Deep Learning

Feedforward neural networks: Multi Layer Perceptron

Technische Hochschule Rosenheim
Sommer 2023
Prof. Dr. Jochen Schmidt

Acknowledgements

Many of the slides presented here are based on the Deep Learning Slides Summer Semester 2020, courtesy of
A. Maier, V. Christlein, K. Breininger, F. Denzinger, F. Thamm,
Pattern Recognition Lab, Friedrich-Alexander-University Erlangen-Nürnberg.
https://lme.tf.fau.de/

Overview

Technische Hochschule Rosenheim

- Perceptrons
- MLP-Training Basics
- MLP – Layer Abstraction
- Universal Approximation Theorem

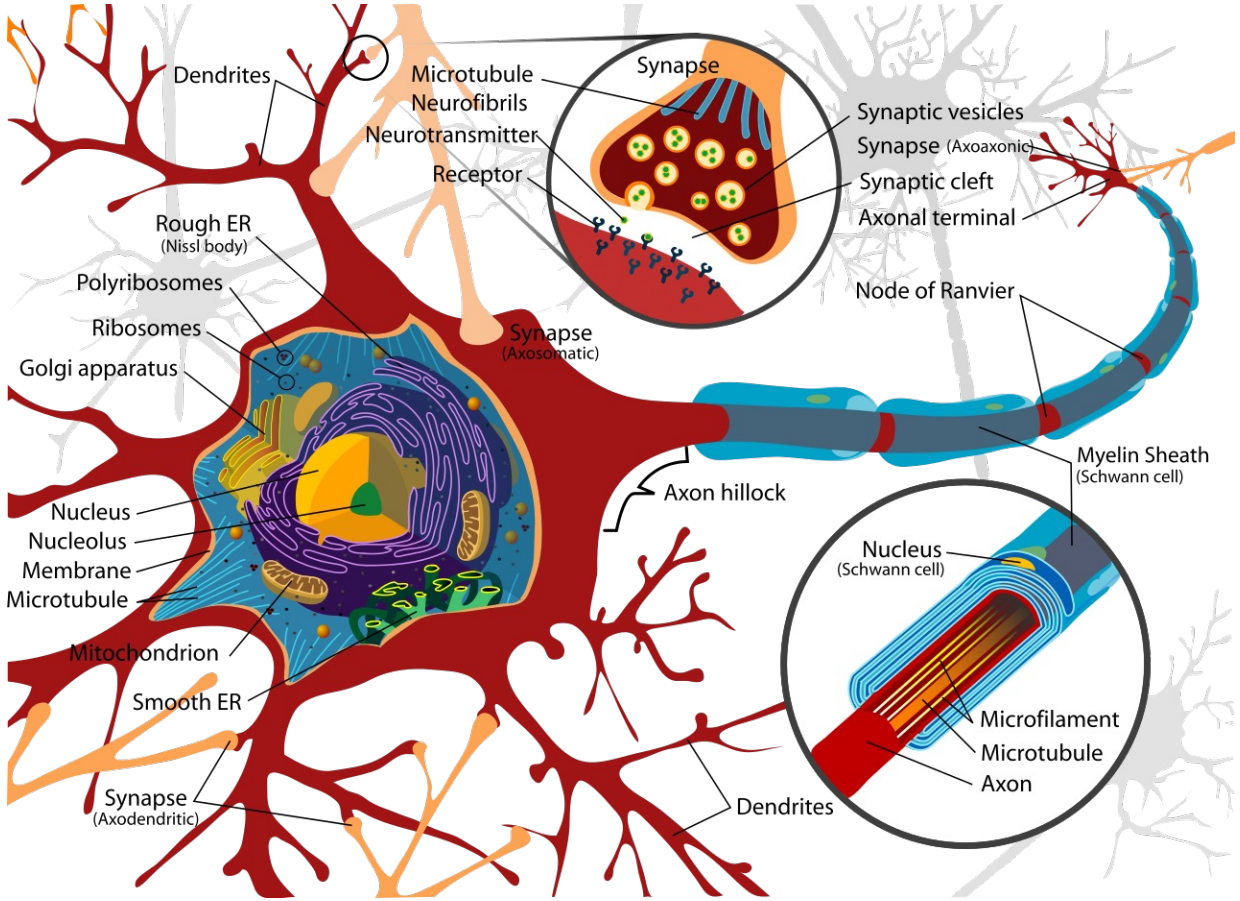
Perceptrons

Real Neurons

Technische Hochschule Rosenheim

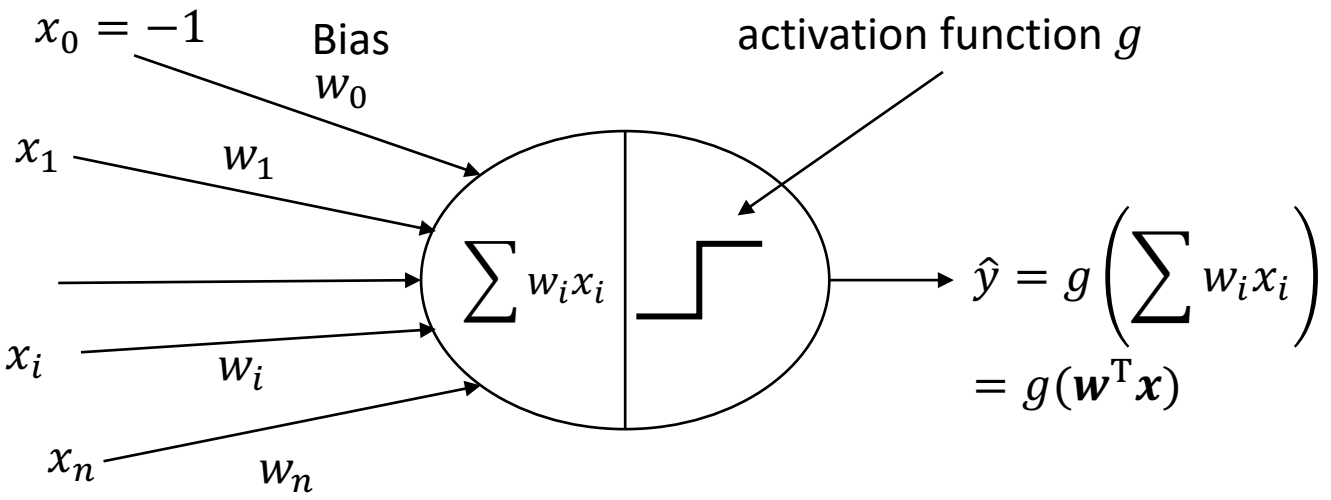
Human brain:

- 10^{11} Neurons
- more than 20 different types
- 10^{14} synapses (connections)
- Cycle time: 1-10 msec

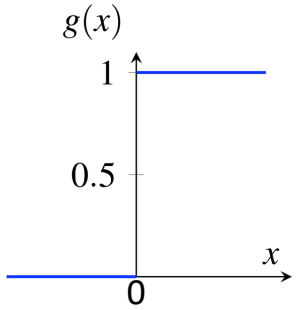


Rosenblatt's Perceptron

- simplified model of a real neuron
- by Frank Rosenblatt 1957
- binary classifier $y \in \{0, 1\}$
- $\boldsymbol{w} = (w_0, w_1, \ldots, w_n)$ set of weights (w_0: bias)
- $\boldsymbol{x} = (-1, x_1, \ldots, x_n)$ input feature vector
- Output: linear function of input + (non-linear) threshold



$$\hat{y} = \begin{cases} 1 & \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} > 0 \\ 0 & \text{otherwise} \end{cases}$$



Bias

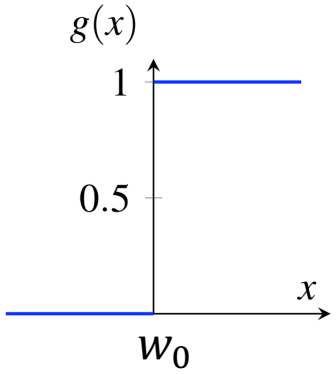
Technische Hochschule Rosenheim

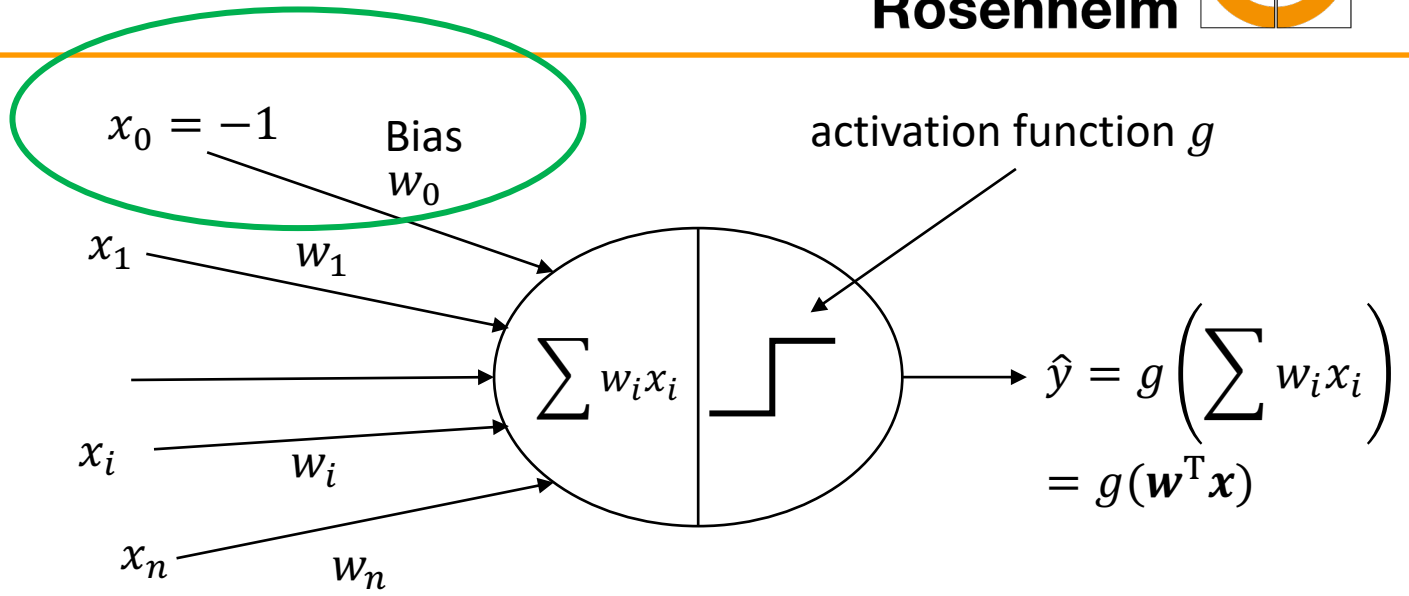
Decision threshold is always zero?
Can we set it to an arbitrary value?
YES: Bias weight = threshold position!

$$\hat{y} = \begin{cases} 1 & \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} = \sum_{i=0}^{n} w_i x_i = w_0 x_0 + \sum_{i=1}^{n} w_i x_i = -w_0 + \sum_{i=1}^{n} w_i x_i$$

$$\sum_{i=0}^{n} w_i x_i > 0 \quad \longrightarrow \quad \sum_{i=1}^{n} w_i x_i > w_0$$



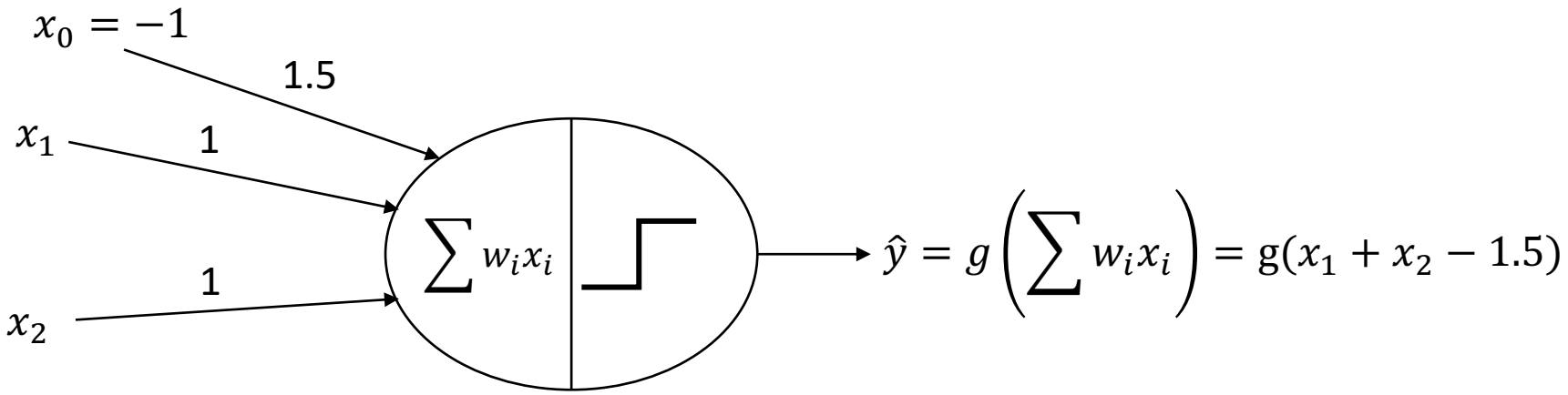


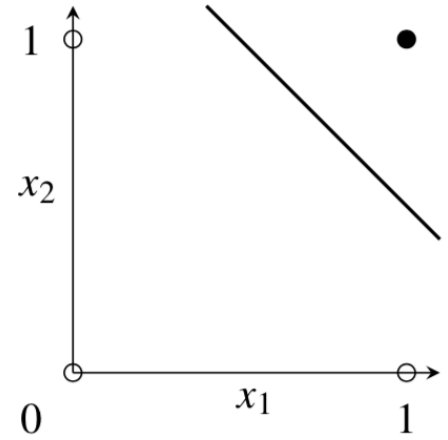
Why?

- The threshold becomes just another weight
- The weights are learned during training
- Thus, the threshold can be trained automatically

Example: Boolean AND

A perceptron can compute the Boolean AND-function:



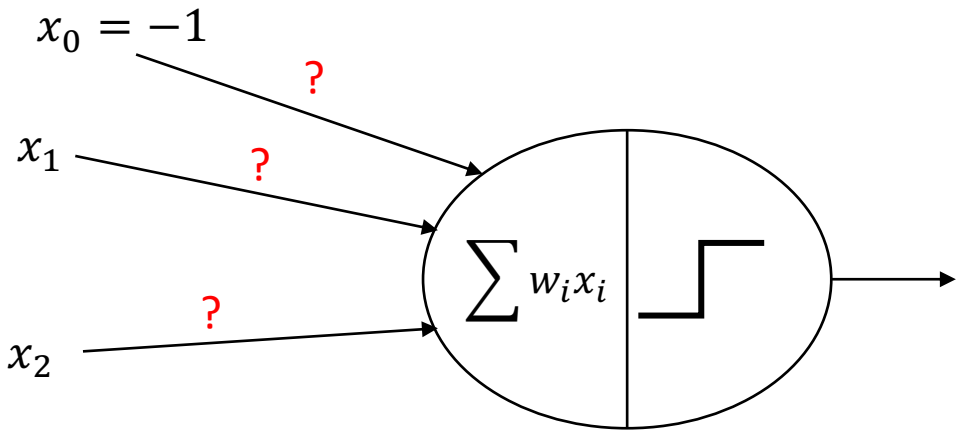


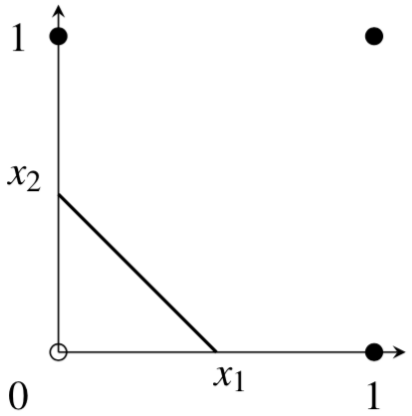
Could we use other weights?

x_1	x_2	$\sum w_i x_i$	$g\left(\sum w_i x_i\right)$
0	0	–1.5	0
0	1	–0.5	0
1	0	–0.5	0
1	1	0.5	1

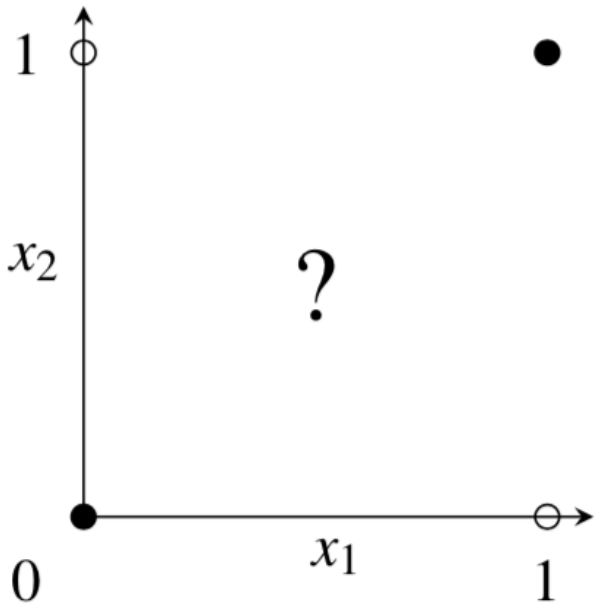
Exercise: Boolean OR

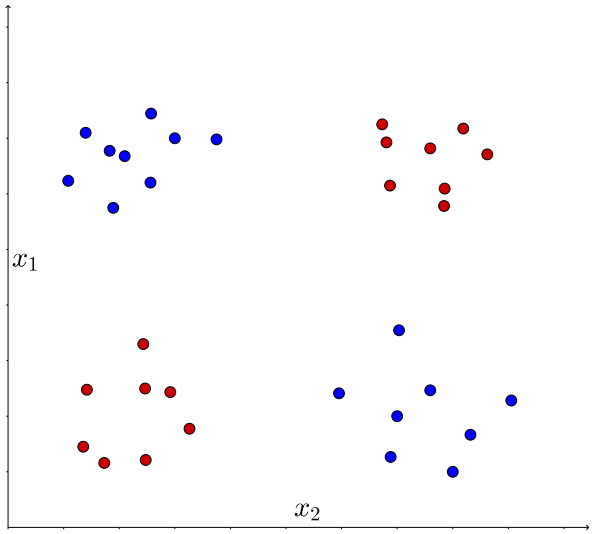
What would the weights look like for the Boolean OR-function?





Boolean XOR?





Observation: A single perceptron computes a linear separation of the feature space

Perceptron – Interpretation

$$f(\boldsymbol{x}) = -w_0 + \sum_{i=1}^{n} w_i x_i = \boldsymbol{w'}^{\mathrm{T}} \boldsymbol{x'} - w_0$$ is the equation of an n-dimensional hyperplane

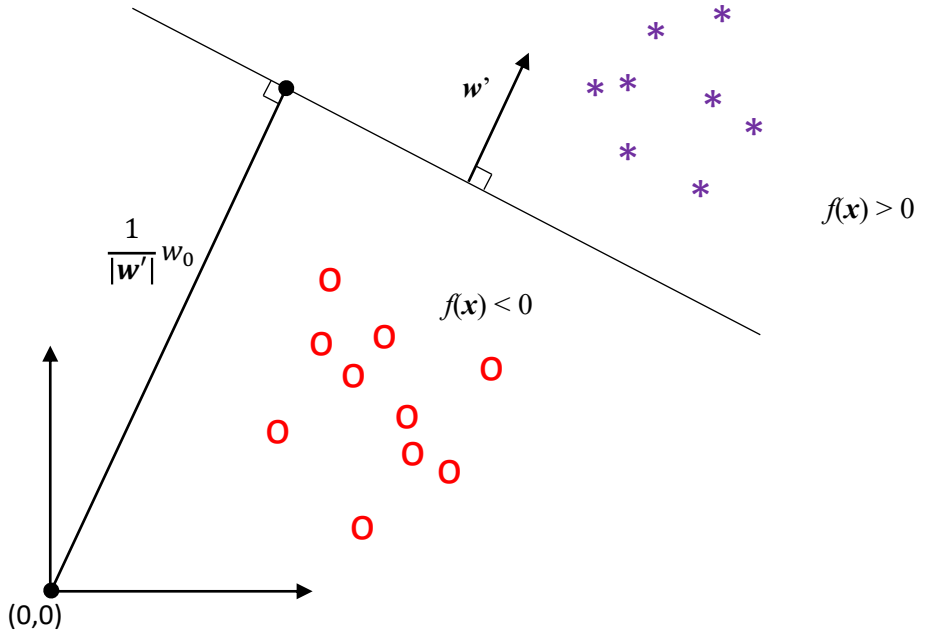
$\boldsymbol{w'} = (w_1, \ldots, w_n)$ set of weights without bias w_0
$\boldsymbol{x'} = (x_1, \ldots, x_n)$ input feature vector

- normal vector: $\boldsymbol{w'}$
  - point on plane: $f(\boldsymbol{x'}) = 0$
  - point above plane: $f(\boldsymbol{x'}) > 0$
  - point below plane: $f(\boldsymbol{x'}) < 0$
  - ”above” = in direction of plane normal

- Signed distance d of a point to hyperplane
  - normalize $\boldsymbol{w'}$, such that $|\boldsymbol{w'}| = 1$:

$$d = f'(\boldsymbol{x'}) = \frac{1}{|\boldsymbol{w'}|} f(\boldsymbol{x'}) = \frac{1}{|\boldsymbol{w'}|} \boldsymbol{w'}^{T} \boldsymbol{x'} - \frac{1}{|\boldsymbol{w'}|} w_0$$

  - distance of plane from origin: $\frac{1}{|\boldsymbol{w'}|} w_0$



P.S.: The perceptron decision rule looks exactly like the one for an SVM (Support Vector Machine)

Perceptron Training

Technische Hochschule Rosenheim

Can we learn the weights automatically?

- We need a training data set: $\mathcal{S} = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_m, y_m)\}$
- Weights $\boldsymbol{w} = (w_0, w_1, \ldots, w_n)$ initialized randomly

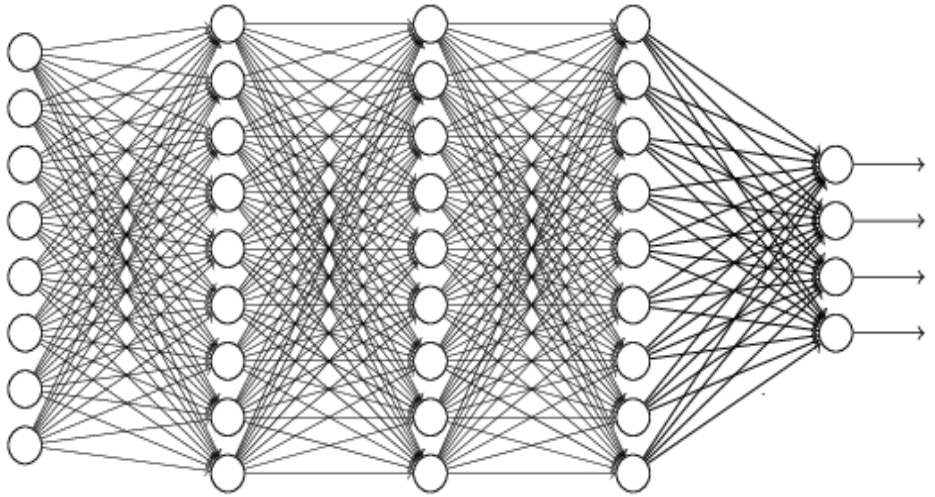
- Repeat until convergence or for a defined number of iterations:
  - for each training sample $(\boldsymbol{x}_e, y_e) \in \mathcal{S}$
    - compute output $\hat{y_e} = \mathrm{g}(\boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_e)$
    - compute residual error: $\varepsilon = y_e - \hat{y_e}$
    - update weights: $\boldsymbol{w} := \boldsymbol{w} + \alpha\, \varepsilon\, \boldsymbol{x}_e$

- α: **Learning Rate**, e.g., $\alpha = 0.1$

- this algorithm updates weights after each sample
- other strategies:
  - process all samples, sum changes, update weights after each iteration
  - process small batches, sum changes, then update

Multi-Layer Perceptron (MLP)

- How to obtain nonlinear decision boundaries? Use more than one neuron.
- Neurons are arranged in layers
- Layer i is (fully) connected with layer i+1
  - no connections within layer
  - no connections to any other layers
  - no feedback
- Information flow from one layer to the next: **feed-forward**
- network has no internal state



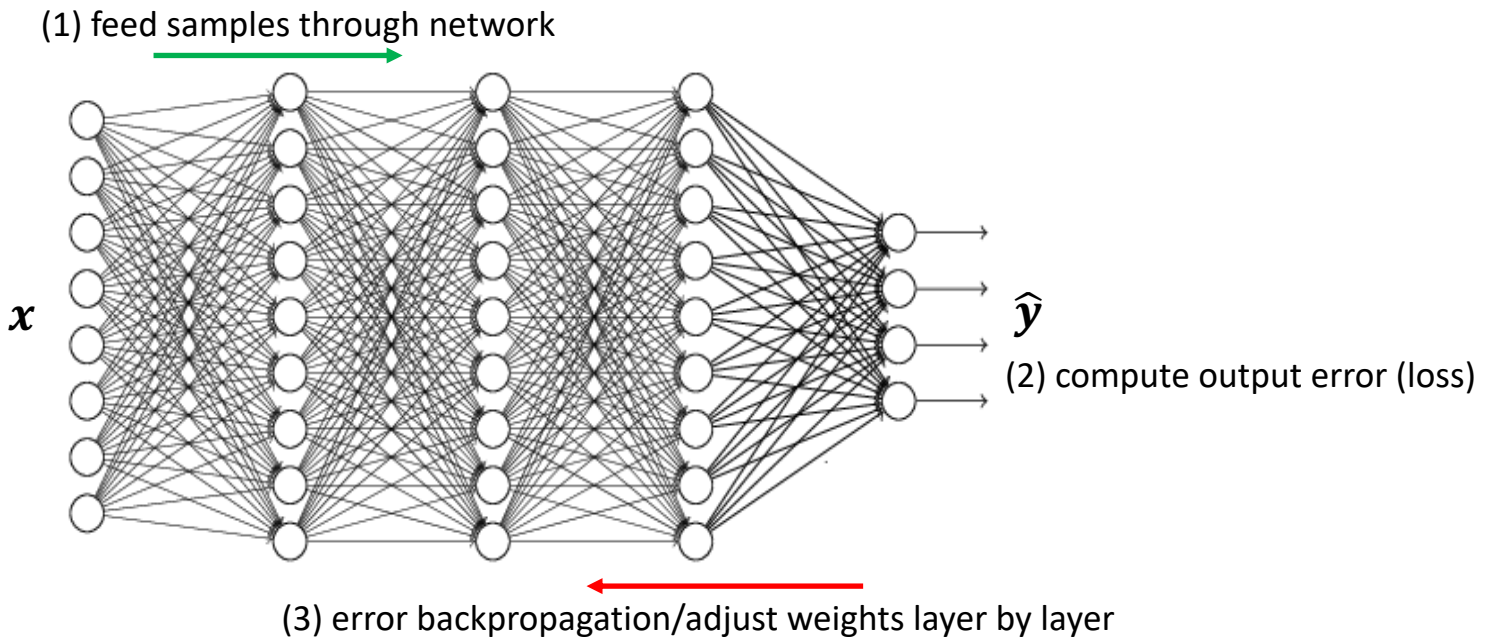
- number of input/output units is problem dependent
- number of layers/hidden units determined by developer
  - … we will discuss this in more detail later

MLP-Training Basics

Training

Training = Determine weights

- define loss function L of output neurons
- minimize loss



Training as an Optimization Problem

Technische Hochschule Rosenheim

Objective: Find optimal weights $\boldsymbol{w}$ for all layers

- Abstraction: Consider the whole neural network as a function

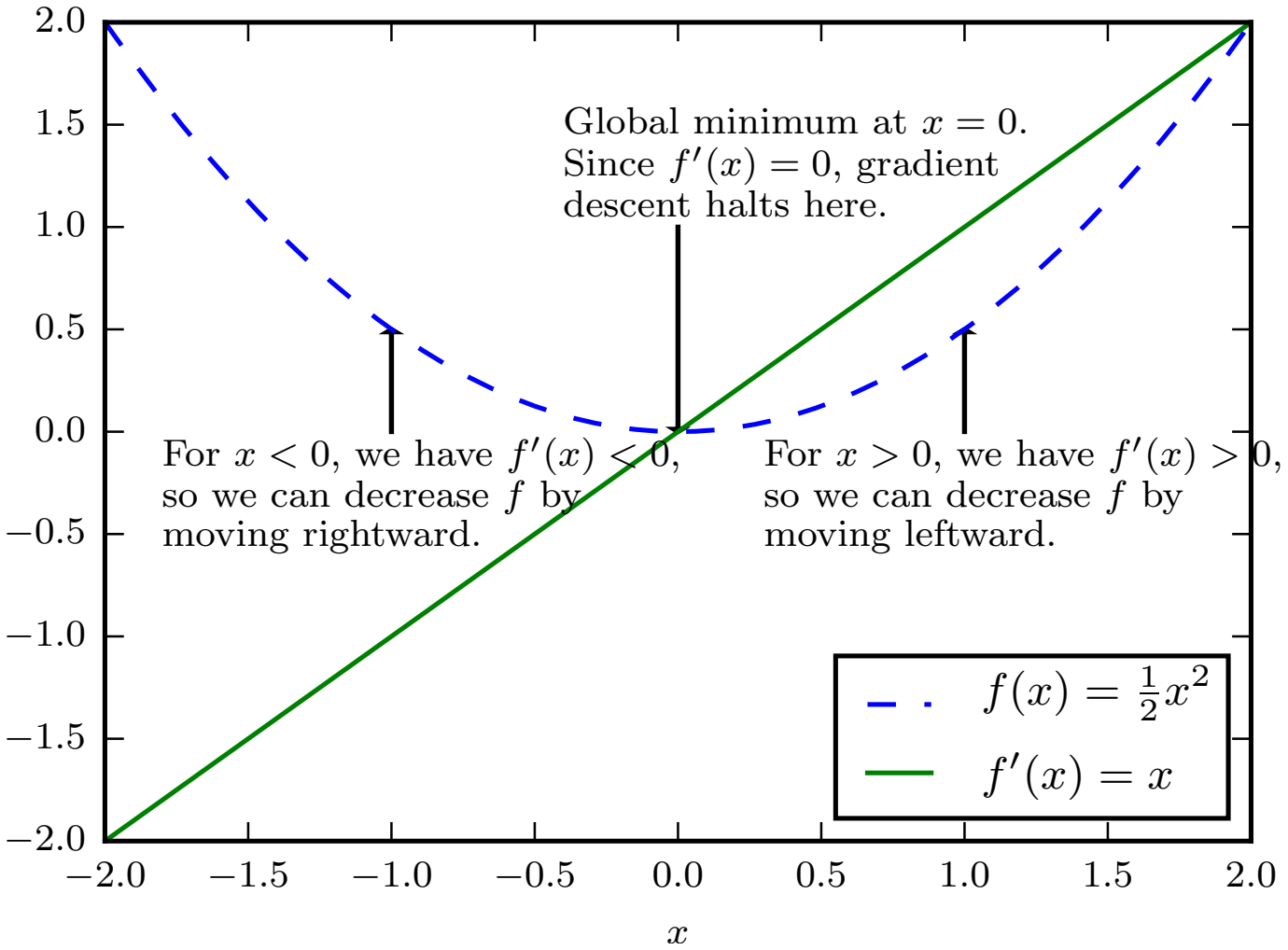
$$L(\boldsymbol{w}, \boldsymbol{x}, \boldsymbol{y})$$

- Consider all M training samples

$$\frac{1}{M} \sum_{m=1}^{M} L(\boldsymbol{w}, \boldsymbol{x}_m, \boldsymbol{y}_m)$$

- Minimize (with unknowns $\boldsymbol{w}$)
  - computing the first derivative and solving for zero crossings will not work (highly nonlinear)
  - method of choice: Gradient Descent

Gradient Descent Illustration

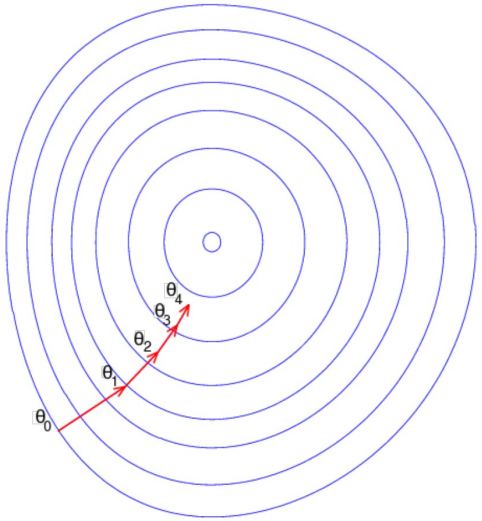


Gradient Descent

$$\frac{1}{M}\sum_{m=1}^{M} L(\boldsymbol{w}, \boldsymbol{x}_m, \boldsymbol{y}_m)$$

1. Initialize weights $\boldsymbol{w}$
2. Iterate until convergence (or for a defined number of iterations)

$$\boldsymbol{w}^{(k+1)} := \boldsymbol{w}^{(k)} - \alpha \nabla_{\boldsymbol{w}} \frac{1}{M}\sum_{m=1}^{M} L(\boldsymbol{w}, \boldsymbol{x}_m, \boldsymbol{y}_m)$$



α: **Learning Rate**

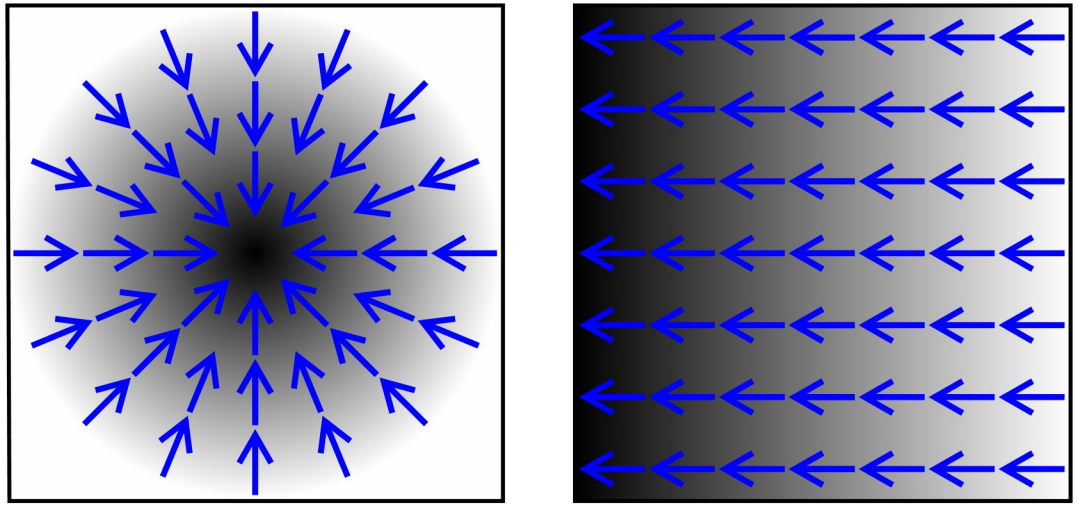
$$\nabla_x f(\boldsymbol{x}, \boldsymbol{y}) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}$$

Gradient of f

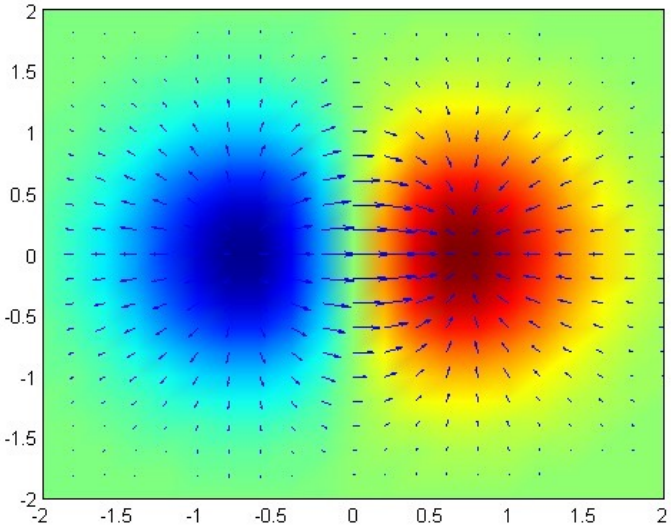
- vector direction: direction of steepest increase
- vector magnitude: rate/strength of increase

Gradient Illustration

Gradient of f
- vector direction: direction of steepest increase
- vector magnitude: rate/strength of increase

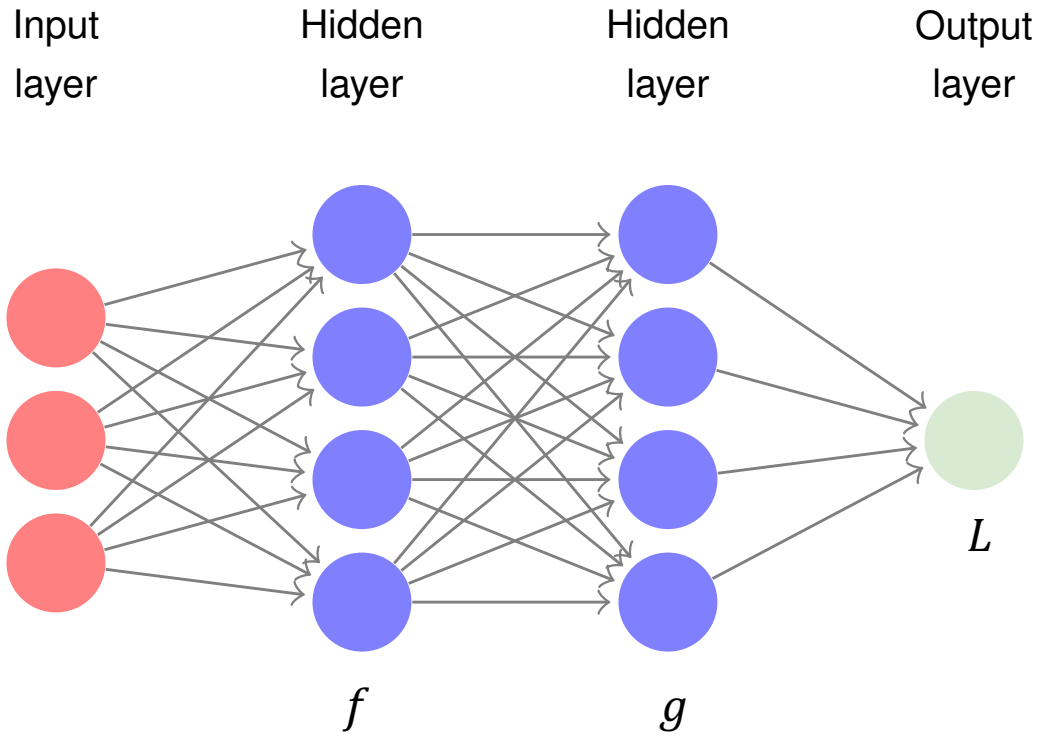


Function values in grayscale:
white = low values
dark = high values



Gradient of $f(x, y) = \frac{x}{e^{x^2+y^2}}$ as pseudo color plot
blue/cold = low values
red/hot = high values

What does this look like in a Neural Network?



A complex network can be looked at as composed functions:

$$L(\boldsymbol{w}, \boldsymbol{x}, \boldsymbol{y}) = L\big(g\big(f\big(\boldsymbol{x}, \boldsymbol{w}_f\big), \boldsymbol{w}_g\big), \boldsymbol{y}\big)$$

How do we Calculate Derivatives in a Neural Network?

Technische Hochschule Rosenheim

Example

- function $\hat{y} = f(\boldsymbol{x}) = f(x_1, x_2) = (2x_1 + 3x_2)^2 + 3$
- evaluate $\frac{\partial}{\partial x_1} f(1, 3)$

Two methods:

- numerical derivative using finite differences
- analytic derivative

Numerical Computation Using Finite Differences

Technische Hochschule Rosenheim

Definition of derivative: $$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

This is asymmetric; due to finite precision, we prefer the symmetric definition:

$$f'(x) = \lim_{h \to 0} \frac{f\left(x + \frac{h}{2}\right) - f\left(x - \frac{h}{2}\right)}{h}$$

Thus, we get the derivative at the exact position of x

Numerical Computation – Example

- function $\hat{y} = f(\boldsymbol{x}) = f(x_1, x_2) = (2x_1 + 3x_2)^2 + 3$
- evaluate $\frac{\partial}{\partial x_1} f(1,3)$

$$f'(x) = \lim_{h \to 0} \frac{f\left(x + \frac{h}{2}\right) - f\left(x - \frac{h}{2}\right)}{h}$$

Set $h = 2 \cdot 10^{-2}$. We get

$$\frac{\partial}{\partial x_1} f(1,3) = \frac{(2(1 + 10^{-2}) + 9)^2 + 3 - ((2(1 - 10^{-2}) + 9)^2 + 3)}{2 \cdot 10^{-2}} = \frac{124.4404 - 123.5604}{2 \cdot 10^{-2}} = 43.999$$

(the exact value is 44)

Numerical Computation – Summary

- for practical purposes it often suffices to use $h = 1 \cdot 10^{-5}$
- for a more accurate derivative use $h = \sqrt[3]{\varepsilon_f} \cdot x$
  - ε_f is the fractional accuracy with which f is computed
    - for a simple function this is approximately the machine accuracy ($\varepsilon_m \approx 10^{-7}$ for 32 Bit floats)
    - for complex functions this may be considerably larger than machine accuracy
  - x is the position, where f is being evaluated
  - prevent division by zero when $x = 0$
- see [Pre07, pp.186-189] for more details

- easy to use
- we only need to be able to evaluate functions
  - can be computed for any function, even if we do not have the derivative
- however:
  - computationally inefficient
  - not as accurate as using the analytical derivative
- frequently used for validating implementations of analytical derivatives

Analytic Derivative – Example

- function $\hat{y} = f(\boldsymbol{x}) = f(x_1, x_2) = (2x_1 + 3x_2)^2 + 3$
- evaluate $\frac{\partial}{\partial x_1} f(1, 3)$

$$\frac{\partial}{\partial x_1} f(x_1, x_2) = \frac{\partial}{\partial x_1}\left((2x_1 + 3x_2)^2 + 3\right)$$

$$= \frac{\partial}{\partial x_1}(2x_1 + 3x_2)^2 + \frac{\partial}{\partial x_1} 3 \qquad (2)$$

$$= \frac{\partial}{\partial x_1}(2x_1 + 3x_2)^2 \qquad (1)$$

$$= \frac{\partial}{\partial z} z^2 \; \frac{\partial}{\partial x_1}(2x_1 + 3x_2) \qquad (4) \text{ and } (2x_1 + 3x_2) = z$$

$$= 2z \cdot 2 = 2 \cdot (2x_1 + 3x_2) \cdot 2 = (8x_1 + 12x_2) \qquad (1) + (2)$$

$$\frac{\partial}{\partial x_1} f(1, 3) = (8 \cdot 1 + 12 \cdot 3) = 44$$

Required Rules:

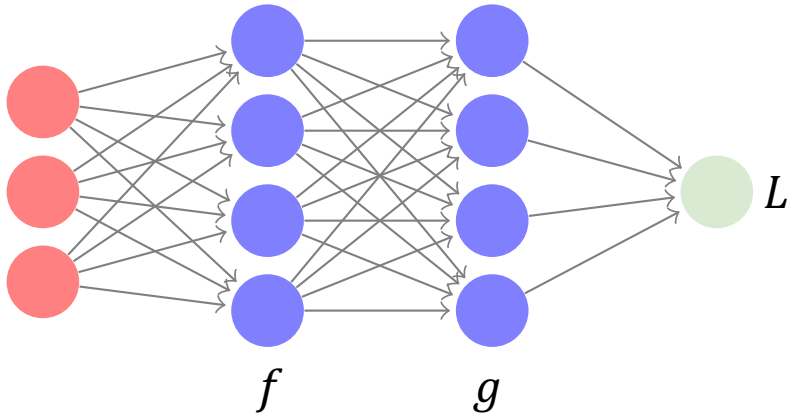
1. $\frac{d}{dx}\,\text{const} = 0$
2. $\frac{d}{dx}$ is a linear operation
3. Monomials: $\frac{d}{dx} x^n = n \cdot x^{n-1}$
4. Chain rule: $\frac{d}{dx} f(g(x)) = \frac{d}{dg} f(g) \cdot \frac{d}{dx} g(x)$

Analytic Derivative – Summary

- **chain rule** and **linearity** enable us to **decompose** complex functions
- analytic derivatives have to be calculated manually
- computationally more efficient and more precise than finite differences

Can we compute analytic gradients of the network loss function automatically?

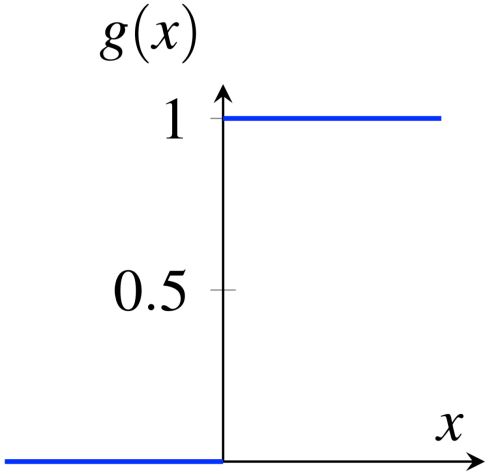
YES: Provided we have the analytic derivative of the activation function – just use chain rule and linearity



$$L(\boldsymbol{w}, \boldsymbol{x}, \boldsymbol{y}) = L\big(g\big(f\big(\boldsymbol{x}, \boldsymbol{w}_f\big), \boldsymbol{w}_g\big), \boldsymbol{y}\big)$$

Ooops – Activation Derivatives Required!

As activation we used the Heaviside step function:



$$g(x) = \begin{cases} 1 & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

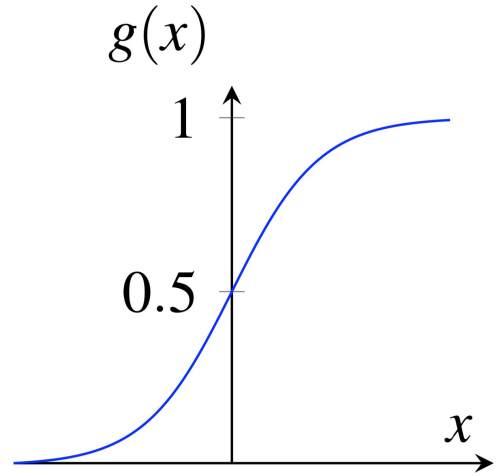
$$g'(x) = \delta(x)$$

replace activation

- not differentiable in the usual sense
- gradient vanishes almost everywhere

Unsuitable for gradient descent!

Sigmoid (also: logistic function)



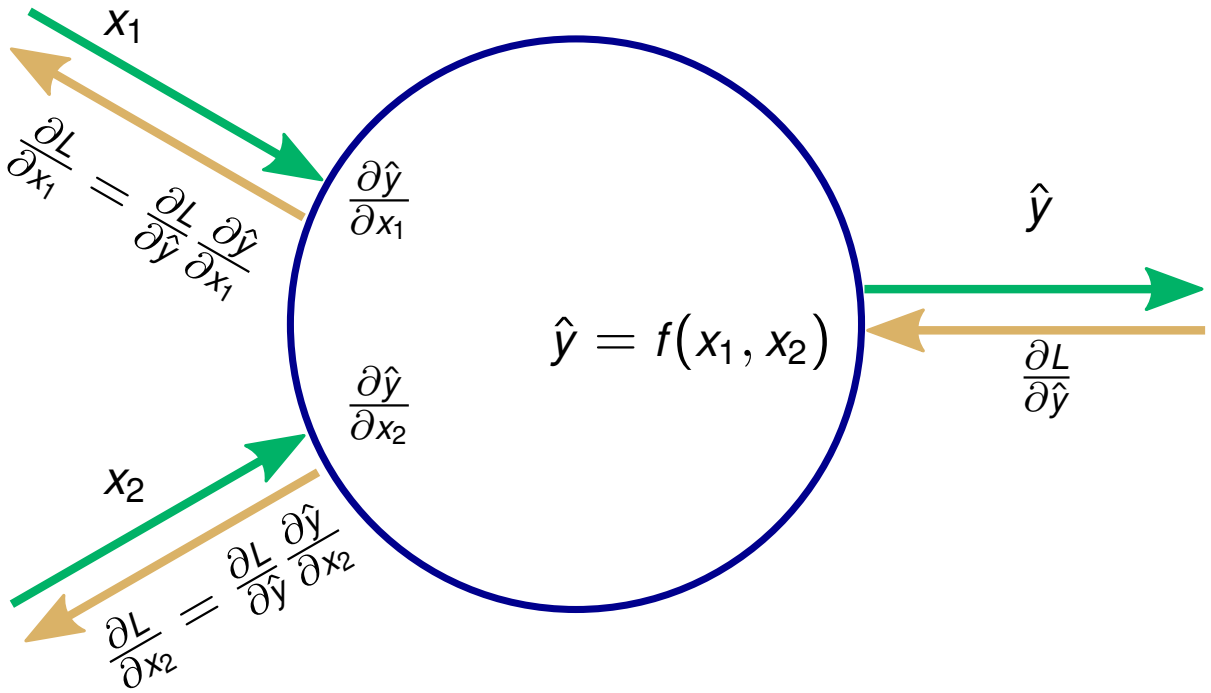
$$g(x) = \frac{1}{1 + e^{-x}}$$

$$g'(x) = g(x)(1 - g(x))$$

- smooth approximation of step function
- efficient computation of derivative
- gradient still vanishes eventually…

we will discuss other activation functions later

Backpropagation algorithm



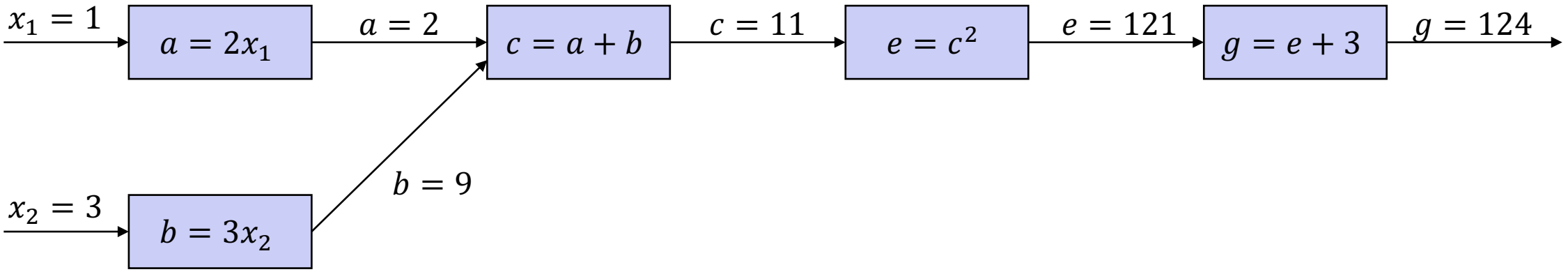
- does not show weights
- to train weights, we also require the derivatives w.r.t. the weights (for update in gradient descent)

1. Forward pass: Compute activations
2. Backward pass: Recursively apply chain rule

Backpropagation – Example / Forward Pass

- function $\hat{y} = f(\boldsymbol{x}) = f(x_1, x_2) = (2x_1 + 3x_2)^2 + 3$
- evaluate $\frac{\partial}{\partial x_1} f(1, 3)$

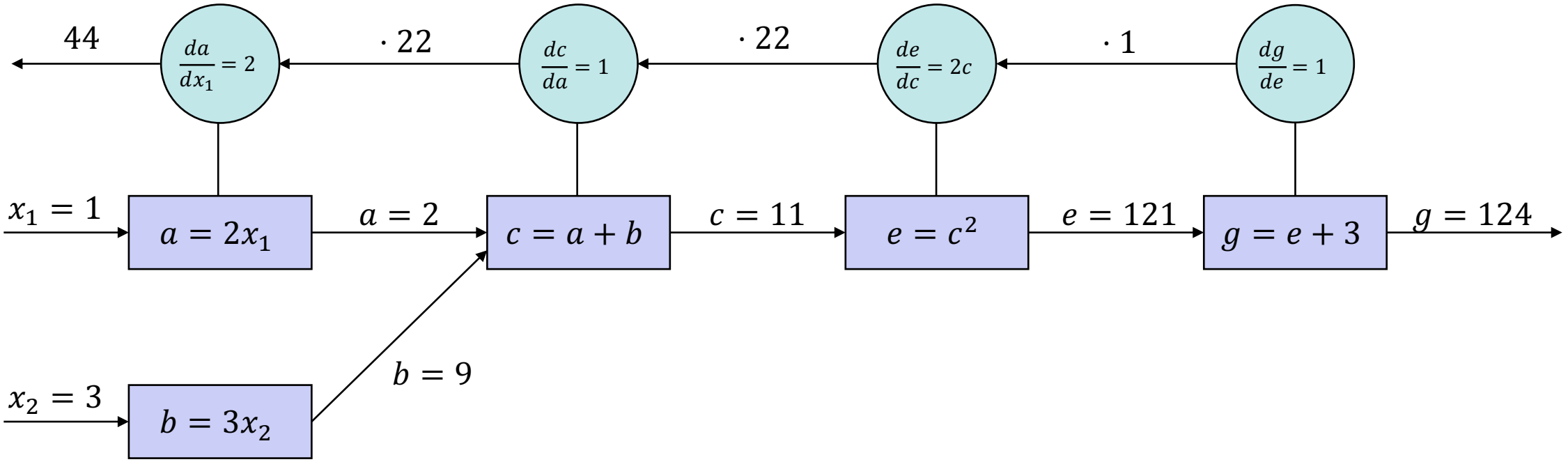
$$\frac{d}{dx} f(g(x)) = \frac{d}{dg} f(g) \cdot \frac{d}{dx} g(x)$$



Backpropagation – Example / Backward Pass

- function $\hat{y} = f(\boldsymbol{x}) = f(x_1, x_2) = (2x_1 + 3x_2)^2 + 3$
- evaluate $\frac{\partial}{\partial x_1} f(1, 3)$

$$\frac{d}{dx} f(g(x)) = \frac{d}{dg} f(g) \cdot \frac{d}{dx} g(x)$$

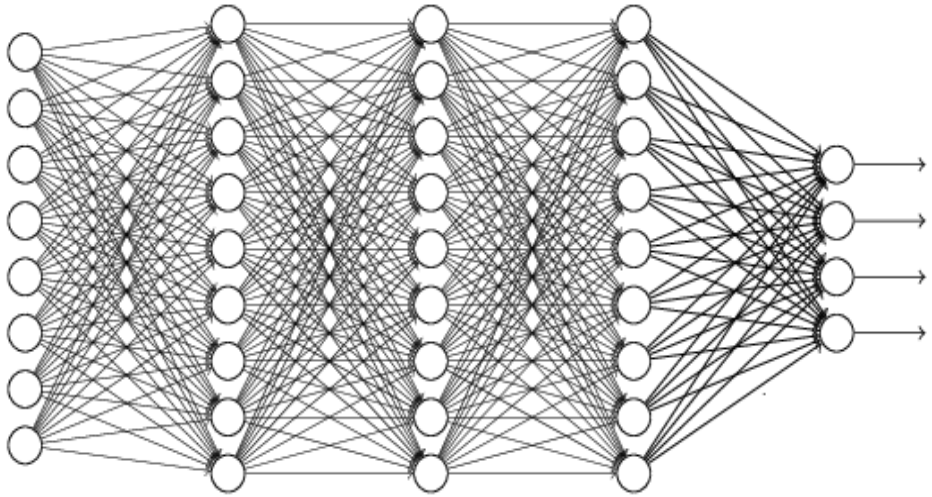


Stability

What do these two systems have in common?

https://youtu.be/esfpcnQW6qs

- Both suffer from positive feedback
- This can lead to desaster

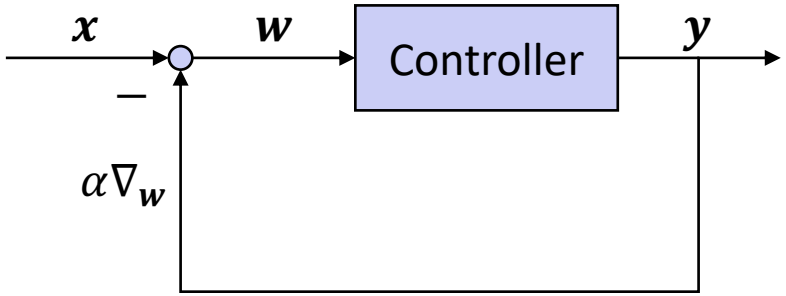


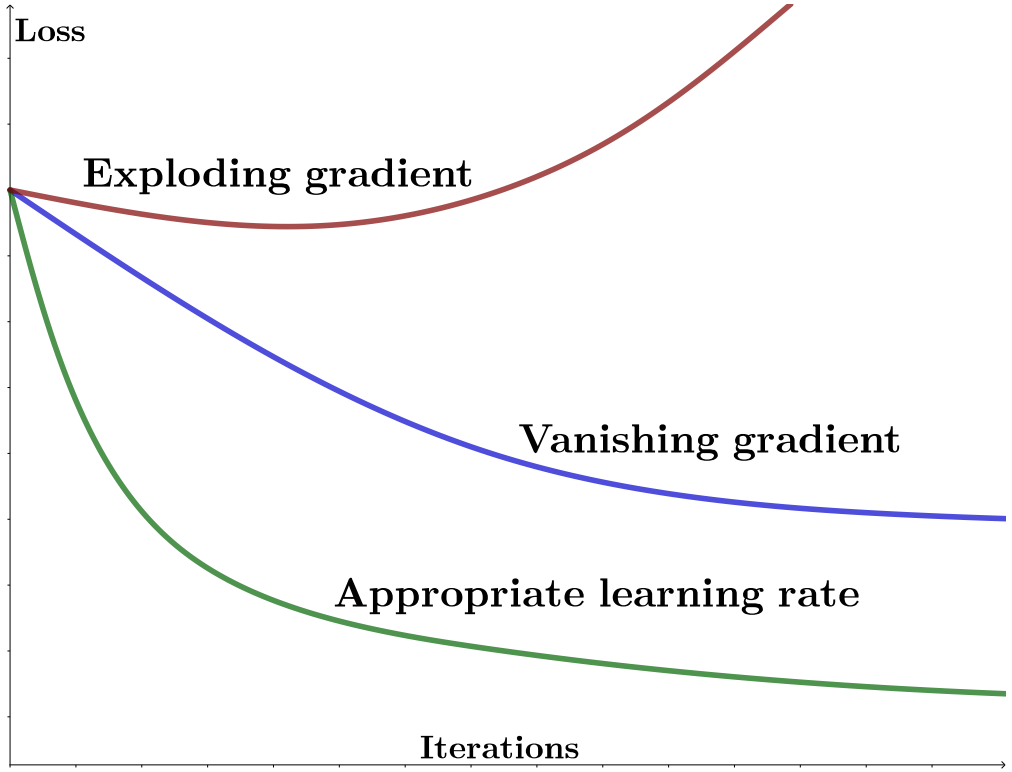
Backpropagation:
Long chains of float multiplications of potentially large and small numbers → positive feedback

Feedback Loop

Technische Hochschule Rosenheim

Analogy to control theory





- if α is too large → **positive feedback** → loss grows without bounds
- if α is too small → **negative feedback** → gradient vanishes
- choice of α is critical for learning

Backpropagation – Summary

- built around the **chain rule**
- uses a **forward pass** computing the function output
- and a **backward pass** for computing the gradient
- computationally very **efficient** by using a dynamic programming approach
  - store inputs and intermediate results during forward pass, so that they are available for backward pass

- product of partials → numerical **errors multiply**
- product of partials → **vanishing** or **exploding** gradient
  - can be mitigated to a certain degree by choosing other activation functions
    → we will discuss these later

MLP – Layer Abstraction

From Graphs of Nodes to Graphs of Layers

- we introduced layers but computed everything on individual nodes
- it is convenient to add further abstraction
  - more compact and clear notation
  - we can talk directly about gradients of entire layers
  - training is implemented on GPUs: matrix/vector notation required

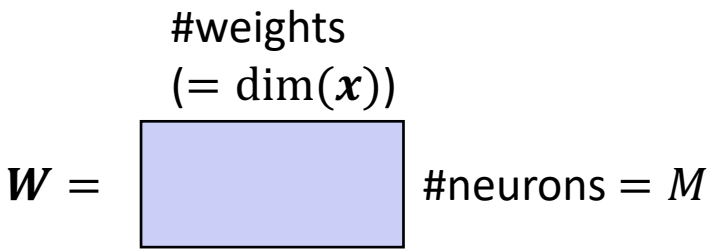
Representing Connections

Technische Hochschule Rosenheim

- recall a single neuron: $\hat{y} = g(\sum w_i x_i) = g(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x})$
- for simplicity of notation, we will use the identity mapping as activation – i.e., we omit g

- assume we have M neurons → M sets of weights $\boldsymbol{w}_m$ for $m \in \{1, \dots, M\}$.
- for each neuron we get $\hat{y}_m = \boldsymbol{w}_m^{\mathrm{T}}\boldsymbol{x}$
- we collect all outputs in a vector and all weights in a matrix:

$$\hat{\boldsymbol{y}} = \boldsymbol{W}\boldsymbol{x}$$



- this is known as a **fully connected layer**
- the non-linear activation g is then applied to each element of $\hat{\boldsymbol{y}}$

Backpropagation

Technische Hochschule Rosenheim

- we can formulate error backpropagation using matrix calculus
- the forward pass is: $\hat{\boldsymbol{y}} = \boldsymbol{W}\boldsymbol{x}$
- after the forward pass through all layers, we compute a loss that depends on our loss function L
- two gradients are required for the backward pass:
  - gradient with respect to **weights** for gradient descent: $\frac{\partial L}{\partial \boldsymbol{W}} = \frac{\partial L}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{W}} = \frac{\partial L}{\partial \hat{\boldsymbol{y}}} \boldsymbol{x}^{\mathrm{T}}$ (this is a matrix)
  - gradient with respect to **inputs** for backpropagation: $\frac{\partial L}{\partial \boldsymbol{x}} = \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{x}} \frac{\partial L}{\partial \hat{\boldsymbol{y}}} = \boldsymbol{W}^{\mathrm{T}} \frac{\partial L}{\partial \hat{\boldsymbol{y}}}$ (this is a vector)

order to be consistent with matrix multiplication

Matrix Notation – Single Layer Example

- assume we have simple network with no activation function. Forward pass: $\hat{\boldsymbol{y}} = \boldsymbol{W}\boldsymbol{x}$
- we want to find weights minimizing the following loss function (L_2 loss):

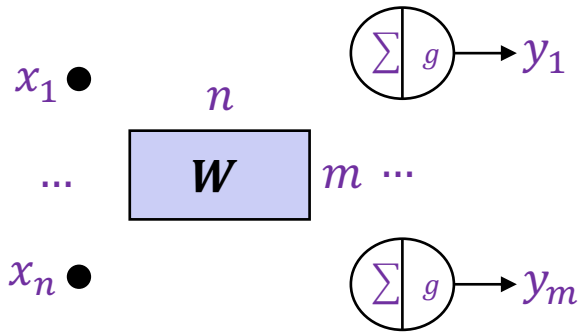
$$L(\boldsymbol{x}, \boldsymbol{W}, \boldsymbol{y}) = \frac{1}{2}\|\hat{\boldsymbol{y}} - \boldsymbol{y}\|_2^2 = \frac{1}{2}(\hat{\boldsymbol{y}} - \boldsymbol{y})^{\mathrm{T}}(\hat{\boldsymbol{y}} - \boldsymbol{y}) = \frac{1}{2}(\boldsymbol{W}\boldsymbol{x} - \boldsymbol{y})^{\mathrm{T}}(\boldsymbol{W}\boldsymbol{x} - \boldsymbol{y})$$

- we get
  - $\frac{\partial L}{\partial \hat{\boldsymbol{y}}} = \hat{\boldsymbol{y}} - \boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} - \boldsymbol{y}$
  - gradient with respect to weights:

$$\underset{m \times n}{\frac{\partial L}{\partial \boldsymbol{W}}} = \underset{m \times 1}{\frac{\partial L}{\partial \hat{\boldsymbol{y}}}} \underset{1 \times n}{\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{W}}} = (\boldsymbol{W}\boldsymbol{x} - \boldsymbol{y})\boldsymbol{x}^{\mathrm{T}}$$

  - gradient with respect to inputs:

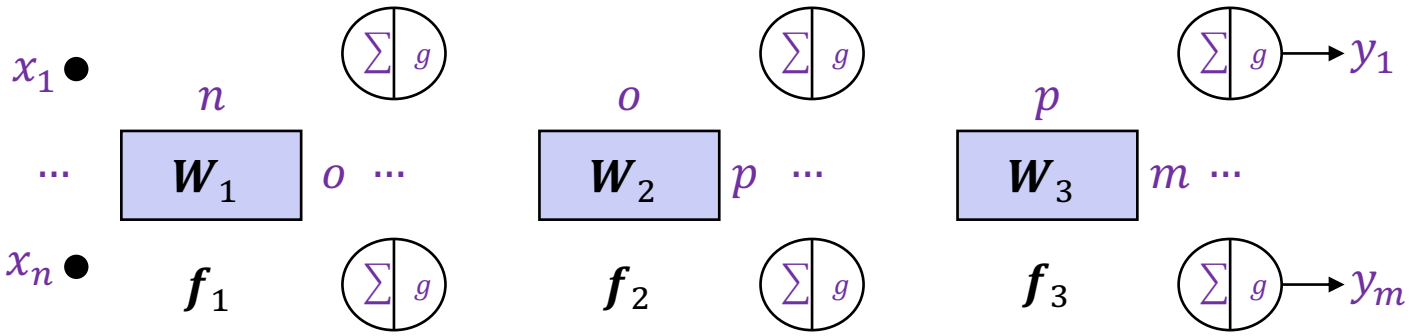
$$\underset{n \times 1}{\frac{\partial L}{\partial \boldsymbol{x}}} = \underset{n \times m}{\frac{\partial L}{\partial \hat{\boldsymbol{y}}}} \underset{m \times 1}{\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{x}}} = \boldsymbol{W}^{\mathrm{T}}(\boldsymbol{W}\boldsymbol{x} - \boldsymbol{y})$$



Linear Network in Matrix Notation

- now we add two more layers: $\hat{\boldsymbol{y}} = \boldsymbol{f}_3\left(\boldsymbol{f}_2\left(\boldsymbol{f}_1(\boldsymbol{x})\right)\right) = \boldsymbol{W}_3\boldsymbol{W}_2\boldsymbol{W}_1\boldsymbol{x}$

this is why non-linear activation is required:
without, this will collapse to a single layer!

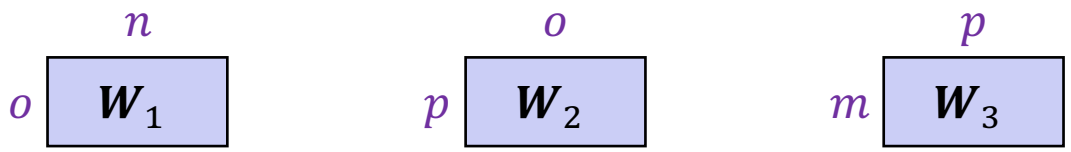


- loss function: $L(\boldsymbol{x}, \boldsymbol{W}_1, \boldsymbol{W}_2, \boldsymbol{W}_3, \boldsymbol{x}, \boldsymbol{y}) = \frac{1}{2}\|\boldsymbol{W}_3\boldsymbol{W}_2\boldsymbol{W}_1\boldsymbol{x} - \boldsymbol{y}\|_2^2$

Linear Network in Matrix Notation

- now we add two more layers: $\hat{\boldsymbol{y}} = \boldsymbol{f}_3\left(\boldsymbol{f}_2\left(\boldsymbol{f}_1(\boldsymbol{x})\right)\right) = \boldsymbol{W}_3\boldsymbol{W}_2\boldsymbol{W}_1\boldsymbol{x}$
- loss function: $L(\boldsymbol{x}, \boldsymbol{W}_1, \boldsymbol{W}_2, \boldsymbol{W}_3, \boldsymbol{x}, \boldsymbol{y}) = \frac{1}{2}\|\boldsymbol{W}_3\boldsymbol{W}_2\boldsymbol{W}_1\boldsymbol{x} - \boldsymbol{y}\|_2^2$
- gradient of last layer: $\frac{\partial L}{\partial \boldsymbol{W}_3} = \frac{\partial L}{\partial \boldsymbol{f}_3}\frac{\partial \boldsymbol{f}_3}{\partial \boldsymbol{W}_3} = (\boldsymbol{W}_3\boldsymbol{W}_2\boldsymbol{W}_1\boldsymbol{x} - \boldsymbol{y})(\boldsymbol{W}_2\boldsymbol{W}_1\boldsymbol{x})^{\mathrm{T}}$

$m \times p$ $\quad$ $m \times 1$ $\quad$ $1 \times p$

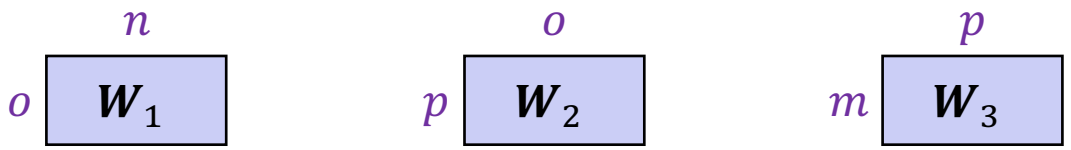


Linear Network in Matrix Notation

- now we add two more layers: $\hat{\boldsymbol{y}} = \boldsymbol{f}_3\left(\boldsymbol{f}_2\left(\boldsymbol{f}_1(\boldsymbol{x})\right)\right) = \boldsymbol{W}_3\boldsymbol{W}_2\boldsymbol{W}_1\boldsymbol{x}$
- loss function: $L(\boldsymbol{x}, \boldsymbol{W}_1, \boldsymbol{W}_2, \boldsymbol{W}_3, \boldsymbol{x}, \boldsymbol{y}) = \frac{1}{2}\|\boldsymbol{W}_3\boldsymbol{W}_2\boldsymbol{W}_1\boldsymbol{x} - \boldsymbol{y}\|_2^2$
- deeper gradients: $\frac{\partial L}{\partial \boldsymbol{W}_2} = \frac{\partial L}{\partial \boldsymbol{f}_3}\frac{\partial \boldsymbol{f}_3}{\partial \boldsymbol{W}_2} = \frac{\partial L}{\partial \boldsymbol{f}_3}\frac{\partial \boldsymbol{f}_3}{\partial \boldsymbol{f}_2}\frac{\partial \boldsymbol{f}_2}{\partial \boldsymbol{W}_2} = \boldsymbol{W}_3^{\mathrm{T}}(\boldsymbol{W}_3\boldsymbol{W}_2\boldsymbol{W}_1\boldsymbol{x} - \boldsymbol{y})(\boldsymbol{W}_1\boldsymbol{x})^{\mathrm{T}}$

order to be consistent with matrix multiplication

$p \times o$ $\quad$ $p \times m$ $\quad$ $m \times 1$ $\quad$ $1 \times o$



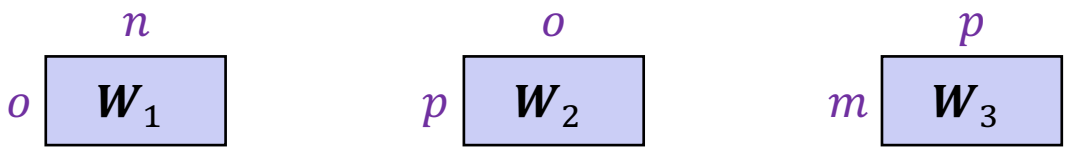
Linear Network in Matrix Notation

- now we add two more layers: $\hat{\boldsymbol{y}} = \boldsymbol{f}_3\left(\boldsymbol{f}_2\left(\boldsymbol{f}_1(\boldsymbol{x})\right)\right) = \boldsymbol{W}_3\boldsymbol{W}_2\boldsymbol{W}_1\boldsymbol{x}$
- loss function: $L(\boldsymbol{x}, \boldsymbol{W}_1, \boldsymbol{W}_2, \boldsymbol{W}_3, \boldsymbol{x}, \boldsymbol{y}) = \frac{1}{2}\|\boldsymbol{W}_3\boldsymbol{W}_2\boldsymbol{W}_1\boldsymbol{x} - \boldsymbol{y}\|_2^2$

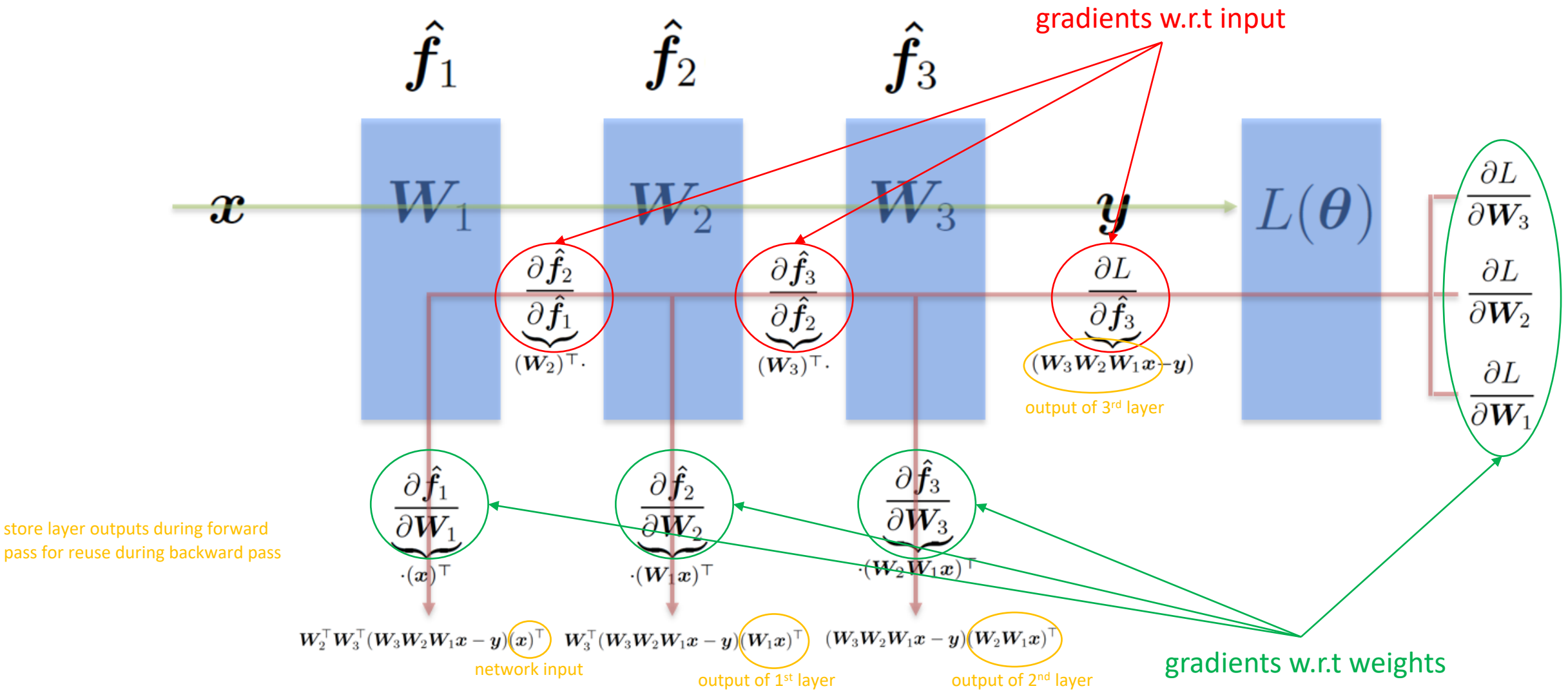
- deeper gradients:

order to be consistent with matrix multiplication

$$\underset{o \times n}{\frac{\partial L}{\partial \boldsymbol{W}_1}} = \frac{\partial L}{\partial \boldsymbol{f}_3}\frac{\partial \boldsymbol{f}_3}{\partial \boldsymbol{W}_1} = \frac{\partial L}{\partial \boldsymbol{f}_3}\frac{\partial \boldsymbol{f}_3}{\partial \boldsymbol{f}_2}\frac{\partial \boldsymbol{f}_2}{\partial \boldsymbol{W}_1} = \frac{\partial L}{\partial \boldsymbol{f}_3}\frac{\partial \boldsymbol{f}_3}{\partial \boldsymbol{f}_2}\frac{\partial \boldsymbol{f}_2}{\partial \boldsymbol{f}_1}\frac{\partial \boldsymbol{f}_1}{\partial \boldsymbol{W}_1} = \underset{o \times p}{\boldsymbol{W}_2^{\mathrm{T}}}\underset{p \times m}{\boldsymbol{W}_3^{\mathrm{T}}}\underset{m \times 1}{(\boldsymbol{W}_3\boldsymbol{W}_2\boldsymbol{W}_1\boldsymbol{x} - \boldsymbol{y})}\underset{1 \times n}{\boldsymbol{x}^{\mathrm{T}}}$$



Linear Network in Matrix Notation



Universal Approximation Theorem

Universal Approximation Theorem

A **single hidden layer** is already sufficient to be a **universal function approximator**

- Proof by [Hor89, Hor91, Cyb89]; for a more recent paper see [Kra21]
- Let $\varphi(\cdot)$ be a non-constant, bounded and monotonically increasing function
- For any $\varepsilon > 0$ and any continuous function f defined on a compact subset of $\mathbb{R}^m$ there exists an integer N, real constants v_i, b_i and real vectors $\boldsymbol{w}_i \in \mathbb{R}^m$ such that

#neurons in hidden layer

activation function

input vector

Bias

$$F(\boldsymbol{x}) = \sum_{i=1}^{N} v_i \varphi(\boldsymbol{w}_i^{\mathrm{T}} \boldsymbol{x} + b_i) \quad \text{with} \quad |F(\boldsymbol{x}) - f(\boldsymbol{x})| < \varepsilon$$

2nd layer of weights

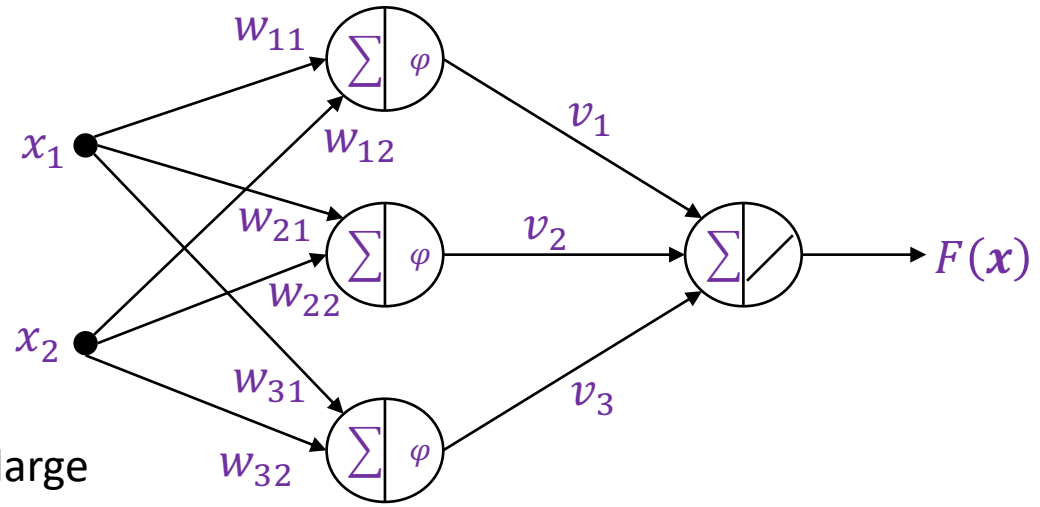
first layer of weights (input – hidden)

only a **single** hidden layer is required!

Example:

$\boldsymbol{x} = (x_1, x_2)$

$\boldsymbol{w}_1 = (w_{11}, w_{12}), \boldsymbol{w}_2 = (w_{21}, w_{22}), \boldsymbol{w}_3 = (w_{31}, w_{32})$

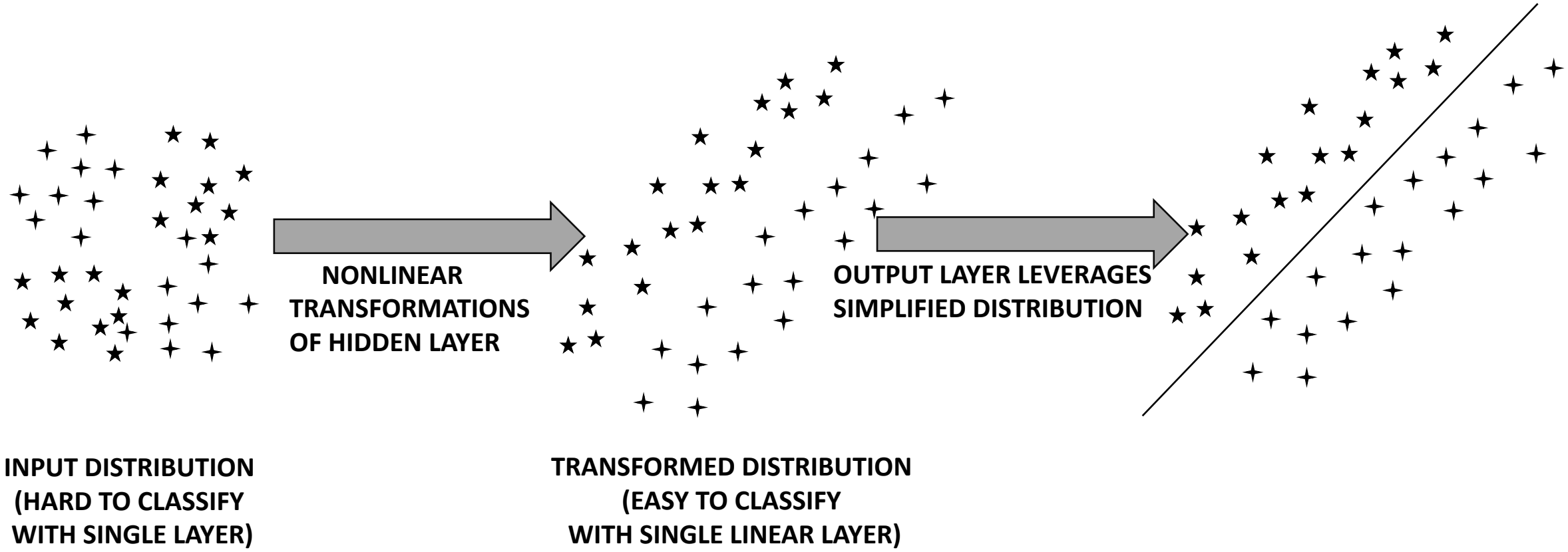


Observations:

- f can be approximated with arbitrary accuracy – just increase N
- it does not say anything about how to choose N, so ε may be quite large
- no nonlinear activation is required in the output neuron

Feature Engineering by Hidden Layer

A **single hidden layer** is also sufficient to train arbitrary (including non-contiguous) class boundaries [Mak89]



MLP – Properties

A MLP can

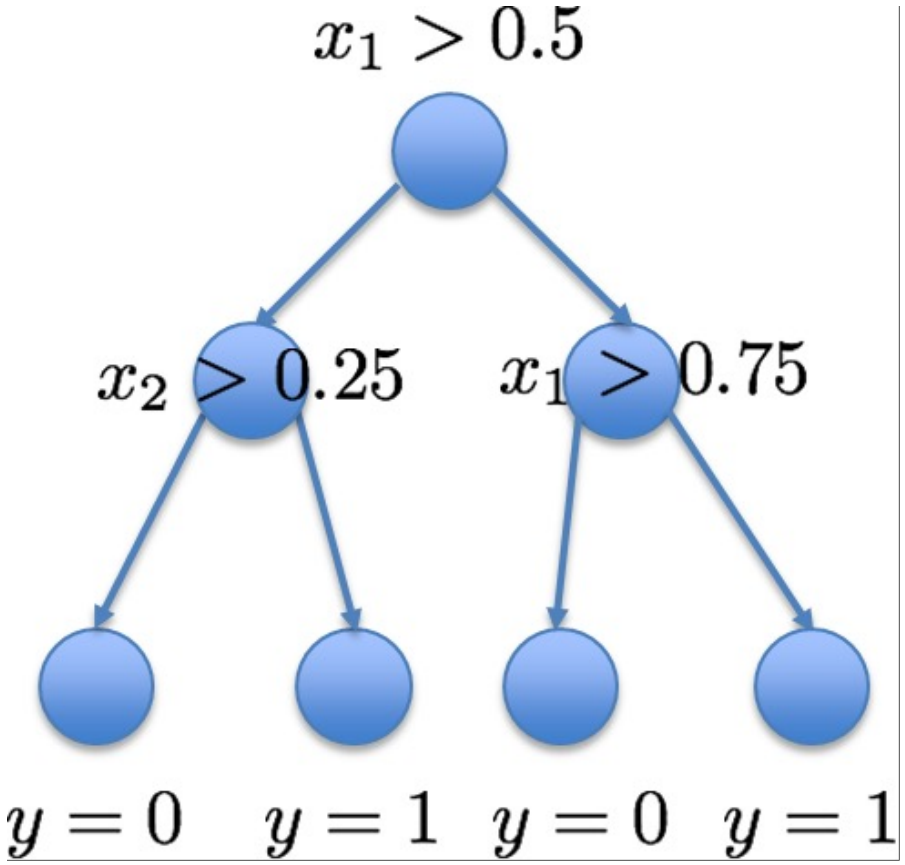
- compute any Boolean function (AND, OR, XOR, …)
- approximate any non-linear function
- define arbitrary class boundaries
- do supervised learning using a sample set (Error Backpropagation)

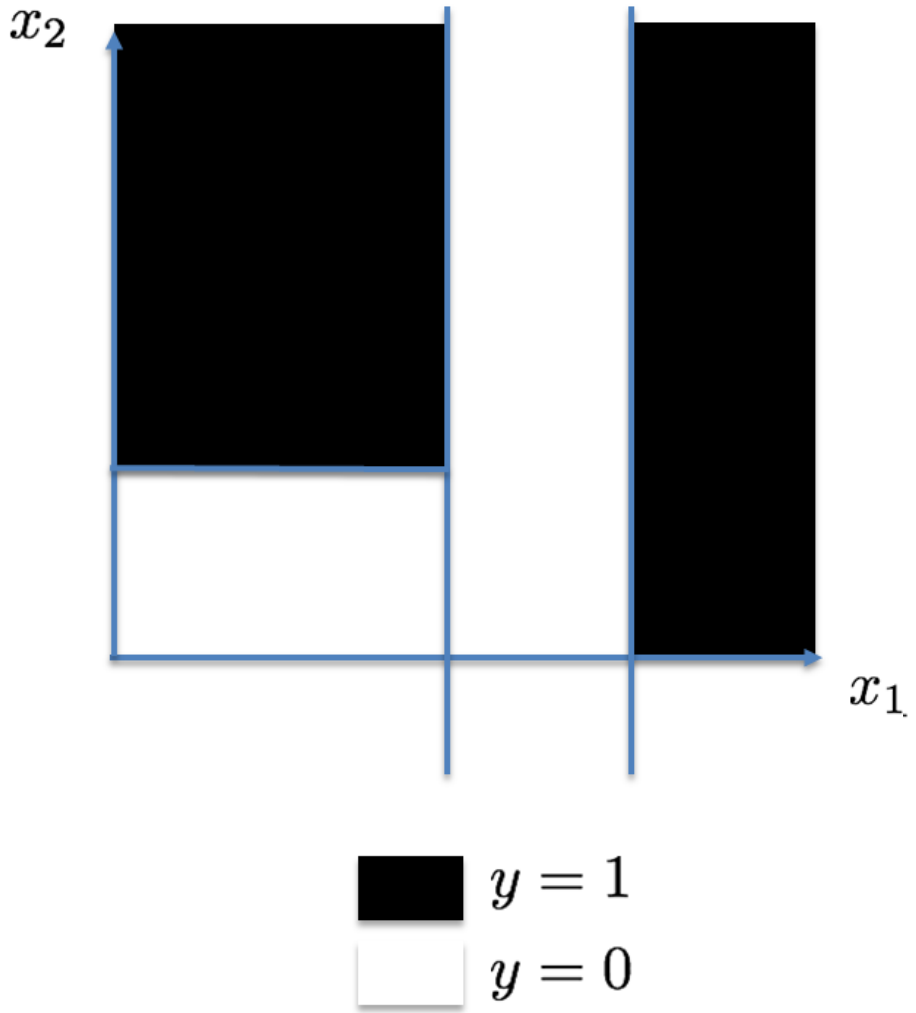
- 2 layers of weights (= 1 hidden layer of neurons) are sufficient for
  - any compact continuous function (= on closed and bounded intervals)
  - arbitrary (including non-contiguous) class boundaries [Mak89]
  - a finite number of neurons in the hidden layer suffices; however, a lot of neurons may be required in that layer
- using 3 layers of weights, the function intervals can be unbounded

So: Why would we need Deep Learning then?

A MLP is not Turing complete!

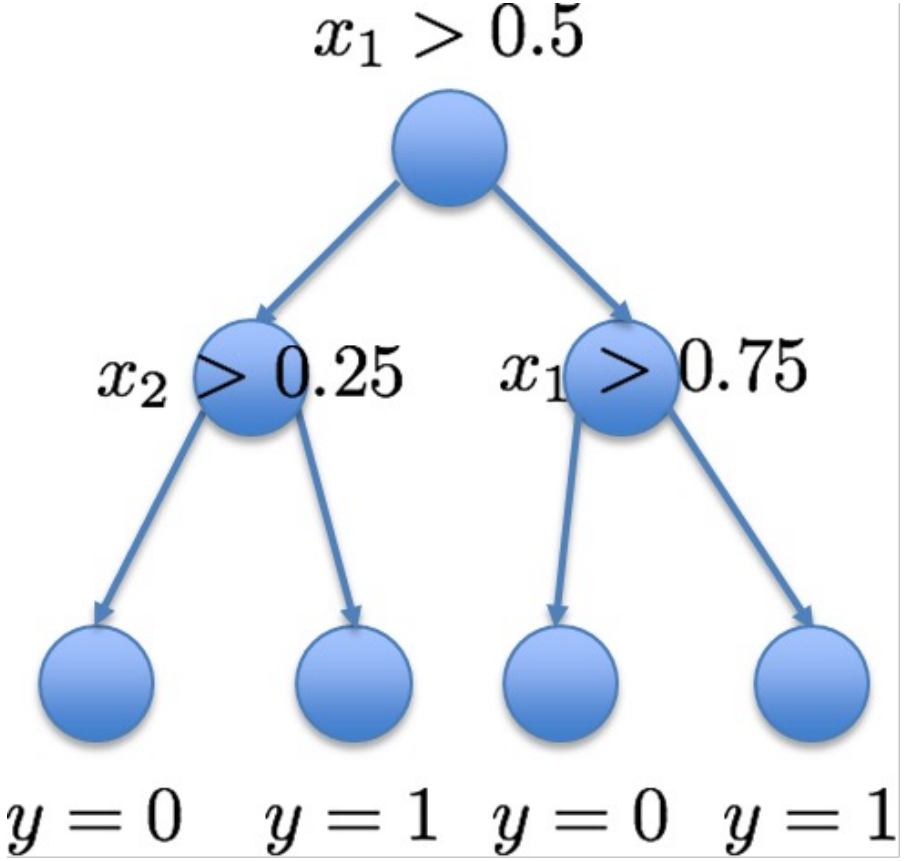
Example: Decision Tree

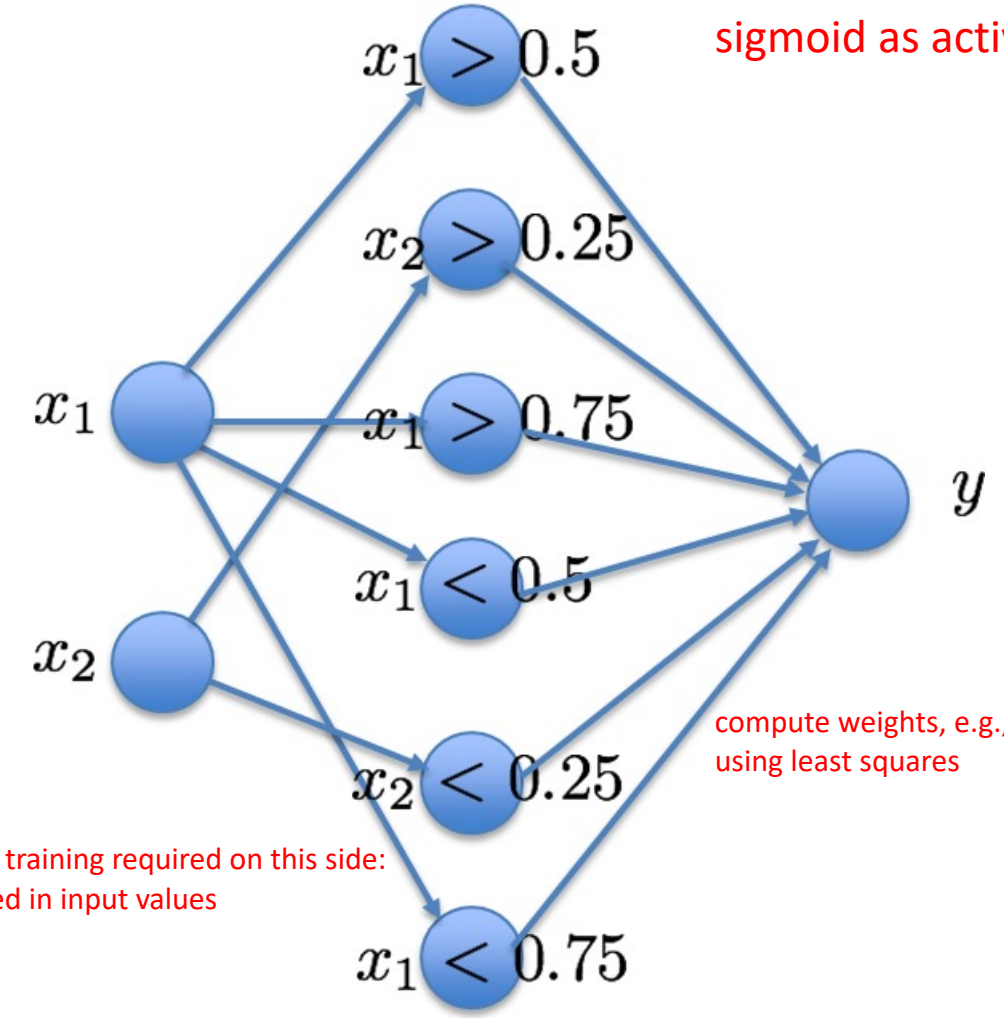




Example: Decision Tree

A neural network is an universal approximator →
We can transform the decision tree into a network:



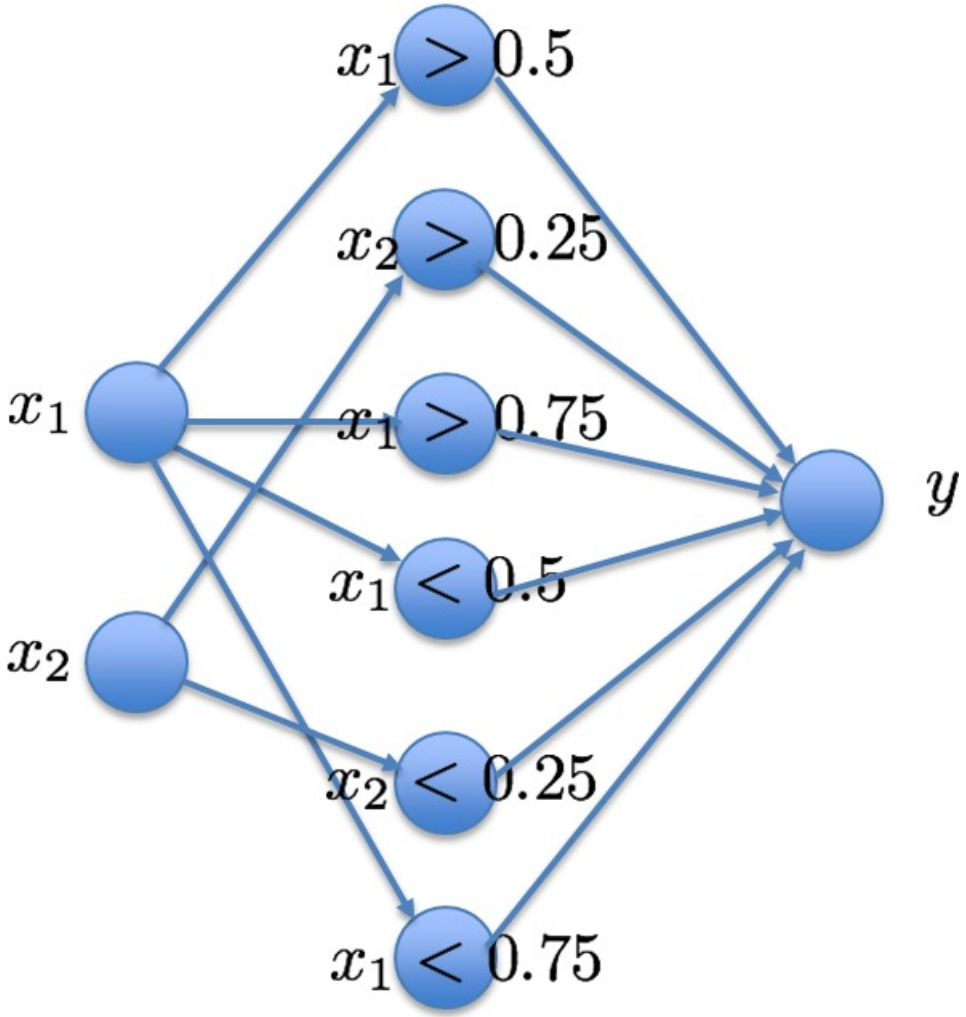


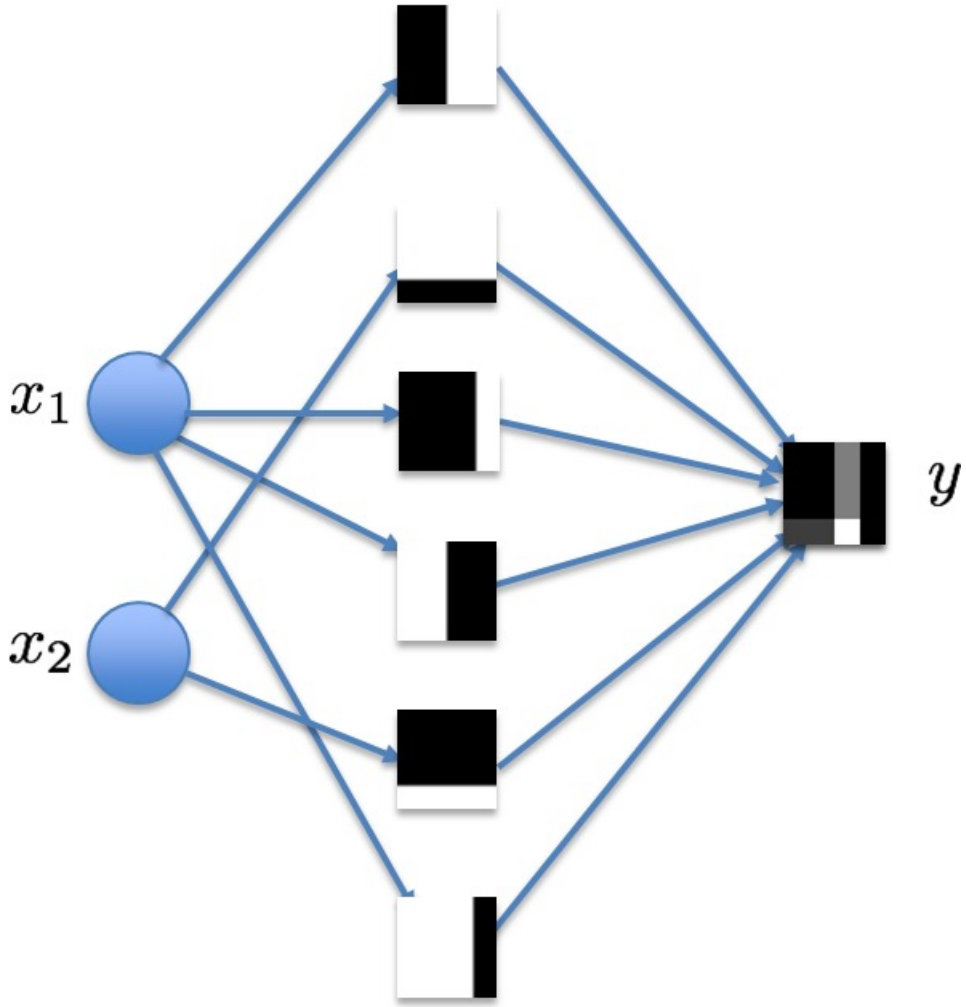
sigmoid as activation

compute weights, e.g., using least squares

no training required on this side: feed in input values

Example: Decision Tree

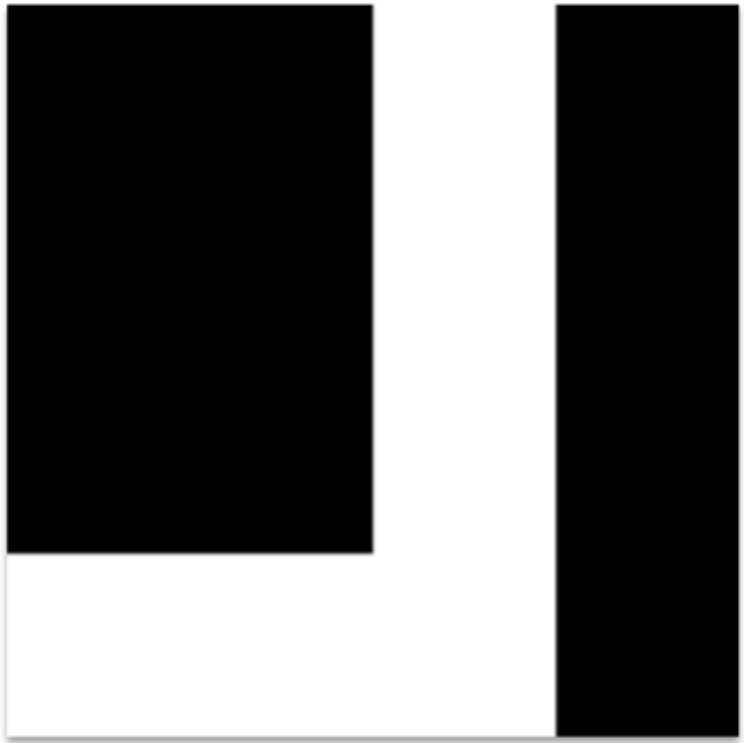




Example: Decision Tree

Technische Hochschule Rosenheim

output of decision tree

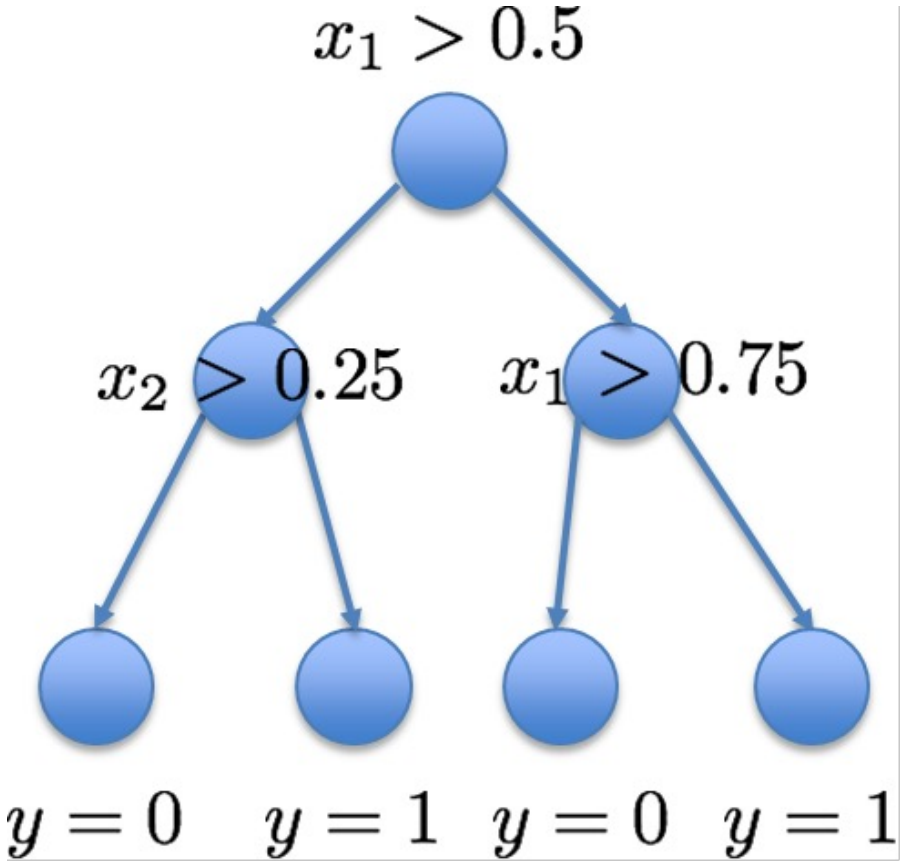


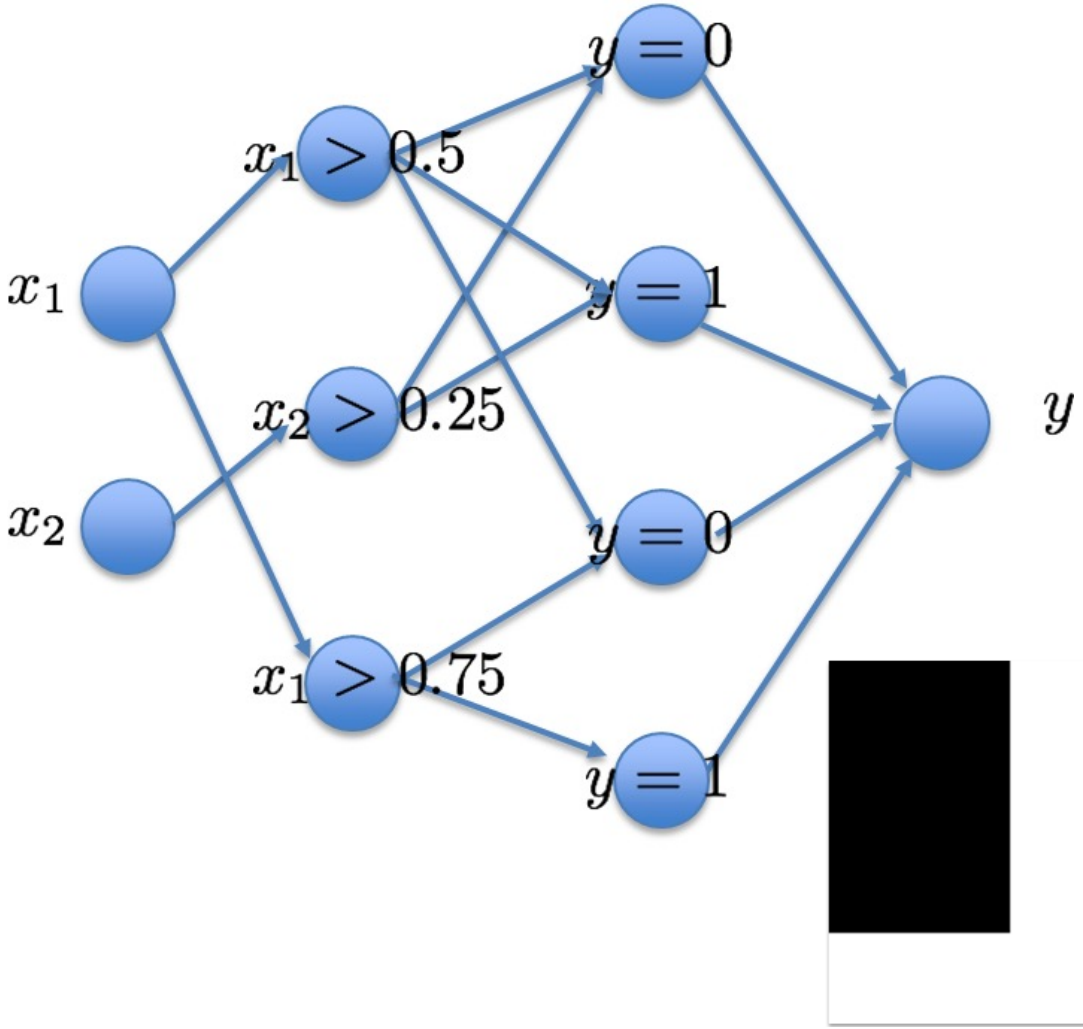
approximation of neural network with six neurons:

this is a quite hard problem – we would have to increase the number of neurons in the layer considerably to obtain a low-error approximation

Example: Decision Tree

add another layer





Depth vs. Width – Further Results

Technische Hochschule Rosenheim

- newer proofs regarding deep networks with ReLU activation [Lu17, Han17]:
  - deep networks require less neurons than flat ones (for the same approximation quality)
  - they showed that
    - any continuous function with d-dimensional input and k-dimensional output can be approximated with arbitrary accuracy
    - using $d + k$ neurons in all hidden layers.
    - less than $d + k$ neurons are never sufficient, no matter how deep the network gets.

- for practical purposes
  - more neurons and deeper networks may be desirable, as the training may converge faster (or at all).
  - strictly speaking we do not have continuous functions anyway:
    - the number of training samples are always finite,
    - we have only discrete samples of the function at certain positions

Summary

- MLPs are a universal function approximator
- gradient descent is the default training algorithm for deep learning
- backpropagation allows for computing gradients efficiently

References

[Han17] B. Hanin and M. Sellke. Approximating Continuous Functions by ReLU Nets of Minimal Width, 2017. arXiv:abs/1710.11278.

[Hor89] K. Hornik, M. Stinchcombe, and H. White. Multilayer Feedforward Networks are Universal Approximators. *Neural Networks*, 2(5):359 – 366, 1989.

[Hor91] K. Hornik. Approximation Capabilities of Multilayer Feedforward Networks. *Neural Networks*, 4(2):251 – 257, 1991.

[Cyb89] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

[Kra21] A. Kratsios. The Universal Approximation Property. *Ann. Math. Artif. Intell.* (2021).

[Lu17] Z. Lu, H. Pu, F. Wang, Z. Hu, and L. Wang. The Expressive Power of Neural Networks: A View from the Width. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan und R. Garnett, editors, Advances in Neural Information Processing Systems 30, S. 6231–6239. Curran Associates, Inc., 2017.

[Mai20] A. Maier, V. Christlein, K. Breininger, F. Denzinger, and F. Thamm: Deep Learning Slides Summer Semester 2020. Pattern Recognition Lab, Friedrich-Alexander-University Erlangen-Nürnberg. CC-BY 4.0.

[Mak89]Makhoul, El-Jaroudi, and Schwartz. Formation of disconnected decision regions with a single hidden layer. In *International 1989 Joint Conference on Neural Networks*, S. 455–460 vol.1. 1989.

[Pre07] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3. Ed., 2007.