Prof. Dr. Carsten Meyer
Faculty of Engineering
Christian-Albrechts-Universität Kiel

Neural Networks and Deep Learning – Summer Term 2018

Exercise sheet 5

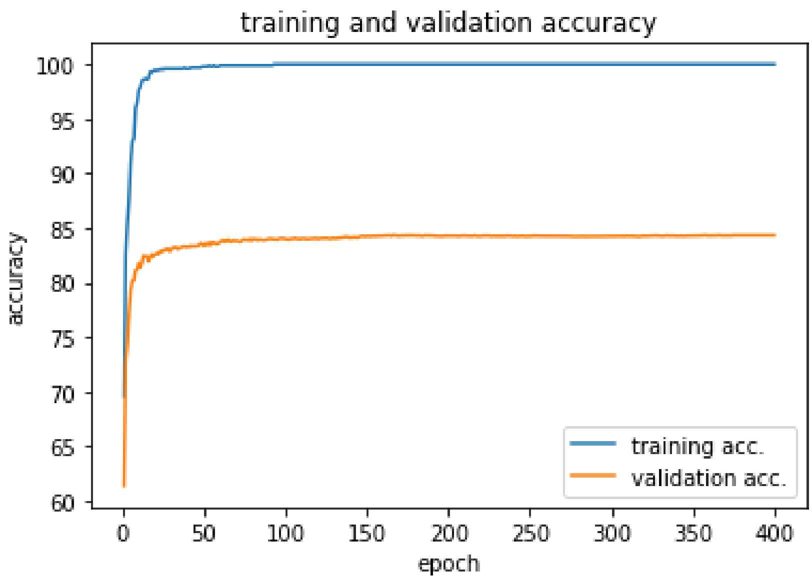
Submission due: Tuesday, June 19, 11:30 sharp

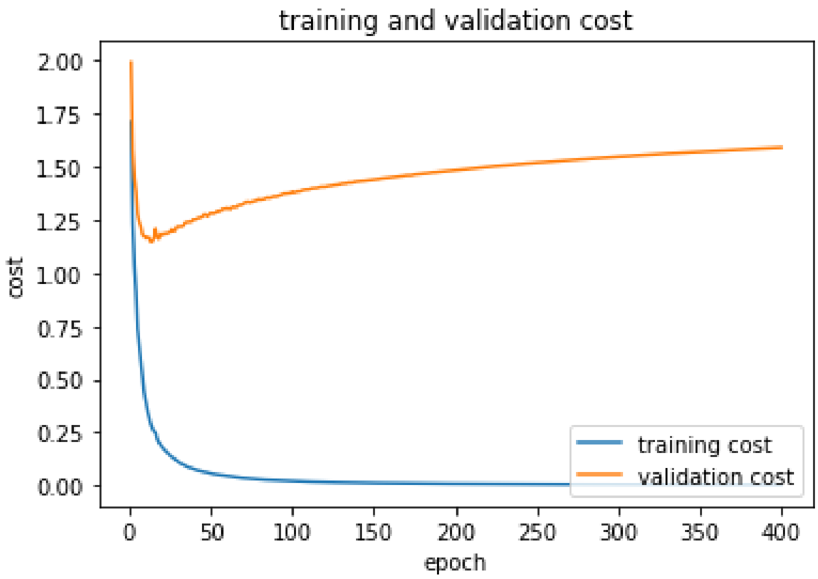
Exercise 1 (Overfitting):

The script **`exercise1.py`** (adapted from Michael Nielsen and Michal Daniel Dobrzanski) configures a simple feedforward neural network with a single hidden layer, the parameters of which are chosen to demonstrate the effect of overfitting on the MNIST digit recognition task. Run the script and discuss the results. Does the overfitting effect depend on the learning rate and on the mini-batch size?
Explore two possibilities to reduce the effect of overfitting (you may refer to the previous exercise for options and code...).

Solution:

Using the validation data as evaluation data, the following results are obtained (I obtained similar results when using the test data as evaluation data):





The training accuracy approaches 100% and the training cost (cross-entropy cost, λ = 0, i.e. no regularization) approaches nearly 0. However, the validation accuracy is only about 83.92%, and the validation cost decreases and then increases again to a value larger than 1.5. This clearly indicates overfitting, i.e. the model learns the training data nearly perfectly, but it does not generalize well to independent validation data.

Note that the first 1000 training images do contain examples for all 10 digits.

Dependence on the learning rate and the mini-batch size:

Modifying the learning rate η and the mini-batch size b, the following results are obtained (number of epochs: 400; we port results for the last epoch; an asterisk * denotes that the training has not converged for 400 epochs):

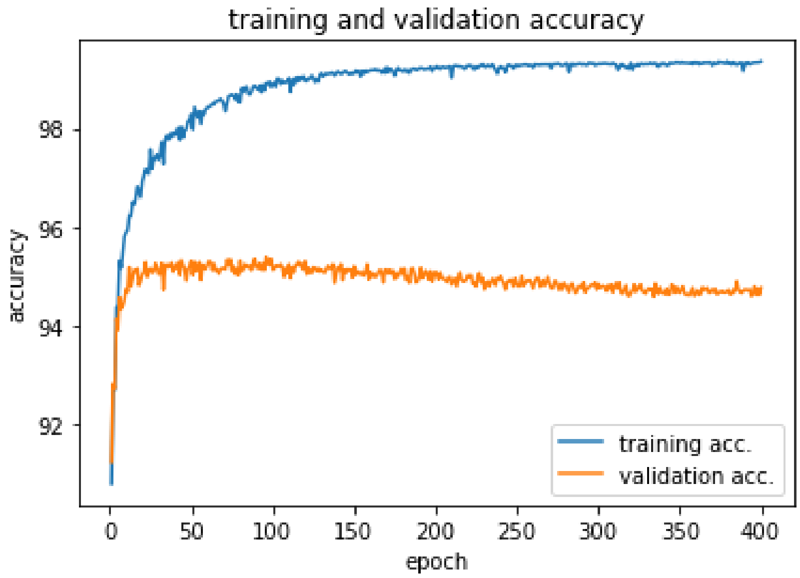
η	b	train. cost	valid. cost	train. acc.	val. acc.	test acc.
0.005	10	0.745467 *	1.319611 *	91.70 *	77.09 *	75.63 *
0.01	10	0.490582 *	1.250634 *	95.30 *	79.40 *	78.32 *
0.05	10	0.069189	1.2597	99.70	82.92	81.40
0.1	10	0.033056	1.337583	100.00	82.93	81.23
0.5	10	0.003733	1.601105	100.00	83.90	82.11
0.05	5	0.036406	1.291454	99.90	83.92	82.28
0.05	10	0.057888	1.345600	100.00	82.15	81.13
0.05	16	0.170670	1.318352	99.10	80.14	78.84
0.05	32	0.287984*	1.196077*	97.80*	80.90*	79.32*

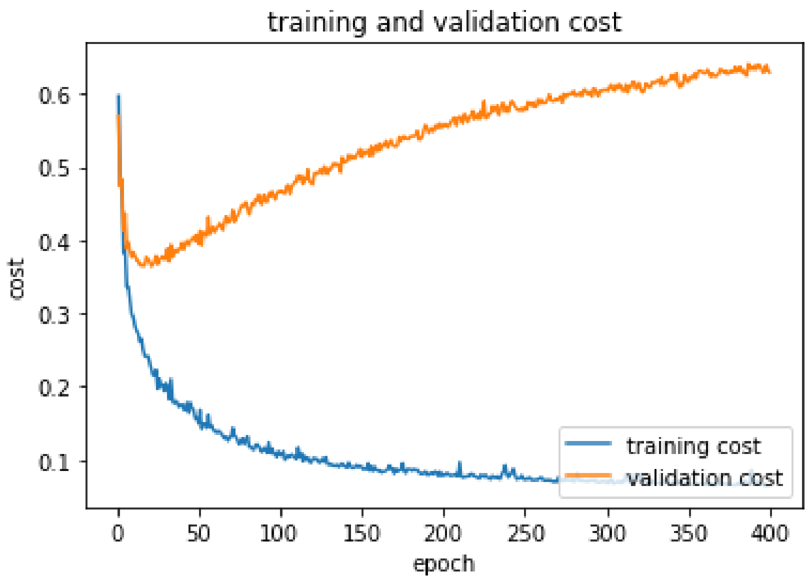
Since the parameter configurations with an asterisk do not seem to have converged (e.g. due to a small learning rate or a large batch size), the results do not support the hypothesis that the overfitting effect depends on the learning rate or the batch size (to really check this hypothesis, the experiments with an asterisk would have to be performed for a much larger number of epochs, and all results would have to be repeated to account for random effects).

Attempts to reduce overfitting:

a) Use more training data

Using all 50000 training images instead of only 10000 training images, the following results are obtained:

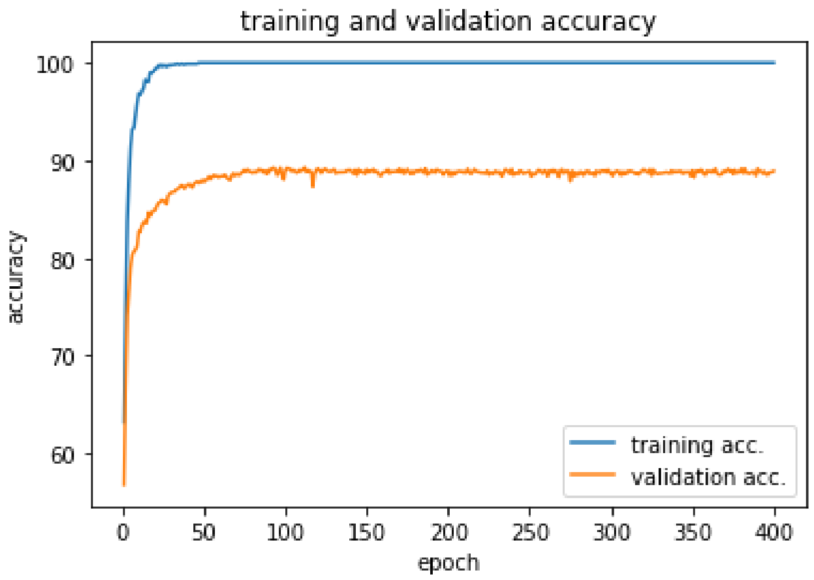


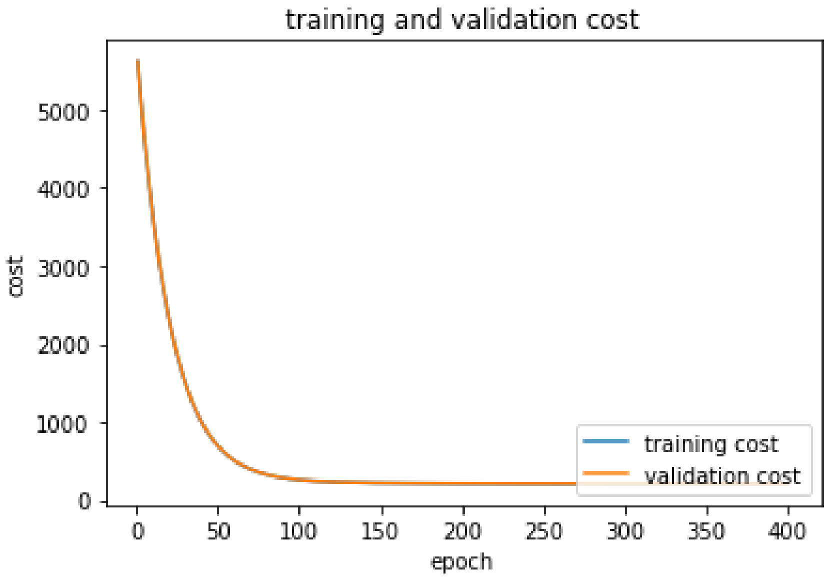


Although the qualitative shape of the training and validation accuracy and cost qualitatively look similar as before, the quantitative values are quite different: The training accuracy is lower than before (on the order of 99.4%), whereas the validation accuracy is much larger (on the order of 94.6%, so the gap between training and validation accuracy is below 5 percent points, compared to about 17 percent points as before). Also, the validation cost is on the order of 0.6 instead of 1.5 as before. So using more training data clearly helps to prevent overfitting in this example.

b) Regularization

Using L_2 regularization with regularization parameter $\lambda = 0.5$ leads to the following results:



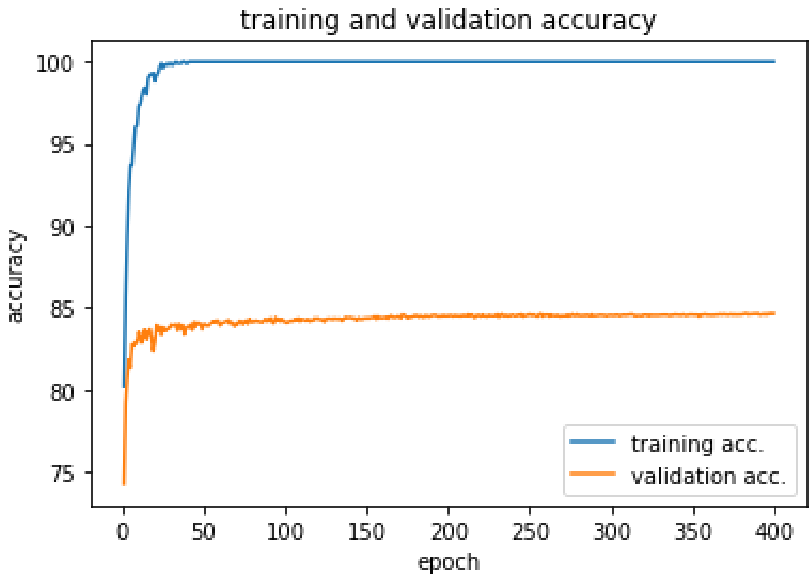


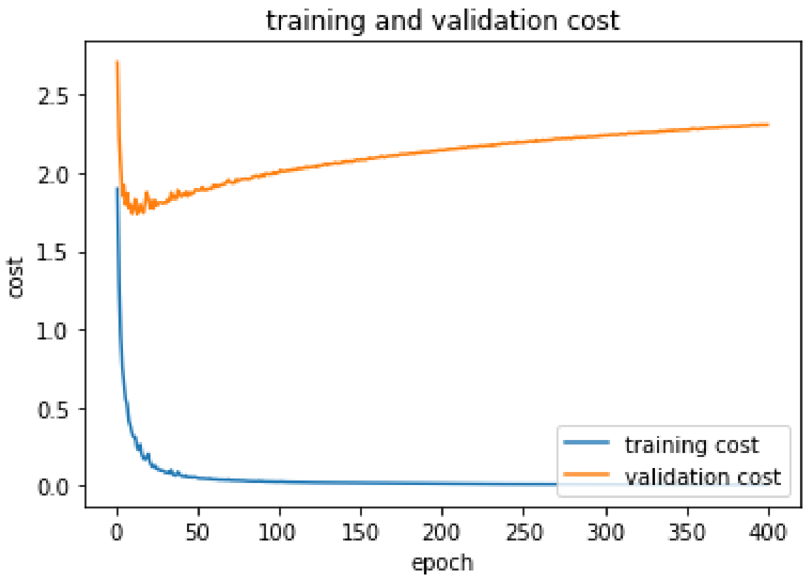
The validation accuracy is on the order of 88.9% (training accuracy 100.0%), so the gap between training and validation has become smaller (11 percent points instead of 17 percent points). While reducing the impact of overfitting, the effect of regularization is, however, not as big as increasing the amount of training data in this example. On the other hand, the parameter λ can further be optimized.

Due to adding costs for the norm of the weights, the costs are now larger: Around epoch 400, the training costs are about 210.17 and the validation costs about 210.8.

c) Reduce the number of parameters

The only possibility to reduce parameters would be to reduce the number of layers and / or the number of hidden neurons. Since there is only one hidden layer with only 30 hidden neurons, this strategy cannot be followed here. The only test is to use a network without hidden layers at all, yielding the following results:





The validation accuracy is about 83.35 and the validation cost about 2.31, so in this example the effect of overfitting is even larger... (generally, reducing the number of parameters may however help in preventing overfitting).

Exercise 2 (Loss functions and weight update formulae, theoretical considerations):

a) Show that for a single-layer perceptron with a linear activation function, the weights can be determined in closed-form from the training data, assuming a mean-squared error loss.

Solution:

This is the same derivation as for least mean squares linear regression:
Training data: $D = \{(\mathbf{x}^{(1)}, y^{(1)}),\ \ldots, (\mathbf{x}^{(n)}, y^{(n)})\}$, $\mathbf{x}^{(\mu)} \in \mathbb{R}^d$, $y^{(\mu)} \in \mathbb{R}$

Write training data (in augmented notation) in matrix form:

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}^{(1)T} \\ \vdots \\ \mathbf{x}^{nT} \end{pmatrix} = \begin{pmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \dots & x_d^{(1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(n)} & x_2^{(n)} & \dots & x_d^{(n)} \end{pmatrix}; \quad \mathbf{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}$$

Perceptron with linear activation function in augmented notation: $\hat{y}(\mathbf{x}) = \mathbf{w}^T \cdot \mathbf{x} = \mathbf{x}^T \cdot \mathbf{w}$
Mean-squared error loss between perceptron output $\hat{y}(\mathbf{x})$ and target $y^{(\mu)}$:

$$\mathrm{L}_{MSE}(\mathbf{w}, y, \hat{y}) = \frac{1}{2p} \sum_{\mu=1}^{p} \mathrm{L}_{MSE}^{(\mu)}(\mathbf{w}, y, \hat{y}) = \frac{1}{2p} \sum_{\mu=1}^{p} \left(\hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)}) - y^{(\mu)}\right)^2 = \frac{1}{2p} (\mathbf{y} - \mathbf{X} \cdot \mathbf{w})^T \cdot (\mathbf{y} - \mathbf{X} \cdot \mathbf{w})$$

Optimal perceptron weights (minimizing the mean squared error loss) are given by:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \mathrm{L}(\mathbf{w}, y, \hat{y}) = \arg\min_{\mathbf{w}} \frac{1}{2p} \sum_{\mu=1}^{p} \left(\hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)}) - y^{(\mu)}\right)^2 = \arg\min_{\mathbf{w}} (\mathbf{y} - \mathbf{X} \cdot \mathbf{w})^T \cdot (\mathbf{y} - \mathbf{X} \cdot \mathbf{w})$$

Minimizing the loss function:

$$\frac{\partial L_{MSE}(\mathbf{w}, y, \hat{y})}{\partial \mathbf{w}^*} = -2(\mathbf{y} - \mathbf{X} \cdot \mathbf{w}^*)^T \mathbf{X} = \mathbf{0} \Leftrightarrow \mathbf{y}^T \mathbf{X} = (\mathbf{X} \cdot \mathbf{w}^*)^T \mathbf{X}$$

$$\Leftrightarrow \mathbf{X}^T \mathbf{y} = \mathbf{X}^T \mathbf{X} \mathbf{w}^* \Leftrightarrow \mathbf{w}^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

assuming that the inverse $(\mathbf{X}^T \mathbf{X})^{-1}$ exists.

This is the desired closed-form solution for the synaptic weights as determined from the (supervised) training data. A closed-form solution exists since due to the linear activation function and the (differentiable) quadratic loss function, the loss function depends quadratically on the synaptic weights; therefore the derivative of the loss function (which should equate to 0) depends linearly on the weights and yields a linear equation for the synaptic weights.

b) The cross-entropy loss for a single-layer perceptron on a training set $D = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \ldots, (\mathbf{x}^{(p)}, y^{(p)})\}$ is defined as

$$\mathrm{L}_{CE}(\boldsymbol{w}, y, \hat{y}) = -\frac{1}{p}\sum_{\mu=1}^{p}\left[y^{(\mu)} \ln \hat{y}(\boldsymbol{w}, \boldsymbol{x}^{(\mu)}) + \left(1 - y^{(\mu)}\right) \ln\left(1 - \hat{y}\left(\boldsymbol{w}, \boldsymbol{x}^{(\mu)}\right)\right)\right],$$

where $\hat{y} = \hat{y}\left(\boldsymbol{w}, \boldsymbol{x}^{(\mu)}\right)$ is the output of the perceptron. Show that this expression is non-negative for $y, \hat{y} \in [0,1]$ and assumes a minimum (as a function of the perceptron output $\hat{y}$) if $\hat{y}\left(\boldsymbol{w}, \boldsymbol{x}^{(\mu)}\right) = y^{(\mu)}$ for all μ, i.e. if the neuron output $\hat{y}\left(\boldsymbol{w}, \boldsymbol{x}^{(\mu)}\right)$ corresponds to the target value $y^{(\mu)}$ for each training sample μ.

Solution:

Define the per-sample cross-entropy loss as $L_{CE}^{\mu} = -y^{(\mu)} \ln \hat{y} - \left(1 - y^{(\mu)}\right) \ln(1 - \hat{y})$.

For $x \in]0,1]$ we have $\ln(x) \leq 0$, and since $\lim_{x \to \infty} x \ln(x) = 0$, it follows that $y^{(\mu)} \ln \hat{y} \leq 0$ for all $y, \hat{y} \in [0,1]$. Similarly, by symmetry, we have $\left(1 - y^{(\mu)}\right) \ln(1 - \hat{y}) \leq 0$ for all $y, \hat{y} \in [0,1]$.

Thus the per-sample cross-entropy loss is non-negative:
$L_{CE}^{\mu} = -y^{(\mu)} \ln \hat{y} - \left(1 - y^{(\mu)}\right) \ln(1 - \hat{y}) \geq 0$ for all $y, \hat{y} \in [0,1]$, and therefore also the cross-entropy loss averaged over the training set is non-negative:

$\mathrm{L}_{CE}(\boldsymbol{w}, y, \hat{y}) = -\frac{1}{p}\sum_{\mu=1}^{p}\left[y^{(\mu)} \ln \hat{y}(\boldsymbol{w}, \boldsymbol{x}^{(\mu)}) + \left(1 - y^{(\mu)}\right) \ln\left(1 - \hat{y}\left(\boldsymbol{w}, \boldsymbol{x}^{(\mu)}\right)\right)\right] \geq 0$ for all $y, \hat{y} \in [0,1]$.

Note that for a sigmoid activation function $f(z)$, we have $0 < \hat{y} = f(z) < 1$, such that $\ln \hat{y}$ and $\ln(1 - \hat{y})$ are both finite. This motivates the definition of the cross-entropy loss in such a way that the perceptron output appears in the argument of the logarithm, whereas the true output (which may be 0 or 1) appears as a pre-factor.

To calculate the minimum of the cross-entropy loss as function of the perceptron output $\hat{y}$, first we take the partial derivative of the per-sample cross-entropy loss with respect to $\hat{y}$ and equate to 0:

$$\frac{\delta L_{CE}^{\mu}(\boldsymbol{w}, y, \hat{y})}{\delta \hat{y}} = -\frac{y^{(\mu)}}{\hat{y}} + \frac{1 - y^{(\mu)}}{1 - \hat{y}} = 0 \Leftrightarrow \frac{y^{(\mu)}}{\hat{y}} = \frac{1 - y^{(\mu)}}{1 - \hat{y}} \Leftrightarrow y^{(\mu)}(1 - \hat{y}) = \hat{y}\left(1 - y^{(\mu)}\right)$$

$$\Leftrightarrow y^{(\mu)} - y^{(\mu)}\hat{y} = \hat{y} - \hat{y}y^{(\mu)} \Leftrightarrow y^{(\mu)} = \hat{y} \quad .$$

Therefore, the per-sample cross-entropy loss is minimized if the perceptron output $\hat{y}$ corresponds to the target output $y^{(\mu)}$. The value of the per-sample cross-entropy loss at the minimum is $L_{CE}^{\mu} = -y^{(\mu)} \ln y^{(\mu)} - \left(1 - y^{(\mu)}\right) \ln\left(1 - y^{(\mu)}\right)$.

Taking the second derivative yields: $\frac{\delta^2 L_{CE}^{\mu}(w,y,\hat{y})}{\delta \hat{y}^2} = \frac{y^{(\mu)}}{\hat{y}^2} + \frac{1-y^{(\mu)}}{(1-\hat{y})^2} > 0 \;\; \forall \;\hat{y}$ such that there is a unique global minimum as a function of $\hat{y}$.

Regarding the full training-set cross-entropy loss $\mathrm{L}_{CE}(w, y, \hat{y}) = \frac{1}{p}\sum_{\mu=1}^{p} L_{CE}^{\mu}$, the derivative with respect to $\hat{y}$ is $\frac{\delta \mathrm{L}_{CE}(w,y,\hat{y})}{\delta \hat{y}} = \frac{1}{p}\sum_{\mu=1}^{p} \frac{\delta L_{CE}^{\mu}}{\delta \hat{y}}$. Inserting the derivative of the per-sample cross-entropy loss and equating to 0 leads to

$$-\frac{1}{p}\sum_{\mu=1}^{p}\left[\frac{y^{(\mu)}}{\hat{y}} - \frac{1-y^{(\mu)}}{1-\hat{y}}\right] = 0 \Leftrightarrow \sum_{\mu=1}^{p}\frac{y^{(\mu)}}{\hat{y}} = \sum_{\mu=1}^{p}\frac{1-y^{(\mu)}}{1-\hat{y}}$$
$$\Leftrightarrow \sum_{\mu=1}^{p} y^{(\mu)}(1-\hat{y}) = \sum_{\mu=1}^{p}\hat{y}\left(1-y^{(\mu)}\right) \Leftrightarrow \sum_{\mu=1}^{p} y^{(\mu)} - \sum_{\mu=1}^{p} y^{(\mu)}\hat{y} = \sum_{\mu=1}^{p}\hat{y} - \sum_{\mu=1}^{p}\hat{y}y^{(\mu)}$$
$$\Leftrightarrow \sum_{\mu=1}^{p} y^{(\mu)} = \sum_{\mu=1}^{p}\hat{y}$$

Thus, it cannot directly been followed that the perceptron output must be identical to the target output for each training pattern. However, if there is a deviation between perceptron output and target for a given training pattern (which is compensated at other training patterns such that the sum of the perceptron outputs corresponds to the sum of the targets), this training pattern yields a larger per-sample cross-entropy loss, and similarly also the other training patterns yields a larger per-sample cross-entropy loss compared to the minimal value (since the per-sample cross-entropy loss has a unique global minimum, which is assumed if the perceptron output corresponds to the target). Therefore, indeed we can follow that $y^{(\mu)} = \hat{y}$ must hold for each training pattern μ.

In the minimum, the value of the cross-entropy loss is
$\mathrm{L}_{CE} = -\frac{1}{p}\sum_{\mu=1}^{p}\left[y^{(\mu)}\ln y^{(\mu)} + \left(1-y^{(\mu)}\right)\ln(1-y^{(\mu)})\right]$.
This is just the binary entropy on the training set.

Exercise 3 (Convolutional neural networks):

a) Explain the following terms related to learning in neural networks:

Solution:

- Convolutional neural network
  - A convolutional neural network is a class of feed-forward artificial neural networks, consisting of a sequence of one or more convolution layers, followed by a pooling layer. If this sequence of convolutions and pooling is repeated a sufficient number of times, the resulting network is called a deep convolutional neural network. Key differences to (deep) multi-layer perceptrons are: 1.) The assumption of a spatial arrangement of the input data (e.g. two- or three-

dimensional), 2.) local connectivity, 3.) parameter sharing, 4.) the existence of pooling / subsampling layers.

The assumption of a spatial arrrangement of the input data make convolutional neural network particularly suited e.g. (but not only) for processing image data. Local connectivity is realised by limiting the receptive field of a hidden neuron (see below) to a local area (“patch”) in its input (as opposed to the full input in a fully connected multi-layer perceptron). Thus, each hidden unit computes the response of applying a weight vector to a small input patch. Parameter sharing means that different hidden units in the same layer share the weight vector; just the input patch is shifted in space corresponding to the “location” of the hidden unit in the spatial arrangement. In this way, the application of the weight vector to the input can be regarded as applying a convolution of the input with the weight vector, which in this context is often called a “filter” or “kernel”. Therefore, such a layer (realizing the ideas of local connectivity and parameter sharing) is called a “convolution layer”. Note that in practice often a correlation is performed instead of a convolution, corresponding to the dot product between the input and the filter (kernel). Pooling (subsampling) serves to reduce the number of parameters and to increase the receptive field of later layers with respect to the input. (Note that in some architectures requiring a high spatial resolution, e.g. in semantic segmentation, pooling / subsampling may be omitted). Historically, after the sequence of convolutional and pooling layers, finally some fully connected layers are being added to compute the output (classification or regression label) corresponding to the input as a whole (e.g. some object category for the input image).

- Filter, Kernel
  - In a convolution layer, the weight vector of a hidden neuron (which is shared among all neurons in that layer) is called a “filter” or “kernel” (or a “feature detector”); these terms are used synonymously. The application of this weight vector to the input by the different hidden units can be interpreted as a convolution of the input with the weight vector, analogously to applying a filter to the input (therefore the term “filter”). In image processing, the term “kernel” refers to a small matrix applied to an image, so the filter applied to an image can be seen as a kernel. Since the filter highlights or detects some particular aspect or “feature” of the image (e.g. the presence of an edge), the filter or kernel can be seen as a feature detector. The result of applying one particular filter (kernel) to an image is called a feature map (see below). Note that while the dimensions of the filter (kernel) are limited with regard to the spatial dimensions of the image (corresponding to the receptive field), the inputs of all input channels (feature maps, see below) are considered without restriction.

- Feature map
  - A feature map is the result of appyling a particular kernel (filter) to an input. The size of the input image, the size of the filter and the type of padding determine the size of the feature map. Each feature map represents the presence of a particular feature (e.g. presence of a horizontal edge) in the input image at various locations in the image (depending on the type of padding). Detecting several features in the input image (e.g. a horizontal and a vertical edge) corresponds to applying several filters to the input image (here, a filter for horizontal edges and a filter for vertical edges). Each filter leads to its

"own" feature maps, i.e. the number of output feature maps corresponds to the number of filters (kernels). Note that if the input consists of several feature maps (or input channels), the filter (or kernel) performs a (weighted) summation over all input feature maps without restriction.

- Receptive field
  - In a biological sense, the receptive field of a neuron (in the visual cortex) is the region of visual space in which a visual stimulus affect the neuron's activity. In the context of artificial neural networks, the receptive field of an artificial neuron is the set of presynaptic neurons (or inputs) which affect the activity of the considered neuron. In a multi-layer feedforward network, this may refer to direct activation (i.e. by neurons in the previous layer only) or to indirect activation (i.e. also considering the layers preceding the previous layer), depending on the context. In a convolutional neural network, the receptive field (at the previous layer) corresponds to the filter size of a neuron.

- Pooling (subsampling) layer
  - A pooling or subsampling layer is a layer in a convolutional neural network which combines the activations of several units (within the unit's receptive field in the previous layer) into a single unit of the current layer. For example, "max pooling" simply uses the maximum activation of all units in the receptive field as new value, whereas "average pooling" uses the average value of the unit's activations within the receptive field (average pooling has often been observed to be less efficient than max pooling). By combining the activations of all units within a receptive field into a single value, a strong data reduction is achieved, generally without a significant performance reduction. This yields the following advantages: The number of parameters and thus the memory demand and computation time are reduced; due to the reduction of the number of parameters, the network is less susceptible to overfitting, which potentially also enables the construction of deeper networks; furthermore, the size of the receptive field of a unit with regard to earlier layers (e.g. the input layer) is increased, i.e. after a pooling layer, any unit is influenced by a larger part of the input image than without a pooling layer (i.e., the context in the input layer is increased).

- Fully convolutional network
  - A fully convolutional network is a convolutional neural network without any fully connected layers. This means that all layers in the network are convolution layers (plus potential pooling layers). The consequence is that – in contrast to other convolutional networks containing fully connected layers – the input to a fully convolutional network is not restricted to a specific size, but can also be of larger size, creating an output pixel map (one pixel for the various fixed-input size portions in the input image), instead of outputs assigned to the image as a whole. Fully convolutional networks are generated from "regular" convolutional networks by transforming fully connected layers into convolution layers.

b) (CNN with Keras – optional) The file **`exercise3b.py`** (adapted from the Keras examples, https://github.com/keras-team/keras/blob/master/examples/mnist_cnn.py) applies a CNN to the MNIST classification problem.

Some hints on installing Keras and Tensorflow (not a complete installation guide!) can be found in **`README_installation_cuda_and_tensorflow.txt`**.

Note that plotting a visualization of the model requires pydotprint to work; to this end, pydot and graphviz have to be installed from an anaconda command prompt:

```
conda install pydot
conda install graphviz
```

If the installation is not successful, comment out the following lines:

```
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot
...
SVG(model_to_dot(model).create(prog='dot', format='svg'))
```

Alternatively, you may visualize the model using the line

```
plot_model(best_model, to_file='best_model.png',
show_shapes=True)
```

Further documentation about Keras can be found at https://keras.io/

In particular:

- Documentation about the Keras Sequential model: https://keras.io/getting-started/sequential-model-guide/
- Documentation about Keras convolutional layers: https://keras.io/layers/convolutional/
- Documentation about Keras pooling layers: https://keras.io/layers/pooling/
- Documentation about Keras core layers (dense, dropout, activation etc.): https://keras.io/layers/core/

Run the script **`exercise3b.py`**, visualize the model structure, discuss the values of the most important default parameters and report the accuracy of the model.
Try to improve the accuracy (and / or the number of model parameters) by modifying the model structure and / or the values of the default parameters.

Solution:

The following default parameters are set and can be obtained by typing
`model.get_config()`

Optimizer: AdaDelta with learning rate 1.0 (default, see https://keras.io/optimizers/)
Batch size: 128 (set in **`exercise3b.py`**)
Number of epochs: 12 (set in **`exercise3b.py`**)
Bias initialization: Set to 0
Kernel initialization: Variance scaling with uniform distribution and scale 1.0
Padding: valid
Strides: (1, 1) (default)
No kernel regularization, no bias regularization
Input is provided in “channels last” format.

Running the script **exercise3b.py**, the following results are obtained:

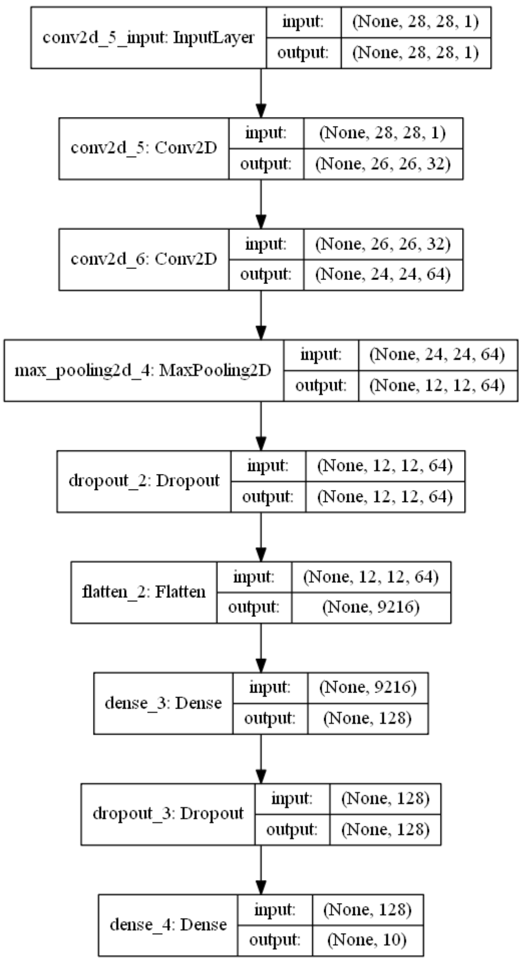
Implementation:

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model. add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

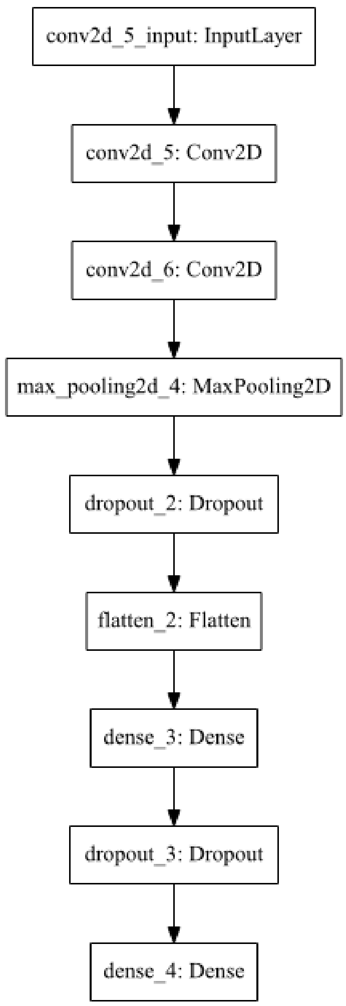
Number of parameters: 1199882

Model structure:

With input and output dimensions:



Without dimensions:



Note: The number of parameters can be verified as follows (see exercise 4):

Output size: (N – F+2P) / S + 1

Number of parameters: $F \cdot F \cdot D_1 \cdot K + K$

N: image heigth /width
D_1: input depth
F: filter (kernel) size
K: number of kernels
P: padding
S: stride

Name	Input	Output dim. calc.	Output	Num. params (% of total parameters)
CONV1 (F=3, K=32, S = 1, P = 0)	[28×28×1]	(28-3)/1+1=26	[26×26×32]	3·3·1·32+32=320 (0.027%)
CONV2 (F=3, K=64, S=1, P=0)	[26×26×32]	(26-3)/1+1 = 24	[24×24×64]	3·3·32·64+64=18496 (1.54%)
MAXPOOL1 (F=2, S=2, P=0)	[24×24×64]	(24-2)/2+1 = 12	[12×12×64]	0
DROPOUT	[12×12×64]	–	[12×12×64]	0
DENSE1 (128 neurons)	[12×12×64]	–	128	12·12·64·128+128=1179776 (98.32%)
DROPOUT	128	–	128	0
DENSE2 (10 neurons)	128	–	10	128·10+10 = 1290 (0.11%)

Total number of parameters: 1.2 million (note that the first dense layer has 98.3% of the parameters!): 320+18496+1179776+1290= 1199882

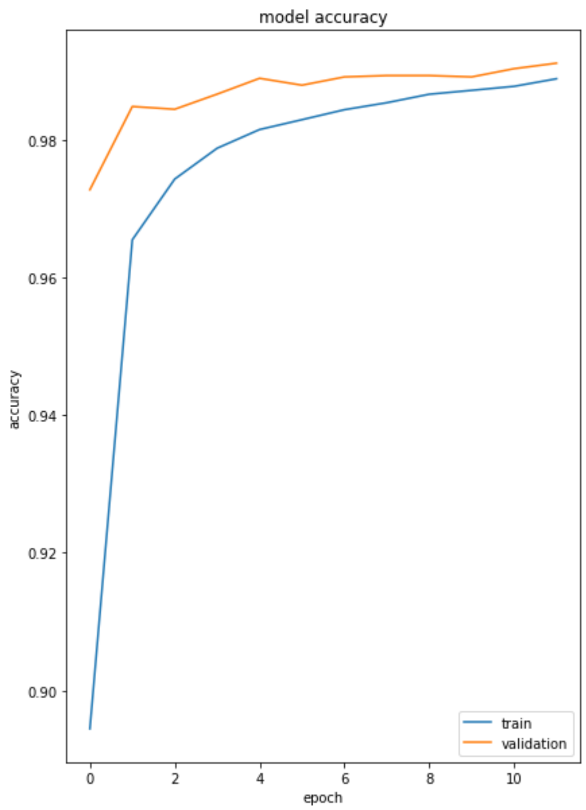
This number of parameters is also obtained by the Keras **`count_param()`** method:

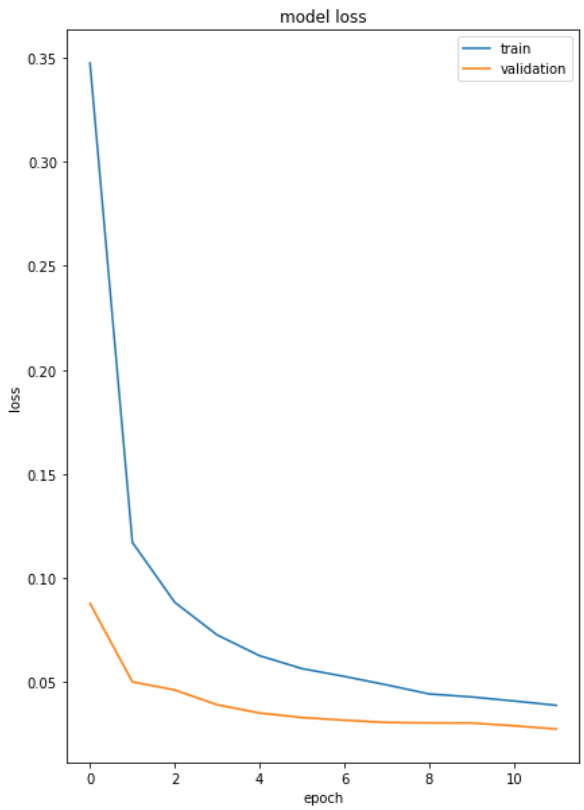
```
model.count_params()
Out[25]: 1199882
```

The script output is as follows (excerpt; due to random component, deviations are possible):

Epoch 1/12
- 63s 1ms/step - loss: 0.3476 - acc: 0.8944 - val_loss: 0.0877 - val_acc: 0.9728
Epoch 2/12
- 62s 1ms/step - loss: 0.1171 - acc: 0.9655 - val_loss: 0.0500 - val_acc: 0.9849
Epoch 3/12
- 62s 1ms/step - loss: 0.0882 - acc: 0.9743 - val_loss: 0.0461 - val_acc: 0.9845
Epoch 4/12
- 62s 1ms/step - loss: 0.0726 - acc: 0.9788 - val_loss: 0.0390 - val_acc: 0.9867
Epoch 5/12
- 62s 1ms/step - loss: 0.0626 - acc: 0.9815 - val_loss: 0.0350 - val_acc: 0.9890
Epoch 6/12
- 61s 1ms/step - loss: 0.0564 - acc: 0.9830 - val_loss: 0.0329 - val_acc: 0.9880
Epoch 7/12

- 61s 1ms/step - loss: 0.0526 - acc: 0.9844 - val_loss: 0.0316 - val_acc: 0.9892
Epoch 8/12
- 61s 1ms/step - loss: 0.0485 - acc: 0.9855 - val_loss: 0.0305 - val_acc: 0.9894
Epoch 9/12
- 62s 1ms/step - loss: 0.0442 - acc: 0.9867 - val_loss: 0.0303 - val_acc: 0.9894
Epoch 10/12
- 61s 1ms/step - loss: 0.0427 - acc: 0.9872 - val_loss: 0.0302 - val_acc: 0.9892
Epoch 11/12
- 61s 1ms/step - loss: 0.0408 - acc: 0.9878 - val_loss: 0.0289 - val_acc: 0.9904
Epoch 12/12
- 62s 1ms/step - loss: 0.0387 - acc: 0.9889 - val_loss: 0.0273 - val_acc: 0.9912
Test loss: 0.0273425589641
Test accuracy: 0.9912



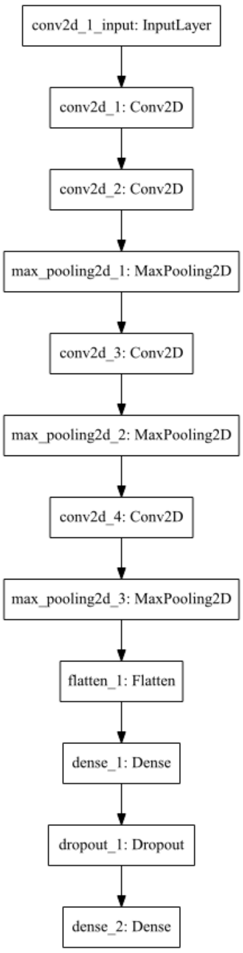


<u>Variation:</u>

- Two more convolutional layers (with 64 filter maps)
- Dense (i.e., fully connected) layer with 64 (instead of 128) units
- Number of parameters: 97482 (instead of 1199882, due to reduction in number of fully connected units)

Implementation:

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax')
```

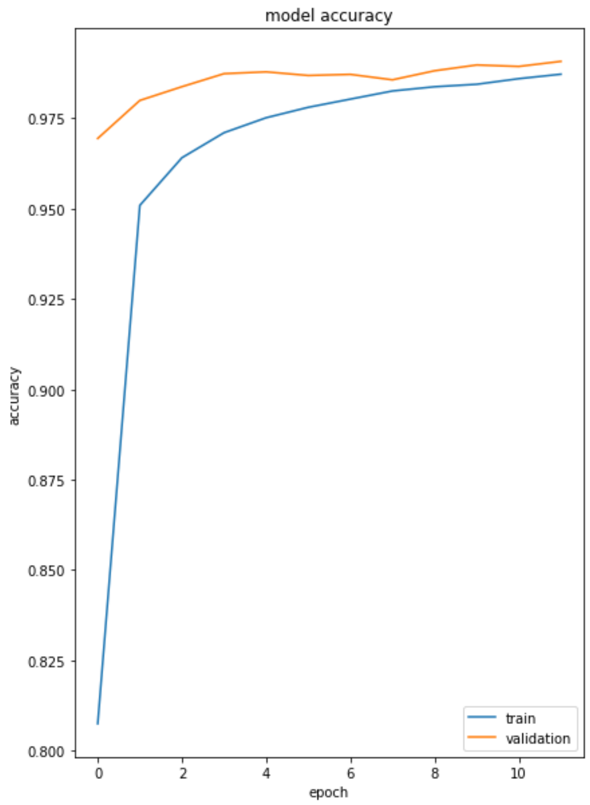


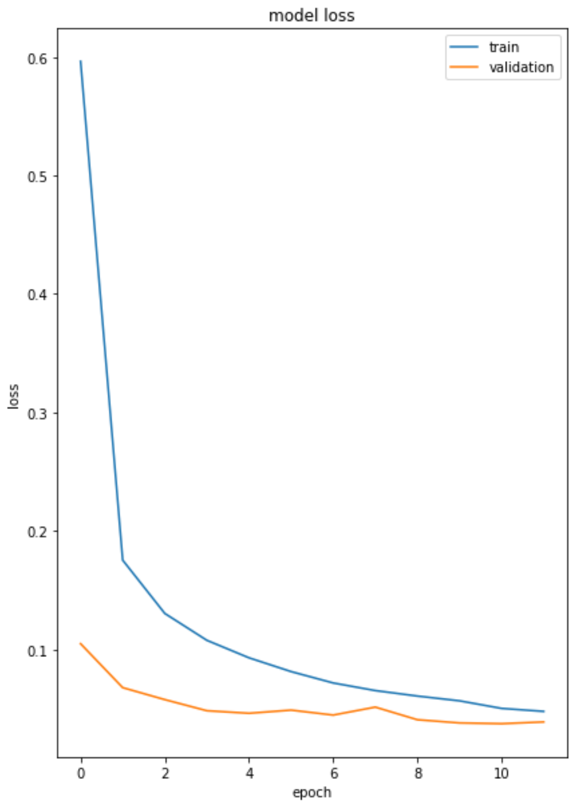
Number of parameters: 97482

Script output (excerpt):

Epoch 1/12
- 64s 1ms/step - loss: 0.5963 - acc: 0.8076 - val_loss: 0.1049 - val_acc: 0.9693
Epoch 2/12
- 60s 1ms/step - loss: 0.1754 - acc: 0.9508 - val_loss: 0.0678 - val_acc: 0.9798
Epoch 3/12
- 60s 993us/step - loss: 0.1304 - acc: 0.9640 - val_loss: 0.0577 - val_acc: 0.9836
Epoch 4/12
- 60s 1ms/step - loss: 0.1077 - acc: 0.9709 - val_loss: 0.0483 - val_acc: 0.9872
Epoch 5/12
- 60s 1ms/step - loss: 0.0930 - acc: 0.9750 - val_loss: 0.0462 - val_acc: 0.9877
Epoch 6/12
- 60s 1ms/step - loss: 0.0814 - acc: 0.9779 - val_loss: 0.0489 - val_acc: 0.9867
Epoch 7/12
- 60s 1ms/step - loss: 0.0718 - acc: 0.9802 - val_loss: 0.0447 - val_acc: 0.9870
Epoch 8/12
- 60s 1ms/step - loss: 0.0653 - acc: 0.9824 - val_loss: 0.0514 - val_acc: 0.9855
Epoch 9/12
- 61s 1ms/step - loss: 0.0607 - acc: 0.9836 - val_loss: 0.0407 - val_acc: 0.9880
Epoch 10/12
- 61s 1ms/step - loss: 0.0567 - acc: 0.9843 - val_loss: 0.0380 - val_acc: 0.9896
Epoch 11/12
- 61s 1ms/step - loss: 0.0503 - acc: 0.9858 - val_loss: 0.0374 - val_acc: 0.9892
Epoch 12/12
- 60s 1ms/step - loss: 0.0478 - acc: 0.9870 - val_loss: 0.0389 - val_acc: 0.9906
Test loss: 0.0388837206187
Test accuracy: 0.9906

i.e. a similar test accuracy with much less parameters!

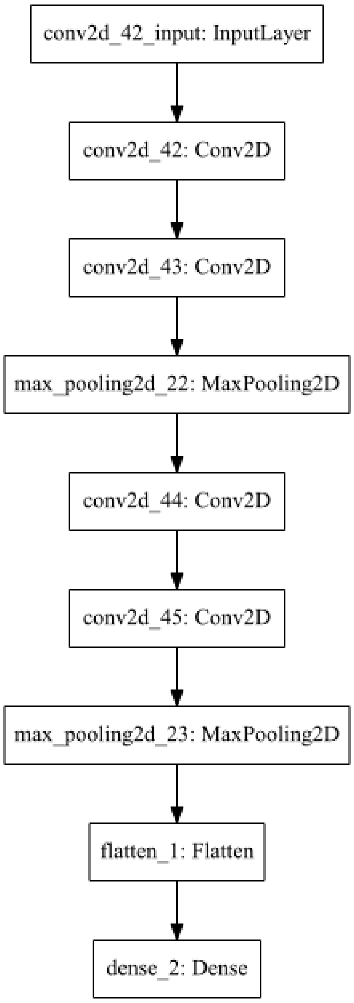




Second variation:

- Less convolutional layers and less feature maps in last convolutional layer
- No dropout

Note that to find an appropriate model configuration, the input and output dimensions must be considered!



Implementation:

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Conv2D(10, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(num_classes, activation='softmax'))
```

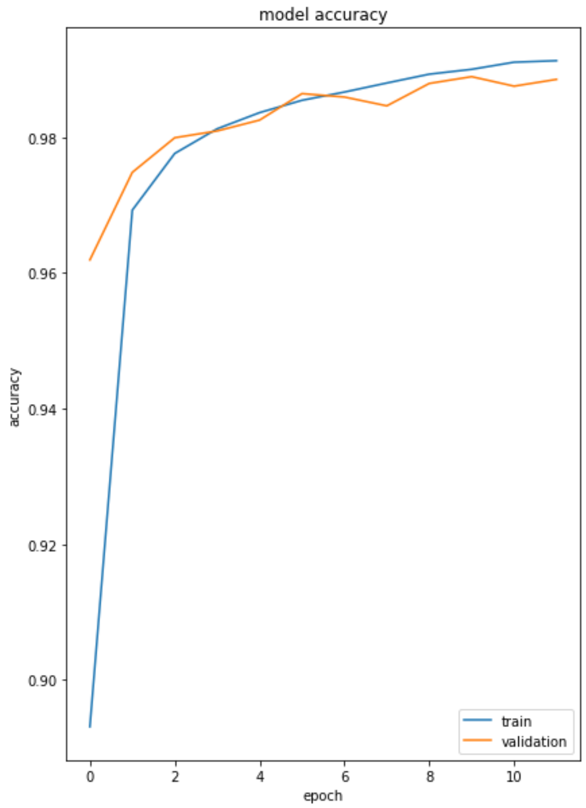
Number of parameters: 63124

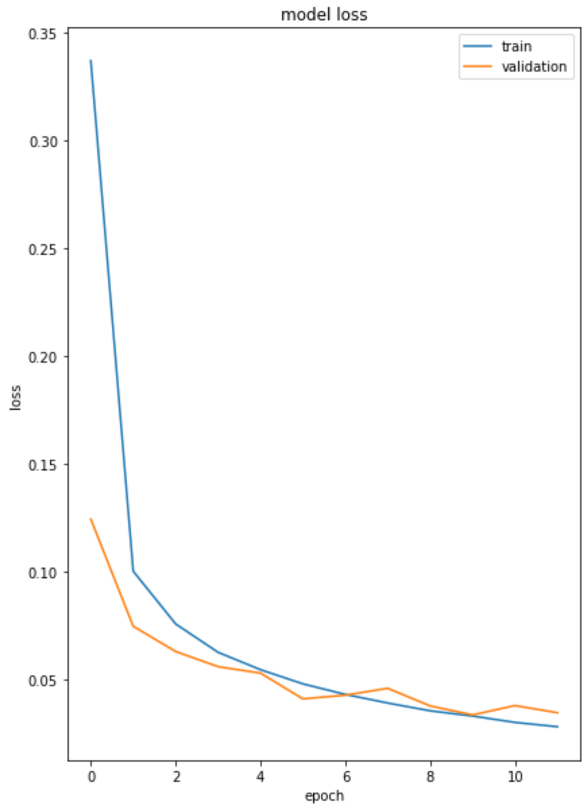
Script output (excerpt):

Epoch 1/12
- 62s 1ms/step - loss: 0.3367 - acc: 0.8931 - val_loss: 0.1240 - val_acc: 0.9619

Epoch 2/12
- 61s 1ms/step - loss: 0.0999 - acc: 0.9692 - val_loss: 0.0744 - val_acc: 0.9748
Epoch 3/12
- 60s 993us/step - loss: 0.0755 - acc: 0.9776 - val_loss: 0.0627 - val_acc: 0.9799
Epoch 4/12
- 60s 992us/step - loss: 0.0623 - acc: 0.9812 - val_loss: 0.0557 - val_acc: 0.9809
Epoch 5/12
- 60s 998us/step - loss: 0.0543 - acc: 0.9836 - val_loss: 0.0527 - val_acc: 0.9825
Epoch 6/12
- 60s 995us/step - loss: 0.0477 - acc: 0.9854 - val_loss: 0.0407 - val_acc: 0.9864
Epoch 7/12
- 59s 986us/step - loss: 0.0428 - acc: 0.9866 - val_loss: 0.0424 - val_acc: 0.9859
Epoch 8/12
- 59s 989us/step - loss: 0.0388 - acc: 0.9880 - val_loss: 0.0456 - val_acc: 0.9846
Epoch 9/12
- 59s 984us/step - loss: 0.0352 - acc: 0.9893 - val_loss: 0.0374 - val_acc: 0.9879
Epoch 10/12
- 60s 1ms/step - loss: 0.0328 - acc: 0.9900 - val_loss: 0.0333 - val_acc: 0.9889
Epoch 11/12
- 59s 987us/step - loss: 0.0298 - acc: 0.9910 - val_loss: 0.0376 - val_acc: 0.9875
Epoch 12/12
- 60s 997us/step - loss: 0.0278 - acc: 0.9913 - val_loss: 0.0343 - val_acc: 0.9885

Test loss: 0.0342879460627
Test accuracy: 0.9885





i.e. slightly worse performance, but lowest number of parameters

<u>Third variation:</u>

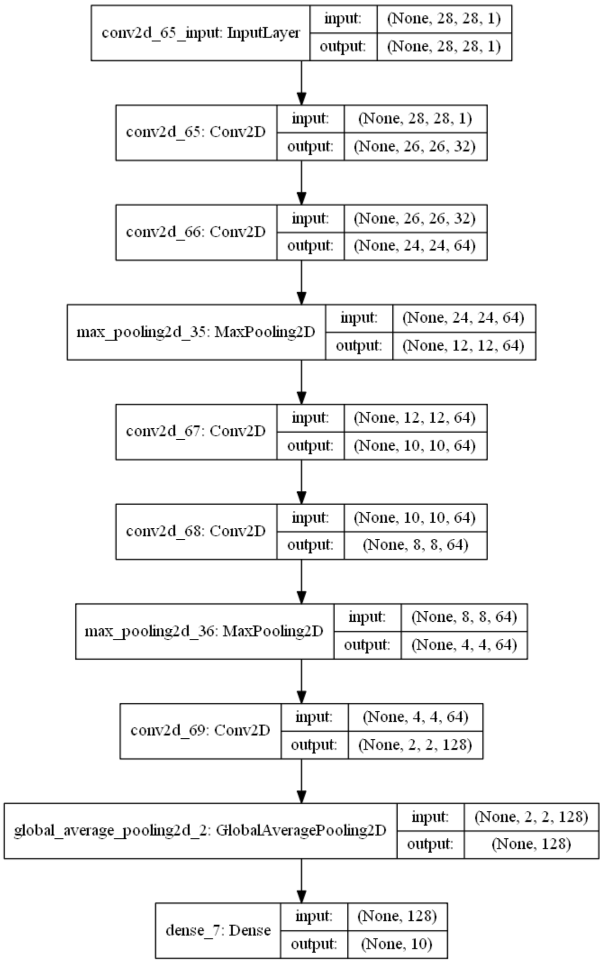
- Global average pooling (please check the literature for its meaning)
- More filter maps, i.e. larger dense layer

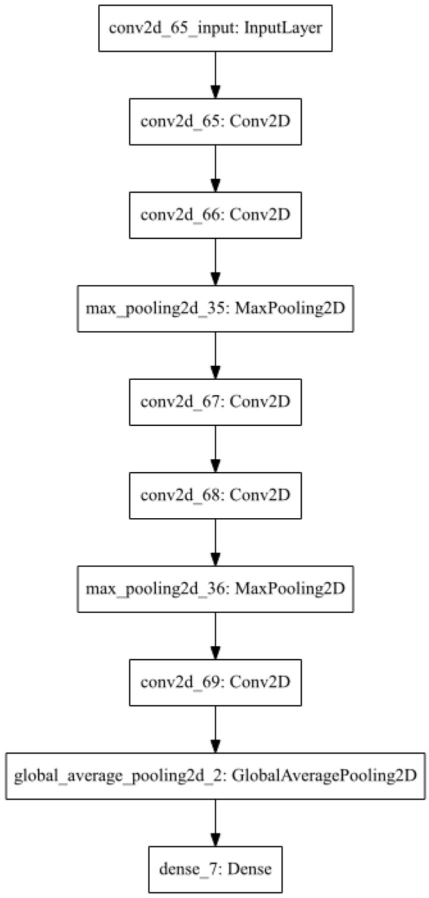
```
from keras.layers import Conv2D, MaxPooling2D,
GlobalAveragePooling2D

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(128, (3,3), activation='relu'))
model.add(GlobalAveragePooling2D(data_format='channels_last'))
model.add(Dense(num_classes, activation='softmax'))
```

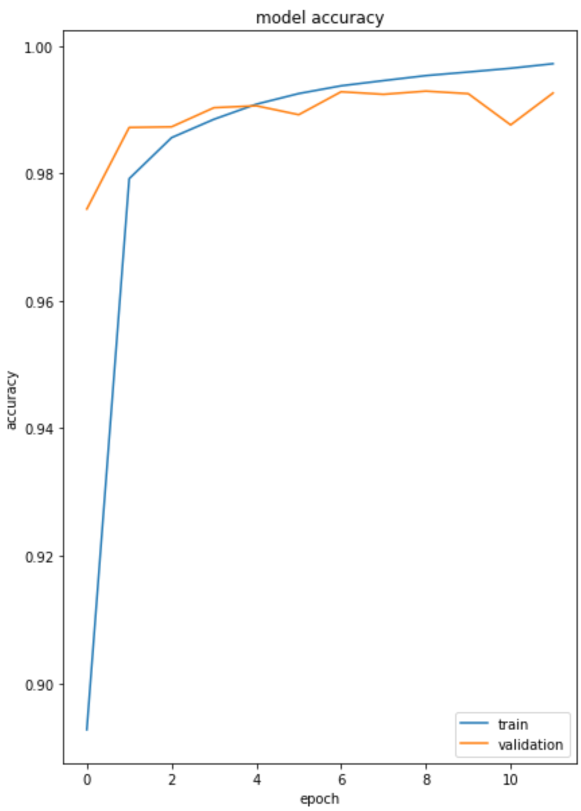
Number of parameters: 167818

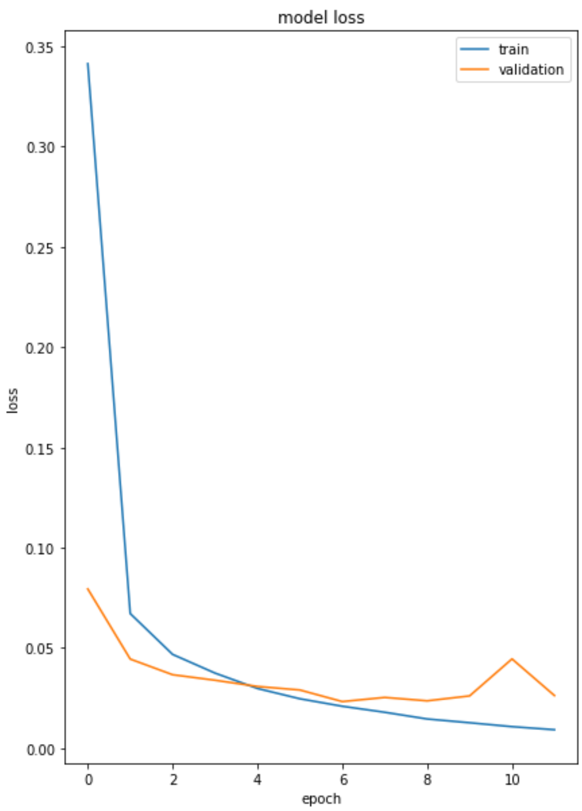
Script output (excerpt):

Epoch 1/12
- 66s 1ms/step - loss: 0.3411 - acc: 0.8927 - val_loss: 0.0794 - val_acc: 0.9744
Epoch 2/12
- 64s 1ms/step - loss: 0.0672 - acc: 0.9792 - val_loss: 0.0445 - val_acc: 0.9872
Epoch 3/12
- 63s 1ms/step - loss: 0.0468 - acc: 0.9856 - val_loss: 0.0367 - val_acc: 0.9873
Epoch 4/12
- 65s 1ms/step - loss: 0.0375 - acc: 0.9885 - val_loss: 0.0339 - val_acc: 0.9903
Epoch 5/12
- 64s 1ms/step - loss: 0.0298 - acc: 0.9908 - val_loss: 0.0308 - val_acc: 0.9906
Epoch 6/12
- 65s 1ms/step - loss: 0.0247 - acc: 0.9925 - val_loss: 0.0291 - val_acc: 0.9892
Epoch 7/12
- 64s 1ms/step - loss: 0.0210 - acc: 0.9937 - val_loss: 0.0233 - val_acc: 0.9928
Epoch 8/12
- 64s 1ms/step - loss: 0.0180 - acc: 0.9945 - val_loss: 0.0254 - val_acc: 0.9924
Epoch 9/12
- 64s 1ms/step - loss: 0.0146 - acc: 0.9953 - val_loss: 0.0237 - val_acc: 0.9929
Epoch 10/12
- 64s 1ms/step - loss: 0.0128 - acc: 0.9959 - val_loss: 0.0261 - val_acc: 0.9925
Epoch 11/12
- 64s 1ms/step - loss: 0.0108 - acc: 0.9965 - val_loss: 0.0446 - val_acc: 0.9876
Epoch 12/12
- 65s 1ms/step - loss: 0.0093 - acc: 0.9972 - val_loss: 0.0263 - val_acc: 0.9926


conv2d_65_input: InputLayer
input: (None, 28, 28, 1)
output: (None, 28, 28, 1)
conv2d_65: Conv2D
input: (None, 28, 28, 1)
output: (None, 26, 26, 32)
conv2d_66: Conv2D
input: (None, 26, 26, 32)
output: (None, 24, 24, 64)
max_pooling2d_35: MaxPooling2D
input: (None, 24, 24, 64)
output: (None, 12, 12, 64)
conv2d_67: Conv2D
input: (None, 12, 12, 64)
output: (None, 10, 10, 64)
conv2d_68: Conv2D
input: (None, 10, 10, 64)
output: (None, 8, 8, 64)
max_pooling2d_36: MaxPooling2D
input: (None, 8, 8, 64)
output: (None, 4, 4, 64)
conv2d_69: Conv2D
input: (None, 4, 4, 64)
output: (None, 2, 2, 128)
global_average_pooling2d_2: GlobalAveragePooling2D
input: (None, 2, 2, 128)
output: (None, 128)
dense_7: Dense
input: (None, 128)
output: (None, 10)


conv2d_65_input: InputLayer
conv2d_65: Conv2D
conv2d_66: Conv2D
max_pooling2d_35: MaxPooling2D
conv2d_67: Conv2D
conv2d_68: Conv2D
max_pooling2d_36: MaxPooling2D
conv2d_69: Conv2D
global_average_pooling2d_2: GlobalAveragePooling2D
dense_7: Dense


model accuracy
accuracy
epoch
train
validation


model loss
loss
epoch
train
validation

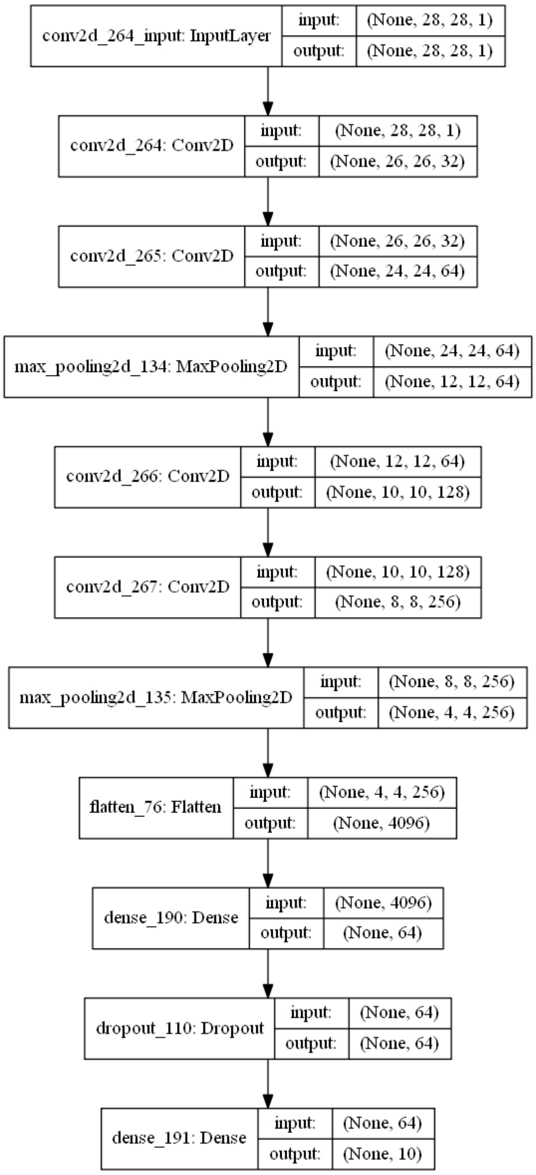
Remark:

The search for the best model configuration can at least partly be automated by using the sklearn grid search functionality; see the script **`exercise3b_grid_search.py`**.

Script output:

```
The parameters of the best model are:
{'dense_layer_sizes': [64], 'filters': [[32, 64], [128, 256]],
'kernel_size': 3, 'pool_size': 2}
10000/10000 [==============================] - 7s 690us/step
loss :  0.0188569785806
acc :  0.9941
Number of parameters of best model: 650698
```

Best model (out of those tested):



For comparison: DNN (i.e. fully connected MLP) with Keras (**`mnist_dnn.py`**):

Epoch 1/30
- 11s 185us/step - loss: 0.3897 - acc: 0.8977
Epoch 2/30
- 9s 154us/step - loss: 0.1902 - acc: 0.9459
Epoch 3/30
- 9s 151us/step - loss: 0.1429 - acc: 0.9599
Epoch 4/30
- 9s 151us/step - loss: 0.1156 - acc: 0.9679
Epoch 5/30
- 9s 149us/step - loss: 0.0966 - acc: 0.9738
Epoch 6/30
- 9s 155us/step - loss: 0.0832 - acc: 0.9773
Epoch 7/30
- 9s 150us/step - loss: 0.0723 - acc: 0.9806
Epoch 8/30
- 10s 159us/step - loss: 0.0643 - acc: 0.9826
Epoch 9/30
- 9s 153us/step - loss: 0.0566 - acc: 0.9849
Epoch 10/30
- 9s 154us/step - loss: 0.0510 - acc: 0.9867
Epoch 11/30
- 10s 162us/step - loss: 0.0457 - acc: 0.9880
Epoch 12/30
- 9s 149us/step - loss: 0.0410 - acc: 0.9898
Epoch 13/30
- 18s 306us/step - loss: 0.0375 - acc: 0.9905
Epoch 14/30
- 12s 202us/step - loss: 0.0340 - acc: 0.9915
Epoch 15/30
- 11s 188us/step - loss: 0.0308 - acc: 0.9924
Epoch 16/30
- 9s 155us/step - loss: 0.0282 - acc: 0.9934
Epoch 17/30
- 9s 148us/step - loss: 0.0259 - acc: 0.9942
Epoch 18/30
- 9s 153us/step - loss: 0.0237 - acc: 0.9947
Epoch 19/30
- 9s 146us/step - loss: 0.0217 - acc: 0.9953
Epoch 20/30
- 9s 149us/step - loss: 0.0199 - acc: 0.9957
Epoch 21/30
- 9s 150us/step - loss: 0.0179 - acc: 0.9965
Epoch 22/30
- 9s 155us/step - loss: 0.0165 - acc: 0.9971
Epoch 23/30
- 10s 160us/step - loss: 0.0152 - acc: 0.9973
Epoch 24/30
- 9s 157us/step - loss: 0.0139 - acc: 0.9977
Epoch 25/30

- 10s 160us/step - loss: 0.0127 - acc: 0.9980
Epoch 26/30
- 9s 157us/step - loss: 0.0116 - acc: 0.9983
Epoch 27/30
- 9s 153us/step - loss: 0.0107 - acc: 0.9984
Epoch 28/30
- 9s 150us/step - loss: 0.0098 - acc: 0.9986
Epoch 29/30
- 9s 153us/step - loss: 0.0090 - acc: 0.9989
Epoch 30/30
- 9s 157us/step - loss: 0.0082 - acc: 0.9989
Test accuracy: 0.9818
Number of parameters:

Name	Input	Output dim. calc.	Output	Num. params (% of total parameters)
DENSE1	784	–	400	784*400+400=314000 (0.027%)
DENSE2	400	–	10	400*10+10=4010 (0.027%)

Total number of parameters: 0.32 million (314000+4010= 318010)

This number of parameters is also obtained by the Keras **`count_param()`** method:

```
model.count_params()
Out[25]: 318010
```

Exercise 4 (Number of parameters in a fully connected and a convolutional network):

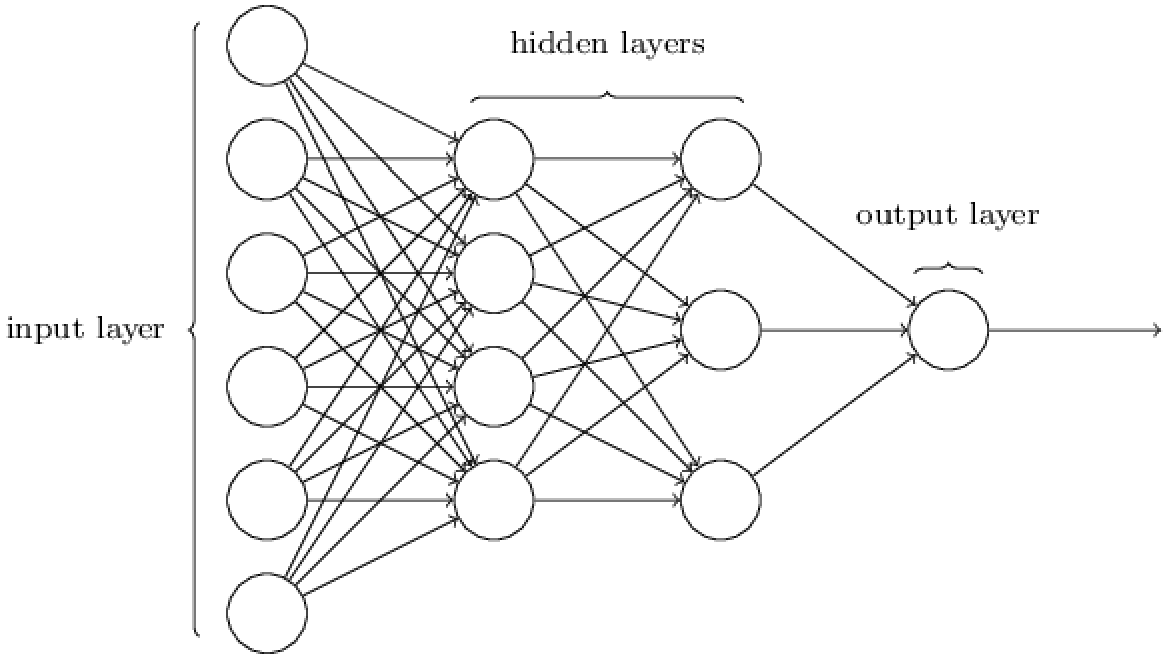
a) For a fully connected multi-layer perceptron containing two hidden layers with 100 hidden units each which is designed for the MNIST classification problem, calculate the number of learnable parameters (i.e. parameters which are learned using the backpropagation algorithm).

Solution:

An input layer suitable for the MNIST digits containing 784 pixel should have 784 input units (remark: input is a 28×28 pixel image, without padding). Full connection to the 100 hidden neurons involves 784*100 synaptic weights, plus 100 bias parameters of the hidden units. Full connection between 100 hidden neurons of the first hidden layer and 100 hidden units of the second hidden layer involves 100*100 synaptic weights, plus 100 bias parameter. Since there are 10 classes, the number of output units is 10. Full connection between 100 hidden neurons and 10 output neurons needs 100*10 synaptic weights, plus 10 bias parameters. So altogether we have

784*100 + 100 + 100*100 + 100 + 100*10 + 10 = 89610 parameters.

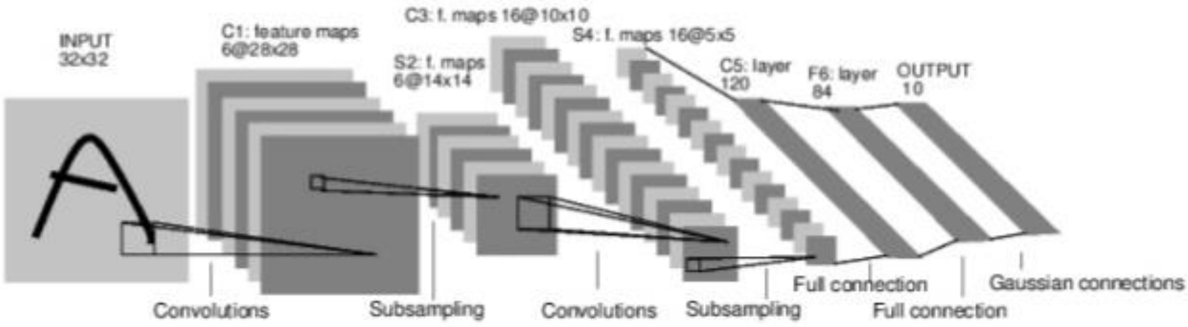
784 input Neurons | 100 hidden neurons | 100 hidden neurons | 10 output neurons



784*100 weights | 100*100 weights | 100*10 weights

100 bias parameter | 100 bias parameter | 10 bias parameter

b) The figure shows the architecture of the famous LeNet-5 convolutional network. Calculate the number of trainable parameters and the number of connections (synaptic weights plus biases, i.e. in augmented space).
Cx: Convolution layer x, Sx: Subsampling layer x, Fx: Fully connected layer x



Solution:

General remark regarding the number of connections: There are two ways of counting the number of connections, either only taking into account pre-synaptic neurons (i.e., without counting the threshold or bias as individual "connection"; referred to as "without bias"), or additionally counting the threshold or bias as individual connection (which makes sense in the "augmented" notation; referred to as "with bias").

Layer C1:
Since the input is a 32 x 32 pixel image (including padding), and the convolutions in C1 yield 28 x 28 feature maps, the filter kernel must be of size 5 x 5, such that each unit of C1 has a 5 x 5 receptive field in the input layer. There are 6 feature maps, so 6 different filter kernels to learn, and each filter has 5*5 weights plus 1 bias.
Number of parameters to learn: (5*5 + 1)*6 = 156 parameter.
Connections (without bias): Each of the 28*28*6 units in C1 is connected to 5*5 units of the receptive field so that the number of connections is 28*28*6*5*5= 117600.
Note that if these layers were fully connected, there were 28*28*6*32*32 = 4816896 connections.
Connections (with bias): Each of the 28*28*6 units in C1 is connected to 5*5 + 1 units of the receptive field (in augmented space), so that the number of connections is 28*28*6*(5*5+1) = 122304.
Note that if these layers were fully connected, there were 28*28*6*(32*32+1) = 4821600 connections.
Receptive field: As mentioned above, each unit of C1 has a 5 x 5 receptive field in the input layer.

Layer S2:
This is a subsampling layer with 6 feature maps of size 14 x 14, i.e. there are 2x2 nonoverlapping receptive fields in C1.
Number of parameters to learn: The subsampling layer introduces zero parameters since it computes a fixed function of the input.
Connections (without bias): Each of the 14*14*6 units in S2 is connected to 2*2 units of the receptive field, so that the number of connections is 14*14*6*2*2 = 4704.

Connections (with bias): Each of the 14*14*6 units in S2 is connected to (2*2 + 1) units of the receptive field (plus bias), so that the number of connections is 14*14*6*(2*2 + 1) = 5880.
Note that the feature maps are processed independently.

Layer C3:
Since the input size is 14 x 14 and the output size 10 x 10, the filter size is again 5 x 5. There are 6 input filter maps (so each filter has 5*5*6 weights and 1 bias) and 16 output filter maps, i.e. there are 16 filter.
Number of parameters to learn: (5*5*6 + 1) * 16 = 2416
Connections (without bias): Each of the 10*10*16 units in C3 is connected to the 5*5*6 units of the receptive field over 6 input filters, so that the number of connections is 10*10*16*5*5*6 = 240000.
Connections (with bias): Each of the 10*10*16 units in C3 is connected to the (5*5*6 + 1) units of the receptive field over 6 input filters, so that the number of connections is 10*10*16*(5*5*6 + 1) = 241600.
Note that each output filter map combines the input filter maps (in contrast to S2).

Note that in the original LeNet-5 architecture, each unit in C3 is connected to several 5 x 5 receptive fields at identical locations in S2, as indicated by the following table:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

TABLE I

EACH COLUMN INDICATES WHICH FEATURE MAP IN S2 ARE COMBINED BY THE UNITS IN A PARTICULAR FEATURE MAP OF C3.

Calculating connections without bias: Therefore, layers 0 – 5 are connected to 3 of the 6 input feature maps, yielding (5*5*3 +1) = 76 parameters to learn and 5*5*3*10*10 = 7500 connections in each of the 6 output layers. Layers 6 – 14 are connected to 4 of the 6 input feature maps, yielding (5*5*4 + 1) = 101 parameters to learn and 5*5*4*10*10 = 10000 connections in each of the 9 output layers. Layer 15 is connected to all 6 input feature maps, yielding (5*5*6 + 1) = 151 parameters to learn and 5*5*6*10*10 = 15000 connections. Therefore, the total number of parameters to learn is 6*76 + 9*101 + 151 = 1516 and the total number of connections is 6*7500 + 9*10000 + 15000 = 150000.
Calculating connections with bias: Layers 0 – 5 are connected to 3 of the 6 input feature maps, yielding (5*5*3 +1) = 76 parameters to learn and (5*5*3 + 1)*10*10 = 7600 connections in each of the 6 output layers. Layers 6 – 14 are connected to 4 of the 6 input feature maps, yielding (5*5*4 + 1) = 101 parameters to learn and (5*5*4 +1)*10*10 = 10100 connections in each of the 9 output layers. Layer 15 is connected to all 6 input feature maps, yielding (5*5*6 + 1) = 151 parameters to learn and (5*5*6 + 1)*10*10 = 15100 connections. Therefore, the total number of parameters to learn is 6*76 + 9*101 + 151 = 1516 and the total number of connections is 6*7600 + 9*10100 + 15100 = 151600.

Layer S4:
This is a subsampling layer with 16 feature maps of size 5 x 5, i.e. there are 2x2 nonoverlapping receptive fields in C3.
Number of parameters to learn: The subsampling layer introduces zero parameters since it computes a fixed function of the input.
Connections without bias: Each of the 5*5*16 units in S4 is connected to (2*2 + 1) units of the receptive field (plus bias), so that the number of connections is 5*5*16*2*2 = 1600. Connections with bias: Each of the 5*5*16 units in S4 is connected to (2*2 + 1) units of the receptive field (plus bias), so that the number of connections is 5*5*16*(2*2 + 1) = 2000.
Again, the subsampling feature maps are processed independently (in contrast to the convolution layers).

Layer C5:
This is a convolution layer with 120 feature maps of size 1 x 1, i.e. the filter size is again 5 x 5. There are 16 input filter maps (so each filter has 5*5*16 weights and 1 bias) and 120 output filter maps, i.e. there are 120 filter.
Number of parameters to learn: (5*5*16 + 1) * 120 = 48120
Connections without bias: Each of the 120 units in C5 is connected to to the 5*5 units of the receptive field in 16 input filters, so the number of connections is 120*5*5*16 = 48000.
Connections with bias: Each of the 120 units in C5 is connected to to the (5*5 + 1) units of the receptive field in 16 input filters, so the number of connections is also 120*(5*5*16 + 1) = 48120 (same as number of parameters to learn).

Layer F6:
This is a fully connected layer with 120 + 1 inputs (including bias) and 84 outputs, so the number of trainable parameters is 84*(120 + 1) = 10164. The number of connections without bias is 84*120 = 10080, the number of connections with bias 10164 (same as number of parameters to learn).

Output layer:
This is again a fully connected layer with 84 + 1 inputs (including bias) and 10 outputs, so the number of trainable parameters is 10*(84 + 1) = 850 and the number of connections without bias is 10*84 = 840. The number of connections with bias is 850 (same as number of parameters to learn).

Summary:

Note: The receptive field size is with respect to the previous layer.
The asterisk (*), i.e., the second number, refers to the original LeNet-5 architecture (see above).

Excluding threshold / bias in the number of connections:

Layer	Receptive field	Parameters to learn	Connections (w/o bias)
C1	5×5	156	117600
S2	2×2	0	4704
C3	5×5	2416 / 1516 (*)	240000 / 150000 (*)
S4	2×2	0	1600
C5	5×5	48120	48000
F6	All neurons (fully connected)	10164	10080
Output	All neurons (fully connected)	850	840
Sum	**–**	**61706 / 60806 (*)**	**422824 / 332824 (*)**

Including threshold / bias in the number of connections:

Layer	Receptive field	Parameters to learn	Connections (with bias)
C1	5×5	156	122304
S2	2×2	0	5880
C3	5×5	2416 / 1516 (*)	241600 / 151600 (*)
S4	2×2	0	2000
C5	5×5	48120	48120
F6	All neurons (fully connected)	10164	10164
Output	All neurons (fully connected)	850	850
Sum	**–**	**61706 / 60806 (*)**	**430918 / 340918 (*)**

c) Calculate the output dimensions and number of parameters of the AlexNet without grouping (the division into two separate paths), i.e. when the AlexNet is realized in a single path.

Solution:

Output size: $(N - F+2P) / S + 1$

Number of parameters: $F \cdot F \cdot D_1 \cdot K + K$

N: image heigth /width
D_1: input depth
F: filter (kernel) size
K: number of kernels
P: padding
S: stride

The results can be presented in table form:

Name	Input	Output dim. calc.	Output	Num. params (% of total parameters)
CONV1 (F=11, K=96, S = 4, P = 0)	[227×227×3]	(227-11)/4+1=55	[55×55×96]	11·11·3·96+96=34944 (0.056%)
MAXPOOL1 (F=3, S=2, P=0)	[55×55×96]	(55-3)/2+1 = 27	[27×27×96]	0
NORM	[27×27×96]	–	[27×27×96]	0
CONV2 (F=5, K=256, S=1, P=2)	[27×27×96]	(27-5+4)/1+1= 27	[27×27×256]	5·5·96·256+256 = 614656 (0.99%)
MAXPOOL2	[27×27×256]	(27-3)/2+1 = 13	[13×13×256]	0
NORM2	[13×13×256]	–	[13×13×256]	0
CONV3 (F=3, K=384, S=1, P=1)	[13×13×256]	(13-3+2)/1+1 = 13	[13×13×384]	3·3·256·384 + 384 = 885120 (1.42%)
CONV4 (F=3, K=384, S=1, P=1)	[13×13×384]	(13-3+2)/1+1 = 13	[13×13×384]	3·3·384·384 + 384 = 1327488 (2.13%)
CONV5 (F=3, K=256, S=1, P=1)	[13×13×384]	(13-3+2)/1+1 = 13	[13×13×256]	3·3·384·256 + 256 = 884992 (1.42%)
MAXPOOL3	[13×13×256]	(13-3)/2+1 = 6	[6×6×256]	0
FC6 (4096 neurons)	[6×6×256]	–	4096	6·6·256·4096+4096=37752832 (60.52%)
FC7 (4096 neurons)	4096	–	4096	4096·4096+4096 = 16781312 (26.90%)
FC8 (1000 neurons)	4096	–	1000	4096·1000+1000 = 4097000 (6.57%)

Total number of parameters: 62 million (note that the fc layers have 94% of the parameters!):

34944+614656+885120+1327488+884992+37752832+16781312+4097000 = 62378344

d) How do these numbers change in the ZF net?

Solution:

Name	Input	Output dim. calc.	Output	Num. params (% of total parameters)
CONV1 (F=7, K=96, S = 2, P = 0)	[227×227×3]	(227-7)/2+1 = 111	[111×111×96]	7·7·3·96+96=14208 (0.003%)
MAXPOOL1 (F=3, S=2, P=0)	[111×111×96]	(111-3)/2+1 = 55	[55×55×96]	0
NORM	[55×55×96]	–	[55×55×96]	0
CONV2 (F=5, K=256, S=1, P=2)	[55×55×96]	(55-5+4)/1+1 = 55	[55×55×256]	5·5·96·256+256 = 614656 (0.16%)
MAXPOOL2	[55×55×256]	(55-3)/2+1 = 27	[27×27×256]	0
NORM2	[27×27×256]	–	[27×27×256]	0
CONV3 (F=3, K=512, S=1, P=1)	[27×27×256]	(27-3+2)/1+1 = 27	[27×27×512]	3·3·256·512 + 512 = 1180160 (0.31%)
CONV4 (F=3, K=1024, S=1, P=1)	[27×27×512]	(27-3+2)/1+1 = 27	[27×27×1024]	3·3·512·1024 + 1024 = 1327488 (0.35%)
CONV5 (F=3, K=512, S=1, P=1)	[27×27×1024]	(27-3+2)/1+1 = 27	[27×27×512]	3·3·384·256 + 256 = 4719616 (1.23%)
MAXPOOL3	[27×27×512]	(27-3)/2+1 = 13	[13×13×512]	0
FC6 (4096 neurons)	[13×13×512]	–	4096	13·13·512·4096+4096=354422784 (92.50%)
FC7 (4096 neurons)	4096	–	4096	4096·4096+4096 = 16781312 (4.38%)
FC8 (1000 neurons)	4096	–	1000	4096·1000+1000 = 4097000 (1.07%)

Total number of parameters: 383 million (note that the fc layers have 98% of the parameters!):
14208+614656+1180160+1327488+4719616+354422784+16781312+4097000 = 383157224