First exercise: Classifying MNIST with MLPs

David-Elias Künstle

November 7, 2017

1 Introduction

In the following I report about my implementation and application of a multilayer perceptron (MLP) for handwritten digit classification with the MNIST dataset.

I briefly describe my problems while implementation but focus more on applying the MLP and an excursion to regularization, because for the former a stricter course was already given.

2 Implementation

The implementation of the MLP was done according to the given framing, wherefore I don't go into detail here. The framing was only changed by moving some code to separate functions to reuse, adding regularization (see section 4) and storing the error rate history while training for plotting. Difficult was, that the parts of the MLP all depend on each other, such that debugging can only be done by changing parts and checking *does the error/loss decrease?* or *are the gradients right?* (with gradient check). Probably consequent test driven development with unit tests even for the layer could help, but was not by myself. Fortunately parts of MLP algorithm themself are simple, such that they easily can be manually compared and verified to the formulas in text books and slides.

I especially had difficulties with the gradient checker. Even after verifying parts to work like `scipy.optimize.check_grad` and a successfully learning network, the check failed. In the end the gradient check still was useful, because it revealed a mistake in the network implementation. We divided the gradient by the number of samples in each layer, which was not a problem for our 1 or 2 hidden-layer networks, but would be for deeper.

3 Application

We use our implementation of a MLP to recognize (*classify*) handwritten digits of the *MNIST* dataset. Setting up training of a MLP which does this classification with a very low error rate in the end involves finding the right meta parameters. We train, verify and test the network on separate datasets. Meta parameters can be of the network (*number of layers or hidden units, choice of activation functions*) or for training itself (*learning rate, batch size, choice of optimization method, ...*). Training and verifying is time consuming, especially with huge datasets, such we used a reduced training set with about 10000 items while searching for meta paramters. Error rate is not exactly the same as with the full training set, but we can see a tendency. Still, training time is about one to three minutes on a *Intel(R) Core(TM) i7-5600U CPU @ 2.60GHz*. Therefore meta parameter search was done manually and very sparse. Some parameter choices are obvious. For gradient descent optimization is very difficult to find a learning rate for which it learns but not overshoots. The stochastic gradient descent (SGD) in contrast is way more robust even if it introduces the batch size as another meta parameter. More difficult is, that several meta parameters affect the error rate in a coupled fashion. It's annoying, if the loss does not decrease at all with the current meta parameters. To at least get a point to start for further parameter tuning I started with meta parameter combinations found in the exercise or internet [1]. Also first searching meta parameters like the learning rate on logarithmic scale worked pretty well.

The setup in Figure 1 is the one I ended up with. Validation error converges after 15 epochs on the full MNIST training training dataset (50000 images) to 1.7%. The test error on the unseen test dataset is 2.0%.

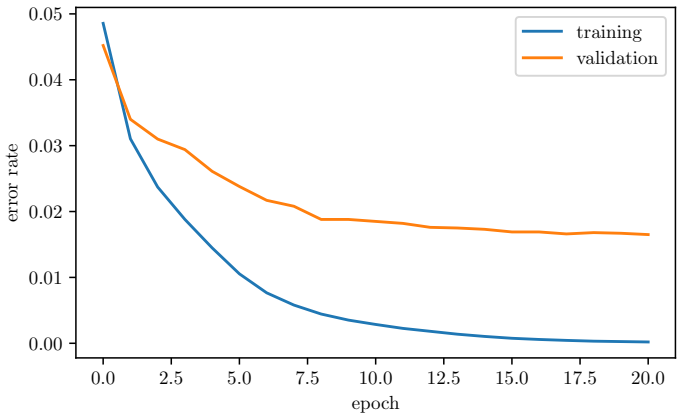
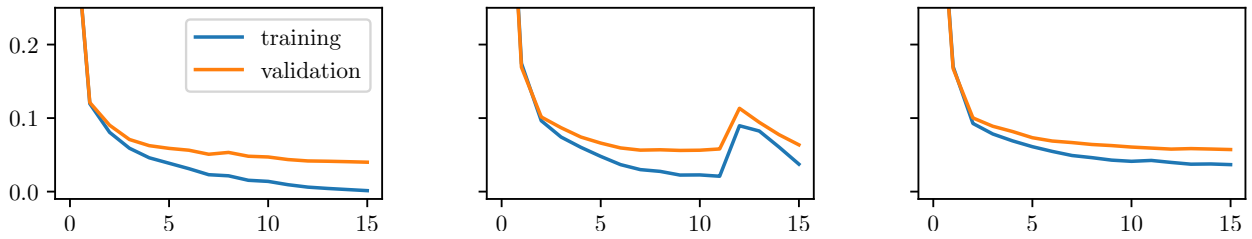


Figure 1: Change of training and validation error while training epochs of MLP on full MNIST dataset. MLP has one hidden layer with 350 units and ReLu activation. Optimization with SGD used a learning rate of 0.4 and batches of 64 images.



(a) none (b) L_1 ($\alpha = 0.0001$) (c) L_2 ($\alpha = 0.005$)

Figure 2: Classification error while SGD training of MLP subset of MNIST without and with regularization. MLP has two hidden layers with each 100 units and ReLu activation.

4 Regularization

With the high amount of parameters (number of weight values), MLPs are capable to express even complex functions. This can lead to the problem of *overfitting* which means the network learns to express exactly the noisy training data instead of the general underlying function. You see this generalization lack in Figure 1 as the gap between training and validation curves. Regularization is an easy method to prevent overfitting by limiting the values of the parameter vectors. Therefore I implemented L_1 and L_2 regularization according to [2].

L_2 regularization adds the squared sum of weights to the loss. This means a large parameter values (huge square weight values) increase the loss a lot and are therefore avoided. L_1 works similar but with the sum of absolute parameter values. The amount of regularization can be controlled with the meta parameter α. In the regularization introduction in lecture we did set $\alpha = \frac{2}{\#\theta}$.

$$L_{L_2}(f_\theta, D) = L(f_\theta, D) + \frac{\alpha}{2}\theta^T\theta, \qquad \theta \leftarrow \theta - \epsilon(\alpha\theta + \nabla_\theta L(f_\theta, D)) \tag{1}$$

$$L_{L_1}(f_\theta, D) = L(f_\theta, D) + \alpha\|\theta\|, \qquad \theta \leftarrow \theta - \epsilon(\alpha\, sign(\theta) + \nabla_\theta L(f_\theta, D)) \tag{2}$$

You see in Figure 2 that - like expected - the validation and training error rates are close. Still also some draws of regularization are visible. Both the train and the validation errors are higher with regularization. Possibly this can be explained, by seeing the regularization update step also as forgetting learned parameters. In Figure 2c epochs 7 and 10 we see another incident which is probably also caused by the forgetting. The error rates jump up before descreasing again.

References

[1] `http://deeplearning.net/tutorial/mlp.html`, last visit 2017/11/06

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning, sec. 7.1*, (MIT Press, 2016)