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

Neural Networks and Deep Learning – Summer Term 2020

Exercise sheet 3

Submission due: Wednesday, May 20, 13:15 sharp

Note:

Some of the following exercises are based on the Keras and TensorFlow libraries. TensorFlow is an open source platform for machine learning, supporting matrix (tensor) multiplications on graphical processing units (GPUs) and therefore ideally suited for (deep) neural networks; for more information see https://www.tensorflow.org/. Keras is a high-level neural networks application programming interface (API) running on top of TensorFlow or other libraries like Theano or CNTK with the aim to facilitate using those libraries; see https://keras.io/.

Python code to configure neural networks using Keras and Tensorflow has been provided in form of a Jupyter notebook, **`Sheet_3.ipynb`**; for documentation on Jupyter and Jupyter notebooks see https://jupyter.org/. **`Sheet_3.ipynb`** needs additional resources like images etc.; all necessary files are contained in a git repository in the directory **`Lab3`**. In order to copy those files to your computer, you have to install **`git`** on your computer; see e.g. https://git-scm.com/download/win.

One way to copy the required lab files to your computer is to use a git bash shell: On Windows, open an explorer and navigate to a folder where you want to copy all relevant lab files. Then (after installing an appropriate git package), right click the mouse in that directory and select the option "Git Bash Here". A git bash shell opens; in that shell, enter the following command:

```
git clone https://gitlab+deploy-token-26:XBza882znMmexaQSpjad@git.informatik.uni-kiel.de/las/nndl.git <your target directory>
```

where **`<your target directory>`** has to be replaced by the path of the directory on your local computer that you want to generate and which should contain the downloaded lab files. After cloning, this directory then should contain a folder **`Lab3`** with the file **`Sheet_3.ipynb`** and other text files.

For the python code involving Keras and TensorFlow to work, there are two possibilities:

a) (recommended) Execute the code in Google Colab (https://colab.research.google.com/notebooks/intro.ipynb). To this end, you need to create a Google account. Log in and open https://colab.research.google.com/. In the upper right corner, select the "Upload" tab and then "Browse". Locate the cloned **`Lab3`** directory and select the file **`Sheet_3.ipynb`**. Note that you have to run the first cell (containing **`! git clone`** ...) before running any other cell; otherwise you may get an error message from subsequent cells like
`OSError: nndl/Lab3/exercise3b_input.txt not found.`

   Note that with this option external images are not displayed correctly; instead, you only see e.g. the following line:
`IMAGE: perceptron`
For external images to be displayed correctly, option b) has to be invoked (or refer to the pdf exercise sheet).

b) Run the notebook locally. In order for the Jupyter notebook to work properly, also the files and the subdirectory of **`Lab3`** have to be loaded into Jupyter notebook. A possibility for that is to invoke Jupyter notebook from an Anaconda installation (see https://www.anaconda.com/) via an Anaconda Prompt (NOT directly selecting the Jupyter Notebook option): Open an Anaconda prompt, navigate to the **`Lab3`** directory just cloned, and then enter
`jupyter-notebook`
into the Anaconda prompt. This opens Jupyter notebook *including* all necessary files. Then, click on **`Sheet_3.ipynb`**; now you can start to work with the Jupyter notebook (and should see external images, which are contained in the **`images`** subdirectory, properly). If you want to execute the cells involving Keras and TensorFlow locally on your computer, you have to install Keras and TensorFlow locally and to configure your environment appropriately.

Run one cell after the other; this should finally provide you with some plots being generated.

Note that changes in **`Sheet_3.ipynb`** which are saved in the colab environment are *not* saved to **`Lab3/Sheet_3.ipynb`**!

Further documentation how to install Jupyter notebooks can be found at:

https://jupyterlab.readthedocs.io/en/stable/getting_started/installation.html
https://jupyter.readthedocs.io/en/latest/install.html#install

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…)

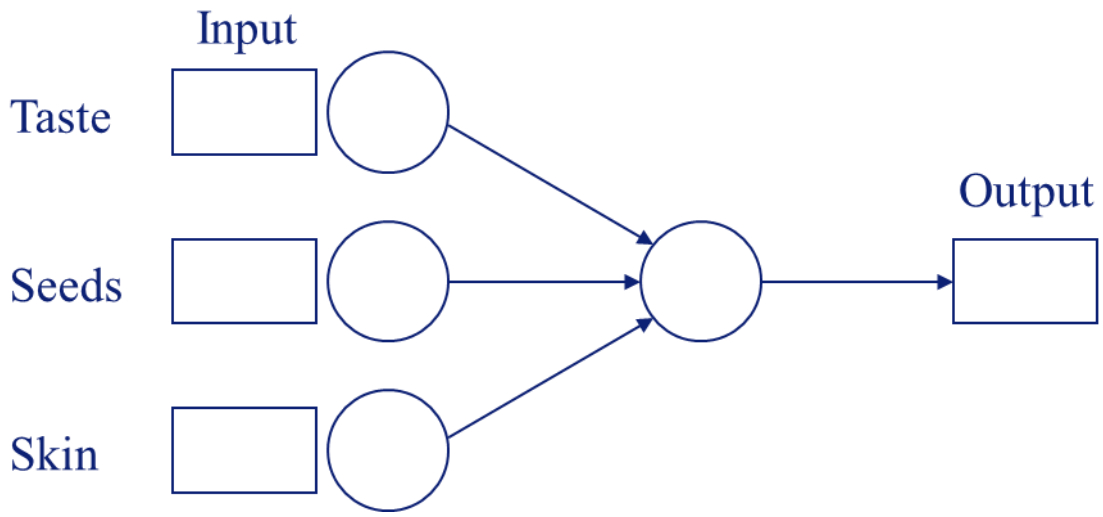
- Cross-validation
    - Description: see above. By estimating the generalization error on *independent* test data, cross-validation can indicate overfitting. To avoid overfitting, early stopping can be used (see above).

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

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

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

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

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

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

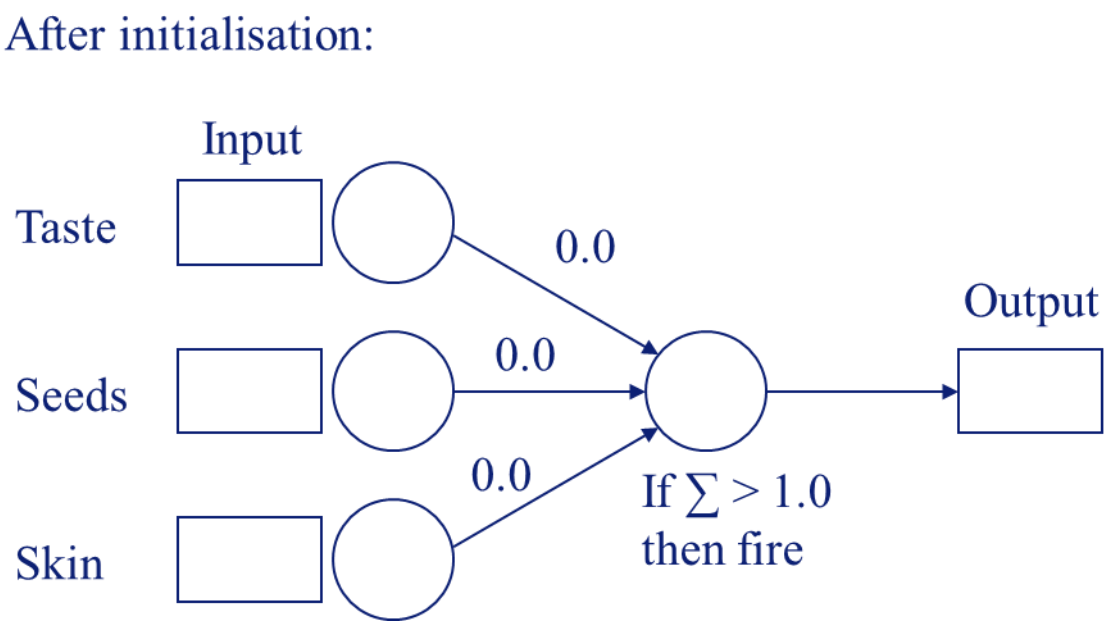
Application of the perceptron learning algorithm involves the following steps:

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

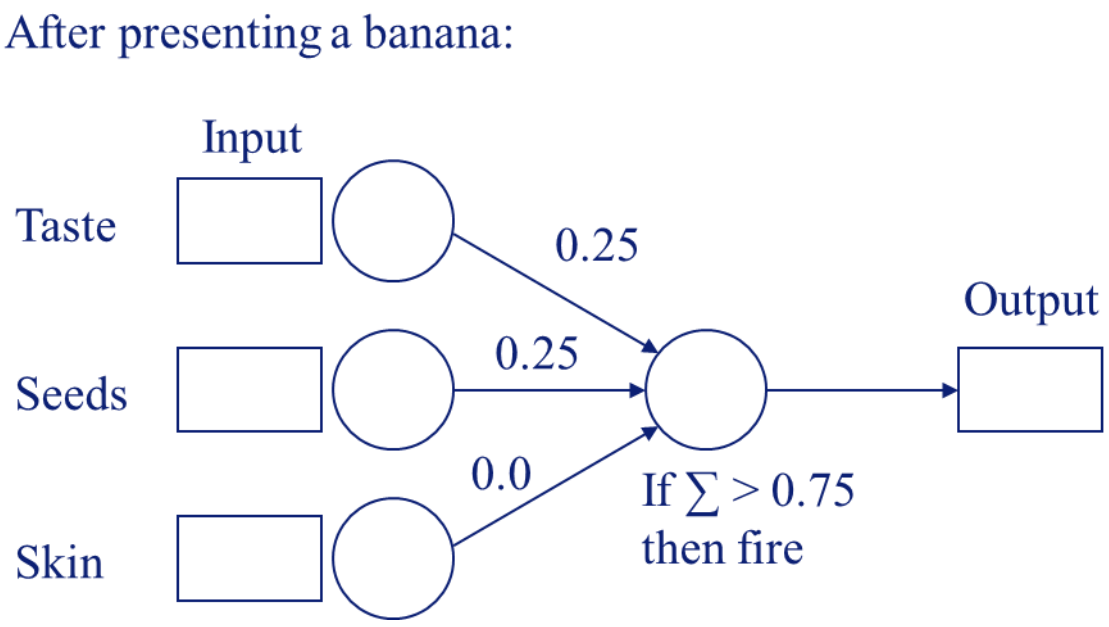
$$\Delta\mathbf{w}(1) = 0.25 \cdot (1-0) \cdot \mathbf{x}^{(1)} = 0.25 \cdot \mathbf{x}^{(1)}$$

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

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.

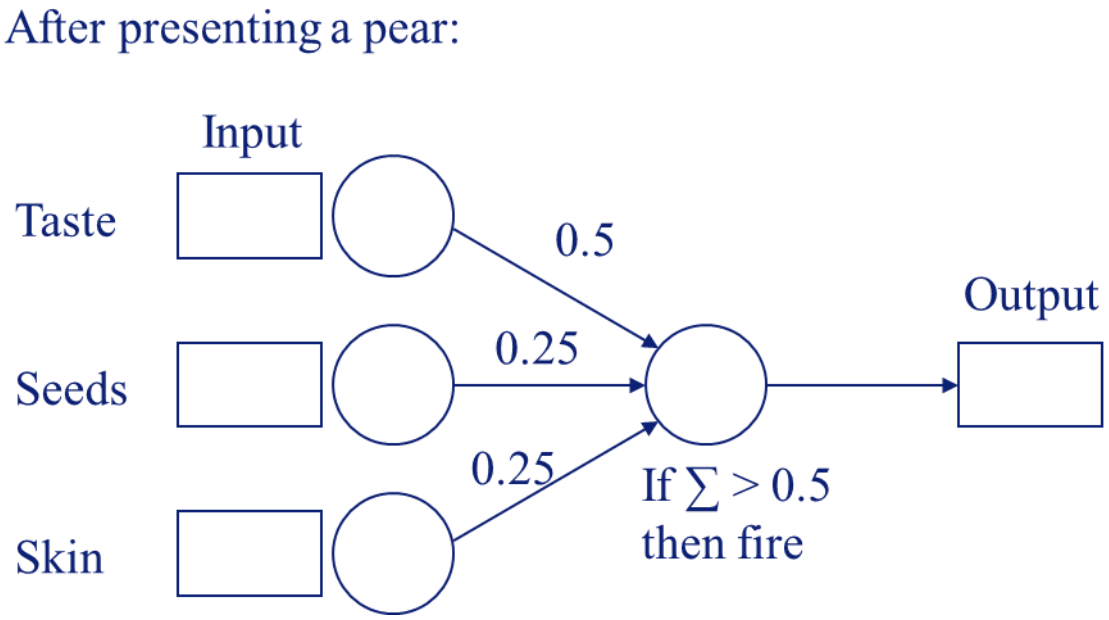
The weight update is given by

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

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

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

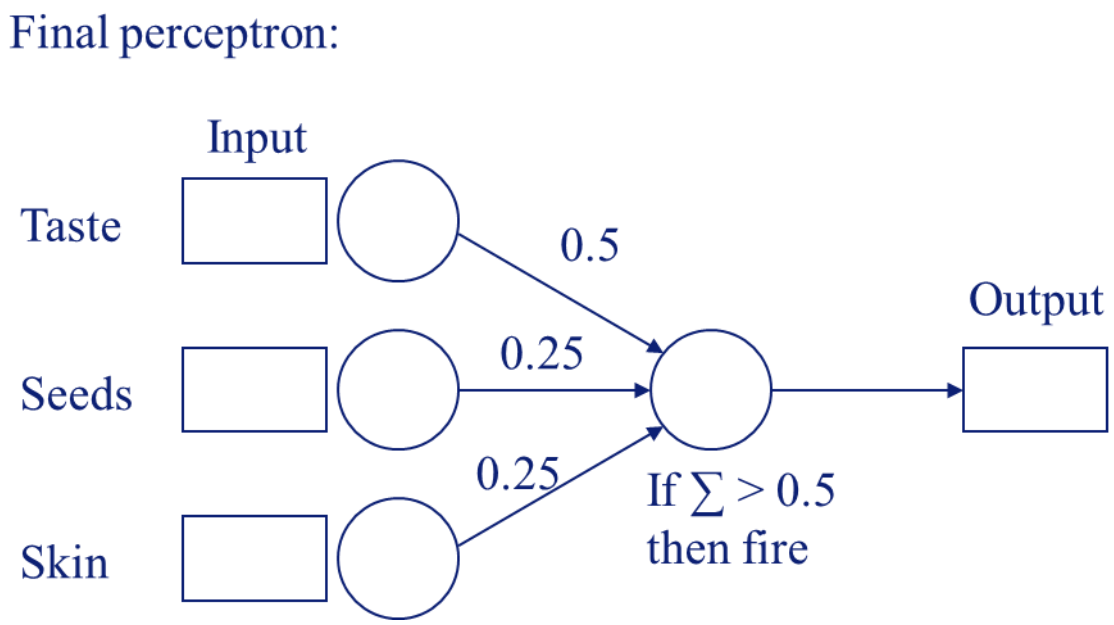
According to the table, the target output for a green apple is $d^{(5)} = 0$. Thus, the perceptron was correct and there is no weight modification. This can also be seen as follows:

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

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 TensorFlow. 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.txt`**. 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 **`sigmoid`** activation function – shall be used to provide a solution to this classification problem. Note that due to the **`sigmoid`** activation function, the output of the perceptron is a real value in [0,1]:

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

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". 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 in python code at # *calculate final decision boundary*):

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

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 TensorFlow and Keras libraries. The source code is given below and can be executed by clicking the play button (see remarks above).

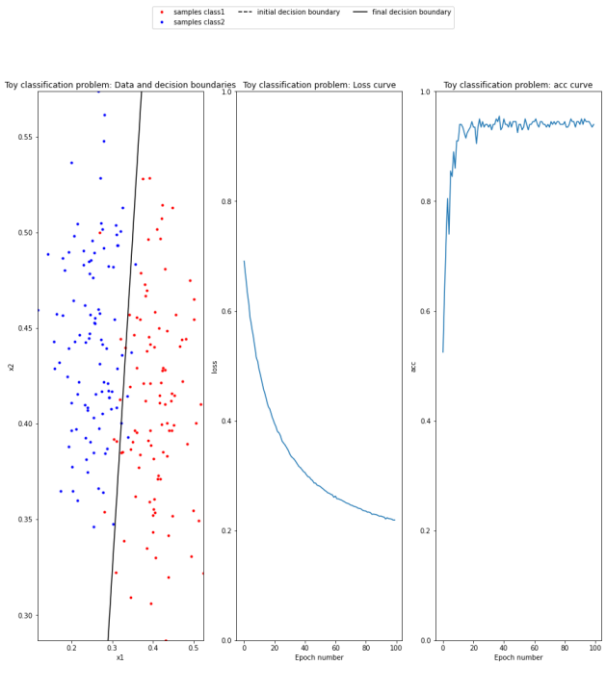
1. Train the model at least three times and report on your findings.
2. Change appropriate parameters (e.g. the learning rate, the batch size, the choice of the solver, potentially the number of epochs etc.) and again report on your findings.

Solution:

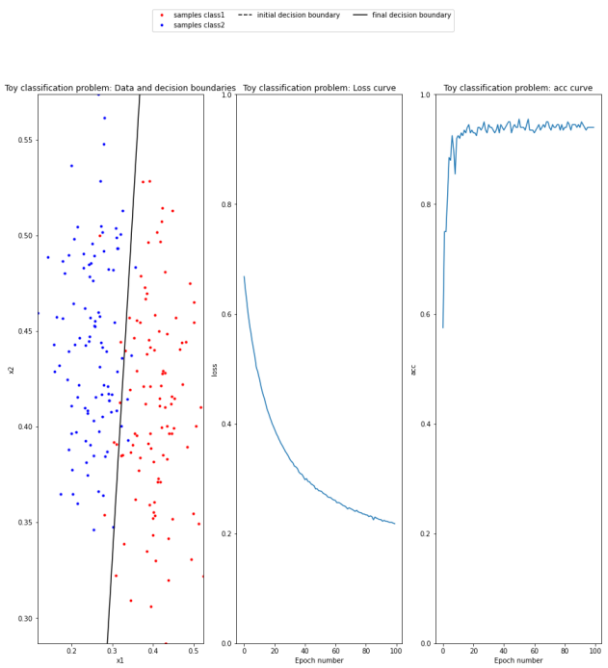
The following results have been obtained for the parameters
Learning rate: 0.1
Batch size: 1
Solver: 'SGD' (stochastic gradient descent)

	1st run	2nd run	3rd run	4th run
Initial weights	(-0.031733, -0.270382)	(-0.888901, -0.606707)	(-1.112785, -0.721842)	(-1.055869, 1.228241)
Initial bias	0.000000	0.000000	0.000000	0.000000
Final weights	(-25.277103, 7.234186)	(-25.457052, 7.044006)	(-25.462023, 7.026671)	(-25.388290, 7.772947)
Final bias	5.254530	5.315825	5.408777	5.049080
Final loss	0.207869	0.206733	0.206840	0.206947
Final accuracy	0.945000	0.945000	0.945000	0.940000
Binary errors	11	11	11	12

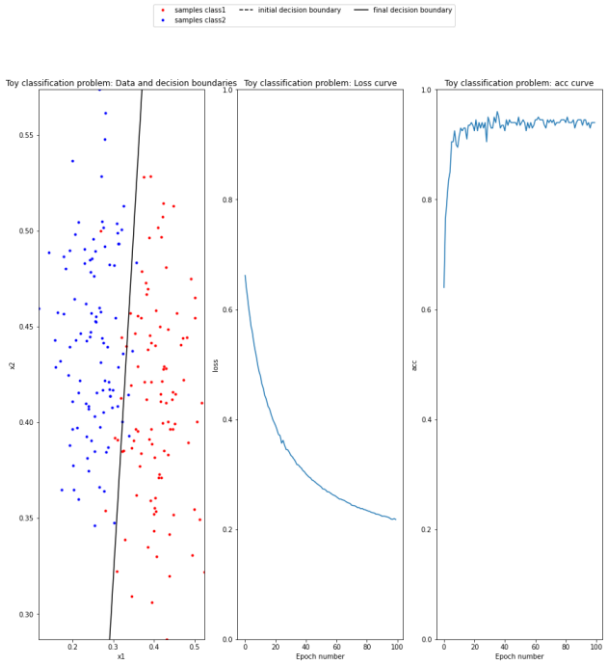
First run:



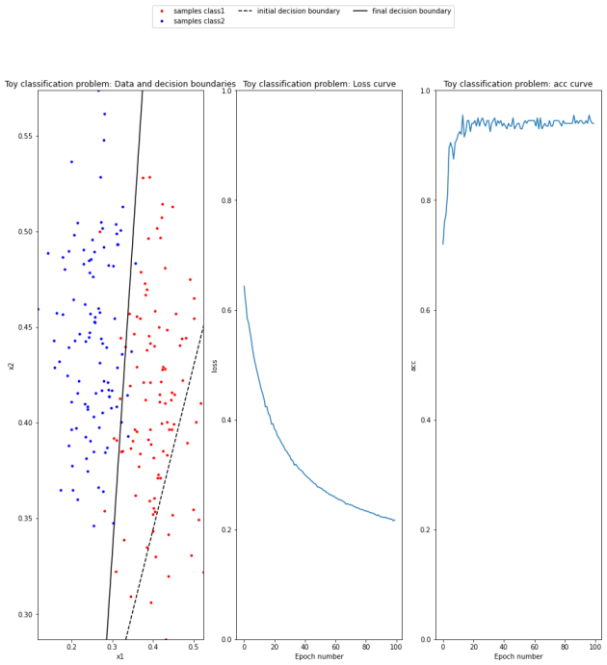
Second run:



Third run:



Fourth run:



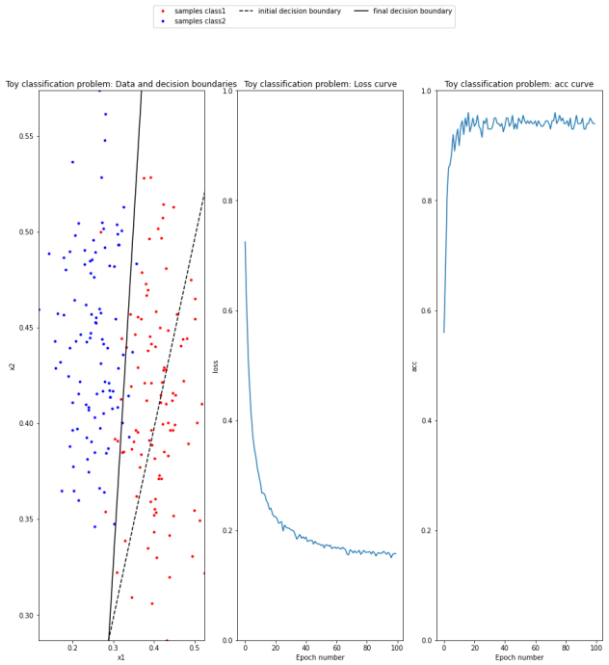
(Note: For the first to third run, the initial decision boundary is not visible in the plots.)

In spite of different initial weights, nearly the same final decision boundary has been obtained and the final loss and accuracy differ only slightly.

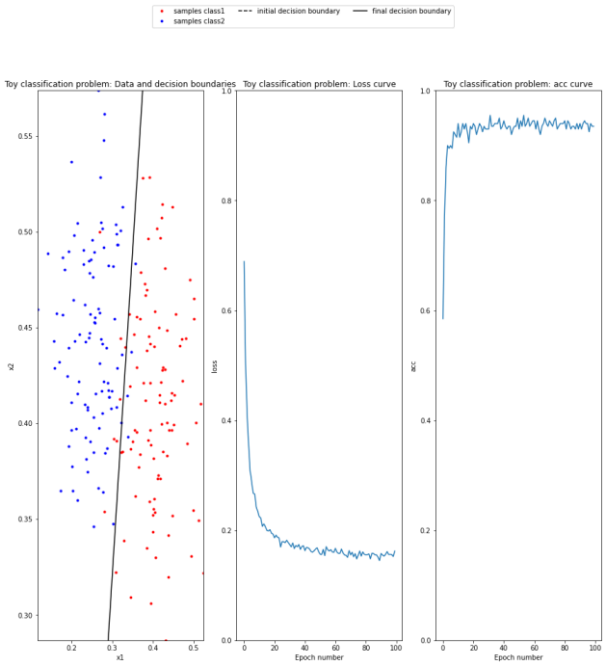
Some modifications of the model and training configuration lead to the following results:

	Config. A: learn. rate 0.5 SGD, batch size 1	Cofig. B: learn. rate 1.0 SGD, batch size 1	Cofig. C: learn. rate 0.01 SGD, batch size 1	Config. D: learn. rate 0.5 Adam, batch size 1
Initial weights	(1.294503, -1.303941)	(0.903776, 0.559002)	(-0.343031, -0.346003)	(1.015592, 1.171410)
Initial bias	0.000000	0.000000	0.000000	0.000000
Final weights	(-46.351772, 13.003973)	(-57.160797, 16.817665)	(-6.120237, 1.552202)	(-93.910973, 27.380003)
Final bias	9.677529	11.772877	1.326252	18.050692
Final loss	0.138034	0.129534	0.485186	0.134807
Final accuracy	0.945000	0.945000	0.945000	0.945000
Binary errors	11	11	11	11

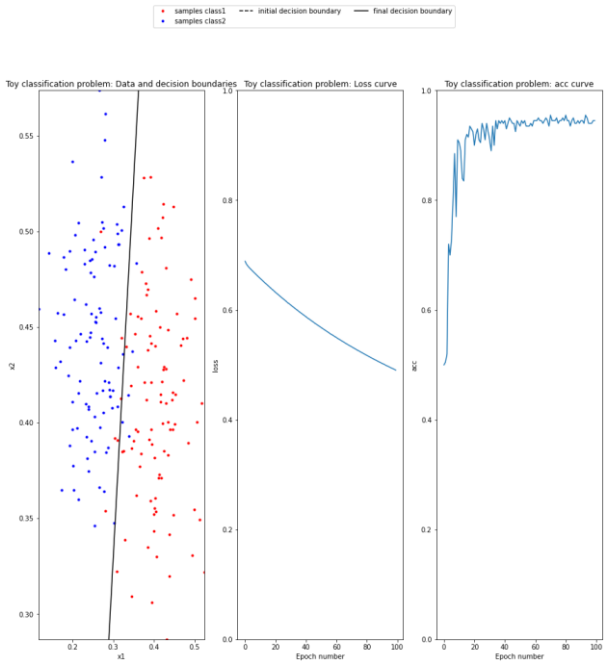
Config. A:



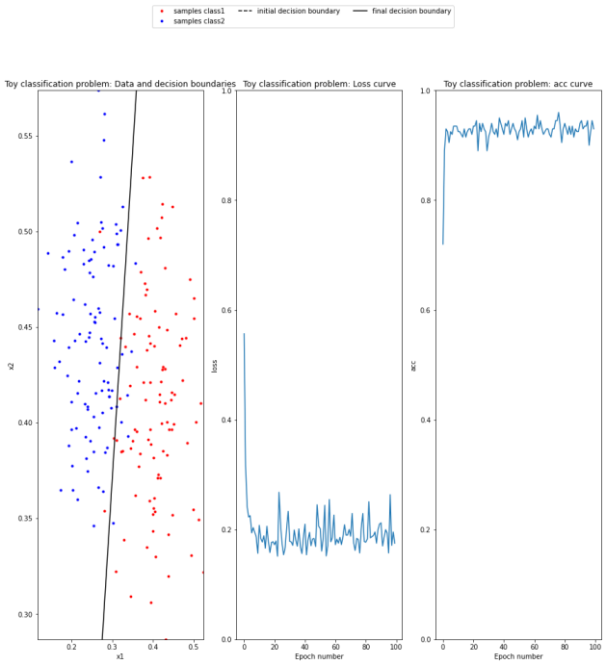
Config. B:



Config. C:



Config. D:

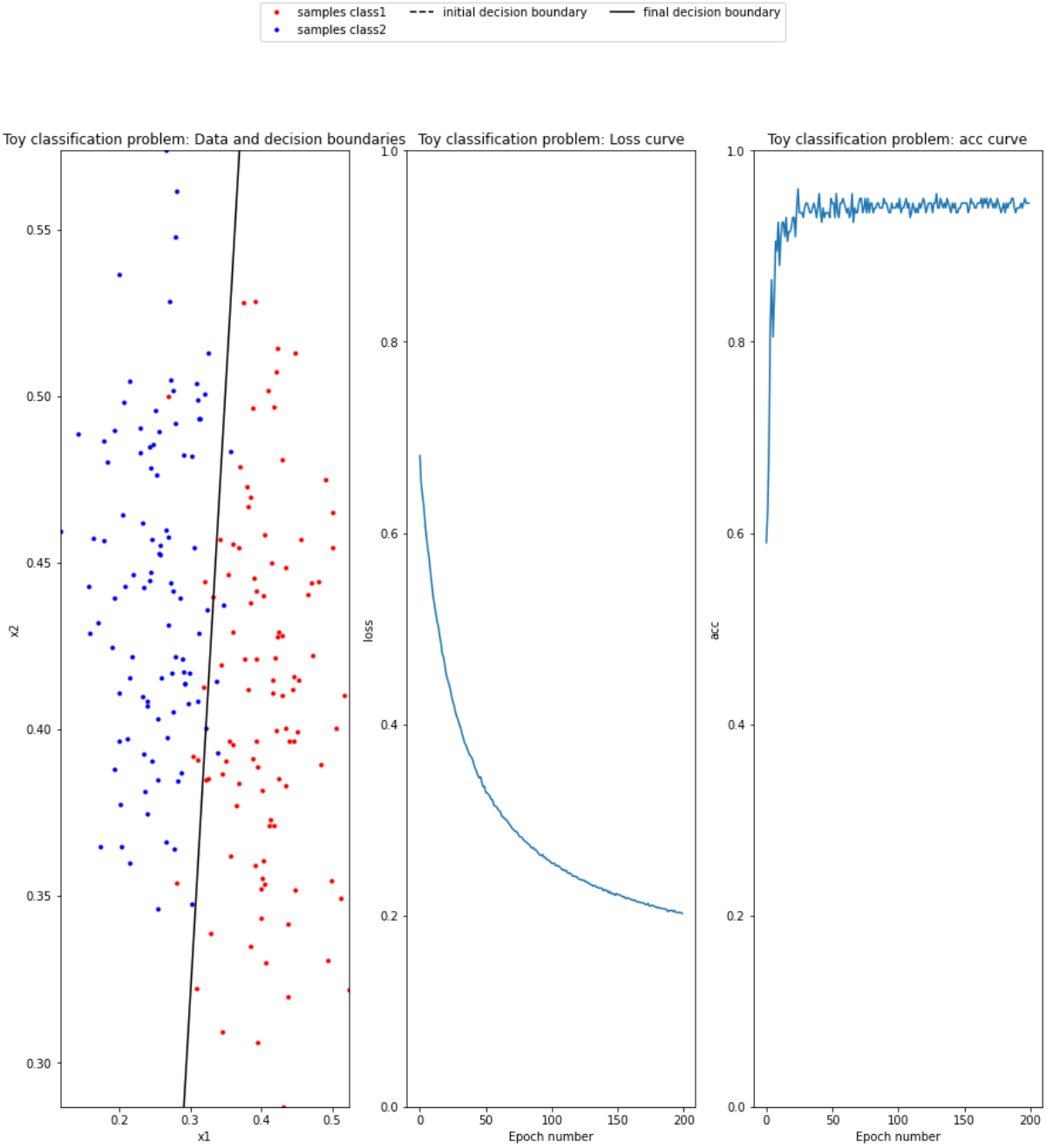


As apparent from the figures and the numbers, all configurations produce similar final results. Only the loss curve reveals that convergence (with respect to the loss) becomes faster with larger learning rates (configuration A), but may fluctuate slightly more (configuration B). For a smaller learning rate (configration C), convergence is even slower. For the Adam optimizer, the loss seems to quickly converge, but also shows stronger fluctuations. Note that these findings only relate to the present task and could be different for other tasks and network configurations!

As a final configuration, a learning rate of 0.5 is tested with the SGD optimizer for a batch size of 8 and 200 epochs:

Initial weights: (-0.953930, -1.073700), initial bias: 0.000000
Final weights: (-27.981892, 7.625711), final bias: 5.963410

Final loss: 0.192292, final accuray: 0.940000
number of binary errors: 12



Again, this leads to a similar solution. Convergence seems to be slower compared to configuration A due to the larger batch size.

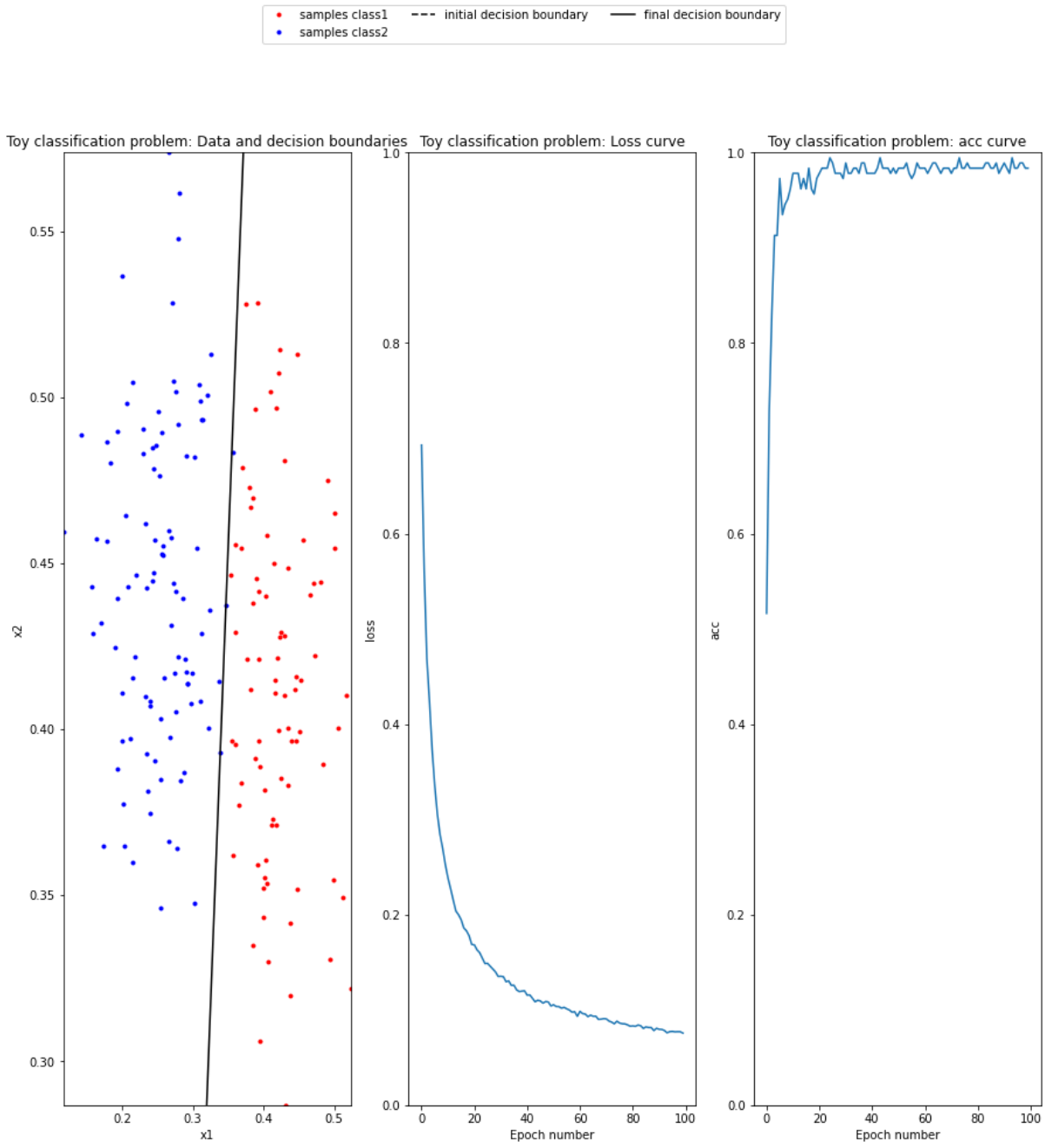
c) Repeat exercise b) with the training set **`exercise3c_input.txt`** and the targets **`exercise3c_target.txt`**. Those points have been generated from the input points of exercise b) by removing points from class 1 (i.e. those points the x-coordinate of which is below 0.35). Do not forget to modify 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 a learning rate of 0.5 and the SGD optimizer with a batch size of 1 the following results have been obtained:

Initial weights: (1.121667, 0.818998), initial bias: 0.000000
Final weights: (-49.002338, 8.845484), final bias: 13.130134

Final loss: 0.072761, final accuray: 0.994565
number of binary errors: 1



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 (in this run there was only 1 error). Correspondingly, the final loss is much lower and the accuracy much larger.

d) Divide the input samples from part b) into separate training and validation sets, where the latter shall comprise 30% of the data. You may use available Keras functionality for this purpose. Run the script at least two times, plot the training and validation loss and accuracy as a function of the epoch number and report on your findings.

Solution:

The available data can be divided into a training and a validation set using the **`validation_split`** parameter in the **`Keras Model.fit()`** method. Note that using this parameter, the validation data is selected from the last samples in the input and target data provided, before shuffling (such that care has to be taken that the input and target data are not ordered, which, however, is satisfied for the data of this exercise):

```
# Train the model
history = model.fit(x=input, y=target, batch_size=1, epochs=100,
                    verbose=True, validation_split=0.3)
```

To plot the training and validation loss and accuracy, the following changes to the code have been made:

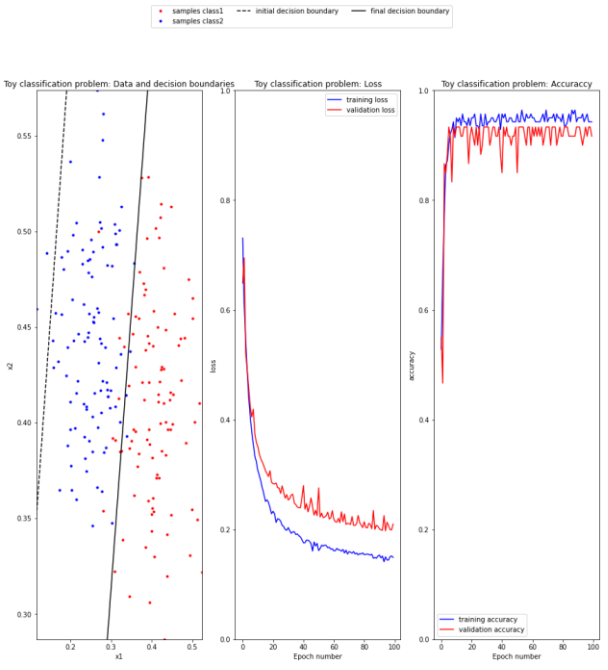
```
# plot final decision boundary
…

# plot loss
axes[1].set_title('Toy classification problem: Loss')
axes[1].set_xlabel('Epoch number')
axes[1].set_ylabel('loss')
axes[1].set_ylim(0, 1)
axes[1].plot(history.history['loss'], color = 'blue',
             label = 'training loss')
axes[1].plot(history.history['val_loss'], color = 'red',
             label = 'validation loss')
axes[1].legend()

# plot accuracy
axes[2].set_title('Toy classification problem: Accuraccy')
axes[2].set_xlabel('Epoch number')
axes[2].set_ylabel('accuracy')
axes[2].set_ylim(0, 1)
axes[2].plot(history.history['acc'], color = 'blue',
             label = 'training accuracy')
axes[2].plot(history.history['val_acc'], color = 'red',
             label = 'validation accuracy')
axes[2].legend()

# show the plot
…
```

Note that the code below **`# final evaluation`** has to be outcommented and ignored, since that evaluation is carried out on all data (i.e., training *and* validation data).

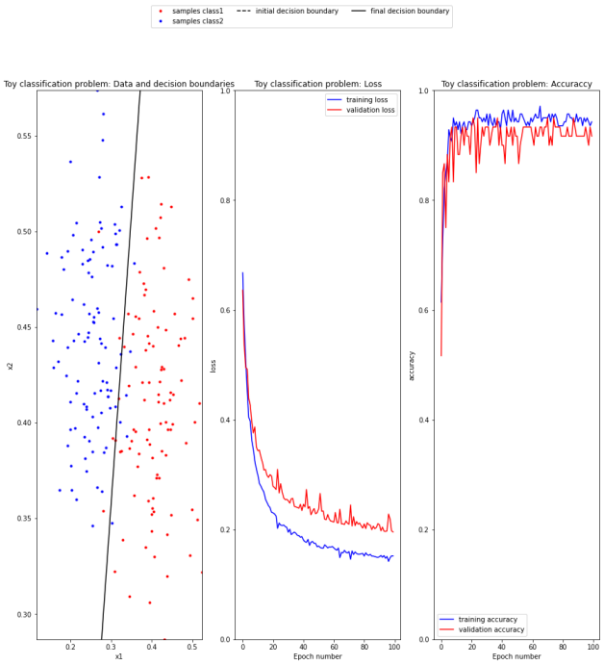
Using a learning rate of 0.5 and the SGD optimizer with a batch size of 1 the following results have been obtained:

	1st run	2nd run
Initial weights	(0.803350, 0.288569)	(-1.314589, -1.263640)
Initial bias	0.000000	0.000000
Final weights	(-40.223831, 13.854180)	(-40.574539, 13.388640)
Final bias	7.73869	7.402119
Final training loss	0.1457	0.1520
Final training accuracy	0.9500	0.9429
Final validation loss	0.2072	0.1955
Final validation accuracy	0.9333	0.9167

First run:

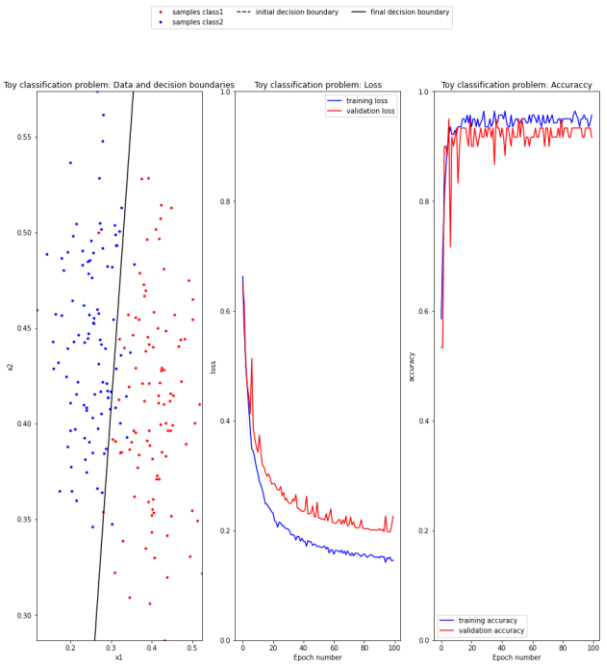


Second run:

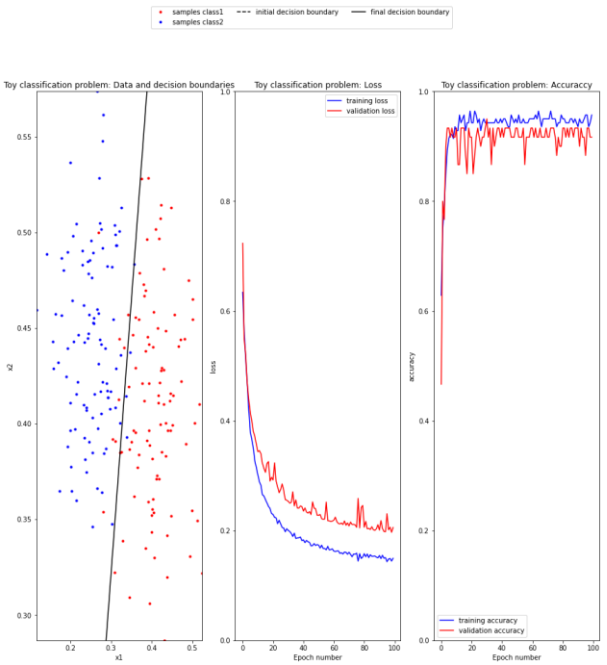


	3rd run	4th run
Initial weights	(-0.636345, -0.541661)	(-0.331303, 1.355813)
Initial bias	0.000000	0.000000
Final weights	(-40.632710, 13.432286)	(-40.314362, 14.083259)
Final bias	6.717554	7.575288
Final training loss	0.1459	0.1494
Final training accuracy	0.9571	0.9571
Final validation loss	0.2258	0.2056
Final validation accuracy	0.9167	0.9167

Third run:



Fourth run:



As expected, the validation loss is larger than the training loss and the validation accuracy lower than the training accuracy. Overfitting is not observed.

e) Modify the script 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.

Useful information about training and evaluation with Tensorflow and Keras can be found at https://www.tensorflow.org/guide/keras/train_and_evaluate

Solution:

The following code modifications have been made:

```
###-----------------
# load training data
###-----------------
input = np.array([[0,0],[0,1],[1,0],[1,1]])
target = np.array([0, 1, 1, 0])
class1 = np.array([[0,0],[1,1]])
class2 = np.array([[0,1],[1,0]])
...

# Train the model
history = model.fit(x=input, y=target, batch_size=1, epochs=100,
                    verbose=True)
```

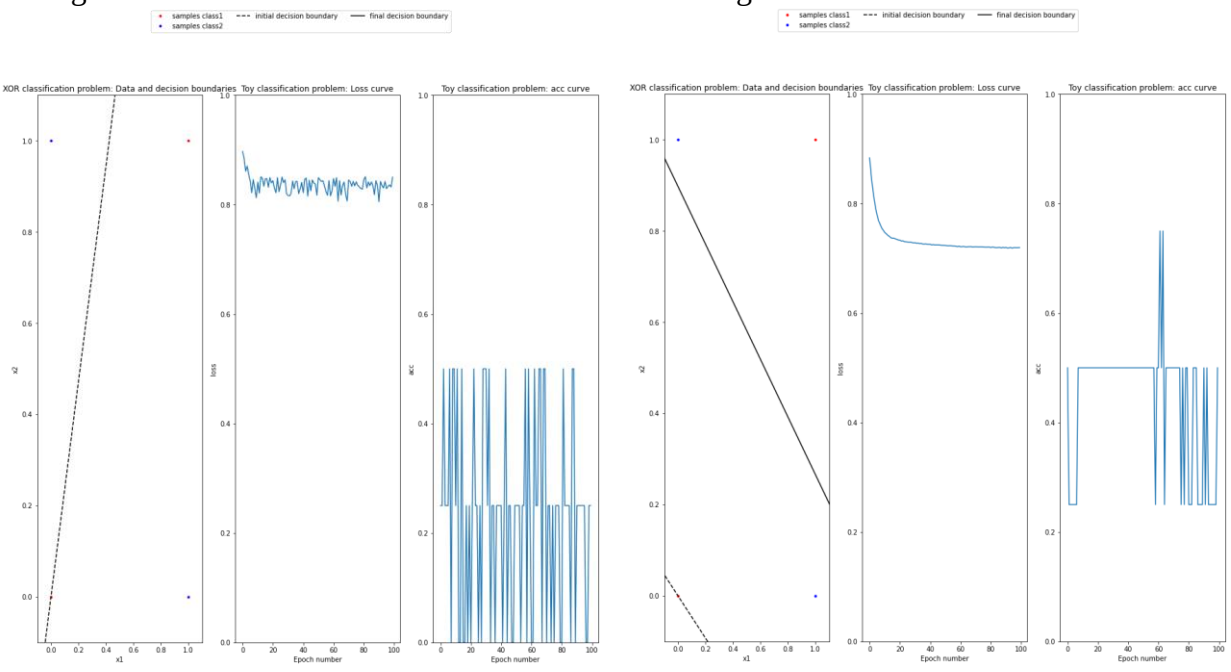
```
...
# plot the data
axes[0].set_title('XOR classification problem: Data and decision
                   boundaries')
axes[0].set_xlabel('x1')
axes[0].set_ylabel('x2')

minx = min(input[:,0]-0.1)
maxx = max(input[:,1]+0.1)
miny = min(input[:,0]-0.1)
maxy = max(input[:,1]+0.1)

...
```

Using SGD and a batch size of 1 the following results have been obtained:

	Config. A: learning rate 0.5	Cofig. B: learning rate 0.1
Initial weights	(1.069273, -0.453007)	(-0.638539, -1.377116)
Initial bias	0.000000	0.000000
Final weights	(-0.090280, -0.015232)	(-0.111185, -0.175935)
Final bias	0.19006	0.157940
Final loss	0.8501	0.7196
Final accuracy	0.2500	0.5000

Note that the values of the final loss and final accuracy (obtained after 100 epochs) do not correspond to the values obtained after the final evaluation.

Config. A:



Config. B:

The loss is (much) larger than 0.5 and the accuracy lower 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 Tensorflow Keras library, to four classification problems provided by the UCI repository (and contained in the scikit learn package; i.e. iris, digits, wine, breast_cancer) 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, a number of solvers (stochastic gradient descent, Adam, Adam with Nesterov momentum, AdaDelta, AdaGrad or RMSProp) shall be applied to the six datasets. An (incomplete) python script for this experiment is provided the Jupyter notebook. Complete the code (model definition, selection and configuration of an optimizer and model "compilation" including selection of an appropriate loss function ; see # TO BE ADAPTED in the Jupyter notebook); consult the Tensorflow Keras documentation if needed. Furthermore, select suitable values of the most important parameters (e.g. learning rate, batch size...). Then, apply the script for at least three different optimizers, for a suitable baseline model configuration. Report the final training and validation loss and accuracy values and provide plots for the training and validation loss and accuracy curves as a function of the number of epochs (see script). What are your conclusions regarding the comparison of the optimization strategies? Also report on the database statistics.

   The optimizer is selected e.g. with
   ```
   opt = SGD(learning_rate=lr) # SGD or Adam, Nadam, Adadelta, Adagrad, RMSProp
   ```
   Note that additional parameters of the optimizers can be set if desired (see the Tensorflow Keras documentation).

Solution:

The network configuration is specified with
```
input_layer = Input(shape=(num_inputs,), name='input')
hidden_1 = Dense(units=num_hidden, activation="relu", name="hidden_1")(input_layer)
out = Dense(units=num_outputs, activation="softmax", name="output")(hidden_1)
```

Since the softmax activation is used, the targets have to be encoded in one-hot notation (see Jupyter notebook below `# one-hot encoding`) and the log-likelihood (or categorical cross-entropy) loss has to be selected as loss function:
```
model.compile(optimizer=opt,loss="categorical_crossentropy",
              metrics=["categorical_accuracy"])
```

The optimizer is selected e.g. with
```
opt = SGD(learning_rate=lr)
```
or
```
opt = Adam(learning_rate=lr)
```
or
```
opt = RMSprop(learning_rate=lr)
```

etc.

In order for the input values to better match potential initial weight values, the inputs are normalized (see Jupyter notebook below **`# normalize inputs`**). You may experiment also with non-normalized inputs.

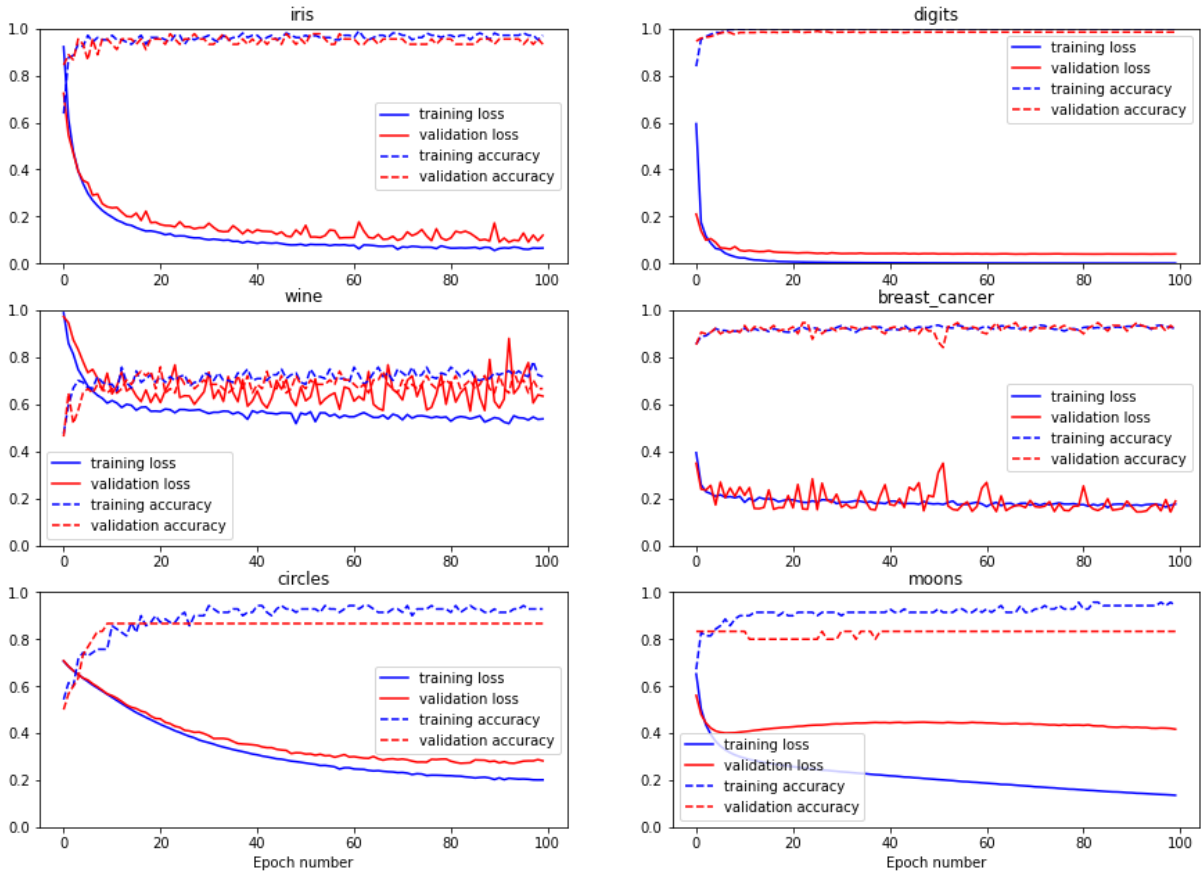
Furthermore, again a separate validation set (containing 30% of the data) is extracted from the data set. In order to avoid any potential bias, the inputs (and targets in the same way) are shuffled (see Jupyter notebook below **`# shuffle data`**).

Before varying the optimizer, a suitable value for the learning rate and an appropriate base network topology should be selected in some initial experiments. Here, the following baseline setting is used:

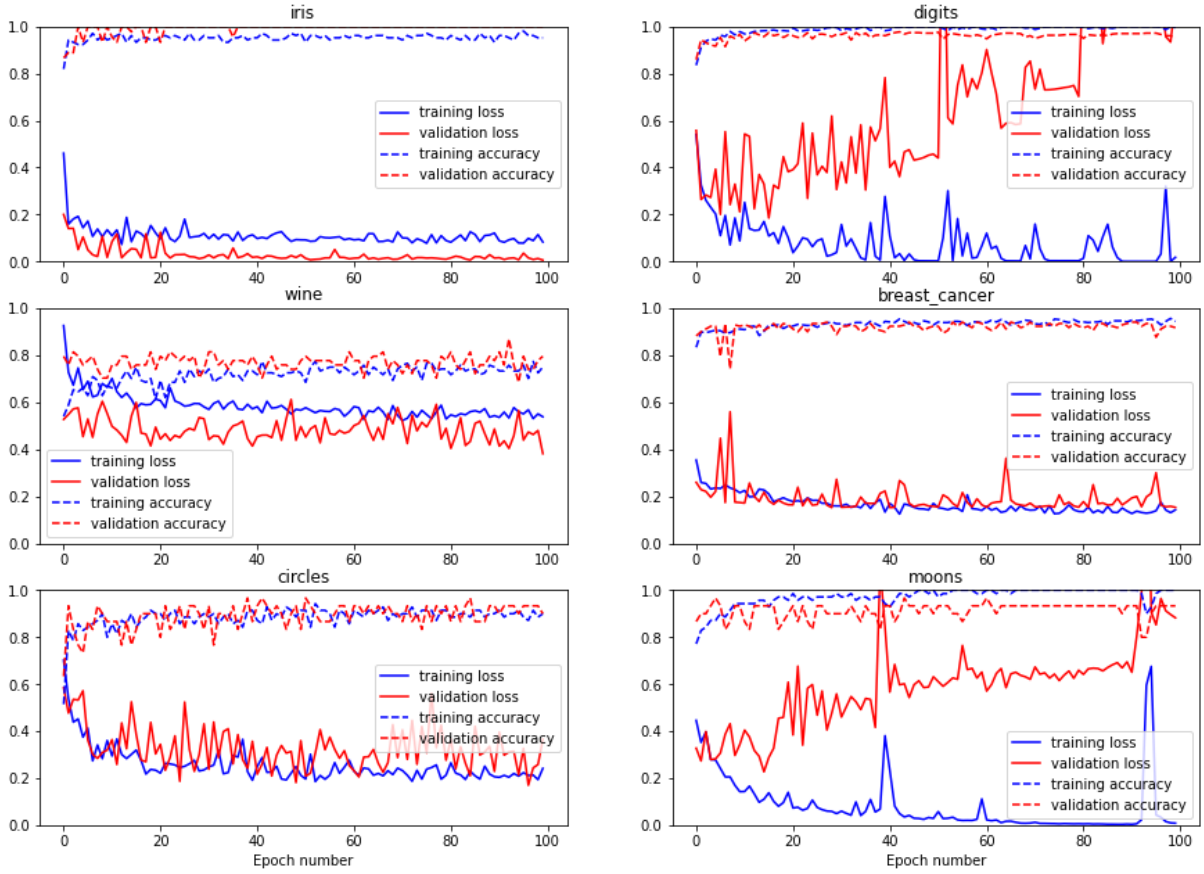
- Network topology: MLP with a single hidden layer containing 100 neurons
- Activation function: ReLU at hidden layer and softmax at output layer
- Initial learning rate: 0.01
- Batch size: 1
- Number of epochs: 100

Using this baseline configuration, the following results have been obtained:

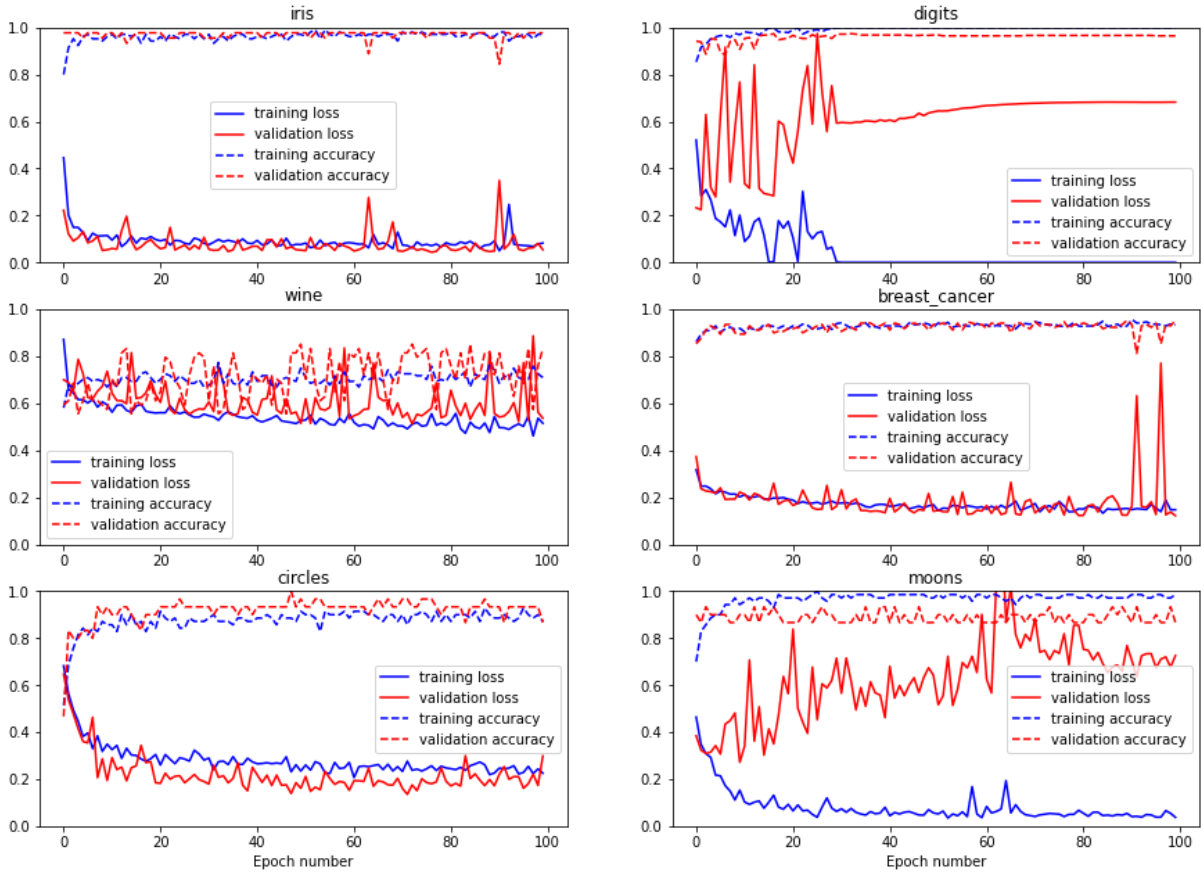
i) Stochastic gradient descent optimizer:



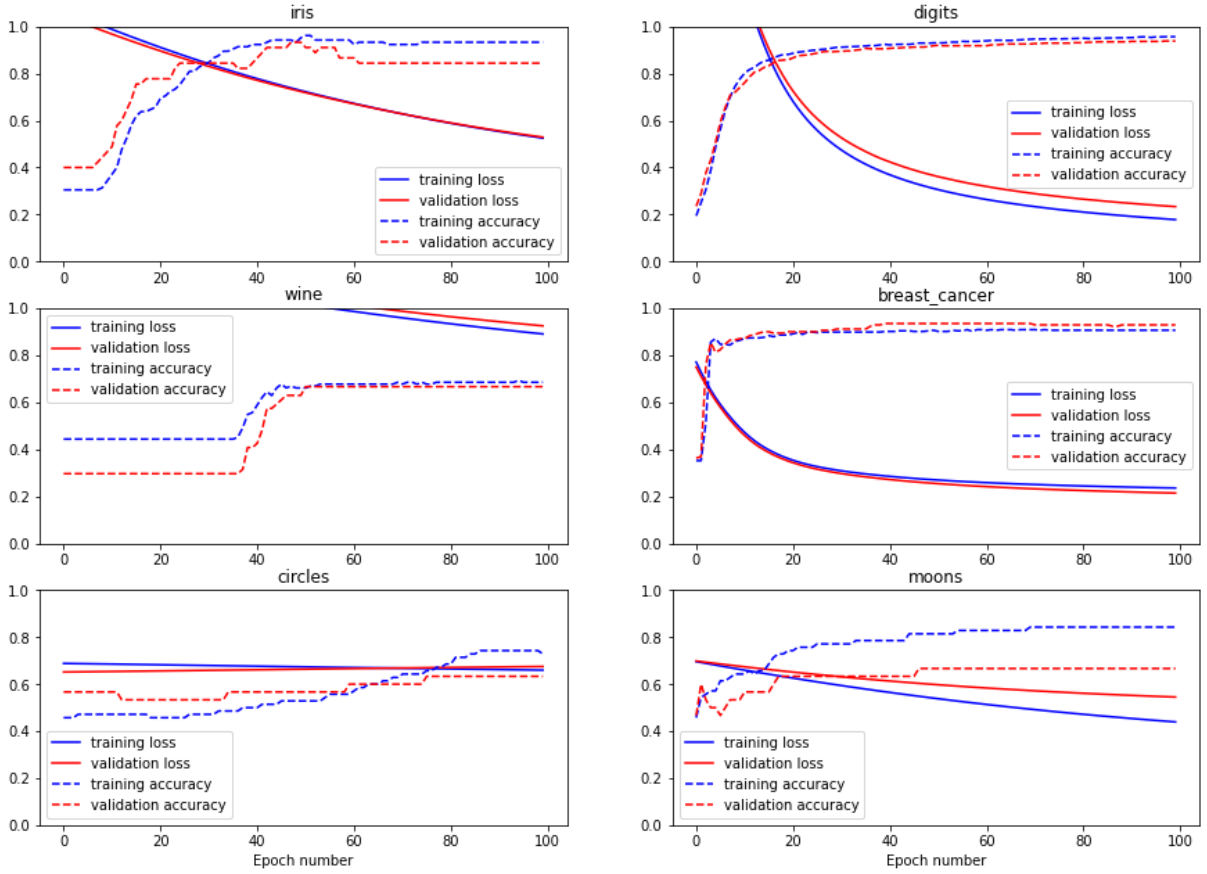
ii) Adam optimizer:



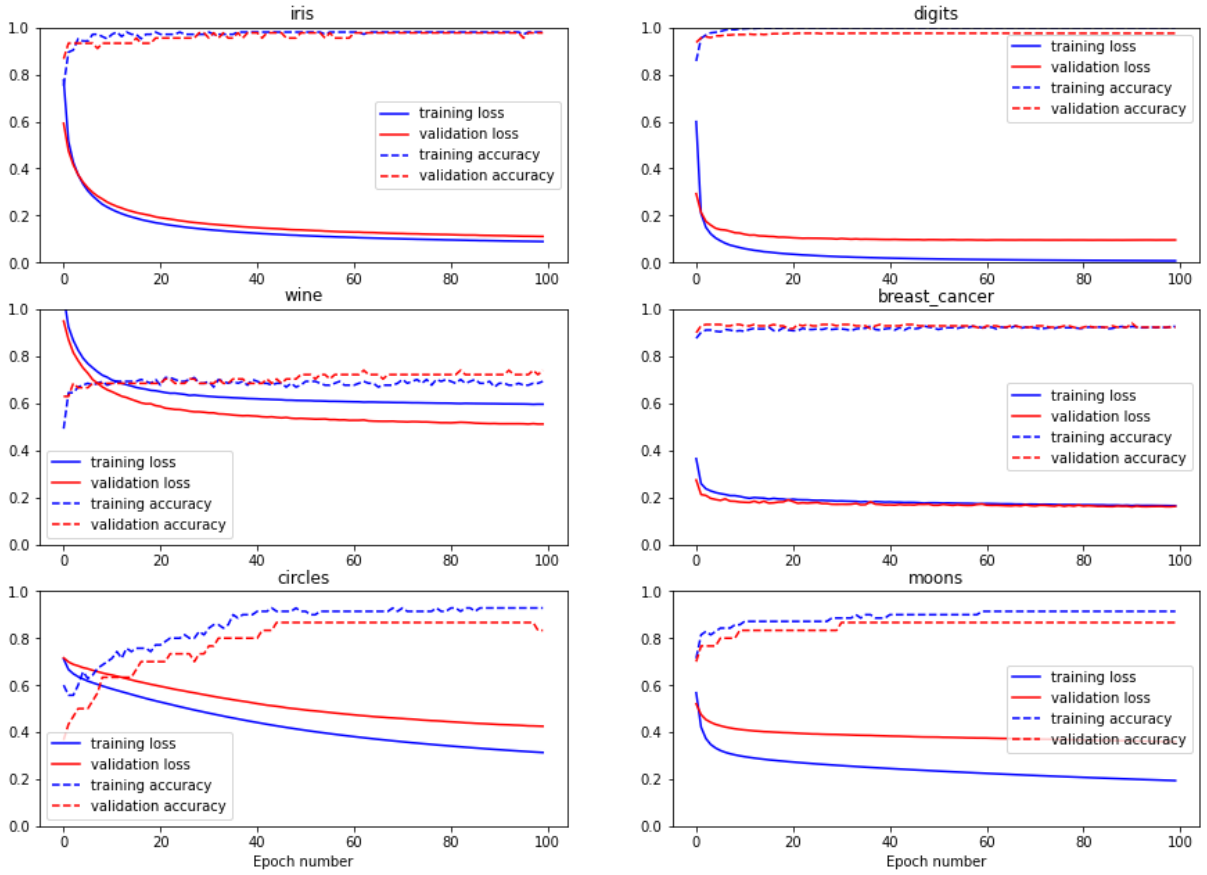
iii) Adam with Nesterov's momentum optimizer:



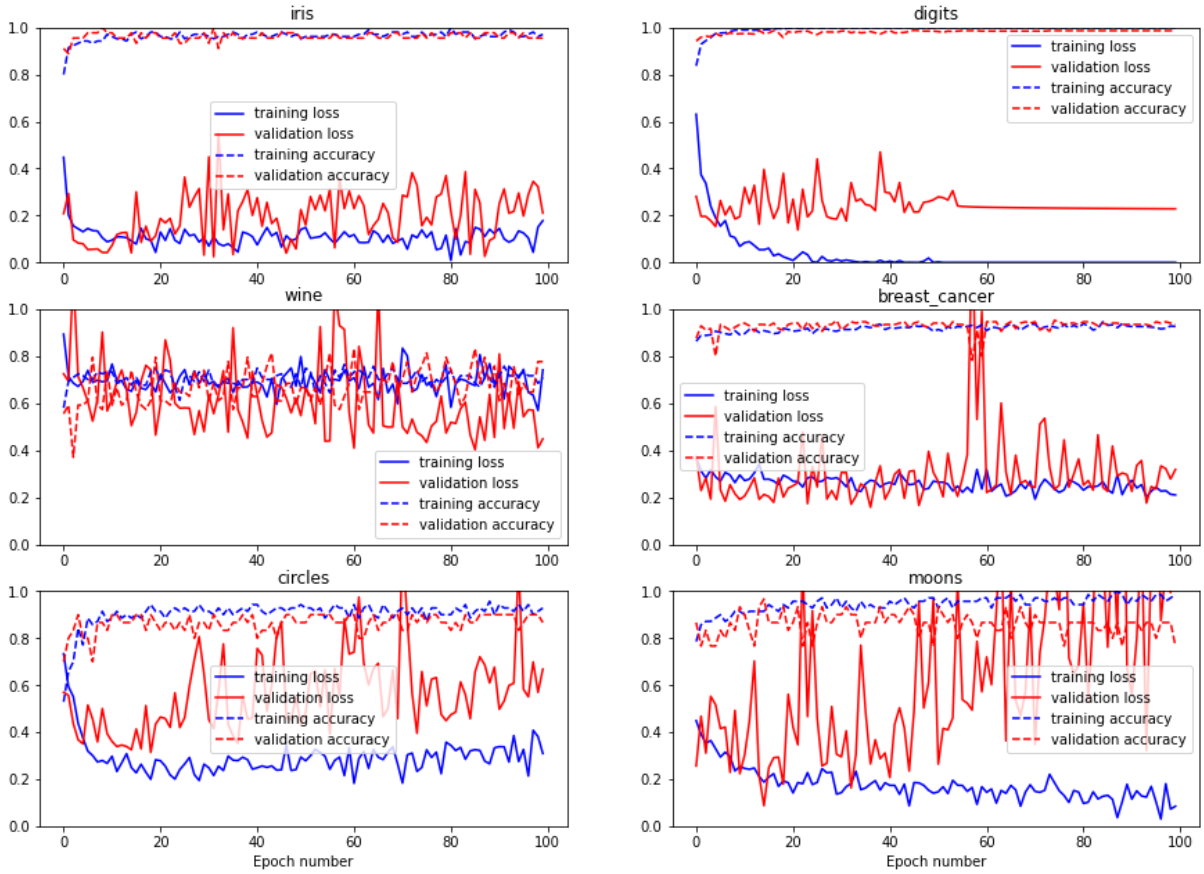
iv) AdaDelta optimizer:



v) AdaGrad optimizer:



vi) RMSProp optimizer:



Conclusions (see also the summary of results on the next page):

- Training on the “iris”, “digits” and “breast_cancer“ 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 “wine” data set yields the lowest test accuracies for this network configuration.
- The Adam and Adam with Nesterov’s momentum optimizers mostly lead to more fluctuations of the training and validation loss and accuracy than the stochastic gradient descent optimizer.
- AdaDelta is on nearly all data sets outperformed by the other optimizers.
- AdaGrad leads to a very smooth convergence of loss and accuracy.
- The RMSProp optimizer leads to very strong fluctuations especially of the loss.
- Thus, stochastic gradient, Adam (with or without Nesterov’s momentum), AdaGrad and RMSProp show only small differences in the final accuracies, with AdaDelta lacking behind. AdaGrad, however, shows the smoothest convergence.
- Note that these conclusions apply for these data sets and the investigated model configuration; in other settings, different conclusions may be obtained.

Remarks:

- Occasionally the validation accuracy is larger than the training accuracy (or the validation loss smaller than the training loss). It remains unclear whether these are statistical effects.
- The increasing validation loss of some optimizers on some data sets (e.g. Adam on “digits”) appears unreasonable and is not yet understood.

Summary of results:

Optimizer	train loss	train acc.	val. loss	val. acc.
iris dataset (4-dim. inputs, 3 classes; 105 training / 45 test samples)				
Stochastic gradient descent	0.065564	0.971429	0.119807	0.933333
Adam	0.082674	0.952381	0.005784	1.000000
Adam with Nesterov's momentum	0.082531	0.971429	0.054153	0.977778
AdaDelta	0.525326	0.933333	0.529071	0.844444
AdaGrad	0.088938	0.980952	0.110767	0.977778
RMSProp	0.177932	0.971429	0.210279	0.955556
digits dataset (4-dim. inputs, 10 classes; 1257 training / 540 test samples)				
Stochastic gradient descent	0.000885	1.000000	0.040355	0.985185
Adam	0.017388	0.996022	1.109858	0.951852
Adam with Nesterov's momentum	0.000000	1.000000	0.682727	0.964815
AdaDelta	0.178240	0.957041	0.233453	0.938889
AdaGrad	0.006989	1.000000	0.095668	0.975926
RMSProp	0.000000	1.000000	0.228302	0.987037
wine dataset (13-dim. inputs, 3 classes; 124 training / 54 test samples)				
Stochastic gradient descent	0.538073	0.717742	0.634721	0.666667
Adam	0.538379	0.750000	0.381470	0.796296
Adam with Nesterov's momentum	0.515205	0.709677	0.536481	0.833333
AdaDelta	0.891200	0.685484	0.925701	0.666667
AdaGrad	0.596262	0.693548	0.511642	0.740741
RMSProp	0.742256	0.709677	0.449061	0.777778
breast_cancer dataset (30-dim. inputs, 2 classes; 398 training / 171 test samples)				
Stochastic gradient descent	0.175433	0.919598	0.187958	0.918129
Adam	0.142339	0.944724	0.152873	0.918129
Adam with Nesterov's momentum	0.146815	0.934673	0.121535	0.953216
AdaDelta	0.235106	0.907035	0.213702	0.929825
AdaGrad	0.164141	0.927136	0.161262	0.923977
RMSProp	0.210003	0.927136	0.318333	0.935673
circles dataset (2-dim. inputs, 2 classes; 70 training / 30 test samples)				
Stochastic gradient descent	0.200863	0.928571	0.281996	0.866667
Adam	0.241319	0.900000	0.364981	0.90000
Adam with Nesterov's momentum	0.224845	0.871429	0.298469	0.866667
AdaDelta	0.659728	0.728571	0.675033	0.633333
AdaGrad	0.313102	0.928571	0.424899	0.833333
RMSProp	0.309003	0.928571	0.668561	0.866667
moon dataset (2-dim. inputs, 2 classes; 70 training / 30 test samples)				
Stochastic gradient descent	0.135250	0.942857	0.416759	0.833333
Adam	0.008084	1.000000	0.882425	0.933333
Adam with Nesterov's momentum	0.037010	0.985714	0.725868	0.866667
AdaDelta	0.439106	0.842857	0.545018	0.666667
AdaGrad	0.193511	0.914286	0.359127	0.866667
RMSProp	0.084378	0.985714	1.580781	0.766667

b) Using the most successful optimizer from part a), 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. To this end, modify the python script accordingly and systematically test the network performance. Provide the final training and validation loss and accuracy and provide the loss and accuracy curves as function of the number of epochs. You may also test further parameter settings. What are your conclusions regarding the network topology?

Solution:

A model with two hidden layers can be defined as follows:

```
input_layer = Input(shape=(num_inputs,), name='input')
hidden_1 = Dense(units=num_hidden, activation="relu", name="hidden_1")(
input_layer)
hidden_2 = Dense(units=num_hidden, activation="relu", name="hidden_2")(
hidden_1)
out = Dense(units=num_outputs, activation="softmax", name="output")(hid
den_2)
```

Using the AdaGrad optimizer, the following results have been obtained:
(Results for one hidden layer with 100 neurons taken from part a)

# hidden neurons	train loss	train acc.	val. loss	val. acc.
iris dataset (4-dim. inputs, 3 classes; 105 training / 45 test samples)				
(50,)	0.096501	0.990476	0.126886	0.955556
(100,)	0.088938	0.980952	0.110767	0.977778
(200,)	0.058057	0.980952	0.117328	0.933333
(50, 50)	0.055991	0.980952	0.091361	0.933333
(100, 100)	0.060059	0.980952	0.061451	0.955556
(200, 200)	0.063073	0.971429	0.114197	0.955556
digits dataset (4-dim. inputs, 10 classes; 1257 training / 540 test samples)				
(50,)	0.019322	0.998409	0.090017	0.970370
(100,)	0.006989	1.000000	0.095668	0.975926
(200,)	0.003642	1.000000	0.067985	0.981481
(50, 50)	0.003884	1.000000	0.078746	0.977778
(100, 100)	0.001265	1.000000	0.065976	0.985185
(200, 200)	0.000480	1.000000	0.081959	0.975926
wine dataset (13-dim. inputs, 3 classes; 124 training / 54 test samples)				
(50,)	0.541700	0.750000	0.644153	0.611111
(100,)	0.596262	0.693548	0.511642	0.740741
(200,)	0.569609	0.717742	0.542377	0.703704
(50, 50)	0.519029	0.750000	0.584597	0.703704
(100, 100)	0.559861	0.701613	0.463661	0.796296

(200, 200)	0.446653	0.814516	0.727554	0.685185
breast_cancer dataset (30-dim. inputs, 2 classes; 398 training / 171 test samples)				
(50,)	0.192433	0.914573	0.186317	0.929825
(100,)	0.164141	0.927136	0.161262	0.923977
(200,)	0.152407	0.937186	0.181820	0.912281
(50, 50)	0.122631	0.952261	0.233857	0.894737
(100, 100)	0.124095	0.947236	0.198816	0.918129
(200, 200)	0.124368	0.952261	0.177268	0.923977
circles dataset (2-dim. inputs, 2 classes; 70 training / 30 test samples)				
(50,)	0.395868	0.885714	0.395779	0.933333
(100,)	0.313102	0.928571	0.424899	0.833333
(200,)	0.298823	0.885714	0.279061	0.933333
(50, 50)	0.240709	0.900000	0.198268	0.933333
(100, 100)	0.171376	0.928571	0.309999	0.866667
(200, 200)	0.059897	0.985714	0.591067	0.833333
moon dataset (2-dim. inputs, 2 classes; 70 training / 30 test samples)				
(50,)	0.275595	0.871429	0.334582	0.800000
(100,)	0.193511	0.914286	0.359127	0.866667
(200,)	0.229606	0.928571	0.241809	0.933333
(50, 50)	0.135041	0.942857	0.178845	0.933333
(100, 100)	0.102645	0.971429	0.137196	0.966667
(200, 200)	0.102838	0.957143	0.128818	0.933333

Conclusion:

- Results are quite similar for the various network topologies. Whereas a modified network topology may improve results on one dataset, results on a different dataset can be lower. Therefore, the baseline topology of a single hidden layer with 100 neurons seems to be sufficient (again note that this refers to the investigated data sets and investigated parameters).