Università della Svizzera italiana

Facoltà di scienze informatiche

Machine Learning **2017**

Student: Timon Willi

Solution for Assignment 1

Due date: 20 October 2017, 21:00

1. The Perceptron *(25 Points)*

1.1.

First some general definitions of the vectors and matrices that will be used for the perceptron. We describe a full batch gradient descent for the forward pass:

- Weights $\mathbf{W} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_k \\ b \end{pmatrix} \in \mathbb{R}^{k+1}$, where $k =$ number of weights, w_i with $i \in \{1, ..., k\}$ is a single weight and b is the bias.
- Input Data $\mathbf{X} = \begin{pmatrix} \mathbf{x_1^T} \\ \mathbf{x_2^T} \\ \vdots \\ \mathbf{x_n^T} \end{pmatrix} \in \mathbb{R}^n$, where $n =$ size of data set and $\mathbf{x}_i$ with $i \in \{1, ..., n\}$ is a single data point.
- Single data point $\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \\ 1 \end{pmatrix} \in \mathbb{R}^{m+1}$, where $m =$ number of attributes and x_i with $i \in \{1, ..., m\}$ is a single attribute. $x_{m+1} = 1$ is used to multiply with the bias in $\mathbf{W}$.

- Target Vector $\mathbf{T} = \begin{pmatrix} t_1 \\ t_2 \\ \vdots \\ t_n \end{pmatrix} \in \mathbb{R}^n$, where $n =$ size of data set and t_i with $i \in \{1, ..., n\}$ is the target of $\mathbf{x}_i \in \mathbf{X}$.

The output of our perceptron is defined as $\mathbf{y} \in \mathbb{R}^n = \mathbf{XW} = \begin{pmatrix} \mathbf{x_1^T W} \\ \mathbf{x_2^T W} \\ \vdots \\ \mathbf{x_n^T W} \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$.

The MSE can be calculated as follows: $\text{MSE} \in \mathbb{R}^+ = \frac{\|(\mathbf{T} - \mathbf{y})\|^2}{2n} = \frac{(\mathbf{T} - \mathbf{y})^T(\mathbf{T} - \mathbf{y})}{2n}$. We define MSE_i as the contribution of x_i to the MSE, which is $\frac{(t_i - y_i)^2}{2}$.

The derivation of the error with respect to the weights goes as follows:

The gradient is defined as $\nabla_{\mathbf{W}} MSE_i = \begin{pmatrix} \frac{\partial MSE_i}{\partial w_1} \\ \frac{\partial MSE_i}{\partial w_2} \\ \vdots \\ \frac{\partial MSE_i}{\partial w_k} \end{pmatrix}$. We can calculate $\frac{\partial MSE_i}{\partial w_j}$, where $j \in \{1, ..., k\}$, by applying the product rule of derivation.

This results in:

$\frac{\partial MSE_i}{\partial w_i} = \frac{\partial MSE_i}{\partial q_i} * \frac{\partial q_i}{\partial y_i} * \frac{\partial y_i}{\partial w_j}$, where $q_i = (t_i - y_i)$ for $i \in \{1, ..., n\}$. Evaluating each element of this equation, we receive the following equations:

- $\frac{\partial MSE_i}{\partial q_i} = q_i$
- $\frac{\partial q_i}{\partial y_i} = -1$
- $\frac{\partial y_i}{\partial w_j} = \mathbf{x}_{i,j}$

Thus $\frac{\partial MSE_i}{\partial w_i} = q_i * -1 * \mathbf{x}_{i,j}$. To achieve a learning effect, we need to subtract this from the original weight w_j. For the whole weight vector this results in $\Delta w_i = \alpha * \mathbf{x}_i * (y_i - t_i)$, where α is the learning rate. For each $\mathbf{x}_i$ the gradients are being calculated, summed and then subtracted from $\mathbf{W}$. In our concrete setting, for one step, we get $x_1 = (1, 8, 1)^T$ and $t_i = 1$. The weights are initialized as $\mathbf{W} = (0.8, -0.5, 2)^T$ and the learning rate is 0.02. We calculate $y_i = x_1^T * \mathbf{W} = -1.2$ by applying the dot product. The MSE is $\frac{(1 - -1.2)^2}{2} = 2.42$ and $\Delta\mathbf{W} = 0.02 * (-1.2 - 1) * (1, 8, 1)^T = (-0.044, -0.352, -0.044)^T$. We then subtract $\mathbf{W} - \Delta\mathbf{W} = (0.8, -0.5, 2)^T - (-0.044, -0.352, -0.044)^T = (0.844, -0.148, 2.044)^T = \mathbf{W}'$.

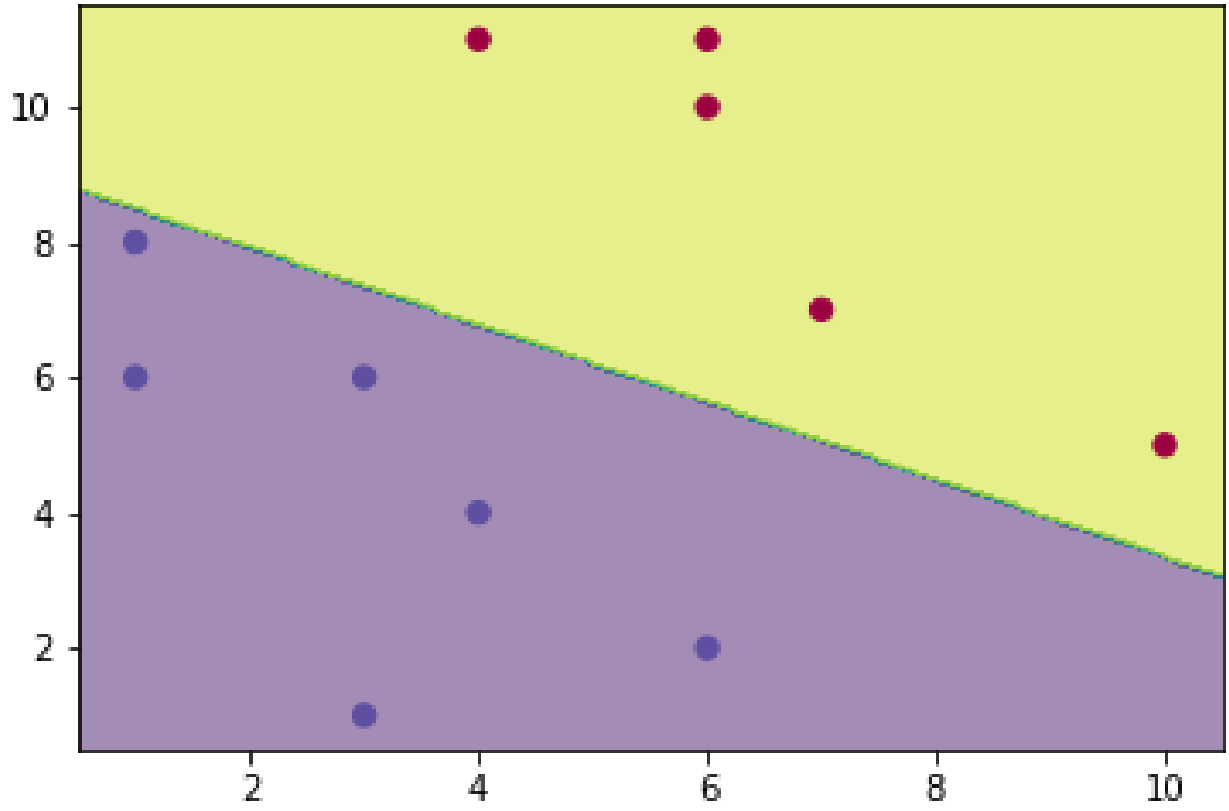


Figure 1. Boundary of Perceptron

1.2.

$\vartheta_{t+1} = \vartheta - \alpha * \nabla\vartheta J(\vartheta)$, where theta are the parameters of the network and J is the cost function. Since we are looking for the minimum of the mean squared error, by subtracting the gradient, we are approaching a local minimum of the MSE function, which leads to a learning effect. Gradient descent is also a good learning algorithm because it is relatively easy to compute if the activation functions are chosen wisely. In combination with Backpropagation, it allows to reuse many of the already computed variables. Gradient descent however is unstable if for example the learning rate is not set correctly. If the learning rate is too big, gradient descent might shoot over the minimum and go in the wrong direction. In some areas of the function gradient descent might also exhibit a zig-zaging behaviour and not choose the most direct path to the minimum, thus making the learning process slower.

2. A Neural Network *(50 Points)*

2.1.

2.2.

We will use the definitions from the first exercise with some additions. We declare $\mathbf{W}^L$ as the ith weight matrix at layer L, where the columns are the different weight vectors $\mathbf{W}^L = (\mathbf{W}_1^L, \mathbf{W}_2^L, \cdots, \mathbf{W}_p^L)$ where p is the size of the layer. We also introduce two non-linear activation function, sigmoid and tanh, which will be represented with $\sigma(x)$ and $tanh(x)$:$\mathbb{R}^n \to \mathbb{R}^n$ respectively. The forward pass goes as follows:

- We define $\mathbf{a}^1 = \mathbf{X}$ as the input data.
- $\mathbf{z}^1 = \mathbf{a}^1 * \mathbf{W}^1$, where * is the dot product and $\mathbf{a}^1 \in \mathbb{R}^{n \times m}, \mathbf{W}^1 \in \mathbb{R}^{m \times 20}, \mathbf{z}^1 \in \mathbb{R}^{n \times 20}$.
- $\mathbf{a}^2 = tanh(\mathbf{z}^1)$
- $\mathbf{z}^2 = \mathbf{a}^2 * \mathbf{W}^2$, where $\mathbf{a}^2 \in \mathbb{R}^{n \times 20}, \mathbf{W}^2 \in \mathbb{R}^{20 \times 15}, \mathbf{z}^2 \in \mathbb{R}^{n \times 15}$.
- $\mathbf{a}^3 = tanh(\mathbf{z}^2)$
- $\mathbf{z}^3 = \mathbf{a}^3 * \mathbf{W}^3$, where $\mathbf{a}^3 \in \mathbb{R}^{n \times 15}, \mathbf{W}^3 \in \mathbb{R}^{15 \times 1}, \mathbf{z}^3 \in \mathbb{R}^{n \times 1}$
- $\mathbf{y} = \sigma(\mathbf{z}^3)$

The MSE can be calculated as follows: MSE $\in \mathbb{R}^+ = \dfrac{\|(\mathbf{T} - \mathbf{y})\|^2}{2n} = \dfrac{(\mathbf{T} - \mathbf{y})^T(\mathbf{T} - \mathbf{y})}{2n}$. We define MSE_i as the contribution of a_i^3 to the MSE, which is $\dfrac{(t_i - y_i)^2}{2}$.

The derivation of the error with respect to the weights goes as follows:

The gradient is defined as $\nabla_{\mathbf{W}^L} MSE_i = \begin{pmatrix} \frac{\partial MSE_i}{\partial \mathbf{W}_{1,1}^L} & \frac{\partial MSE_i}{\partial \mathbf{W}_{2,1}^L} & \cdots & \frac{\partial MSE_i}{\partial \mathbf{W}_{p,1}^L} \\ \frac{\partial MSE_i}{\partial \mathbf{W}_{1,2}^L} & \frac{\partial MSE_i}{\partial \mathbf{W}_{2,2}^L} & \cdots & \frac{\partial MSE_i}{\partial \mathbf{W}_{p,2}^L} \\ \vdots & \vdots & \cdots & \vdots \\ \frac{\partial MSE_i}{\partial \mathbf{W}_{1,k}^L} & \frac{\partial MSE_i}{\partial \mathbf{W}_{2,k}^L} & \cdots & \frac{\partial MSE_i}{\partial \mathbf{W}_{p,k}^L} \end{pmatrix}$. We can calculate $\dfrac{\partial MSE_i}{\partial \mathbf{W}^L}$, where $j \in \{1, ..., k\}, h \in \{1, ..., p\}$, by applying the product rule of derivation.
For the first layer of our network, this results in:

$\dfrac{\partial MSE_i}{\partial \mathbf{W}^1} = \dfrac{\partial MSE_i}{\partial q_i} * \dfrac{\partial q_i}{\partial y_i} * \dfrac{\partial y_i}{\partial z_i^3} * \dfrac{\partial z_i^3}{\partial a_i^3} * \dfrac{\partial a_i^3}{\partial z_i^2} * \dfrac{\partial z_i^2}{\partial a_i^2} * \dfrac{\partial a_i^2}{\partial z_i^1} * \dfrac{\partial z_i^1}{\partial \mathbf{W}^1}$, where $q_i = (t_i - q_i)$ for $i \in \{1, ..., n\}$.
Evaluating each element of this equation, we receive the following equations:

- $\dfrac{\partial MSE_i}{\partial q_i} = q_i$

- $\frac{\partial q_i}{\partial y_i} = -1$
- $\frac{\partial y_i}{\partial z_i^3} = \sigma'(z_i^3)$. We define $\delta^3 = \frac{\partial MSE_i}{\partial q_i} * \frac{\partial q_i}{\partial y_i} * \frac{\partial y_i}{\partial z_i^3}$, $\delta^3 \in \mathbb{R}^{1\times 15}$.
- $\frac{\partial z_i^3}{\partial a_i^3} = \mathbf{W}^3$
- $\frac{\partial a_i^3}{\partial z_i^2} = \sigma'(z_i^2)$. We define $\delta^2 = \frac{\partial MSE_i}{\partial q_i} * \frac{\partial q_i}{\partial y_i} * \frac{\partial y_i}{\partial z_i^3} * \frac{\partial z_i^3}{\partial a_i^3} * \frac{\partial a_i^3}{\partial z_i^2} = (\delta^3 * (\mathbf{W}^3)^T) \circ \sigma'(z_i^2)$, $\delta^2 \in \mathbb{R}^{1\times 20}$, where $\circ$ is the element-wise multiplication.
- $\frac{\partial z_i^2}{\partial a_i^2} = \mathbf{W}^2$
- $\frac{\partial a_i^2}{\partial z_i^1} = \sigma'(z_i^1)$. We define $\delta^1 = (\delta^2 * (\mathbf{W}^2)^T) \circ \sigma'(z_i^1)$, $\delta^1 \in \mathbb{R}^{1\times 3}$
- $\frac{\partial z_i^1}{\partial \mathbf{W}^1} = a_i^1 \rightarrow \frac{\partial MSE_i}{\partial \mathbf{W}^1} = (a_i^1)^T * \delta^1$. As one can see, it is possible to generalize by writing $\frac{\partial MSE_i}{\partial \mathbf{W}^l} = (a_i^l)^T * \delta^l$. Thus we can write:
- $\frac{\partial MSE_i}{\partial \mathbf{W}^1} = (a_i^1)^T * \delta^1$
- $\frac{\partial MSE_i}{\partial \mathbf{W}^2} = (a_i^2)^T * \delta^2$
- $\frac{\partial MSE_i}{\partial \mathbf{W}^3} = (a_i^3)^T * \delta^3$

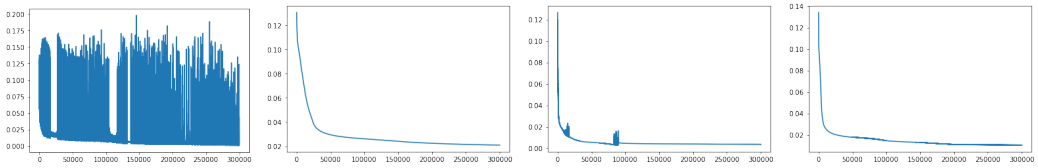
2.3.

2.4.

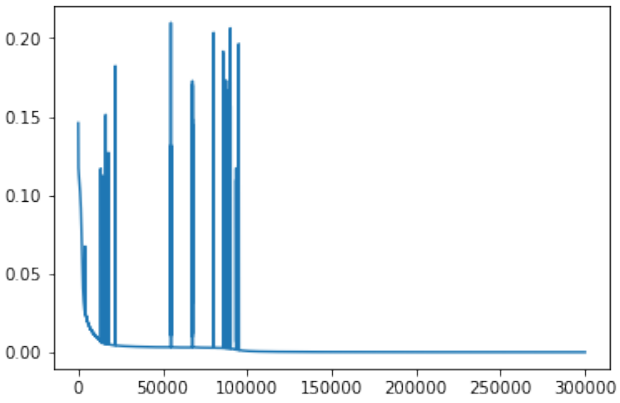
I chose a 80/20 split such that I would get data from both classes. I also chose it based on experience since often 5-fold cross-validation works well. There is a danger of overfitting since the classes are not equally distributed. There is also a skew in the test-data, since it only consists of one class. This makes it less robust to noise. It is recommended to first shuffle the data set. The learning rate of 0.1 achieved the lowest cost of 4.39473081029e-05.

2.5.

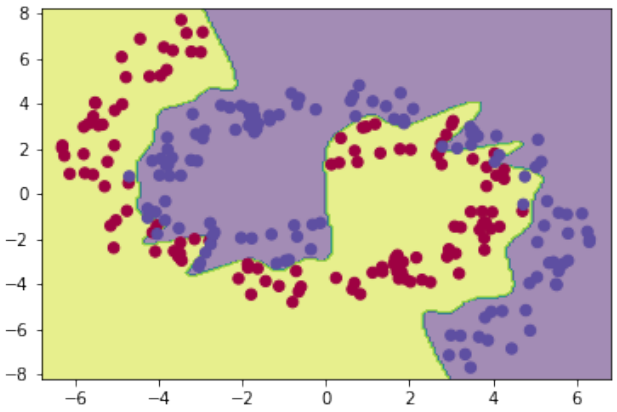
As one can see from the plotted boundary, the network is not able to properly fit to spiral-like data. If there was more data to come with more noise or a higher twist, the network would fail to predict properly. It is not robust to a lot of variance in data, if one assumes, that the data exhibits a spiral form. There are many ways which could possibly improve the performance of a neural network. Adding more layers would provide the network with more flexibility. However, there is also the danger that it then might overfit. Maybe other activation functions will increase performance.



(a) Learning Rate = 0.5 *(b)* Learning Rate = 0.01 *(c)* Learning Rate = 0.02 *(d)* Learning Rate = 0.05



(e) Learning Rate = 0.1



(f) Learning Rate = 0.1

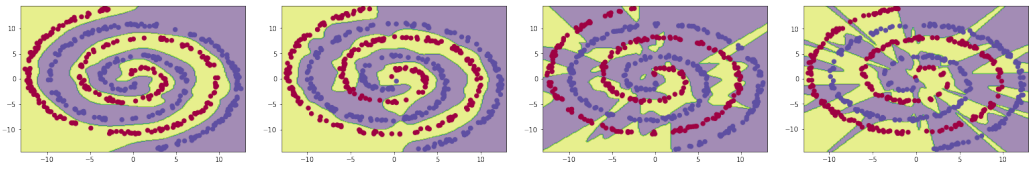
Figure 2. Cost convergence and boundary for different learning rates.

3. Further Improvements *(25 Points)*

For one, I try to use a different activation function which might provide more non-linearity than the tanh. I will try the recently published swish activation function, which looks very similar to ReLU[1]. I will also apply the stochastic gradient descent. To be more exact, I will be using mini-batch stochasitc gradient descent, with batch-size of 10. Due to the stochasticity, it is more robust and can find the shorter way to the minimum in a smaller amount of time, thus reducing the training time. Additionally I will use softmax instead of sigmoid in the final layer. Although one can be transformed into the other in the two-class classification, I am interested in the one-hot encoding. Also maybe there might be a performance difference, since it is not unreasonable to assume that in practice, the two

[1] https://arxiv.org/pdf/1710.05941.pdf

methods might perform differently.



(a) *(b)* *(c)* *(d)*

Figure 3. Cost convergence and boundary for different learning rates.

In (a) we can see the boundary of the softmax combined with the swish activation and the stochastic learning. It achieved an accuracy of 1.0 on the test set with 1000 epochs and a batch-size of 10. In (b) the boundary of the model with sigmoid output with swish can be seen. It was trained in 30'000 epochs. In (c) we use tanh instead of swish for the softmax and stochastic. It is also trained in 1000 epochs. In (d) we see the boundaries plotted with the model from exercise 2 trained in 100'000 epochs.

We can see that the biggest impact was made by the change of activation function. It trains a lot faster with swish. However, the best performance was achieved with the combination of softmax, stochastic and swish together.