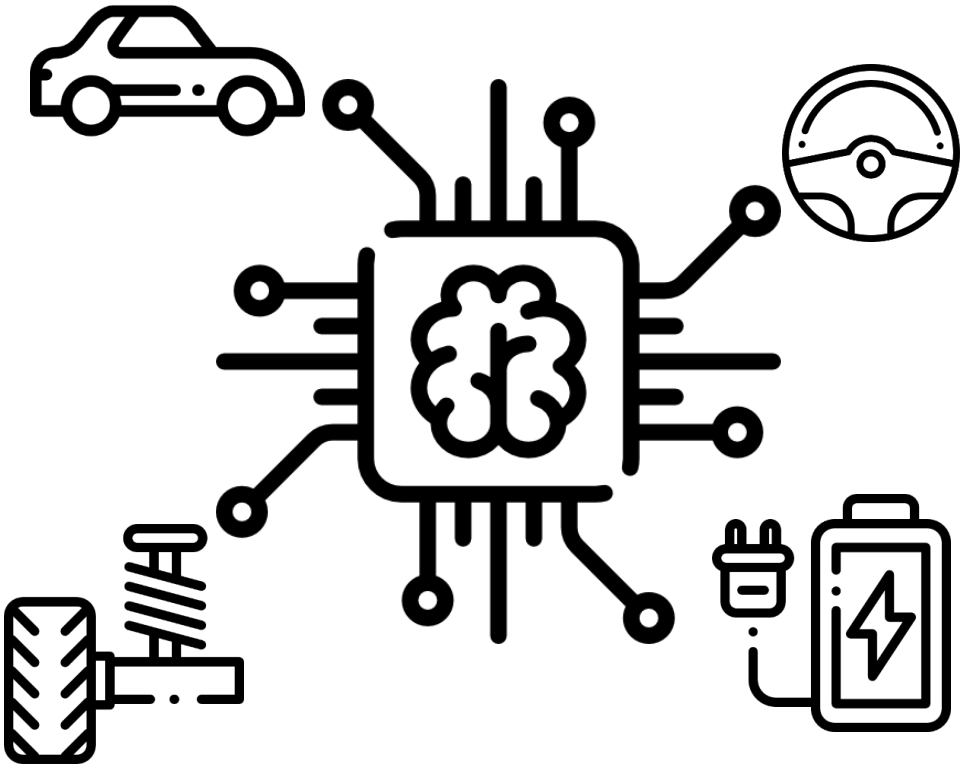
Institute of Automotive Technology
TUM School of Engineering and Design
Technical University of Munich

Artificial Intelligence in Automotive Technology

Maximilian Geißlinger / Fabian Netzler

Prof. Dr.-Ing. Markus Lienkamp

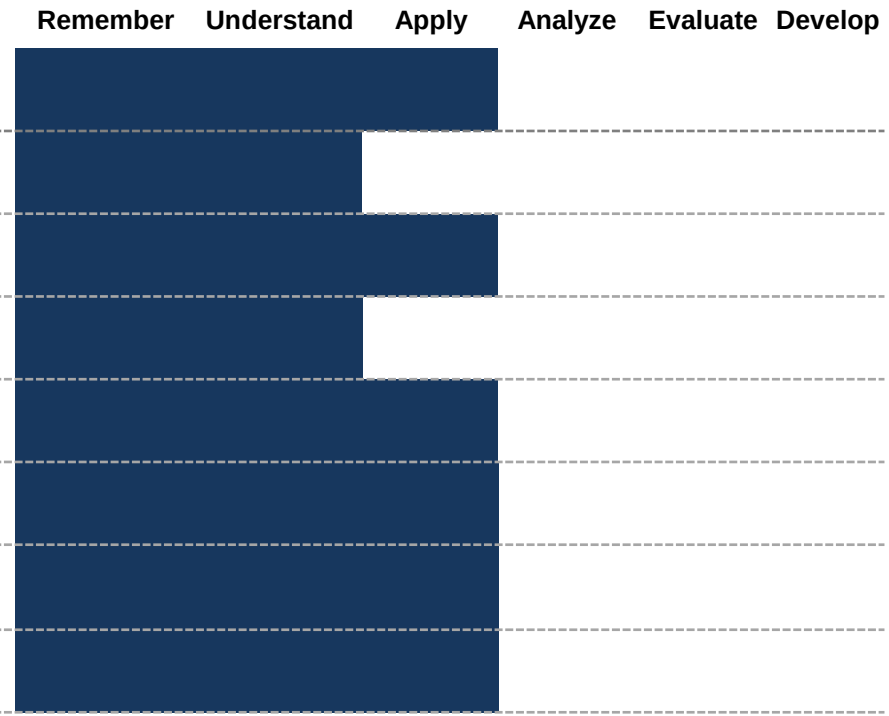


Lecture Overview

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

Objectives for Lecture 7: Deep Neural Networks

After the lecture you will be able to…



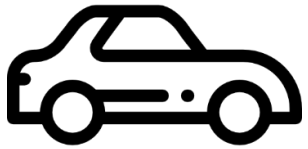
Deep Neural Networks
Prof. Dr. Markus Lienkamp

(Domagoj Majstorović, M. Sc.)

Agenda

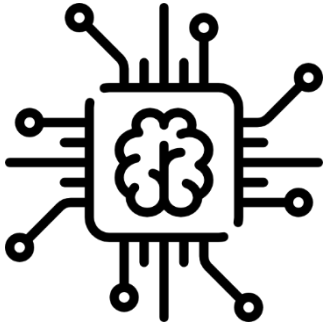
1. Chapter: Backpropagation

- 1.1 Computational Graph
- 1.2 Single Neuron
- 1.3 Neural Chain
- 1.4 Neural Network



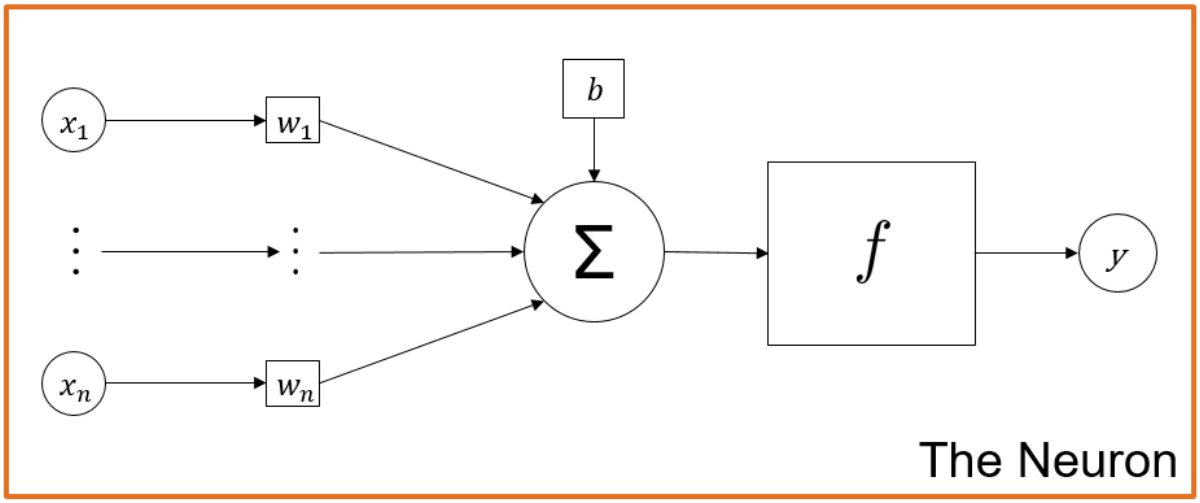
2. Chapter: Neuronal Networks

- 2.1 Activation Functions
- 2.2 Fully Connected Layer
- 2.3 Batches and Weight Initialization
- 2.4 Training the Network



3. Chapter: Summary

Last Week - Repetition



$L = w^2$

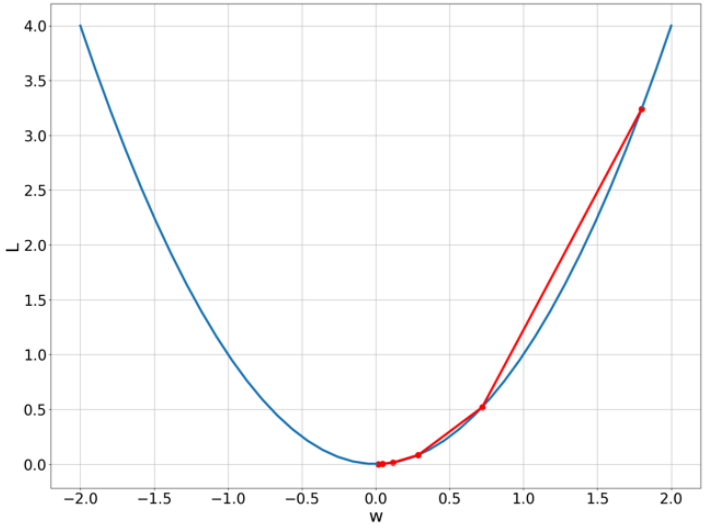
$\nabla L = \frac{dL}{dw} = 2 \cdot w$

$\alpha = 0.3$

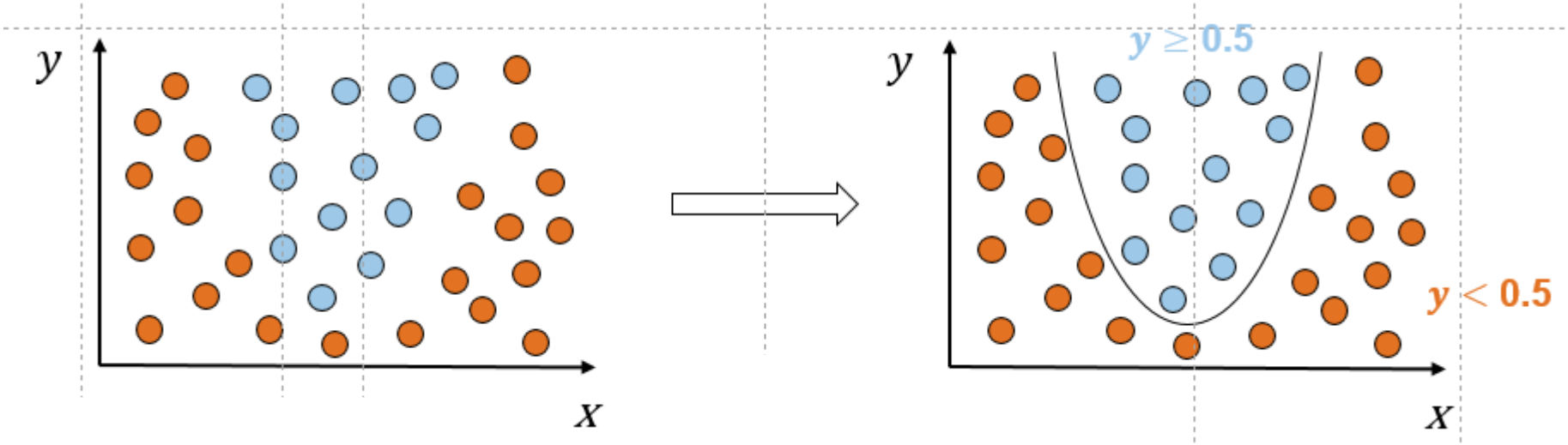
$w_0 = 1.8$

$\epsilon = 0.001$

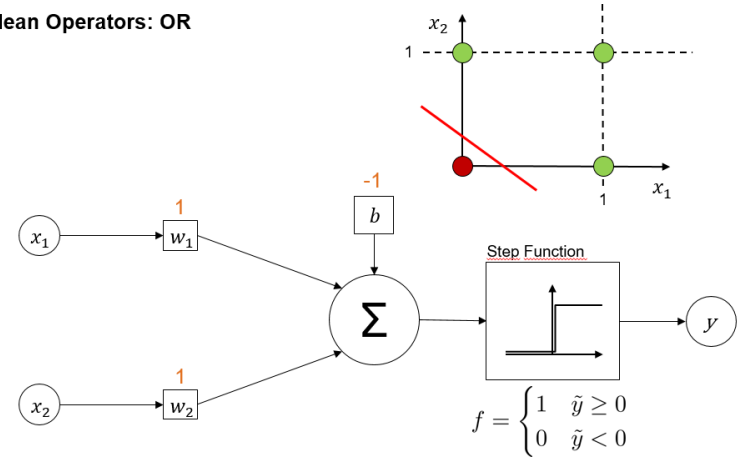
$N = 5$



Repetition

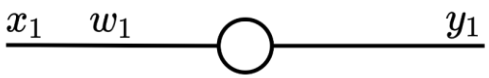


Boolean Operators: OR

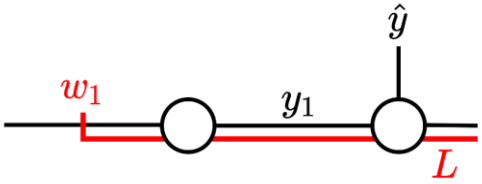


Last Week:

Single Neuron



1D Gradient Descent



$L = w^2$

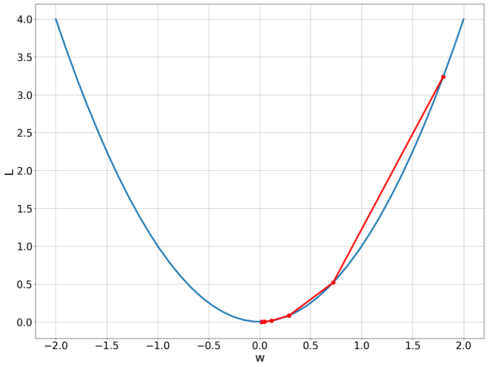
$\nabla L = \frac{dL}{dw} = 2 \cdot w$

$\alpha = 0.3$

$w_0 = 1.8$

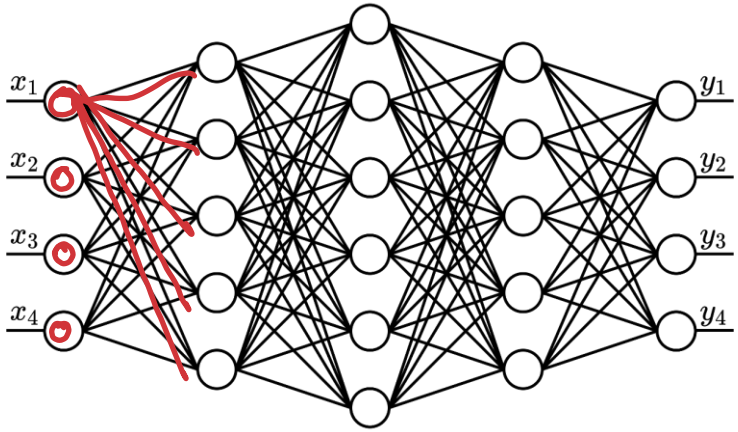
$\epsilon = 0.001$

$N = 5$

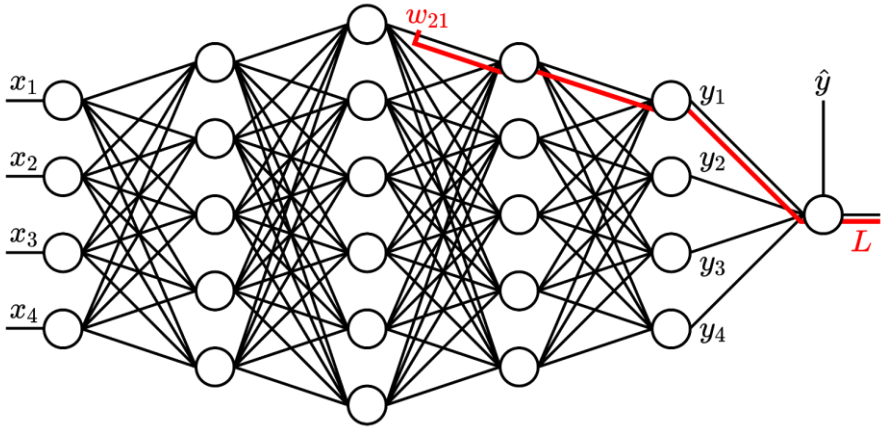


Today:

Neural Network



Backpropagation

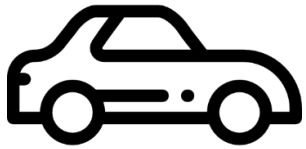


Deep Neural Networks
Prof. Dr. Markus Lienkamp

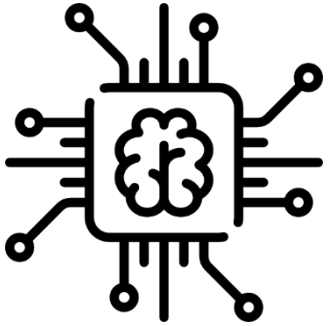
(Domagoj Majstorović, M. Sc.)

Agenda

1. Chapter: Backpropagation
- 1.1 Computational Graph
- 1.2 Single Neuron
- 1.3 Neural Chain
- 1.4 Neural Network



2. Chapter: Neuronal Networks
- 2.1 Activation Functions
- 2.2 Fully Connected Layer
- 2.3 Batches and Weight Initialization
- 2.4 Training the Network



3. Chapter: Summary

Computational Graph

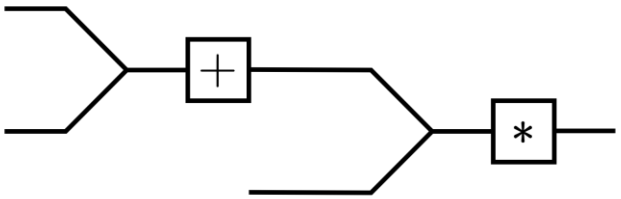
We need: $\Delta w = -\alpha \frac{\partial L}{\partial w}$

Options:

1. Analytical solution

$$\frac{\partial f}{\partial x_1} = w_1 \frac{e^{-w_1 x_1 - w_2 x_2 - w_3}}{(e^{-w_1 x_1 - w_2 x_2 - w_3} + 1)^2}$$

2. Computational graph



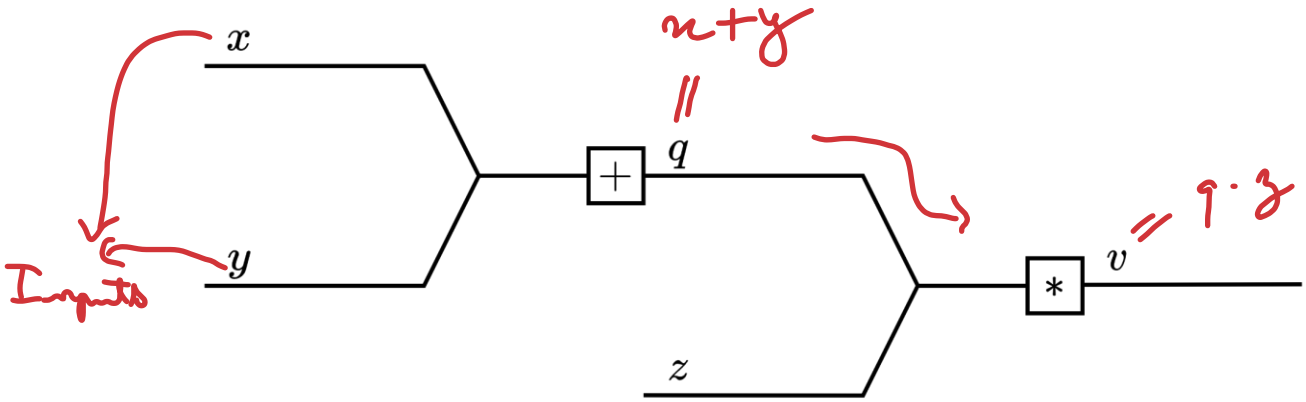
→ Chain of Derivation → **Backpropagation**

Additional Slides

To understand the backpropagation algorithm, you need to understand the role of local gradients and how they work. First, we look at a simple example which demonstrates the computational graph. Each operation box (node) represents a mathematic operation with the given input and calculated output data.

Computational Graph

Simple example

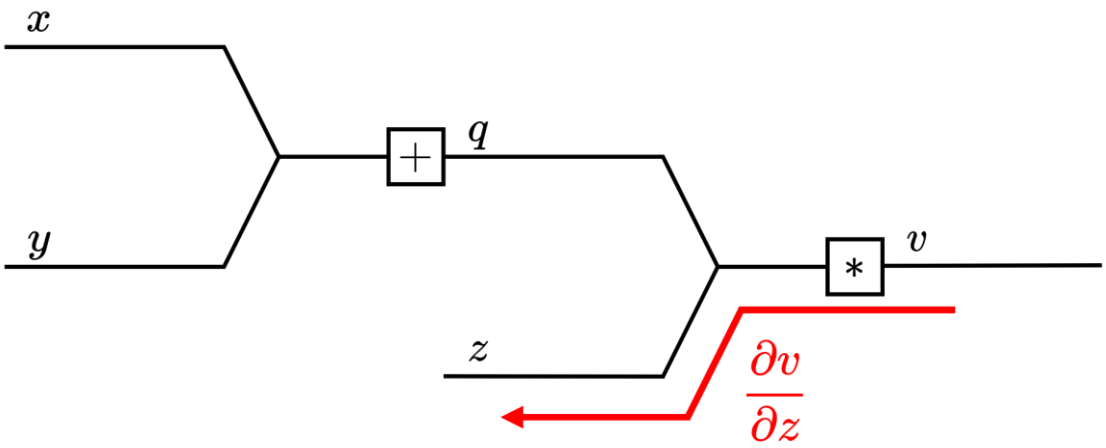


Calculate:

$$\frac{\partial v}{\partial z}, \frac{\partial v}{\partial q}, \frac{\partial v}{\partial x}, \frac{\partial v}{\partial y}$$

Computational Graph

Simple example



Calculate:

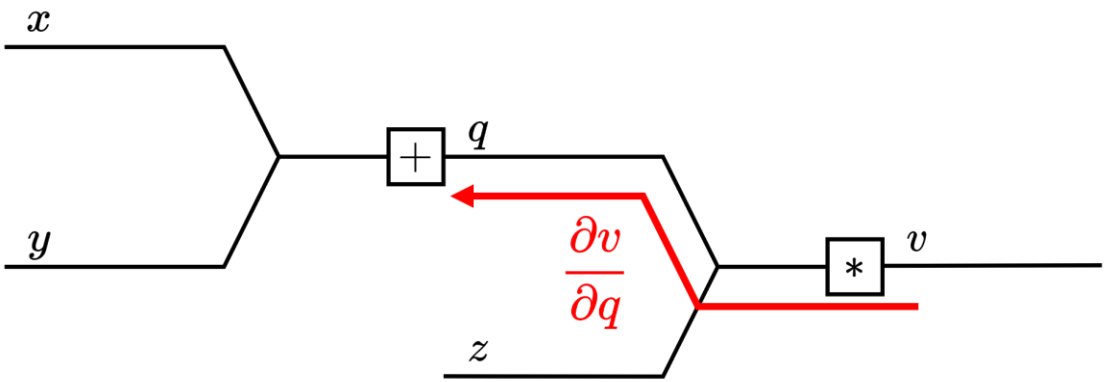
$$\frac{\partial v}{\partial z}, \frac{\partial v}{\partial q}, \frac{\partial v}{\partial x}, \frac{\partial v}{\partial y}$$

Additional Slides

The gradient between v and z can be calculated as shown on the slide. The gradients “flow” backwards (upstream) along the graph.

Computational Graph

Simple example



Calculate:

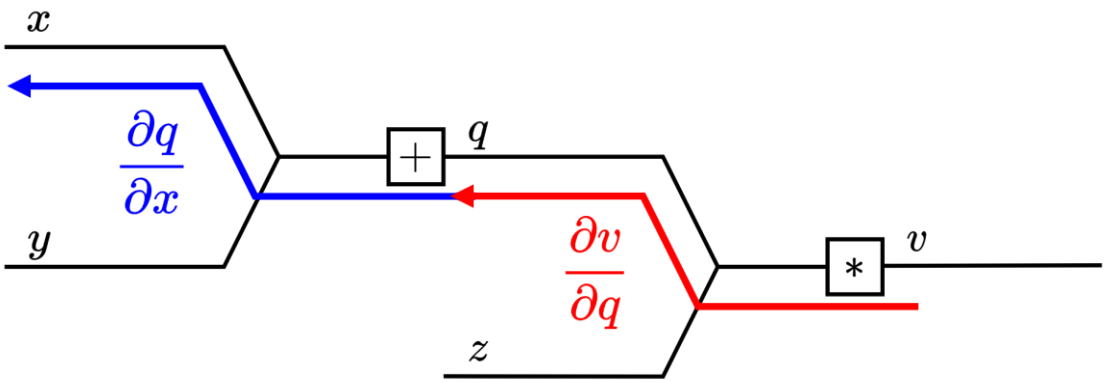
$$\frac{\partial v}{\partial z}, \boxed{\frac{\partial v}{\partial q}}, \frac{\partial v}{\partial x}, \frac{\partial v}{\partial y}$$

Additional Slides

The gradient between x and q can calculated in a similar manner.

Computational Graph

Simple example

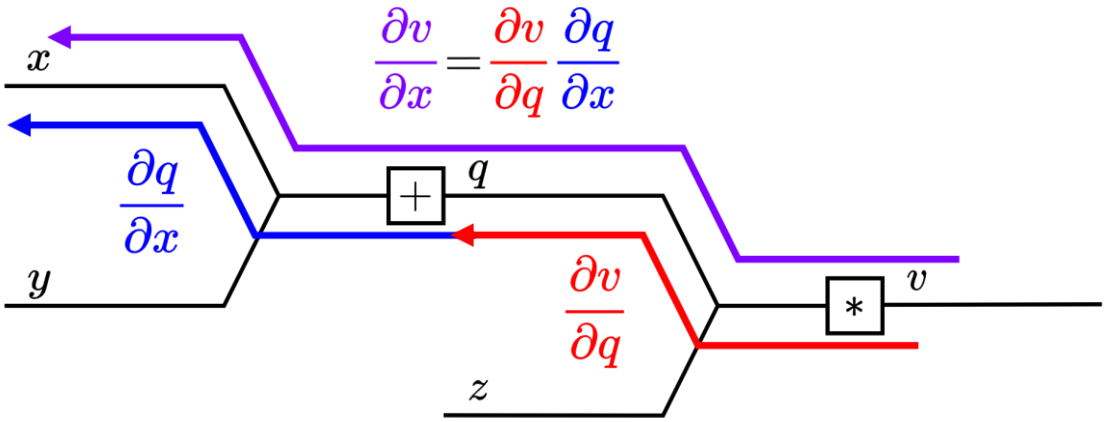


Calculate:

$$\frac{\partial v}{\partial z}, \boxed{\frac{\partial v}{\partial q}}, \boxed{\frac{\partial v}{\partial x}}, \frac{\partial v}{\partial y}$$

Computational Graph

Simple example

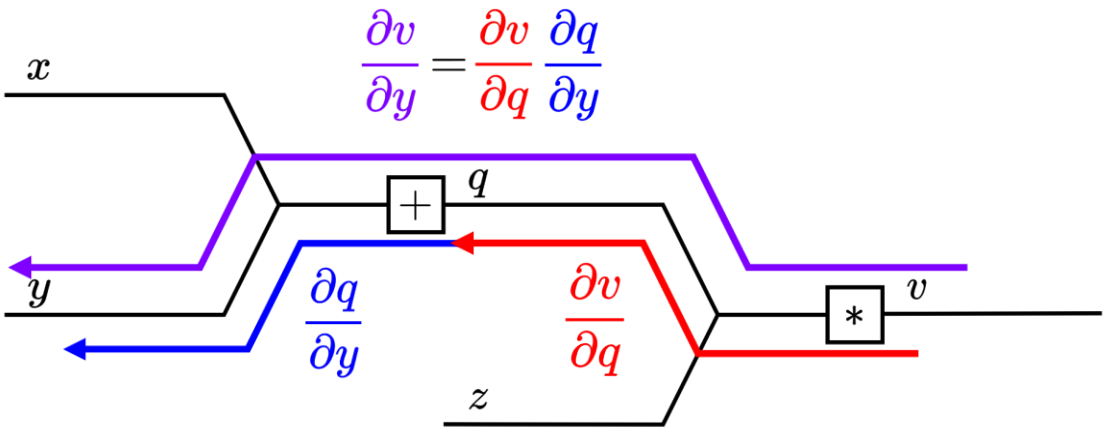


Calculate:

$$\frac{\partial v}{\partial z}, \frac{\partial v}{\partial q}, \frac{\partial v}{\partial x}, \frac{\partial v}{\partial y}$$

Computational Graph

Simple example



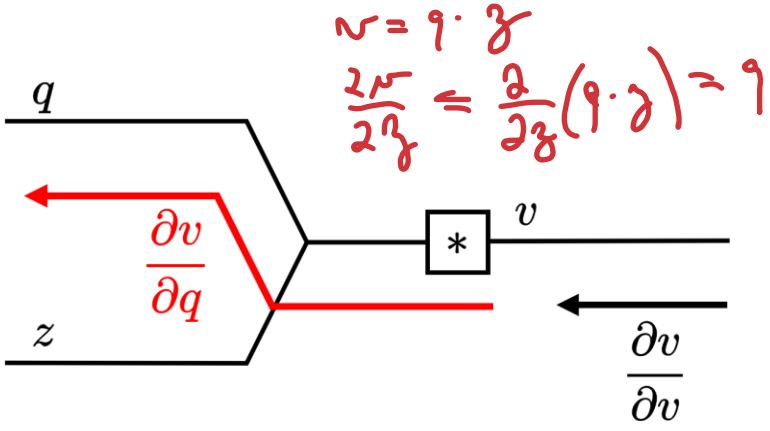
Calculate:

$$\frac{\partial v}{\partial z}, \frac{\partial v}{\partial q}, \frac{\partial v}{\partial x}, \frac{\partial v}{\partial y}$$

Computational Graph

Standard operations and some rules

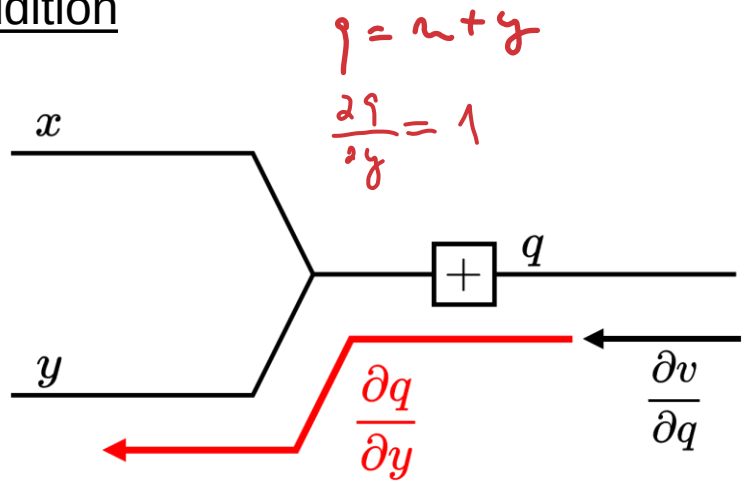
Multiplication



$$\frac{\partial v}{\partial q} = \frac{\partial v}{\partial v} z$$

"Gradient **switches** to other factor"

Addition



$$\frac{\partial v}{\partial y} = \frac{\partial v}{\partial q}\frac{\partial q}{\partial y} = \frac{\partial v}{\partial q} \cdot 1$$

"Gradient **remains the same**"

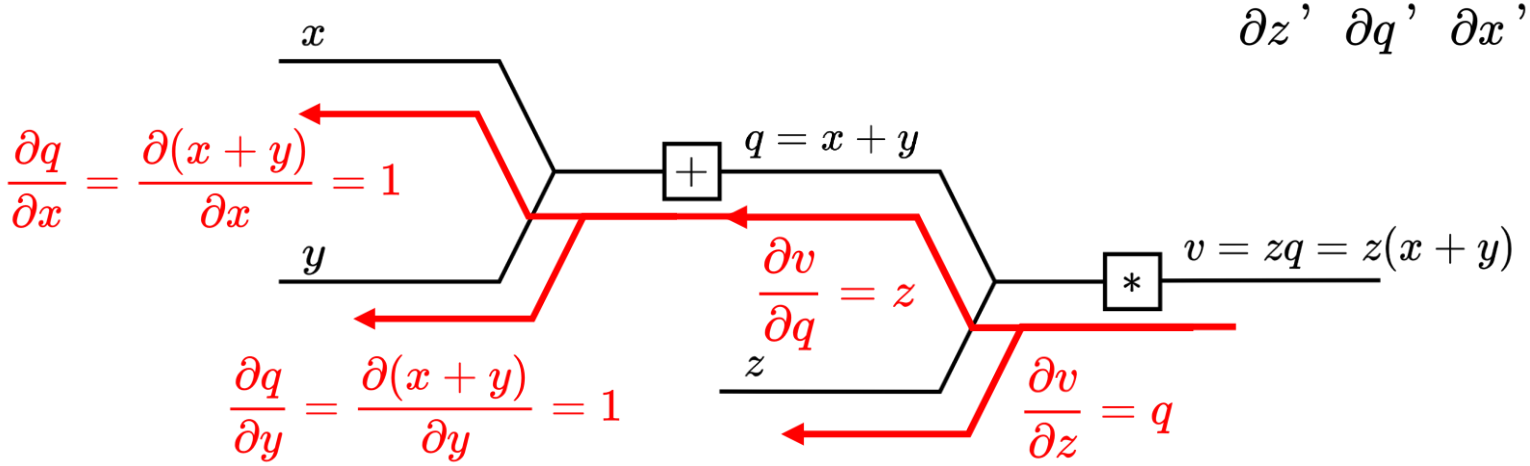
Simple rules for calculation of the upstream gradient can be applied to these operations of addition and multiplication which make the overall calculation much easier:

- **Addition**: The incoming (upstream) gradient remains the same as it is multiplied with number 1.
- **Multiplication**: The incoming (upstream) gradient needs to be multiplied with the factor on another branch.

Computational Graph

Simple example

$$\frac{\partial v}{\partial z}, \frac{\partial v}{\partial q}, \frac{\partial v}{\partial x}, \frac{\partial v}{\partial y}$$

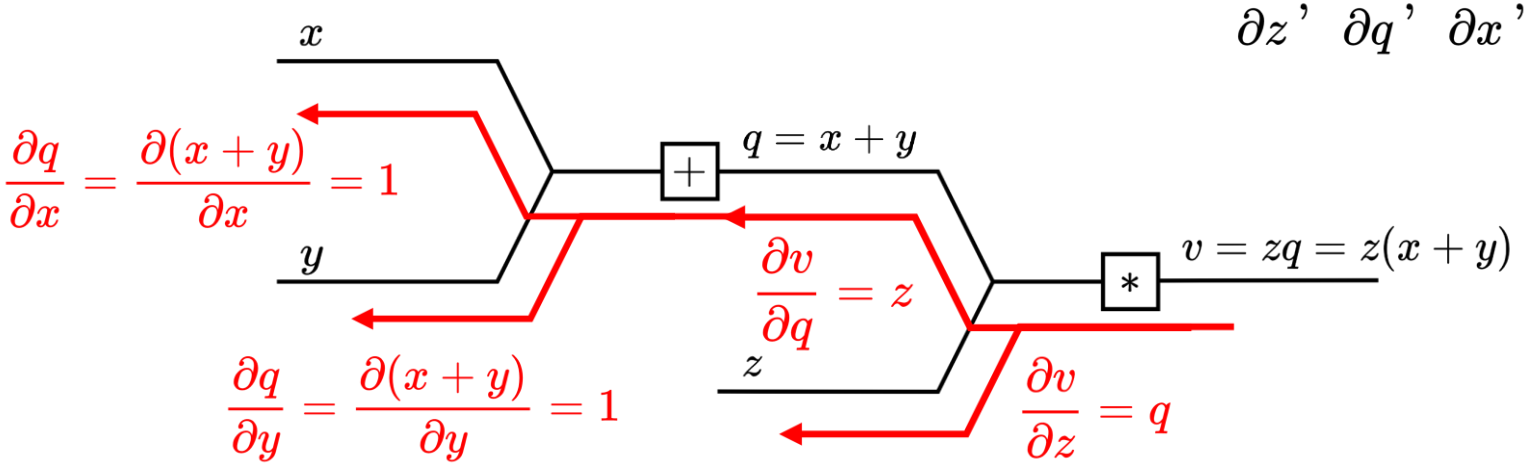


$$\frac{\partial v}{\partial z} = q \qquad \frac{\partial v}{\partial q} = z \qquad \frac{\partial v}{\partial x} = \frac{\partial v}{\partial q}\frac{\partial q}{\partial x} = z \cdot 1 = z \qquad \frac{\partial v}{\partial y} = \frac{\partial v}{\partial q}\frac{\partial q}{\partial y} = z \cdot 1 = z$$

Computational Graph

Simple example

$$\frac{\partial v}{\partial z}, \frac{\partial v}{\partial q}, \frac{\partial v}{\partial x}, \frac{\partial v}{\partial y}$$



$$\frac{\partial v}{\partial z} = \frac{\partial v}{\partial v} q \qquad \frac{\partial v}{\partial q} = \frac{\partial v}{\partial v} z$$

$$\frac{\partial v}{\partial x} = \frac{\partial v}{\partial q}\frac{\partial q}{\partial x} = z \cdot 1 = z \qquad \frac{\partial v}{\partial y} = \frac{\partial v}{\partial q}\frac{\partial q}{\partial y} = z \cdot 1 = z$$

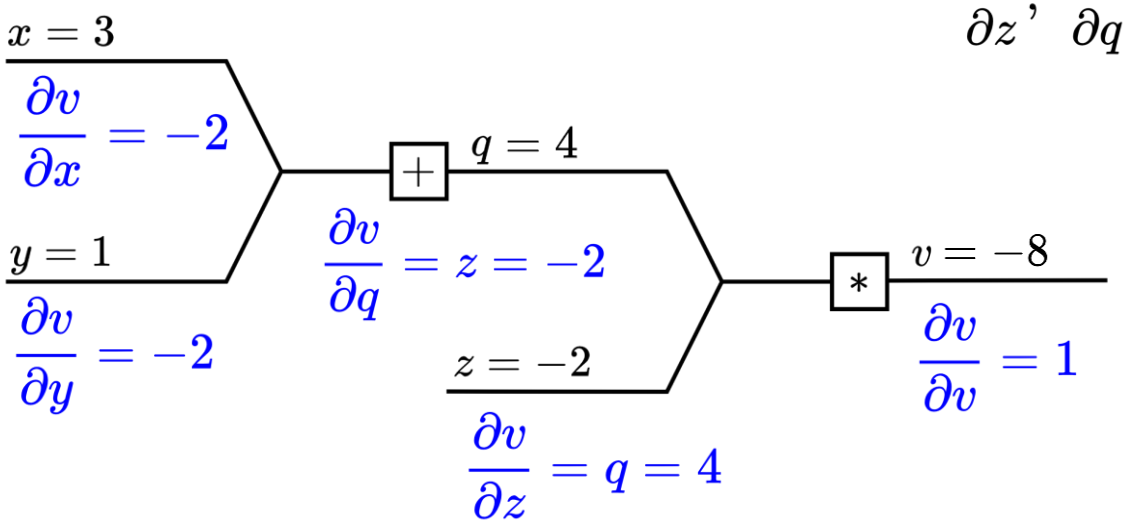
Additional Slides

We can use the very same method to calculate the gradient between v and y.

Computational Graph

Simple example, now with numbers

$$\frac{\partial v}{\partial z}, \frac{\partial v}{\partial q}, \frac{\partial v}{\partial x}, \frac{\partial v}{\partial y}$$

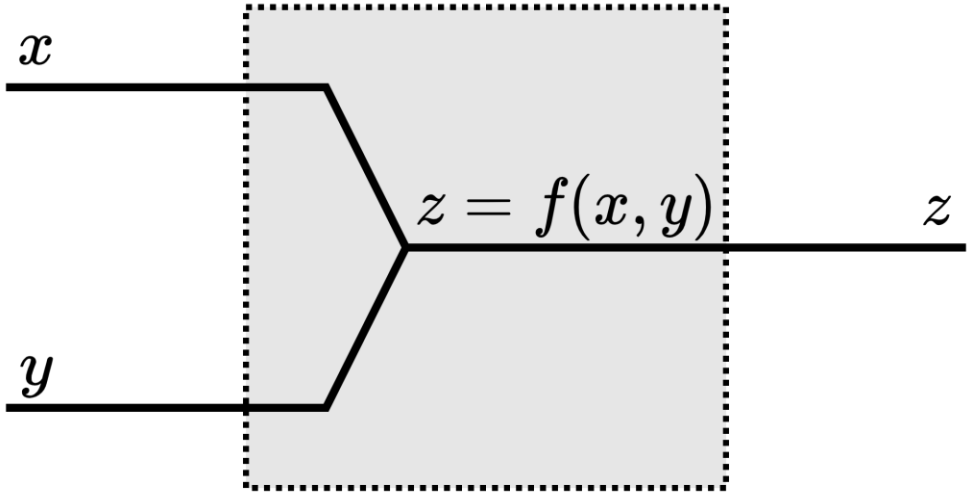


$$\frac{\partial v}{\partial z} = \frac{\partial v}{\partial v} q = 1 \cdot 4 = 4 \qquad \frac{\partial v}{\partial q} = \frac{\partial v}{\partial v} z = 1 \cdot (-2) = -2$$

$$\frac{\partial v}{\partial x} = \frac{\partial v}{\partial q}\frac{\partial q}{\partial x} = (-2) \cdot 1 = -2 \qquad \frac{\partial v}{\partial y} = \frac{\partial v}{\partial q}\frac{\partial q}{\partial y} = (-2) \cdot 1 = -2$$

Computational Graph

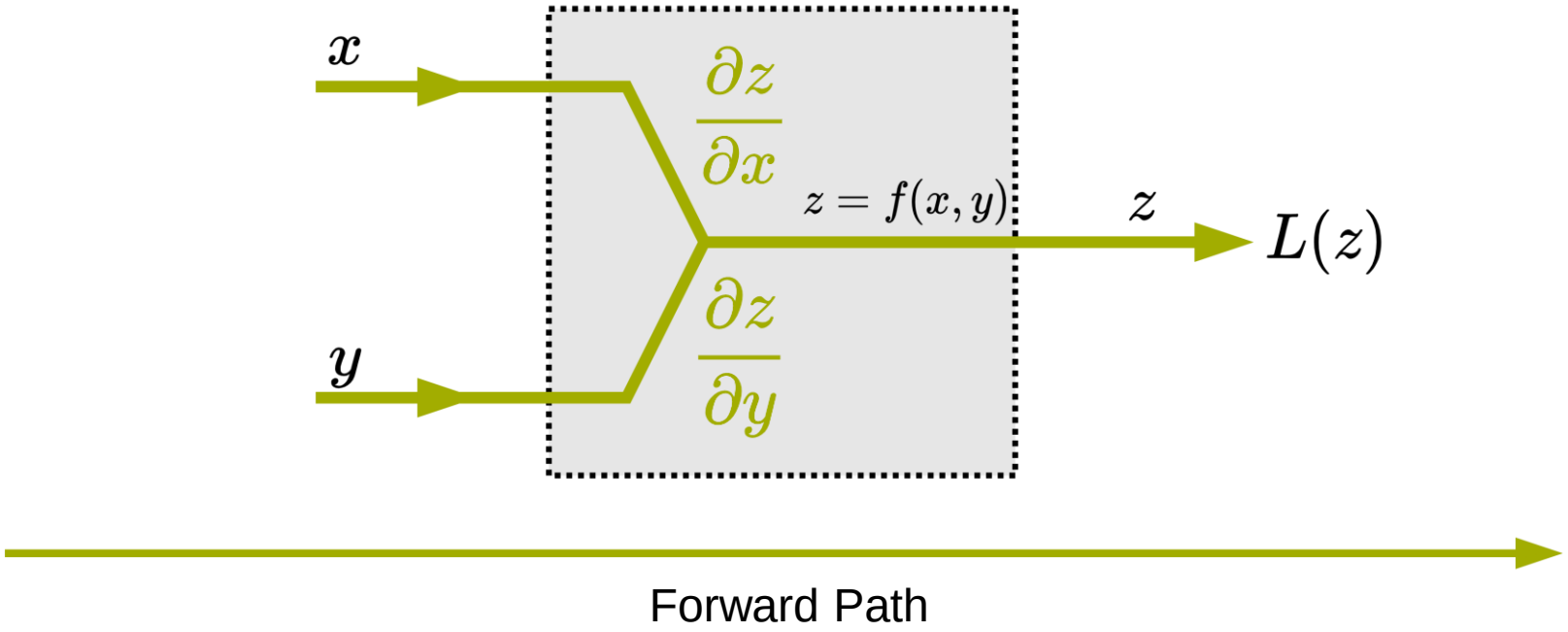
Abstract function



Computational Graph

Abstract function

Saving local gradients during the forward path phase



Additional Slides

So far, we have looked at the most basic case with single mathematical operations. A more abstract form of those operations can be represented through functions. Nevertheless, the procedure for the calculation of local gradients will remain the same, no matter how complex the function might get.

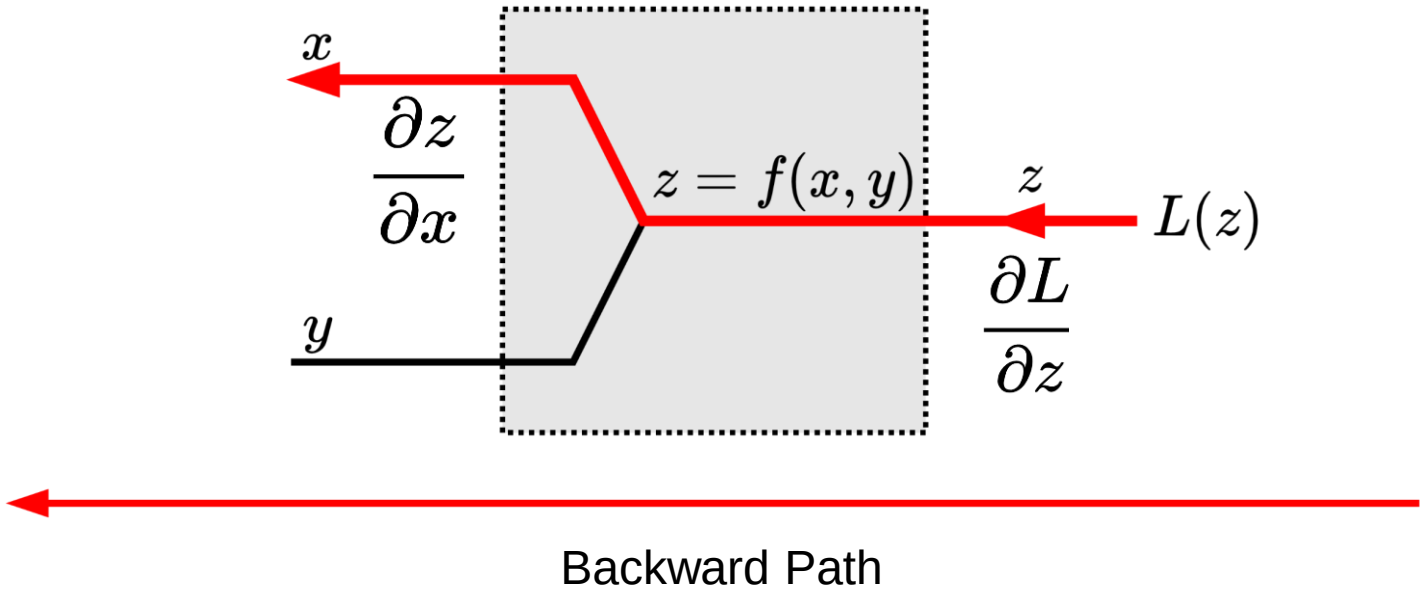
In practice, local gradients are usually computed and stored to memory during the forward path phase. This way the whole procedure can be carried out in an efficient way since the operation of the function derivation is computationally inexpensive.

Computational Graph

Abstract function

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x} = \frac{\partial L}{\partial z} f'_x(x, y)$$

Already calculated during forward path

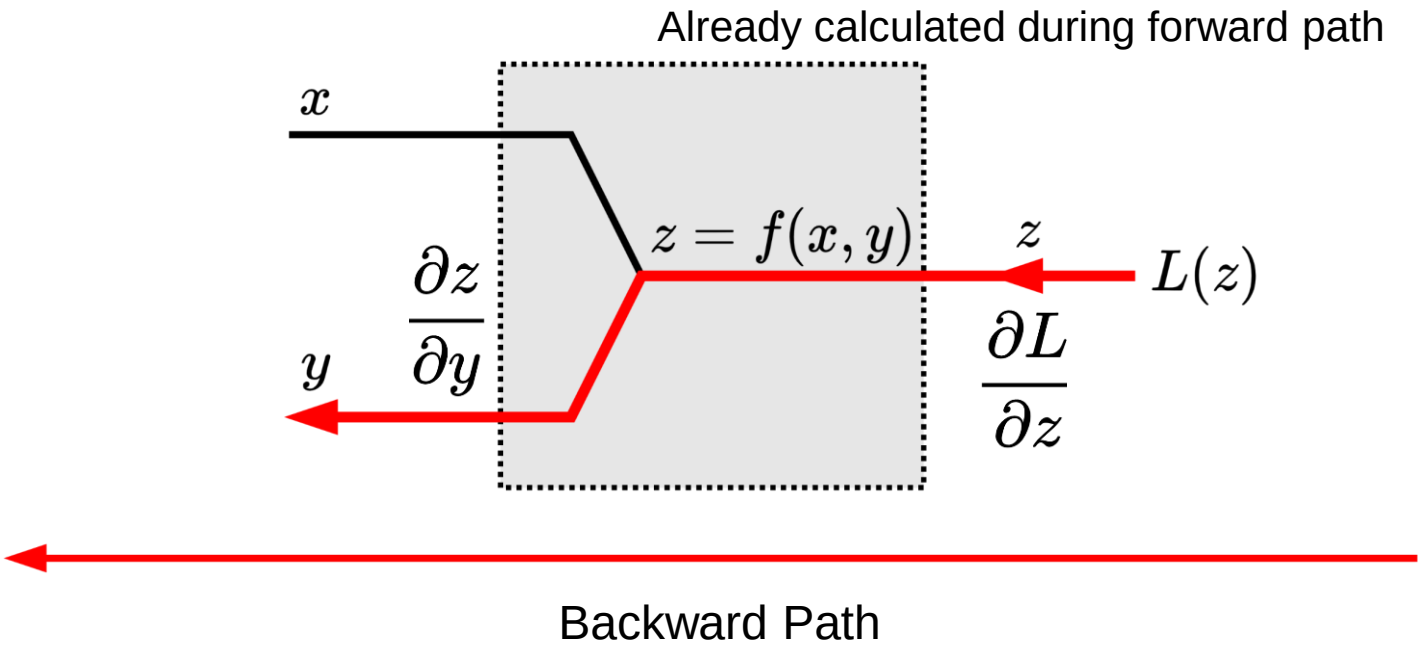


Finally, in order to calculate how does the error (or Loss function) change when x is being changed (dL/dx), we just need to multiply all downstream gradients along the way.

Computational Graph

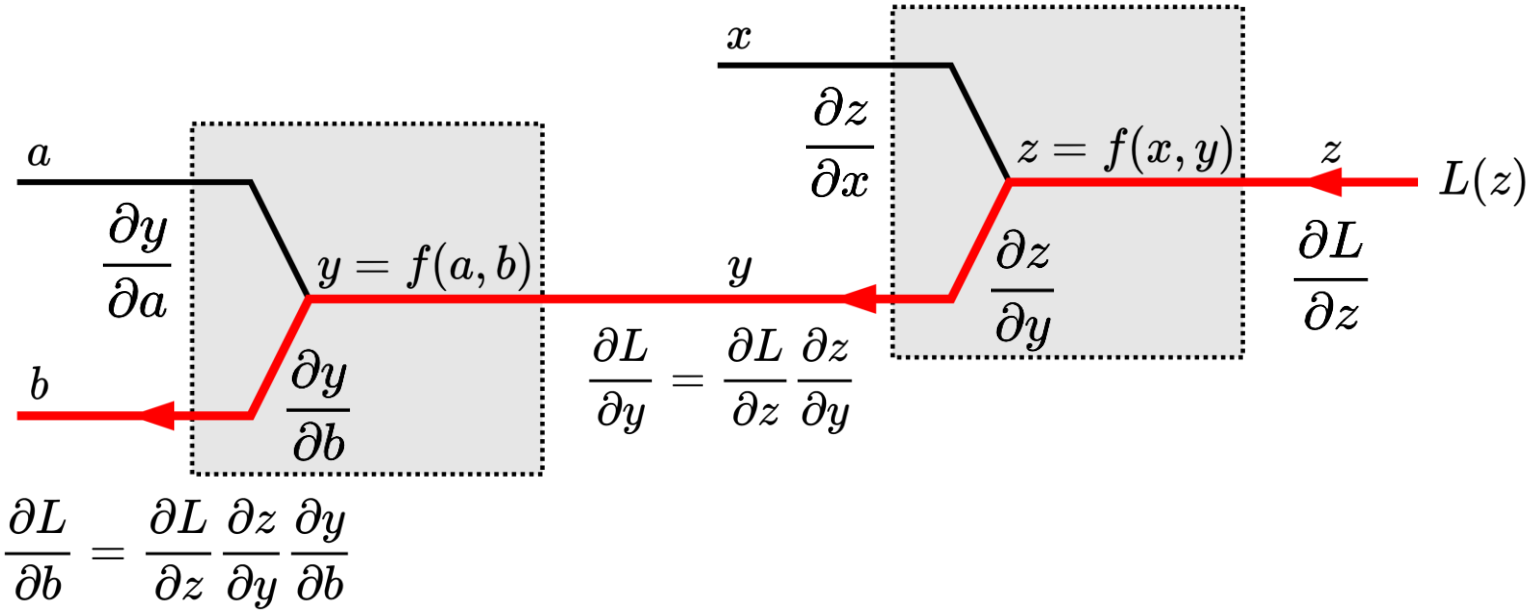
Abstract function

$$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y} = \frac{\partial L}{\partial z} f'_y(x, y)$$



Computational Graph

Abstract function



Now chaining these functions as well as overall derivative calculation becomes very easy, since the gradients are being propaged backwards through the network - hence the name backpropagation!.

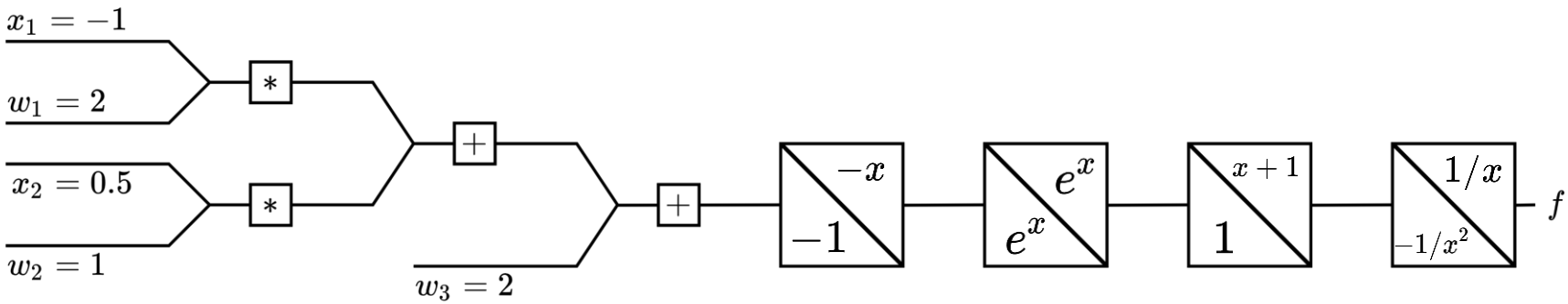
E.g., the complete gradient dL/db is just a multiplication of all local gradients along the path.

Computational Graph

Complex Example

$$f(x_1, x_2) = \frac{1}{1 + e^{-(x_1 w_1 + x_2 w_2 + w_3)}}$$

Local gradient 



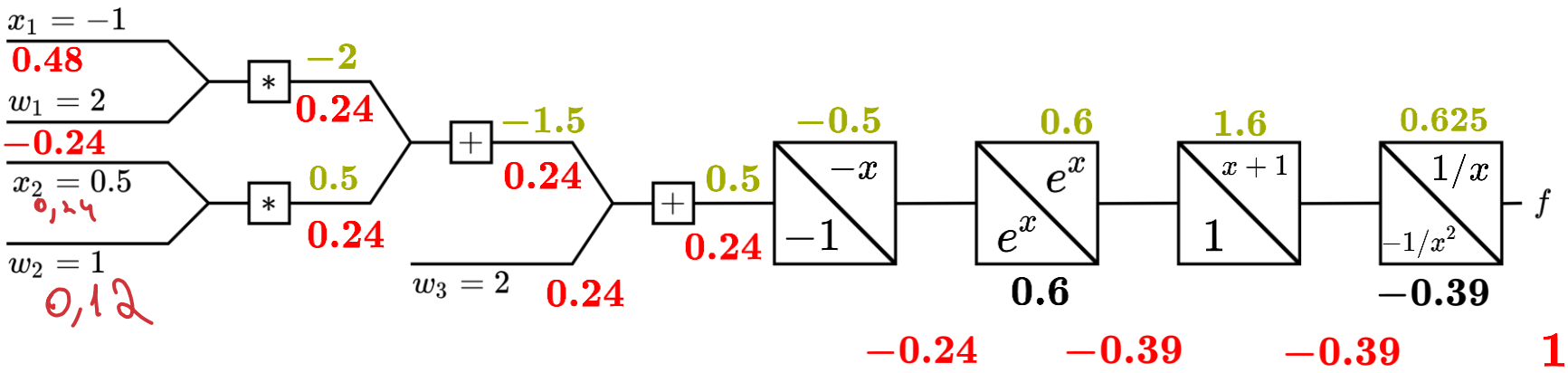
$$\frac{\partial f}{\partial w_1} = ? \qquad \frac{\partial f}{\partial x_1} = ?$$

Computational Graph

Complex Example

$$f(x_1, x_2) = \frac{1}{1 + e^{-(x_1 w_1 + x_2 w_2 + w_3)}}$$

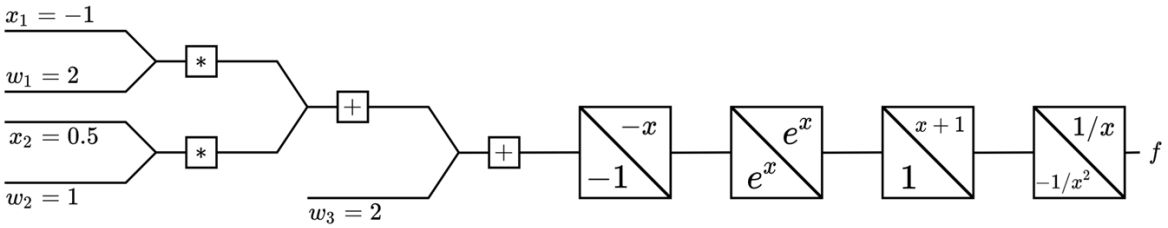
Local gradient 



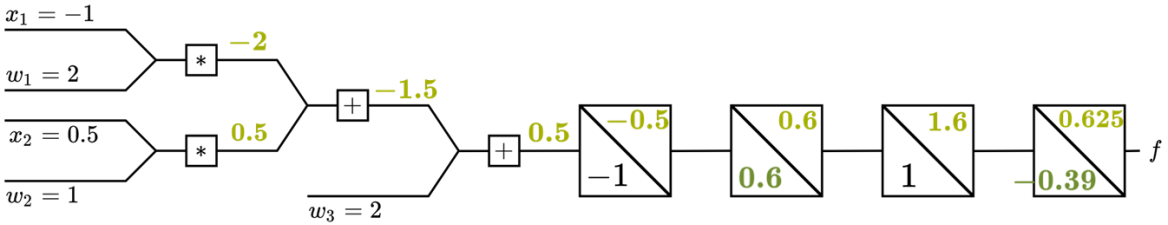
$$\frac{\partial f}{\partial x_1} = w_1 \frac{e^{-w_1 x_1 - w_2 x_2 - w_3}}{(e^{-w_1 x_1 - w_2 x_2 - w_3} + 1)^2} = 0.470074$$

$$\frac{\partial f}{\partial w_1} = x_1 \frac{e^{-w_1 x_1 - w_2 x_2 - w_3}}{(e^{-w_1 x_1 - w_2 x_2 - w_3} + 1)^2} = -0.235004$$

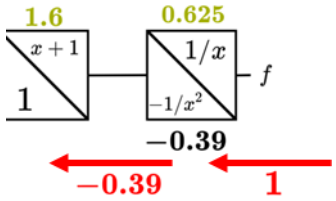
Let's have a look at a more complex example and calculate a derivate using this method. Here the complex function $f(x_1,x_2)$ is shown in the form of a computational graph, and we want to calculate df/dx_1:



First, we calculate the forward path:

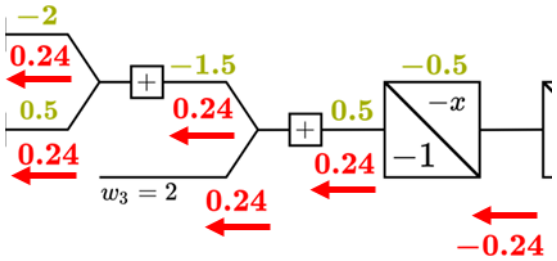


To get to df/dx_1 we just need to multiply the local gradients from f to x_1. At the end of the graph, we start with df/df = 1:

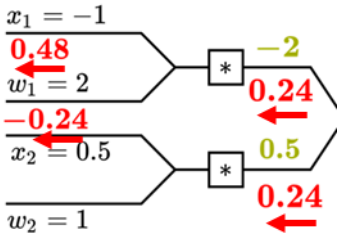


The next gradient can be calculated by simply multiplying the upstream gradient (df/df = 1) with the local derivative (-0.39), resulting in -0.39.

This get repeated along the computational graph. Applying previously introduced rules we know for additions that the gradient remains the same (0.24):



whereas for multiplications we need to multiply the upstream gradient (0.24) by the factor from other branch:



df / dx_1 = 0.24 * w_1 = 0.48

df / dw_1 = 0.24 * x_1 = -0.24

Finally, we can also show that we would get the same result if we used the analytic method:

$$\frac{\partial f}{\partial x_1} = w_1 \frac{e^{-w_1 x_1 - w_2 x_2 - w_3}}{(e^{-w_1 x_1 - w_2 x_2 - w_3} + 1)^2} = 0.470074$$

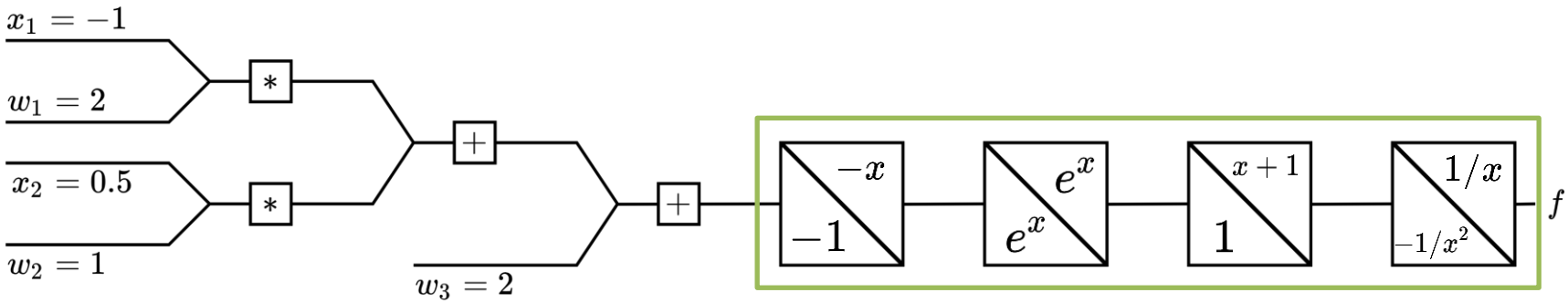
$$\frac{\partial f}{\partial w_1} = x_1 \frac{e^{-w_1 x_1 - w_2 x_2 - w_3}}{(e^{-w_1 x_1 - w_2 x_2 - w_3} + 1)^2} = -0.235004$$

Computational Graph

Complex Example

$$f(x_1, x_2) = \frac{1}{1 + e^{-(x_1 w_1 + x_2 w_2 + w_3)}}$$

Local gradient



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

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

These four functions can be combined into one with a simple derivative.

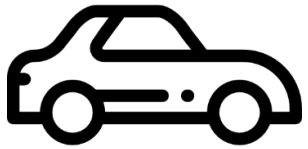
Deep Neural Networks
Prof. Dr. Markus Lienkamp

(Domagoj Majstorović, M. Sc.)

Agenda

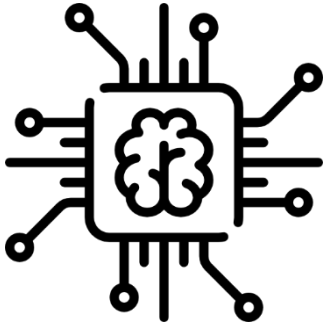
1. Chapter: Backpropagation

- 1.1 Computational Graph
- 1.2 Single Neuron
- 1.3 Neural Chain
- 1.4 Neural Network



2. Chapter: Neuronal Networks

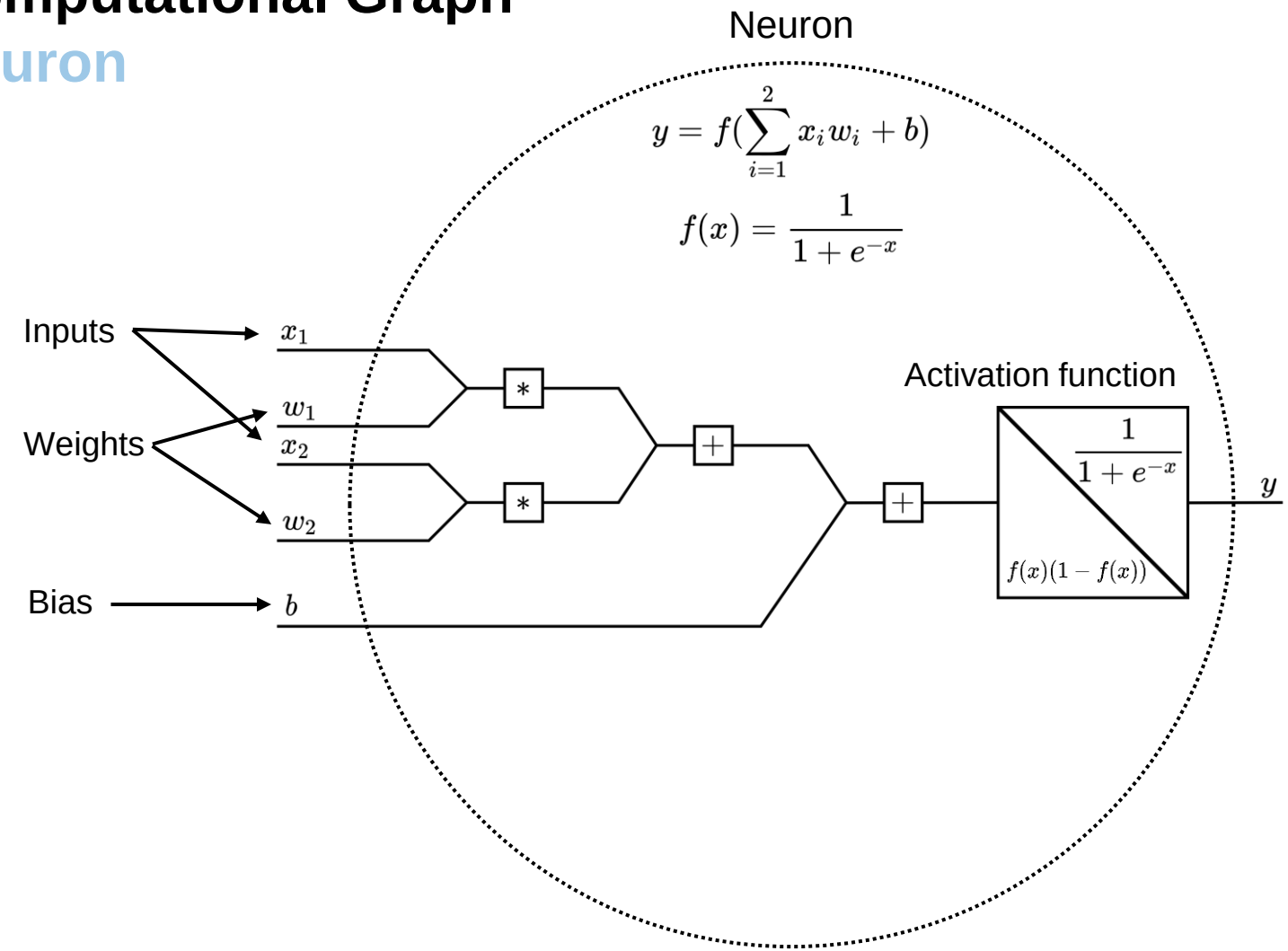
- 2.1 Activation Functions
- 2.2 Fully Connected Layer
- 2.3 Batches and Weight Initialization
- 2.4 Training the Network



3. Chapter: Summary

Computational Graph

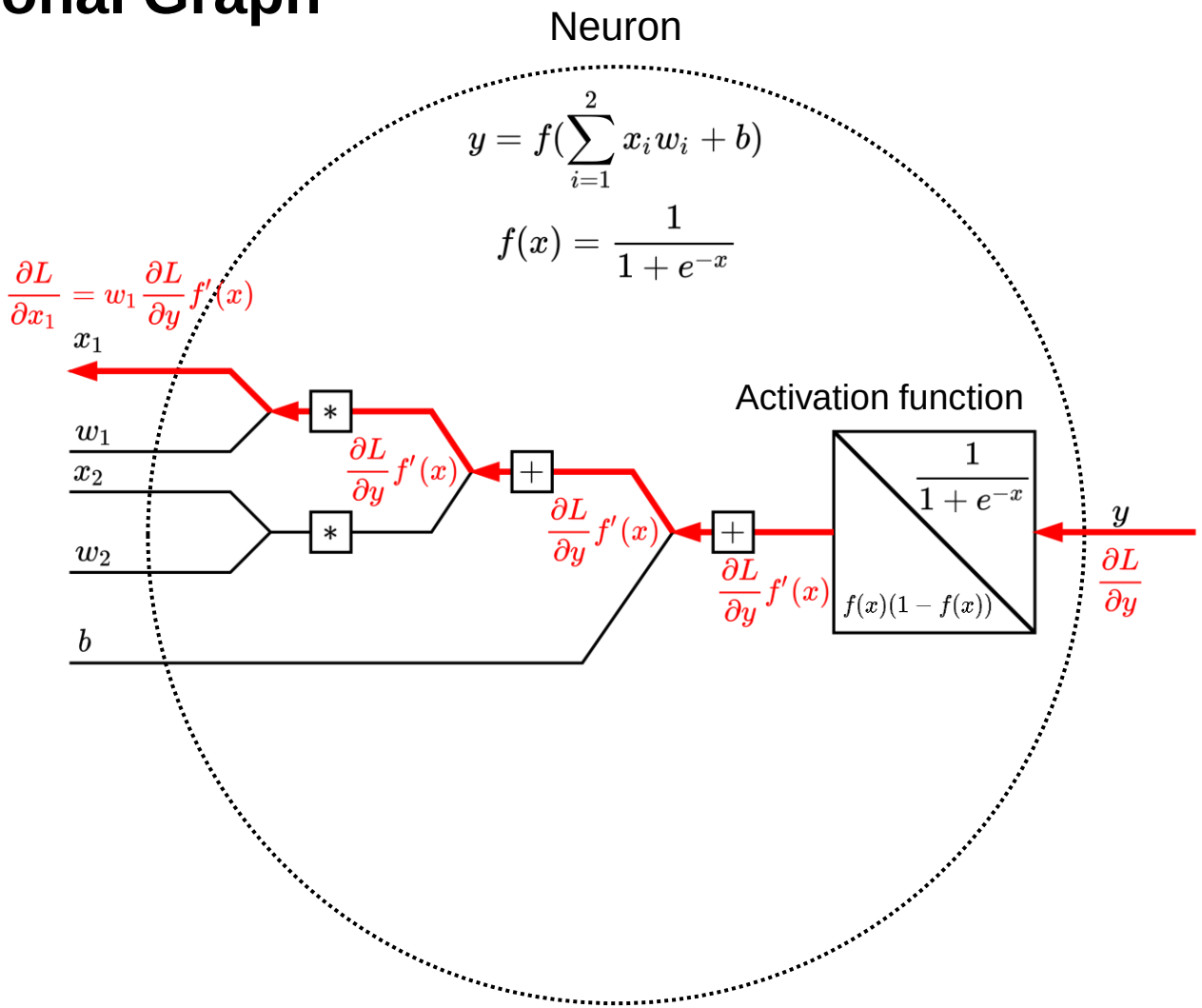
Neuron



The result is the computational graph for a single neuron.

Computational Graph

Neuron



Now we can apply the backpropagation algorithm to the single Neuron.

The downstream gradient dL/dx_1 can be calculated by multiplying the upstream gradient with the local derivative of the activation function. Since the additions do not impact gradients, the gradient will remain the same.

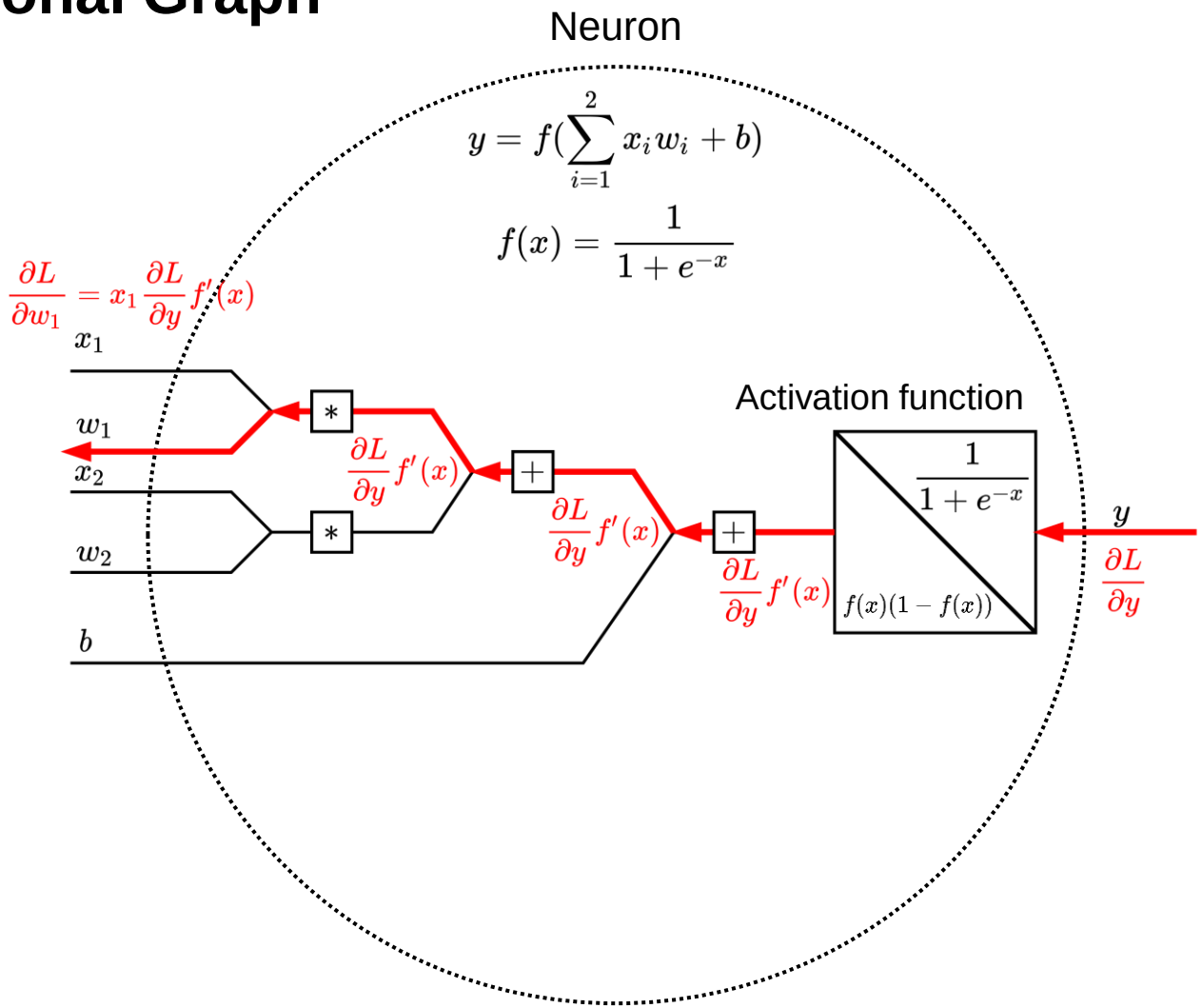
Lastly, we need to multiply the gradient with the other factor (w_1) which will result with the following expression:

$dL/dx_1 = dL/dy\ f'(x)\ w_1$ where,

- dL/dy = upstream gradient
- f'(x) = local derivative of activation function
- w1 = weight (other) factor

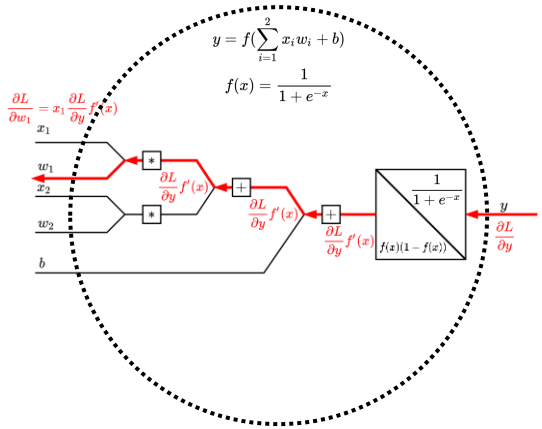
Computational Graph

Neuron

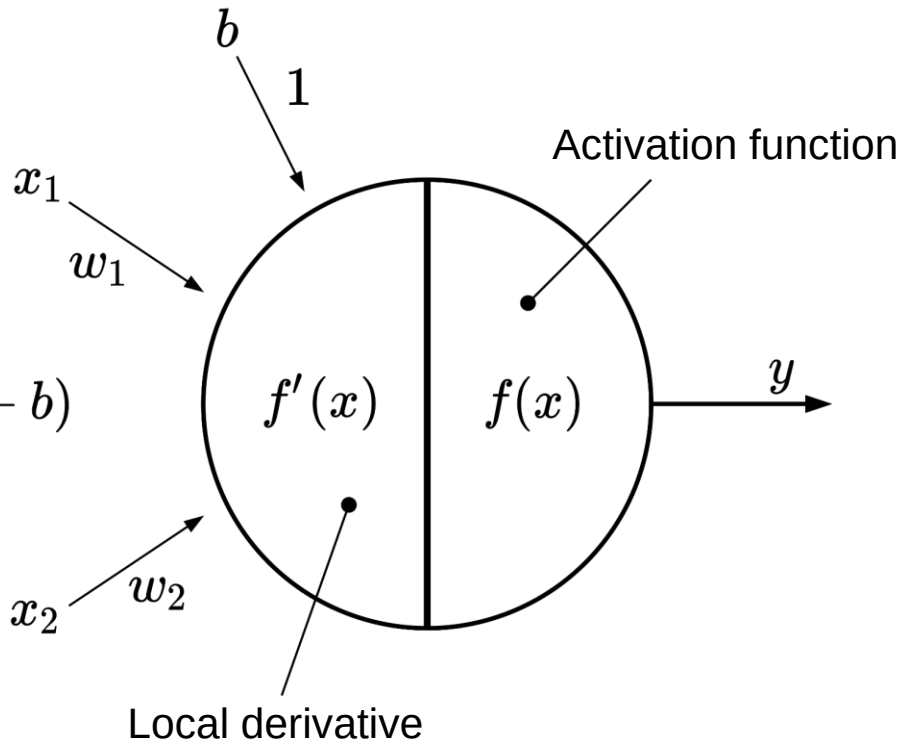


Computational Graph

Neuron

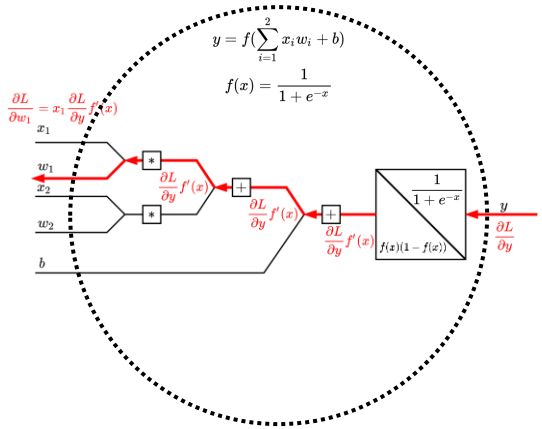


$$y = f(\sum_{i=1}^{2} x_i w_i + b)$$



Computational Graph

Neuron



$$\Delta w = -\alpha \frac{\partial L}{\partial w}$$

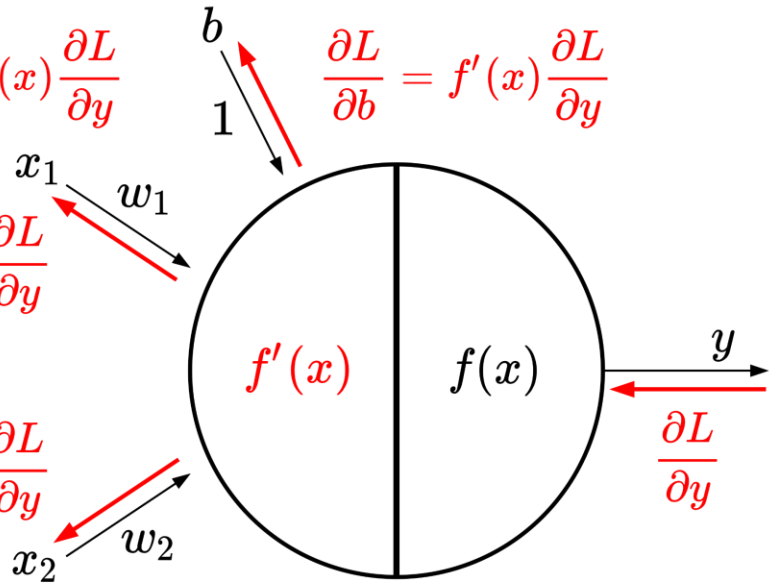
$$\frac{\partial L}{\partial x_1} = w_1 f'(x) \frac{\partial L}{\partial y}$$

$$\frac{\partial L}{\partial b} = f'(x) \frac{\partial L}{\partial y}$$

$$\frac{\partial L}{\partial w_1} = x_1 f'(x) \frac{\partial L}{\partial y}$$

$$\frac{\partial L}{\partial w_2} = x_2 f'(x) \frac{\partial L}{\partial y}$$

$$\frac{\partial L}{\partial x_2} = w_2 f'(x) \frac{\partial L}{\partial y}$$



The computational graph of a neuron can be simplified by hiding the inner additions and multiplications and just showing the function with a corresponding local derivative.

In such case downstream gradients can be calculated as shown.

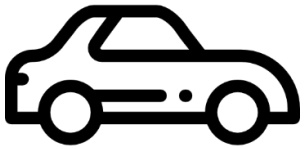
Deep Neural Networks
Prof. Dr. Markus Lienkamp

(Domagoj Majstorović, M. Sc.)

Agenda

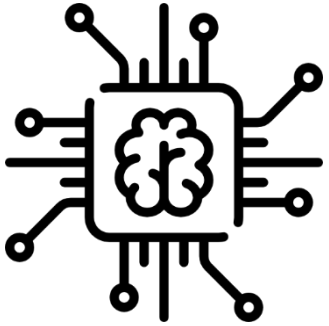
1. Chapter: Backpropagation

- 1.1 Computational Graph
- 1.2 Single Neuron
- 1.3 Neural Chain
- 1.4 Neural Network



2. Chapter: Neuronal Networks

- 2.1 Activation Functions
- 2.2 Fully Connected Layer
- 2.3 Batches and Weight Initialization
- 2.4 Training the Network



3. Chapter: Summary

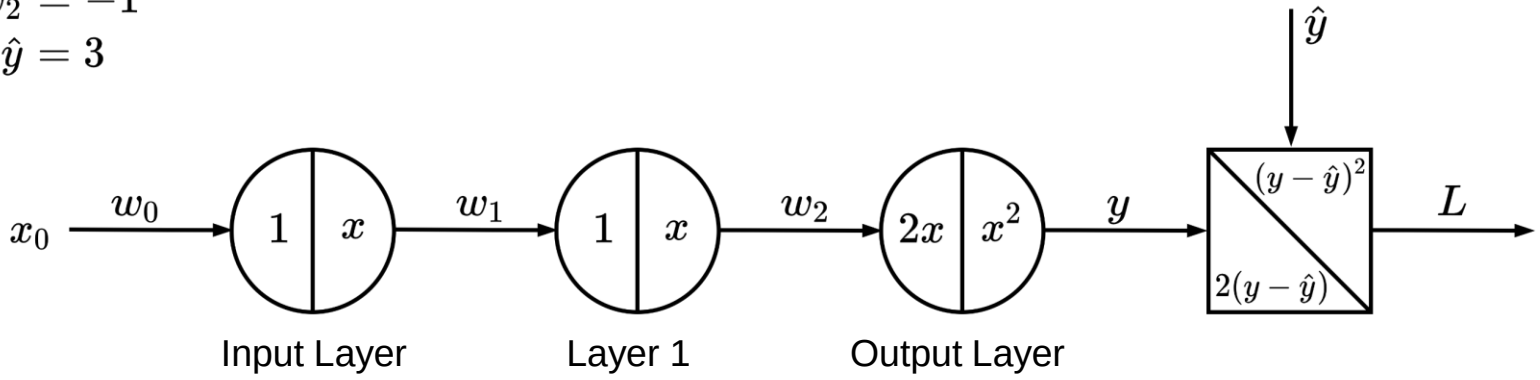
Backpropagation

Neural Chain

$$x_0 = 3$$
$$w_0 = 1$$
$$w_1 = 0.5$$
$$w_2 = -1$$
$$\hat{y} = 3$$

We want to find:

$$\Delta w_1 = -\alpha \frac{\partial L}{\partial w_1} \qquad \Delta w_2 = -\alpha \frac{\partial L}{\partial w_2}$$



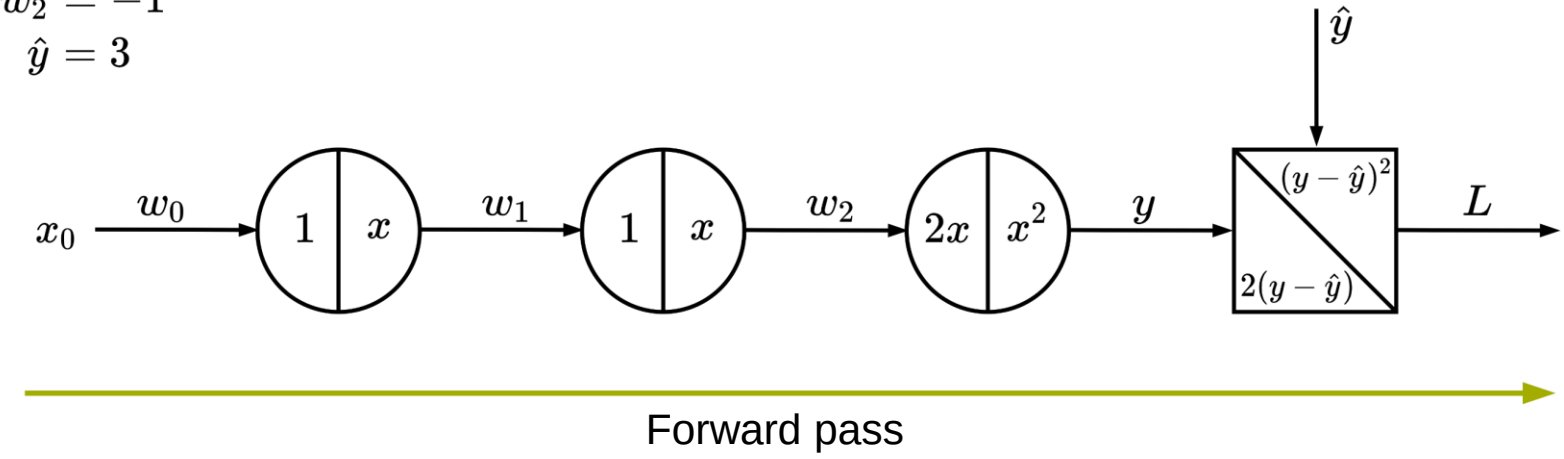
Backpropagation

Neural Chain

$$x_0 = 3$$
$$w_0 = 1$$
$$w_1 = 0.5$$
$$w_2 = -1$$
$$\hat{y} = 3$$

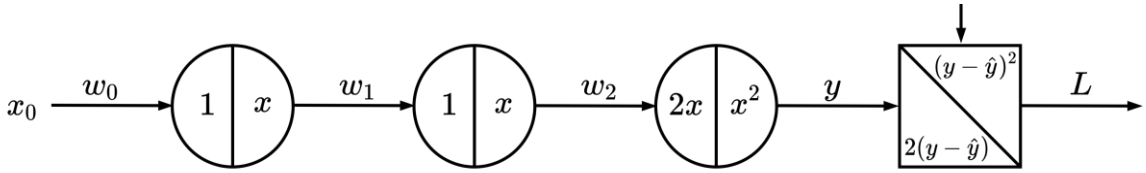
We want to find:

$$\Delta w_1 = -\alpha \frac{\partial L}{\partial w_1} \qquad \Delta w_2 = -\alpha \frac{\partial L}{\partial w_2}$$



Backpropagation

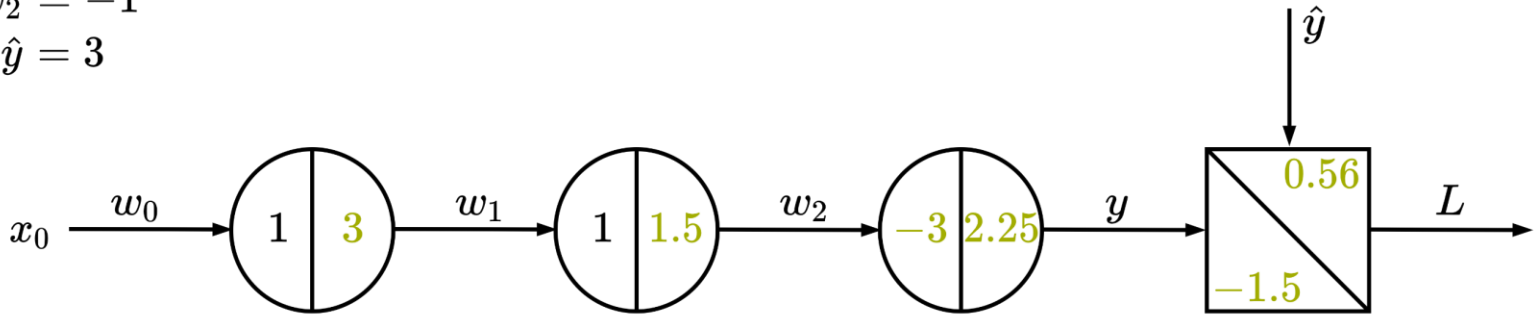
Neural Chain



$$x_0 = 3$$
$$w_0 = 1$$
$$w_1 = 0.5$$
$$w_2 = -1$$
$$\hat{y} = 3$$

We want to find:

$$\Delta w_1 = -\alpha \frac{\partial L}{\partial w_1} \qquad \Delta w_2 = -\alpha \frac{\partial L}{\partial w_2}$$



Forward pass

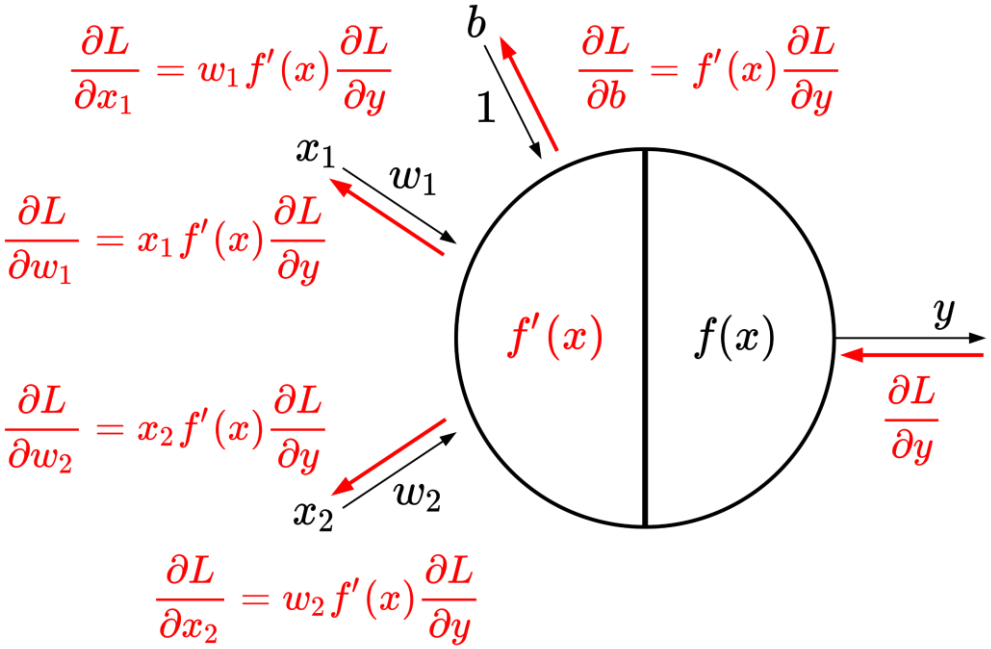
Backward pass

$$\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}$$

Backpropagation

Neural Chain

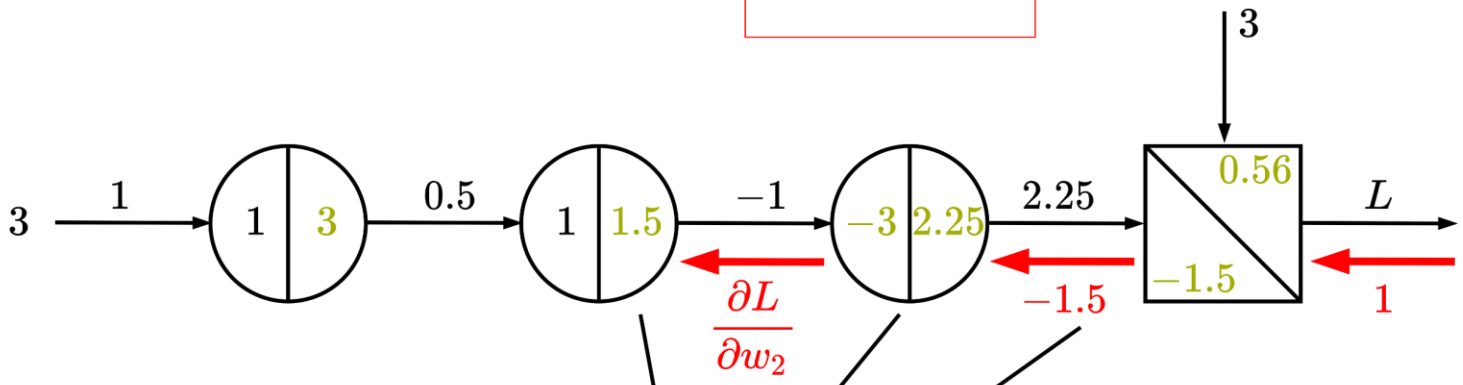
$$\Delta w = -\alpha \frac{\partial L}{\partial w}$$



Backpropagation

Neural Chain

$$\Delta w_1 = -\alpha \frac{\partial L}{\partial w_1} \qquad \boxed{\Delta w_2 = -\alpha \frac{\partial L}{\partial w_2}}$$



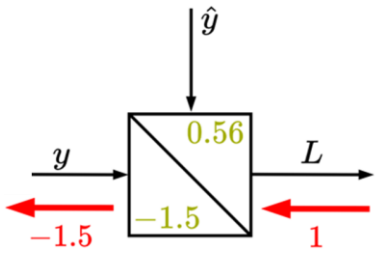
$$\frac{\partial L}{\partial w_2} = x_2 f'(x_2) \frac{\partial L}{\partial y} = 1.5 \cdot (-3) \cdot (-1.5) = 6.75$$

$$\frac{\partial L}{\partial x_2} = w_2 f'(x_2) \frac{\partial L}{\partial y} = (-1) \cdot (-3) \cdot (-1.5) = -4.5$$

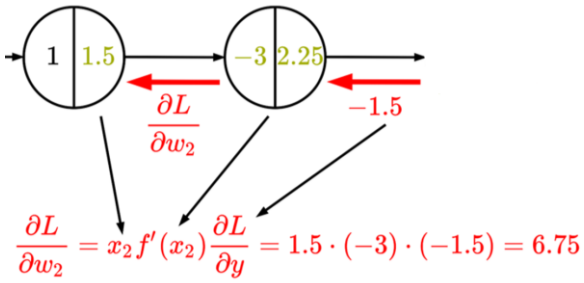
Let's look now at a chain of neurons (i.e., a small neural network). Here we have three neurons linked together and a loss function at the very end. Since we want to minimize the output of the loss function and only the parameters, we can change are w_1 and w_2, we need to know if we need to increase or decrease w_1 and w_2 values. This is defined with the derivative of dL/dw_1 or dL/dw_2. If a derivative is positive, that means that the Loss function will increase if we increase the weight parameter. Therefore, we must change w in the other direction (negative multiplication). Finally, we multiply the gradient with a predefined learning rate parameter alpha to influence the rate of change.

$$\Delta w_1 = -\alpha \frac{\partial L}{\partial w_1}$$

To calculate this, we use the backpropagation algorithm. dL/dL is the starting upstream gradient which always equals to 1. To calculate the next gradient, we simply multiply dL/dL with the local gradient of the loss function:



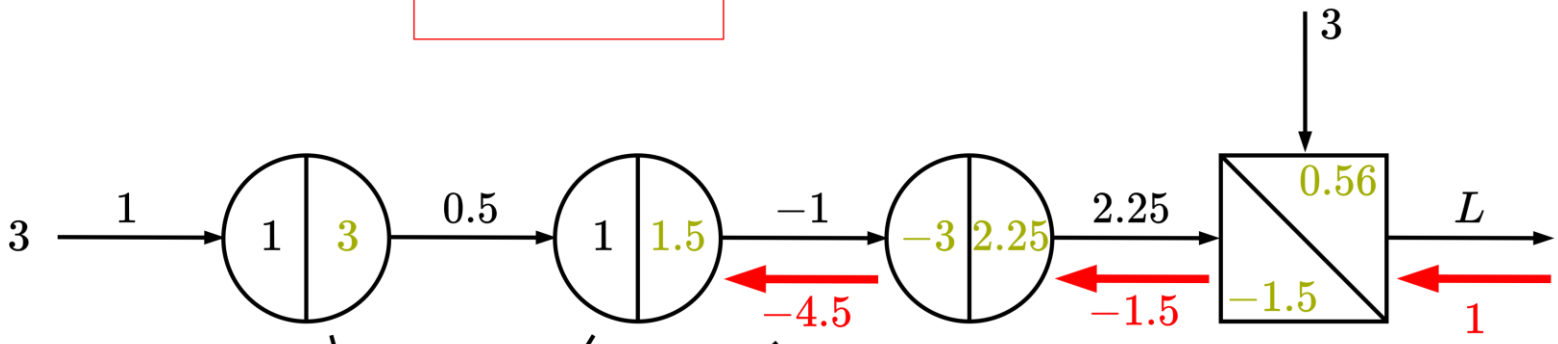
dL/dw_2 can be calculated by multiplying the downstream gradient (-1.5) with the local gradient (-3) and "the other factor" (1.5)



Backpropagation

Neural Chain

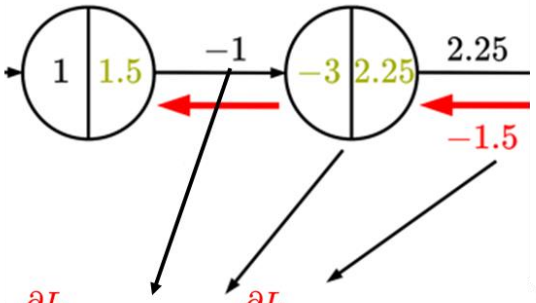
$$\Delta w_1 = -\alpha \frac{\partial L}{\partial w_1} \qquad \Delta w_2 = -\alpha \frac{\partial L}{\partial w_2}$$



$$\frac{\partial L}{\partial w_1} = x_1 f'(x_1) \frac{\partial L}{\partial x_2} = 3 \cdot 1 \cdot (-4.5) = -13.5$$

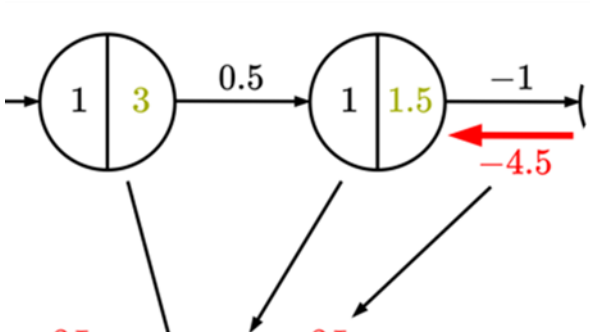
$$\frac{\partial L}{\partial x_1} = w_1 f'(x_1) \frac{\partial L}{\partial x_2} = 0.5 \cdot 1 \cdot (-4.5) = -2.25$$

To calculate dL/dx_2 we need the upstream gradient dL/dw_2 as well as include the "other" coefficient being the weight w_2 which produces the following:



$$\frac{\partial L}{\partial x_2} = w_2 f'(x_2) \frac{\partial L}{\partial y} = (-1) \cdot (-3) \cdot (-1.5) = -4.5$$

Having this gradient, it is possible to calculate dL/dw_1 which gives us the change of w_1, as well as the change of x_1:



$$\frac{\partial L}{\partial w_1} = x_1 f'(x_1) \frac{\partial L}{\partial x_2} = 3 \cdot 1 \cdot (-4.5) = -13.5$$

$$\frac{\partial L}{\partial x_1} = w_1 f'(x_1) \frac{\partial L}{\partial x_2} = 0.5 \cdot 1 \cdot (-4.5) = -2.25$$

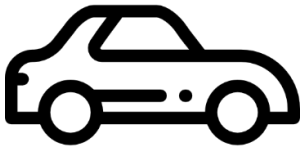
Deep Neural Networks
Prof. Dr. Markus Lienkamp

(Domagoj Majstorović, M. Sc.)

Agenda

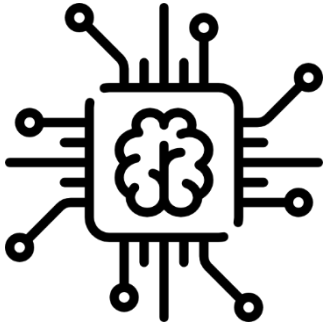
1. Chapter: Backpropagation

- 1.1 Computational Graph
- 1.2 Single Neuron
- 1.3 Neural Chain
- 1.4 Neural Network



2. Chapter: Neuronal Networks

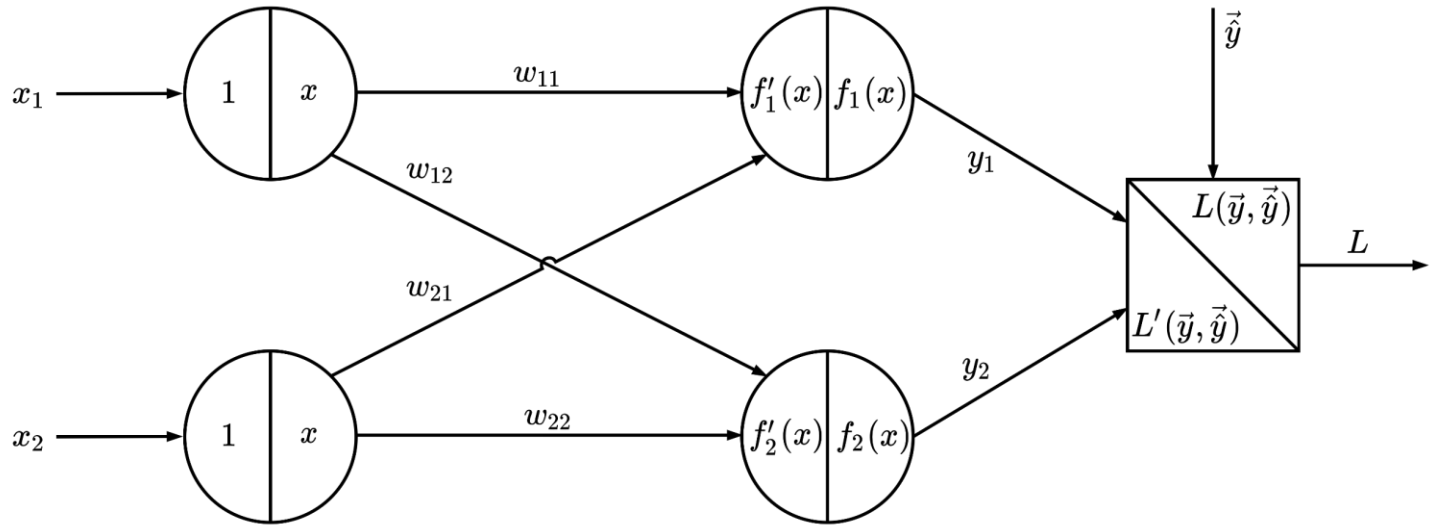
- 2.1 Activation Functions
- 2.2 Fully Connected Layer
- 2.3 Batches and Weight Initialization
- 2.4 Training the Network



3. Chapter: Summary

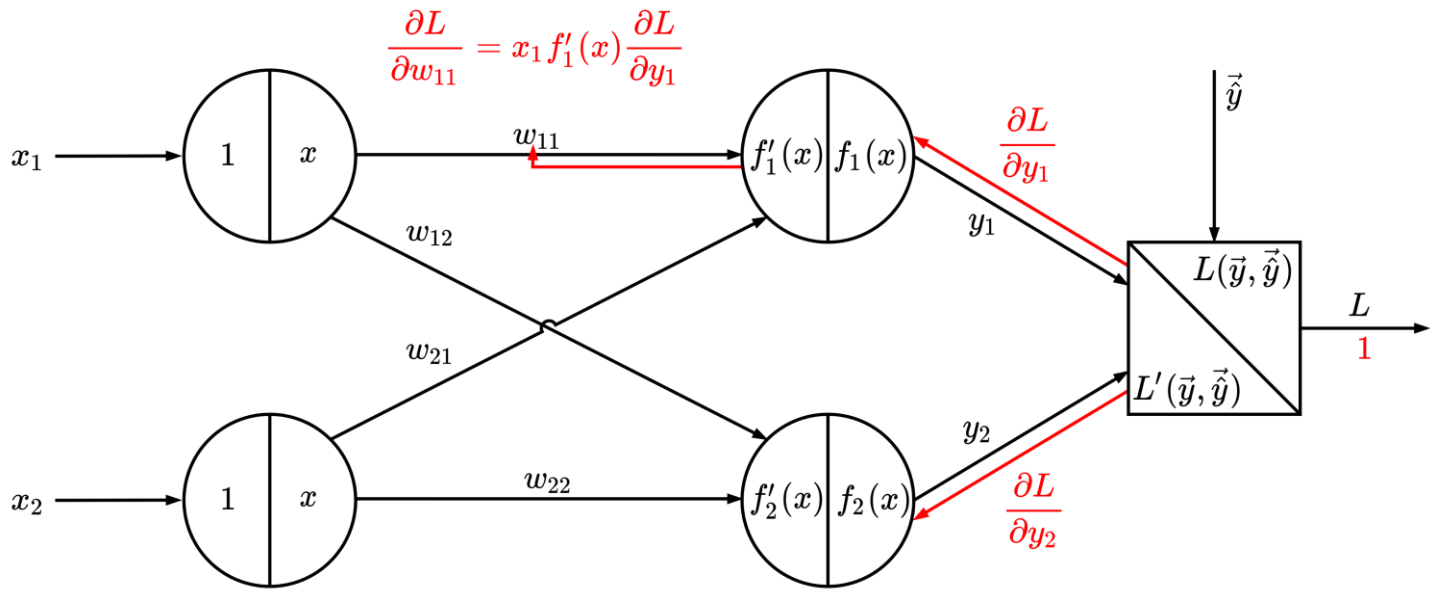
Backpropagation

Neural Network



Backpropagation

Neural Network



$$\frac{\partial L}{\partial w_{11}} = x_1 f_1'(x) \frac{\partial L}{\partial y_1} \quad \frac{\partial L}{\partial w_{12}} = x_1 f_2'(x) \frac{\partial L}{\partial y_2} \quad \frac{\partial L}{\partial w_{21}} = x_2 f_1'(x) \frac{\partial L}{\partial y_1} \quad \frac{\partial L}{\partial w_{22}} = x_2 f_2'(x) \frac{\partial L}{\partial y_2}$$

Finally, we can connect multiple neurons into a network. The very same rules for calculating the local gradients still apply.

Backpropagation
Neural Network

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \qquad \Delta W = -\alpha \begin{pmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{12}} \\ \frac{\partial L}{\partial w_{21}} & \frac{\partial L}{\partial w_{22}} \end{pmatrix}$$

$$\vec{y} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

$$W = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} \qquad \Delta W = -\alpha \begin{pmatrix} x_1 \frac{\partial L}{\partial y_1} f_1' & x_1 \frac{\partial L}{\partial y_2} f_2' \\ x_2 \frac{\partial L}{\partial y_1} f_1' & x_2 \frac{\partial L}{\partial y_2} f_2' \end{pmatrix}$$

Hadamard Product

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \odot \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 1a & 2b \\ 3c & 4d \end{pmatrix} \Rightarrow \quad \Delta W = -\alpha \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \left[\begin{pmatrix} \frac{\partial L}{\partial y_1} \\ \frac{\partial L}{\partial y_2} \end{pmatrix} \odot \begin{pmatrix} f_1' \\ f_2' \end{pmatrix} \right]^T$$

Backpropagation

Neural Network

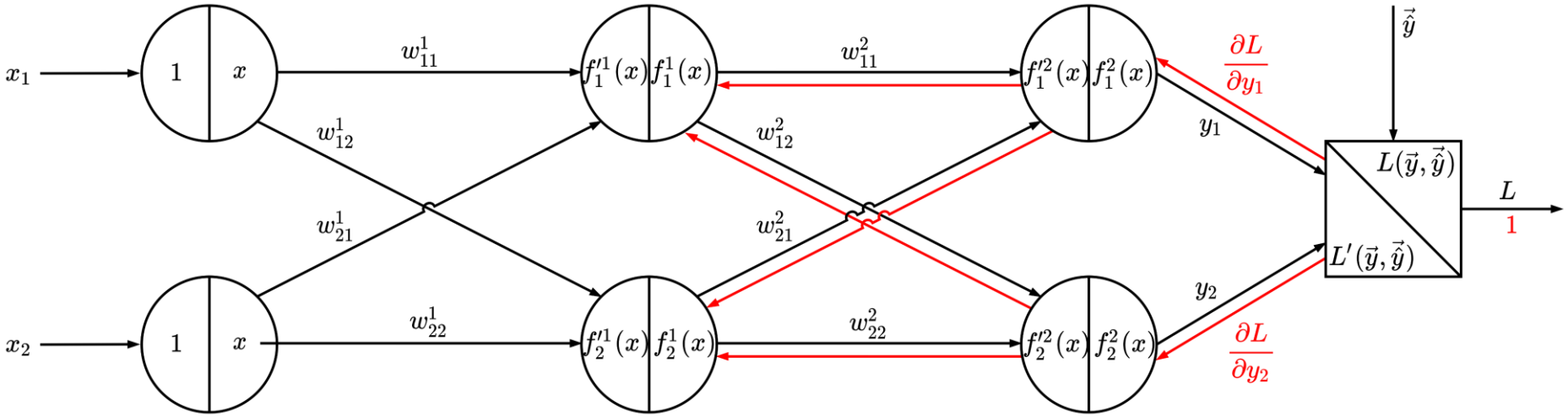
$$\Delta W = -\alpha \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \left[\begin{pmatrix} \frac{\partial L}{\partial y_1} \\ \frac{\partial L}{\partial y_2} \end{pmatrix} \odot \begin{pmatrix} f_1' \\ f_2' \end{pmatrix} \right]^T = -\alpha \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \left[\begin{pmatrix} \frac{\partial L}{\partial y_1} f_1' \\ \frac{\partial L}{\partial y_2} f_2' \end{pmatrix} \right]^T$$

$$= -\alpha \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \begin{pmatrix} \frac{\partial L}{\partial y_1} f_1' & \frac{\partial L}{\partial y_2} f_2' \end{pmatrix} = -\alpha \begin{pmatrix} x_1 \frac{\partial L}{\partial y_1} f_1' & x_1 \frac{\partial L}{\partial y_2} f_2' \\ x_2 \frac{\partial L}{\partial y_1} f_1' & x_2 \frac{\partial L}{\partial y_2} f_2' \end{pmatrix}$$

In order to calculate change of weights in the network we can simply use vector operations (linear algebra).

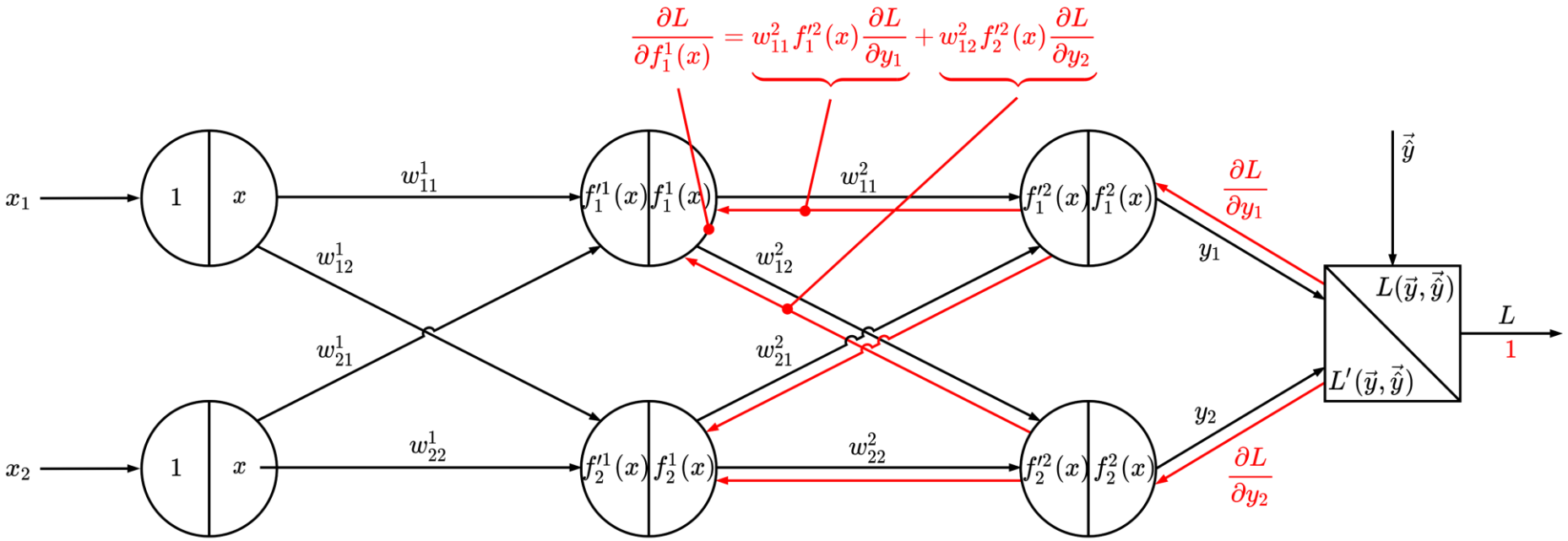
Backpropagation

Neural Network, one hidden layer



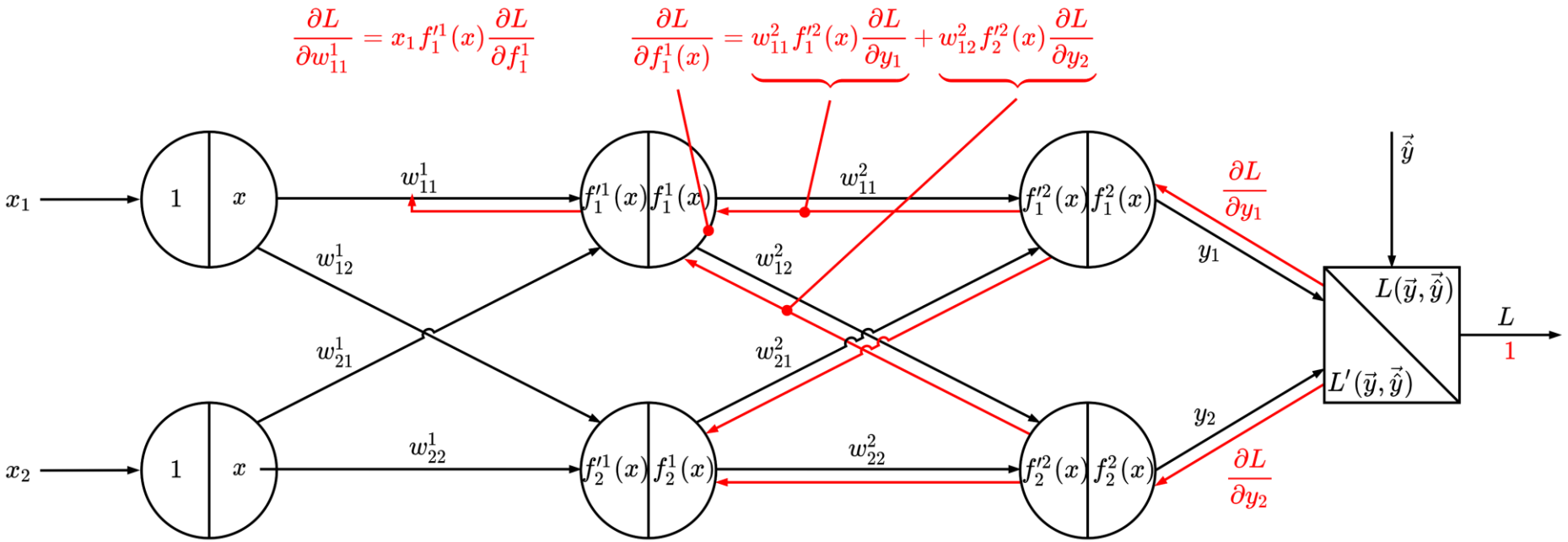
Backpropagation

Neural Network, one hidden layer



Backpropagation

Neural Network, one hidden layer

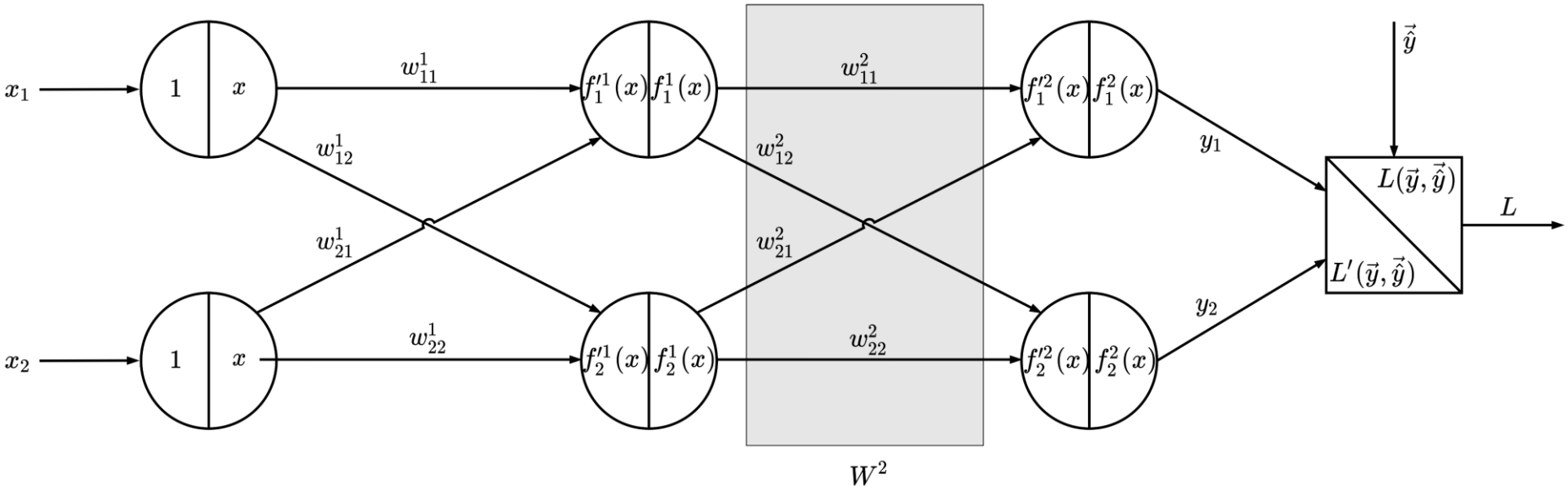


$$\frac{\partial L}{\partial w_{11}^1} = x_1 f_1'^1(x) \frac{\partial L}{\partial f_1^1} \quad \frac{\partial L}{\partial w_{12}^1} = x_1 f_2'^1(x) \frac{\partial L}{\partial f_2^1} \quad \frac{\partial L}{\partial w_{21}^1} = x_2 f_1'^1(x) \frac{\partial L}{\partial f_1^1} \quad \frac{\partial L}{\partial w_{22}^1} = x_2 f_2'^1(x) \frac{\partial L}{\partial f_2^1}$$

Once two upstream gradients arrive at the neuron, the resulting gradient will be the sum of both gradients.

Backpropagation

Neural Network, one hidden layer



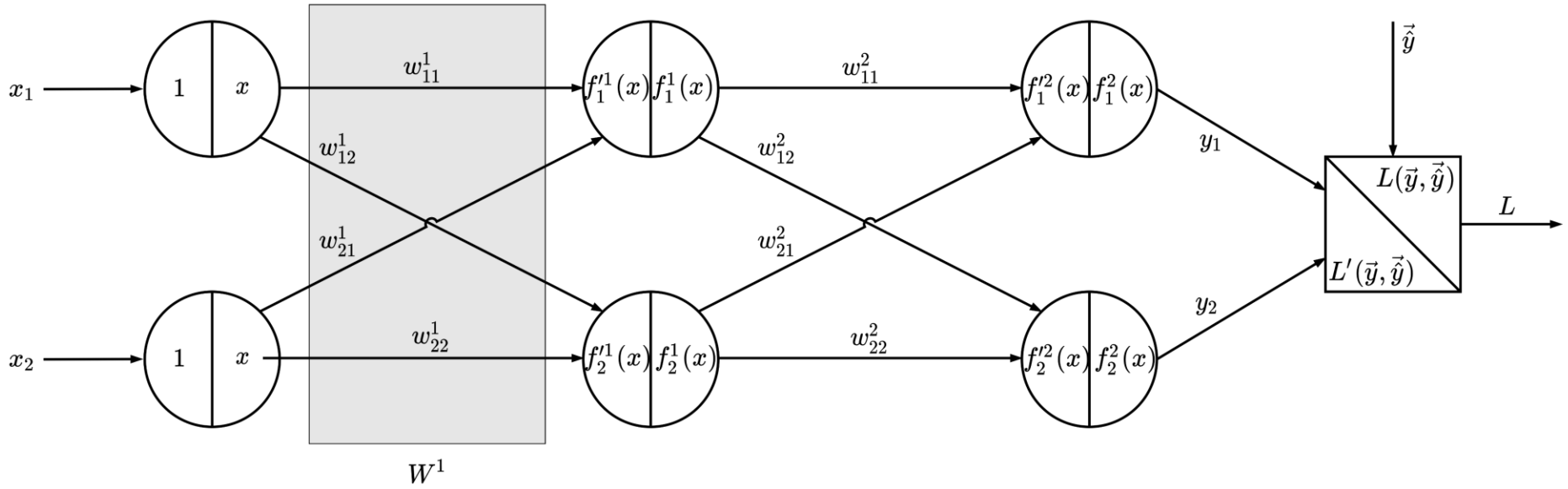
$$\Delta W^2 = -\alpha \begin{pmatrix} f_1^1 \\ f_2^1 \end{pmatrix} \left[\begin{pmatrix} \frac{\partial L}{\partial y_1} \\ \frac{\partial L}{\partial y_2} \end{pmatrix} \odot \begin{pmatrix} f_1'^2 \\ f_2'^2 \end{pmatrix} \right]^T$$

Forward Path

Backward Path

Backpropagation

Neural Network, one hidden layer



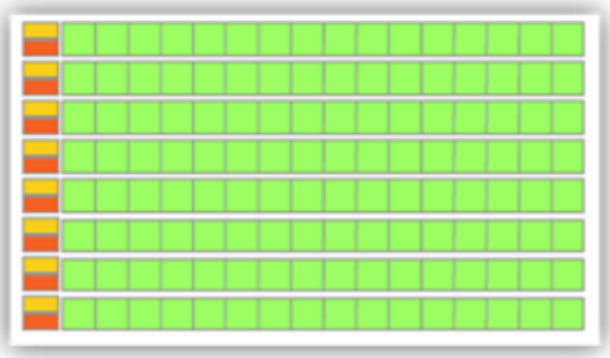
$$\Delta W^1 = -\alpha \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \left[\begin{pmatrix} \frac{\partial L}{\partial f_1^1} \\ \frac{\partial L}{\partial f_2^1} \end{pmatrix} \odot \begin{pmatrix} f_1'^1 \\ f_2'^1 \end{pmatrix} \right]^T = -\alpha \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \left[\left[\begin{pmatrix} w_{11}^2 & w_{12}^2 \\ w_{21}^2 & w_{22}^2 \end{pmatrix} \left[\begin{pmatrix} \frac{\partial L}{\partial y_1} \\ \frac{\partial L}{\partial y_2} \end{pmatrix} \odot \begin{pmatrix} f_1'^2 \\ f_2'^2 \end{pmatrix} \right]^T \right] \odot \begin{pmatrix} f_1'^1 \\ f_2'^1 \end{pmatrix} \right]^T$$

CPU

- Complex logic control
- Low compute density
- Large Cache
- Low latency tolerance

vs.

GPU



- Optimized for parallel computing
- High compute density
- Many calculation per memory access
- High latency tolerance

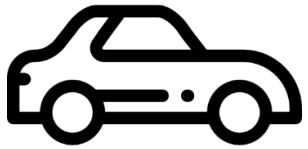
Deep Neural Networks
Prof. Dr. Markus Lienkamp

(Domagoj Majstorović, M. Sc.)

Agenda

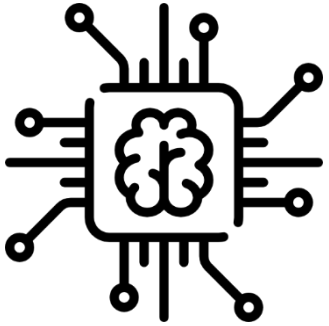
1. Chapter: Backpropagation

- 1.1 Computational Graph
- 1.2 Single Neuron
- 1.3 Neural Chain
- 1.4 Neural Network



2. Chapter: Neuronal Networks

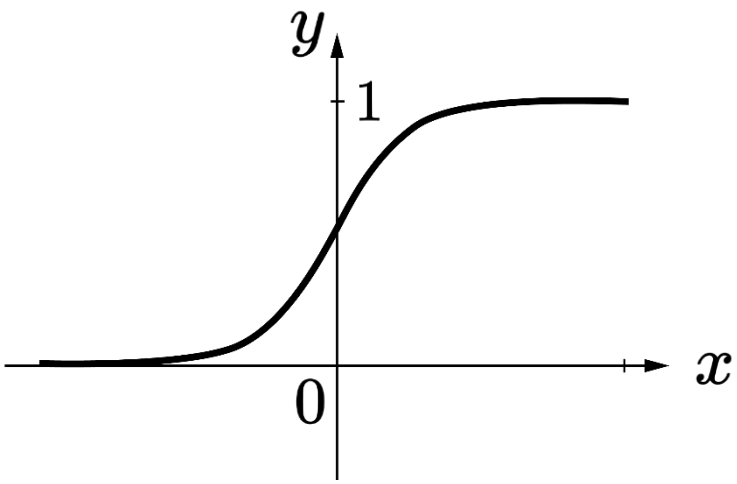
- 2.1 Activation Functions
- 2.2 Fully Connected Layer
- 2.3 Batches and Weight Initialization
- 2.4 Training the Network



3. Chapter: Summary

Activation Functions

Sigmoid function



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

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

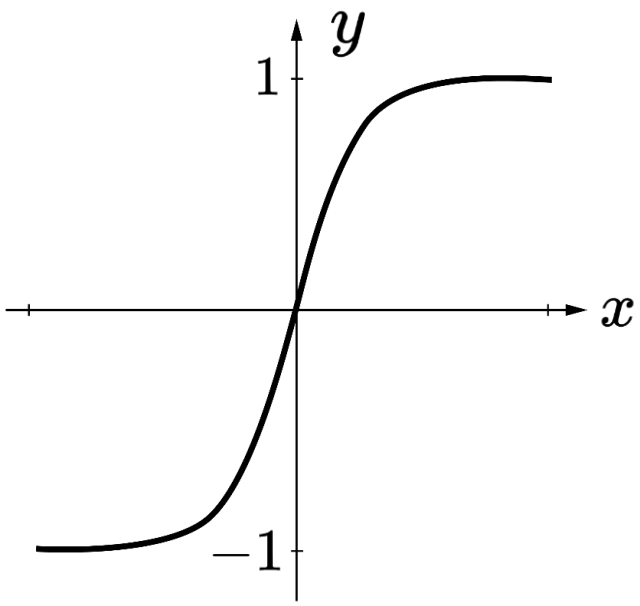
Two main problems:

1. Causes vanishing gradient:
   - Gradient nearly zero for very large or small x
   - *Kills* gradient and network stops to learn

2. Output isn‘t zero centered
   - Always all gradients positive or all negative, inefficient weight updates

Activation Functions

Tangens hyperbolicus (TanH)



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

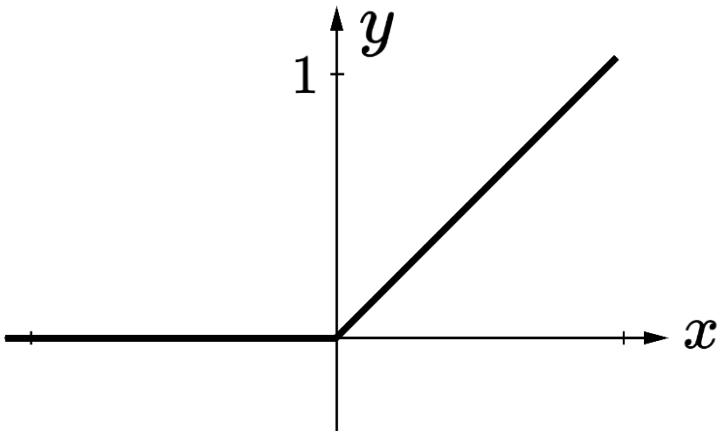
$$f'(x) = \frac{4}{(e^x + e^{-x})^2}$$

Better than sigmoid:

- Output is now zero-centered
- But still causes vanishing gradient

Activation Functions

Rectified Linear Unit (ReLU)



$$f(x) = \begin{cases} x, x \geq 0 \\ 0, x < 0 \end{cases}$$

$$f'(x) = \begin{cases} 1, x \geq 0 \\ 0, x < 0 \end{cases}$$

Most common activation function:

- Computationally efficient
- Converges very fast
- Does not activate all neurons at the same time

Problem:

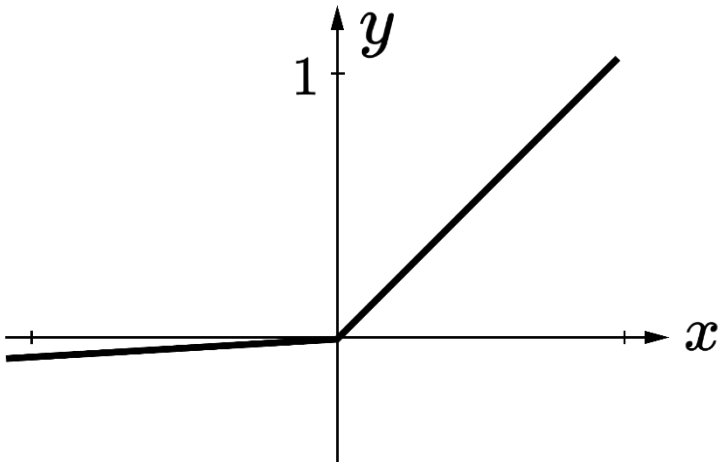
- Gradient is zero for x<0 and can cause vanishing gradient → dead ReLUs may happen
- Not zero-centered

Usage:

- Mostly used in hidden layers

Activation Functions

Leaky ReLU



$$f(x) = \begin{cases} x, x \geq 0 \\ ax, x < 0 \end{cases}$$

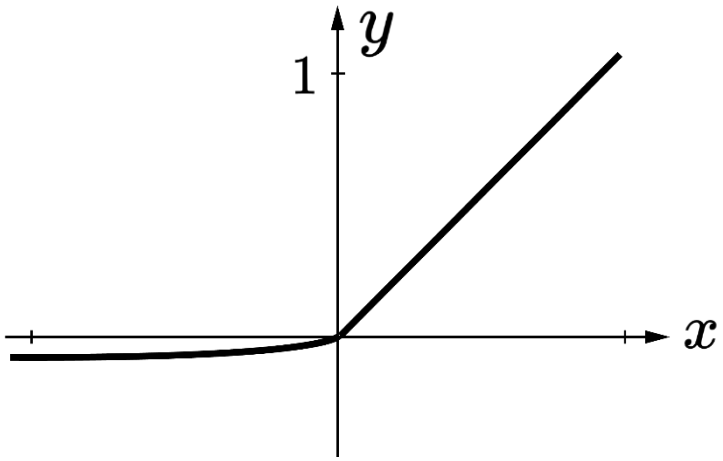
$$f'(x) = \begin{cases} 1, x \geq 0 \\ a, x < 0 \end{cases}$$

Improves on ReLU:

- Removes zero part of ReLU by adding a small slope. More stable then ReLU, but adds another parameter
- Computationally efficient
- Converges very fast
- Doesn‘t *die*
- Parameter ***a*** can also be learned by the network

Activation Functions

Exponential Linear Unit (ELU)



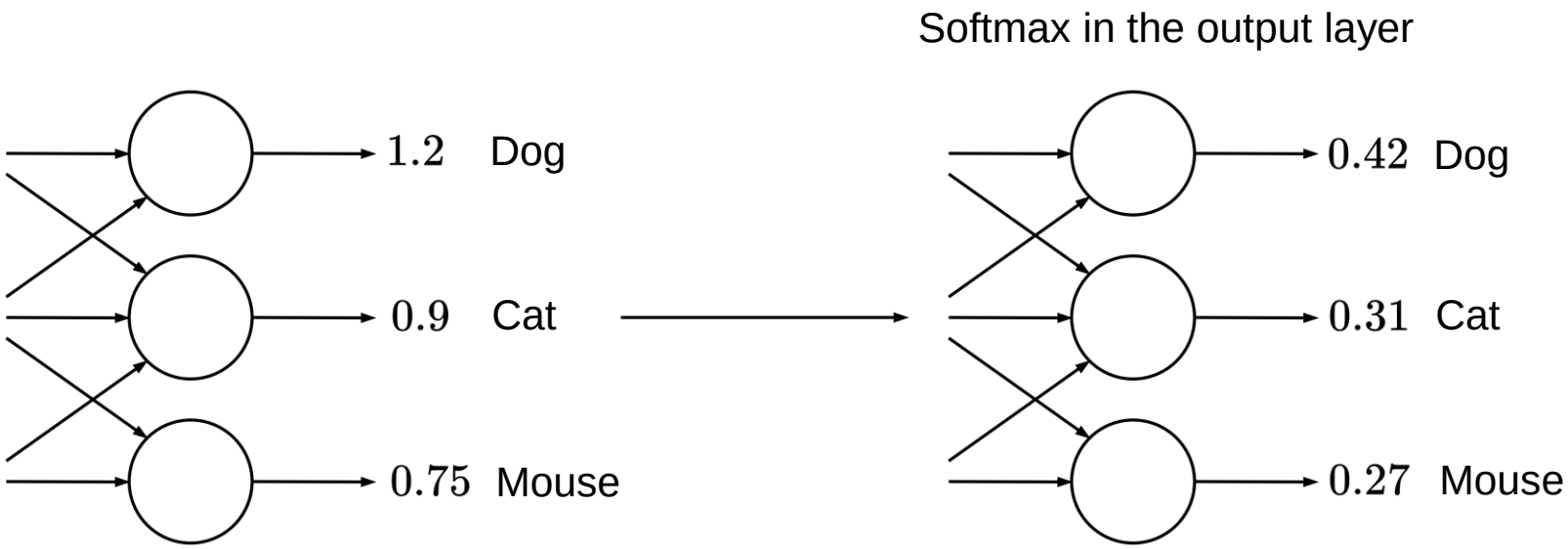
- Benefits of ReLU and Leaky ReLU
- Computation requires e^x

$$f(x) = \begin{cases} x, x \geq 0 \\ a(e^x - 1), x < 0 \end{cases}$$

$$f'(x) = \begin{cases} 1, x \geq 0 \\ ae^x, x < 0 \end{cases}$$

Activation Functions

Softmax function



$$f(x) = \frac{e^{y_i}}{\sum_{k=1}^{K} e^{y_k}}$$

- Type of Sigmoid function
- Handy for **classification** problems
- Divides by the sum of all outputs, allows for percentage representation

Activation Functions
Rule of thumb

- Sigmoid / ReLU + Softmax for classifiers
- Sigmoid, TanH sometimes avoided due to vanishing gradient
- ReLU mostly used today
- Start with ReLU. If you don‘t get optimal results, go for LeakyReLU or ELU

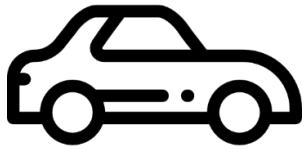
Deep Neural Networks
Prof. Dr. Markus Lienkamp

(Domagoj Majstorović, M. Sc.)

Agenda

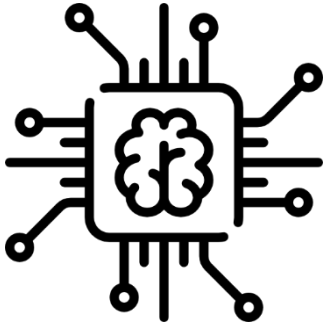
1. Chapter: Backpropagation

- 1.1 Computational Graph
- 1.2 Single Neuron
- 1.3 Neural Chain
- 1.4 Neural Network



2. Chapter: Neuronal Networks

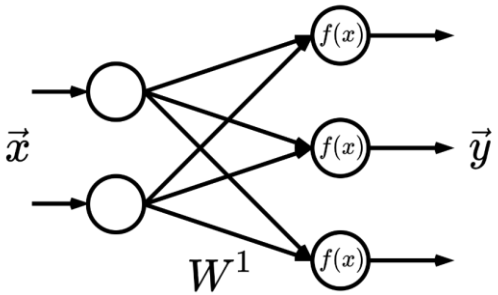
- 2.1 Activation Functions
- 2.2 Fully Connected Layer
- 2.3 Batches and Weight Initialization
- 2.4 Training the Network



3. Chapter: Summary

Fully Connected Layer

Dimensions



$$\vec{y} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} f(x_1 w_{11} + x_2 w_{21} + b_1) \\ f(x_1 w_{12} + x_2 w_{22} + b_2) \\ f(x_1 w_{13} + x_3 w_{23} + b_3) \end{pmatrix} = f \begin{pmatrix} x_1 w_{11} + x_2 w_{21} + b_1 \\ x_1 w_{12} + x_2 w_{22} + b_2 \\ x_1 w_{13} + x_3 w_{23} + b_3 \end{pmatrix}$$

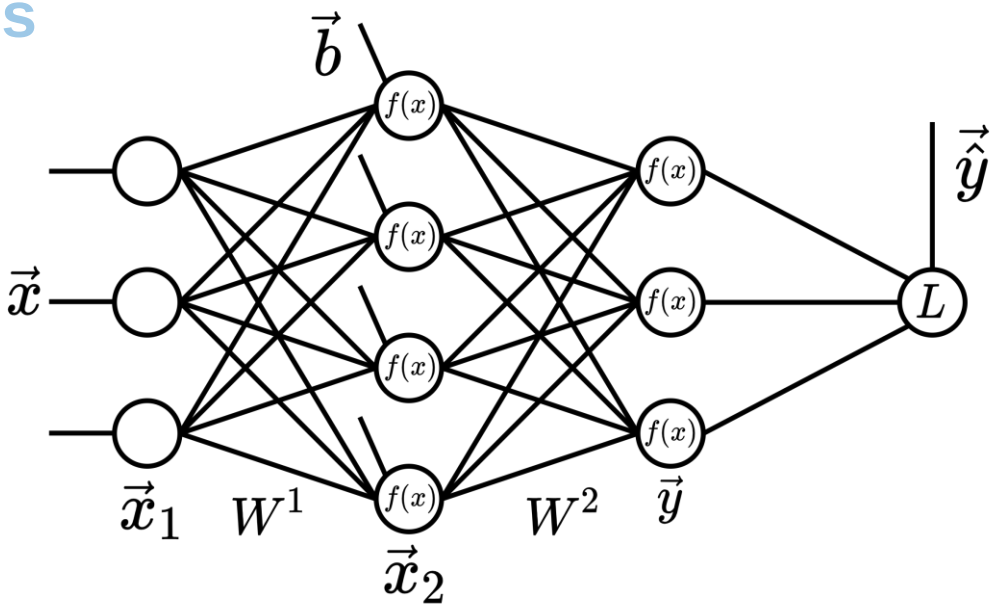
$$= f \left(\begin{pmatrix} x_1 w_{11} + x_2 w_{21} \\ x_1 w_{12} + x_2 w_{22} \\ x_1 w_{13} + x_3 w_{23} \end{pmatrix} + \vec{b} \right) = f \left(\begin{pmatrix} w_{11} + w_{21} \\ w_{12} + w_{22} \\ w_{13} + w_{23} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \vec{b} \right) = f(W^1 \vec{x} + \vec{b})$$

Alternative:

$$\vec{y}^T = \begin{pmatrix} y_1 & y_2 & y_3 \end{pmatrix} = f(W^1 \vec{x} + \vec{b})^T = f(\vec{x}^T W^{1T} + \vec{b}^T)$$

Fully Connected Layer

Dimensions



$$\vec{x}_2 = f\left(\begin{pmatrix} x_1 & x_2 \end{pmatrix} \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{pmatrix} + \vec{b} \right)$$

$$(1 \times 2)(2 \times 4) = (1 \times 4)$$

Since the weight change can be described through matrix operations, a corresponding forward path can be described in such form. For an input vector x, output vector y and a weight matrix W in a fully connected layer the following formula can be derived:

$$\vec{y}^T = \begin{pmatrix} y_1 & y_2 & y_3 \end{pmatrix} = f(W^1\vec{x} + \vec{b})^T = f(\vec{x}^T W^{1T} + \vec{b}^T)$$

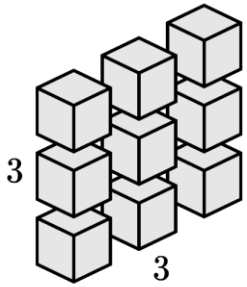
Dimensions of the matrix W are n x m, where n is the number of input neurons and m the number of output neurons:

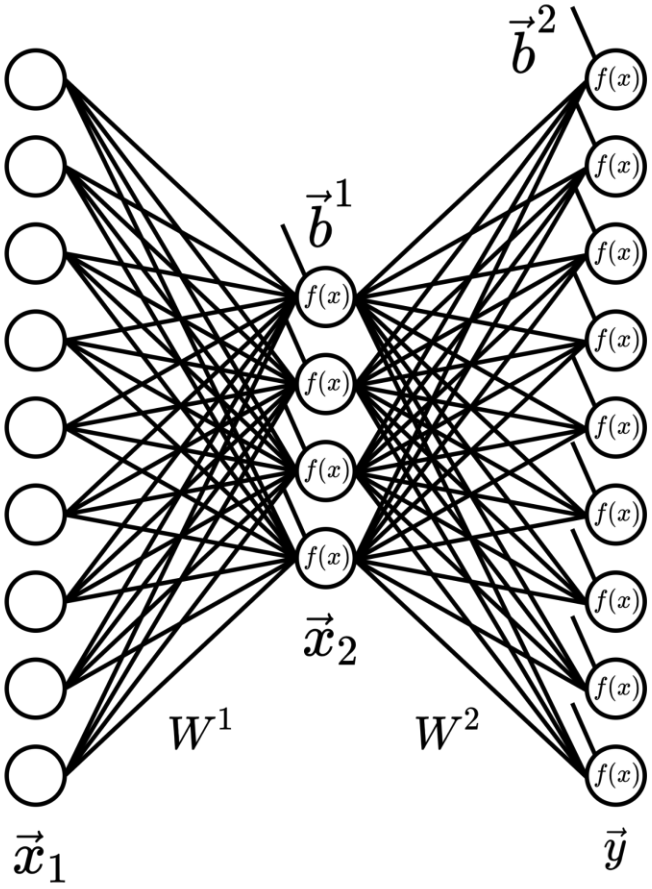
$$\vec{x}_2 = f\left(\begin{pmatrix} x_1 & x_2 \end{pmatrix}\begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{pmatrix} + \vec{b}\right)$$

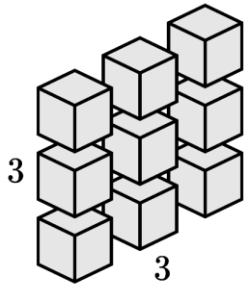
$$(1 \times 2)(2 \times 4) = (1 \times 4)$$

Fully Connected Layer

Image Data





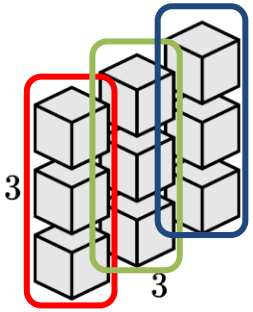


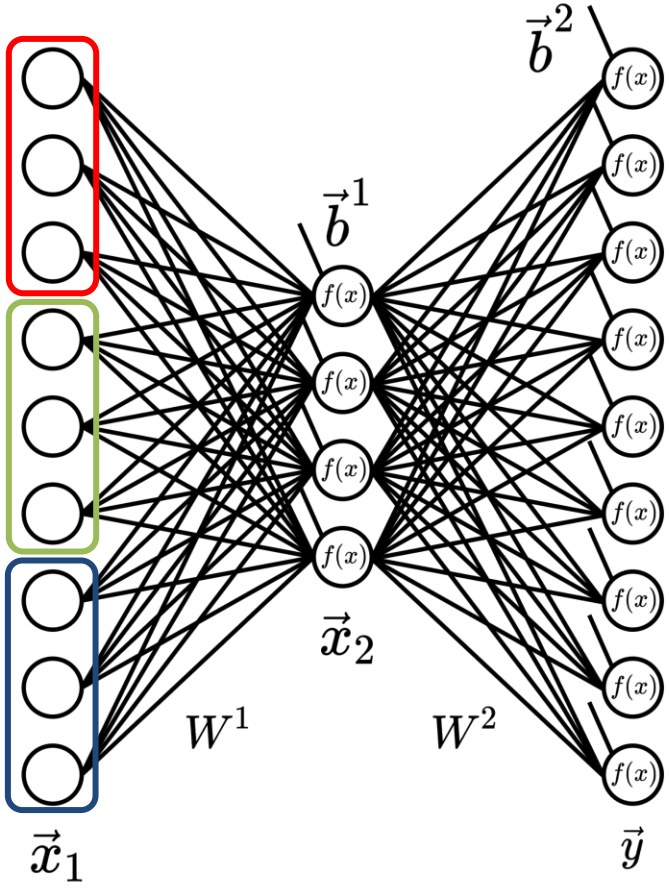
Problem:

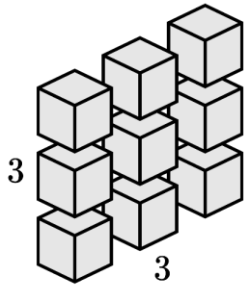
$$(3 \times 3)(9 \times 4)$$

Fully Connected Layer

Image Data







Solution: Flatten (reshape)

$$(3 \times 3) \Rightarrow (1 \times 9)$$

$$(1 \times 9)(9 \times 4)$$

Additional Slides

Since images have two dimensions (width and height), and a fully connected layer requires a 1D vector as an input, these image pixels need to be flattened (stacked) to get a column vector representation of the image which is then suitable for further calculations.

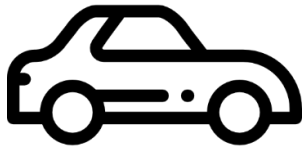
Deep Neural Networks
Prof. Dr. Markus Lienkamp

(Domagoj Majstorović, M. Sc.)

Agenda

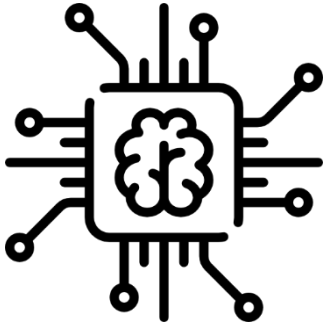
1. Chapter: Backpropagation

- 1.1 Computational Graph
- 1.2 Single Neuron
- 1.3 Neural Chain
- 1.4 Neural Network



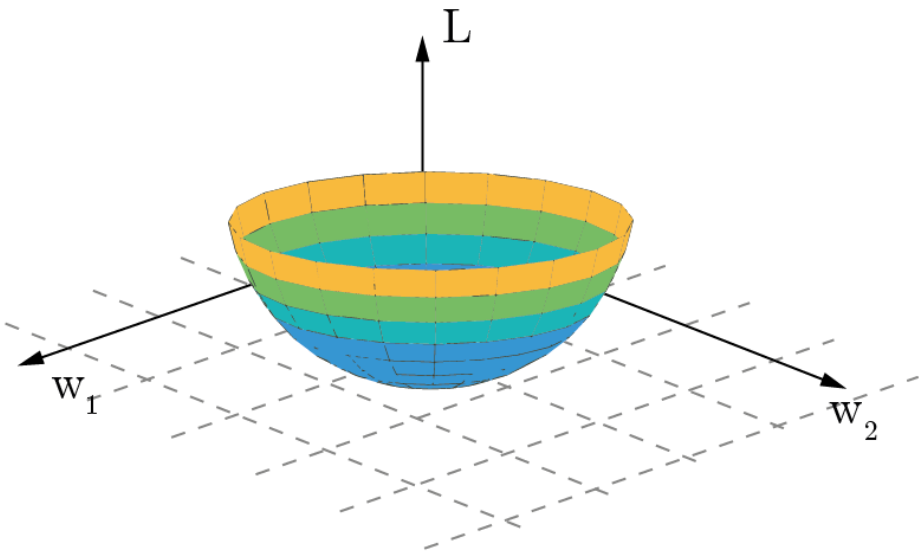
2. Chapter: Neuronal Networks

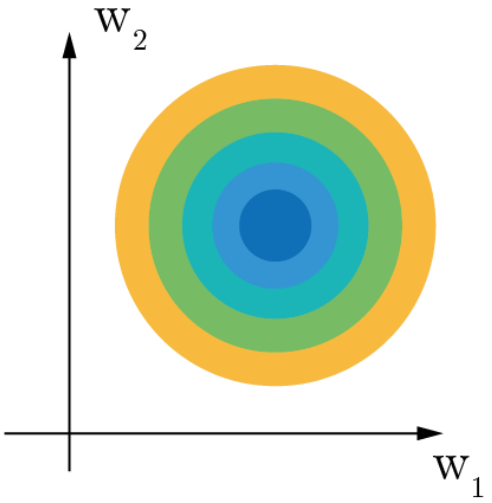
- 2.1 Activation Functions
- 2.2 Fully Connected Layer
- 2.3 Batches and Weight Initialization
- 2.4 Training the Network

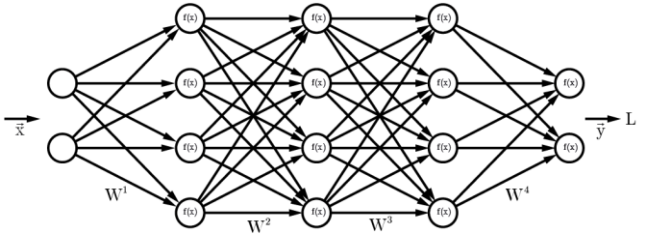


3. Chapter: Summary

Gradient Descent





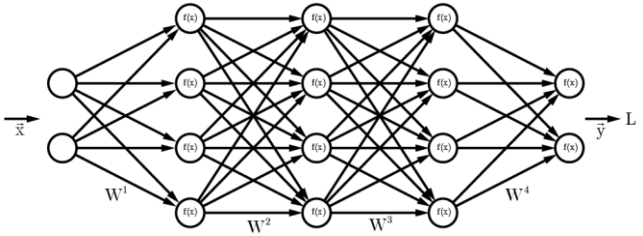


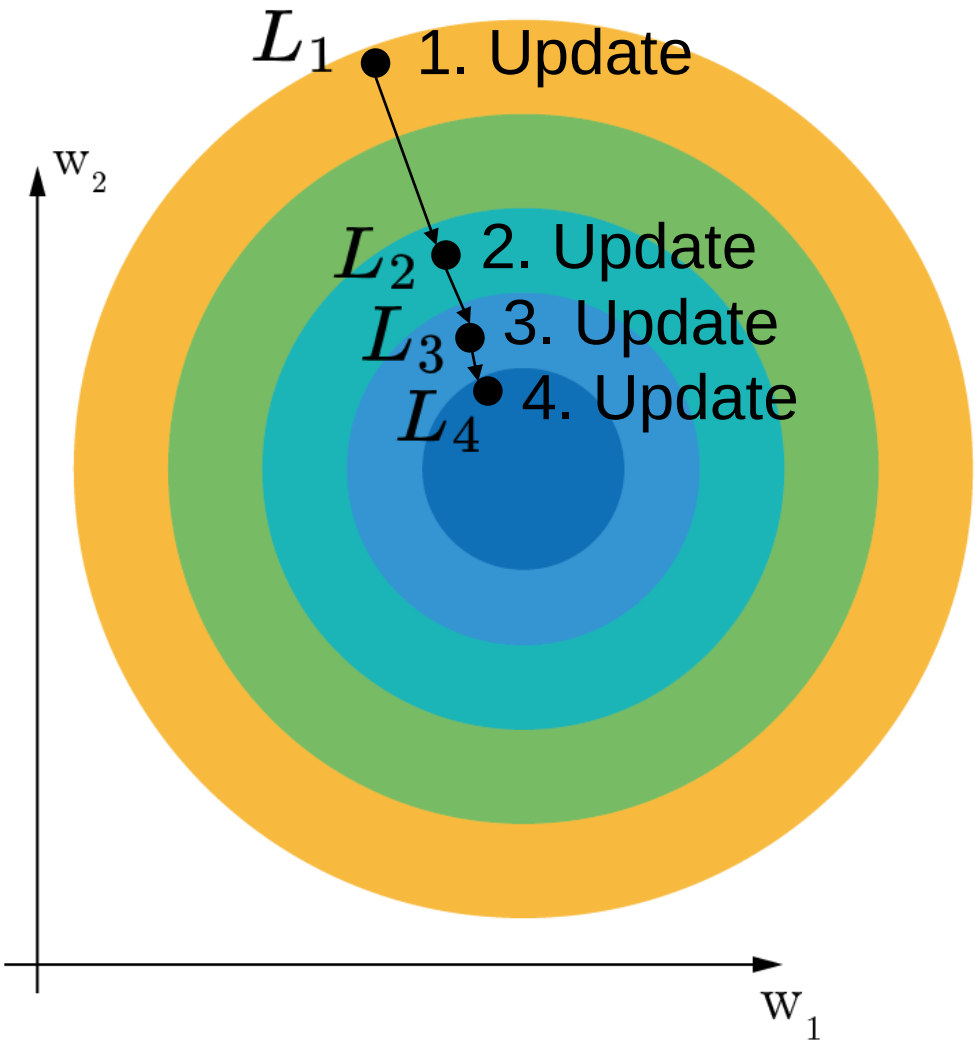
$$L(x) = \frac{1}{n}\sum_{n=1}^{n}(y_i - \hat{y}_i)^2$$

Training:

for n < max_epochs:
- Forward path
- Backward path
- Update weights

Batch Gradient Descent





$$L(x) = \frac{1}{n}\sum_{n=1}^{n}(y_i - \hat{y}_i)^2$$

All the data

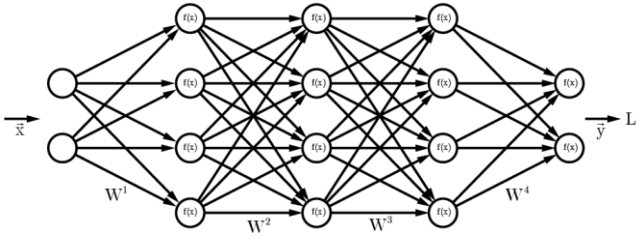
$$-\alpha \nabla L_W$$

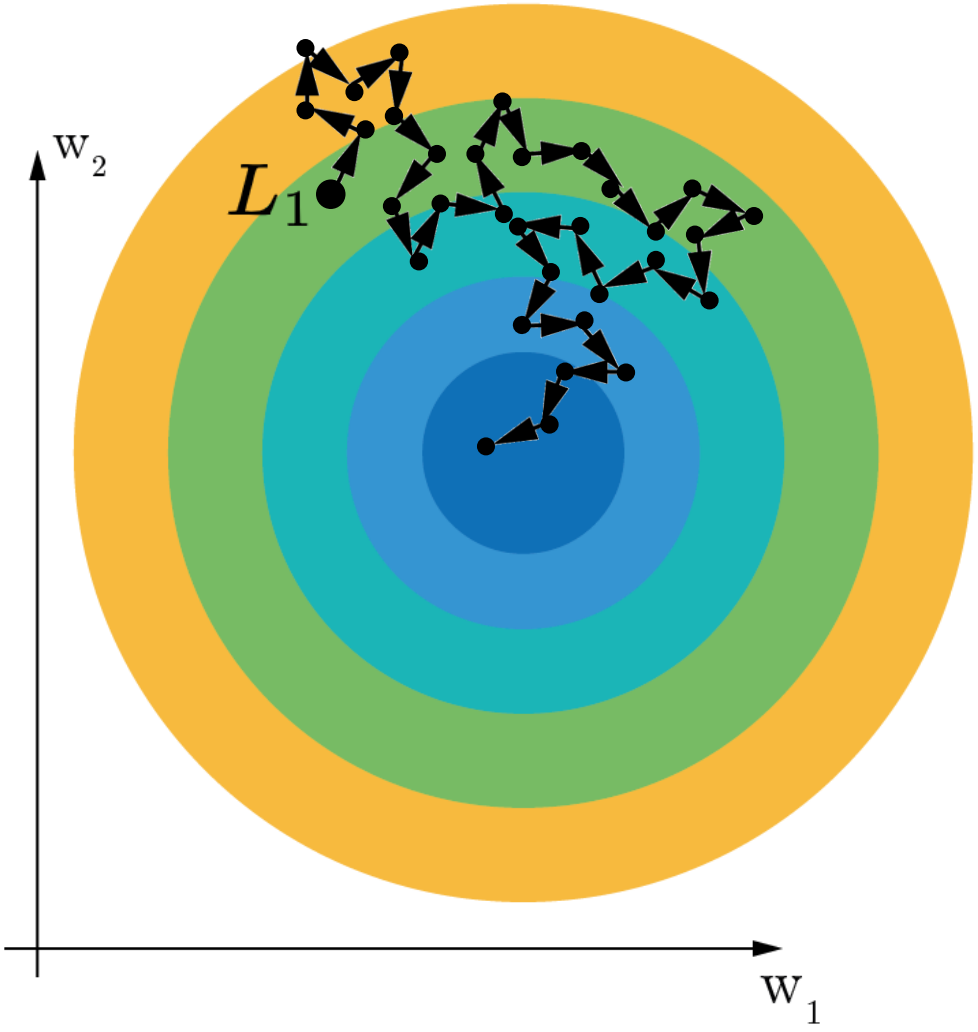
Additional Slides

Backpropagation is a calculation method for optimizing the loss of a neural network with gradient descent. If there is a local minima over the whole data set (dark blue circle), the starting loss L1 decreases towards this minimum with each iteration.

Stochastic Gradient Decent

Alternative to Gradient Decent





Only use one random datapoint at a time:

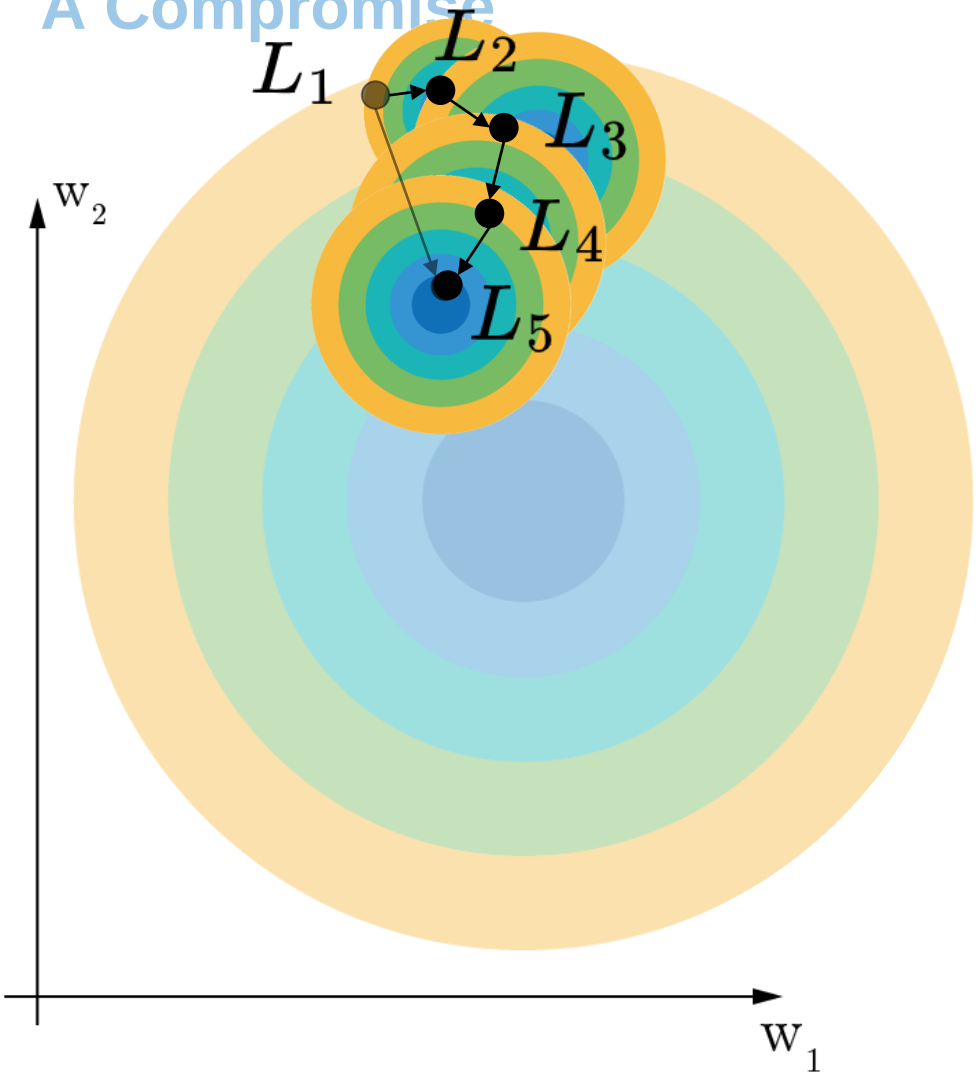
- Reduces compute time per optimization step
- Finding local minimums takes longer

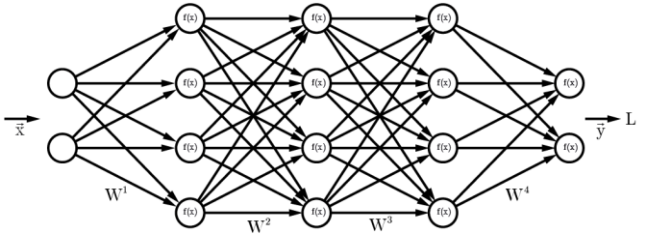
$$L(x) = \frac{1}{n}\sum_{n=1}^{n}(y_i - \hat{y}_i)^2$$

$$n = 1 \Rightarrow L(x) = (y - \hat{y})^2$$

Mini-Batch Gradient Descent

A Compromise





$$L(x) = \frac{1}{n}\sum_{n=1}^{n}(y_i - \hat{y}_i)^2$$

Dataset is now splitted into batches.

Trying to minimize global loss function with local cost functions

It usually not possible to train over the whole data set (since it would require a lot of memory and computing power), which makes small subsets of the dataset better input to calculate the new change of weights. The gradient descent then only applies to those small subsets, which usually results in a suboptimal change of weights which doesn't lead towards to the global loss minimum but the minimum of the subset.

In case of the stochastic gradient decent the subset size is one.

Increasing the subset size results in mini-batch gradient descent.

Gradient descent based on mini-batches stacks the matrix operations that happen inside the network on top of each other, which results in 3D matrix operations. This all is well handled by the deep learning framework.

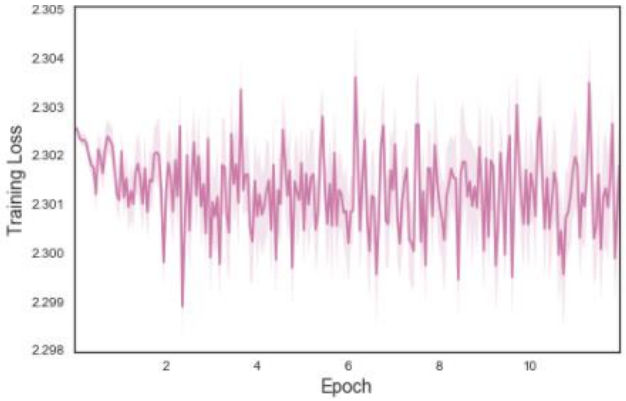
Stochastic vs. Mini-Batch

- **Stochastic** training can miss local minimums because of the randomness of each input and training takes longer in general.
- **Mini-batches** approach finds minimum quickly but may need more computational power.

Weight Initialisation

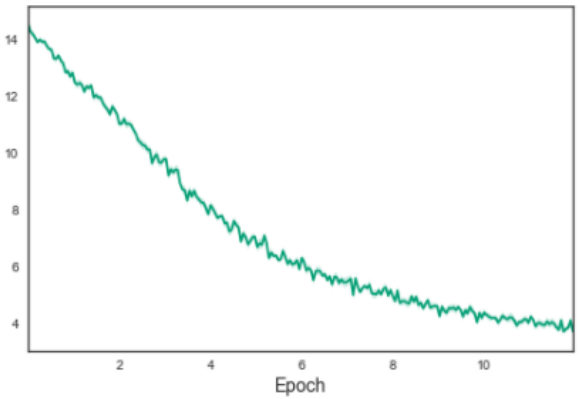
$$W = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{pmatrix}$$

Init as 0



$$W = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

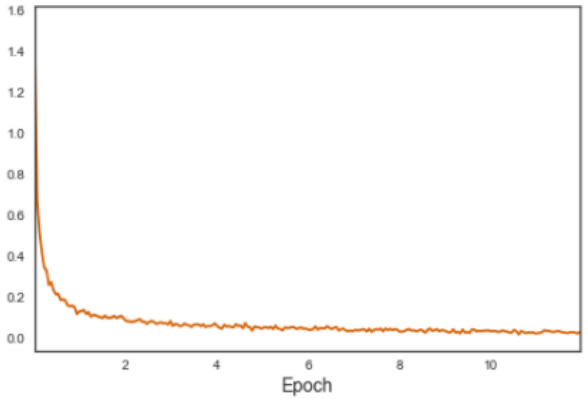
Normal Distribution
(0, $\sigma = 0.4$)



$$W = \begin{pmatrix} 0.01 & -0.50 & 0.14 \\ -0.10 & 0.11 & -0.06 \\ -0.33 & 0.47 & 0.32 \end{pmatrix}$$

Normal Distribution (0, $\sigma \sim \sqrt{\frac{2}{n_i}}$)

n_i: Number of Neurons in previous layer



$$W = \begin{pmatrix} 1.20 & -1.50 & 1.14 \\ -1.20 & 0.97 & -0.6 \\ 0.63 & -0.47 & 0.52 \end{pmatrix}$$

$$n_i = 2; \sigma = 1$$

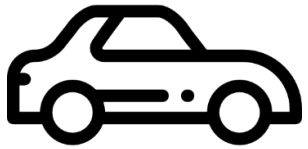
Deep Neural Networks
Prof. Dr. Markus Lienkamp

(Domagoj Majstorović, M. Sc.)

Agenda

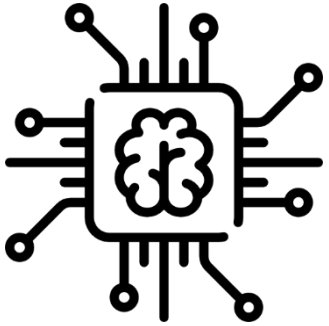
1. Chapter: Backpropagation

- 1.1 Computational Graph
- 1.2 Single Neuron
- 1.3 Neural Chain
- 1.4 Neural Network



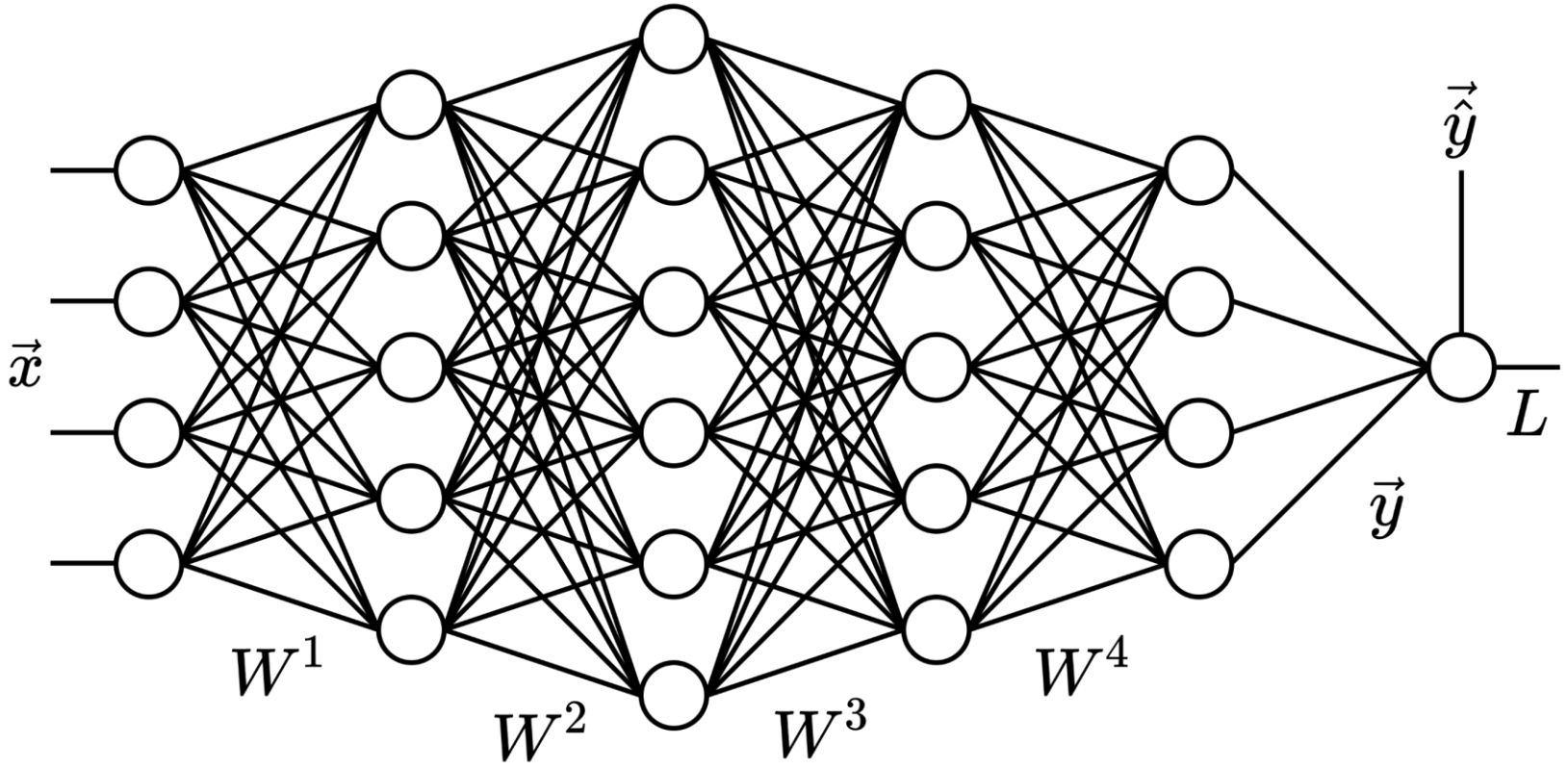
2. Chapter: Neuronal Networks

- 2.1 Activation Functions
- 2.2 Fully Connected Layer
- 2.3 Batches and Weight Initialization
- 2.4 Training the Network

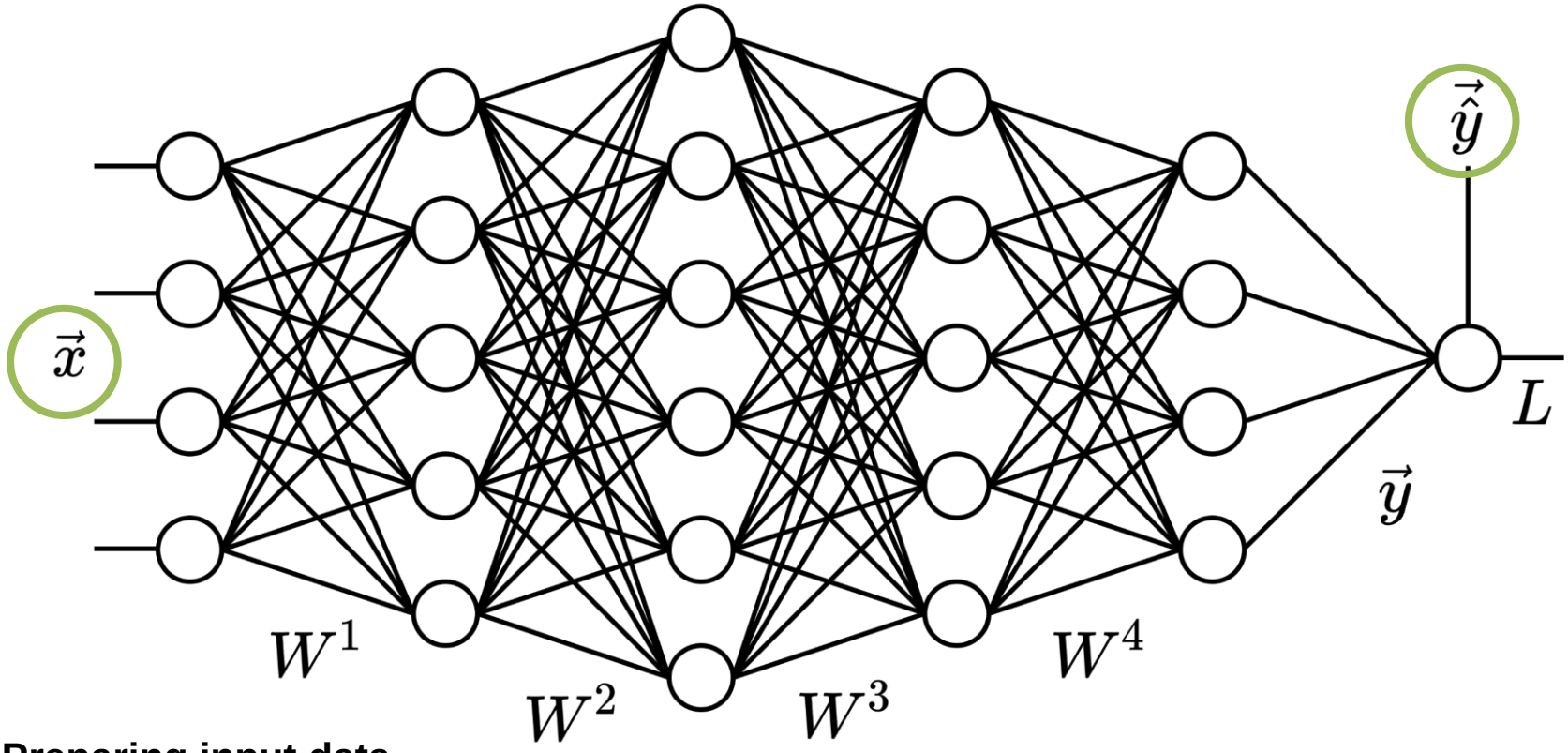


3. Chapter: Summary

Training a Neural Network

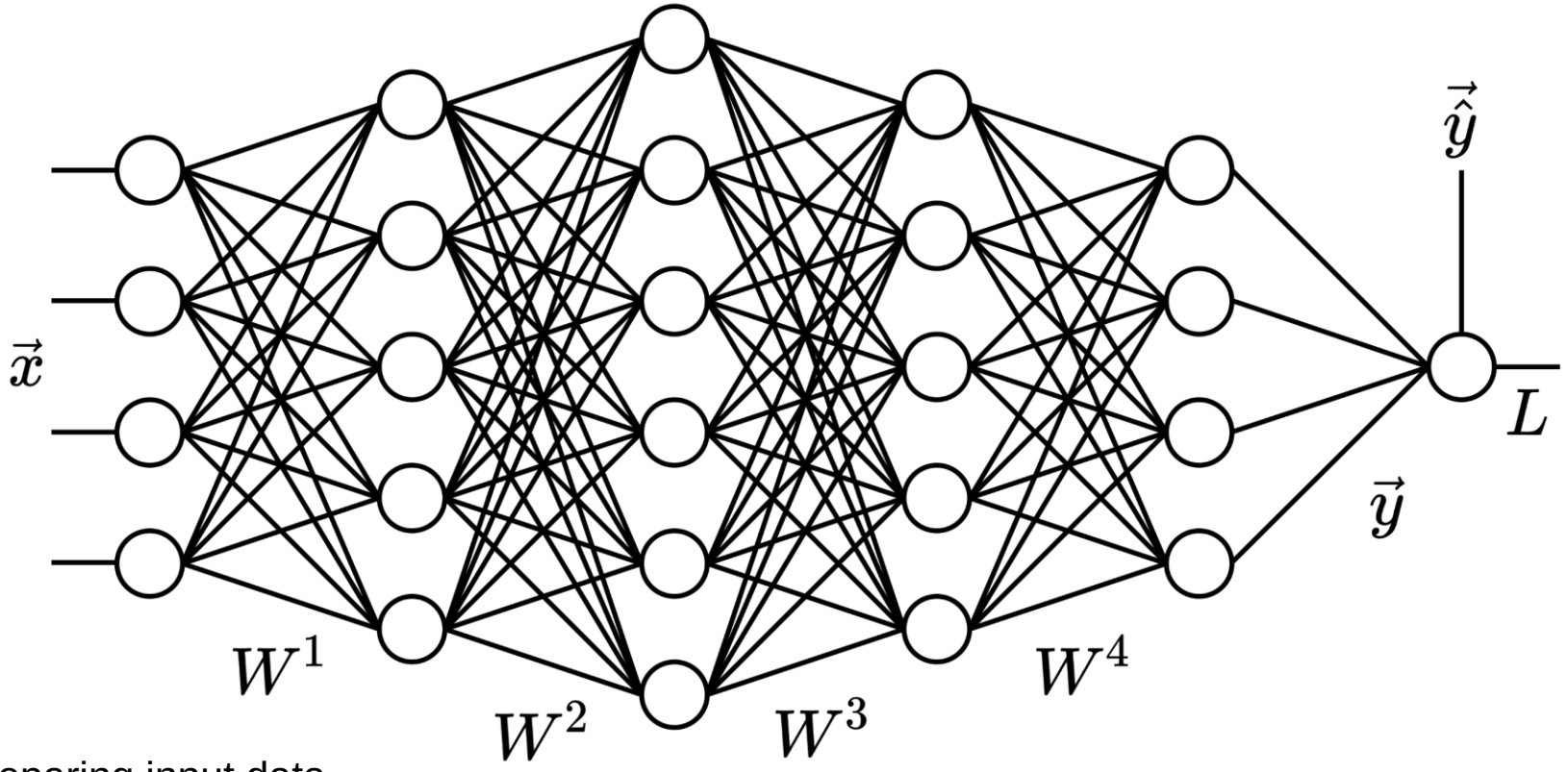


Training a Neural Network



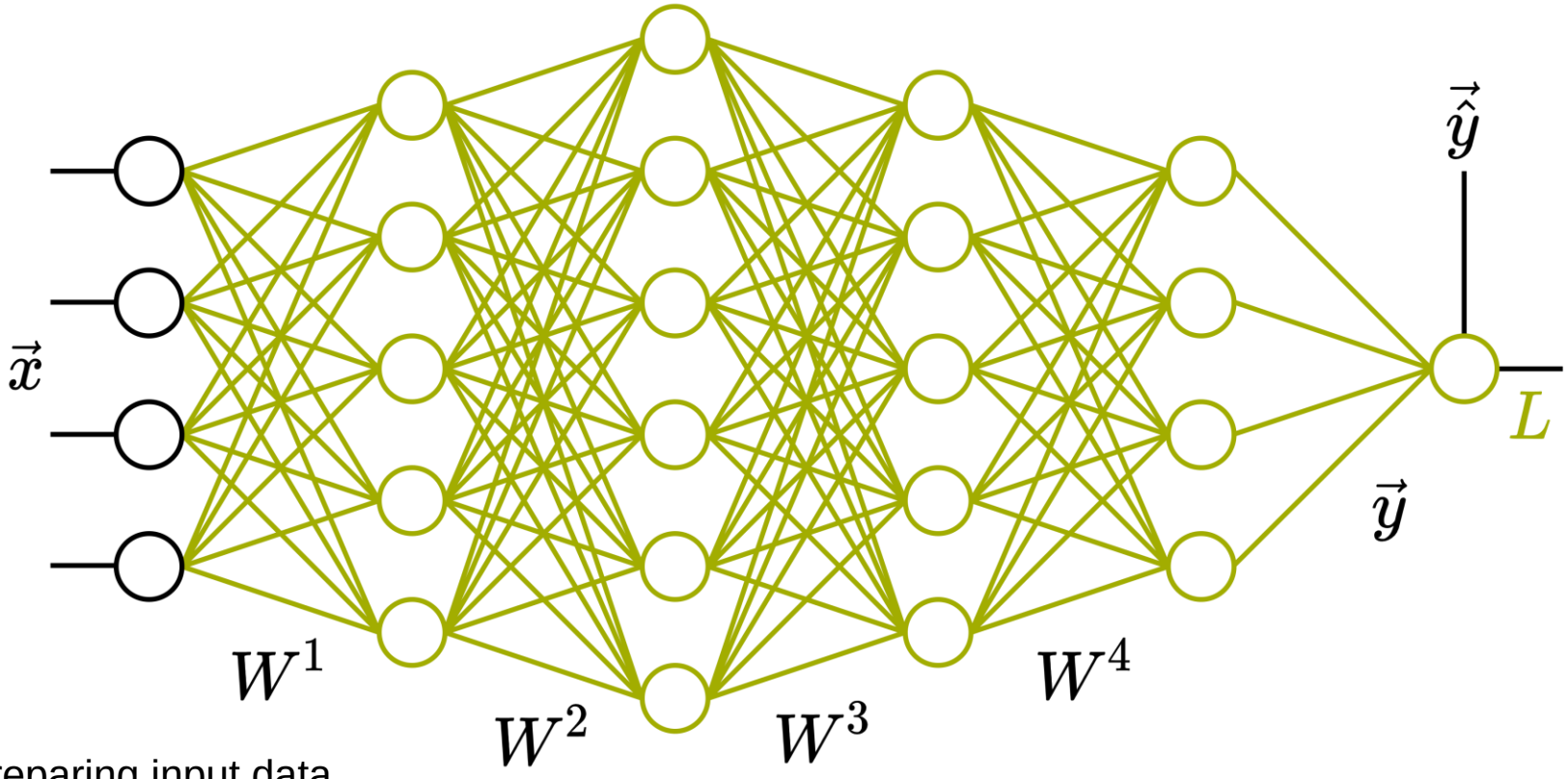
1. Preparing input data

Training a Neural Network



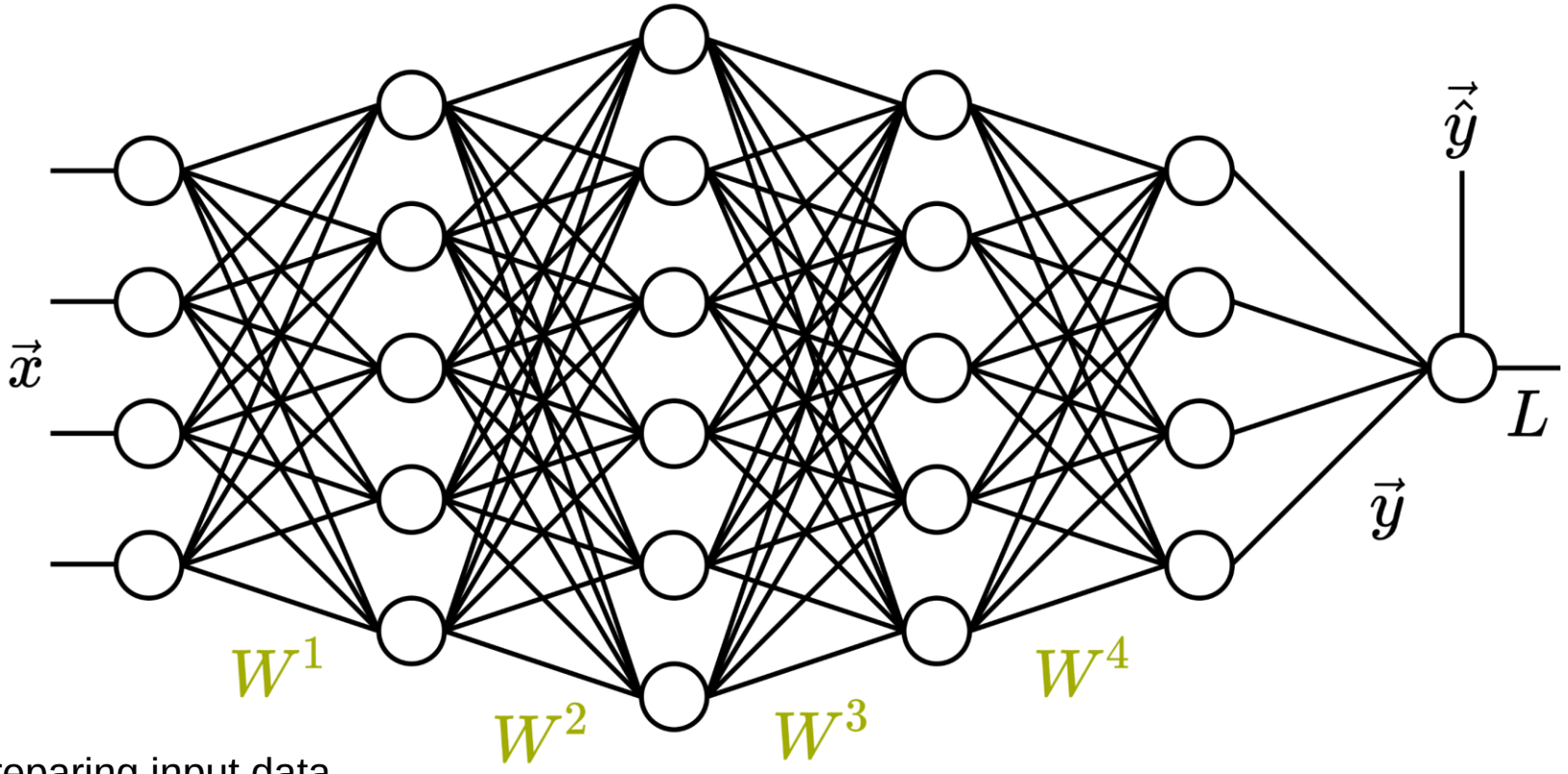
1. Preparing input data
2. **Prepare the data-pipeline to get the data into the network**

Training a Neural Network



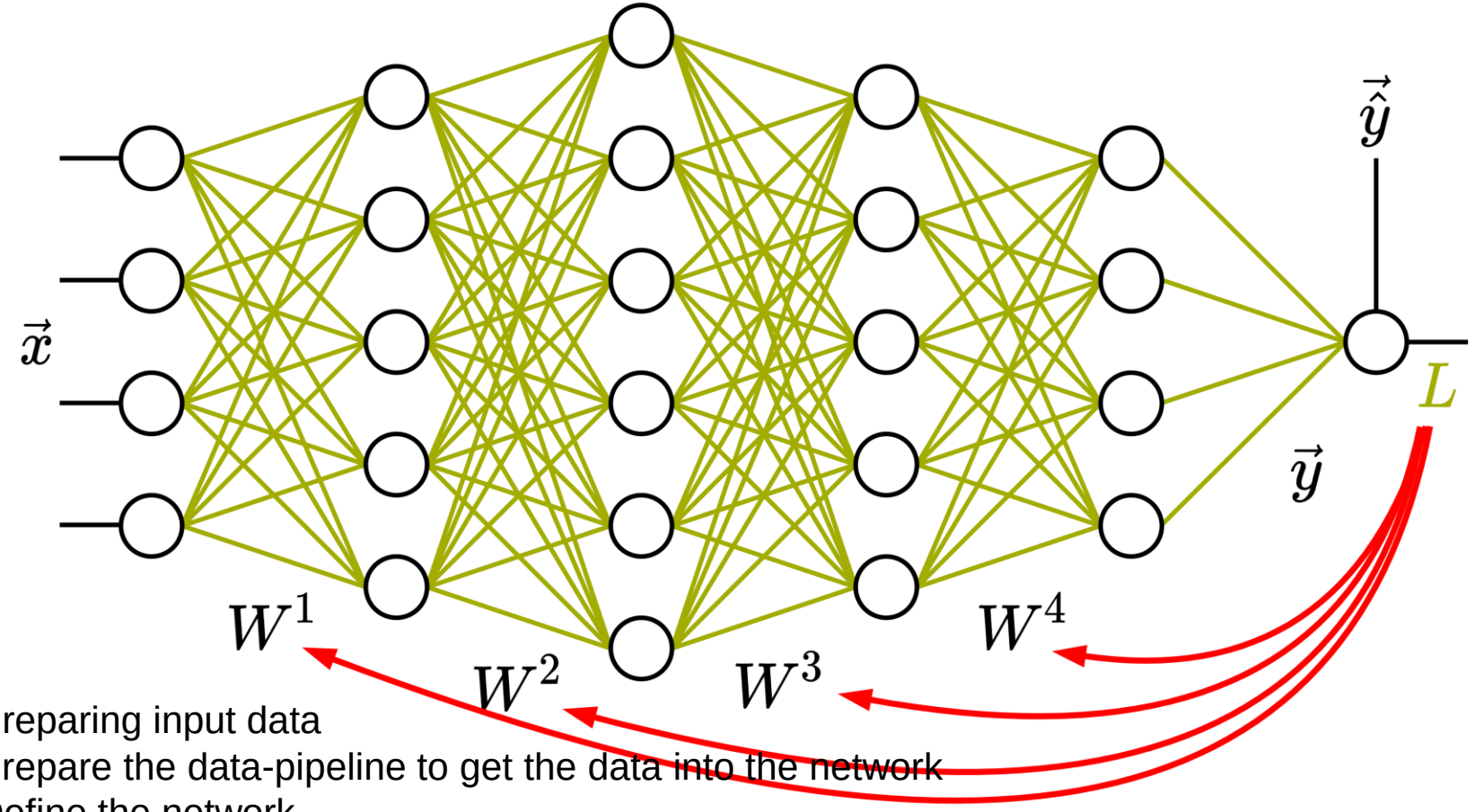
1. Preparing input data
2. Prepare the data-pipeline to get the data into the network
3. **Define the network**

Training a Neural Network



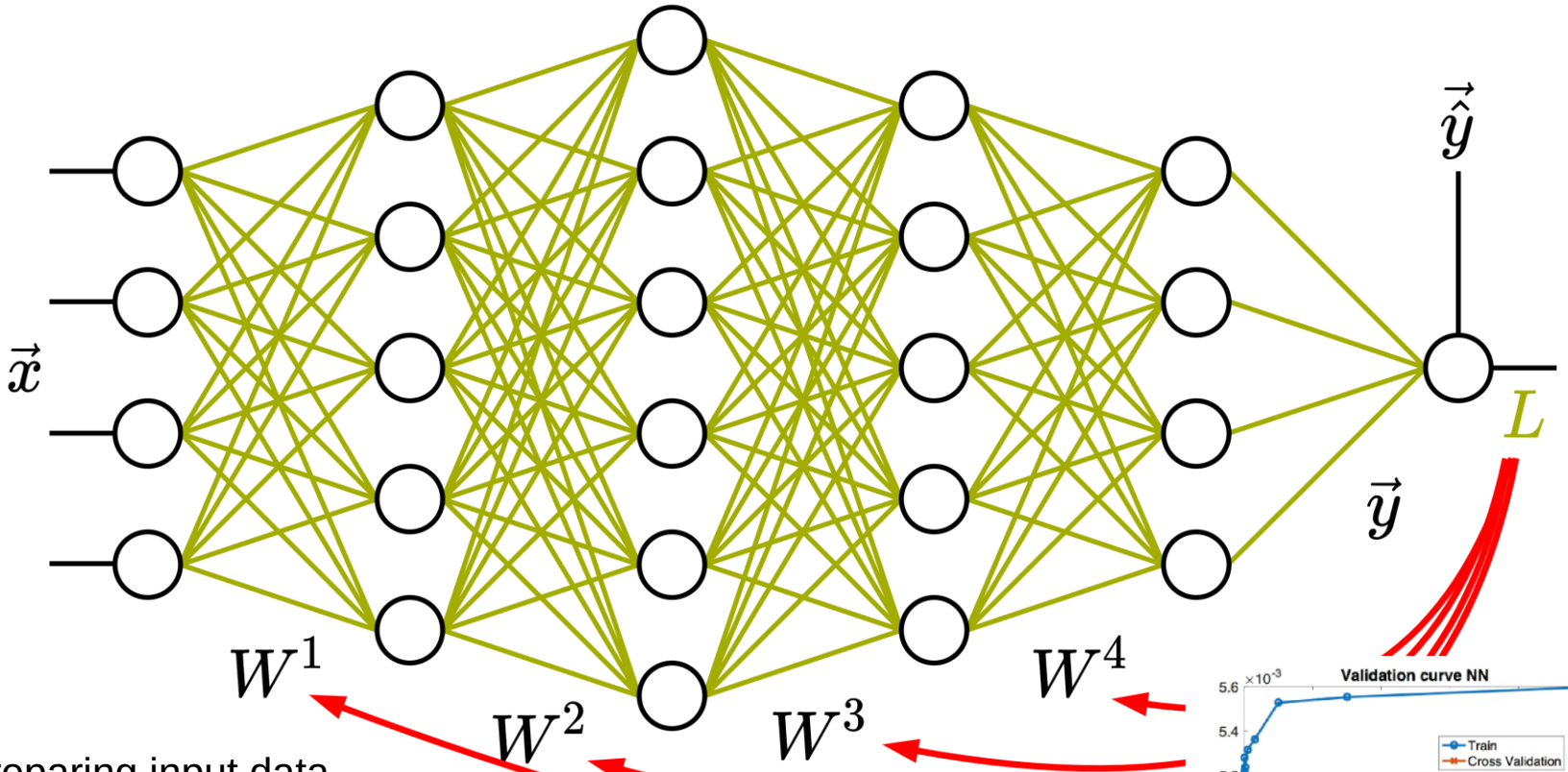
1. Preparing input data
2. Prepare the data-pipeline to get the data into the network
3. Define the network
4. **Initialize all variables**

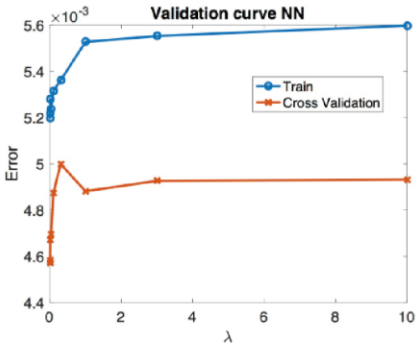
Training a Neural Network



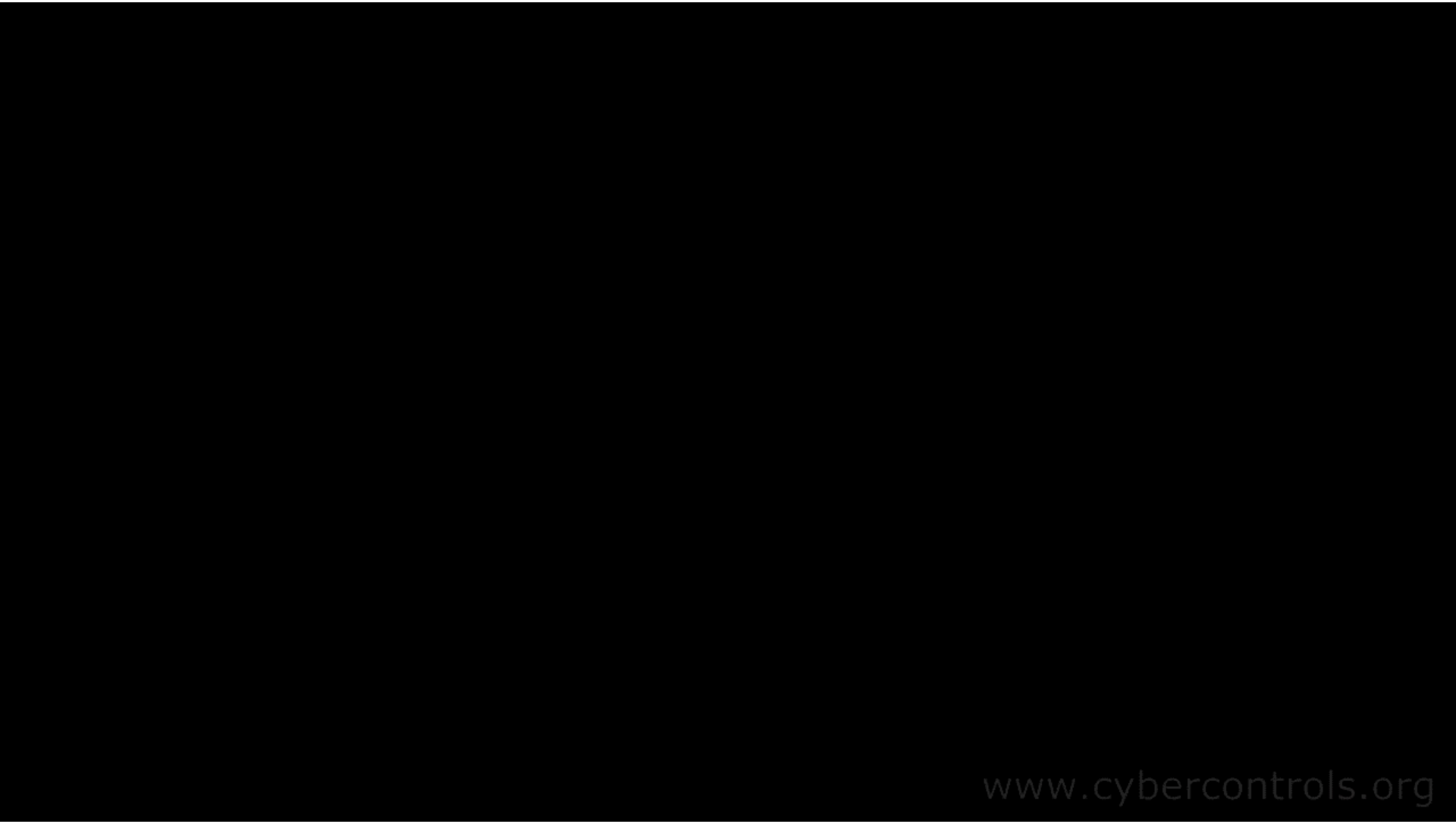
1. Preparing input data
2. Prepare the data-pipeline to get the data into the network
3. Define the network
4. Initialize all variables
5. **Train the network**

Training a Neural Network





1. Preparing input data
2. Prepare the data-pipeline to get the data into the network
3. Define the network
4. Initialize all variables
5. Train the network
6. **Supervise the training**


www.cybercontrols.org

Summary

What we learned today:

How to use computational graphs to calculate local gradients in any mathematical function

How to backpropagate local gradients to calculate weight updates

Most common activation functions and their use cases

Benefit of using batches for the training of neural networks

How to initialize weights correctly

What are the steps to train a neural network

End