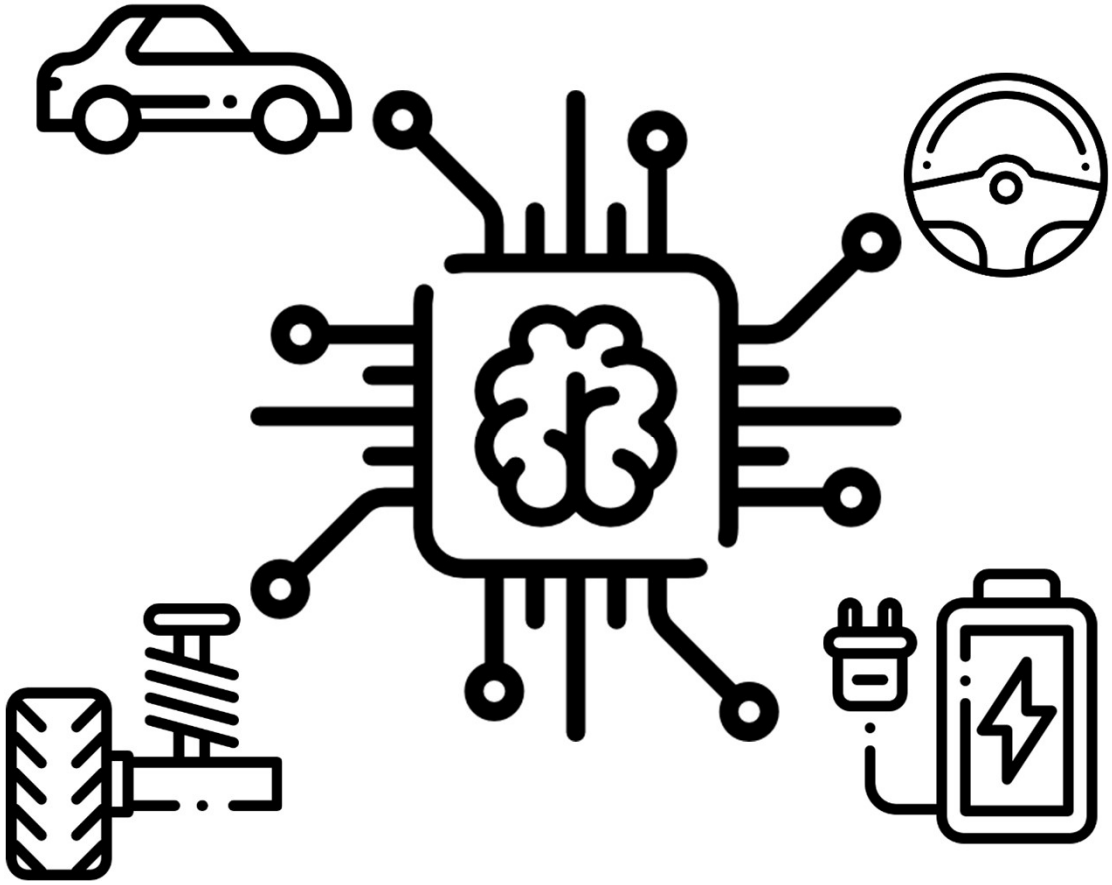
Lehrstuhl für Fahrzeugtechnik
Fakultät für Maschinenwesen
Technische Universität München

Artificial Intelligence in Automotive Technology

Johannes Betz / Prof. Dr.-Ing. Markus Lienkamp / Prof. Dr.-Ing. Boris Lohmann

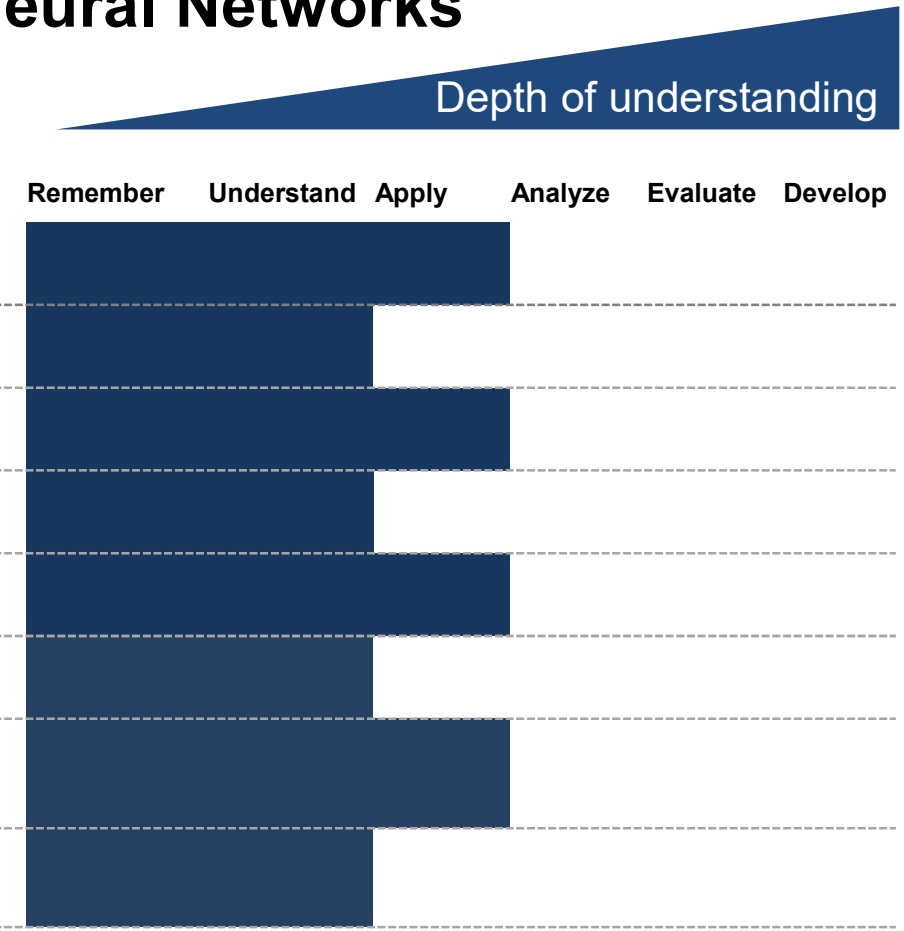


Lecture Overview

Lecture 16:15 – 17:45	Practice 17:45 – 18:30
1 Introduction: Artificial Intelligence 17.10.2019 – Johannes Betz	**Practice 1** 17.10.2019 – Johannes Betz
2 Perception 24.10.2019 – Johannes Betz	**Practice 2** 24.10.2019 – Johannes Betz
3 Supervised Learning: Regression 31.10.2019 – Alexander Wischnewski	**Practice 3** 31.10.2019 – Alexander Wischnewski
4 Supervised Learning: Classification 7.11.2019 – Jan Cedric Mertens	**Practice 4** 7.11.2019 – Jan Cedric Mertens
5 Unsupervised Learning: Clustering 14.11.2019 – Jan Cedric Mertens	**Practice 5** 14.11.2019 – Jan Cedric Mertens
6 Pathfinding: From British Museum to A* 21.11.2019 – Lennart Adenaw	**Practice 6** 21.11.2019 – Lennart Adenaw
7 Introduction: Artificial Neural Networks 28.11.2019 – Lennart Adenaw	**Practice 7** 28.11.2019 – Lennart Adenaw
8 Deep Neural Networks 5.12.2019 – Jean-Michael Georg	**Practice 8** 5.12.2019 – Jean-Michael Georg
9 Convolutional Neural Networks 12.12.2019 – Jean-Michael Georg	**Practice 9** 12.12.2019 – Jean-Michael Georg
10 Recurrent Neural Networks 19.12.2019 – Christian Dengler	**Practice 10** 19.12.2019 – Christian Dengler
11 Reinforcement Learning 09.01.2020 – Christian Dengler	**Practice 11** 09.01.2020 – Christian Dengler
12 AI-Development 16.01.2020 – Johannes Betz	**Practice 12** 16.01.2020 – Johannes Betz
13 Guest Lecture: VW Data:Lab 23.01.2020 –	

Objectives for Lecture 8: Neural Networks

After the lecture you are able to…

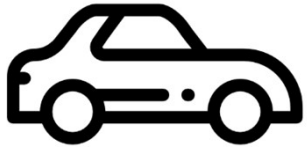


Deep Neural Networks

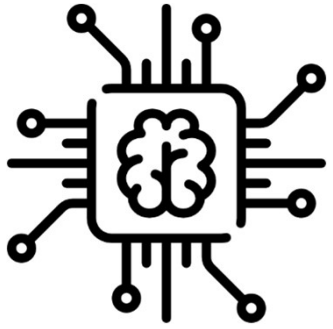
Johannes Betz / Prof. Dr. Markus Lienkamp / Prof. Dr. Boris Lohmann
(Jean-Michael Georg, M. Sc.)

Agenda

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

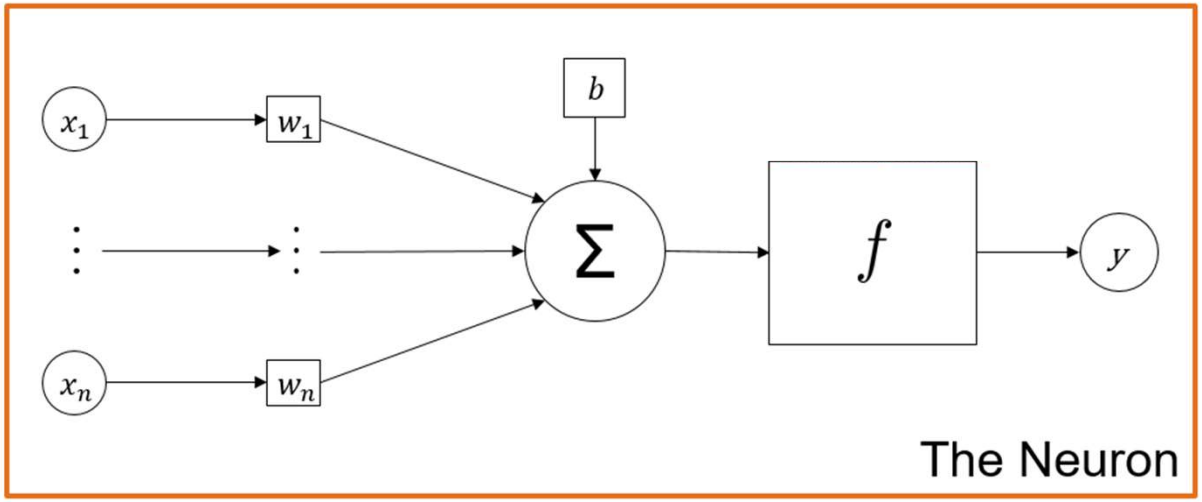


2. Chapter: Neuronal Networks
    - 2.1 Activation Functions
    - 2.2 Fully Connected Layer
    - 2.3 Batches
    - 2.3 Weight Inititalisation



3. Chapter: Overview

Repetition



$L = w^2$

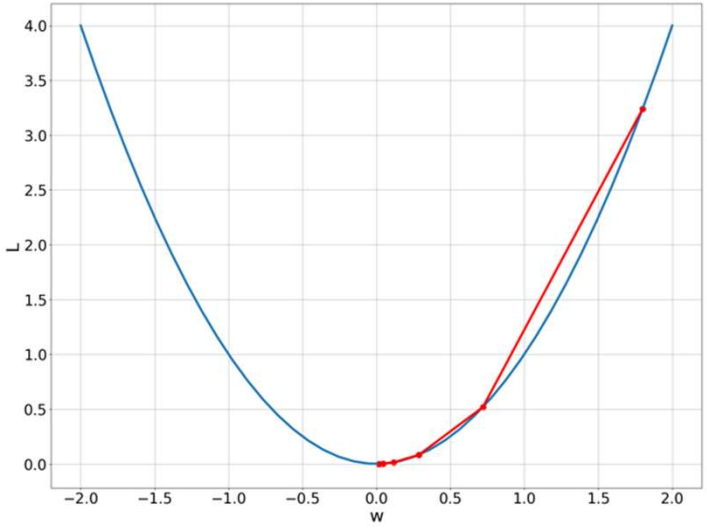
$\nabla L = \frac{dL}{dw} = 2 \cdot w$

$\alpha = 0.3$

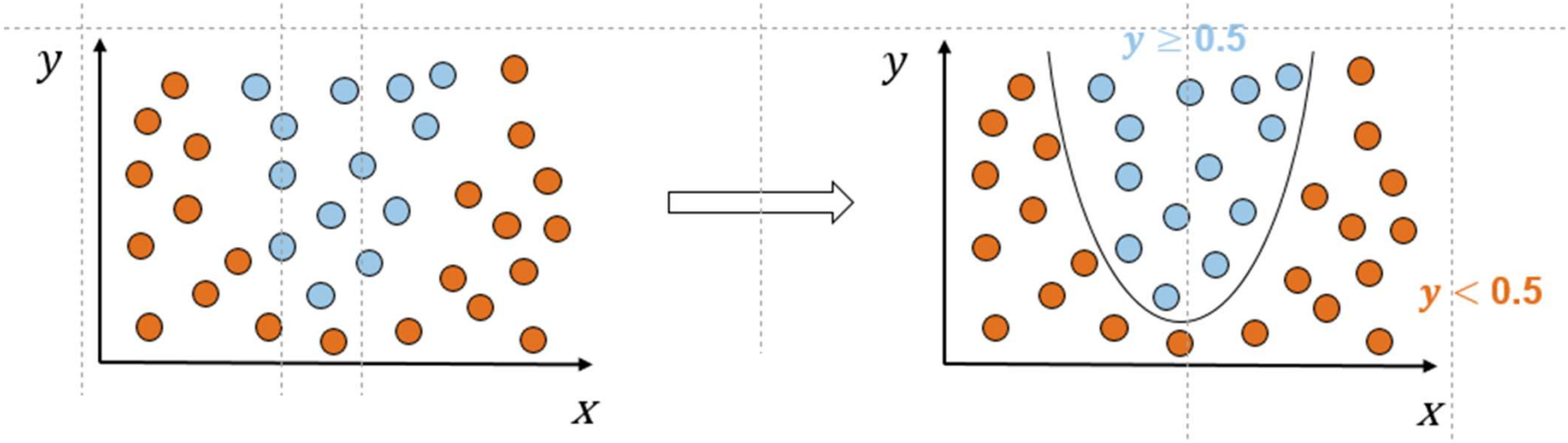
$w_0 = 1.8$

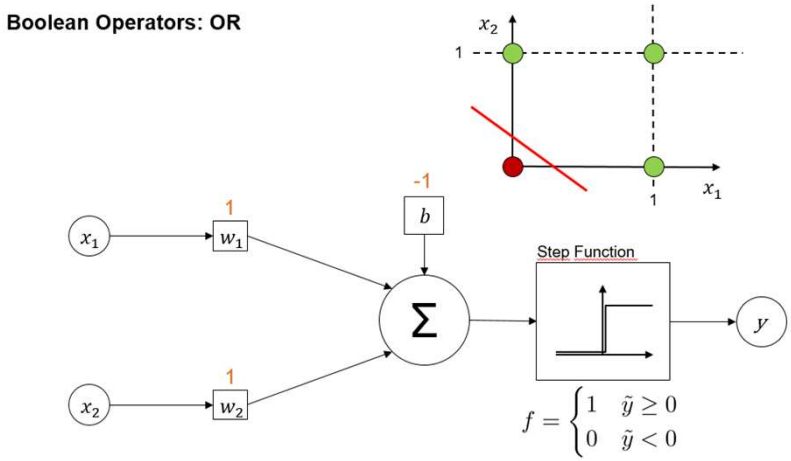
$\epsilon = 0.001$

$N = 5$



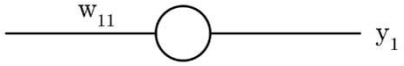
Repetition


y
X
y
X
$y \geq 0.5$
$y < 0.5$

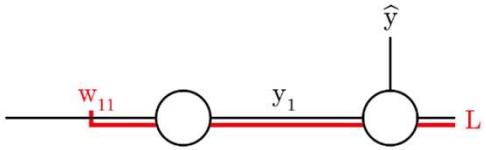

Boolean Operators: OR
x_2
1
x_1
1
-1
b
x_1
1
w_1
x_2
1
w_2
Σ
Step Function
y
$f = \begin{cases} 1 & \tilde{y} \geq 0 \\ 0 & \tilde{y} < 0 \end{cases}$

Last Week:

Single Neuron



1D Gradient Descent



$L = w^2$

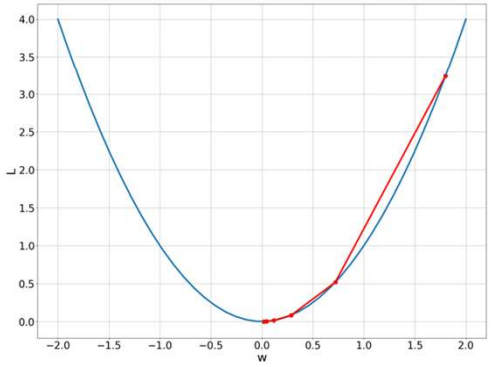
$\nabla L = \frac{dL}{dw} = 2 \cdot w$

$\alpha = 0.3$

$w_0 = 1.8$

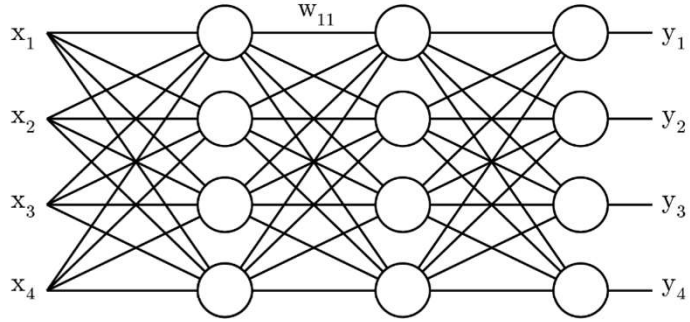
$\epsilon = 0.001$

$N = 5$

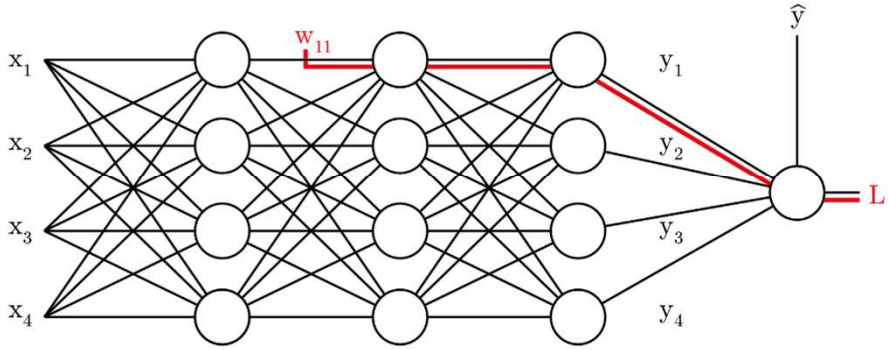


Today:

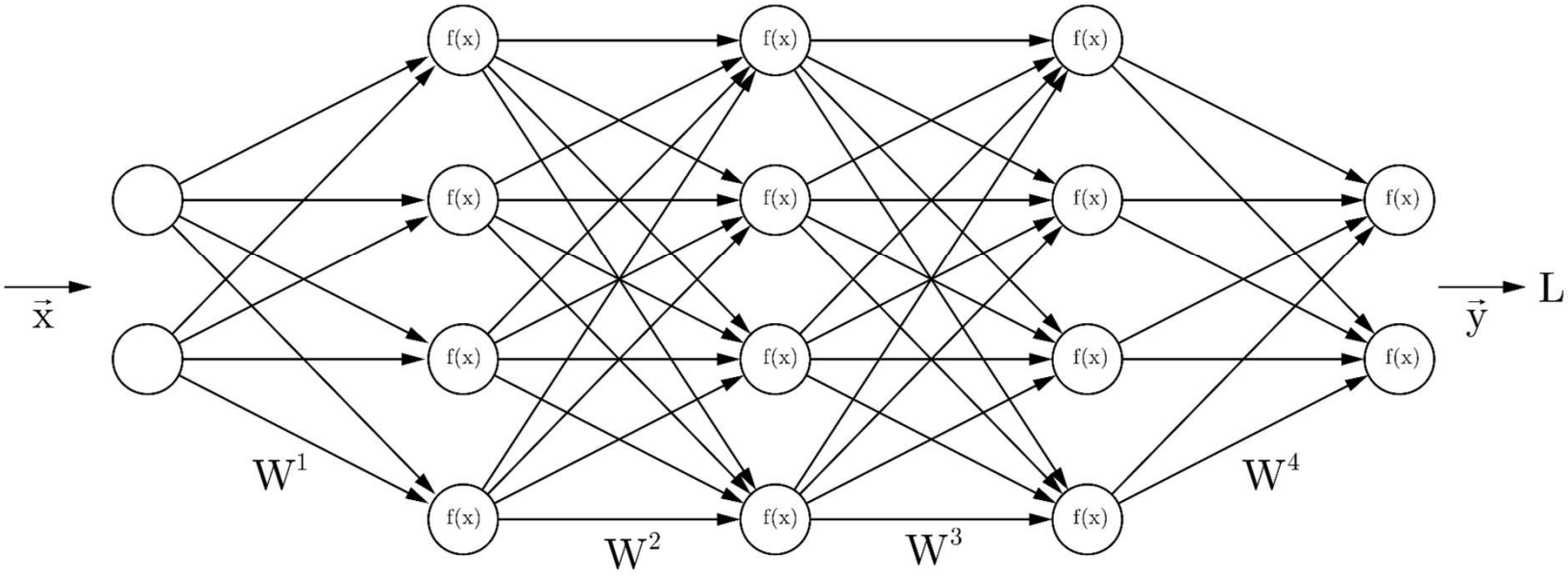
Neural Network



Backpropagtion



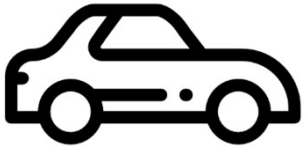
Why?

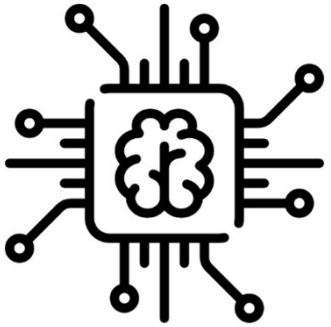


Deep Neural Networks

Johannes Betz / Prof. Dr. Markus Lienkamp / Prof. Dr. Boris Lohmann
(Jean-Michael Georg, M. Sc.)

Agenda



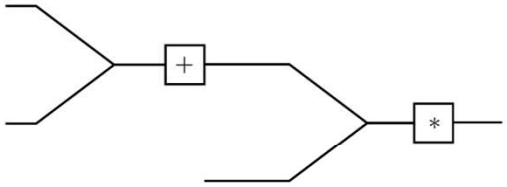


Computational Graph

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

Options:
1. Analytical
2. Difference quotient
3. Computational graph

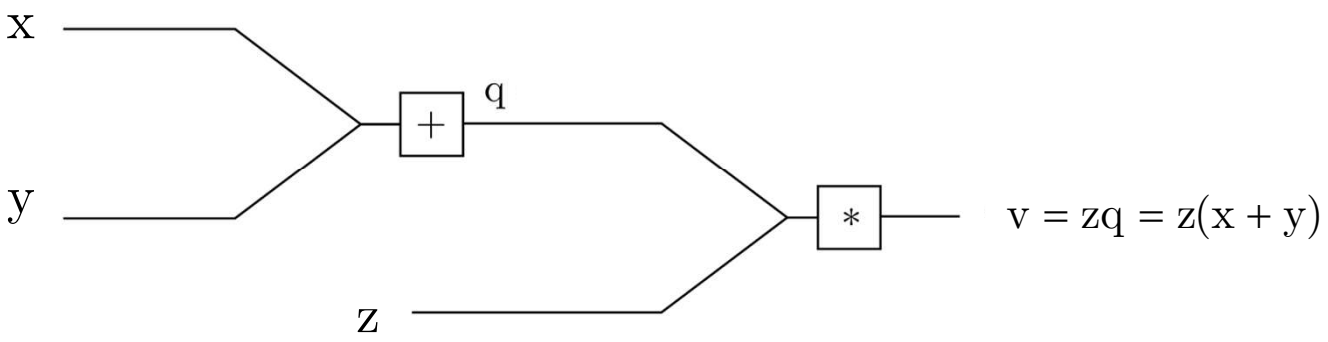
$$\lim_{h \to 0} \frac{L(w+h) - L(w)}{h}$$



$\Rightarrow$ Chain of derivitives $\Rightarrow$ Backpropagation

Computational Graph

Simple example

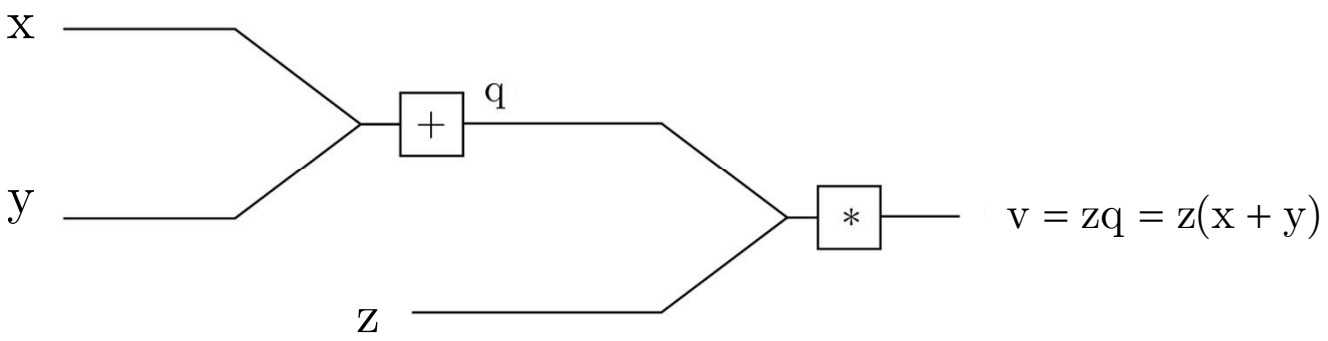


Lets calculate some derivatives!

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

Computational Graph

Simple example

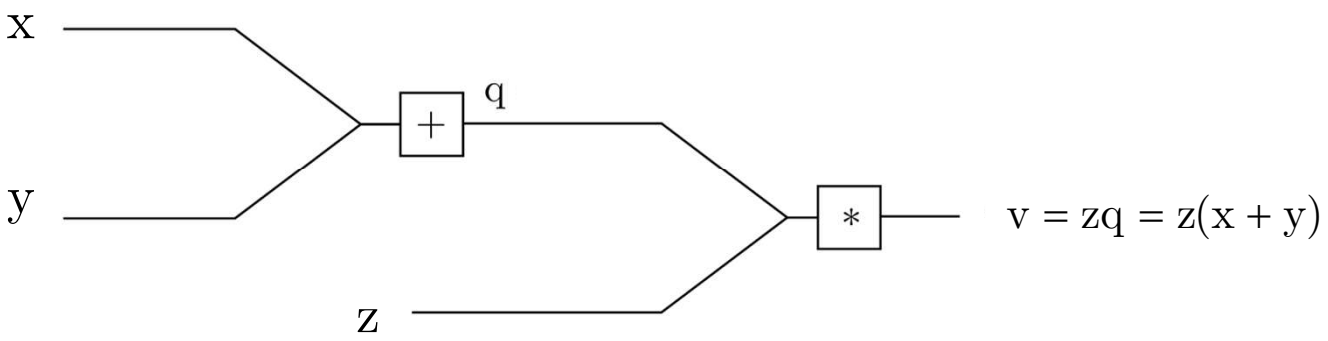


Lets calculate some derivatives!

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

Computational Graph

Simple example

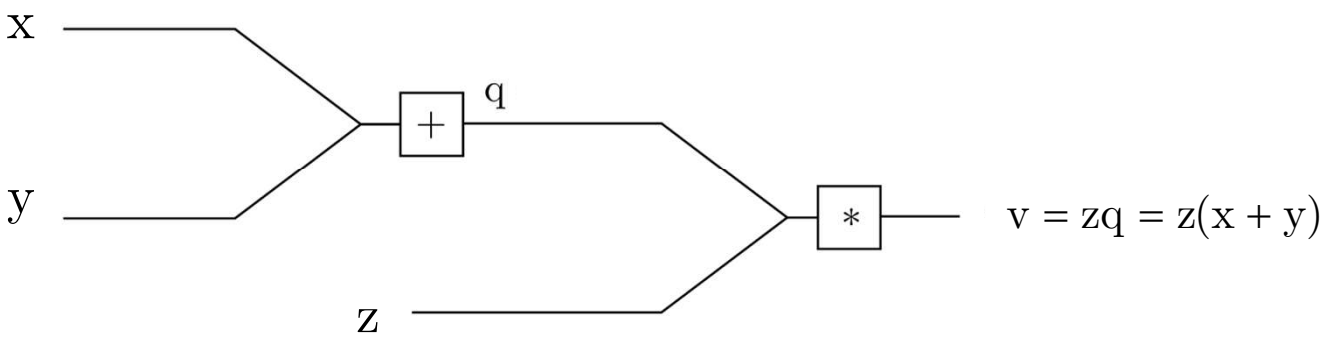


Lets calculate some derivatives!

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

Computational Graph

Simple example

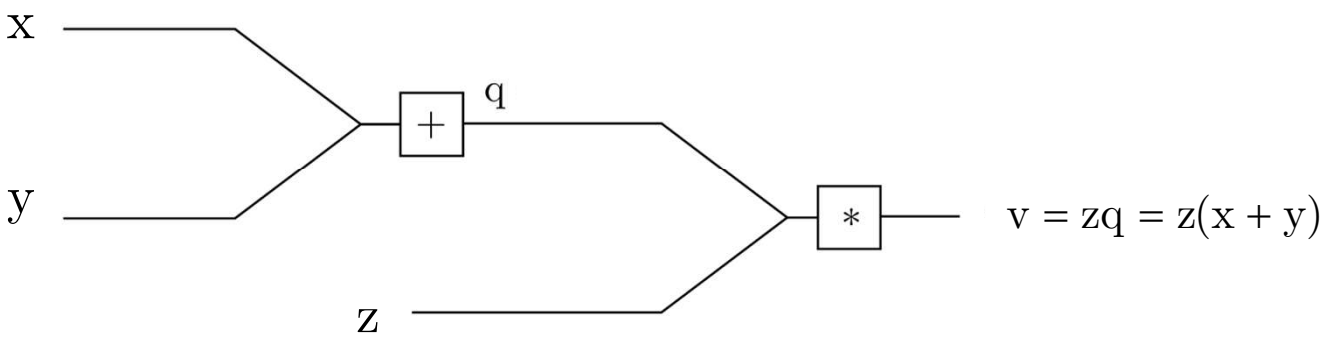


Lets calculate some derivatives!

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

Computational Graph

Simple example



Lets calculate some derivatives!

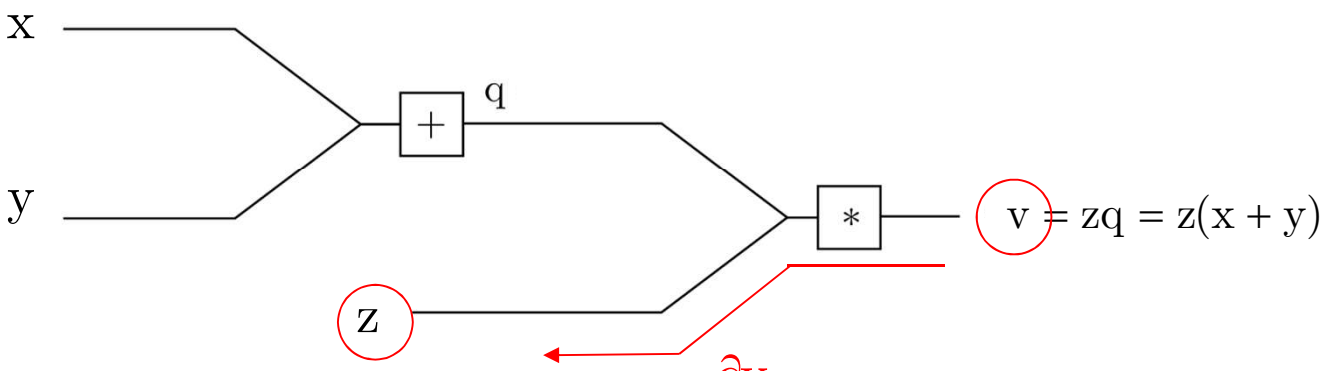
$$\frac{\partial v}{\partial z} = q \qquad \frac{\partial v}{\partial q} = z \qquad \frac{\partial v}{\partial x} = \frac{\partial v}{\partial q}\frac{\partial q}{\partial x} = z \cdot 1 \qquad \frac{\partial v}{\partial y} = \frac{\partial v}{\partial q}\frac{\partial q}{\partial y} = z \cdot 1$$

Additional Slides

To understand the backpropagation algorithm you need to understand how local gradients function. Therefore we first have a look at an easy example, the computational graph. Here every box represents a mathematic operation, with the given input and calculated output data.

Computational Graph

Simple example



Definition gradient along graph:

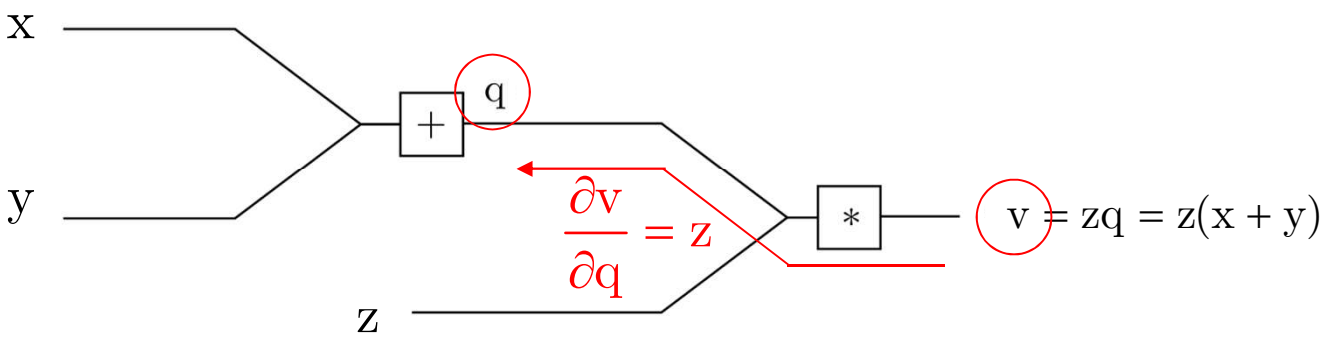
$$\frac{\partial v}{\partial z} = q$$

Additional Slides

The gradient between v and z can be calculated as shown. The gradients “flow” backwards a long the graph. Therefore the upstream gradient regarding z is dv/dz

Computational Graph

Simple example

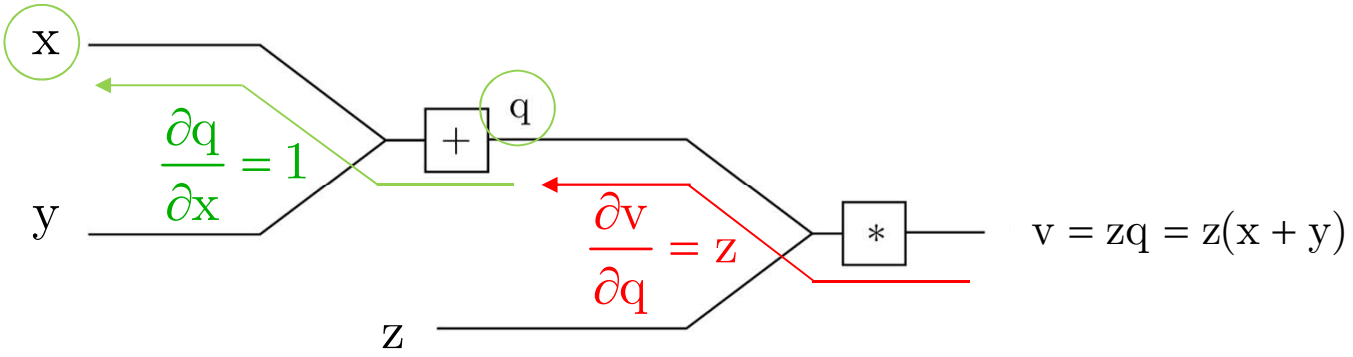


Additional Slides

The gradient between v and z can be calculated as shown. The gradients “flow” backwards a long the graph. Therefore the upstream gradient of z is dv/dz

Computational Graph

Simple example

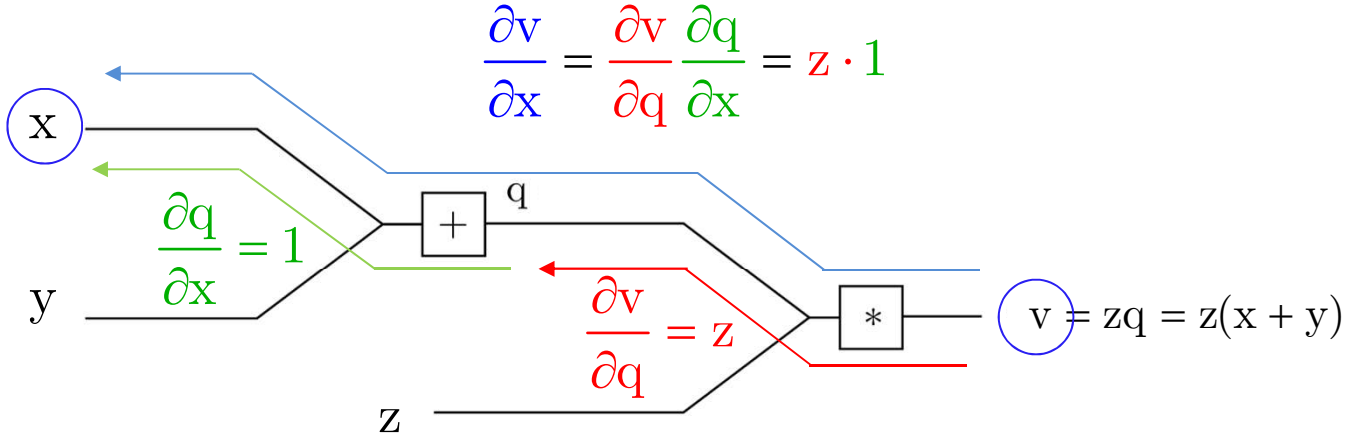


Additional Slides

The gradient between q and b can calculated similary as dv/dq

Computational Graph

Simple example

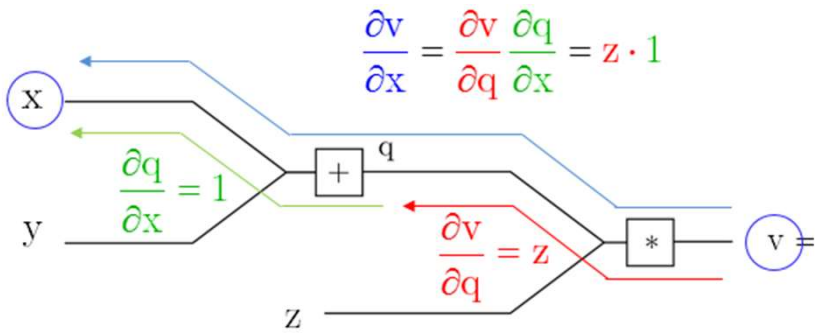


To calculate the gradient dv/dx just multipliy the gradients along the way

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

Additional Slides

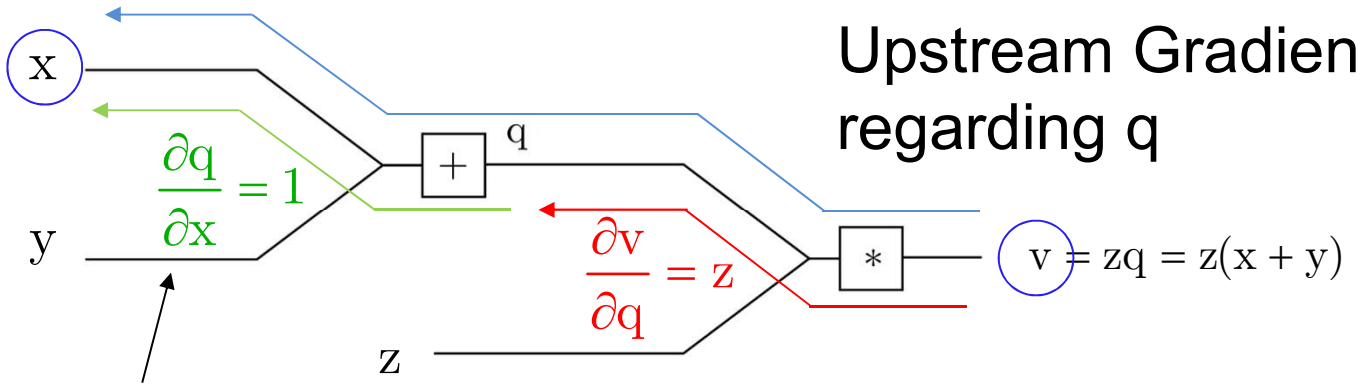
To calculate the gradient dv/dx you can simply multiple the gradients between v and x.



Computational Graph

Simple example

Gradient „flows“

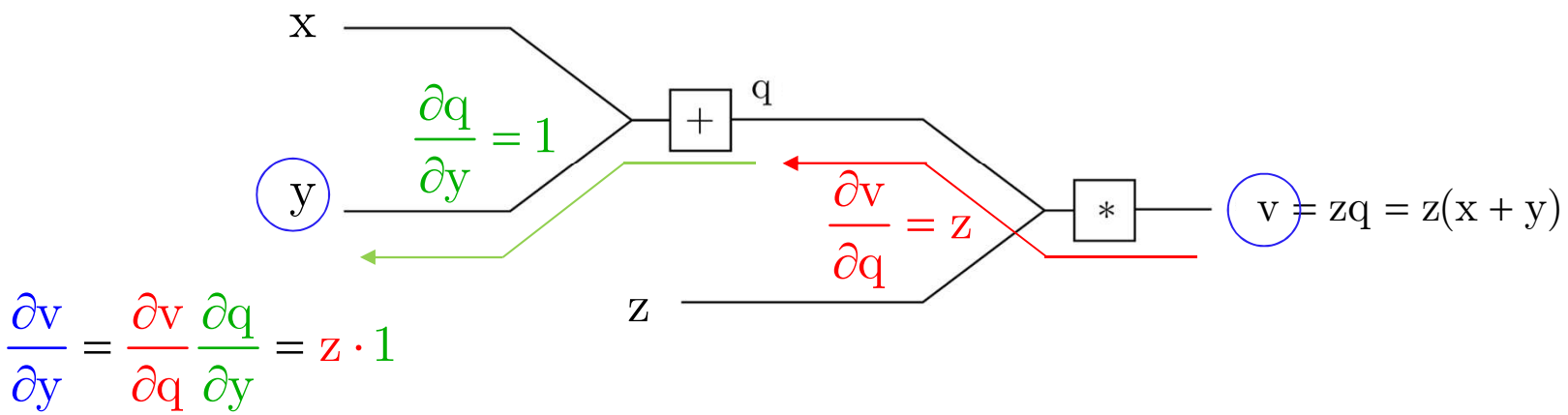


Upstream Gradient regarding q

Downstream Gradient regarding q

Computational Graph

Simple example



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

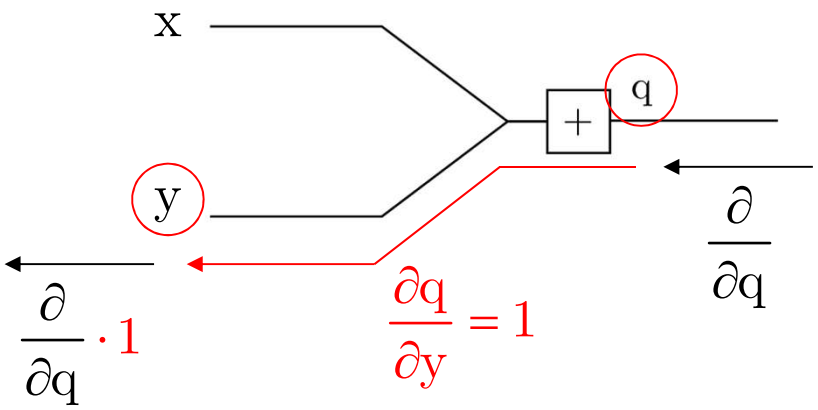
Additional Slides

With the same method the gradient between dv / dy can be cacluated.

Computational Graph

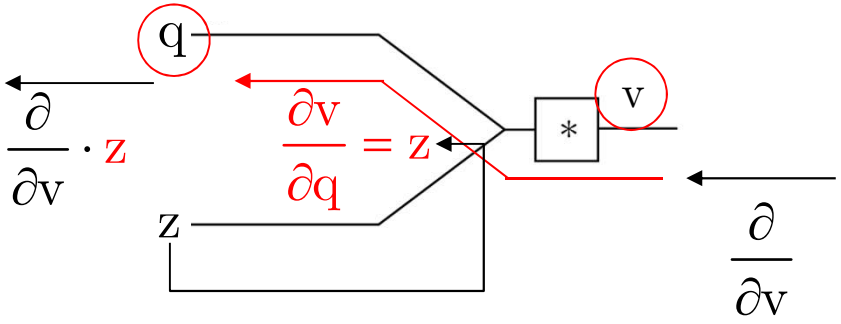
Standard operations

Addition:



Downstream gradient remains the same

Multiplication



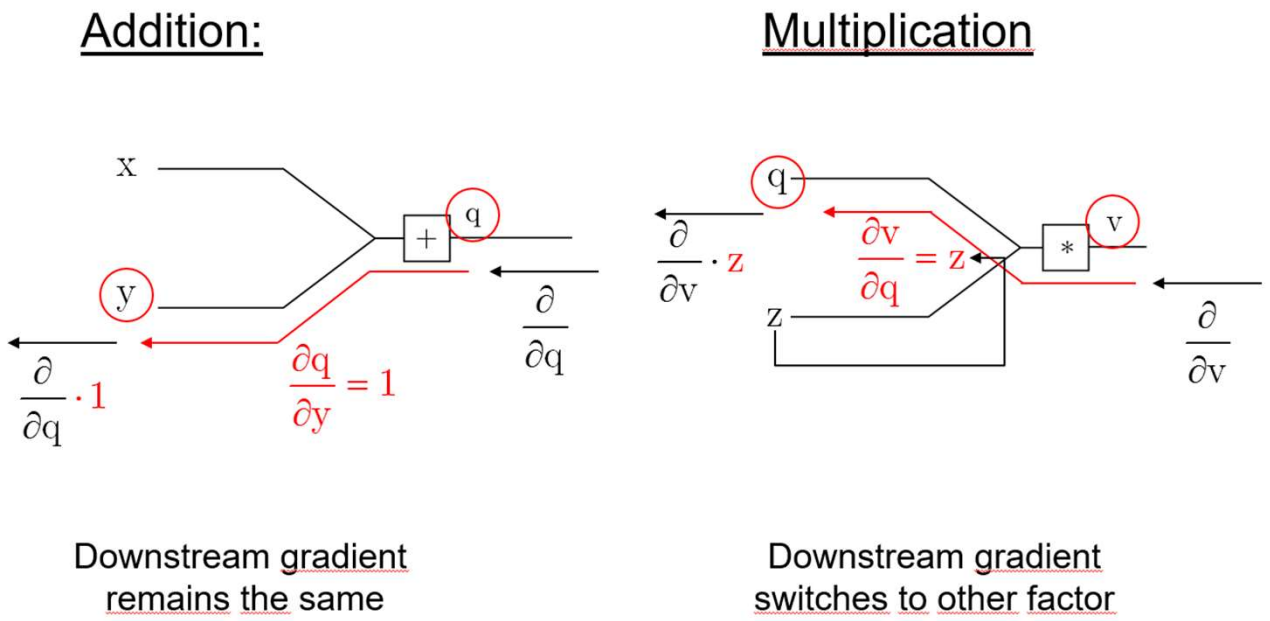
Downstream gradient switches to other factor

Additional Slides

For Additions and multiplications simple rules for calculating the downstream gradient can be defined

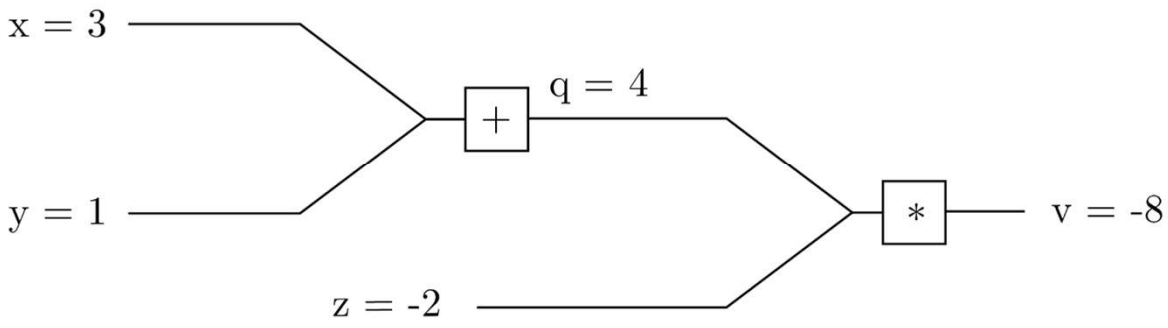
Additions: Additions don’t change the incoming upstream gradient (here for example (d / dv))

Multiplications: For Multiplications the upstream gradient (d / dv) needs to be multiplied with the other factor (here z)



Computational Graph

Simple example, with numbers



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

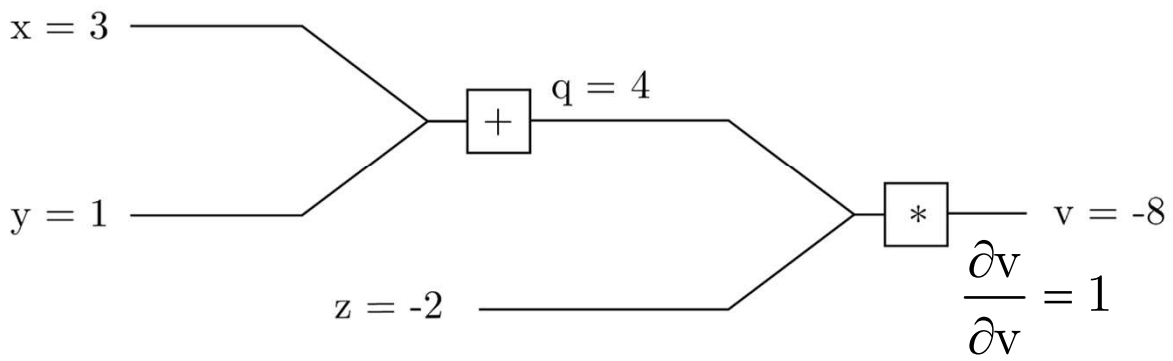
$$\frac{\partial v}{\partial q} =$$

$$\frac{\partial v}{\partial x} =$$

$$\frac{\partial v}{\partial y} =$$

Computational Graph

Simple example, with numbers



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

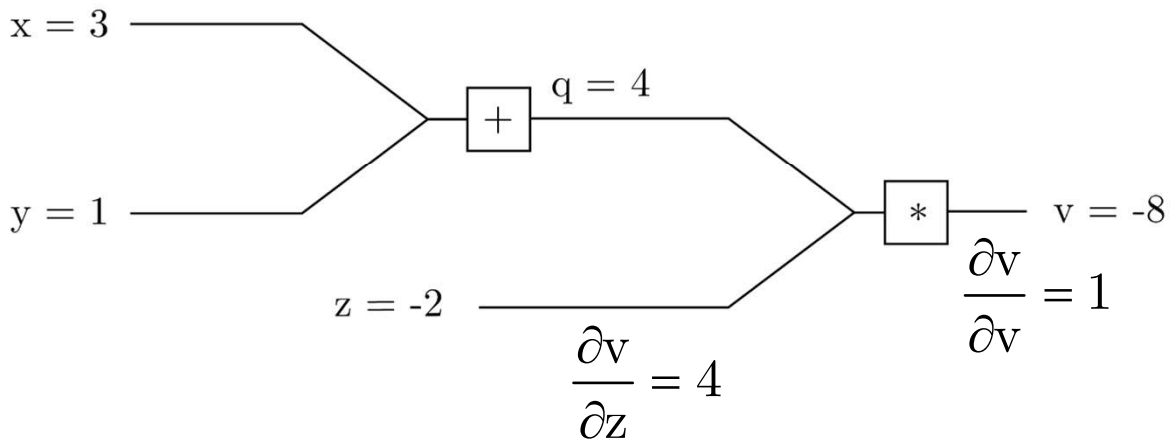
$$\frac{\partial v}{\partial q} =$$

$$\frac{\partial v}{\partial x} =$$

$$\frac{\partial v}{\partial y} =$$

Computational Graph

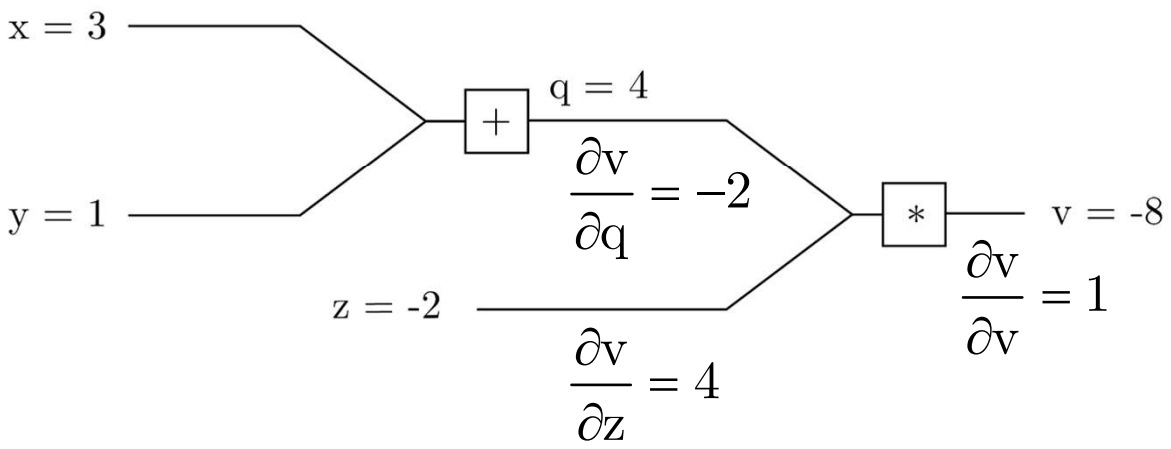
Simple example, with numbers



$$\frac{\partial v}{\partial z} = 1 \cdot 4 = 4 \qquad \frac{\partial v}{\partial q} = \qquad \frac{\partial v}{\partial x} = \qquad \frac{\partial v}{\partial y} =$$

Computational Graph

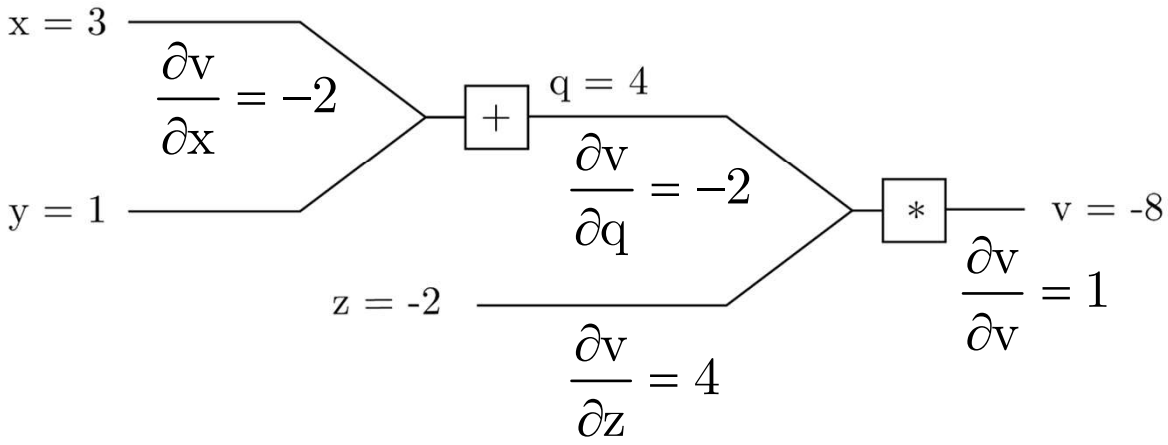
Simple example, with numbers



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

Computational Graph

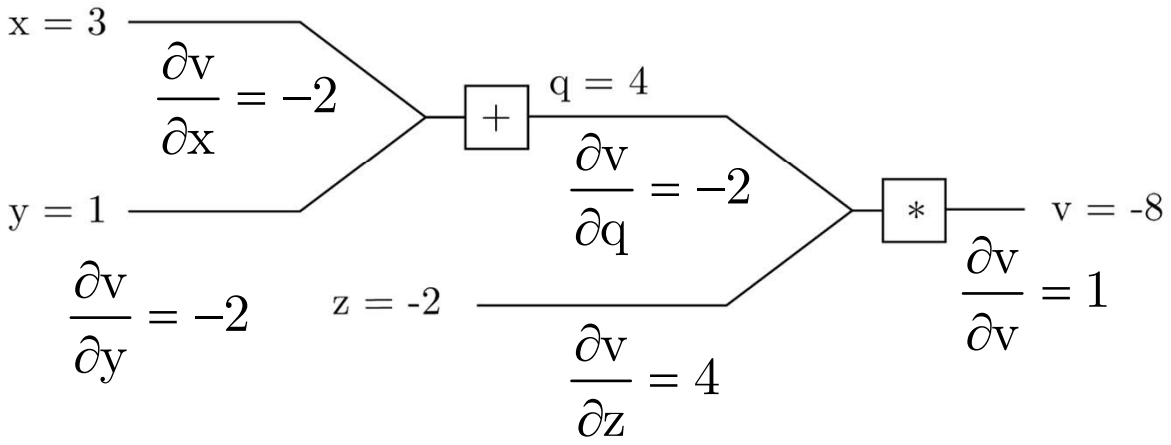
Simple example, with numbers



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

Computational Graph

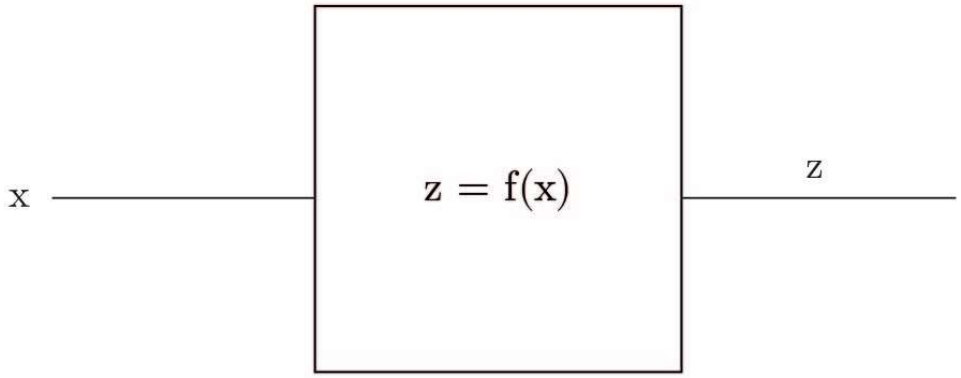
Simple example, with numbers



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

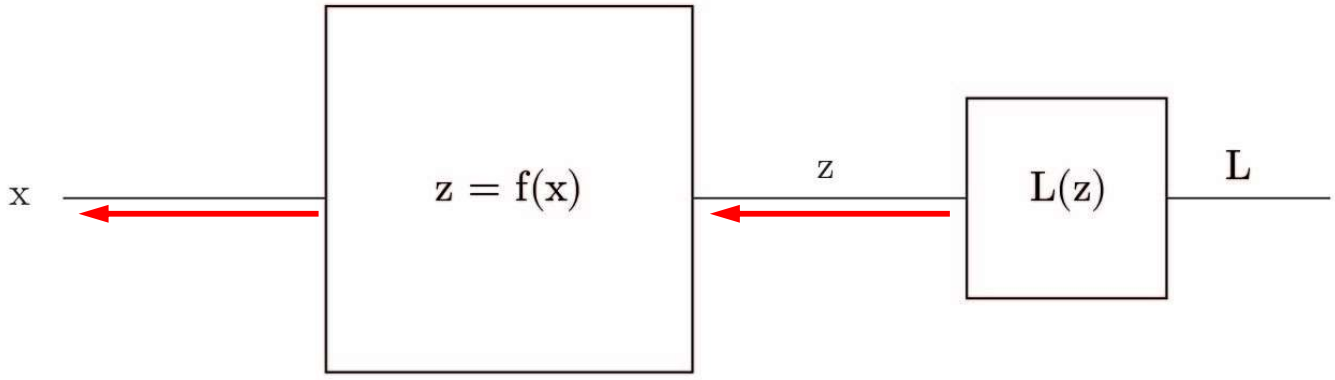
Computational Graph

Abstract function



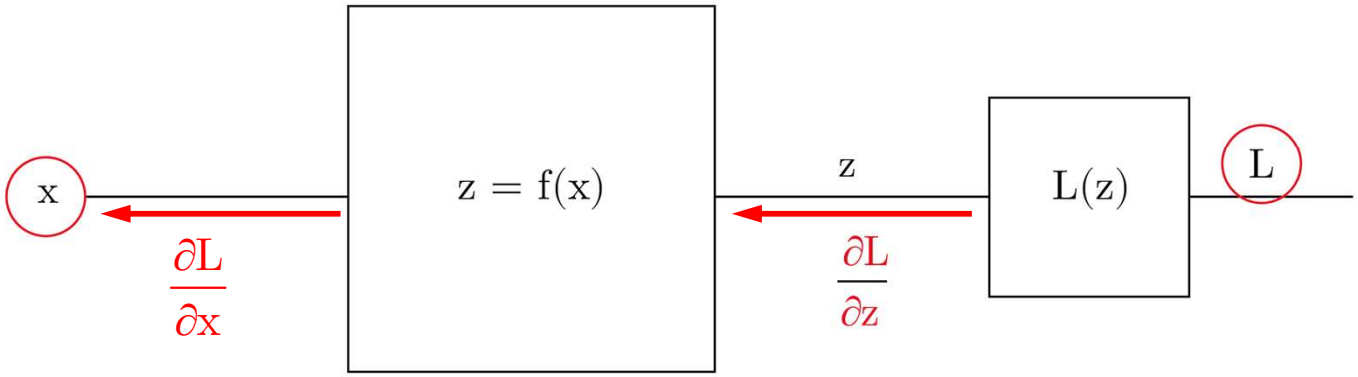
Computational Graph

Abstract function



Computational Graph

Abstract function



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

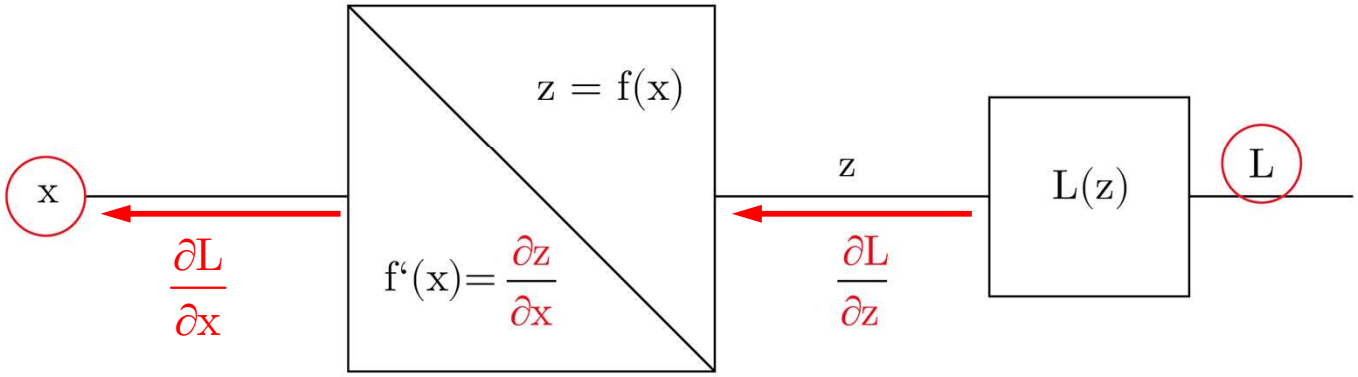
Additional Slides

In the computational graph example we looked at simple mathematical operations. A more abstract form of those operations are functions. Same as for addition and multiplication the local gradient can be calculated for complex function. (Also a multiplication can be a function f(x,y) = x*y)

Usually the local gradient is computed and saved during the forward path. Since the local gradient is just the derivative of the function, the operation can be computationally inexpensive.

Computational Graph

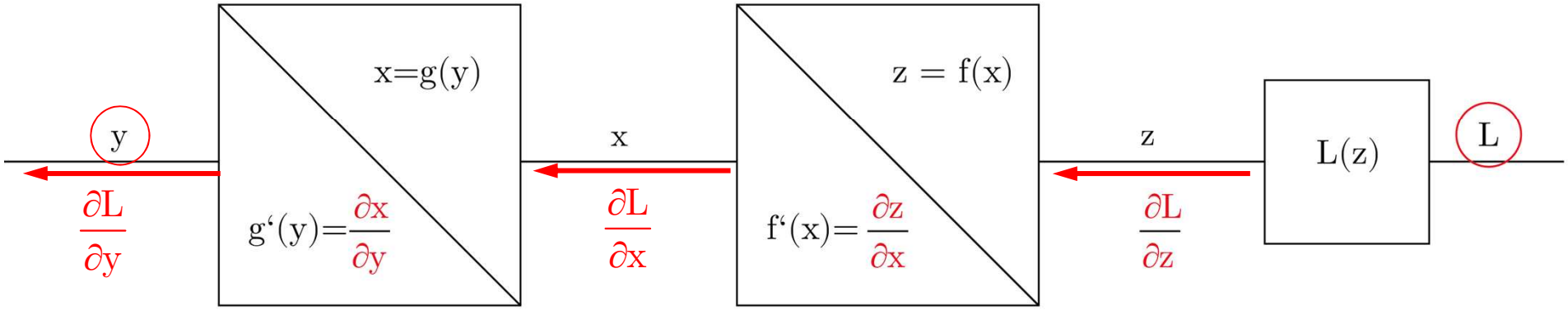
Abstract function



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

Computational Graph

Abstract function



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

Additional Slides

For calculating the derivative dL/dy, (how does the error or Loss function change, when x is being changed) we just need to multiple the downstream gradients along the way.

Now chaining functions and calculating derivaties becomes very easy, since the gradients get propaged backwards (therefore backpropagtion) through the network, the compley gradient dL/db is just a multiplication of local gradients.

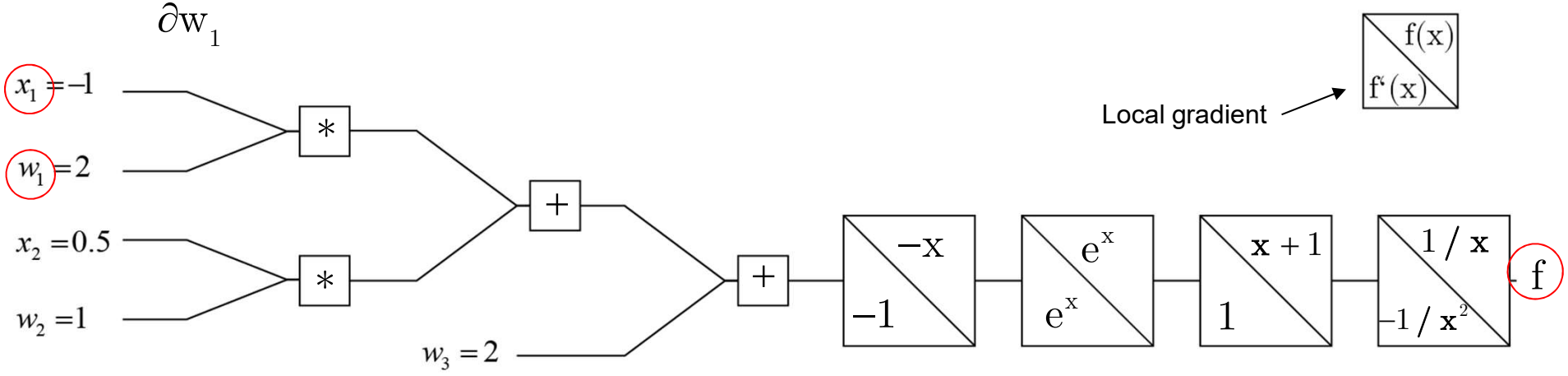
Computational Graph

Complex Example

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

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

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



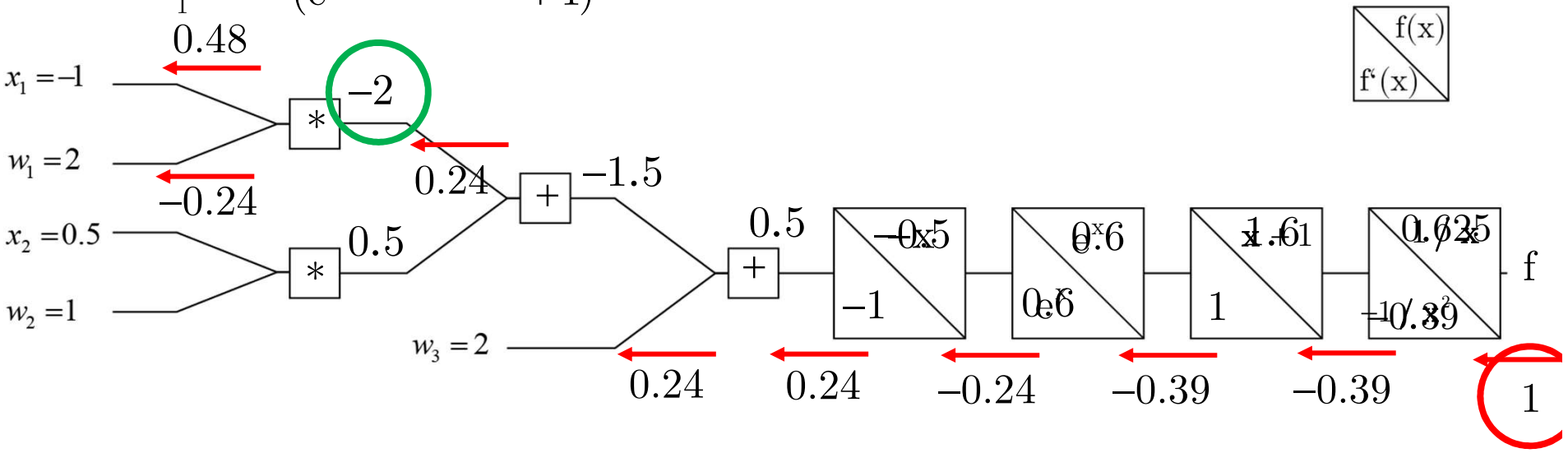
Computational Graph

Complex Example

$$\frac{\partial f}{\partial x_1} = w_1 \frac{e^{-w_1x_1-w_2x_2-w_3}}{(e^{-w_1x_1-w_2x_2-w_3}+1)^2} = 0.470074$$

$$\frac{\partial f}{\partial w_1} = x_1 \frac{e^{-w_1x_1-w_2x_2-w_3}}{(e^{-w_1x_1-w_2x_2-w_3}+1)^2} = -0.235004$$

$$f(x_1, x_2) = \frac{1}{1+e^{-(x_1w_1+x_2w_2+w_3)}}$$



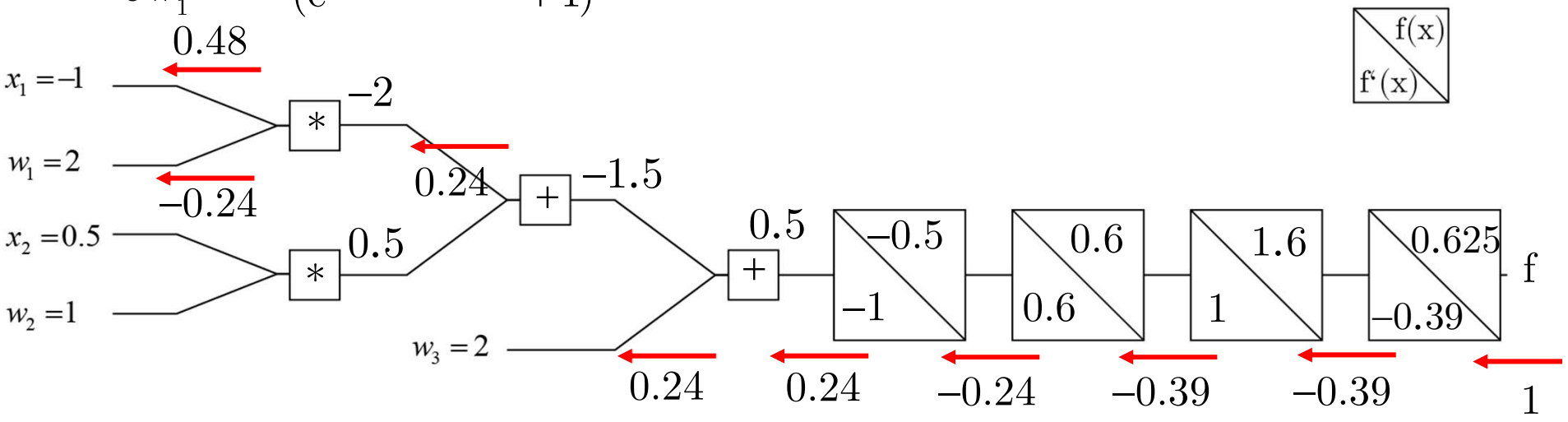
Computational Graph

Complex Example

$$\frac{\partial f}{\partial x_1} = w_1 \frac{e^{-w_1x_1-w_2x_2-w_3}}{(e^{-w_1x_1-w_2x_2-w_3}+1)^2} = 0.470074$$

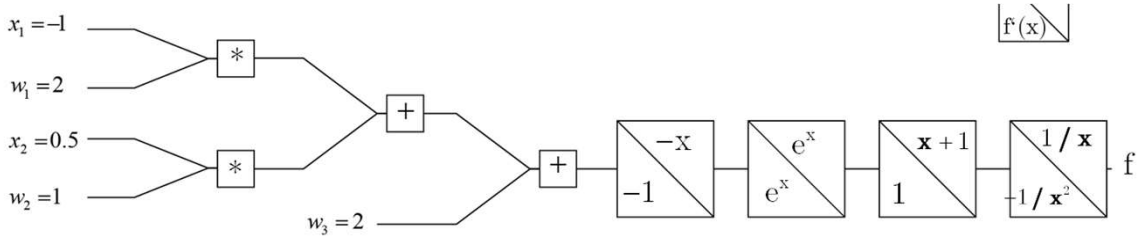
$$\frac{\partial f}{\partial w_1} = x_1 \frac{e^{-w_1x_1-w_2x_2-w_3}}{(e^{-w_1x_1-w_2x_2-w_3}+1)^2} = -0.235004$$

$$f(x_1, x_2) = \frac{1}{1+e^{-(x_1w_1+x_2w_2+w_3)}}$$

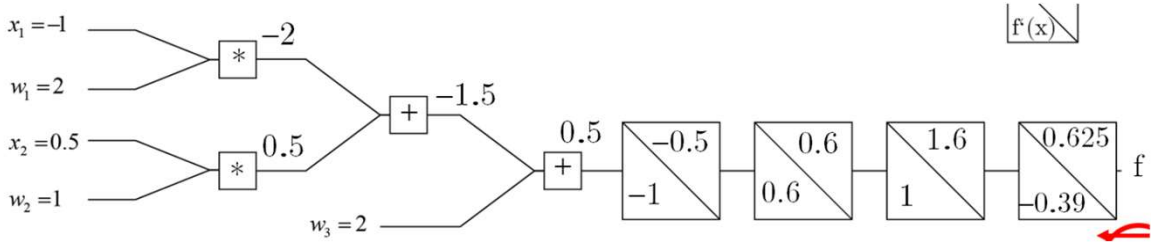


Additional Slides

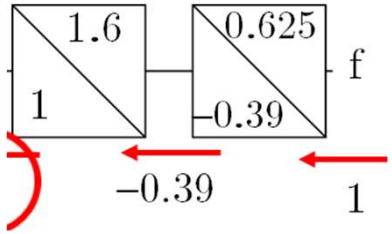
Let's have a look at a more complex example and calculate a derivate with the new method. Here the complex function f(x1,x2) is shown in form of a computational graph, and we want to calculate df/dx1:



First we calculate the forward path:



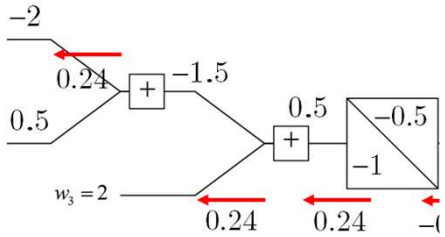
To get to df/dx1 we just need to multiply the local gradients from f to x1. At the end of the graph we start with df/df = 1:



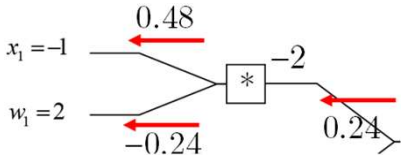
The next gradient can be calculated by simply multiplying the upstream gradient (df/df = 1) with the local derivative (-0.39), resulting in -0.39

Additional Slides

This get repeated along the computational graph. For additions we know that the gradient remains the same (0.24) :



And for multiplications we know we need to multiply the upstream gradient (0.24) with the other factor:



df / dx1 = 0.24 * w1 = 0.48

df / dw1 = 0.24 * x1 = -0.24

We get the same result as with the analytic method:

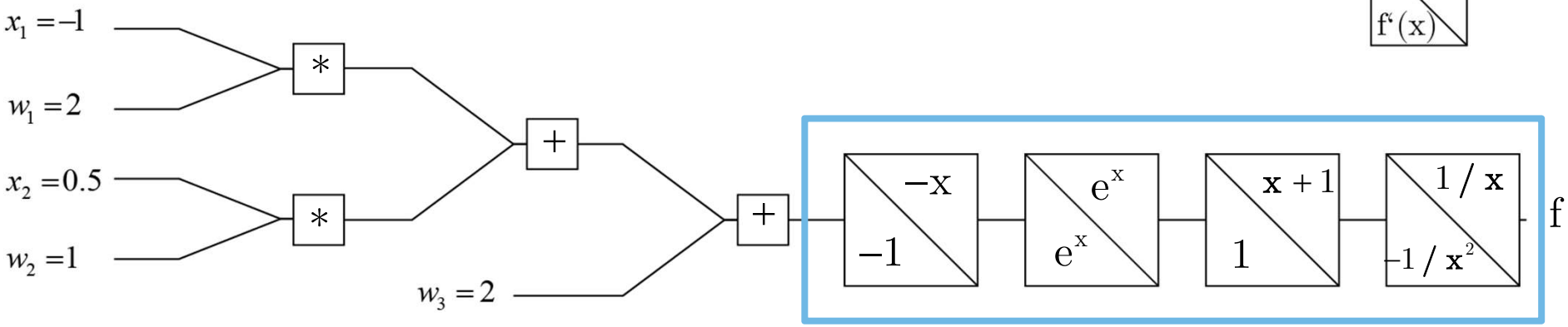
$$\frac{\partial f}{\partial x_1} = w_1 \frac{e^{-w_1x_1 - w_2x_2 - w_3}}{(e^{-w_1x_1 - w_2x_2 - w_3} + 1)^2} = 0.470074$$

$$\frac{\partial f}{\partial w_1} = x_1 \frac{e^{-w_1x_1 - w_2x_2 - w_3}}{(e^{-w_1x_1 - w_2x_2 - w_3} + 1)^2} = -0.235004$$

Computational Graph

Complex Example

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



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

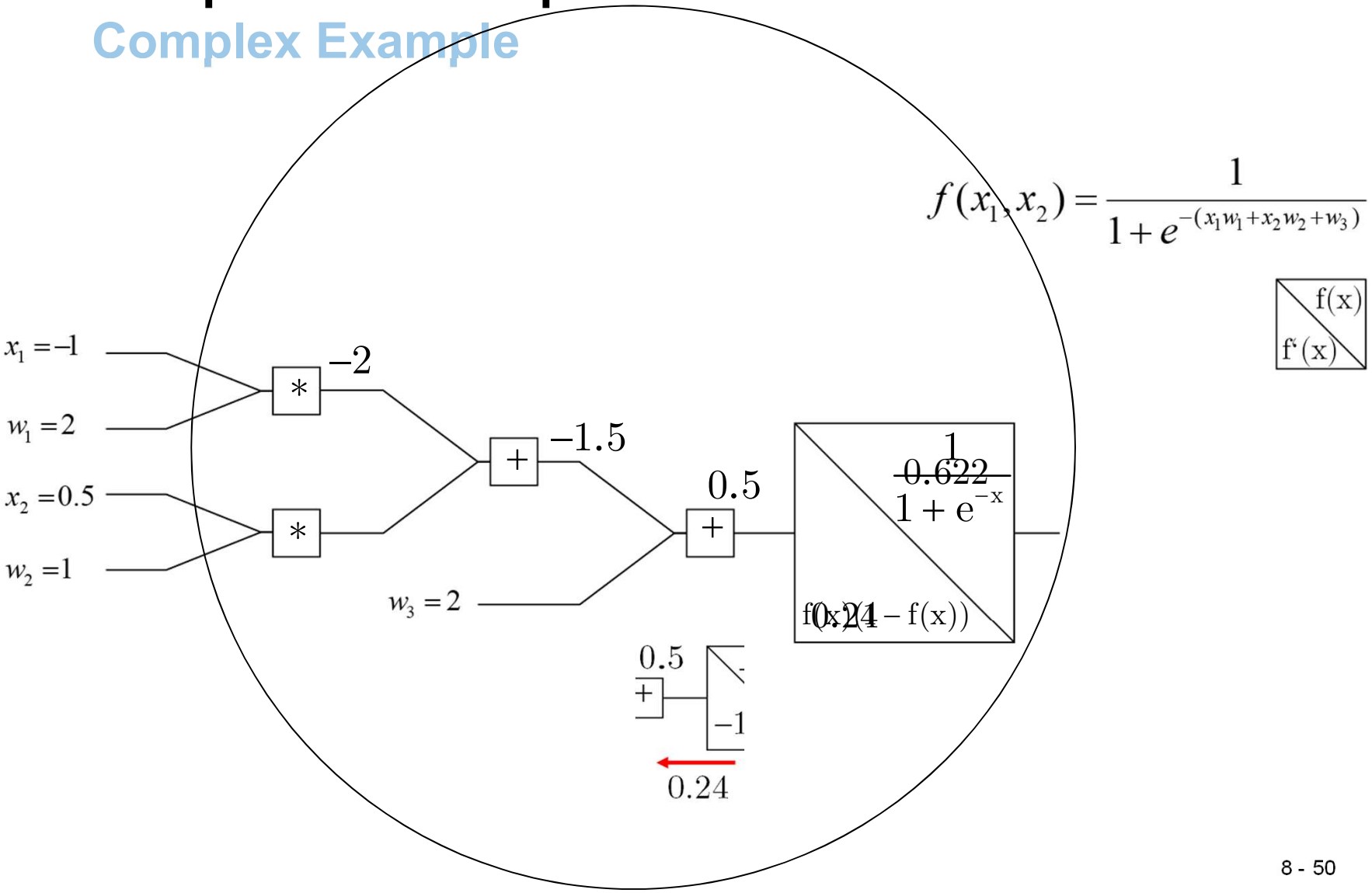
$$f'(x) = f(x)(1 - f(x))$$

Additional Slides

The four functions can be combined to one function with a simple derivative

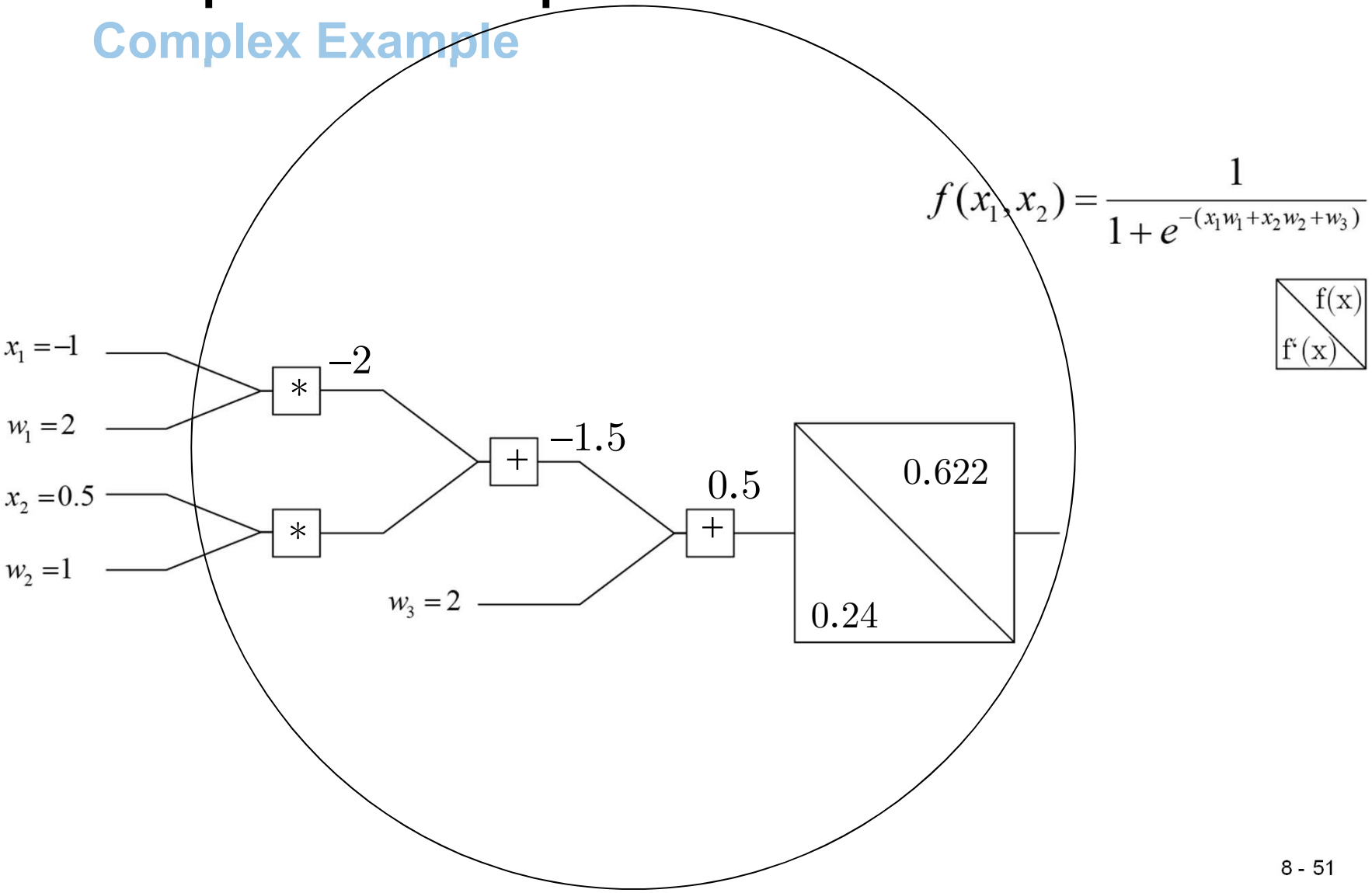
Computational Graph

Complex Example



Computational Graph

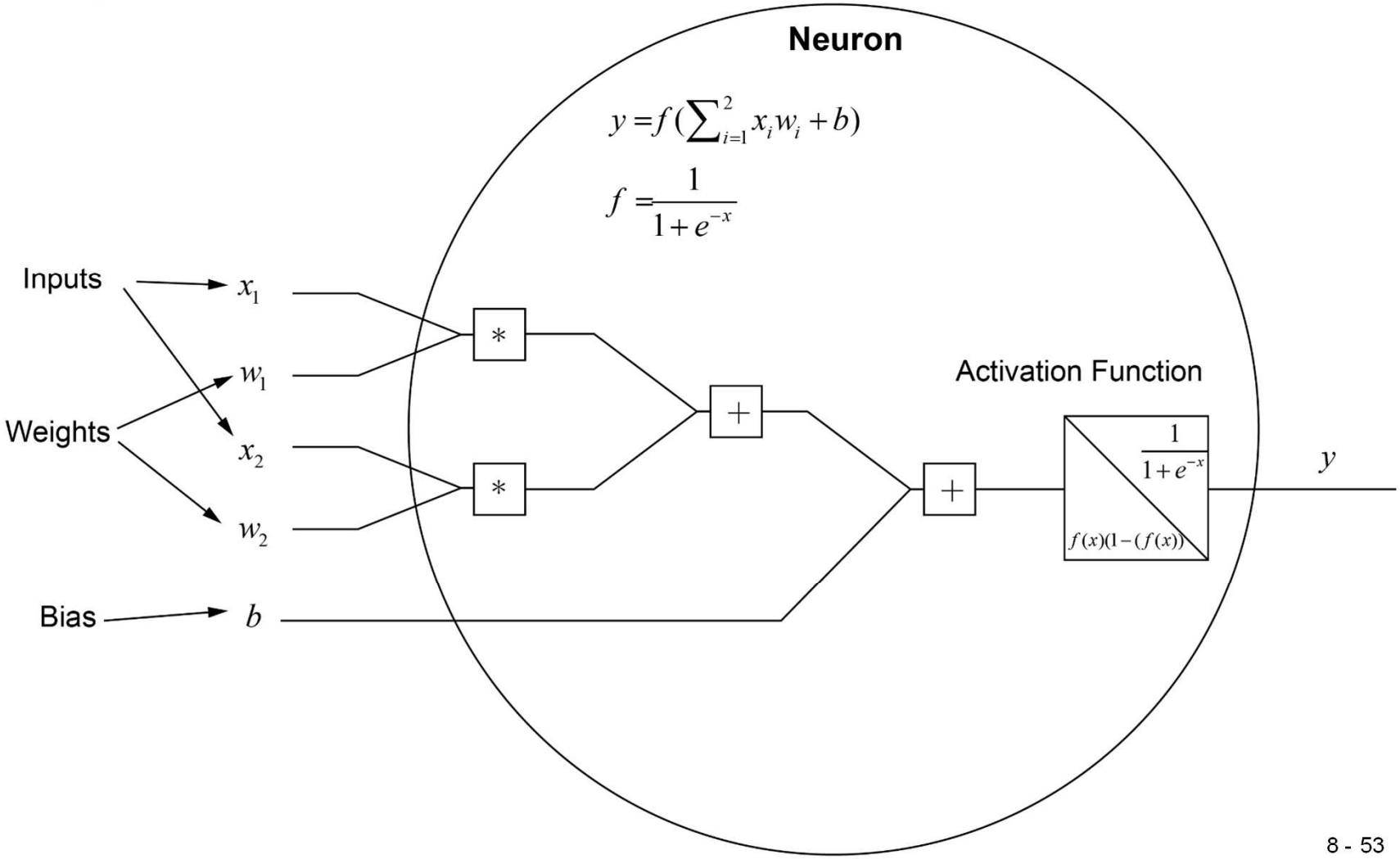
Complex Example



The result is the computational graph for a single neuron.

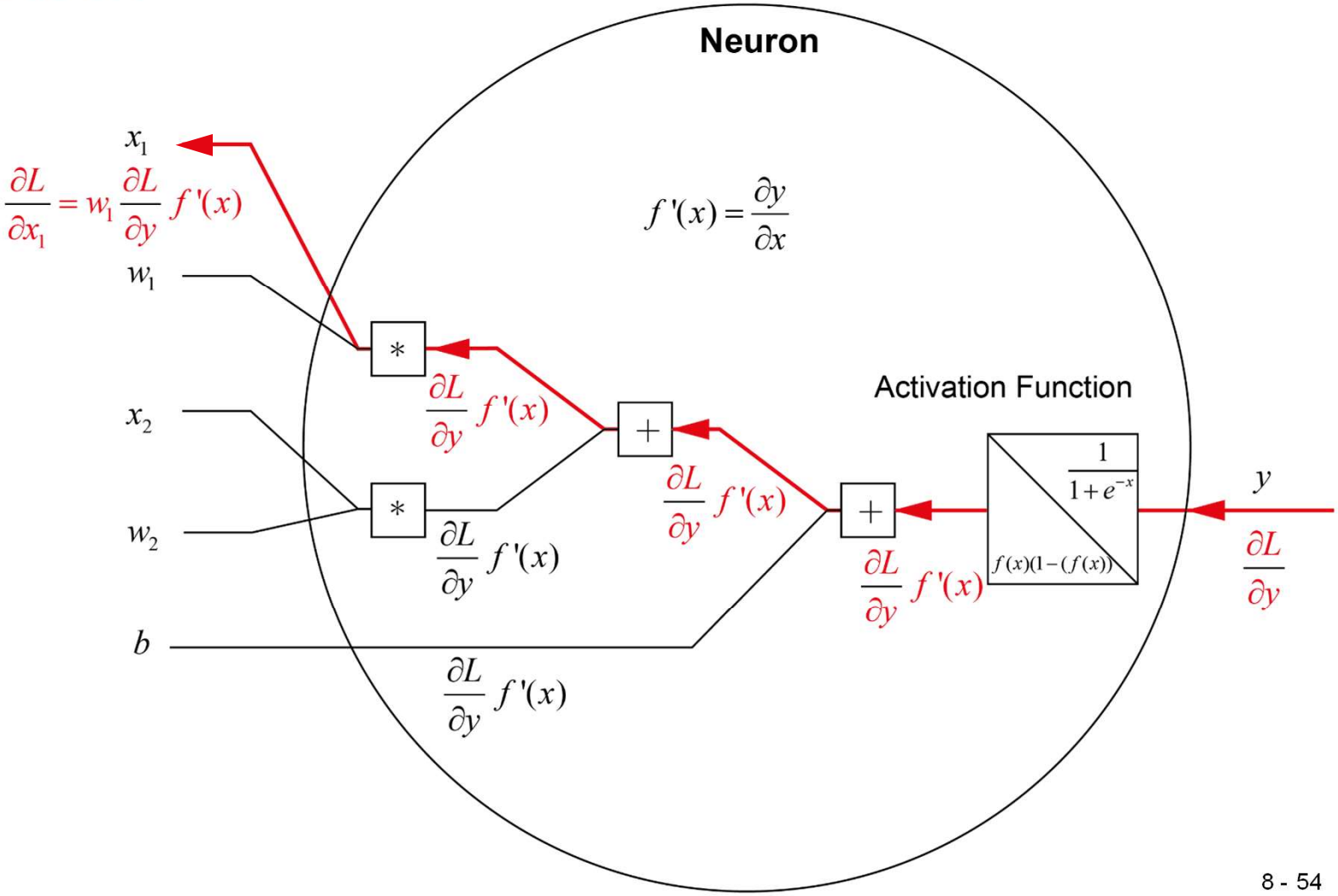
Computational Graph

Neuron



Computational Graph

Neuron



Additional Slides

We can now apply the backpropagtion algorithm to the single Neuron. The downstream gradient dL/dx1 can be calculated by multiplying the upstream gradient with the local derivative of the activation function. Since the additions don't have an impact on the gradient, the gradient remains the same. Due to the last multiplication we need to multiply the gradient with the other factor (w1) resulting in

$dL/dx1 = dL/dy\ f'(x)\ w1$

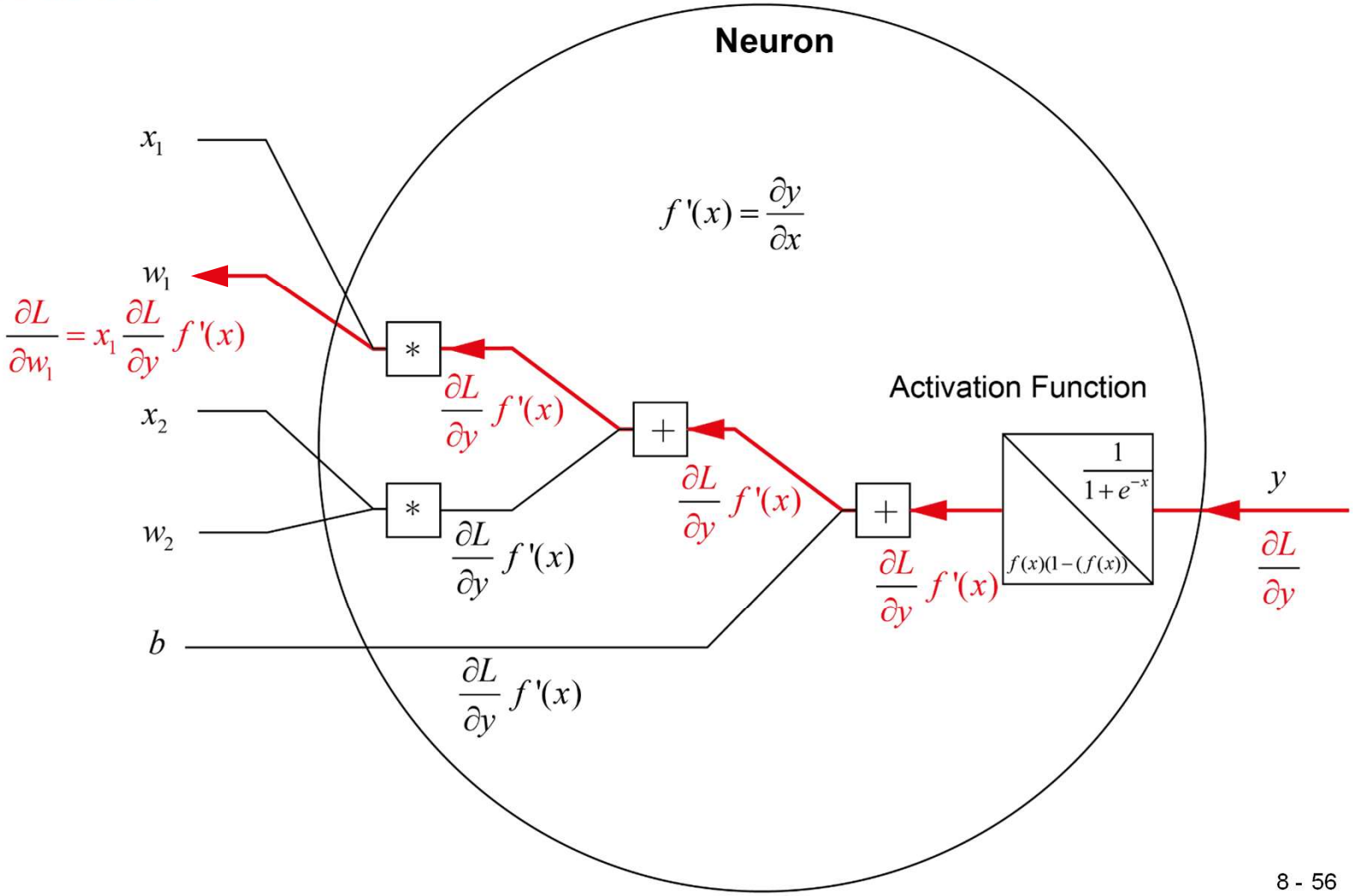
dL/dy = upstream gradient

$f'(x)$ = local derivative of activation function

$W1$ = other factor

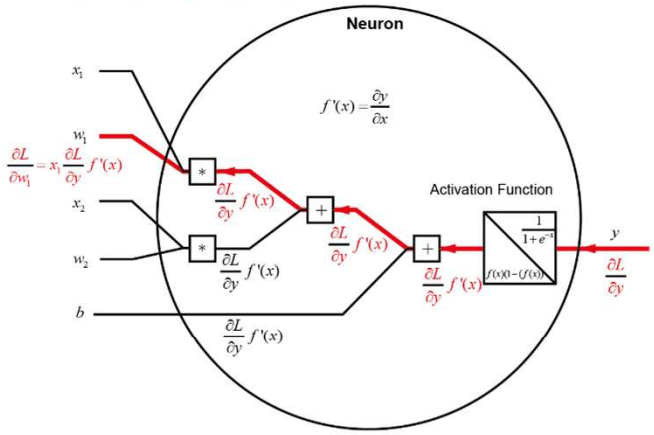
Computational Graph

Neuron

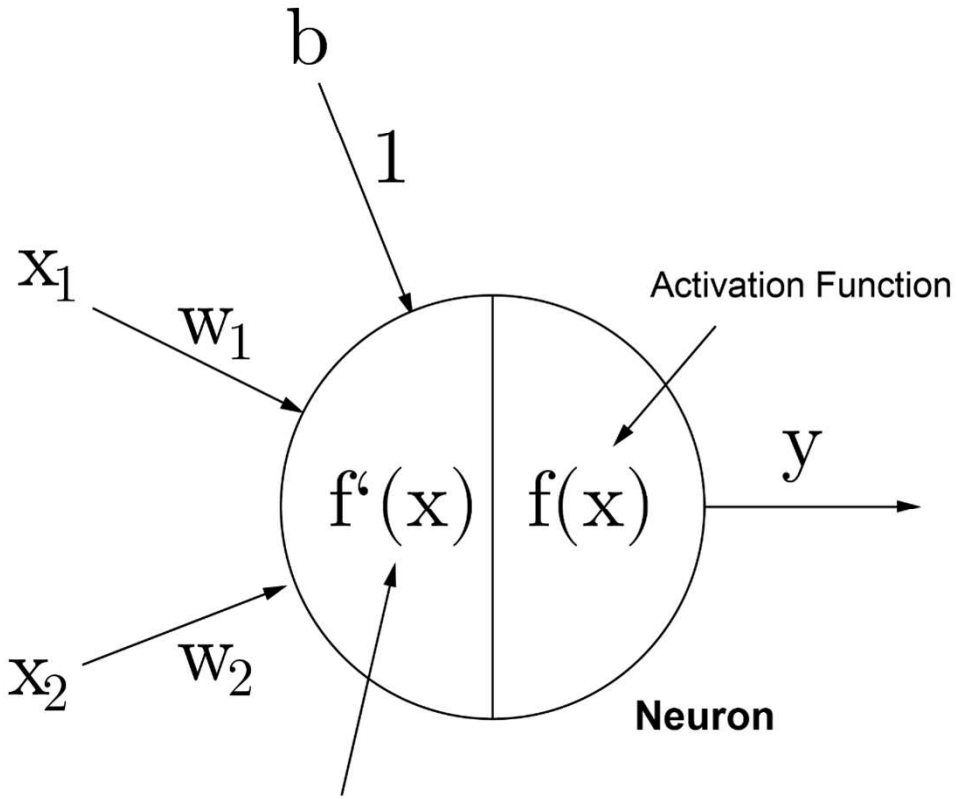


Computational Graph

Neuron

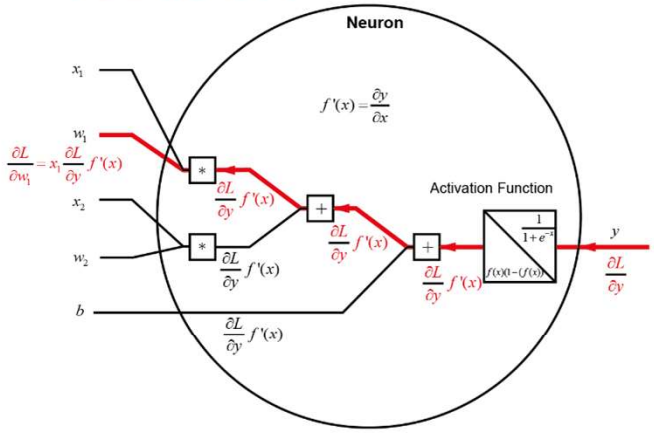


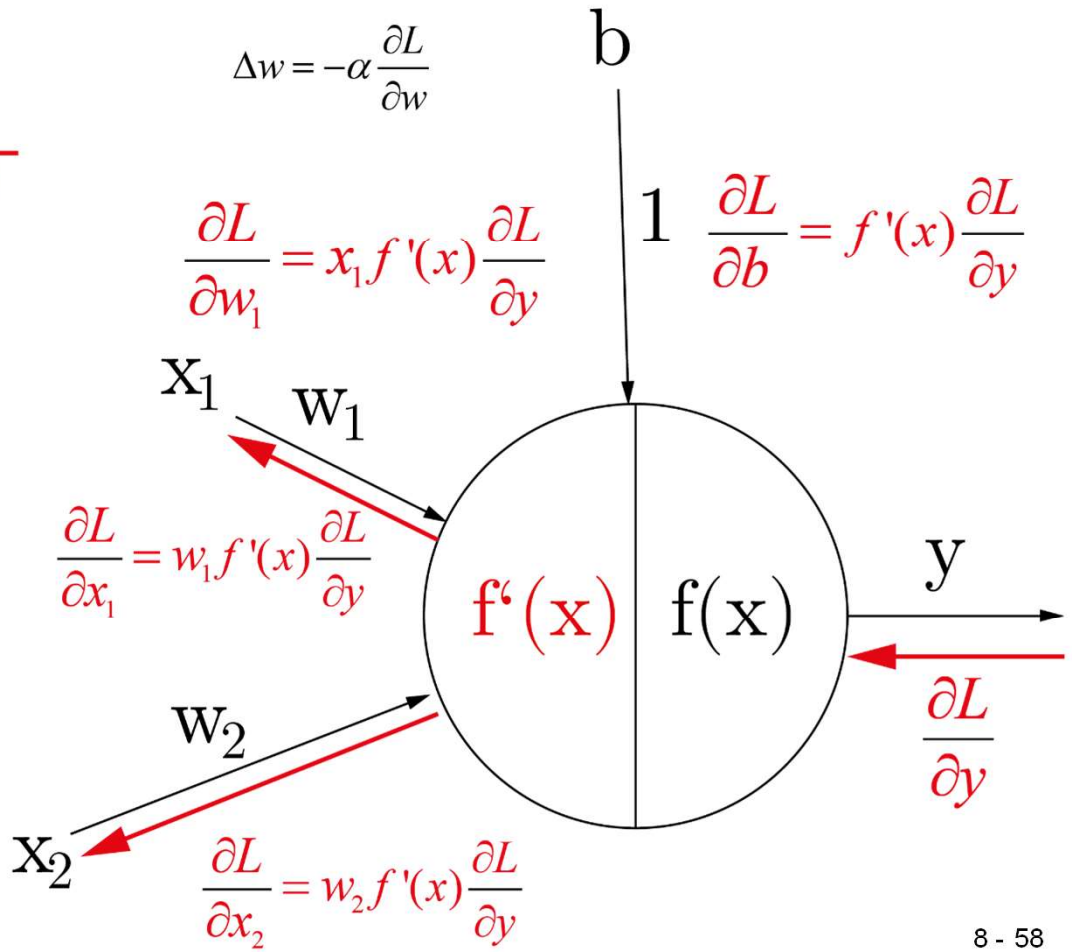
$$y = f\left(\sum_{i=1}^{2} x_i w_i + b\right)$$



Computational Graph

Neuron





Additional Slides

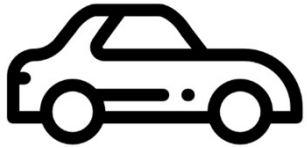
The computational graph of a neuron can be simplified by hiding the inner additions and multiplications and just showing the function and local derivative. Then the downstream gradients can be calcuclated as shown.

Deep Neural Networks

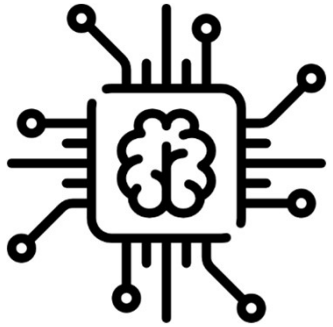
Johannes Betz / Prof. Dr. Markus Lienkamp / Prof. Dr. Boris Lohmann
(Jean-Michael Georg, M. Sc.)

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
    - 2.3 Weight Inititalisation



3. Chapter: Overview

Backpropagation

Neural Chain

we want:

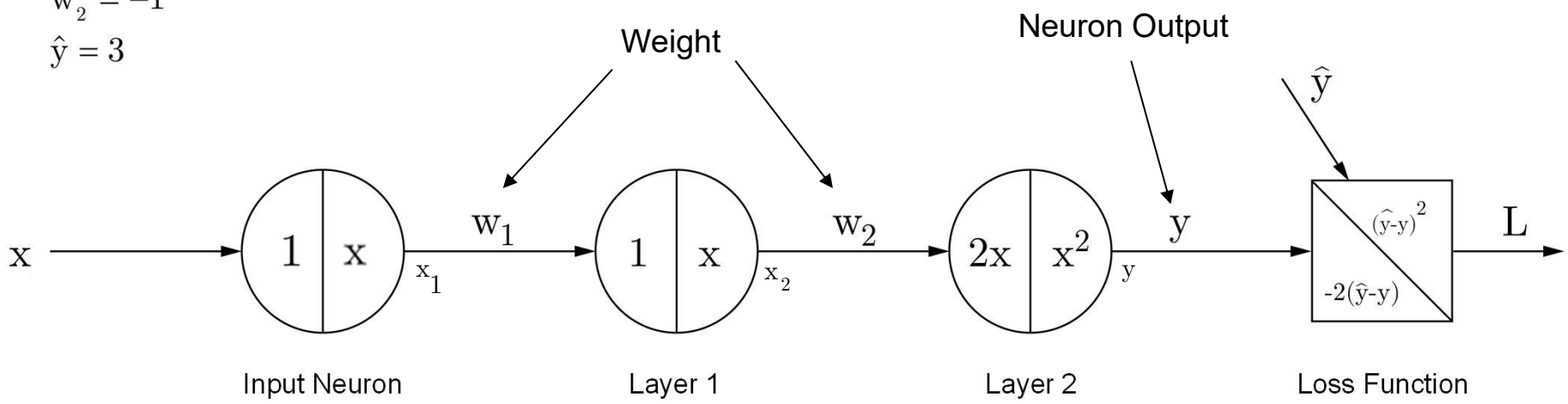
$$\Delta w_1 = -\alpha \frac{\partial L}{\partial w_1}, \Delta w_2 = -\alpha \frac{\partial L}{\partial w_2}$$

$x = 3$

$w_1 = 0.5$

$w_2 = -1$

$\hat{y} = 3$



Backpropagation

Neural Chain

we want:

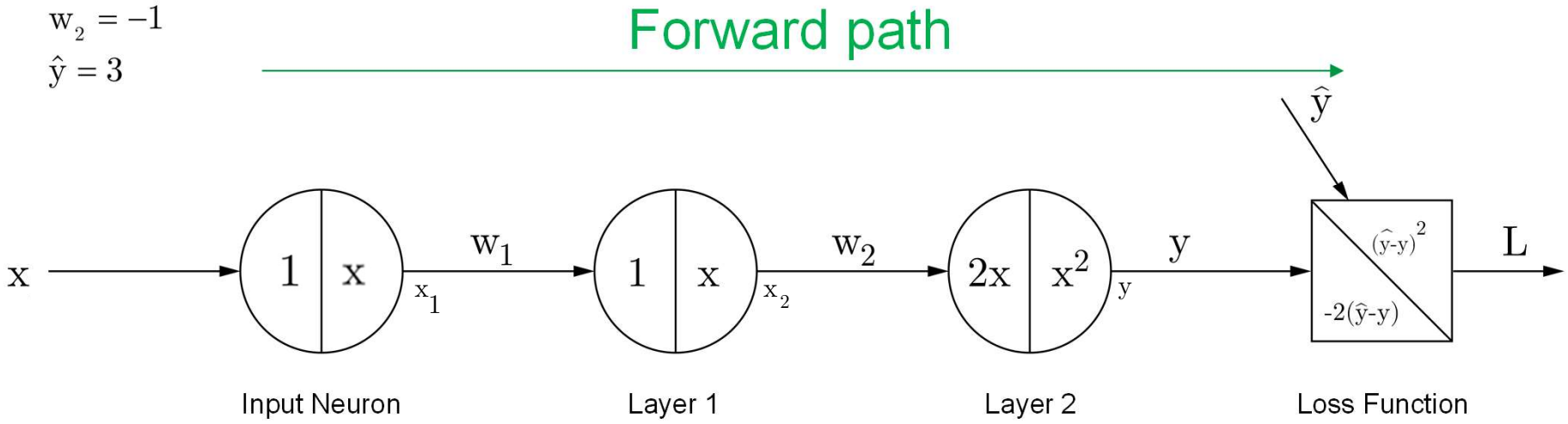
$$\Delta w_1 = -\alpha \frac{\partial L}{\partial w_1}, \Delta w_2 = -\alpha \frac{\partial L}{\partial w_2}$$

$x = 3$

$w_1 = 0.5$

$w_2 = -1$

$\hat{y} = 3$



Backpropagation

Neural Chain

we want:

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

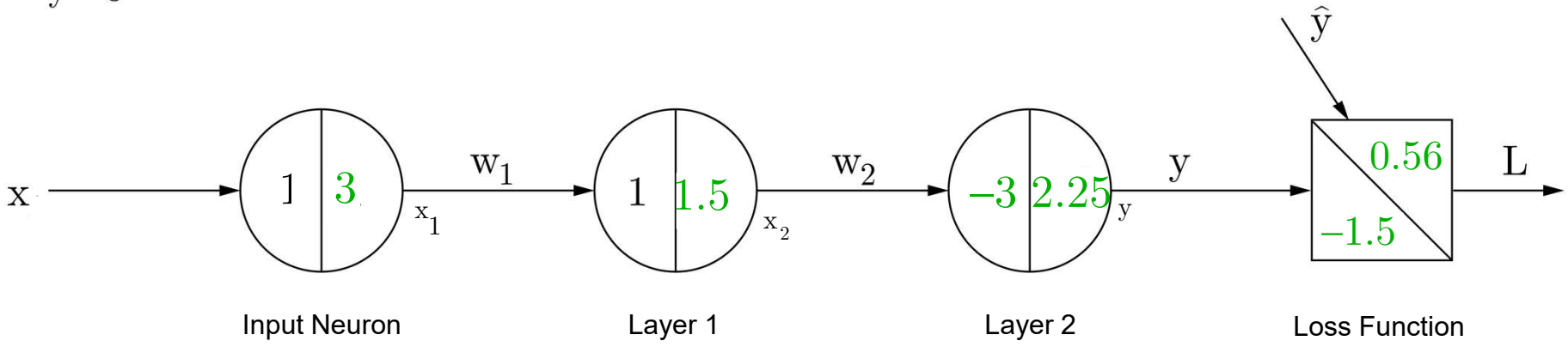
$x = 3$

$w_1 = 0.5$

$w_2 = -1$

$\hat{y} = 3$

Forward path



Backpropagation

Neural Chain

we want:

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

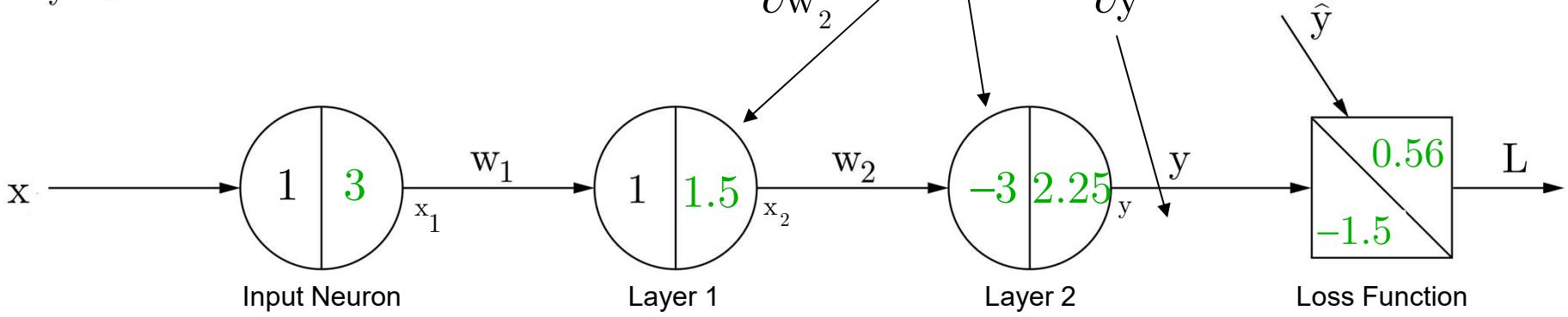
$x = 3$

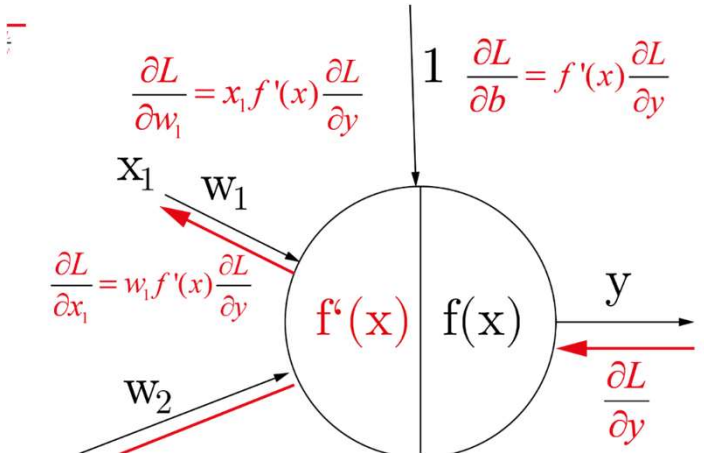
$w_1 = 0.5$

$w_2 = -1$

$\hat{y} = 3$

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





Backpropagation

Neural Chain

we want:

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

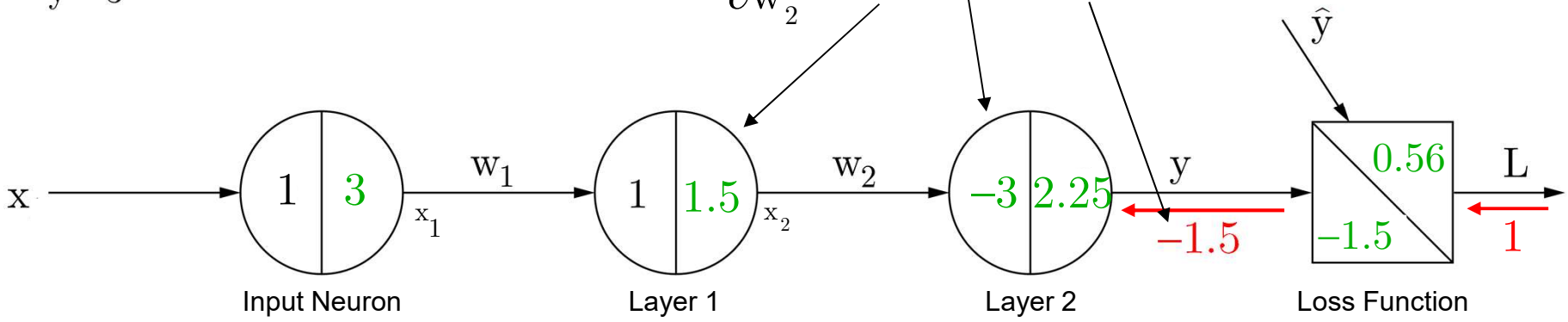
$x = 3$

$w_1 = 0.5$

$w_2 = -1$

$\hat{y} = 3$

$$\frac{\partial L}{\partial w_2} = 1.5 \cdot -3 \cdot -1.5 = 6.75$$



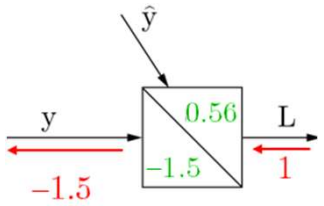
Additional Slides

Let's look at a chain of neurons (/ small neural network). We have three neurons linked together and a loss function at the end. Since we want to minimize the output of the loss function and the only parameters we can change are w1 and w2 we need to know if we need to increase or decrease w1 and w2. The direction is given be the derivative of dL/dw1 or dL/dw2. If dL / dw1 is positive the Loss function increases when we increase w1. Therefore we must change w1 in the other direction. In addition we multiple the gradient with a predefined learn rate.

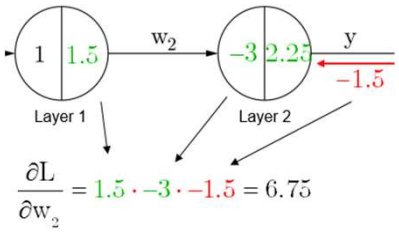
$$\Delta w_1 = -\alpha \frac{\partial L}{\partial w_1}$$

For calculating dL/dw1 we can use the backpropagation algorithm:

dL/dL is 1 and therefore the starting upstream gradient. To calculate the next gradient we simply multiply dL/dL with the local gradient of the loss function.



dL/dw2 can then be calculated by multiplying the downstream gradient (-1.5) with the local gradient (-3) and "the other factor" (1.5)



Backpropagation

Neural Chain

we want:

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

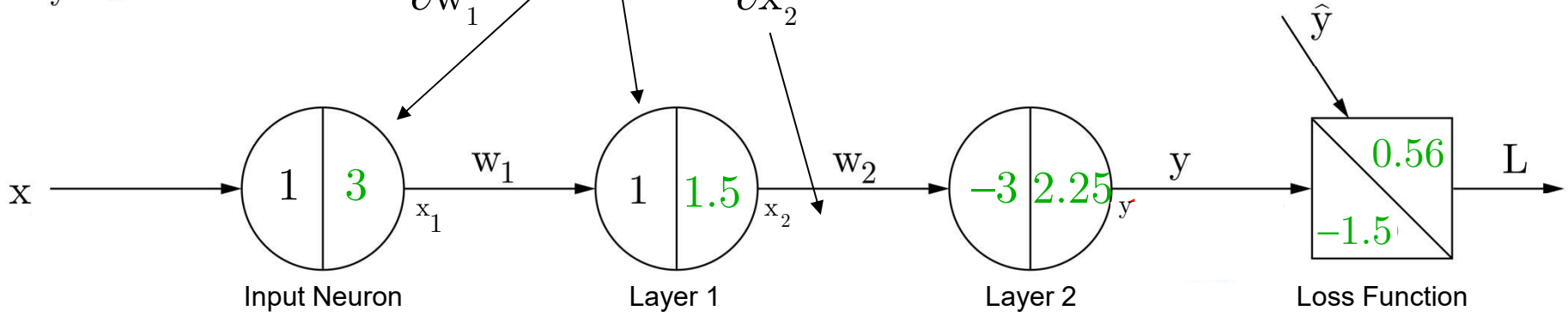
$x = 3$

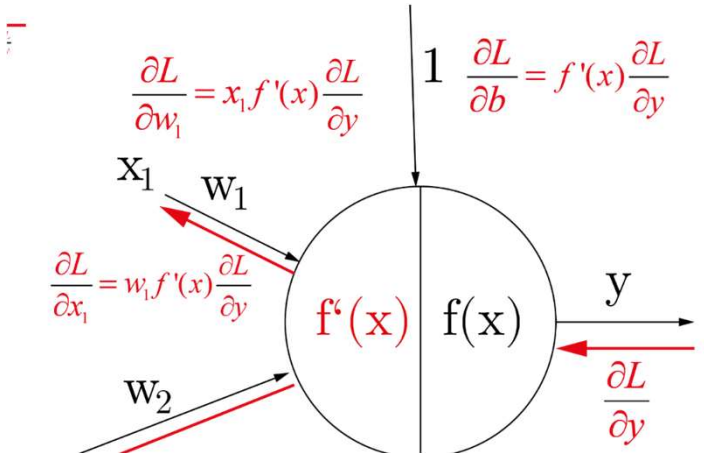
$w_1 = 0.5$

$w_2 = -1$

$\hat{y} = 3$

$$\frac{\partial L}{\partial w_1} = x_1 \cdot f'(x) \cdot \frac{\partial L}{\partial x_2}$$





Backpropagation

Neural Chain

we want:

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

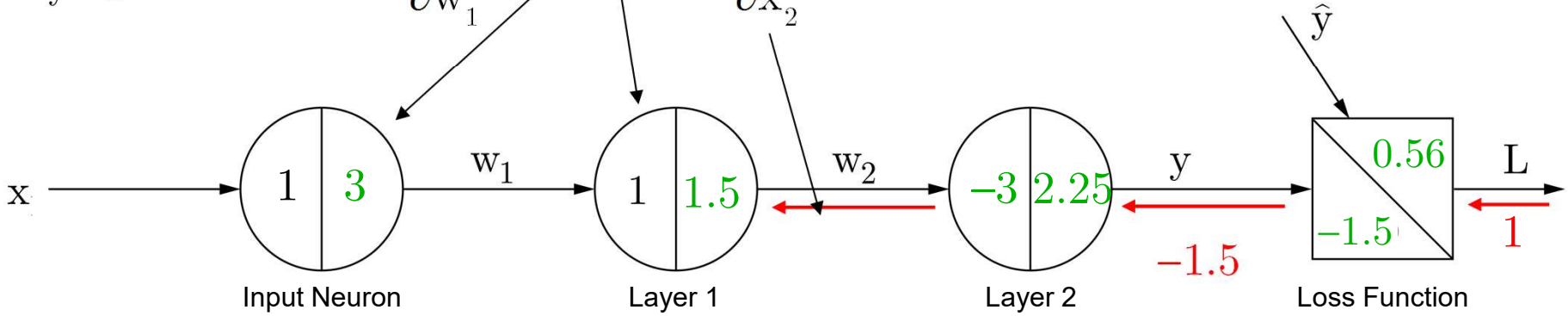
$x = 3$

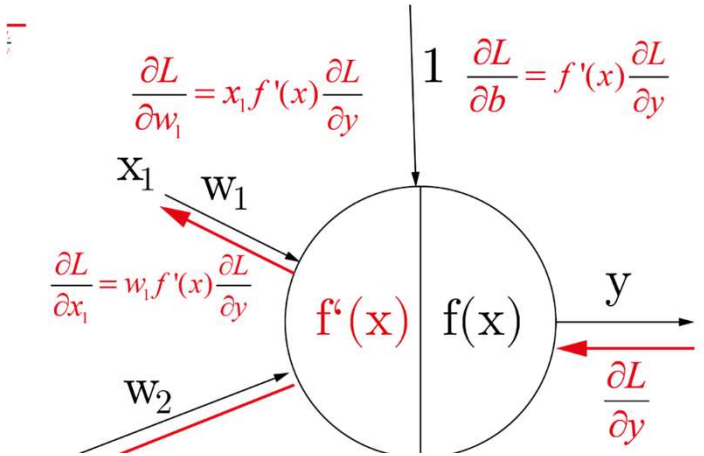
$w_1 = 0.5$

$w_2 = -1$

$\hat{y} = 3$

$$\frac{\partial L}{\partial w_1} = x_1 \cdot f'(x) \cdot \frac{\partial L}{\partial x_2}$$





Backpropagation

Neural Chain

we want:

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

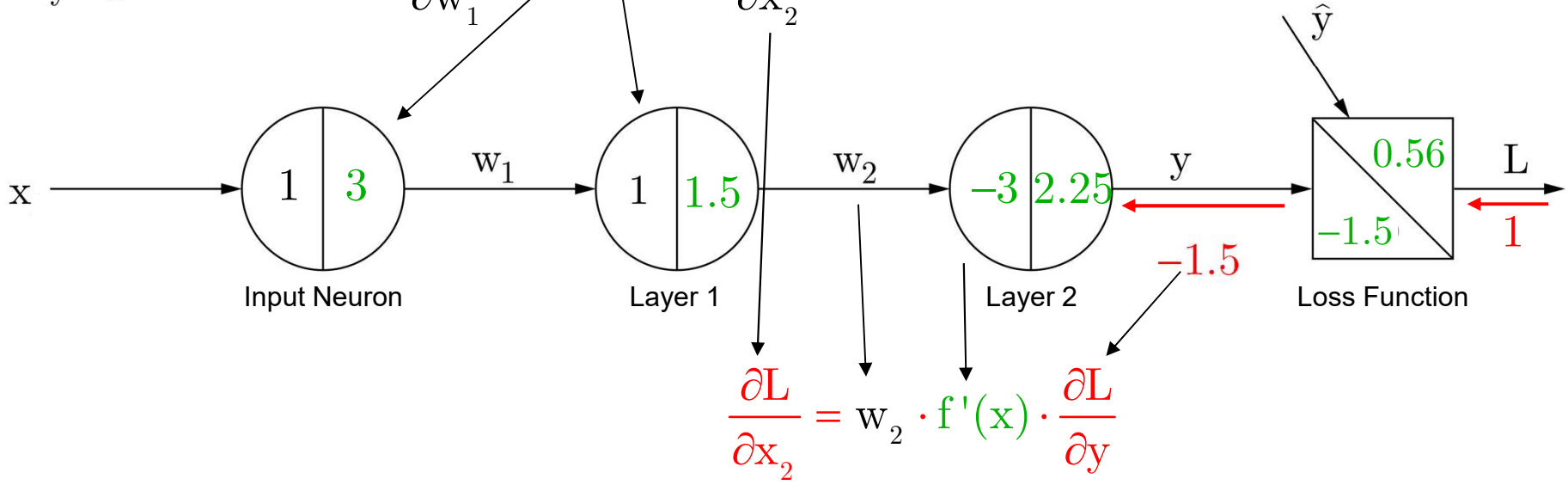
$x = 3$

$w_1 = 0.5$

$w_2 = -1$

$\hat{y} = 3$

$$\frac{\partial L}{\partial w_1} = x_1 \cdot f'(x) \cdot \frac{\partial L}{\partial x_2}$$



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

Backpropagation

Neural Chain

we want:

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

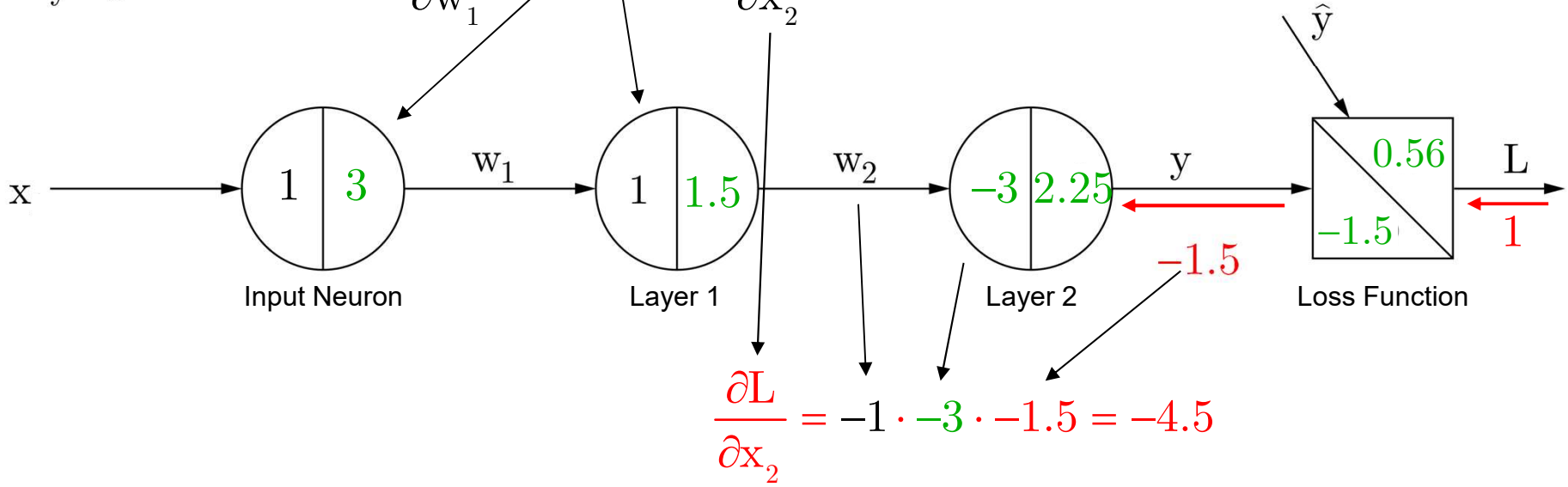
$x = 3$

$w_1 = 0.5$

$w_2 = -1$

$\hat{y} = 3$

$$\frac{\partial L}{\partial w_1} = x_1 \cdot f'(x) \cdot \frac{\partial L}{\partial x_2}$$



$$\frac{\partial L}{\partial x_2} = -1 \cdot -3 \cdot -1.5 = -4.5$$

Backpropagation

Neural Chain

we want:

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

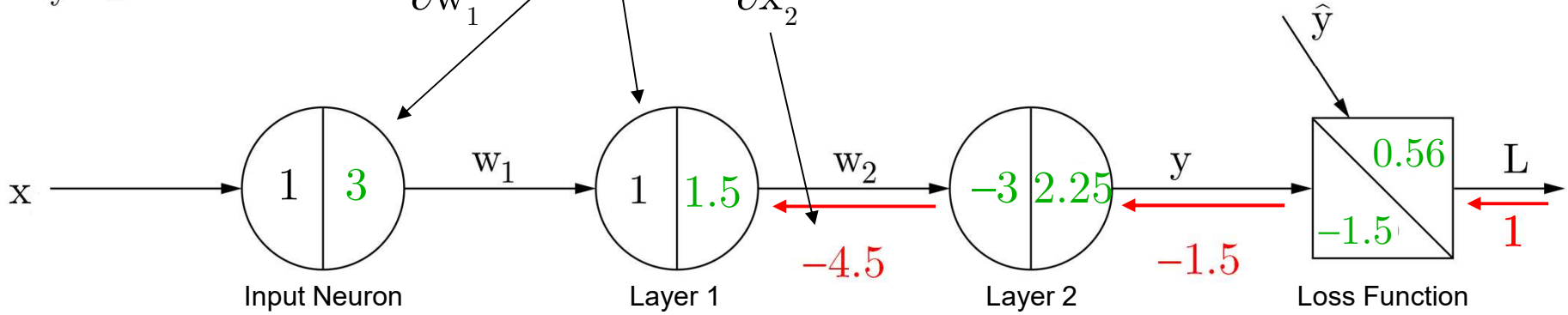
$x = 3$

$w_1 = 0.5$

$w_2 = -1$

$\hat{y} = 3$

$$\frac{\partial L}{\partial w_1} = x_1 \cdot f'(x) \cdot \frac{\partial L}{\partial x_2}$$



Backpropagation

Neural Chain

we want:

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

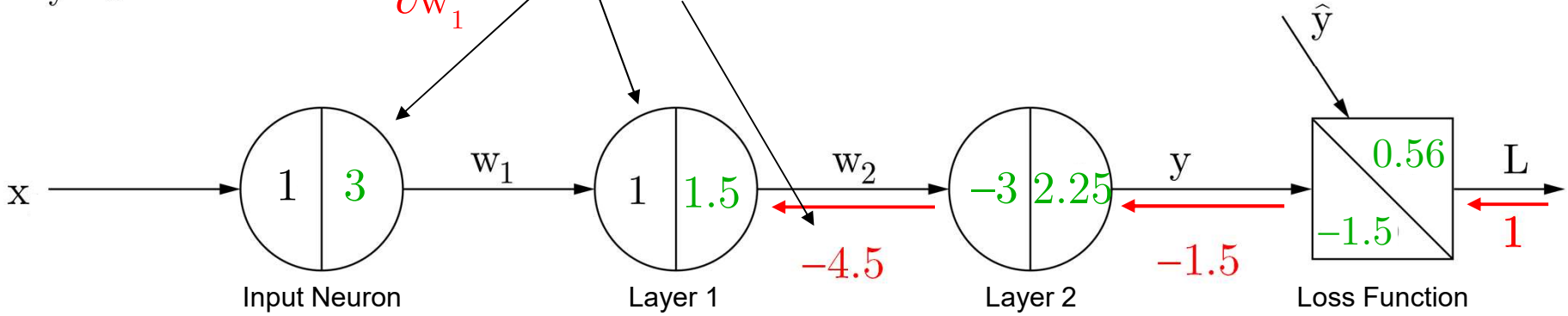
$x = 3$

$w_1 = 0.5$

$w_2 = -1$

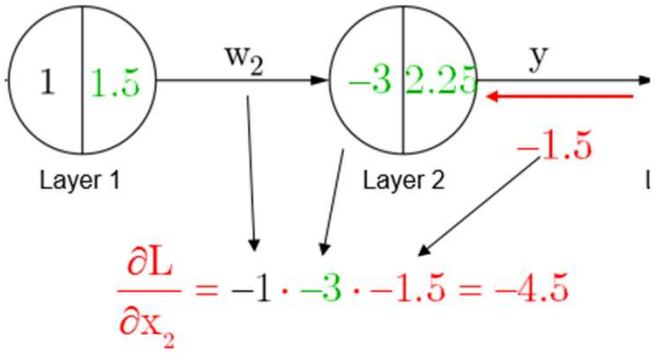
$\hat{y} = 3$

$$\frac{\partial L}{\partial w_1} = 3 \cdot 1 \cdot -4.5 = -13.5$$

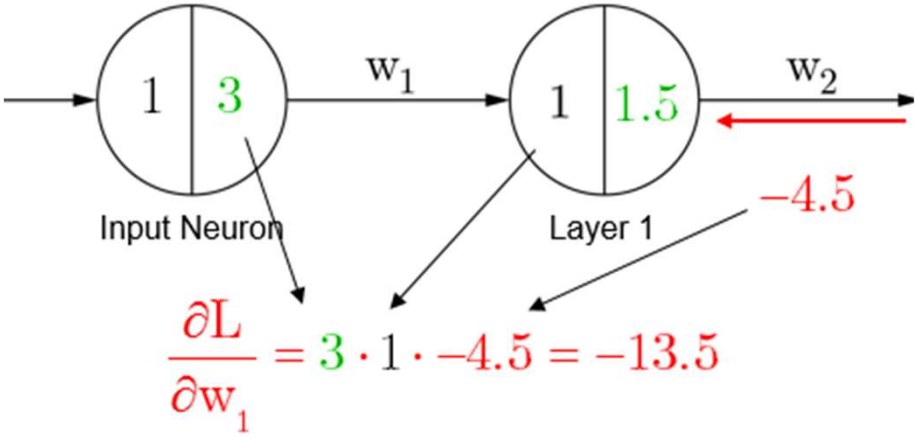


Additional Slides

To calculate dL/dw2 we need the upstream gradient dL/dx2. We just need to change “the other factor” in dL/dw2 with w2 instead of x2.



With this gradient it is possible to calculate dL/dw1 and therefore the change of w1.



Backpropagation

Neural Chain

we want:

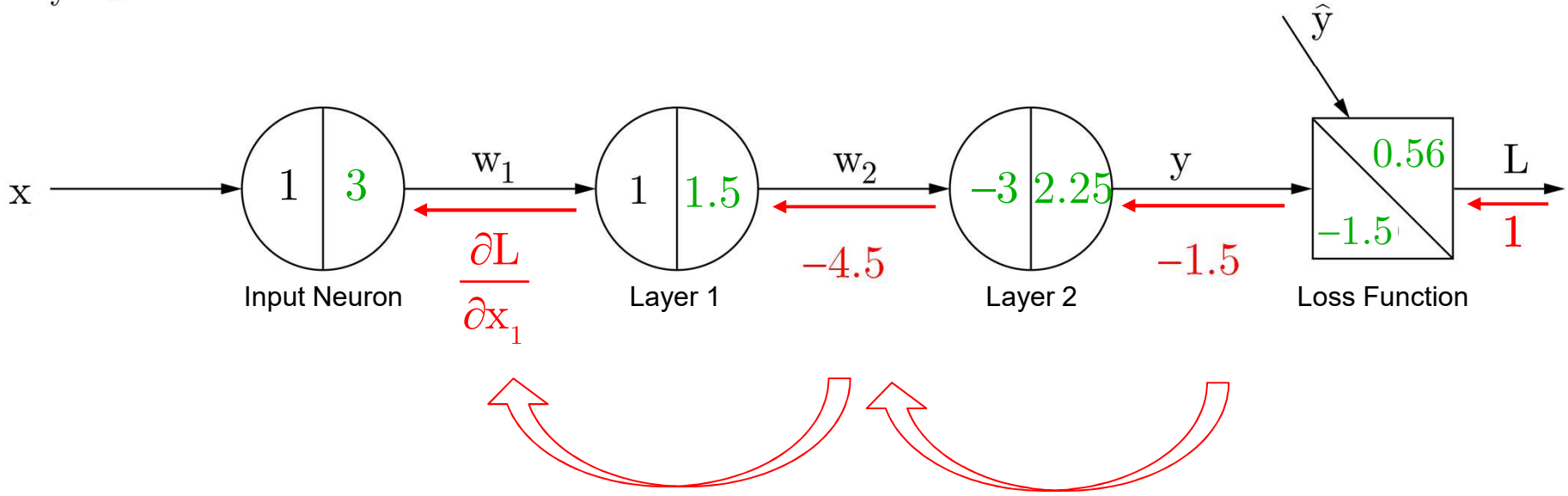
$$\Delta w_1 = -\alpha \frac{\partial L}{\partial w_1}, \Delta w_2 = -\alpha \frac{\partial L}{\partial w_2}$$

$x = 3$

$w_1 = 0.5$

$w_2 = -1$

$\hat{y} = 3$



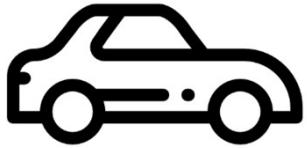
Backpropagation of the error to previous layers

Deep Neural Networks

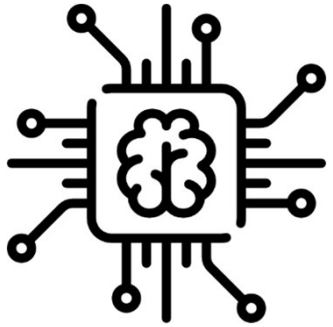
Johannes Betz / Prof. Dr. Markus Lienkamp / Prof. Dr. Boris Lohmann
(Jean-Michael Georg, M. Sc.)

Agenda

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



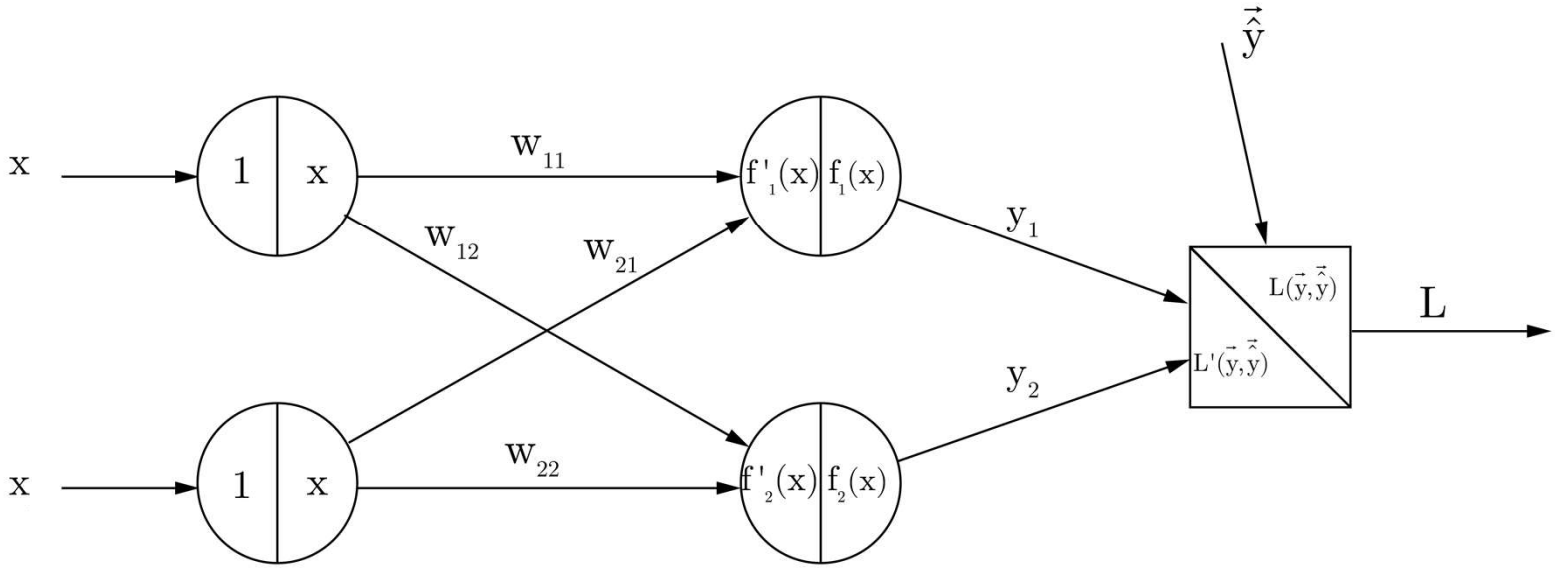
2. Chapter: Neuronal Networks
    - 2.1 Activation Functions
    - 2.2 Fully Connected Layer
    - 2.3 Batches
    - 2.3 Weight Inititalisation



3. Chapter: Overview

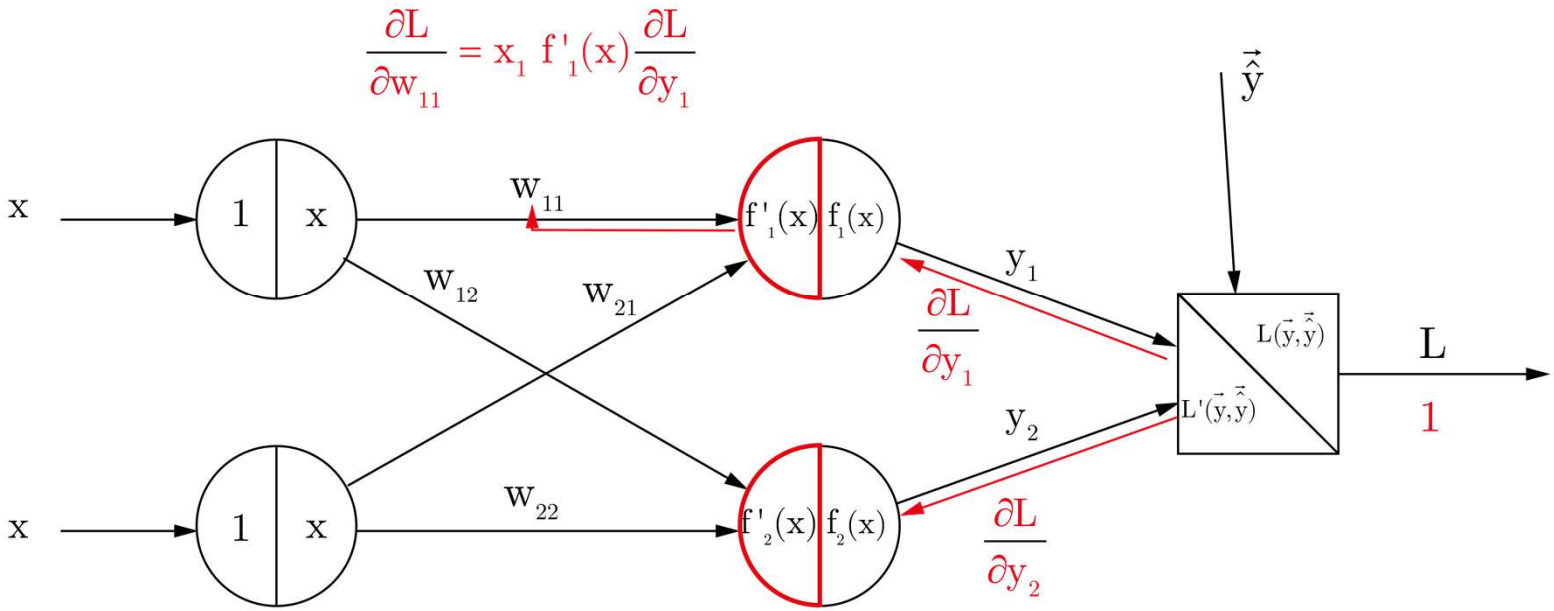
Backpropagation

Neural Network



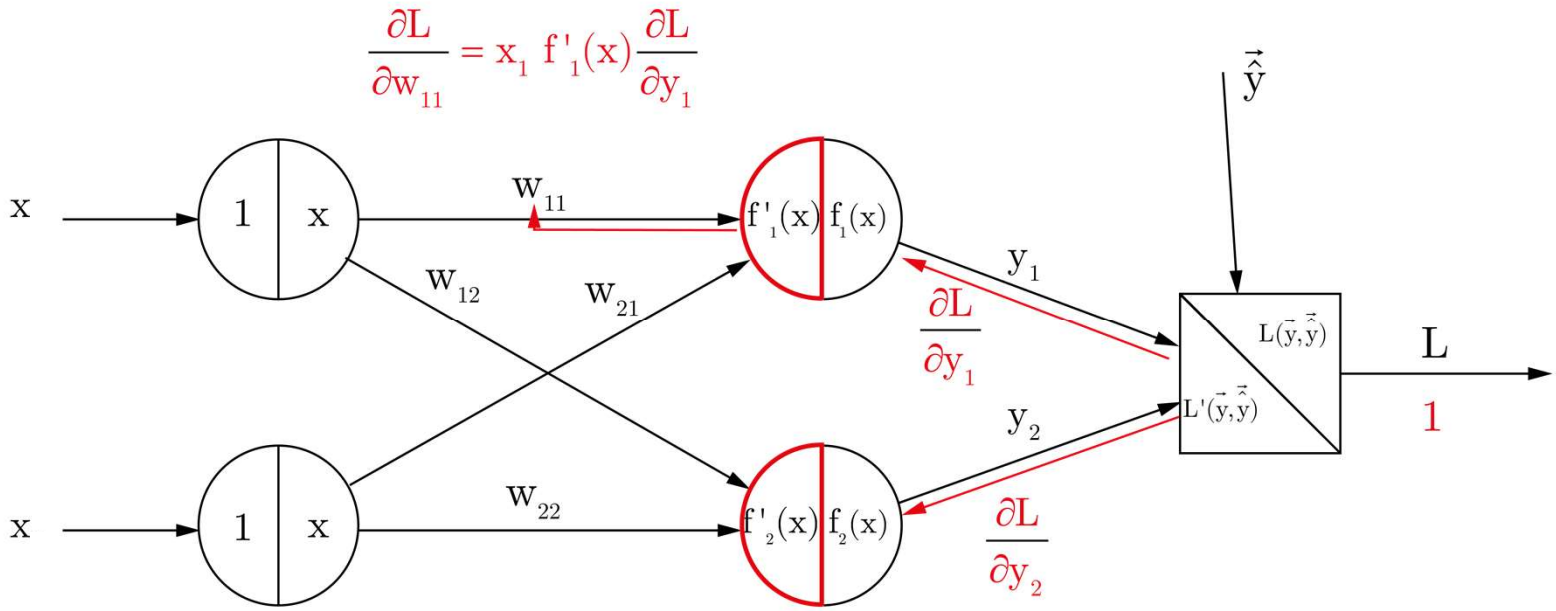
Backpropagation

Neural Network



Backpropagation

Neural Network



$$\frac{\partial L}{\partial w_{11}} = x_1 \, f'_1(x) \frac{\partial L}{\partial y_1} \qquad \frac{\partial L}{\partial w_{12}} = x_1 \, f'_2(x) \frac{\partial L}{\partial y_2} \qquad \frac{\partial L}{\partial w_{22}} = x_2 \, f'_2(x) \frac{\partial L}{\partial y_2} \qquad \frac{\partial L}{\partial w_{21}} = x_2 \, f'_1(x) \frac{\partial L}{\partial y_1}$$

Additional Slides

Backpropagation

Neural Network

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

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

$$W = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix}$$

$$\Rightarrow \Delta W = -\alpha \begin{pmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{12}} \\ \frac{\partial L}{\partial w_{21}} & \frac{\partial L}{\partial w_{22}} \end{pmatrix}$$

$$\Rightarrow \Delta W = -\alpha \begin{pmatrix} x_1 \frac{\partial L}{\partial y_1} f'_1 & x_1 \frac{\partial L}{\partial y_2} f'_2 \\ x_2 \frac{\partial L}{\partial y_1} f'_1 & x_2 \frac{\partial L}{\partial y_2} f'_2 \end{pmatrix}$$

Hadamard product

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \circ \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 1a & 2b \\ 3c & 4d \end{pmatrix}$$

$$\Rightarrow \Delta W = -\alpha \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \left[\begin{pmatrix} \frac{\partial L}{\partial y_1} \\ \frac{\partial L}{\partial y_2} \end{pmatrix} \circ \begin{pmatrix} f'_1 \\ f'_2 \end{pmatrix} \right]^T$$

Backpropagation

Neural Network

$$\Rightarrow \Delta W = -\alpha \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \left[\begin{pmatrix} \frac{\partial L}{\partial y_1} \\ \frac{\partial L}{\partial y_2} \end{pmatrix} \circ \begin{pmatrix} f'_1 \\ f'_2 \end{pmatrix} \right]^T = -\alpha \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \left[\begin{pmatrix} \frac{\partial L}{\partial y_1} f'_1 \\ \frac{\partial L}{\partial y_2} f'_2 \end{pmatrix} \right]^T$$

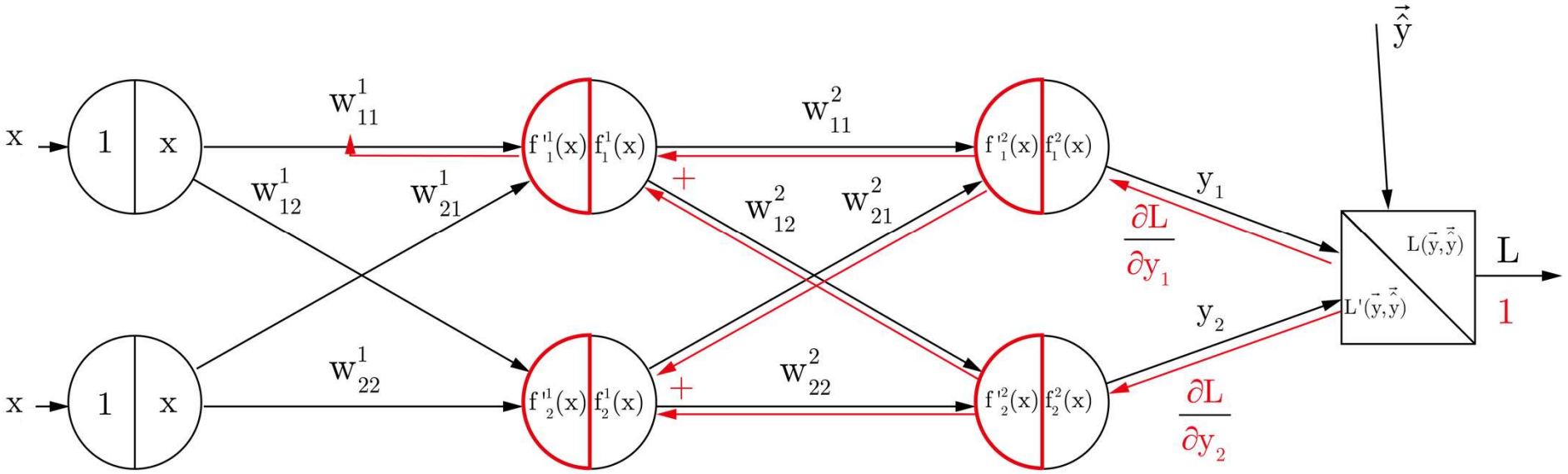
$$= -\alpha \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \begin{pmatrix} \frac{\partial L}{\partial y_1} f'_1 & \frac{\partial L}{\partial y_2} f'_2 \end{pmatrix} = -\alpha \begin{pmatrix} x_1 \frac{\partial L}{\partial y_1} f'_1 & x_1 \frac{\partial L}{\partial y_2} f'_2 \\ x_2 \frac{\partial L}{\partial y_1} f'_1 & x_2 \frac{\partial L}{\partial y_2} f'_2 \end{pmatrix}$$

Additional Slides

For neural networks the change of weights can be simply calculate with a vector operations

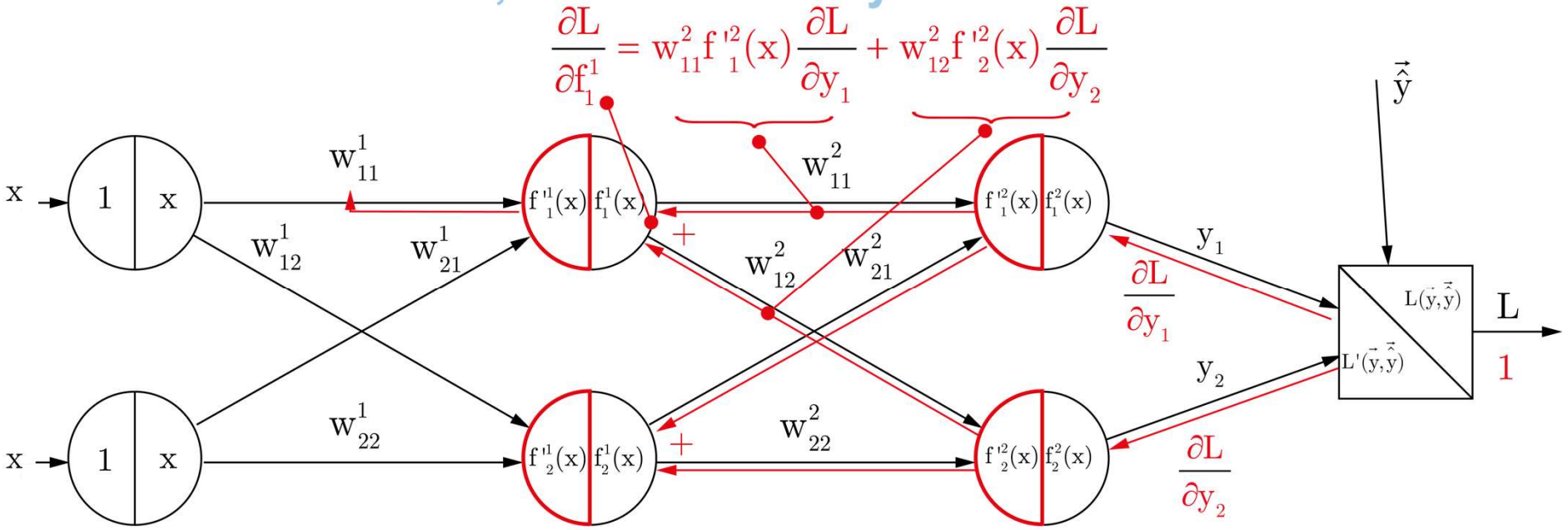
Backpropagation

Neural Network, one hidden layer



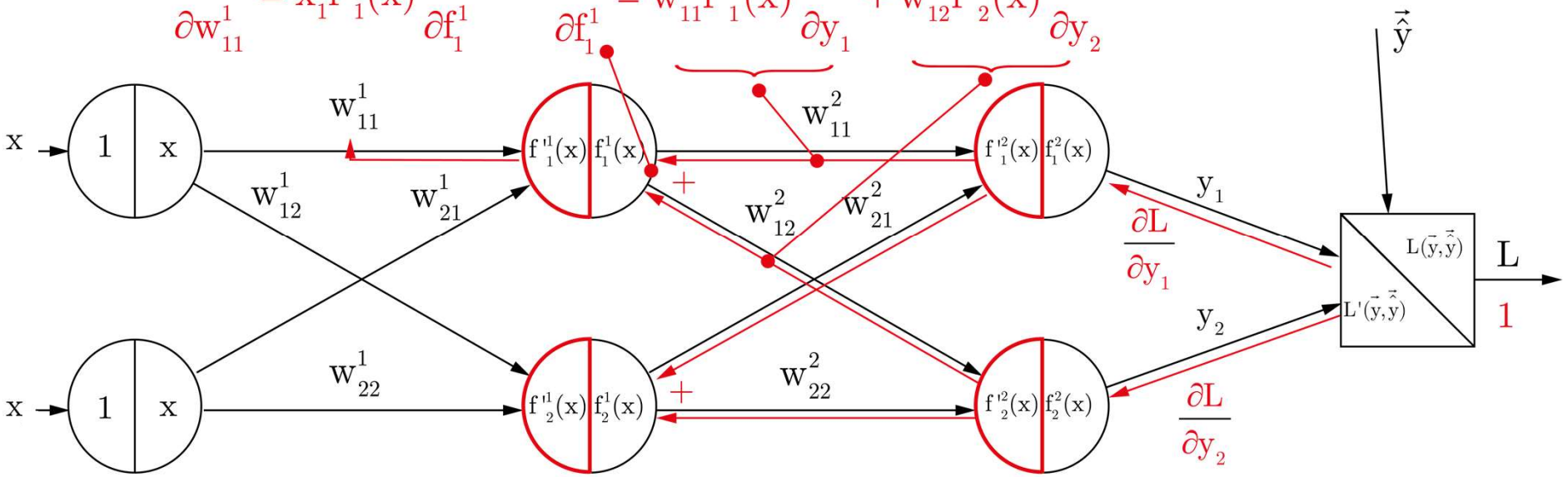
Backpropagation

Neural Network, one hidden layer



Backpropagation

Neural Network, one hidden layer



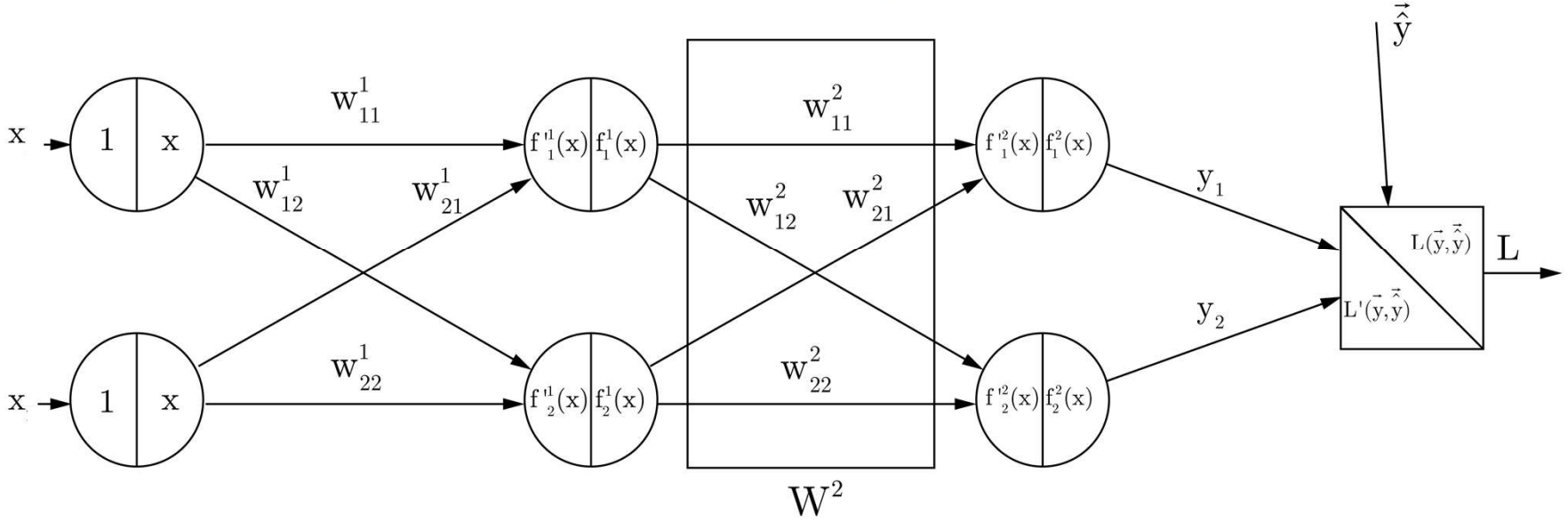
$$\frac{\partial L}{\partial w_{11}^1} = x_1 f'^1_1(x) \frac{\partial L}{\partial f_1^1}, \qquad \frac{\partial L}{\partial w_{12}^1} = x_1 f'^1_2(x) \frac{\partial L}{\partial f_2^1}, \qquad \frac{\partial L}{\partial w_{21}^1} = x_2 f'^1_1(x) \frac{\partial L}{\partial f_1^1}, \qquad \frac{\partial L}{\partial w_{22}^1} = x_2 f'^1_2(x) \frac{\partial L}{\partial f_2^1}$$

Additional Slides

When two upstream gradients arrive at a neuron, the resulting gradient is the sum of both gradients.

Backpropagation

Neural Network, one hidden layer



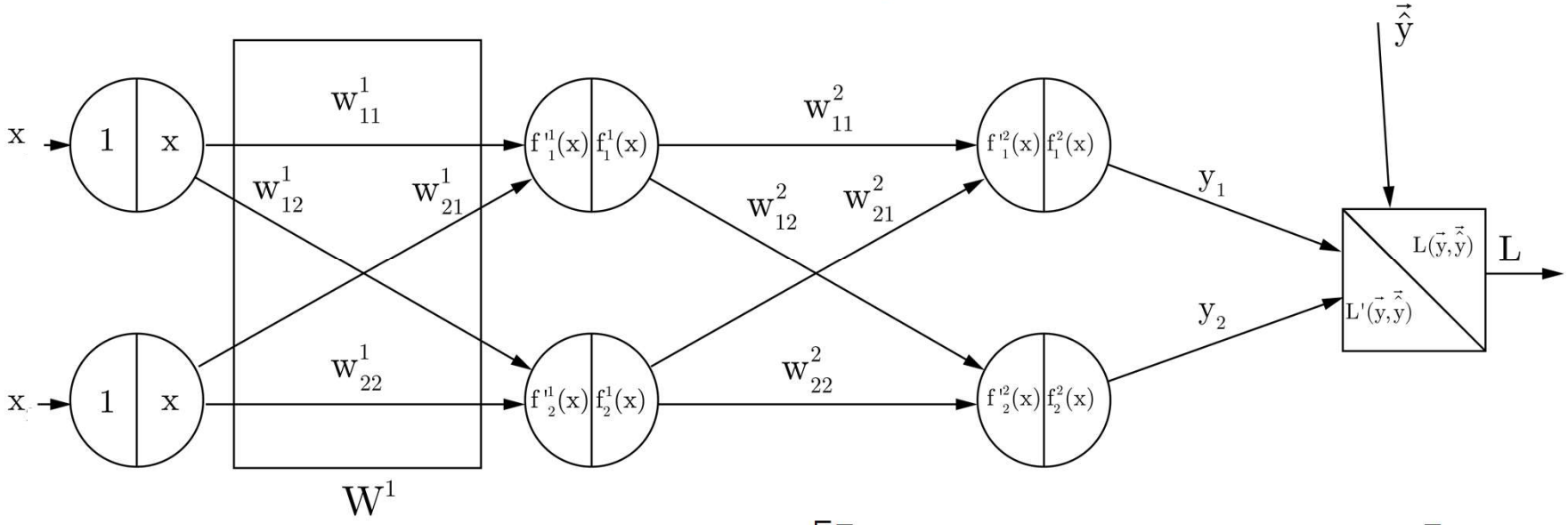
$$\Rightarrow \Delta W^2 = -\alpha \begin{pmatrix} f^1_1 \\ f^1_2 \end{pmatrix} \left[\begin{pmatrix} \frac{\partial L}{\partial y_1} \\ \frac{\partial L}{\partial y_2} \end{pmatrix} \circ \begin{pmatrix} f'^2_1 \\ f'^2_2 \end{pmatrix} \right]^T$$

Forward path

Backward path

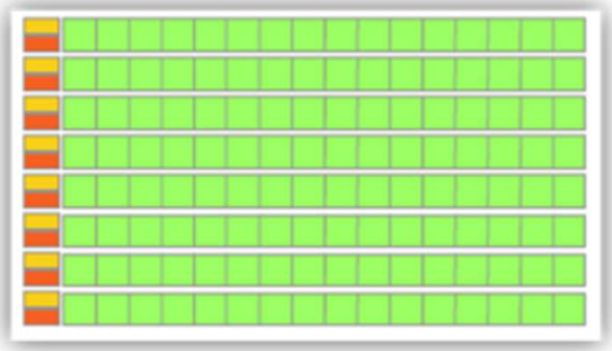
Backpropagation

Neural Network, one hidden layer



$$\Rightarrow \Delta W^1 = -\alpha \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \left[\begin{pmatrix} \frac{\partial L}{\partial f^1_1} \\ \frac{\partial L}{\partial f^1_2} \end{pmatrix} \circ \begin{pmatrix} f'^1_1 \\ f'^1_2 \end{pmatrix} \right]^T = -\alpha \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \left[\begin{pmatrix} w^2_{11} & w^2_{12} \\ w^2_{21} & w^2_{22} \end{pmatrix} \left[\begin{pmatrix} \frac{\partial L}{\partial y_1} \\ \frac{\partial L}{\partial y_2} \end{pmatrix} \circ \begin{pmatrix} f'^2_1 \\ f'^2_2 \end{pmatrix} \right]^T \right] \circ \begin{pmatrix} f'^1_1 \\ f'^1_2 \end{pmatrix} \Bigg]^T$$

CPU vs GPU

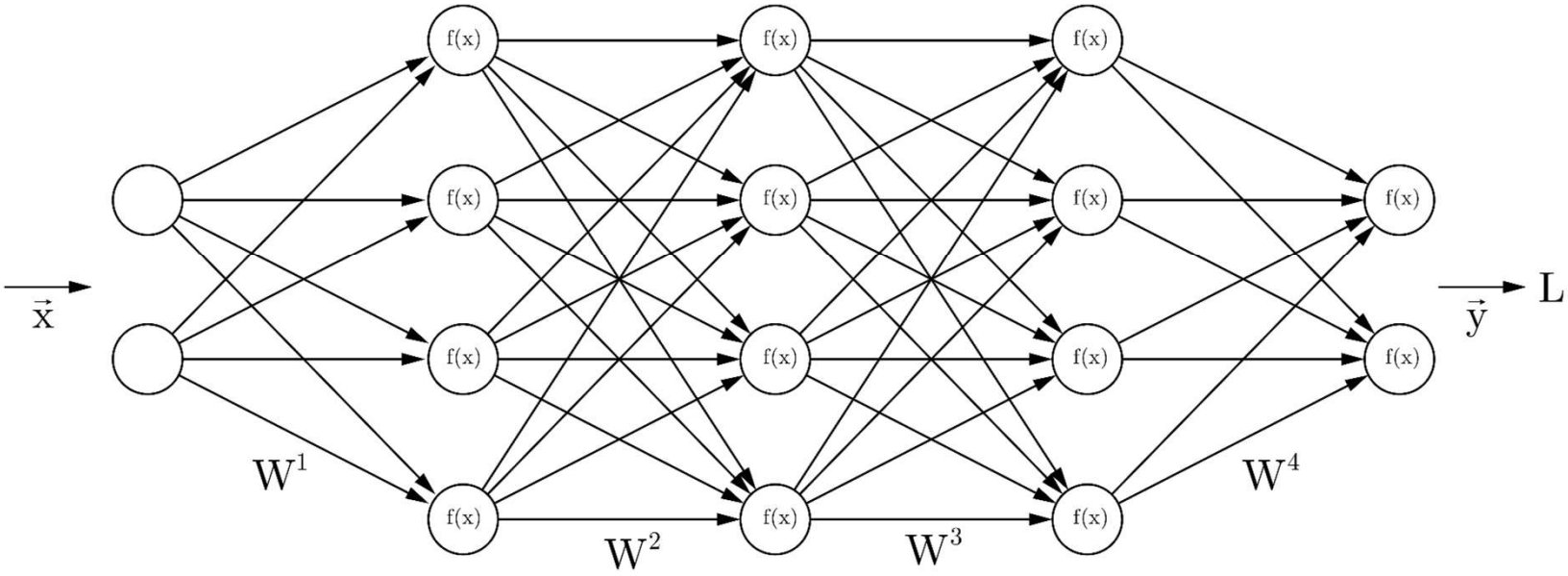


- Low computedensity
- Complex logic control
- Large Cache
- Low latencytollerance

- High computedensity
- Many calculation per memory acces
- Optimizer for parallel computing
- Low latency tolerance

Backpropagation

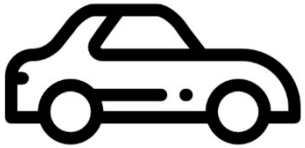
Neural Network

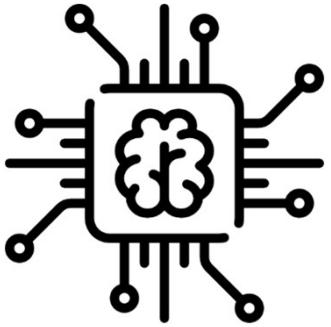


Deep Neural Networks

Johannes Betz / Prof. Dr. Markus Lienkamp / Prof. Dr. Boris Lohmann
(Jean-Michael Georg, M. Sc.)

Agenda



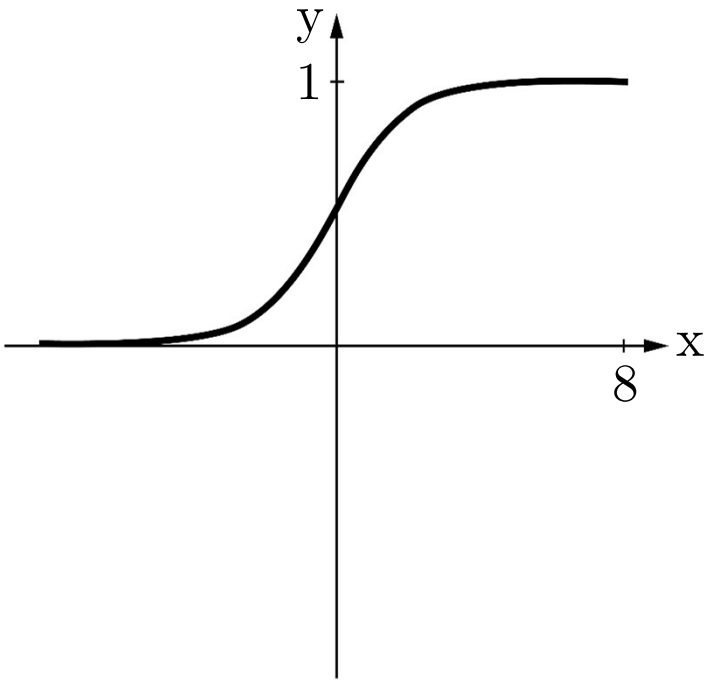


Activation Functions

Sigmoid

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

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



Two main problems:

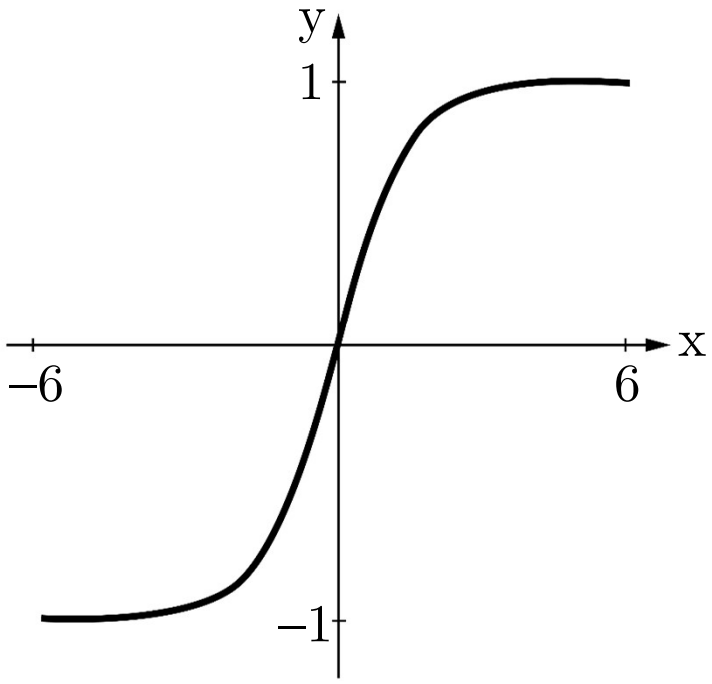
- Causes vanishing gradient: Gradient nearly zero for very large or small x, kills gradient and network stops learning
- Output isn‘t zero centered: Always all gradients positive or all negativ, inefficient weight updates

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:

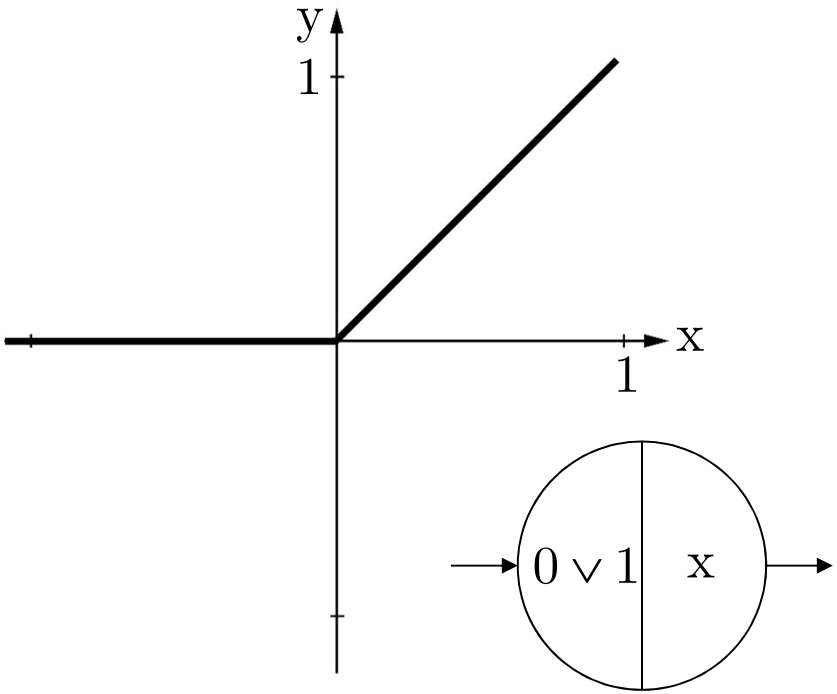
- Output is zero centered
- But still causes vanishing gradient

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:

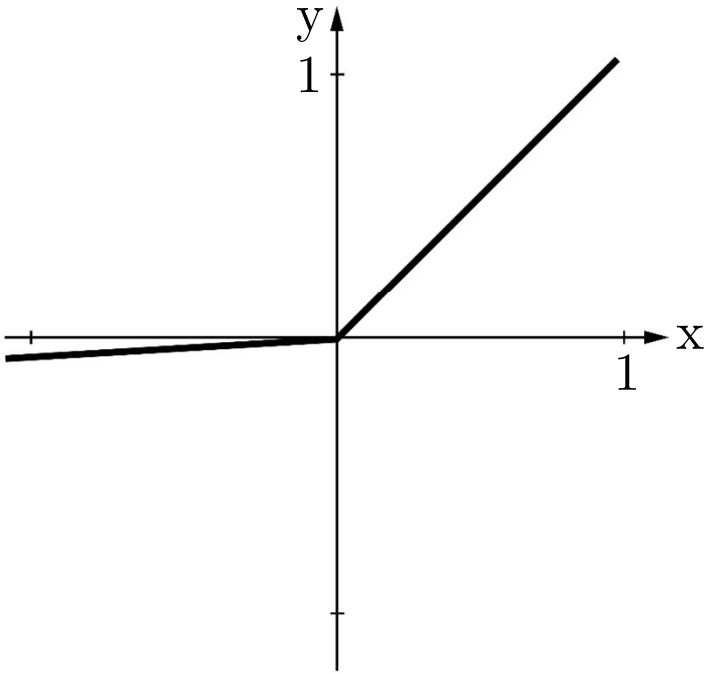
- Mostly used in hidden layers
- Positive bias at init to get active ReLU

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:

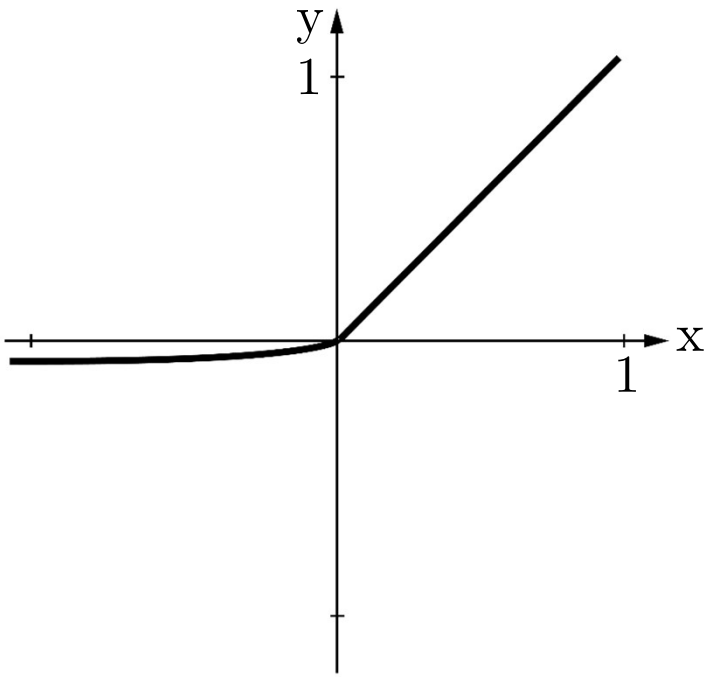
- Removes zero part of ReLU by adding a small slope. More stable then relu, but adds another paramter
- Computationally efficient
- Converges very fast
- Doesn‘t die
- Parameter a can also be learned by the network

Activation Functions

Exponential Linea Unit (ELU)

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

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



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

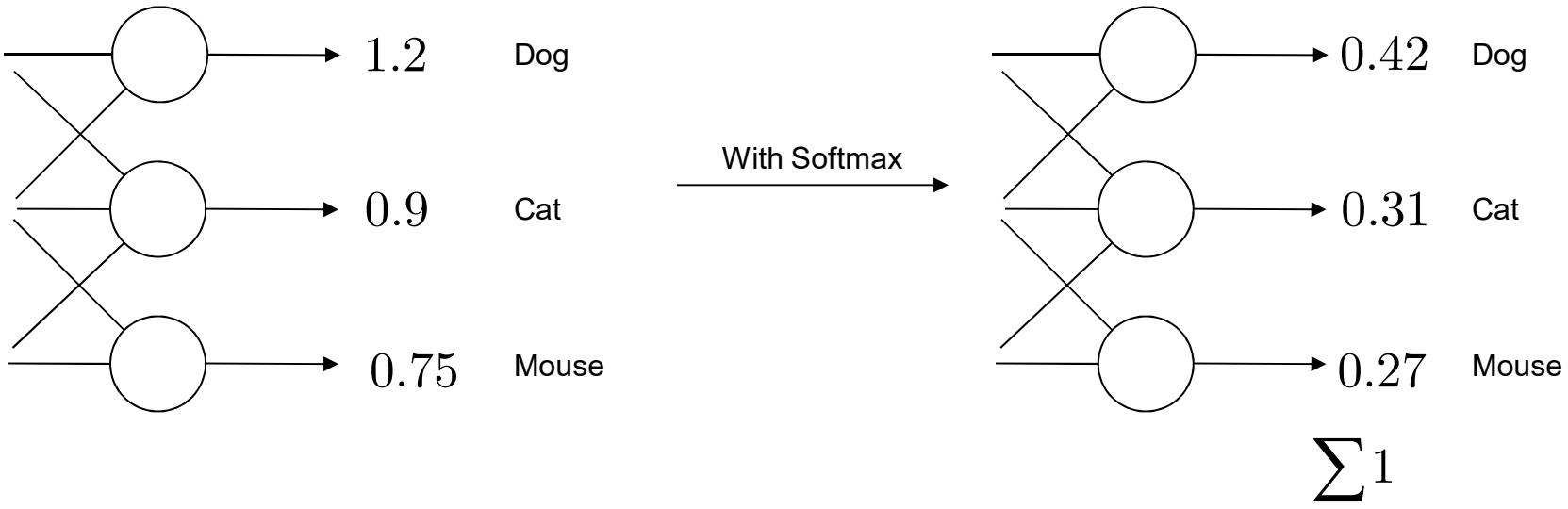
Activation Functions

Softmax

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

- Type of Sigmoid, handy for classification problems.
- Divides by the sum of all outputs, allows for percentage representation

Classifier:



Activation Functions

Rule of thumb

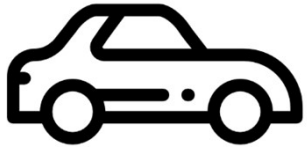
- Sigmoid / Relu + Softmax for classifiers
- Sigmoid, tanh sometimes avoided due to vanishing gradient
- ReLU mostly used today
- Start with ReLU if you don‘t get optimal results go for LeakyReLU od ELU

Deep Neural Networks

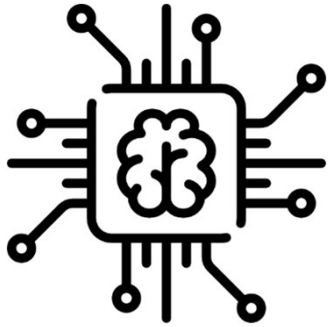
Johannes Betz / Prof. Dr. Markus Lienkamp / Prof. Dr. Boris Lohmann
(Jean-Michael Georg, M. Sc.)

Agenda

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



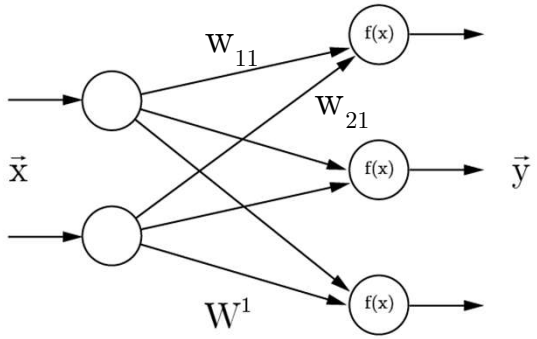
2. Chapter: Neuronal Networks
    - 2.1 Activation Functions
    - 2.2 Fully Connected Layer
    - 2.3 Batches
    - 2.3 Weight Inititalisation



3. Chapter: Overview

Fully Connected Layer

Dimensions



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

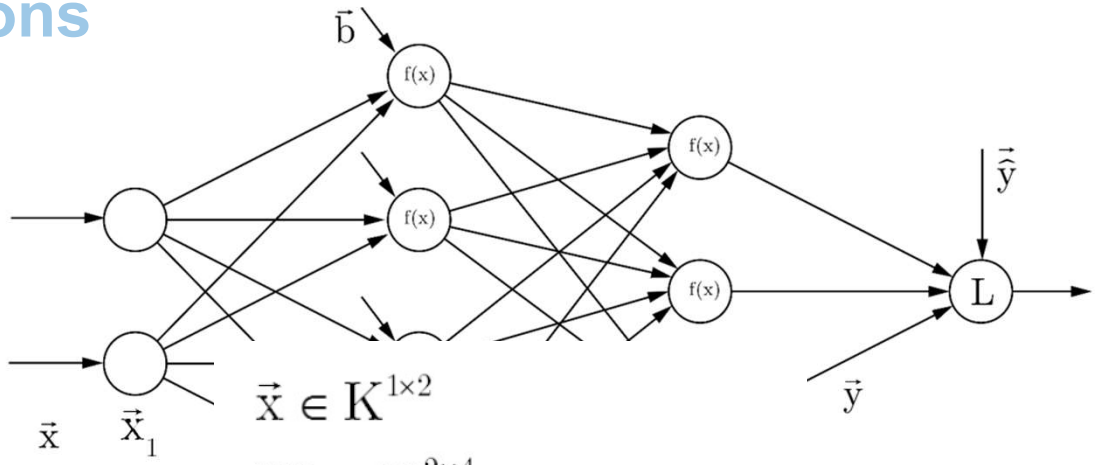
$$= f\left(\begin{pmatrix} x_1 w_{11} + x_2 w_{21} \\ x_1 w_{12} + x_2 w_{22} \\ x_1 w_{13} + x_2 w_{23} \end{pmatrix} + \vec{b}\right) = f\left(\begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{23} & w_{33} \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \vec{b}\right) = f\left(W^1\vec{x} + \vec{b}\right)$$

Alternativ:

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

Fully Connected Layer

Dimensions



$$\vec{x} \in K^{1\times 2}$$

$$W_1 \in K^{2\times 4}$$

$$\vec{b} \in K^{1\times 4}$$

$$f(\vec{x}, W^1, \vec{b}_1) = \vec{x}_2 \in K^{1\times 4}$$

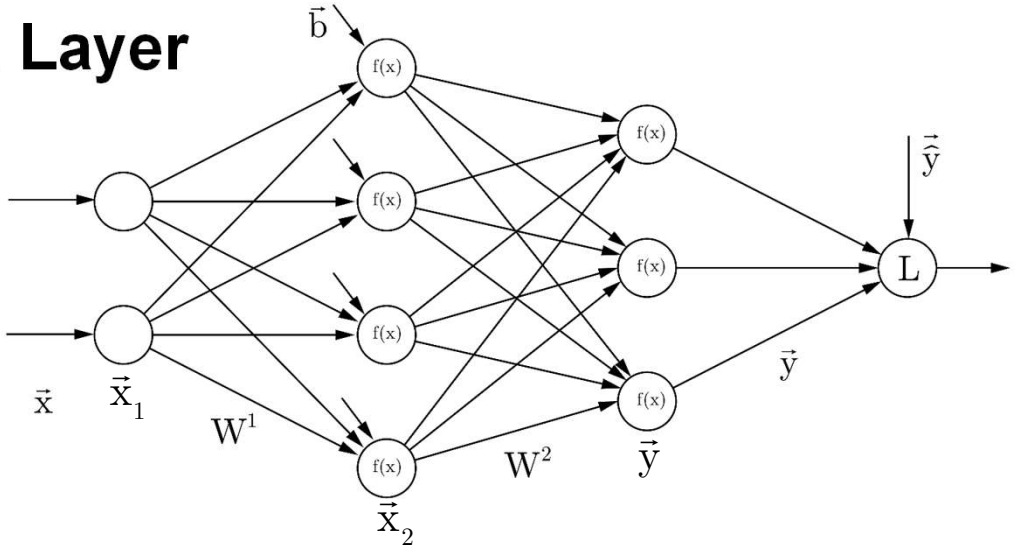
$$\vec{x}_2 = f\left(\underbrace{\begin{pmatrix} x_1 & x_2 \end{pmatrix}}_{\vec{x}} \cdot \underbrace{\begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{pmatrix}}_{W^1} + \vec{b}\right)$$

Here:

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

Fully Connected Layer

Dimensions



$$\vec{x} \in K^{1\times 2}$$

$$W_1 \in K^{2\times 4}$$

$$\vec{b} \in K^{1\times 4}$$

$$f(\vec{x}, W^1, \vec{b}_1) = \vec{x}_2 \in K^{1\times 4}$$

Here :

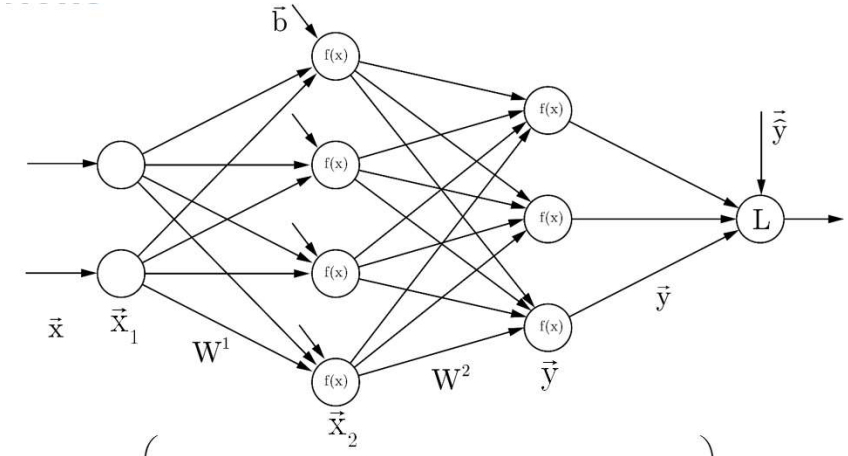
$$\vec{x} \cdot W^1$$

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

Since the weight change can be described as matrix operations, also the forward path can be described in matrix operations. For an input vector x, output vector y and a weigtmatrix in a fully connected layer the following formula can be derived:

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

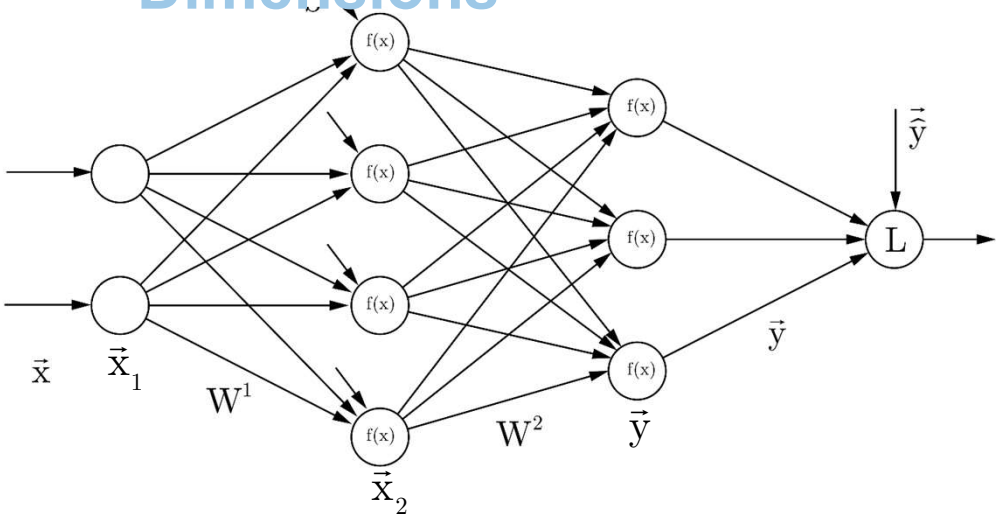
Where the dimensions of W impact the dimensions are n x m, where n is the number of input neurons and m the number of output neurons.



$$\vec{x}_2 = f\left(\underbrace{\begin{pmatrix} x_1 & x_2 \end{pmatrix}}_{\vec{x}} \cdot \underbrace{\begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{pmatrix}}_{W^1} + \vec{b}\right)$$

Fully Connected Layer

Dimensions



$$\vec{x} \in K^{1\times 2}$$

$$W_1 \in K^{2\times 4}$$

$$\vec{b} \in K^{1\times 4}$$

$$f(\vec{x}, W^1, \vec{b}_1) = \vec{x}_2 \in K^{1\times 4}$$

$$\vec{x}_2 = f\left(\underbrace{\begin{pmatrix} x_1 & x_2 \end{pmatrix}}_{\vec{x}} \cdot \underbrace{\begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{pmatrix}}_{W^1} + \vec{b}\right)$$

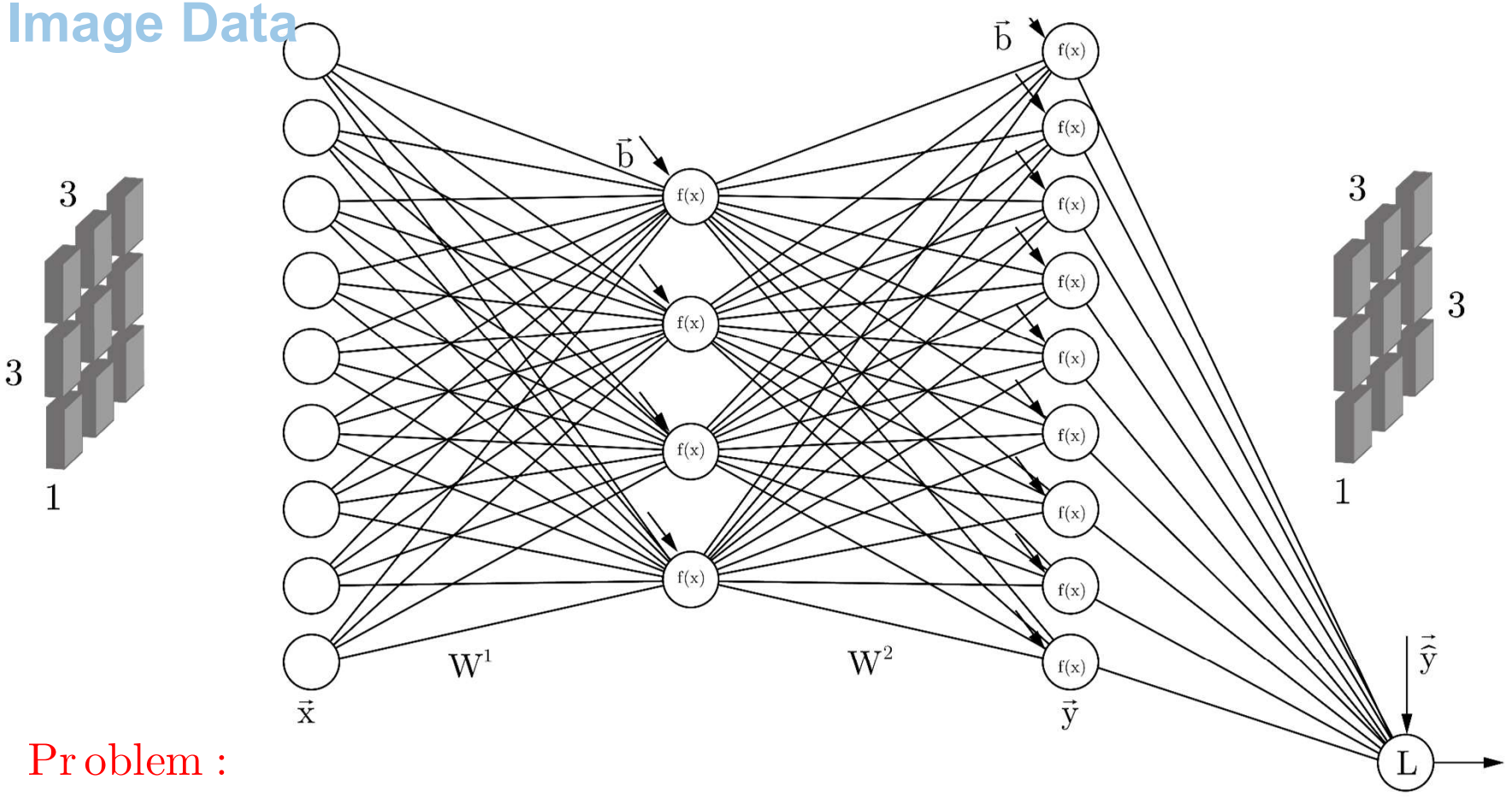
Here :

$$\vec{x} \cdot W^1$$

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

Fully Connected Layer

Image Data

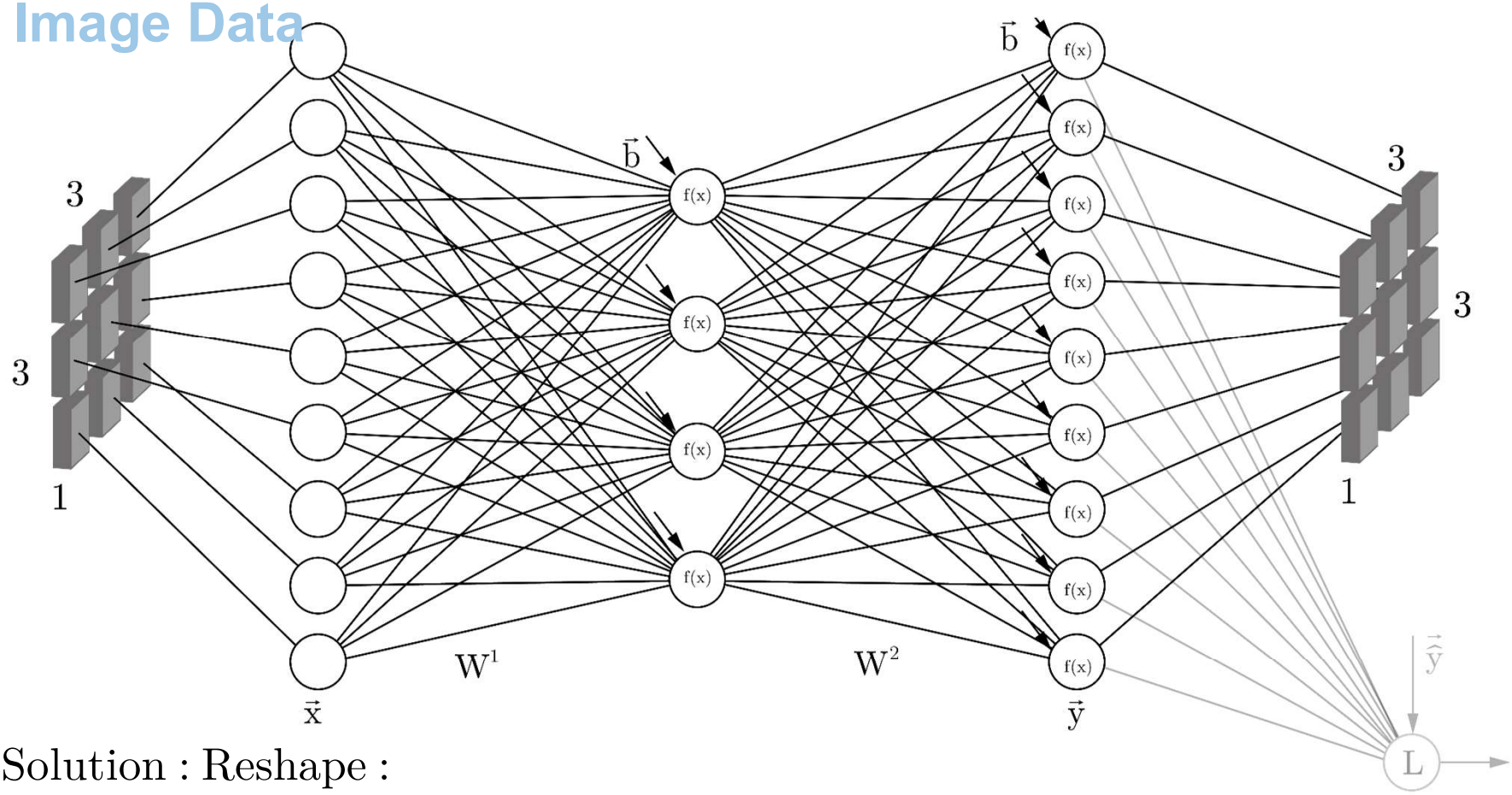


Problem :

$(1 \times 3 \times 3) \cdot (9 \times 4)$

Fully Connected Layer

Image Data



Solution : Reshape :

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

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

Additional Slides

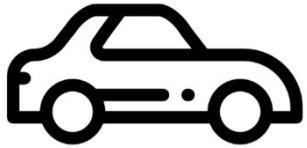
Since images are 2D but fully connected layer require a 1D vector, the image pixels are stacked to get a vector representation of the image

Deep Neural Networks

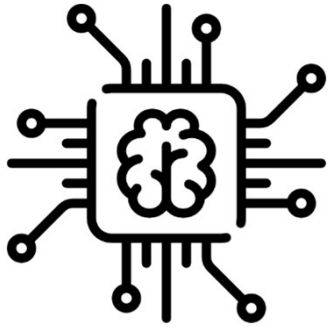
Johannes Betz / Prof. Dr. Markus Lienkamp / Prof. Dr. Boris Lohmann
(Jean-Michael Georg, M. Sc.)

Agenda

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

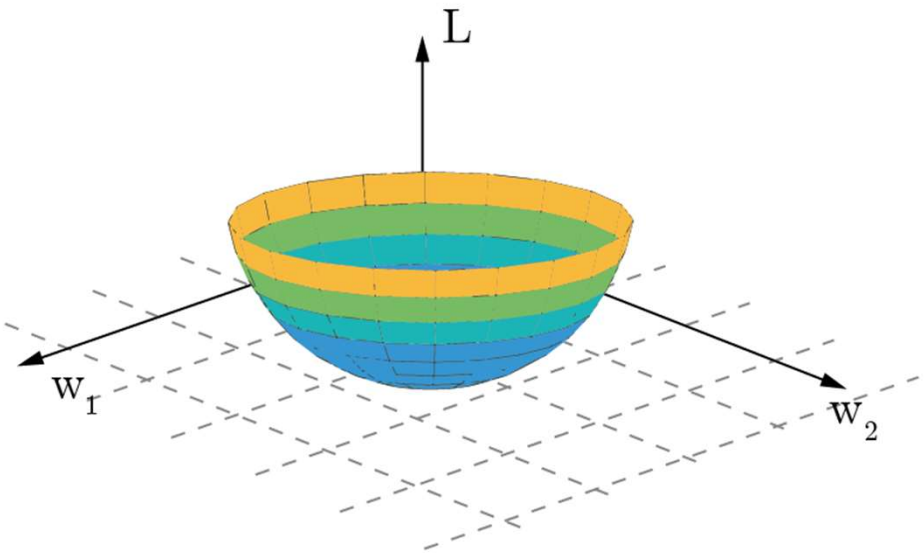


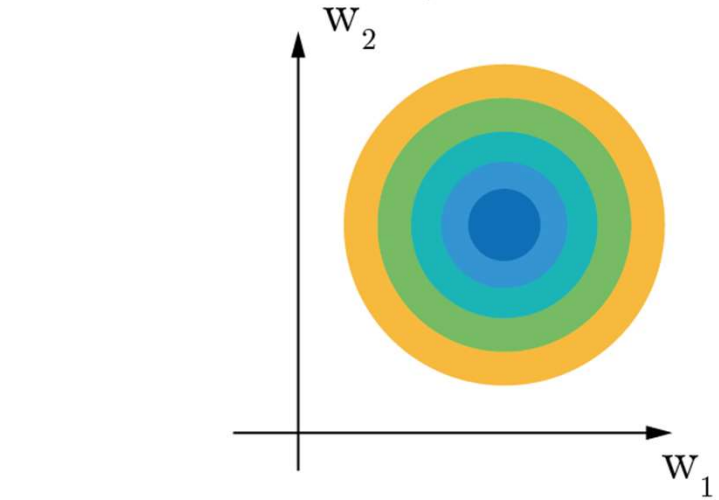
2. Chapter: Neuronal Networks
    - 2.1 Activation Functions
    - 2.2 Fully Connected Layer
    - 2.3 Batches
    - 2.3 Weight Inititalisation

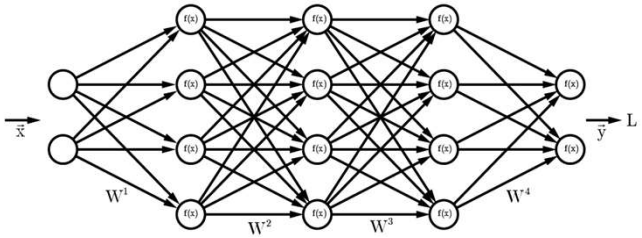


3. Chapter: Overview

Gradient Descent





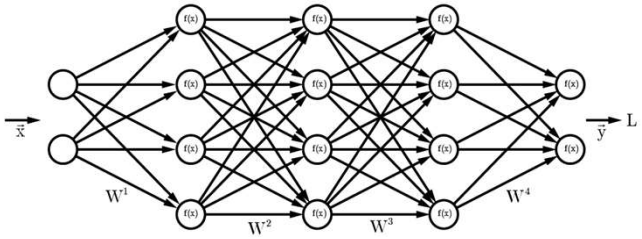


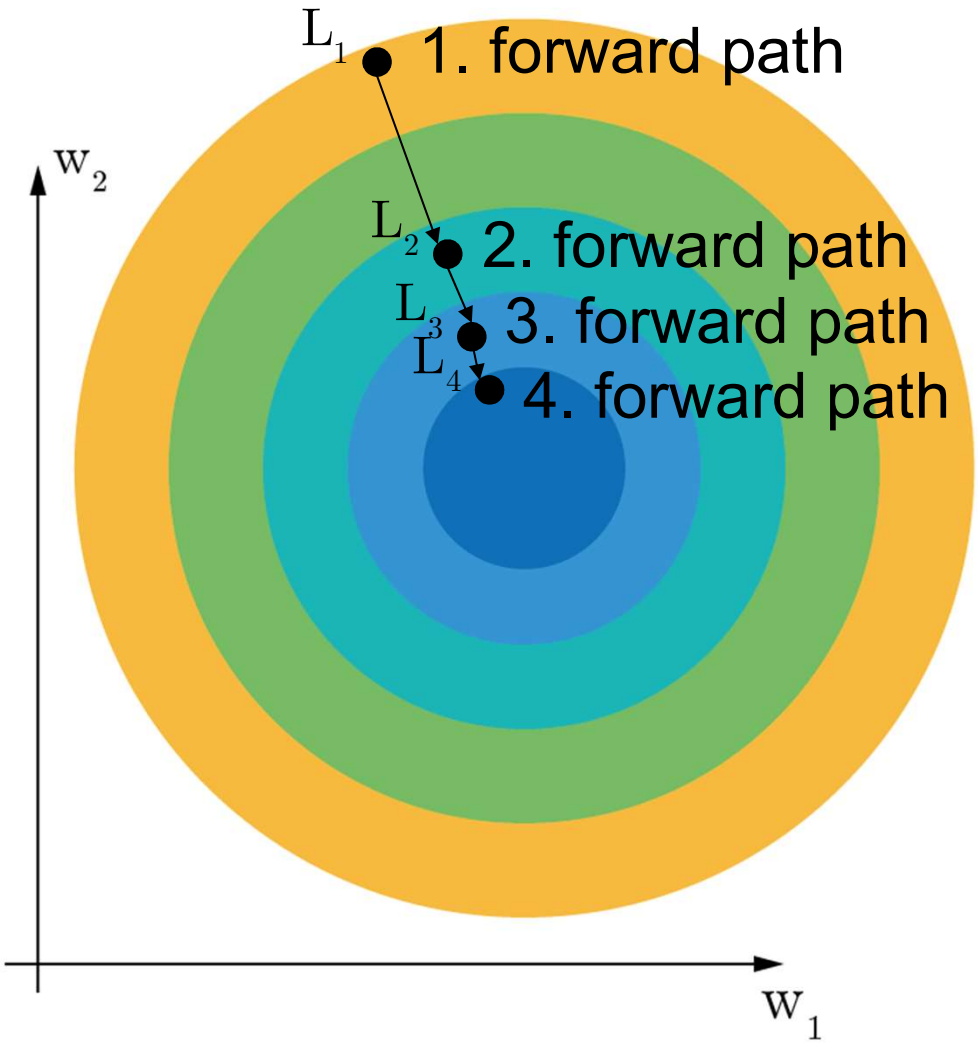
$$L(X) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Training:

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

Gradient Descent





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

All the data

$$-\alpha \nabla L_W$$

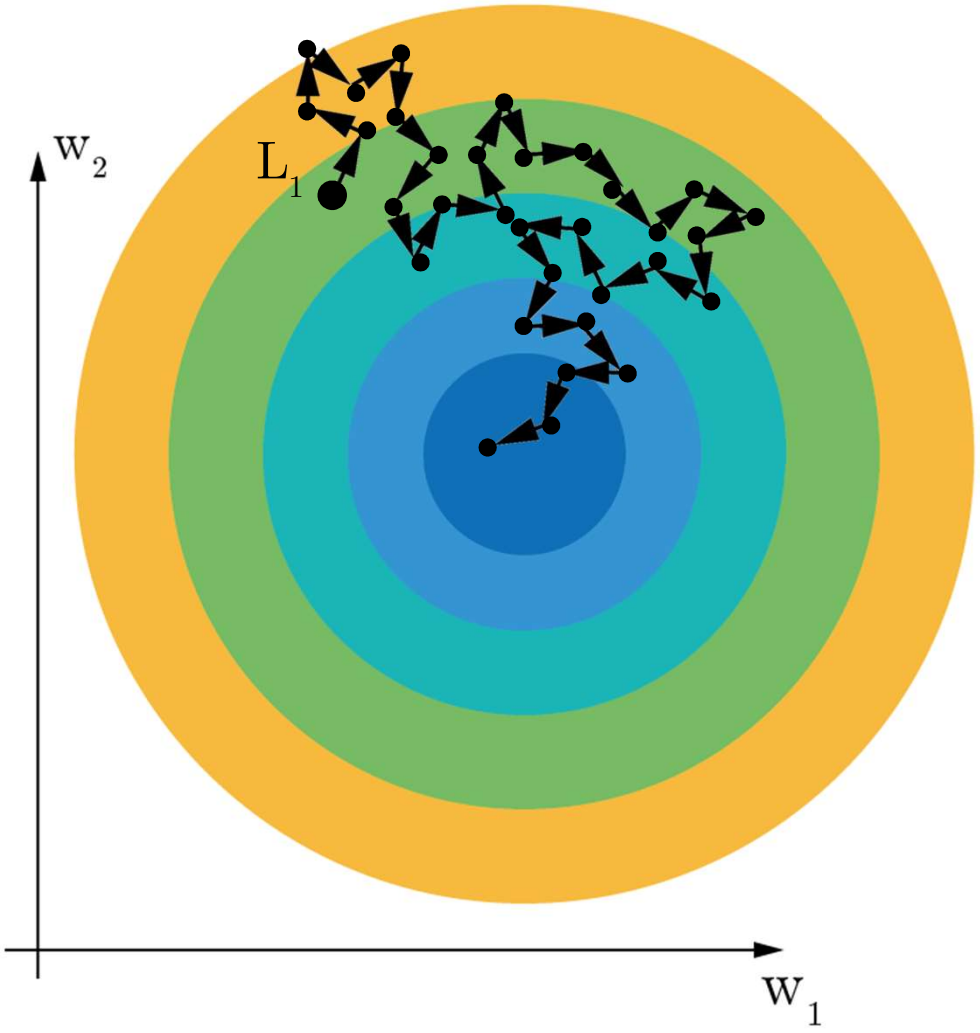
3x compute time

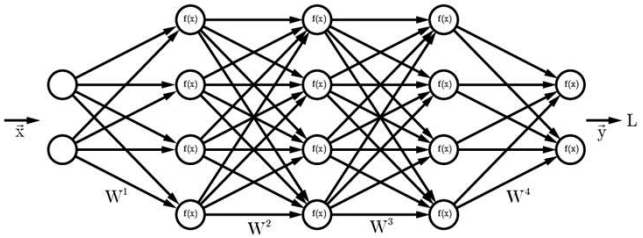
Additional Slides

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 Descent

Alternative to Gradient Descent





Only use one random datapoint at a time

$\Rightarrow$ Reduces compute time per optimisation step

$\Rightarrow$ But finding local minimums takes longer

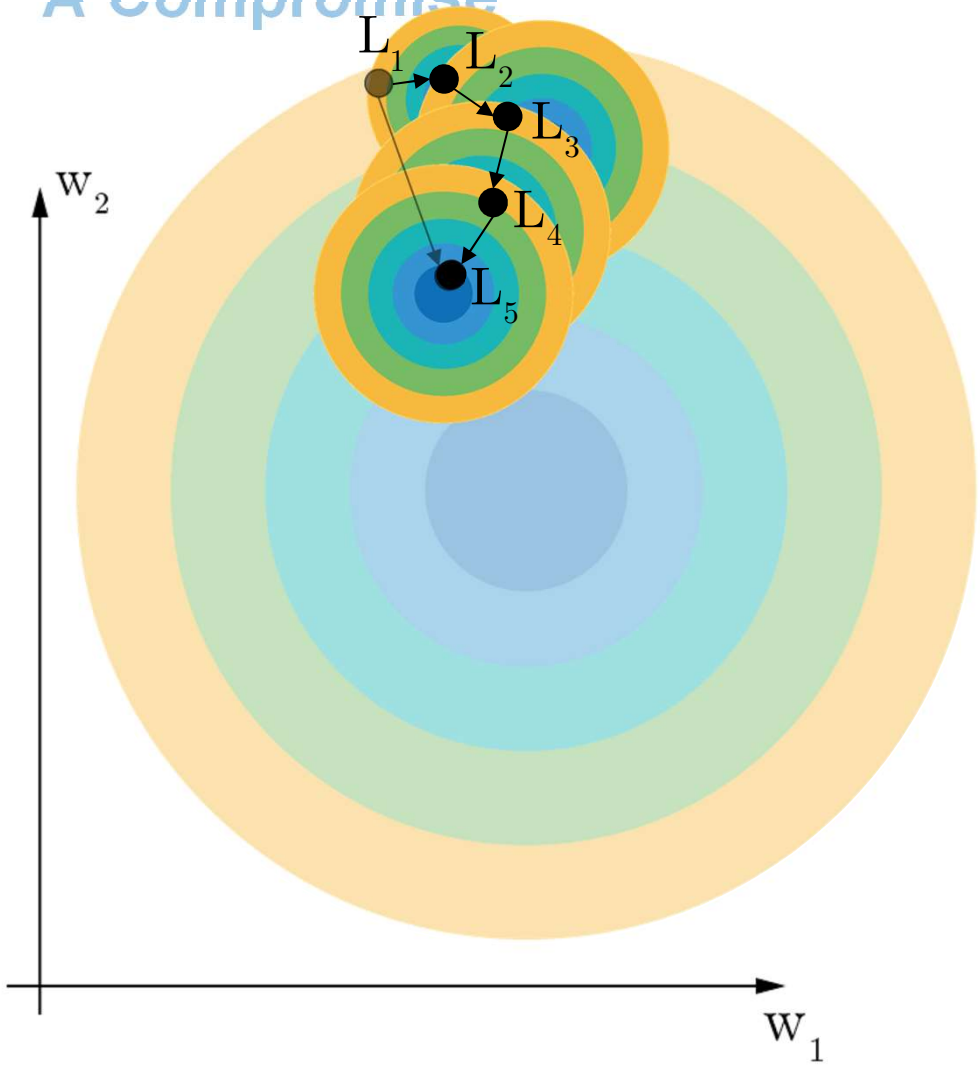
$$L(X) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

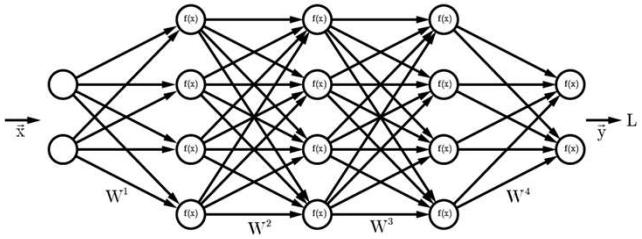
$$n = 1$$

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

Batch Gradient Descent

A Compromise





$$L(X) = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

Split dataset in batches,

=> Trying to minimize global loss function with local cost functions

Additional Slides

The problem is, usually in machine learning it is not possible to train over the whole data set, therefore small subsets of dataset are used to calculate the new change of weights. The gradient descent then only applies to those small subsets, which may result in a change of weights which doesn’t change to global loss towards the global minimum but toward the minimum of the subset.

For stochastic gradient decent the subset size is one.

Increasing the subset size results to batch gradient descent.

Dimensions with batch gradient descent

$$\vec{x}_2 = f\left(\underbrace{\begin{pmatrix} x_1 & x_2 \end{pmatrix}}_{\vec{x}} \cdot \underbrace{\begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{pmatrix}}_{W^1} + \vec{b}\right)$$

$$\vec{x} \in K^{1\times 2}$$
$$W_1 \in K^{2\times 4}$$
$$\vec{b} \in K^{1\times 4}$$

For batchsize n:

$$\begin{matrix} \vec{x}_{2,1} \\ \vec{x}_{2,2} \\ \vdots \\ \vec{x}_{2,n} \end{matrix} = f\left(\underbrace{\begin{matrix} \begin{pmatrix} x_1 & x_2 \end{pmatrix}_1 \\ \begin{pmatrix} x_1 & x_2 \end{pmatrix}_2 \\ \vdots \\ \begin{pmatrix} x_1 & x_2 \end{pmatrix}_n \end{matrix}}_{\vec{x}} \cdot \underbrace{\begin{matrix} \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{pmatrix}_1 \\ \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{pmatrix}_2 \\ \vdots \\ \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{pmatrix}_n \end{matrix}}_{W^1} + \begin{matrix} \vec{b}_1 \\ \vec{b}_2 \\ \vdots \\ \vec{b}_n \end{matrix}\right)$$

$$\vec{x} \in K^{n\times 1\times 2}$$
$$W_1 \in K^{n\times 2\times 4}$$
$$\vec{b} \in K^{n\times 1\times 4}$$

3D Matrix multiplcation
-> handled by tensorflow

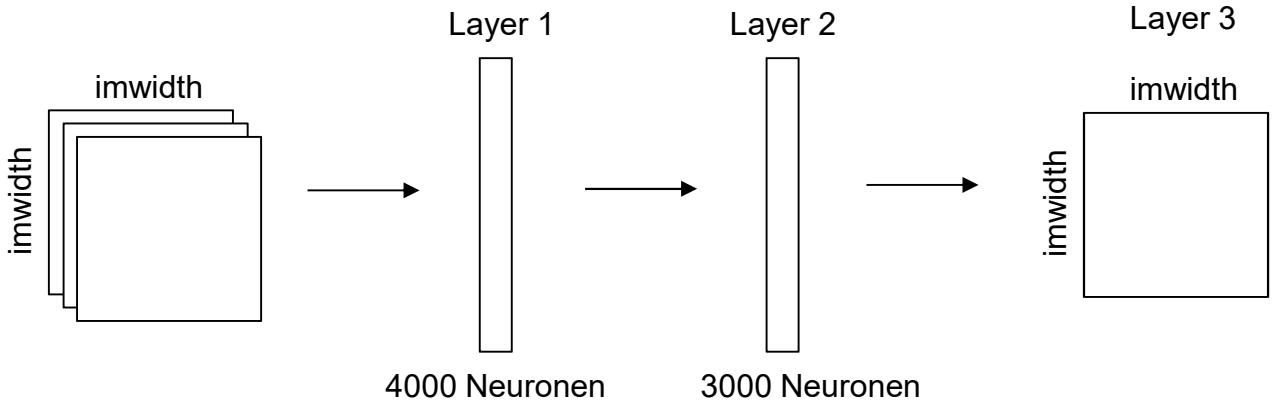
Additional Slides

In batch gradient descent the matrix operations happening inside the network are basically stacked on top of each other, resulting in 3d matrix operations. This all happens inside the computer and is usually handlet by the deep learning frame framework.

Stochastic vs. Batch

- Stochastic training can miss local minimums because of the randomness of each input
- Takes longer in general
- Mini batches finds minimum quickly but may take more computational power

Codeexample Tensorflow:

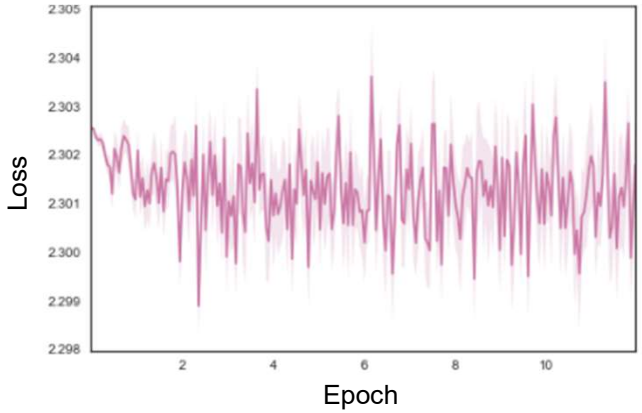



```
with tf.name_scope("Network"):
    with tf.name_scope("Layer1"):
        w1 = tf.Variable(tf.random_normal([batchsize,imwidth*imwidth*3,4000],mean = 0.0, stddev = 0.01))
        b1 = tf.Variable(tf.random_normal([batchsize,1,4000],mean = 0.0, stddev = 0.01))
        a1 = tf.matmul(x,w1)
        l1 = tf.nn.relu(tf.add(a1,b1))
    with tf.name_scope("Layer2"):
        w2 = tf.Variable(tf.random_normal([batchsize,4000,3000],mean = 0.0, stddev = 0.01))
        b2 = tf.Variable(tf.random_normal([batchsize,1,3000],mean = 0.0, stddev = 0.01))
        a2 = tf.matmul(l1,w2)
        l2 = tf.nn.relu(tf.add(a2,b2))
    with tf.name_scope("Layer3"):
        w3 = tf.Variable(tf.random_normal([batchsize,3000,imwidth*imwidth],mean = 0.0, stddev = 0.01))
        b3 = tf.Variable(tf.random_normal([batchsize,1,imwidth*imwidth],mean = 0.0, stddev = 0.01))
        a3 = tf.matmul(l2,w3)
        y_out = tf.sigmoid(tf.add(a3,b3))
```

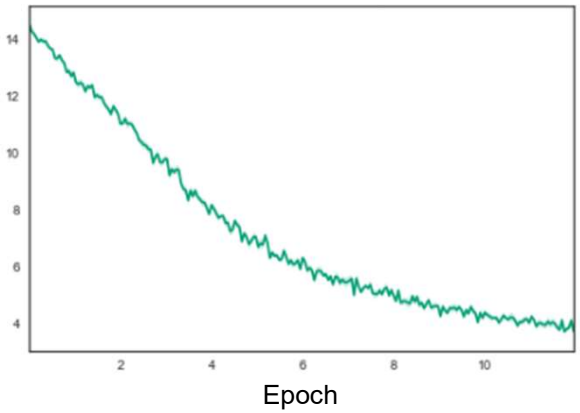
Weight Initialisation

$$W_1 = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{pmatrix}$$

- **Init as 0**
- **z.B.:** $W_1 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$



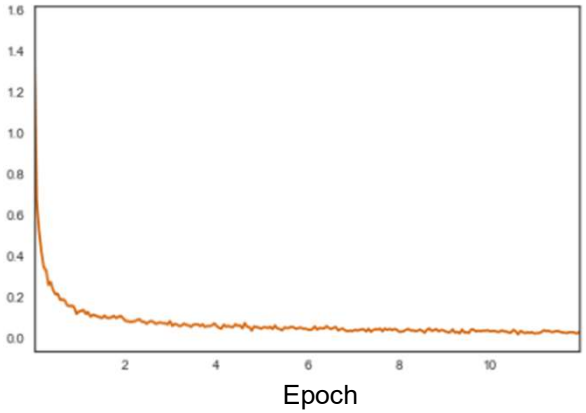
- **Normal Distribution**
- **(0, $\sigma = 0.4$)**



$$W_1 = \begin{pmatrix} 0.01 & -0.50 & 0.14 \\ -0.10 & 0.11 & -0.06 \\ -0.33 & 0.47 & 0.32 \end{pmatrix}$$

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

n_i: Number of neurons in previous layer



$$W_1 = \begin{pmatrix} 1.20 & -1.50 & 1.4 \\ -1.2 & 0.97 & -0.6 \\ 0.63 & -0.47 & 0.52 \end{pmatrix}$$

n_i = 2
=> σ = 1

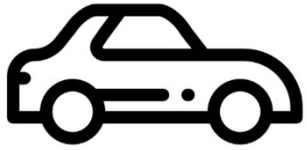
https://intoli.com/blog/neural-network-initialization/

Deep Neural Networks

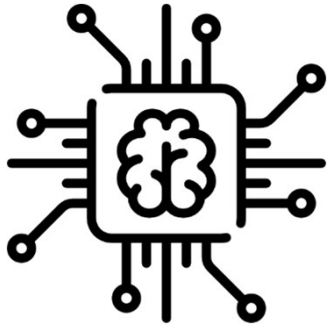
Johannes Betz / Prof. Dr. Markus Lienkamp / Prof. Dr. Boris Lohmann
(Jean-Michael Georg, M. Sc.)

Agenda

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

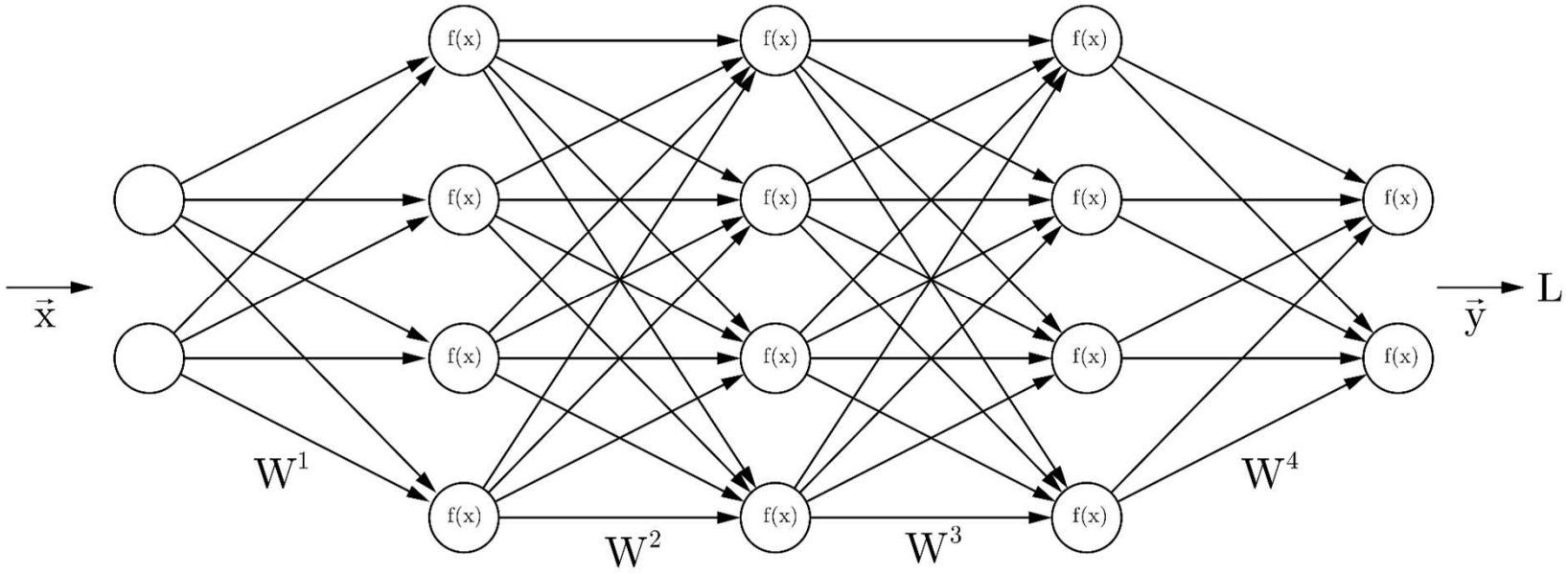


2. Chapter: Neuronal Networks
    - 2.1 Activation Functions
    - 2.2 Fully Connected Layer
    - 2.3 Batches
    - 2.3 Weight Inititalisation



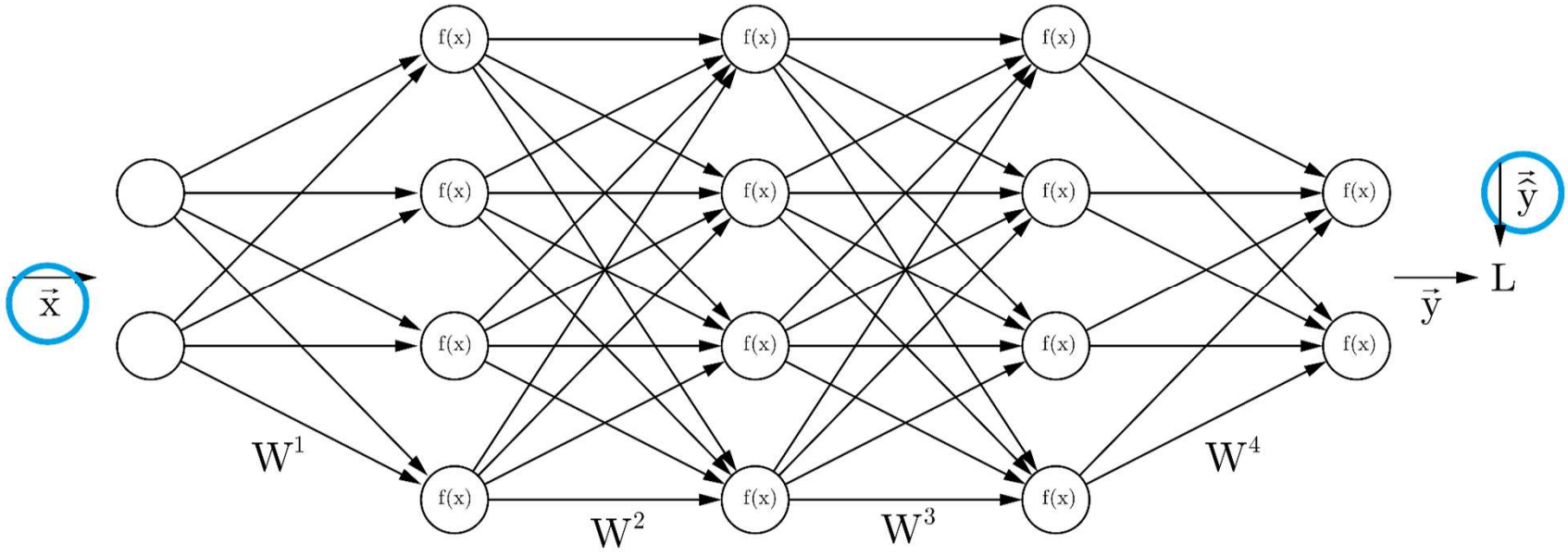
➡ 3. Chapter: Overview

Training a Neural Network



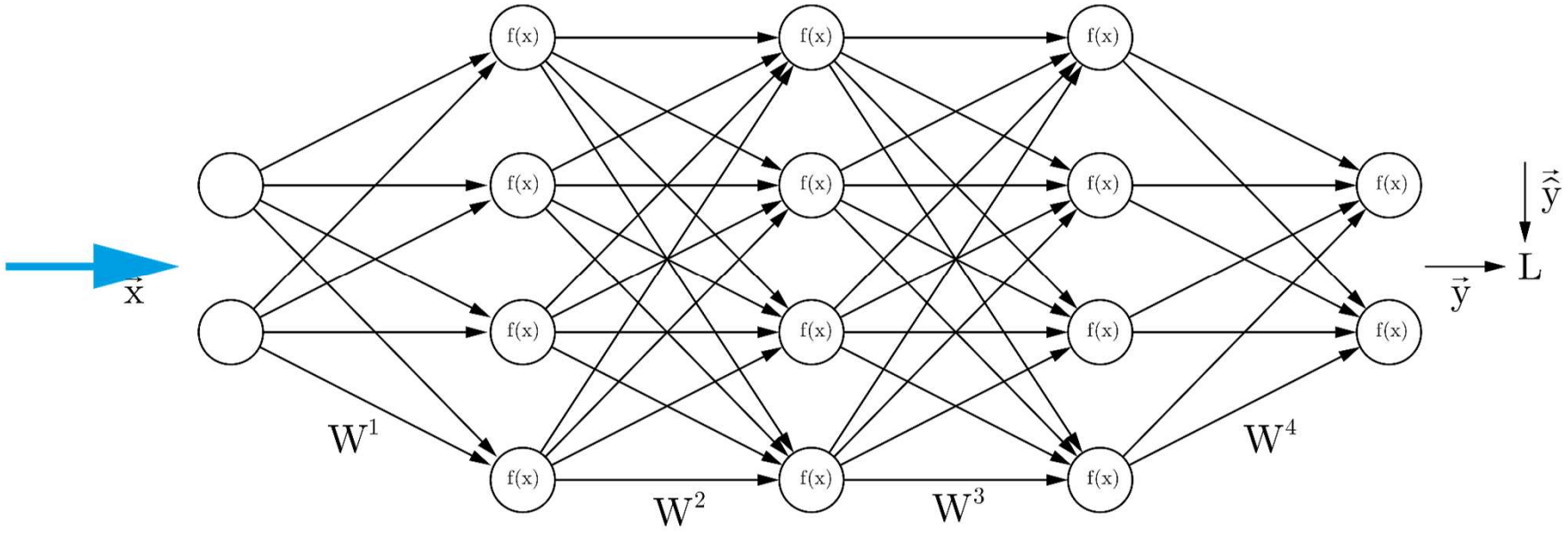
1. Preparing input data
2. Prepare datapipeline to get the data into the network
3. Define the network
4. Initialize all variables
5. Train the network
6. Supervise the training

Training a Neural Network



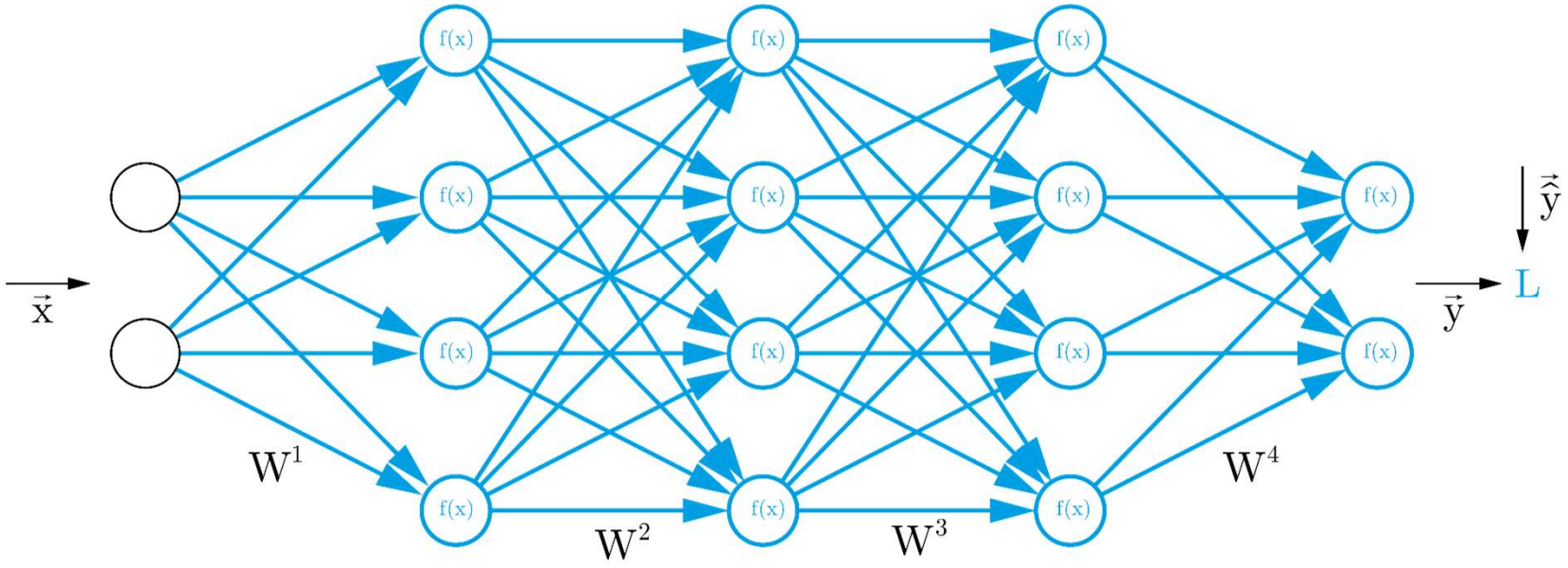
1. **Preparing input data**
2. Prepare datapipeline to get the data into the network
3. Define the network
4. Initialize all variables
5. Train the network
6. Supervise the training

Training a Neural Network



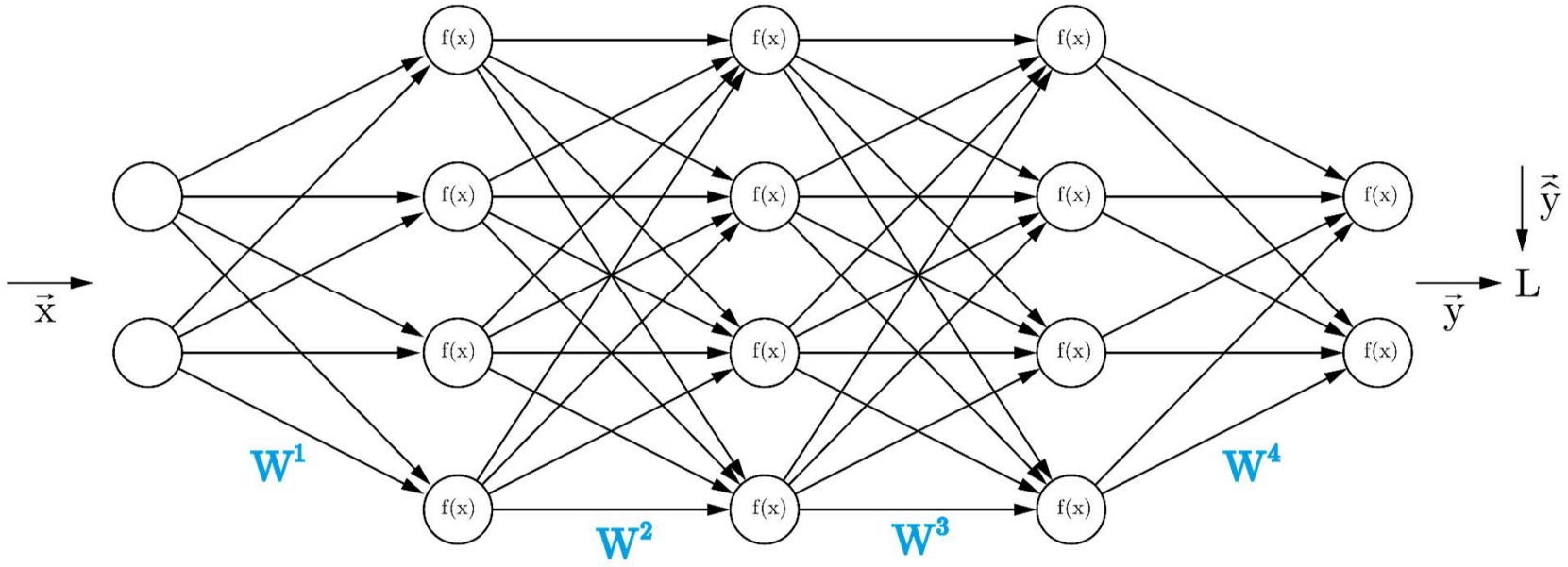
1. Preparing input data
2. **Prepare datapipeline to get the data into the network**
3. Define the network
4. Initialize all variables
5. Train the network
6. Supervise the training

Training a Neural Network



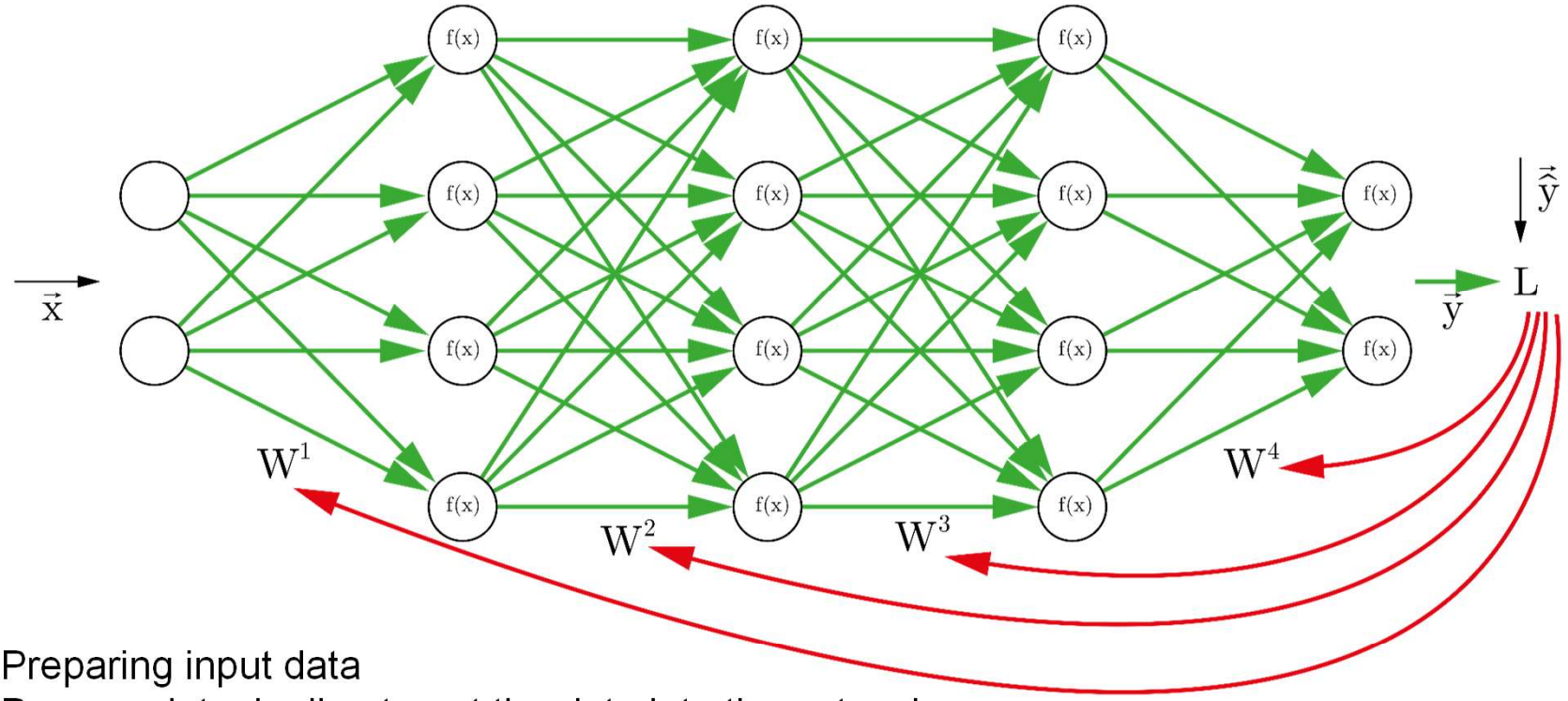
1. Preparing input data
2. Prepare datapipeline to get the data into the network
3. **Define the network**
4. Initialize all variables
5. Train the network
6. Supervise the training

Training a Neural Network



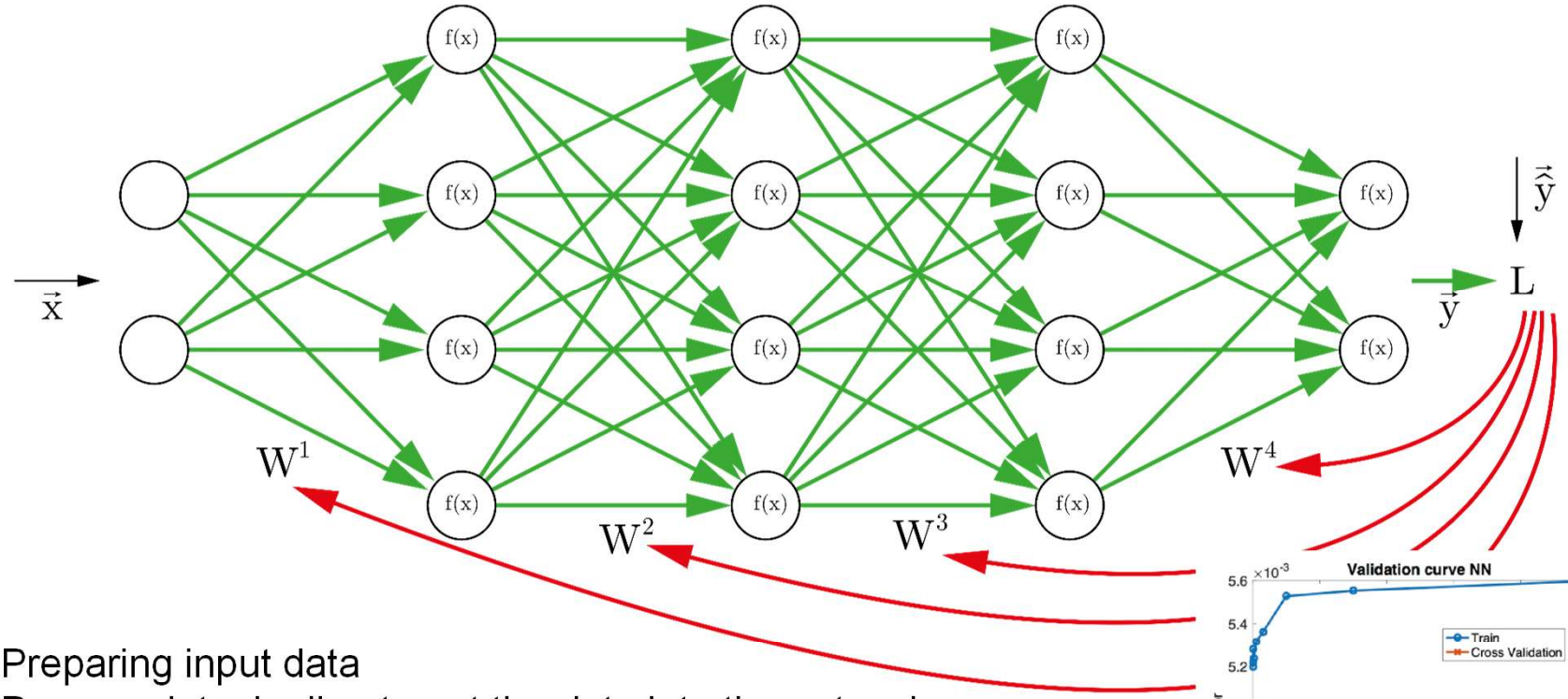
1. Preparing input data
2. Prepare datapipeline to get the data into the network
3. Define the network
4. **Initialize all variables**
5. Train the network
6. Supervise the training

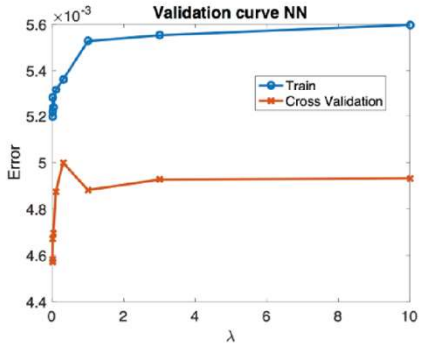
Training a Neural Network



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

Training a Neural Network





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

Summary

What we learned today:

Using computational graphs to calculate local gradients in any mathematical function
Backpropagating local gradients to calculate weight updates
Most common activation functions and their use cases
Use of batches for training neural networks
Initialize weights correctly
Steps to take in training a neural network

End