

Implementing ANNs with TensorFlow

Session 03 - Backpropagation

Organizational Stuff

**QnA on next Friday is cancelled. Substitute on Monday!
Time and place will be announced via Mail.**

Last Week

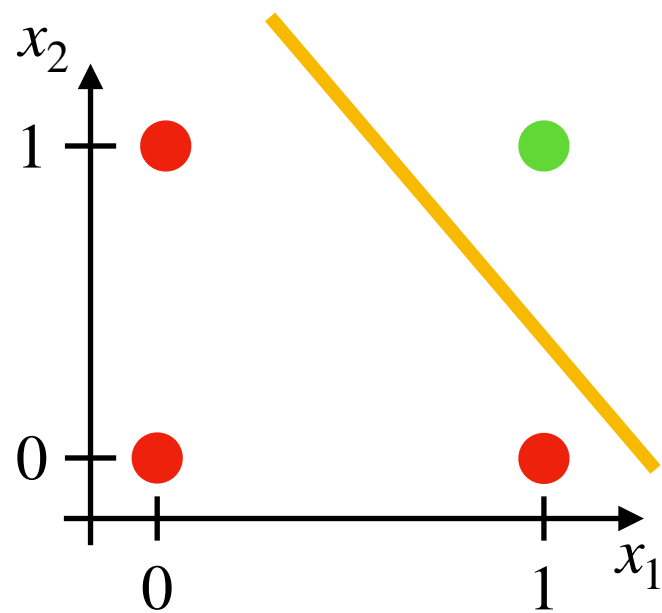
XOR Problem

- Perceptrons are not able to solve the XOR problem.
- Famously published in *Perceptrons: An introduction to computational geometry* (Minsky & Papert)
[mp]
- Why? Simply put, because a perceptron can only solve linear problems.



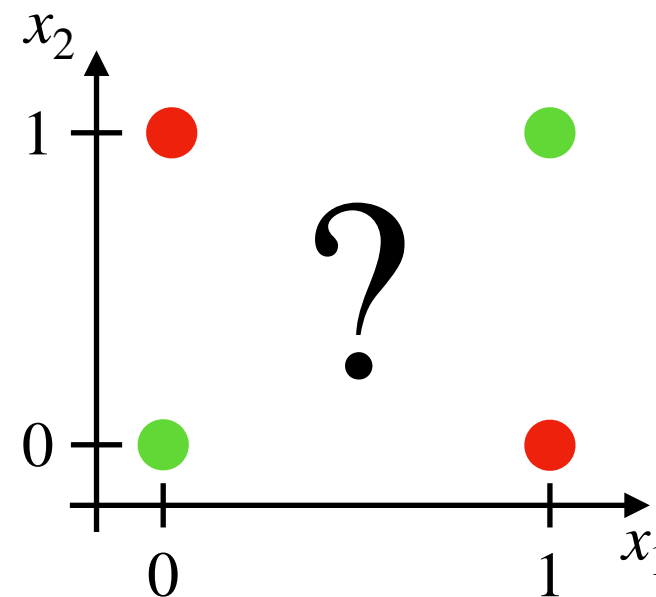
Marvin Minsky & Seymour Papert

[w1, w2]



AND

Linearly separable



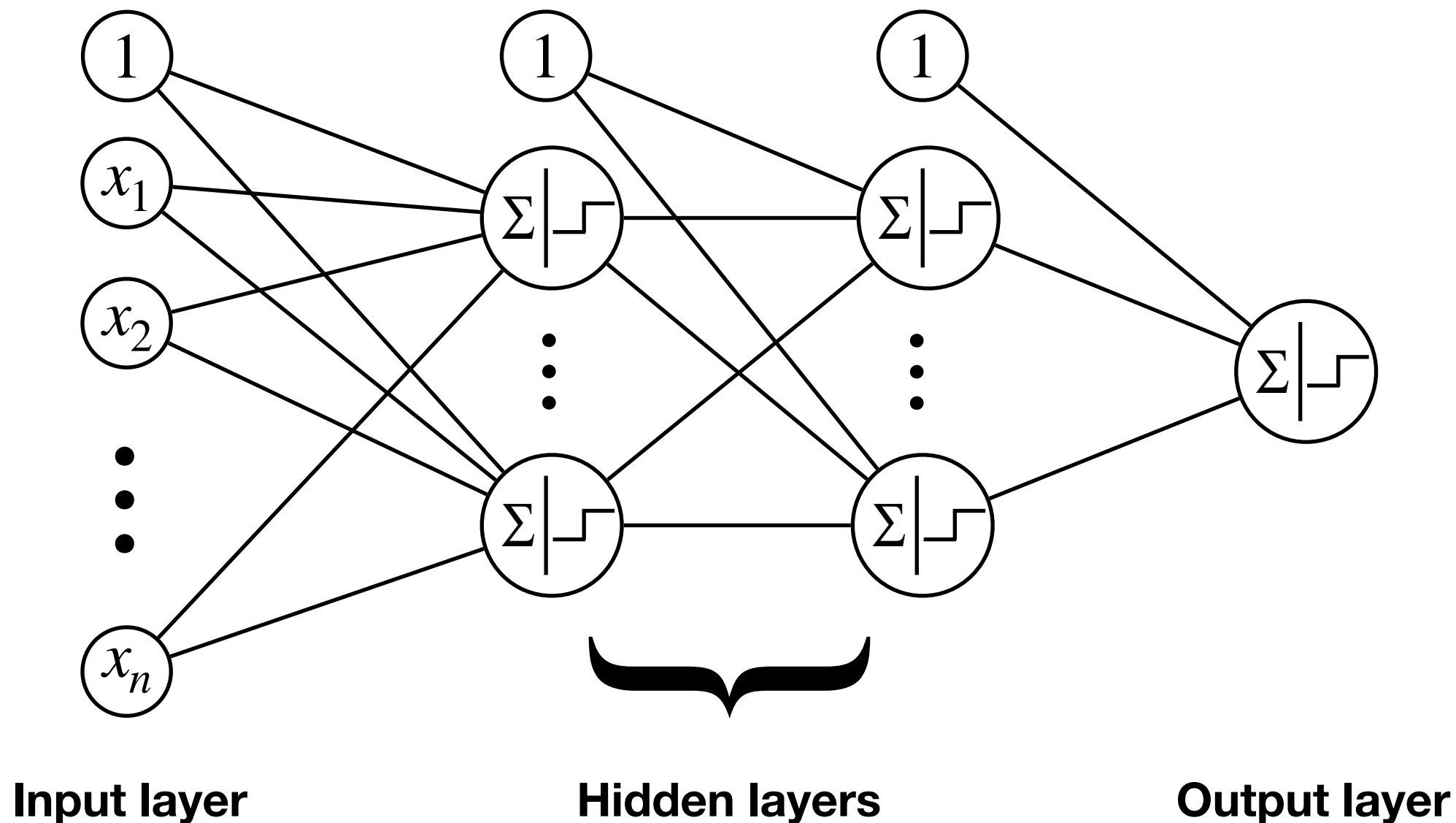
XOR

Not linearly separable



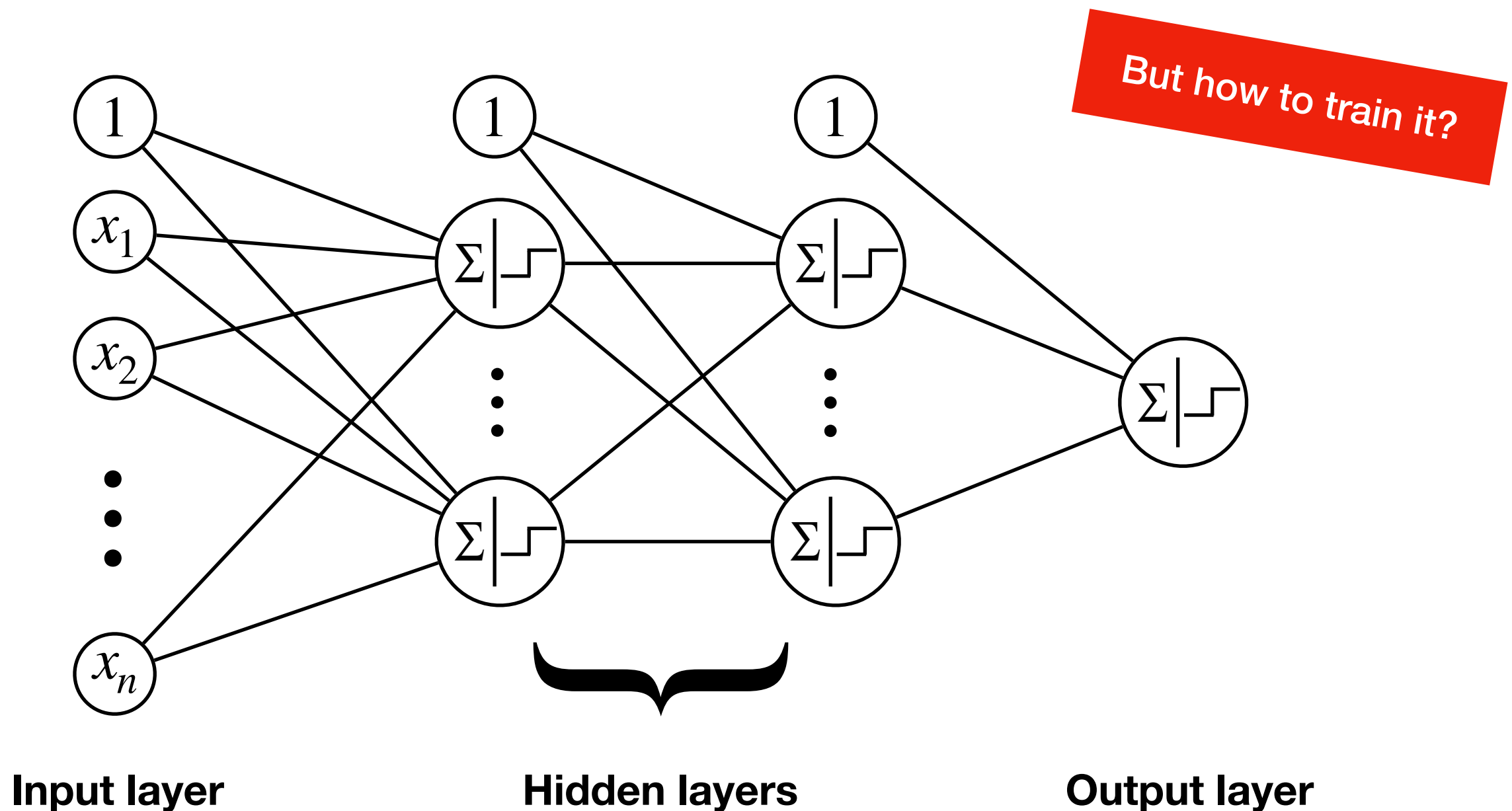
Multi-Layer Perceptron

- Stacking multiple perceptrons onto one another gives us a so called multi-layer perceptron (MLP), which would be able to solve the XOR gate.



Multi-Layer Perceptron

- Stacking multiple perceptrons onto one another gives us a so called multi-layer perceptron (MLP), which would be able to solve the XOR gate.



Agenda

1. Supervised Learning
2. Gradient Descent
3. Backpropagation

Supervised Learning

Supervised Learning

- We will start with the most common training regime for artificial neural networks: supervised learning.
- It means that we give labeled training samples to the network, such that it can learn the desired mapping.

E.g.

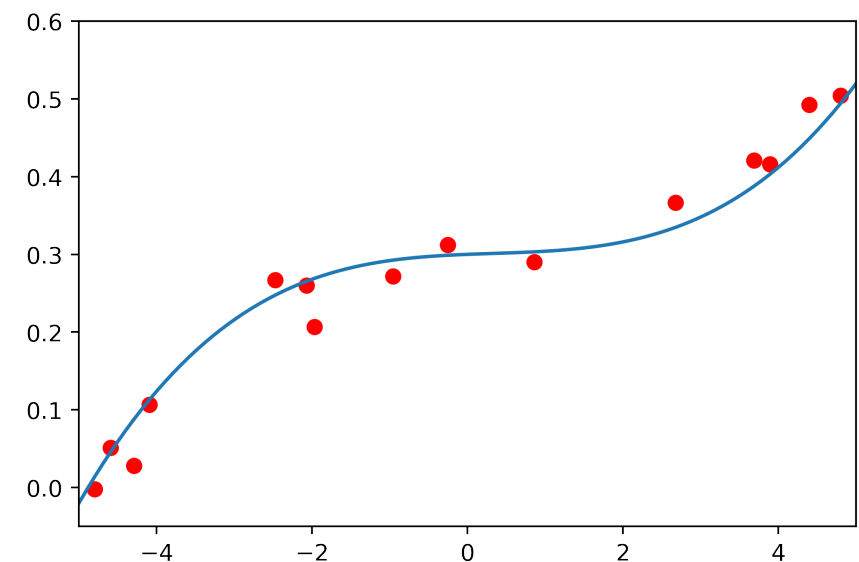
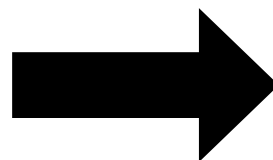
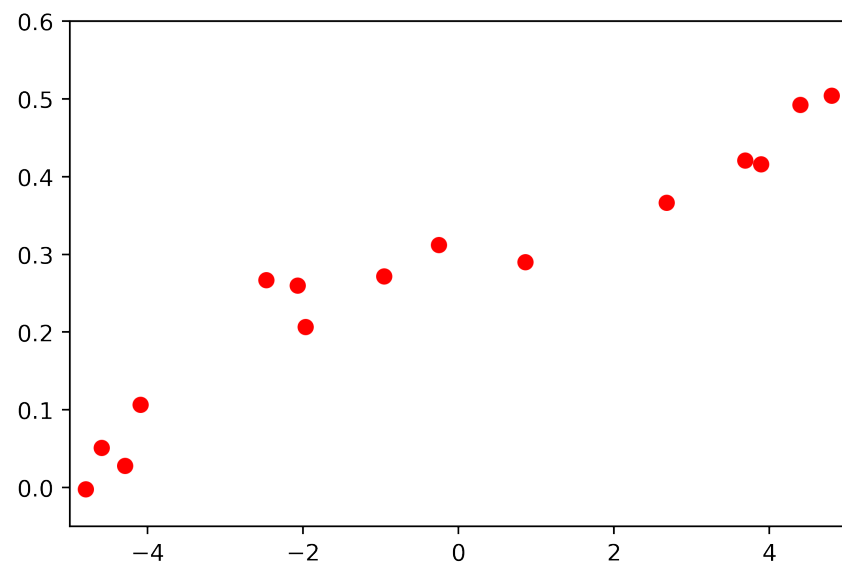
Logical XOR

x_1	x_2	$x_1 \oplus x_2$
0	0	0
1	0	1
0	1	1
1	1	0

**This is a classification task.
Although we used it to motivate
learning in perceptrons we will
switch the task setup for this
lecture!**

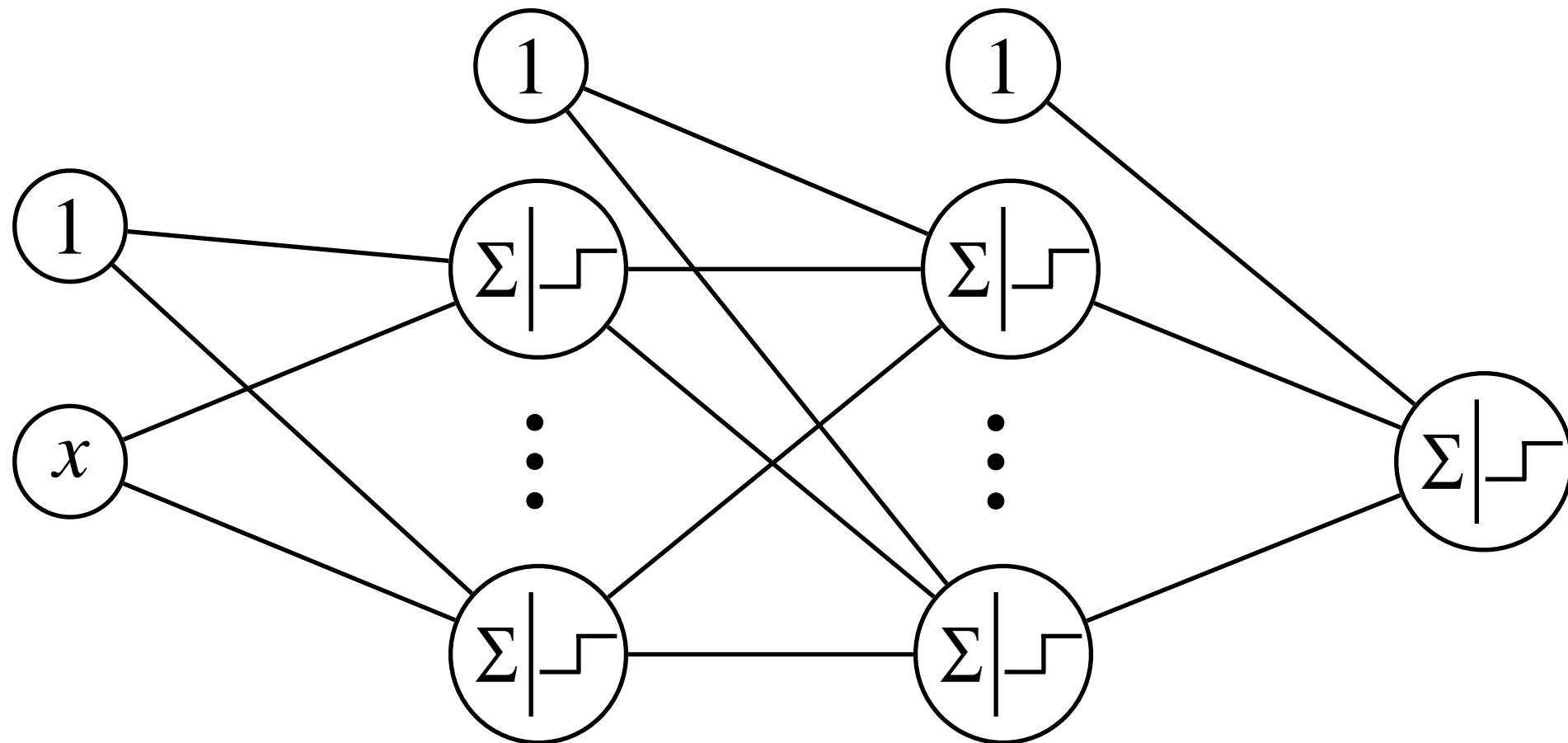
Regression

- Let's have a look at a simple task setup: regression.
- Regression is the task to learn an underlying function from a set of given points.



Multi-Layer Perceptron

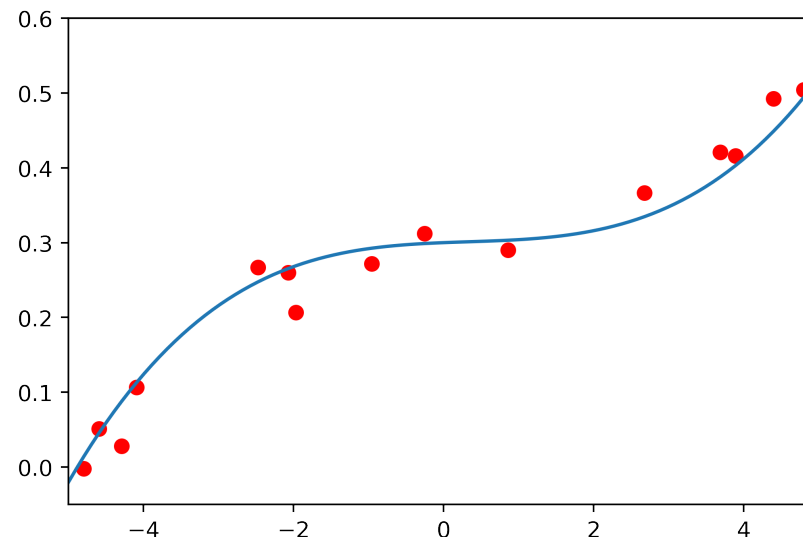
To solve this task we can use a multi-layer perceptron.



But we have to make one last change!

A New Activation Function

- We need a different activation function. Why?
 1. We want to approximate a smooth continuous function that is not binary as the logical gates.

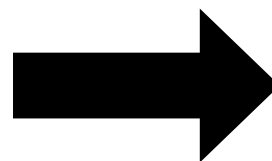
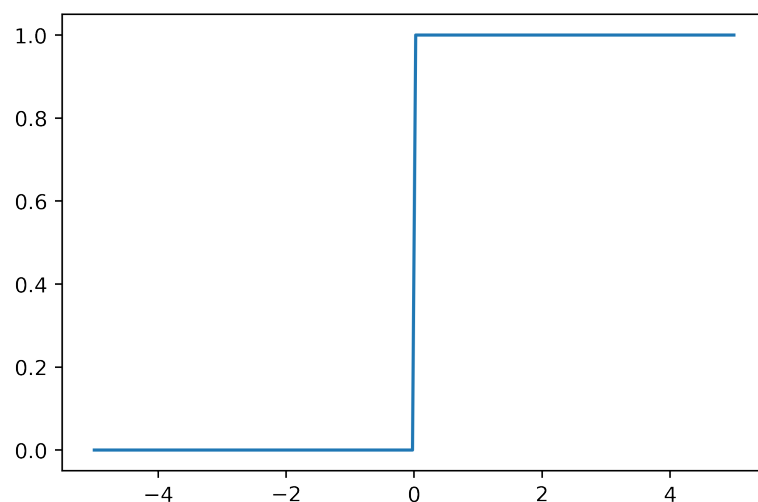


2. For the backpropagation algorithm we will need a differentiable function!

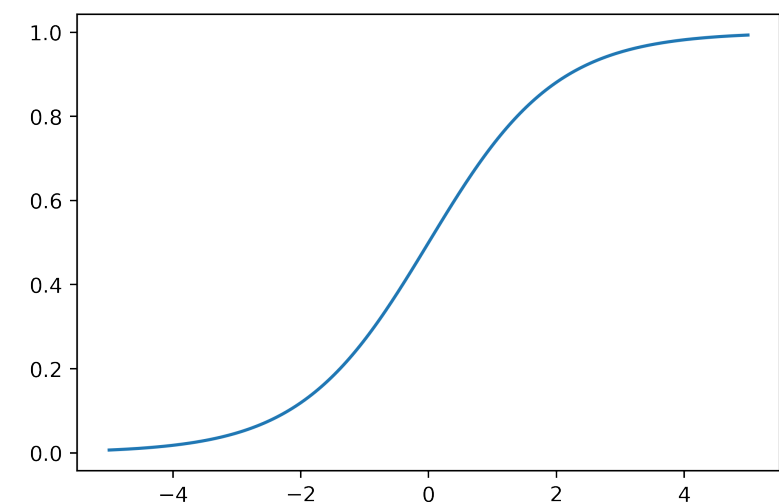
Logistic Function

- The logistic function is an activation function that can be viewed as a smooth continuous version of the step function.
- There are other activation functions, which will be covered in later lectures.

Heaviside step function



Logistic function

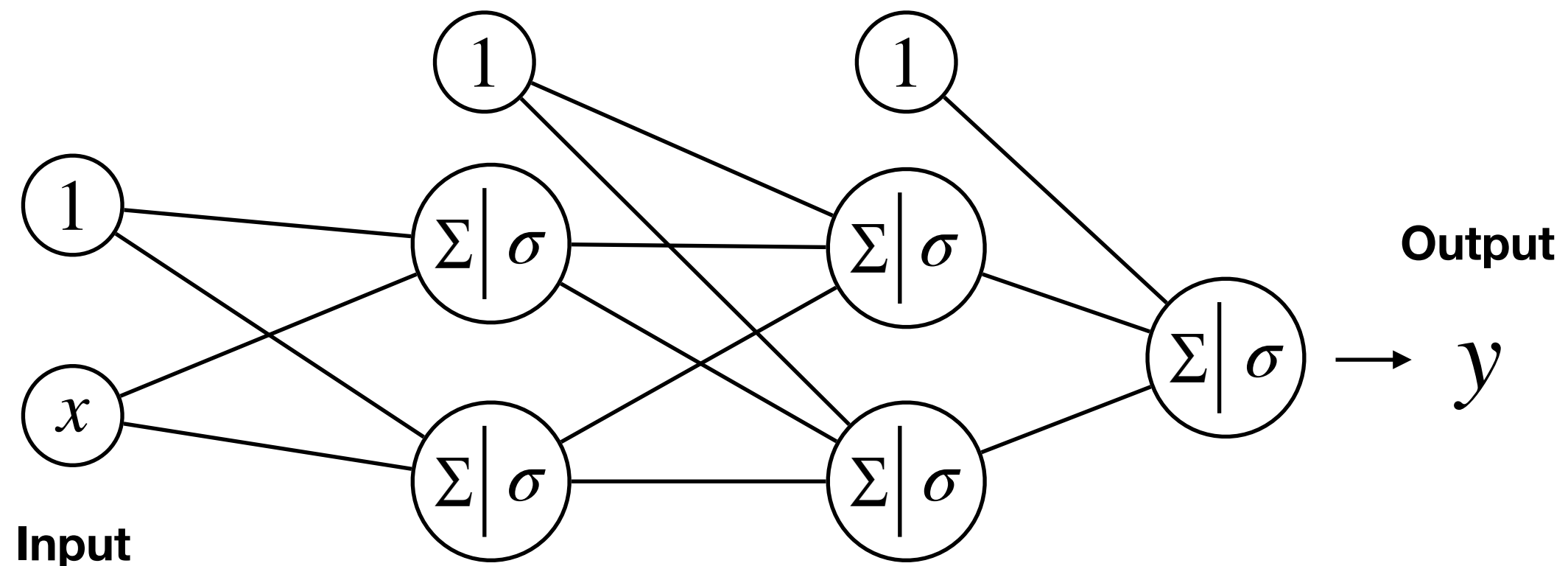
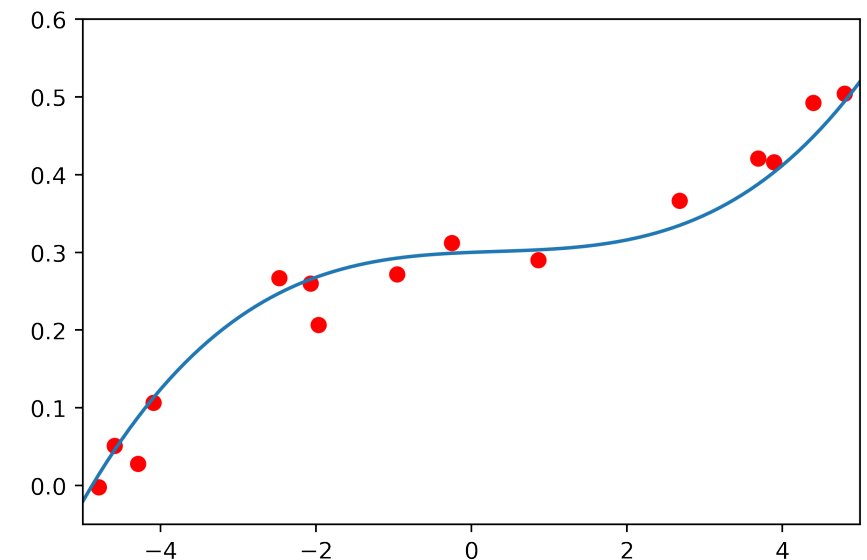


$$\sigma(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{else} \end{cases}$$

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

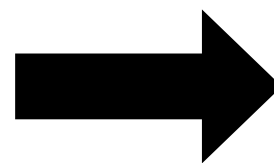
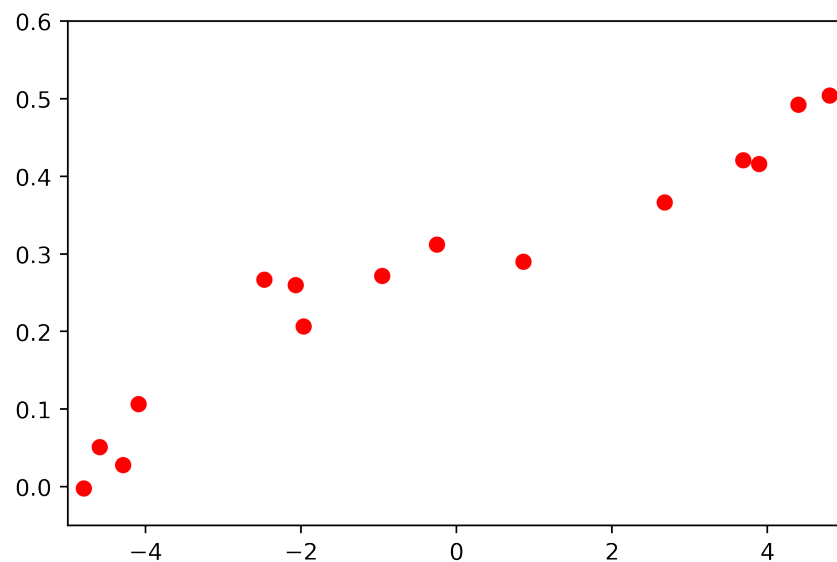
Multi-Layer Perceptron

- This is the multi-layer perceptron that we want to approximate the underlying function.



Supervised Learning

- We want to train this neural network to solve the presented regression task as good as possible.
- The network is provided with the labeled dataset $\{(x_i, t_i)\}_{i=1}^N$, consisting out of N pairs of input x_i and target t_i .

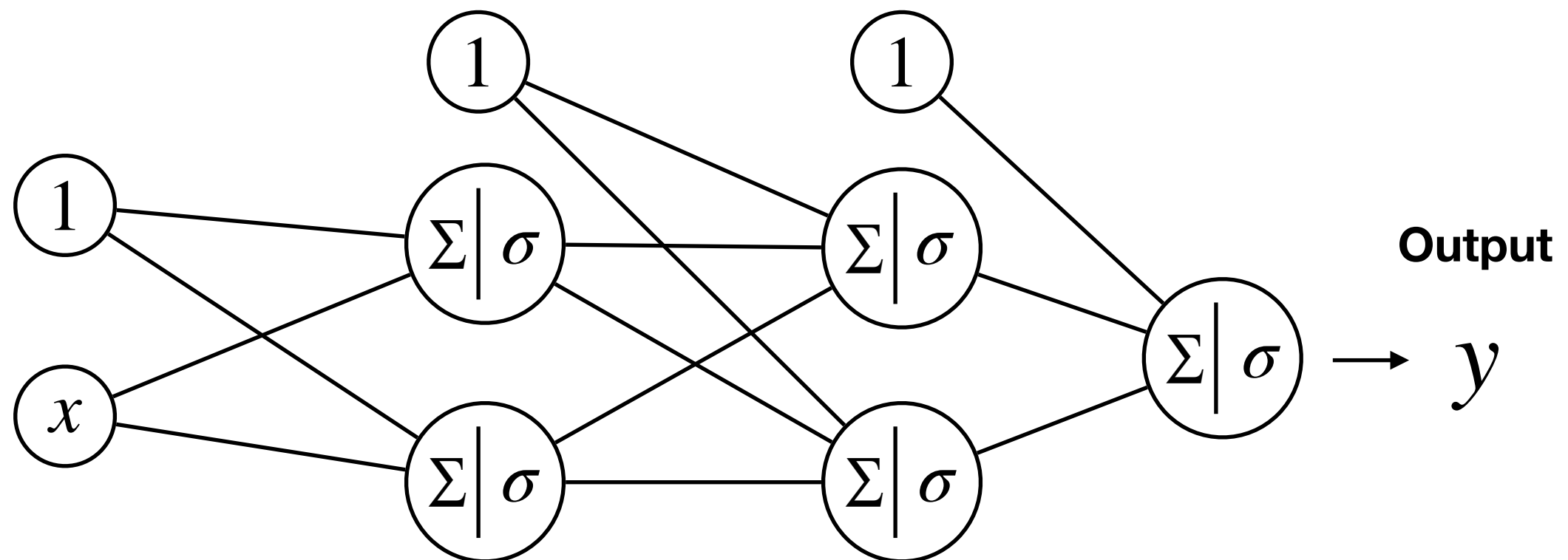


Dataset

$\{(-4.8, 0.0), (-4.5, 0.5), (-4.3, 0.3), \dots\}$

Multi-Layer Perceptron

The network is defined through all its variables (weights and biases). We summarize them as parameters θ .



$$\vec{w}^{(1)} \in \mathbb{R}^2$$

$$\vec{b}^{(1)} \in \mathbb{R}^2$$

$$W^{(2)} \in \mathbb{R}^{2 \times 2}$$

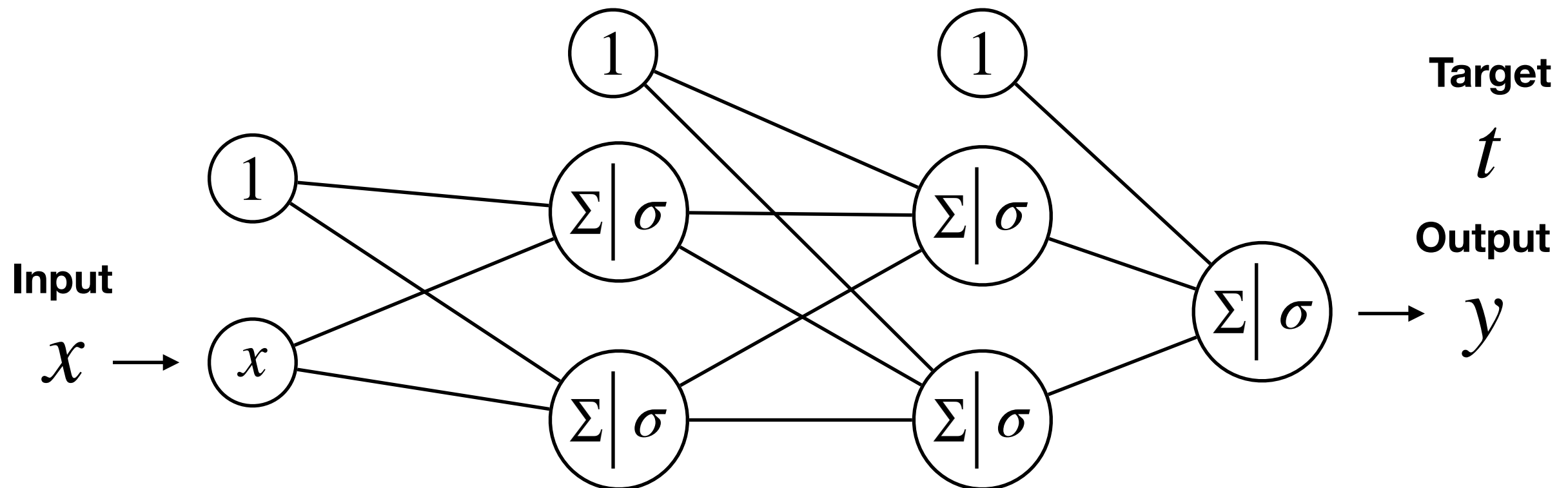
$$\vec{b}^{(2)} \in \mathbb{R}^2$$

$$\vec{w}^{(3)} \in \mathbb{R}^2$$

$$\vec{b}^{(3)} \in \mathbb{R}$$

The Goal

- Our goal is to find the parameters θ of the network such that for all inputs the output matches the target!
- But what does “match” mean?



Loss Function

Loss Function

- For that purpose we define the loss function.
- The loss function computes how bad the network's performance is.
- By minimizing the loss function we therefore maximize the performance of the network.
- E.g. for regression we can use the MSE (mean squared error - average error on all samples in the dataset):

$$L_{\theta} = \frac{1}{N} \sum_{i=1}^N \frac{1}{2} (t_i - y_i)^2$$

Target Output

Minimizing the Loss Function

How to minimize the loss function L_θ ?

$$L_\theta = \frac{1}{N} \sum_{i=1}^N \frac{1}{2} (t_i - y_i)^2$$
$$= \frac{1}{N} \sum_{i=1}^N \frac{1}{2} (t_i - \underbrace{(\sigma(\vec{w}^{(3)}) \sigma(W^{(2)}) \sigma(\vec{w}^{(1)} x_i + \vec{b}^{(1)}) + \vec{b}^{(2)}) + \vec{b}^{(3)})}_{\text{Network function}}))^2$$

Minimizing the Loss Function

How to minimize the loss function? $L_{\theta} = \frac{1}{N} \sum_{i=1}^N \frac{1}{2} (t_i - y_i)^2$

Calculate the first derivative of L_{θ} and set it to zero?

Minimizing the Loss Function

How to minimize the loss function?

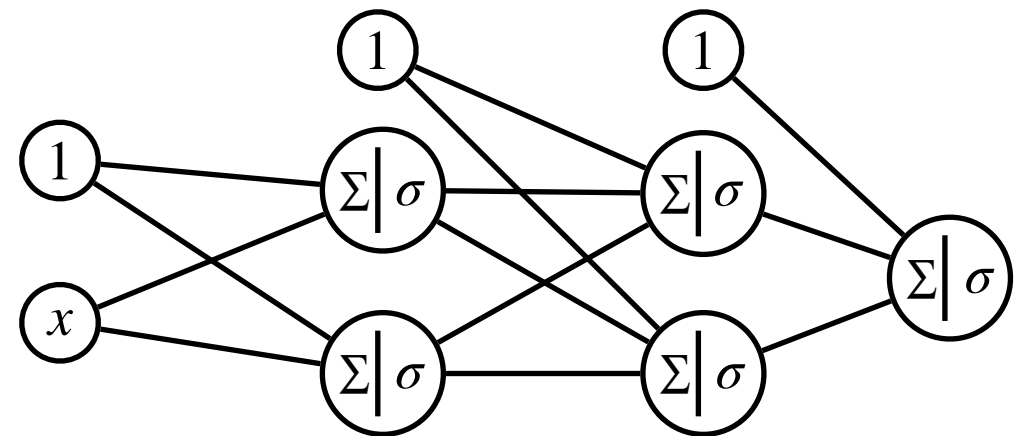
$$L_{\theta} = \frac{1}{N} \sum_{i=1}^N \frac{1}{2} (t_i - y_i)^2$$

Calculate the first derivative of L_{θ} and set it to zero?



This simple network already has 13 variables. You would have to differentiate L_{θ} in respect to all thirteen variables and then solve the resulting system of equations (with thirteen (non-linear) equations).

Too complex!



Minimizing the Loss Function

How to minimize the loss function? $L_{\theta} = \frac{1}{N} \sum_{i=1}^N \frac{1}{2} (t_i - y_i)^2$

Calculate the explicit loss values for certain combinations of weights and then take the best combination?

Minimizing the Loss Function

How to minimize the loss function?

$$L_{\theta} = \frac{1}{N} \sum_{i=1}^N \frac{1}{2} (t_i - y_i)^2$$

Calculate the explicit loss values for certain combinations of weights and then take the best combination?

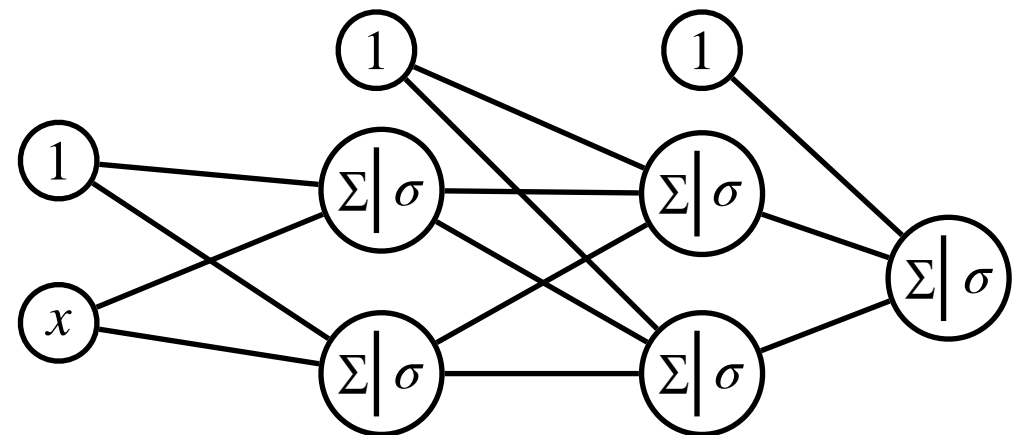


This network has 13 variables.
Say you would like to check for
50 different values for each
variable. This would result in:

$$50^{13} = 10.220.703.125. \\ 000.000.000.000$$

combinations.

Too much!



Minimizing the Loss Function

How to minimize the loss function?

$$L_{\theta} = \frac{1}{N} \sum_{i=1}^N \frac{1}{2} (t_i - y_i)^2$$

Minimize in an iterative manner!

Minimizing the Loss Function

How to minimize the loss function?

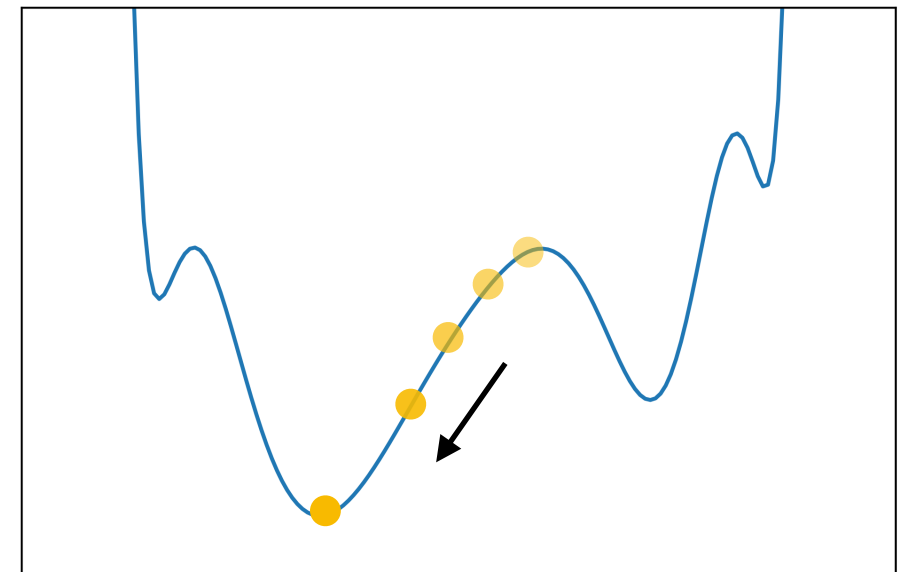
$$L_{\theta} = \frac{1}{N} \sum_{i=1}^N \frac{1}{2} (t_i - y_i)^2$$

Minimize in an iterative manner!



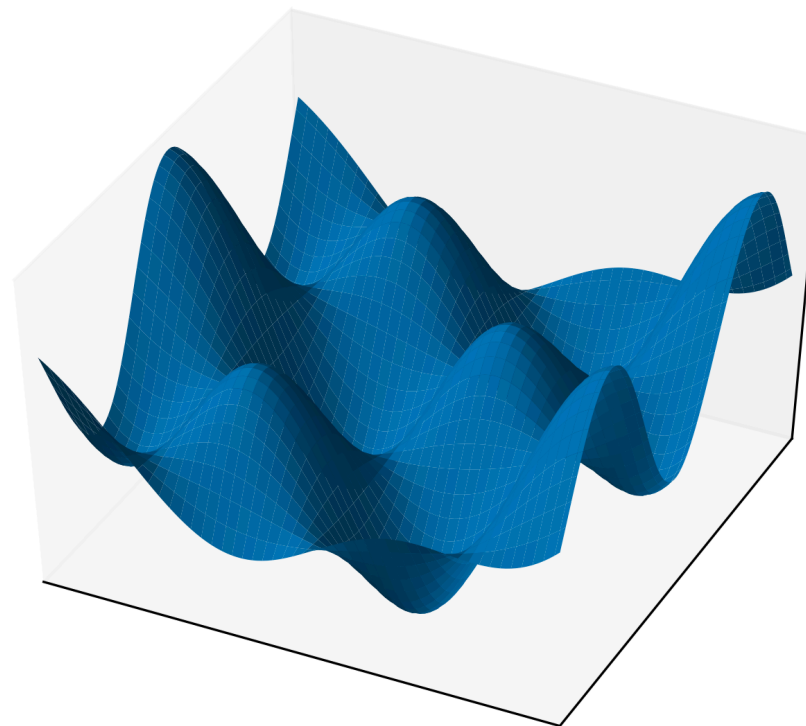
We can start at a random point and use calculus to compute the direction in which the loss function decreases the steepest.

Through many iterative updates along the steepest path we can reach a minimum.



Loss Surface

- If we talk and think about the function on which we want to find the minimum we usually talk about the loss surface.
- On the other hand the term loss function rather describes the chosen loss (e.g. mean squared error).
- On the loss surface we perform the iterative minimization.



Gradient Descent

Gradient

How do you compute the direction of the steepest decrease in a high-dimensional function?

Gradient - Example

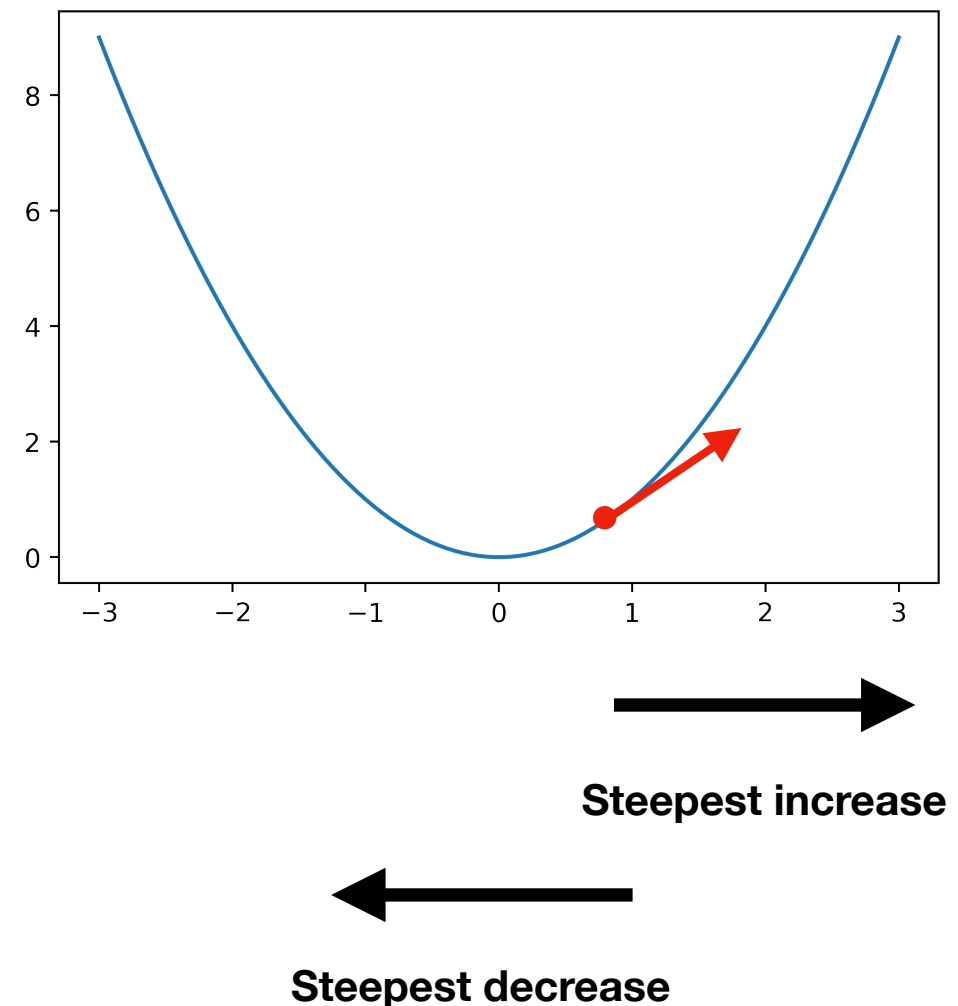
Example in 1-D

Consider the function $f(x) = x^2$.

The derivative $f'(x) = 2x$ computes the slope of the function at a given point x_0 .

E.g. the slope at point $x_0 = 0.8$ is $f'(x_0) = 1.6$.

Now that means that the steepest increase of the function is in the positive direction. The opposite direction has the steepest decrease accordingly.



Gradient

Consider a high-dimensional function:

$$f : \mathbb{R}^n \rightarrow \mathbb{R}, (x_1, \dots, x_n) \rightarrow f(x_1, \dots, x_n)$$

The derivate of a high dimensional function is called the gradient. The gradient is a high-dimensional vector containing the partial derivatives in regard to all variables.

$$\nabla f = \left(\frac{\partial f}{\partial x_1} \frac{\partial f}{\partial x_2} \cdots \frac{\partial f}{\partial x_n} \right)$$

Gradient- Example

Example in 2-D

Consider the function $f(x, y) = xy + 3x^2$.

The partial derivatives of this function are

$$\frac{\partial}{\partial x} f = y + 6x \qquad \frac{\partial}{\partial y} f = x$$

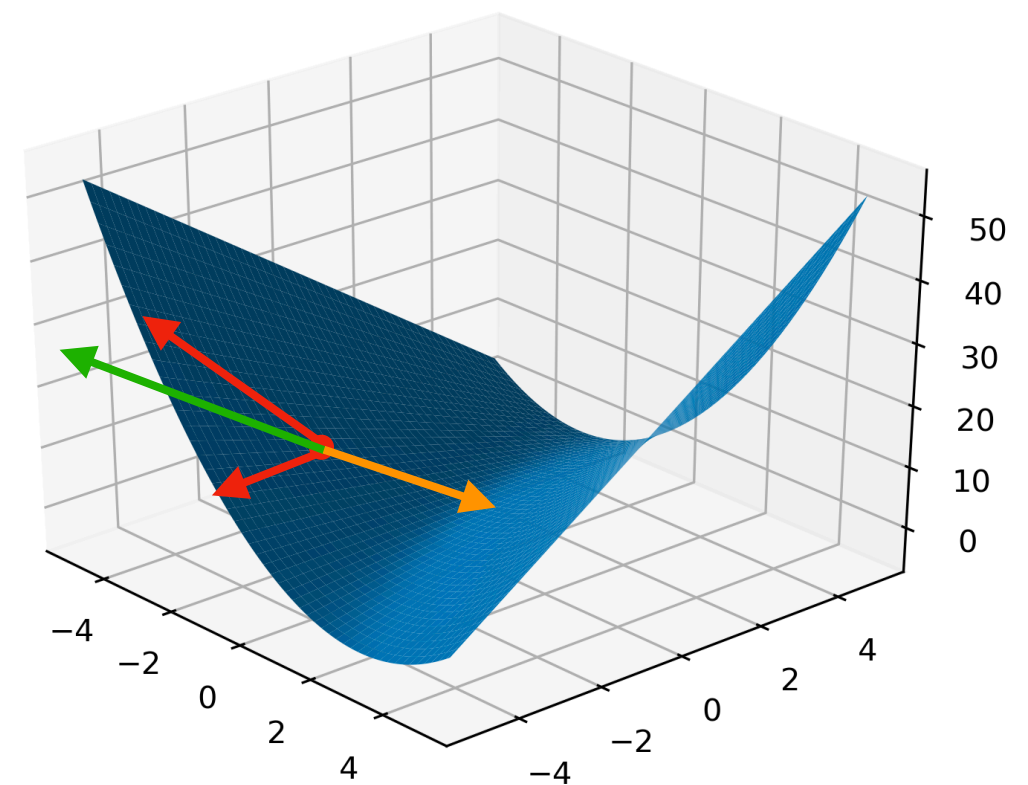
Resulting in the gradient

$$\nabla f = (y + 6x \quad x)$$

The gradient also shows in the direction of the steepest increase: e.g. in

$(x_0, y_0) = (-2, -2)$ this direction is

$$\nabla f(-2, -2) = (-14 \quad -2)$$



Partial derivatives

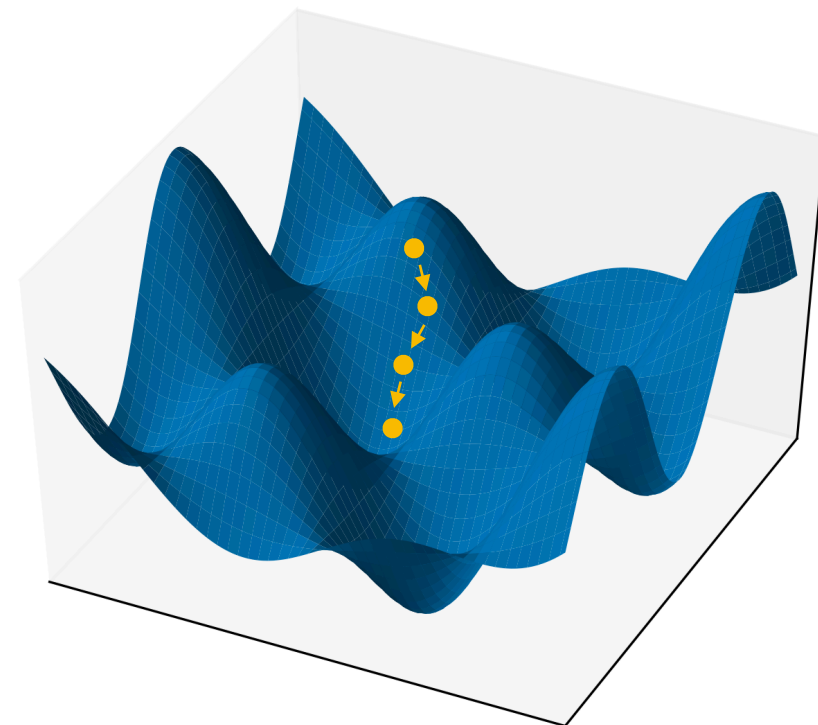
Gradient = steepest increase

-Gradient = steepest decrease

Gradient Descent

- For a given combination of parameters we can now compute the direction of the steepest decrease by computing the gradient of the loss surface.
- Through iterative updates we can reach a minimum.

1. Random initialization of parameters θ .
2. Compute loss value L_{θ} .
3. Compute gradient ∇L_{θ} .
4. Update parameters:
$$\theta_{new} = \theta_{old} - \gamma \nabla L_{\theta}.$$
5. Go back to 2.



Gradient Descent Update Rule

1. Random initialization of parameters θ .
2. Compute loss value L_θ .
3. Compute gradient ∇L_θ .
4. Update parameters:
$$\theta_{new} = \theta_{old} - \gamma \nabla L_\theta.$$
5. Go back to 2.

A closer look at the update rule:

Parameters before update

$\theta_{new} = \theta_{old} - \gamma \nabla L_\theta$

Parameters after update

Descent

Learning Rate

Gradient

The diagram illustrates the gradient descent update rule. At the top, the text 'Parameters before update' has an arrow pointing to the θ_{old} term in the equation $\theta_{new} = \theta_{old} - \gamma \nabla L_\theta$. Below the equation, the text 'Parameters after update' has an arrow pointing to the θ_{new} term. The minus sign in the equation is labeled 'Descent' with an arrow. The γ term is labeled 'Learning Rate' with an arrow. The ∇L_θ term is labeled 'Gradient' with an arrow.

Backpropagation

Backpropagation

- How can we compute the gradient of the loss function for all the parameters in a multi-layer perceptron?
- The solution to this is an algorithm called backpropagation, invented by Paul Werbos (1974, 1982).

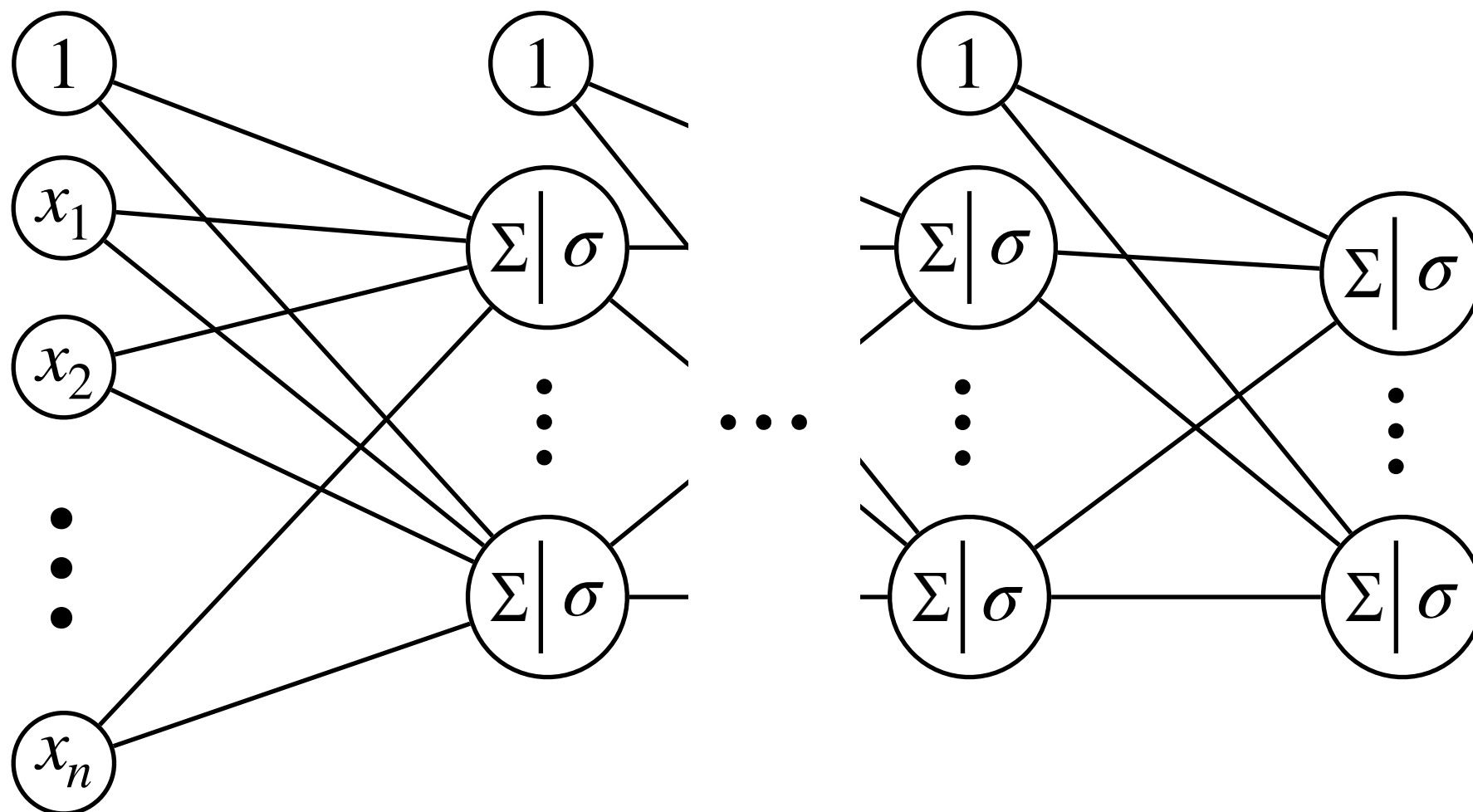
Not available due
to copyright
issues.

Paul Werbos

WARNING
This is going to be mathy!

Backpropagation

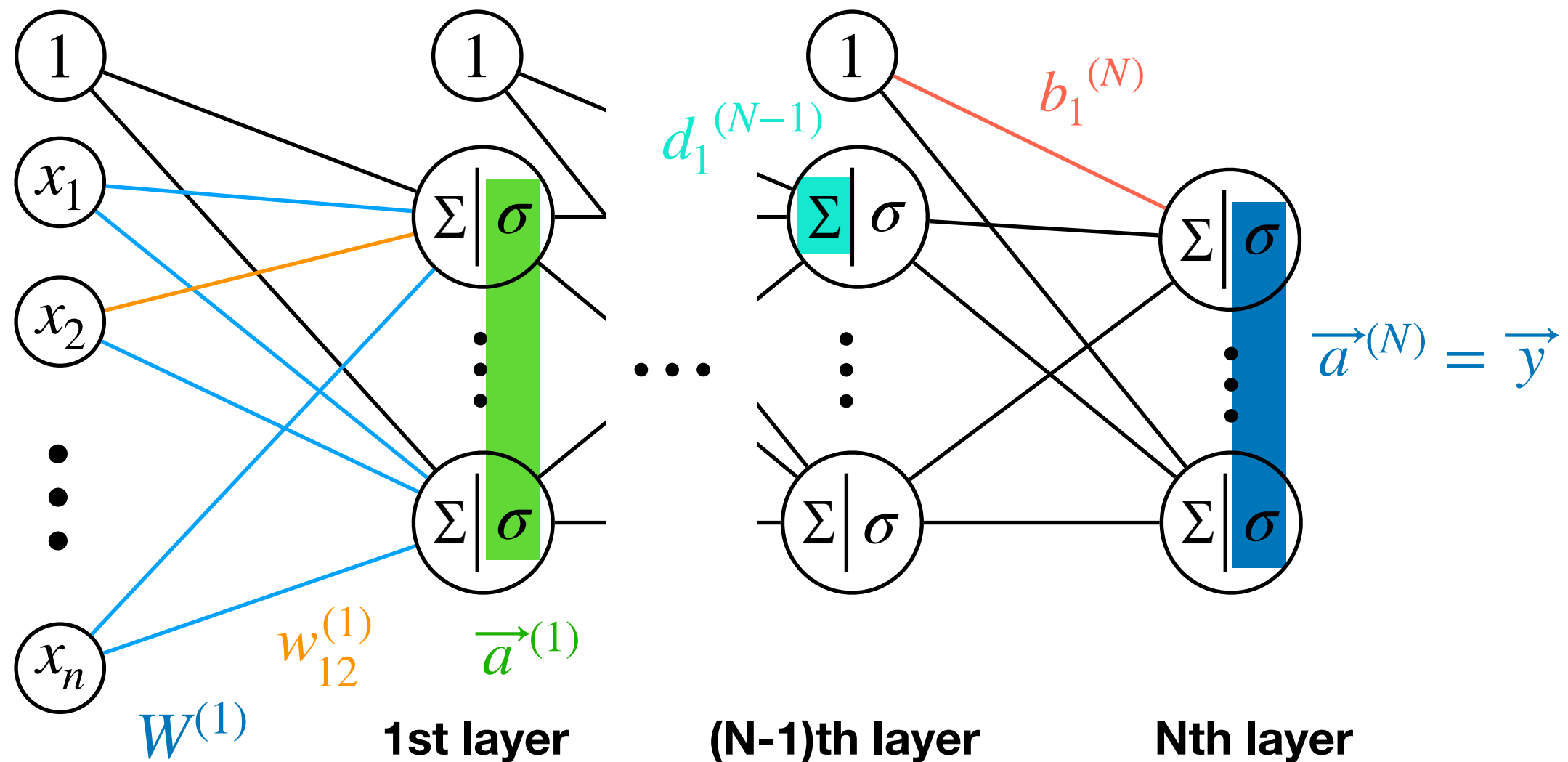
- For simplicity we consider the loss for only one input (instead of a whole dataset): (\vec{x}, \vec{t}) .
- Furthermore we consider the following (very general) multi-layer perceptron:



Backpropagation

- Backpropagation calculates the gradient of the loss.
- For that first the loss and thus the output has to be calculated.
- Backpropagation is always preceded by a forward step.

Notation Reminder



Differentiating the Loss Function

- The goal is to compute the gradient of the loss function:

Example loss function: Sum-squared error: $L = \sum_{k=1}^m \frac{1}{2}(t_k - y_k)^2$

- This means we have to calculate the partial derivative of the loss function in respect to every single variable!
- Let's start with the partial derivative in respect to a weight in the last layer:

$$\frac{\partial L}{\partial w_{ij}^{(N)}}$$

Differentiating the Loss Function

$$\frac{\partial L}{\partial w_{ij}^{(N)}}$$

- L is the loss for one sample

$$L = \sum_{k=1}^m \frac{1}{2} (t_k - y_k)^2$$

- L is not directly dependent on $w_{ij}^{(N)}$. L is a nested function!



Chain rule!

Chain Rule

- How to differentiate a nested function?

Simple Example $f(x) = (\cos(x))^2$

Classic chain rule

$$f(x) = g(h(x)) \quad \text{where}$$

$$g(\cdot) = (\cdot)^2 \quad \text{and} \quad h(x) = \cos(x)$$

Chain Rule: $f'(x) = g'(h(x))h'(x)$

$$\begin{aligned} \Rightarrow f'(x) &= 2\cos(x)(-\sin(x)) \\ &= -2\sin(x)\cos(x) \end{aligned}$$

Leibniz's Notation

$$z = y^2 \quad \text{where} \quad y = \cos(x)$$

Leibniz's Notation: $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$

$$\begin{aligned} \Rightarrow \frac{\partial z}{\partial x} &= 2y(-\sin(x)) \\ &= 2\cos(x)(-\sin(x)) \\ &= -2\sin(x)\cos(x) \end{aligned}$$

Backpropagation - Derivation

Let's write down the nested loss function!

$$\frac{\partial L}{\partial w_{ij}^{(N)}}$$

$$L = \sum_{k=1}^m \frac{1}{2} (t_k - y_k)^2$$

with

$$y_i = \sigma(d_i^{(N)})$$

$$d_i^{(N)} = \sum_j w_{ij}^{(N)} a_j^{(N-1)} + b_i^{(N)}$$

Write down the partial derivative in Leibniz's notation:

$$\frac{\partial L}{\partial w_{ij}^{(N)}} = \frac{\frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial d_i^{(N)}} \frac{\partial d_i^{(N)}}{\partial w_{ij}^{(N)}}}{\frac{\partial y_i}{\partial d_i^{(N)}} \frac{\partial d_i^{(N)}}{\partial w_{ij}^{(N)}}}$$

Now we can
simply compute it
step-by-step!

Backpropagation - Derivation

$$L = \sum_{k=1}^m \frac{1}{2} (t_k - y_k)^2$$

$$\frac{\partial L}{\partial w_{ij}^{(N)}} = \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial d_i^{(N)}} \frac{\partial d_i^{(N)}}{\partial w_{ij}^{(N)}}$$

$$\begin{aligned} \frac{\partial L}{\partial y_i} &= \frac{\partial}{\partial y_i} \sum_{k=1}^m \frac{1}{2} (t_k - y_k)^2 \\ &= \frac{\partial}{\partial y_i} \frac{1}{2} (t_i - y_i)^2 \\ &= \frac{1}{2} 2(t_i - y_i)(-1) \\ &= -(t_i - y_i) \end{aligned}$$

Backpropagation - Derivation

$$y_i = \sigma(d_i^{(N)})$$

$$\frac{\partial L}{\partial w_{ij}^{(N)}} = \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial d_i^{(N)}} \frac{\partial d_i^{(N)}}{\partial w_{ij}^{(N)}}$$

$$\begin{aligned} \frac{\partial y_i}{\partial d_i^{(N)}} &= \frac{\partial}{\partial d_i^{(N)}} \sigma(d_i^{(N)}) \\ &= \sigma'(d_i^{(N)}) \end{aligned}$$

**Depends on the chosen
activation function!**



Backpropagation - Derivation

$$d_i^{(N)} = \sum_k w_{ik} a_k^{(N-1)} + b_i^{(N)}$$

$$\frac{\partial L}{\partial w_{ij}^{(N)}} = \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial d_i^{(N)}} \frac{\partial d_i^{(N)}}{\partial w_{ij}^{(N)}}$$

$$\begin{aligned} \frac{\partial d_i^{(N)}}{\partial w_{ij}^{(N)}} &= \frac{\partial}{\partial w_{ij}^{(N)}} \sum_k w_{ik}^{(N)} a_k^{(N-1)} + b_i^{(N)} \\ &= \frac{\partial}{\partial w_{ij}^{(N)}} w_{ij}^{(N)} a_j^{(N-1)} \\ &= a_j^{(N-1)} \end{aligned}$$

Backpropagation - Derivation

Putting it all together:

$$\begin{aligned}\frac{\partial L}{\partial w_{ij}^{(N)}} &= \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial d_i^{(N)}} \frac{\partial d_i^{(N)}}{\partial w_{ij}^{(N)}} \\ &= - (t_i - y_i) \sigma'(d_i^{(N)}) a_j^{(N-1)}\end{aligned}$$

Backpropagation - Derivation

Let's proceed the partial derivative in respect to a weight one layer before the last layer:

$$\frac{\partial L}{\partial w_{ij}^{(N-1)}}$$

Again we can use Leibniz's Notation:

$$\frac{\partial L}{\partial w_{ij}^{(N-1)}} = \frac{\partial L}{\partial \vec{y}} \frac{\partial \vec{y}}{\partial \vec{d}^{(N)}} \frac{\partial \vec{d}^{(N)}}{\partial a_i^{(N-1)}} \frac{\partial a_i^{(N-1)}}{\partial d_i^{(N-1)}} \frac{\partial d_i^{(N-1)}}{\partial w_{ij}^{(N-1)}}$$

Take a minute and try to understand these dependencies!

Backpropagation - Derivation

$$\frac{\partial L}{\partial w_{ij}^{(N-1)}} = \frac{\partial L}{\partial \vec{y}} \frac{\partial \vec{y}}{\partial \vec{d}^{(N)}} \frac{\partial \vec{d}^{(N)}}{\partial a_i^{(N-1)}} \frac{\partial a_i^{(N-1)}}{\partial d_i^{(N-1)}} \frac{\partial d_i^{(N-1)}}{\partial w_{ij}^{(N-1)}}$$

Gradient

$$\begin{aligned} \frac{\partial L}{\partial \vec{y}} &= \frac{\partial}{\partial \vec{y}} \sum_{k=1}^m \frac{1}{2} (t_k - y_k)^2 \\ &= \left(\frac{\partial}{\partial y_1} \sum_{k=1}^m \frac{1}{2} (t_k - y_k)^2 \cdots \frac{\partial}{\partial y_m} \sum_{k=1}^m \frac{1}{2} (t_k - y_k)^2 \right) \\ &= \left(-(t_1 - y_1) \cdots -(t_m - y_m) \right) \end{aligned}$$

Slide 43

Backpropagation - Derivation

$$\frac{\partial L}{\partial w_{ij}^{(N-1)}} = \frac{\partial L}{\partial \vec{y}} \frac{\partial \vec{y}}{\partial \vec{d}^{(N)}} \frac{\partial \vec{d}^{(N)}}{\partial a_i^{(N-1)}} \frac{\partial a_i^{(N-1)}}{\partial d_i^{(N-1)}} \frac{\partial d_i^{(N-1)}}{\partial w_{ij}^{(N-1)}}$$

Jacobian*

$$\frac{\partial \vec{y}}{\partial \vec{d}^{(N)}} = \begin{pmatrix} \frac{\partial y_1}{\partial d_1^{(N)}} & \dots & \frac{\partial y_1}{\partial d_m^{(N)}} \\ \vdots & & \vdots \\ \frac{\partial y_m}{\partial d_1^{(N)}} & \dots & \frac{\partial y_m}{\partial d_m^{(N)}} \end{pmatrix}$$

$$= \begin{pmatrix} \sigma'(d_1^{(N)}) & & 0 \\ & \sigma'(d_2^{(N)}) & \\ & & \ddots \\ 0 & & & \sigma'(d_m^{(N)}) \end{pmatrix}$$

Slide 44

***Jacobian matrix - the partial derivatives of a function that maps many variables to many variables**

Backpropagation - Derivation

$$\frac{\partial L}{\partial w_{ij}^{(N-1)}} = \frac{\partial L}{\partial \vec{y}} \frac{\partial \vec{y}}{\partial \vec{d}^{(N)}} \frac{\partial \vec{d}^{(N)}}{\partial a_i^{(N-1)}} \frac{\partial a_i^{(N-1)}}{\partial d_i^{(N-1)}} \frac{\partial d_i^{(N-1)}}{\partial w_{ij}^{(N-1)}}$$

Jacobian with
one column


$$\frac{\partial \vec{d}^{(N)}}{\partial a_i^{(N-1)}} = \begin{pmatrix} \frac{\partial d_1^{(N)}}{\partial a_i^{(N-1)}} \\ \vdots \\ \frac{\partial d_m^{(N)}}{\partial a_i^{(N-1)}} \end{pmatrix} = \begin{pmatrix} \frac{\partial}{\partial a_i^{(N-1)}} w_{1i}^{(N)} a_i^{(N-1)} \\ \vdots \\ \frac{\partial}{\partial a_i^{(N-1)}} w_{mi}^{(N)} a_i^{(N-1)} \end{pmatrix} = \begin{pmatrix} w_{1i}^{(N)} \\ \vdots \\ w_{mi}^{(N)} \end{pmatrix}$$

Slide 45

Backpropagation - Derivation

$$\frac{\partial L}{\partial w_{ij}^{(N-1)}} = \frac{\partial L}{\partial \vec{y}} \frac{\partial \vec{y}}{\partial \vec{d}^{(N)}} \frac{\partial \vec{d}^{(N)}}{\partial a_i^{(N-1)}} \frac{\partial a_i^{(N-1)}}{\partial d_i^{(N-1)}} \frac{\partial d_i^{(N-1)}}{\partial w_{ij}^{(N-1)}}$$

$$\frac{\partial a_i^{(N-1)}}{\partial d_i^{(N-1)}} = \frac{\partial}{\partial d_i^{(N-1)}} \sigma(d_i^{(N-1)})$$

Slide 44 

$$= \sigma'(d_i^{(N-1)})$$

Backpropagation - Derivation

$$\frac{\partial L}{\partial w_{ij}^{(N-1)}} = \frac{\partial L}{\partial \vec{y}} \frac{\partial \vec{y}}{\partial \vec{d}^{(N)}} \frac{\partial \vec{d}^{(N)}}{\partial a_i^{(N-1)}} \frac{\partial a_i^{(N-1)}}{\partial d_i^{(N-1)}} \frac{\partial d_i^{(N-1)}}{\partial w_{ij}^{(N-1)}}$$

$$\begin{aligned} \frac{\partial d_i^{(N-1)}}{\partial w_{ij}^{(N-1)}} &= \frac{\partial}{\partial w_{ij}^{(N-1)}} \sum_k w_{ik}^{(N-1)} a_k^{(N-2)} + b_i^{(N-1)} \\ &= \frac{\partial}{\partial w_{ij}^{(N-1)}} w_{ij}^{(N-1)} a_j^{(N-2)} \\ &= a_j^{(N-2)} \end{aligned}$$

Slide 45

Backpropagation - Derivation

Putting it all together:

$$\begin{aligned}
 \frac{\partial L}{\partial w_{ij}^{(N-1)}} &= \frac{\partial L}{\partial \vec{y}} \frac{\partial \vec{y}}{\partial \vec{d}^{(N)}} \frac{\partial \vec{d}^{(N)}}{\partial a_i^{(N-1)}} \frac{\partial a_i^{(N-1)}}{\partial d_i^{(N-1)}} \frac{\partial d_i^{(N-1)}}{\partial w_{ij}^{(N-1)}} \\
 &= \begin{pmatrix} -(t_1 - y_1) & \cdots & -(t_m - y_m) \end{pmatrix} \begin{pmatrix} \sigma'(d_1^{(N)}) & & \\ & \sigma'(d_2^{(N)}) & \\ & & \ddots \\ & & & \sigma'(d_m^{(N)}) \end{pmatrix} \frac{\partial \vec{d}^{(N)}}{\partial a_i^{(N-1)}} \frac{\partial a_i^{(N-1)}}{\partial d_i^{(N-1)}} \frac{\partial d_i^{(N-1)}}{\partial w_{ij}^{(N-1)}} \\
 &= \begin{pmatrix} -(t_1 - y_1) & \sigma'(d_1^{(N)}) & \cdots & -(t_m - y_m) & \sigma'(d_m^{(N)}) \end{pmatrix} \begin{pmatrix} w_{1i}^{(N)} \\ \vdots \\ w_{mi}^{(N)} \end{pmatrix} \frac{\partial a_i^{(N-1)}}{\partial d_i^{(N-1)}} \frac{\partial d_i^{(N-1)}}{\partial w_{ij}^{(N-1)}} \\
 &= \left(\sum_{k=1}^m -(t_k - y_k) \sigma'(d_k^{(N)}) w_{ki}^{(N)} \right) \frac{\partial a_i^{(N-1)}}{\partial d_i^{(N-1)}} \frac{\partial d_i^{(N-1)}}{\partial w_{ij}^{(N-1)}}
 \end{aligned}$$

Backpropagation - Derivation

- Now we can make the following beautiful simplification

$$\frac{\partial L}{\partial w_{ij}^{(N)}} = - \underbrace{(t_i - y_i) \sigma'(d_i^{(N)})}_{:= \delta_i^{(N)}} a_j^{(N-1)}$$

$$\Rightarrow \frac{\partial L}{\partial w_{ij}^{(N)}} = \delta_i^{(N)} a_j^{(N-1)}$$

Backpropagation - Derivation

- Now we can make the following beautiful simplification

$$\begin{aligned}\frac{\partial L}{\partial w_{ij}^{(N-1)}} &= \left(\sum_{k=1}^m - (t_k - y_k) \sigma'(d_k^{(N)}) w_{ki}^{(N)} \right) \sigma'(d_i^{(N-1)}) a_j^{(N-2)} \\ &= \underbrace{\left(\sum_{k=1}^m \delta_k^{(N)} w_{ki}^{(N)} \right)}_{:= \delta_i^{(N-1)}} \sigma'(d_i^{(N-1)}) a_j^{(N-2)}\end{aligned}$$

$$\Rightarrow \frac{\partial L}{\partial w_{ij}^{(N-1)}} = \delta_i^{(N-1)} a_j^{(N-2)}$$

Backpropagation - Formula

Important slide!

Which results in the following simple formula

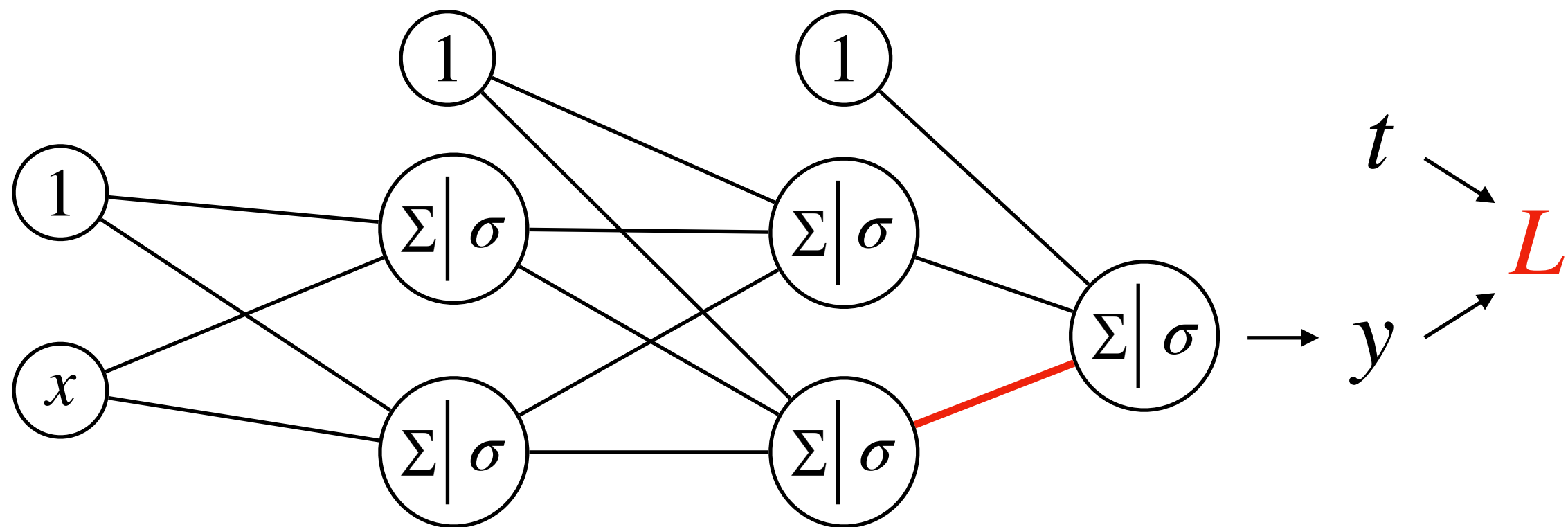
$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} a_j^{(l-1)}$$

where

$$\delta_i^{(l)} = \begin{cases} -(t_i - y_i) \sigma'(d_i^{(l)}) & \text{if } l = N, \\ \left(\sum_{k=1}^m \delta_k^{(l+1)} w_{ki}^{(l+1)} \right) \sigma'(d_i^{(l)}) & \text{else.} \end{cases}$$

Backpropagation - Intuition

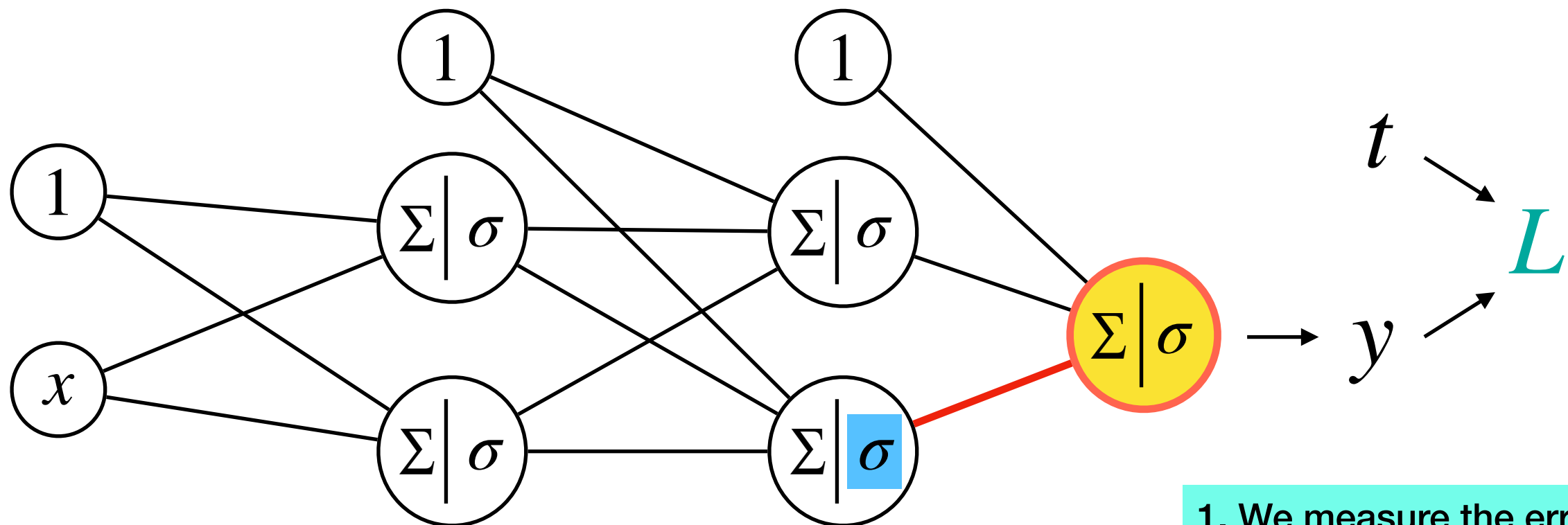
$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} a_j^{(l-1)} \quad \delta_i^{(l)} = \begin{cases} -(t_i - y_i) \sigma'(d_i^{(N)}) & \text{if } l = N, \\ \left(\sum_{k=1}^m \delta_k^{(l+1)} w_{ki}^{(l+1)} \right) \sigma'(d_i^{(l)}) & \text{else.} \end{cases}$$



To update this weight we are interested in how much this weight contributed to the error!

Backpropagation - Intuition

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} a_j^{(l-1)} \quad \delta_i^{(l)} = \begin{cases} -(t_i - y_i) \sigma'(d_i^{(N)}) & \text{if } l = N, \\ \left(\sum_{k=1}^m \delta_k^{(l+1)} w_{ki}^{(l+1)} \right) \sigma'(d_i^{(l)}) & \text{else.} \end{cases}$$



1. We measure the error that was done at the output!

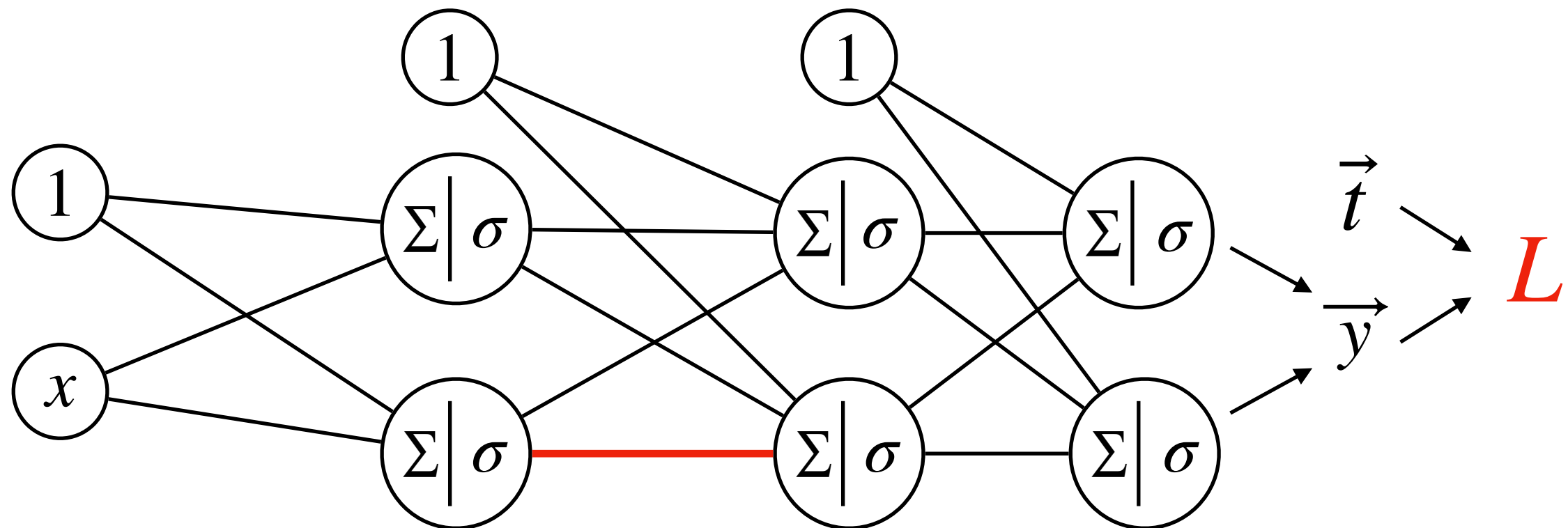
2. We backtrack the error to the drive by multiplying with the derivative of the activation function!

3. Thus we get the estimated error at this particular neuron (delta).

4. Finally we scale this error term with the activation that actually "drove through" this weight.

Backpropagation - Intuition

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} a_j^{(l-1)} \quad \delta_i^{(l)} = \begin{cases} -(t_i - y_i) \sigma'(d_i^{(N)}) & \text{if } l = N, \\ \left(\sum_{k=1}^m \delta_k^{(l+1)} w_{ki}^{(l+1)} \right) \sigma'(d_i^{(l)}) & \text{else.} \end{cases}$$

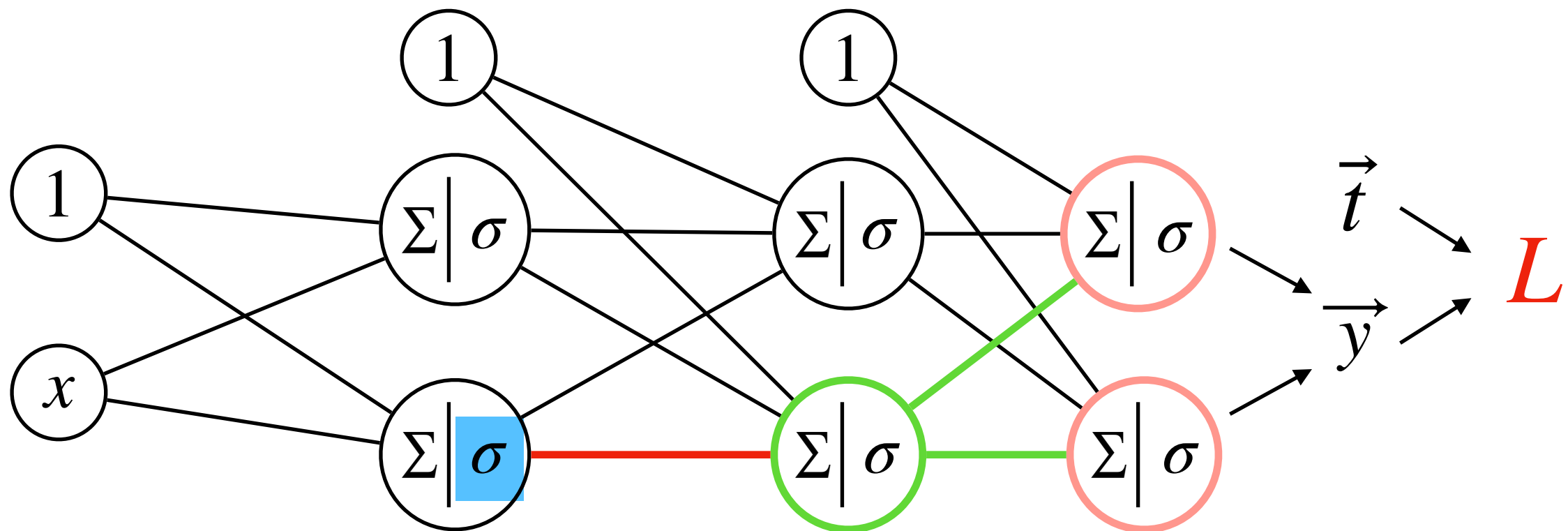


**Calculating the update
for weight in a hidden
layer is harder!**

**We only know the error
at the output node!**

Backpropagation - Intuition

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} a_j^{(l-1)} \quad \delta_i^{(l)} = \begin{cases} -(t_i - y_i) \sigma'(d_i^{(N)}) & \text{if } l = N, \\ \left(\sum_{k=1}^m \delta_k^{(l+1)} w_{ki}^{(+1)} \right) \sigma'(d_i^{(l)}) & \text{else.} \end{cases}$$



3. Lastly we again scale with the activation of the previous layer.

2. We recursively calculate the delta in the layer before.

1. We already know the error estimations at the layers: delta.

Any questions?

Conclusion & Outlook

Conclusion

- We talked about how a multi-layer perceptron can be optimized to solve a task.
- This is achieved through iterative optimization via gradient descent.
- Gradient descent requires to calculate the partial derivatives for all weights, which is achieved via the backpropagation algorithm.
- You will probably never again calculate the partial derivatives inside a neural network again though!

Outlook

- In the homework you will have to program the backpropagation algorithm in NumPy.
- This will be the motivation for the next lecture.
- Next week we will introduce TensorFlow.
- TensorFlow automates the gradient computation for you!

See you next week!

Resources

- [w1] Rama at English Wikipedia (https://commons.wikimedia.org/wiki/File:Marvin_Minsky_at_OLPCb.jpg)
- [w2] Rodrigo Mesquita at English Wikipedia (https://commons.wikimedia.org/wiki/File:Seymour_Papert.png)
- [mp] M. Minsky, S. Papert, “Perceptrons: An Introduction to Computational Geometry” *MIT Press*, 1969.