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

Neural Networks and Deep Learning – Summer Term 2020

Exercise sheet 4

Submission due: Wednesday, June 03, 13:15 sharp

Remark:

Some of the following experiments are performed on the MNIST data set. The data are contained in the file **`mnist.pkl.gz`** and are loaded using the module **`mnist_loader.py`**. Note that the data are normalized to the range [0, 1].

Generally, the data are divided into a training set (first 60000 samples) and a test set (last 10000 samples). Here, however, we additionally use a validation set of 10000 samples, so we use the first 50000 data samples for training, the next 10000 data samples for validation and the last 10000 samples for testing.

In order to verify the distribution of patterns to classes, the following python code can be executed after loading the data, i.e. after defining the variables **`training_target`**, **`validation_target`** and **`test_target`**:

```
for i in range(10):
    print("%d: %d" % (i, (training_target==i).sum()))

for i in range(10):
    print("%d: %d" % (i, (validation_target==i).sum()))

for i in range(10):
    print("%d: %d" % (i, (test_target==i).sum()))
```

The output is summarized in the following table:

	0	1	2	3	4	5	6	7	8	9
train	4932	5678	4968	5101	4859	4506	4951	5175	4842	4988
valid.	991	1064	990	1030	983	915	967	1090	1009	961
test	980	1135	1032	1010	982	892	958	1028	974	1009

Regarding this training / validation / test split, we see that the data are not fully homogeneously distributed over the splits; however, each digit is sufficiently well represented.

Exercise 1 (Learning in neural networks):

a) Explain the following terms related to neural networks as short and precise as possible:

Solution:

- Loss function
  - In mathematical optimization, statistics, machine learning etc. a loss function (or cost function) is a function that maps an event or values of one or more variables (in our case mostly the outputs of a neural network or learning machine for a set of input samples, the training set, compared to the target values) onto a real number intuitively representing some "loss" or "cost" associated with the event [Wikipedia] (e.g., a true or wrong prediction of the target output for a particular input sample by the learning machine). The loss function is often used for parameter estimation and in this context calculated on the training set, the intuition being that the parameters of the learning machine should be estimated in such a way as to minimize the loss on a set of representative examples (the training set). The hope is that in this way, the predictions on new examples are also as accurate as possible. However, the loss function may also involve – apart from some quantity measuring a prediction error, which is computed by a comparison of the output of the learning machine with the target output – a regularization term (see below), which guides the parameter estimation in parameter space and does not necessarily involve the target values (in the case involving regularization, I call the loss function a "cost function", but this distinction is not common in literature). Note that principally the loss function can also be defined and computed on a "development corpus", e.g. to optimize the meta-parameters of the learning algorithm (e.g. learning rate, network topology). An example of a loss function is the mean (or sum) squared error loss, often used for regression problems. (Partly based on "Wikipedia).

- Stochastic gradient descent
  - Stochastic gradient descent is a variant, namely stochastic approximation, of the gradient descent optimization method to minimize a (differentiable) objective function (loss function) that is written as a sum of contributions from individual training samples. In the "regular" gradient descent algorithm, the gradient of the (total) loss function is used to compute the parameter update in each iteration, involving contributions from *all* training samples ("batch learning"). Stochastic gradient descent approximates this "total" gradient by the gradient computed against a *single, randomly chosen ("stochastic") training example* ("online learning"). This often converges faster than the "regular" (i.e., batch) gradient descent method. Practically, often a randomly chosen *subset* of training examples (i.e., more than a single training example, but much less then all training samples, called "mini-batch"; see below) is used to compute the gradient.

- Mini-batch
  - "Small" subset of the training data used in an (iterative) learning algorithm, e.g. (stochastic) gradient descent; compromise between using the full training set to calculate the gradient of the loss function for parameter update ("batch learning") and using a single training example to calculate the gradient for

parameter update ("online learning"). The size of the mini-batch is often chosen to match hardware constraints (e.g. the number of cores, such that the gradient can be calculated in parallel for all elements of the mini-batch, memory constraints) and algorithmic requirements (speed and quality of convergence). If memory permits, often the values 8 or 16 are used as size of the mini-batch (or less in case of memory constraints). Note that varying the size of the mini-batch may influence the optimal value of the learning rate.

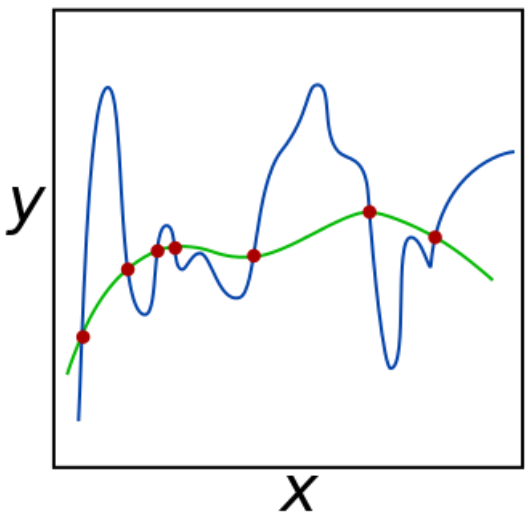
- Regularization
  - In mathematics, statistics, machine learning etc., regularization is a process of introducing additional information in order to solve an ill-posed problem or to prevent overfitting [Wikipedia]. The general goal in machine learning is to minimize the generalization error. Since the generalization error cannot be computed, any learning machine (e.g., neural network) is constructed ("trained" or "learned") on a finite, empirical set of examples (the "training set"). This is, however, an underdetermined problem since we try to infer a function of any x given only some examples x_1, x_2, ..., x_n (i.e. many solutions are possible which are consistent with the training set, but which may predict examples not contained in the training set differently). In particular, a learning machine trained on the training set may overfit to the training data, i.e. perform well on the training set, but worse on other samples. Regularization aims to "guide" the learning process towards solutions improving the generalization error, by incorporating additional ("prior") knowledge about the form of the solution. Technically, regularization can be realized by adding a regularization term $R(f)$ to a loss function $L(f(x_i), y_i)$ measuring the loss of predicting an output $f(x_i)$ to an input sample x_i, while the true output is y_i (see above):

    $$J(f, \{x_i\}, \{y_i\}) = \sum_{i=1}^{N} L(f(x_i), y_i) + \lambda R(f)$$

    Here, λ is a parameter controlling the importance of the regularization term. $R(f)$ is typically chosen to impose a penalty on the complexity of f. In case of a neural network specified by a parameter vector $\mathbf{w}$, the prior knowledge can be imposed by a penalty on the norm of $\mathbf{w}$ ("weight regularization"). For example, one may impose the prior knowledge that the weights shall be "small". This can be realized by using the L1 norm ("L1 regularization") or the L2 norm ("L2 regularization"):

    $$R(f) = \Omega(\mathbf{w}) = \sum_{i=1}^{n} |w_i| \quad \text{(L1 regularization)}$$

    $$R(f) = \Omega(\mathbf{w}) = \sum_{i=1}^{n} w_i^2 \quad \text{(L2 regularization)}$$



The green and blue functions both incur zero loss on the given data points. A learned model can be induced to prefer the green function, which may generalize better to more points drawn from the underlying unknown distribution, by adjusting λ, the weight of the regularization term. (Source of text and figure: Wikipedia)

- Dropout
  - Dropout is a regularization technique to reduce the risk of overfitting, mostly applied in fully connected layers of neural networks. It consists in randomly removing a (non-output) unit in a neural network with probability $1- p$ (i.e. keeping it in the network with probability p). Whether a unit is removed from the network or not, is determined independently for each unit in the network, leading to a certain network configuration (with a randomly selected set of neurons missing). For each training sample, the decision to remove a unit or not is drawn again randomly for each unit (independent of the input or the weights). For the weights, the current estimate of the learning algorithm is used (i.e., the learning algorithm proceeds as usual, but each training sample sees a different, "thinned" network).

    Technically, to remove a unit can be realized by setting the output of that unit to zero (this works in multi-layer perceptrons computing a scalar product of the activation – which is set to zero – and the weights, but it doesn't work in radial basis function networks which compute a distance between the activation and the weights). Since by this dropout process the receptive field and thus the input of each hidden unit may be different from training sample to training sample, the risk of co-adaptation of the hidden units is reduced (i.e., the hidden units cannot learn the same, but must learn different aspects of the input data, thus improving generalization).

    Dropout can be interpreted as an approximation to performing model averaging over an ensemble of models. The different realizations of the models generated by the dropout process define the members of the ensemble. However, each test sample is not evaluated by the different members of the ensemble (this would be infeasible since the number of different models is too large); instead, each test sample is only evaluated by a single model, namely the network containing *all* units. In order to keep the expected value of the output from any unit i at test time (i.e., with all units), the weights going out of unit i have to be multiplied after training by the probability of including unit i in training. (Note that an alternative way of keeping the expected value similar in training and test is to scale the weights during training by a factor $1/p$. This is done e.g. in Keras.) The output generated in this way by the full model is regarded as an approximation to the output that would be generated by averaging the outputs of all ensemble elements applied to the input sample.

    For hidden units in fully connected layers of feedforward networks, p is usually set to 0.5, i.e., half of the units are removed independently for each new training sample. For input layers, p is set to 0.8, i.e., only 20% of the units are removed (if at all), since not too much input information should be thrown away (if at all). In convolutional layers, dropout is normally not applied since the number of weights in a filter is rather small (in contrast to fully connected layers).

    Dropout can be realized by artificially inserting a "dropout layer" after the layer to which dropout shall be applied to. This dropout layer realizes the random selection of units the output of which are set to zero.

- Batch normalization
  - Batch normalization is a technique applied in neural networks to reduce the internal covariate shift. The internal covariate shift is the change in the distribution of network activations due to the change in network parameters during training. "It is known that the network training converges faster if its

input are “whitened” (normalized), i.e. linearly transformed to have zero means and unit variances, and decorrelated. As each layer observes the inputs produced by the layers below, it would be advantageous to achieve the same whitening of the inputs of each layer. By whitening the inputs to each layer, we would take a step towards achieving the fixed distributions of inputs that would remove the ill effects of the internal covariate shift” [Ioffe and Szegedy: “Batch normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”]. Intuitively, even if the inputs to the network are normalized, their postsynaptic potentials are not, and may drive e.g. a sigmoid nonlinearity into saturation. The result is a very small error signal during backpropagation, slowing down learning. This effect is amplified from layer to layer in a deep neural network. Moreover, if the next minibatch presented to the network contains data with different statistics than in the current minibatch, the postsynaptic potentials and activations may also exhibit different data statistics at internal nodes of the network (“internal covariate shift”). Learning by backpropagation then has to “calibrate” those data (implicitly) before the learning process can concentrate on discriminating different data samples. This may considerably slow down learning.

As a consequence, batch normalization normalizes the layer outputs for every minibatch for each output (feature) independently and applies an affine transformation to preserve representation of the layer. More precisely, for each input dimension independently, the mean and the variance of the elements of the current minibatch are calculated. Then, the data are normalized by subtracting the mean and afterwards dividing by the square root of the variance (to increase stability a small constant ε is added to the variance). This normalized input is then transformed by an affine transformation with learnable parameters γ (slope) and β (offset), learned independently for each feature dimension, in order to flexibly choose the “operation regime” of the activation function (e.g., to not only work in the linear regime in case of a sigmoid activation function).

- Learning with momentum
  - Learning with momentum is a technique applied in gradient descent learning to improve convergence. For small learning rates, gradient descent based learning converges to a local minimum, but learning may be slow. If the learning rate is too large, the weight update may overshoot (or even diverge), leading to an oscillating (or increasing) loss function. In stochastic gradient descent, the true gradient of the loss function is approximated by the average gradient calculated on a small minibatch of training examples (only one example in case of online learning). Thus, the weight changes will not be perpendicular to the isocontours of the loss function, and take different directions at each weight update step. If the learning rate is small enough, this erratic behaviour of the weight updates will still lead to the local minimum of the loss function. Learning with momentum is a compromise that smoothes the erratic behaviour of the minibatch updates, without slowing down the learning too much. Technically, this is achieved by adding a “momentum” term to the weight update which is parallel to the last weight update. In analogy to the physical “momentum”, this term dampens the weight update if the previous weight update was in opposite direction (oscillation) and accelerates the weight modification if the weight update is roughly in the same direction as the previous weight update. Thus, the weight update is smoothed, stabilizing

learning. However, a new parameter representing the influence of the momentum term has to be estimated, which is not always useful.

- Data augmentation
  - Data augmentation is a technique in machine learning to artificially increase the size of the training set by applying some transformation(s) to existing training data. Huge training sets are essential to reliably train a large learning machine like a deep neural network with a huge number of parameters and to prevent overfitting: "There is no data than more data." However, in certain domains, training data may be scarce; in another scenario, huge amounts of (un-annotated) data may be available, but annotated data (needed for supervised learning) may be scarce. The goal in both cases is to artificially generate more data by applying defined transformations to existing data (such that the annotations might be transformed as well, if annotations are needed). There are many approaches to augmenting data, depending on the type of problem and the type of data. For example, for image classification, geometric transformations like translations, scaling, rotations, crop, flip etc. can be applied to existing input images to create new images. Or small perturbations ("noise") can be applied to images or other data (e.g. for regression problems). These transformations, however, have to be applied with care: They should create variability in the training data that can be expected in the test data. For example, it doesn't make sense to apply large rotations to the objects, if such rotations create object poses that are never to be expected in the test data (or which might be confused with other labels). Applying data augmentation in a smart way (depending on the problem and the type of data), better models can often be trained which are more robust to the variations covered by the data augmentation strategy and which are less prone to overfitting.

- Unsupervised pre-training / supervised fine-tuning
  - As stated above, huge training sets are essential to reliably train a large learning machine like a deep neural network with a huge number of parameters. On the other hand, learning algorithms like backpropagation are supervised and thus need annotated data. Often, it is infeasible to provide such amounts of supervised training data. In such cases (and in the beginning of deep learning), a way out is to use huge amounts of unannotated data to extract, layer by layer, features from the input data. In this way, the individual layers of a deep neural network can be initialised with "meaningful" values obtained on huge data sets ("unsupervised pre-training"). Then, the supervised learning algorithm like backpropagation is applied on a much smaller, annotated set of examples. Starting from the initial values, the parameters of the network can be modified ("fine-tuned") using the set of annotated values ("supervised fine-tuning"). This allows to train deep neural networks in cases where only a limited set of annotated data (but huge amounts of unannotated data) exists. The annotated data set may be further increased by data augmentation.

- Deep learning
  - Deep learning are machine learning methods based on learning hierarchical data representations. This can be realised e.g. by neural networks with a large number of layers, each generating increasingly abstract representations of the input data.

b) Name the most important output activation functions $f(z)$, i.e., activation function of the output neuron(s), together with a corresponding suitable loss function L (in both cases, give the mathematical equation). Indicate whether such a perceptron is used for a classification or a regression task.

Solution:

Overview:

Activation function $f(z)$	Loss function L	Learning task
Linear	Mean squared error	Regression (number of output neurons matches dimensionality of target space)
Logistic	Cross-entropy	Binary classification (probability for class 1) / logistic regression (in interval [0,1]) (single output neuron or depending on problem)
Softmax	Log-Likelihood	Classification (number of output neurons matches number of target classes)

Mathematical equations:

j output units $\hat{y}_j$, $j = 1, ..., m$ (vector $\hat{\boldsymbol{y}}$); p training samples $\boldsymbol{x}^{(\mu)}$ with targets $\boldsymbol{y}^{(\mu)}$, $\mu = 1, ..., p$:

1.) *Linear* activation function: $f(\boldsymbol{z}) = \boldsymbol{z}$

Mean squared error loss: $\mathrm{L}_{MSE}(\mathbf{W}, \boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{2p} \sum_{\mu=1}^{p} \sum_{j=1}^{m} \left(\hat{y}_j \left(\mathbf{W}, \boldsymbol{x}^{(\mu)} \right) - y_j^{(\mu)} \right)^2$

2.) *Logistic* activation function: $f(z) = \frac{1}{1+e^{-z}}$

Cross-entropy loss:
$\mathrm{L}_{CE}(\mathbf{W}, y, \hat{y}) = -\frac{1}{p} \sum_{\mu=1}^{p} \sum_{j=1}^{m} \left[y_j^{(\mu)} \ln \hat{y}_j(\mathbf{W}, \boldsymbol{x}^{(\mu)}) + \left(1 - y_j^{(\mu)}\right) \ln(1 - \hat{y}_j(\mathbf{W}, \boldsymbol{x}^{(\mu)})) \right]$

3.) *Softmax* activation function: $f(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{m} e^{z_k}}$

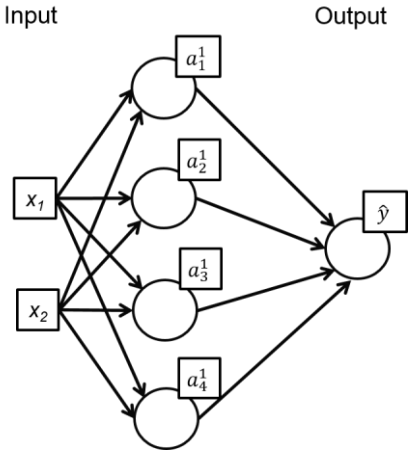
Log-likelihood loss: $\mathrm{L}_{LL}(\mathbf{W}, y, \hat{y}) = -\sum_{\mu=1}^{p} \ln \hat{y}_{j(\mu)}(\mathbf{W}, \boldsymbol{x}^{(\mu)})$

Further activation functions:

- Binary (Heaviside function) or bipolar (sign) for binary classification problems
- Tanh (for applications similar to the logistic activation function)
- ReLU plus variants (ReLU for non-negative regression; also used for hidden neurons)

Exercise 2 (Multi-layer perceptron: Backprogagation, regression problem):

a) Consider the multi-layer perceptron in the following figure:



The activation function at all hidden nodes is ReLU and at the output node linear.

Perform one iteration of plain backpropagation (without momentum, regularization etc.), based on a mini-batch composed of two input samples $\mathbf{x}^{(\mu)}$ with corresponding target values $y^{(\mu)}$, learning rate $\eta = 0.5$ and SSE loss:

$\mathbf{x}^{(1)} = (-1, 1)^T$ with target $y^{(1)} = 1$ and $\mathbf{x}^{(2)} = (2, -1)^T$ with target $y^{(2)} = -1$

The initial weights and biases are given as (t is the iteration index):

$$\mathbf{W}^1(t=0) = \begin{pmatrix} 1 & 2 \\ 0 & -1 \\ -1 & -3 \\ -2 & 2 \end{pmatrix}; \; \mathbf{W}^2(t=0) = (1 \quad 0 \quad -1 \quad 2)$$

$$\boldsymbol{b}^1(t=0) = \begin{pmatrix} -2 \\ 2 \\ 0 \\ -2 \end{pmatrix}; \; b^2(t=0) = -2$$

For the forward path, calculate the postsynaptic potential (PSP), the activations and outputs and insert them into the following table:

Input $\boldsymbol{x} = (x_1, x_2)^T = \boldsymbol{a}^0$	PSP $\boldsymbol{z}^1$	Activation $\boldsymbol{a}^1$	Output $\hat{y} = a^2$
$(-1, 1)^T$			
$(2, -1)^T$			

For the backward path, calculate the updated weights and biases for the hidden and output layer and insert them into the following table:

Weights $\mathbf{W}^1(t=1)$	Bias $\boldsymbol{b}^1(t=1)$	Weights $\mathbf{W}^2(t=1)$	Bias $\boldsymbol{b}^2(t=1)$

Solution:

Calculation of activations and output:

$$\boldsymbol{a}^1 = ReLU\{\mathbf{W}^1 \cdot \boldsymbol{x} + \boldsymbol{b}^1\}$$
$$\hat{y} \;=\; a^2 = \mathbf{W}^2 \cdot \boldsymbol{a}^1 + b^2$$

Input $\boldsymbol{x} = (x_1\,, x_2)^T = (-1,\, 1)^T$:

$$\boldsymbol{a}^1 = ReLU\left\{\begin{pmatrix} 1 & 2 \\ 0 & -1 \\ -1 & -3 \\ -2 & 2 \end{pmatrix} \cdot \begin{pmatrix} -1 \\ 1 \end{pmatrix} + \begin{pmatrix} -2 \\ 2 \\ 0 \\ -2 \end{pmatrix}\right\} = ReLU\left\{\begin{pmatrix} 1 \\ -1 \\ -2 \\ 4 \end{pmatrix} + \begin{pmatrix} -2 \\ 2 \\ 0 \\ -2 \end{pmatrix}\right\} = ReLU\left\{\begin{pmatrix} -1 \\ 1 \\ -2 \\ 2 \end{pmatrix}\right\}$$
$$= \begin{pmatrix} 0 \\ 1 \\ 0 \\ 2 \end{pmatrix}$$

$$\hat{y} \;=\; a^2 = (1 \quad 0 \quad -1 \quad 2) \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \\ 2 \end{pmatrix} - 2 = 2$$

Input $\boldsymbol{x} = (x_1\,, x_2)^T = (\,2, -1)^T$:

$$\boldsymbol{a}^1 = ReLU\left\{\begin{pmatrix} 1 & 2 \\ 0 & -1 \\ -1 & -3 \\ -2 & 2 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ -1 \end{pmatrix} + \begin{pmatrix} -2 \\ 2 \\ 0 \\ -2 \end{pmatrix}\right\} = ReLU\left\{\begin{pmatrix} 0 \\ 1 \\ 1 \\ -6 \end{pmatrix} + \begin{pmatrix} -2 \\ 2 \\ 0 \\ -2 \end{pmatrix}\right\} = ReLU\left\{\begin{pmatrix} -2 \\ 3 \\ 1 \\ -8 \end{pmatrix}\right\}$$
$$= \begin{pmatrix} 0 \\ 3 \\ 1 \\ 0 \end{pmatrix}$$

$$\hat{y} \;=\; a^2 = (1 \quad 0 \quad -1 \quad 2) \cdot \begin{pmatrix} 0 \\ 3 \\ 1 \\ 0 \end{pmatrix} - 2 = -3$$

Input $\boldsymbol{x} = (x_1\,, x_2)^T = \boldsymbol{a}^0$	PSP $\boldsymbol{z}^1$	Activation $\boldsymbol{a}^1$	Output $\hat{y} = a^2$
$(-1,\, 1)^T$	$(-1,\, 1,\, -2,\, 2)^T$	$(0,\, 1,\, 0,\, 2)^T$	2
$(2,\, -1)^T$	$(-2,\, 3,\, 1,\, -8)^T$	$(0,\, 3,\, 1,\, 0)^T$	–3

The SSE loss is given by $\mathrm{L}_{MSE}(\boldsymbol{w}, y, \hat{y}) = \frac{1}{2}\sum_{\mu=1}^{p}\left(\hat{y}(\boldsymbol{w}, \boldsymbol{x}^{(\mu)}) - y^{(\mu)}\right)^2$

Thus we get for the derivative of the loss function with respect to the output for sample μ:
$\nabla_{\hat{y}}\mathrm{L} = \hat{y}\left(\boldsymbol{w}, \boldsymbol{x}^{(\mu)}\right) - y^{(\mu)}$

For the error $\boldsymbol{\Delta}^L$ in the output layer $L = 2$ we have, using $f'(\boldsymbol{z}^2) = 1$, since f is linear:

$\boldsymbol{\Delta}^{2,\mu} = \nabla_{\hat{y}}\mathrm{L} \odot f'(\boldsymbol{z}^2) = \nabla_{\hat{y}}\mathrm{L} = \hat{y}\left(\boldsymbol{w}, \boldsymbol{x}^{(\mu)}\right) - y^{(\mu)}$ (scalar, since there is only 1 output)

Inserting $\hat{y}^{(1)} = 2$, $y^{(1)} = 1$ leads to

$$\boldsymbol{\Delta}^{2,1} = \hat{y}^{(1)} - y^{(1)} = 2 - 1 = 1$$

Inserting $\hat{y}^{(2)} = -3$, $y^{(2)} = -1$ leads to

$$\boldsymbol{\Delta}^{2,2} = \hat{y}^{(2)} - y^{(2)} = -3 + 1 = -2$$

The new weights and biases of the output layer are then given by, using $\eta = 0.5$, $m = 2$, the initial weights $\mathbf{W}^2(t=0) = \mathbf{W}^2$ given above, $\boldsymbol{a}^{1,1\,T}$ =(0, 1, 0, 2), $\boldsymbol{a}^{1,2\,T}$ =(0, 3, 1, 0) and $\boldsymbol{b}^2(t=0) = \boldsymbol{b}^2 = -2$:

$$\begin{aligned}\mathbf{W}^2(t=1) = \mathbf{W}^2(t=0) - \frac{\eta}{m}\sum_{\mu=1}^{m} \boldsymbol{\Delta}^{2,\mu} \cdot \ (\boldsymbol{a}^{1,\mu})^T = (1 \quad 0 \quad -1 \quad 2) \\ -\frac{1}{4}\big(1 \cdot (0,1,0,2) - 2 \cdot (0,3,1,0)\big) \\ = (1 \quad 0 \quad -1 \quad 2) - (0, 0.25, 0, 0.5) + (0, 1.5, 0.5, 0) = (1 \quad 1.25 \quad -0.5 \quad 1.5)\end{aligned}$$

$$\boldsymbol{b}^2(t=1) = \boldsymbol{b}^2(t=0) - \frac{\eta}{m}\sum_{\mu=1}^{m} \boldsymbol{\Delta}^{2,\mu} = -2 - \frac{1}{4}(1-2) = -1.75$$

Backpropagating the error:

$$\boldsymbol{\Delta}^{1,\mu} = ((\mathbf{W}^2)^T \cdot \boldsymbol{\Delta}^{2,\mu}) \odot \ f'(\boldsymbol{z}^{1,\mu})$$

Since f is the ReLU function, we have $f'(\boldsymbol{z}^{1,\mu}) = \Theta[\boldsymbol{z}^{1,\mu}]$ and using $\boldsymbol{z}^{1,1} = (-1, 1, -2, 2)^{\,T}$ and $\boldsymbol{z}^{1,2} = (-2, 3, 1, -8)^{\,T}$ we get:
$f'(\boldsymbol{z}^{1,1}) = (0, 1, 0, 1)^T$ and $f'(\boldsymbol{z}^{1,2}) = (0, 1, 1, 0)^T$.

Thus we obtain:

$$\boldsymbol{\Delta}^{1,1} = ((\mathbf{W}^2)^T \cdot \boldsymbol{\Delta}^{2,1}) \odot \ f'(\boldsymbol{z}^{1,1}) = \left(\begin{pmatrix}1\\0\\-1\\2\end{pmatrix} \cdot 1\right) \odot \begin{pmatrix}0\\1\\0\\1\end{pmatrix} = \begin{pmatrix}0\\0\\0\\2\end{pmatrix}$$

$$\boldsymbol{\Delta}^{1,2} = ((\mathbf{W}^2)^T \cdot \boldsymbol{\Delta}^{2,2}) \odot \ f'(\boldsymbol{z}^{1,2}) = \left(\begin{pmatrix}1\\0\\-1\\2\end{pmatrix} \cdot (-2)\right) \odot \begin{pmatrix}0\\1\\1\\0\end{pmatrix} = \begin{pmatrix}0\\0\\2\\0\end{pmatrix}$$

The updated weights and biases are then given by, using $\mathbf{W}^1(t=0) = \mathbf{W}^1$ and $(\boldsymbol{a}^{0,1})^T = \mathbf{x}^{(1),T} =$ (–1, 1) and $(\boldsymbol{a}^{0,2})^T = \mathbf{x}^{(2),T} = (2, -1)$ as well as $\boldsymbol{b}^1(t=0) = \boldsymbol{b}^1 = \begin{pmatrix}-2\\2\\0\\-2\end{pmatrix}$:

$$\mathbf{W}^1(t=1) = \mathbf{W}^1(t=0) - \frac{\eta}{m}\sum_{\mu=1}^{m}\boldsymbol{\Delta}^{1,\mu}\cdot\ (\boldsymbol{a}^{0,\mu})^T = \begin{pmatrix} 1 & 2 \\ 0 & -1 \\ -1 & -3 \\ -2 & 2 \end{pmatrix} - \frac{1}{4}\left(\begin{pmatrix} 0 \\ 0 \\ 0 \\ 2 \end{pmatrix}\cdot(-1 \quad 1) - \right.$$

$$\left.\frac{1}{4}\left(\begin{pmatrix} 0 \\ 0 \\ 2 \\ 0 \end{pmatrix}\cdot(2 \quad -1)\right) = \begin{pmatrix} 1 & 2 \\ 0 & -1 \\ -1 & -3 \\ -2 & 2 \end{pmatrix} - \frac{1}{4}\begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ -2 & 2 \end{pmatrix} - \frac{1}{4}\begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 4 & -2 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 0 & -1 \\ -1 & -3 \\ -2 & 2 \end{pmatrix} + \right.$$

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0.5 & -0.5 \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ -1 & 0.5 \\ 0 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 2 \\ 0 & -1 \\ -2 & -2.5 \\ -1.5 & 1.5 \end{pmatrix}$$

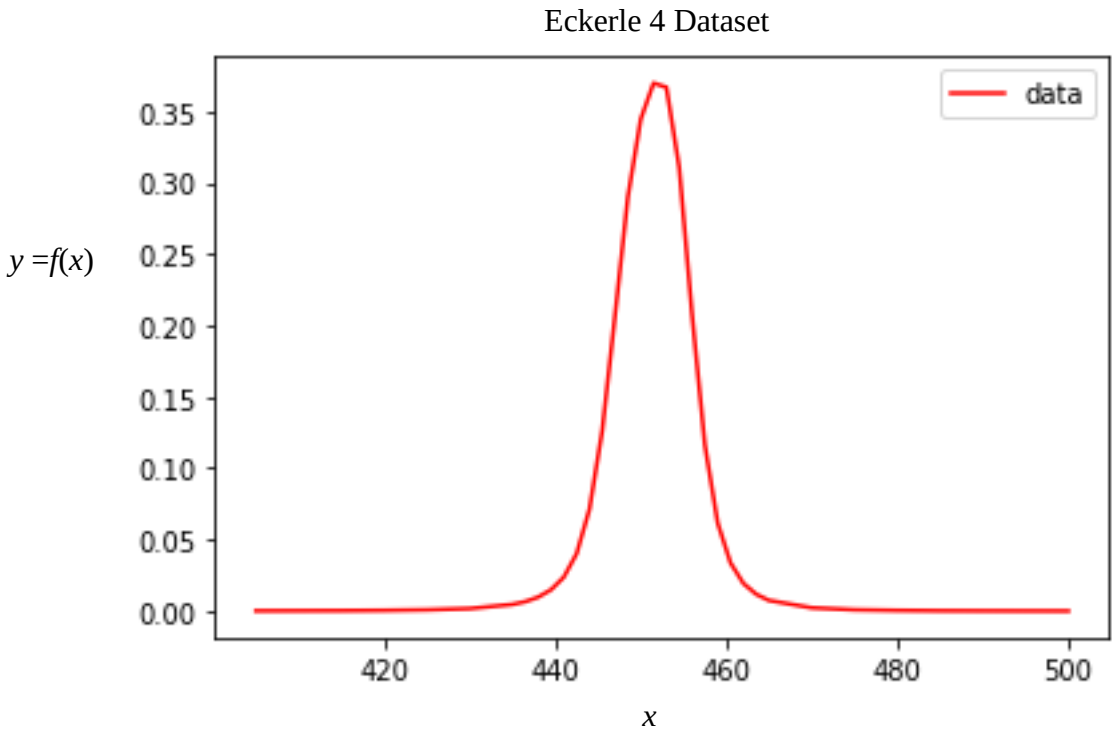
$$\boldsymbol{b}^1(t=1) = \boldsymbol{b}^1(t=0) - \frac{\eta}{m}\sum_{\mu=1}^{m}\boldsymbol{\Delta}^{1,\mu} = \begin{pmatrix} -2 \\ 2 \\ 0 \\ -2 \end{pmatrix} - \frac{1}{4}\begin{pmatrix} 0 \\ 0 \\ 0 \\ 2 \end{pmatrix} - \frac{1}{4}\begin{pmatrix} 0 \\ 0 \\ 2 \\ 0 \end{pmatrix} = \begin{pmatrix} -2 \\ 2 \\ 0 \\ -2 \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0.5 \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ 0.5 \\ 0 \end{pmatrix}$$

$$= \begin{pmatrix} -2 \\ 2 \\ -0.5 \\ -2.5 \end{pmatrix}$$

This leads to the following table of results after one iteration:

Weights $\mathbf{W}^1(t=1)$	Bias $\boldsymbol{b}^1(t=1)$	Weights $\mathbf{W}^2(t=1)$	Bias $\boldsymbol{b}^2(t=1)$
$\begin{pmatrix} 1 & 2 \\ 0 & -1 \\ -2 & -2.5 \\ -1.5 & 1.5 \end{pmatrix}$	$\begin{pmatrix} -2 \\ 2 \\ -0.5 \\ -2.5 \end{pmatrix}$	$(1 \quad 1.25 \quad -0.5 \quad 1.5)$	-1.75

b) The goal of this exercise is to train a multi-layer perceptron to solve a high difficulty level nonlinear regression problem. The data has been generated using an exponential function with the following shape:



This graph corresponds to the values of a dataset that can be downloaded from the Statistical Reference Dataset of the Information Technology Laboratory of the United States on this link: http://www.itl.nist.gov/div898/strd/nls/data/eckerle4.shtml

This dataset is provided in the file **`Eckerle4.csv`**. Note that this dataset is divided into a training and test corpus comprising 60% and 40% of the data samples, respectively. Moreover, the input and output values are normalized to the interval [0, 1]. Basic code to load the dataset and divide it into a training and test corpus, normalizing the data and to apply a multi-layer perceptron is provided in the Jupyter notebook.

Choose a suitable network topology (number of hidden layers and hidden neurons, potentially include dropout, activation function of hidden layers) and use it for the multi-layer perceptron defined in the Jupyter notebook. Set further parameters (learning rate, loss function, optimizer, number of epochs, batch size; see the lines marked with **`# FIX!!!`** in the Jupyter notebook). Try to avoid underfitting and overfitting. Vary the network and parameter configuration in order to achieve a network performance as optimal as possible. For each network configuration, due to the random components in the experiment, perform (at least) 4 different training and evaluation runs and report the mean and standard deviation of the training and evaluation results. Report on your results and conclusions.

(Source of exercise: http://gonzalopla.com/deep-learning-nonlinear-regression)

Note: In the script, data scaling is performed on the whole data set (estimating means, standard deviation, minimum, maximum etc. on the whole data set), before dividing into a training and test set. Thus, knowledge of the test data influences scaling. Normally, scaling would be performed only on training data, and the resulting scaling applied to the test data.

Solution:

Since we deal with a regression problem, we choose the mean squared error ("MSE") loss. As optimizer, we can choose e.g. stochastic gradient descent ("SGD") with learning rate 0.01 (and later test different optimizers or vary the learning rate):

```
opt = SGD(learning_rate = 0.01)
model.compile(loss='mse', optimizer=opt, metrics=["mse"])
```

Note that the Tensorflow Keras functionality has to be imported before using them!

Momentum and in particular Nesterov momentum can be activated e.g. in the stochastic gradient descent optimizer as follows:

```
opt = SGD(learning_rate = 0.01, momentum=0.9, nesterov=True)
```

As activation function of the hidden layers, we choose the ReLU function:

```
activation = 'relu'  # activation of hidden layers
```

To use 3 hidden layers with 128, 64 and 64 neurons, respectively, the following syntax is used:

```
num_hidden = [128, 64, 64]  # for each hidden layer: number of hidden units in form of a pytho
n list
```

Each hidden layer – as well as dropout – is implemented as a Tensorflow Keras layer in a "sequential" model structure (see below):

```
model = Sequential()
```

A first fully connected ("dense") hidden layer – connected to the input – is implemented via

```
model.add(Dense(num_hidden[0], input_dim=num_inputs, activation=activat
ion))
```

Further hidden layers are added in a similar way.

Dropout is invoked using an (optional) additional layer:

```
model.add(Dropout(dropout)) # dropout with dropout fraction
```

Dropout is first disabled (value 0) and may later be switched on to disable 20% of the neurons:

```
dropout = 0.2  # 0 if no dropout, else fraction of dropout units, e.g. 0.2
```

The final layer – connected to the output – is added via

```
model.add(Dense(1))
```

where the argument in the "`Dense`" layer is 1 since we use one output neuron for a (scalar) regression problem.

The number of epochs and the batch size can be set as follows (and may be varied):

```
num_epochs = 256
batch_size = 1
```

Training is invoked with the `fit` method:

```
history = model.fit(X_train, y_train, epochs=num_epochs, batch_size=bat
ch_size, verbose=2)
```

Storing the **history** object allows to recover the training loss and accuracy over all training epochs.
To realize a number of repetitions of training and evaluation, the following code can be used (code lines after **dropout = 0**):

```
num_repetitions = 4
training_error = np.zeros(num_repetitions)  # store training error for the different runs
test_error = np.zeros(num_repetitions)  # store test error for the different runs

for run in range(num_repetitions):
  # repeat training and evaluation num_repetitions times
  # define sequential network structure.
  model = Sequential()

...

  training_error[run] = history.history['loss'][num_epochs-
1] # remember training error
  print("final (mse) training error: %f" % training_error[run])

...
  test_error[run] = model.evaluate(X_test, y_test)[0] # remember test error
  print("test error: %f" % test_error[run])

# final statistics
print("\n")
print("training error: %s" % training_error)
print("... mean +/- std: %f +/ %f" %
      (np.mean(training_error), np.std(training_error, ddof=1)))
print("test error: %s" % test_error)
print("... mean +/- std: %f +/- %f" %
      (np.mean(test_error), np.std(test_error, ddof=1)))
```

The following results have been obtained (if not stated otherwise, the following parameters have been used: no dropout, learning rate 0.01, batch size 2, 256 epochs):

Network configuration	MSE training		MSE test	
	Individual results	mean ± std.	Individual results	mean ± std.
[128, 64, 64], Adam	0.00093958 0.00833894 0.00165187 0.00326627	0.003549 +/- 0.003338	0.00161392 0.00906388 0.00456719 0.00360921	0.004714 +/- 0.003150
[128, 64, 64], Adam with Nesterov's momentum	0.00646319 0.02903142 0.07766966 0.0176913	0.032714 +/- 0.031355	0.02101597 0.01363953 0.15481262 0.02773787	0.054301 +/- 0.067254
[128, 64, 64], SGD	Did not work			
[128, 64, 64], SGD, momentum 0.9, Nesterov = False	0.00183113 0.00872862 0.00381526 0.0084471	0.005706 +/- 0.003427	0.00576381 0.0047414 0.00776304 0.00772263	0.006498 +/- 0.001497

[128, 64, 64], SGD, momentum 0.99, Nesterov = False	Did not work			
[128, 64, 64], SGD, momentum 0.99, Nesterov = True	1.22455542e-03 2.00341060e-06 4.16077748e-02 8.79973825e-03	0.012909 +/- 0.019525	0.00220307 0.00071479 0.0610942 0.0072879	0.017825 +/- 0.028983
[128, 64, 64], AdaGrad	0.04243932 0.04302144 0.04062312 0.04262973	0.042178 +/- 0.001065	0.07727005 0.07931674 0.07554453 0.07854231	0.077668 +/- 0.001648
[128, 64, 64], RMSProp	0.02474733 0.04057043 0.03173536 0.00456203	0.025404 +/- 0.015329	0.08768415 0.01324433 0.02053881 0.00976548	0.032808 +/- 0.036858
[128, 64, 64], Adam, learning rate 0.001	0.00848493 0.00386298 0.00205245 0.00293786	0.004335 +/- 0.002864	0.00570708 0.00797188 0.00199791 0.00284647	0.004631 +/- 0.002735
[128, 64, 64], Adam, learning rate 0.001, dropout 0.2	0.00785327 0.00822205 0.0119867 0.01259177	0.010163 +/- 0.002472	0.00467526 0.00338786 0.00840496 0.00929517	0.006441 +/- 0.002854
[128, 64, 64, 64], Adam, learning rate 0.001	0.00987816 0.00259155 0.00428888 0.01030929	0.006767 +/- 0.003907	0.01623591 0.00896986 0.00674867 0.00522683	0.009295 +/- 0.004876
[128, 64], Adam, learning rate 0.001	0.02207843 0.03519015 0.02986959 0.03315998	0.030075 +/- 0.005764	0.03971228 0.07290985 0.05201817 0.06567006	0.057578 +/- 0.014727
[128, 128, 64], Adam, learning rate 0.001	**0.00409798 0.00187738 0.00123317 0.0043017**	**0.002878 +/- 0.001552**	**0.00427128 0.0015368 0.00365665 0.00279937**	**0.003066 +/- 0.001185**
[128, 128, 64], Adam, learning rate 0.001, dropout 0.2	0.00789899 0.02421053 0.00957814 0.00806093	0.012437 +/- 0.007885	0.01010753 0.04547878 0.00425082 0.00948531	0.017331 +/- 0.018948
[128, 128, 128], Adam, learning rate 0.001	0.00150278 0.00259968 0.00518235 0.01440458	0.005922 +/- 0.005861	0.00233353 0.00236104 0.00407558 0.0039278	0.003174 +/- 0.000957

“Did not work” refers to a situation where the network predicted (nearly) the same value for each input. Note that for some configurations, some training runs were successful, while others were not, i.e., the configuration was found to be “unstable”.

Conclusion: The best network configuration found had 3 hidden layers with 128, 128 and 64 neurons, respectively, and used the Adam optimizer. Dropout was not found to be beneficial. Note however that dropout has been applied to the input layer (which is not always recommended) and not to the output layer. Applying dropout only to the output layer resulted in strong fluctuations of the training loss as a function of the training epochs.The learning rate was 0.001 (which showed less oscillations of the training loss than for a learning rate of 0.01).

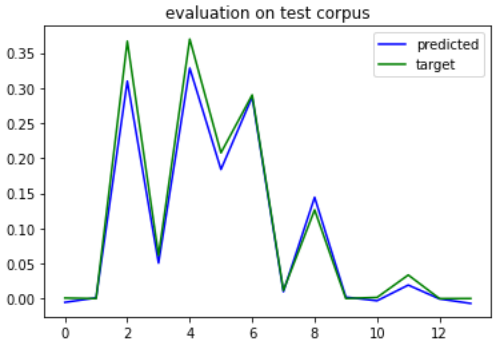
The optimal model configuration found was:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_408 (Dense)            (None, 128)               256
_________________________________________________________________
dense_409 (Dense)            (None, 128)               16512
_________________________________________________________________
dense_410 (Dense)            (None, 64)                8256
_________________________________________________________________
dense_411 (Dense)            (None, 1)                 65
=================================================================
Total params: 25,089
Trainable params: 25,089
Non-trainable params: 0
```

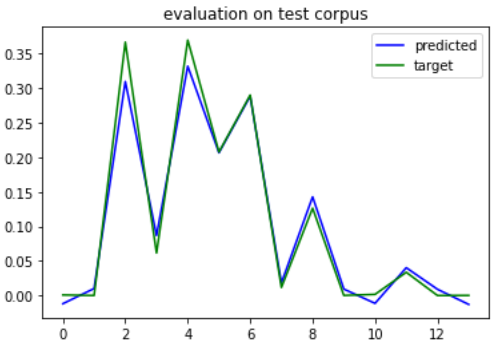
For this configuration, some sample plots are being provided on the next plage. In a second experiment, these results have been obtained (compare with the respective values in the table above in bold):

Network configuration	training		test	
	results	summary	results	summary
[128, 128, 64], Adam, learning rate 0.001	[0.00064292 0.0042784 0.01114108 0.0008195]	0.004220 +/- 0.004908	[0.00328204 0.00336861 0.00425401 0.0017935]	0.003175 +/- 0.001020

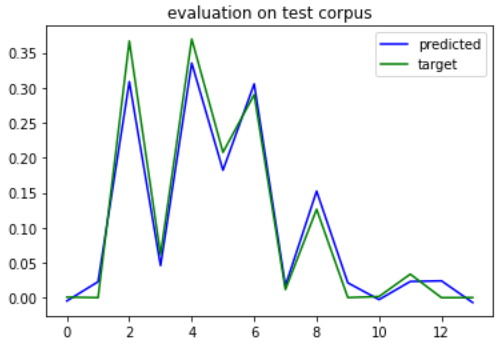
Run 1:



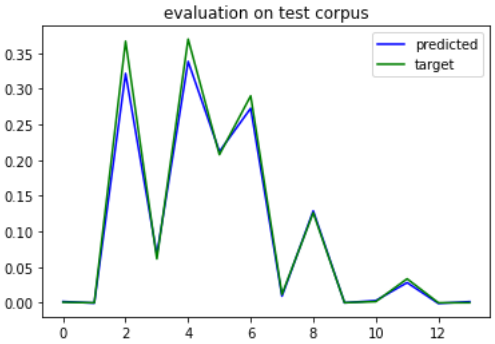
Run 2:



Run 3:



Run 4:



In experiments with a multi-layer perceptron using the scikit-learn library, an alternative optimizer – namely the L-BFGS optimization method – has been found to yield good results. L-BFGS is a second order (quasi-Newton) method, where the inverse of the Hessian matrix is estimated to steer parameter search through the variable space. Instead of the BFGS method, which uses a dense matrix approximation to the Hessian matrix, L-BFGS ("limited memory BFGS") "stores a few vectors that represent the approximation implicitly" [Wikipedia]. It is well suited for optimization problems with a large number of variables, for problems where batch optimization" makes sense, and if the optimization function is convex (e.g. least mean squares), since in those cases second order methods are a good choice.
More details on L-BFGS can be found at
http://aria42.com/blog/2014/12/understanding-lbfgs
https://en.wikipedia.org/wiki/Limited-memory_BFGS

Exercise 3 (Parameters of a multi-layer perceptron – digit recognition):

In the following exercises, we use Tensorflow and Keras to configure, train and apply a multi-layer perceptron to the problem of recognizing handwritten digits (the famous "MNIST" problem). The MNIST data are loaded using a Tensorflow Keras built-in function.

Perform experiments on this pattern recognition problem trying to investigate the influence of a number of parameters on the classification performance. This may refer to

- the learning rate and potentially learning schedule,
- the number of hidden neurons (in a network with a single hidden layer),
- the number of hidden layers as well as applying dropout and / or batch normalization,
- the solver (including momentum),
- the activation function at hidden layers,
- regularization.

The script in the Jupyter notebook can serve as a basis or starting point.

Report your findings and conclusions.

Note: These experiments may require a lot of computation time!

Further investigations and experiments as well as code extensions and modifications are welcome!

Solution:

The MNIST handwritten digit recognition task is a multi-class classification problem with 10 classes; an example input image is provided below. There are two possibilities to handle multi-class classification problems in the Tensorflow Keras framework:

a) Encoding target labels in one-hot representation, using a softmax activation function at the output layer and "categorical cross-entropy loss" (see lab sheet 3, exercise 4),

b) Encoding target labels as single number, using the "sparse categorical cross-entropy loss", set the option "**`from_logits = True`**" and set the activation function of the output layer to "linear".

Here, we follow variant b). Due to the random initialization of the learning algorithms, a first series of experiments has been performed, where each experiment is repeated **`numRepetitions = 4`** times. For each of the initial network configuration, three learning rates are being evaluated. The best learning rate is selected according to the mean accuracy (with respect to the 4 repetitions) on the validation set. However, a single model for this best learning rate is stored for later reference, namely the single experiment (out of the 4 repetitions) that yielded the lowest validation accuracy for the respective learning rate. For this single model, the training loss and accuracy are plotted as a function of the number of epochs, and the final training, validation / test losses and accuracies are reported. The code is written such that the training history and validation / test results for the best model are remembered for a later summary. The first experiments (using a single hidden layer) shall

provide some “feeling” for suitable values for the (constant) learning rate (i.e. no learning schedule yet) and the number of hidden neurons.

Additional variations of the model configuration and training setup are implemented as follows (note to import the corresponding Tensorflow Keras functionality; dropout and momentum have already been explained in the last exercise):

- Batch normalization is implemented via a batch normalization layer (potentially with additional parameters):
  `model.add(BatchNormalization())`

- A learning rate schedule can be invoked by
  `lr_schedule = schedules.ExponentialDecay(initial_learning_rate = learningRate, decay_steps=100000, decay_rate=0.96, staircase=True)`

- The learning rate schedule, momentum and Nesterov momentum can be invoked in some of the optimizers as e.g.
  `opt = SGD(learning_rate=lr_schedule, momentum=momentum, nesterov=nesterov)`
  Note that a momentum parameter is only accepted by the SGD and RMSprop optimizers; Nesterov momentum is only accepted by SGD (in case of Adam use Nadam)

- Regularization is invoked via (**`regularization_weight`** is set to e.g. 0.01):
  `regularizer = tfr.l1(l=regularization_weight)`
  and using the regularizer in the definition of the layers:
  `model.add(Dense(num_hidden[i], activation=activation, kernel_regularizer=regularizer, bias_regularizer=regularizer))`

It is recommended to inspect the summary of the model configuration for control purposes, since this also shows options for further parameter adjustments, e.g.:

```
Now running tests for config {'learningRates': [0.01], 'hiddenLayerSizes':
[100, 100], 'solver': 'SGD', 'activation': 'relu'}

Model configuration:
{'name': 'sequential_6', 'layers': [{'class_name': 'Dense', 'config':
{'name': 'dense_6', 'trainable': True, 'batch_input_shape': (None, 784),
'dtype': 'float32', 'units': 100, 'activation': 'relu', 'use_bias': True,
'kernel_initializer': {'class_name': 'GlorotUniform', 'config': {'seed':
None}}, 'bias_initializer': {'class_name': 'Zeros', 'config': {}},
'kernel_regularizer': {'class_name': 'L1L2', 'config': {'l1':
0.009999999776482582, 'l2': 0.0}}, 'bias_regularizer': {'class_name':
'L1L2', 'config': {'l1': 0.009999999776482582, 'l2': 0.0}},
'activity_regularizer': None, 'kernel_constraint': None, 'bias_constraint':
None}}, {'class_name': 'Dropout', 'config': {'name': 'dropout_1',
'trainable': True, 'dtype': 'float32', 'rate': 0.2, 'noise_shape': None,
'seed': None}}, {'class_name': 'BatchNormalization', 'config': {'name':
'batch_normalization_1', 'trainable': True, 'dtype': 'float32', 'axis':
ListWrapper([1]), 'momentum': 0.99, 'epsilon': 0.001, 'center': True,
'scale': True, 'beta_initializer': {'class_name': 'Zeros', 'config': {}},
'gamma_initializer': {'class_name': 'Ones', 'config': {}},
'moving_mean_initializer': {'class_name': 'Zeros', 'config': {}},
'moving_variance_initializer': {'class_name': 'Ones', 'config': {}},
'beta_regularizer': None, 'gamma_regularizer': None, 'beta_constraint':
None, 'gamma_constraint': None}}, {'class_name': 'Dense', 'config':
```

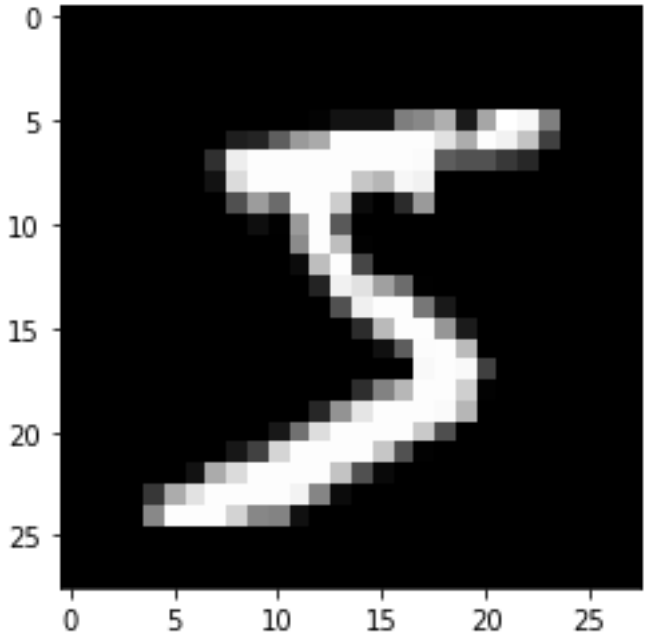
{'name': 'dense_7', 'trainable': True, 'dtype': 'float32', 'units': 100,
'activation': 'relu', 'use_bias': True, 'kernel_initializer':
{'class_name': 'GlorotUniform', 'config': {'seed': None}},
'bias_initializer': {'class_name': 'Zeros', 'config': {}},
'kernel_regularizer': {'class_name': 'L1L2', 'config': {'l1':
0.009999999776482582, 'l2': 0.0}}, 'bias_regularizer': {'class_name':
'L1L2', 'config': {'l1': 0.009999999776482582, 'l2': 0.0}},
'activity_regularizer': None, 'kernel_constraint': None, 'bias_constraint':
None}}, {'class_name': 'Dropout', 'config': {'name': 'dropout_2',
'trainable': True, 'dtype': 'float32', 'rate': 0.2, 'noise_shape': None,
'seed': None}}, {'class_name': 'BatchNormalization', 'config': {'name':
'batch_normalization_2', 'trainable': True, 'dtype': 'float32', 'axis':
ListWrapper([1]), 'momentum': 0.99, 'epsilon': 0.001, 'center': True,
'scale': True, 'beta_initializer': {'class_name': 'Zeros', 'config': {}},
'gamma_initializer': {'class_name': 'Ones', 'config': {}},
'moving_mean_initializer': {'class_name': 'Zeros', 'config': {}},
'moving_variance_initializer': {'class_name': 'Ones', 'config': {}},
'beta_regularizer': None, 'gamma_regularizer': None, 'beta_constraint':
None, 'gamma_constraint': None}}, {'class_name': 'Dense', 'config':
{'name': 'output', 'trainable': True, 'dtype': 'float32', 'units': 10,
'activation': 'linear', 'use_bias': True, 'kernel_initializer':
{'class_name': 'GlorotUniform', 'config': {'seed': None}},
'bias_initializer': {'class_name': 'Zeros', 'config': {}},
'kernel_regularizer': {'class_name': 'L1L2', 'config': {'l1':
0.009999999776482582, 'l2': 0.0}}, 'bias_regularizer': {'class_name':
'L1L2', 'config': {'l1': 0.009999999776482582, 'l2': 0.0}},
'activity_regularizer': None, 'kernel_constraint': None, 'bias_constraint':
None}}], 'build_input_shape': TensorShape([None, 784])}

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_6 (Dense)              (None, 100)               78500
_________________________________________________________________
dropout_1 (Dropout)          (None, 100)               0
_________________________________________________________________
batch_normalization_1 (Batch (None, 100)               400
_________________________________________________________________
dense_7 (Dense)              (None, 100)               10100
_________________________________________________________________
dropout_2 (Dropout)          (None, 100)               0
_________________________________________________________________
batch_normalization_2 (Batch (None, 100)               400
_________________________________________________________________
output (Dense)               (None, 10)                1010
=================================================================
Total params: 90,410
Trainable params: 90,010
Non-trainable params: 400
```

First, the data dimensions and statistics and some example data are inspected:

```
training input shape: (50000, 28, 28), training target shape: (50000,)
validation input shape: (10000, 28, 28), validation target shape: (10000,)
test input shape: (10000, 28, 28), test target shape: (10000,)
```

```
Example image, true label: 5
```



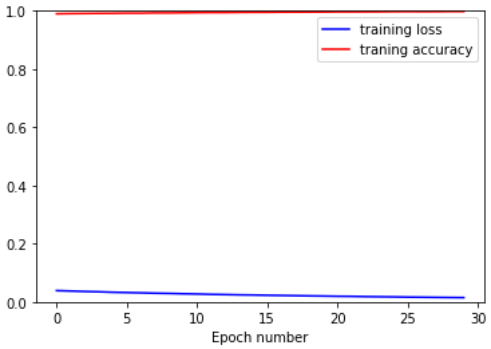
```
min. training data: 0.000000
max. training data: 1.000000
min. validation data: 0.000000
max. validation data: 1.000000
min. test data: 0.000000
max. test data: 1.000000
(50000, 784)
(10000, 784)
(10000, 784)
```

The first series of experiments uses a rather simple model topology (called “baseline”, i.e., a single hidden layer, stochastic gradient descent without momentum, a ReLU activation function at the hidden layers, constant learning rate, no dropout, no batch normalization, no regularization, batch size 1) in order to get a feeling for appropriate values of the learning rate and for fluctuations of the model performance due to random components.

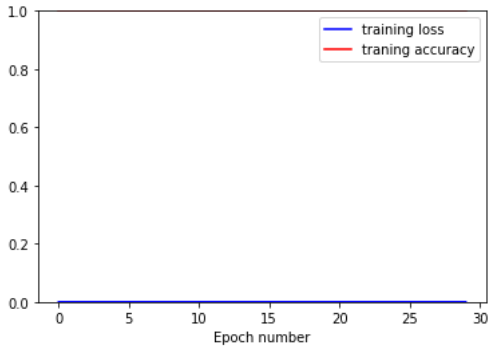
Results for experiments with this baseline model configuration are reported as follows (using a single hidden layer with 50 neurons):

Training loss and training accuracy as function of the number of epochs:

Learning rate 0.001,
best repetition: experiment 1



Learning rate 0.01,
best repetition: experiment 3



The table below contains results of the individual runs (first four lines, without brackets) and the average and standard deviation of the four runs in brackets:

Configuration: 1 hidden layer with 50 neurons	Training loss	Training accuracy	Validation loss	Validation accuracy	Test loss	Test accuracy
Learning rate 0.001	0.043315 0.016557 0.007840 0.004493 (0.018051 +/- 0.017594)	0.988200 0.997180 0.999520 0.999900 (0.996200 +/- 0.005467)	0.087656 0.095606 0.106282 0.114017 (0.100890 +/- 0.011611)	0.973900 0.975000 0.974900 0.974500 (0.974575 +/- 0.000499)	0.087428 0.094776 0.103976 0.110947 (0.099282 +/- 0.010311)	0.972600 0.972300 0.973100 0.973500 (0.972875 +/- 0.000532)
Learning rate 0.01	0.000230 0.000091 0.000064 0.000050 (0.000109 +/- 0.000082)	1.000000 1.000000 1.000000 1.000000 (1.000000 +/- 0.000000)	0.165957 0.171261 0.174457 0.177031 (0.172176 +/- 0.004771)	0.977200 0.977400 0.977500 0.977400 (0.977375 +/- 0.000126)	0.165052 0.170423 0.173927 0.176723 (0.171531 +/- 0.005030)	0.974500 0.974200 0.974900 0.974600 (0.974550 +/- 0.000289)
Learning rate 0.1	1.119319 1.012933 1.259970 1.261435 (1.163414 +/- 0.120444)	0.643020 0.636480 0.549640 0.517320 (0.586615 +/- 0.062815)	1.950709 1.643393 1.684036 2.039909 (1.829512 +/- 0.195584)	0.667900 0.674300 0.568800 0.535000 (0.611500 +/- 0.070239)	1.826685 1.677376 2.649775 1.913383 (2.016805 +/- 0.433092)	0.650400 0.654800 0.558100 0.508900 (0.593050 +/- 0.071658)

Optimal learning rate for this configuration: 0.01

Unfortunately the connection to the cloud was terminated so the job crashed.

Results for a single hidden layer with 100 neurons:

Configuration: 1 hidden layer with 100 neurons	Training loss	Training accuracy	Validation loss	Validation accuracy	Test loss	Test accuracy
Learning rate 0.001	0.028927 0.009216 0.004314 0.002615 (0.011268 +/- 0.012100)	0.993380 0.999380 0.999920 1.000000 (0.998170 +/- 0.003205)	0.076257 0.080250 0.084184 0.087504 (0.082049 +/- 0.004868)	0.978500 0.979200 0.978600 0.978800 (0.978775 +/- 0.000310)	0.071632 0.075381 0.077591 0.081287 (0.076473 +/- 0.004043)	0.976600 0.978600 0.979000 0.978500 (0.978175 +/- 0.001072)
Learning rate 0.01	0.000194 0.000108 0.000077 0.000059 (0.000110 +/- 0.000060)	1.000000 1.000000 1.000000 1.000000 (1.000000 +/- 0.000000)	0.110189 0.114601 0.117470 0.119842 (0.115525 +/- 0.004153)	0.979800 0.979500 0.980100 0.980100 (0.979875 +/- 0.000287)	0.095330 0.098926 0.101281 0.103145 (0.099671 +/- 0.003369)	0.981200 0.981500 0.982000 0.981900 (0.981650 +/- 0.000370)
Learning rate 0.1	1.016015 0.980988 1.179593 1.184637 (1.090308 +/- 0.106989)	0.684720 0.664380 0.593540 0.574640 (0.629320 +/- 0.053443)	1.411853 2.686976 2.546006 2.213479 (2.214578 +/- 0.570781)	0.754300 0.708900 0.572700 0.584900 (0.655200 +/- 0.090283)	1.388084 2.699833 2.537734 2.029452 (2.163776 +/- 0.590746)	0.738100 0.686700 0.566700 0.574200 (0.641425 +/- 0.084654)

Optimal learning rate for this configuration: 0.01

Again, the job unfortunately crashed since the connection to the cloud was terminated.

The following findings can be drawn from the experiments:
For both model configurations (50 or 100 neurons in the single hidden layer), the best learning rate (out of the three values 0.001, 0.01 and 0.1) was found to be 0.01. This value not only provided the best validation and test results, but also the lowest differences between different runs (i.e., the lowest standard deviation of results). Whereas the difference in performance to a learning rate of 0.001 was quite small, convergence was faster for a learning rate of 0.01 (see the plot of the accuracy as function of the number of epochs above). A learning rate of 0.1 is clearly too large, significantly impacting validation and test loss and accuracies. The difference between validation and test accuracies is small (as expected), whereas the difference between validation and test loss is larger (for unknown reasons).

To investigate further configurations in reasonable runtime, the experimental settings are modified as follows (dropout is disabled in these experiments and later added):

- The number of repetitions of each experiment is restricted to 1: `numRepetitions = 1`
  (motivated by the very low standard deviation of results for a learning rate of 0.1 / 0.001).
- Only a single learning rate is investigated: `'learningRates': [0.01],`
  (motivated since this value yielded the best results in the experiments so far).
- The number of epochs is reduced to 20: `num_epochs = 20`
  (motivated by the convergence properties of previous experiments).

Next, the influence of the optimizer shall be addressed. In order to perform these experiments with a reasonable model configuration, a few tests with regard to the model topology (varying the number of neurons in a single hidden layer and varying the number of layers) are performed. For those experiments, stochastic gradient descent has been selected since it yields an acceptable performance so far. Keeping the other parameters fixed as mentioned above, the following results have been obtained (if not mentioned otherwise, a constant learning rate has been used, no dropout, no batch normalization, no regularization and batch size 1):

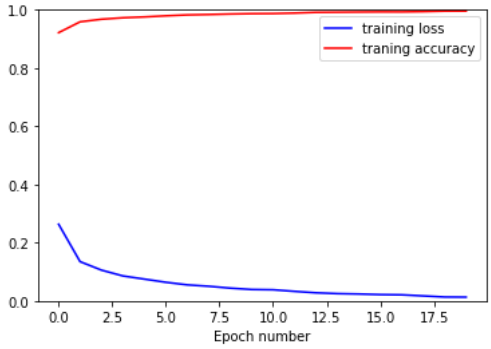
Experimental configurations:

A) Single hidden layer with 50 neurons ([50]), SGD, ReLU activation of hidden layers (39250 + 510 = 39760 parameter)
B) Single hidden layer with 100 neurons ([100]), SGD, ReLU activation of hidden layers (78500 + 1010 = 79510 parameter)
C) Single hidden layer with 200 neurons ([200]), SGD, ReLU activation of hidden layers (157000 + 2010 = 159010 parameter)
D) Two hidden layers with 100 neurons each ([100, 100]), SGD, ReLU
(78500 + 10100 + 1010 = 89610 parameter)
E) Three hidden layers with 100 neurons each ([100, 100, 100]), SGD, ReLU
(78500 + 10100 + 10100 + 1010 = 99710 parameter)
F) Four hidden layers with 100 neurons each ([100, 100, 100, 100]), SGD, ReLU
(78500 + 10100 + 10100 + 10100 + 1010 = 109810 parameter)
G) Single hidden layer with 100 neurons ([100]), Adam, ReLU
(78500 + 1010 = 79510 parameter)
H) Single hidden layer with 100 neurons ([100]), AdaGrad, ReLU
(78500 + 1010 = 79510 parameter)
I) Single hidden layer with 100 neurons ([100]), AdaDelta, ReLU
(78500 + 1010 = 79510 parameter)
J) Single hidden layer with 100 neurons ([100]), RMSProp, ReLU
(78500 + 1010 = 79510 parameter)

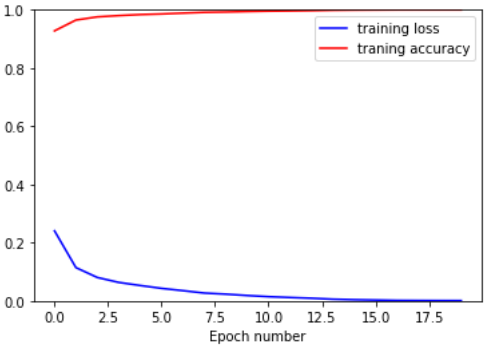
Summary of results:

Configuration: Learning rate 0.01	Training Loss	Training accuracy	Validation loss	Validation accuracy	Test loss	Test accuracy
A: [50], SGD, ReLU	0.012859	0.995900	0.136605	0.971900	0.147408	0.972600
B: [100], SGD, ReLU	0.000902	1.000000	0.093322	0.979300	0.086790	0.979700
C: [200], SGD, ReLU	0.000721	1.000000	0.076720	0.983800	0.070349	0.983000
D: [100, 100], SGD, ReLU	0.009871	0.996900	0.138232	0.979600	0.121062	0.981000
E: [100, 100, 100], SGD, ReLU	0.020172	0.993760	0.147756	0.972800	0.144868	0.973300
F: [100, 100, 100, 100], SGD, ReLU	0.016737	0.995020	0.119888	0.977900	0.128177	0.976000
G: [100], Adam, ReLU	0.727100	0.861640	1.257219	0.864800	1.682080	0.857300
H: [100], AdaGrad, ReLU	0.068415	0.982160	0.095219	0.972600	0.093076	0.972400
I: [100], AdaDelta, ReLU	0.267616	0.925260	0.243221	0.930900	0.254327	0.930100
J: [100], RMSProp, ReLU	0.847191	0.973760	1.757821	0.963500	1.660859	0.965500

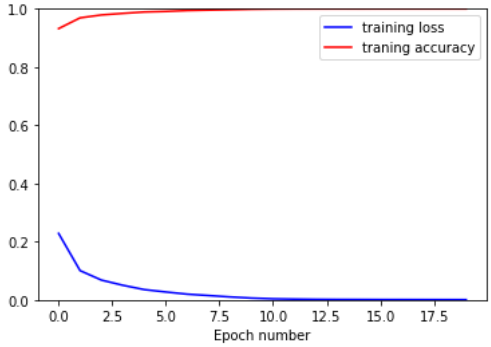
Configuration A:



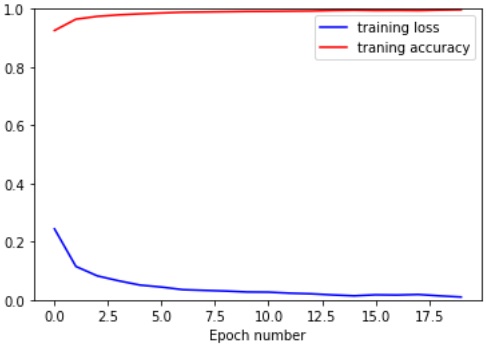
Configuration B:



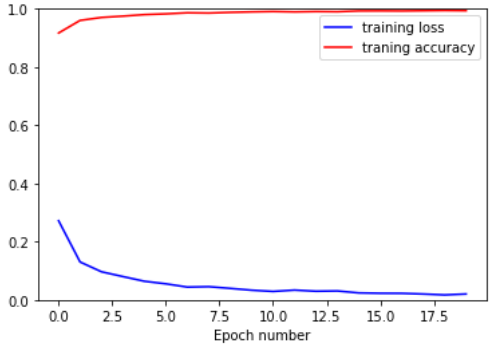
Configuration C:



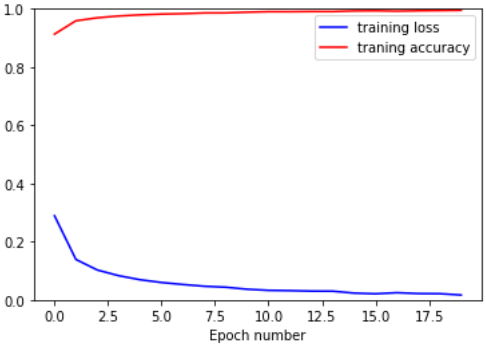
Configuration D:



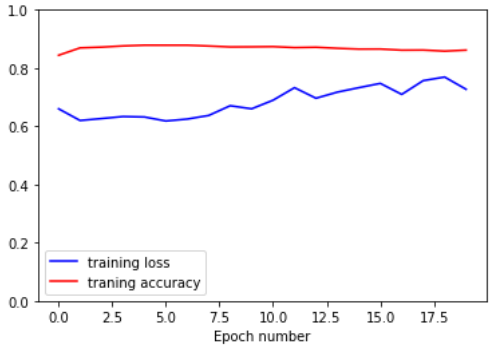
Configuration E:



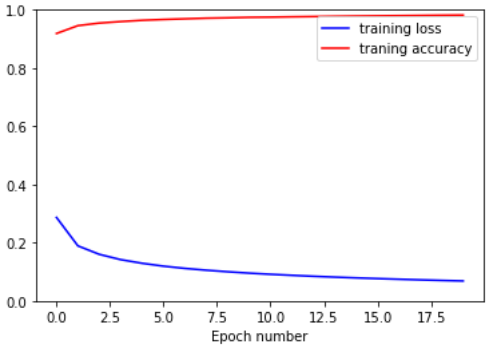
Configuration F:



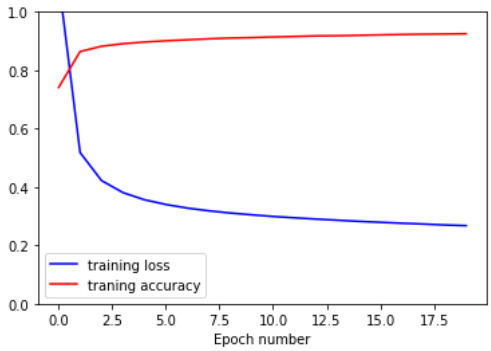
Configuration G:



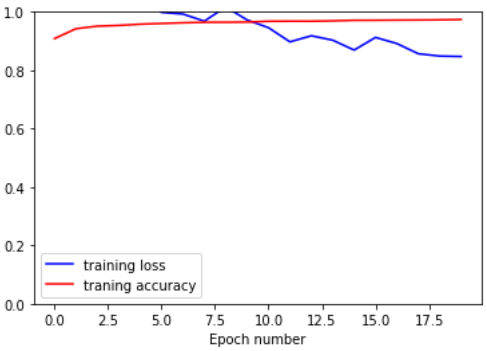
Configuration H:



Configuration I:



Configuration J:



The following findings can be drawn from the experiments:

- For a single hidden layer, the performance improves with increasing number of neurons.
- Adding further layers, performance minimally deteriorates using the indicated experimental settings, suggesting that a single hidden layer appears to be sufficient to achieve a reasonable performance.
- Best results have been obtained with the stochastic gradient descent optimizer (yet without momentum). Adam and Nadam (which performed similarly as Adam; results not shown) as well as AdaDelta provided sub-optimal results in these experiments.
- As seen from the plots of the training loss and accuracy, some configurations have not converged yet (e.g., configurations A, I). Other configurations (with best performance) converge faster.
- The computation time slightly increases with increasing network size, i.e., number of parameters (where the number of parameters of a fully connected network increases significantly faster when increasing the number of hidden neurons per layer than when adding more layers!) and also depends on other settings, e.g. the solver. Mostly, the computation time was between 50 and 70 seconds per epoch (for a batch size 1). With increasing batch size, the training time is however considerably reduced: For a batch size of 8, with a single hidden layer with 100 neurons and SGD solver, the computation time per epoch was only 7 seconds. Note however, that modifying the batch size generally implies to adjust the learning rate since both parameters are related).

Further experiments address some variants of the stochastic gradient descent algorithm, i.e., applying momentum and potentially Nesterov momentum. Experiments with momentum 0.9 (potentially added by Nesterov momentum) however were not successful (neither in a single-layer network with 100 nor 1000 neurons).

Therefore, the following experiments use the standard stochastic gradient descent algorithm without momentum. Here, the influence of the number of neurons in a single-layer network is further investigated; then, additional variations of the configuration are addressed. Again, if not stated otherwise, the following experiments use the parameters reported above: Single repetition, 20 epochs, constant learning rate of 0.01, no dropout, no batch normalization, no regularization and batch size 1:

K) Single hidden layer with 300 neurons ([300]), SGD, ReLU activation of hidden layers (235500 + 3010 = 238510 parameter)
L) Single hidden layer with 500 neurons ([500]), SGD, ReLU activation of hidden layers (392500 + 5010 = 397510 parameter)
M) Single hidden layer with 750 neurons ([750]), SGD, ReLU activation of hidden layers (588750 + 7510 = 596260 parameter)
N) Single hidden layer with 1000 neurons ([1000]), SGD, ReLU activation of hidden layers (785000 + 10010 = 795010 parameter)
O) Single hidden layer with 1500 neurons ([1500]), SGD, ReLU activation of hidden layers (1177500 + 15010 = 1192510 parameter)
P) Single hidden layer with 1000 neurons ([1000]), SGD, ReLU activation of hidden layers, dropout 0.2 (785000 + 10010 = 795010 parameter)
Q) Single hidden layer with 1000 neurons ([1000]), SGD, ReLU activation of hidden layers, batch size 8 (785000 + 10010 = 795010 parameter)

Note that dropout is usually not applied at the input layer. In configuration P, dropout was applied not to the input, but to the output layer (in some tests, the accuracy on the validation and test data did not change much with the dropout configuration, i.e. whether dropout was applied only to the "inner" hidden layers or including the input and / or including the output layer).

These experiments yielded the following results (where results from previous experiments have been repeated in gray for better readability):

Configuration: Learning rate 0.01	Training loss	Training accuracy	Validation loss	Validation accuracy	Test loss	Test accuracy
A: [50], SGD, ReLU	0.012859	0.995900	0.136605	0.971900	0.147408	0.972600
B: [100], SGD, ReLU	0.000902	1.000000	0.093322	0.979300	0.086790	0.979700
C: [200], SGD, ReLU	0.000721	1.000000	0.076720	0.983800	0.070349	0.983000
K: [300], SGD, ReLU	0.000695	1.000000	0.071323	0.983800	0.066822	0.982900
L: [500], SGD, ReLU	0.000666	1.000000	0.069209	0.984700	0.064297	0.983900
M: [750], SGD, ReLU	0.000657	1.000000	0.068998	0.983600	0.061477	0.983900
N: [1000], SGD, ReLU	0.000635	1.000000	0.066217	0.984900	0.059083	0.984200
O: [1500], SGD, ReLU	0.000627	1.000000	0.067720	0.983800	0.057673	0.984500
P: [1000], SGD, ReLU, dropout 0.2	0.187101	0.891420	0.077229	0.985100	0.061507	0.985300
Q: [1000], SGD, ReLU, batch size 8	0.023458	0.995620	0.071826	0.978600	0.068080	0.978400

The learning curves (training loss and accuracy as function of the number of epochs) were similar to those above (for configuration A, B or C), except for configuration P.

Note that the optimal configuration is selected according to the validation accuracy (assuming that the "accuracy" metric is defined to be the metric of interest). The corresponding test accuracy then is an "independent" estimation of the accuracy to be expected in further experiments.

In these experiments, the optimal validation accuracy is achieved (without dropout) with 1000 hidden neurons (although the difference in validation accuracy is rather small in the investigated range of neuron numbers). The final test accuracy is then 0.9842, which is not the best test accuracy obtained. Note that other considerations (e.g. runtime) might influence the choice of the final network configuration; this should be reflected in a suitable definition of an evaluation metric which is then optimized on the validation data. Dropout minimally improved the validation and the test accuracy; a dropout rate of 0.2 yielded a test accuracy of 0.9860.

Due to the lower runtime, all following experiments were performed with a batch size of 8. The other parameters were left unchanged (unless otherwise stated), in particular a single hidden layer with 1000 neurons was used (i.e., 795010 parameter). Results were as follows ("learn. schedule" refers to an exponential decay of the learning rate with initial learning rate of 0.01, "regularization" refers to L1 regularization):

Configuration: Learning rate 0.01, batch size 8	Training loss	Training accuracy	Validation loss	Validation accuracy	Test loss	Test accuracy
Q: [1000], SGD, ReLU	0.023458	0.995620	0.071826	0.978600	0.068080	0.978400
R: [1000], SGD, ReLU, dropout 0.2	0.220987	0.886130	0.064576	0.982300	0.058527	0.980700
S: [1000], SGD, ReLU, learn. schedule	0.023576	0.995840	0.066890	0.979500	0.065112	0.980200
T: [1000], SGD, ReLU, sigmoid activ.	0.230429	0.934460	0.223759	0.936400	0.230059	0.932700
U:[1000], SGD, ReLU, batch normaliz.	0.025358	0.992040	0.081662	0.980800	0.068359	0.982900

In conclusion, these experiments did not lead to additional improvements: Dropout again minimally improved results; the exponential learning rate schedule performed similarly in this configuration to a constant learning rate, and the sigmoid activation function performed worse than the ReLU activation. Batch normalization only (slightly) improved the test accuracy, but not the validation accuracy (note that batch normalization) is recommended generally for networks with a larger number of layers; see exercise 4). L1-Regularisation (with regularisation weight 0.01) did not work (results not shown).

To summarize, in these experiments the optimal network configuration was found to have a single hidden layer, 100 to about 1000 hidden neurons, a learning rate of about 0.01 and the stochastic gradient descent optimizer, and dropout of 0.2. The optimal test accuracy achieved was 0.985 (with a batch size of 1).

Exercise 4 (Vanishing gradient):

a) The Jupyter notebook implements a multi-layer perceptron for use on the MNIST digit classification problem. Apart from the training loss and accuracy, it also displays a histogram of the weights (between the input and the first hidden layer) after initialization and at the end of the training, and visualizes the weights (between the input layer and 16 hidden neurons of the first hidden layer). Using a sigmoid activation function, compare the output for a single hidden layer, five and six hidden layers. Then change to a ReLU activation function and inspect the results for six hidden layers. Discuss your findings.

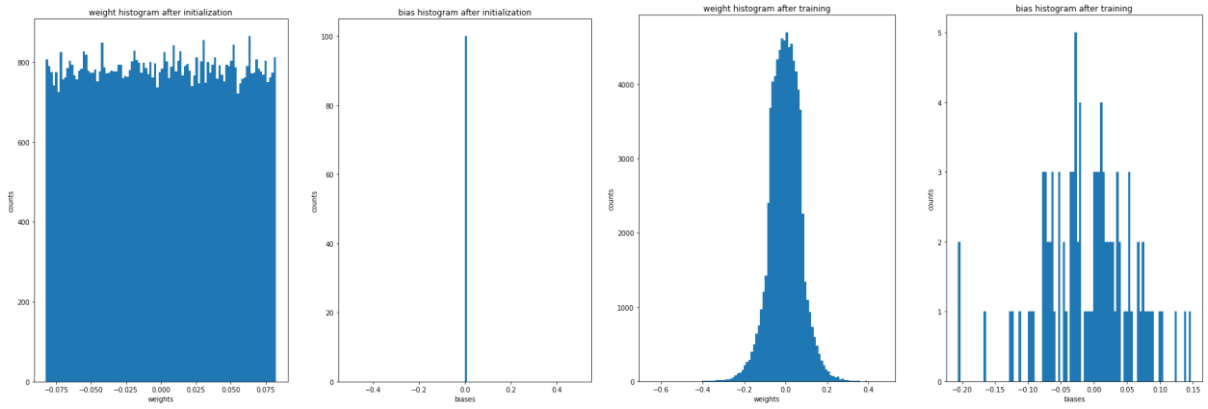
Solution:

Considering a network with a single, five and six hidden layer(s) with 100 hidden units (each), running for 20 epochs, sigmoid activation function, stochastic gradient descent with constant schedule, learning rate 0.01:

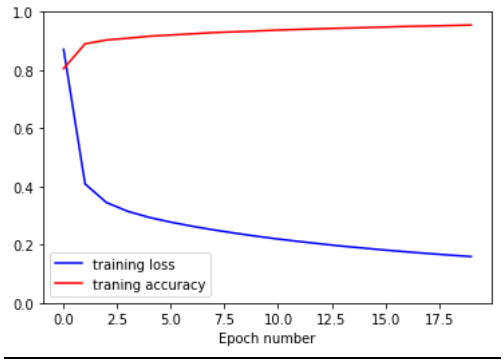
Baseline configuration refers to a configuration without batch normalization, without dropout and without regularization, using Glorot uniform initialization for the weights and zero initialization for the biases and a batch size of 8.

<u>Single hidden layer, sigmoid activation function, baseline configuration:</u>

Weight / bias histogram after initialization: *Weight / bias histogram after training:*

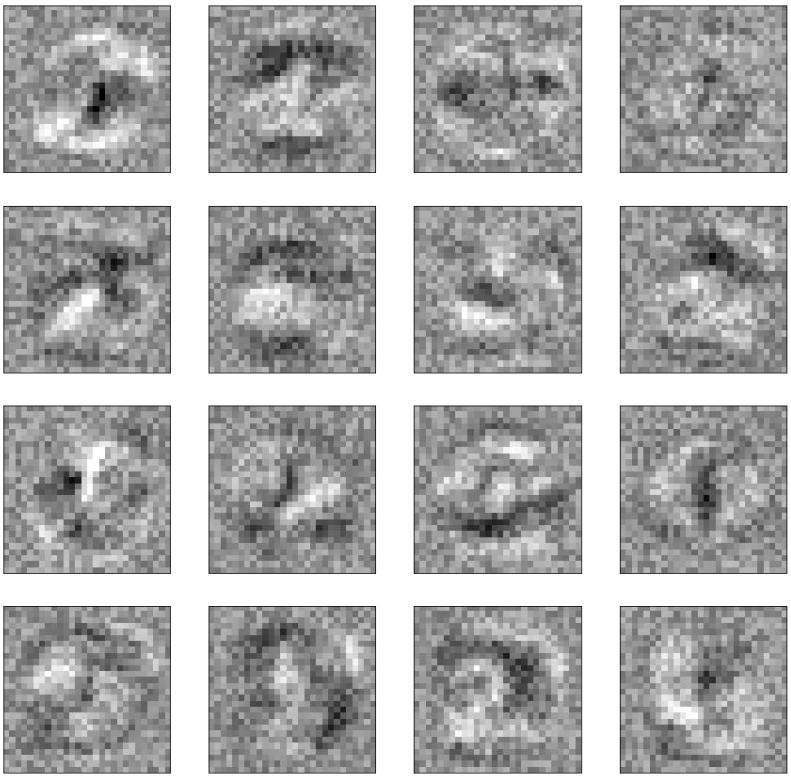


Learning curves:



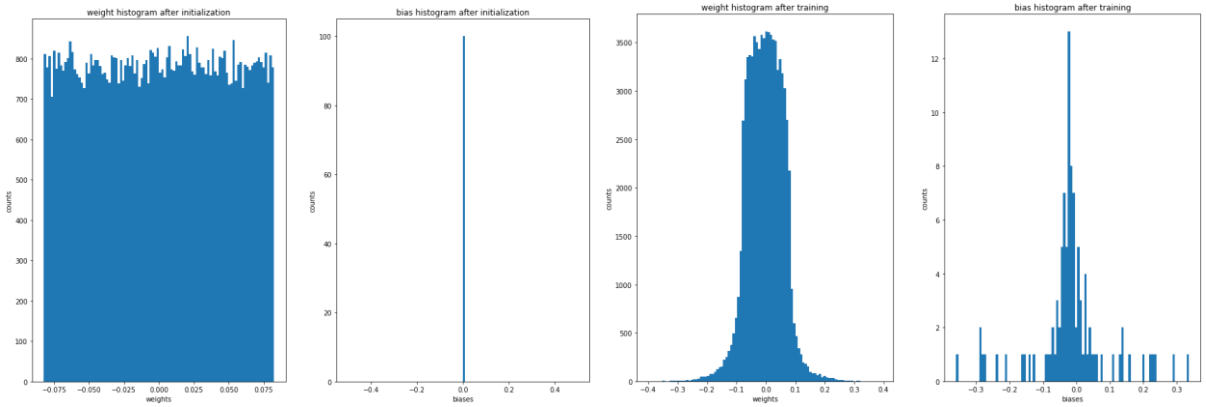

```
final training loss: 0.159168
final training accuracy: 0.954880
final validation loss: 0.156789
final validation accuracy: 0.956200
final test loss: 0.159964
final test accuracy: 0.954000
```

Visualization of the weights between input and some of the hidden neurons of the first hidden layer after training:

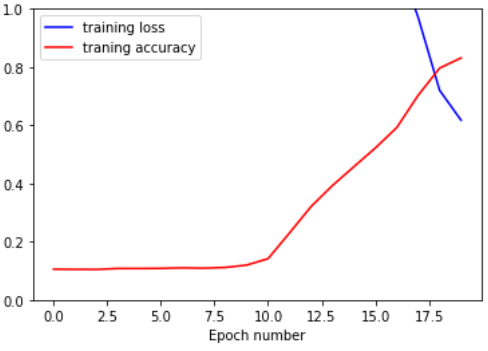


Five hidden layers, sigmoid activation function, baseline configuration:

Weight / bias histogram after initialization: *Weight / bias histogram after training:*

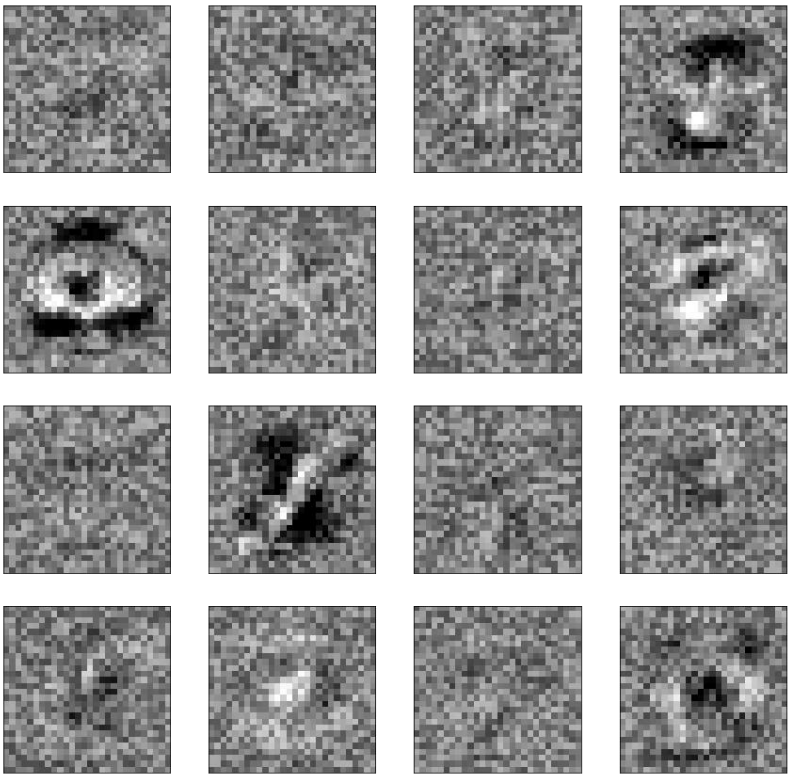


Learning curves:



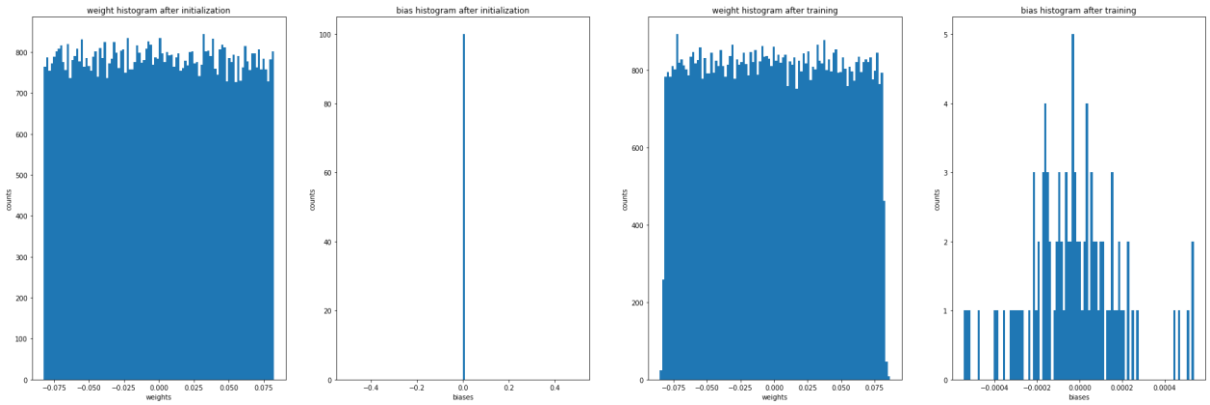

```
final training loss: 0.617239
final training accuracy: 0.831420
final validation loss: 0.552177
final validation accuracy: 0.857500
final test loss: 0.577704
final test accuracy: 0.852800
```

Visualization of the weights between input and some of the hidden neurons of the first hidden layer after training:

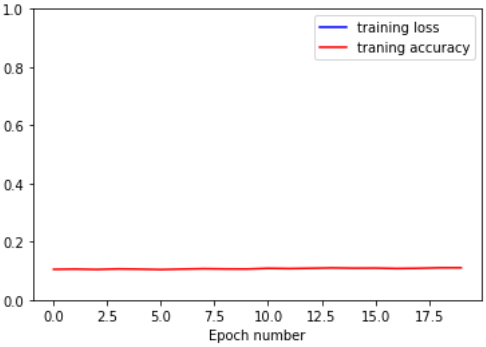


Six hidden layers, sigmoid activation function, baseline configuration:

Weight / bias histogram after initialization: *Weight / bias histogram after training:*

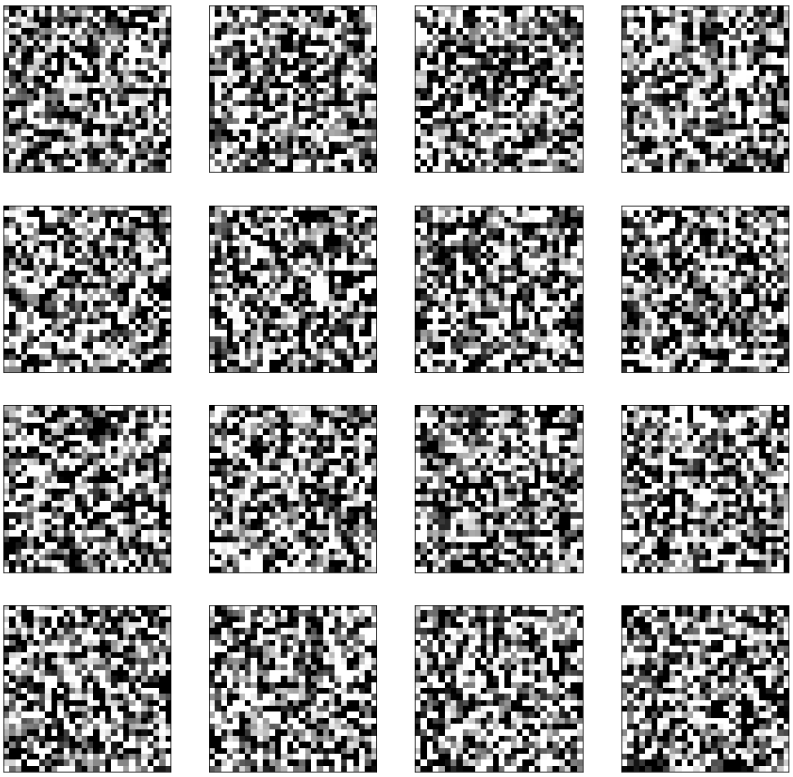


Learning curves:

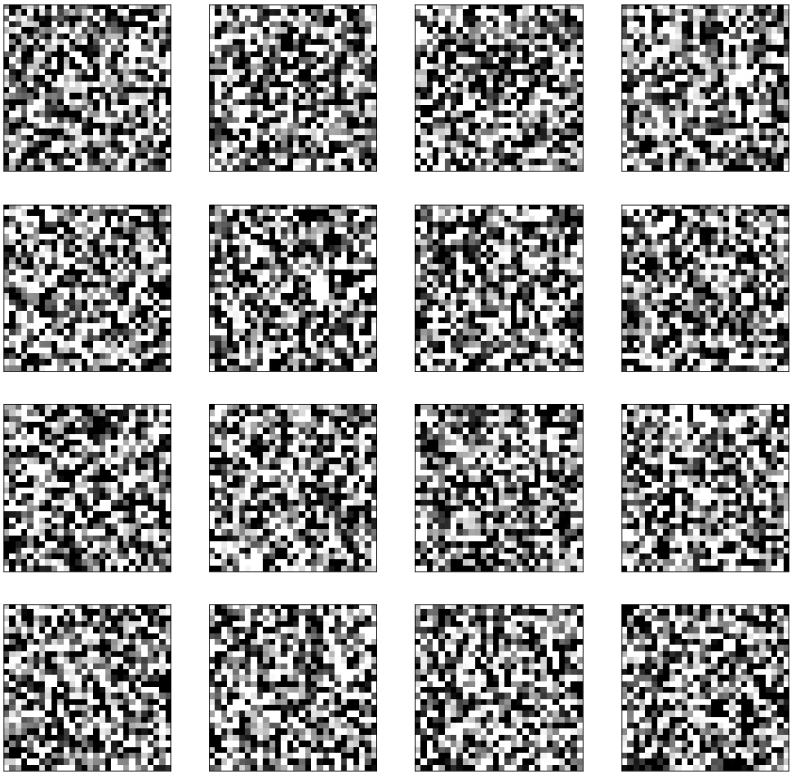



```
final training loss: 2.302763
final training accuracy: 0.110340
final validation loss: 2.304596
final validation accuracy: 0.106400
final test loss: 2.302881
final test accuracy: 0.113500
```

Visualization of the weights between input and some of the hidden neurons of the first hidden layer after initialization:

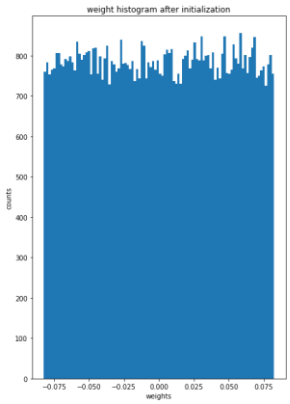


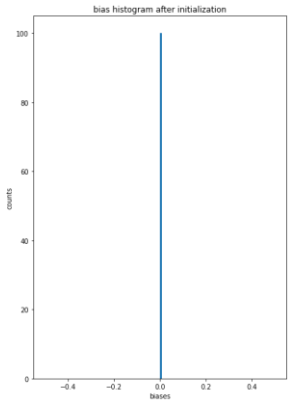
Visualization of the weights between input and some of the hidden neurons of the first hidden layer after training:



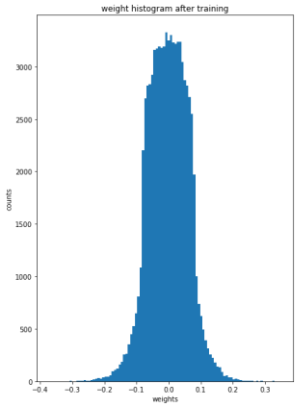
Six hidden layers, ReLU activation function, baseline configuration:

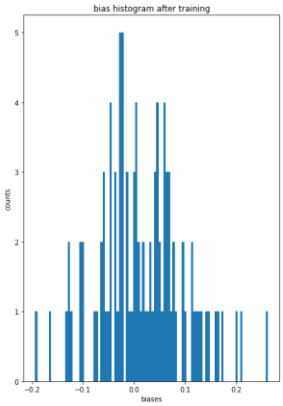
Weight / bias histogram after initialization:



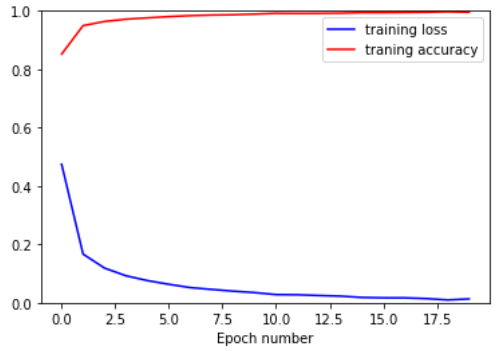


Weight / bias histogram after training:



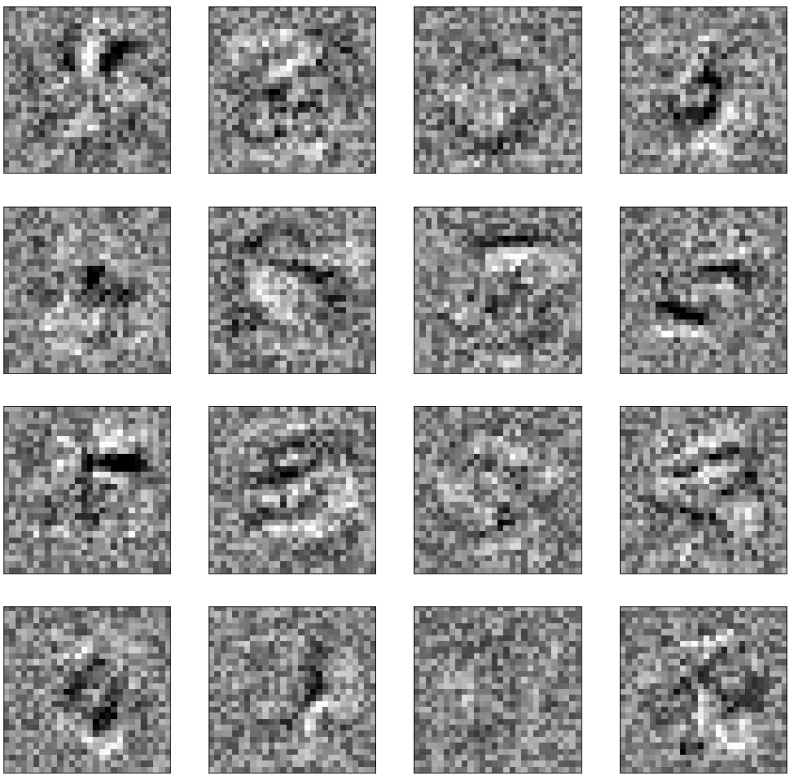


Learning curves:




```
final training loss: 0.013037
final training accuracy: 0.995540
final validation loss: 0.104106
final validation accuracy: 0.977900
final test loss: 0.102228
final test accuracy: 0.977600
```

Visualization of the weights between input and some of the hidden neurons of the first hidden layer after training:



Interpretation:

For a single hidden layer (with 100 hidden neurons), training succeeds (as in the previous exercise). This can be seen particularly in the loss and accuracy measures on training, validation and test data and in the learning curves. Comparing the weight and bias histograms before and after training, a large qualitative and quantitative change can be observed. The visualization of the weights reveals some stroke-like patterns, in contrast to the weight visualizations after initialization (see below for 6 hidden layers). Therefore, the weights (even between the input and the first hidden layer) are modified significantly, and apparently in a constructive way.

With five hidden layers (of 100 neurons each), training has not converged yet (as can be seen from the learning curves). This is reflected in larger losses and lower accuracy as compared to a single hidden layer. The weight histogram after training, however, is similar to the weight histogram with a single hidden layer. The visualization of the weights indicates that some neurons of the first hidden layer have weights similar to those of a single hidden layer network, whereas other neurons of the first hidden layer do not seem to have converged yet.

Using six hidden layers, training does not succeed. This is seen in the losses and accuracies as well as in the learning curves. The histogram of the weights (between input and hidden layer) before and after training is very similar. Note that the bias values after training are extremely low, so that the shape of the bias histogram after training might be misleading; bias values are only minimally modified. Also the visualization of the weights before and after training hardly shows any changes. This again suggests that the weights between the input and the hidden layer are hardly modified.

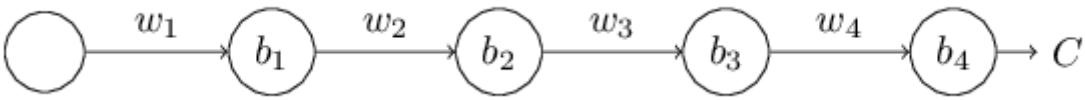
Switching to the ReLU activation function with six hidden layers, training succeeds (as seen from the losses and accuracies as well as the learning curves); in fact, this configuration significantly outperforms a single layer with sigmoid activation function. The weight and bias histograms after training are comparable to those of a network with a single or five hidden layers and sigmoid activation function. The weight visualizations are more or less comparable (although less expressive) than those of a network with a single hidden layer and sigmoid activation function.

These experiments clearly demonstrate the advantages of the ReLU activation function compared to a sigmoid activation function with regard to the vanishing gradient problem.

b) Give a theoretical justification, why the weights and biases of neurons in the first hidden layers in a multi-layer perceptron with many hidden layers are modified only slowly when using a sigmoid activation function and gradient descent. To this end, consider – as an example – a simplified network with three hidden layers (and a single neuron per layer), compute and analyse the change Δb_1 of the bias of the first hidden neuron with respect to a change in the cost function C. What changes in your analysis when using a ReLU activation function instead of a sigmoid?

Solution:

Consider a simplified network with three hidden layers:



Analyse the modification of the bias b_1 of the first neuron: The bias b_1 is modified according to the partial derivative of the cost function C with respect to the bias b_1. Specifically, a small change Δb_1 will lead to a change $\Delta C \approx \frac{\partial C}{\partial b_1}\Delta b_1$. Compute the gradient $\frac{\partial C}{\partial b_1}$ by tracking the change of b_1 through the cascade:

Generally, the postsynaptic potential z_j is given by $z_j = w_j a_{j-1} + b_j$ and the activation a_j by $a_j = \sigma(z_j)$ with a sigmoid activation function $\sigma(z_j)$, e.g. $\sigma(z) = \frac{1}{1+e^{-z}}$.

For the activation a_1 we have (using an input $x = a_0$): $a_1 = \sigma(z_1) = \sigma(w_1 a_0 + b_1)$.

A change in b_1 leads to $\Delta a_1 \approx \frac{\partial \sigma(w_1 a_0 + b_1)}{\partial b_1}\Delta b_1 = \sigma'(z_1)\Delta b_1$.

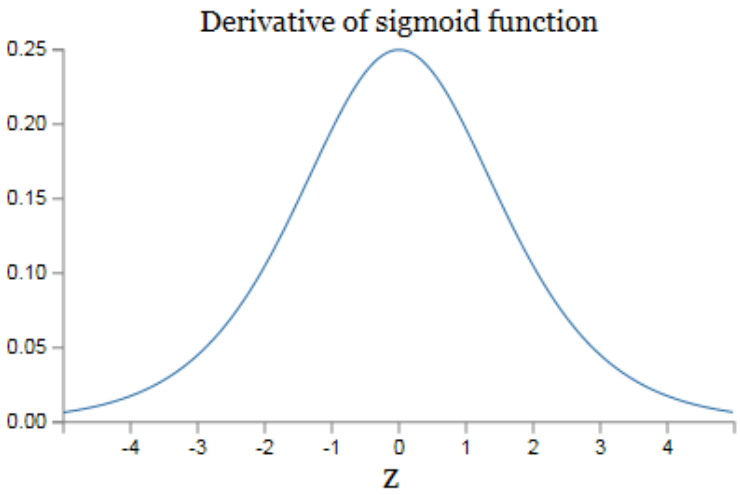
With $z_2 = w_2 a_1 + b_2$, this leads to a change in z_2: $\Delta z_2 \approx \frac{\partial z_2}{\partial a_1}\Delta a_1 = w_2 \Delta a_1 \approx \sigma'(z_1) w_2 \Delta b_1$.

This then modifies the activation a_2, which in turn leads to a change in the postsynaptic potential z_3, etc. Finally, we obtain: $\Delta C \approx \sigma'(z_1) w_2 \cdot \sigma'(z_2) w_3 \cdot \sigma'(z_3) w_4 \sigma'(z_4) \frac{\partial C}{\partial a_4}\Delta b_1$.

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$

Now consider the derivative of a sigmoid function:

$$\sigma(z) = \frac{1}{1+e^{-z}} \Rightarrow \sigma'(z) = \frac{e^{-z}}{(1+e^{-z})^2}$$



The derivative of the sigmoid has a maximal value of 0.25 at $z = 0$.

Using random initial weights in the interval [0, 1], this means that $\left|w_j \cdot \sigma'(z_j)\right| < \frac{1}{4}$.

Thus, there is an exponential decrease over the cascade:

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1)\overbrace{w_2\sigma'(z_2)}^{<\frac{1}{4}}\overbrace{w_3\sigma'(z_3)}^{<\frac{1}{4}}\underbrace{w_4\sigma'(z_4)\frac{\partial C}{\partial a_4}}$$

$$\updownarrow \text{ common terms}$$

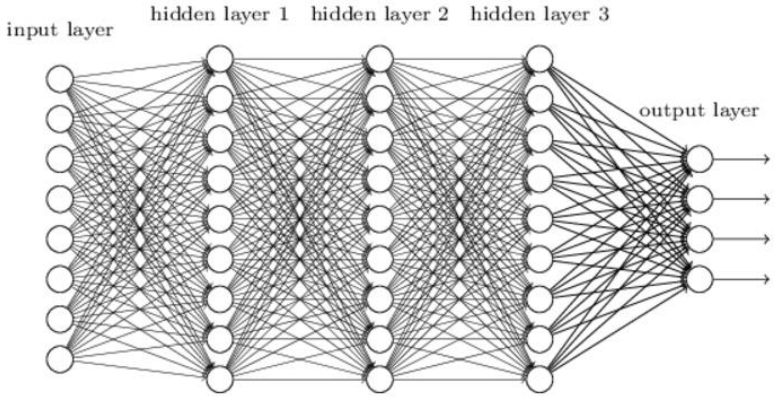
$$\frac{\partial C}{\partial b_3} = \sigma'(z_3)\overbrace{w_4\sigma'(z_4)\frac{\partial C}{\partial a_4}}$$

Thus we see that the gradient of the cost function *C* with respect to the bias is being reduced with each layer (similar arguments hold for the weight parameters). Since the amount of change of the bias (and similarly of the weights) is proportional to the gradient, the bias (and weight parameters) of early layers are modified less and less with increasing number of layers ("vanishing gradient problem").

The opposite problem ("exploding gradient problem") can occur if the initial weights are large, say on the order of 100. In this case, in spite of the maximum of the derivative of the sigmoid function being 0.25, the product $\left|w_j \cdot \sigma'(z_j)\right|$ is much larger than 1 for a broad range of values of the postsynaptic potential z_j. This can lead to parameter changes with are too large.

Switching the activation function from sigmoid to ReLU, the derivative of the activation function becomes constant in case of positive postsynaptic potentials and 0 otherwise. For positive postsynaptic potentials, the decrease of the gradient of earlier layers with increasing number of layers is therefore much weaker. This may avoid the vanishing gradient problem (if, however, any postsynaptic potential in any of the layers is negative, the gradient becomes 0, motivating variations of the standard ReLU function which are nonzero and strictly monotonous even for negative postsynaptic potentials).

In case of more complex network, the analysis becomes more involved:



Now, the gradient in the *l*-th layer in a network with *L* layers becomes:

$$\delta^l = \Sigma'(z^l)(w^{l+1})^T\Sigma'(z^{l+1})(w^{l+2})^T\ldots\Sigma'(z^L)\nabla_a C$$

Diag. matrix with $\sigma'(z) < 0.25$ — weight matrix — vector of partial derivatives w.r.t. output activation

The matrices $\Sigma'(z^l)$ have small entries on the diagonal (using the logistic activation function none is larger than 0.25). Provided the entries in the weight matrices w^j aren't too large, each additional term $(w^j)^T \Sigma'(z^l)$ tends to make the gradient vector smaller, leading to a vanishing gradient. Since the number of terms in the expression is large, some terms may lead to an exploding and thus altogether to an unstable gradient. Due to the vanishing gradient problem, learning slows down in early layers. This might be overcome by variants of the ReLU activation function.

c) Starting from your analysis for the multi-layer perceptron with six hidden layers and sigmoid activation function in part a), try to find other model configurations which lead to a successful training. You may modify e.g. the learning rate and batch size, the weight and bias initialization, apply batch normalization and / or dropout, and add regularization.

Solution:

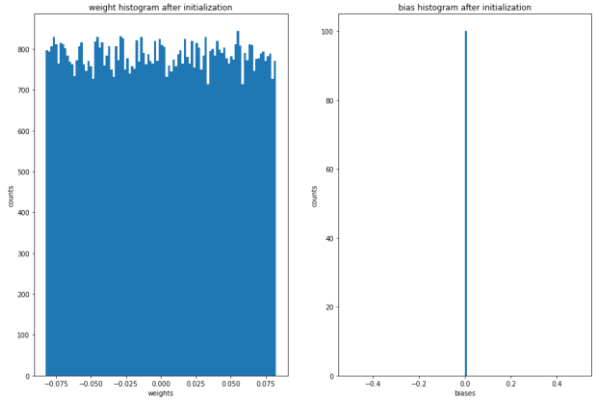
The initialization scheme of the weights and biases can be modified by changing

```
weight_init = tfi.glorot_uniform()  # default: glorot_uniform(); e.g. glorot_normal(), he_normal(), he_uniform(), lecun_normal(), lecun_uniform(), RandomNormal(), RandomUniform(), Zeros() etc.
bias_init = tfi.Zeros()  #   default: Zeros(); for some possible values see weight initializers
```

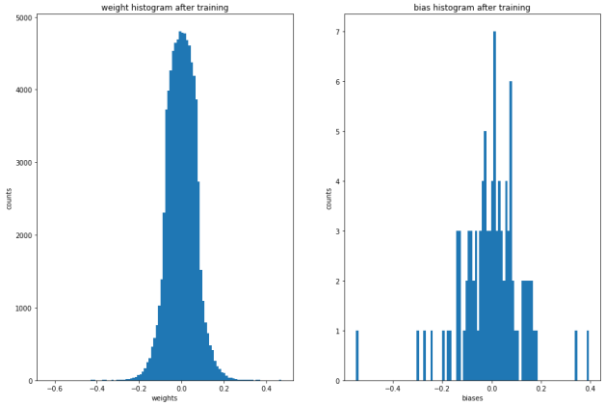
The choice of the initialization scheme is reflected in the initial weight and bias histograms (see examples below).
Note that in the following configurations, all changes compared to the baseline configuration are indicated in bold face:

Applying batch normalization: Six hidden layers, sigmoid activation function, baseline configuration but enabling batch normalization:

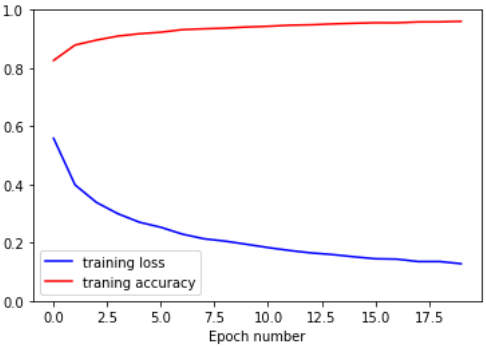
Weight / bias histogram after initialization:



Weight / bias histogram after training:

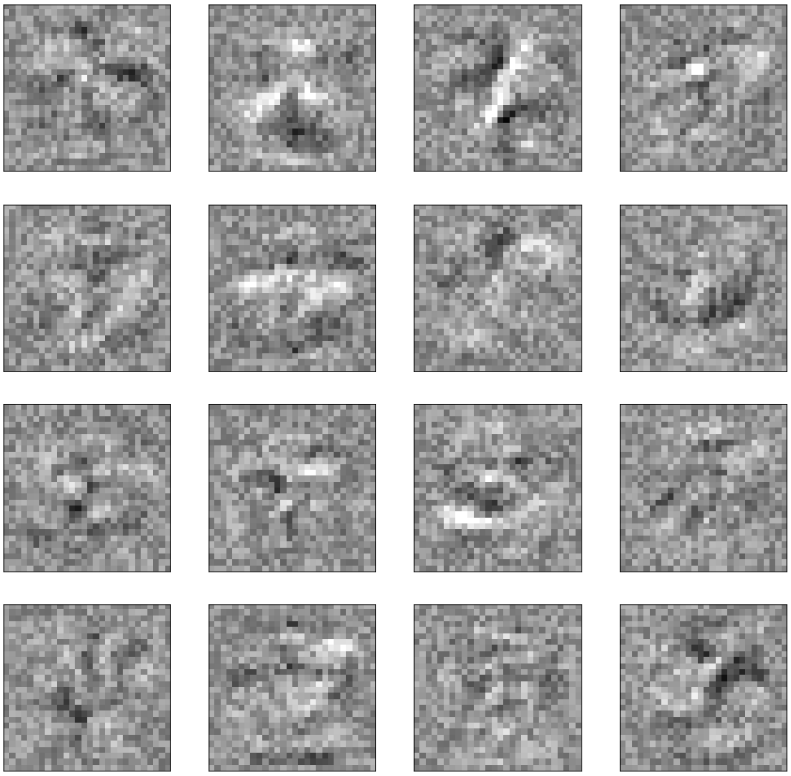


Learning curves:




```
final training loss: 0.127679
final training accuracy: 0.960480
final validation loss: 0.071721
final validation accuracy: 0.979000
final test loss: 0.071740
final test accuracy: 0.977400
```

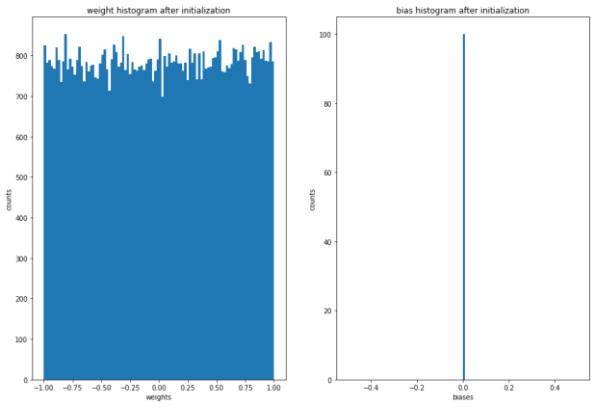
Visualization of the weights between input and some of the hidden neurons of the first hidden layer after training:



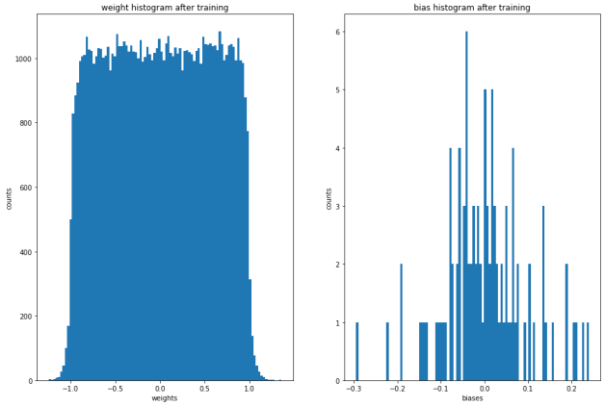
Different weight initialization: Six hidden layers, sigmoid activation function, baseline configuration but using "random uniform" weight initialization in the interval [-1, 1[and zero initialization for the biases:

```
weight_init = tfi.RandomUniform(minval=-1.0, maxval=1.0)
bias_init = tfi.Zeros()
```

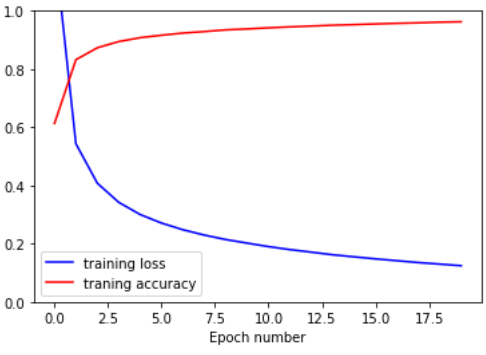
Weight / bias histogram after initialization:



Weight / bias histogram after training:

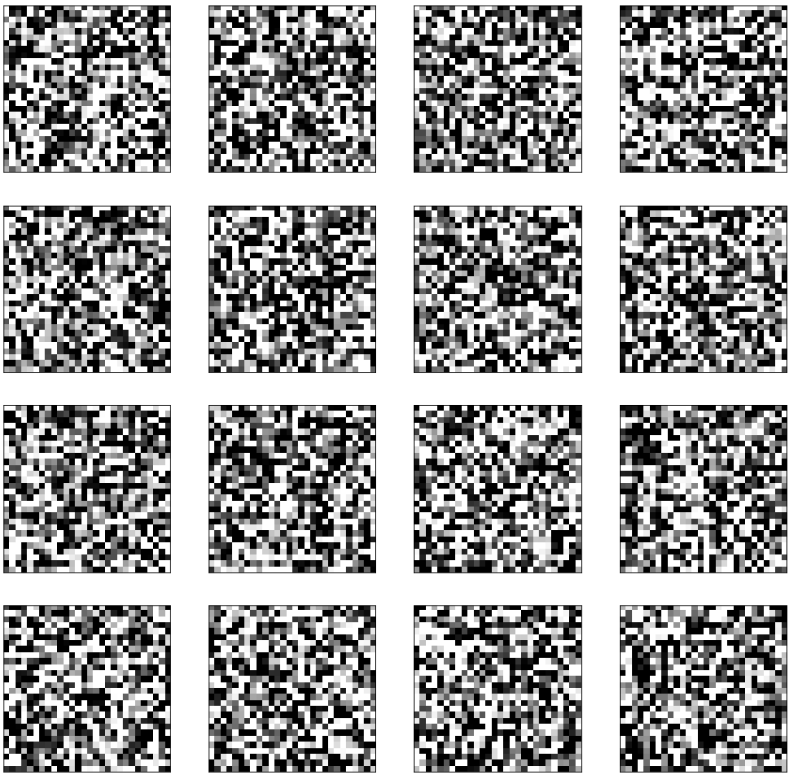


Learning curves:

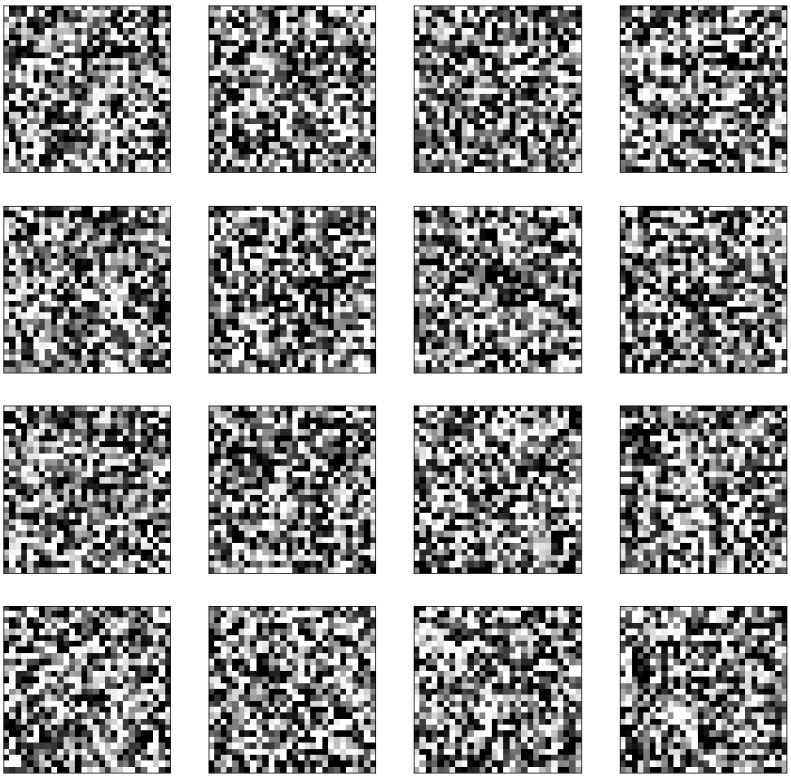



```
final training loss: 0.124293
final training accuracy: 0.962560
final validation loss: 0.175328
final validation accuracy: 0.949100
final test loss: 0.177092
final test accuracy: 0.946200
```

Visualization of the weights between input and some of the hidden neurons of the first hidden layer after initialization:



Visualization of the weights between input and some of the hidden neurons of the first hidden layer after training:

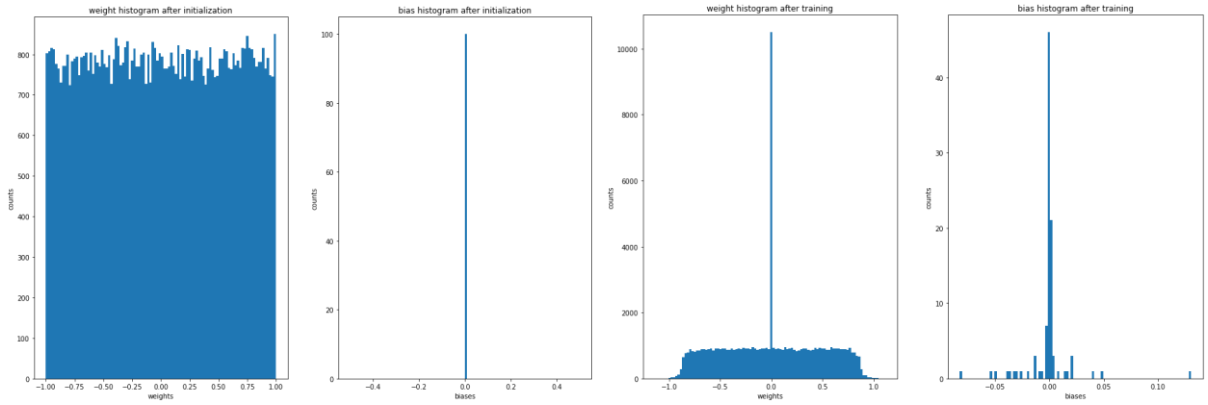


Different weight initialization plus L1-regularization: Six hidden layers, sigmoid activation function, baseline configuration but using "random uniform" weight initialization in the interval [-1, 1[and zero initialization for the biases and L1 regularization of weights and biases with coefficient 0.0001:

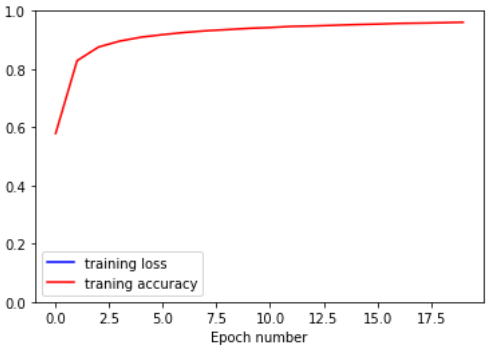
```
weight_init = tfi.RandomUniform(minval=-1.0, maxval=1.0)
bias_init = tfi.Zeros()

regularization_weight = 0.0001
regularizer = tfr.l1(l=regularization_weight)
```

Weight / bias histogram after initialization: *Weight / bias histogram after training:*

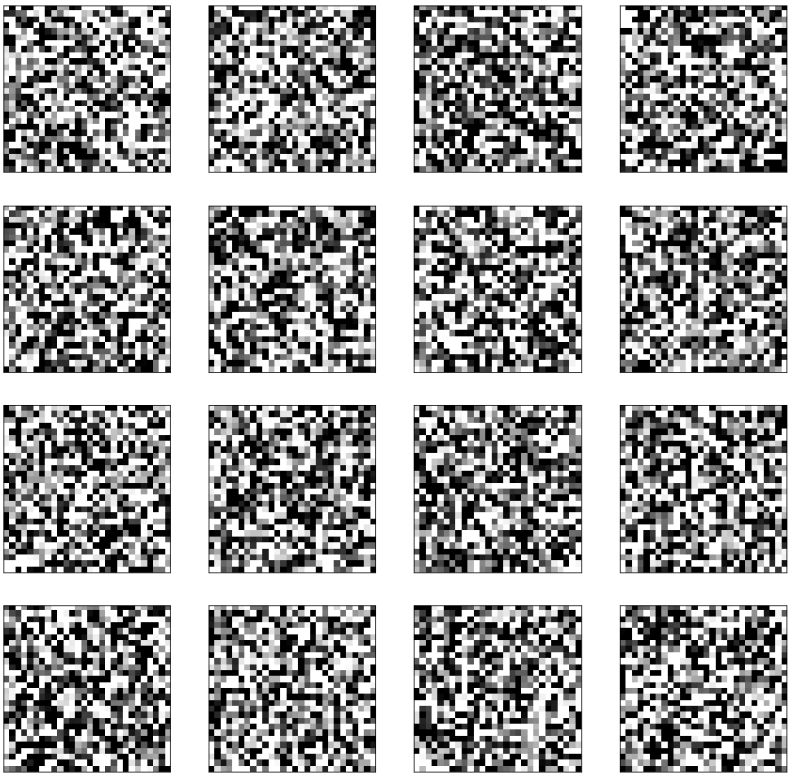


Learning curves:

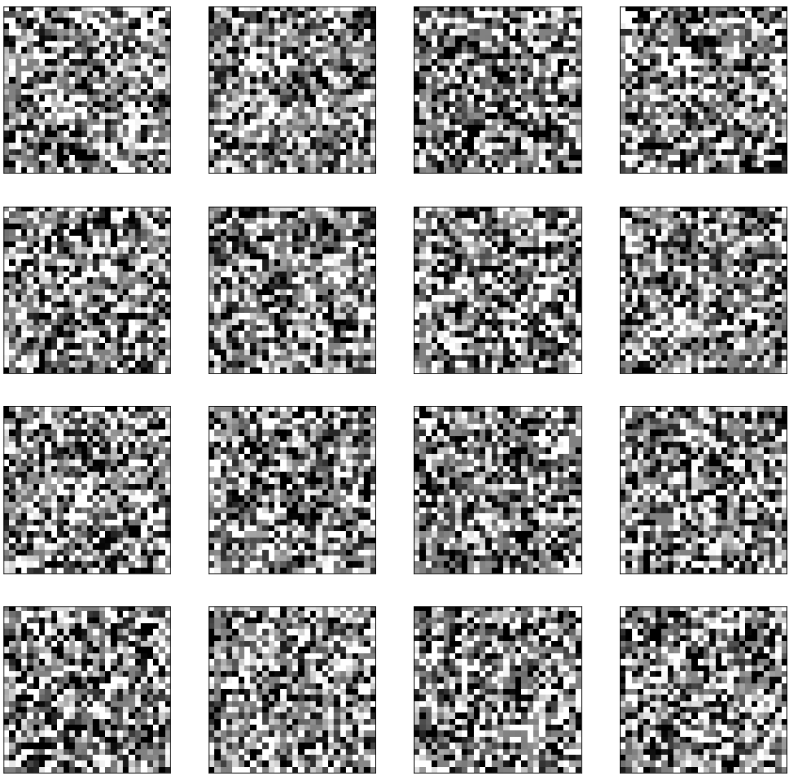



```
final training loss: 5.152094
final training accuracy: 0.960780
final validation loss: 5.146176
final validation accuracy: 0.956600
final test loss: 5.146333
final test accuracy: 0.952500
```

Visualization of the weights between input and some of the hidden neurons of the first hidden layer after initialization:



Visualization of the weights between input and some of the hidden neurons of the first hidden layer after training:

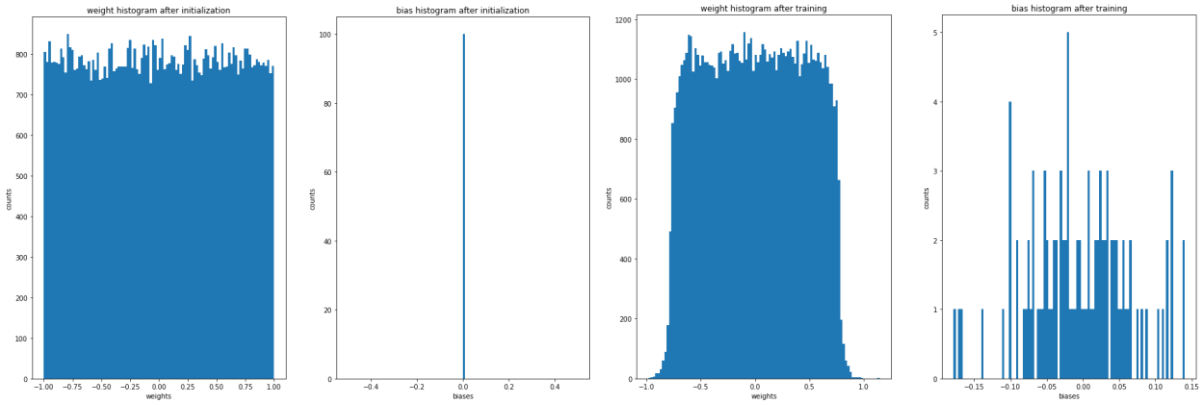


Different weight initialization plus L2-regularization: Six hidden layers, sigmoid activation function, baseline configuration but using "random uniform" weight initialization in the interval [-1, 1[and zero initialization for the biases and L1 regularization of weights and biases with coefficient 0.0001:

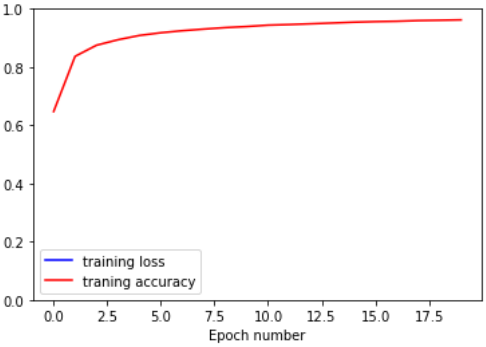
```
weight_init = tfi.RandomUniform(minval=-1.0, maxval=1.0)
bias_init = tfi.Zeros()

regularization_weight = 0.0001
regularizer = tfr.l2(l=regularization_weight)
```

Weight / bias histogram after initialization: *Weight / bias histogram after training:*

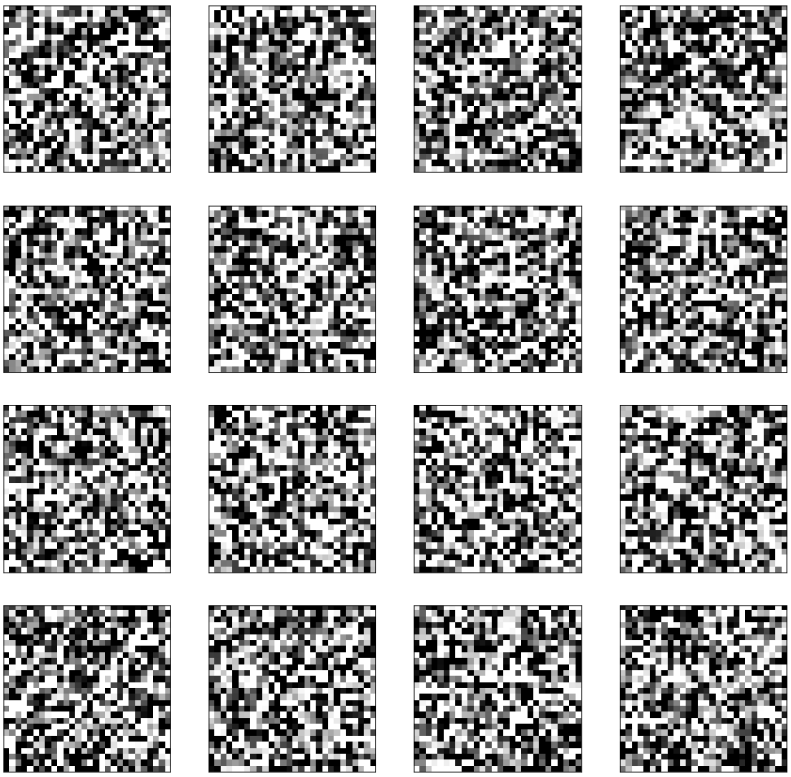


Learning curves:

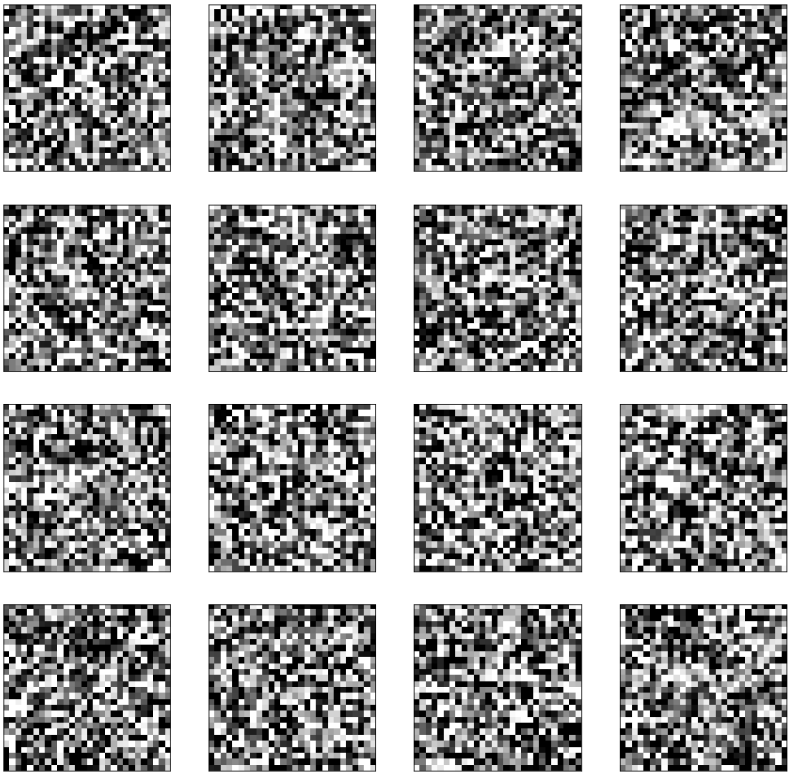



```
final training loss: 2.824241
final training accuracy: 0.962200
final validation loss: 2.822695
final validation accuracy: 0.951200
final test loss: 2.826861
final test accuracy: 0.951900
```

Visualization of the weights between input and some of the hidden neurons of the first hidden layer after initialization:



Visualization of the weights between input and some of the hidden neurons of the first hidden layer after training:



Interpretation:

Batch normalization normalizes the activations at intermediate layers and thus prevents the activations to be in the saturation regime of the sigmoid activation function, thus enabling a successful training.

Using a larger range of values for the initial weights (appropriate for the sigmoid activation function) plus appropriate regularization to decrease the weight and bias values also leads to a successful training. With regularization, the losses are much larger (see statistics above) due to the additional regularization term. L1 regularization "puts more pressure" to make small values even smaller than L2 regularization, which is reflected in the final weight histograms, and leads to larger losses. Note that the regularization coefficient has to be chosen in a suitable range such that the original loss and the regularization term match. Interestingly, the visualization of the final weights does not reveal the stroke-like patterns as before. In, fact it seems that the weights have been modified only minimally (if at all). Still, validation and test accuracy are very large.

Starting from the baseline configuration, dropout (with factor 0.2) or L1-regularization (with factor 0.01 or 0.0001; in both cases using Glorot uniform initialization for the weights and zero initialization for the biases) did not lead to a successful training. Also, decreasing or increasing the learning rate by a factor of 10 did not lead to a successful training.

Combining methods, the following results have been achieved:

Batch normalization plus different weight initialization plus L1-regularization, sigmoid activation: Six hidden layers, sigmoid activation function, baseline configuration but using batch normalization and "random uniform" weight initialization in the interval [-1, 1[and zero initialization for the biases and L1 regularization of weights and biases with coefficient 0.0001:

```
final training loss: 5.356184
final training accuracy: 0.889180
final validation loss: 5.145382
final validation accuracy: 0.942400
final test loss: 5.152021
final test accuracy: 0.939700
```

Batch normalization plus different weight initialization plus L1-regularization, ReLU activation: Six hidden layers, ReLU activation function, baseline configuration but using batch normalization and "random uniform" weight initialization in the interval [-1, 1[and zero initialization for the biases and L1 regularization of weights and biases with coefficient 0.0001:

```
final training loss: 5.376817
final training accuracy: 0.881740
final validation loss: 5.126177
final validation accuracy: 0.949500
final test loss: 5.132136
final test accuracy: 0.944700
```

Thus, the combination of the different methods did not further improve results, neither for the sigmoid nor the ReLU activation function.