Project 1: Applying Convolutional Neural Networks on MNIST

Marcus Brenscheidt
Heinrich Heine University
marcus.brenscheidt@uni-duesseldorf.de

Bastian Berndt
Heinrich Heine University
bastian.berndt@uni-duesseldorf.de

Tobias Uelwer
Heinrich Heine University
tobias.uelwer@uni-duesseldorf.de

Abstract

In this short paper we explore how to use convolutional neural networks for the MNIST challenge. Furthermore, we explore how the design of our network, as well as the choice of hyperparameters affects the performance of our CNN-approach. We will conclude with the proposal of a minimal CNN, which represents an optimal tradeoff between complexity and prediction accuracy.

1 Introduction to MNIST and Convolutional Neural Networks

1.1 Data

In this paper we try to classify handwritten digits with convolutional neural networks. Our dataset, which is known as MNIST, is commonly used in Machine Learning tutorials and competitions. MNIST is an acronym for Mixed National Institute of Standards and Technology database. This database is basically a collection of 28x28 greyscale pictures.

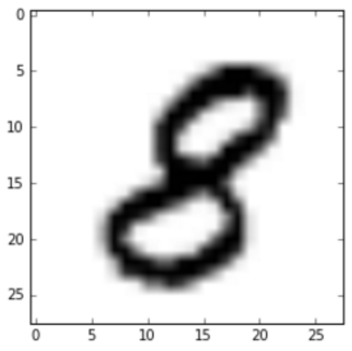

Figure 1: Handwritten eight from MNIST

1.2 Why Convolutional Networks?

Conventional fully connected neural networks often fail to take local structure in the data into account. For a lot of machine learning tasks we don't have spatial relationship between the dimensions in our data points. However, this clearly does not hold true for image data, where the position of a pixel is almost as important as the value it holds.

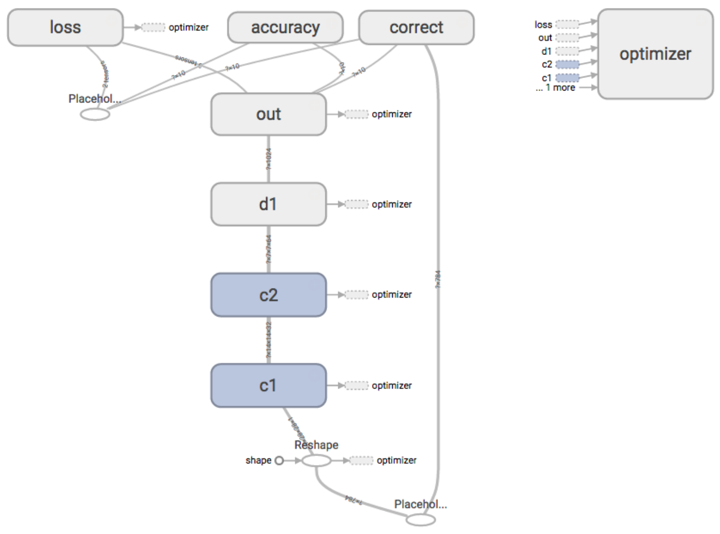

Figure 2: Abstract representation of our computation graph, c: convolutional, d: fully connected

Convolutional neural networks take spatial structure in our data into acount by learning and applying filters used for convolution in our neural networks. Often times, we use convolutional filters as intermediary layers in our neural network architecture and pass their output into conventional fully connected layers.

2 Implementing a simple CNN for the MNIST challenge

2.1 Basic Architecture

In this section we will start with a prototypical CNN architecture, consisting of two convolutional layers, a fully connected layer and an output layer. The output of every layer (except from the output layer) is activated by a rectified linear unit. Furthermore, the output of each convolutional layer is max-pooled with window-size 2 and step size 2. For a visualization of the architecture, please refer to figure 2.

Convolutional Layers

Our first convolutional layer consists of 32 5x5 kernels. Our second convolutional layer translates these 32 channels into 64 channels with 5x5 kernels. Each output is passed into a rectifier and fed into a maxpooling layer. As a step size for convolution, we chose single steps.

Fully Connected Layer and Output Layer

Finally we apply a fully connected layer that connects the pooled output from our convolutional layers with our output layer. The fully connected layer has a size of 7*7*64x1024. The output layer translates these 1024 inputs to the 10 classes (numbers). It has a shape of 1024x10.

2.2 Training Process

For training our CNNs, we developed our own framework that allows rapid creation of different CNN architectures. Another central goal for our framework was logging and discoverability, so that we can easily compare and analyze the many different architectures that we want to test. Therefore we implemented Tensorboard support, so that for any high level specification of a CNN, we can see the correct Graph in Tensorboard and have properly labeled, exhaustive statistics, such as distributions, scalar summaries, and image summaries of misclassified inputs as well as convolved inputs.

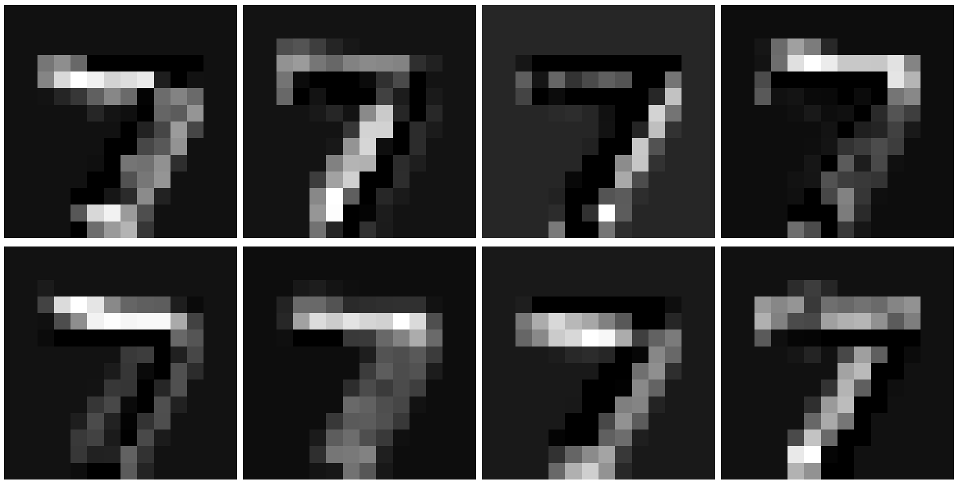

Figure 3: first 8 convolutions of a handwritten seven

For each CNN we construct a proper prediction graph with proper logging capabilities. This graph is then passed into a loss node, which, in turn, is passed into an optimization node. For optimization we used the Adam Optimizer.

The Adam Optimizer has empirically shown to be a rather good and stable optimizer.

We also monitor the convolutions themselves to analyze to which features our convolutional layers react. A visualization of the ouput from our convolutional layers can be found in 3.

2.3 Evaluating Test Performance

2.3.1 Quantitative Performance Analysis

In general, all of our tested architectures achieved good results, hovering between 98 - 99 % accuracy on the test-set.

In rare cases, we even achieved accuracies higher than 99 % with our standard configuration.

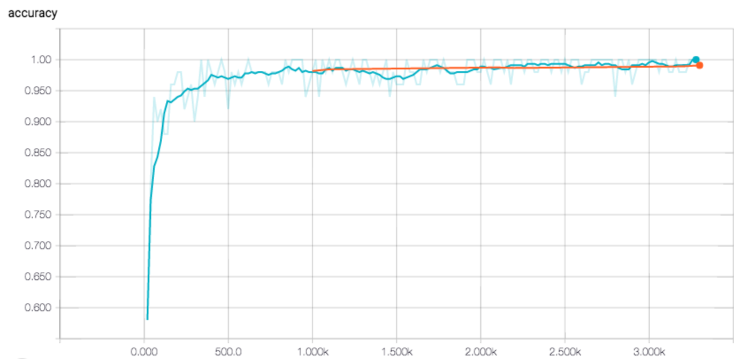

Figure 4: Blue: Training Perfomance, Orange: Test Performance

For the performance measurement we logged accuracy on our training set every 20th iteration step, whereas we measured the performance on the whole test set only every 1000th iteration. Overall we performed our tests over the range of 3 epochs.

However, we usually see the trend in our performance evaluations, that most neural network architectures with sufficient complexity perform equivalently good, with 98.XX % accuracy after a few epochs.

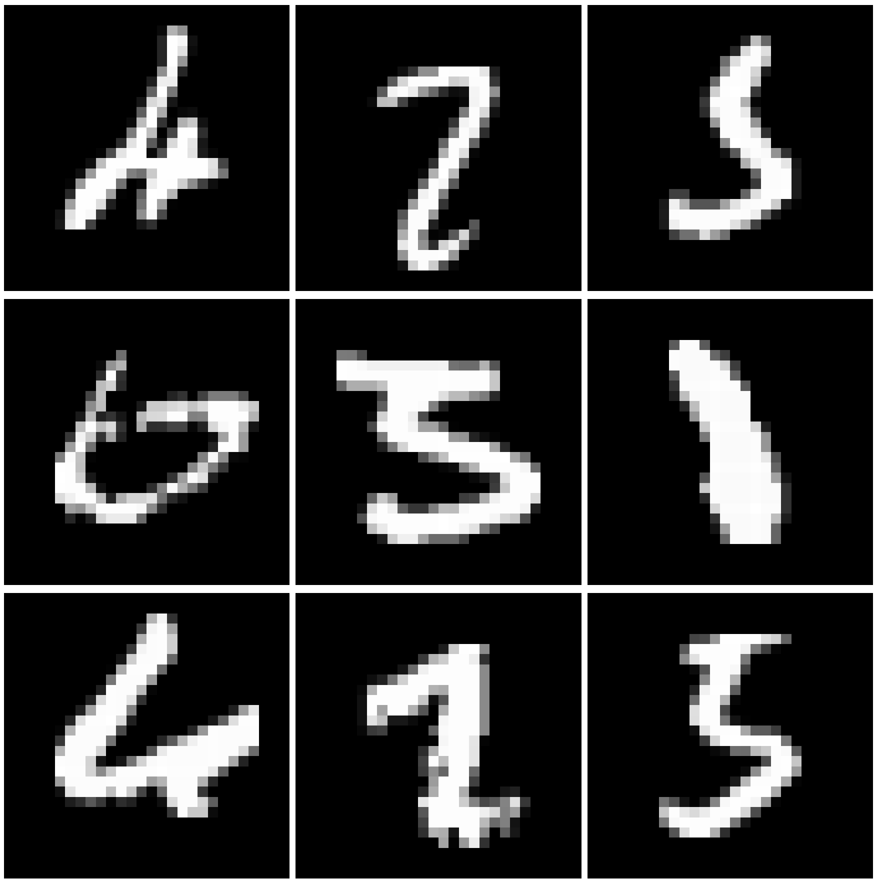

Figure 5: 9 Misclassified Images from our Test set

2.3.2 Qualitative Performance Analysis

Aside from a quantitative analysis, which only looks at the raw accuracy achieved by a neural network, we also try get a *qualitative* understanding of how our networks performs. This can be done by investigating which subsets of the test data gets misclassified. A subset of the misclassified inputs is represented in figure 5.

On closer inspection, we can see that our misclassified samples are rather irregular representations of their respective numbers and might even be ambiguous for the human perception (5 vs. 3, last image).

3 Effect of Architecture and Hyperparameters on performance

3.1 Preface: On Randomness and Determination in Neural Network training

Training Neural Networks is usually not a deterministic process. While the optimization algorithms themselves are often deterministic, the initialization of the layer variables usually follows some random distributions.

3.2 Effect of Iteration Count

Optimization of neural networks is usually done by some form of gradient descent. Gradient descent follows a decline in a loss function and thus seeks some form of localized minimum. This means that during the optimization process our network configuration converges against a local minimum.

Thus, increasing iteration count has a beneficial effect on accuracy, as long as we have not yet converged against our optimal configuration. However, one can observe diminishing returns once the optimization process has reached a loss minimum. Increasing iteration count does not allow us to improve the accuracy of our predictions beyond a certain threshold, which is determined by our network architecture and a little bit of chance (since the local maximum we find is dependent on our starting point).

3.3 Effect of Batch-Size during Training Process

We ran an empirical investigation into the optimal setting of the batch size. We tested batch sizes between 10 and 5000 and averaged each over five trials. Each trial consisted of training one initial epoch where we measured final accuracy and computation times.

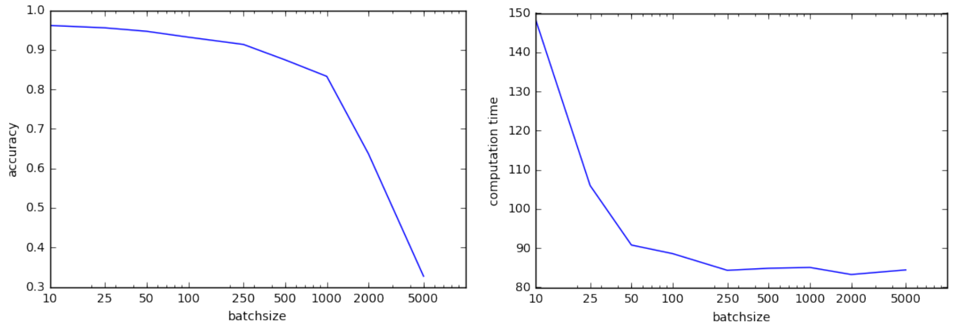

Figure 6: Batchsize plotted against Accuracy and Computation time for one epoch.

For a plot of our results, please refer to 6. We can see that the accuracy degrades slowly with a growing batch size, while the computation time reduces rather fast with bigger batches.

From these experiments we concluded an optimal batch-size of 50 for our training process.

3.4 Effect of Learning Rate on the Optimization Process

Since we used the Adam Optimizer, the learning rate does not have quite the same impact on our performance compared to more conservative stochastic gradient descent algorithms.

The Adam Optimizer adjusts the step size dynamically based on adaptive estimations of lower order moments of our cost-function. Thus setting an initial step-size does only have a minor influence on the actual steps being made in the optimization process. More information can be found in [2].

3.5 Effect of Pooling

Pooling is both a tool for reducing the dimensionality of our feature set, but can also be seen as rectifier unit in the sense that it introduces nonlinearity into the functions computable by a neural network [1].

In our tests, early pooling has shown to reduce computation times by quite a margin. For pooling, we relied exclusively on Max-Pooling, which was applied to part of our convolutional layers. Stride and filter size, were both set to 2, which translates to only picking the maximum signal from mutually exclusive 2x2 patches.

If we leave out the pooling layers completely, we can observe, that the time required for the optimization process increases noticeably. Furthermore, we have the additional downside that our network tends to overfit.

3.6 Effect of general network shape on Learning

Even with a small number of hidden layers and convolutional layer, we get rather close to a practical threshold of 99%. We presume that the issue with achieving a higher accuracy lies not within a lack of theoretical complexity representable through our neural networks, but rather in the irregularities in the data itself and a small number of outliers.

When adding more convolutional layers or fully connected layers, we hardly see a noticeable improvement in accuracy. However, deeper networks and more variables require more computations for the optimization process. Adding more and/or wider layers mainly increases computation time per epoch while decreasing rate of convergence, however leading to a similar absolute outcome in terms of accuracy.

4 Designing a minimal efficient CNN

We measured the size of a network through the number of variables that are optimized as a key performance indicator.

Other possible metrics could also consider layer depth and diversity of layer types, as well as the usage of activation functions and pooling layers.

To minimize the number of parameters, we relied primarily on small convolutional filters, which were then passed into a maxpooling layer in order to reduce the size of our final fully connected layer.

Our chosen minimal architecture consists of five layers in total. We start with three convolutional layers, each followed by a maxpooling layer. The result is then passed into one fully connected layer, which is, again, passed into the final output layer.

type	**weights**	**bias**	**activation**	**# parameter**
convolutional	3x3x1x4	4	Rectifier, Maxpool2d	40
convolutional	3x3x4x8	8	Rectifier, Maxpool2d	296
convolutional	3x3x8x4	4	Rectifier, Maxpool2d	292
fully connected	64x8	8	Rectifier	520
out	8x10	10	Rectifier	90
total				**1238**

From the table we can see that our minimal design only uses 1238 Variables. With this size we can achieve an accuracy greater than 95%.

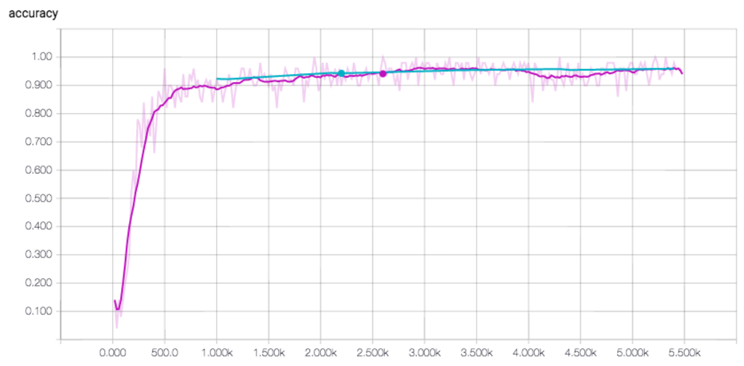

Figure 7: Minimal Network, Blue: Training Perfomance, Orange: Test Performance

After five epochs of training, we finally reached an accuracy of 96.01% on our test set.

This experiment has shown that convolutional neural networks are able to achieve quite good performance, even with a very limited amount of hidden variables and thus, limited complexity in the structure of our predictions.

References

[1] Montufar, G. F., Pascanu, R., Cho, K., & Bengio, Y. (2014). On the number of linear regions of deep neural networks. In Advances in neural information processing systems (pp. 2924-2932).

[2] Kingma, D., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.