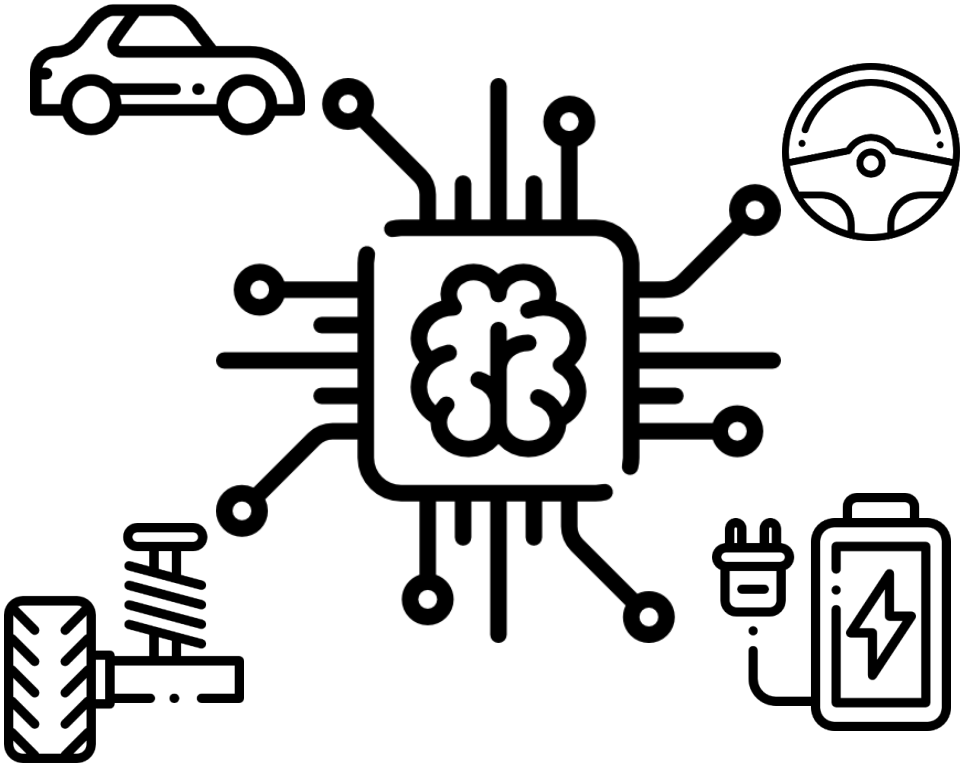
Institute of Automotive Technology
TUM School of Engineering and Design
Technical University of Munich

Artificial Intelligence in Automotive Technology

Maximilian Geißlinger / Fabian Netzler

Prof. Dr.-Ing. Markus Lienkamp



Lecture Overview

Lecture 16:15-17:45 | Practice 17:45-18:30	
1 Introduction: Artificial Intelligence	20.10.2022 – Maximilian Geißlinger
2 Perception	27.10.2022 – Sebastian Huber
3 Supervised Learning: Regression	03.11.2022 – Fabian Netzler
4 Supervised Learning: Classification	10.11.2022 – Andreas Schimpe
5 Unsupervised Learning: Clustering	17.11.2022 – Andreas Schimpe
6 Introduction: Artificial Neural Networks	24.11.2022 – Lennart Adenaw
7 Deep Neural Networks	08.12.2022 – Domagoj Majstorovic
8 Convolutional Neural Networks	15.12.2022 – Domagoj Majstorovic
9 Knowledge Graphs	12.01.2023 – Fabian Netzler
10 Recurrent Neural Networks	19.01.2023 – Matthias Rowold
11 Reinforcement Learning	26.01.2023 – Levent Ögretmen
12 AI-Development	02.02.2023 – Maximilian Geißlinger
13 Guest Lecture	09.02.2023 – to be announced

Introduction: Artificial Neural Networks
Prof. Dr.-Ing. Markus Lienkamp

(Lennart Adenaw, M. Sc.)

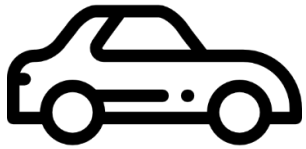
Agenda

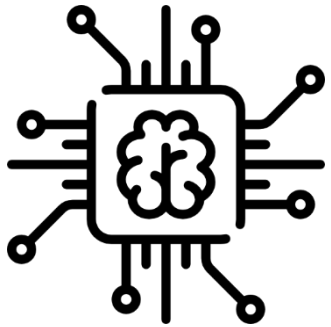
1. Chapter: Introduction

2. Chapter: Towards Artificial Neurons
    - 2.1 Linear Regression
    - 2.2 Gradient Descent
    - 2.3 The Neuron

3. Chapter: Multilayer Networks
    - 3.1 Functional Completeness
    - 3.2 MNIST Example

4. Chapter: Summary

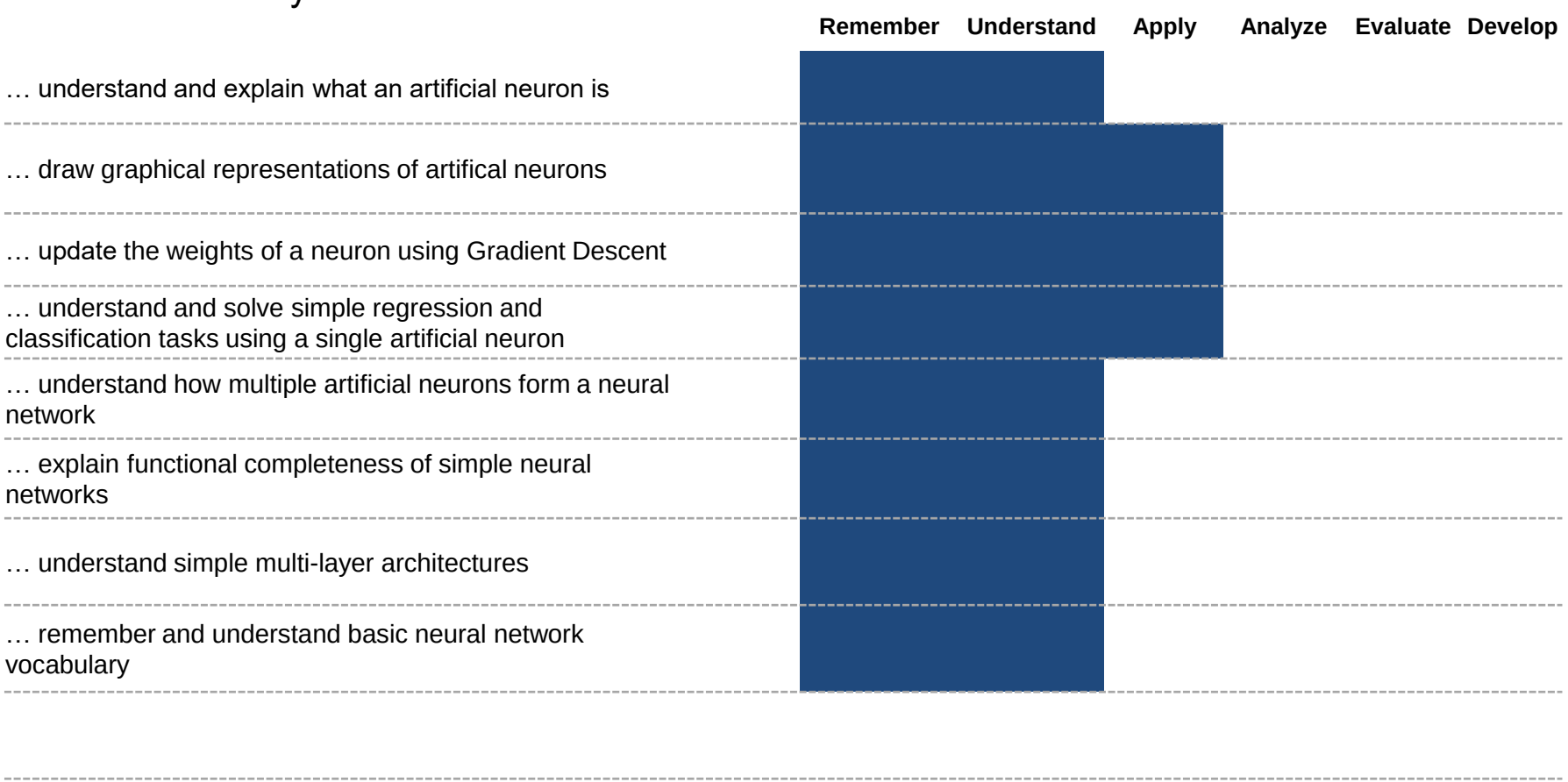




Objectives for Lecture 7: Introduction to Neural Nets

Depth of understanding

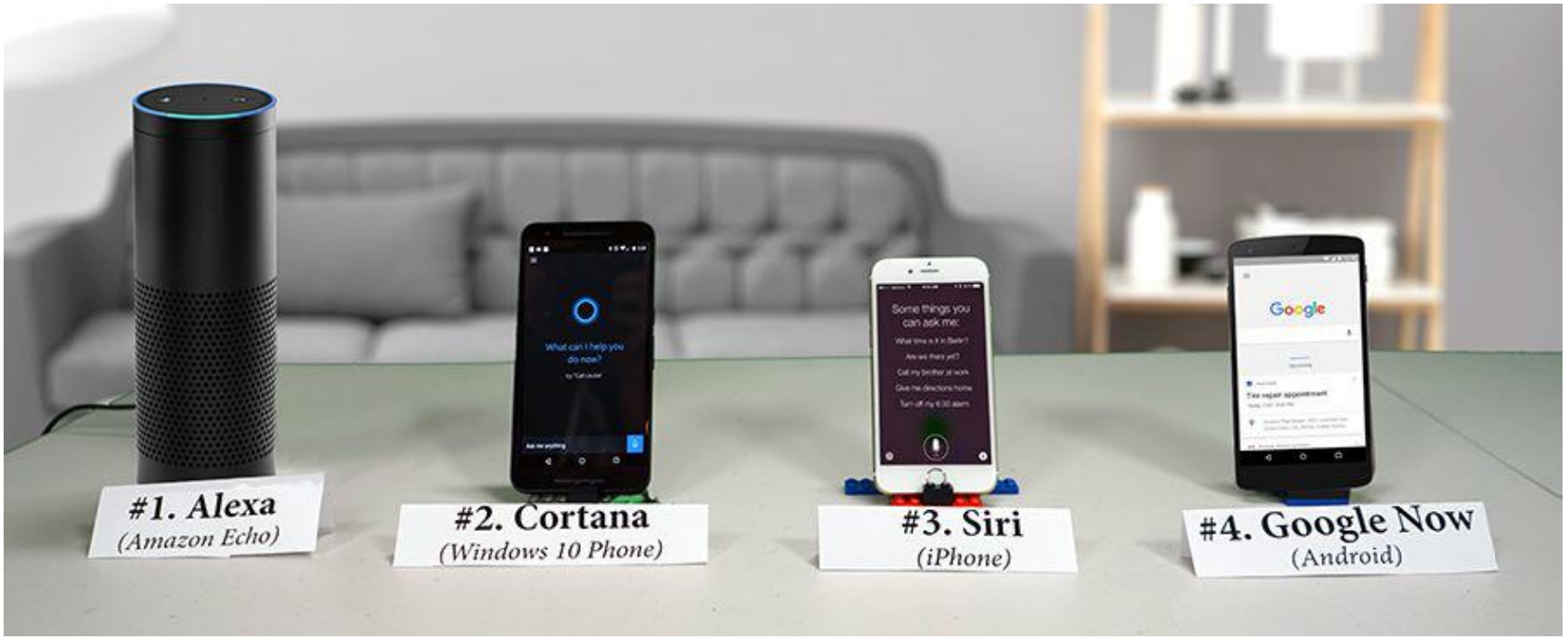
After the lecture you are able to…



Introduction
Neural Nets

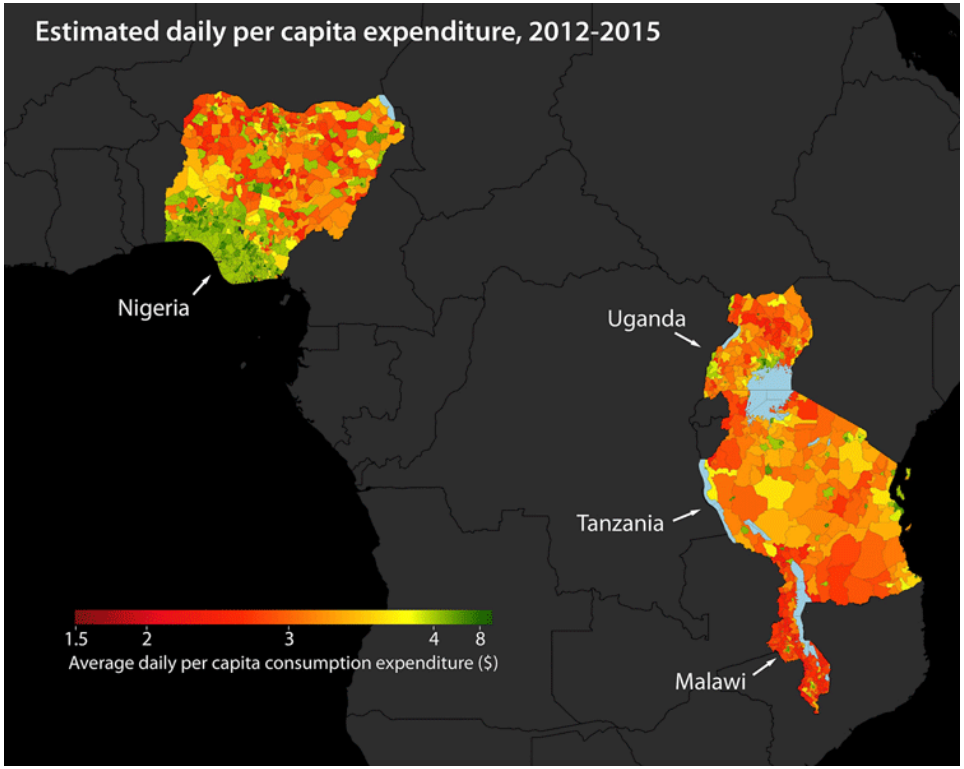
Image to Text Translation

Introduction
Neural Nets



Speech Recognition
Speech Segmentation
Text-to-Speech

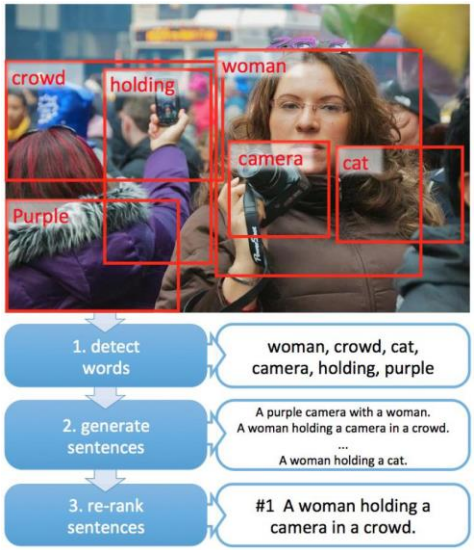
Introduction
Neural Nets



Using Machine Learning to Map Poverty from Satellite Imagery

Introduction
Neural Nets

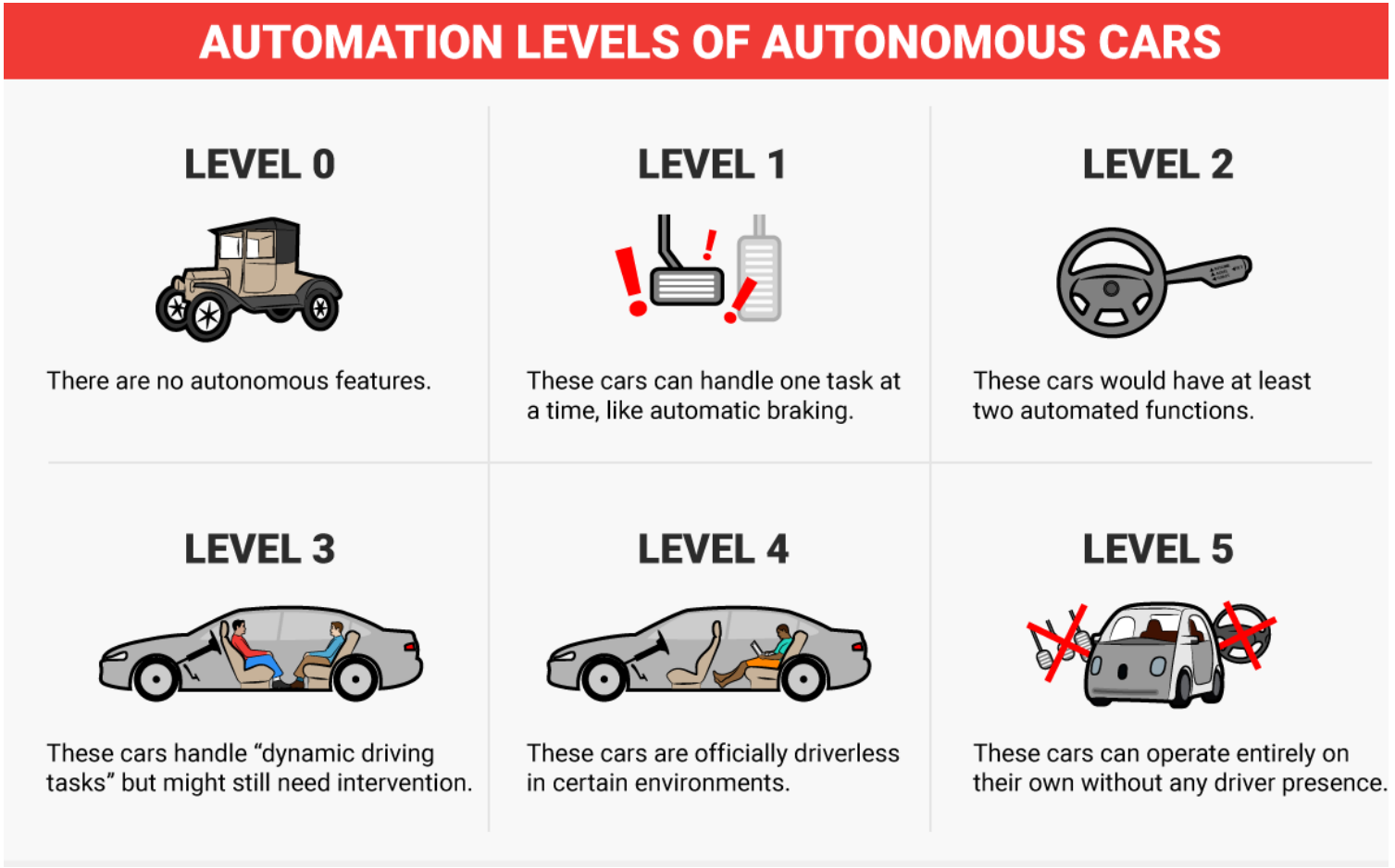
Image Colorization



Caption Generation

Artistic Style Transfer

Introduction
Neural Nets



Introduction
Neural Nets

What is the similarity between these tasks?

How can one general approach fit them all?

Additional Slides

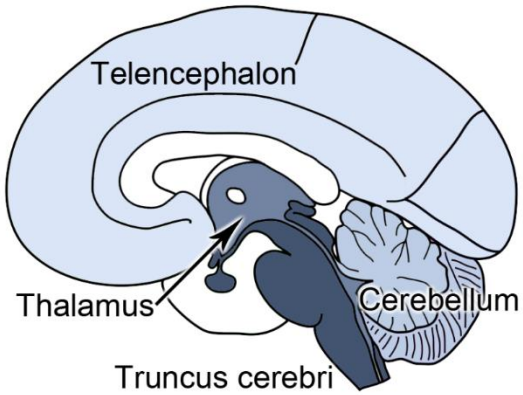
From the introductory lecture as well as the previous lectures, you are already familiar with a lot of examples for possible applications of machine learning. Many of the presented applications are realized by using Artificial Neural Networks (ANN). Among these are tasks of image processing, speech recognition, classification, forecasting, pattern detection, optical character recognition and last but not least automated driving. It is due to the diverse applications of one method, that ANN are often referred to as artificial intelligence.

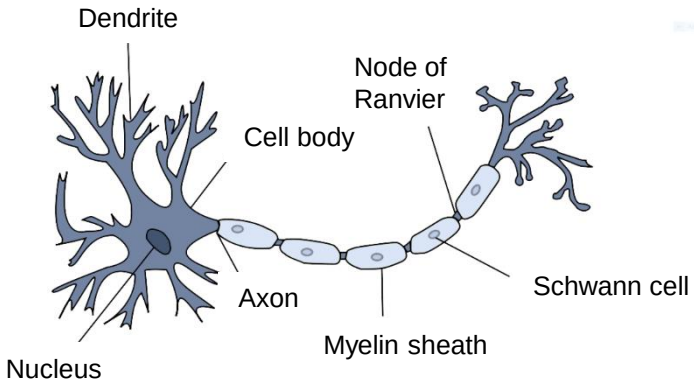
In order to understand ANN and the principals governing their theories, it is paramount to understand the similarities between the various tasks named above. In the introductory section of this lecture, we will deal with the similarities in order to unravel the general concept of ANN before introducing the basic building block of ANN: the artificial neuron.

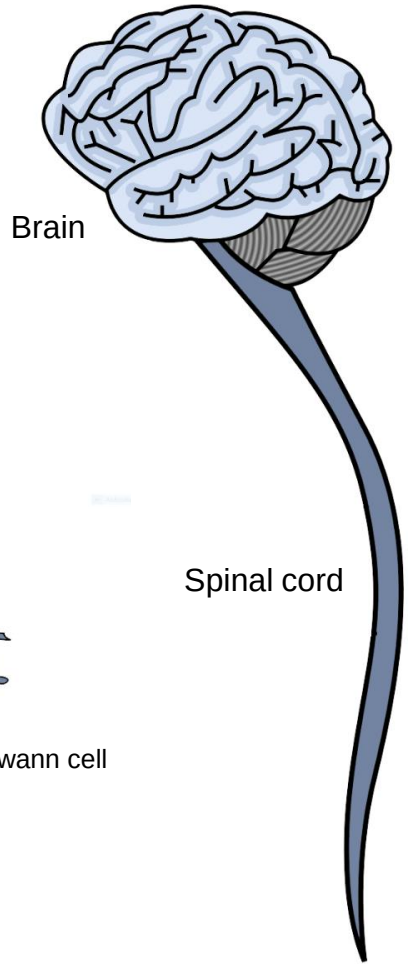
Starting with the artificial neuron, this lecture will give you an overview over the most important vocabulary, ideas of artificial neurons and ANN, including training and netting together multiple neurons to form a net.

Introduction

How to introduce Neural Nets



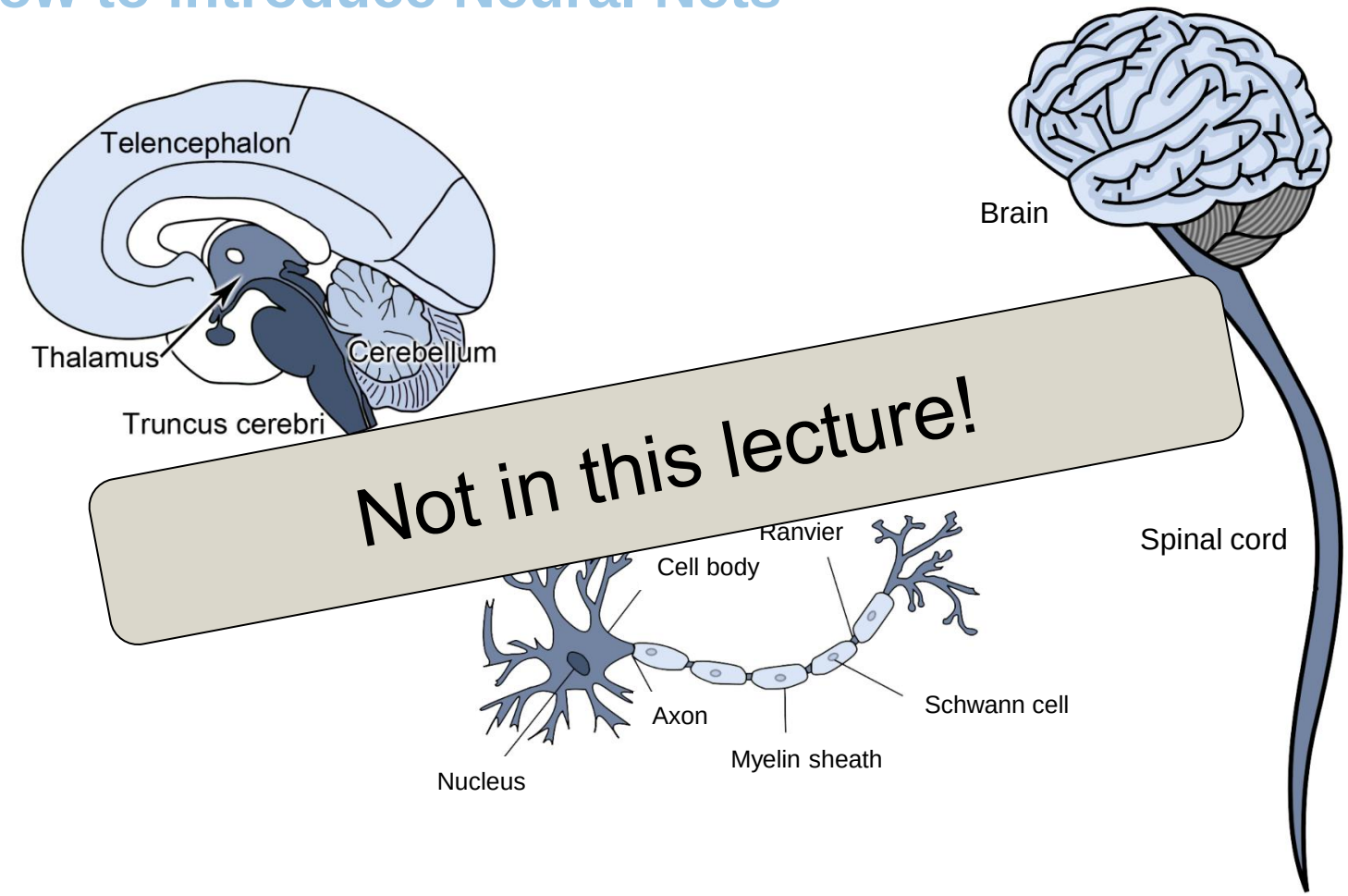




http://www.dkriesel.com/_media/science/neuronalenetze-en-zeta2-1col-dkrieselcom.pdf

Introduction

How to introduce Neural Nets



http://www.dkriesel.com/_media/science/neuronalenetze-en-zeta2-1col-dkrieselcom.pdf

Additional Slides

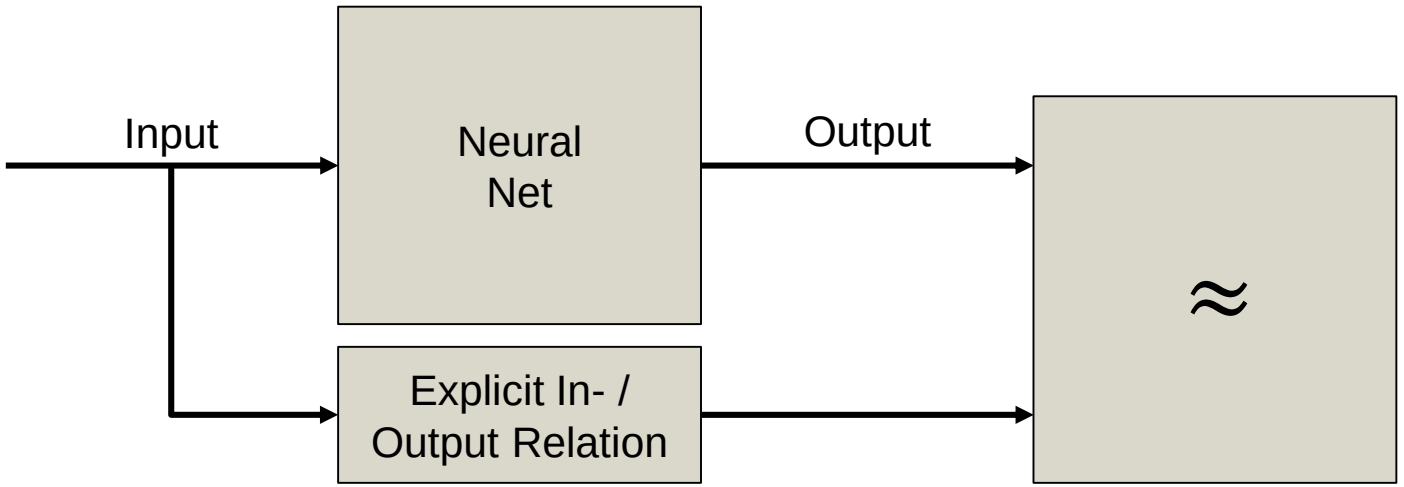
Many lectures, tutorials and talks introduce the topic of artificial neural networks (ANN) by retracing contemporary ANN research to its early roots, one of which being Rosenblatt‘s *Perceptron* from 1958.

Although this approach is appealing at first glance, it also has its shortcomings: Originally, ANN have been developed in an attempt to model the human brain technically. However, during the past 60 years, research has clearly shown that ANN do not accurately reflect the processes taking place in our brains. Nonetheless, they have proved to be a very powerful mathematical tool. As such (a mathematical tool!) they shall be introduced in this lecture. This introduction is to demystify their functionality, utility and usage.

Nevertheless, because of its wide acceptance, we will use the term „Neuron“ in this lecture.

Introduction
Universal Approximation Theorem

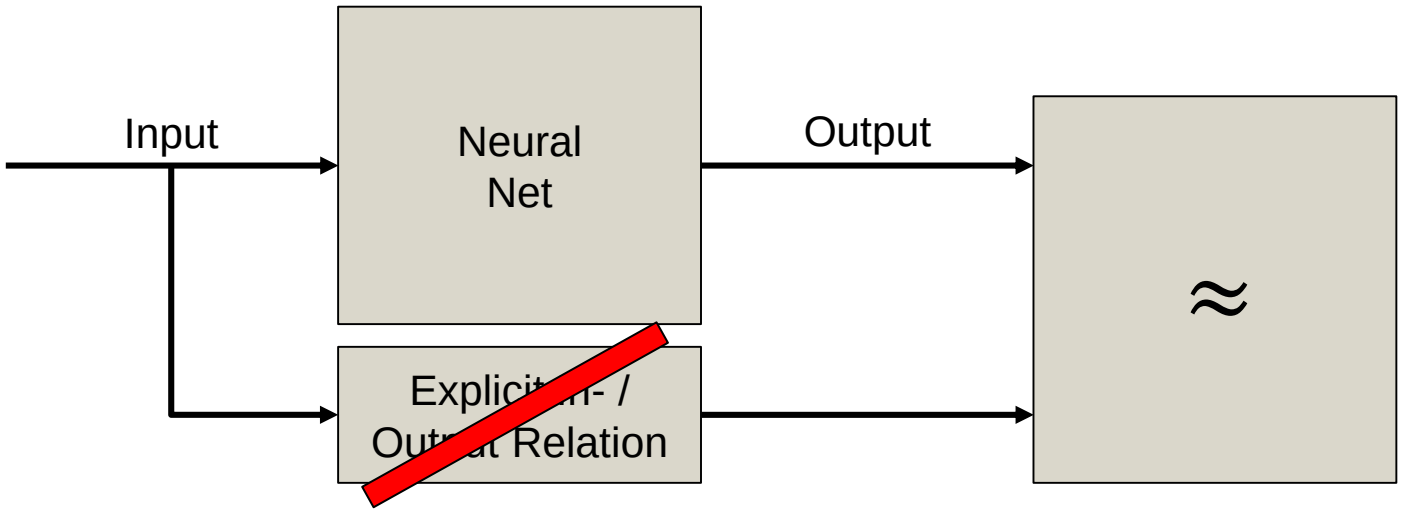
Neural Nets are Universal Approximators:



Introduction
Universal Approximation Theorem

Benefit of Neural Nets:

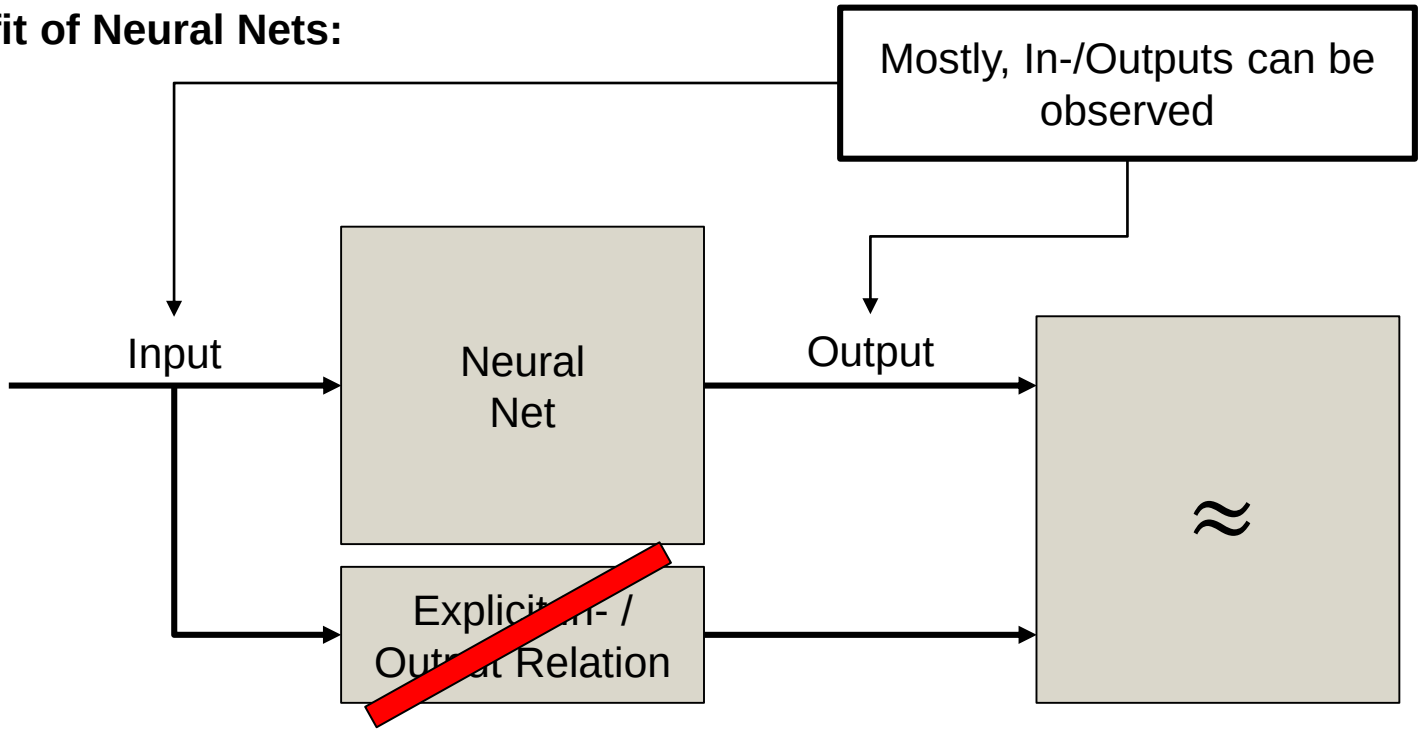
Often, no explicit In- / Output Relation is known!



Introduction

Universal Approximation Theorem

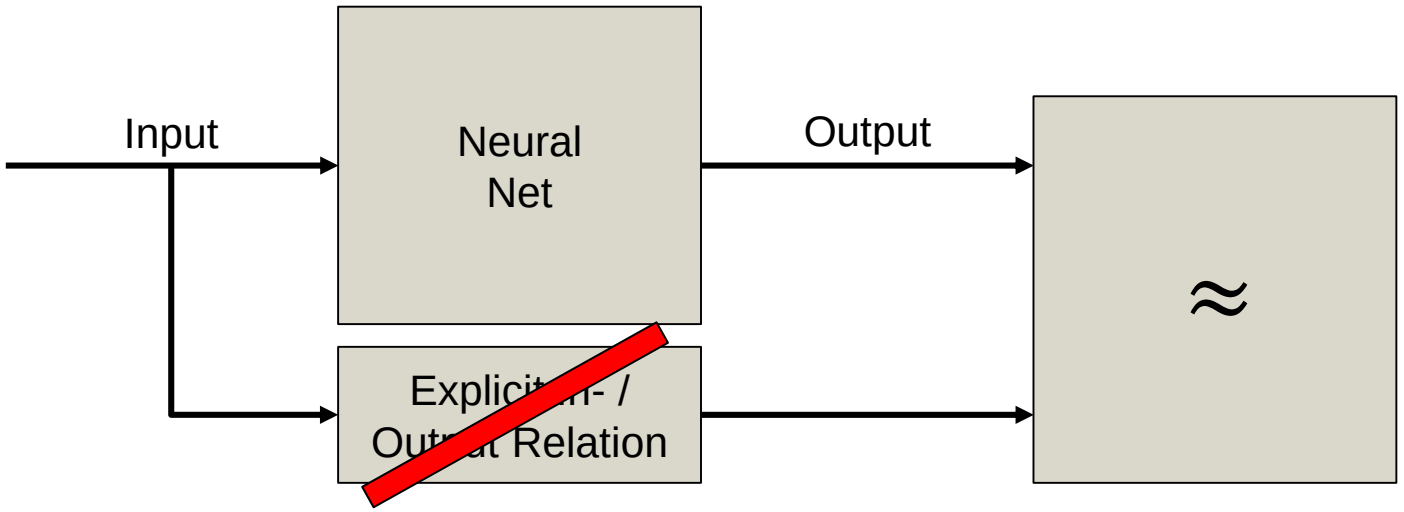
Benefit of Neural Nets:



Introduction
Universal Approximation Theorem

Benefit of Neural Nets:

Neural Nets can learn from observations



(Supervised) Machine Learning

Additional Slides

The Universal Approximation Theorem for ANN shows that even relatively simple ANN can approximate any analytic function with arbitrary accuracy. Even though, the Universal Approximation Theorem does not proof that the necessary parameters and architectures of the desired ANN can be found easily or by the means of finite time or input data, countless practical examples prove the applicability to real life problems.

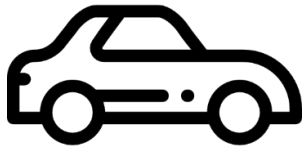
Since ANN are trained only by input data and expected outputs, they do not require full analytic knowledge of the physical domains being approximated. Hence, they are powerful tools when analytical solutions are unknown, too complex or not real-time ready as long as inputs and outputs of the physical system to be modeled can be observed.

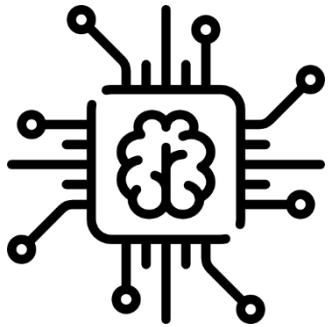
That being said, analytical solutions still ought to be sought whenever possible because they offer extended insights into the modeled domain as well as a potentially higher numeric accuracy.

Introduction: Artificial Neural Networks
Prof. Dr.-Ing. Markus Lienkamp

(Lennart Adenaw, M. Sc.)

Agenda





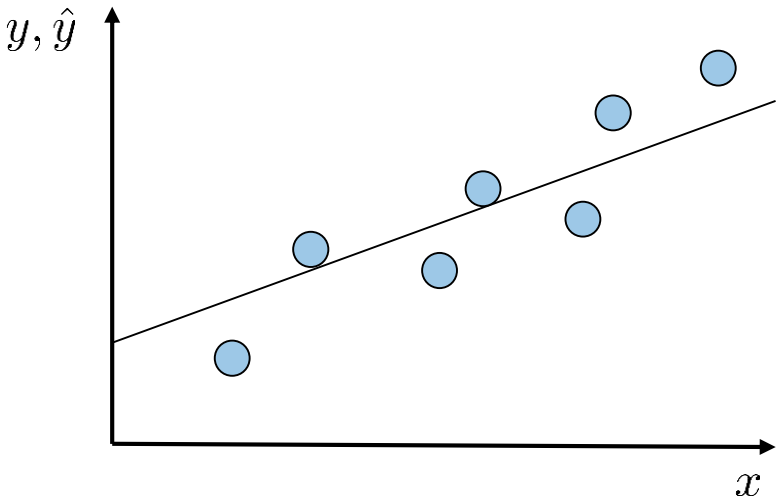
Towards Artificial Neurons

Linear Regression

The Simplest Approximation:

$$y = f(\vec{w} \cdot \vec{x}, b) = \sum_i w_i x_i + b = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b$$

$\vec{x}$:	$Input\ Vector$
$\vec{w}$:	$Weight\ Vector$
b :	$Bias$
y :	$Output$
$\hat{y}$:	$Training\ Data$



Additional Slides

In order to derive the necessary ideas and math for the understanding of neural networks, which are proven to be „universal approximators“, we start by taking a closer look at the simplest form of mathematical approximation: linear regression without basis functions.

Linear regression can be used to define and introduce a number of important concepts in the context of deep learning like Weights, Biases, Loss Function, Forward Pass and Gradient Descent.

From linear regression, one can then derive more complex forms of regression by introducing non-linearities into the corresponding models. From this line of thought, the basic model of a Neuron as it is used in ANN can be derived.

Additional Slides

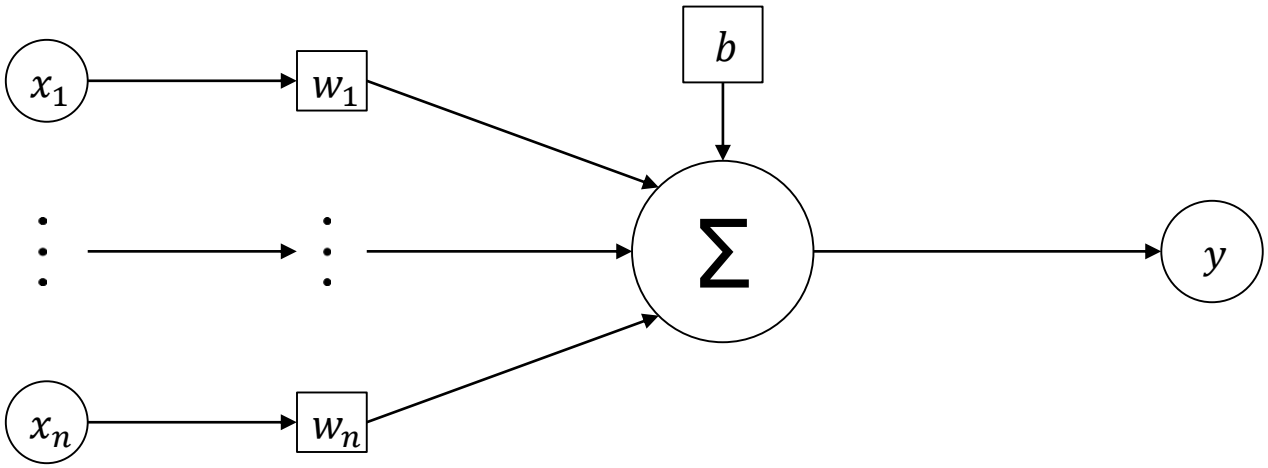
The idea of linear regression is to find Weights $\vec{w}$ and Bias b, such that the resulting function $y = f(\vec{w} \cdot \vec{x}, b) = \sum_i w_i x_i + b$ approximates a data set with inputs $\vec{x} \in X$ and Outputs $\hat{y} \in \hat{Y}$ as accurately as possible. This concept should be familiar to you from lecture 3.

Towards Artificial Neurons

Linear Regression

Graphical Representation:

$$y = f(\vec{w} \cdot \vec{x}, b) = \sum_i w_i x_i + b = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b$$

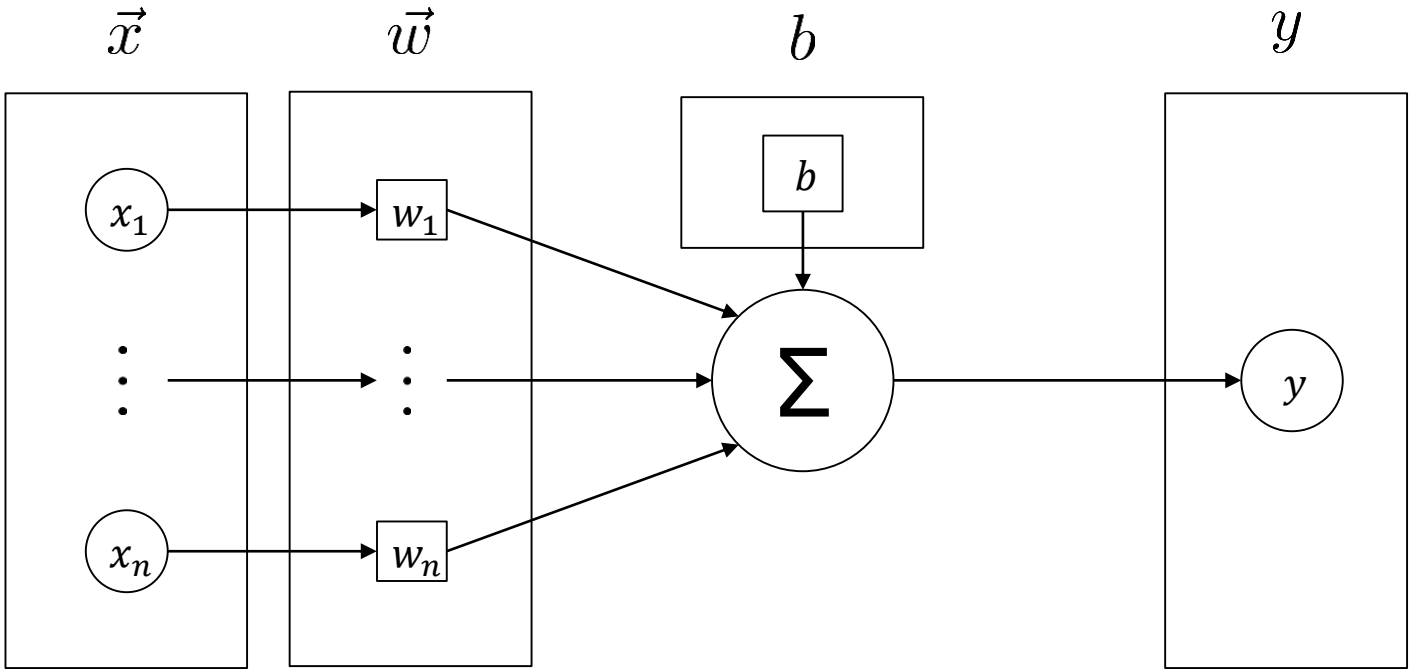


Towards Artificial Neurons

Linear Regression

Graphical Representation:

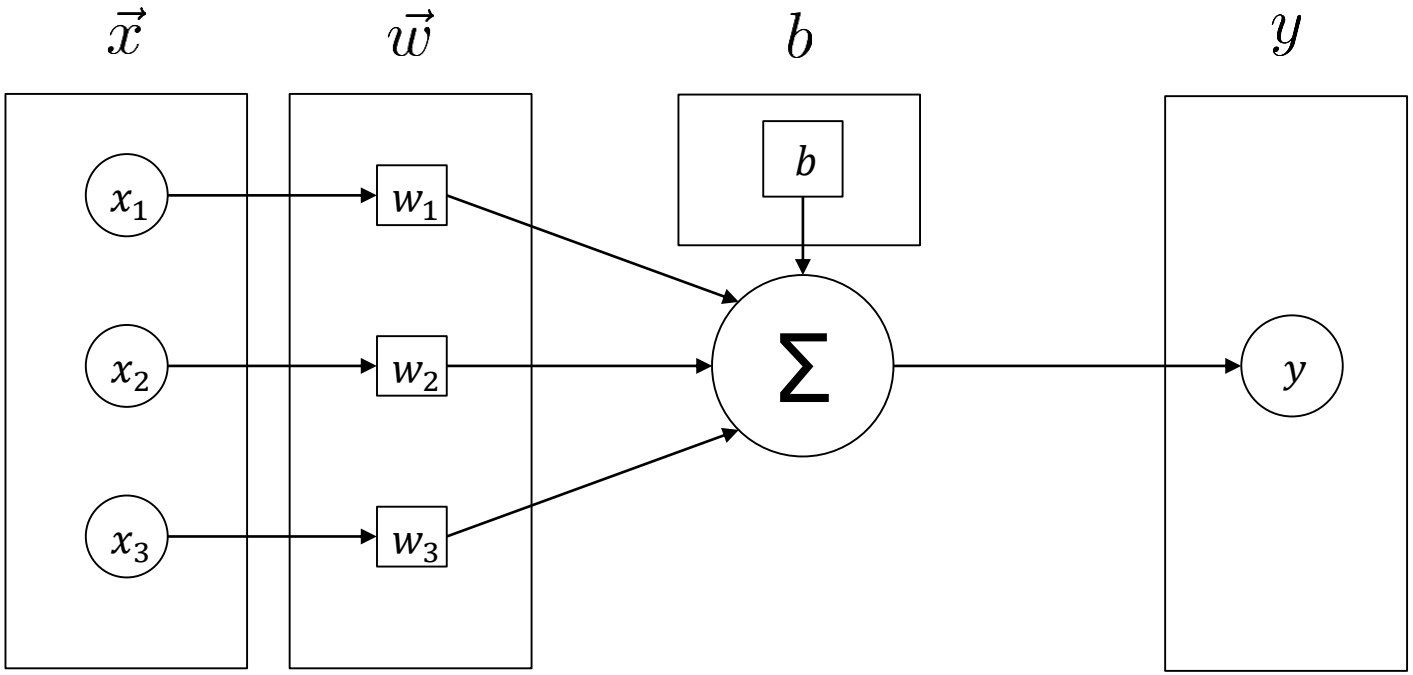
$$y = f(\vec{w} \cdot \vec{x}, b) = \sum_i w_i x_i + b = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b$$



Towards Artificial Neurons

Linear Regression

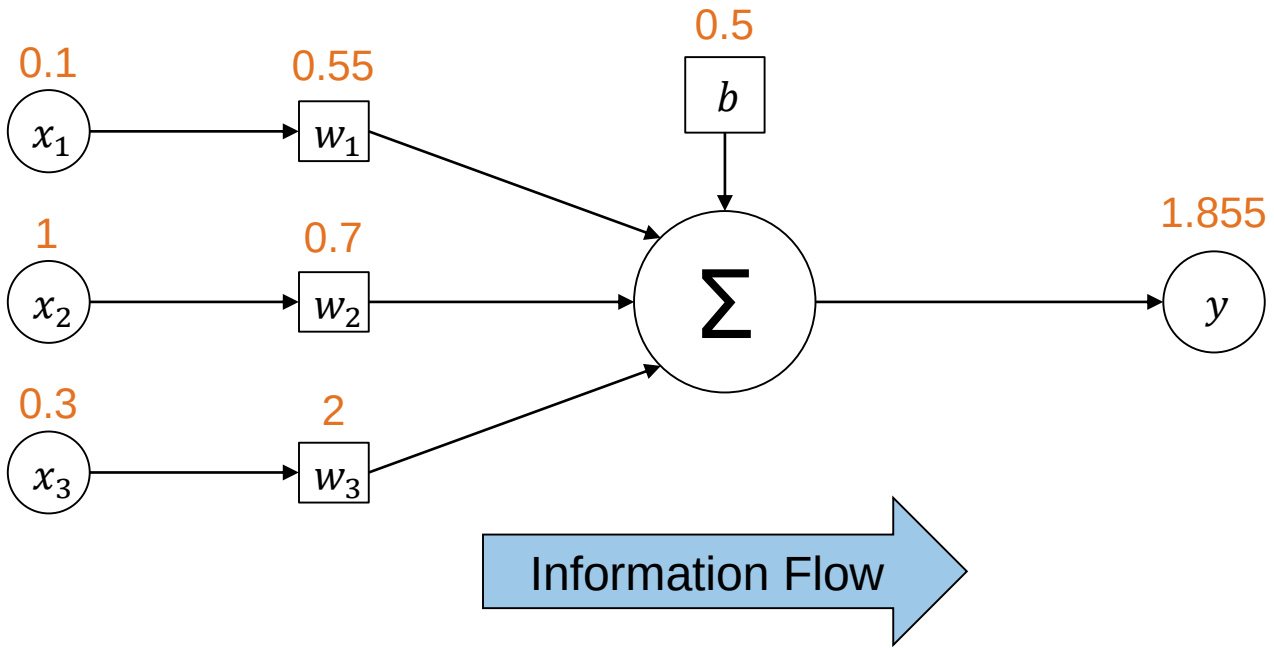
Graphical Representation – 3 Inputs Example:



Towards Artificial Neurons

Linear Regression

Forward Pass:



Additional Slides

Since this is an introduction to neural networks, neural network vocabulary is employed. Hence, the neural network term „Forward Pass“ is used, although it is not a term used in linear regression, because it refers to the corresponding process in an artificial neuron.

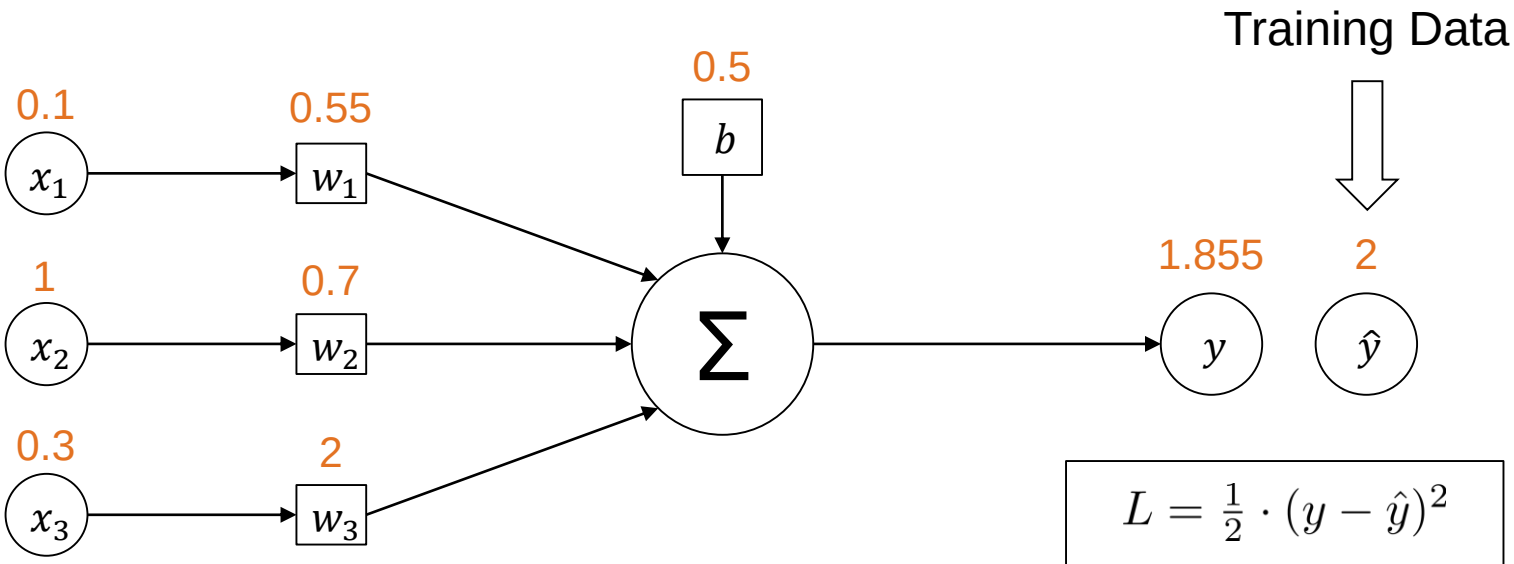
The „Backward Pass“ which is part of the „Backpropagation“ idea will be introduced the next lecture.

During the forward pass, the output of a linear regression (i.e. an artificial neuron respectively) is calculated by multiplying the inputs with weights and adding a bias term to them.

Towards Artificial Neurons

Linear Regression

Loss Function:



Towards Artificial Neurons

Linear Regression

Optimization Problem:

$$minimize_{\vec{w},b}(L(y,\hat{y}))$$

$$L = \frac{1}{2} \cdot (y - \hat{y})^2$$

$$y = f(\vec{w},\vec{x}) = \sum_i (w_i \cdot x_i + b)$$

Additional Slides

During the training phase, the task is to tweak the weights in the linear regression model to fit the input/output relation in the training data as good as possible. For a single pair of in- and output-values, a loss can be calculated, measuring the error of the regression for that xy pair. In order to fit the model to the training data, weights and biases have to be adjusted such that the Loss on the training data set is minimized. After training, a linear regression generalizes a linear relation between its input and output space. You can think of a linear regression as a linear data model that is able to parametrize itself using training data.

The governing idea of the remainder of this lecture is the generalization of this concept to arbitrary, possibly non-linear in-/output relations which are previously unknown.

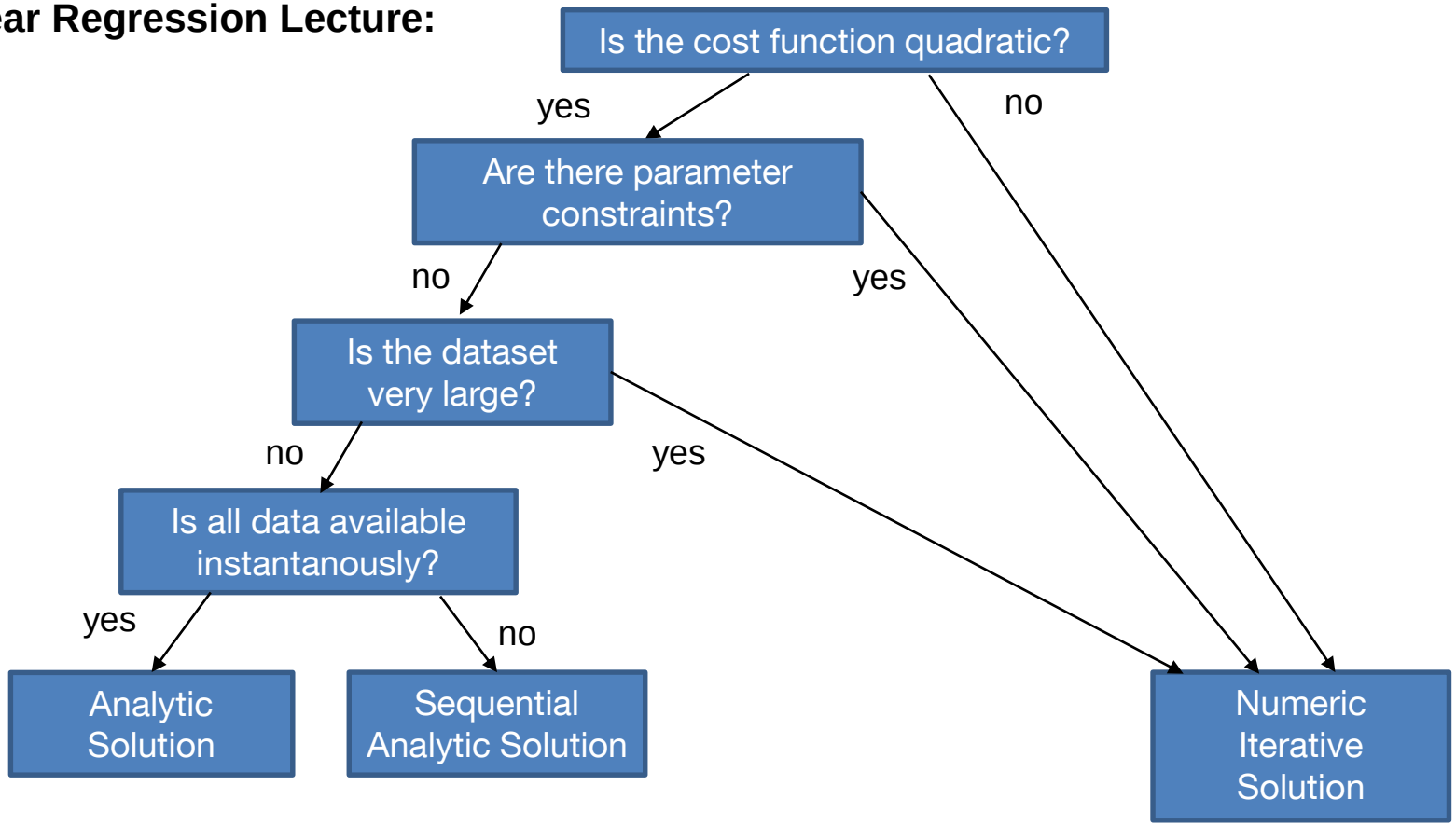
To introduce ANN, the road map of this lecture includes the transition from a linear regression model to an artificial neuron and from an artificial neuron to an artificial neural net.

We will start by having a closer look at the optimization problem in linear regression models, that closely resembles the training of a single artificial neuron.

Towards Artificial Neurons

Linear Regression

Linear Regression Lecture:



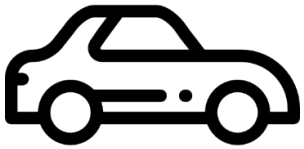
Additional Slides

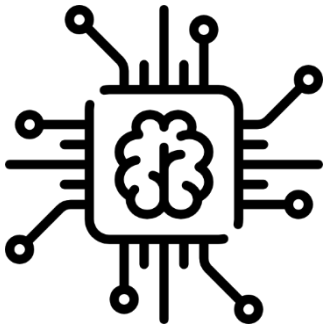
From lecture 3, you already know how to solve the optimization problem in linear regression models. In the context of ANN we will mostly deal with non-quadratic cost (loss) functions and large amounts of data. Furthermore, it is in the nature of ANN in practical applications to form highly interwoven structures that defy any tries to analytically solve the optimization problem. Therefore, we will have a look at a universal, numeric iterative method to solve the optimization problem for both linear regression models and ANN. This method is called gradient descent and will be presented in the next chapter.

Introduction: Artificial Neural Networks
Prof. Dr.-Ing. Markus Lienkamp

(Lennart Adenaw, M. Sc.)

Agenda





Towards Artificial Neurons

Gradient Descent

Approach:

- Gradient defines steepest ascent ($-\nabla L$)

- Update weights by a step in the opposite direction (i.e. steepest descent, length α)

- Stop when optimization criterium is met (e.g. loss threshold ϵ) or after N iterations

$$\nabla L = \begin{pmatrix} \frac{\delta L}{\delta w_1} \\ \frac{\delta L}{\delta w_2} \\ \vdots \\ \frac{\delta L}{\delta w_n} \\ \frac{\delta L}{\delta b} \end{pmatrix} \qquad \vec{w_{new}} = \vec{w_{old}} - \alpha \cdot \nabla L$$

Additional Slides

Gradient Descent leverages the mathematical finding that the gradient of a multidimensional function points into the direction of the steepest ascent of the function. The gradient is a vector that contains the partial derivatives of the function with respect to the input variables of the function.

Hence, when trying to minimize a function (e.g. the Loss function in a linear regression model) with a set of parameters (weights and bias), the gradient of the function can be computed and an optimized set of parameters can be determined by adding the negative of the gradient (steepest descent) to the original set of parameters. The step length or *learning rate* α determines the magnitude of parameter adjustments. The effects of different learning rates will be investigated later in this lecture. Gradient Descent is stopped either after a certain number of iterations, or when another optimization criteria is met (e.g. the loss is smaller than a predefined threshold).

A parameter update can be calculated for every in-/output relation in the training data set to parametrize the model. By the superposition of parameter updates, the model parameters converge to a state where the function is minimized (e.g. the model output approximates the observed data in the best way).

In the context of ANN, a set of parameters is called a weight vector.

To simplify the notation, the bias – which is also a parameter – is included into the weight vector. Therefore, updates of bias and weights can be conveniently written in a single line and the partial derivative of the Loss function with regards to the bias appears in the gradient.

Towards Artificial Neurons

Gradient Descent

Well behaved:

$$\vec{w_{new}} = \vec{w_{old}} - \alpha \cdot \nabla L$$

$L = w^2$

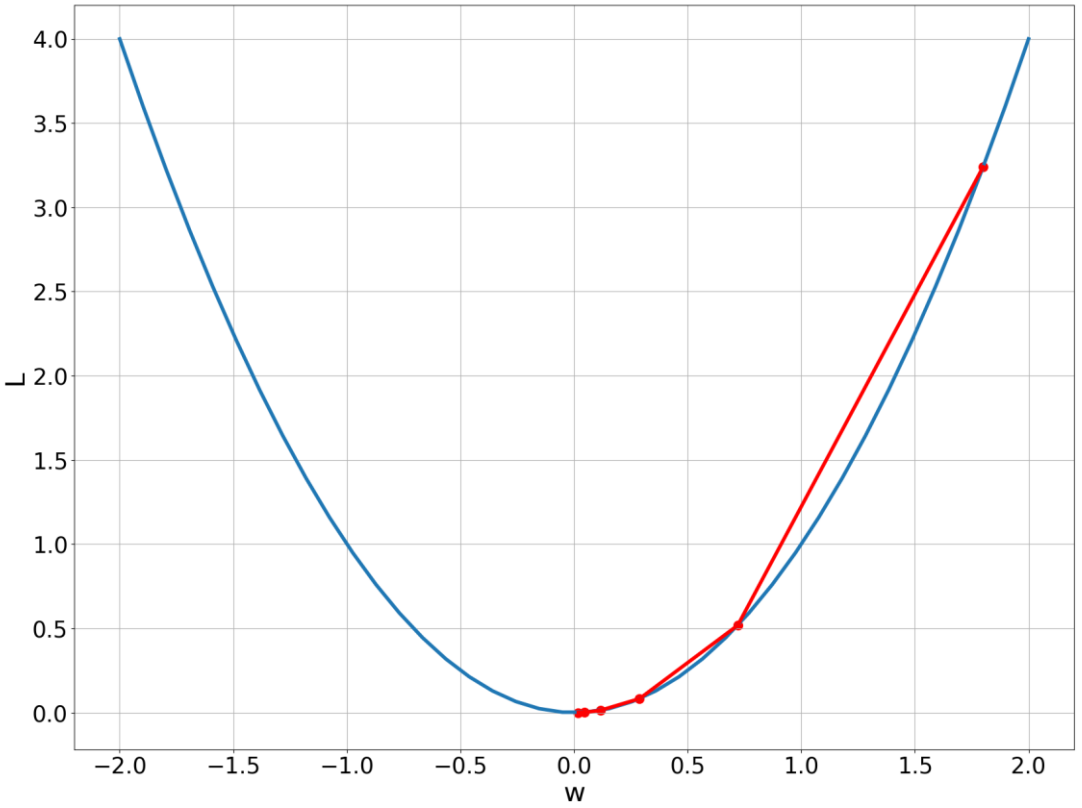
$\nabla L = \frac{dL}{dw} = 2 \cdot w$

$\alpha = 0.3$

$w_0 = 1.8$

$\epsilon = 0.001$

$N = 5$



Additional Slides

The example shows a very simple loss function that depends only on one weight. The weight function is minimized in five Gradient Descent iterations. At each step, the gradient of the function with respect to the weight is evaluated and the weight is updated. The training is stopped after a weight threshold of 0.001 is met (resulting in five iterations).

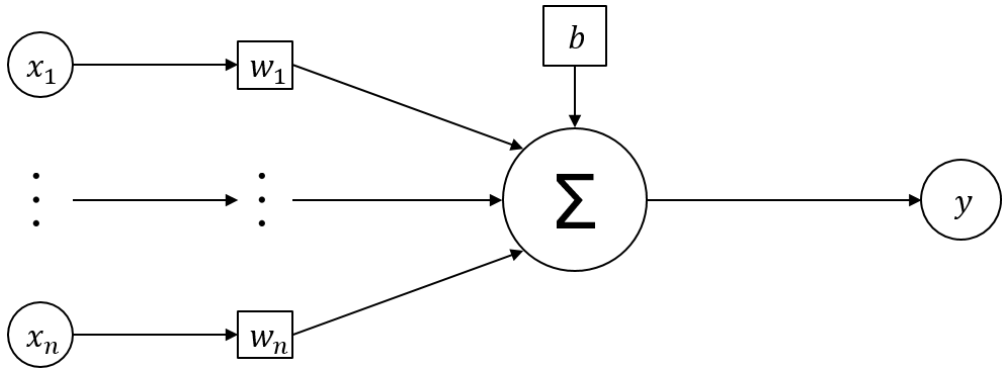
Towards Artificial Neurons

Gradient Descent

Finding the Gradient:

$$L = \frac{1}{2} \cdot (y - \hat{y})^2$$

$$\nabla L = \begin{pmatrix} \frac{\delta L}{\delta w_1} \\ \frac{\delta L}{\delta w_2} \\ \vdots \\ \frac{\delta L}{\delta w_n} \\ \frac{\delta L}{\delta b} \end{pmatrix}$$



$$\frac{\delta L}{\delta w_i} = \frac{dL}{dy} \cdot \frac{\delta y}{\delta w_i}$$

$$\frac{\delta L}{\delta b} = \frac{dL}{dy} \cdot \frac{\delta y}{\delta b}$$

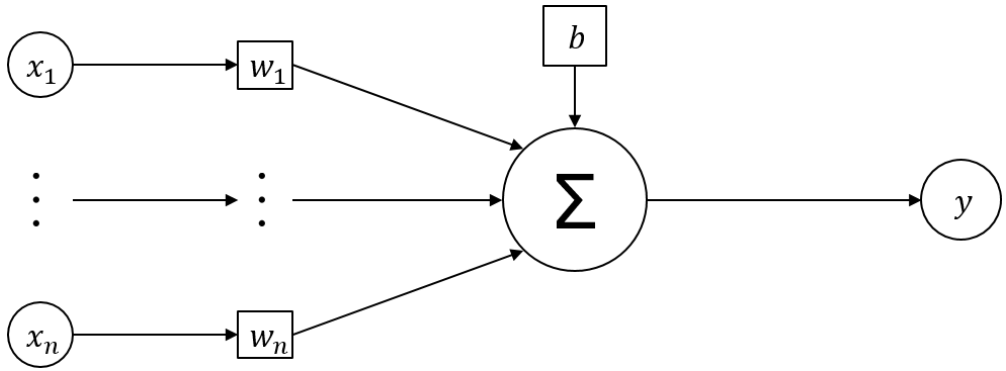
Towards Artificial Neurons

Gradient Descent

Finding the Gradient:

$$L = \frac{1}{2} \cdot (y - \hat{y})^2$$

$$\nabla L = \begin{pmatrix} \frac{\delta L}{\delta w_1} \\ \frac{\delta L}{\delta w_2} \\ \vdots \\ \frac{\delta L}{\delta w_n} \\ \frac{\delta L}{\delta b} \end{pmatrix}$$



$$\frac{\delta L}{\delta w_i} = \frac{dL}{dy} \cdot \frac{\delta y}{\delta w_i} \qquad \frac{\delta L}{\delta b} = \frac{dL}{dy} \cdot \frac{\delta y}{\delta b}$$

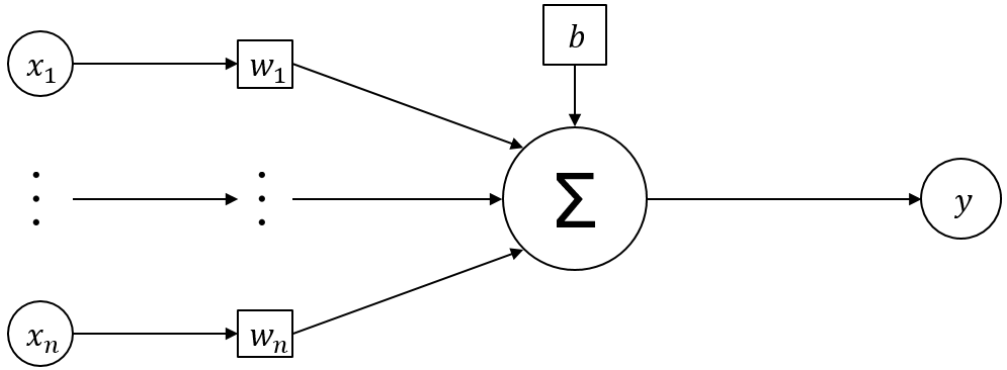
Towards Artificial Neurons

Gradient Descent

Finding the Gradient:

$$L = \frac{1}{2} \cdot (y - \hat{y})^2$$

$$\nabla L = \begin{pmatrix} \frac{\delta L}{\delta w_1} \\ \frac{\delta L}{\delta w_2} \\ \vdots \\ \frac{\delta L}{\delta w_n} \\ \frac{\delta L}{\delta b} \end{pmatrix}$$



$$\frac{\delta L}{\delta w_i} = \frac{dL}{dy} \cdot \frac{\delta y}{\delta w_i} \qquad \frac{\delta L}{\delta b} = \frac{dL}{dy} \cdot \frac{\delta y}{\delta b}$$

$$\frac{dL}{dy} = \frac{d}{dy} \frac{1}{2} \cdot (y - \hat{y})^2 = y - \hat{y} = \Delta y$$

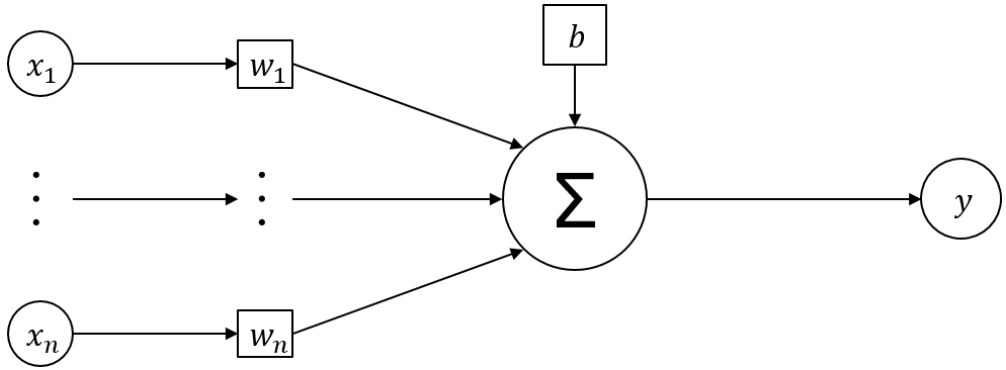
Towards Artificial Neurons

Gradient Descent

Finding the Gradient:

$$L = \frac{1}{2} \cdot (y - \hat{y})^2$$

$$\nabla L = \begin{pmatrix} \frac{\delta L}{\delta w_1} \\ \frac{\delta L}{\delta w_2} \\ \vdots \\ \frac{\delta L}{\delta w_n} \\ \frac{\delta L}{\delta b} \end{pmatrix}$$



$$\frac{\delta L}{\delta w_i} = \frac{dL}{dy} \cdot \frac{\delta y}{\delta w_i} = \Delta y \cdot x_i$$

$$\frac{\delta y}{\delta w_i} = \frac{\delta}{\delta w_i} \sum_i w_i \cdot x_i + b = x_i$$

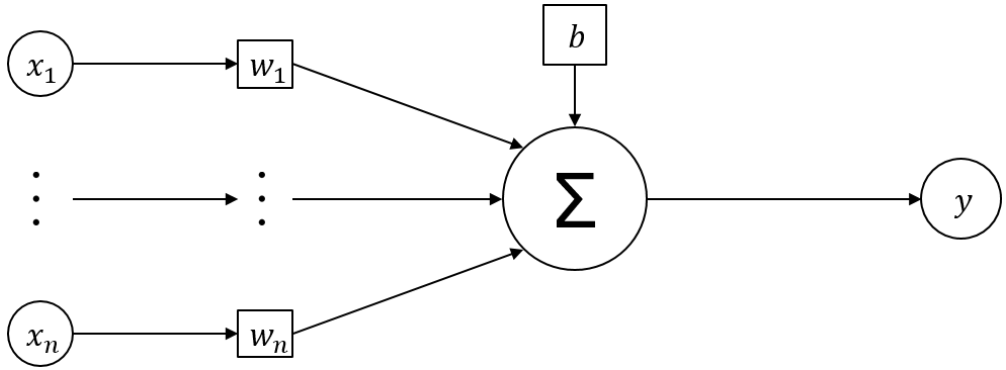
Towards Artificial Neurons

Gradient Descent

Finding the Gradient:

$$L = \frac{1}{2} \cdot (y - \hat{y})^2$$

$$\nabla L = \begin{pmatrix} \frac{\delta L}{\delta w_1} \\ \frac{\delta L}{\delta w_2} \\ \vdots \\ \frac{\delta L}{\delta w_n} \\ \frac{\delta L}{\delta b} \end{pmatrix}$$



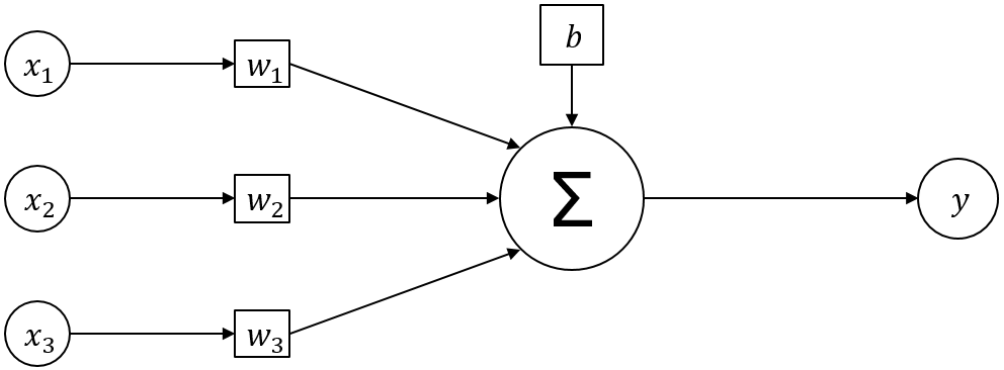
$$\frac{\delta L}{\delta b} = \frac{dL}{dy} \cdot \frac{\delta y}{\delta b} = \frac{dL}{dy} \cdot \frac{\delta y}{\delta w_i} = \Delta y$$

$$\frac{\delta y}{\delta b} = \frac{\delta}{\delta b} \cdot \sum_i w_i \cdot x_i + b = 1$$

Towards Artificial Neurons

Gradient Descent

Numerical Example:

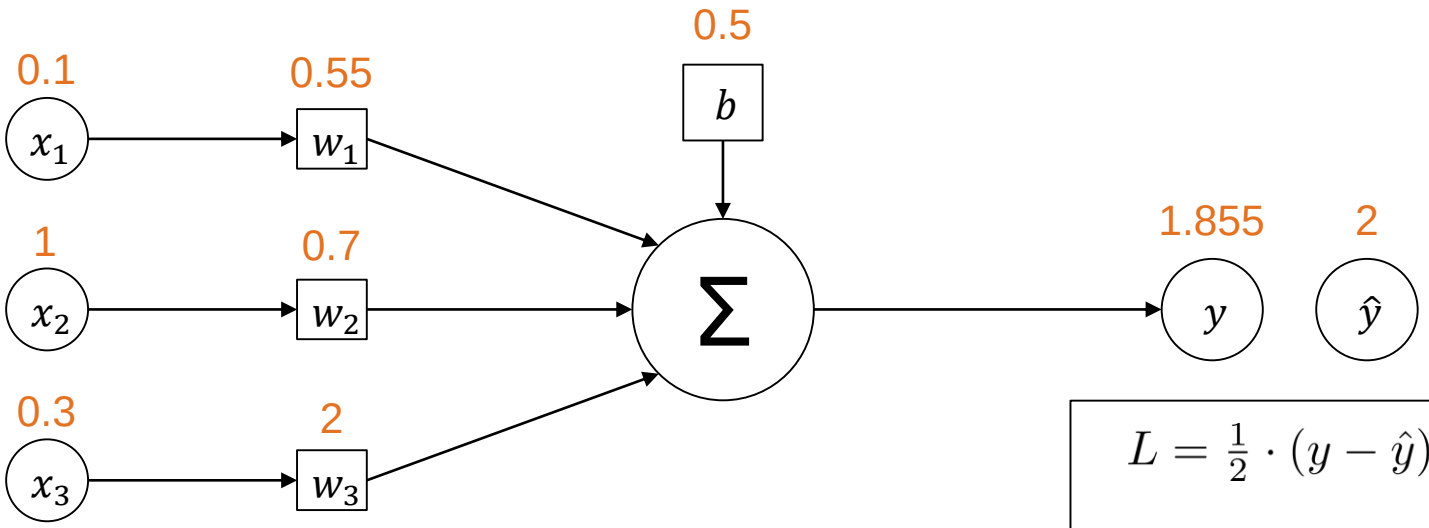


$$\nabla L = \begin{pmatrix} \frac{\delta L}{\delta w_1} \\ \frac{\delta L}{\delta w_2} \\ \frac{\delta L}{\delta w_3} \\ \frac{\delta L}{\delta b} \end{pmatrix} = \begin{pmatrix} \Delta y \cdot x_1 \\ \Delta y \cdot x_2 \\ \Delta y \cdot x_3 \\ \Delta y \end{pmatrix}$$

Towards Artificial Neurons

Gradient Descent

$$\nabla L = \begin{pmatrix} \Delta y \cdot x_1 \\ \Delta y \cdot x_2 \\ \Delta y \cdot x_3 \\ \Delta y \end{pmatrix} = - \begin{pmatrix} 0.145 \cdot 0.1 \\ 0.145 \cdot 1 \\ 0.145 \cdot 0.3 \\ 0.145 \end{pmatrix} = - \begin{pmatrix} 0.0145 \\ 0.145 \\ 0.0435 \\ 0.145 \end{pmatrix}$$



$$L = \frac{1}{2} \cdot (y - \hat{y})^2$$

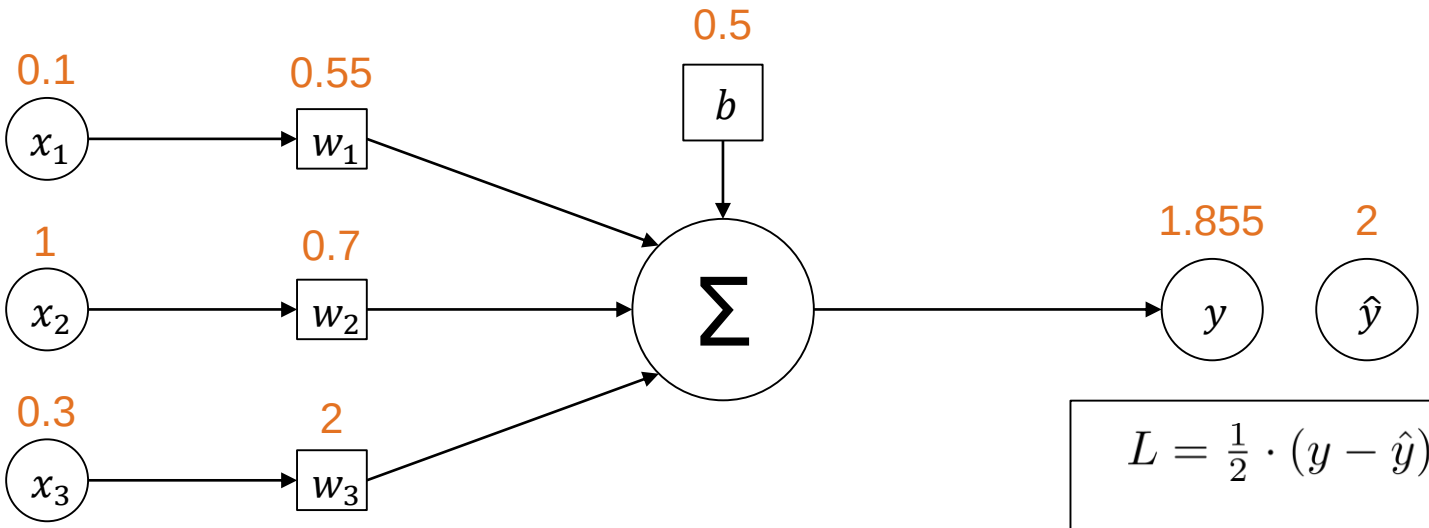
$$L = 0.01051125$$

Numerical Example – One Iteration

Towards Artificial Neurons

Gradient Descent

$$\nabla L = -\begin{pmatrix} 0.0145 \\ 0.145 \\ 0.0435 \\ 0.145 \end{pmatrix} \quad \alpha = 0.5, \vec{w_{new}} = \vec{w_{old}} - 0.5 \cdot \nabla L$$



Numerical Example – One Iteration

Towards Artificial Neurons

Gradient Descent

$$\nabla L = -\begin{pmatrix} 0.0145 \\ 0.145 \\ 0.0435 \\ 0.145 \end{pmatrix} \quad \alpha = 0.5, \vec{w_{new}} = \vec{w_{old}} - 0.5 \cdot \nabla L$$

$$\vec{w_{new}} = \vec{w_{old}} - 0.5 \cdot \nabla L$$

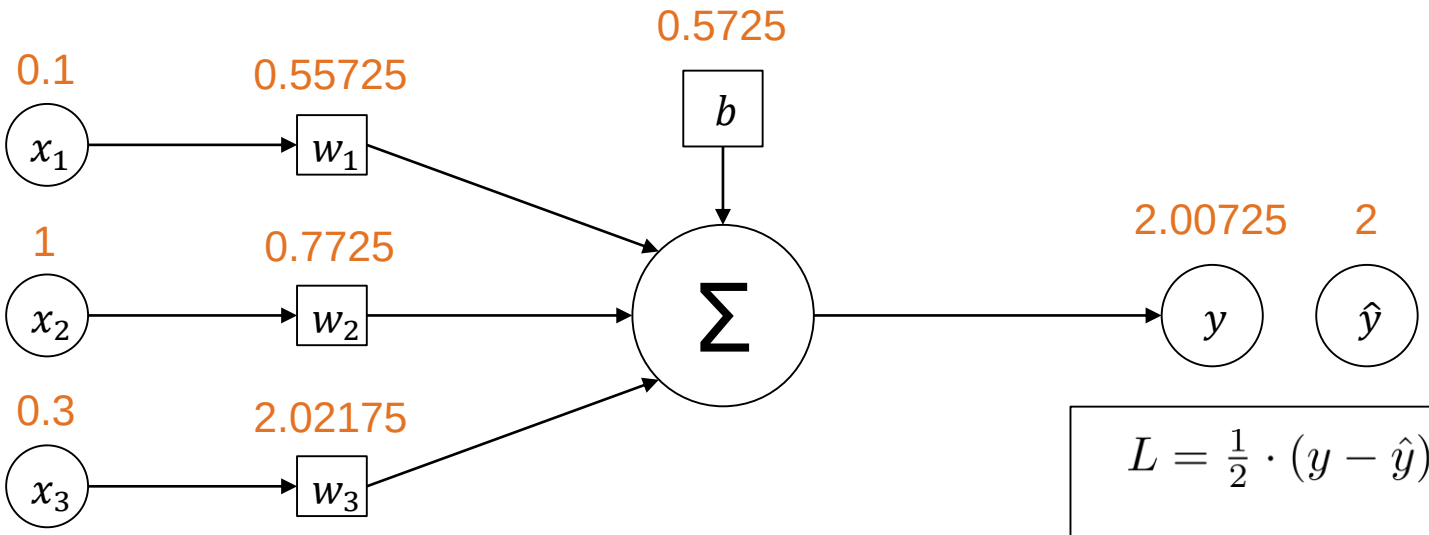
$$\begin{pmatrix} 0.55 \\ 0.7 \\ 2 \\ 0.5 \end{pmatrix} + 0.5 \cdot \begin{pmatrix} 0.0145 \\ 0.145 \\ 0.0435 \\ 0.145 \end{pmatrix} = \begin{pmatrix} 0.55725 \\ 0.7725 \\ 2.02175 \\ 0.5725 \end{pmatrix}$$

Numerical Example – One Iteration

Towards Artificial Neurons

Gradient Descent

$$\vec{w_{new}} = \begin{pmatrix} 0.55725 \\ 0.7725 \\ 2.02175 \\ 0.5725 \end{pmatrix}$$



$$L = \frac{1}{2} \cdot (y - \hat{y})^2$$

$$L = 0.00002628125$$

Numerical Example – One Iteration

Additional Slides

Although Gradient Descent works well in most scenarios, there are a number of pitfalls to be kept in mind when the optimization does not behave as expected. The next slides will introduce you to a couple of phenomena that may hinder the convergence of Gradient Descent. Most of these phenomena can be tackled by a proper adjustment of the learning rate or by advanced optimization methods enhancing Gradient Descent (e.g. Stochastic Gradient Descent or AdaGrad).

Towards Artificial Neurons

Gradient Descent

Stuck in Local Minimum:

$$\vec{w_{new}} = \vec{w_{old}} - \alpha \cdot \nabla L$$

$$L = w^4 - w^3 - 7w^2 + w + 20$$

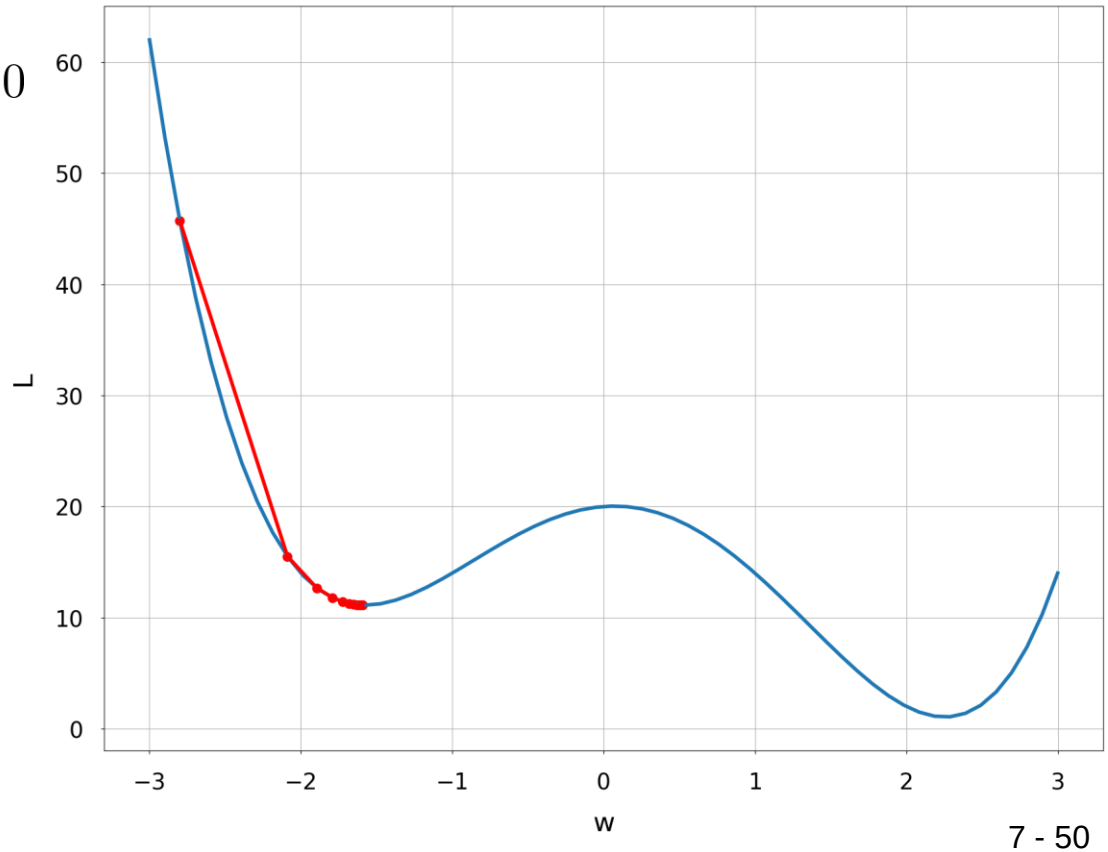
$$\nabla L = 4w^3 - 3w^2 - 14w + 1$$

$$\alpha = 0.01$$

$$w_0 = -2.8$$

$$\epsilon = 0.001$$

$$N = 10$$



Additional Slides

Gradient Descent can get stuck in local minima easily because the gradient of the function to be minimized is positive on one side of the local minimum and negative on the other. Additionally, the gradient around local minima is small, resulting in small weight adjustments. Therefore, Gradient Descent may jump around the local minimum in small steps without any progress regarding convergence.

Towards Artificial Neurons

Gradient Descent

Vanishing Gradient:

$$\vec{w_{new}} = \vec{w_{old}} - \alpha \cdot \nabla L$$

$L = w^3$

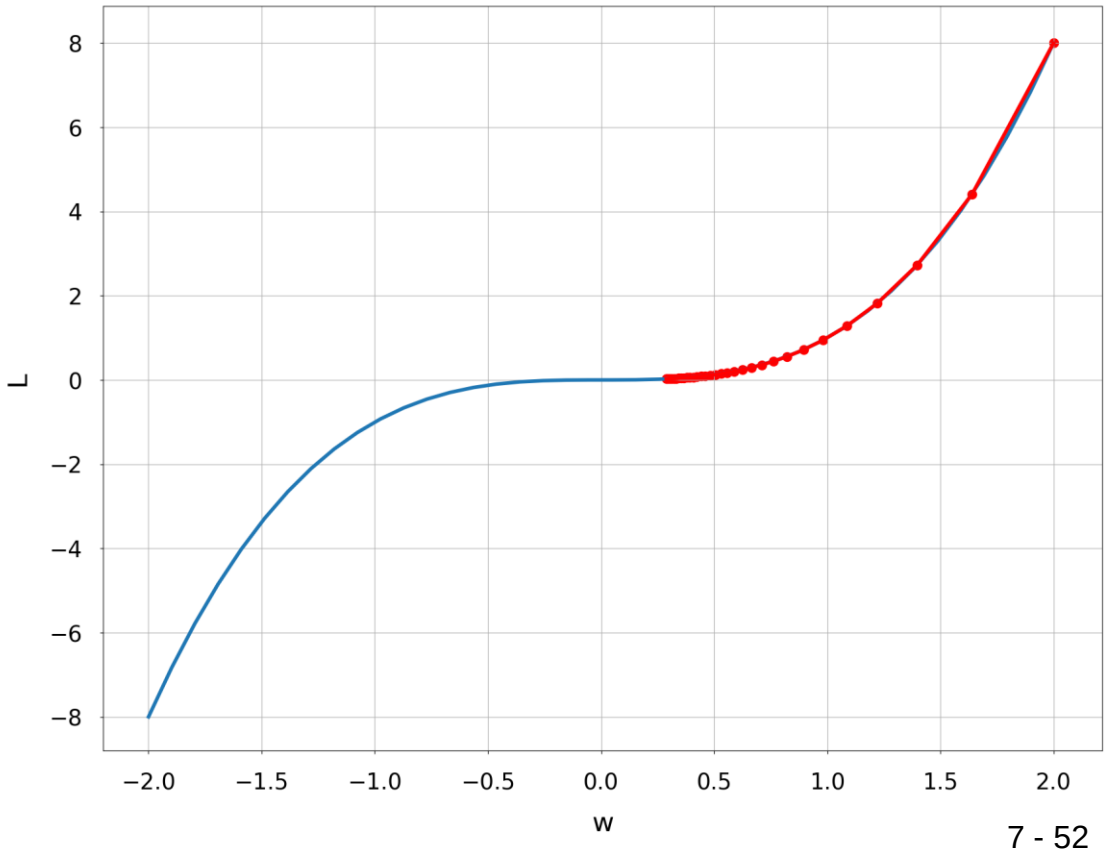
$\nabla L = \frac{dL}{dw} = 3w^2$

$\alpha = 0.03$

$w_0 = 2$

$\epsilon = 0.001$

$N = 30$



Additional Slides

Especially when the learning rate is small, Gradient Descent may get stuck on plateaus of the function to be minimized. This is due to the fact that on plateaus, the gradient of the function is small, hence weight updates are small, causing the next iteration point to be very close to the previous one. An increased learning rate may compensate for the slowing effect of the small slope.

Towards Artificial Neurons

Gradient Descent

Oscillating:

$$\vec{w_{new}} = \vec{w_{old}} - \alpha \cdot \nabla L$$

$$L = w^2$$

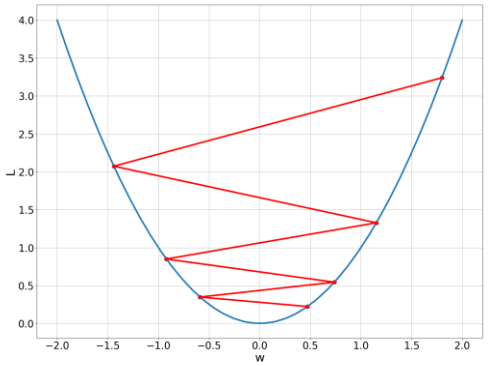
$$\nabla L = \frac{dL}{dw} = 2 \cdot w$$

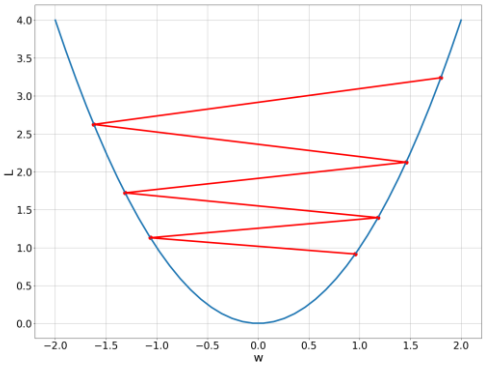
$$\alpha = [0.9, 0.95, 0.99, 1]$$

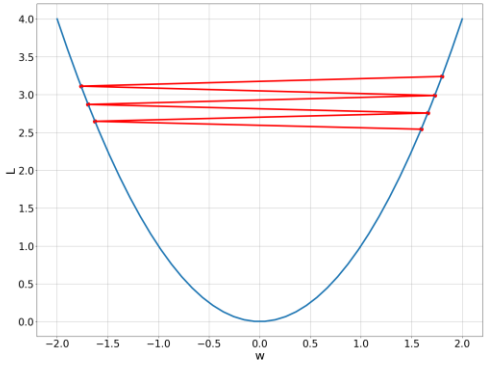
$$w_0 = 1.8$$

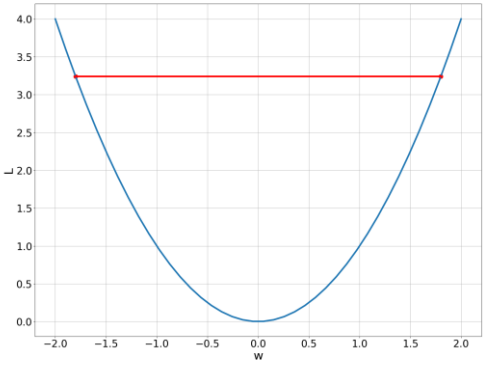
$$\epsilon = 0.001$$

$$N = 5$$









Additional Slides

Another effect that may hinder convergence of Gradient Descent is the phenomenon of oscillation. Especially in symmetric parts of the function, the iterations of Gradient Descent may jump between two mirror-inverted slopes with no or small progress.

Towards Artificial Neurons

Gradient Descent

Jumping out of Minima:

$$\vec{w_{new}} = \vec{w_{old}} - \alpha \cdot \nabla L$$

$$L = w^4 - w^3 - 7w^2 + w + 20$$

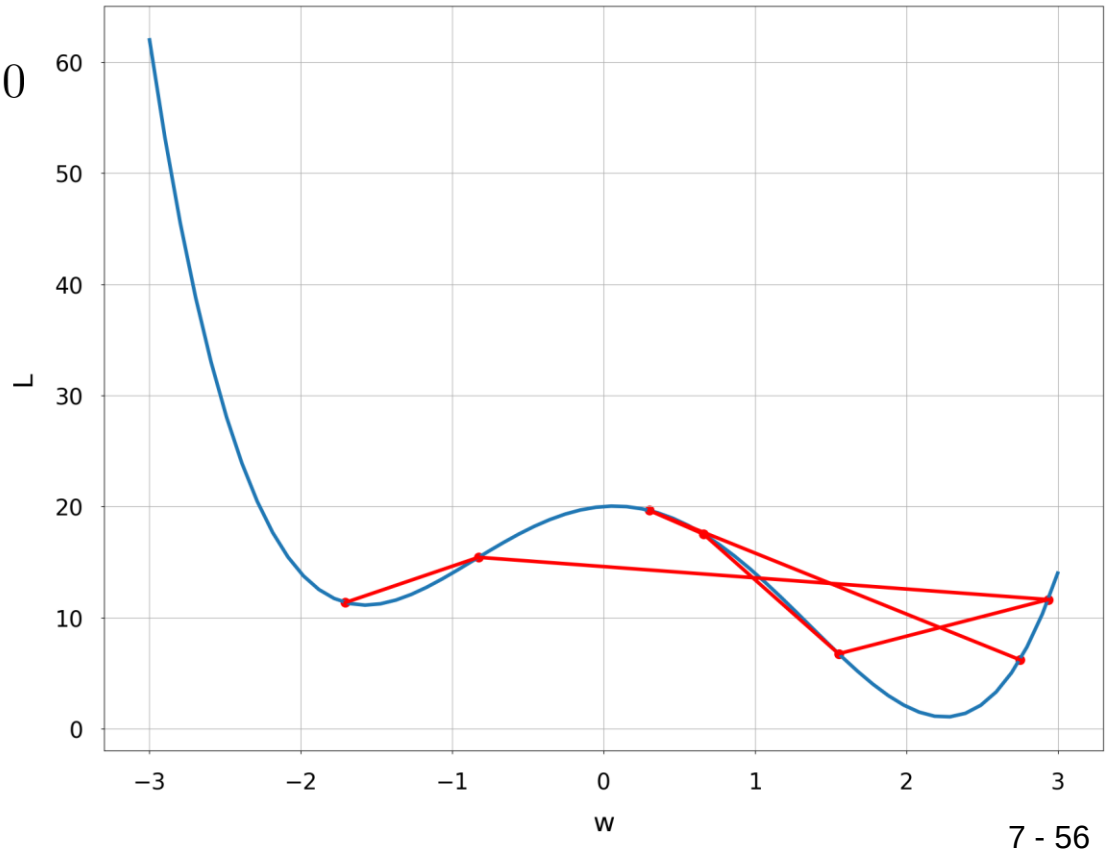
$$\nabla L = 4w^3 - 3w^2 - 14w + 1$$

$$\alpha = 0.1065$$

$$w_0 = 2.75$$

$$\epsilon = 0.001$$

$$N = 5$$



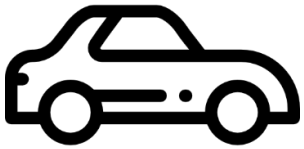
Additional Slides

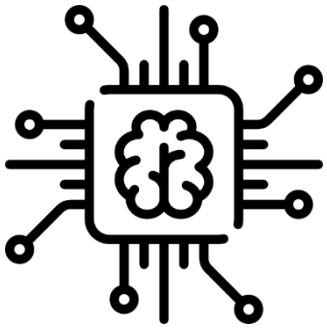
When learning rates are too large and/or minima quite shallow, Gradient Descent may jump over the ridge surrounding a minimum instead of converging to the minimum. In the example, you see how a Gradient Descent approach that started close to the global minimum of the function ultimately converges to a local minimum of the function.

Introduction: Artificial Neural Networks
Prof. Dr.-Ing. Markus Lienkamp

(Lennart Adenaw, M. Sc.)

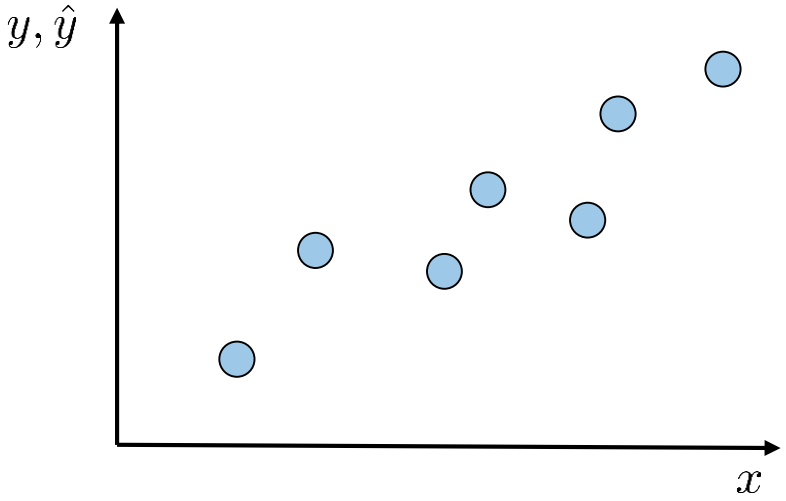
Agenda



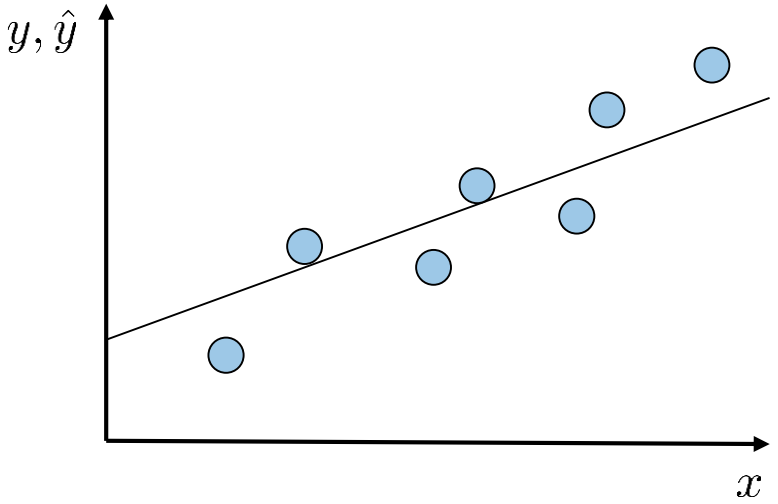


Towards Artificial Neurons

Wrap Up:



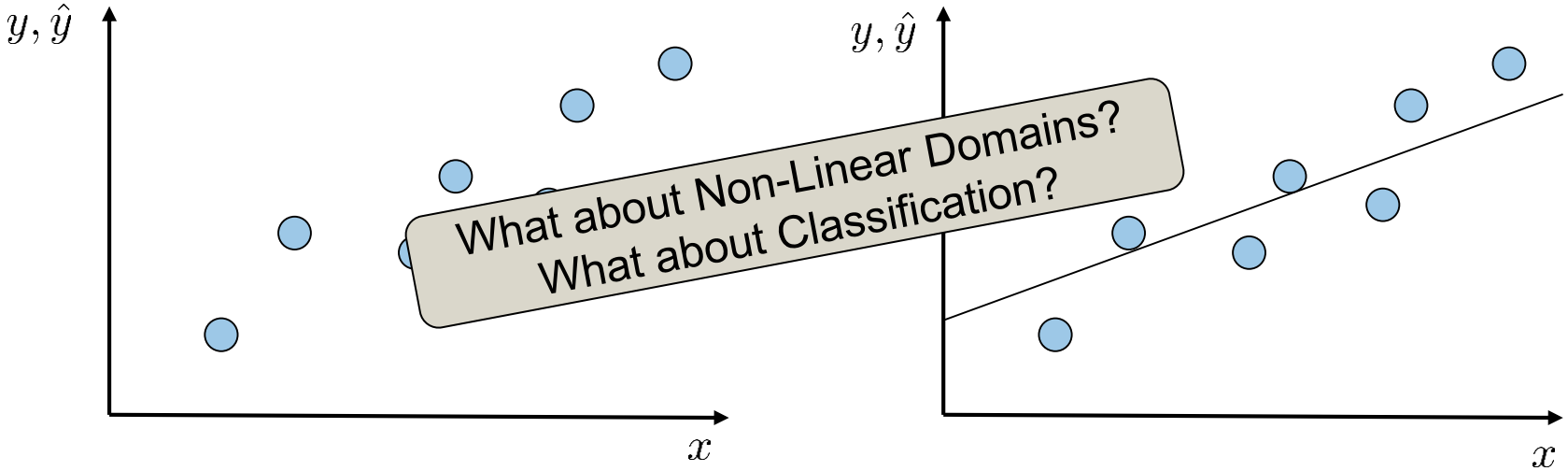
Input Data
Specific Observations of the Domain



Abstraction of the Domain
Regression

Towards Artificial Neurons

Wrap Up:



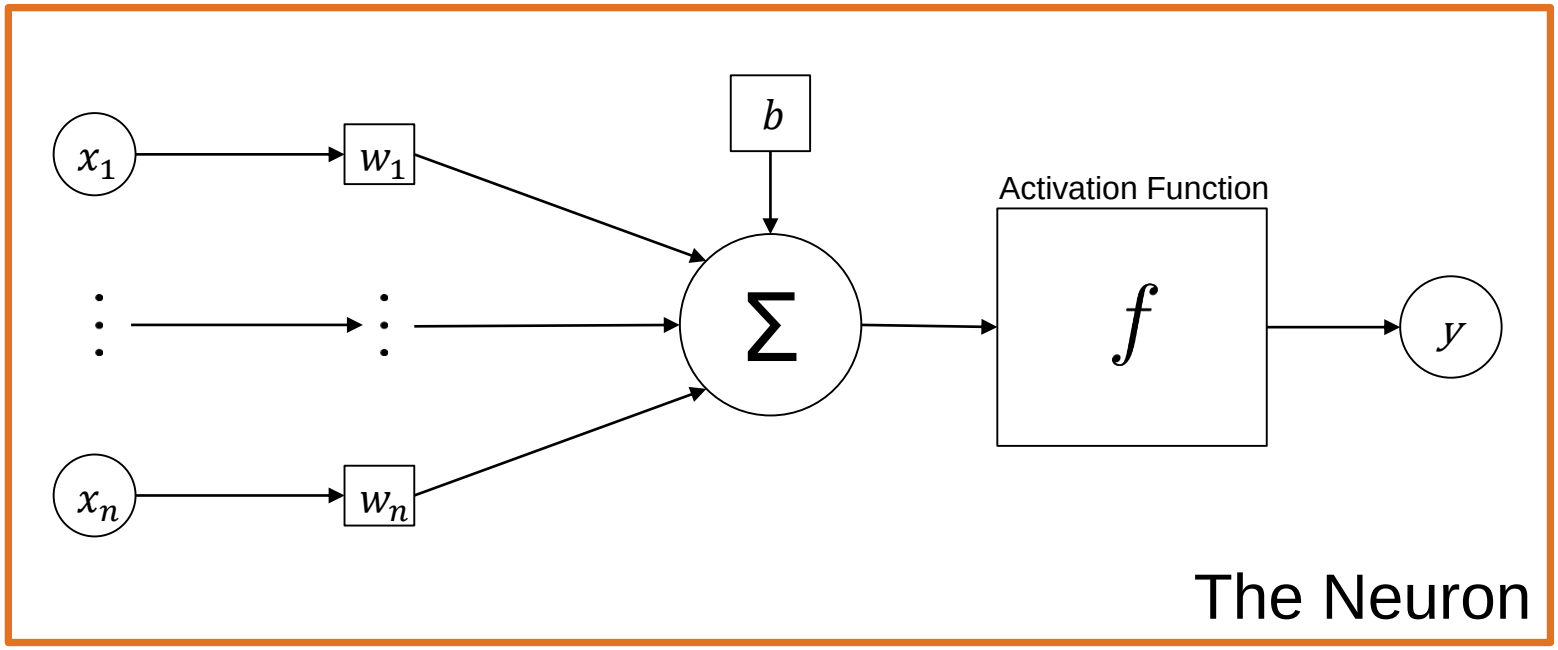
Input Data
Specific Observations of the Domain

Abstraction of the Domain
Regression

Towards Artificial Neurons

The Neuron

Introduction of an Activation Function:

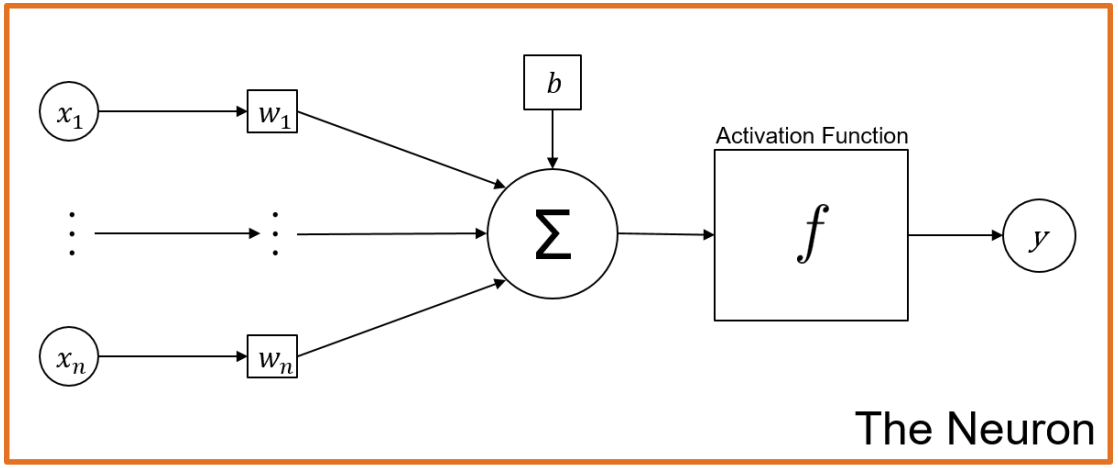


Towards Artificial Neurons

The Neuron

Properties:

- One Output
- One or many Inputs
- Bias b, Weights w
- Activation Function f
- $y = f(\sum_i w_i \cdot x_i + b)$



Additional Slides

As mentioned before, the road map of this lecture comprises the transition from a linear regression model to universal approximators, e.g. ANN. Up to this point, we mainly revised the contents of lecture 3 and used vocabulary and concepts from the domain of ANN to reinterpret linear regression models.

Furthermore, the concept of Gradient Descent has been introduced.

The question that remains is how these concepts can be applied to non-linear domains and how classification tasks can be solved by using a similar approach.

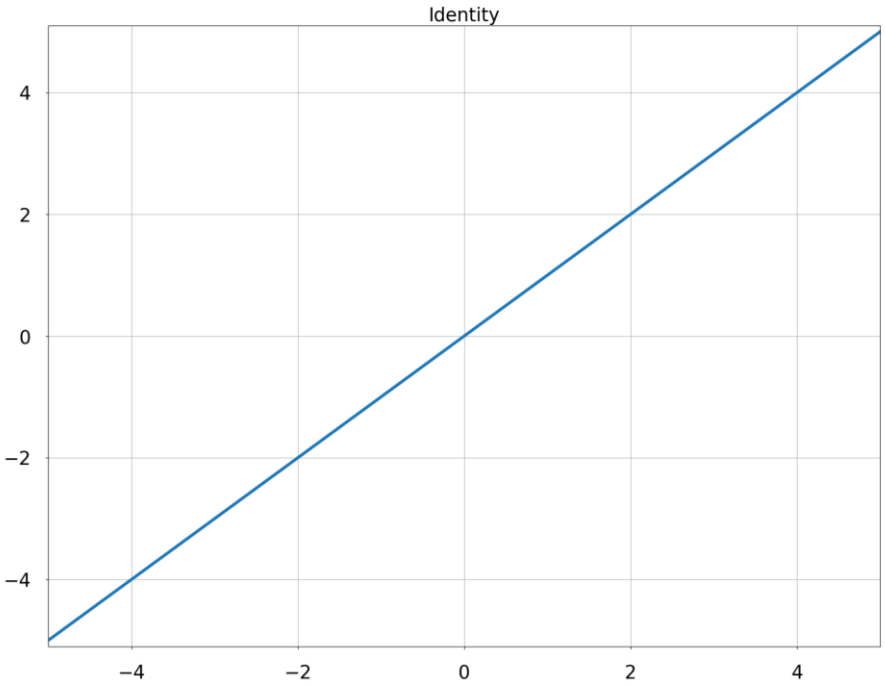
In order to enable approximation of non-linear domains, the Linear Regression model is augmented by a non-linear activation function. The resulting model is called an Artificial Neuron and serves as the base component of an ANN.
Artificial Neurons can be wired together to form arbitrarily large structures in which arrays of Neurons of the same hierarchy level are called „Layers“.

Each ANN has an „Input Layer“ where the data is fed into the network, a number of „Hidden Layers“ made up of artificial Neurons and an „Output Layer“ which contains the results of the forward pass.

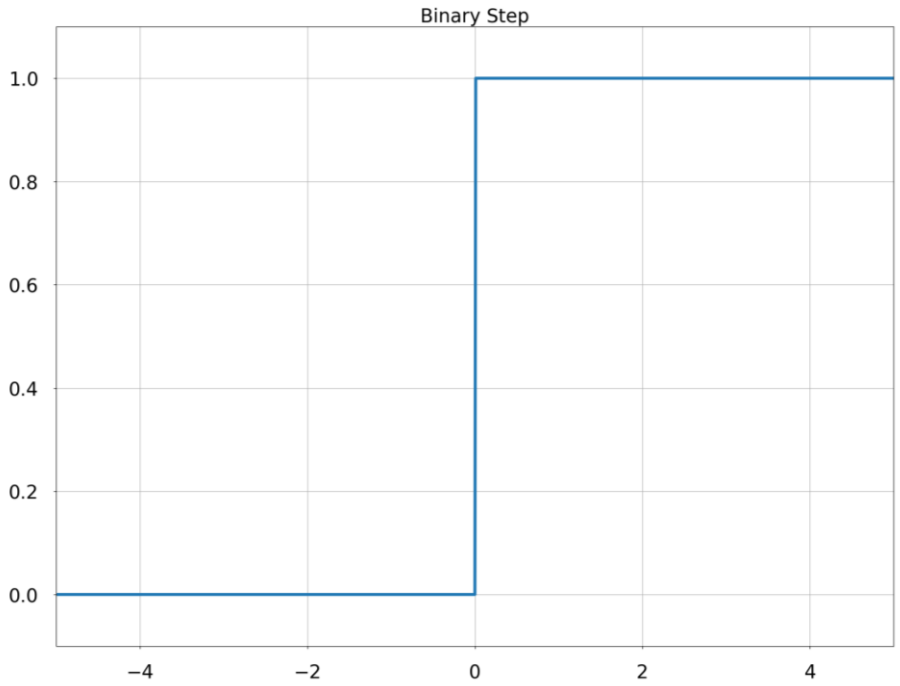
Towards Artificial Neurons

The Neuron

Linear Regression



Binary Classification



Additional Slides

The most simple activation functions are the identity function and the binary step. The identity function projects its input on itself. Thereby, it is effectively redundant. Hence, an artificial neuron with an identity activation function behaves identical to a linear regression model. Other linear activation functions may scale the neurons output. All linear activation functions including the identity function can be used in artificial neurons for regression tasks.

In contrast to linear activation functions, non-linear activation functions can be used to tackle non-linear tasks like classification.

Using a binary step activation function on an artificial neuron restricts the output of the neuron to two discrete values: 0 and 1. These values can be used to distinguish two classes. Therefore, a single artificial neuron with a binary step function can be used to determine whether a set of inputs belong to class 1 or class 0 if the right weights are applied to the neuron's inputs. This type of classification is also known as binary classification.

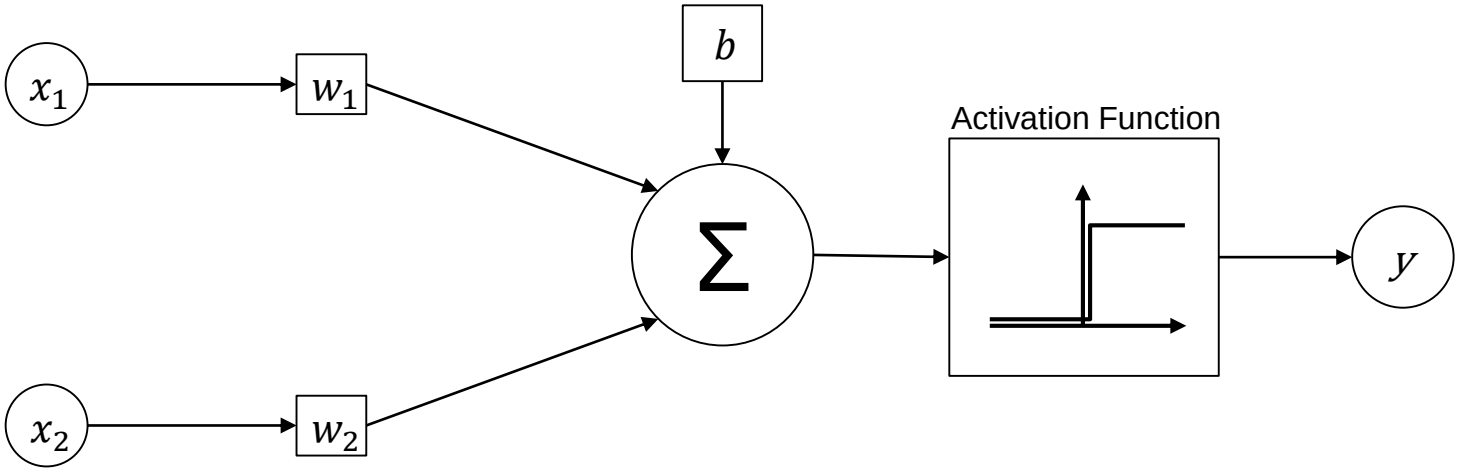
Example:

Let the inputs of an artificial neuron be the volume and the maximum velocity of a vehicle. Suppose that the artificial neuron has a binary step function. It is then possible to determine a set of weights w1, w2, b for which the neuron behaves in such a way that it outputs a 0 for each value combination belonging to a passenger car and a 1 for each value combination characteristic for a bus.

Towards Artificial Neurons

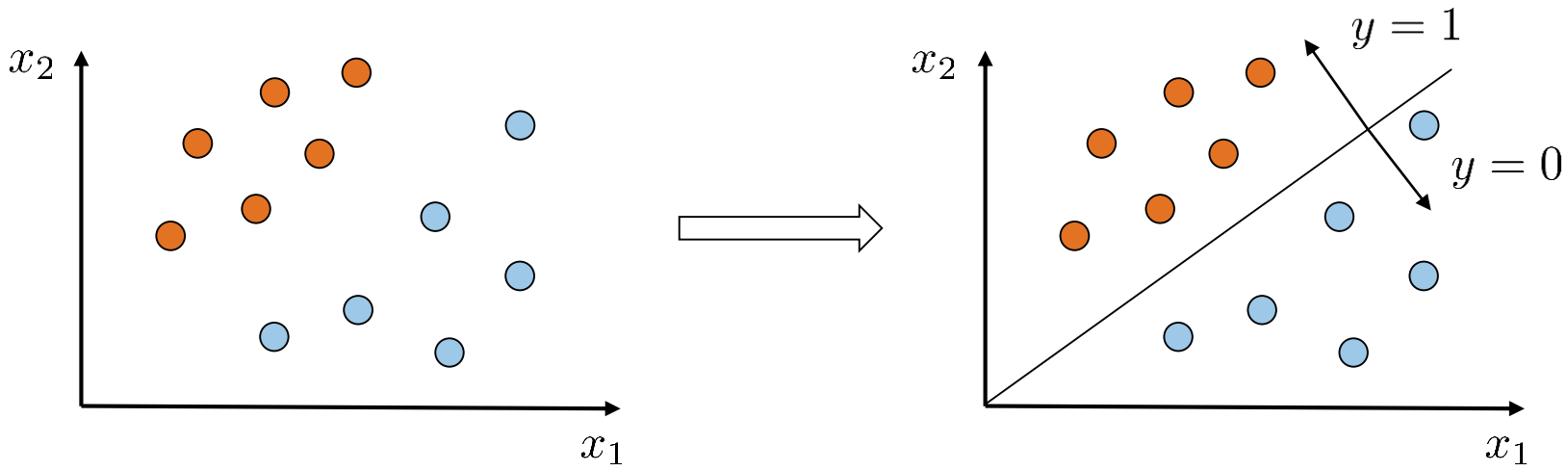
The Neuron

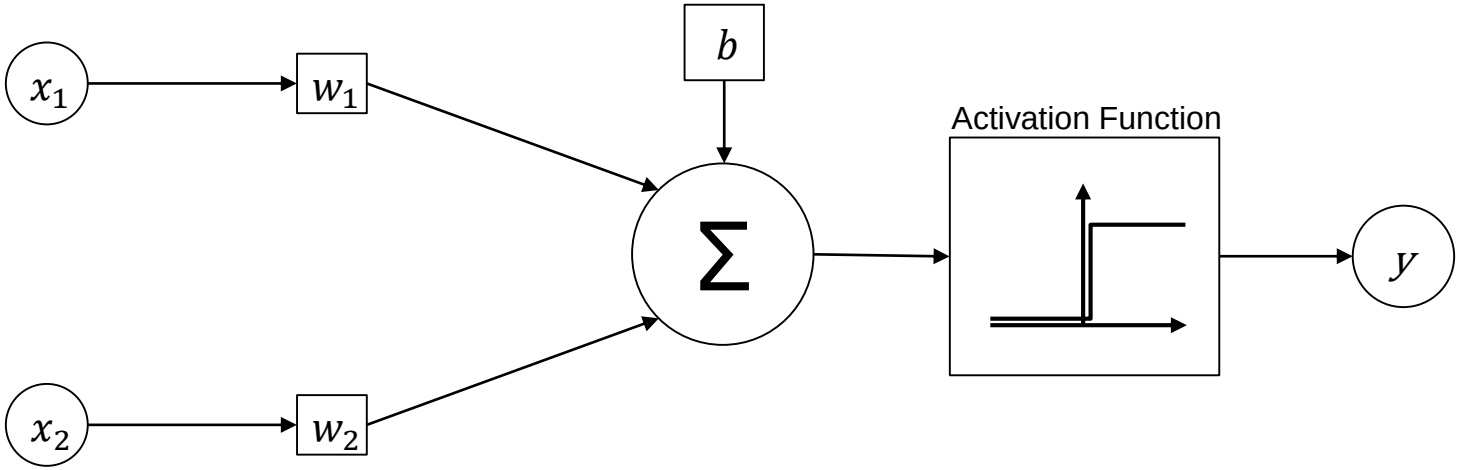
Suppose the following Setup:



Towards Artificial Neurons

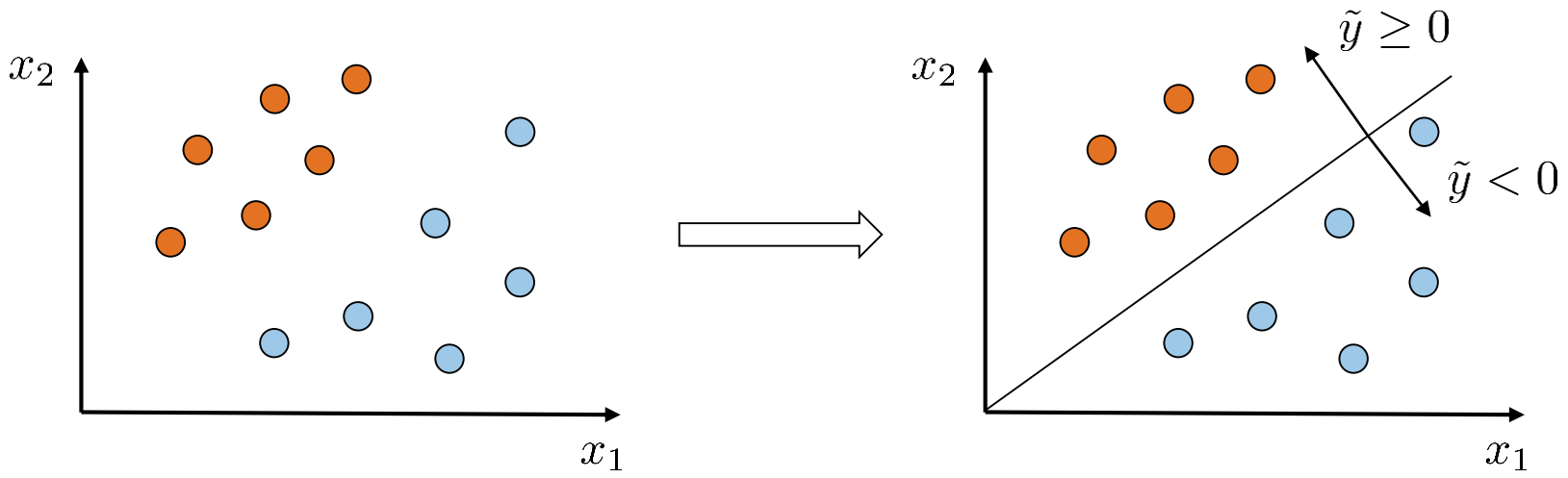
The Neuron

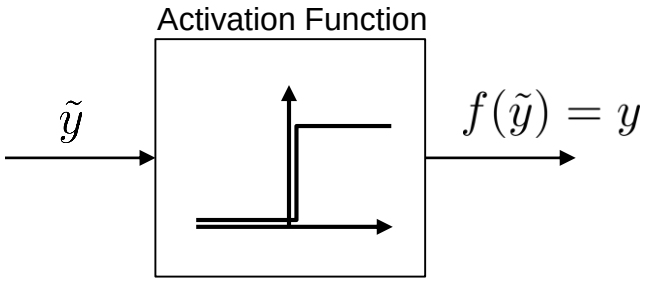




Towards Artificial Neurons

The Neuron



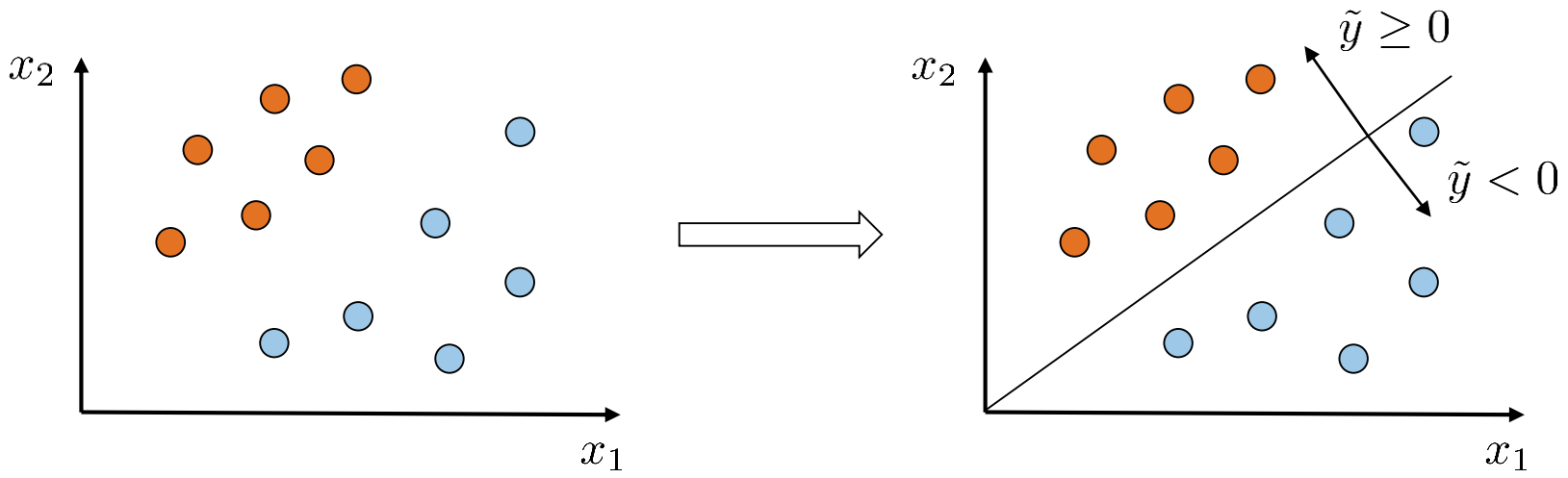


$$f = \begin{cases} 1 & \tilde{y} \geq 0 \\ 0 & \tilde{y} < 0 \end{cases}$$

Step Function

Towards Artificial Neurons

The Neuron



Activation Fu

$$\frac{df}{d\tilde{y}} = \begin{cases} 0 & \tilde{y} > 0 \\ \infty & \tilde{y} = 0 \\ 0 & \tilde{y} < 0 \end{cases}$$

$$f = \begin{cases} 1 & \tilde{y} \geq 0 \\ 0 & \tilde{y} < 0 \end{cases}$$

Step Function

Additional Slides

To derive an Artificial Neuron from the Linear Regression model, an activation function was added. In order to perform a simple binary classification task with only one artificial neuron, a „Threshold“ or „Step Function“ is used as the activation function. The idea behind this is to generate an output which is either 1 or 0 depending on the class the input is coming from. For linear separable classes, it is certain that a specific combination of weights will ensure a correct classification of two distinct classes.

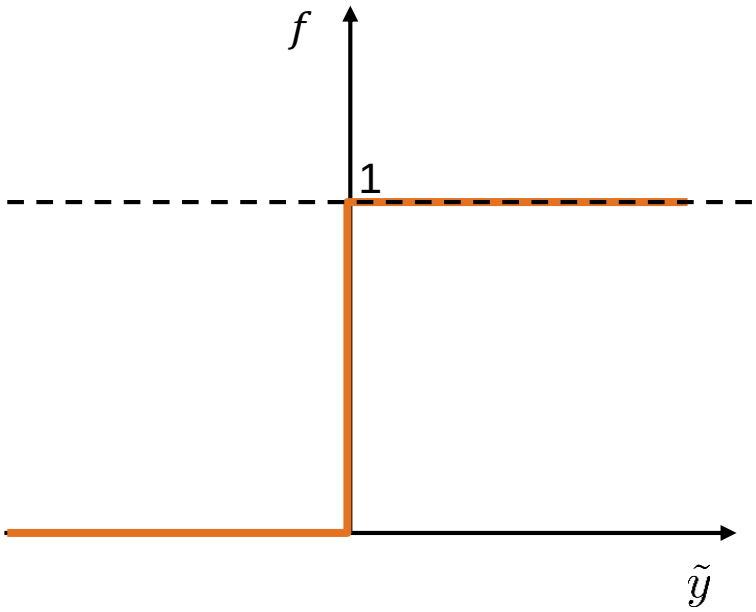
Nevertheless, the problem with the binary step is that gradient descent fails to converge toward a reasonable set of weights, because the derivative of the loss function will be either 0 or infinite through the introduction of a binary step activation function. Hence, a binary step can be used to solve binary classification problems with a single Artificial Neuron, but it is not possible to train the weights of the neuron using Gradient Descent.

To circumvent this problem, a continuously differentiable function that approximates the binary step can be used. In order to use Gradient Descent together with this new activation function, the new function’s gradient shall never be 0 or infinite. A function meeting these requirements is the sigmoid function.

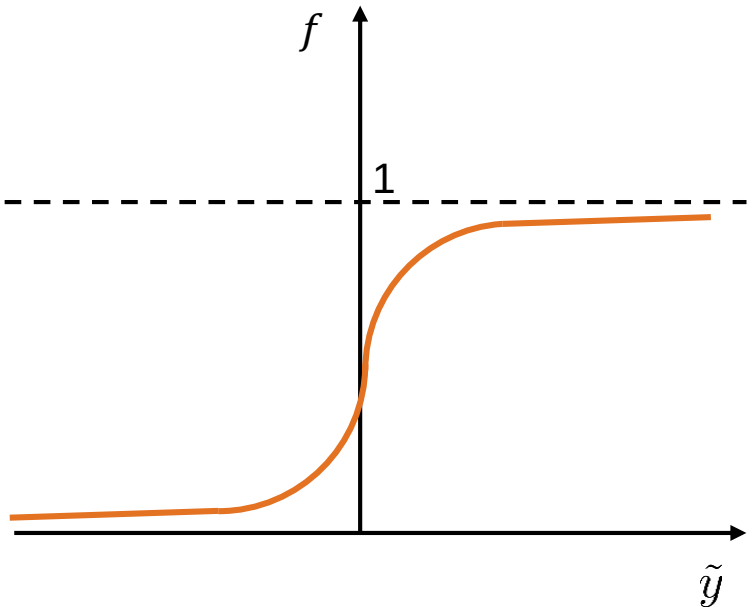
Towards Artificial Neurons

The Neuron

Step Function



Sigmoid Function



Towards Artificial Neurons

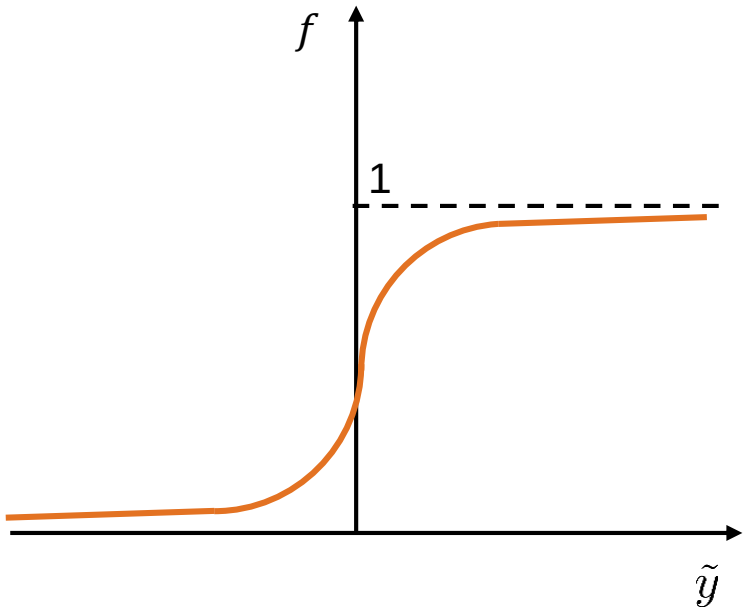
The Neuron

$$f = \frac{1}{1+e^{-\tilde{y}}} = \frac{e^{\tilde{y}}}{1+e^{\tilde{y}}}$$

$$\frac{df}{d\tilde{y}} = f \cdot (1 - f)$$

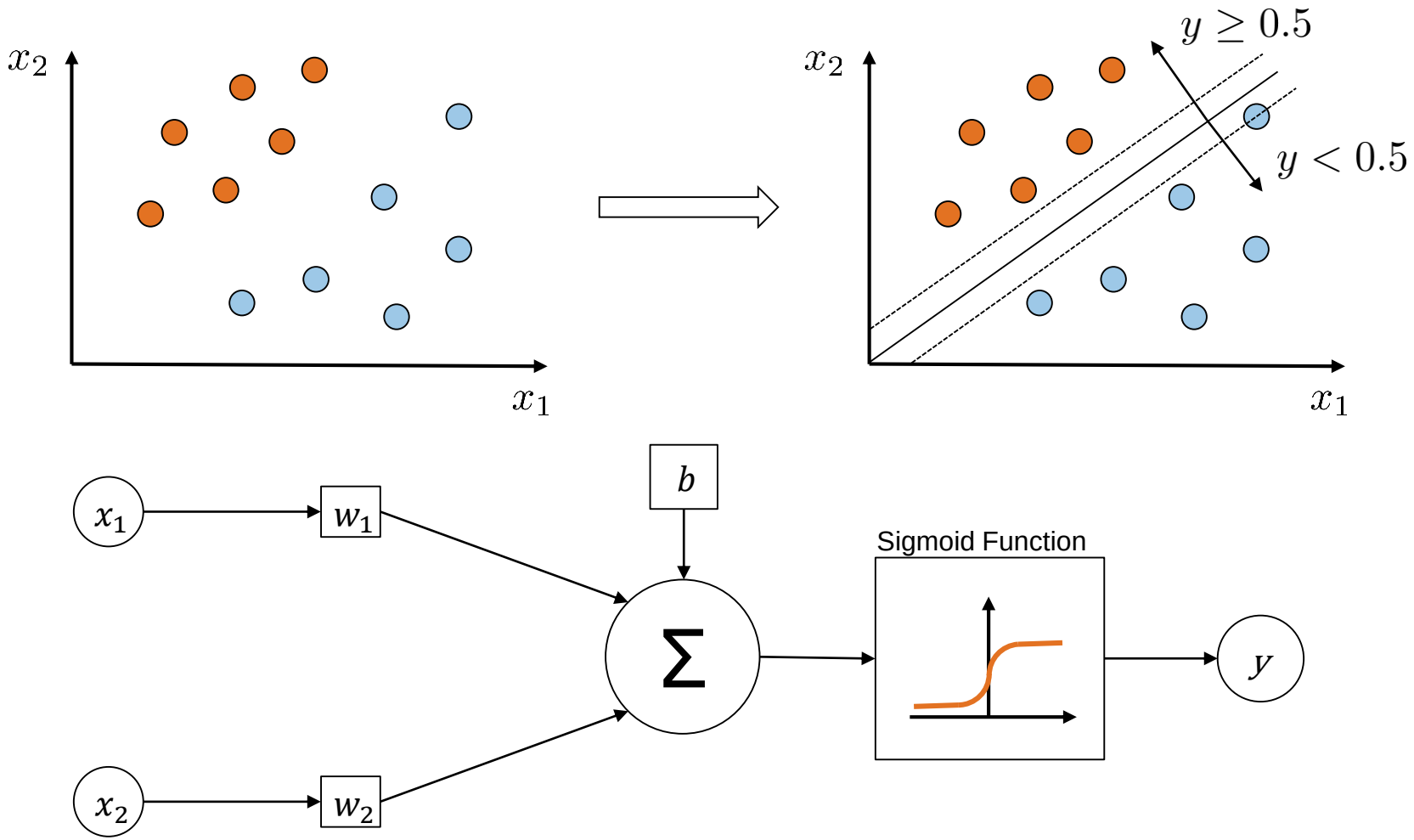
Continuously Differentiable!

Sigmoid Function



Towards Artificial Neurons

The Neuron



Additional Slides

The sigmoid function – just like the binary step – is well suited for binary classification because its output values are mostly close to 1 or close to 0. Nevertheless, it is continuously differentiable in every point. Therefore, artificial neurons that have a sigmoid activation function are trainable using Gradient Descent, because the gradient of the Loss function can be calculated and each component of the gradient is a real number.

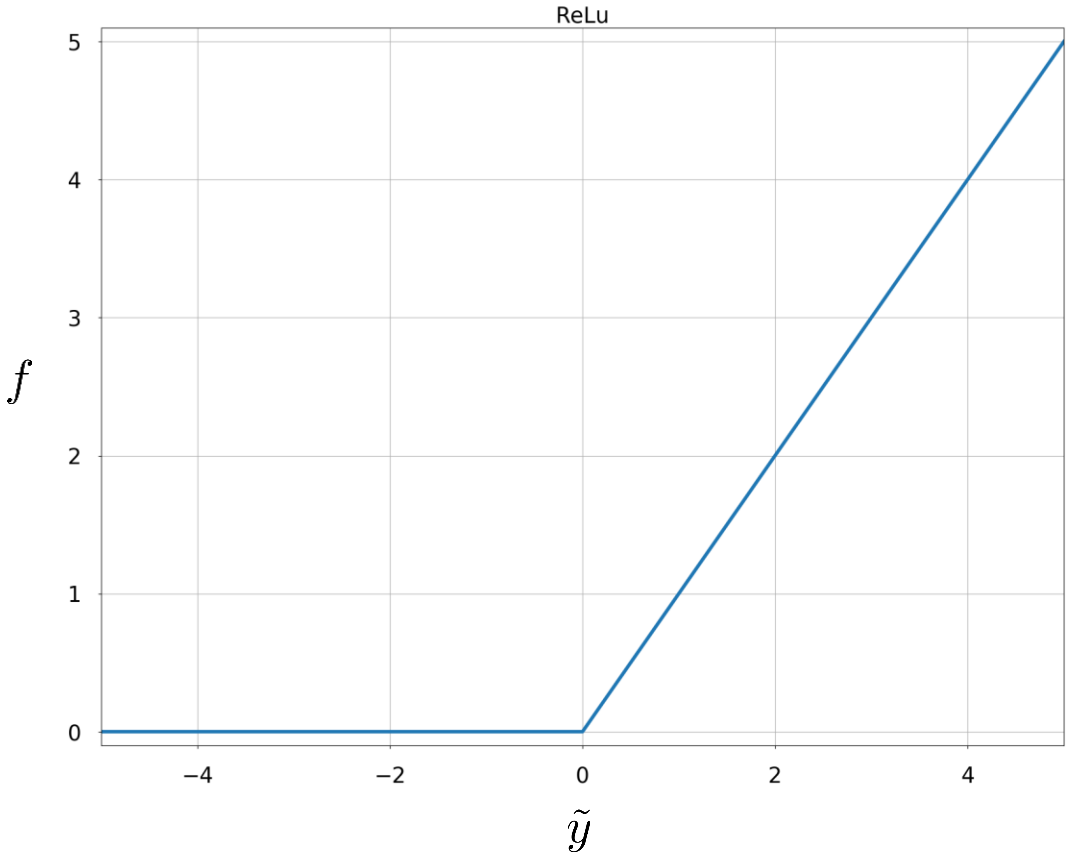
The downside of the sigmoid function in comparison to the binary step, however, is the fact that it introduces fuzziness into the binary classification task because the output values of the artificial neuron are not binary but real numbers on the interval]0,1[. Depending on the application, this behavior can be favorable, because it is a measure for how probable a classification result is. If this behavior is not desired in a specific application, it can be circumvented by training the neuron using a sigmoid function only and introducing a binary step or threshold into the calculation when using the neuron for classification.

Note that the derivative of the sigmoid function is a function of the sigmoid function itself. Hence, the derivative at every point of the function can easily be determined with the function value in the same point. This is not necessary for neuron training, but computationally efficient when using Gradient Descent together with the sigmoid function.

Towards Artificial Neurons

The Neuron

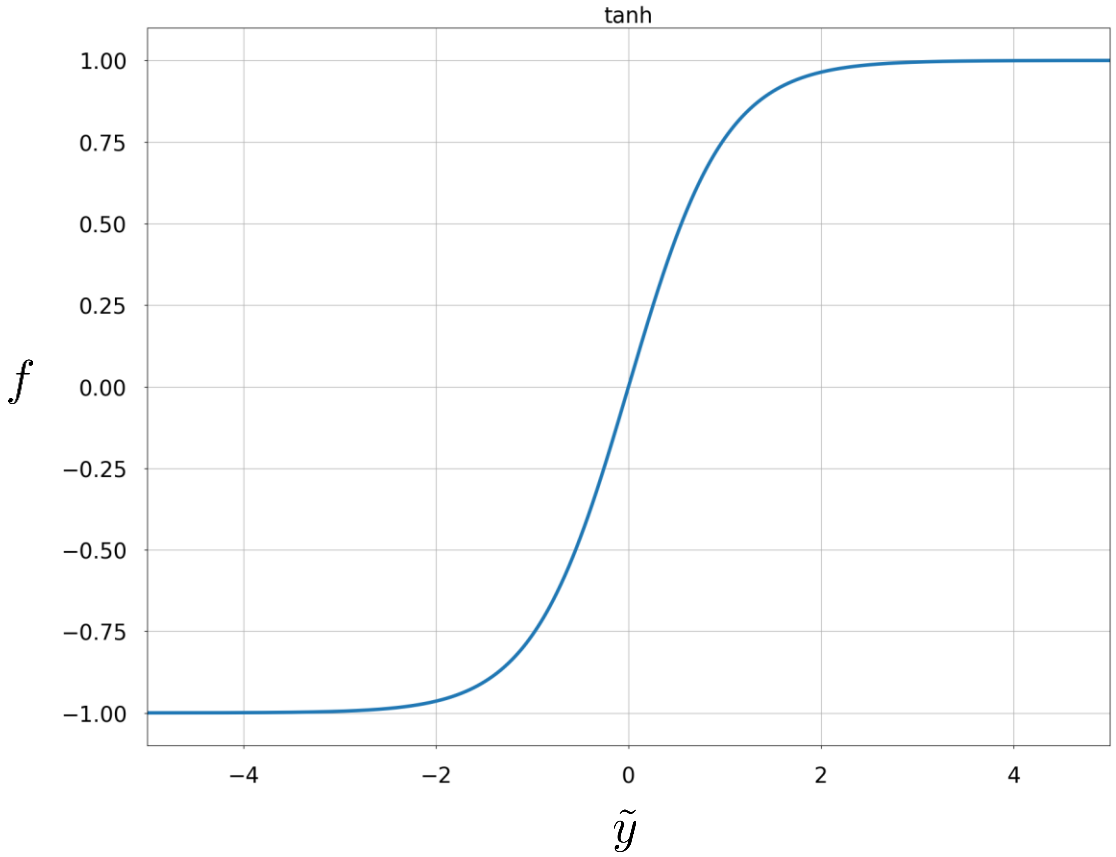
$$f = \begin{cases} \tilde{y} & \tilde{y} \geq 0 \\ 0 & \tilde{y} < 0 \end{cases}$$



Towards Artificial Neurons

The Neuron

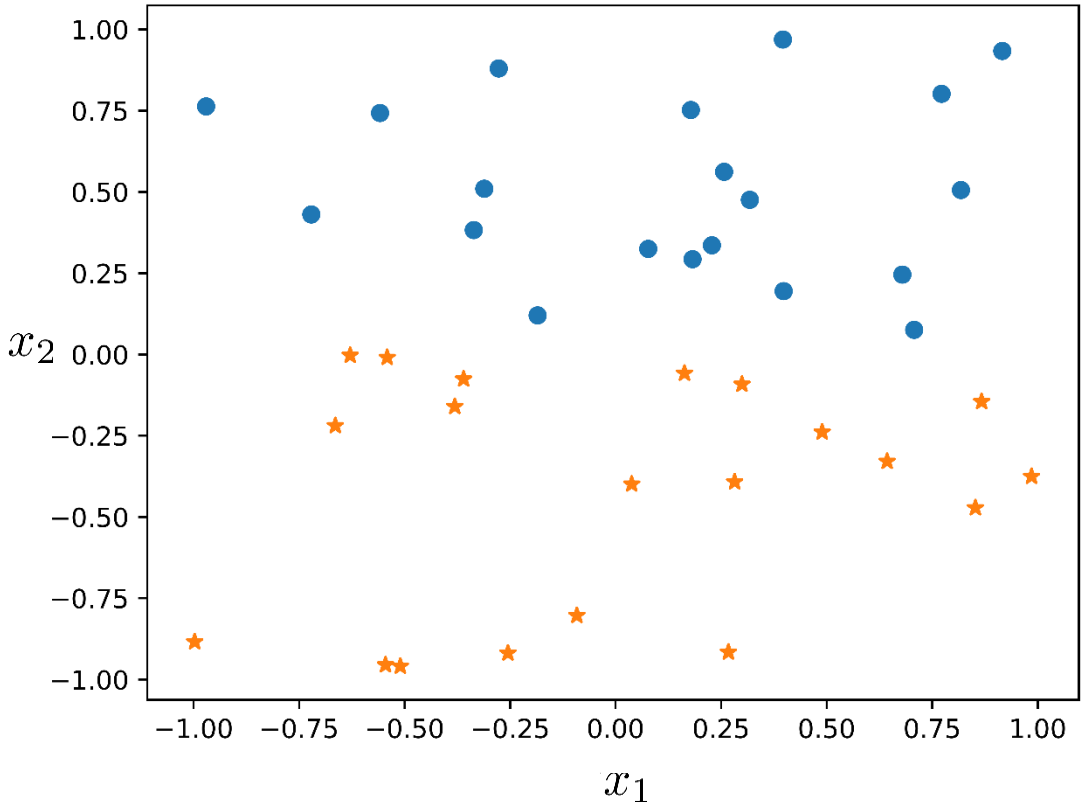
$$f = \tanh(\tilde{y})$$



Towards Artificial Neurons

The Neuron

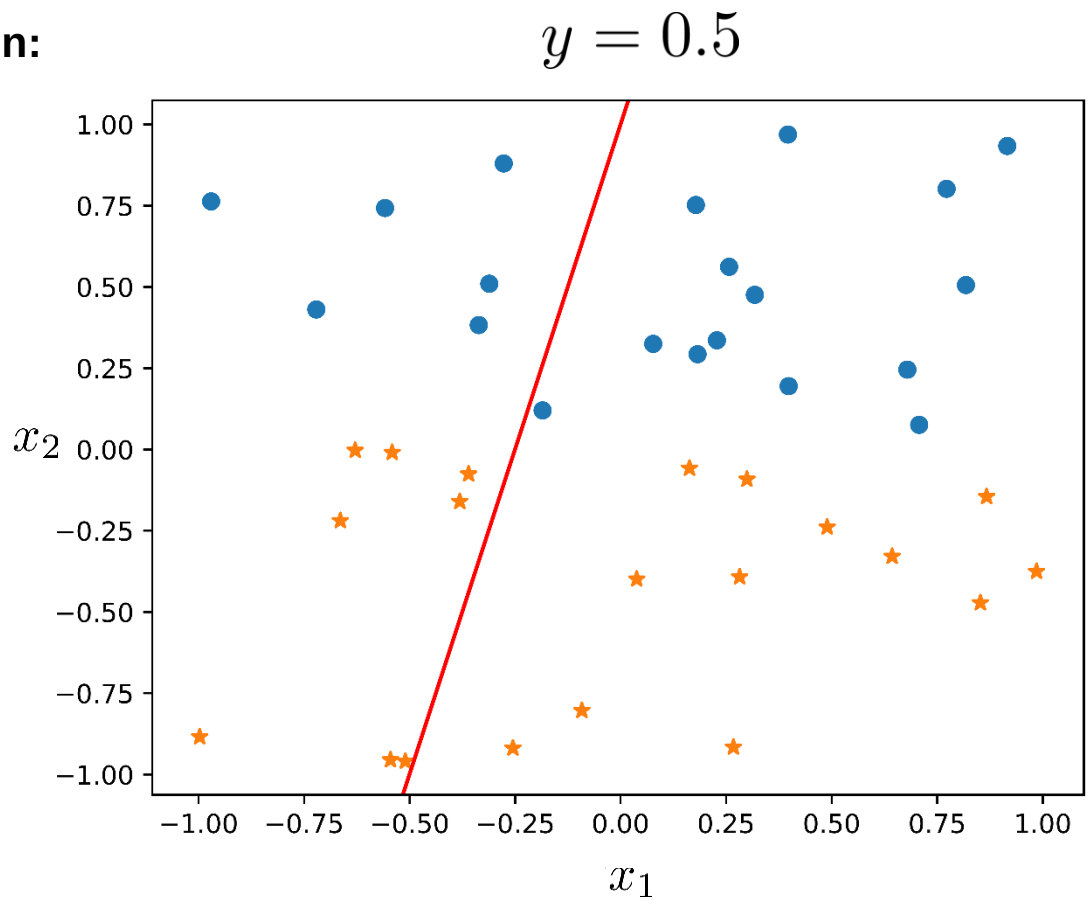
Input Data:



Towards Artificial Neurons

The Neuron

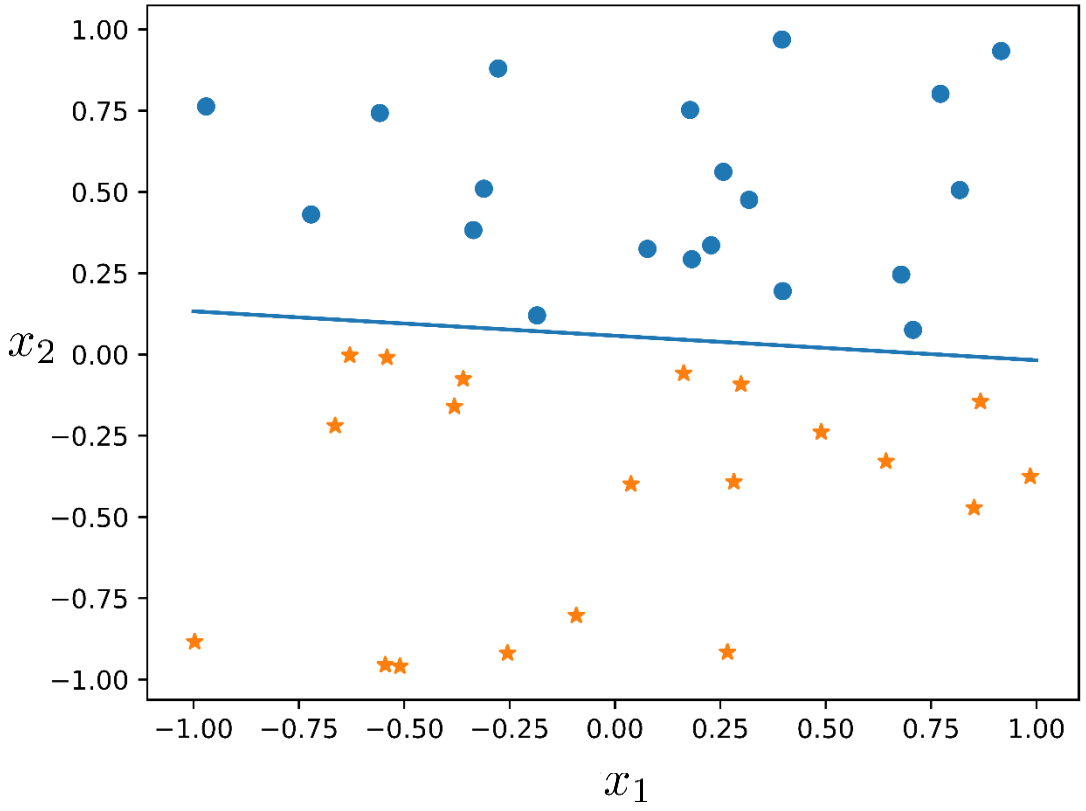
Initialization:



Towards Artificial Neurons

The Neuron

After Training:



$y = 0.5$

Additional Slides

The example shows a binary classification task on a data set with two input variables x_1 and x_2. Consequentially, the artificial neuron used for this task has two inputs, two weights, one bias and a sigmoid activation function.

The first figure shows the data points in the training data set. The data points in the training data set are labeled and the two classes (blue dots and orange stars) are clearly clustered into distinguishable groups in the input data space.

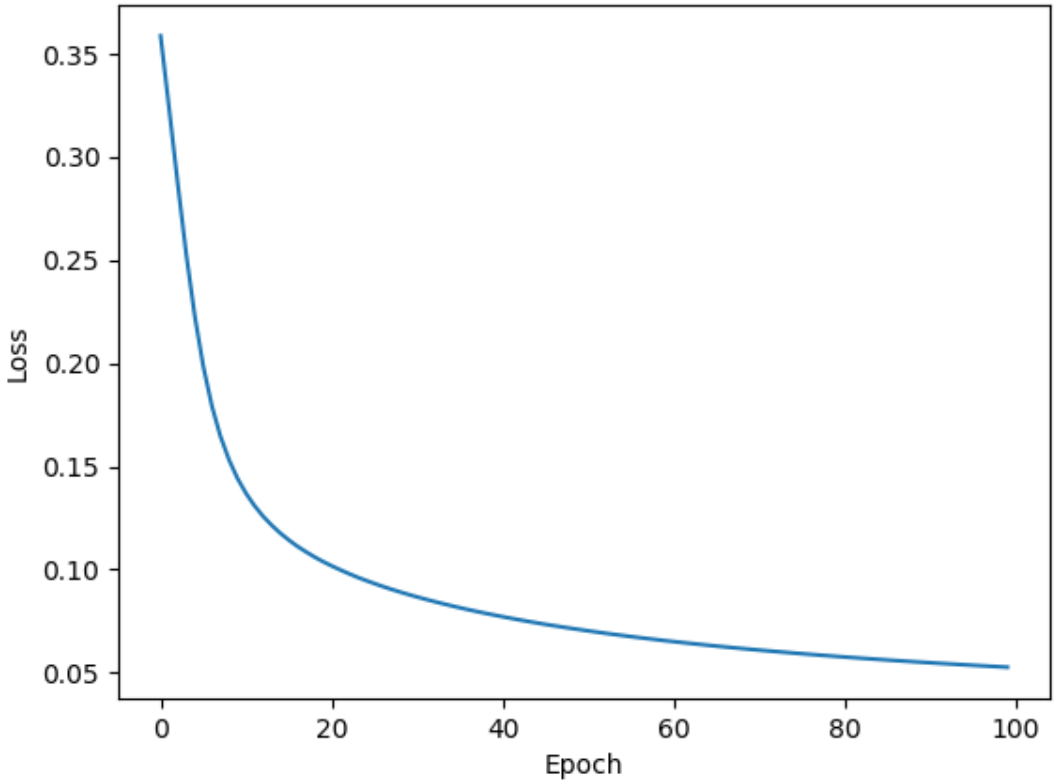
During training, the neuron‘s weights are updated by inputting the input data set and using Gradient Descent to calculate the weight update. Initially, the weights are set to random values.

The second and third figure show the line on which the artificial neuron outputs 0.5, which is the class separation line. As you can see, the line does not separate the two classes very well after initialization. After training, however, a clear separation of the classes is achieved.

Towards Artificial Neurons

The Neuron

Loss History:



Epoch:

An epoch has passed when all training vectors have been used once to update the weights.

Additional Slides

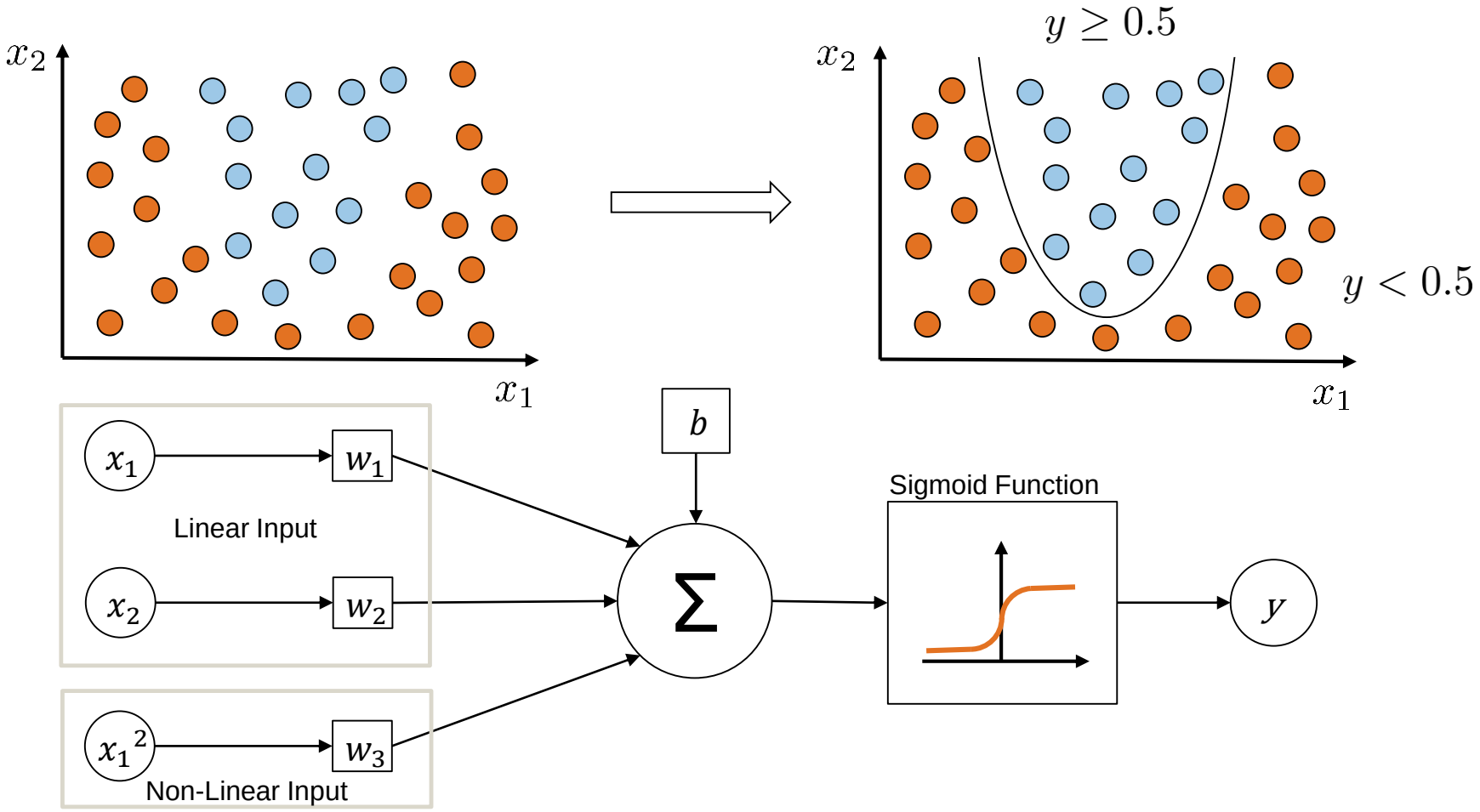
The graph shows the Loss history of the example training procedure. In it, the average Loss for each epoch is plotted. An epoch is defined to have passed when all training vectors have been used for weight update once. Therefore, epochs are a measure of the number of training iterations.

It is evident that for this example the Loss diminishes with a growing number of inputs. The maximum Loss occurs at the start of the training when the weights are still close to their initial values. This situation corresponds to the odd separation line of example figure 1, in which the two classes are not separated well. As training progresses, the Loss of each epoch decreases because the separation of the classes improves. Visually, the separation line in this example rotates and translates until it separates the classes well.

Note that – for this example – the Loss is never exactly 0 because the Loss function used for Gradient Descent is based on the difference between the neuron‘s output and a binary class label (0/1). Since the neuron does have a sigmoid activation function, this difference will never be exactly 0.

Towards Artificial Neurons

The Neuron



Additional Slides

Although the sigmoid function makes it possible to train artificial neurons to solve classification problems, it does not help with the regression or classification in non-linear domains. This is due to the fact that the activation function mainly manipulates the output of the neuron in a way that introduces a threshold between the classes to be distinguished. The inputs to the activation function, however, are strictly linear due to the nature of the linear weights and inputs. It is, therefore, not even possible to distinguish two classes that are separated by a parabola in a two dimensional input space.

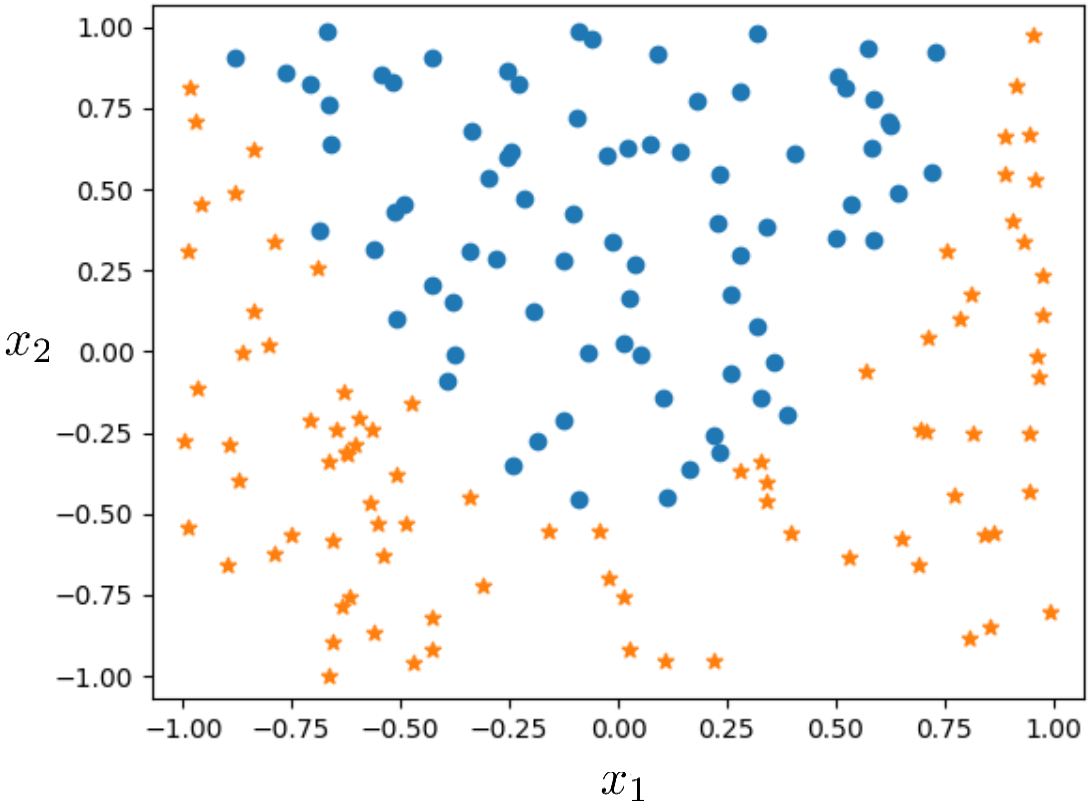
An idea to overcome this shortcoming is to introduce non-linearities into the inputs of the artificial neuron. In case of a quadratic class separation line, it is feasible to use the same neuron as in the previous example and add another input to it. The input is then chosen to be the square of the input variable that is supposed to have a quadratic relation to the expected separation line between the classes.

The following slides are an example of this approach.

Towards Artificial Neurons

The Neuron

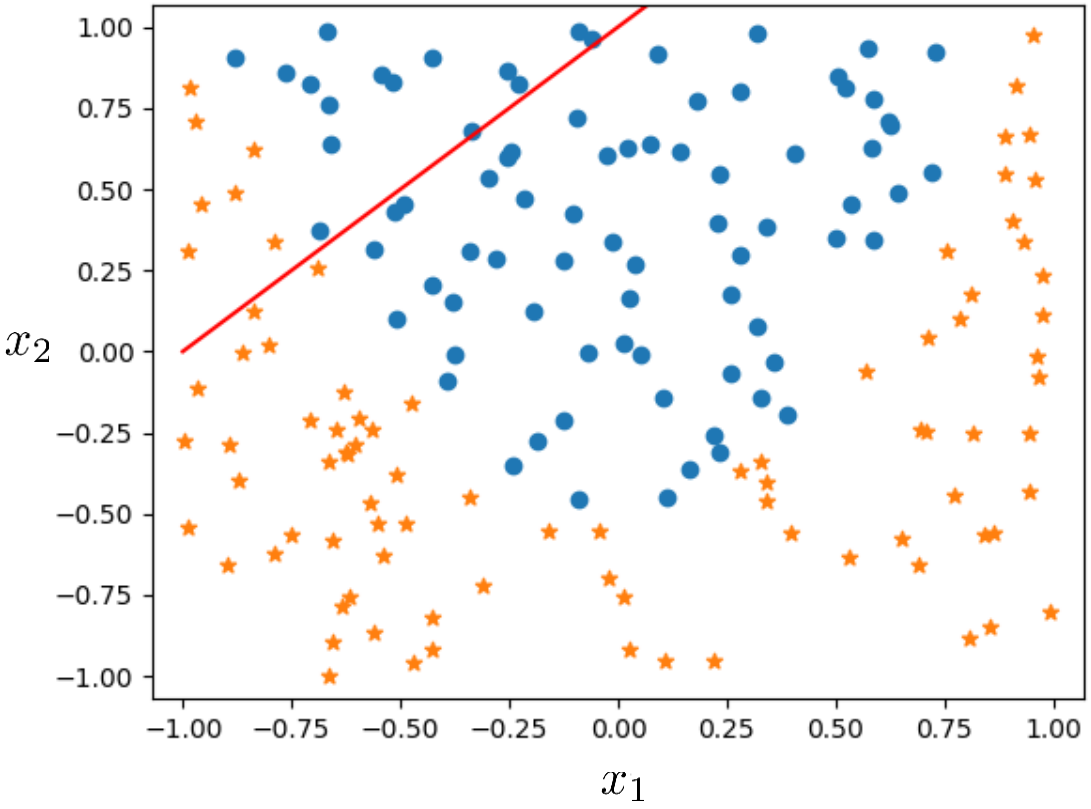
Input Data:



Towards Artificial Neurons

The Neuron

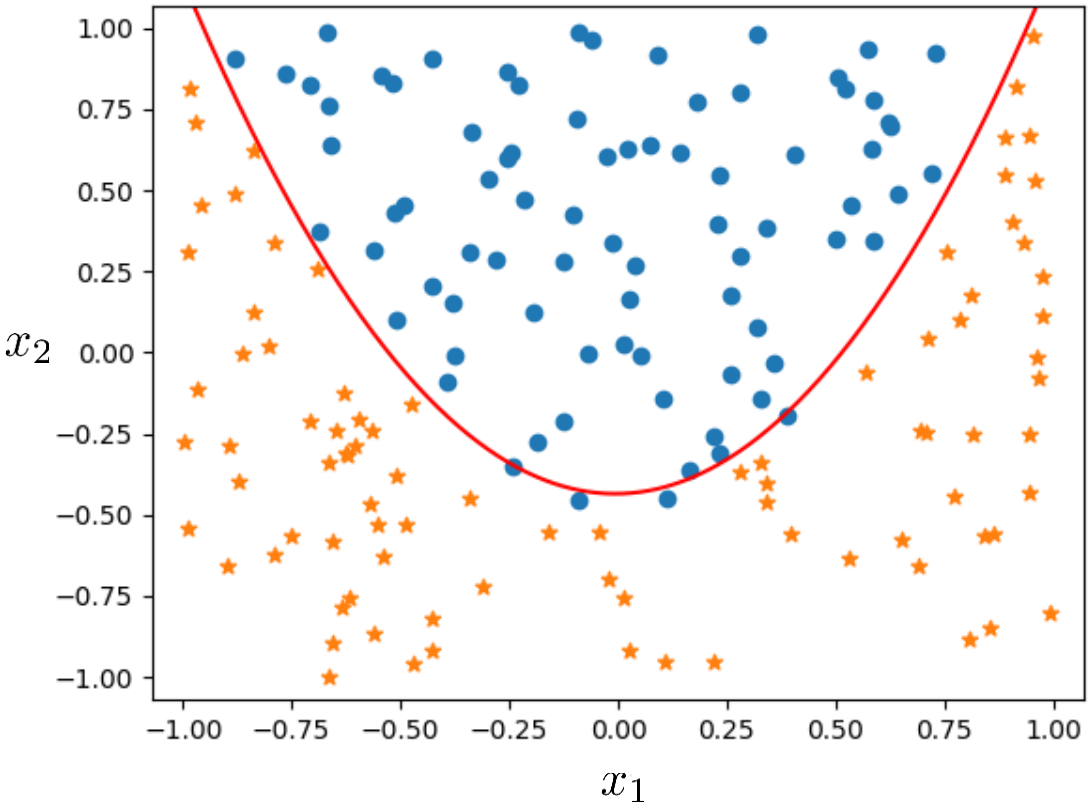
Initialization:



Towards Artificial Neurons

The Neuron

After Training :



Additional Slides

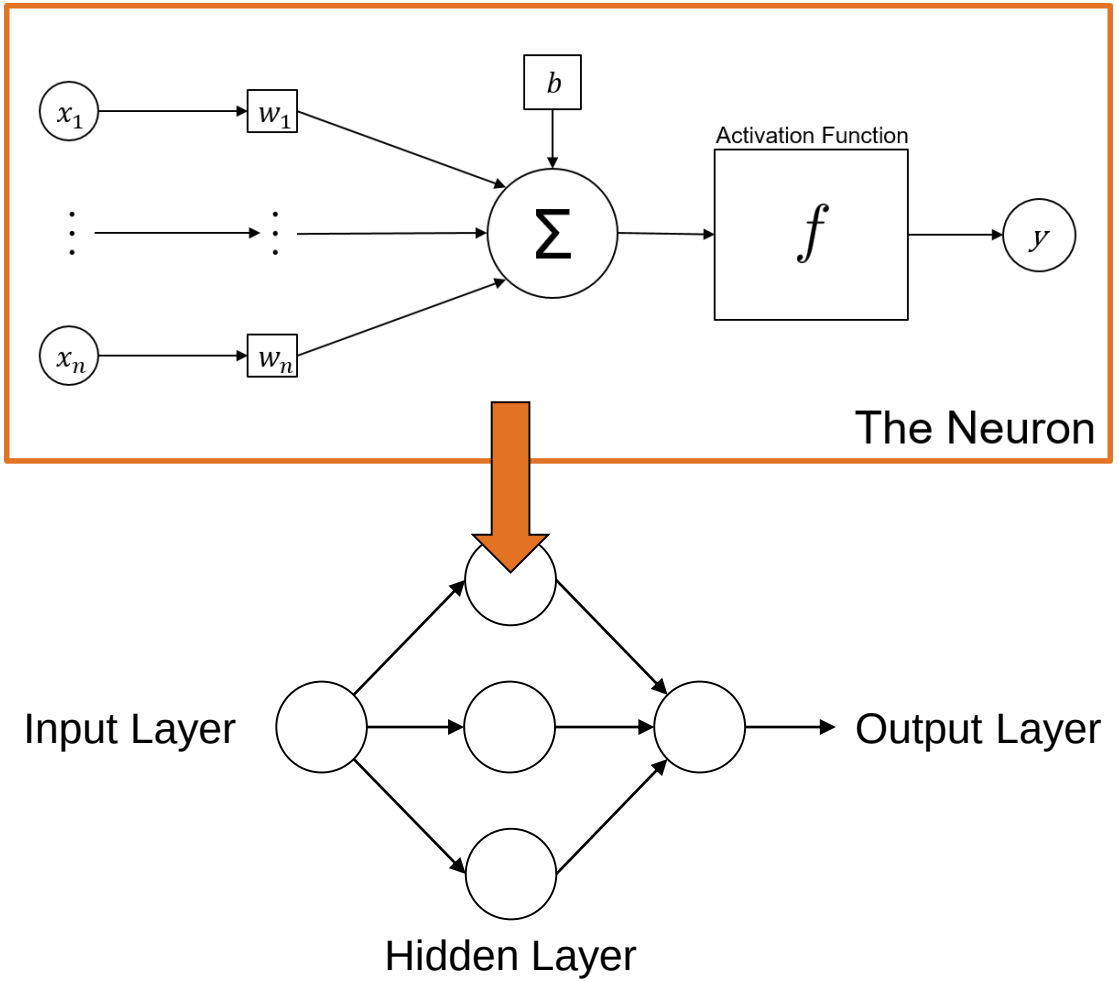
In the example, the weights of the artificial neuron are successfully trained to define a non-linear (i.e. quadratic) separation line between the classes. This indicates that it is possible to use this approach for classification and regression in non-linear domains.

Unfortunately, in the context of machine learning, this observation does not do us any good. The sole reason why this approach worked for the quadratic example is that we explicitly modeled the quadratic relation in the input data space, meaning that for any non-linear relation we would need to be able to tell which form of non-linearity is present in the data.

If we knew about the relation a priori, we would not need machine learning. Therefore, the introduced approach defeats the purpose of ANN altogether.

Towards Artificial Neurons

The Neuron



Additional Slides

The solution to the challenge of non-linear domains is the chaining of several artificial neurons with linear input. It has been shown that any analytic function can be approximated by a finite number of neurons up to an arbitrary accuracy. The corresponding theorem is known as the „Universal Approximation Theorem”. The connection of artificial neurons to larger structures is called an artificial neural network.

Unfortunately, the mathematical proof of this concept is beyond the scope of this lecture. Nevertheless, we will be able to demonstrate the practical usability of ANN in multiple examples.

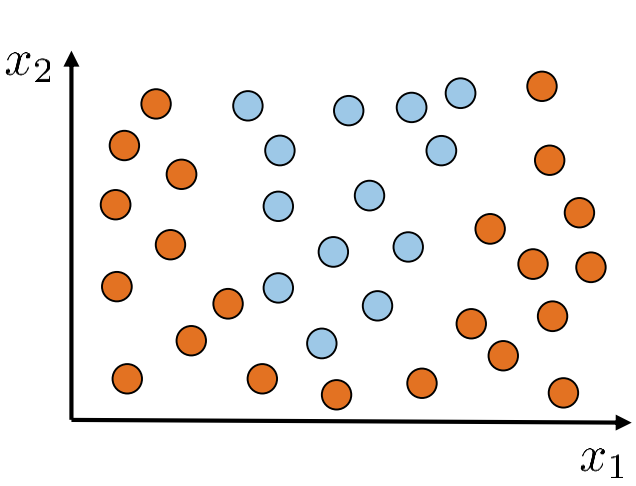
ANN are built layer by layer. Following an input layer that is used to feed the input data to the network, several neurons are stacked into a layer. An arbitrary number of layers can be queued after the input layer. The outputs of the neurons in a previous layer are then the inputs of neurons in the next layer. Results of the ANN are passed to its environment via an output layer which is essentially a layer with as many neurons as the desired result has dimensions. All layers between the input and the output layer are called hidden layers.

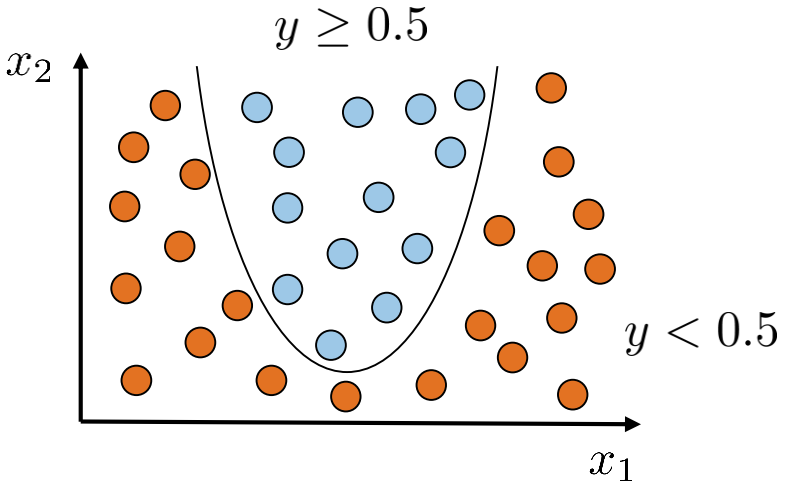
Training of an ANN also works using Gradient Descent and a method called Backpropagation which you will learn about in the next lecture.

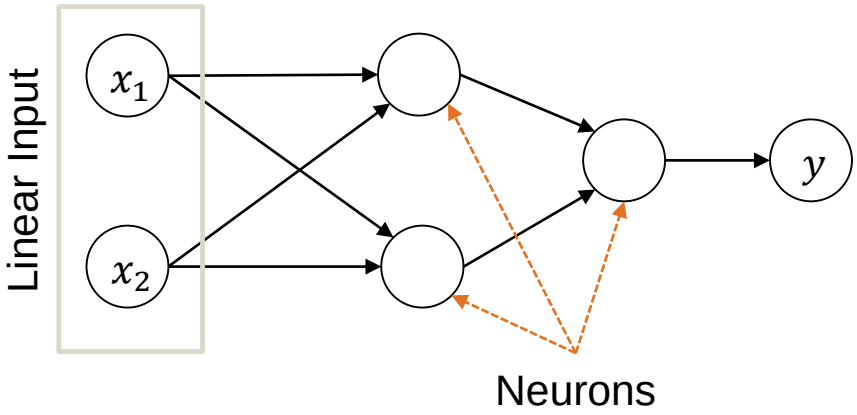
When describing an ANN, some pieces of information are regularly used to characterize the net.

Towards Artificial Neurons

The Neuron







Net Properties:

Loss Function: Mean Squared Error

Activation Function: Sigmoid / Linear

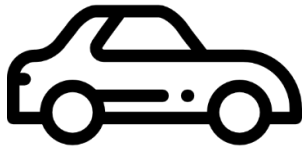
Optimizer: Gradient Descent

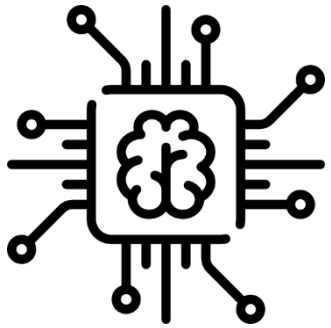
Architecture: Number of Layers, Nodes per Layer, (Type of Layer)

Introduction: Artificial Neural Networks
Prof. Dr.-Ing. Markus Lienkamp

(Lennart Adenaw, M. Sc.)

Agenda





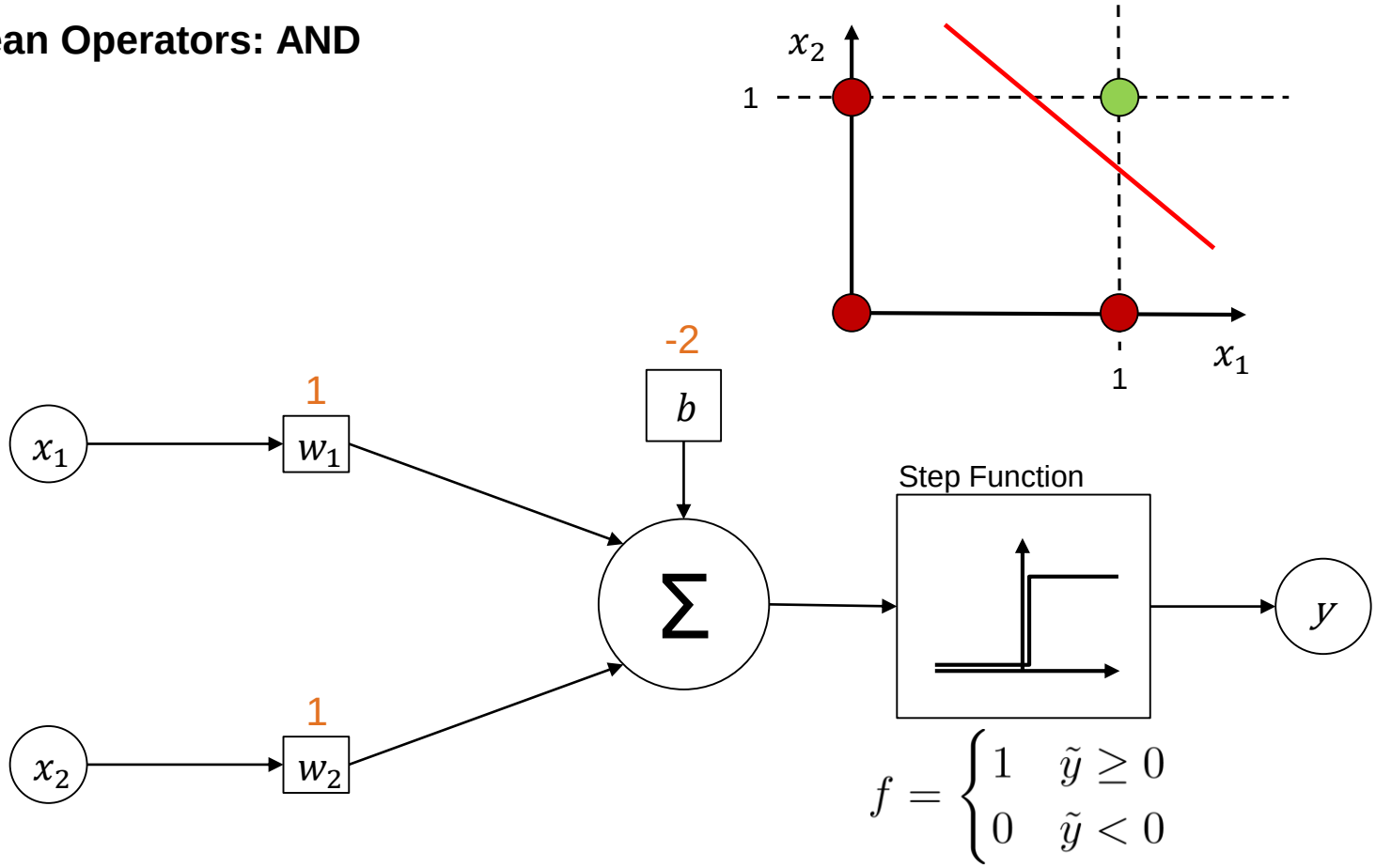
Additional Slides

Although the proof for the Universal Approximation Theorem is beyond this lecture, we can take a look at the functional completeness of ANN in contrast to the functional completeness of a single artificial neuron: the term functional completeness comes from the domain of mathematical logic. A set of boolean operators is said to be functional complete if it can express all possible truth tables by combining members of the set into a boolean expression.

Multilayer Networks

Functional Completeness

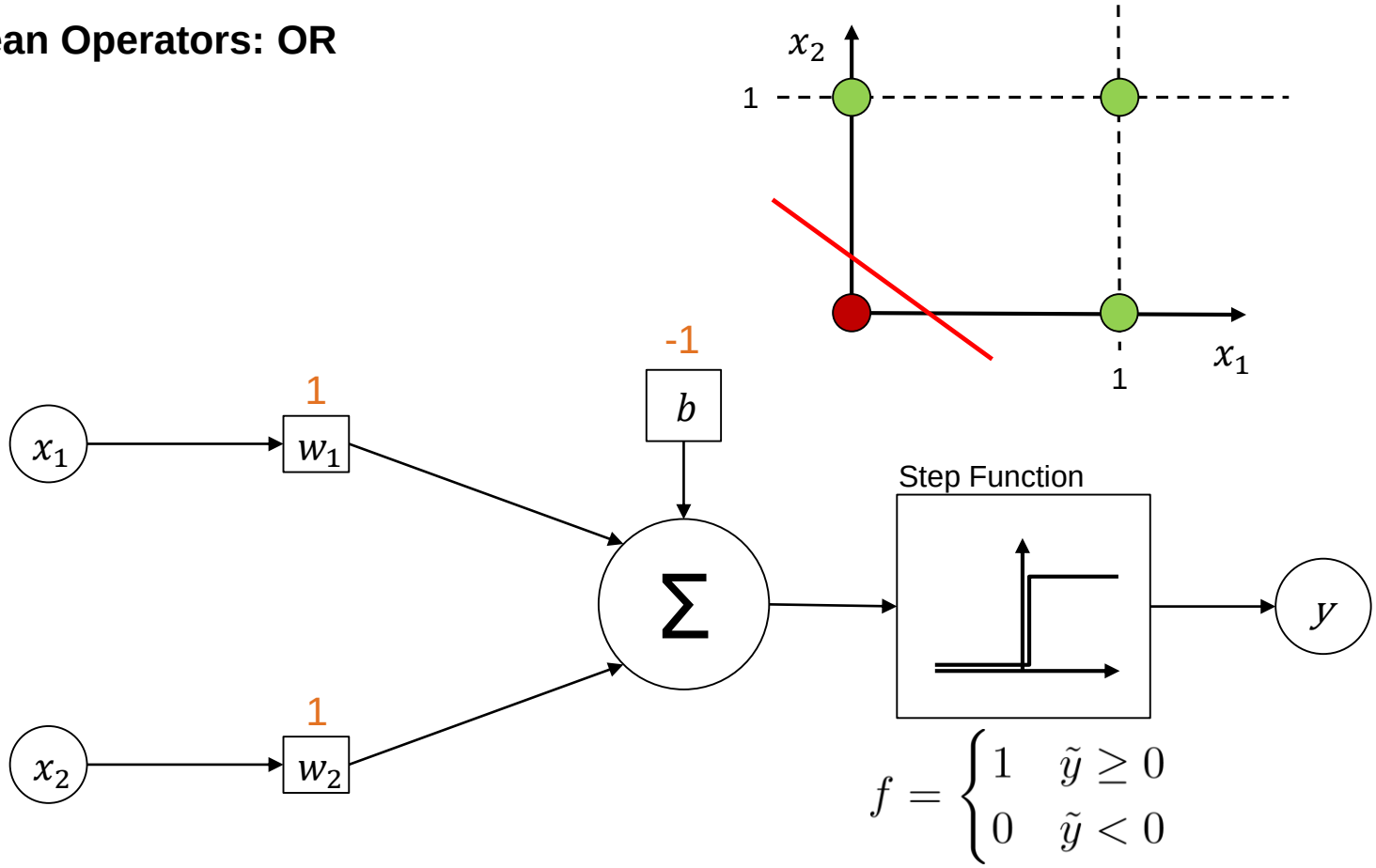
Boolean Operators: AND



Multilayer Networks

Functional Completeness

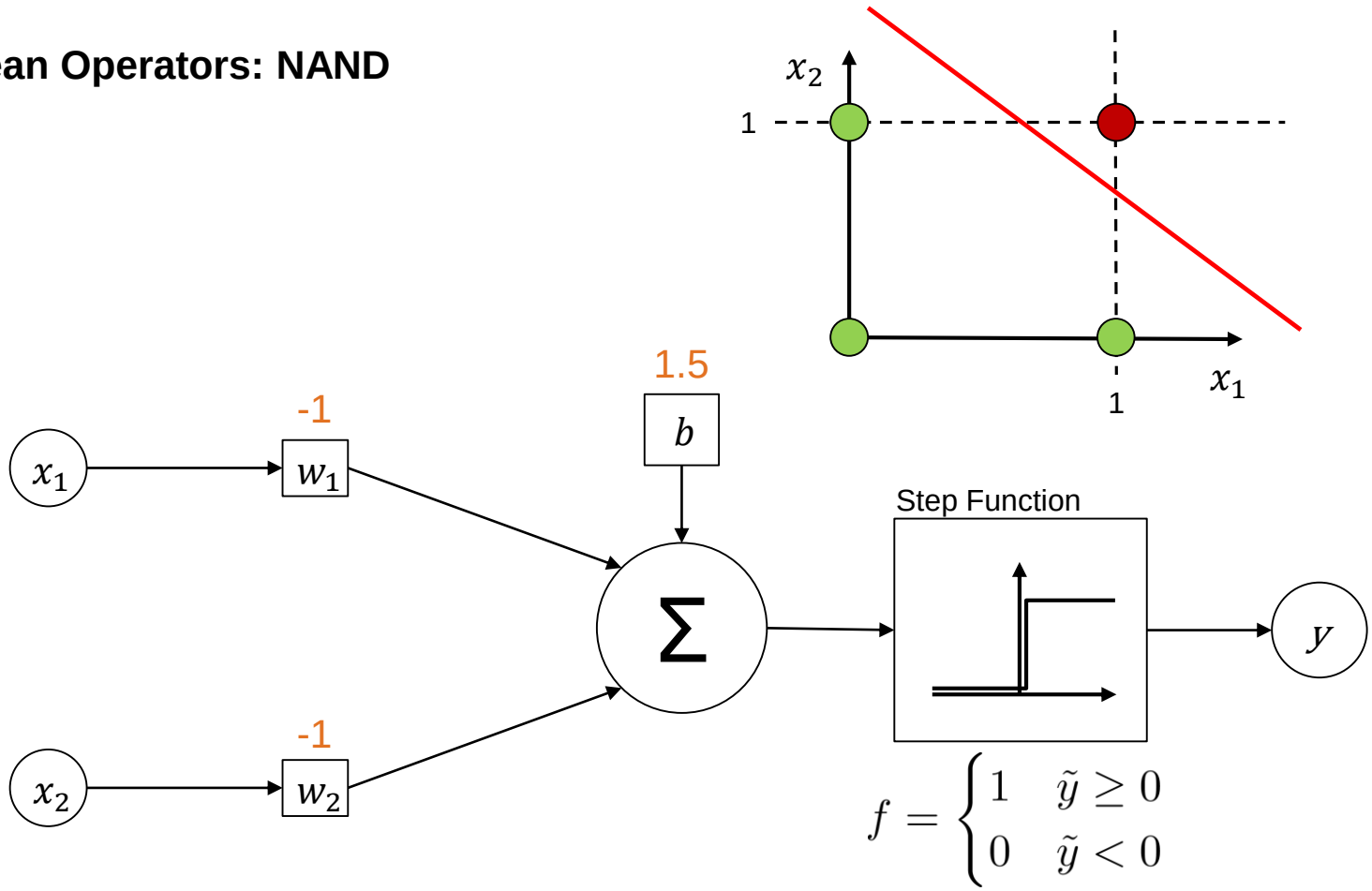
Boolean Operators: OR



Multilayer Networks

Functional Completeness

Boolean Operators: NAND



Additional Slides

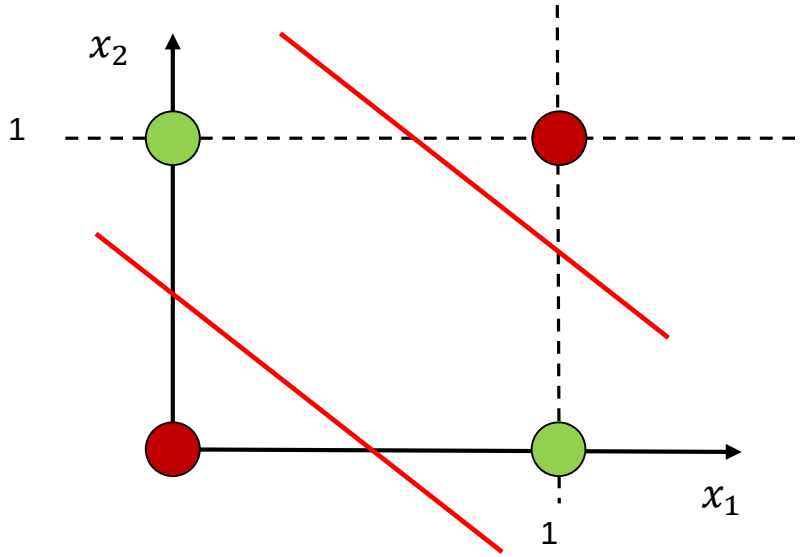
It is evident that a single artificial neuron can produce the truth tables of the basic boolean operators AND, OR and NAND. That is, there are combinations of weights and bias that – in combination with a binary step activation function – cause the neuron to behave like any of these boolean operators by classifying its inputs correctly as either 1 or 0. Hence, the neuron is essentially performing a binary classification task in the input data space.

Another truth table that appears regularly in boolean logic is the XOR table. The XOR operator is 1 if and only if one of its inputs is 1 and all the other inputs are 0. Looking at the corresponding input data space, it is evident that a single class separation line can not correctly distinguish the two result classes (they are not linear separable). Therefore, binary classification using a single neuron fails to model the XOR operator correctly.

Multilayer Networks

Functional Completeness

Boolean Operators: XOR

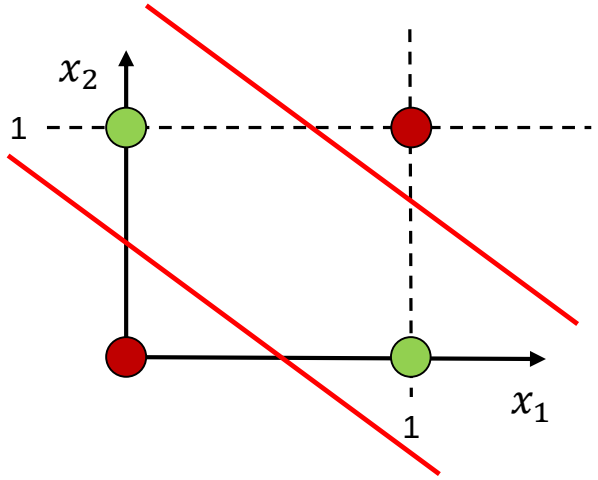


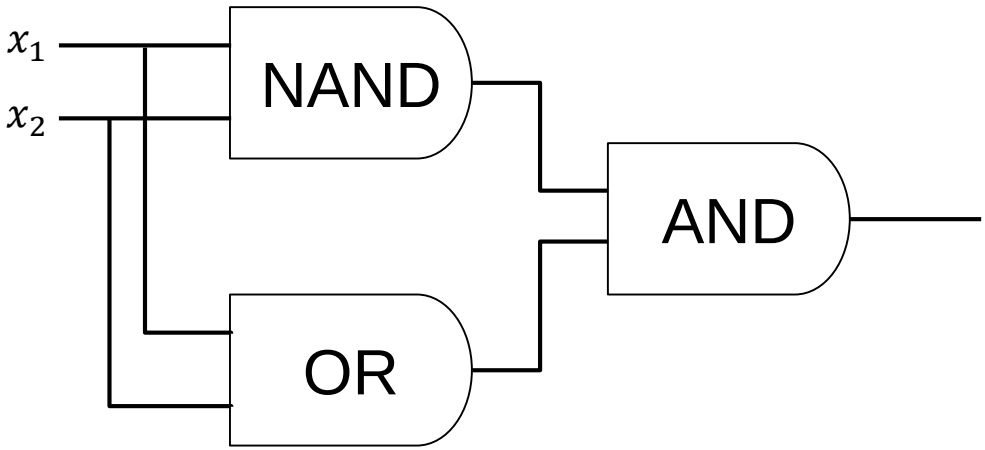
No linear separability

Multilayer Networks

Functional Completeness

Boolean Operators: XOR





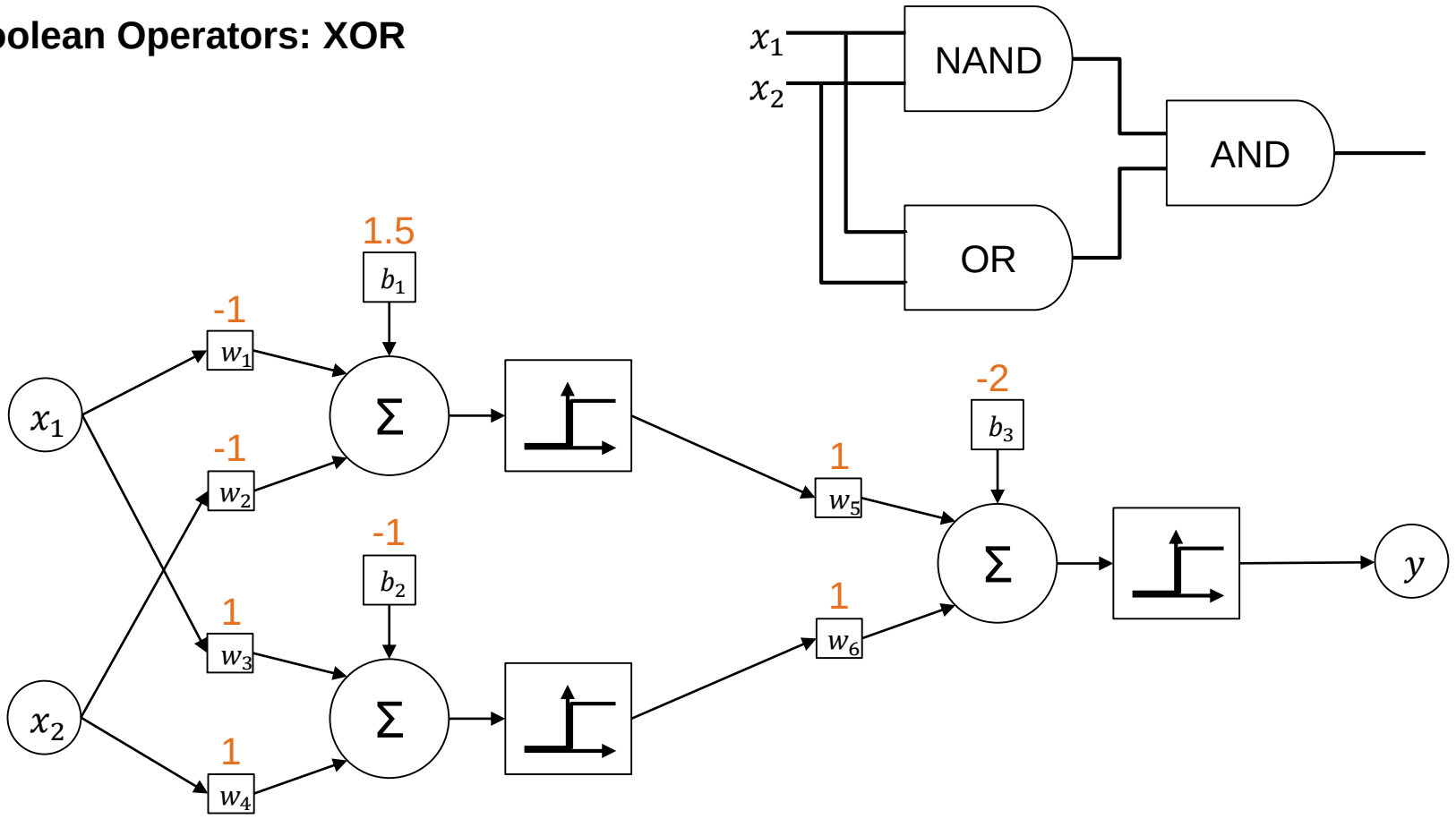
Additional Slides

From boolean algebra, we know that there is a possibility to use NAND, OR and AND operators to emulate an XOR operator. To do this, NAND, OR and AND have to be wired together. The resulting network of operators behaves like a single XOR operator.

Multilayer Networks

Functional Completeness

Boolean Operators: XOR



Additional Slides

By chaining the neurons analogue to the boolean network on the previous slide, a XOR operator can be realized using artificial neurons. Therefore, ANN are functional complete, whereas a single artificial neuron is not. This simple proof of functional completeness means that an ANN can potentially learn all logic expressions that can be formulated using boolean algebra.

As you know from earlier parts of this lecture, binary step functions cannot be used to train artificial neurons and sigmoid functions do not provide a binary output. Due to this fact, it is possible to construct all boolean operators using artificial neurons or a small ANN, but – although possible – training them to show the desired behavior requires additional considerations. If such a training is desired, sigmoid functions have to be used and the results have to be projected to either 1 or 0 to rule out fuzziness after training.

Introduction: Artificial Neural Networks
Prof. Dr.-Ing. Markus Lienkamp

(Lennart Adenaw, M. Sc.)

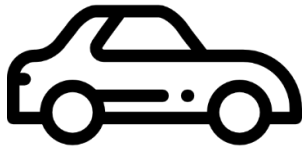
Agenda

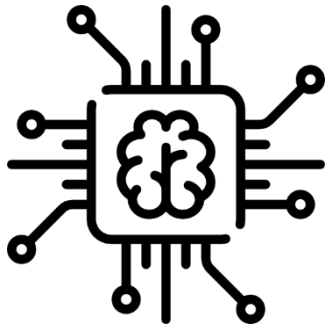
1. Chapter: Introduction

2. Chapter: Towards Artificial Neurons
    - 2.1 Linear Regression
    - 2.2 Gradient Descent
    - 2.3 The Neuron

3. Chapter: Multilayer Networks
    - 3.1 Functional Completeness
    - **3.2 MNIST Example**

4. Chapter: Summary

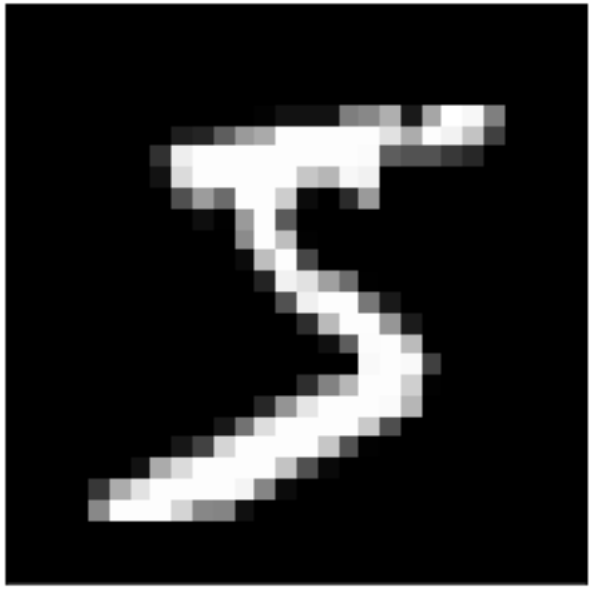




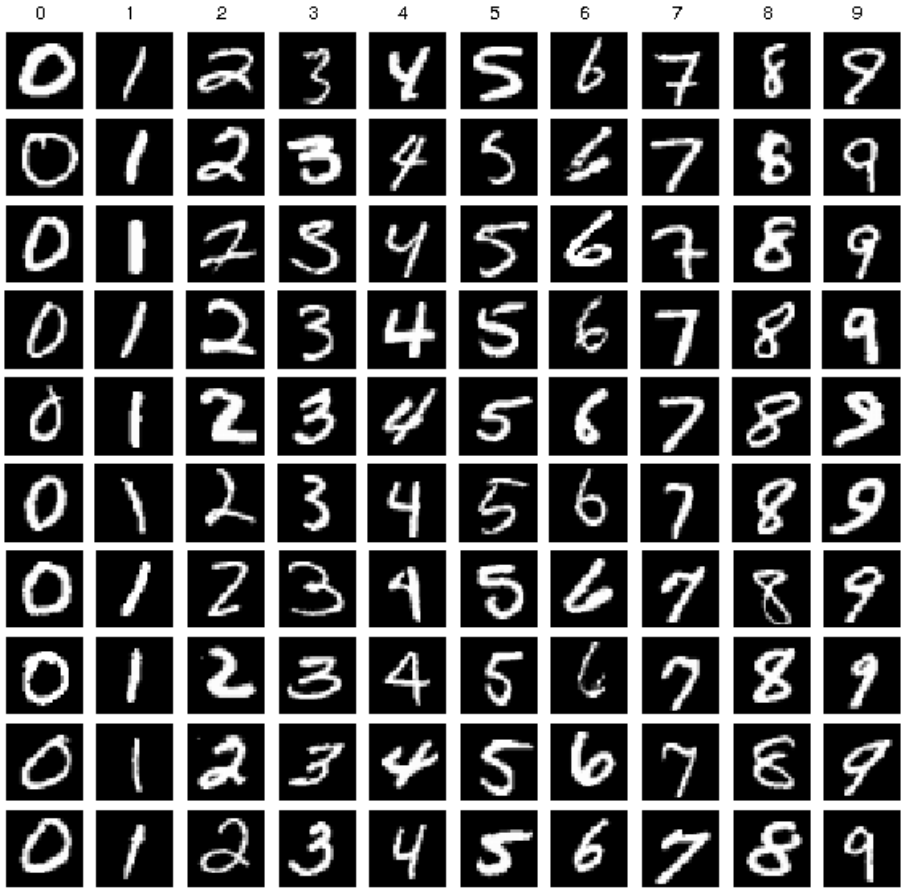
Multilayer Networks

MNIST Example

28x28 Grayscale



http://conx.readthedocs.io/en/latest/_images/MNIST_6_0.png



https://www.researchgate.net/publication/306056875_An_analysis_of_image_storage_systems_for_scalable_training_of_deep_neural_networks/figures?lo=1

Multilayer Networks

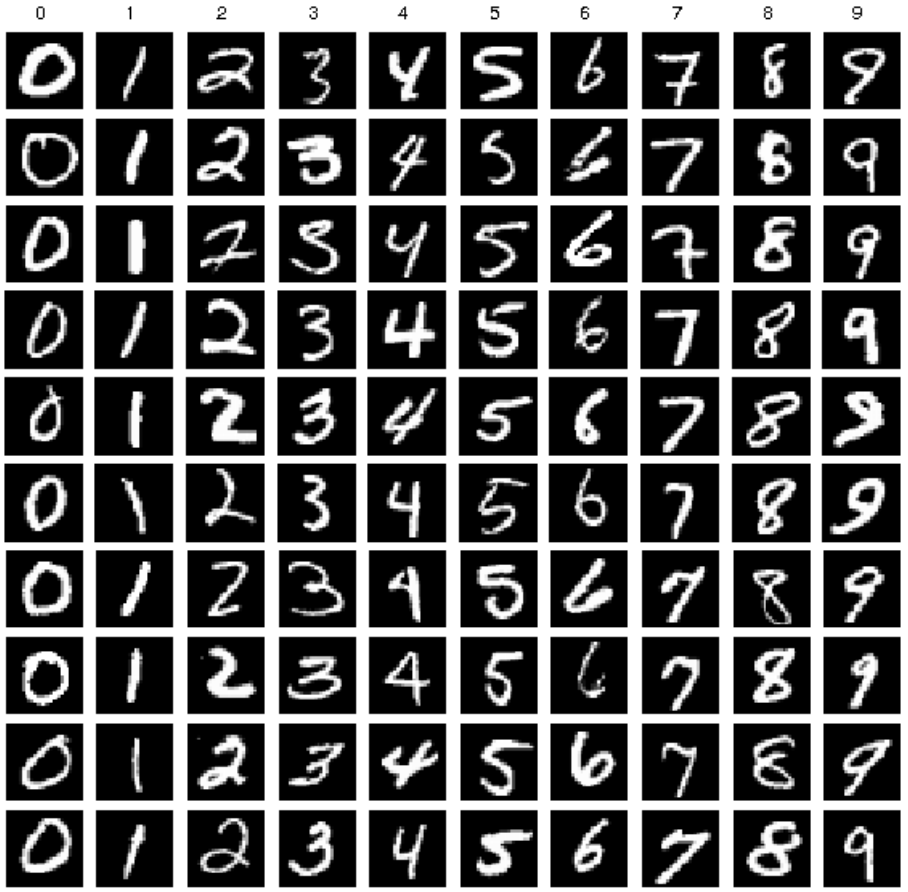
MNIST Example

Properties:

- 60.000 handwritten numbers
- 28x28 pixels
- 0 to 255 grayscale
- Numbers 0 to 9

Task

Train a classifier that can identify the handwritten numbers

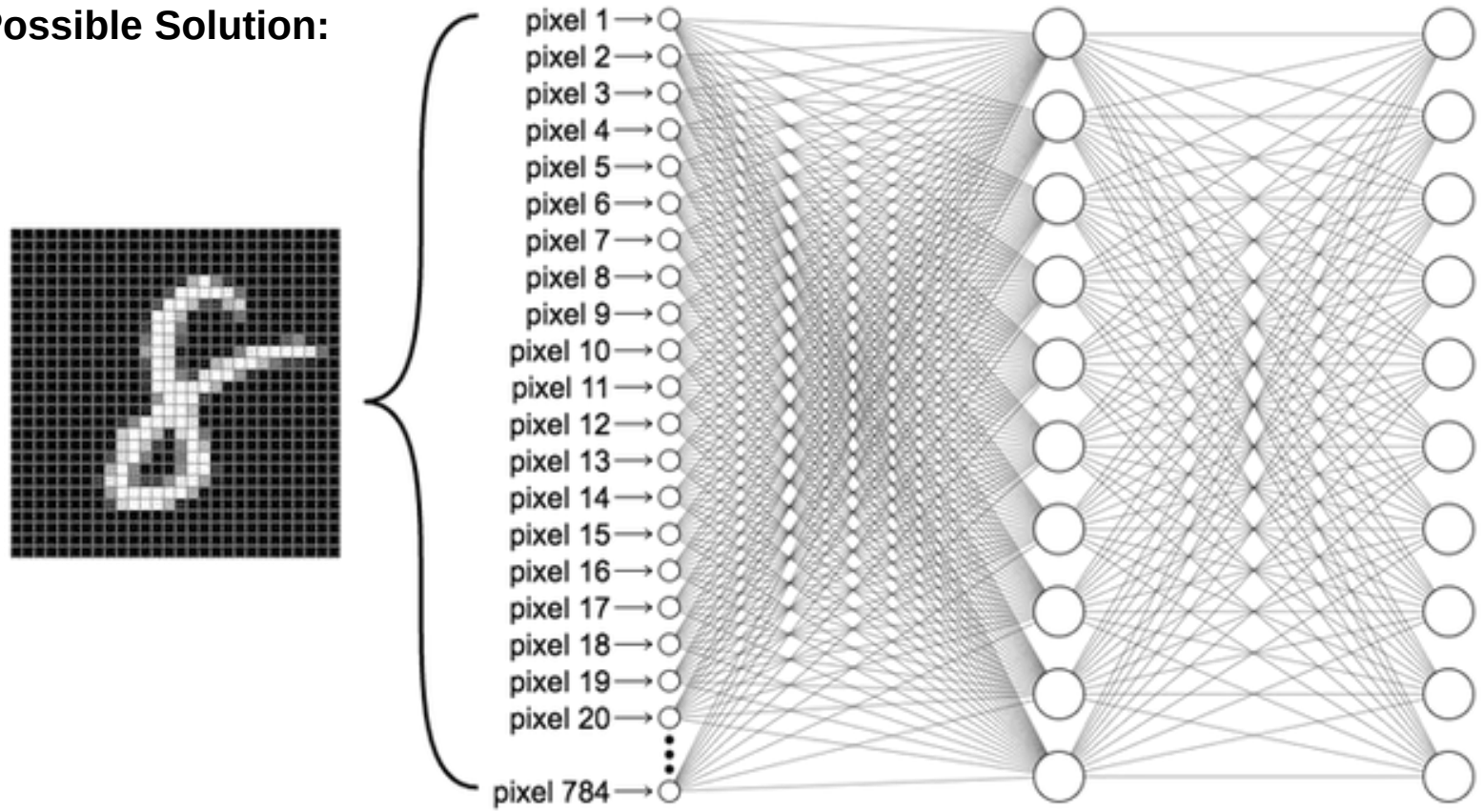


https://www.researchgate.net/publication/306056875_An_analysis_of_image_storage_systems_for_scalable_training_of_deep_neural_networks/figures?lo=1

Multilayer Networks

MNIST Example

Possible Solution:

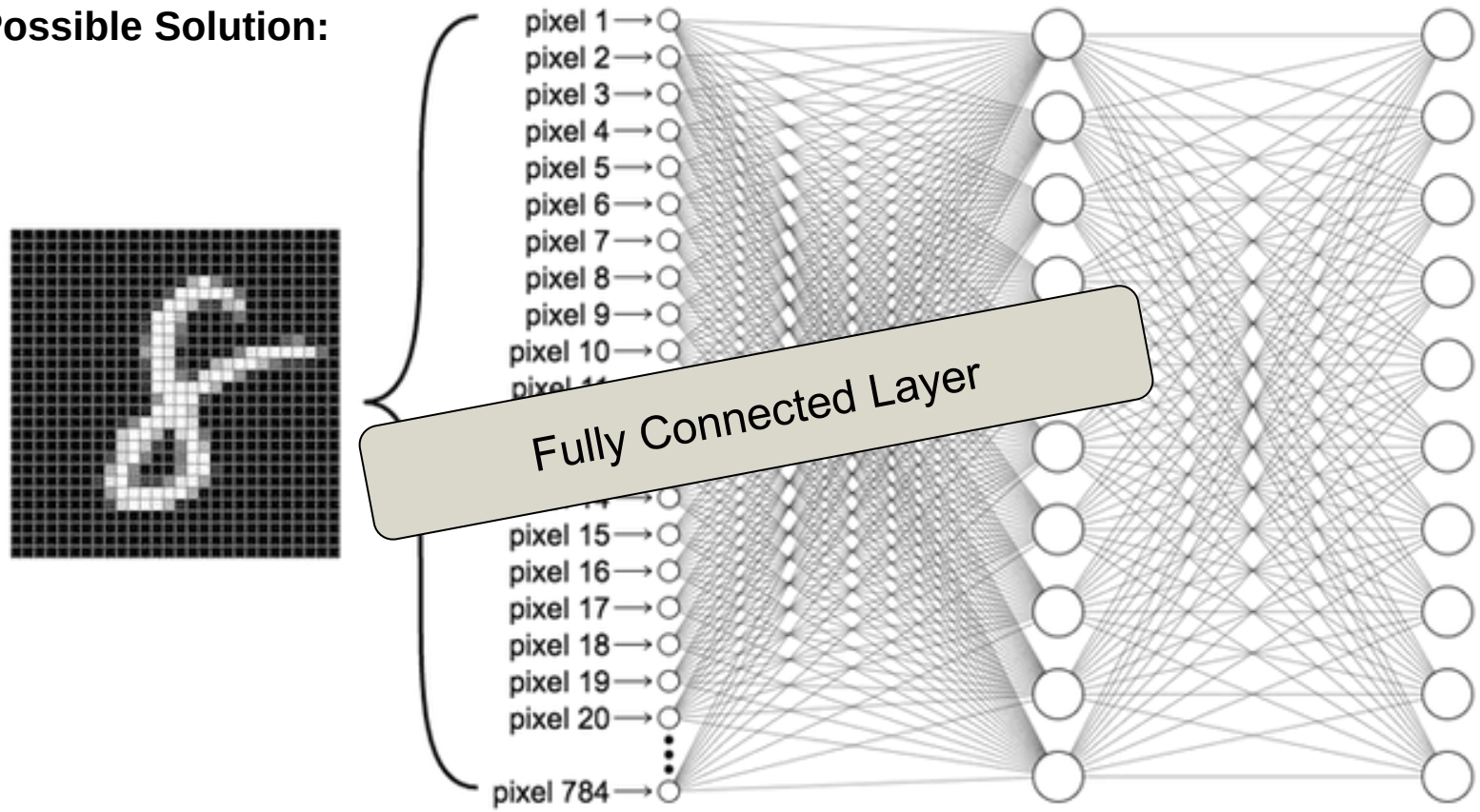


https://achintavarna.wordpress.com/2017/11/17/keras-tutorial-for-beginners-a-simple-neural-network-to-identify-numbers-mnist-data/

Multilayer Networks

MNIST Example

Possible Solution:



https://achintavarna.wordpress.com/2017/11/17/keras-tutorial-for-beginners-a-simple-neural-network-to-identify-numbers-mnist-data/

Introduction: Artificial Neural Networks
Prof. Dr.-Ing. Markus Lienkamp

(Lennart Adenaw, M. Sc.)

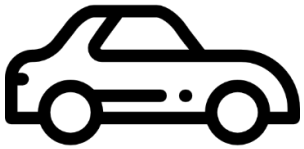
Agenda

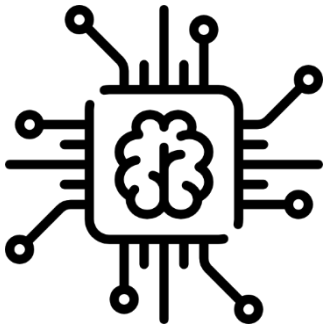
1. Chapter: Introduction

2. Chapter: Towards Artificial Neurons
    - 2.1 Linear Regression
    - 2.2 Gradient Descent
    - 2.3 The Neuron

3. Chapter: Multilayer Networks
    - 3.1 Functional Completeness
    - 3.2 MNIST Example

4. Chapter: Summary





Summary

What we learned today:

Neural Networks are mathematical tools that can approximate any mathematical function

Gradient Descent is an approach suitable for weight adjustments

A single Neuron can perform Linear Regression and Binary Classification

Non-Linear, Multiple Classification and Regression is best performed by Neural Networks

Vocabulary and Ideas