Deep Learning Lab - Report - Exercise 1

Rabea Laura Turon

October 30, 2018

1 Introduction

This report gives a summary of the implementation and testing process of Exercise 1 of the Deep Learning Lab Course 2018 at the Albert-Ludwigs-University of Freiburg. The first task was to implement a general framework for Multilayer Perceptrons (from now on abbreviated to MLPs) based on some code structure provided by the university. As a second task this framework should be used to train some MLPs on the MNIST dataset to classify digits from a test data set with an accuracy of about 1% to 3%.

2 MLP framework implementation

I will not go into detail about the implementation of the marked code snippets since their function was clearly defined by the given comments and I do not think that the specific implementation is of interest but that it only matters that it works. However I do admit that the only way I checked the correctness of the implementation was to use the gradient checking provided in the code and to see if the training of the networks worked as expected, i.e. that the loss decreased over time and that stochastic gradient descent had a faster rate of decrease than the normal gradient descent.

Additionally to the marked code snippets I implemented weight decay as a regularization technique for the parameters of the layers. From a theoretical point of view this means adding a second term to the loss function which adds the squared values of the weights to the already used loss function. Since the goal is to minimize the overall loss this leads to a trade off between minimizing the classification or regression error and minimizing the weights' values. In general this leads to smaller weights which means that the network overfits less.

Moreover I changed the training function to not only print the loss and error values during training, but return them in order to be able to plot them later. I also added a function that takes these returned values as input and then plots the curves of the loss and error progress. An example of such curves can be seen in Figure 1.

3 MNIST classification

3.1 The task

The MNIST dataset consists of 28x28 pixel images of handwritten digits. It is typically divided into a training set of 50,000 images, a validation set of 10,000 images and a test set of 10,000 images. The task is to train a MLP on the training set that classifies the ten digits from zero to nine correctly. The validation set is used during the training process to validate the generalization properties of the network on data that the network has not used for training. As a final assessment of the network the accuracy of the network on the test set can be used.

3.2 Testing procedure

Even though the time was limited I tried to perform an exhaustive test of the hyperparameter space. By some example networks – inspired by the already given example network – I found a set of hyperparameters that seemed to work well for most network architectures. These were 0.1 as the initialized standard deviation of the weights, 0.1 as the learning rate, a batch size of 64 and no weight decay or a weight decay of 0.001. Then I performed a grid search over the just mentioned hyperparameters (two values were tested for each one) and the additional hyperparameters network depth (between 2 and 4 including the fully connected layer with ten units for the classification), number of units per layer (three different values for each layer) and learning algorithm, which was either gradient descent or stochastic gradient descent. As activation function I only tried ReLUs because these are normally used for classification on images. I only trained the networks during this grid search for five epochs because on the one hand I did not have the time to do more and on the other hand I hoped to already see from these five epochs if the networks learned anything and how fast.

The results showed that the hyperparameters 0.1 for the initialization of the standard deviation of the weights, 0.1 for the learning rate, 128 as batchsize, 0 for the weight decay and stochastic gradient descent worked well on all tested network architectures. Therefore I chose some of the better performing architectures and trained them again with these hyperparameters for 30 epochs. Note that all the testing up to here was not performed on the whole training and validation set, but only on 20,000 images of the training set and 200 images of the validation set.

3.3 Results

Most of the networks from the last testing round performed similar with a classification error between 2% and 4% on the smaller validation set. However most of them also seemed to overfit since their classification error on the smaller training set was either 0 or 0.0001. Three improvements could help with this problem. First, the implemented weight decay could help. Second, using the whole dataset instead of just 20,000 samples could already lead to less overfitting. And third, training could be stopped after the training error goes below a certain threshold. Of course, other regularization techniques like dropout could also help but were not implemented here. On the other hand it cold also be the case that the networks are simply to big and the only way to avoid overfitting is to make them smaller.

However I had to choose one network with a good performance. I decided to use a smaller network to avoid too much overfitting and because simpler solutions are normally preferred. The final network had 3 layers, 100 units in the first layer, 50 units in the second layer and the 10 units for classification in the final layer. You can see the networks' training in Figure 1. The curves of the training and validation set are not too far from each other which suggests that the hyperparameters are chosen well and the network actually generalizes. The final test error is 2.4%.

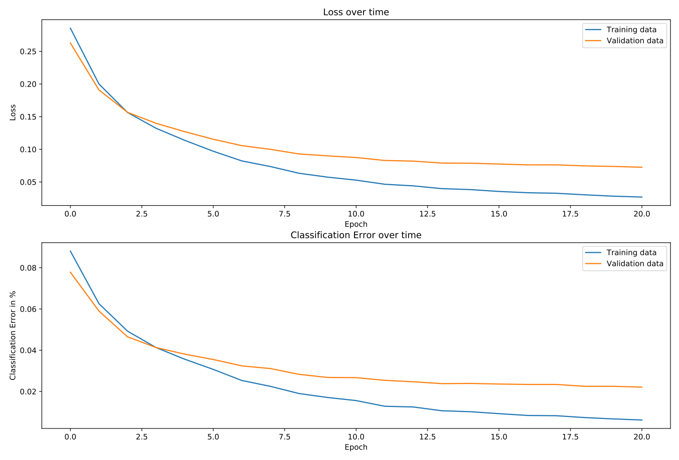

Figure 1: Development of the loss and classification error on the training and validation set during training. A network with 3 layers (100,50,10 units) was used. Initialization of standard deviation: 0.1, learning rate: 0.2, weight decay: 0.0005, batchsize: 128, number of epochs: 20