Deep Learning Lab - Report on the Second Exercise

Joshua Heipel

matriculation number: 3706603

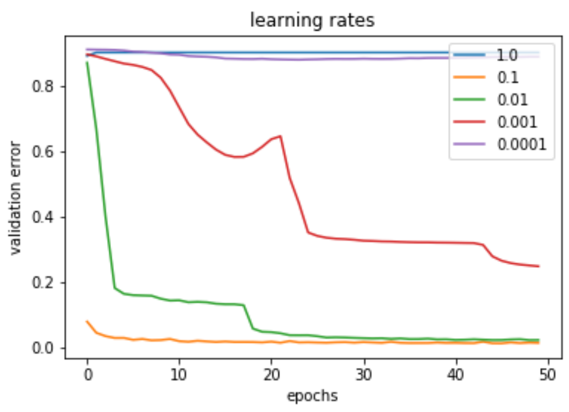
1. Implementing a CNN in Tensorflow

The first part of the exercise was to implement a convolutional neural network with Tensorflow. Instead of using the simple API "Keras" a specific framework for the tasks of this exercise was created. The implemented classes and functions contained in the "CNN.py" file are mainly based on the implementation of a fully connected neural network from the last exercise sheet. A pooling option is directly integrated into the convolutional layers. The output layer comes with a softmax activation function in order to calculate the loss. Training and validation of the network are implemented inside the given dummy functions of "cnn_mnist.py" file.

2. Learning Rate

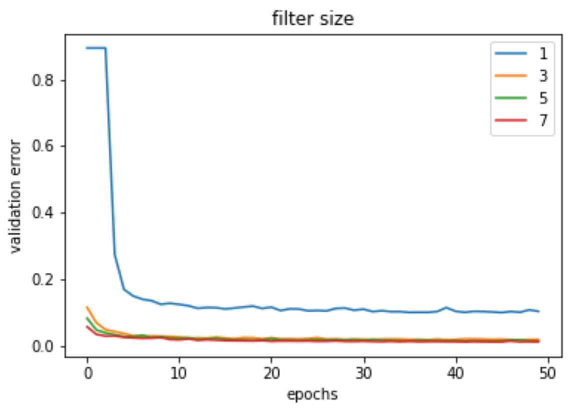
In the second exercise the learning rate of the stochastic gradient descent algorithm was gradually changed. The following figure shows the learning curves (validation error) for different parameter values after 50 epochs. At a low learning rate of 0.0001 the validation error doesn't improve significantly. With increasing values (0.001 - 0.1), a greater decline can be found - with some plateaus and transient climbs in between. A high learning rate of 1.0 doesn't improve of the validation error at all.

From the figure it can be concluded, that the learning rate is an important optimization parameter. At low values the convergence of the algorithm towards a minimum is immensely down immensely. At very high values the step size is too large to reduce the objective function and the algorithm doesn't converge at all. Because the minimization is calculated on the loss rather than on the validation error, the illustrated learning curves show some plateaus and temporary increases.



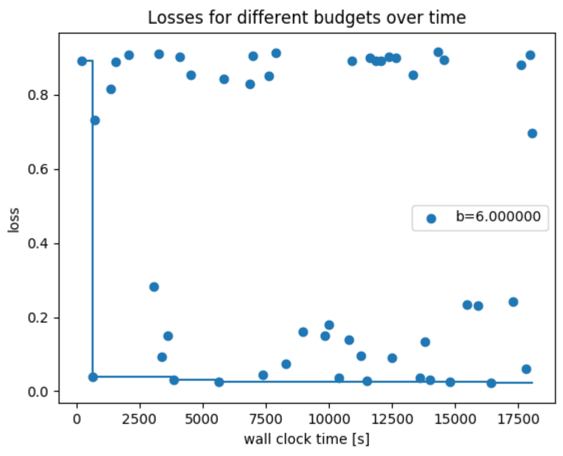
3. Convolution Type

The third part of the exercise was to try out different network architectures by varying the filter sizes of the convolutional layers while keeping the rest of the configurations unchanged. The learning curves for various parameters after 50 training epochs are shown in the figure below. In all cases the validation error decreases with the number of training cycles. Regardless of the filter size, convergence is achieved after 10-15 epochs. With a size of 1x1 the validation error is about 10.5%. With larger filters it can be further reduced while the difference between the configurations becomes smaller. The lowest validation error of about 1.4% is achieved with a filter size of 7x7. As a major drawback, the runtime of the gradient descent optimization algorithm increases immensely with larger filter sizes. Therefore, large filters must be used carefully. In case of small differences between the gray values of adjacent pixels large filters shouldn't give much advantage over small filters. However, if the size is too small, the validation error becomes significantly larger.



4. Random Search

For the last exercise an automatic hyperparameter optimization scheme should be used to find the best configuration of the parameters learning rate, batch size, number of filters and filter size. The training and validation function as well as the configuration space are defined inside the given "random_search.py" file. Optimization was then run for 50 iterations with a budget of 6 training epochs each. The following figure shows the calculated losses for various randomly selected parameters. As the distribution of points indicates, the differences in the remaining loss are very high. The best result was achieved with a learning rate of 0.066 a batch size of 36, a number of 33 filters per layer and a filter size of 5x5. The remaining test error was about 2.4%.


Losses for different budgets over time
loss
wall clock time [s]
b=6.000000