1. Introduction

In this assignment, the task was to create a two layered neural network. To compute the gradients, the approach of mini batching was selected. The idea of mini batching is, for each epoch, to iteratively select random subsets (batches) of the training data, make predictions on the subsets, and ultimately update the model's parameters before the next epoch. Additionally, the approach of cyclical learning rates was used in order to enhance the network's accuracy. This means that the learning rate was increased and decreased periodically during training. Furthermore, a random and fine grid search for the best lambda value was done in order to regularize the network optimally.

For forward pass, the softmax function was used to determine the probabilities of an image belonging to each class. For the back propagation in this model, cross-entropy was used as a loss function. By combining the cross-entropy loss and a regularization term, the full cost function was defined.

2. Analytic gradient and numerical gradient tests

The model used an analytical approach in order to compute the gradients of each mini batch. In order to make sure that the analytical gradient was successfully computed, it was compared to a function that rather calculated the numerical gradient. The tests were done using various (non-random) dimensions of the first training data instance. To pass the test, the absolute relative error for the model's weights (W) and bias (b) should not exceed 1e-5. Below, the test implementation and results are depicted and conclusively, the testing of a small training batch example passed. The function below was called with a threshold of 1e-5 and a reduced dimensionality of the input vectors equivalent to the assignment instructions.

```
def CompareGradients(act_vals, X,Y, W1, W2, b1,b2, lambd, threshold):

    P, act_vals = EvaluateClassifier(X, W1, W2, b1,b2)

    #Calculate gradients
    grad_W1_a, grad_W2_a, grad_b1_a, grad_b2_a = ComputeGradients(act_vals, X, Y,P, W1,W2, lambd)
    grad_W1_n,grad_W2_n, grad_b1_n, grad_b2_n = ComputeGradsNum(W1, W2, b1,b2, X, Y,lambd, h=0.00001)

    # Calculate differences
    w_rel_error_1 = np.sum(np.abs(grad_W1_a - grad_W1_n)) / np.maximum(0.001, np.sum(np.abs(grad_W1_a) + np.abs(grad_W1_n)))
    w_rel_error_2 = np.sum(np.abs(grad_W2_a - grad_W2_n)) / np.maximum(0.001, np.sum(np.abs(grad_W2_a) + np.abs(grad_W2_n)))

    b_rel_error_1 = np.sum(np.abs(grad_b1_a - grad_b1_n)) / np.maximum(0.001, np.sum(np.abs(grad_b1_a) + np.abs(grad_b1_n)))
    b_rel_error_2 = np.sum(np.abs(grad_b2_a - grad_b2_n)) / np.maximum(0.001, np.sum(np.abs(grad_b2_a) + np.abs(grad_b2_n)))

    # Check differences
    if (w_rel_error_1 and w_rel_error_2) and (b_rel_error_2 and b_rel_error_1) < threshold:
        print("Analytical ok")
    else:
        print("Gradient difference too high")
```

Figure 1: Implementation of analytic gradient testing.

```
In [1]: runfile('/Users/akseluhr/Documents/test/untitled0.py', wdir='/
Users/akseluhr/Documents/test')
Cost:  2.8426652682206583
Accuracy:  0.0898
Analytical ok
```

Figure 2: Test results of the analytic gradient testing.

3. Results

3.1 First setting

In the first setting, only one cycle was used which explains why the curves are similar to the previous assignment. Hence, the periodical change obtained by the cycle learning rate approach is not expressed in the graphs, since the cycle is constrained (i.e. non-existing) by setting it to 1. The cost curve seems reasonable (~1.5 vs ~1.85) but given the difference in the accuracy curves (~.60 vs. ~.42), the model is overfitted.

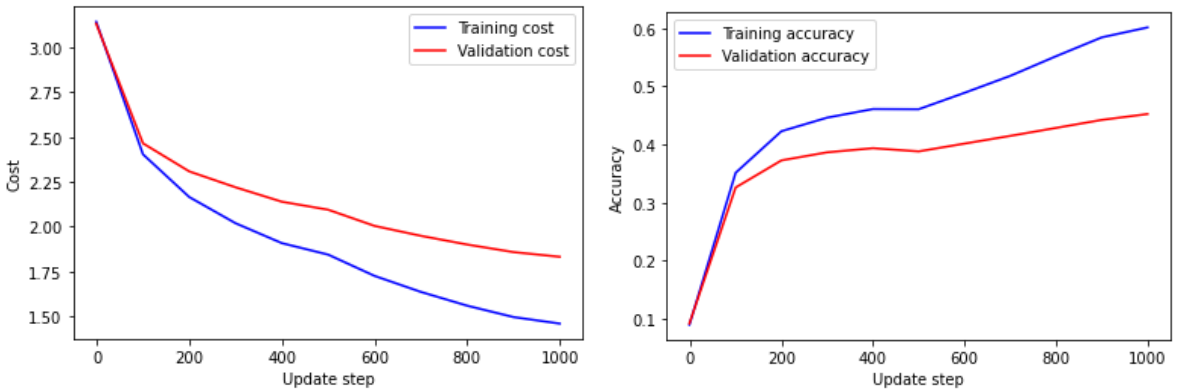

Figure 3: First parameter settings cost and accuracy. Lambda = 0.01.

3.2 Second setting

For this setting, the number of cycles were increased to 3 which is clearly depicted below with the local minimums and maximums. This is the result of the variation of nt. The value of n_s was increased to 800 and the model is still overfitted. Probably even worse now given the difference in the cost of training and validation, as well as their accuracies (~.71 vs ~.43). It is somewhat expected though, since the weights are initialized w.r.t. the training data, and so far, the optimal regularization term (i.e. value of lambda) has not been found.

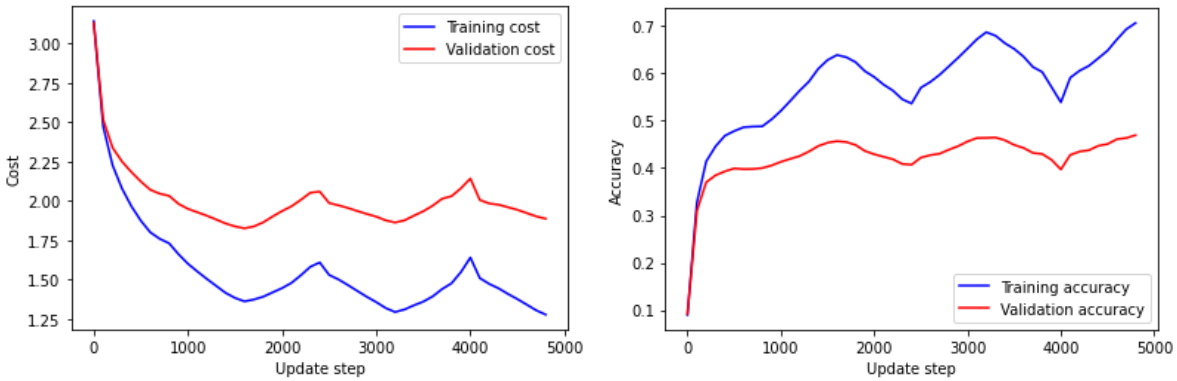

Figure 4: Second parameter setting results and the learnt weight matrix. Lambda = 0.01.

3.3 Lambda coarse search

Below are the randomly generated lambda values used for the coarse search. The lambdas were generated on a log scale.

```
[0.0004950159553733192, 0.0005627932047415164, 0.0015119336467640998, 0.0015676677195506057,
0.002576638574613588, 0.0038333321561566606, 0.007257005721594274, 0.03690557729213758]
```

Figure 5: Third test. Lambda values for the coarse search.

Depicted below is a random search for an optimal lambda value. The graphs can be summarized as the higher the lambda value and thus regularization term, the higher loss and lower accuracy. Given the performance graphs, the lambda values of ~0.0005 and ~0.00156 are good indicators on what values to select for the fine search.

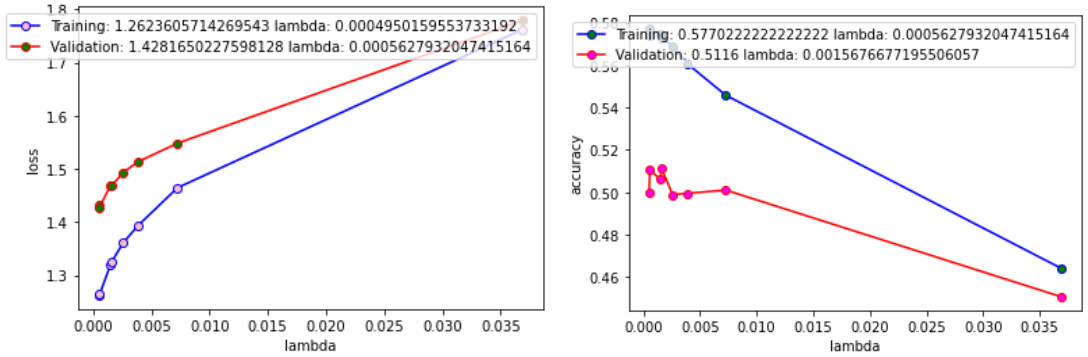

Figure 6: Third test. Results of a random search of an optimal lambda value.
Hyper-params: eta_min=1e-5, eta_max=1e-1, step_size=800, n_batch=100, cycles=2.

3.4 Lambda fine search

Below are the lambda values used for the fine search. The lambdas were manually added to a list of lambdas and inputted in the grid search function, having a range with respect to the results from the coarse search.

```
[0.00025, 0.0005, 0.001, 0.0015, 0.002, 0.0025, 0.003, 0.005]
```

Figure 7: Fourth test. Lambda values for the fine search.

Results indicate that the lower the lambda, the lower the cost. But given the highest accuracy on the validation set when lambda = 0.002, this is the final lambda value that will be used for the test set.

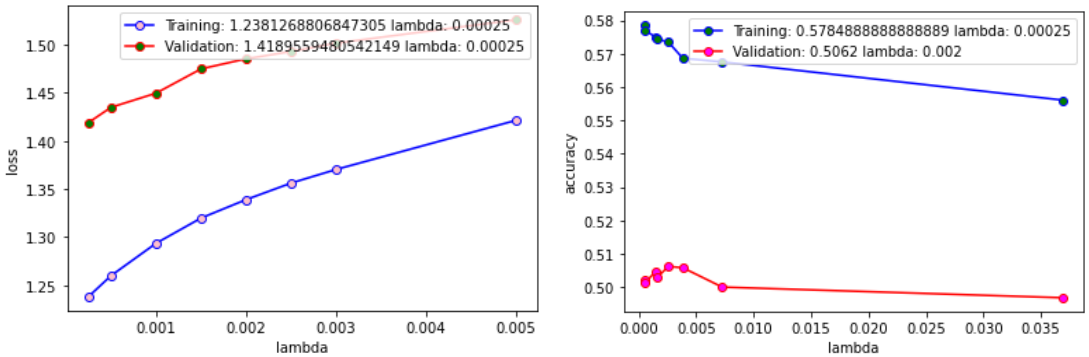

Figure 8: Forth test. Results of a fine search of an optimal lambda value.
Hyper-params: eta_min=1e-5, eta_max=1e-1, step_size=800, n_batch=100, cycles=2.

3.4 Testing the network on the test set

Final tests with 1000 validation data instances are depicted below. Compared to the two first tests (see fig 3 and 4), the validation and training curve does not differ as much, which indicates that the model is less overfitted after searching for the optimal lambda value for the regularization term. The accuracy for the test set achieved a score of 51.43 % which is *slightly* better than guessing. Nice! :)

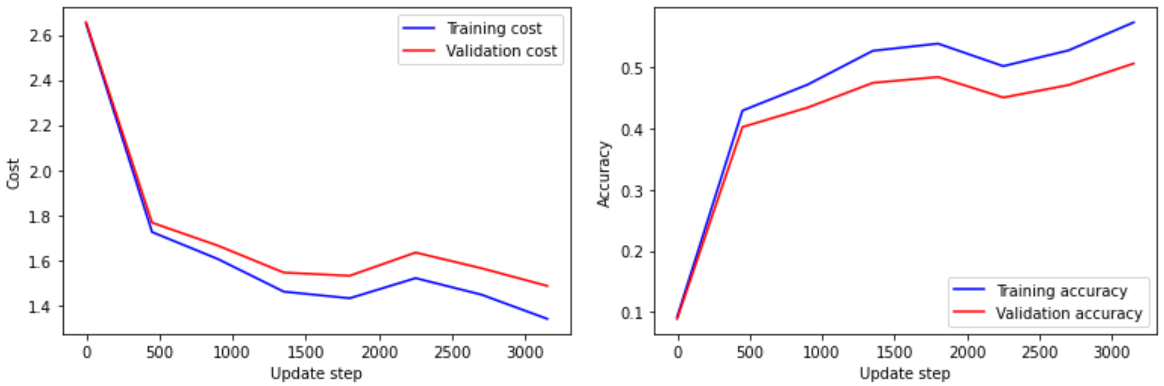

Figure 9: Final test. Results of the network with an optimal lambda value (=0.002).
Hyper-params: eta_min=1e-5, eta_max=1e-1, step_size=800, n_batch=100, cycles=3.

```
Final Accuracy on test set:  0.5143
```

Figure 10: Final test set accuracy.

4. Final remarks

Note: This is my reflection. Feedback is gladly welcome on this part.

Since the weights were initialized with respect to the training data, the model, without regularization, will be biased towards this population. In essence, this means that the model will perform well on data which is similar to the training data, but not achieve the same performance on unseen data (e.g. validation data). This is what the results in figure 4 and 3 are showing: fluctuations in training data in terms of accuracy and loss, whereas the validation data scores significantly lower. In other words, the value of lambda was not optimal as the regularization did not affect the model to the most optimal extent. However, grid searching for an optimal value of lambda resulted in a decrease in the difference of the performance (training vs validation) since large weights (again, drawn from the training data) were penalized in a more favorable way, as seen in figure 9.