Introduction to Data Analysis

Teilnehmer:

Name
2 Teilnehmende des Andreas-Gymnasium
4 Teilnehmende des Heinrich-Hertz-Gymnasium
3 Teilnehmende des Herder-Gymnasium
1 Teilnehmer des Käthe-Kollwitz-Gymnasium

mit tatkräftiger Unterstützung durch:

Lydia Gehrke	Humboldt-Universität zu Berlin

Gruppenleiter:

Benjamin Couéraud	DLR, Berlin

1. Introduction

Data can be found everywhere in everyday life. It is important to evaluate existing ones in order to make predictions about the future, for example in case of illness or real estate prices.

2. Linear Regression

The easiest way to evaluate data is by using linear regression. Here, existing inputs are approximated linearly and then a relationship to an output is established.

2.1. Univariate Case

First, consider the case that there is only one input variable x available for the data, but also one output variable y. Overall, we have a data set $\{(x_i, y_i)\}_{1\leq i\leq N}$ consisting of N pairs of data.

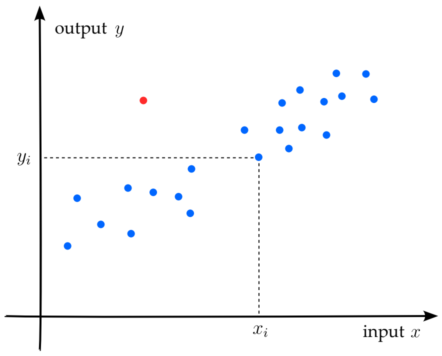


Abbildung 1: Illustration of some dataset

Now, we would like to find a linear function that approximates the values in the best way. So, we try to find parameter θ_0 and θ_1 in the linear function

$$f_\theta(x) = \theta_0 + \theta_1 x$$

The optimal function would be the one where the difference between the actual and the approximated values is the smallest. So for every x_i in our data set, we try to minimize the error $|f(x_i) - y_i|$. Therefore, we introduce the *cost function* $L : \mathbb{R}^2 \to \mathbb{R}$, which is defined as follows:

$$L(\theta_0, \theta_1) = \frac{1}{2N}\sum_{i=1}^{N}[f_\theta(x_i) - y_i]^2 = \frac{1}{2N}\sum_{i=1}^{N}[(\theta_0 + \theta_1 x_i) - y_i]^2$$

This cost function describes the average of the squared error $(f(x_i) - y_i)^2$, divided by 2. We square the difference because this way, we have no negative differences while still only using completely derivable functions. Now, we try to find the parameters θ_0^* and θ_1^* for which the value of the cost function is minimized:

$$(\theta_0^*, \theta_1^*) = \underset{(\theta_0,\theta_1)\in\mathbb{R}^2}{\text{arg min}}\ L(\theta_0, \theta_1)$$

This means that the problem has now become a minimisation problem. To find the candidates for the local minima of L, you have to search for values θ_0, θ_1 where the derivative of L at this point is equal to zero. But because we have two input variables in L, we have to find the solutions for the following system of equations:

$$\frac{\partial L}{\partial \theta_0}(\theta_0^*, \theta_1^*) = 0$$
$$\frac{\partial L}{\partial \theta_1}(\theta_0^*, \theta_1^*) = 0$$

Here, we use partial derivatives of the cost function. The second criterion for these candidates is the *Hessian matrix* $H(\theta_0^*, \theta_1^*)$:

$$H(\theta_0^*, \theta_1^*) = \begin{bmatrix} \frac{\partial^2 L}{\partial \theta_0^2}(\theta_0^*, \theta_1^*) & \frac{\partial^2 L}{\partial \theta_1 \partial \theta_0}(\theta_0^*, \theta_1^*) \\ \frac{\partial^2 L}{\partial \theta_0 \partial \theta_1}(\theta_0^*, \theta_1^*) & \frac{\partial^2 L}{\partial \theta_1^2}(\theta_0^*, \theta_1^*) \end{bmatrix}$$

In this case, (θ_0^*, θ_1^*) is a local minimum of L if and only if $H(\theta_0^*, \theta_1^*)$ is positive definite. This means that, for every $u \in \mathbb{R}^N \setminus \{\vec{0}\}$, the inequality $u^T H(\theta_0^*, \theta_1^*) u > 0$ holds.

Now, with this knowledge, we compute the minimum of the cost function L as follows. First we bring the cost function into a form that allows us to use less notation.

$$\begin{aligned} L(\theta_0, \theta_1) &= \frac{1}{2N} \sum_{i=1}^{N} [(\theta_0 + \theta_1 x_i) - y_i]^2 \\ &= \frac{1}{2N} \sum_{i=1}^{N} \theta_0^2 + \theta_1^2 x_i^2 + 2\theta_0\theta_1 x_i + y_i^2 - 2\theta_0 y_i - 2\theta_1 x_i y_i \\ &= \frac{1}{2}\theta_0^2 + \frac{1}{2}\theta_1^2 \overline{x^2} + \theta_0\theta_1 \overline{x} + \frac{1}{2}\overline{y^2} - \theta_0 \overline{y} - \theta_1 \overline{xy} \end{aligned}$$

With $\overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$ and $\overline{xy} = \frac{1}{N}\sum_{i=1}^{N} x_i \cdot y_i$. Now the first derivative can be computed. The derivative with respect to θ_0,

$$\frac{\partial L}{\partial \theta_0}(\theta_0, \theta_1) = \theta_0 + \theta_1 \overline{x} - \overline{y}$$

and with respect to θ_0,

$$\frac{\partial L}{\partial \theta_1}(\theta_0, \theta_1) = \theta_1 \overline{x^2} + \theta_0 \overline{x} - \overline{xy}$$

can now be used to calculate critical points, that may be the global minimum. This can be accomplished by taking advantage on the fact that for the points (θ_0^*, θ_1^*) to be the minimum, the partial derivatives need to be zero. It follows that

$$\theta_0 + \theta_1 \overline{x} - \overline{y} = 0$$
$$\theta_1 \overline{x^2} + \theta_0 \overline{x} - \overline{xy} = 0$$

which yields:

$$\theta_0 = \overline{y} - \theta_1 \overline{x}$$

$$\theta_1 = \frac{\overline{xy} - \overline{x} \cdot \overline{y}}{\overline{x^2} - \overline{x}^2} = \frac{\text{Cov}(x, y)}{\text{Var}(x)}$$

Here we can make the observation, that $\text{Var}(x) \neq 0 \Rightarrow \overline{x^2} - \overline{x}^2 > 0$. Now that the critical points have been obtained, we only need to show that the Hessian Matrix $H(\theta_0^*, \theta_1^*)$ is positive definite, meaning that for all $, u \in \mathbb{R}^2 \setminus \{0\}$ holds, that $u^T H(\theta_0^*, \theta_1^*)u > 0$. This can be shown, by computing the given formula.

$$\begin{aligned}
u^T H(\theta_0^*, \theta_1^*)u &= \begin{bmatrix} u_1 & u_2 \end{bmatrix} \begin{bmatrix} 1 & \overline{x} \\ \overline{x} & \overline{x^2} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \\
&= u_1(u_1 + u_2\overline{x}) + u_2(u_1\overline{x} + u_2\overline{x^2}) \\
&= u_1^2 + 2u_1u_2\overline{x} + u_2^2\overline{x^2} \\
&> u_1^2 + 2u_1u_2\overline{x} + u_2^2\overline{x}^2 \\
&= (u_1 + u_2\overline{x})^2 \geq 0
\end{aligned}$$

This computation hold, because of the observation that we can suppose $\overline{x^2} - \overline{x}^2 > 0$ for most datasets represented by x.

2.2. Multivariate Case

But what if we have more than one input determining the result? For this, we can use multivariate regression. A function with n features would look somewhat like this:

$$f_\theta : \mathbb{R}^n \to \mathbb{R} \text{ with } f_\theta(x_1, ...x_n) = \theta_0 + \theta_1 \cdot x_1 + ... + \theta_n \cdot x_n$$

Like before, we need a cost function, which is still very similar to the univariate cost function:

$$L(\theta_0, ...\theta_n) = \frac{1}{2N} \sum_{i=1}^{M} (\theta_0 + \theta_1 \cdot x_1 + ... + \theta_n \cdot x_n - y)^2$$

To minimise this cost function, we have two options; normal equations or gradient descent, a numerical approach. First we will deal with the normal equations, for which we will use matrices as a means of representing the functions, where X is the matrix of features, θ is the vector of $\theta_0, ...\theta_n$ and y is the vector of the outputs. When we write the loss function like this, we get

$$L(\theta) = \frac{1}{N}(X\theta - y)(X\theta - y)^T$$

As before, we want to find the vector θ so that the derivative of L is 0. Assuming the matrix $X^T X$ is invertible, we calculated θ^* and found:

$$\theta^* = (X^T X)^{-1} X^T y$$

We say that the matrix A, of size $n \times n$, is *invertible* if there exists a matrix B, which has the same size as A, such that:

$$BA = AB = I_n,$$

where I_n denotes the *identity matrix* of size $n \times n$ (with ones only on the diagonal). In the case such a matrix B exists, we say that A is invertible and we denote the inverse by A^{-1}. Now let's take a look at an example: for a invertible matrix $M \in \mathbb{R}^{2\times 2}$, we can manually calculate the inverse $M^{-1} = \begin{pmatrix} e & f \\ g & h \end{pmatrix}$.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

From this, we get the following equations:

$$1 = ae + bf \tag{1}$$
$$0 = ce + df \tag{2}$$
$$0 = ag + bh \tag{3}$$
$$1 = cg + dh \tag{4}$$

That gives us:

$$\begin{aligned}
e &= \frac{1-bf}{a} && \text{rearrange (1)}\\
\Rightarrow f &= \frac{-c\cdot\frac{1-bf}{a}}{d} && \text{insert e in (2)}\\
\Rightarrow f &= \frac{-c}{ad-cb}\\
\Leftrightarrow e &= \frac{1-b\frac{-c}{ad+cb}}{a} = \frac{d}{ad-bc}
\end{aligned}$$

Next we have:

$$\begin{aligned}
g &= \frac{bh}{a} && \text{rearrange (3)}\\
\Rightarrow h &= \frac{1-c\frac{bh}{a}}{d} && \text{insert g in (4)}\\
\Rightarrow h &= \frac{a}{ad-cb}\\
\Leftrightarrow g &= \frac{-b}{ad-cb}
\end{aligned}$$

When we write this out, we get:

$$\frac{1}{ad-bc}\cdot\begin{pmatrix} d & -c \\ -b & a \end{pmatrix}$$

because M is invertible, the determinant $det(M) = ad - bc$ is not equal to zero. We can apply this result to recover the results obtained in the univariate case. Indeed, we have

$$X = \begin{pmatrix} 1 & x_1 \\ \vdots & \vdots \\ 1 & x_n \end{pmatrix},$$

so by using the definition of the multiplication of matrices:

$$X^T X = \begin{pmatrix} 1 & \overline{x} \\ \overline{x} & \overline{x^2} \end{pmatrix}$$

and so by applying the previous result we obtain:

$$(X^T X)^{-1} = \frac{1}{n}\frac{1}{\mathrm{Var}(x)}\begin{pmatrix} \overline{x^2} & -\overline{x} \\ \overline{x} & 1 \end{pmatrix}.$$

Finally we obtain the parameters of univariate regression again:

$$\theta^* = (X^T X)^{-1} X y = \frac{1}{n}\frac{1}{\mathrm{Var}(x)}\begin{pmatrix} \overline{x^2} & -\overline{x} \\ \overline{x} & 1 \end{pmatrix}\begin{pmatrix} n\overline{y} \\ n\overline{xy} \end{pmatrix} = \frac{1}{\mathrm{Var}(x)}\begin{pmatrix} \overline{x^2}\cdot\overline{y} - \overline{x}\cdot\overline{xy} \\ \overline{xy} - \overline{x}\cdot\overline{y} \end{pmatrix}.$$

2.3. Gradient descent

The presented method is impractical in the case of very big matrices because it takes a lot of time for the computer to invert the matrix $X^T X$ as the number of rows and columns in the matrix increases. A shorter, but still very precise algorithm for optimization problems is the gradient descent.

First, we define the *gradient* $\nabla F(a)$ of a function $F : \mathbb{R}^d \to \mathbb{R}$ at a point a as the column vector of the partial derivatives of the function in each component:

$$\nabla F(a) = \begin{pmatrix} \frac{\partial F}{\partial x_1}(a) \\ \vdots \\ \frac{\partial F}{\partial x_d}(a) \end{pmatrix}$$

The gradient consists of all the partial derivatives of the function F. Also, we define the *directional derivative* of the function $F : \mathbb{R}^d \to \mathbb{R}$ in the point $a \in \mathbb{R}^d$ and in the direction $h \in \mathbb{R}^d$ to be:

$$D_h F(a) = \lim_{\varepsilon \to 0} \frac{F(a + \varepsilon h) - F(a)}{\varepsilon}$$

For existent and continuous partial derivatives, the relation $D_h F(a) = \langle \nabla F(a), h \rangle$ holds (for the Euclidean scalar product on $\mathbb{R}^d$). The direction $h = -\nabla F(a)$ is always a direction of descent on the function because $D_{-\nabla F(a)} F(a) = \langle \nabla F(a), -\nabla F(a) \rangle = -\|\nabla F(a)\|^2 \leq 0$. This means that, due to the definition of the directional derivative, there exists a sufficiently small value ε for which $F(a - \varepsilon \nabla F(a)) < F(a)$. This means that the algorithm brings us closer to the minimum by going in the opposite direction of the gradient.

Now we can define the algorithm of the gradient descent. Here, you start at a point $\theta_0 \in \mathbb{R}^d$ and compute the sequence (θ_n) with the following recursion:

$$\theta_{n+1} = \theta_n - \alpha \nabla L(\theta_n)$$

In this definition, α is a parameter which controls how far we are going by following the gradient. Also, this algorithm is converging towards the local minimum of the function, if the value of α is chosen in the right way. To implement this algorithm, we need a certain condition for it to stop. This can be done by choosing a small threshold δ, and end the iteration in the case $|F(\theta_{n+1}) - F(\theta_n)| \leq \delta$. Also, you have to note that if the function has several local minima, the choice of the starting point x_0 is important because the algorithm then converges towards the nearest local minimum (this is why often in the implementation, several starting points are chosen for the gradient descent and in the end, one just takes the one that gave the smallest value for the cost function). This algorithm can be applied to our cost function $L(\theta)$ to find an approximation for the global minimum of the function. Thereby, the gradient of the multivariate cost function $L(\theta)$ can be rewritten as follows by applying the definition of the matrices X and the vector y :

$$\nabla L(\theta) = \frac{1}{N} X^T (X\theta - y)$$

Finally we note that, despite what its name suggests, linear regression can also be applied to nonlinear cases. To do so, we take an additional feature x, which is really $f(x_i)$ where x_i is an already existing feature, and f is another function, for example the quadratic one. Using this method, we can pretend a function is linear, and use the methods from linear regression on it, by going into a higher dimensional space. This lets us find parameters θ for virtually any function with linear regression, although the more features we add, the harder it becomes to compute a solution quickly.

3. Logistic regression

Sometimes, we want to decide/predict whether some data belongs to some class or not, for example when we want to know if an email is spam or not or if a picture shows a cat or not. There are also applications in medicine, when we want to know whether a patient has a certain illness or not. This is achieved by using logistic regression. How it works, and how we use it for classification (as it is, in fact, a regression technique), is explained in the following. To achieve this, we have to apply a function g that returns a (mostly) binary output. We again have an input matrix

$$X = (x_{i,j})_{1 \leq i,j \leq M+1,N}$$

of the features x_i, where $x_{1,j} = 1$. Also, we again have an output vector $y = (y_j)_{1 \leq j \leq N}$. The function we now want to fit is the function $f_\theta(x_1, \dots, x_M) = g(\theta_0 + \theta_1 x_1 + \cdots + \theta_M x_M)$ where g is the function we presented before. One possible function is the logistic function $g(z) = \frac{1}{1+e^{-z}}$ (thus the name *logistic* regression). It strictly returns values between 0 and 1, of which most are quite close to 0 (class 0) or 1 (class 1), which is fitting, since we wanted a function with (mostly) binary output. But, any sigmoid function and other function with (mostly) binary output would work for the function g.

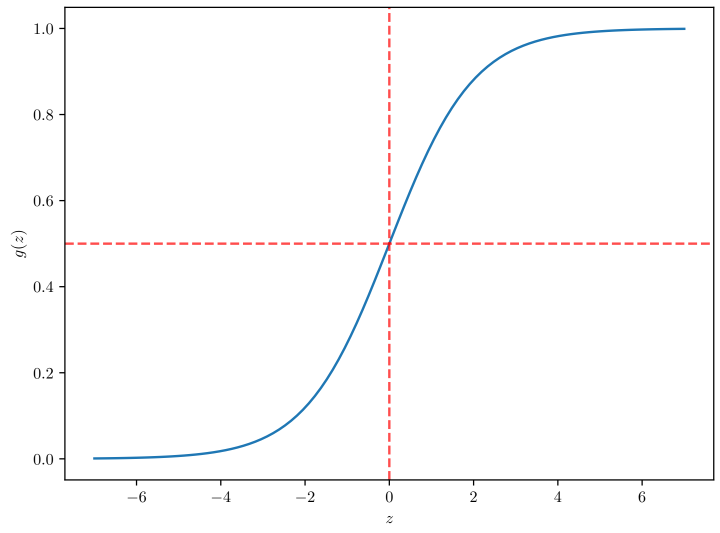


Abbildung 2: Graph of the logistic function

Once we have found those θ_i, then we can plot the decision boundary by plotting the linear function $\theta_0 + \theta_1 x_1 + \cdots + \theta_M x_M$ due to the nature of the linear function: we set the output variable y of the model to be equal to 1 if $f_\theta(x_1, \dots, x_M) = g(\theta_0 + \cdots + \theta_M x_M) \geq \frac{1}{2}$ (which is equivalent to $\theta^T x \geq 0$ after setting $x_0 = 1$), and $y = 0$ if $f_\theta(x_1, \dots, x_M) = g(\theta_0 + \cdots + \theta_M x_M < \frac{1}{2}$ (which is equivalent to $\theta^T x < 0$). Thus, we can assume as the decision boundary for the data to be the set of points that satisfy the equation

$$\theta_0 + \theta_1 x_1 + \cdots + \theta_M x_M = 0$$

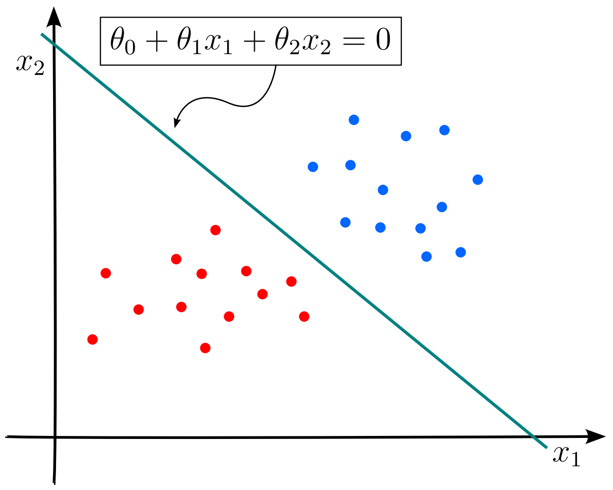


Abbildung 3: Illustration of the decision boundary, which separates the two classes

Similarly to linear regression, we now try to fit the θ_i to the data set. Similar to linear regression, we get a function to predict something, but this time, whether some data belongs to the first (0) or second (1) class. The closer the value is to 0, the more likely it is for the data to belong to the 0-class, and vice versa. To achieve the fit, we again have to minimize a cost function for the data set. We can express it as the following:

$$L(\theta_0, \dots, \theta_M) = \frac{1}{N} \sum_{j=1}^{N} \ell(f_\theta(x_{1,j}, \dots, x_{M,j}), y_j)$$

$\ell(f_\theta(x), y)$ is called the loss function for one sample of the data set that we sum up to get the total cost. It tells us how much difference there is between the real class and the prediction. This cost function is better for logistic regression as the one used for linear regression would have too many local minima in our case. We instead use this function, called 'log-loss function' which contains a logarithm:

$$\ell(f_\theta(x_1, \dots, x_M), y) = \begin{cases} -\log f_\theta(x_1, \dots, x_M) & \text{if } y = 1, \\ -\log(1 - f_\theta(x_1, \dots, x_M) & \text{if } y = 0 \end{cases}$$

(log, in this case, is the natural log, but any would work)

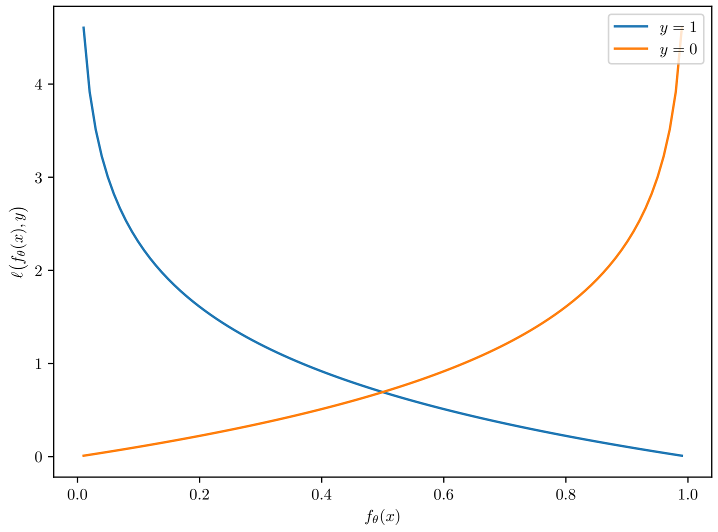


Abbildung 4: Graph of the log-loss function

This function does everything we need: in case that $y = 1$, but $\ell(f_\theta(x), y) \to 0$, the loss function is converging to ∞. In case that $y = 0$ and $\ell(f_\theta(x), y) \to 1$, it is the same. This is important, because there are sometimes high stakes on whether the data has been classified right or wrong, for example in the illness example. Additionally, if the class was predicted exactly, that means $y = 1$ and $\ell(f_\theta(x), y) \to 1$ or $y = 0$ and $\ell(f_\theta(x), y \to 0)$, the loss function is converging to 0. We can also write the function in a more compact way:

$$\ell(f_\theta(x), y) = -y \log f_\theta(x) - (1 - y) \log \left(1 - f_\theta(x)\right.$$

as the left term is 0 when $y = 0$ and 1 if $y = 1$, and the right term is 0 when $y = 1$ and 1 if $y = 0$. The presence of the logistic function in the loss function makes it hard to calculate the theta via something analogue to normal equations, thus, we use gradient descent. By using the relation $g'(z) = g(z) \cdot (1 - g(z))$, which can easily be proven, we can calculate the partial derivatives for the gradients with respect to a θ_i

by applying the chain rule multiple times:

$$\begin{aligned}\frac{\partial \ell}{\partial \theta_i} &= -y \cdot \frac{1}{g(\theta_T x)} \cdot g(\theta^T) \cdot (1 - g(\theta^T x)) \cdot x_i \\ &\quad - (1-y) \cdot \frac{1}{1 - g(\theta_T x)} \cdot g(\theta^T) \cdot (1 - g(\theta^T x)) \cdot x_i \\ &= [-y \cdot (1 - g(\theta^T x)) + (1-y) \cdot g(\theta^T x)] \cdot x_i \\ &= [-y \cdot + y \cdot g(\theta^T x)) + g(\theta^T x) \cdot -y \cdot g(\theta^T x)] \cdot x_i \\ &= [-y + g(\theta^T x)] \cdot x_i\end{aligned}$$

Thus, we obtain this gradient, which is similar to the linear regression:

$$\nabla L(\theta) = \frac{1}{N} \cdot X^T [g(X\theta) - y].$$

4. Neural networks

4.1. Layout of a neural network

When the function which should be approximated is highly nonlinear, traditional techniques are computationally too slow. Neural networks are also more effective when the source data is a high-dimensional vector space. Nowadays neural networks are capable of playing chess, predicting stock markets or perform image recognition. The basic unit in a neural network is a neuron. It takes multiple inputs and generates an output. In the picture below you can see a neuron with 3 features as an input. The neuron multiplies the inputs with a weight we call θ_i and then adds up all of these modified feature values. The orange neuron, so called *bias unit*, makes sure the output of the neuron doesn't get close to 0, even if all the features are close to 0.

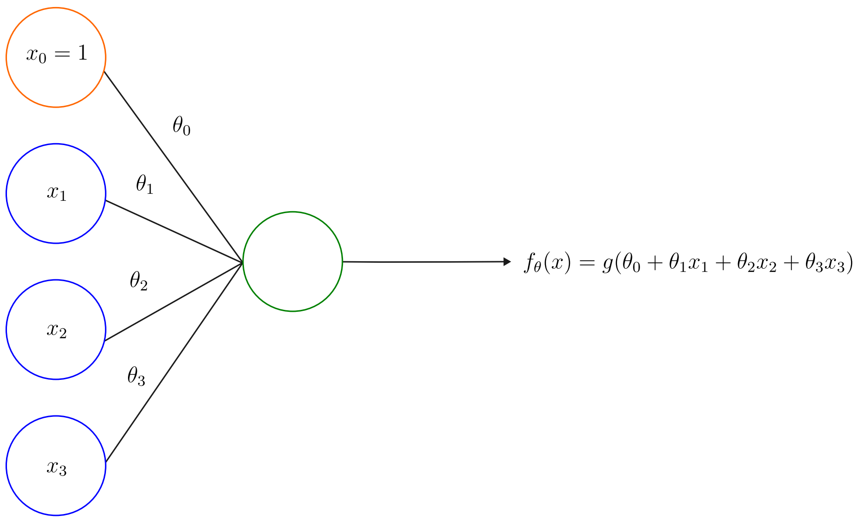

$x_0 = 1$
θ_0
x_1
θ_1
θ_2
x_2
θ_3
x_3
$f_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3)$

Abbildung 5: Illustration of a neuron

By using gradient descent we are going to estimate the vector $\theta = (\theta_0, \theta_1, \theta_2, \theta_3)$. These parameters are also called *weights* in the context of neural networks. The function g we are using is the logistic function, which we also used in the logistic regression. So the idea of a neuron is to approximate a specific function by using another function $f_\theta : \mathbb{R}^N \to \mathbb{R}$, in which N represents the number of features. When we stack multiple of these neurons together the neural network becomes more and more powerful. An example of a neural network containing multiple neurons can be found below.

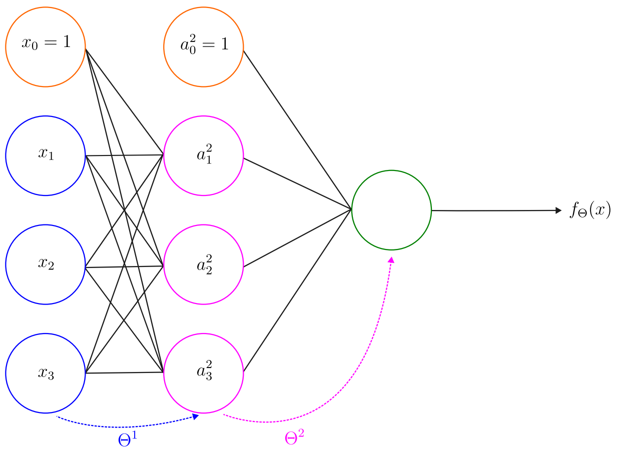


Abbildung 6: Illustration of a small neural network

This neural network has $L = 3$ layers. The first layer is called input layer, then there is a hidden layer and the last neuron is in the output layer. We call the second layer *dense* if all neurons are connected to the input neurons, like in this case. The number of neurons in a layer is notated as ν_l where $l \in \{1, \ldots, L\}$, excluding the bias unit. So in this example $\nu_2 = 3$. The values in the hidden layers are called *activation values*, you can see them as a kind of internal features in the neural network. These values are passed through the different hidden layers. This is the reason neural networks are sometimes seen as "smart" by some people.

While training the neural network, which just means computing the weights by minimizing some cost function, these internal features are allowing the neural network to understand the data somehow. We will call these values a_i^l, where l represents the layer and i the neuron in the layer. The vector of parameters is now called Θ, where Θ^l are the parameters used from layer l to layer $l + 1$. So the activation values can be calculated as follows:

$$\begin{aligned}
&a_0^1 = 1, a_1^1 = x_1, a_2^1 = x_2, a_3^1 = x_3 \text{ and } a_0^2 = 1\\
&a_1^2 = g(\Theta_{1,0}^1 + \Theta_{1,1}^1 x_1 + \Theta_{1,2}^1 x_2 + \Theta_{1,3}^1 x_3)\\
&a_2^2 = g(\Theta_{2,0}^1 + \Theta_{2,1}^1 x_1 + \Theta_{2,2}^1 x_2 + \Theta_{2,3}^1 x_3)\\
&a_3^2 = g(\Theta_{3,0}^1 + \Theta_{3,1}^1 x_1 + \Theta_{3,2}^1 x_2 + \Theta_{3,3}^1 x_3)\\
&a_1^3 = g(\Theta_{1,0}^2 + \Theta_{1,1}^2 a_1^2 + \Theta_{1,2}^2 a_2^2 + \Theta_{1,3}^2 a_3^2) = f_\Theta(x).
\end{aligned}$$

4.2. Examples of neural networks and approximation theorems

An example for a function that can be approximated with a simple neural network, consisting of a single neuron is the AND-function: $\{0,1\}^2 \to \{0,1\}$, $(x_1, x_2) \mapsto F(x_1, x_2) := x_1$ AND x_2. Using the logistic function as an activation-function and the following parameters $\theta = (-30, 20, 20)$ we get $f_\theta(x_1, x_2) = g(-30 + 20x_1 + 20x_2)$ resulting in the outputs being in accordance with the respective outputs of the AND-Function:

x_1	x_2	$f_\theta(x_1, x_2)$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

The parameters used in this example are not optimal, but serve as an illustration. The AND-Network we just computed will be useful in a more powerful example. Consider a line which divides the Cartesian plane into two regions.

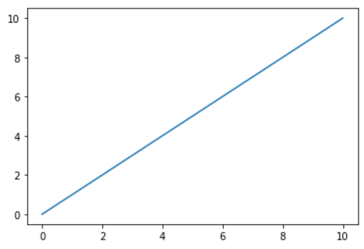


Abbildung 7: Example realisation with one line

A single-neuron-network is able to tell if a set of points (x_1, x_2) is in one of these regions, given the adequate weights. A network consisting of three of these networks and the AND-network will be able to tell if a set of points is contained within the area of three intersecting lines, which would be a triangle.

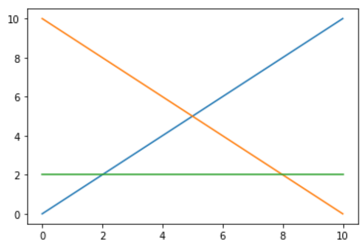


Abbildung 8: Example realisation with three lines

This makes neural networks powerful for classification, as it can easily be expanded to all convex polygons, by simply adding more lines. With the means of Delaunay triangulation any polygon can be broken down into triangles. Therefore any polygon can be realized by a neural network. As every Jordan curve can be approximated to arbitrary precision by a polygon, a single neural network exists, such that:

$$\forall (x_1, x_2) \in \text{Int}\, A \;\; f_\theta(x_1, x_2) = 1, \;\; \forall (x_1, x_2) \in \text{ExtA} \;\; f_\theta(x_1, x_2) = 0$$

.

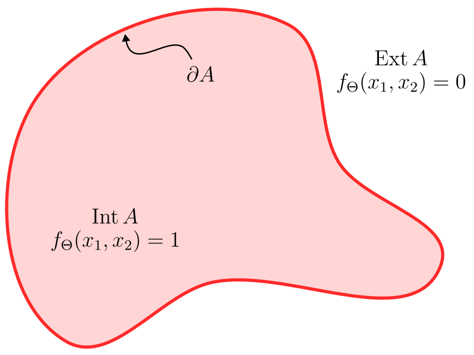


Abbildung 9: example illustration of a set

with ∂A being the boundary of the curve. Similarly, using a simplified version of the universal approximation theorem, which states that every continuous function $F : [a, b] \to \mathbb{R}$ can be approximated using a neural network, as for any continuous function on a compact set there is exists a sequence of step functions that uniformly converges towards that function. Any step function can in turn be exactly represented by a neural network. This shows the potential of neural networks in regression.

4.3. Computing the parameters with backpropagation

As with linear and logistic regression, the parameters best fitting the data set are found by minimizing a cost-function. An example for a cost-function is defined as follows.

$$L(\Theta) = \frac{1}{N}\sum_{i=1}^{K}\sum_{k=1}^{N} l(f_\Theta(x)_k, y_k) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}\left[y_{i,k}\log f_\Theta(x_i)_k + (1-y_{i,k})\log(1-f_\Theta(x_i)_k)\right]$$

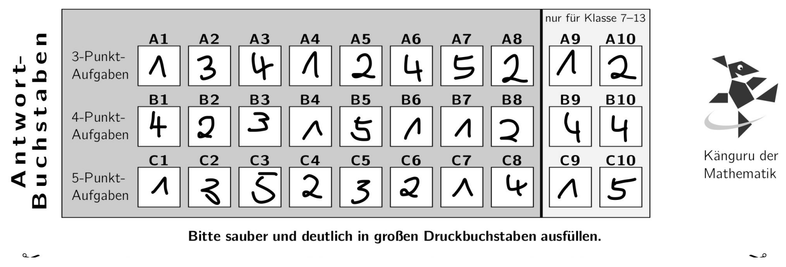
This function takes into account the number of samples as well as the classes. Taking the partial derivatives of the cost-function and applying the chain rule multiple times leaves us with an expression for the gradient.

$$\delta^{L-1} = [(\Theta^{L-1})^T\delta^L] \odot g'(z^{L-1})$$

with δ^l being the gradient of the weighted input of layer l, δ^L being the gradient of the last layer, Θ representing the according weights, $\odot$ denoting the Hadamard Product (element-wise multiplication) and z^l being a vector introduced to apply the chain rule to the cost function: $\frac{\partial L}{\partial \Theta^l_{i,v}}(\Theta) = \frac{\partial L}{\partial z^l_j} \cdot \frac{\partial z^l_j}{\partial \Theta^l_{i,j}}(\Theta)$. Thus, by going backward we are able to compute the gradients of the cost function with the formula above. Then we can use the gradients to perform gradient descent, finding the optimal weights corresponding to our data set.

4.4. An example of image recognition

To further understand the possibilities artificial neural networks open, let us work with an example: Every year, the 'Känguru-Wettbewerb', a multiple-choice math quiz for grades 3 to 13 takes place in over 80 countries. Students are given up to 30 questions of different difficulties and then have to write their answers on an answer sheet. But, after collecting every single one of over 6 million answer sheets, manually correcting each sheet would take a lot of time! This is why using image recognition is way easier, letting computers do the work.



Antwort-Buchstaben

	1	2	3	4	5	6	7	8	9	10
3-Punkt-Aufgaben	A1: 1	A2: 3	A3: 4	A4: 1	A5: 2	A6: 4	A7: 5	A8: 2	A9: 1	A10: 2
4-Punkt-Aufgaben	B1: 4	B2: 2	B3: 3	B4: 1	B5: 5	B6: 1	B7: 1	B8: 2	B9: 4	B10: 4
5-Punkt-Aufgaben	C1: 1	C2: 3	C3: 5	C4: 2	C5: 3	C6: 2	C7: 1	C8: 4	C9: 1	C10: 5

nur für Klasse 7–13

Känguru der Mathematik

Bitte sauber und deutlich in großen Druckbuchstaben ausfüllen.

Abbildung 10: example answer sheet from 'Känguru-Wettbewerb'

The most interesting part of this sheet is highlighted in grey: The actual answers of students. To find out what those students wrote down, we need to extract each answer square on its own. In the following, only the example for A1 will be shown:

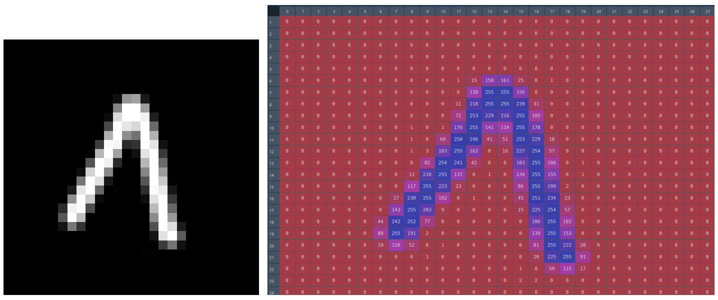
```
from PIL import Image
import PIL.ImageOps

width, height = im.size

###Relative Coordinates of row A
```

```
topA = height * 478/1025
bottomA = height * 587/1025

###Relative position of column 1
left1 = width * 432/2181
right1 = width * 535/2181

a1 = im.crop((left1, topA, right1, bottomA)
```

For each answer square, we crop out the square itself through its relative postion on the answer sheet. As the neural network used for number recognition works only on 28 by 28 images and white numbers on black ground, we afterwards invert the colors and scale down to the necessary image size.

```
a1 = PIL.ImageOps.fit(PIL.ImageOps.invert(a1), (28, 28))
```

The image a1 now looks like this:



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	1	15	150	161	25	0	1	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	130	255	255	156	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	11	218	255	255	239	31	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	72	253	229	216	255	105	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	1	0	1	176	255	141	114	255	178	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	1	0	60	250	246	41	51	253	229	18	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	1	3	183	255	162	0	16	227	254	57	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	82	254	241	42	0	0	183	255	106	0	1	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	13	210	255	137	0	1	0	134	255	155	0	1	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	117	255	223	23	0	0	0	86	255	199	2	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	27	230	255	102	0	1	0	0	45	251	234	23	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	143	255	203	9	0	0	0	0	15	225	254	57	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	44	242	252	77	0	0	0	0	0	0	186	255	102	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	88	255	191	2	0	0	0	0	0	0	139	255	153	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	19	120	52	0	1	0	0	0	0	0	81	255	222	28	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	20	225	255	91	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	59	115	17	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	0	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Abbildung 11: How a number looks like in the matrix

On the right, you can see a 28 × 28-Matrix representing the gray scale values of each pixel:

$$A_1 = \begin{bmatrix} a_{1,1} & \dots & a_{1,28} \\ \vdots & \ddots & \vdots \\ a_{28,1} & \dots & a_{28,28} \end{bmatrix}$$

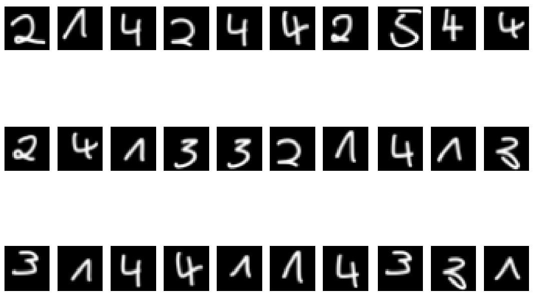
As the neural network works by having all 30 images as rows in a big array, we now need to convert every single image array into a row-vector and then combine these vectors into one 30 × 784-matrix:

```
a1 = numpy.ravel(a1)
```

This appends all other rows of the 28 × 28-Matrix onto the first row, so we get a 1 × 784 row vector:

$$A_1 = \begin{bmatrix} a_{1,1} & \dots & a_{1,28} & a_{2,1} & \dots & a_{28,28} \end{bmatrix}$$

After connecting all these row vectors, the 30 × 784 matrix can then be inserted into the neural network. This neural network has been trained with 60,000 images of numbers 0 to 9, partially mirrored or rotated, of the MNIST database. Having 25 neurons in its hidden layer, it achieves an accuracy of 88.62% on its test set, and 53.33% on the Känguru images. One could argue that 53.33% does not seem to be a lot, but that is already more than 5 times as good as random guesses!



2	9	4	7	4	4	2	5	4	4
2	4	1	3	3	7	0	6	3	9
2	1	4	4	3	0	6	2	9	3

Abbildung 12: Comparison of the input and output

On the left, you can see images taken from the Känguru sheet. The corresponding numbers on the right are corresponding guesses of the neural network on which numbers they are. Such a decrease in accuracy is actually quite normal, as the images of a test data set are most likely compiled and processed the same way as the images of the training data. The Känguru images could, for example, have a lower contrast with more grey areas around the white numbers, making it harder for the program to actually identify certain numbers. Overall, we can say that in these few hours we worked with neural networks, we achieved a good image recognition program, programming it all by ourselves, deriving the formulas needed to optimise the neural network instead of using ready to use packages from the Internet.