Assignment 2

DD2424 Deep Learning in Data Science

Márk Antal Csizmadia
macs@kth.se

April 2021

1 Introduction

This short report describes my solution to Assignment 2, the implementation of which may be found in the script uploaded alongside with the report.

2 Implementation

In this section, a brief description of my implementation is followed by the explanation of the numerical gradient check.

2.1 Neural network blocks

In my implementation, I built on top of my code from Assignment 1. At the core of my implementation is the `Dense` layer (fully-connected layer) that can have activations `SoftmaxActivation`, and `ReLUActivation`. The learnable (or trainable) parameters of the `Dense` layer can be initialized with the `XavierInitializer`, and can be regularized with the `L2Regularizer`. A `Model` comprises any number of layers, its loss function is `CategoricalCrossEntropyLoss`. The trainable parameters are optimized with the mini-batch gradient descent algorithm via the `SGDOptimizer` and the learning rate schedule of the optimizer is `LRCyclingSchedule` [1]. The `Model` is first compiled with the loss function, some metrics such as the `AccuracyMetrics`, and the optimizer, and then it is fit to the data with the `Model.fit` method.

In this assignment, I implemented a two-layer neural network for multi-class classification such that both of the `Dense` layers' trainable parameters are initialized with the `XavierInitializer` and regularized with the `L2Regularizer`. The input dimension and the output dimension of the first `Dense` layer were 3072 (=32*32*3) and 50, respectively. The input and output dimensions of the second `Dense` layer were 50 and 10 (=the number of class labels), respectively. The first `Dense` layer has `ReLUActivation`, and the second `Dense` layer has `SoftmaxActivation`. The `Model` is then compiled with the `CategoricalCrossEntropyLoss` loss function, the `AccuracyMetrics` performance metric, and the `SGDOptimizer` optimizer that uses the `LRCyclingSchedule` learning rate schedule.

2.2 Numerical gradient check

The correctness of the gradient computation and backpropagation was assured via numerical gradient check. I implemented the double centered gradient check with the combination of three methods - `grad_check_without_reg`, `get_num_gradient`, and `loss_func`. The `grad_check_without_reg` function iterates through the learnable weights of the model, computes the analytical and numerical gradients, and then compares them. The full functionality of the gradient check is in `test_grad_check`. The gradients were checked with 20 data points and the first 10 dimensions of the 3072 dimensions of the data to make the running time realizable on my CPU.

In the case of the aforementioned model, in the first `Dense` layer, the maximum relative error among the values in 3072x50 analytical and numerical gradients with respect to the weights was 5.906446e-07, while the same metric was 1.420300e-08 in the case of the 1x50 gradients with respect to the bias. The same measures in the second `Dense` layer for the 50x10 weight and 1x10 weight and bias gradients were 1.649889e-06 and 4.036460e-09, respectively.

As outlined in [2], the aforementioned discrepancies are satisfactory to assert that the analytical gradients of the model are correct. To avoid kinks in the objective function, only a few, namely 20, data points, and the first 10 dimensions were used in the gradient check. The step size for computing the numerical gradient was set to 1e-06. In computing the relative error, the formula provided in the Assignment 1 document was used with a slight modification based on [2]. Furthermore, the `numpy.testing.assert_array_almost_equal` function was also used to make the comparison.

3 Results

Please note that the accuracy measures are given as real numbers between 0.0 and 1.0, where 0.0 mean 0.0% and 1.0 means 100.0%.

3.1 Replicating Known Cases

To make sure that my implementation was correct, I first reproduced Figure 3 and Figure 4 in the laboratory guide. The training data was `data_batch_1`, the validation data was `data_batch_2`, and the testing data was `test_batch`. All of the training, validation, and test set included 10000 images. The data was pre-processed as described in the laboratory guide.

The results of the run for Figure 3 are shown in Figure 1. The `Model` is as described before. The L2 regularization hyper-parameter (`lambda` in the laboratory guide) was set to 0.01. The `LRCyclingSchedule` cyclical learning rate schedule's base learning rate was $1e-5$ and the maximum learning rate was $1e-1$. The step size (half-cycle) of the `LRCyclingSchedule` was $n\_s = 500$ (i.e.: one fully cycle is 1000 steps). The training on the training set of 10000 images went on for 10 epochs with a mini-batch size of 100 that is equivalent to 10 000 / 100 = 1000 update steps, or equivalently for 1 full-cycle.

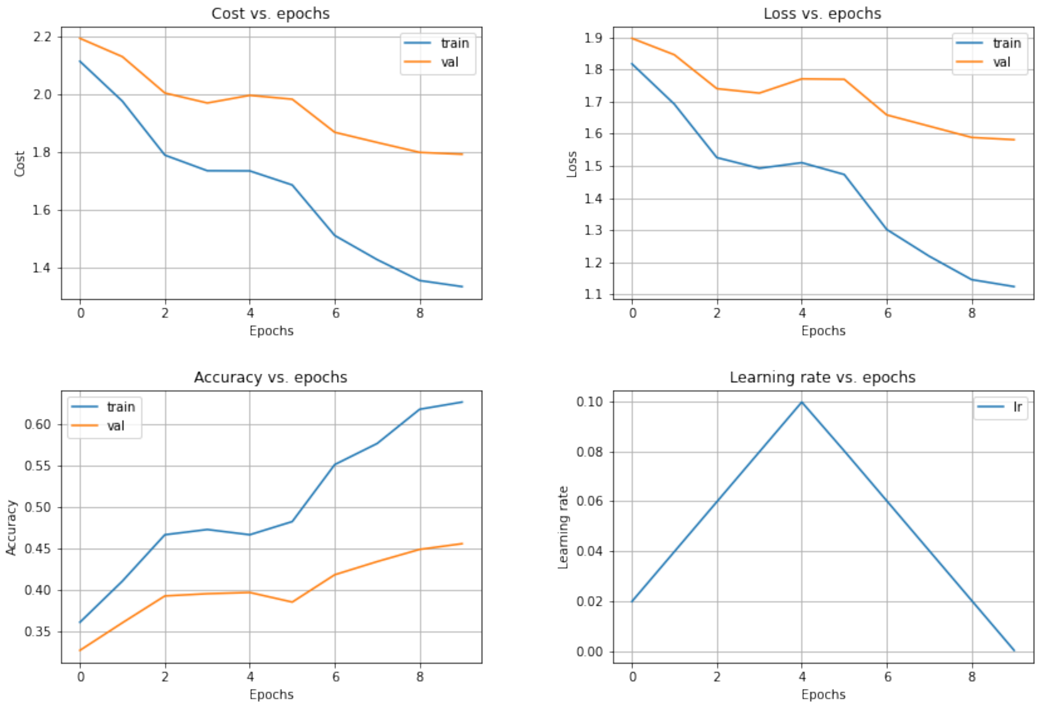

Figure 1: The cost, loss, accuracy, and learning rate curves of training for 10 epochs with a mini-batch size of 100 (equivalent to 1000 update steps or 1 full cycle). Note that accuracy measures are given as real numbers in the range 0-1.

Note that the reason for the learning rate curve to be shifted is that the learning rate is always recorded at the end of an epoch (i.e.: after the learning rate has been updated for 100 update steps). A shown in Figure 1, the replication is almost identical to Figure 3 in the laboratory guide, and therefore confirms the correctness of my implementation. The test accuracy of the trained model on `test_batch` was 0.4572.

The results of the run for Figure 4 are shown in Figure 2. The `Model` is as described before. The L2 regularization hyper-parameter (`lambda` in the laboratory guide) was set to 0.025, instead of 0.01 since my curves looked slightly different that way. The `LRCyclingSchedule` cyclical learning rate schedule's base learning rate was $1e-5$ and the maximum learning rate was $1e-1$. The step size (half-cycle) of the `LRCyclingSchedule` was $n\_s = 800$ (i.e.: one fully cycle is 1600 steps). The training on the training set of 10000 images went on for 48 epochs with a mini-batch size of 100 that is equivalent to 48 000 / 100 = 4800 update steps, or equivalently for 3 full-cycles.

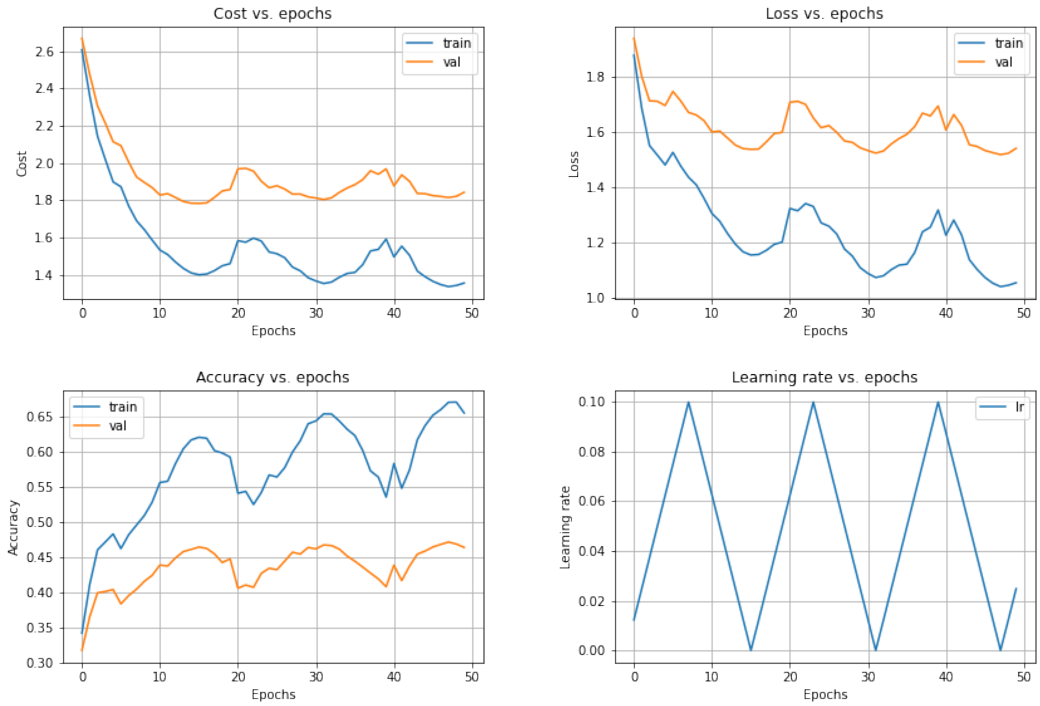

Figure 2: The cost, loss, accuracy, and learning rate curves of training for 48 epochs with a mini-batch size of 100 (equivalent to 4800 update steps or 3 full cycles). Note that accuracy measures are given as real numbers in the range 0-1.

Note that the reason for the learning rate curve to be shifted is that the learning rate is always recorded at the end of an epoch (i.e.: after the learning rate has been updated for 100 update steps). A shown in Figure 2, my replication is almost identical to Figure 4 in the laboratory guide, and therefore confirms the correctness of my implementation. The test accuracy of the trained model on `test_batch` was 0.4713.

3.2 Coarse-to-fine Random Search of the L2 Regularization Hyper-Parameter, `lambda`

In this section, the coarse-to-fine random search to find the set the L2 regularization hyper-parameter (`lambda` in the laboratory guide) is discussed.

In this part of the laboratory, all of the 5 batches were used for training and validation. The 5 batches were concatenated and 5000 randomly selected images were separated into the validation set. The test set was the same as before. As a result, the training, validation, and test set comprised 45000, 5000, and 10000 images, respectively.

The step size `n_s`, base learning rate, and maximum learning rate of the `LRCyclingSchedule` learning rate schedule were set to $n\_s = 2 \times \text{floor}(n\_data/n\_batch) = 2 \times \text{floor}(45000/100) = 2 \times 450 = 900$, $1e-5$, and $1e-1$, respectively. The mini-batch size was set to 100. Therefore, one fully cycle was $2 * 900 = 1800$ update steps, or equivalently, $(1800 * 100)/45000 = 180000/45000 = 4$ epochs. The `Model` was the same as before otherwise. In each run of the hyper-parameter search, the model was trained for 2 full

LRCyclingSchedule cycles, or equivalently, for 8 epochs.

In each run of the coarse search, the L2 regularization learning rate lambda was sampled as

$$lambda = 10^{l_{min}+(l_{max}-l_{min})*U(0,1)} \tag{3.1}$$

where $U(0,1)$ is a random number sampled from the uniform distribution between 0 and 1. In the coarse search 10 different, randomly sampled values were explored. The lambda of the model with the best performance on the validation set, lambda_best_coarse was then used in the subsequent fine search. In the fine search, the L2 regularization learning rate lambda was sampled as

$$lambda = U(0.8 \times lambda\_best\_coarse, 1.2 \times lambda\_best\_coarse) \tag{3.2}$$

that is, a narrow interval around the lambda_best_coarse was futher explored. In the fine search, 10 different, randomly sampled values were explored.

The results of the coarse-to-fine search are shown in Table 1 and 2, respectively. Table 1 shows the results of the coarse search, and Table 2 shows the results of the subsequent fine search. Note that accuracy measures are given as real numbers in the range 0-1.

lambda	validation accuracy
0.000426632	0.5098
0.000510532	0.5024
3.665951E-05	0.5090
0.000932242	0.5034
0.079010204	0.4426
3.495252E-05	0.5044
0.007931035	0.5058
0.011035776	0.5058
0.000499186	0.4986
0.073193151	0.4432

Table 1: Coarse hyper-parameter search for lambda

As shown in Table 1, the model with the highest validation accuracy of 0.5098 turned out to be the one with lambda = 0.000426632. Therefore, in the fine search, more lambda parameters were explored around this value, as shown in Table 2. Note that accuracy measures are given as real numbers in the range 0-1.

lambda	validation accuracy
0.000410849	0.5040
0.000414176	0.5056
0.000365376	0.5134
0.000425332	0.5020
0.000507594	0.5020
0.000364492	0.5060
0.000465001	0.5044
0.000471122	0.5026
0.000413759	0.5042
0.000506177	0.5044

Table 2: Fine hyper-parameter search for `lambda`

As shown in Table 2, the model with the highest validation accuracy of 0.5134 turned out to be the one with `lambda`=0.000365376. Also note that generally the validation accuracy is greater in Table 2 than in Table 1, which supports that the fine search is exploring a worthwhile interval around the best value found in the coarse search.

Using the best `lambda` value of 0.000365376, a final model was trained. The dataset was altered so that 49000 and 1000 randomly sampled images comprised the training and the validation set, respectively. The test set was the same as before.
The step size `n_s`, base learning rate, and maximum learning rate of the `LRCyclingSchedule` learning rate schedule were set to $n\_s = 2 \times \text{floor}(n\_data/n\_batch) = 2 \times \text{floor}(49000/100) = 2 \times 490 = 980$, $1e-5$, and $1e-1$, respectively. The mini-batch size was set to 100. Therefore, one fully cycle was $2 * 980 = 1960$ update steps, or equivalently, $(1960 * 100)/49000 = 196000/49000 = 4$ epochs. The `Model` was the same as before otherwise, and it was trained for 3 full cycles, or equivalently, for 12 epochs. The results of training the final model are shown in Figure 3.

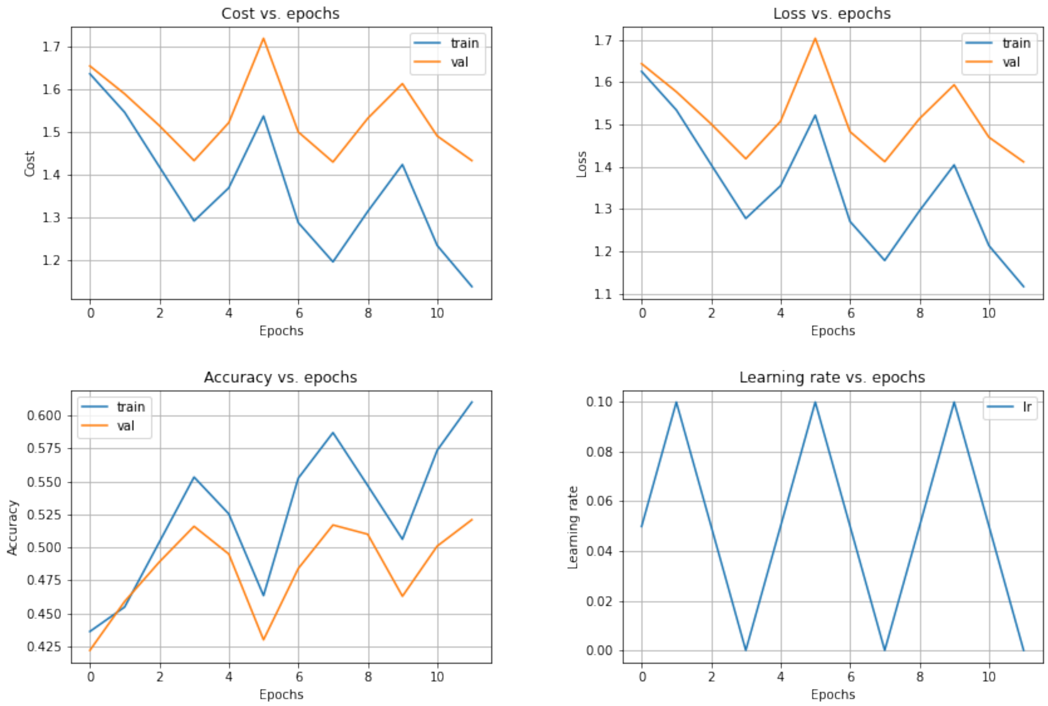

Figure 3: The cost, loss, accuracy, and learning rate curves of training the final model after hyper-parameter search for 12 epochs with a mini-batch size of 100 (equivalent to 3 full cycles). Note that accuracy measures are given as real numbers in the range 0-1.

Note that the reason for the learning rate curve to be shifted is that the learning rate is always recorded at the end of an epoch (i.e.: update steps have already taken place). The final model's test accuracy was 0.5170.

4 Conclusions

In this assignment, a two-layer neural network for multi-class classification was trained and evaluated on the CIFAR-10 data set. For optimizing the model, the cyclical learning rate schedule was applied. The best value of the L2 learnable parameter regularization rate, or `lambda`, was found via coarse-to-fine search. The final model with this `lambda` achieved a test accuracy of 0.5170, or equivalently, 51.70 %.

References

[1] L. N. Smith, "Cyclical learning rates for training neural networks," in *2017 IEEE winter conference on applications of computer vision (WACV)*, IEEE, 2017, pp. 464–472.

[2] CS231n Team, *Learning*. Stanford University, 2020. [Online]. Available: `https://cs231n.github.io/neural-networks-3/`.