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

Neural Networks and Deep Learning – Summer Term 2018

Exercise sheet 3

Submission due: Tuesday, May 22, 11:30 sharp

Note:

Some of the following exercises are based on the "scikit learn" python library, a set of python modules for machine learning and data mining (see www.scikits.appspot.com/learn).

This can easily be installed using the Anaconda package (scikit-learn is already included in the Anaconda distribution):

1.) Download Anaconda from (caution: download takes quite a while!)
https://www.continuum.io/downloads
(Python 3.5) → Anaconda3-4.0.0-Windows-x86_64.exe

2.) Install Anaconda by double-clicking the executable

Remark:
A **S**cientific **PY**thon **D**evelopment **E**nvi**R**onment ("Spyder") is included in the Anaconda package.

3.) From the start menu, start an Jupyter QTconsole (Anaconda3 → Jupyter QTConsole)

Further documentation on SciKit can be found at:

http://scikit-learn.org/stable/install.html

Exercise 1 (Learning in neural networks):

a) Explain the following terms related to neural networks:

Solution:

- Learning in neural networks
  - In the context of neural networks, „learning“ refers to specifying the organization of the network (connectivity, neuronal elements, parameters etc.) in such a way that a desired network response is achieved for a given set of input patterns (the „*training set*“). Often, the term “learning” is further restricted to specify only the adjustment of the *weights* and the *thresholds* of the neurons in the network.

- Training set
  - The training set is the set of input patterns (“examples”) used for learning, i.e. used to specify the weights and thresholds of the network.

- Supervised learning
  - Learning mode where – apart from the input patterns – also the correct output (“target”) for each input pattern is presented to the network.

- Unsupervised learning
  - Learning mode where only the input patterns are presented to the network, but not the desired target output.

- Online (incremental) learning
  - Learning mode where learning (i.e. modification of the weights and thresholds) is done after presentation of each individual training sample to the network.

- Offline (batch) learning
  - Learning mode where learning (i.e. modification of the weights and thresholds) is done only after presentation of the whole set of training samples to the network.

- Training error
  - Quantity measuring the difference between the actual network output and the target output, *calculated on the set of training examples.*

- Generalisation error
  - Quantity measuring the difference between the actual network output and the target output, *ideally calculated on all possible input patterns.* In practice, the generalization error is approximated by representing the set of all possible input patterns by an appropriately chosen set of representative input patterns (different from the training set).

- Overfitting
  - Phenomenon in learning where the network learns some details of individual training patterns which are not relevant for most of the remaining patterns (learning of “noise” instead of the “signal”).

- Cross-validation
    - Learning method where an available (annotated) data set is split into a training set used for learning (i.e. for estimation of thresholds and weights) and an *independent* test set for evaluation. The core idea of cross-validation is to use a small fraction of the data for testing and the remaining fraction for training, and to circulate the role of the test data among all available data – performing a new training run on each new definition of training and test data. In this way, training and test data are used efficiently (each element of the available data is used for testing, and the extreme case of leaving-one-out training is performed on all available elements of the data except one). The final test performance is calculated by a weighted average on all test results. Note that the separation into training and test data must be such that all created training and test corpora are representative of the problem (so be careful with "ordered" data) and be independent (so e.g. in case of multiple images of a person, one must perform "leaving-one-person-out" instead of "leaving-one-image-out"). If a further independent validation set is needed to optimize meta-parameters (e.g. the learning rate), the available data have to be split into three sets (training, validation and test set).

b) Name and briefly describe some methods to indicate or avoid overfitting when training neural networks.

Solution:

- Early stopping
    - Method to stop an iterative training process when the "validation error" – i.e. the error on an independent validation set, as an estimate for the generalization error – starts to increase (while the training error might continue to decrease even with further iterations). Therefore, early stopping is mostly applied using cross-validation.

- Regularisation
    - Restricting the number of degrees of freedom of a model (i.e. of the network) in order to have a more meaningful estimate for the parameters to be learned. Example: Limit the number of weights or layers, use the same weight parameter for multiple connections…)

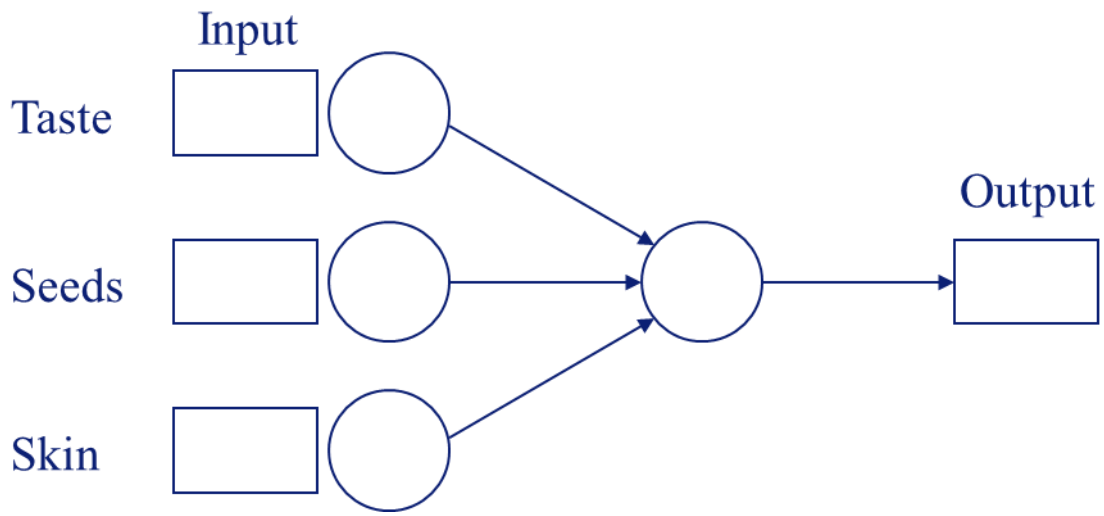
- Cross-validation
    - Description: see above. By estimating the generalization error on *independent* test data, cross-validation can indicate overfitting. To avoid overfitting, early stopping can be used (see above).

Exercise 2 (Perceptron learning – analytical calculation):

The goal of this exercise is to train a single-layer perceptron (threshold element) to classify whether a fruit presented to the perceptron is going to be liked by a certain person or not, based on three features attributed to the presented fruit: its taste (whether it is sweet or not), its seeds (whether they are edible or not) and its skin (whether it is edible or not). This generates the following table for the inputs and the target output of the perceptron:

Fruit	Perceptron input (features of the fruit)			Target output person likes = 1, doesn't like = 0
	Taste sweet = 1, not sweet = 0	Seeds edible = 1, not edible = 0	Skin edible = 1, not edible = 0	
Banana	1	1	0	1
Pear	1	0	1	1
Lemon	0	0	0	0
Strawberry	1	1	1	1
Green apple	0	0	1	0

Since there are three (binary) input values (taste, seeds and skin) and one (binary) target output, we will construct a single-layer perceptron with three inputs and one output.



Since the target output is binary, we will use the perceptron learning algorithm to construct the weights.

To start the perceptron learning algorithm, we have to initialize the weights and the threshold. Since we have no prior knowledge on the solution, we will assume that all weights are 0 ($w_1 = w_2 = w_3 = 0$) and that the threshold is $\theta = 1$ (i.e. $w_0 = -\theta = -1$). Furthermore, we have to specify the learning rate η. Since we want it to be large enough that learning happens in a reasonable amount of time, but small enough so that it doesn't go too fast, we set $\eta = 0.25$.

Apply the perceptron learning algorithm – in the incremental mode – analytically to this problem, i.e. calculate the new weights and threshold after successively presenting a banana, pear, lemon, strawberry and a green apple to the network (in this order).

Draw a diagram of the final perceptron indicating the weight and threshold parameters and verify that the final perceptron classifies all training examples correctly.

Note: The iteration of the perceptron learning algorithm is easily accomplished by filling in the following table for each iteration of the learning algorithm:

μ =1; current training sample: banana						
Input $\mathbf{x}^{(\mu)}$	Current weights $\mathbf{w}(t)$	Network Output $y^{(\mu)}$	Target output $d^{(\mu)}$	Learning rate η	Weight update $\Delta\mathbf{w}(t)$	New Weights $\mathbf{w}(t+1)$
$x_0 = 1$	$w_0 =$			0.25		
$x_1 =$	$w_1 =$					
$x_2 =$	$w_2 =$					
$x_3 =$	$w_3 =$					

(Source of exercise: Langston, Cognitive Psychology)

Solution:

The perceptron learning algorithm (incremental mode) is given by

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \eta \cdot \left(d^{(\mu)} - y^{(\mu)}\right) \cdot \mathbf{x}^{(\mu)} \tag{1}$$

where $\mathbf{w}(t)$ is the weight vector at iteration t, η is the learning rate, $\mathbf{x}^{(\mu)}$ is the current input (training) sample presented to the network, $d^{(\mu)}$ is the target output for that training sample and $y^{(\mu)}$ is the actual network output for the current training sample.

Since we have a threshold element, the output of the perceptron is given by

$$y = \Theta\left[\sum_{i=1}^{3} x_i \cdot w_i - \theta\right] = \Theta\left[\mathbf{x} \cdot \mathbf{w}\right] \tag{2}$$

where we use the bias $w_0 = -\theta$ corresponding to the input $x_0 = 1$:

$$\mathbf{x} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad ; \quad \mathbf{w} = \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{pmatrix} = \begin{pmatrix} -\theta \\ w_1 \\ w_2 \\ w_3 \end{pmatrix}$$

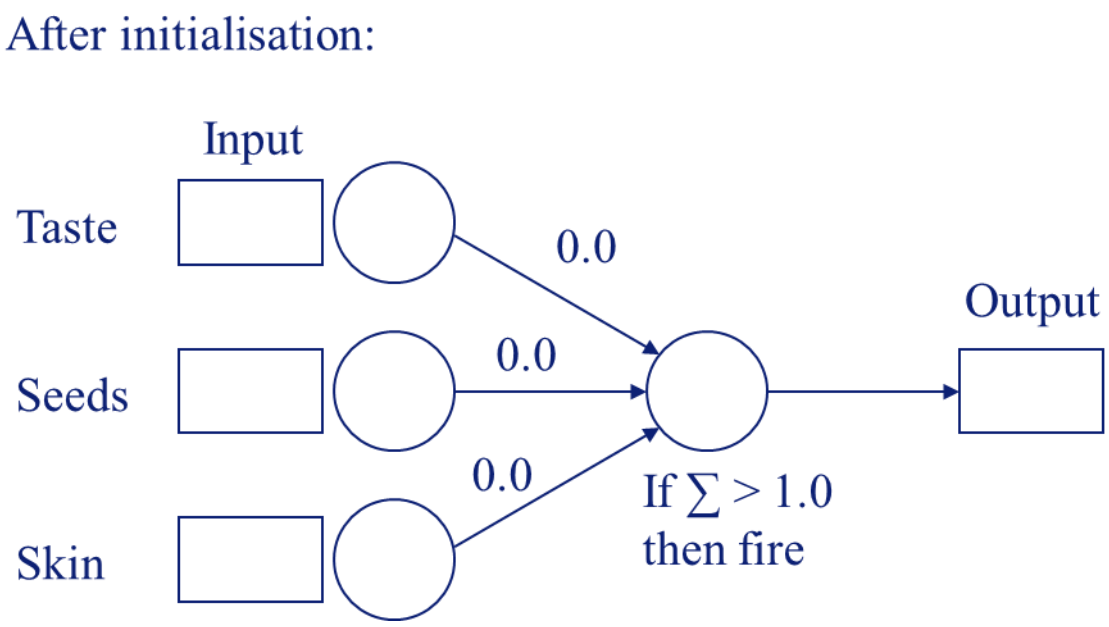
Application of the perceptron learning algorithm involves the following steps:

1.) Calculation of network output $y^{(\mu)}$ for the current input sample $\mathbf{x}^{(\mu)}$ using the current weights $\mathbf{w}(t)$ according to equation (2),
2.) Comparison of network output $y^{(\mu)}$ with the current target $d^{(\mu)}$,
3.) Modification of weights according to the perceptron learning formula, equation (1).

Using the weight update $\Delta\mathbf{w}(t)$ we have

$$\Delta\mathbf{w}(t) = \eta \cdot \left(d^{(\mu)} - y^{(\mu)}\right) \cdot \mathbf{x}^{(\mu)} \quad \Rightarrow \quad \mathbf{w}(t+1) = \mathbf{w}(t) + \Delta\mathbf{w}(t)$$

After initialization, our perceptron looks like



First iteration:

In the first iteration we present a banana to the network, so we have $x_0 = 1$ (as always), $x_1 = 1$ (taste), $x_2 = 1$ (seeds), $x_3 = 0$ (skin).

Current weights after initialization: $w_0 = -\theta = -1$ (bias term corresponding to threshold), $w_1 = w_2 = w_3 = 0$ (weight parameters).

$$\mathbf{x}^{(1)} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \end{pmatrix} \quad ; \quad \mathbf{w}(1) = \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{pmatrix} = \begin{pmatrix} -1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

The network output is given by

$$y^{(1)} = \Theta\left[\sum_{i=1}^{3} x_i^{(1)} \cdot w_i(1) - \theta(1)\right] = \Theta\left[\mathbf{x}^{(1)} \cdot \mathbf{w}(1)\right] = \Theta[-1] = 0$$

According to the table, the target output for a banana is $d^{(1)} = 1$. Thus, the perceptron made a mistake and we have to update the weights.

The weight update is given by

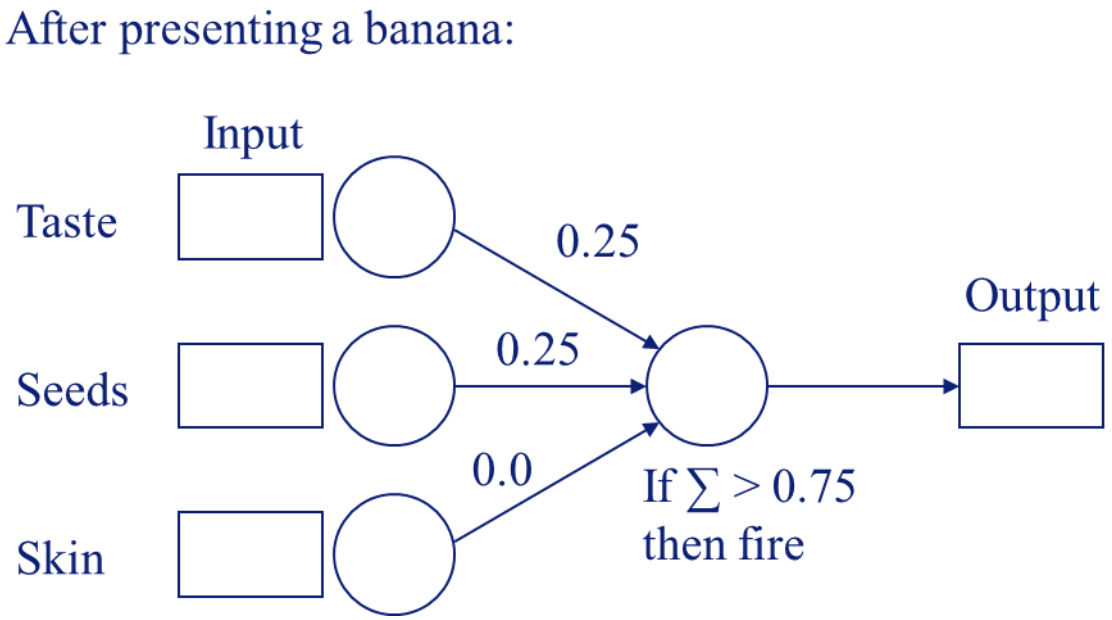
$$\Delta\mathbf{w}(1) = 0.25 \cdot (1-0) \cdot \mathbf{x}^{(1)} = 0.25 \cdot \mathbf{x}^{(1)}$$

$$\Rightarrow \quad \mathbf{w}(2) = \mathbf{w}(1) + 0.25 \cdot \mathbf{x}^{(1)}$$

We can calculate the new weights by filling out the table:

μ =1; current training sample: banana						
Input $\mathbf{x}^{(\mu)}$	Current weights $\mathbf{w}(t)$	Network Output $y^{(\mu)}$	Target output $d^{(\mu)}$	Learning rate η	Weight update $\Delta\mathbf{w}(t)$	New Weights $\mathbf{w}(t+1)$
$x_0 = 1$	$w_0 = -1$	0	1	0.25	+0.25	–0.75
$x_1 = 1$	$w_1 = 0$				+0.25	0.25
$x_2 = 1$	$w_2 = 0$				+0.25	0.25
$x_3 = 0$	$w_3 = 0$				0	0

Thus, after presenting a banana and adjusting the weights, we get the following perceptron (the threshold is given by $\theta = -w_0 = 0.75$):



Second iteration:

In the second iteration, we present a pear, so we have $x_0 = 1$ (as always), $x_1 = 1$ (taste), $x_2 = 0$ (seeds), $x_3 = 1$ (skin).

Using the calculated weights, we have

$$\mathbf{x}^{(2)} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} \quad ; \quad \mathbf{w}(2) = \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{pmatrix} = \begin{pmatrix} -0.75 \\ 0.25 \\ 0.25 \\ 0 \end{pmatrix}$$

The network output is given by

$$y^{(2)} = \Theta\left[\sum_{i=1}^{3} x_i^{(2)} \cdot w_i(2) - \theta(2)\right] = \Theta\left[\mathbf{x}^{(2)} \cdot \mathbf{w}(2)\right] = \Theta\left[-0.25\right] = 0$$

According to the table, the target output for a pear is $d^{(2)} = 1$. Thus, the perceptron made a mistake and we have to update the weights.

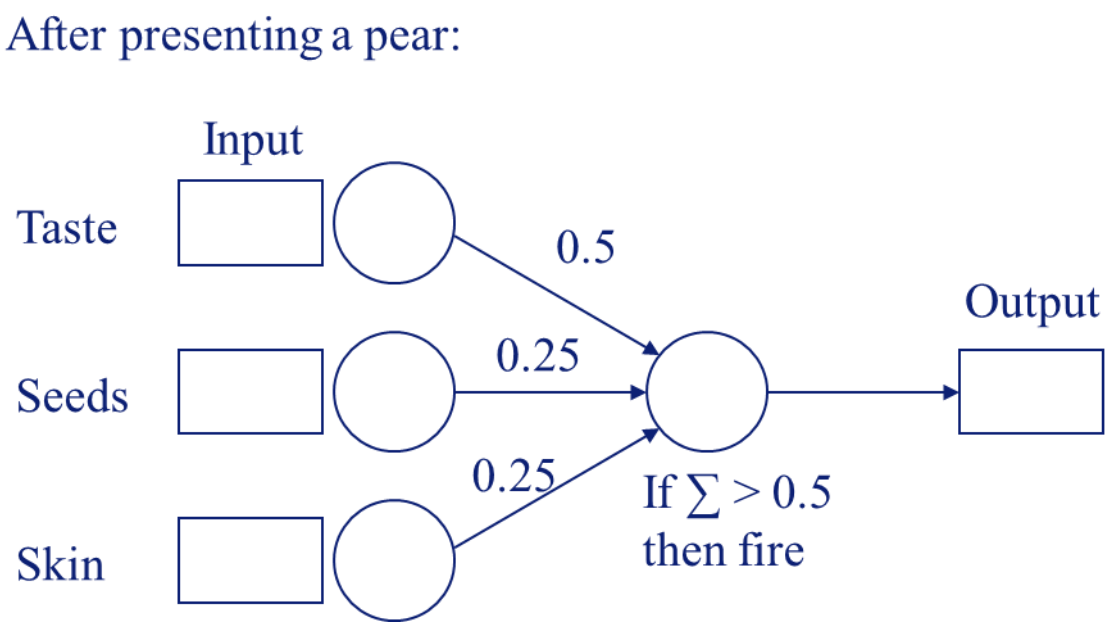
The weight update is given by

$$\Delta\mathbf{w}(2) = 0.25 \cdot (1-0) \cdot \mathbf{x}^{(2)} = 0.25 \cdot \mathbf{x}^{(2)}$$
$$\Rightarrow \quad \mathbf{w}(3) = \mathbf{w}(2) + 0.25 \cdot \mathbf{x}^{(2)}$$

We can calculate the new weights by filling out the table:

μ =2; current training sample: pear						
Input $\mathbf{x}^{(\mu)}$	Current weights $\mathbf{w}(t)$	Network Output $y^{(\mu)}$	Target output $d^{(\mu)}$	Learning rate η	Weight update $\Delta\mathbf{w}(t)$	New Weights $\mathbf{w}(t+1)$
$x_0 = 1$	$w_0 = -0.75$	0	1	0.25	+0.25	–0.5
$x_1 = 1$	$w_1 = 0.25$				+0.25	0.5
$x_2 = 0$	$w_2 = 0.25$				0.0	0.25
$x_3 = 1$	$w_3 = 0$				+0.25	0.25

Thus, after presenting a pear and adjusting the weights, we get the following perceptron (the threshold is given by $\theta = -w_0 = 0.5$):



Third iteration:

In the third iteration, we present a lemon, so we have $x_0 = 1$ (as always), $x_1 = 0$ (taste), $x_2 = 0$ (seeds), $x_3 = 0$ (skin).

Using the calculated weights, we have

$$\mathbf{x}^{(3)} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad ; \quad \mathbf{w}(3) = \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{pmatrix} = \begin{pmatrix} -0.5 \\ 0.5 \\ 0.25 \\ 0.25 \end{pmatrix}$$

The network output is given by

$$y^{(3)} = \Theta\left[\sum_{i=1}^{3} x_i^{(3)} \cdot w_i(3) - \theta(3)\right] = \Theta\left[\mathbf{x}^{(3)} \cdot \mathbf{w}(3)\right] = \Theta\left[-0.5\right] = 0$$

According to the table, the target output for a lemon is $d^{(3)} = 0$. Thus, the perceptron was correct and there is no weight modification. This can also be seen as follows:

$$\Delta\mathbf{w}(3) = 0.25 \cdot (0-0) \cdot \mathbf{x}^{(3)} = 0$$
$$\Rightarrow \quad \mathbf{w}(4) = \mathbf{w}(3)$$

Now the table looks like:

μ=3; current training sample: lemon						
Input $\mathbf{x}^{(\mu)}$	Current weights $\mathbf{w}(t)$	Network Output $y^{(\mu)}$	Target output $d^{(\mu)}$	Learning rate η	Weight update $\Delta\mathbf{w}(t)$	New Weights $\mathbf{w}(t+1)$
$x_0 = 1$	$w_0 = -0.5$	0	0	0.25	0.0	–0.5
$x_1 = 0$	$w_1 = 0.5$				0.0	0.5
$x_2 = 0$	$w_2 = 0.25$				0.0	0.25
$x_3 = 0$	$w_3 = 0.25$				0.0	0.25

Fourth iteration:

In the fourth iteration, we present a strawberry, so we have $x_0 = 1$ (as always), $x_1 = 1$ (taste), $x_2 = 1$ (seeds), $x_3 = 1$ (skin).

Using the calculated weights, we have

$$\mathbf{x}^{(4)} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \quad ; \quad \mathbf{w}(4) = \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{pmatrix} = \begin{pmatrix} -0.5 \\ 0.5 \\ 0.25 \\ 0.25 \end{pmatrix}$$

The network output is given by

$$y^{(4)} = \Theta\left[\sum_{i=1}^{3} x_i^{(4)} \cdot w_i(4) - \theta(4)\right] = \Theta\left[\mathbf{x}^{(4)} \cdot \mathbf{w}(4)\right] = \Theta\left[0.5\right] = 1$$

According to the table, the target output for a strawberry is $d^{(4)} = 1$. Thus, the perceptron was correct and there is no weight modification. This can also be seen as follows:

$$\Delta\mathbf{w}(4) = 0.25 \cdot (0-0) \cdot \mathbf{x}^{(4)} = 0$$
$$\Rightarrow \quad \mathbf{w}(5) = \mathbf{w}(4)$$

Now the table looks like:

μ=4; current training sample: strawberry						
Input $\mathbf{x}^{(\mu)}$	Current weights $\mathbf{w}(t)$	Network Output $y^{(\mu)}$	Target output $d^{(\mu)}$	Learning rate η	Weight update $\Delta\mathbf{w}(t)$	New Weights $\mathbf{w}(t+1)$
$x_0 = 1$	$w_0 = -0.5$	1	1	0.25	0.0	–0.5
$x_1 = 1$	$w_1 = 0.5$				0.0	0.5
$x_2 = 1$	$w_2 = 0.25$				0.0	0.25
$x_3 = 1$	$w_3 = 0.25$				0.0	0.25

Fifth iteration:

In the fifth iteration, we present a green apple, so we have $x_0 = 1$ (as always), $x_1 = 0$ (taste), $x_2 = 0$ (seeds), $x_3 = 1$ (skin).

Using the calculated weights, we have

$$\mathbf{x}^{(5)} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \quad ; \quad \mathbf{w}(5) = \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{pmatrix} = \begin{pmatrix} -0.5 \\ 0.5 \\ 0.25 \\ 0.25 \end{pmatrix}$$

The network output is given by

$$y^{(5)} = \Theta\left[\sum_{i=1}^{3} x_i^{(5)} \cdot w_i(5) - \theta(5)\right] = \Theta\left[\mathbf{x}^{(5)} \cdot \mathbf{w}(5)\right] = \Theta\left[-0.25\right] = 0$$

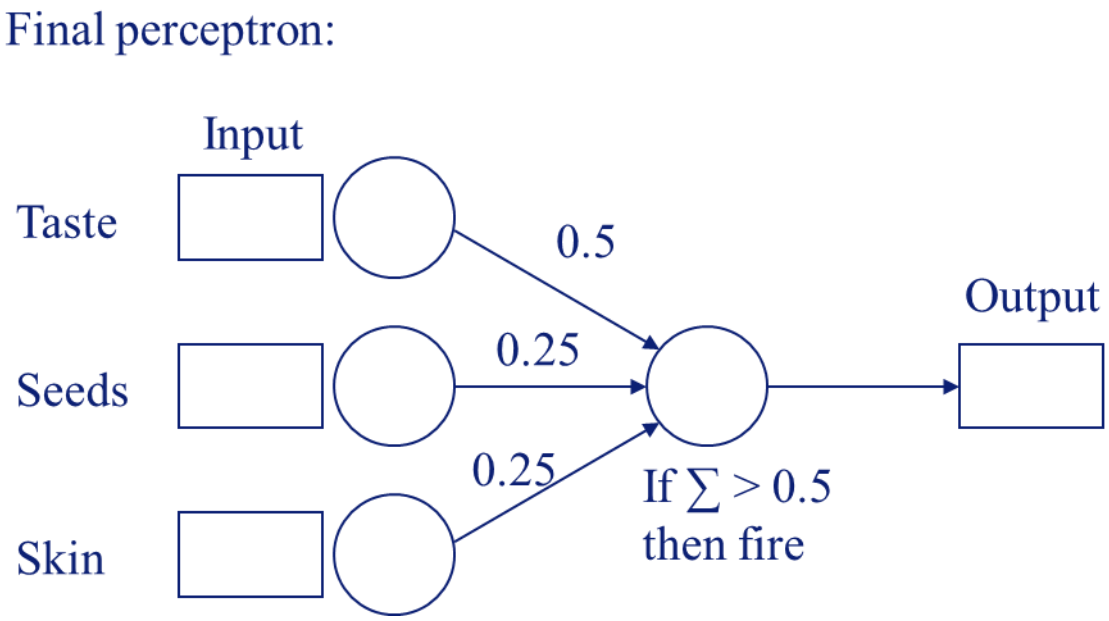
According to the table, the target output for a green apple is $d^{(5)} = 0$. Thus, the perceptron was correct and there is no weight modification. This can also be seen as follows:

$$\Delta\mathbf{w}(5) = 0.25 \cdot (0 - 0) \cdot \mathbf{x}^{(5)} = 0$$
$$\Rightarrow \quad \mathbf{w}(6) = \mathbf{w}(5)$$

Finally, the table looks like:

μ=5; current training sample: green apple						
Input $\mathbf{x}^{(\mu)}$	Current weights $\mathbf{w}(t)$	Network Output $y^{(\mu)}$	Target output $d^{(\mu)}$	Learning rate η	Weight update $\Delta\mathbf{w}(t)$	New Weights $\mathbf{w}(t+1)$
$x_0 = 1$	$w_0 = -0.5$	0	0	0.25	0.0	–0.5
$x_1 = 0$	$w_1 = 0.5$				0.0	0.5
$x_2 = 0$	$w_2 = 0.25$				0.0	0.25
$x_3 = 1$	$w_3 = 0.25$				0.0	0.25

Thus, the final perceptron is given by



Using the calculated parameters $\theta = -w_0 = 0.5$ and $w_1 = 0.5$, $w_2 = 0.25$, $w_3 = 0.25$, the output of the perceptron is given by

$$y = \Theta\left[\sum_{i=1}^{3} x_i \cdot w_i - \theta\right] = \Theta\left[0.5 \cdot x_1 + 0.25 \cdot x_2 + 0.25 \cdot x_3 - 0.5\right]$$

This leads to the following outputs:

Fruit	Perceptron input			Network output y	Target output d
	Taste x_1	Seeds x_2	Skin x_3		
Banana	1	1	0	$\Theta[0.25]=1$	1
Pear	1	0	1	$\Theta[0.25]=1$	1
Lemon	0	0	0	$\Theta[-0.5]=0$	0
Strawberry	1	1	1	$\Theta[0.5]=1$	1
Green apple	0	0	1	$\Theta[-0.25]=0$	0

Remarks:

- Two examples were sufficient to obtain the final perceptron, i.e. the last three examples did not contribute to weight modification.
- The target output in this example is identical to the first feature x_1 (taste). From this, you could directly construct an appropriate perceptron.

Exercise 3 (Single-layer perceptron, gradient learning, 2dim. classification):

The goal of this exercise is to solve a two-dimensional binary classification problem with gradient learning, using the sklearn python library. Since the problem is two-dimensional, the perceptron has 2 inputs. Since the classification problem is binary, there is one output. The (two-dimensional) inputs for training are provided in the file **`exercise3b_input.txt`**, the corresponding (1-dimensional) targets in the file **`exercise3b_target`**. To visualize the results, the training samples corresponding to class 1 (output label "0") have separately been saved in the file **`exercise3b_class1.txt`**, the training samples corresponding to class 2 (output label "1") in the file **`exercise3b_class2.txt`**. The gradient learning algorithm – using the **`logistic()`** activation function – shall be used to provide a solution to this classification problem. Note that due to the **`logistic()`** activation function, the output of the perceptron is a real value in [0,1]:

$$\texttt{logistic}(h) = \frac{1}{1+e^{-h}}$$

To assign a binary class label (either 0 or 1) to an input example, the perceptron output y can be passed through the Heaviside function $\Theta[\, y - 0.5 \,]$ to yield a binary output y^{binary}. Then, any perceptron output between 0.5 and 1 is closer to 1 than to 0 and will be assigned the class label "1". Conversely, any perceptron output between 0 and <0.5 is closer to 0 than to 1 and will be assigned the class label "0" (see also the scikit learn documentation, "Neural network models (supervised), Mathematical formulation)". As usual, denote the weights of the perceptron w_1 and w_2 and the bias $w_0 = -\theta$.

a) Using the above-mentioned post-processing step $\Theta[\, y - 0.5 \,]$ applied to the perceptron output y, show that the decision boundary separating the inputs $\mathbf{x}=(\, x_1 \,,\, x_2 \,)$ assigned to class label "1" from those inputs assigned to class label "0" is given by a straight line in two-dimensional space corresponding to the equation (see lines 83 and 108 in file **`exercise3b.py`**):

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{w_0}{w_2}$$

Solution:

The (binary) output of the perceptron is given by

$$y^{\text{binary}} = \Theta[y - 0.5] = \Theta\left[\text{logSig}(\, h) - 0.5\right] = \Theta\left[\frac{1}{1+e^{-h}} - 0.5\right] = \Theta\left[\frac{1}{1+e^{-\mathbf{w}\cdot\mathbf{x}}} - 0.5\right]$$

where the postsynaptic potential h is given by

$$h = \mathbf{w}\cdot\mathbf{x} = w_0 + x_1 \cdot w_1 + x_2 \cdot w_2$$

The decision boundary is given by the points $\mathbf{x}$ where y^{binary} is close to both class labels, which is the case if the argument of the Heaviside function is 0. Thus, the condition for the decision boundary is:

$$\frac{1}{1+e^{-\mathbf{w}\cdot\mathbf{x}}} - 0.5 = 0 \Leftrightarrow \frac{1}{1+e^{-\mathbf{w}\cdot\mathbf{x}}} = \frac{1}{2} \Leftrightarrow 1+e^{-\mathbf{w}\cdot\mathbf{x}} = 2 \Leftrightarrow e^{-\mathbf{w}\cdot\mathbf{x}} = 1 \Leftrightarrow -\mathbf{w}\cdot\mathbf{x} = \ln(1) = 0$$

$$\Leftrightarrow \mathbf{w}\cdot\mathbf{x} = w_0 + w_1 \cdot x_1 + w_2 \cdot x_2 = 0 \Leftrightarrow w_2 \cdot x_2 = -w_1 \cdot x_1 - w_0 \Leftrightarrow x_2 = -\frac{w_1}{w_2} \cdot x_1 - \frac{w_0}{w_2}$$

b) The classification problem (defined by the training data provided in **`exercise3b_input.txt`** and the targets provided in **`exercise3b_target.txt`**) shall now be solved using the scikit learn python library. Corresponding software has been provided in the file **`exercise3b.py`**. Run the file by typing

`run exercise3b.py`

in a python console at least three times and report on your findings. Change appropriate parameters (e.g. the learning rate, the batch size, the choice of the solver, potentially the number of epochs etc.) and again report on your findings.

Note: By setting

`hidden_layer_sizes=()`

no hidden layers are generated, realising a single-layer perceptron. The number of layers is 2 (scikit learn counts input and output as two layers, instead of counting the number of learnable weight layers). Training is invoked with the "**`fit`**" method.

The weights can be accessed by **`net.coefs_`**, the biases by **`net.intercepts_`** (both are lists of numpy arrays, so several indices are needed to access individual weights).

In scikit learn, a single-layer perceptron can also be realized as perceptron linear model:

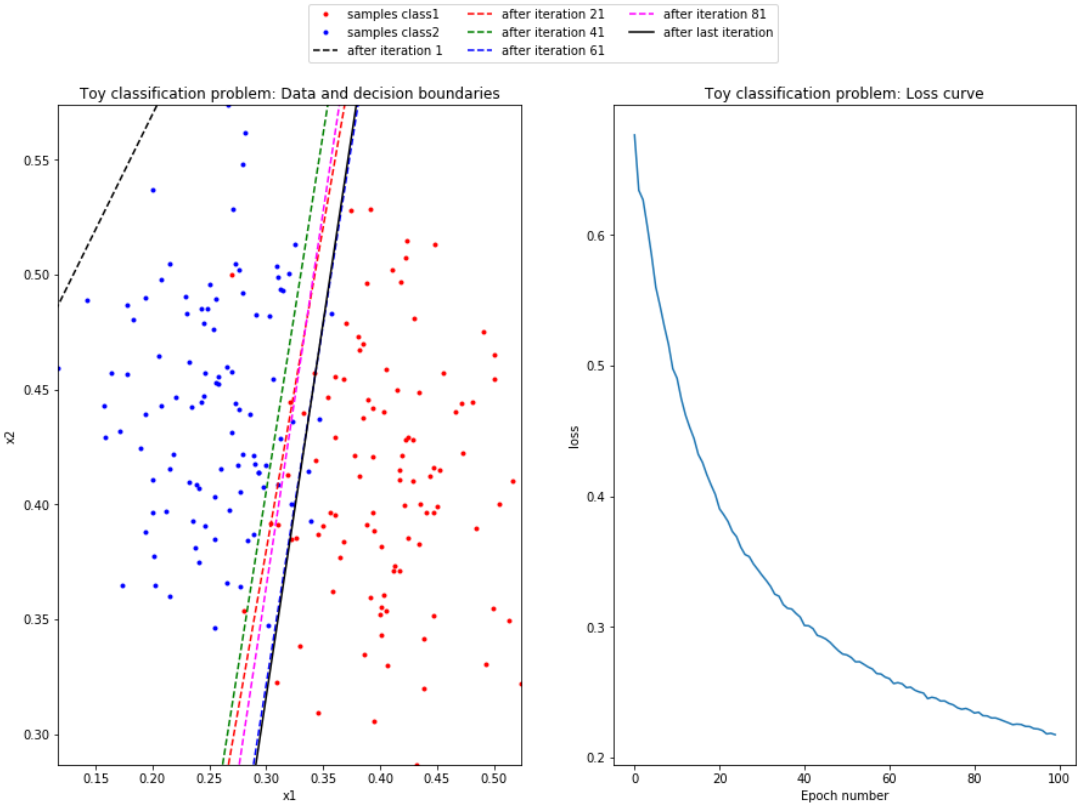
`from sklearn.linear_model import perceptron`

Here, however, the activation function is always linear.

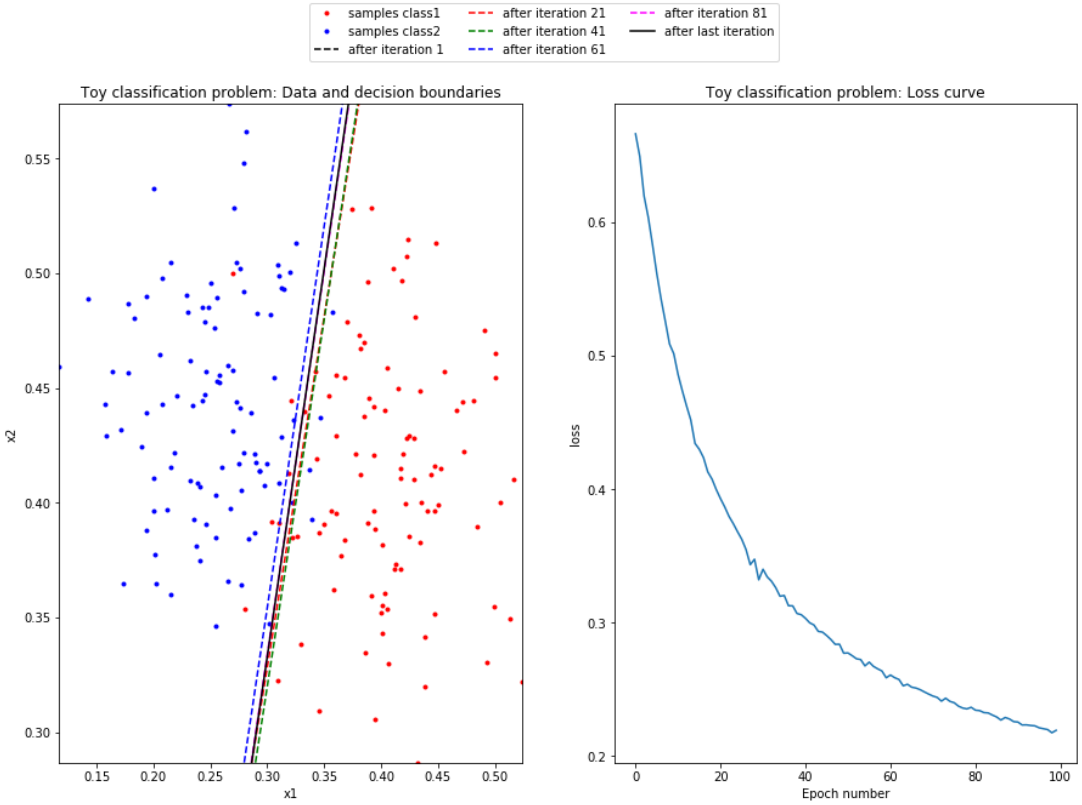
Solution:

The following results have been obtained for the parameters
Learning rate: 0.1
Batch size: 1
Solver: 'sgd' (stochastic gradient descent)

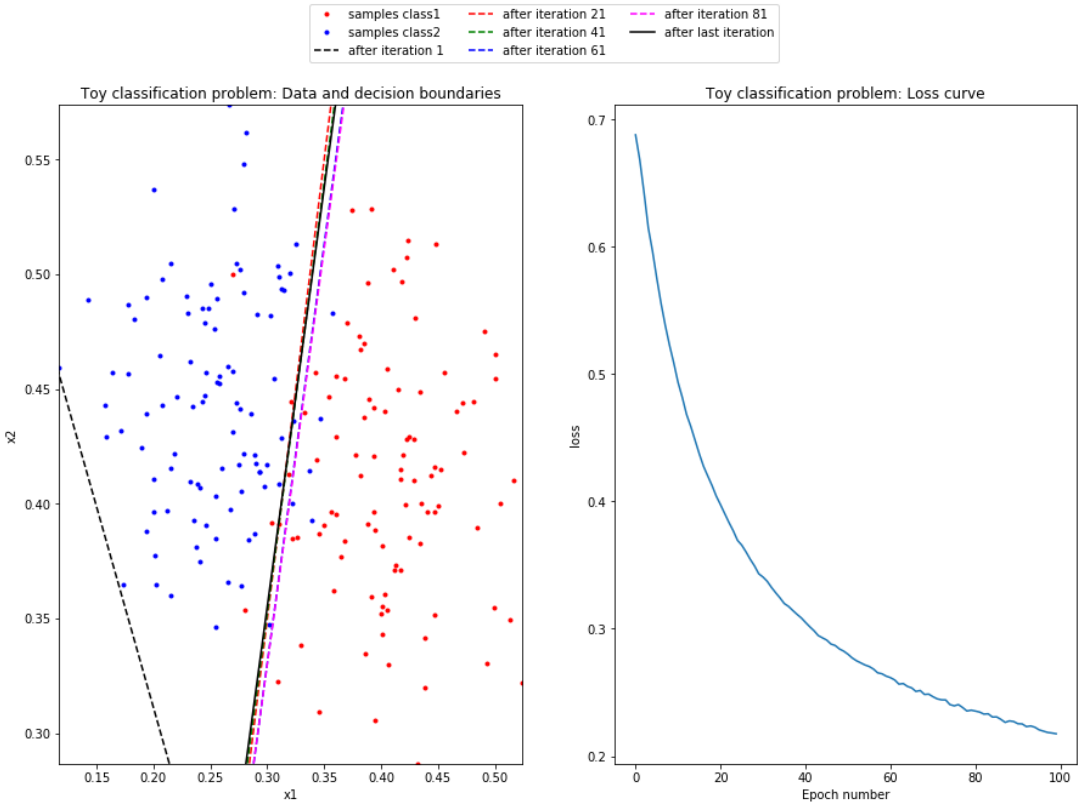
First run (Number of binary errors: 12):



Second run (Number of binary errors: 11):

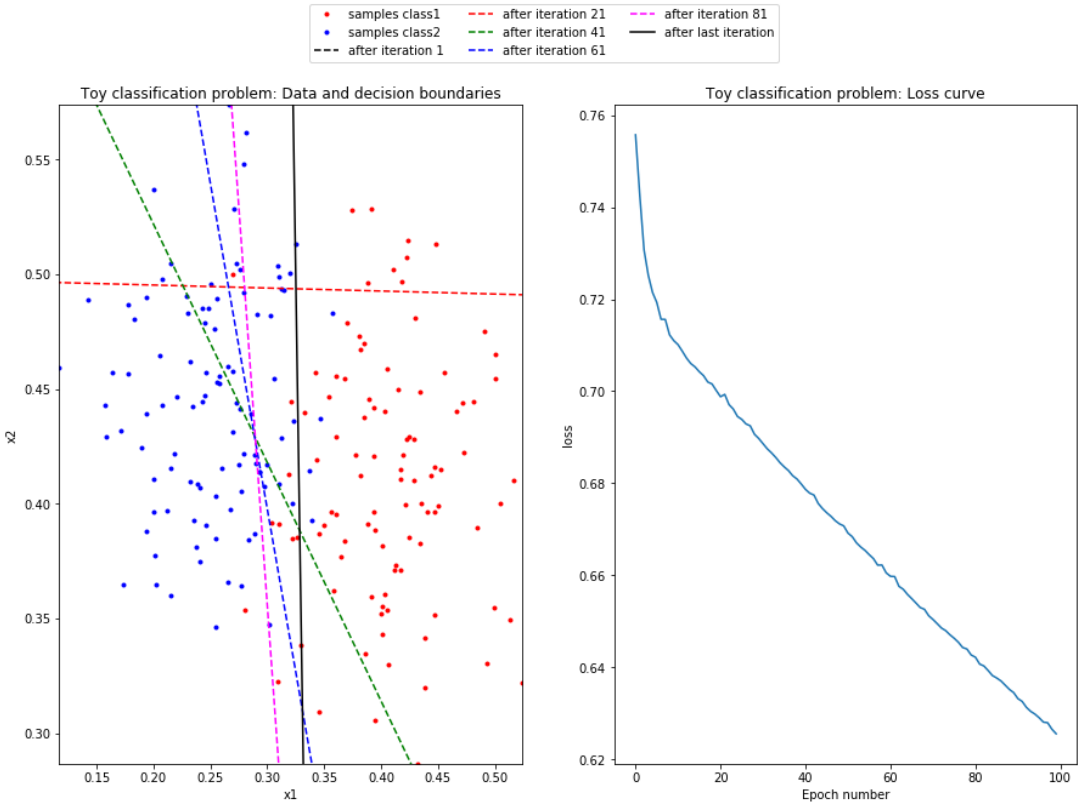


Third run (Number of binary errors: 11):



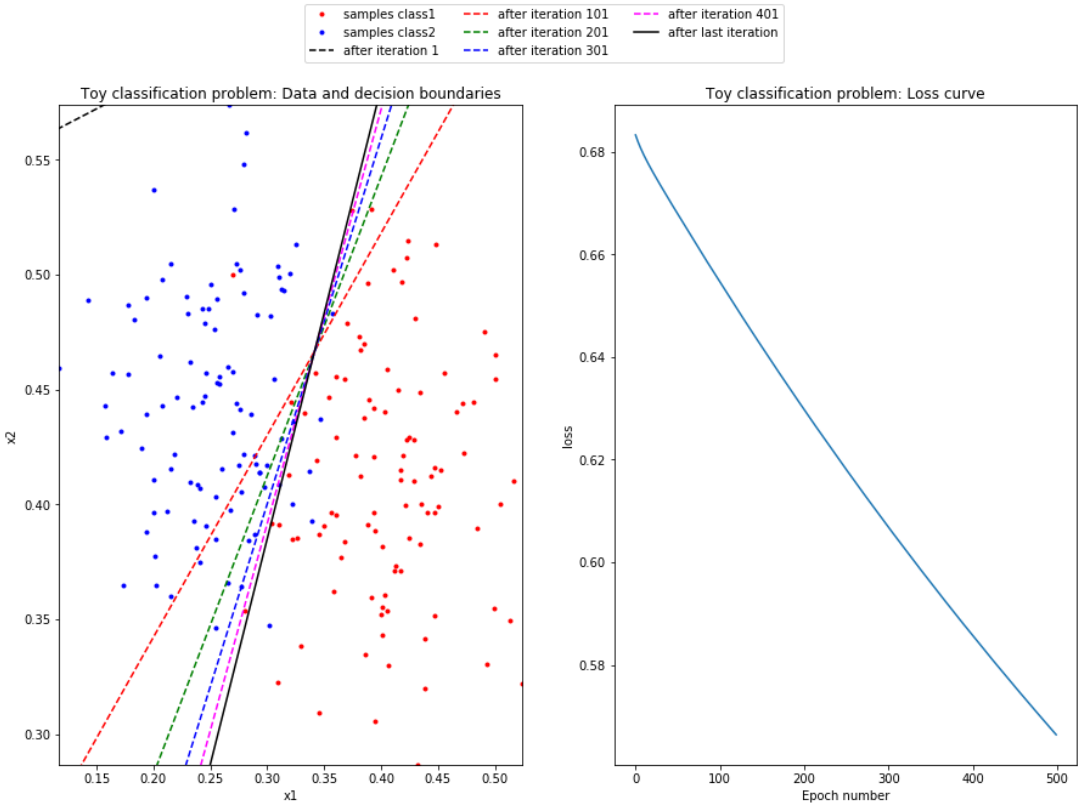
Due to the randomness of the initial weights, different runs may produce different decision boundaries (as seen in the plots above). However, the number of binary errors is mostly 11 (sometimes 12).

If the learning rate is too low, convergence is much slower:
Learning rate: 0.001,
Batch size: 1
Solver: ‘sgd’ (stochastic gradient descent)
18 binary errors



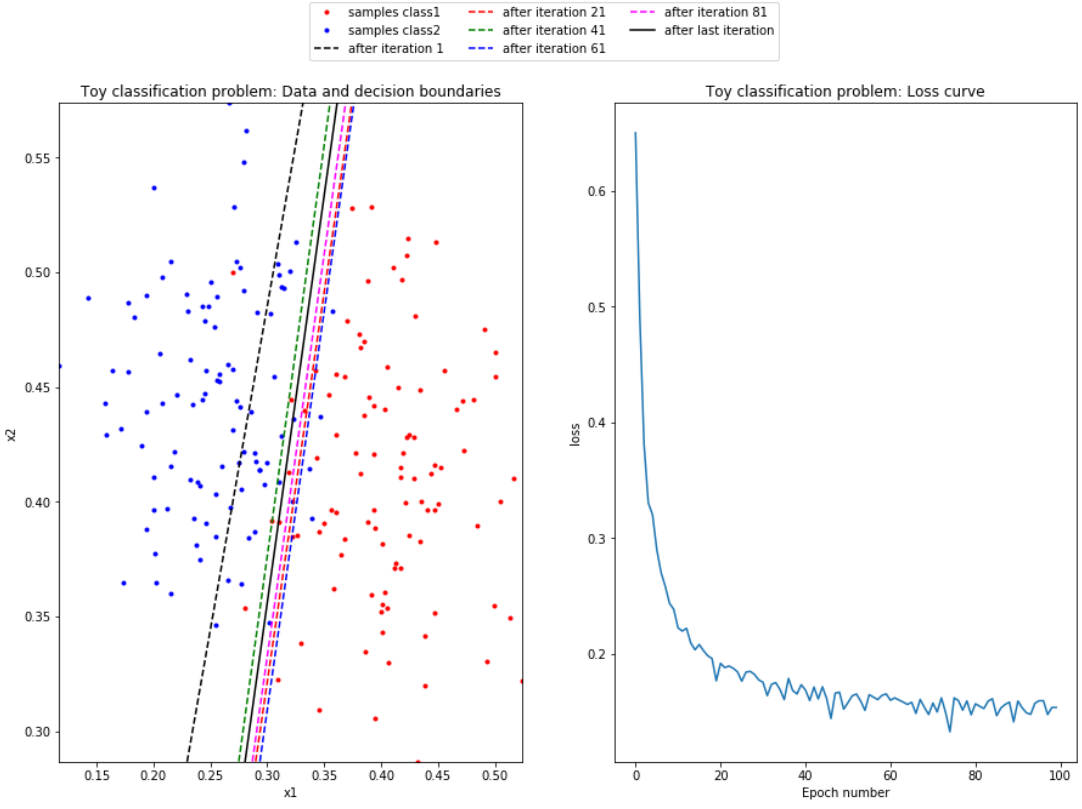
However, increasing the number of iterations to 500 (and plotting intermediate decision boundaries only after each 100 iterations) leads to even fewer (9) binary errors:

Learning rate: 0.001,
Batch size: 1
Solver: ‘sgd’ (stochastic gradient descent)
9 binary errors

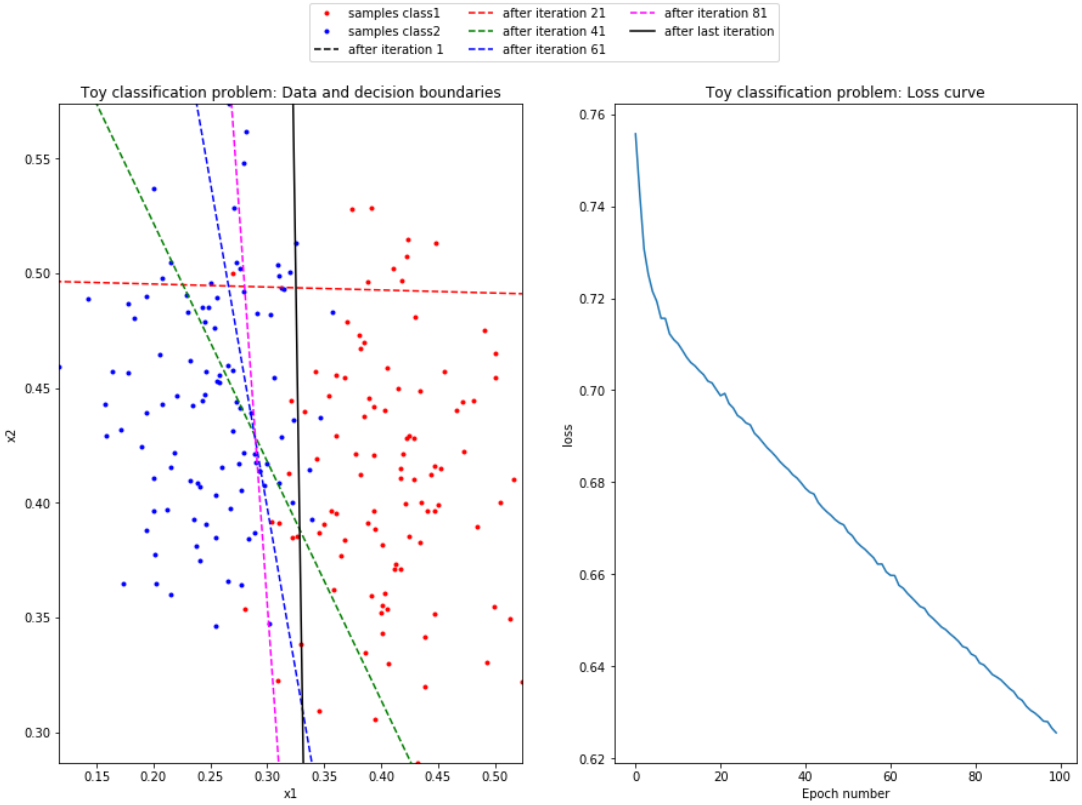


A larger learning rate leads to faster, but “less stable” convergence:

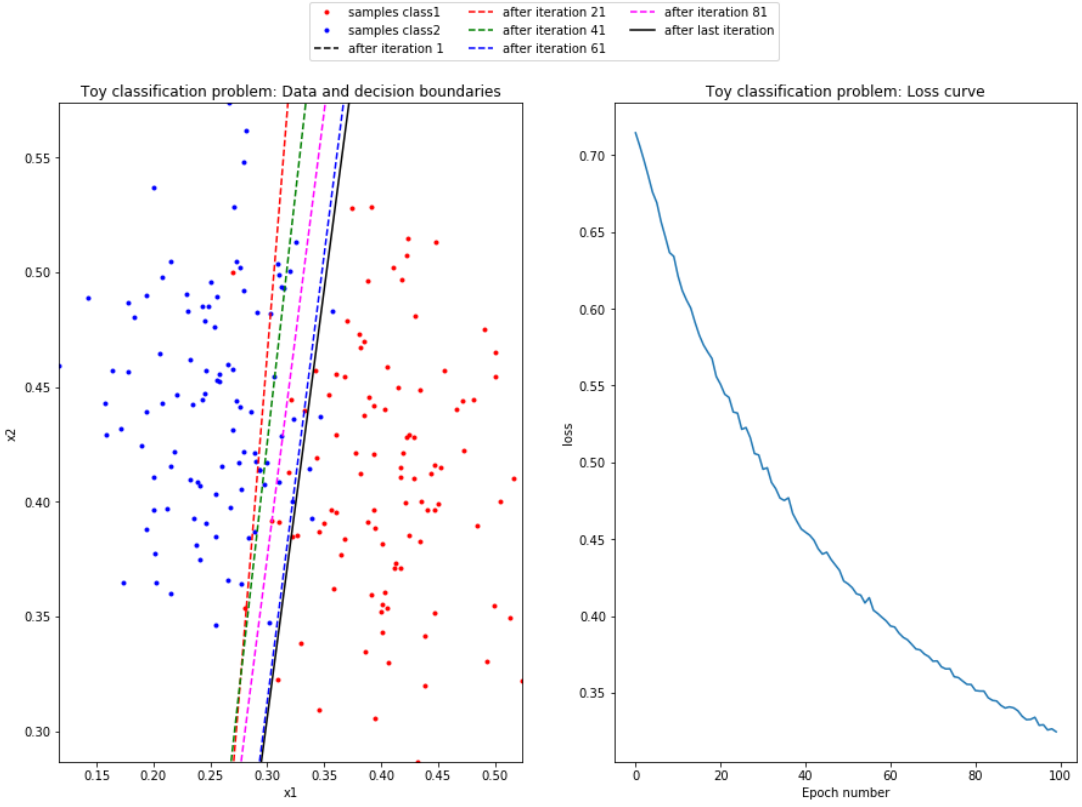
Learning rate: 1.0
Batch size: 1
Solver: ‘sgd’ (stochastic gradient descent)
11 binary errors



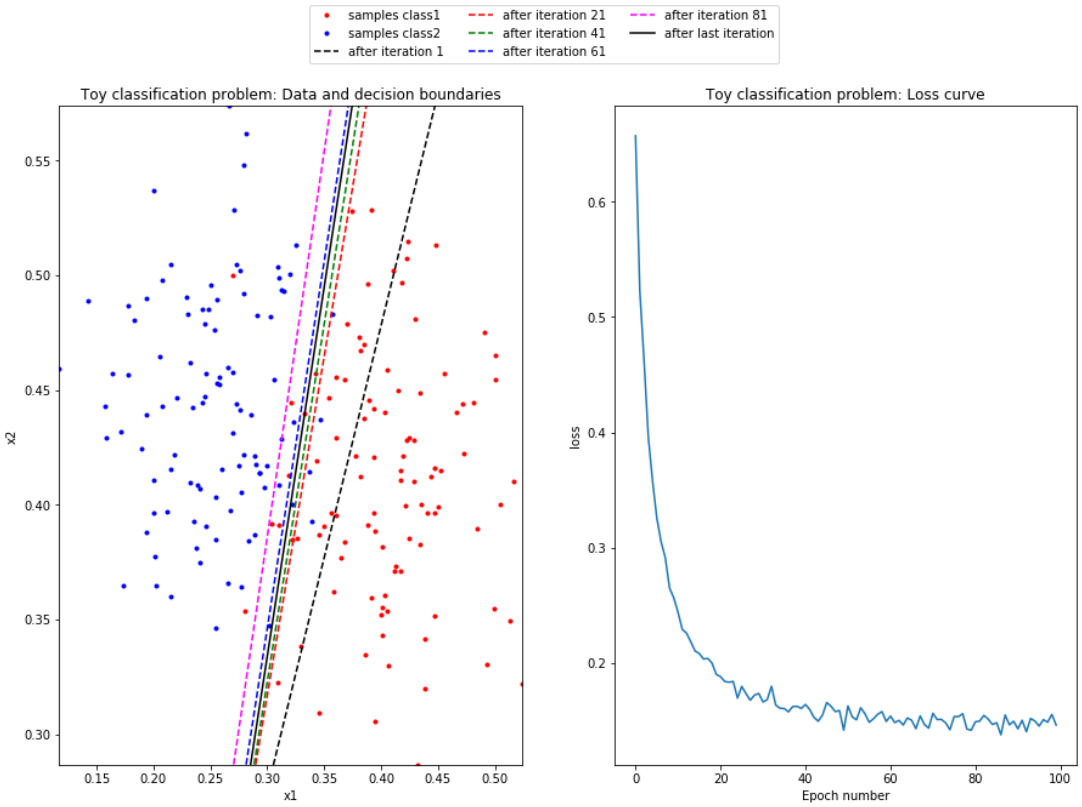
Increasing the batch size leads to slower convergence, so that the learning rate might be increased:
Learning rate: 0.1, batch size: 32, solver: ‘sgd’ (stochastic gradient descent)
15 binary errors



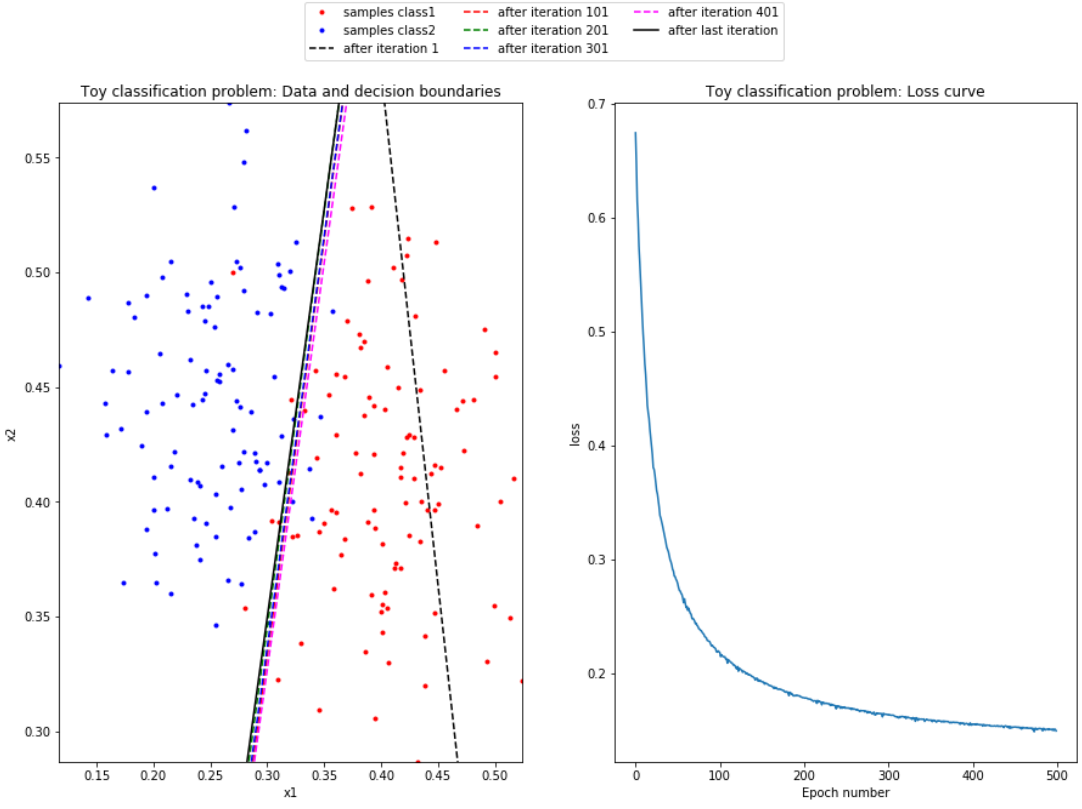
Learning rate: 1.0, batch size: 32, solver: ‘sgd’ (stochastic gradient descent)
11 binary errors



The ‘adam’ solver lead to a less smooth learning curve than the ‘sgd’ solver:
Learning rate: 0.1, batch size: 1, solver: ‘adam’
12 binary errors



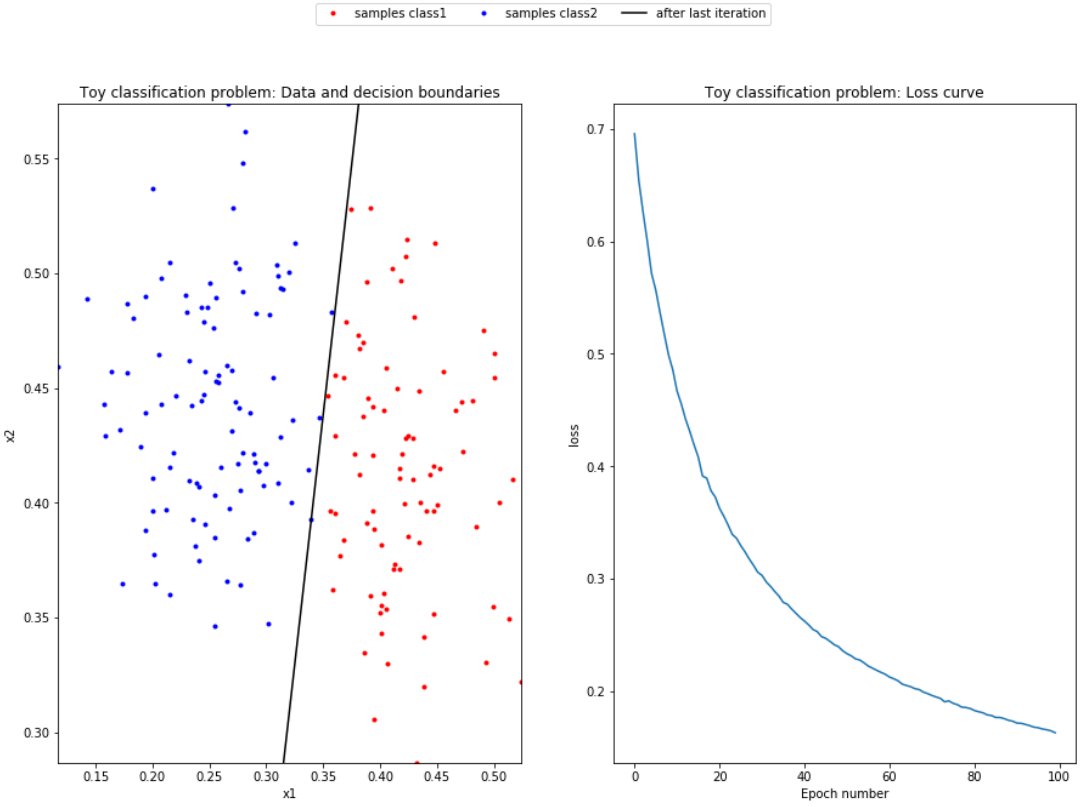
Finally, iterating longer with the default setting does not improve performance:
Learning rate: 0.1, batch size: 1, solver: ‘sgd’ (stochastic gradient descent), 500 epochs:
11 binary errors



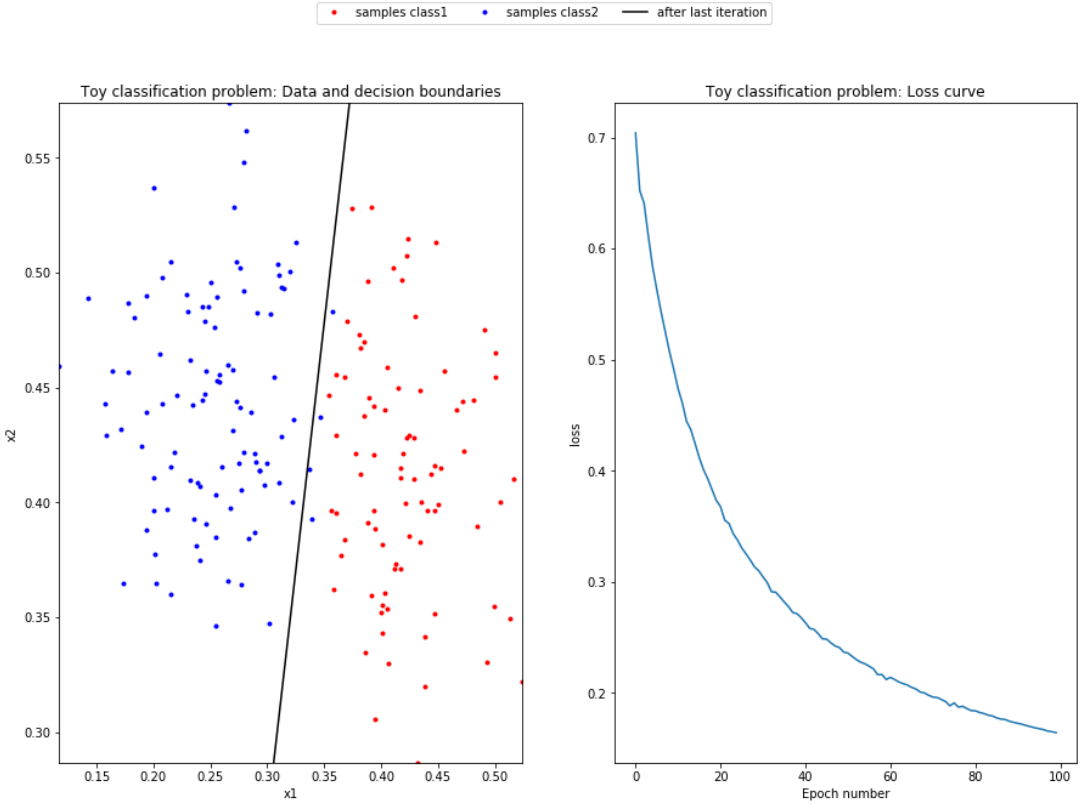
c) Repeat exercise b) with the training set **`exercise3c_input.txt`** and the targets **`exercise3c_target.txt`**. Those points have been generated from the input points of exercise b) by removing points from class 1 (i.e. those points the x-coordinate of which is below 0.35). Do not forget to modify the variables **`class1`** and **`class2`** to load the files **`exercise3c_class1.txt`** and **`exercise3c_class1.txt`**, respectively! Discuss the output of the training algorithm in terms of the resulting decision boundary and the final training error.

Solution:

Using the default setting, the following results were obtained:
Learning rate: 0.1, batch size: 1, solver: 'sgd' (stochastic gradient descent), 500 epochs:
First run: 0 binary errors



Second run: 4 binary errors



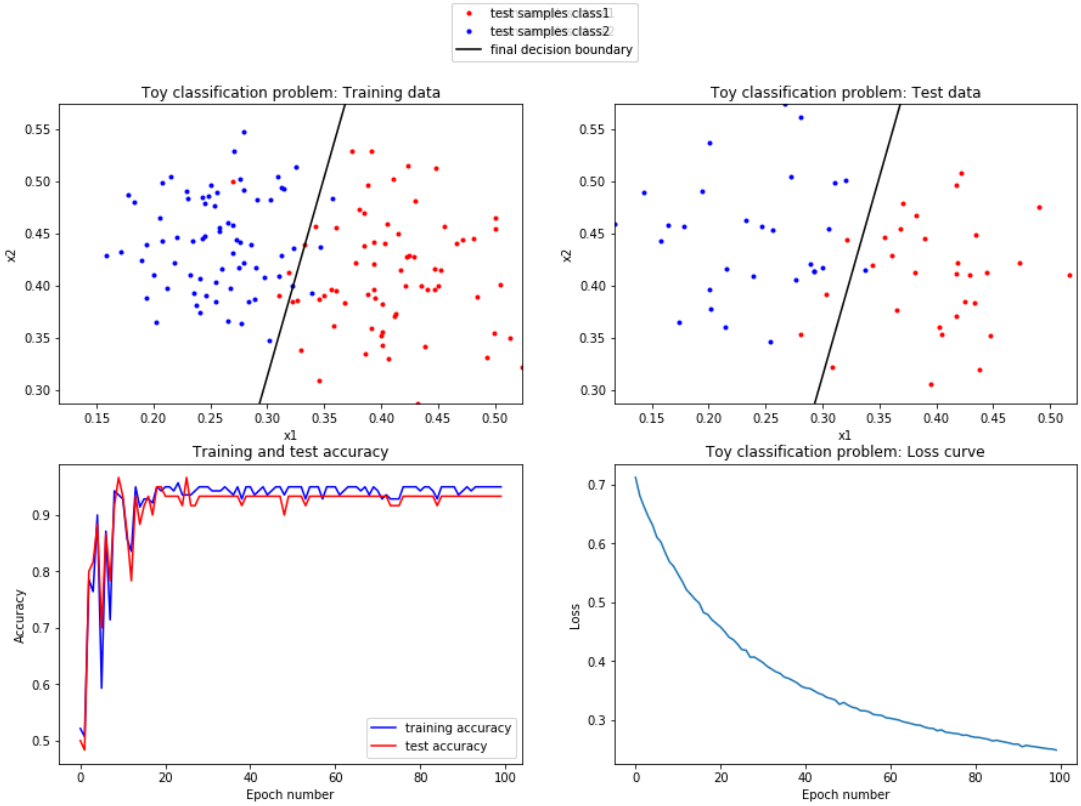
The decision boundary has moved towards larger x-values, since the points of class 1 with the lowest x-values have been removed. The classes can be separated more clearly now; the resulting loss is smaller than before and the number of binary errors much lower (between 0 and about 4).

d) Divide the input samples into a separate training and a test set. To this end, you may use the **`train_test_split`** function of sklearn. To use this function, the following line has to be included in the python file:
`from sklearn.model_selection import train_test_split`
Adapt the script from part b) of this exercise to perform a training on the training corpus and a test on the training and the test corpus created by **`train_test_split`**. The evaluation of the model can be performed using the **`score`** function, e.g.
`net.score(X_train, y_train)`
`net.score(X_test, y_test)`
Plot the training and test score (accuracy) as a function of the iteration number. Run the script at least two times and report on your findings.

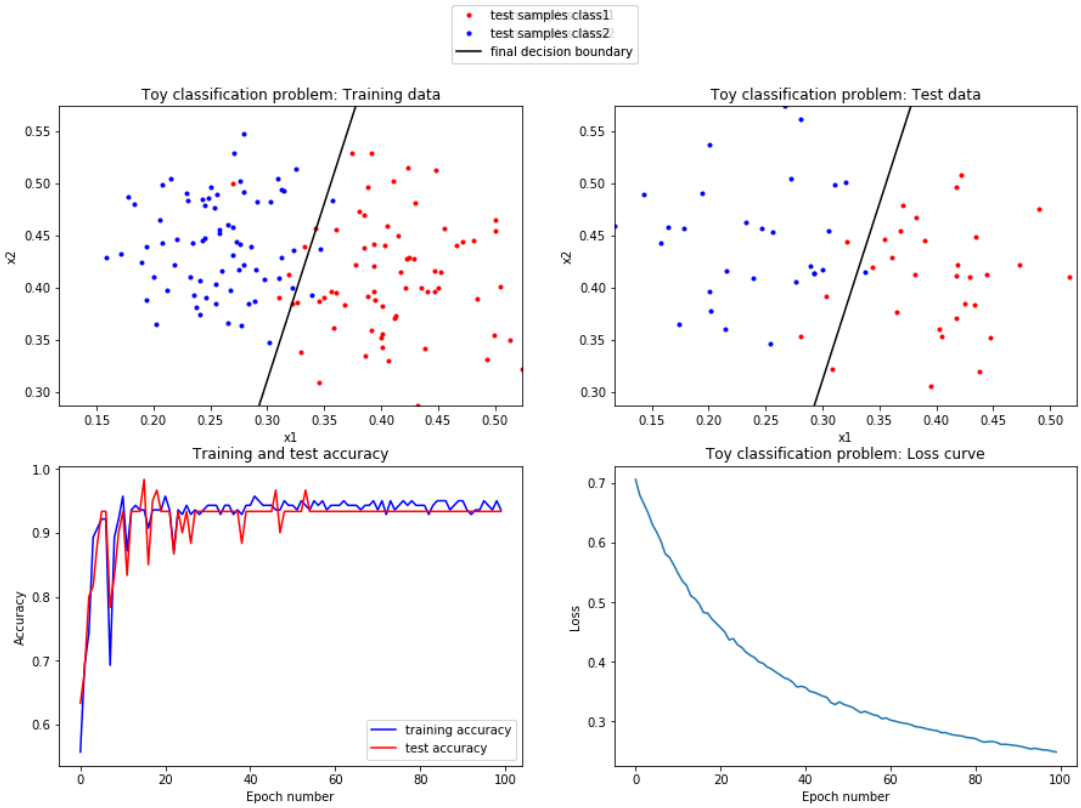
Solution:

Using the default setting, the following results were obtained:
Learning rate: 0.1
Batch size: 1
Solver: ‘sgd’ (stochastic gradient descent)
100 epochs:

First run: 7 binary train errors, 4 binary test errors



Second run: 9 binary train errors, 4 binary test errors



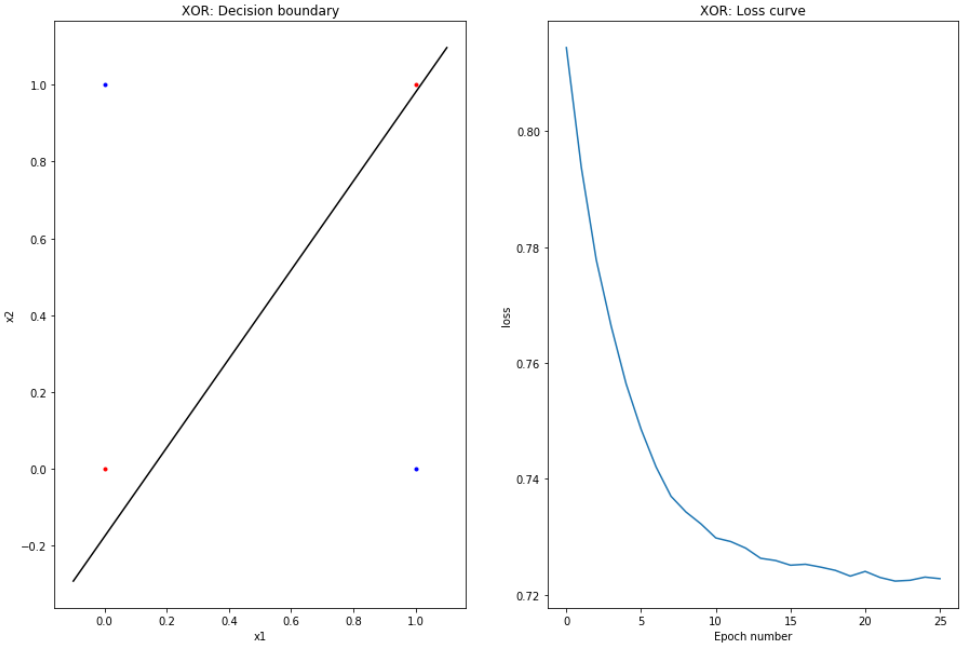
As expected, the test accuracy is lower than the training accuracy. Overfitting is not observed.

e) Modify the script **`exercise3b.py`** to handle the XOR-problem, i.e. set
```
input = np.array([[0,0],[0,1],[1,0],[1,1]])
target = np.array([0, 1, 1, 0])
```
and plot the final decision boundary and the loss function. Report on your findings.

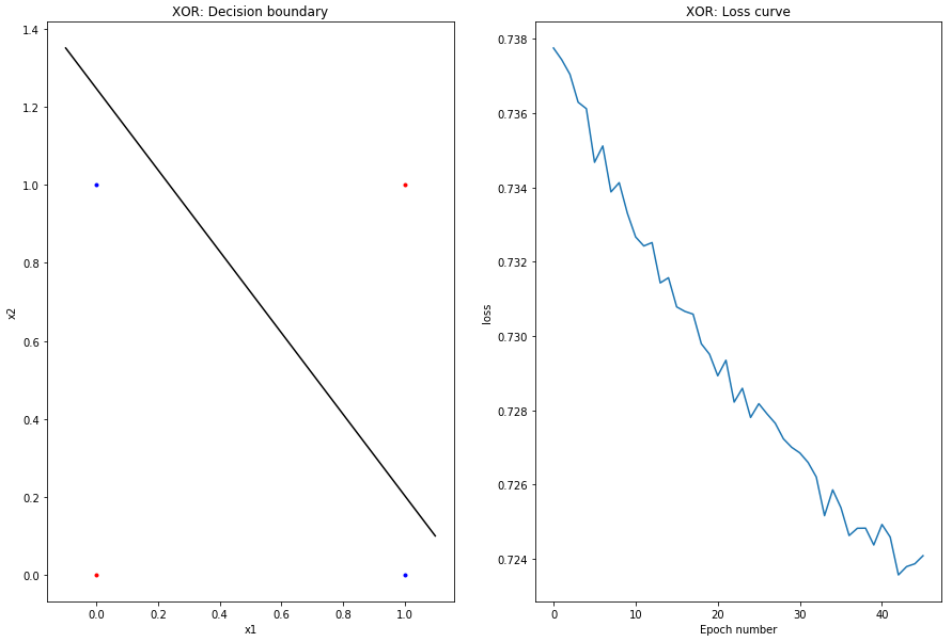
Solution:

Using the default setting, the following results were obtained:
Learning rate: 0.1, batch size: 1, solver: 'sgd' (stochastic gradient descent)
100 epochs (however, the actual number of epochs is smaller, since the training loss did not improve more than tol=0.000010 for two consecutive epochs and the algorithm stops.

First run:



Second run:



The loss is larger than 0.5; XOR cannot be learned by a single-layer perceptron.

Exercise 4 (Multi-layer perceptron and backpropagation – small datasets):

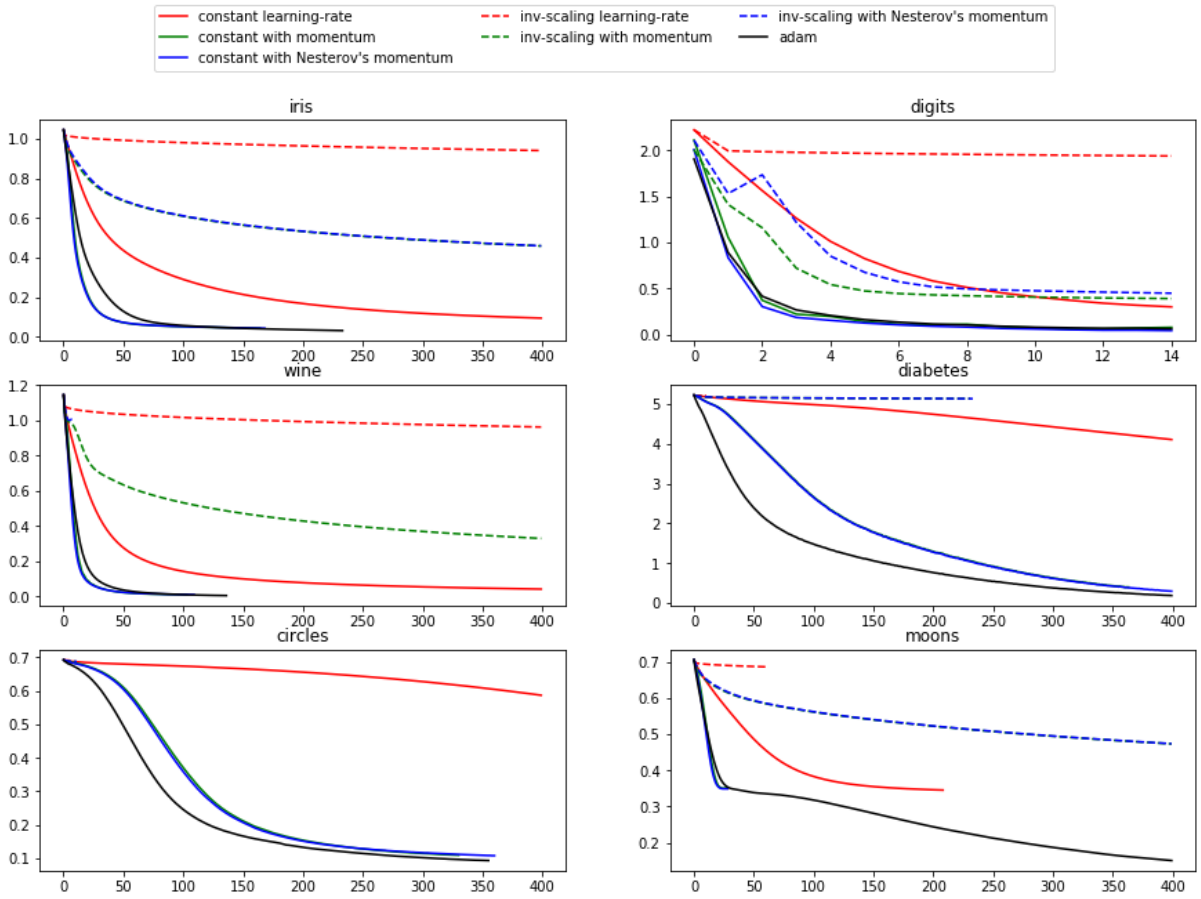
The goal of this exercise is to apply a multi-layer perceptron (MLP), trained with the backpropagation algorithm as provided by the scikit learn python library, to four classification problems provided by the UCI repository (and contained in the scikit learn package; i.e. iris, digits, wine, diabetes) and two artificially generated classification problems (circles, moon). In particular, the influence of the backpropagation solver and of the network topology shall be investigated in parts a) and b) of the exercise, respectively.

a) In this part of the exercise, seven solvers (stochastic gradient descent with constant or adapted learning rate and / or momentum or not, adam optimizer) shall be applied to the six datasets. A python script for this experiment is provided in **`exercise4a.py`**. Find out the values of the most important parameters (number of hidden layers and of hidden neurons, activation function, batch size, learning rate, momentum ...) as provided by the script. Apply the script and record the values of the training loss, training score and test score. You may also test different values for important parameters. What are your conclusions regarding the comparison of the solver strategies and the learning success? Report on the database statistics.

Solution:

Important parameter settings:

- Network topology: By default, the MLP has a single hidden layer with 100 neurons.
- Activation function: ReLU
- Batch size: Set to min(200, number of samples)
- Initial learning rate: 0.01
- Momentum: 0.9



Summary of results:

stochastic learning strategy	train loss	train score	test score
iris dataset (4-dim. inputs, 3 classes; 105 training / 45 test samples)			
constant learning rate	0.095441*	0.952381*	0.977778*
constant with momentum	0.044485	0.980952	0.977778
constant with Nesterov's momentum	0.044220	0.980952	0.977778
inv-scaling learning rate	0.941184	0.657143	0.555556
inv-scaling with momentum	0.459448	0.866667	0.777778
inv-scaling with Nesterov's momentum	0.460127	0.866667	0.777778
Adam	0.032600	0.990476	0.977778
digits dataset (4-dim. inputs, 10 classes; 1257 training / 540 test samples)			
constant learning rate	0.298351*	0.943516*	0.925926*
constant with momentum	0.075761	0.985680	0.940741
constant with Nesterov's momentum	0.040823	0.996818	0.964815
inv-scaling learning rate	1.936591	0.566428	0.494444
inv-scaling with momentum	0.388572	0.911695	0.888889
inv-scaling with Nesterov's momentum	0.446801	0.910103	0.890741
Adam	0.060121	0.993636	0.970370
wine dataset (13-dim. inputs, 3 classes; 124 training / 54 test samples)			
constant learning rate	0.042257	1.000000	1.000000
constant with momentum	0.010652	1.000000	1.000000
constant with Nesterov's momentum	0.010837	1.000000	1.000000
inv-scaling learning rate	0.962229	0.661290	0.703704
inv-scaling with momentum	0.329851	0.967742	0.962963
inv-scaling with Nesterov's momentum	1.005831	0.395161	0.407407
Adam	0.005622	1.000000	1.000000
diabetes dataset (10-dim. inputs, 181 classes; 309 training / 133 test samples)			
constant learning rate	4.106060	0.103560	0.007519
constant with momentum	0.395595	0.970874	0.007519
constant with Nesterov's momentum	0.293299	0.987055	0.007519
inv-scaling learning rate	5.213980	0.009709	0.000000
inv-scaling with momentum	5.133636	0.012945	0.007519
inv-scaling with Nesterov's momentum	5.133436	0.012945	0.007519
Adam	0.182448	0.993528	0.000000
circles dataset (2-dim. inputs, 2 classes; 70 training / 30 test samples)			
constant learning rate	0.586886	0.857143	0.800000
constant with momentum	0.109379	0.942857	0.866667
constant with Nesterov's momentum	0.108059	0.942857	0.866667
inv-scaling learning rate	0.691072	0.514286	0.466667
inv-scaling with momentum	0.687417	0.671429	0.700000
inv-scaling with Nesterov's momentum	0.687591	0.642857	0.700000
Adam	0.093452	0.957143	0.866667
moon dataset (2-dim. inputs, 2 classes; 70 training / 30 test samples)			
constant learning rate	0.345146	0.842857	0.866667
constant with momentum	0.351435	0.814286	0.833333
constant with Nesterov's momentum	0.348738	0.814286	0.866667
inv-scaling learning rate	0.685614	0.485714	0.533333
inv-scaling with momentum	0.472911	0.814286	0.833333
inv-scaling with Nesterov's momentum	0.473382	0.814286	0.833333
Adam	0.150447	0.942857	0.933333

*: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (400) reached and the optimization hasn't converged yet.

Conclusions:

- Training on the "iris", "digits" and "wine" datasets are successful (with test accuracies larger than 0.95 for the best learning strategy). Training on the "circles" and "moon" datasets yield lower test accuracies (between 0.85 and 0.95). Training on the "diabetes" data set produces extreme overfitting (for the best learning strategies), presumably due to the very large number of output classes (compared to the number of training samples), such that the test score is close to zero.
- The adam optimizer yields the lowest training loss and the largest training score among all learning strategies for the specified number of iterations, on all tested data sets except for the "digits" data set where it is slightly outperformed by the "constant with Nesterov's momentum" learning strategy. It also yields the largest test score on all datasets except for the "diabetes" dataset where it completely fails.
- The "constant with Nesterov's momentum" and "adam" learning strategies clearly outperform the other learning strategies. The "inv-scaling" learning strategy does not work on these datasets.

Remarks:

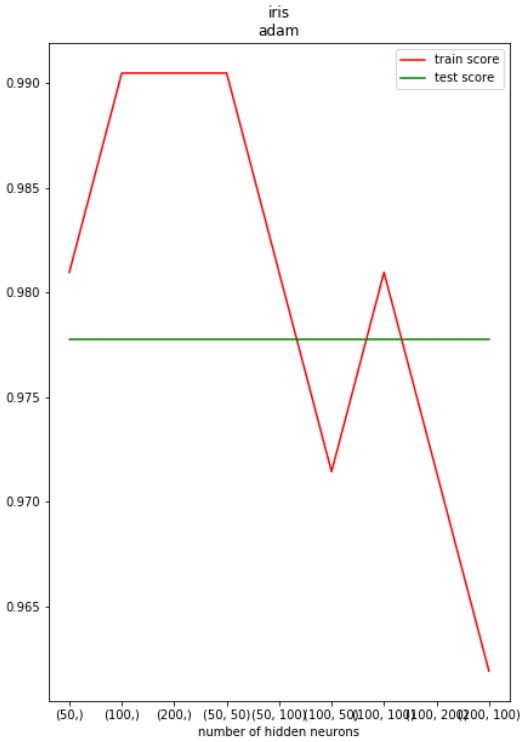
- By definition, the "score" method returns the fraction of correctly classified samples, for the training set ("train score") and the test set ("test score").
- Occasionally the test score is larger than the training score. It remains unclear whether these are statistical effects.
- Note that the "inv-scaling" learning strategy gradually decreases the learning rate at each time step t using an inverse scaling exponent of 0.5 (default), i.e. the effective learning rate is the initial learning rate (0.001 per default) divided by $\sqrt{t}$.
- The classifier loss function is the cross-entropy function, since this is the only loss function for classifiers currently supported by scikit-learn.
- The "adaptive" learning rate strategy produces similar results on these datasets as the "constant" learning rate strategy.
- Varying the batch size to 8 does not improve results on these datasets.

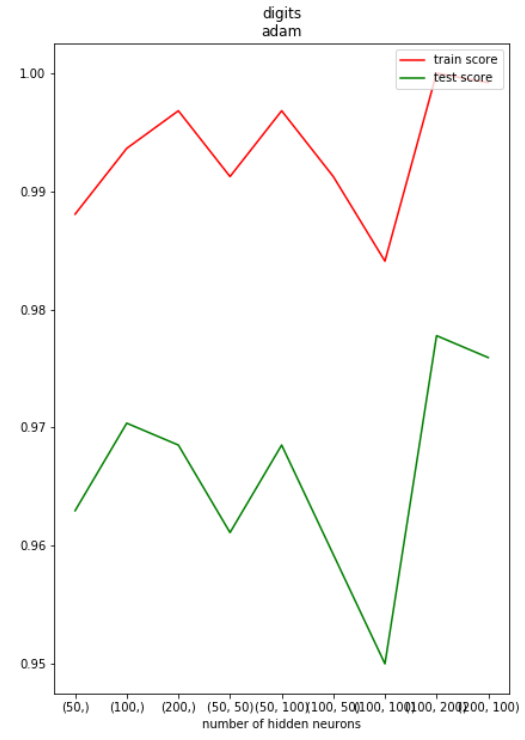
b) Using the most successful solvers from part a), namely the "constant with Nesterov's momentum" and the "adam" solver, in this part of the exercise different network topologies shall be investigated, i.e. the number of hidden layers and of hidden neurons shall be varied. A corresponding python script for this experiment is provided in **`exercise4b.py`**. Run the script, record the training loss, training score and test score and report on your conclusions regarding the network topology. You may also test further parameter settings.

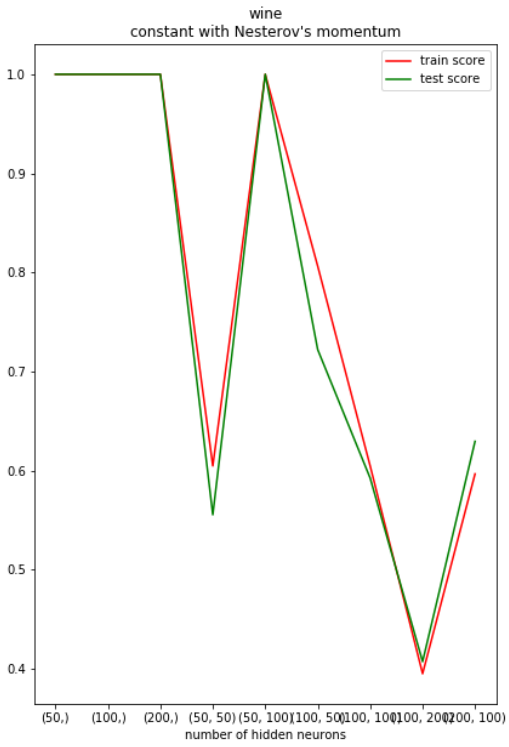
Solution:

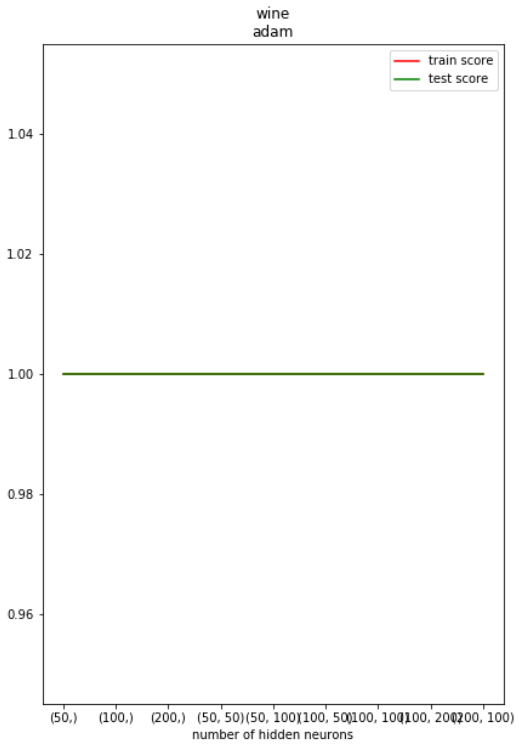
	constant with Nesterov's momentum			adam		
# hidden neurons	train loss	train score	test score	train loss	train score	test score
iris dataset (4-dim. inputs, 3 classes; 105 training / 45 test samples)						
(50,)	0.044191	0.980952	0.977778	0.041137	0.980952	0.977778
(100,)	0.044220	0.980952	0.977778	0.032600	0.990476	0.977778
(200,)	0.043957	0.980952	0.977778	0.028640	0.990476	0.977778
(50, 50)	2.417185	0.952381	0.977778	0.017459	0.990476	0.977778
(50,100)	1.802817	0.695238	0.600000	0.026727	0.980952	0.977778
(100, 50)	3.116060	0.695238	0.600000	0.036996	0.971429	0.977778
(100, 100)	7.687775	0.371429	0.244444	0.045795	0.980952	0.977778
(100, 200)	3.039461	0.714286	0.622222	0.035818	0.971429	0.977778
(200, 100)	2.684730	0.790476	0.622222	0.050978	0.961905	0.977778
digits dataset (4-dim. inputs, 10 classes; 1257 training / 540 test samples)						
(50,)	0.045275	0.992840	0.962963	0.079332	0.988067	0.962963
(100,)	0.040823*	0.996818*	0.964815*	0.060121	0.993636	0.970370
(200,)	0.038060	0.994431	0.959259	0.036015	0.996818	0.968519
(50, 50)	0.011243	0.998409	0.962963	0.038718	0.991249	0.961111
(50,100)	0.009057	1.000000	0.964815	0.036127	0.996818	0.968519
(100, 50)	0.008487	1.000000	0.964815	0.044263	0.991249	0.959259
(100, 100)	0.006815	1.000000	0.968519	0.027282	0.984089	0.950000
(100, 200)	0.004969	1.000000	0.972222	0.008503	1.000000	0.977778
(200, 100)	0.005606	1.000000	0.975926	0.011343	0.999204	0.975926
wine dataset (13-dim. inputs, 3 classes; 124 training / 54 test samples)						
(50,)	0.010980	1.000000	1.000000	0.010044	1.000000	1.000000
(100,)	0.010837	1.000000	1.000000	0.005622	1.000000	1.000000
(200,)	0.010758	1.000000	1.000000	0.003275	1.000000	1.000000
(50, 50)	0.789918	0.604839	0.555556	0.002438	1.000000	1.000000
(50,100)	0.002503	1.000000	1.000000	0.001251	1.000000	1.000000
(100, 50)	3.305745	0.806452	0.722222	0.001171	1.000000	1.000000
(100, 100)	4.139268	0.604839	0.592593	0.000968	1.000000	1.000000
(100, 200)	1.604230	0.395161	0.407407	0.000677	1.000000	1.000000
(200, 100)	0.478603	0.596774	0.629630	0.000731	1.000000	1.000000
diabetes dataset (10-dim. inputs, 181 classes; 309 training / 133 test samples)						
(50,)	0.644235	0.889968	0.000000	0.586971	0.877023	0.007519
(100,)	0.293299*	0.987055*	0.007519*	0.182448	0.993528	0.000000
(200,)	0.248848	0.993528	0.007519	0.058787	1.000000	0.007519
(50, 50)	2.229896	0.530744	0.007519	0.060670	1.000000	0.000000
(50,100)	2.695266	0.372168	0.000000	0.014533	1.000000	0.007519
(100, 50)	0.641581	0.834951	0.000000	0.130080	0.996764	0.007519
(100, 100)	3.159868	0.268608	0.000000	0.035263	0.996764	0.000000
(100, 200)	1.246339	0.634304	0.007519	0.009171	1.000000	0.007519
(200, 100)	2.710164	0.417476	0.000000	0.011300	1.000000	0.022556
circles dataset (2-dim. inputs, 2 classes; 70 training / 30 test samples)						
(50,)	0.134656	0.942857	0.866667	0.097614	0.957143	0.866667
(100,)	0.108059	0.942857	0.866667	0.093452	0.957143	0.866667
(200,)	0.130756	0.942857	0.866667	0.089193	0.957143	0.866667
(50, 50)	0.948953	0.485714	0.566667	0.090924	0.957143	0.866667
(50,100)	0.751686	0.528571	0.533333	0.085618	0.957143	0.866667
(100, 50)	0.823761	0.585714	0.533333	0.090043	0.957143	0.866667
(100, 100)	0.873508	0.485714	0.533333	0.094700	0.971429	0.866667
(100, 200)	1.032544	0.557143	0.466667	0.085112	0.957143	0.866667
(200, 100)	0.949931	0.585714	0.500000	0.085948	0.971429	0.866667

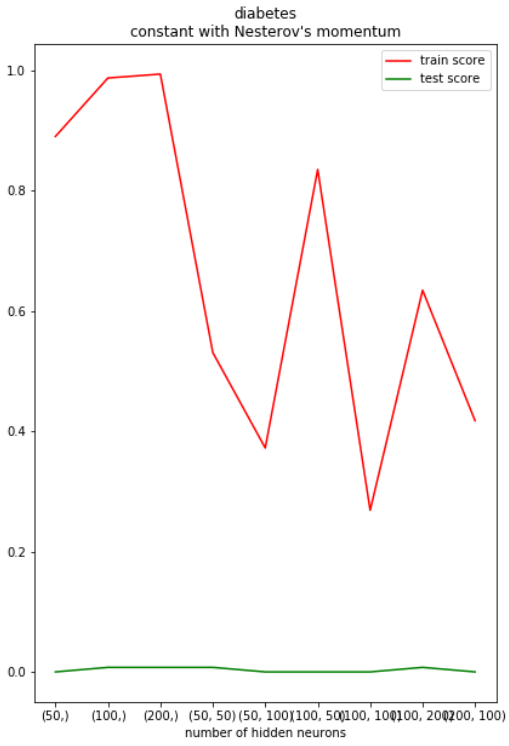
moon dataset (2-dim. inputs, 2 classes; 70 training / 30 test samples)						
(50,)	0.348124	0.814286	0.866667	0.293208	0.871429	0.800000
(100,)	0.348738	0.814286	0.866667	0.150447	0.942857	0.933333
(200,)	0.349493	0.814286	0.866667	0.287515	0.871429	0.800000
(50, 50)	0.357380	0.842857	0.866667	0.350794	0.842857	0.866667
(50,100)	0.357855	0.828571	0.866667	0.354062	0.842857	0.866667
(100, 50)	0.357650	0.842857	0.866667	0.143236	0.957143	0.933333
(100, 100)	0.358395	0.842857	0.866667	0.355984	0.857143	0.900000
(100, 200)	0.357861	0.842857	0.866667	0.156541	0.942857	0.933333
(200, 100)	0.359623	0.814286	0.866667	0.360911	0.828571	0.900000

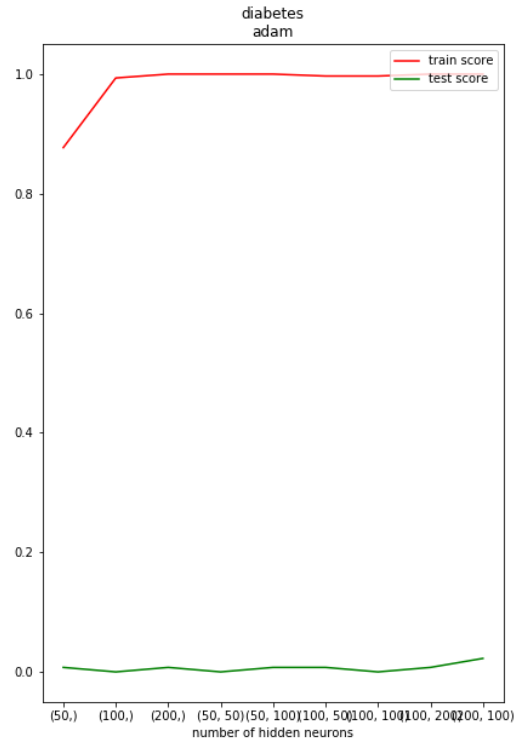


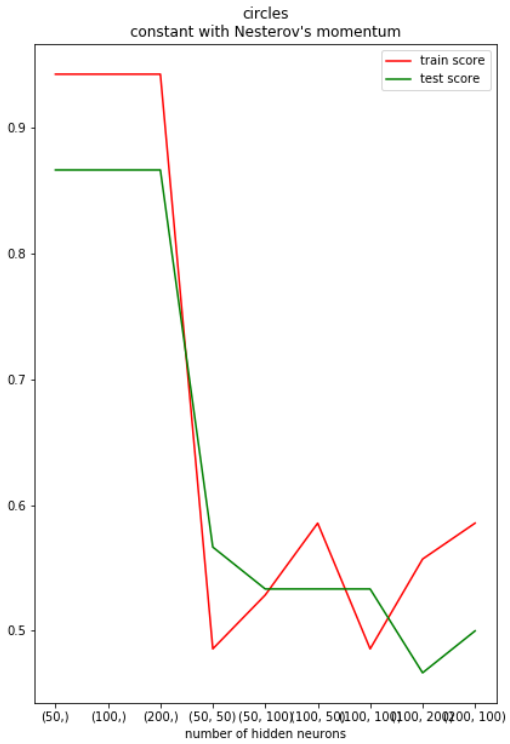


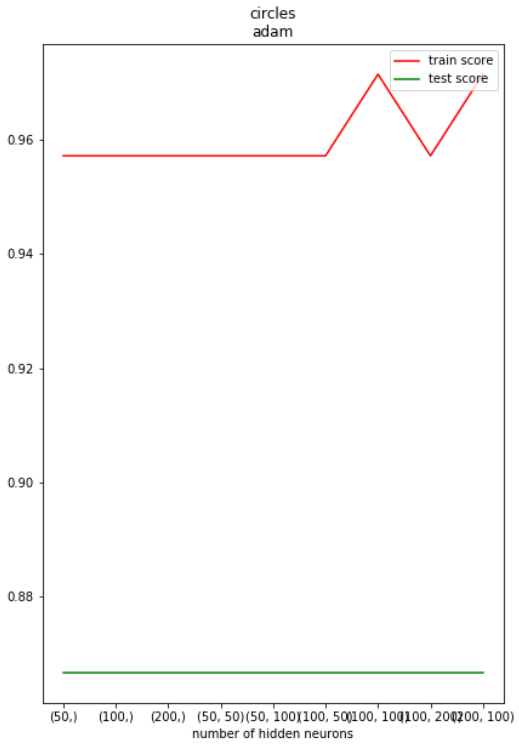

wine
constant with Nesterov's momentum
train score
test score
1.0
0.9
0.8
0.7
0.6
0.5
0.4
(50,)
(100,)
(200,)
(50, 50)
(50, 100)
(100, 50)
(100, 100)
(100, 200)
(200, 100)
number of hidden neurons

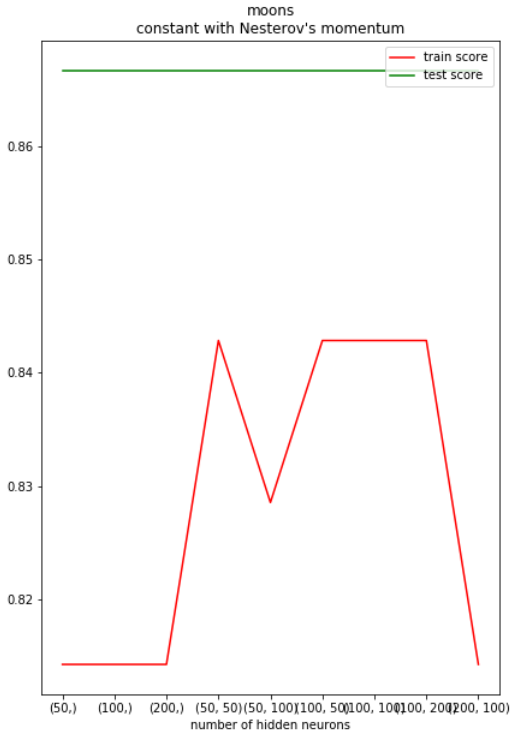

wine
adam
train score
test score
1.04
1.02
1.00
0.98
0.96
(50,)
(100,)
(200,)
(50, 50)
(50, 100)
(100, 50)
(100, 100)
(100, 200)
(200, 100)
number of hidden neurons

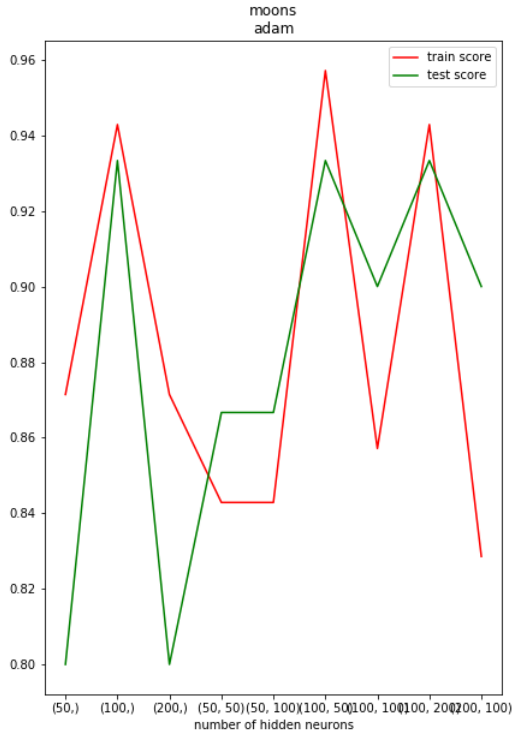

diabetes
constant with Nesterov's momentum
train score
test score
1.0
0.8
0.6
0.4
0.2
0.0
(50,)
(100,)
(200,)
(50, 50)
(50, 100)
(100, 50)
(100, 100)
(100, 200)
(200, 100)
number of hidden neurons


diabetes
adam
train score
test score
1.0
0.8
0.6
0.4
0.2
0.0
(50,)
(100,)
(200,)
(50, 50)
(50, 100)
(100, 50)
(100, 100)
(100, 200)
(200, 100)
number of hidden neurons









Conclusions:

- By varying the network topology, results are hardly improved. There is a very tiny improvement on the “digits” dataset and on the “diabetes” dataset – but on the “diabetes” dataset the test error is still close to zero. The digits dataset consists of a comparably large number of data samples such that a network with more parameters can be trained. For the other datasets, this does not seem to be the case.

- Therefore, a multi-layer perceptron with a single hidden layer and 100 hidden units is sufficient for these datasets.