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

Neural Networks and Deep Learning – Summer Term 2018

Exercise sheet 4

Submission due: Tuesday, June 05, 11:30 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 learning in neural networks:

Solution:

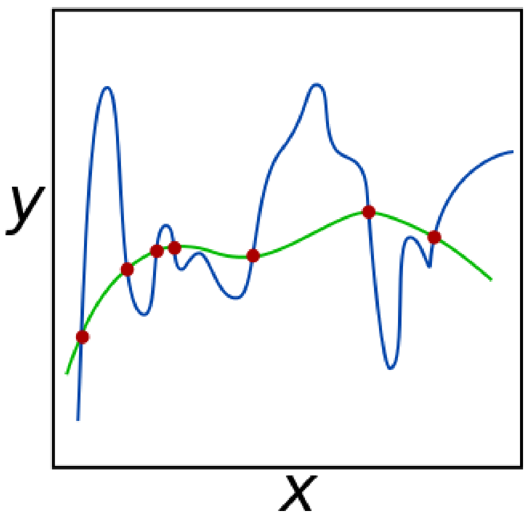
- Loss function
    - In mathematical optimization, statistics, machine learning etc. a loss function (or cost function) is a function that maps an event or values of one or more variables (in our case mostly the outputs of a neural network or learning machine for a set of input samples, the training 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 $x_1, x_2, \ldots, x_n$ (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 **w**, the prior knowledge can be imposed by a penalty on the norm of **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"):
    $R(f) = \Omega(\mathbf{w}) = \sum_{i=1}^{n} |w_i|$ (L1 regularization)
    $R(f) = \Omega(\mathbf{w}) = \sum_{i=1}^{n} w_i^2$ (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)

<u>Mathematical equations:</u>

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

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

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

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

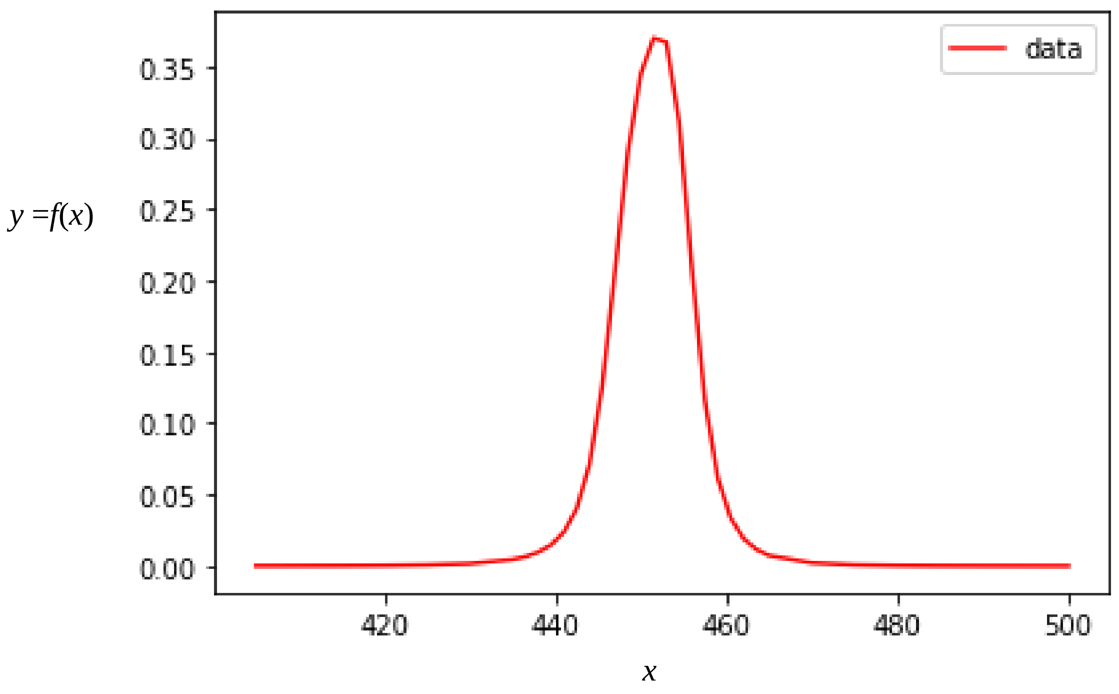
Further activation functions:

- 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 – regression problem):

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:

Eckerle 4 Dataset



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 python file **`exercise2.py`**.

Choose a suitable network topology (number of hidden layers and hidden neurons) and use it for the multi-layer perceptron defined in the file **`exercise2.py`** (see the lines marked with **`# FIX!!!`**). Try to avoid underfitting and overfitting. Furthermore, define a suitable multi-layer perceptron (again see the lines marked with **`# FIX!!!`**). Define and vary important parameters of the multi-layer perceptron in order to achieve a network performance as good as possible. Report on the found network configuration and on your conclusions.

Remark: You may also use any deep learning framework to solve the exercise, e.g. the “Keras” python library with the “Tensorflow” python deep learning libaray, as suggested in http://gonzalopla.com/deep-learning-nonlinear-regression. This allows e.g. to apply stochastic dropout in order to reduce the risk of overfitting. It is however recommended to use a rather small number of hidden layers.

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

Solution:

Replacing the lines

```
# hidden layer sizes
hidden_layer_size = () # FIX!!!
```

in `exercise2.py` by

```
hidden_layer_size = (96,64,64,32)
```

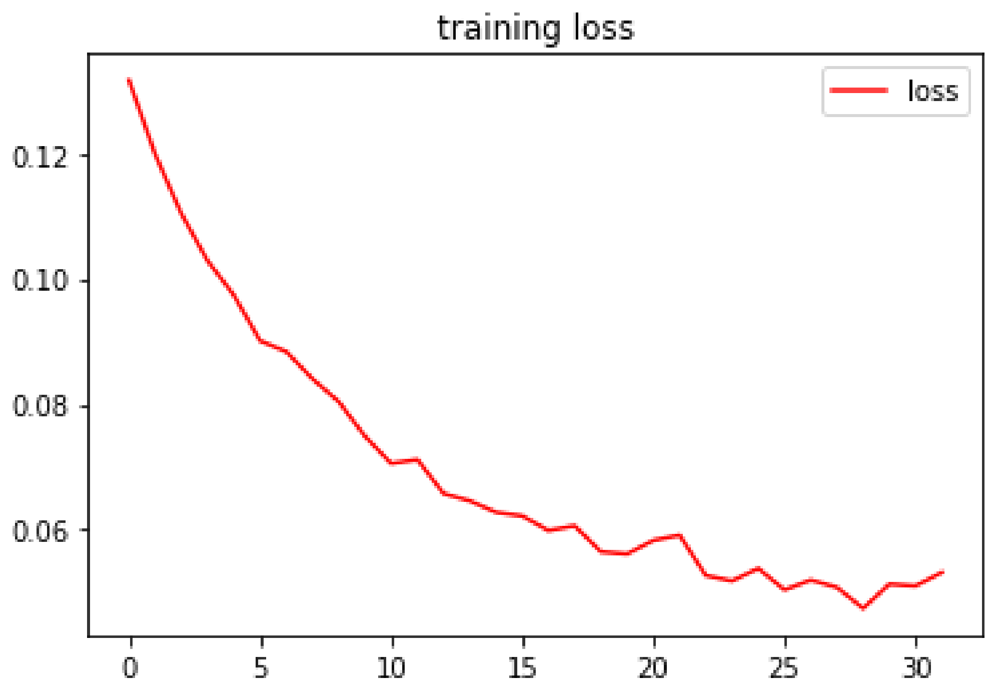
(i.e. a multi-layer perceptron with 4 hidden layers with 96 hidden neurons in the first hidden layer, 64 neurons in the second and third hidden layer and 32 neurons in the fourth hidden layer) and further replacing the lines

```
# construct MLP regressor
mlp = MLPRegressor() # FIX!!!
```

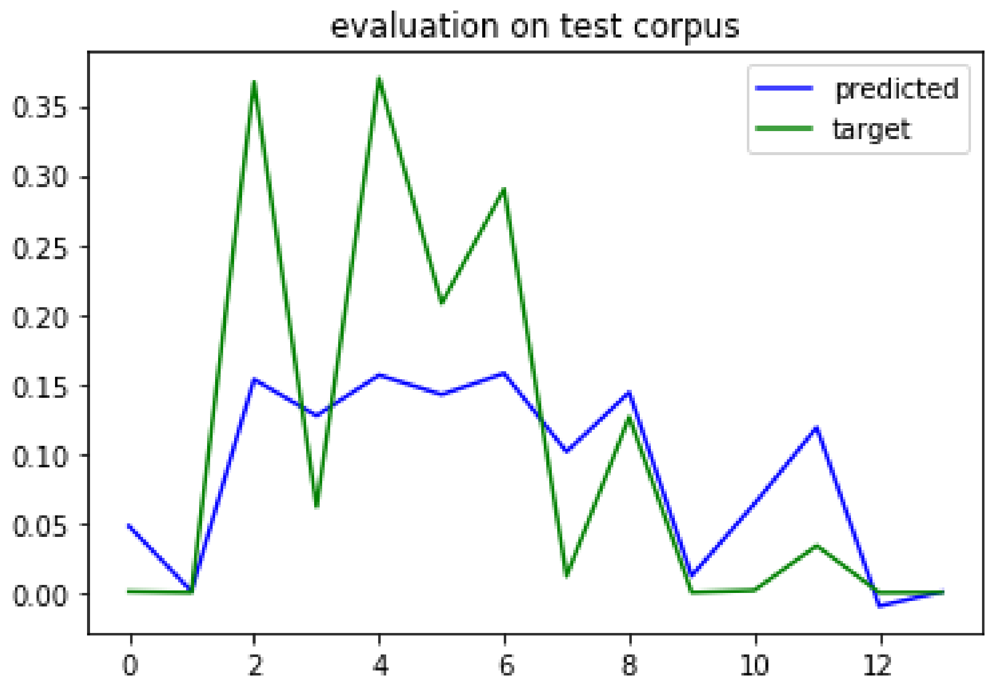
by

```
# construct MLP regressor
mlp = MLPRegressor(hidden_layer_sizes=hidden_layer_size,
alpha=0.002, batch_size=2, verbose=0, random_state=0,
max_iter=256, learning_rate_init=0.001, solver='adam')
```

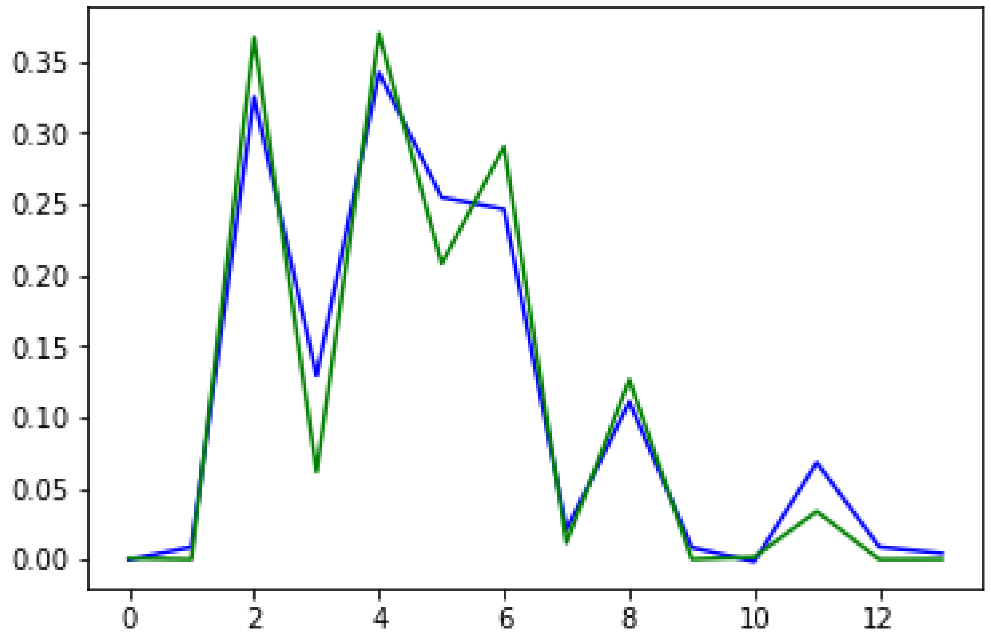
(i.e. using the above-defined topology, settting the regularization coefficient α to 0.002, using a batch size of 2, the initial learning rate to 0.001 and using the “adam” optimization scheme), the following output is obtained:



```
number of network inputs: 1
number of network outputs: 1
number of network layers: 6
Training set score: 0.339641
Training set loss: 0.053189
Test set score: 0.475428
```



It can be seen that this solution is far from optimal (providing a training set score of only about 34.0% and a test set score of 47.5%). The solution suggested in http://gonzalopla.com/deep-learning-nonlinear-regression/
(see **`exercise2_keras.py`**) works much better (even without dropot):



The reason for this deviation is not understood yet. It may be that the sklearn training does not really run for 256 epochs – or any other reasons (hints are welcome...!).

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

In the following exercises, we use the neural network library of **`scikit-learn`** to configure, train and apply a multi-layer perceptron to the problem of recognizing handwritten digits (the famous "MNIST" problem). The MNIST data are provided in **`mnist.pkl.gz`**; they can be loaded using the script **`mnist_loader.py`**.

Perform experiments on this pattern recognition problem trying to investigate the influence of a number of parameters on the classification performance, namely

a) the learning rate,
b) the number of hidden neurons (in a network with a single hidden layer),
c) the number of hidden layers,
d) the solver and learning rate schedule used in optimization,
e) the activation function.

(Further aspects as the loss function and the influence of regularization are addressed in Exercise 4). The script **`exercise3.py`** can serve as a basis or starting point.

Report on and summarize your findings.

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

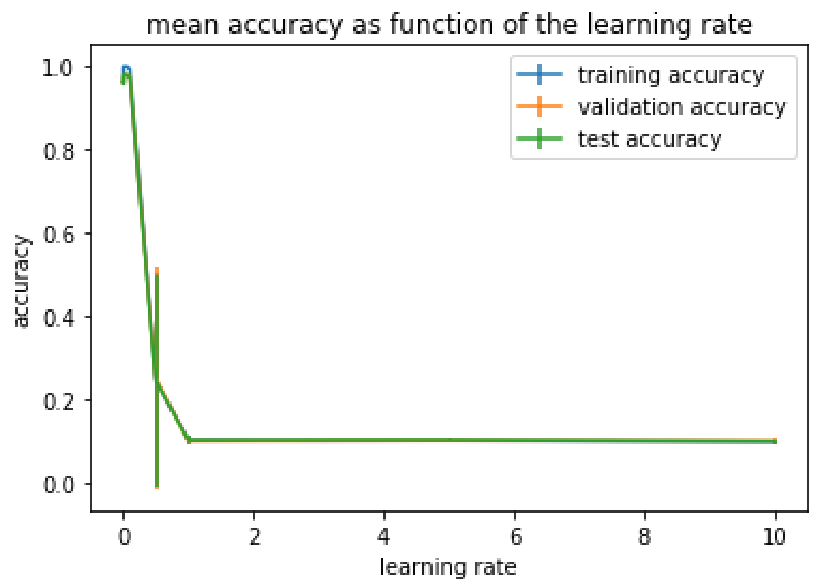
Solution:

a) Running the script **`exercise3_solution.py`**, the following results were obtained:

Modifying the learning rate in a network with two hidden layers and 100 hidden units each, running for 30 epochs:

```
net = nn.MLPClassifier(hidden_layer_sizes=hiddenLayerSizes,
          activation='logistic', solver=solver, alpha=0.0,
          batch_size=batchSize, learning_rate=schedule,
          learning_rate_init=learningRate,max_iter=numEpochs,
          shuffle=False)
```

Learning rate	Training accuracy (mean +/- std)	Validation accuracy mean +/- std	Test accuracy mean +/- std
0.001	0.9632 +/- 0.0004	0.9623 +/- 0.0014	0.9586 +/- 0.0010
0.005	0.9975 +/- 0.0003	0.9758 +/- 0.0010	0.9763 +/- 0.0007
0.010	1.0000 +/- 0.0000	0.9778 +/- 0.0021	0.9779 +/- 0.0007
0.050	1.0000 +/- 0.0000	0.9799 +/- 0.0006	0.9799 +/- 0.0009
0.100	0.9941 +/- 0.0025	0.9734 +/- 0.0016	0.9728 +/- 0.0016
0.500	0.2456 +/- 0.2589	0.2493 +/- 0.2660	0.2431 +/- 0.2547
1.000	0.1005 +/- 0.0084	0.0984 +/- 0.0054	0.1004 +/- 0.0090
3.000	0.0996 +/- 0.0017	0.0991 +/- 0.0025	0.1008 +/- 0.0018
5.000	0.0999 +/- 0.0012	0.0994 +/- 0.0023	0.1008 +/- 0.0030
10.000	0.0976 +/- 0.0049	0.0994 +/- 0.0065	0.0976 +/- 0.0052

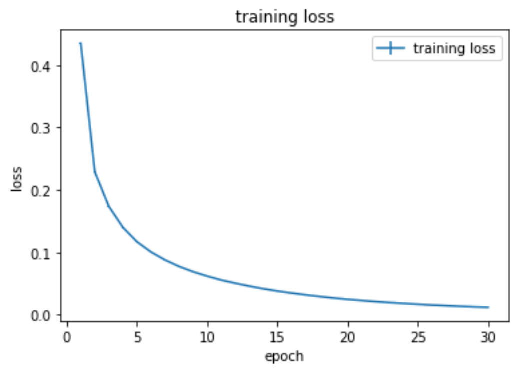


learning rates tested: [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1.0, 3.0, 5.0, 10.0]
mean training accuracy: [0.963215 0.997465 0.99996 1. 0.994125 0.24561 0.100515 0.09958 0.09994 0.097555]
mean validation accuracy: [0.96225 0.975775 0.977825 0.97985 0.97335 0.2493 0.0984 0.0991 0.099425 0.099375]
mean test accuracy: [0.9586 0.976275 0.977925 0.979875 0.97275 0.243075 0.100425 0.100825 0.1008 0.097575]

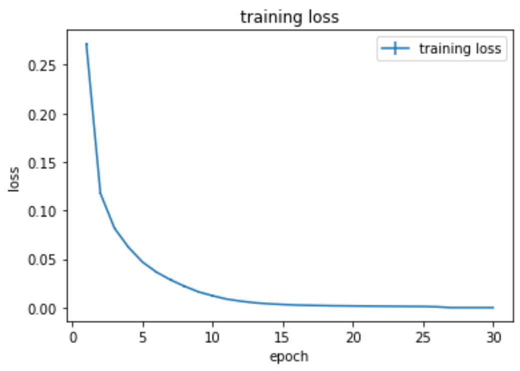
optimal learning rate (determined on validation data): 0.050000

Some plots of the training loss as a function of the epoch number for selected learning rates:

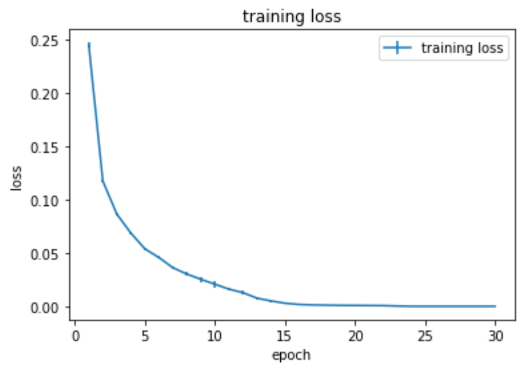
Learning rate 0.01:



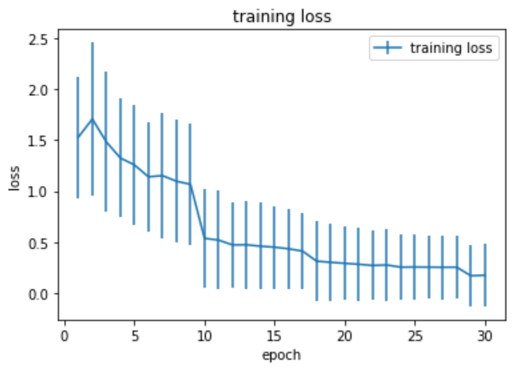
Learning rate 0.05:



Learning rate 0.1:



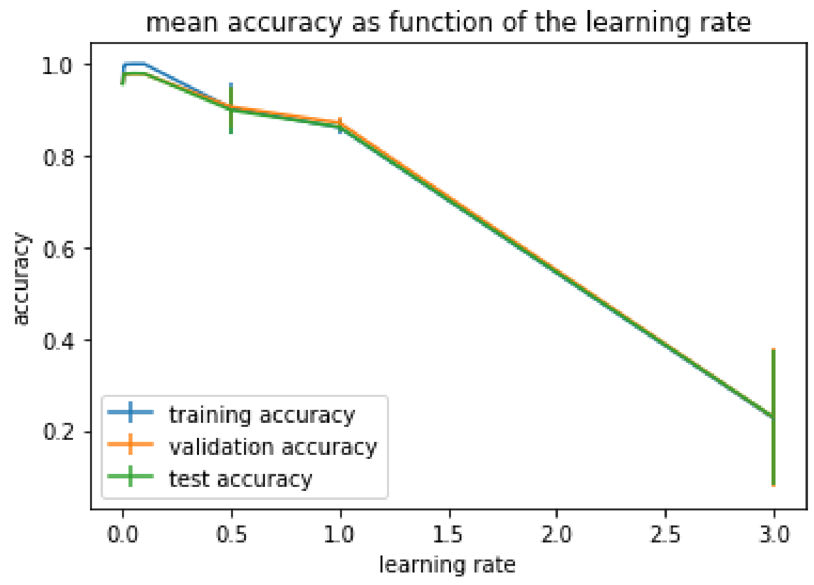
Learning rate 0.5:



Interpretation:
For values of the learning rate smaller than 0.05, convergence is relatively slow (the training loss does not seem to have converged at 30 epochs, so the accuracies are not optimal). For a learning rate of 0.1, some fluctuations (differences between different training runs) in the training loss can be observed between epoch 5 and 10. For larger learning rates these fluctuations become much more pronounced, so the convergence is instable and performance drops significantly (no valid optimum found, training not successful). The learning rate leading to the most stable convergence and the best accuracies is 0.05. Note the similar performance on all three corpora (training, validation and test).

A similar picture is obtained for a network with one hidden layer with 100 hidden units, running for 30 epochs:

Learning rate	Training accuracy (mean +/- std)	Validation accuracy mean +/- std	Test accuracy mean +/- std
0.001	0.9605 +/- 0.0005	0.9603 +/- 0.0005	0.9564 +/- 0.0006
0.005	0.9937 +/- 0.0005	0.9771 +/- 0.0005	0.9765 +/- 0.0003
0.010	0.9992 +/- 0.0001	0.9779 +/- 0.0007	0.9784 +/- 0.0006
0.050	1.0000 +/- 0.0000	0.9787 +/- 0.0005	0.9796 +/- 0.0006
0.100	1.0000 +/- 0.0000	0.9785 +/- 0.0009	0.9790 +/- 0.0008
0.500	0.9033 +/- 0.0541	0.9063 +/- 0.0449	0.8991 +/- 0.0500
1.000	0.8612 +/- 0.0162	0.8716 +/- 0.0123	0.8624 +/- 0.0115
3.000	0.2269 +/- 0.1471	0.2290 +/- 0.1519	0.2292 +/- 0.1471



```
learning rates tested: [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1.0, 3.0]
mean training accuracy: [ 0.9605    0.99371   0.999185  1.        0.999995  0.90329   0.86123
  0.22688 ]
mean validation accuracy: [ 0.96025   0.97705   0.9779    0.978675  0.978525  0.906325
0.8716  0.228975]
mean test accuracy: [ 0.956425  0.97645   0.978425  0.979575  0.979025  0.89905   0.862425
  0.2292  ]
```

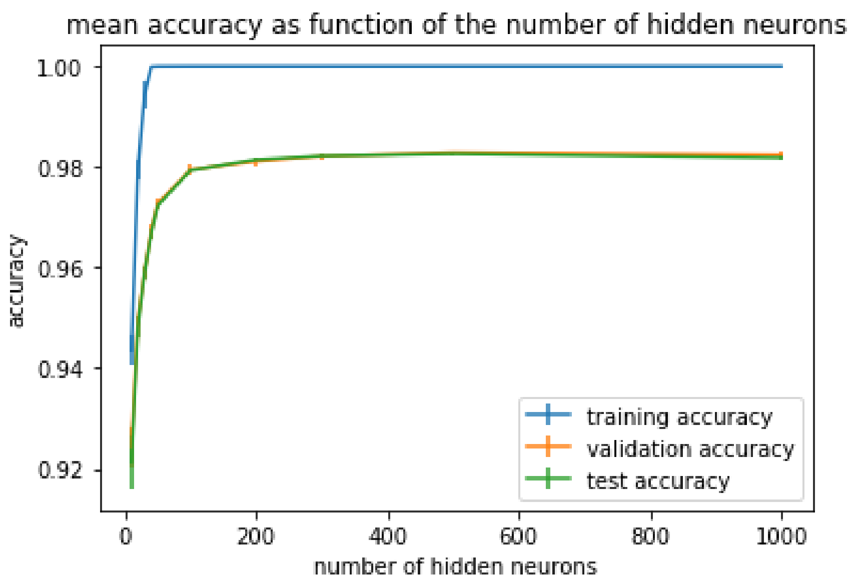
optimal learning rate (determined on validation data): 0.050000

The qualitative picture is comparable (although a larger learning rate, i.e. 0.5, still leads to acceptable performance in case of a single hidden layer).

b) Again running the script **`exercise3_solution.py`**, the following results were obtained:

Modifying the number of hidden neurons in a network with a single hidden layer, learning rate 0.05, running for 100 epochs:

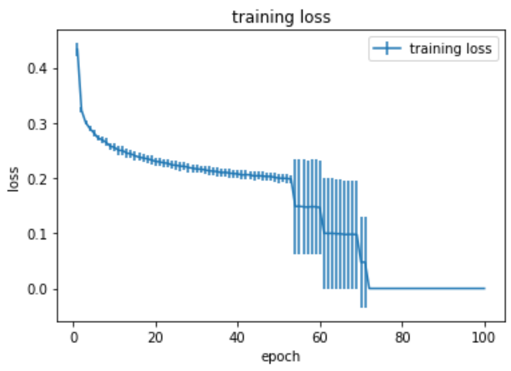
Number of hidden neurons	Training accuracy (mean +/- std)	Validation accuracy mean +/- std	Test accuracy mean +/- std
10	0.9435 +/- 0.0030	0.9244 +/- 0.0040	0.9214 +/- 0.0054
20	0.9796 +/- 0.0019	0.9494 +/- 0.0013	0.9483 +/- 0.0022
30	0.9944 +/- 0.0028	0.9592 +/- 0.0010	0.9590 +/- 0.0013
40	0.9999 +/- 0.0000	0.9673 +/- 0.0015	0.9667 +/- 0.0011
50	1.0000 +/- 0.0000	0.9727 +/- 0.0009	0.9723 +/- 0.0005
100	1.0000 +/- 0.0000	0.9795 +/- 0.0009	0.9794 +/- 0.0005
200	1.0000 +/- 0.0000	0.9812 +/- 0.0008	0.9815 +/- 0.0006
300	1.0000 +/- 0.0000	0.9822 +/- 0.0006	0.9822 +/- 0.0005
500	1.0000 +/- 0.0000	0.9829 +/- 0.0003	0.9827 +/- 0.0004
1000	1.0000 +/- 0.0000	0.9824 +/- 0.0008	0.9819 +/- 0.0004



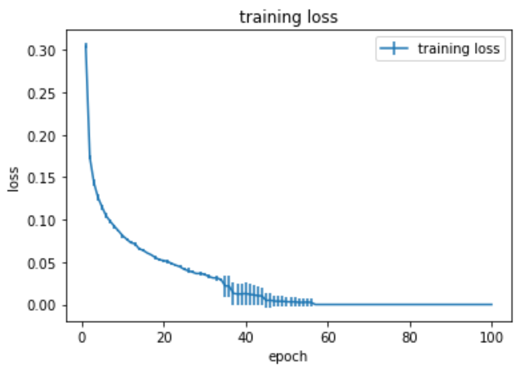
```
number of hidden units tested: [10, 20, 30, 40, 50, 100, 200, 300, 500, 1000]
mean training accuracy: [ 0.94352   0.979605  0.994435  0.999905  0.99998   1.        1.        1.
1.        1.]
mean validation accuracy: [ 0.92435   0.94935   0.95915   0.96725   0.972725  0.979475
0.98115  0.98215   0.982875   0.982425 ]
mean test accuracy: [ 0.921425  0.948325  0.959025  0.966725  0.972325  0.97935   0.98145
  0.982225   0.98265   0.98185 ]

optimal number of hidden neurons (determined on validation data): 500.000000
```

Some plots of the training loss as a function of the epoch number for selected numbers of hidden neurons:

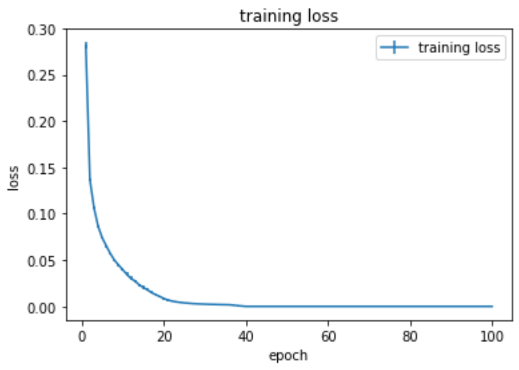
10 hidden neurons:



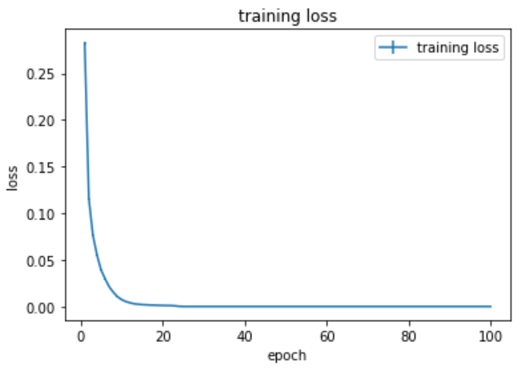
30 hidden neurons:



50 hidden neurons:



300 hidden neurons:

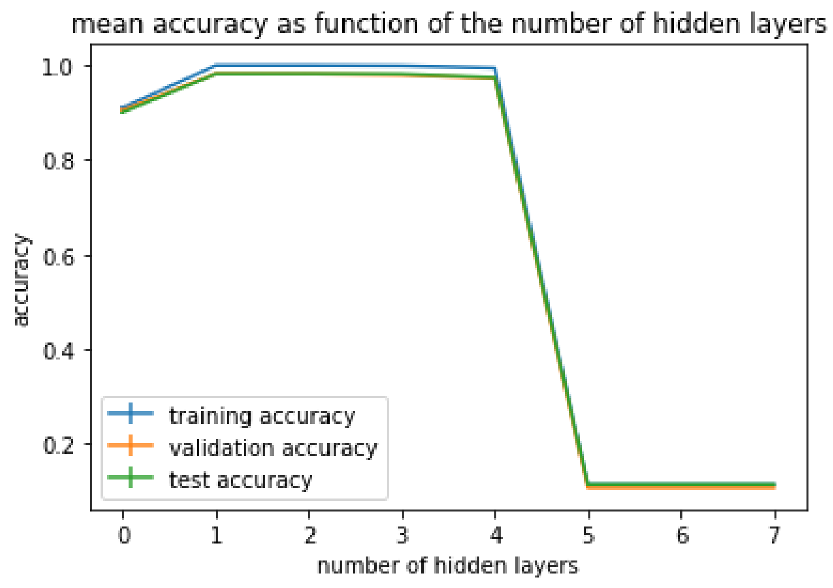


Interpretation:
For fewer hidden neurons (less than 50), convergence is rather slow and accompanied by significant fluctuations (between different training runs). Moreover, only for a larger number of neurons a suitable optimum is found. Increasing the number of hidden neurons, convergence is faster and fluctuations between training runs are negligible. With 50 hidden neurons or more, the training accuracy is 100%, with 200 hidden neurons or more, the validation and test error are above 98%. With 1000 hidden neurons, there is a minimal decrease in accuracy (apart from an increase in runtime with increasing number of hidden neurons).

c) Using the script **`exercise3_solution.py`**, the following results were obtained:

Modifying the number of hidden layers in a network with 500 neurons per hidden layer, learning rate 0.05, running for 50 epochs (for 3 hidden layers or more, only a single iteration has been performed and the number of epochs has been reduced to 30 due to runtime reasons):

Number of hidden layers	Training accuracy (mean +/- std)	Validation accuracy mean +/- std	Test accuracy mean +/- std
0	0.9108 +/- 0.0024	0.9057 +/- 0.0033	0.9011 +/- 0.0023
1	1.0000 +/- 0.0000	0.9828 +/- 0.0005	0.9821 +/- 0.0005
2	1.0000 +/- 0.0000	0.9821 +/- 0.0006	0.9824 +/- 0.0006
3	0.9992	0.9797	0.9818
4	0.9946	0.9731	0.9745
5	0.1136	0.1064	0.1135
6	0.1136	0.1064	0.1135
7	0.1136	0.1064	0.1135

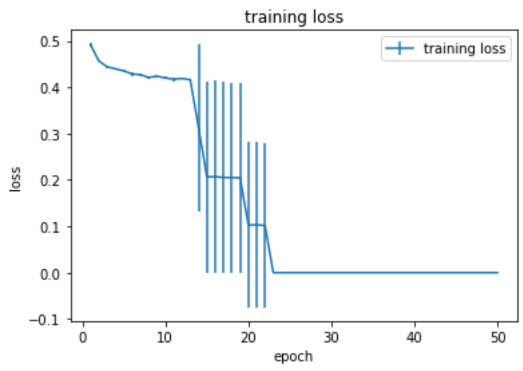


number of hidden layers tested: range(0, 8)
mean training accuracy: [0.910805, 1.0, 1.0, 0.99916, 0.99458, 0.11356, 0.11356, 0.11356]
mean validation accuracy: [0.90565, 0.982775, 0.982075, 0.9797, 0.9731, 0.1064, 0.1064, 0.1064]
mean test accuracy: [0.901075, 0.9821, 0.9824, 0.9818, 0.9745, 0.1135, 0.1135, 0.1135]

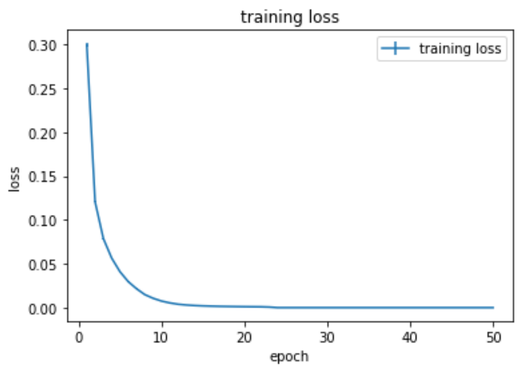
optimal number of hidden layers (determined on validation data): 1

Some plots of the training loss as a function of the epoch number for selected numbers of hidden layers:

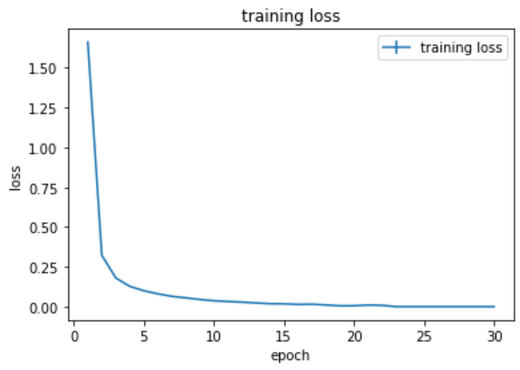
No hidden layer:



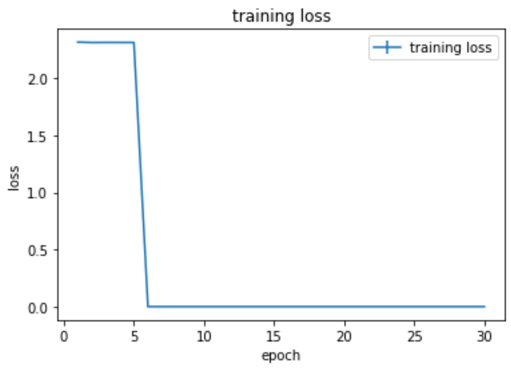
1 hidden layer:



3 hidden layers:



5 hidden layers:



Interpretation:

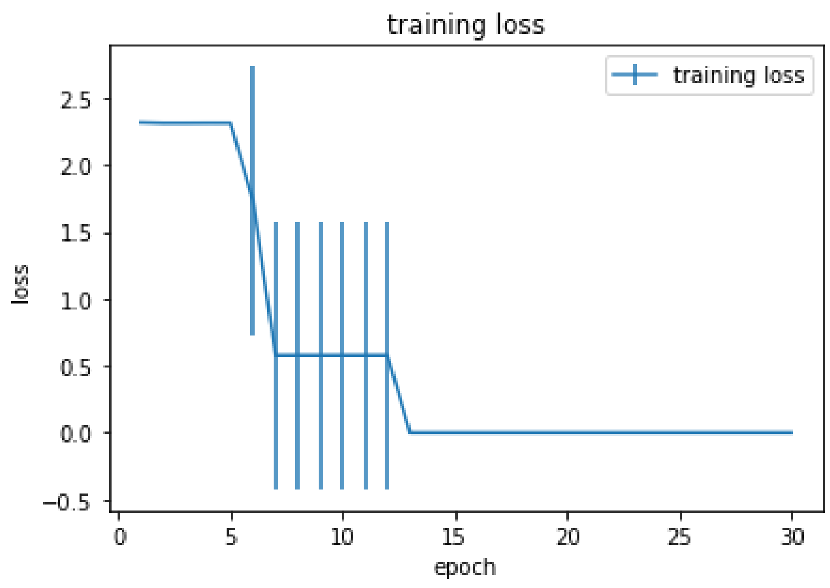
With up to 4 hidden layers, a large test accuracy is obtained. Using no hidden layer is clearly inferior to one or two hidden layers. The maximum performance is obtained with a single hidden layer (using two hidden layers yields comparable performance). Using more than 4 hidden layers results in bad accuracies although the training loss is close to 0.

Repeating the experiment for 5 hidden layers with 500 neurons per hidden layer, learning rate 0.05, running for 50 epochs, four times yields the following output:

training accuracy: 0.111045 +/- 0.004356

validation accuracy: 0.107050 +/- 0.001126

test accuracy: 0.110825 +/- 0.004633



In all four repetitions, the accuracies are very low (in spite of a training loss which is nearly 0). Thus, for the given settings, the network does not find suitable weights and thresholds and learning is not successful.

d) Modifying default settings in the script **exercise3_solution.py**, the following results were obtained in a network with a single hidden layer with 500 neurons, logistic activation function, initial learning rate 0.05, running for 50 epochs:

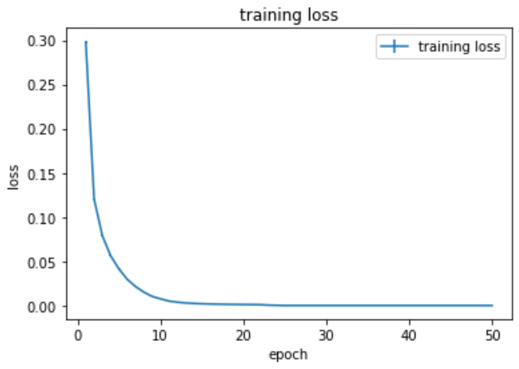
```
  net = nn.MLPClassifier(hidden_layer_sizes=hiddenLayerSizes,
                 activation='logistic', solver=solver, alpha=0.0,
                 batch_size=batchSize, learning_rate=schedule,
                 learning_rate_init=learningRate,
max_iter=numEpochs,
                 shuffle=False)
```

Solver	Schedule	Training accuracy (mean +/- std)	Validation accuracy mean +/- std	Test accuracy mean +/- std
sgd	constant	1.0000 +/- 0.0000	0.9823 +/- 0.0002	0.9819 +/- 0.0004
sgd	const, momentum	1.0000 +/- 0.0000	0.9828 +/- 0.0006	0.9826 +/- 0.0006
sgd	const, Nest. mom.	1.0000 +/- 0.0000	0.9826 +/- 0.0007	0.9824 +/- 0.0005
sgd	invscaling	0.9698 +/- 0.0004	0.9692 +/- 0.0005	0.9660 +/- 0.0007
sgd	adaptive	1.0000 +/- 0.0000	0.9833 +/- 0.0006	0.9828 +/- 0.0003
sgd	adaptive, mom.	1.0000 +/- 0.0000	0.9831 +/- 0.0008	0.9822 +/- 0.0006
sgd	adap., Nest. mom.	1.0000 +/- 0.0000	0.9825 +/- 0.0006	0.9821 +/- 0.0006
lbfgs	constant	0.9337 +/- 0.0054	0.9390 +/- 0.0042	0.9352 +/- 0.0039
lbfgs	invscaling	0.9346 +/- 0.0033	0.9387 +/- 0.0008	0.9363 +/- 0.0022
lbfgs	adaptive	0.9332 +/- 0.0060	0.9373 +/- 0.0048	0.9343 +/- 0.0043
adam	constant	0.8552 +/- 0.0197	0.8647 +/- 0.0155	0.8532 +/- 0.0181
adam	invscaling	0.8530 +/- 0.0219	0.8655 +/- 0.0193	0.8522 +/- 0.0217
adam	adaptive	0.8704 +/- 0.0246	0.8800 +/- 0.0218	0.8699 +/- 0.0275

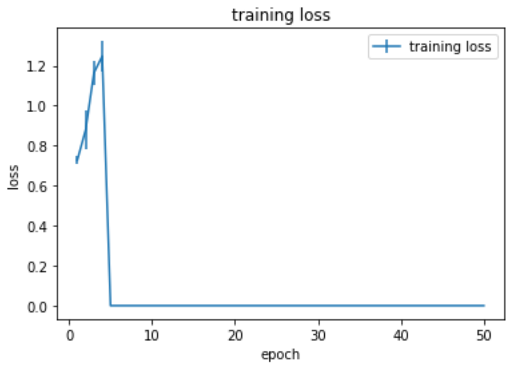
(“const.” = constant, momentum = 0.9, “Nest. mom.” = Nesterov momentum)

Some plots of the training loss as a function of the epoch number for selected numbers of hidden layers:

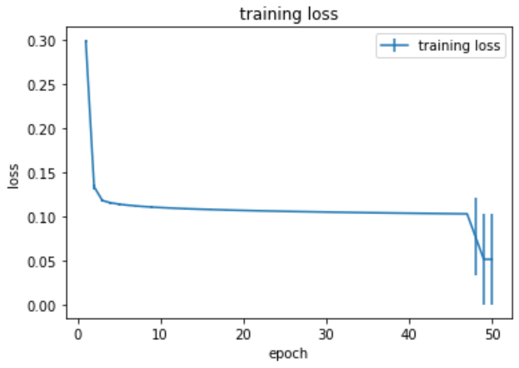
sgd, constant schedule:



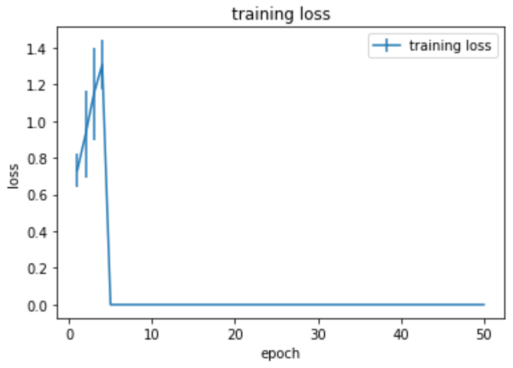
adam, constant schedule:



sgd, invscaling schedule:



adam, adaptive schedule:



Interpretation:

For this task, stochastic gradient descent with either constant or adaptive schedule works best yielding a smooth decrease of the training loss and best accuracies. (Nesterov) momentum does not improve results in this example. Stochastic gradient descent with invscaling schedule is clearly inferior. The adam and lbfgs solver yield suboptimal accuracies, although the training loss might decrease to 0.

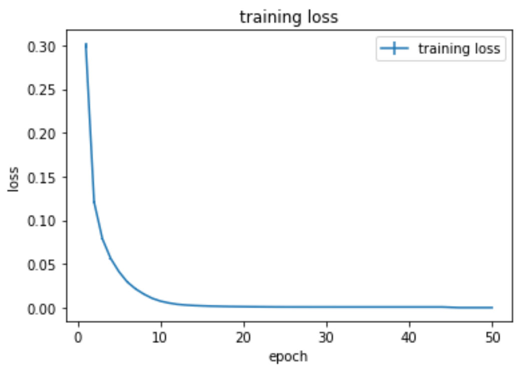
e) Modifying default settings in the script **`exercise3_solution.py`**, the following results were obtained in a network with a single hidden layer with 500 neurons, stochastic gradient descent with adaptive schedule, initial learning rate 0.05, running for 50 epochs:

```
net = nn.MLPClassifier(hidden_layer_sizes=hiddenLayerSizes,
         activation='logistic', solver=solver, alpha=0.0,
         batch_size=batchSize, learning_rate=schedule,
         learning_rate_init=learningRate,max_iter=numEpochs,
         shuffle=False)
```

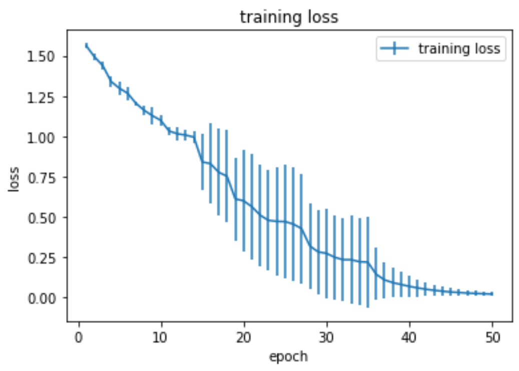
Activation function	Training accuracy (mean +/- std)	Validation accuracy mean +/- std	Test accuracy mean +/- std
logistic	1.0000 +/- 0.0000	0.9825 +/- 0.0005	0.9823 +/- 0.0004
tanh	0.9966 +/- 0.0017	0.9642 +/- 0.0013	0.9626 +/- 0.0025
relu	0.9995 +/- 0.0003	0.9768 +/- 0.0010	0.9771 +/- 0.0009
identity	0.8986 +/- 0.0066	0.8971 +/- 0.0047	0.8927 +/- 0.0059

Some plots of the training loss as a function of the epoch number for selected numbers of hidden layers:

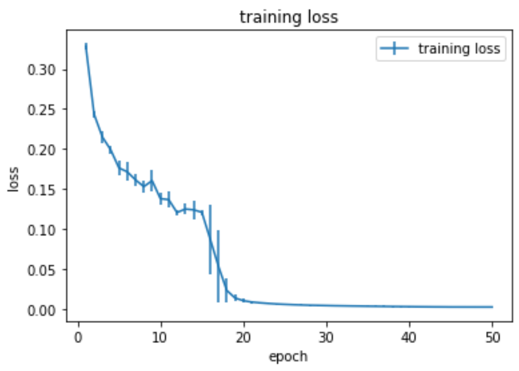
logistic activation:



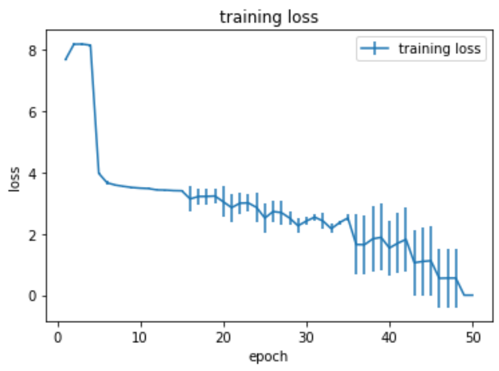
tanh activation:



relu activation:



identity activation:



Interpretation:

For this task, the logistic activation function yields the best performance and fastest convergence. Relu and tanh follow with slightly inferior performance, but convergence is slower and (in case of tanh) accompanied with large fluctuations between different training runs. The identity activation function doesn’t work (as can be expected).

Exercise 4 (Vanishing gradient):

a) The script **`exercise4.py`** implements a multi-layer perceptron for use on the MNIST digit classification problem. Apart from the training loss, it also displays a histogram of the weights (between the input and the first hidden layer) after 1 training epoch 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 logistic activation function, compare the output for a single hidden layer, four and five hidden layers. Then change to a ReLU activation function and inspect the results for five hidden layers. Discuss your findings.

Solution:

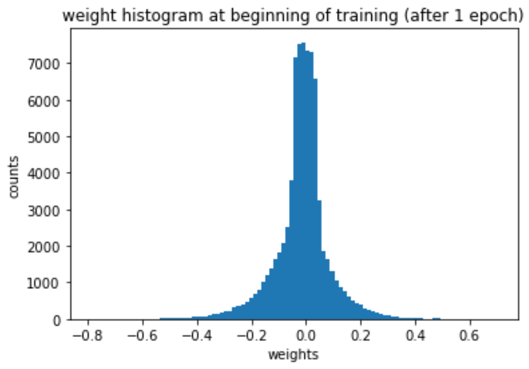
Interpretation of the plots (which follow on the next pages):

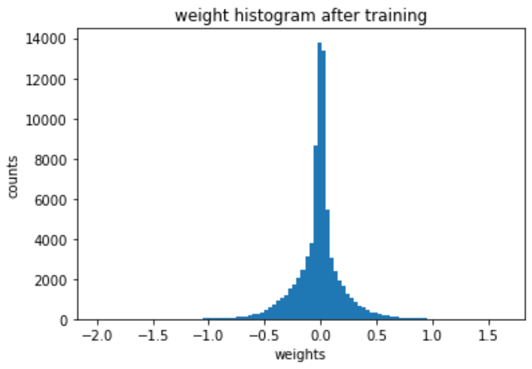
For a single or four hidden layers (with 100 hidden neurons each), training succeeds (as in exercise 1). Comparing a single and four hidden layers, after one training epoch the range of the synaptic weights (between input and first hidden layer) is already larger than for four hidden layers, indicating that training of these weights is faster than with four hidden layers. This is also supported by the graph of the training loss (note the different scaling of the ordinate and the abscissa). The histogram of the weights after training suggests that the weights with four hidden layers are on average slightly larger than with a single hidden layer. The graphical representation of the weights is hard to compare between a single and four hidden layers.
Using five hidden layers, training does not succeed. This is seen in the values of the training loss and the training, validation and test accuracies. The histogram of the weights (between input and hidden layer) suggests that hardly any training occurs, which is also supported by the graphical representation of the weights which look completely random. A similar picture is obtained when using seven hidden layers.
Switching to the ReLU activation function, the accuracies are still very low (and the training loss large). The weight histograms suggest, however, that some training takes place. This is alos supported by the graphical representation of the weights (between the input and the first hidden layer). It seems that some inappropriate local minimum has been found.

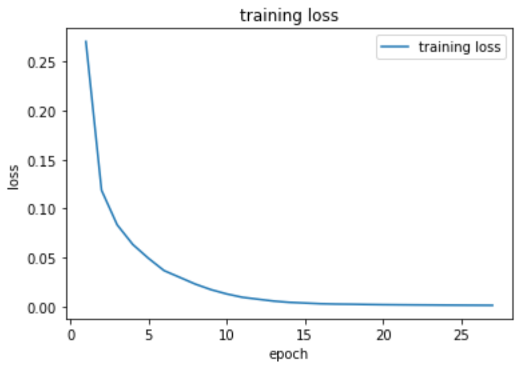
(Following pages: Plots of the results of the experiments)

Considering a network with a single, four and five hidden layer(s) with 100 hidden units (each), running for maximally 100 epochs, logistic activation function, stochastic gradient descent with constant schedule, learning rate 0.05:

Single hidden layer, logistic activation function:

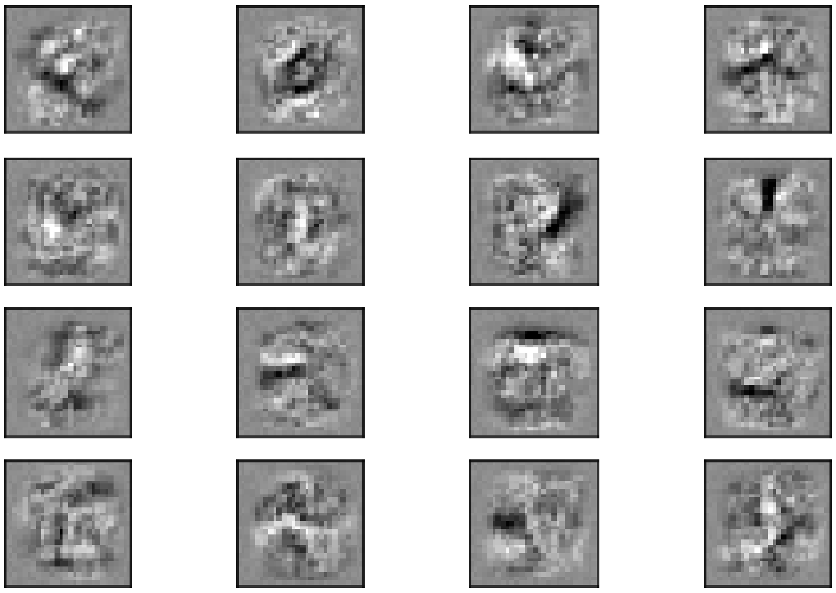




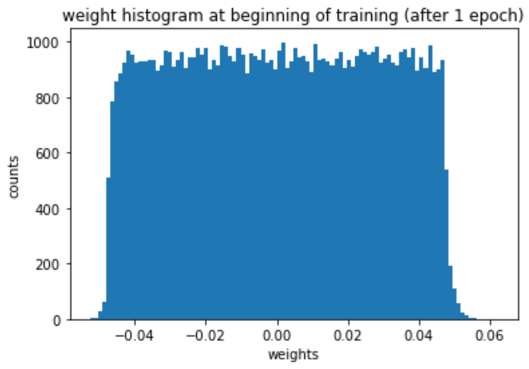


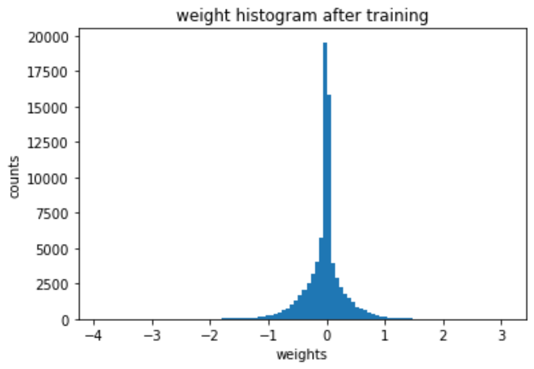
training accuracy: 1.000000
validation accuracy: 0.978200
test accuracy: 0.980600

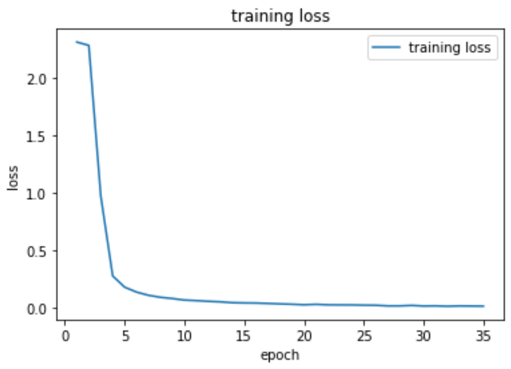
Visualization of the weights between input and some of the hidden neurons of the first hidden layer:



Four hidden layers, logistic activation function:

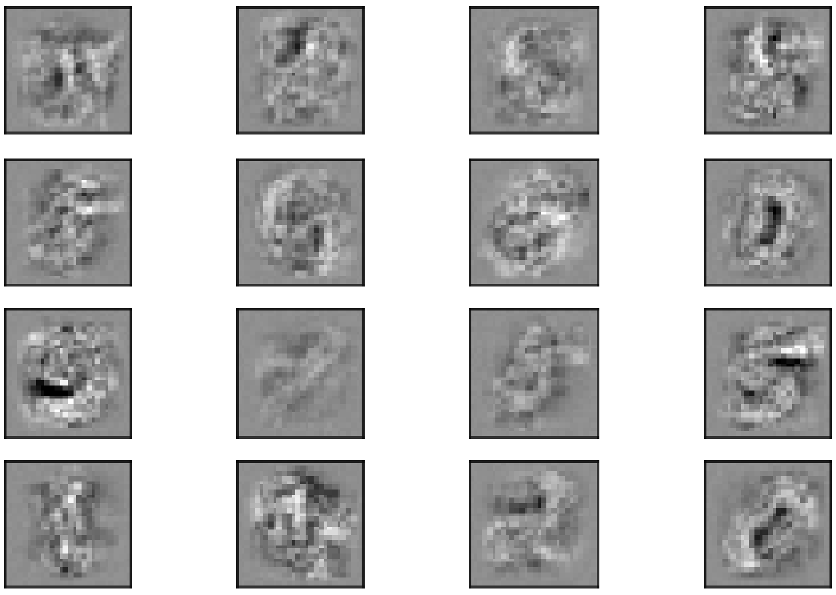




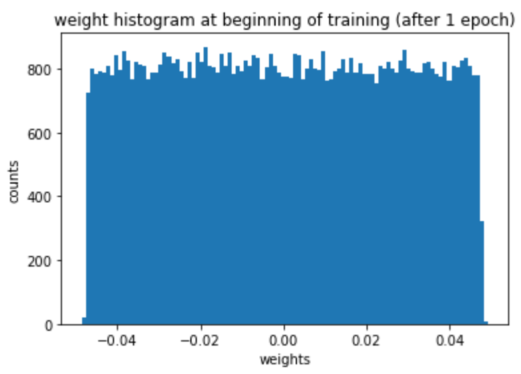


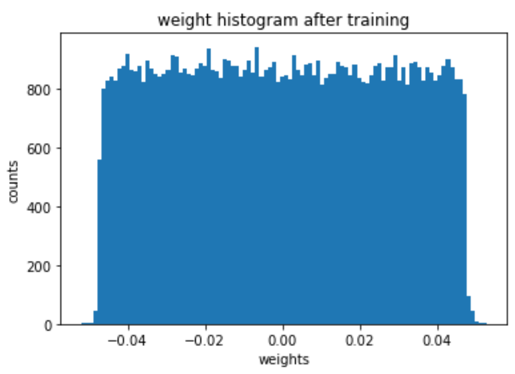
training accuracy: 0.995360
validation accuracy: 0.970600
test accuracy: 0.972100

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

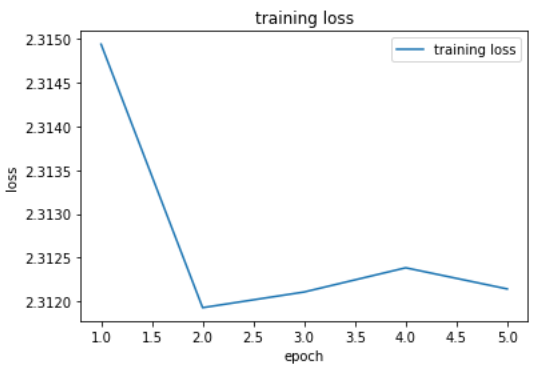


Five hidden layers, logistic activation function:

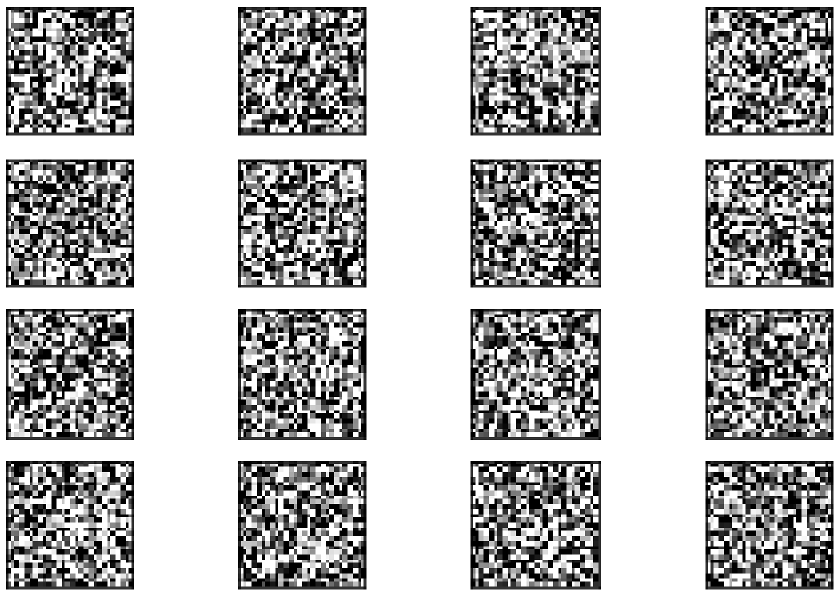




training accuracy: 0.099360
validation accuracy: 0.099000
test accuracy: 0.103200

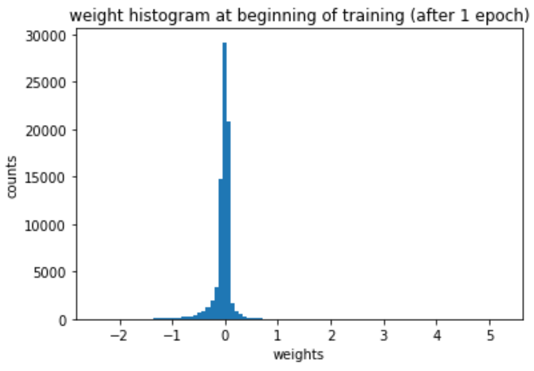


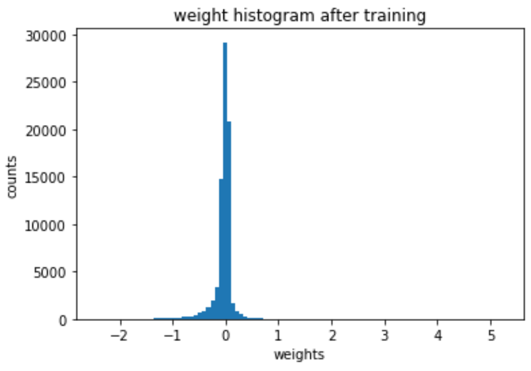
Visualization of the weights between input and some of the hidden neurons of the first hidden layer:

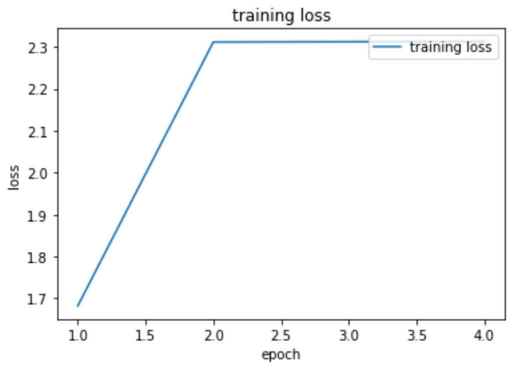


Replacing the logistic activation function by the ReLU activation function:

Five hidden layers, ReLU activation function:

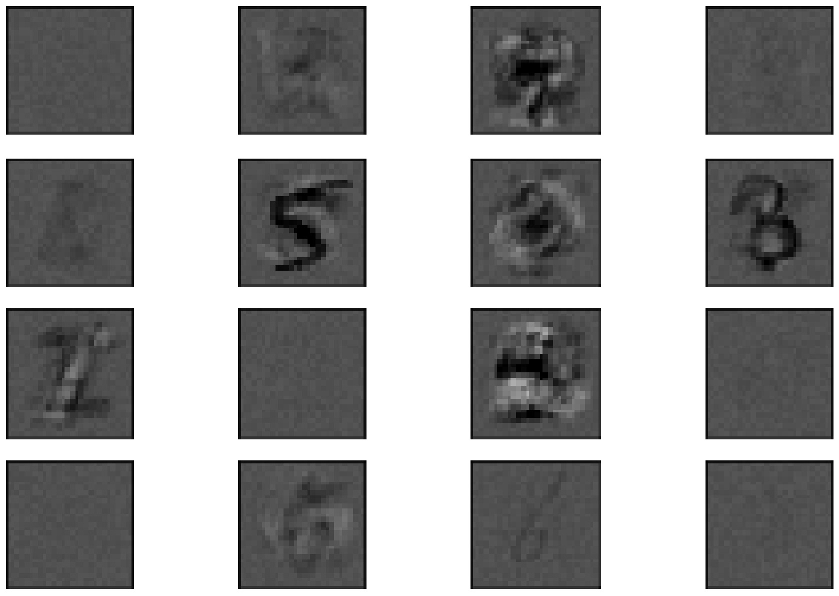






training accuracy: 0.099020
validation accuracy: 0.096700
test accuracy: 0.095800

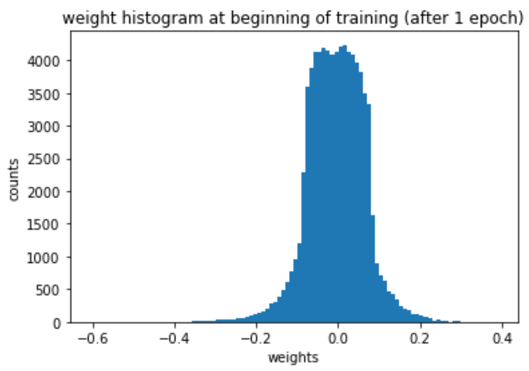
Visualization of the weights between input and some of the hidden neurons of the first hidden layer:

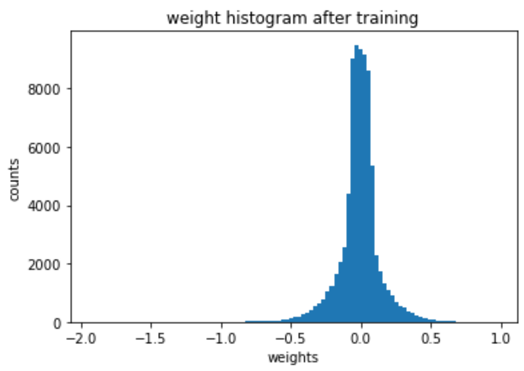


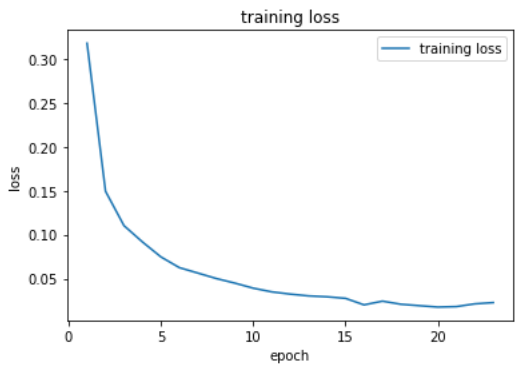
b) Starting from your analysis for the multi-layer perceptron with five hidden layers and ReLU activation function, modify parameters such that the training becomes successful.

Solution:
Considering a network with five hidden layers with 100 hidden units (each), running for maximally 100 epochs, ReLU activation function, stochastic gradient descent with constant schedule:

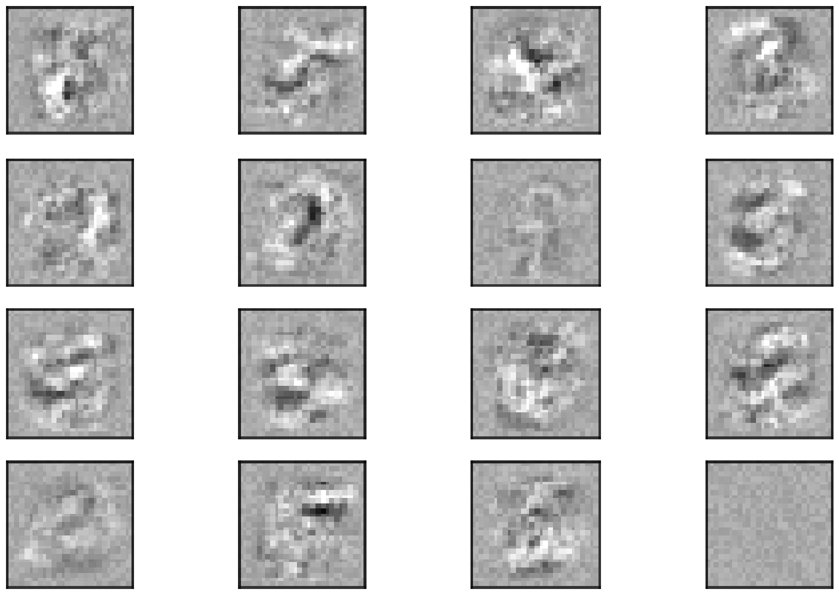
i. Decrease learning rate e.g. to 0.01 (instead of 0.05):





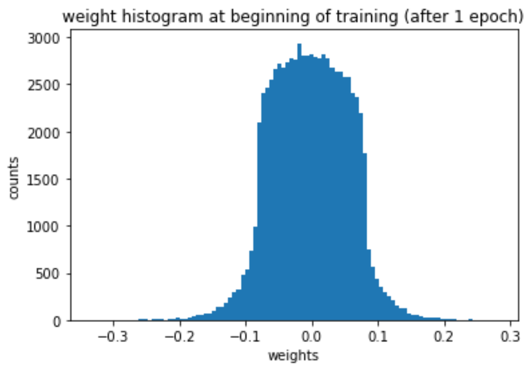


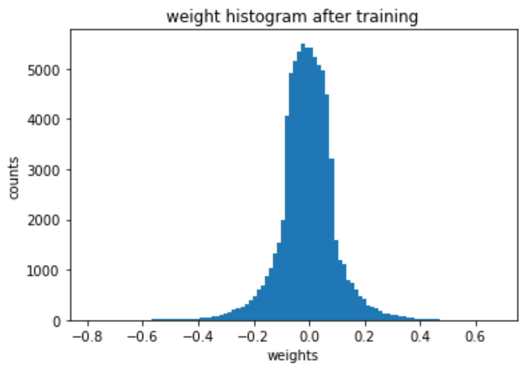
training accuracy: 0.995940
validation accuracy: 0.980300
test accuracy: 0.976200



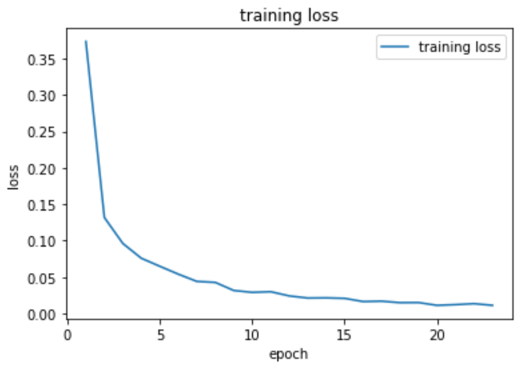
Note that in case of a logistic activation function, a learning rate of 0.01 did not lead to a successful training in my experiments.

ii. Increase batch size to 100 (instead of 10), using learning rate of 0.05:

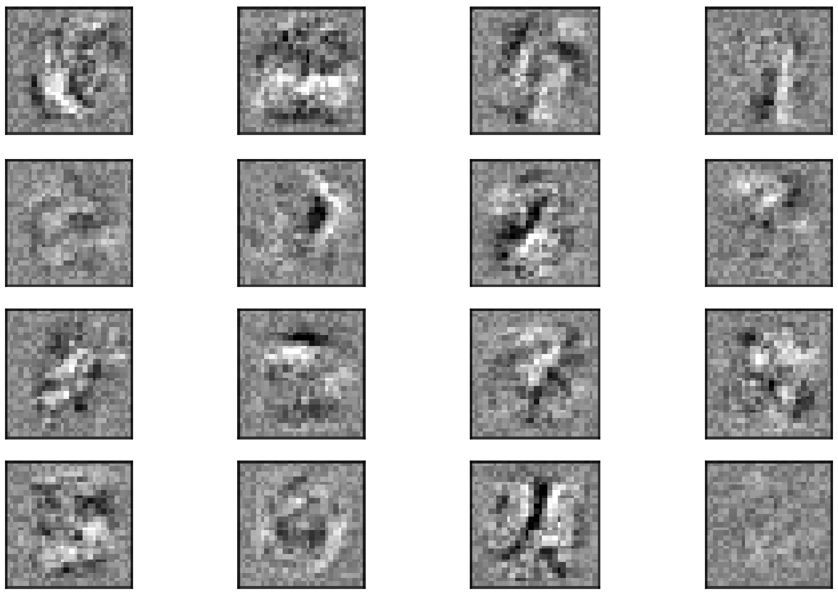




training accuracy: 0.998260
validation accuracy: 0.980000
test accuracy: 0.979700



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



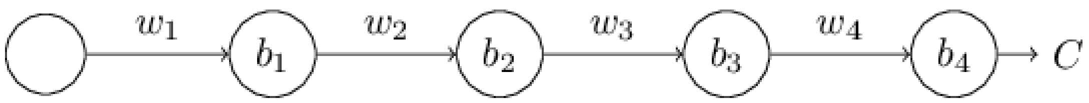
Increasing the batch size also leads to a faster training.
Again, using the logistic activation function, training was not successful with these parameter settings.

c) 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?

(In case of an errror message just re-run the script.)

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. $\sigma(z) = \frac{1}{1+e^{-z}}$.

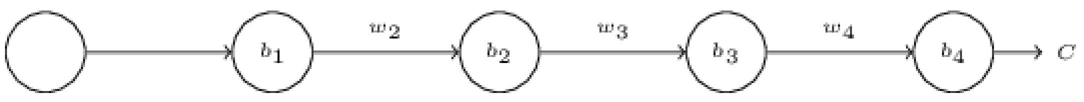
For the activation a_1 we have (using an input $x = a_0$): $a_1 = \sigma(z_1) = \sigma(w_1 a_0 + b_1)$.

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: $\Delta z_2 \approx \frac{\partial z_2}{\partial a_1}\Delta a_1 = w_2 \Delta a_1 \approx \sigma'(z_1) w_2 \Delta b_1$.

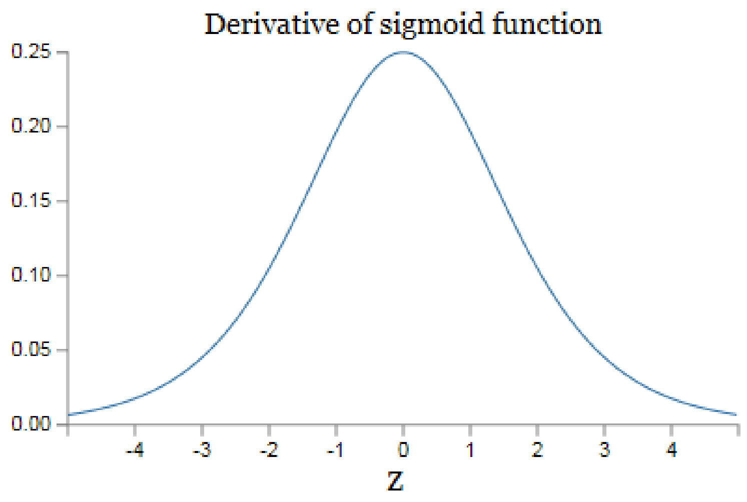
This then modifies the activation a_2, which in turn leads to a change in the postsynaptic potential z_3, etc. Finally, we obtain: $\Delta C \approx \sigma'(z_1) w_2 \cdot \sigma'(z_2) w_3 \cdot \sigma'(z_3) w_4 \sigma'(z_4) \frac{\partial C}{\partial a_4}\Delta b_1$.

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



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:

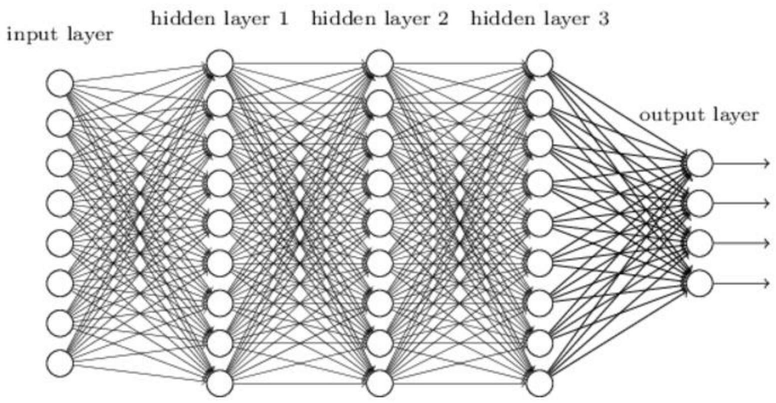
$$\frac{\partial C}{\partial b_1} = \sigma'(z_1)\,\overbrace{w_2\sigma'(z_2)}^{<\frac{1}{4}}\,\overbrace{w_3\sigma'(z_3)}^{<\frac{1}{4}}\,\underbrace{w_4\sigma'(z_4)\frac{\partial C}{\partial a_4}}$$

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

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

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.