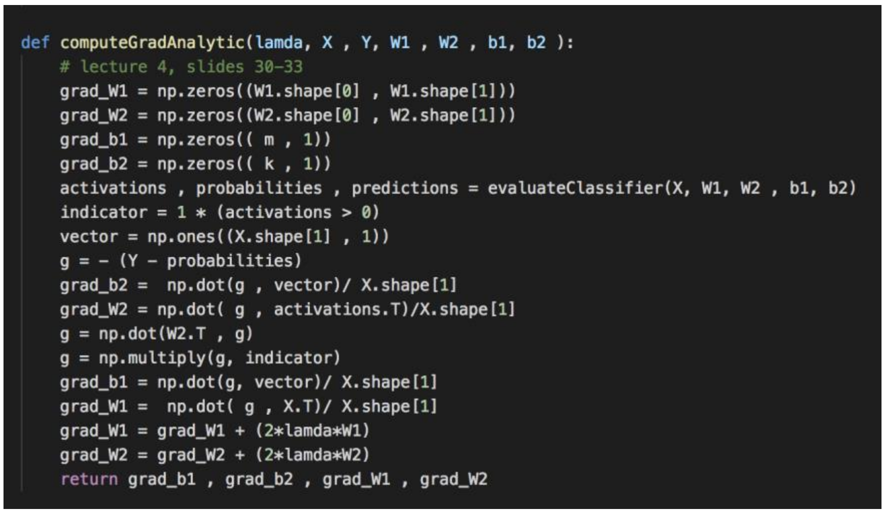
Note: In all figures presented in this report, the red colour corresponds to the training set and the blue colour corresponds to the validations set.

Implementation and results:

The core implementation of the network was very similar to that of assignment 1. However, a few changes were needed:

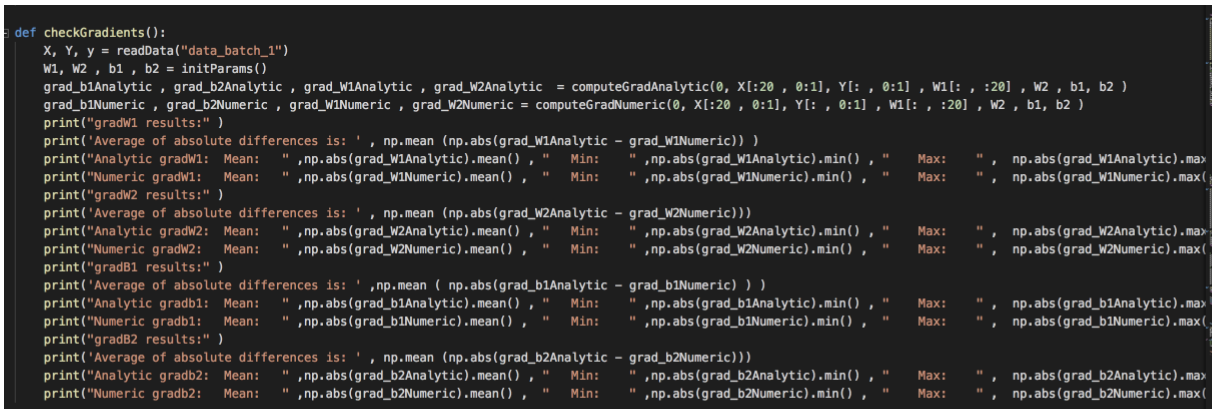
Gradient Evaluation:

The gradient computations were based on the analytical gradients presented in lecture 4, slides 30-33. The implementation used in this assignment is shown below.



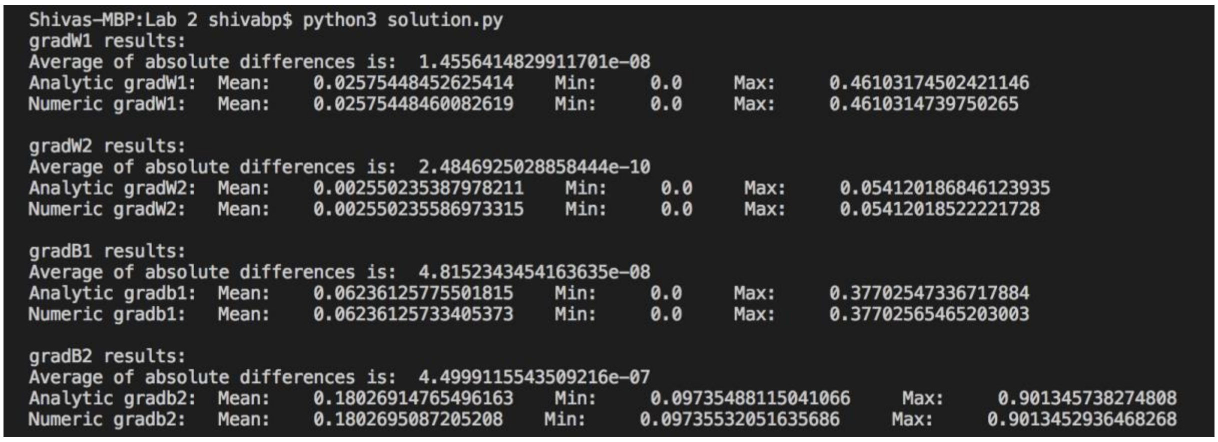
```
def computeGradAnalytic(lamda, X , Y, W1 , W2 , b1, b2 ):
    # lecture 4, slides 30-33
    grad_W1 = np.zeros((W1.shape[0] , W1.shape[1]))
    grad_W2 = np.zeros((W2.shape[0] , W2.shape[1]))
    grad_b1 = np.zeros(( m , 1))
    grad_b2 = np.zeros(( k , 1))
    activations , probabilities , predictions = evaluateClassifier(X, W1, W2 , b1, b2)
    indicator = 1 * (activations > 0)
    vector = np.ones((X.shape[1] , 1))
    g = - (Y - probabilities)
    grad_b2 =  np.dot(g , vector)/ X.shape[1]
    grad_W2 = np.dot( g , activations.T)/X.shape[1]
    g = np.dot(W2.T , g)
    g = np.multiply(g, indicator)
    grad_b1 = np.dot(g, vector)/ X.shape[1]
    grad_W1 =  np.dot( g , X.T)/ X.shape[1]
    grad_W1 = grad_W1 + (2*lamda*W1)
    grad_W2 = grad_W2 + (2*lamda*W2)
    return grad_b1 , grad_b2 , grad_W1 , grad_W2
```

In order to make sure that I have implemented gradient computations correctly, I used the numerical gradient computations for which we were given MATLAB implementation for. I then implemented the following function, that takes 1 data point and 20 dimensions of the data, in order for both the numerical and analytical gradient functions to process.



```
def checkGradients():
    X, Y, y = readData("data_batch_1")
    W1, W2 , b1 , b2 = initParams()
    grad_b1Analytic , grad_b2Analytic , grad_W1Analytic , grad_W2Analytic  = computeGradAnalytic(0, X[:20 , 0:1], Y[: , 0:1] , W1[: , :20] , W2 , b1, b2 )
    grad_b1Numeric , grad_b2Numeric , grad_W1Numeric , grad_W2Numeric = computeGradNumeric(0, X[:20 , 0:1], Y[: , 0:1] , W1[: , :20] , W2 , b1, b2 )
    print("gradW1 results:" )
    print('Average of absolute differences is: ' , np.mean (np.abs(grad_W1Analytic - grad_W1Numeric)) )
    print("Analytic gradW1:  Mean:    " ,np.abs(grad_W1Analytic).mean() , "   Min:     " ,np.abs(grad_W1Analytic).min() , "    Max:     " ,  np.abs(grad_W1Analytic).max
    print("Numeric gradW1:   Mean:    " ,np.abs(grad_W1Numeric).mean() ,  "   Min:     " ,np.abs(grad_W1Numeric).min() ,  "    Max:     " ,  np.abs(grad_W1Numeric).max(
    print("gradW2 results:" )
    print('Average of absolute differences is: ' , np.mean (np.abs(grad_W2Analytic - grad_W2Numeric)))
    print("Analytic gradW2:  Mean:    " ,np.abs(grad_W2Analytic).mean() , "   Min:     " ,np.abs(grad_W2Analytic).min() , "    Max:     " ,  np.abs(grad_W2Analytic).max
    print("Numeric gradW2:   Mean:    " ,np.abs(grad_W2Numeric).mean() ,  "   Min:     " ,np.abs(grad_W2Numeric).min() ,  "    Max:     " ,  np.abs(grad_W2Numeric).max(
    print("gradB1 results:" )
    print('Average of absolute differences is: ' ,np.mean ( np.abs(grad_b1Analytic - grad_b1Numeric) ) )
    print("Analytic gradb1:  Mean:    " ,np.abs(grad_b1Analytic).mean() , "   Min:     " ,np.abs(grad_b1Analytic).min() , "    Max:     " ,  np.abs(grad_b1Analytic).max
    print("Numeric gradb1:   Mean:    " ,np.abs(grad_b1Numeric).mean() ,  "   Min:     " ,np.abs(grad_b1Numeric).min() ,  "    Max:     " ,  np.abs(grad_b1Numeric).max(
    print("gradB2 results:" )
    print('Average of absolute differences is: ' , np.mean (np.abs(grad_b2Analytic - grad_b2Numeric)))
    print("Analytic gradb2:  Mean:    " ,np.abs(grad_b2Analytic).mean() , "   Min:     " ,np.abs(grad_b2Analytic).min() , "    Max:     " ,  np.abs(grad_b2Analytic).max
    print("Numeric gradb2:   Mean:    " ,np.abs(grad_b2Numeric).mean() ,  "   Min:     " ,np.abs(grad_b2Numeric).min() ,  "    Max:     " ,  np.abs(grad_b2Numeric).max(
```

Once the processing was done, the evaluation resulted in the following:

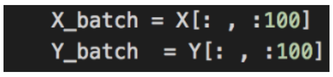


```
Shivas-MBP:Lab 2 shivabp$ python3 solution.py
gradW1 results:
Average of absolute differences is:  1.4556414829911701e-08
Analytic gradW1:  Mean:    0.02575448452625414    Min:     0.0     Max:     0.46103174502421146
Numeric gradW1:   Mean:    0.02575448460082619    Min:     0.0     Max:     0.4610314739750265

gradW2 results:
Average of absolute differences is:  2.4846925028858444e-10
Analytic gradW2:  Mean:    0.002550235387978211    Min:     0.0     Max:     0.054120186846123935
Numeric gradW2:   Mean:    0.002550235586973315    Min:     0.0     Max:     0.05412018522221728

gradB1 results:
Average of absolute differences is:  4.8152343454163635e-08
Analytic gradb1:  Mean:    0.06236125775501815    Min:     0.0     Max:     0.37702547336717884
Numeric gradb1:   Mean:    0.06236125733405373    Min:     0.0     Max:     0.37702565465203003

gradB2 results:
Average of absolute differences is:  4.4999115543509216e-07
Analytic gradb2:  Mean:    0.18026914765496163    Min:     0.09735488115041066     Max:     0.901345738274808
Numeric gradb2:   Mean:    0.1802695087205208    Min:     0.09735532051635686     Max:     0.9013452936468268
```

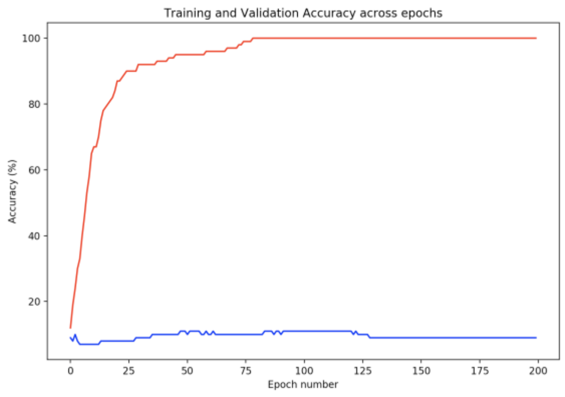
Therefore, it was confirmed for me that since the average difference between the numerical gradients and analytical gradients for both the weights and bias in both of the hidden layers is very small, the analytical implementation is reliable.

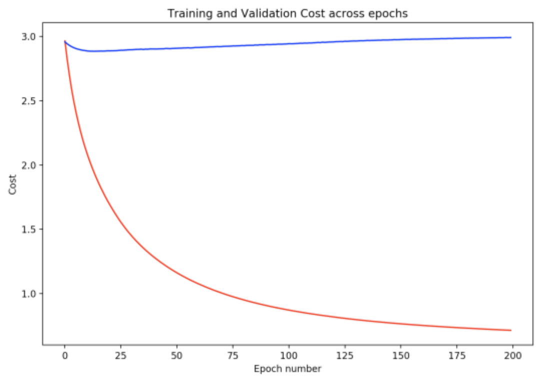
Initial training:

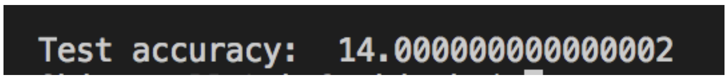
I implemented the rough training process once reliability of my gradient computations was confirmed. In order to double check that everything is fine and the network training is as it should, I tested the network with the following parameters:
Lambda: 0
Learning rate: 0.01
n_epochs: 200
and 100 data points. I then expected the network to overfit, because of too much training and too little data. Thus, as a result of overfitting, the network would achieve a very low loss and a very low accuracy on the validation and test sets. The actual results were as follows:



```
X_batch = X[: , :100]
Y_batch  = Y[: , :100]
```





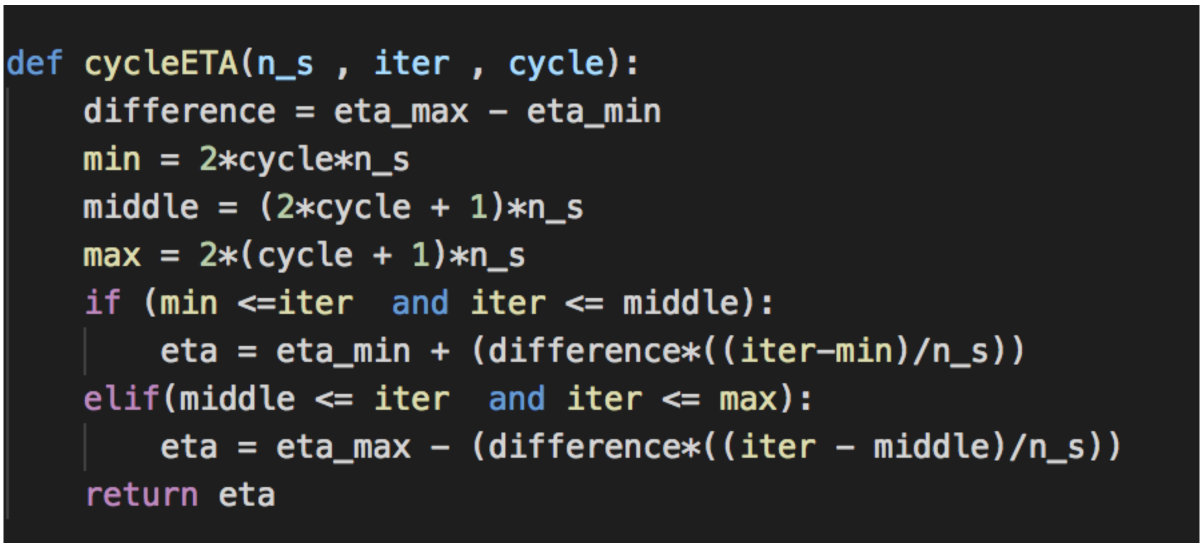


```
Test accuracy:  14.000000000000002
```

As the figures show, training accuracy is very high and its loss is very low. On the other hand, loss on the validation data is very high and its accuracy is relatively low. The test accuracy is also very low and thereby overfitting is confirmed. Thus, the implementation so far seems to be as it should and we can move forward with tuning the parameters.

Cyclic eta:

I implemented the function that varies eta values according to the assignment instructions as follows:

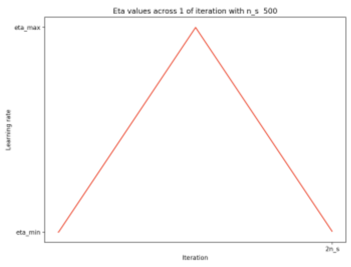


```
def cycleETA(n_s , iter , cycle):
    difference = eta_max - eta_min
    min = 2*cycle*n_s
    middle = (2*cycle + 1)*n_s
    max = 2*(cycle + 1)*n_s
    if (min <=iter  and iter <= middle):
        eta = eta_min + (difference*((iter-min)/n_s))
    elif(middle <= iter  and iter <= max):
        eta = eta_max - (difference*((iter - middle)/n_s))
    return eta
```

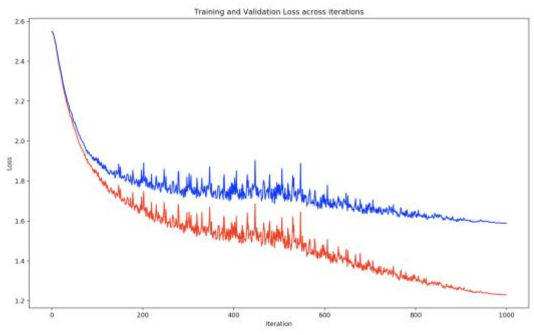
This function was then called for each iteration of training. I first tested this function with the following parameters:

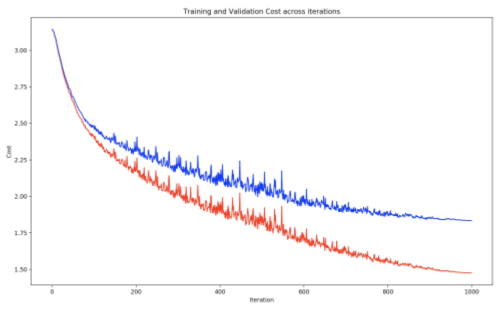
n_s (Stepsize): 500
n_epochs: 10
eta_max: 1e-1
eta_min: 1e-5
lambda: 0.01
n_cycles: 1

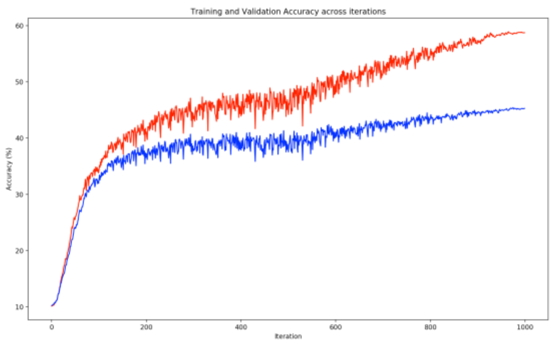
on the entire data, to get an overall image of the training performance and then I changed the plots to plot every 100th point of the iteration to get a less sparse view of the training performance. The results are as follows:

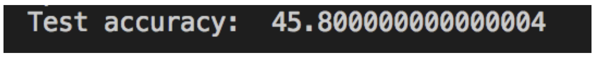


1 cycle is supposed to represent 2n_s. Eta is supposed to be eta_min at the beginning of a cycle, eta_max once half a cycle is reached and back to eta_min at the end of a cycle just as the figure shows.



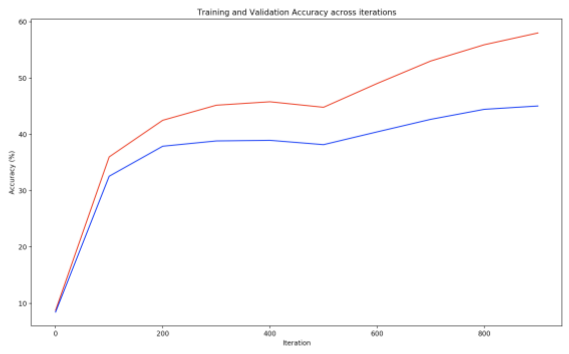


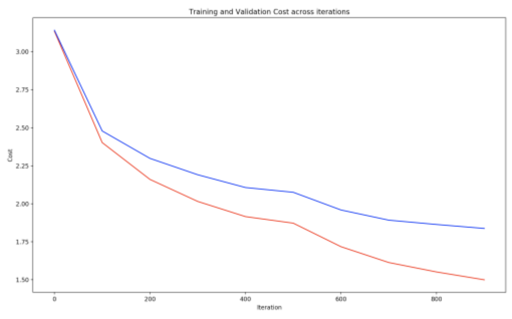


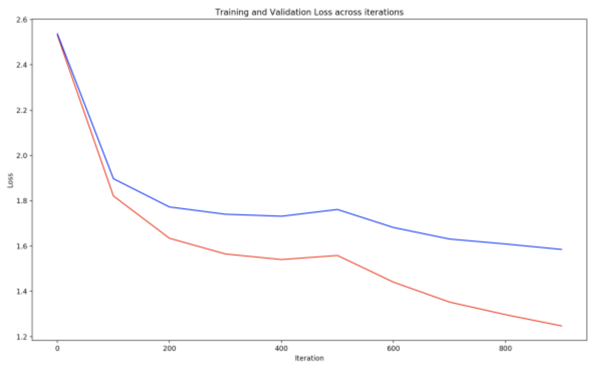


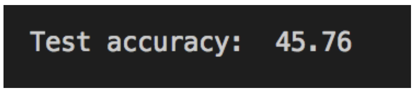
```
Test accuracy:  45.800000000000004
```

These results were achieved when all the data were used for the plots. The test accuracy and starting, ending as well as the direction of the accuracy, loss and the cost functions are very similar to those presented in Figure 3 of the assignment. In order to reduce the sparsity of the plots, I replotted on every 100th data point and iteration and the following results were obtained.



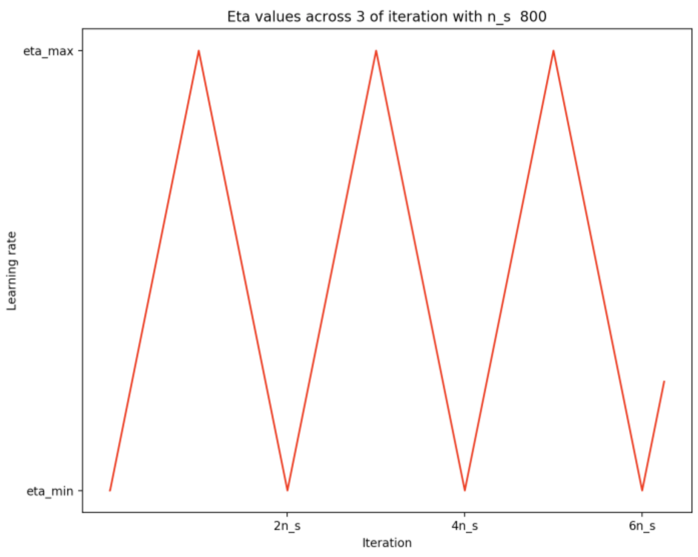




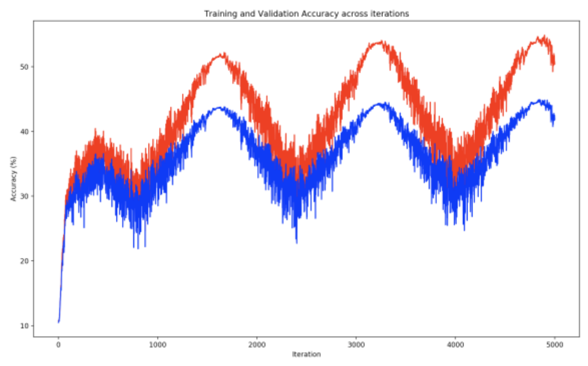


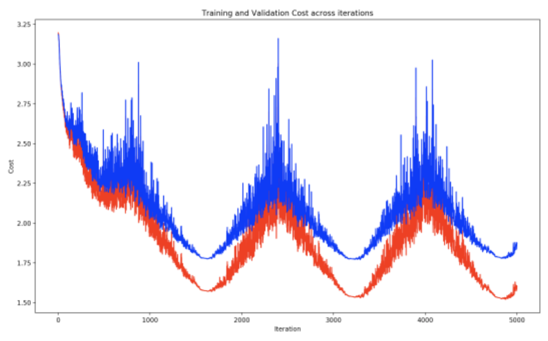
```
Test accuracy:  45.76
```

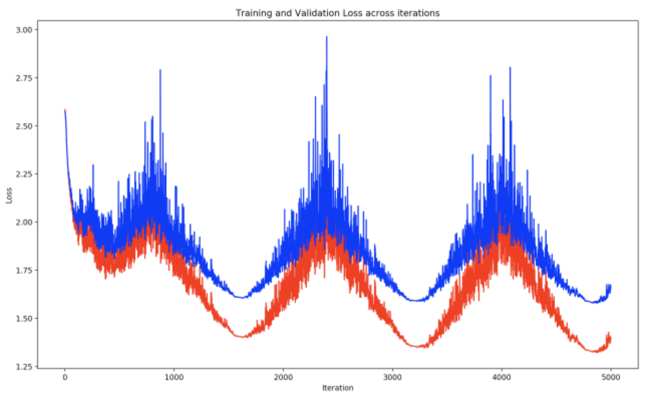
These results again are similar to that of assignment's figure 3 which was the confirmation of the reliability of my implementation.
The same experiment was repeated with the following parameters:
n_s (Stepsixe): 800
n_epochs: 10
eta_max: 1e-1
eta_min: 1e-5
lambda: 0.01
n_cycles: 3

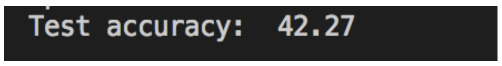


Again, plotting the results for all data at each iteration, resulted in the following:



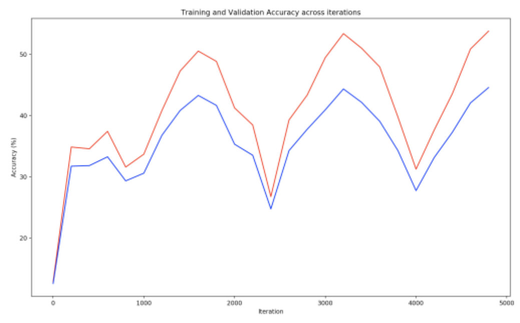


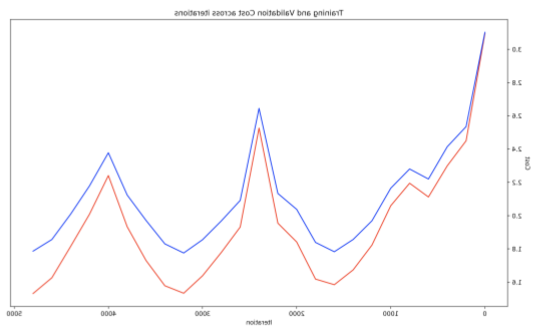


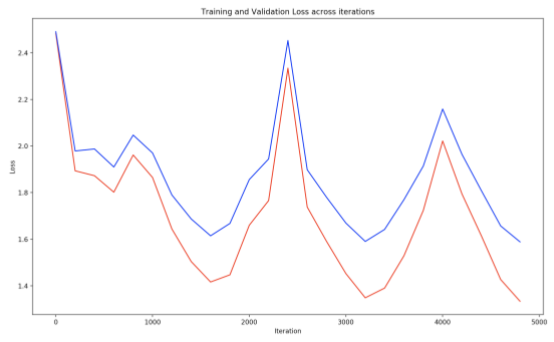


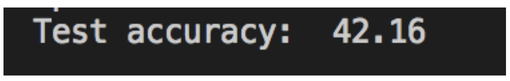
```
Test accuracy:  42.27
```

The sparsity of the curves is very high as the figures show since we are dealing with 10000 data points, ~5000 iterations as well as a cyclic learning rate. Plotting the same test for every 100th data point resulted in the following:







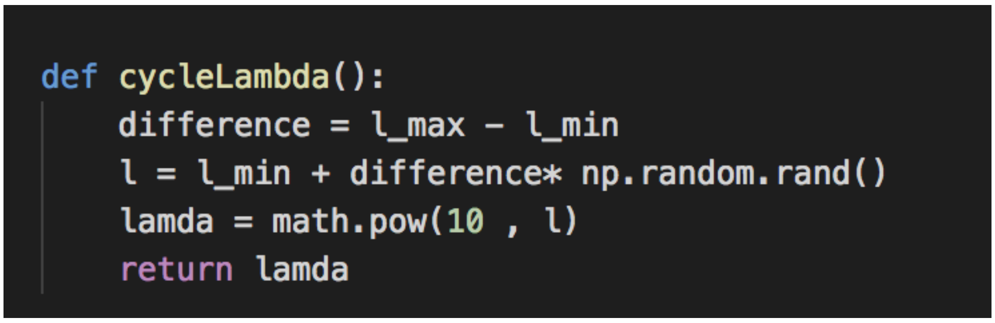


```
Test accuracy:  42.16
```

The curves are confirmed using figure 4 of the assignment instructions. I would justify the ups- and downs due to the learning rate going up and down in each cycle. Otherwise, the overall direction of the curves seems to be as they should. Accuracy is on the increase whereas loss and cost are on the decrease. Since the regularization coefficient is not 0, loss and cost values should not be the same and they are not. The accuracy of the training set is higher than that for the validation set and the cost of the training set is lower than that of the validation set.

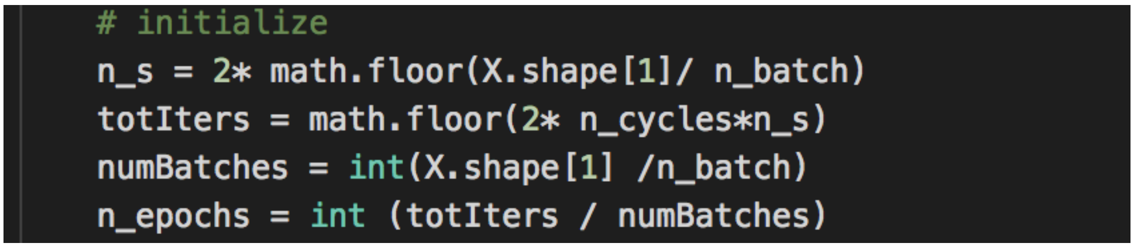
Dynamic lambda:

Lambda is the regularization coefficient of the network. Up until now, it has had a fixed value. However, I implemented the following function that dynamically varies lambda value.



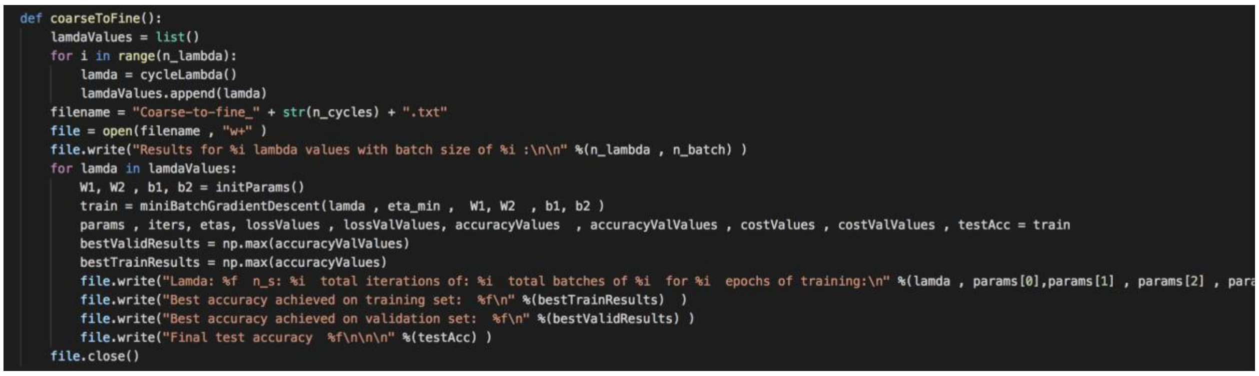
```python
def cycleLambda():
    difference = l_max - l_min
    l = l_min + difference* np.random.rand()
    lamda = math.pow(10 , l)
    return lamda
```

The parameters of the training such as n_s needed to change as well. Thus, I implemented dynamic assigning of the relevant parameters.



```python
    # initialize
    n_s = 2* math.floor(X.shape[1]/ n_batch)
    totIters = math.floor(2* n_cycles*n_s)
    numBatches = int(X.shape[1] /n_batch)
    n_epochs = int (totIters / numBatches)
```

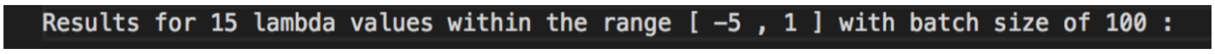
Having implemented the basic ingredients, I implemented the following function that performs greedy search in order for me to find the lambda value that results in the best model with the best performance.



```python
def coarseToFine():
    lamdaValues = list()
    for i in range(n_lambda):
        lamda = cycleLambda()
        lamdaValues.append(lamda)
    filename = "Coarse-to-fine_" + str(n_cycles) + ".txt"
    file = open(filename , "w+" )
    file.write("Results for %i lambda values with batch size of %i :\n\n" %(n_lambda , n_batch) )
    for lamda in lamdaValues:
        W1, W2 , b1, b2 = initParams()
        train = miniBatchGradientDescent(lamda , eta_min ,  W1, W2  , b1, b2 )
        params , iters, etas, lossValues , lossValValues, accuracyValues  , accuracyValValues , costValues , costValValues , testAcc = train
        bestValidResults = np.max(accuracyValValues)
        bestTrainResults = np.max(accuracyValues)
        file.write("Lamda: %f  n_s: %i  total iterations of: %i  total batches of %i  for %i  epochs of training:\n" %(lamda , params[0],params[1] , params[2] , para
        file.write("Best accuracy achieved on training set:  %f\n" %(bestTrainResults)  )
        file.write("Best accuracy achieved on validation set:  %f\n" %(bestValidResults) )
        file.write("Final test accuracy  %f\n\n\n" %(testAcc) )
    file.close()
```

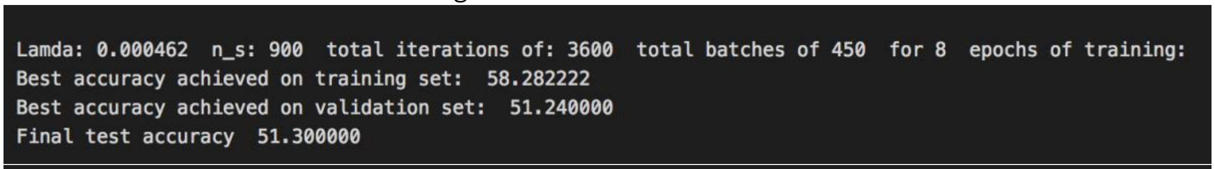
I varied the number of lambdas between 10- 15. For most of my experiments, I set n_lambdas to 15.
I then ran the following tests:

Test 1:

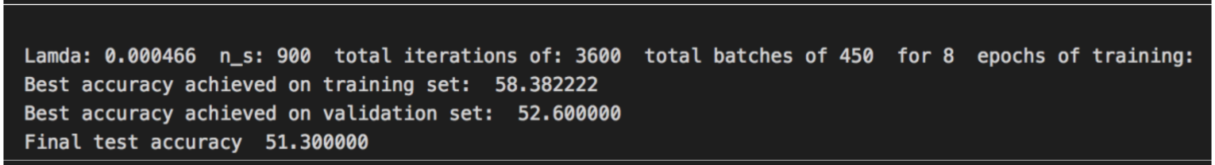


```
Results for 15 lambda values within the range [ -5 , 1 ] with batch size of 100 :
```

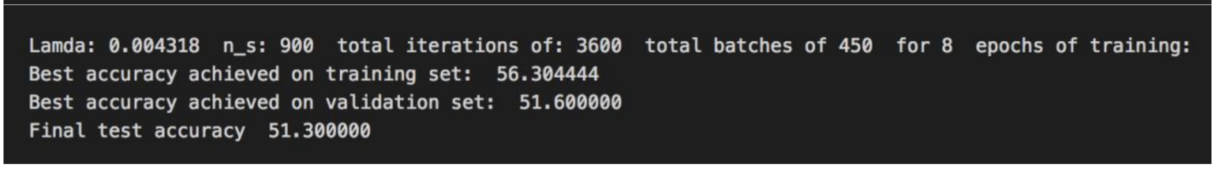
And the 3 best results in descending order:



```
Lamda: 0.000462  n_s: 900  total iterations of: 3600  total batches of 450  for 8  epochs of training:
Best accuracy achieved on training set:  58.282222
Best accuracy achieved on validation set:  51.240000
Final test accuracy  51.300000
```



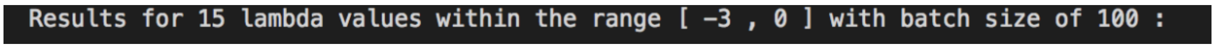
```
Lamda: 0.000466  n_s: 900  total iterations of: 3600  total batches of 450  for 8  epochs of training:
Best accuracy achieved on training set:  58.382222
Best accuracy achieved on validation set:  52.600000
Final test accuracy  51.300000
```



```
Lamda: 0.004318  n_s: 900  total iterations of: 3600  total batches of 450  for 8  epochs of training:
Best accuracy achieved on training set:  56.304444
Best accuracy achieved on validation set:  51.600000
Final test accuracy  51.300000
```

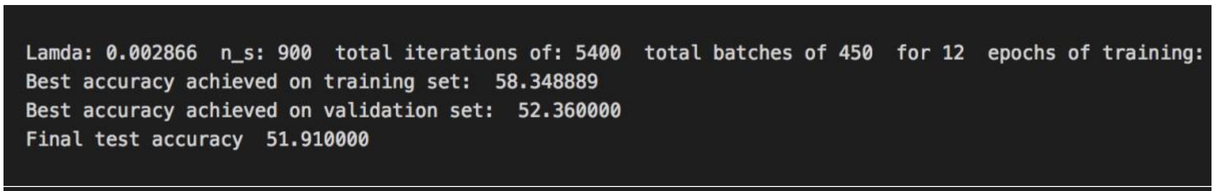
Looking at the results printed to the .txt file, I noticed that I should modify the [l_min, l_max] range to be more negative since they seemed to get better accuracies on the test set.

Test 2:

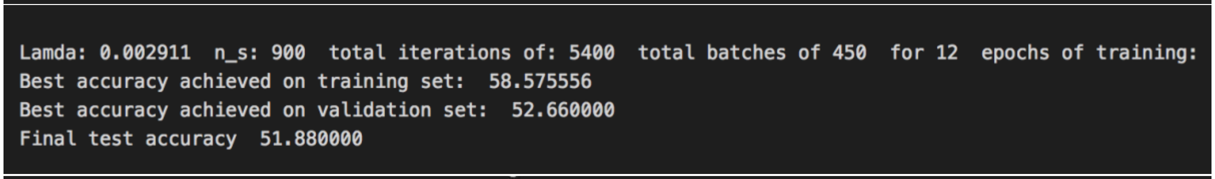


```
Results for 15 lambda values within the range [ -3 , 0 ] with batch size of 100 :
```

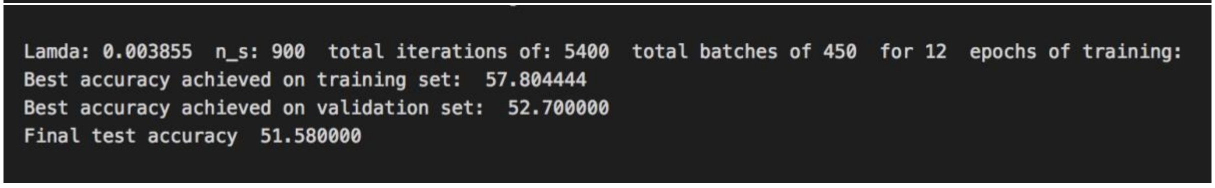
And the 3 best results in descending order:



```
Lamda: 0.002866  n_s: 900  total iterations of: 5400  total batches of 450  for 12  epochs of training:
Best accuracy achieved on training set:  58.348889
Best accuracy achieved on validation set:  52.360000
Final test accuracy  51.910000
```



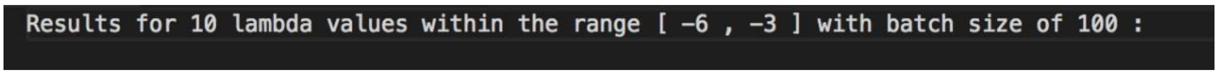
```
Lamda: 0.002911  n_s: 900  total iterations of: 5400  total batches of 450  for 12  epochs of training:
Best accuracy achieved on training set:  58.575556
Best accuracy achieved on validation set:  52.660000
Final test accuracy  51.880000
```



```
Lamda: 0.003855  n_s: 900  total iterations of: 5400  total batches of 450  for 12  epochs of training:
Best accuracy achieved on training set:  57.804444
Best accuracy achieved on validation set:  52.700000
Final test accuracy  51.580000
```

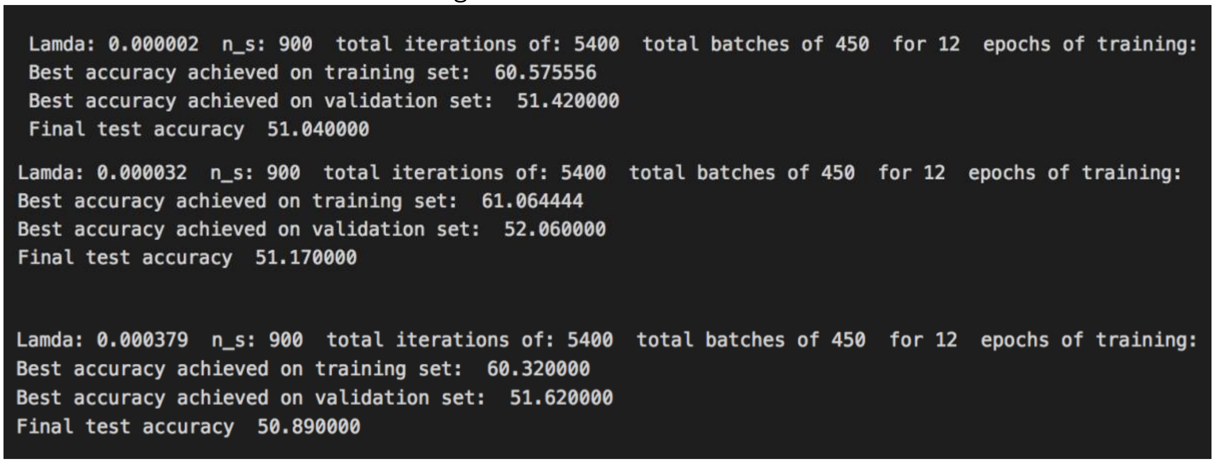
I then attempted to experiment with whether the results will get better if I push the [l_min, l_max] range to be more negative.

Test 3:



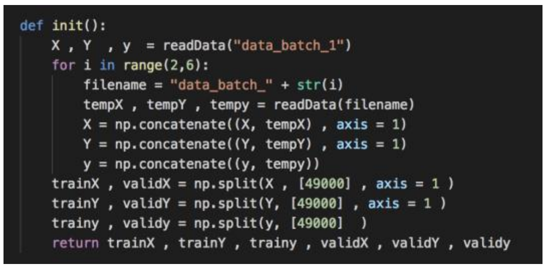
```
Results for 10 lambda values within the range [ -6 , -3 ] with batch size of 100 :
```

And the 3 best results in descending order:



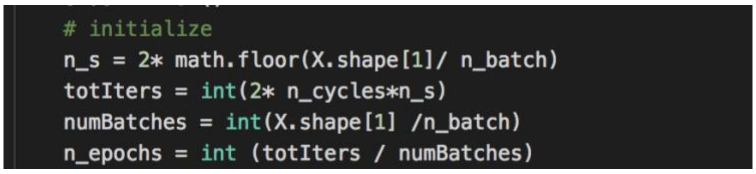
```
Lamda: 0.000002  n_s: 900  total iterations of: 5400  total batches of 450  for 12  epochs of training:
Best accuracy achieved on training set:  60.575556
Best accuracy achieved on validation set:  51.420000
Final test accuracy  51.040000
Lamda: 0.000032  n_s: 900  total iterations of: 5400  total batches of 450  for 12  epochs of training:
Best accuracy achieved on training set:  61.064444
Best accuracy achieved on validation set:  52.060000
Final test accuracy  51.170000


Lamda: 0.000379  n_s: 900  total iterations of: 5400  total batches of 450  for 12  epochs of training:
Best accuracy achieved on training set:  60.320000
Best accuracy achieved on validation set:  51.620000
Final test accuracy  50.890000
```

It turns out that the best performance was achieved in test case 2, when lambda is 0.002866. Thus, I tried training the entire network by using all 5 batches for mostly training data and only 1000 data points for validation as stated by assignment instructions. The split of data batches was as follows:



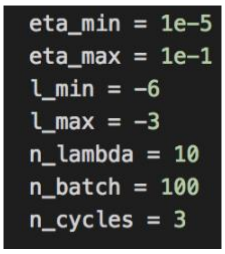
```
def init():
    X , Y  , y  = readData("data_batch_1")
    for i in range(2,6):
        filename = "data_batch_" + str(i)
        tempX , tempY , tempy = readData(filename)
        X = np.concatenate((X, tempX) , axis = 1)
        Y = np.concatenate((Y, tempY) , axis = 1)
        y = np.concatenate((y, tempy))
    trainX , validX = np.split(X , [49000] , axis = 1 )
    trainY , validY = np.split(Y, [49000] , axis = 1 )
    trainy , validy = np.split(y, [49000]  )
    return trainX , trainY , trainy , validX , validY , validy
```

The dynamic parameters are defined as follows where n_batch is 100.



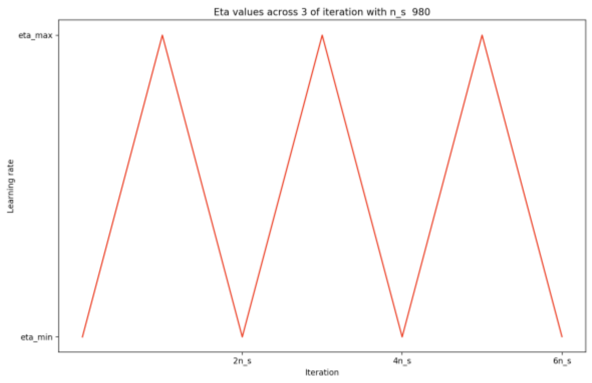
```
    # initialize
    n_s = 2* math.floor(X.shape[1]/ n_batch)
    totIters = int(2* n_cycles*n_s)
    numBatches = int(X.shape[1] /n_batch)
    n_epochs = int (totIters / numBatches)
```

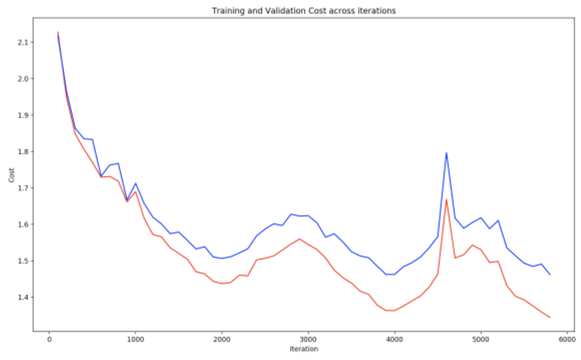
And the rest of the parameters that are fixed are as follows:

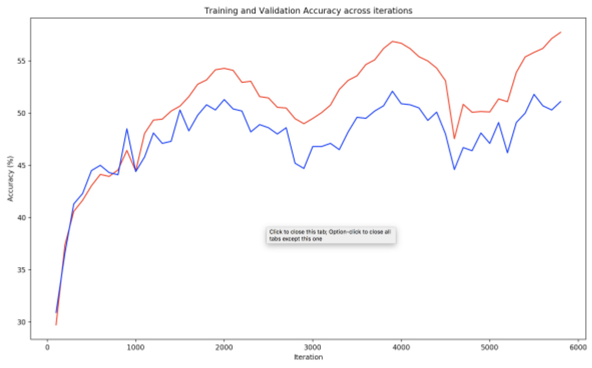


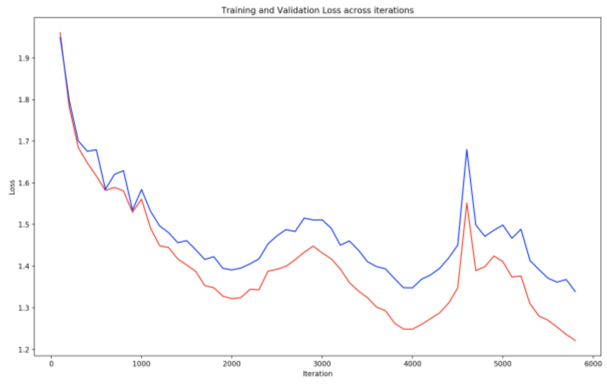
```
eta_min = 1e-5
eta_max = 1e-1
l_min = -6
l_max = -3
n_lambda = 10
n_batch = 100
n_cycles = 3
```

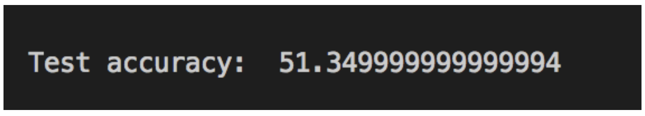
Training the network in these settings resulted in the following:











```
Test accuracy:  51.349999999999994
```