UNIVERSITY OF FREIBURG

Exercise 2: Convolutional Neural Networks

Hendrik Vloet

November 19, 2017

1 Theory

1.1 Convolutional Neural Networks

This section is meant to provide a quick recap of Convolutional Neural Networks (CNNs), so I tried to keep it short.
A convolutional neural network is a feed-forward neural network that does not rely solely on fully connected layers. Typically a CNN consists of several layers that are designed in a way to detect more and more advanced features of the input data, the deeper the network propagates forth. This principle setup is shown in figure 1.1

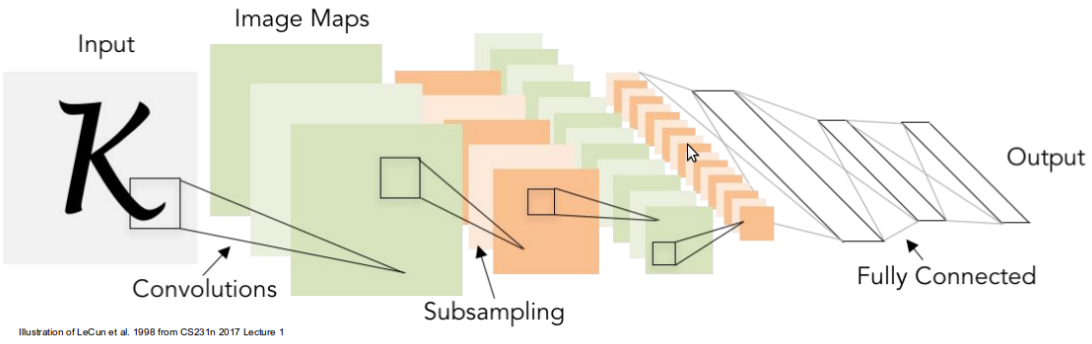


Figure 1.1: Example CNN, taken from LeCun et al. 1998 from CS231n 2017 Lecture 1

The basic workflow works like this:

1. Image is fed into the Network
2. The image is convolved with several layers of various dimensions, e.g. 16 filters with dimensions 3x3
3. depending on the convolution operation (same/valid), the image dimensions are changed, let's assume we use same convolution, i.e. the dimensions of the input image are not changed when going through a convolution layer.
4. after convolution, there can be some sort of sub sampling in order to make the data more manageable. One example could be a 2x2 max pooling layer, which reduces the dimensions of its input data by half. So for example, if we fed an 28x28 image into a max-pool-2x2 layer, we get as output a lower resolved 14x14 image.
5. By using more and more convolution and pooling layers, the network extracts more and more complex features from the original data, e.g. first it detects some contours, than edges and in the end it can classify an object with the help of these "high-level" features.
6. The last stage is one fully connected layer, like in classic neural networks. It uses an activation and a loss function in order to get correct classification results. Note that the input of the fully connected layer is now longer an multi-dimensional image, but a flattened array according to the designed network.
7. Finally the output of this network is a classification.

1.1.1 Hyperparameters

There are several hyperparameters for CNNs one can tweak, here is a short overview over the most common (there are many more):

- Number of Layers
- Choice of Pooling
- Choice of Loss Function
- Filter dimensions
- Zero Padding
- same/valid convolution
- Stride
- convolution method:
  - Convolution: $S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$
  - Cross-Correlation: $S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)$
    (flipping the kernel first and then applying cross-correlation is equivalent to convolution)

2 Task 1: Setup

The general Setup as it is implmented is shown in figure CITE. It shows the most basic configuration, i.e. the parameters of Task 1. Dimensions are chosen accordingly and are of course modified later on for different filter depths.
I used a SGD optimising approach with a fixed batch size of 50.
I chose 20 epochs to train and test.

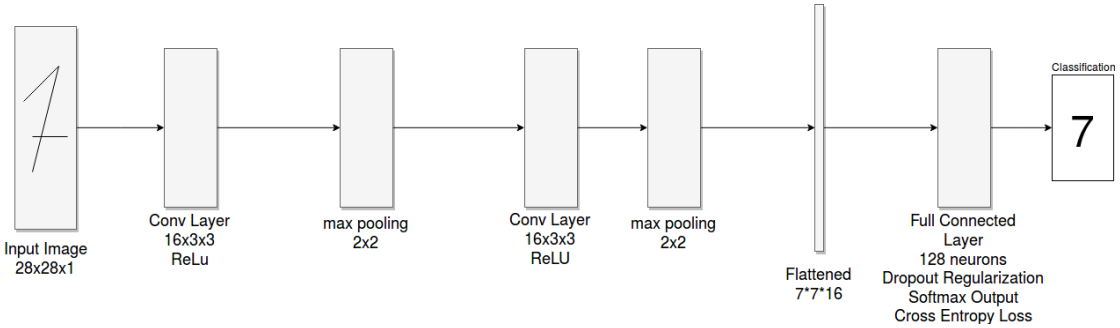


Figure 2.1: Principal structure of the used CNN

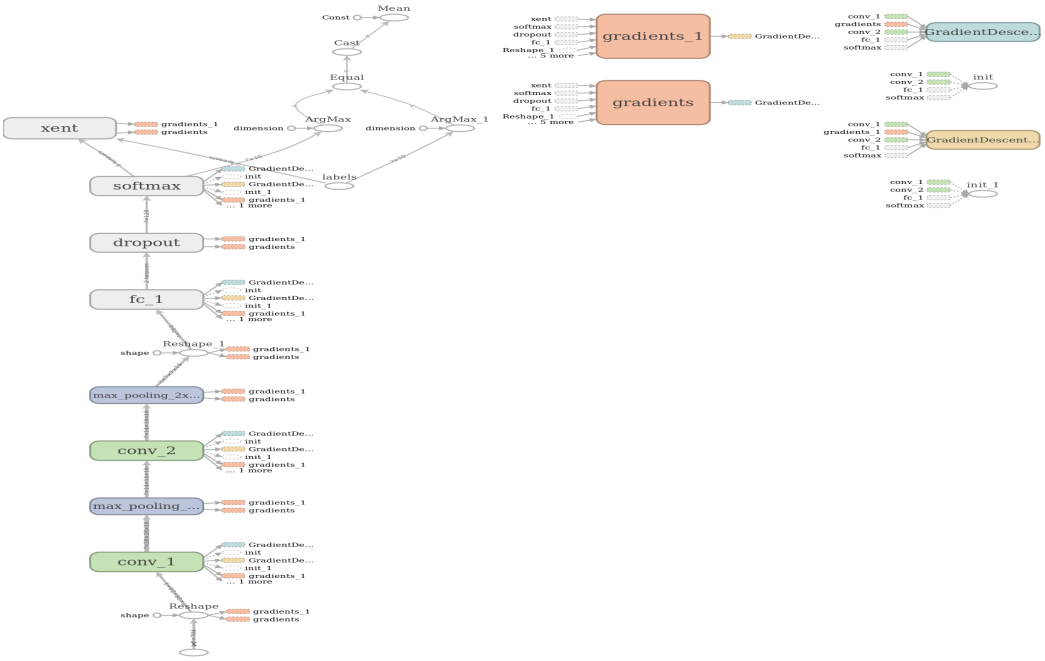
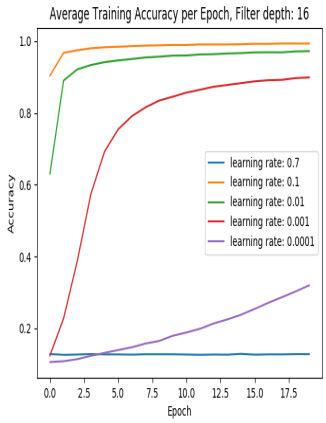


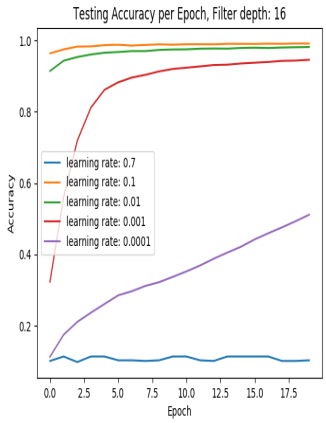
Figure 2.2: Tensorboard graph of used architecture

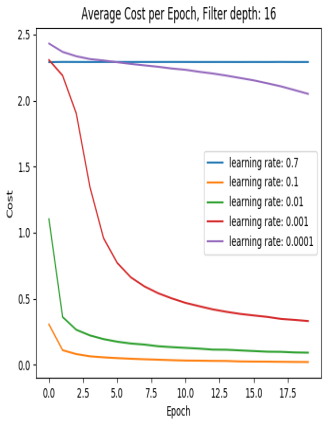
3 Results

3.1 Task 2: Comparison of Learning Rates

I compared several learning rates with a constant filter size of 16. I added a learning rate of 0.7 to the learning rates under consideration in order to emphasize the divergent behaviour of the optimiser (SGD) if the learning rate is chosen too high.
It can be nicely seen that for very small learning rates, e.g. 0.0001 the learning does not converge since the steps the optimiser makes during the gradient descent optimisation are too small. Eventually this small learning rate will lead to convergence, but it would take far more epochs (»20). On the other hand we can see for a rather high learning rate, e.g. 0.7 the optimisation does not converge, too. On the contrary, it diverges: the steps the optimiser makes are now so big that it comes to an "overshooting" of the minimum. This results in almost constant training/testing accuracy and also the costs will not decrease. Now, convergence is not guaranteed anymore.
A rather good result is obtained with the learning 0.1. It is not to small and not to big and the optimiser manages it find a minimum within the 20 epochs.
Overall one can say a CNN can be easy to implement and lead to very good result, but the designer has to have some intuition about good hyperparameters or if he lacks that knowledge, knows how to do a hyperparameter optimisation with for example random search, grid search, hill climbing, etc...







3.2 Task 3: Runtime Comparison

The runtime comparison shows that the runtime increases exponentially for the increasing number of filters, since the parameters also increases. Overall the GPU runtimes are much lower than the CPU runtimes. This is because the GPU memory can be used more efficiently than the CPU

The filter sizes of 128 and 256 I could not inspect, since the workstation I used terminated due to out-of-memory errors. I attached the error message. The number of parameters missing I calculated by hand, but I could not verify them due to the OOM error(Please take a look at the according file MEMORY_ERROR.txt in the same directory as the report).

Parameters for 128 filters:

$$[(3 \cdot 3 \cdot 1 + 1) \cdot 128] + [(3 \cdot 3 \cdot 128 + 1) \cdot 128] + [7 \cdot 7 \cdot 128 \cdot 128 + 128] + [128 \cdot 10 + 10] = 953098$$

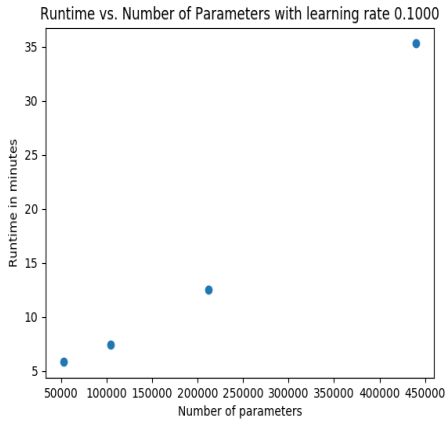
Parameters for 256 filters:

$$[(3 \cdot 3 \cdot 1 + 1) \cdot 256] + [(3 \cdot 3 \cdot 256 + 1) \cdot 256] + [7 \cdot 7 \cdot 256 \cdot 128 + 128] + [128 \cdot 10 + 10] = 2198410$$

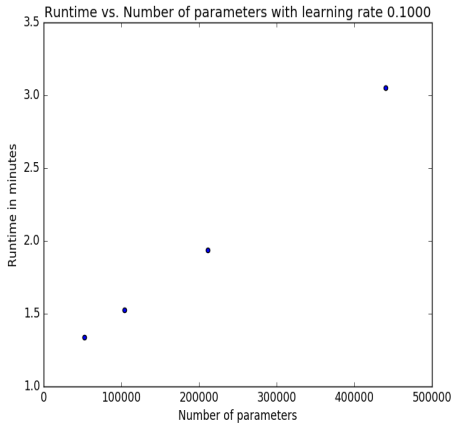
Nevertheless, I can still conclude what was stated above: the usage of GPU is way more efficient than the use of the CPU.

CPU				
Filter depth	8	16	32	64
Parameters	52258	104250	211690	440394
Runtime	5.8	7.4	12.5	35.3

GPU						
Filter depth	8	16	32	64	128	256
Parameters Parameters	52258	104250	211690	440394	953098	2198410
Runtime	1.3411	1.5256	1.9356	3.0511	OOM	OOM



CPU Runtime vs. no. of parameters



GPU Runtime vs. no. of parameters