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

Neural Networks and Deep Learning – Summer Term 2020

Exercise sheet 5

Submission due: Wednesday, June 17, 13:15 sharp

Exercise 1 (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: $D = \left\{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(n)}, y^{(n)})\right\}$, $\mathbf{x}^{(\mu)} \in \mathbb{R}^d$, $y^{(\mu)} \in \mathbb{R}$

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

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

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 $D = \left\{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \ldots, (\mathbf{x}^{(p)}, y^{(p)})\right\}$ is defined as

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

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 $\lim_{x \to 0} x ln(x) = 0$, 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:

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

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 $L_{CE}^{\mu} = -y^{(\mu)} \ln y^{(\mu)} - \left(1-y^{(\mu)}\right)\ln(1-y^{(\mu)})$.

Taking the second derivative yields: $\frac{\delta^2 L_{CE}^{\mu}(\boldsymbol{w},y,\hat{y})}{\delta\hat{y}^2} = \frac{y^{(\mu)}}{\hat{y}^2} + \frac{1-y^{(\mu)}}{(1-\hat{y})^2} > 0 \quad \forall\ \hat{y}$ 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 2 (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 on MNIST) The Jupyter notebook provides code to apply a CNN to the MNIST classification problem. Complete the script – choosing a suitable network architecture and providing values for the hyperparamters – and report the accuracy of the model. Try to improve the accuracy of the CNN (and / or to reduce the number of model parameters) by modifying the model structure and / or the values of the hyperparameters and settings. Visualize and discuss the model structure. Compare your results to the results of MNIST classification using a multilayer perceptron (lab sheet 4, exercise 3).

Note: The model can be visualized using the **`plot_model`** method of **`tensorflow.keras.utils`**:
`plot_model(model, to_file='model.png', show_shapes=False, show_layer_names=True)`
This should write a visualization of the model to the file **`model.png`**. However, this may not work in Colab. In this case, you may execute the method locally (without training).

Solution:

The following default configuration is being used in the following experiments (if not stated otherwise):

- Using ReLU activation
- Two convolutional layers with filter size 3 x 3
- Followed by max pooling with stride 2 x 2
- Potentially further convolutional layers following the same scheme
- One or more fully connected layers
- With dropout 0.5 (in fully connected layers)
- Softmax activation function (10 output classes for MNIST)
- Training using the Adam optimizer (without momentum)
- Learning rate 0.001
- No data augmentation
- No batch normalization
- No regularization
- No noise addition
- 50 epochs
- Batch size 128

The following configurations are being investigated:

1) Two convolutional layers with 32 feature maps each, followed by max pooling, and a fully connected layer with 64 hidden units ("[32], [64]")

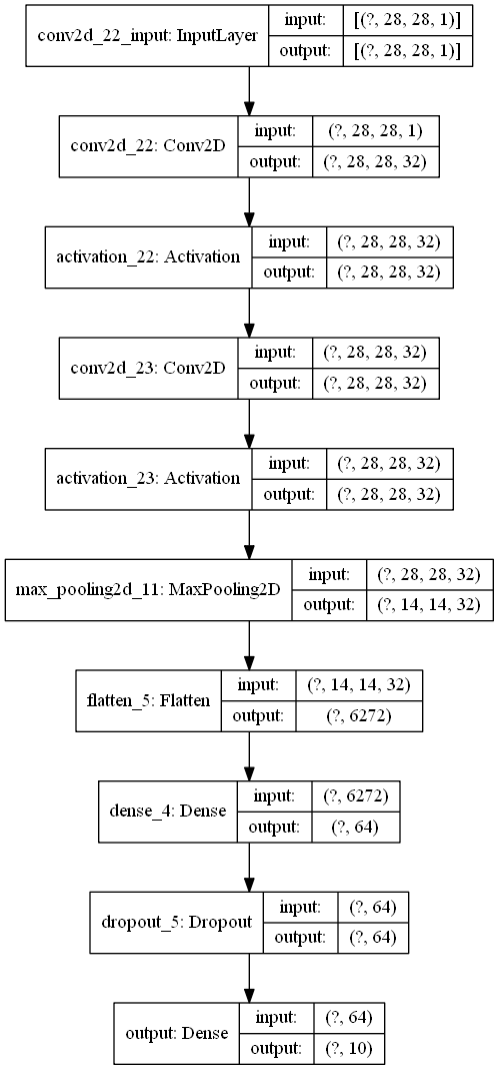
2) Two convolutional layers with 32 feature maps each, followed by max pooling, followed by two additional convolutional layers with 64 feature maps each, followed by a fully connected layer with 128 hidden units ("[32, 64], [128]")

3) Two convolutional layers with 32 feature maps each, followed by max pooling, followed by two additional convolutional layers with 64 feature maps each, followed by a fully connected layer with 64 hidden units ("[32, 64], [64]")

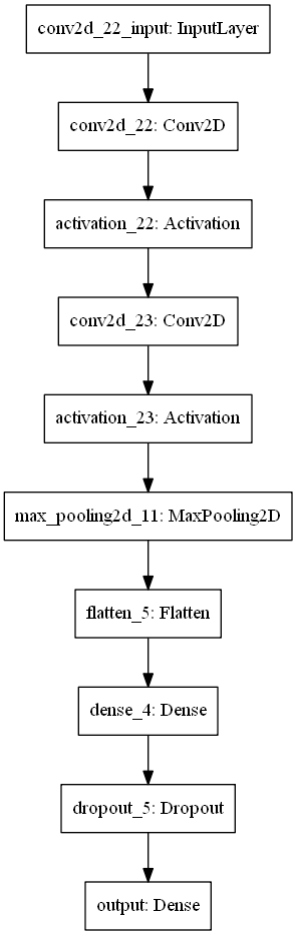
4) Two convolutional layers with 32 feature maps each, followed by max pooling, followed by two additional convolutional layers with 64 feature maps each, followed by max pooling, and again followed by two further convolutional layers with 128 feature maps each, followed by a fully connected layer with 64 hidden units ("[32, 64, 128], [64]"), **including batch normalization** (this was found to be necesary to obtain good performance).

Model structure: Configuration 1)

With input and output dimensions:



Without dimensions:



(The question marks refer to the number of input samples)

Note that Keras counts the activation function, flatten and dropout as individual layers.

In detail with parameters, using the `model.summary()` method:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_97 (Conv2D)           (None, 28, 28, 32)        320
_________________________________________________________________
activation_94 (Activation)   (None, 28, 28, 32)        0
_________________________________________________________________
conv2d_98 (Conv2D)           (None, 28, 28, 32)        9248
_________________________________________________________________
activation_95 (Activation)   (None, 28, 28, 32)        0
_________________________________________________________________
max_pooling2d_48 (MaxPooling (None, 14, 14, 32)        0
_________________________________________________________________
flatten_19 (Flatten)         (None, 6272)              0
_________________________________________________________________
dense_9 (Dense)              (None, 64)                401472
_________________________________________________________________
dropout_10 (Dropout)         (None, 64)                0
_________________________________________________________________
output (Dense)               (None, 10)                650
=================================================================
Total params: 411,690
Trainable params: 411,690
```

Note: The number of parameters can be verified as follows (see exercise 3):

Output size: (N – F+2P) / S + 1

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

Padding: 1 (“same” convolution)

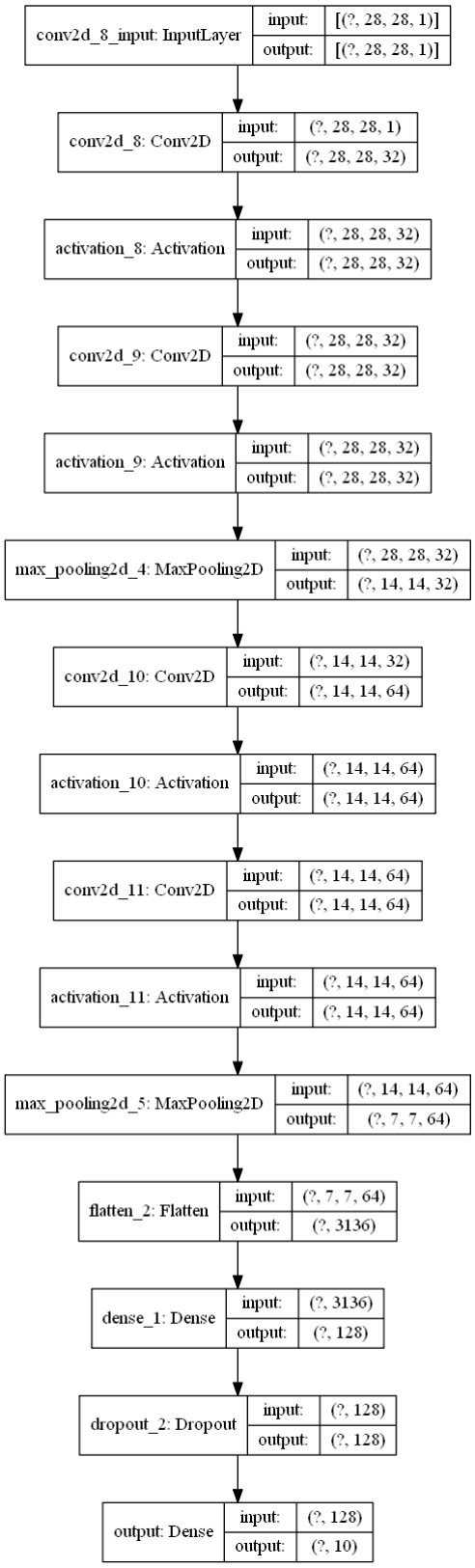
N: image heigth /width
D_1: input depth
F: filter (kernel) size
K: number of kernels
P: padding
S: stride

Name	Input	Output dim. calc.	Output	Num. params (% of total parameters)
CONV1 (F=3, K=32, S = 1, P = 1)	[28×28×1]	(28-3+2)/1+1=28	[28×28×32]	3·3·1·32+32=320 (0.0777%)
CONV2 (F=3, K=32, S=1, P=1)	[28×28×32]	(28-3+2)/1+1 = 28	[28×28×32]	3·3·32·32+32=9248 (2.2464%)
MAXPOOL1 (F=2, S=2, P=0)	[28×28×32]	(28-2)/2+1 = 14	[14×14×32]	0
DENSE1 (128 neurons)	[14×14×32]	–	64	14·14·32·64+64=401472 (97.5180%)
DROPOUT	64	–	64	0
DENSE2 (10 neurons)	64	–	10	64·10+10 = 650 (0.1579%)

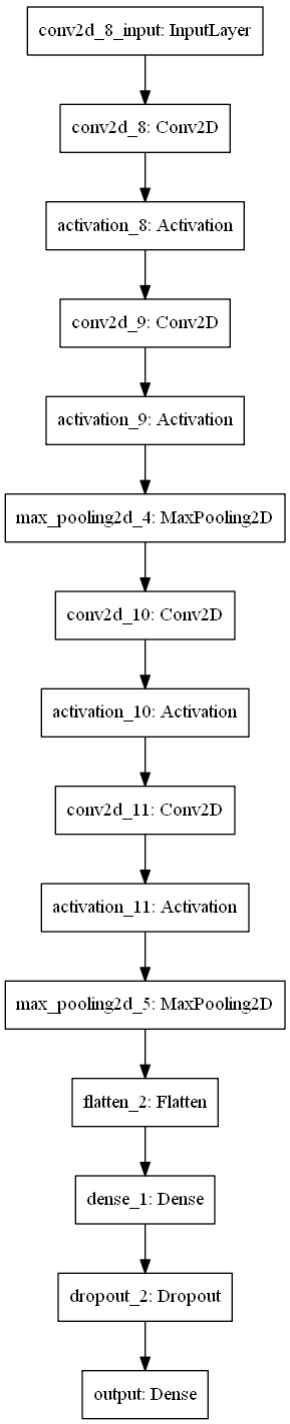
Total number of parameters: 411.690 (note that the first dense layer has 97.5% of the parameters!): 320 + 9248 + 401472 + 650 = 411690

Model structure: Configuration 2)

With input and output dimensions:



Without dimensions:



In detail with parameters, using the **`model.summary()`** method:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        320
_________________________________________________________________
activation (Activation)      (None, 28, 28, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 32)        9248
_________________________________________________________________
activation_1 (Activation)    (None, 28, 28, 32)        0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 64)        18496
_________________________________________________________________
activation_2 (Activation)    (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 14, 14, 64)        36928
_________________________________________________________________
activation_3 (Activation)    (None, 14, 14, 64)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 3136)              0
_________________________________________________________________
dense (Dense)                (None, 128)               401536
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
output (Dense)               (None, 10)                1290
=================================================================
Total params: 467,818
Trainable params: 467,818
```

Note: Again verifying the number of parameters:

Output size: (N – F+2P) / S + 1

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

Padding: 1 (“same” convolution)

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 = 1)	[28×28×1]	(28-3+2)/1+1=28	[28×28×32]	3·3·1·32+32=320 (0.0684%)
CONV2 (F=3, K=32, S=1, P=1)	[28×28×32]	(28-3+2)/1+1 = 28	[28×28×32]	3·3·32·32+32=9248 (1.9768%)
MAXPOOL1 (F=2, S=2, P=0)	[28×28×32]	(28-2)/2+1 = 14	[14×14×32]	0

CONV3 (F=3, K=64, S = 1, P = 1)	[14×14×32]	(14-3+2)/1+1=14	[14×14×64]	3·3·32·64+64=18496 (3.9537%)
CONV4 (F=3, K=64, S=1, P=1)	[14×14×64]	(14-3+2)/1+1 = 14	[14×14×64]	3·3·64·64+64=36928 (7.8937%)
MAXPOOL2 (F=2, S=2, P=0)	[14×14×64]	(14-2)/2+1 = 7	[7×7×64]	0
DENSE1 (128 neurons)	[7×7×64]	–	128	7·7·64·128+128=401536 (85.8317%)
DROPOUT	128	–	128	0
DENSE2 (10 neurons)	128	–	10	128·10+10 = 1290 (0.2758%)

Total number of parameters: 467.818 (note that the first dense layer has 85.8% of the parameters!): 320 + 9248 + 18496 + 36928 + 401536 + 1290 = 467818

Thus one can see that the most influential factor determining the total number of parameters is the number of connections in (first) fully connected layer. In the second configuration, the number of hidden units in the first fully connected layer is twice as large than in the first configuration, and the number of feature maps is also twice as large than in the first configuration, but this compensated by an additional pooling layer which reduces the size of the input of the first convolutional layer by a factor of 4. The number of additional parameters by the increased number of convolutional layers in the second configuration is rather small compared to the total number of parameters (about 55.000 parameters more than in configuration 1).

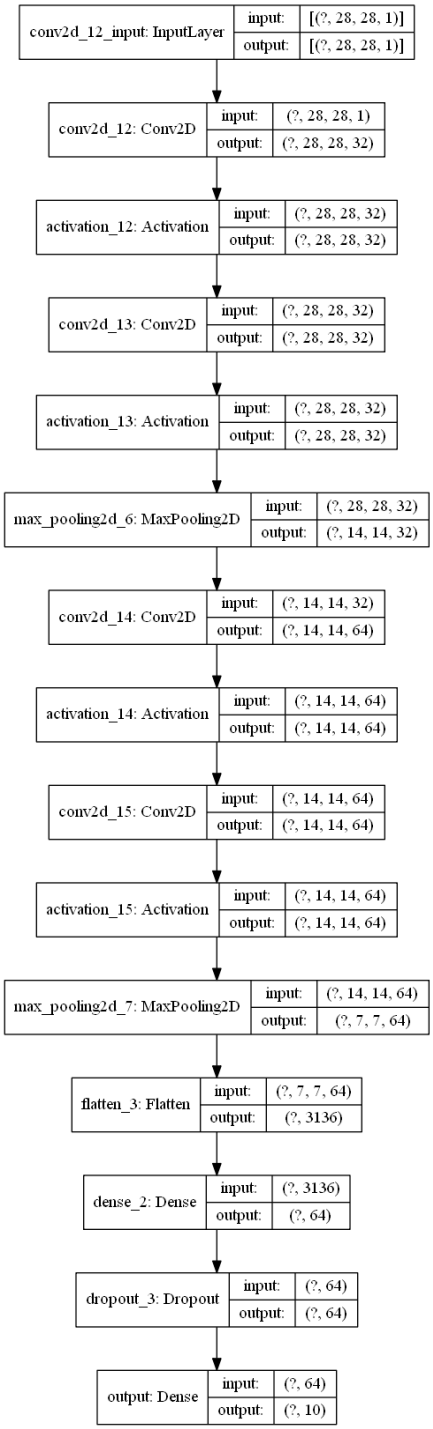
Model structure: Configuration 3)

In detail with parameters, using the **`model.summary()`** method:

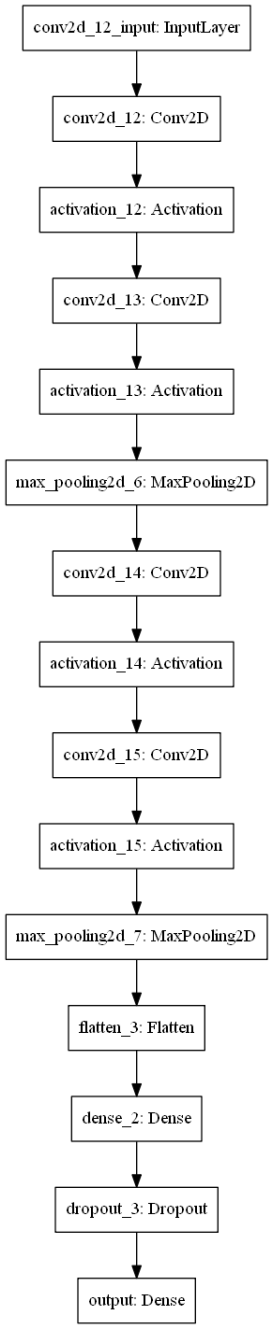
```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (None, 28, 28, 32)        320
_________________________________________________________________
activation_4 (Activation)    (None, 28, 28, 32)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 28, 28, 32)        9248
_________________________________________________________________
activation_5 (Activation)    (None, 28, 28, 32)        0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 14, 14, 64)        18496
_________________________________________________________________
activation_6 (Activation)    (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 14, 14, 64)        36928
_________________________________________________________________
activation_7 (Activation)    (None, 14, 14, 64)        0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
```

```
flatten_1 (Flatten)          (None, 3136)              0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                200768
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
output (Dense)               (None, 10)                650
=================================================================
Total params: 266,410
Trainable params: 266,410
Non-trainable params: 0
```

With input and output dimensions:

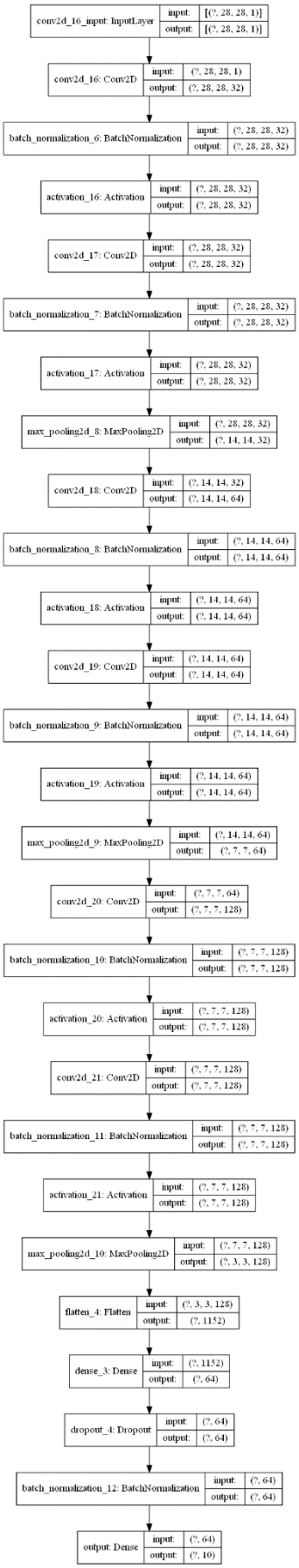


Without dimensions:

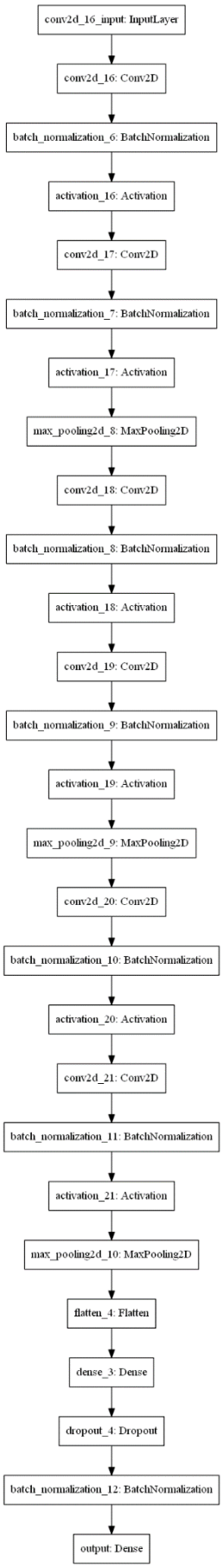


Model structure: Configuration 4)

With input and output dimensions:



Without dimensions:



In detail with parameters, using the **`model.summary()`** method:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        320
_________________________________________________________________
batch_normalization (BatchNo (None, 28, 28, 32)        128
_________________________________________________________________
activation (Activation)      (None, 28, 28, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 32)        9248
_________________________________________________________________
batch_normalization_1 (Batch (None, 28, 28, 32)        128
_________________________________________________________________
activation_1 (Activation)    (None, 28, 28, 32)        0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 64)        18496
_________________________________________________________________
batch_normalization_2 (Batch (None, 14, 14, 64)        256
_________________________________________________________________
activation_2 (Activation)    (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 14, 14, 64)        36928
_________________________________________________________________
batch_normalization_3 (Batch (None, 14, 14, 64)        256
_________________________________________________________________
activation_3 (Activation)    (None, 14, 14, 64)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 7, 7, 128)         73856
_________________________________________________________________
batch_normalization_4 (Batch (None, 7, 7, 128)         512
_________________________________________________________________
activation_4 (Activation)    (None, 7, 7, 128)         0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 7, 7, 128)         147584
_________________________________________________________________
batch_normalization_5 (Batch (None, 7, 7, 128)         512
_________________________________________________________________
activation_5 (Activation)    (None, 7, 7, 128)         0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 3, 3, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 1152)              0
_________________________________________________________________
dense (Dense)                (None, 64)                73792
_________________________________________________________________
dropout (Dropout)            (None, 64)                0
_________________________________________________________________
batch_normalization_6 (Batch (None, 64)                256
_________________________________________________________________
output (Dense)               (None, 10)                650
=================================================================
Total params: 362,922
Trainable params: 361,898
Non-trainable params: 1,024
```

Note that “valid” padding cannot be applied to this architecture since after the second pooling operation the input size would be 4 x 4 prohibiting two further convolutions.

Experimental results for the different configurations:
(results for the multilayer perceptron from the last lab sheet are included in gray)

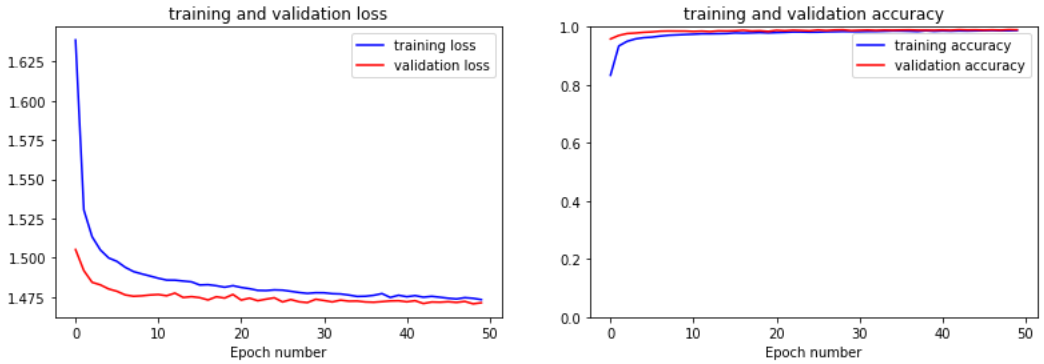
Configuration	Training loss	Training accuracy	Validation loss	Validation accuracy	Test loss	Test accuracy
MLP, 1 hidden layer with 200 units (configuration "C", 159.010 param.)	0.000721	1.000000	0.076720	0.983800	0.070349	0.983000
MLP, 1 hidden layer with 1000 units, (configuration "P", 795.010 param.)	0.187101	0.891420	0.077229	0.985100	0.061507	0.985300
CNN, conf. 1: [32], [64], 411.690 param.	1.473386	0.987800	1.471790	0.989300	1.471394	0.989700
CNN, conf. 2: [32, 64], [128], 467,818 param.	1.480255	0.980900	1.476277	0.984900	1.473528	0.987600
CNN, conf. 3: [32, 64], [64], 266,410 param.	1.487485	0.973660	1.472647	0.988500	1.473321	0.987700
CNN, conf. 4: [32, 64, 128], [64], 362,922 param.	1.463429	0.997960	1.465828	0.995200	1.466105	0.995300

Interpretation:

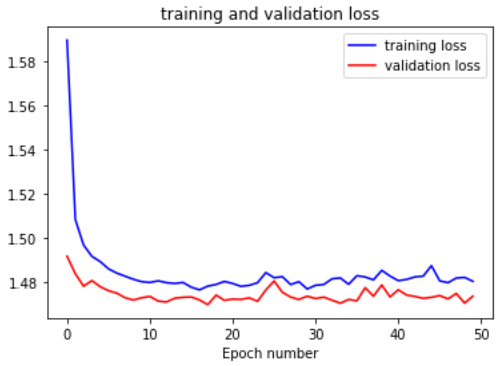
- The CNNs yield accuracies at least as good as the MLPs (with an improvement especially for configuration 4).
- Since the MLPs had only a single hidden layer, the number of parameters of the MLP crucially depends on the number of hidden units (see the number of parameters of the MLP configurations in the first two lines of the table above). Although the CNNs have many more layers than the MLP, the number of parameters is comparable to the number of parameters of the MLP (for a MLP with more hidden layers, the number of parameters would strongly increase!).
- The learning curves (see below) show that the training and validation loss and accuracy have converged (except for configuration 1 where the training loss seems to still decrease).
- Further improvements may be realized by further parameter optimizations (learning rate, batch size, optimizer with variants) and adding data augmentation / regularization.

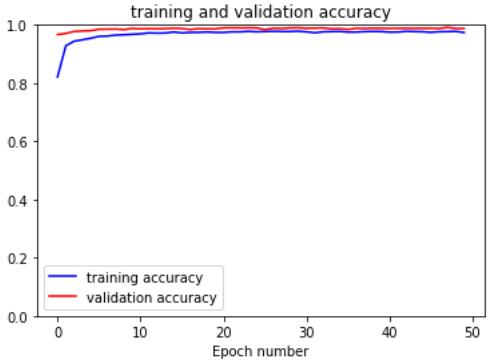
Learning curves:

Configuration 1)

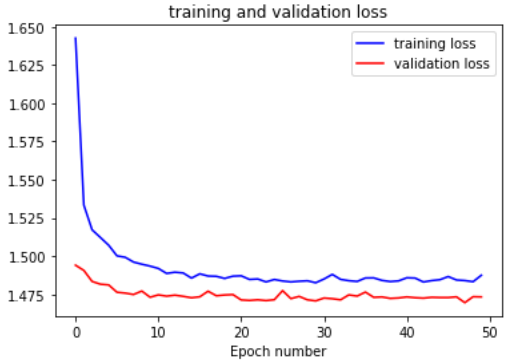


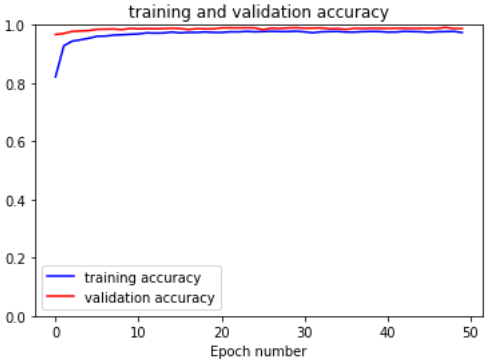
Configuration 2)



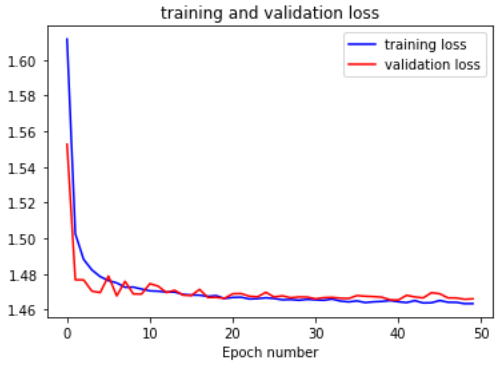


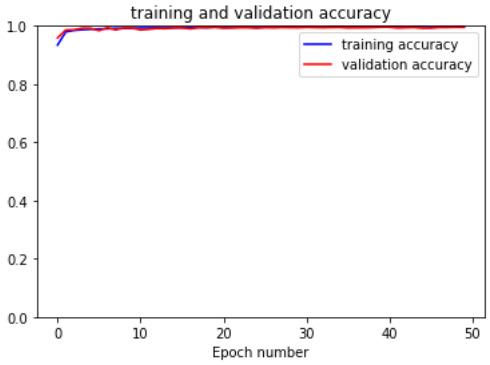
Configuration 3)





Configuration 4)





For comparison: Calculation of parameters for a multilayer perceptron (MLP) with a single hidden layer with 1000 hidden neurons:

Name	Input	Output dim. calc.	Output	Num. params (% of total parameters)
DENSE1	784	–	1000	784*1000+1000=785000 (98.7421%)
DENSE2	400	–	10	1000*10+10=10010 (1.2591%)

Total number of parameters: 795.010 (785000+10010= 795010)

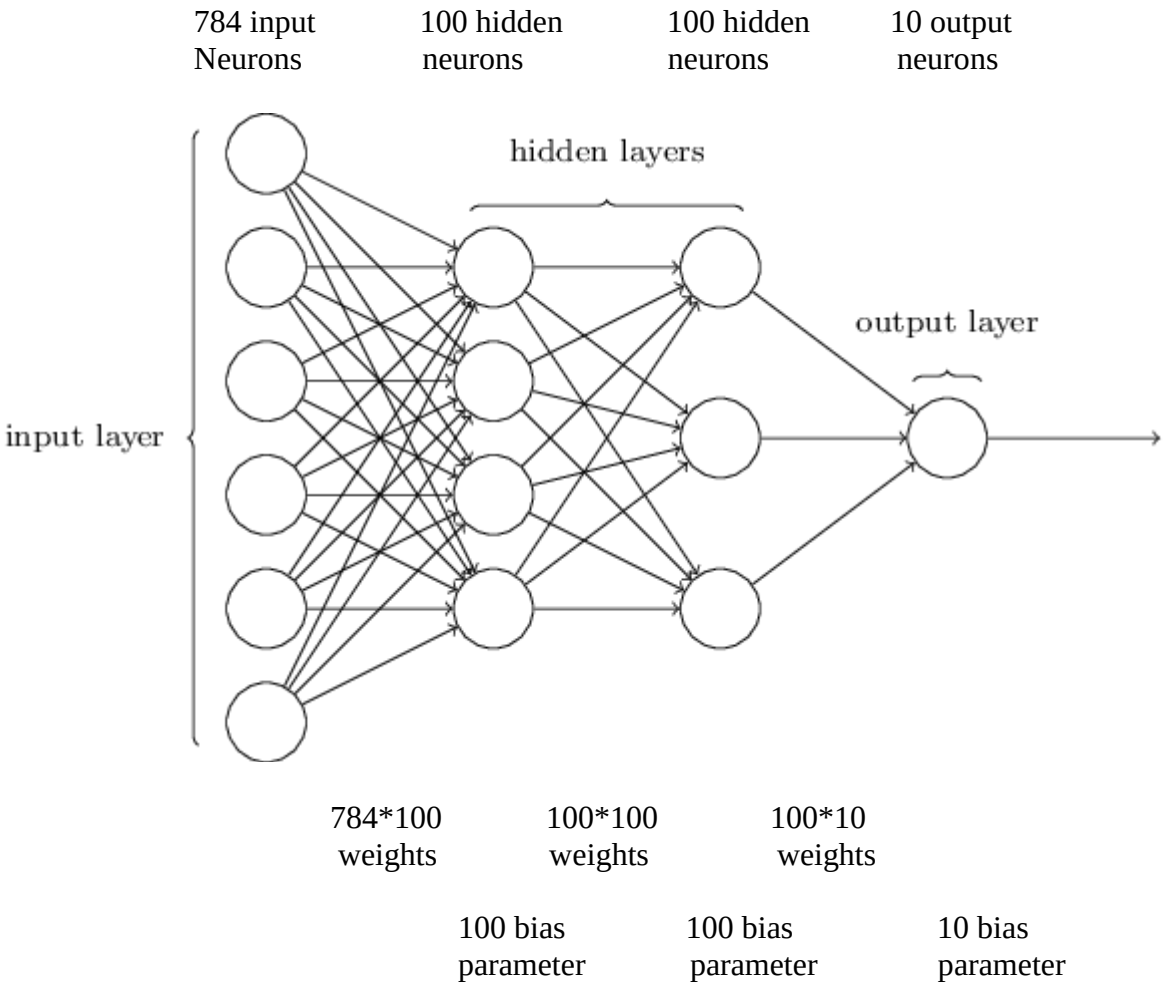
Exercise 3 (Number of parameters in a fully connected and a convolutional network):

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).

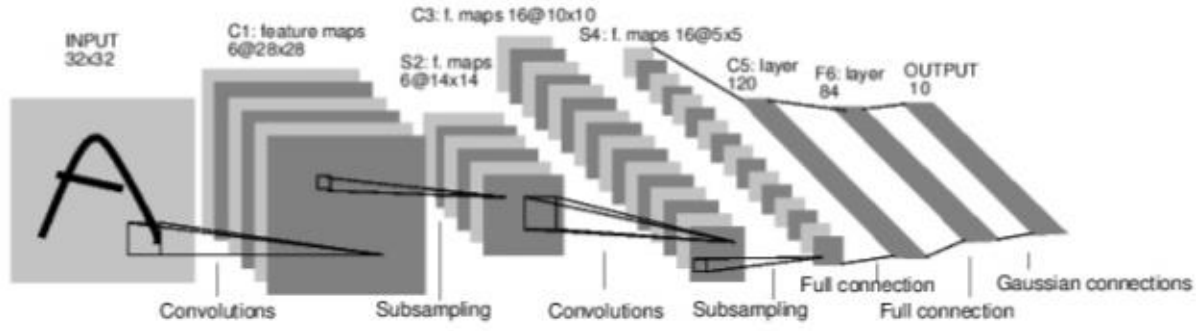
Solution:

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.



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) (optional) 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) (optional) 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

Exercise 4 (Overfitting – CNN on CIFAR-10):

The jupyter notebook provides code to train a network to classify images of the CIFAR-10 dataset (https://www.cs.toronto.edu/~kriz/cifar.html). Training is performed on a subset of 10000 images from the dataset. Run the code and discuss the results. Then, explore possibilities to counteract the observed behavior.

Remarks:

1) If you perform the experiments in Colab, you may explicitly select GPU execution: Select the "Edit" tab in Colab, then choose "Notebook settings", and then select "GPU" as hardware accelerator. See also the accompanying documentation in Colab.

2) For Tensorboard to work, you have to explicitly activate cookies from all third parties in your browser. Furthermore, it has been found that Tensorboard could not be run successfully in all configuration or browsers. If it doesn't work in your configuration (e.g. error "Google 403. That's an error. That's all we know.") please outcomment the lines referring to Tensorboard.

3) Experiments with data augmentation may substantially increase runtime! Further information on data augmentation in Keras can be found at https://keras.io/preprocessing/image/.

As before, you can visualize your different runs via Tensorboard:

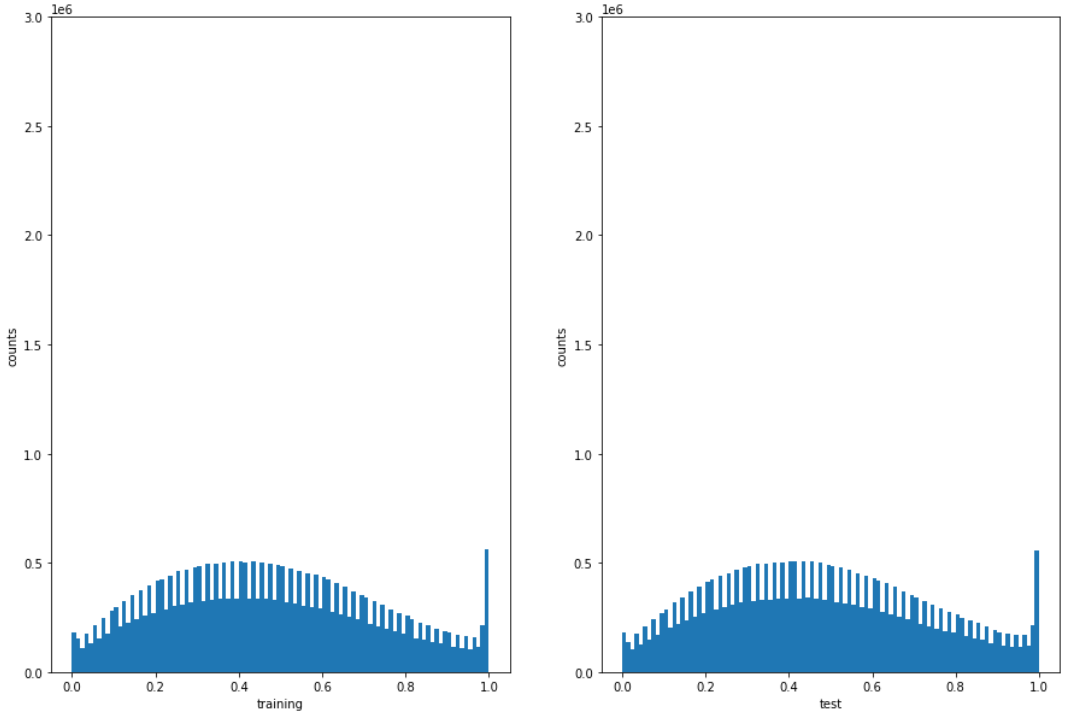
```
%load_ext tensorboard
%tensorboard --logdir {logs_base_dir}
```

If you want to download the created logs you can create a zip file with the following command. Remember that any saved data in Colab will be lost after a reboot.

```
# create zip from logs
!zip -r ./logs.zip ./logs
```

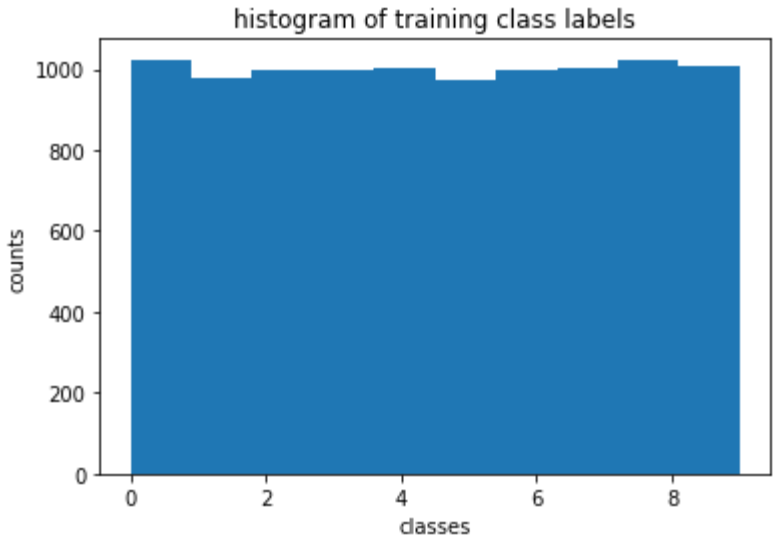
Solution:

The provided example images give a first impression on the quality and difficulty of the data set (in any machine learning problem, you should first get an idea of the data at hand!). The next step is to calculate some statistics on the training and (if available) test data, for example about the input values to check for potential mismatch between training and test, data heterogeneity etc.:



These plots show that the histogram of (flattened) input values is quite similar between training and test data sets, providing no cues for potential data mismatch.

The next test checks whether the 10 classes are actually contained in the training data:



It can be concluded that all 10 classes (numbered from 0 to 9) are contained at nearly equal fractions in the training data.

The baseline configuration provided in the Jupyter notebook can be summarized as follows:

- 2 blocks, each consisting of a convolutional layer with 32 feature maps, followed by batch normalization and ELU activation (filter size 3 x 3)
- Max pooling (stride 2 x 2)
- 2 blocks, each consisting of a convolutional layer with 64 feature maps, followed by batch normalization and ELU activation (filter size 3 x 3)
- Max pooling (stride 2 x 2)
- 2 blocks, each consisting of a convolutional layer with 128 feature maps, followed by batch normalization and ELU activation (filter size 3 x 3)
- Max pooling (stride 2 x 2)
- One fully connected layer with 2048 units followed by a softmax activation function (10 output classes for CIFAR-10)

This network topology can be denoted [32, 64, 128], [2048].

Further settings are:

- Training using the RMSProp optimizer, learning rate 0.001 and decay 1e-6,
- Log-likelihood (categorical cross-entropy) loss function,
- No dropout,
- No batch normalization,
- No regularization,
- No additive Gaussian noise,
- No data augmentation,
- 125 epochs,
- Batch size 64,

Note that the activation function has been set to "ELU" (exponential linear unit), which is defined as:

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ a(e^x - 1) & \text{otherwise} \end{cases}$$

The RMSProp algorithm has been chosen since it seemed to provide best results regarding the training and validation loss and accuracy in initial experiments (in this way, also other hyperparameters can be set).

Please also note the implementation of the **`fit`** method used in this example specifies that validation is performed on the test data. In theory, both corpora should be separate: The validation data are used to optimize hyperparameters and define the early stopping point (if applicable); the final system performance is then determined on independent test data. The settings used here have been chosen for simplicity.

The performance of the baseline system ("configuration A": training on 10000 images, no dropout at all, no additive Gaussian noise, no weight regularization, no data augmentation, with batch normalization) is summarized as follows:

Final training loss: 0.000000
Final training accuracy: 1.000000
Final test loss: 3.182173 (recall that here the test data are identical to the validation data)
Final test accuracy: 0.708000 (dito)

Note that without batch normalization, the validation loss was found to monotonically increase; therefore, all configurations use batch normalization and the batch size (64) was chosen sufficiently "large".

From the figures above it can be seen that the training loss decreases monotonically with increasing number of training epochs, while the validation (= test) loss first decreases and then increases again. Both the training and validation accuracy increase monotonically (except for fluctuations) with increasing number of training iterations, and the validation accuracy is much smaller than the training accuracy. This indicates **overfitting**, since (a) the model performs considerably worse on the validation than on the training data and (b) the system modifications as a result of training beyond about 10 epochs still continue to reduce the training loss, but increase the validation loss (at nearly constant or only slightly increasing validation accuracy). The relation between loss and accuracy is discussed further below.

The following experiments try to investigate the influence of the following system configurations on overfitting:

- number of training examples (10000 versus 50000 training images),
- dropout (in fully connected layers, yes or no),
- additive Gaussian noise (0, i.e. no Gaussian noise, or 0.2, i.e. with Gaussian noise),
- Weight regularization (0, i.e. no weight regularization, or L2 weight 0.0001)

Important remarks:

Dropout:
Dropout is usually being applied to fully connected layers. Due to parameter sharing, dropout in convolutional layers has a different effect than in fully connected layers (and the name "dropout" is misleading in this case). For this reason, dropout is often even discouraged for this reason (see also https://towardsdatascience.com/dropout-on-convolutional-layers-is-weird-5c6ab14f19b2). Since the effect of dropout seems to be the addition of (Bernoulli) noise to the inputs of a layer, the Jupyter notebook contains the addition of noise (for simplicity Gaussian

noise) as an alternative to dropout. Experimentally, dropout in the convolutional layers was found to improve results on this task (therefore it is included in the code), but generally it should be used with care (or even avoided) in convolutional layers.

Batch normalization:
In fully connected layers, batch normalization is normally being applied to the activations, i.e., after the activation function. In convolutional layers, batch normalization is normally being applied to the postsynaptic potentials, i.e., before the activation function. This has been realized in the Jupyter notebook (by separating the activation function from the convolutional layers); experimentally, the influence of the location of batch normalization on the system performance has been found to be rather small in this task.
When batch normalization is used, the batch size should not be too small; moreover, the batch size directly influences the normalization of the input statistics of each layer and thus might also affect model performance (in addition to the estimation of the gradient on each batch). Therefore, the optimal batch size with batch normalization can be different than that without batch normalization.

Weight regularization:
As an example for weight regularization, L2 regularization has been chosen (with weight 0.0001).

Individually varying the above-mentioned settings, the following **system configurations** (numbered from "A" to "L") have been defined (including the baseline configuration):

A: 10000 training images, no dropout at all, no additive Gaussian noise, no weight regularization, no data augmentation, with batch normalization

B: 10000 training images, dropout 0.5 in fully connected layers (no dropout in convolutional layers), no additive Gaussian noise, no weight regularization, no data augmentation, with batch normalization

C: 10000 training images, no dropout at all, with additive Gaussian noise, no weight regularization, no data augmentation, with batch normalization

D: 10000 training images, no dropout at all, no additive Gaussian noise, L2-regularization with weight 0.0001, no data augmentation, with batch normalization

E: 10000 training images, dropout 0.5 in fully connected layers (no dropout in convolutional layers), with additive Gaussian noise, L2-regularization with weight 0.0001, no data augmentation, with batch normalization

F: 50000 training images, no dropout at all, no additive Gaussian noise, no weight regularization, no data augmentation, with batch normalization

G: 50000 training images, dropout 0.5 in fully connected layers (no dropout in convolutional layers), no additive Gaussian noise, no weight regularization, no data augmentation, with batch normalization

H: 50000 training images, no dropout at all, with additive Gaussian noise, no weight regularization, no data augmentation, with batch normalization

I: 50000 training images, no dropout at all, no additive Gaussian noise, L2-regularization with weight 0.0001, no data augmentation, with batch normalization

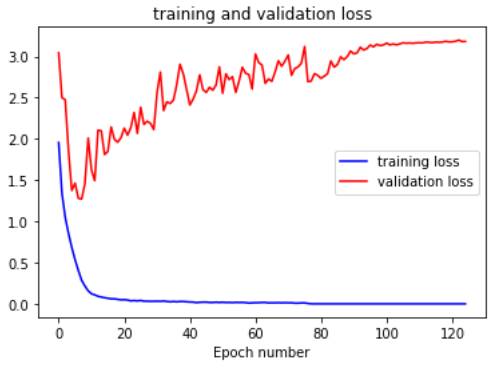
J: 50000 training images, dropout 0.5 in fully connected layers (no dropout in convolutional layers, no additive Gaussian noise, L2-regularization with weight 0.0001, no data augmentation, with batch normalization

K: 50000 training images, dropout 0.5 in fully connected layers (no dropout in convolutional layers), with additive Gaussian noise, L2-regularization with weight 0.0001, no data augmentation, with batch normalization

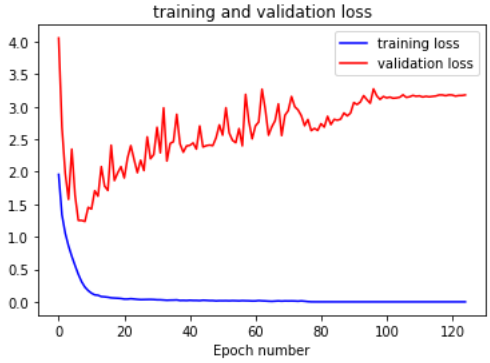
L: 50000 training images, dropout 0.5 in fully connected layers plus dropout in convolutional layers, with additive Gaussian noise, L2-regularization with weight 0.0001, no data augmentation, with batch normalization

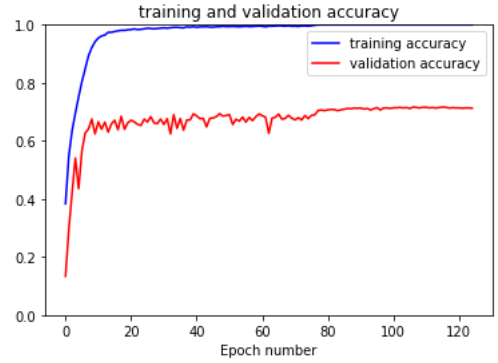
The results are summarized as follows (including the baseline configuration "A" from above):

A: 10000 training images, no dropout at all, no additive Gaussian noise, no weight regularization, no data augmentation, with batch normalization

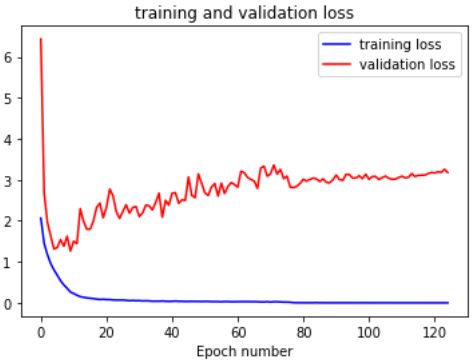


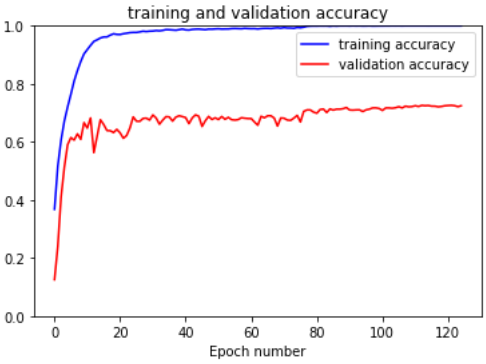
B: 10000 training images, dropout 0.5 in fully connected layers (no dropout in convolutional layers), no additive Gaussian noise, no weight regularization, no data augmentation, with batch normalization



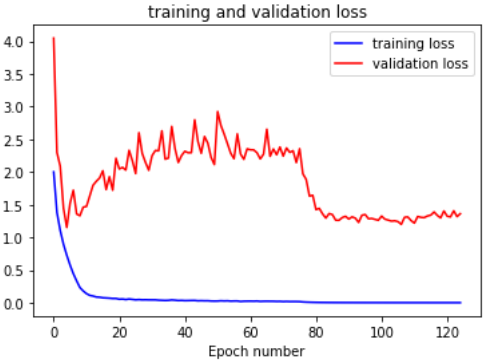


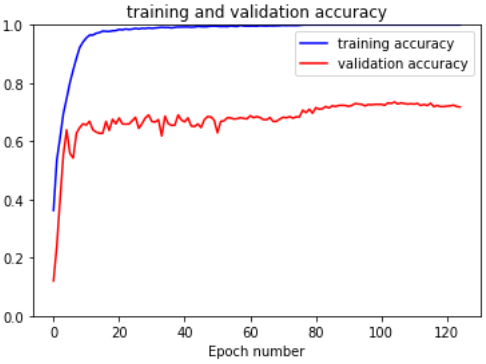
C: 10000 training images, no dropout at all, additive Gaussian noise, no weight regularization, no data augmentation, with batch normalization



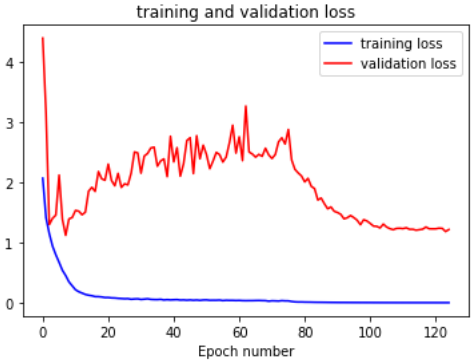


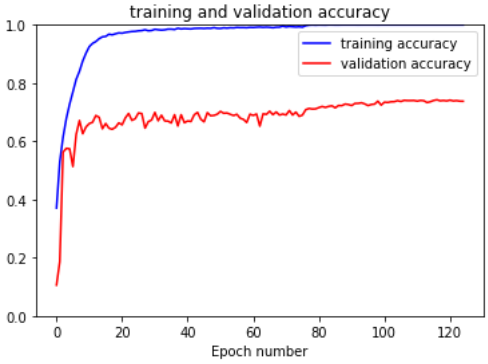
D: 10000 training images, no dropout at all, no additive Gaussian noise, L2-regularization with weight 0.0001, no data augmentation, with batch normalization



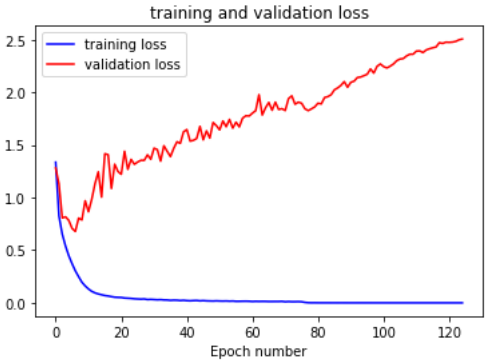


E: 10000 training images, dropout 0.5 in fully connected layers (no dropout in convolutional layers), with additive Gaussian noise, L2-regularization with weight 0.0001, no data augmentation, with batch normalization

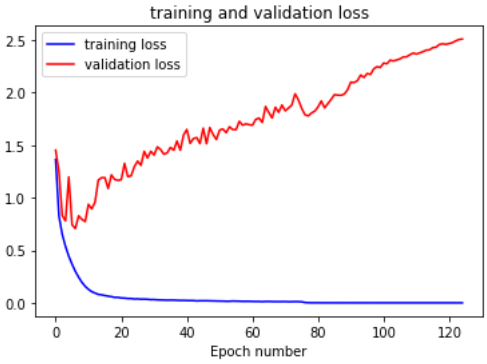


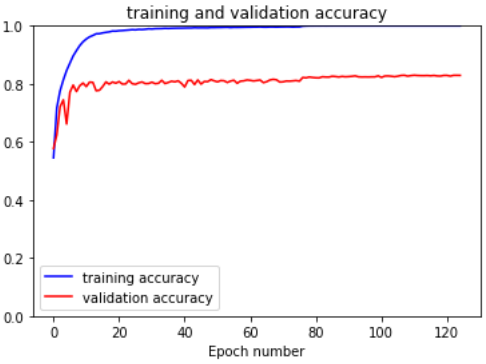


F: 50000 training images, no dropout at all, no additive Gaussian noise, no weight regularization, no data augmentation, with batch normalization

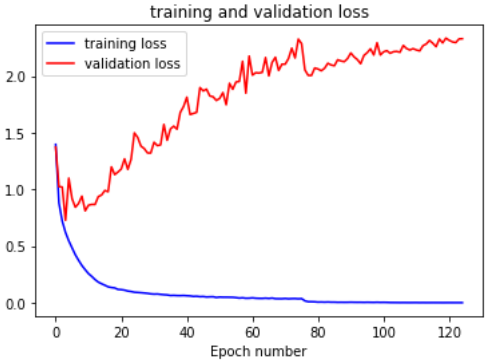


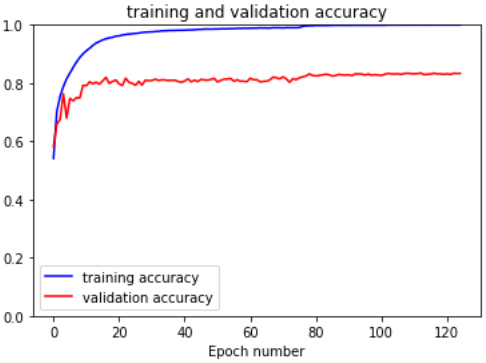
G: 50000 training images, dropout 0.5 in fully connected layers (no dropout in convolutional layers), no additive Gaussian noise, no weight regularization, no data augmentation, with batch normalization





H: 50000 training images, no dropout at all, with additive Gaussian noise, no weight regularization, no data augmentation, with batch normalization

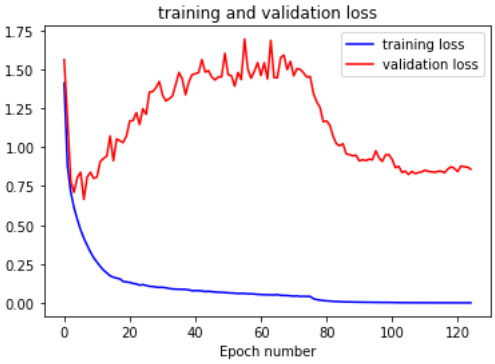




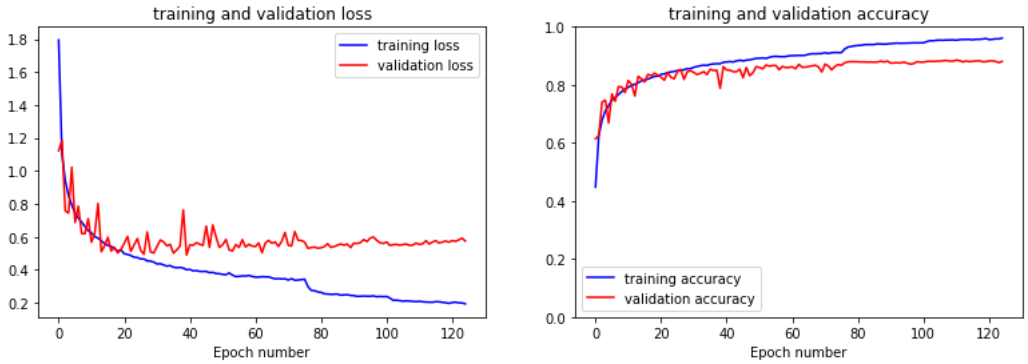
I: 50000 training images, no dropout at all, no additive Gaussian noise, L2-regularization with weight 0.0001, no data augmentation, with batch normalization

J: 50000 training images, dropout 0.5 in fully connected layers (no dropout in convolutional layers, no additive Gaussian noise, L2-regularization with weight 0.0001, no data augmentation, with batch normalization

K: 50000 training images, dropout 0.5 in fully connected layers (no dropout in convolutional layers), with additive Gaussian noise, L2-regularization with weight 0.0001, no data augmentation, with batch normalization



L: 50000 training images, dropout 0.5 in fully connected layers plus dropout in convolutional layers, with additive Gaussian noise, L2-regularization with weight 0.0001, no data augmentation, with batch normalization



The final errors are summarized in the following table (recall that here the test data are identical to the validation data):

N: Number of training samples
Drop.: Dropout (only applied to fully connected layers, if not stated otherwise)
W. Reg.: Weight regularization
Gauss.: applying additive Gaussian noise after max pooling layers

Conf.	N	Drop.	Gauss.	W. Reg.	Final training		Final test	
					loss	accuracy	loss	accuracy
A	10k	0	0	0	0.000000	1.000000	3.182173	0.708000
B	10k	1	0	0	0.000000	1.000000	3.180658	0.712800
C	10k	0	1	0	0.000003	1.000000	3.178606	0.724500
D	10k	0	0	1	0.000196	1.000000	1.361189	0.718000
E	10k	1	1	1	0.000586	1.000000	1.215237	0.737700
F	50k	0	0	0	0.000000	1.000000	2.506742	0.828500
G	50k	1	0	0	0.000000	1.000000	2.510752	0.829300
H	50k	0	1	0	0.000242	0.999920	2.329829	0.833300
I	50k	0	0	1	0.000424	1.000000	0.939606	0.830900
J	50k	1	0	1	0.000426	1.000000	0.898669	0.833500
K	50k	1	1	1	0.000885	1.000000	0.858634	0.836100
L	50k	1+conv	1	1	0.194032	0.961420	0.575020	0.881000

Interpretation:

The largest effect is clearly observed by increasing the amount of training material (50000 instead of 10000 training images). This considerably reduces the gap between the observed training and test

L2 weight regularization prevents the validation loss to further increase and leads to its reduction after 70 to 80 epochs. The effect of the other settings on overfitting is rather small.

When combining settings (dropout, Gaussian noise and weight regularization), additional (minimal) perfomance improvements are observed (which are apparently not statistically significant). Interestingly, when applying dropout also to convolutional layers (see the remark

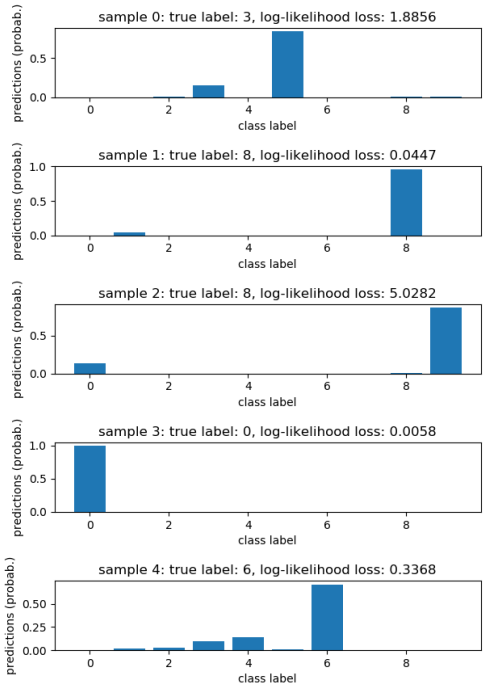
above), further performance improvements can be observed and the increase of the validation loss is (nearly) removed (see configuration L above).
Data augmentation required too much time in my configuration (20 minutes per epoch), so I could not test its performance.

Why can the validation loss increase while the validation accuracy remains more or less constant?
To investigate this effect, the first 5 test samples have been considered and the model predictions (as probabilities, i.e. the softmax output for the 10 classes) have been recorded after 7 and 125 epochs, respectively, for two different training runs (on 10000 and 50000 training samples), using the baseline configuration. Further, the log-likelihood loss has been calculated as the negative logarithm of the prediction for the correct class.

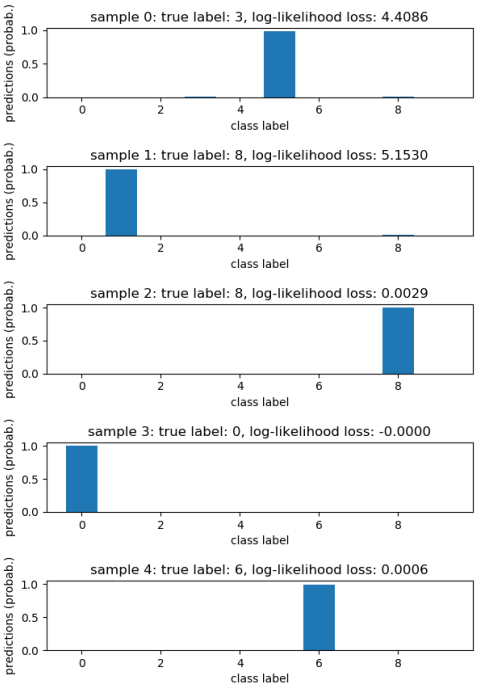
Here are the results:

i) Training on 10000 samples:

predictions (probabilities) after 7 epochs:



predictions (probabilities) after 125 epochs:



Interpretation (loss: log-likelihood loss):

Sample no. / true label	Prediction after 7 epochs	Prediction after 125 epochs
0 (true label = 3)	Error; loss = 1.8856	Error; loss = 4.4086
1 (true label = 8)	Correct; loss = 0.0447	Error; loss = 5.1530
2 (true label = 8)	Error; loss = 5.0282	Correct; loss = 0.0029
3 (true label = 0)	Correct; loss = 0.0058	Correct; loss = 0.0000
4 (true label = 6)	Correct; loss = 0.3368	Correct; loss = 0.0006

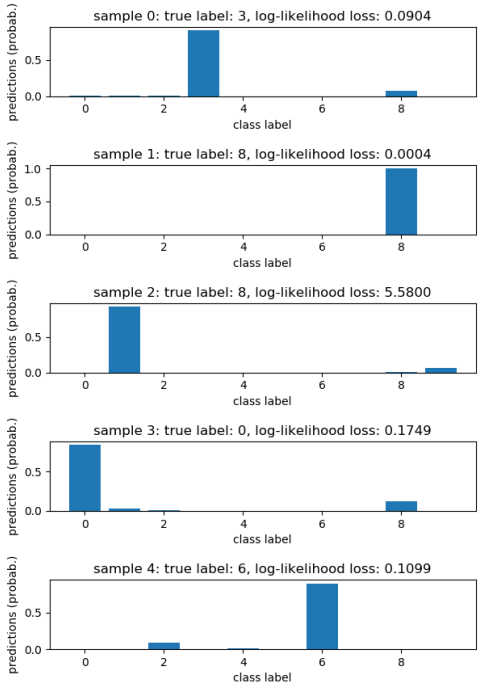
Loss after 7 epochs: 1.8856 + 0.0447 + 5.0282 + 0.0058 + 0.3368 = 7.3011
Loss after 125 epochs: 4.4086 + 5.1530 + 0.0029 + 0.0000 + 0.0006 = 9.5651

Note that the loss for correct samples is rather small, while for incorrect samples it is substantially larger (and larger than 1).

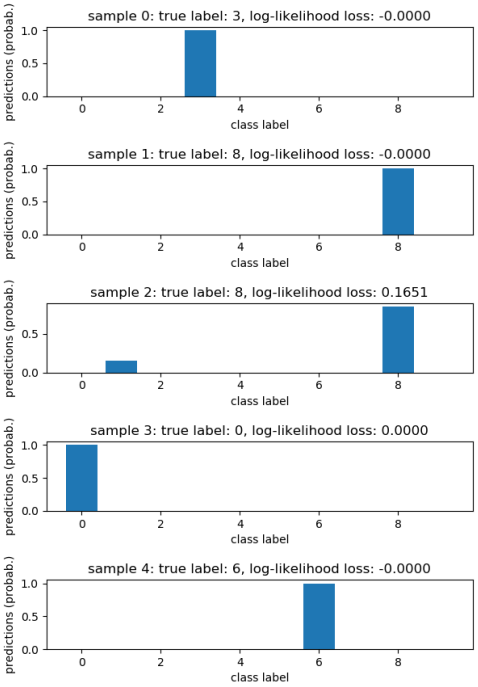
Although this simple calculation should be treated with care (since only 5 examples are involved), it shows how the error rate (in this case two errors after 7 and 125 epochs) can remain constant, while the loss is increasing. It also shows the basic mechanism for this behavior (see sample 0): For a wrong example, after a longer training time the model gets even more confident of its incorrect predictions. Thus, the probability for the correct class decreases even further for these incorrect examples and eventually may approach 0, so that the negative logarithm increases to (very) large values, contributing to a (much) larger loss.

ii) Training on 50000 samples:

predictions (probabilities) after 7 epochs:



predictions (probabilities) after 125 epochs:



Interpretation (loss: log-likelihood loss):

Sample no. / true label	Prediction after 7 epochs	Prediction after 125 epochs
0 (true label = 3)	Correct; loss = 0.0904	Correct; loss = 0.0000
1 (true label = 8)	Correct; loss = 0.0004	Correct; loss = 0.0000
2 (true label = 8)	Error; loss = 5.5800	Correct; loss = 0.1651
3 (true label = 0)	Correct; loss = 0.1749	Correct; loss = 0.0000
4 (true label = 6)	Correct; loss = 0.1099	Correct; loss = 0.0000

Loss after 7 epochs: 0.0904 + 0.0004 + 5.5800 + 0.1749 + 0.1099 = 5.9556
Loss after 125 epochs: 0.0000 + 0.0000 + 0.1651 + 0.0000 + 0.0000 = 0.1651

When trained on 50000 training examples, the model makes fewer errors (even after 7 epochs only). More training epochs reduce the error rate and the loss even further. This is mainly due to the fact that the model gets more confident of its correct predictions, increasing their probabilities and reducing the loss.