BACKPROPAGATION

LUCAS SAS BRUNSCHIER

SS20

INTRO

Präsentation ist auch online

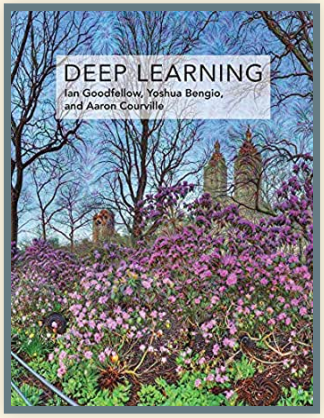
sirbubbls.github.io/backpropagation-seminar

GitHub:

https://github.com/sirbubbls/backpropagation-seminar

ZUSÄTZLICHE RESSOURCEN

Deep Learning (Ian Goodfellow, Yoshua Bengio & Aaron Courville)



https://www.deeplearningbook.org

Backpropagation Kapitel

JUPYTER NOTEBOOK

Beispiele für alle praktischen Beispiele dieser Präsentation sind in diesem Repository zu finden.

AGENDA

1. Vorwissen
    - Gradients & Stochastic Gradient Descent
    - Computational Graphs
2. Neuronale Netze
3. Forward Propagation
4. Backpropagation
    - Kettenregel
    - Backpropagation
    - Implementation in Python
5. General Backpropagation
6. Historisches
7. Quellen

VORWISSEN

GRADIENT

Der Gradient ist der Vektor aller partiellen Ableitungen einer Funktion f.

Notation: $\nabla f(x)$

BEISPIEL

$$f(x) = 2x_1^2 + x_2^3$$

$$\rightarrow \nabla_x f(x) = \begin{pmatrix} f'_{x_1} \\ f'_{x_2} \end{pmatrix} = \begin{pmatrix} 4x_1 \\ 3x_2^2 \end{pmatrix}$$

JACOBI MATRIX

Bei mehreren Funktionen (o.a. ***Komponenten Funktionen****).*

Bei mehreren Funktionen bilden deren Gradienten eine Jacobi Matrix.

$$\boldsymbol{J}(f) = \begin{pmatrix} \nabla f_1 \\ \nabla f_2 \end{pmatrix}$$

STOCHASTIC GRADIENT DESCENT

Der Gradient Descent Algorithmus wird dafür verwendet ein lokales Minimum einer Funktion zu bestimmen.

BEISPIEL

Funktion $f(x) = x_1^2 - x_2^2$ ist gegeben.

Also: $\nabla_x f(x) = \begin{pmatrix} f'_{x_1} \\ f'_{x_2} \end{pmatrix} = \begin{pmatrix} 2x_1 \\ -2x_2 \end{pmatrix}$

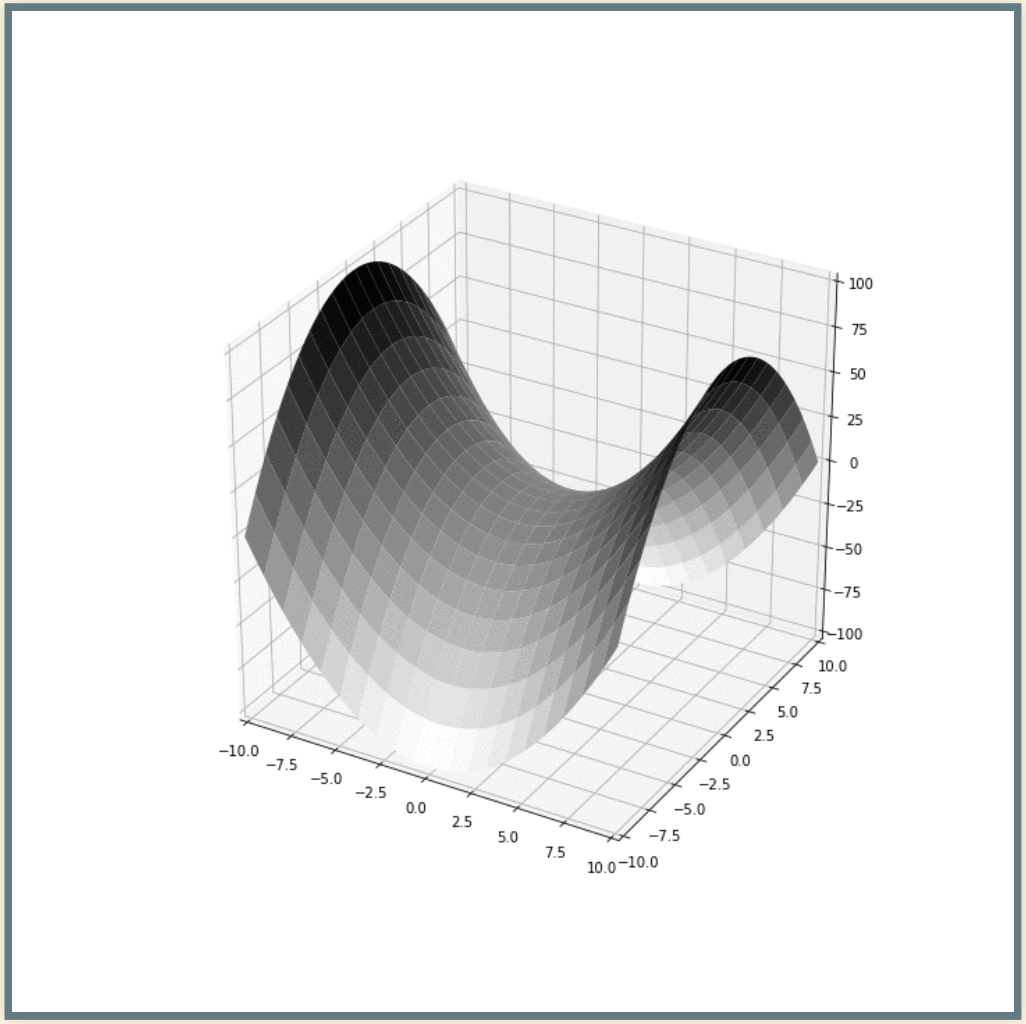
Wir starten mit einem beliebigen Punkt: z.B. $\begin{pmatrix} 2 \\ 1 \end{pmatrix}$ und setzen ein:

$$\begin{pmatrix} 2x_1 \\ -2x_2 \end{pmatrix} = \begin{pmatrix} 2 * 2 \\ -2 * 1 \end{pmatrix} = \begin{pmatrix} 4 \\ -2 \end{pmatrix}$$

$$Neuer\ Punkt = \begin{pmatrix} 2 \\ 1 \end{pmatrix} - \lambda \begin{pmatrix} 4 \\ -2 \end{pmatrix} \text{ mit}$$

$$\lambda : learning\ rate$$

VISUALISIERT



COMPUTATIONAL GRAPHS

Typischerweise werden Operationen in artificial neural networks nicht mit mathematischen Formeln angegeben, sondern als Graph dargestellt.

REPRÄSENTATION

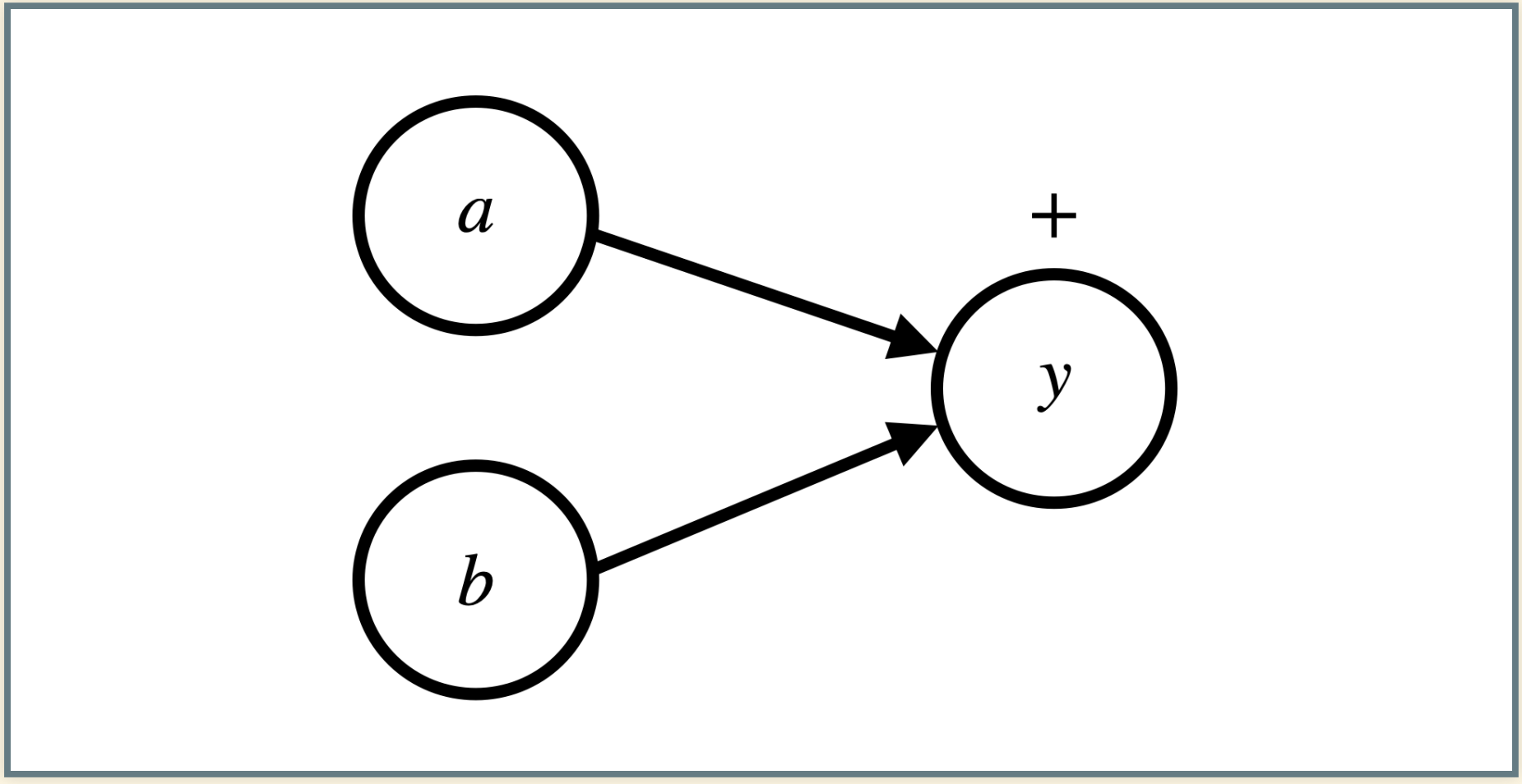
Jede Node in einem Graph G repräsentiert eine mathematische Operation oder eine Input Variable.

Beispielsweise:

- Matrix Multiplikation
- Addition
- Skalare Multiplikation

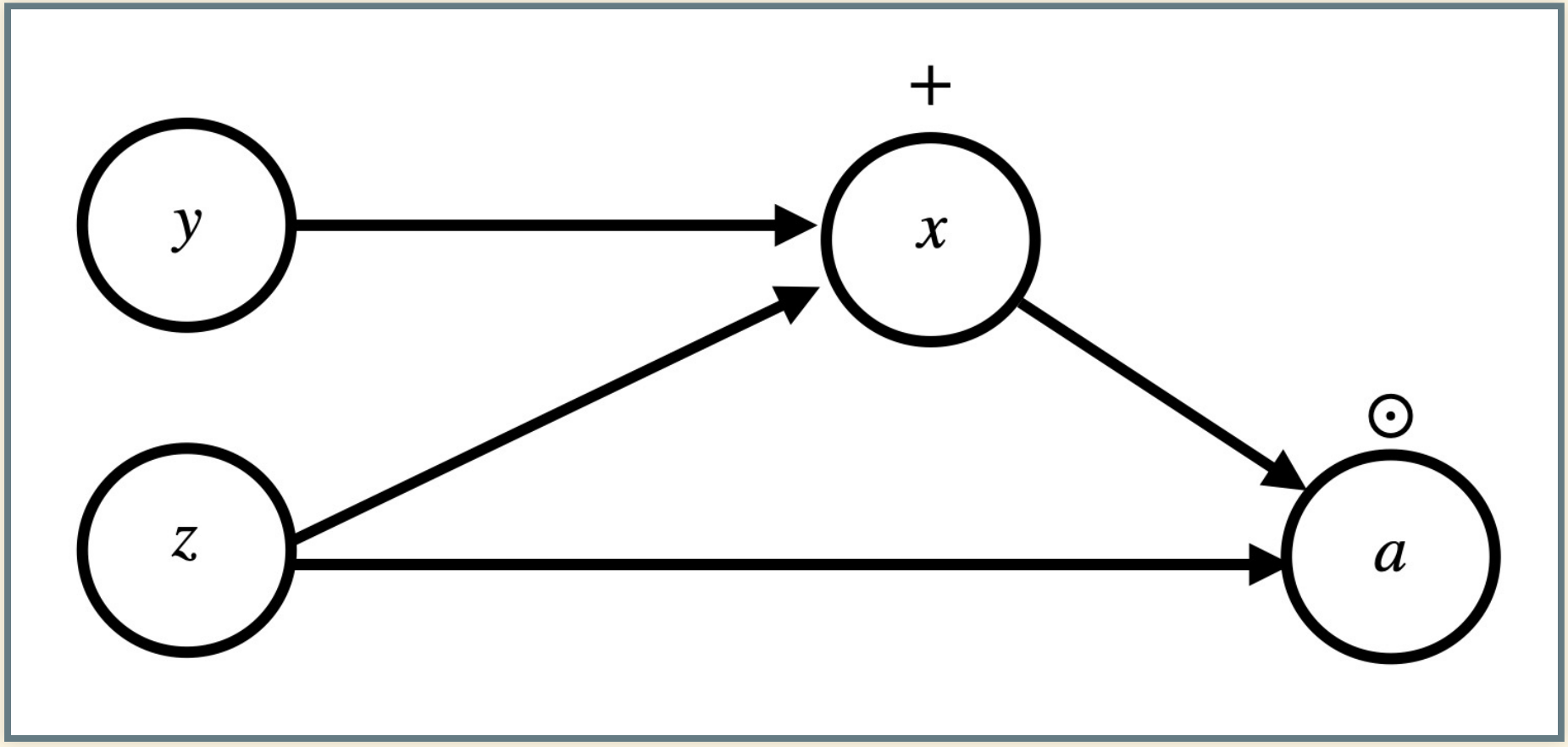
ADDITION BEISPIEL

$$y = a + b$$



KOMPLEXERE BEISPIELE

$$x = y + z$$
$$a = x \odot z$$



KÜNSTLICHE NEURONALE NETZE & DEEP LEARNING

KÜNSTLICHE NEURONALE NETZE

Als Vorbild dienen Neuronale Netze in der Biologie, jedoch sind beide Felder doch unterschiedlicher als man vielleicht erwarten würde.

In diesem Vortrag werden nur fully connected feed forward networks behandelt.

FORMALE DEFINITION

Parameter werden üblicherweise als Theta (θ) notiert.
Der Lernalgorithmus soll die Parameter θ so verändern, dass sich f so nah wie möglich an f^* annähert.

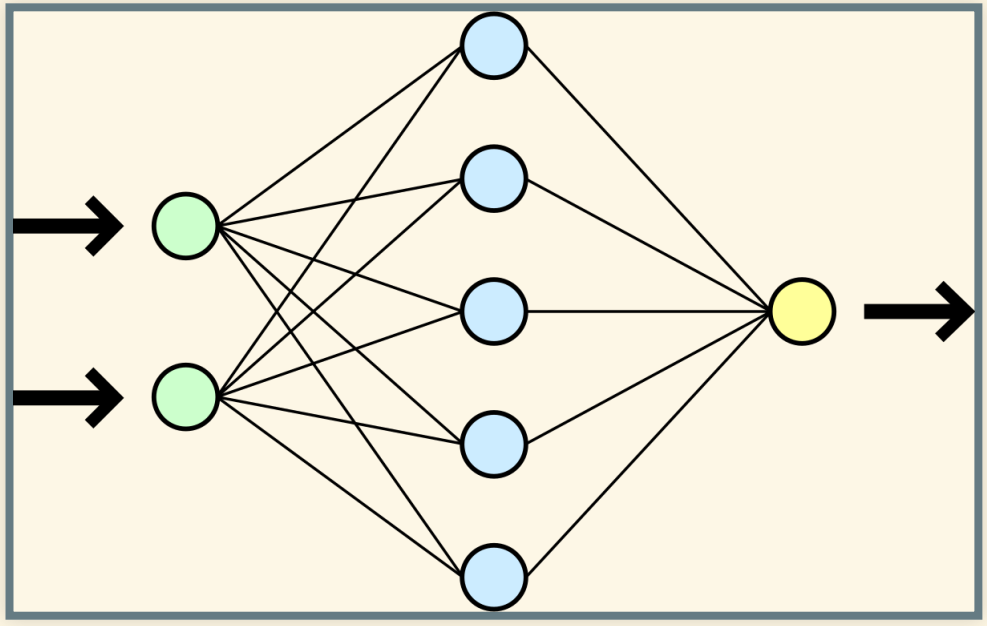
Formale Definition für ein neuronales Netz:

$y = f(x; \theta)$ und $y = f^*(x)$

- y ist den Wert den unser NN voraussagen soll
- x sind die Input Daten, die das NN erhält
- θ sind alle Parameter eines neuronalen Netzwerks
- f^* ist unsere Zielfunktion

WIE IST NUN EIN NEURONALES NETZWERK AUFGEBAUT?

Wir teilen das Netzwerk in Schichten (Layer) auf.



Jeder Layer bildet eine Funktion f^i, mit $i = Layer\ Index$ ab.

FORMELL

Somit ist ein neurales Netzwerk eine Kette an Funktionen f.

Ein Netz mit 3 Layern wäre somit
$f^2(f^1(f^0(X)))$ *mit*
$X = Input\ Data$

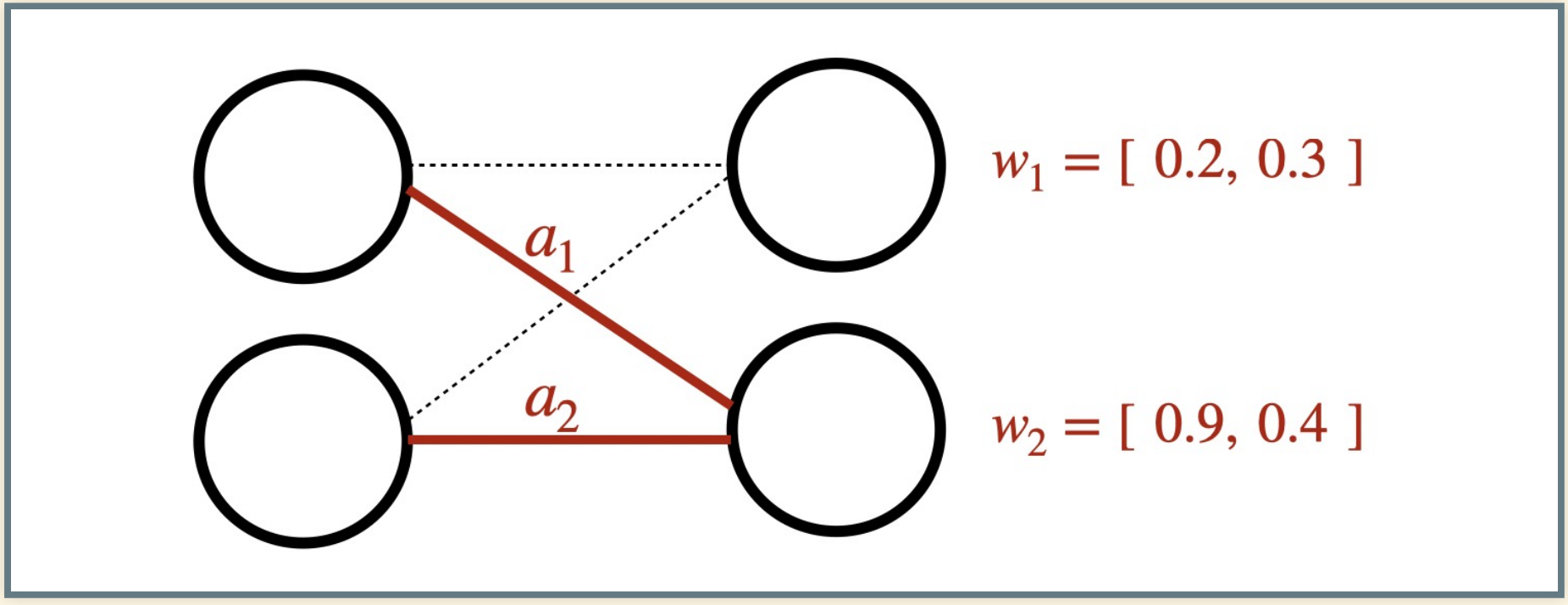
AUFBAU EINES LAYERS

Jeder Layer enthält **mindestens** folgende Informationen:

- Eine Weight Matrix (w)
- Einen Bias Vektor (b)
- Aktivierungsfunktion (σ)

VERBINDUNG DER LAYER

Jedes Neuron eines Layers L_i ist mit allen Neuronen des Layers L_{i-1} verbunden.



AKTIVIERUNGSFUNKTION

Da wir bei Neural Networks oft versuchen non-lineare Zusammenhänge zu approximieren, benötigen wir auch eine nicht-lineare Komponente in unserem NN.

BELIEBTE AKTIVIERUNGSFUNKTIONEN

- Rectified Linear Unit ($ReLU$)

$$f(x) = max(0, x)$$

- $Leaky\ ReLU$

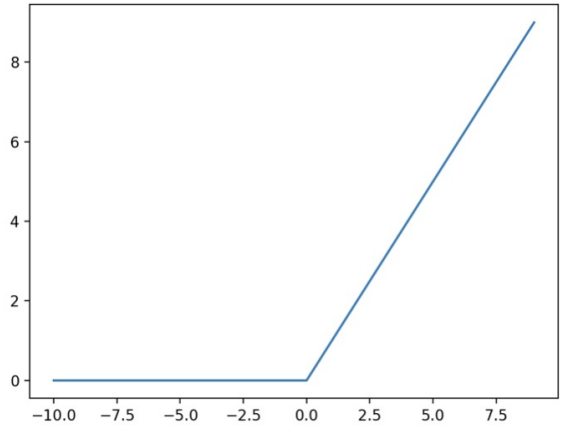
$$f(x) = \begin{cases} x & if\ x > 0 \\ 0.1x & else \end{cases}$$

- Sigmoid Function

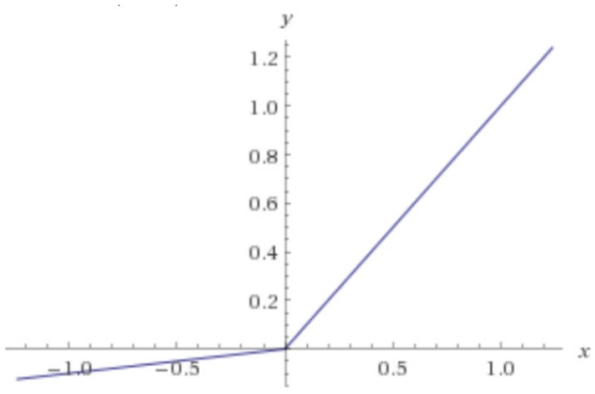
$$f(x) = \frac{1}{1+e^{-t}}$$

FUNCTION PLOT

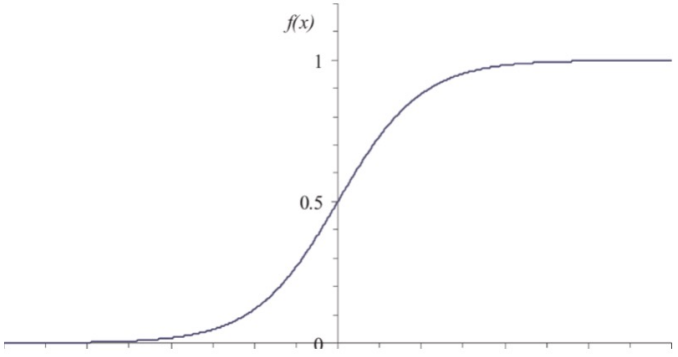
ReLU



LeakyReLU



Sigmoid



COST FUNCTION (J)

Eine Funktion um zu bestimmen wie 'nah' wir uns an unserem erwarteten Inference Wert befinden.

In dieser Präsentation benutzen wir die Euklidean-Distance $(x - y)^2$ *als Cost Function.*

FORWARD PROPAGATION

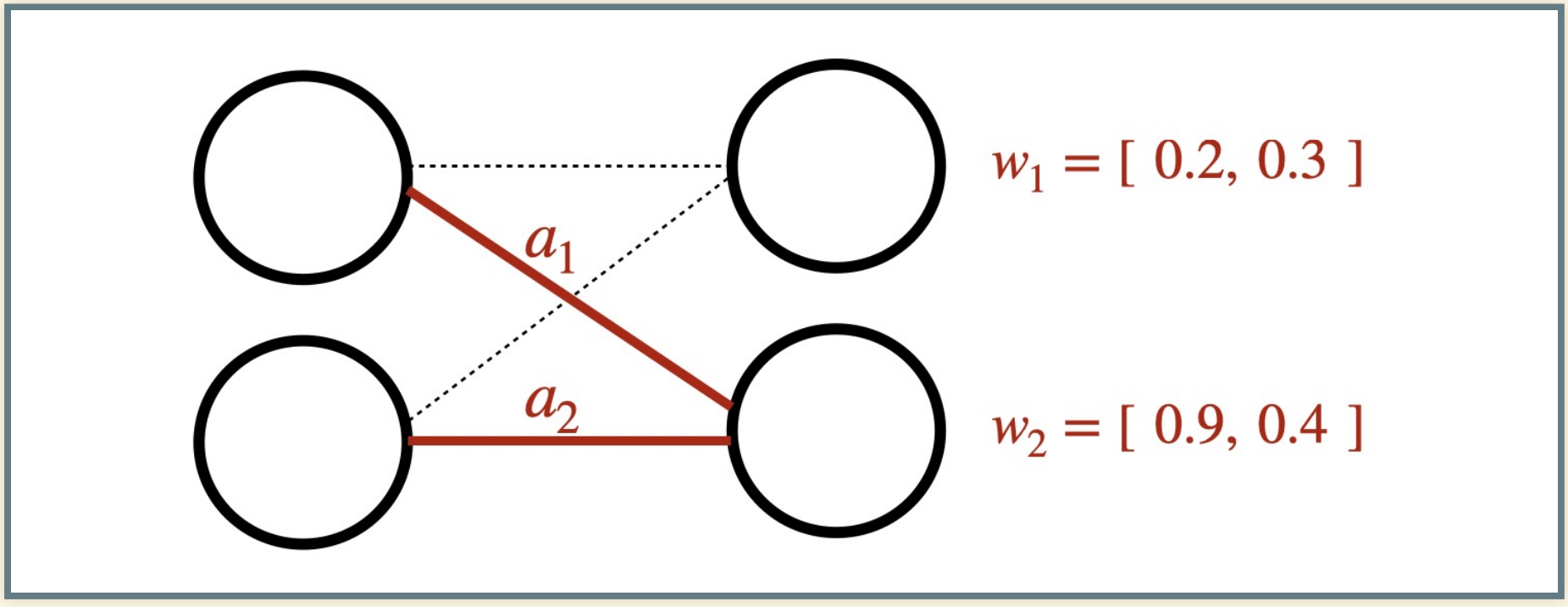
Ein Layer in einem Feed-Forward Neural Network besteht aus folgenden Elementen:

- Inputs (X)
- Weights (W)
- Biases (b)
- Output (a)

BERECHNUNG DES INPUTS

Jedes Neuron enthält einen Vektor mit Weights w, der Angibt wie stark jeder Input gewichtet wird.

$a_1 * w_1 + a_2 * w_2$ oder $a_{L-1} W_L$



FORMELL

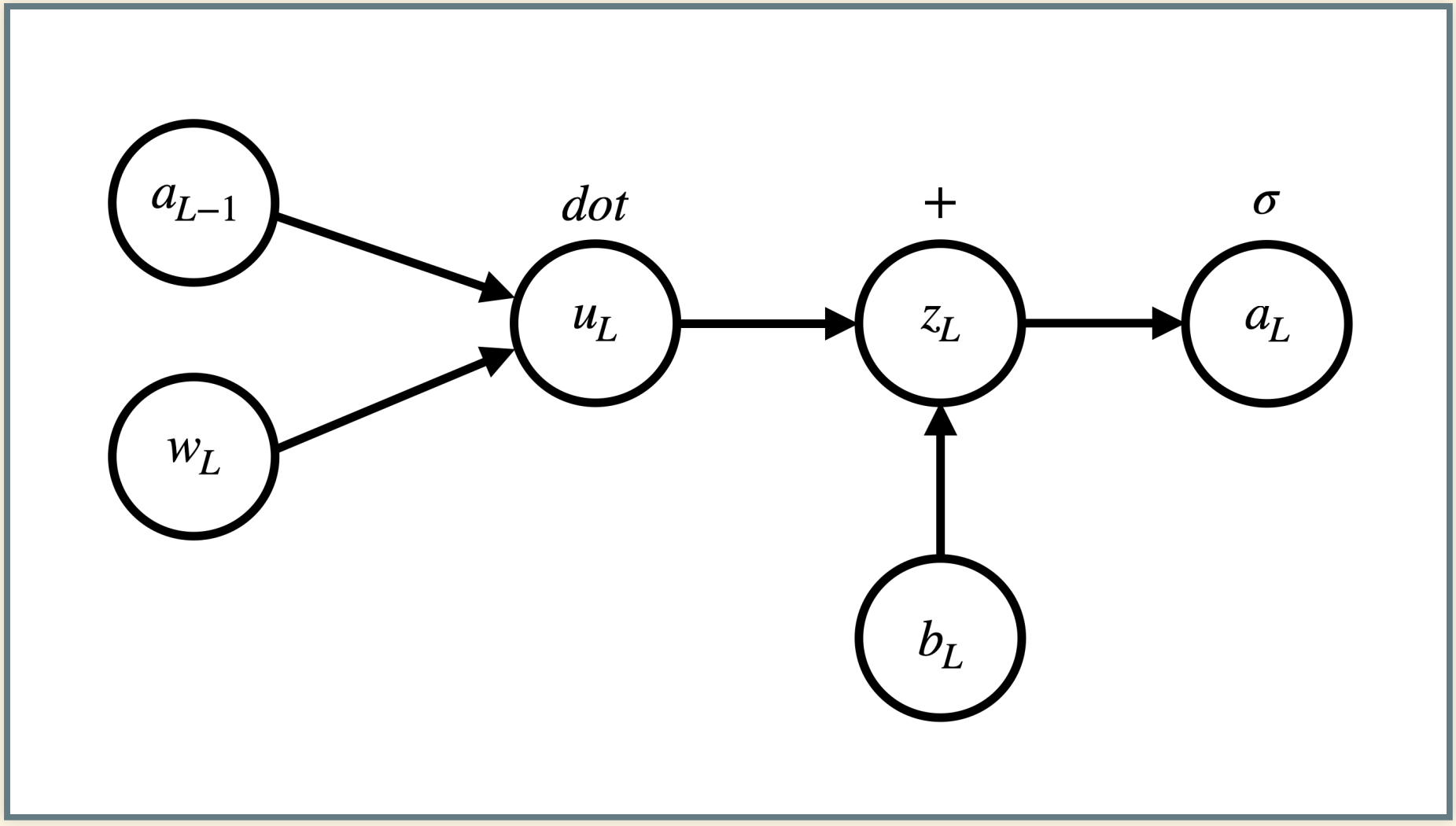
Um die Aktivierungen (a) eines Layers zu berechen können wir folgende Formel benutzen:

$$a_L = \sigma(W_L a_{L-1} + b_L)$$

Der berechnete Vektor a_L dient dem Layer $L + 1$ als Input.

COMPUTATIONAL GRAPH

$$a = \sigma(a_{L-1} W_L + b)$$


a_{L-1}
w_L
dot
u_L
$+$
z_L
b_L
σ
a_L

BEISPIEL (XOR)

$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$b = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

MULTIPLIZIEREN DER WEIGHTS (W) UND INPUTS (X)

$$XW = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$$

ADDIEREN DES BIAS VEKTORS (b)

$$XW + b = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix} + \begin{pmatrix} 0 \\ -1 \end{pmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

AKTIVIERUNGSFUNKTION (IN DIESEM FALL $ReLU$)

$$ReLU := f(x) = max(0, x)$$

$$relu(XW + b) = relu(\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}) = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

Die Aktivierungsfunktion wird auf jedes Element der Matrix ausgeführt.

OUTPUT LAYER

Multiplizieren der Output Matrix des ersten Layers mit den Weights des Output Layers (w).

$$w = relu(XW + b) * \begin{bmatrix} 1 \\ -2 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} * \begin{bmatrix} 1 \\ -2 \end{bmatrix} =$$

PREDICTIONS & INPUT

Input: $\begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$

Predictions: $\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$

CODE BEISPIEL

```
def forward(X):
    a = X
    for layer in L:
        a = layer.weights @ a + layer.bias
    return a
```

LAUFZEITKOMPLEXITÄT

$$O(w)$$

- w Anzahl der Weights in neuronalem Netz.

BACKPROPAGATION

WOZU BRAUCHEN WIR DEN BACKPROPAGATION ALGORITHMUS?

Ein fundamentaler Baustein, von neuralen Netzen.

Backpropagation ist kein Lernalgorithmus/Optimierungsalgorithmus, sondern aussschließlich für die Generierung der Gradients jedes Layers zuständig.

Also suchen wir folgende Gradients:

- $\nabla_{b^k} J$
- $\nabla_{w^k} J$

KETTENREGEL

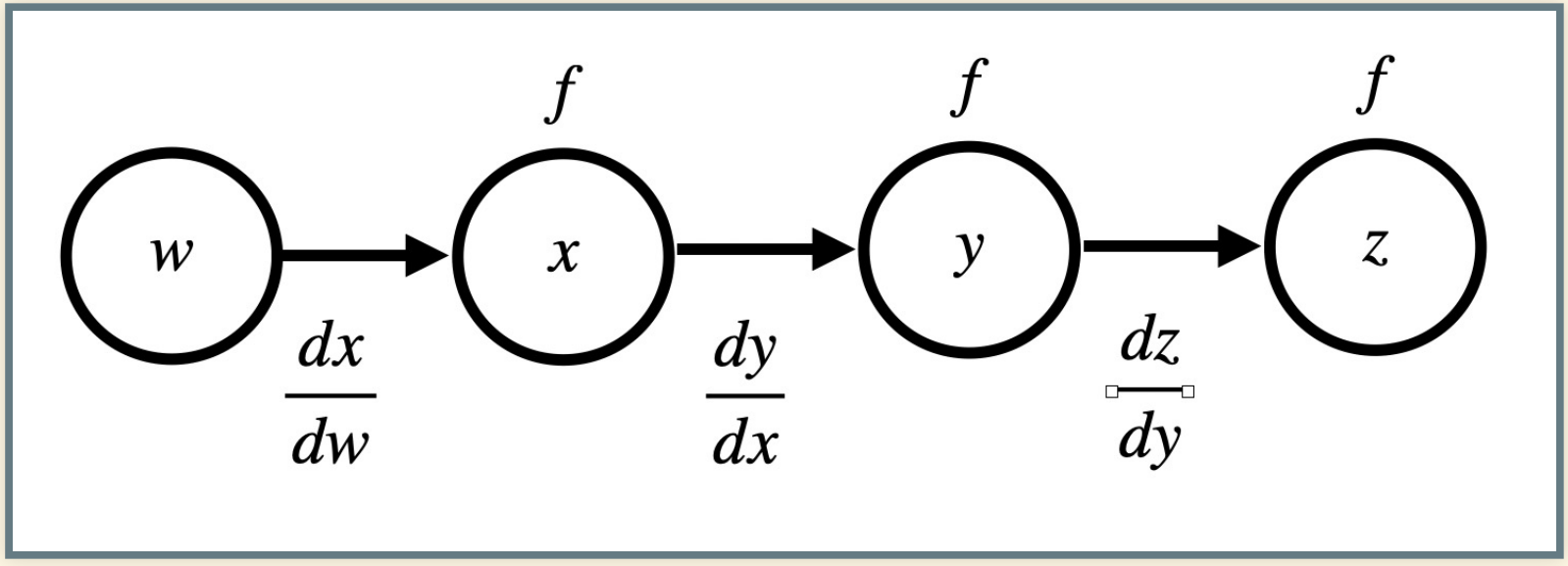
Die Kettenregel ist nützlich um Ableitungen aus schon bereits vorhandenen Ableitungen zu konstruieren.

$$y = g(x) \; und \; z = f(g(x)) = f(y)$$

Dann besagt die Kettenregel: $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$

KETTENREGEL ALS GRAPH

$$x = f(w),\ y = f(x),\ z = f(y)$$



$$\frac{dz}{dw} = \frac{dz}{dy}\frac{dy}{dx}\frac{dx}{dw} = f'(y)f'(x)f'(w)$$

$$= f'(f(f(w)))f'(f(w))f'(w)$$

ANPASSUNG DER FORWARD PROPAGATION

Wir benötigen folgende Werte aus jedem Layer um den Backpropagation Algorithmus ausführen zu können.

- a Aktivation Vektor
- z Pre Activation Function Vektor

$f'(y)f'(x)f'(w)$: *Speichern der Zwischenergebnisse in Variablen*
$f'(f(f(w)))f'(f(w))f'(w)$: *Neu Evaluierung der Zwischenergebnisse*

BESCHREIBUNG DES ALGORITHMUS

SCHRITT 1

Forward Propagation ausführen.

SCHRITT 2

Wir berechnen den Gradienten der Cost Function J.

$$J = \frac{1}{2}(y - X)^2 \rightarrow \nabla_y J = X - y$$

SCHRITT 3

Erst müssen wir den Gradienten in Relation zu den pre activation function values berechnen.

$$\nabla_{a^k} J = g \odot f'(a^{(k)})$$

mit

$$f'(x) := Ableitung\ der\ Aktivierungsfunktioin$$

SCHRITT 4

Weight Gradienten berechnen.

$$f(w, a, b) = w * a + b$$

$$g * \frac{\partial}{\partial w} = g * (a + 0)$$

$$\nabla_{w^k} J = g a^{k-1}$$

SCHRITT 5

Bias Gradienten berechnen.

$$f(w, a, b) = w * a + b$$

$$g * \frac{\partial}{\partial b} = g * 1$$

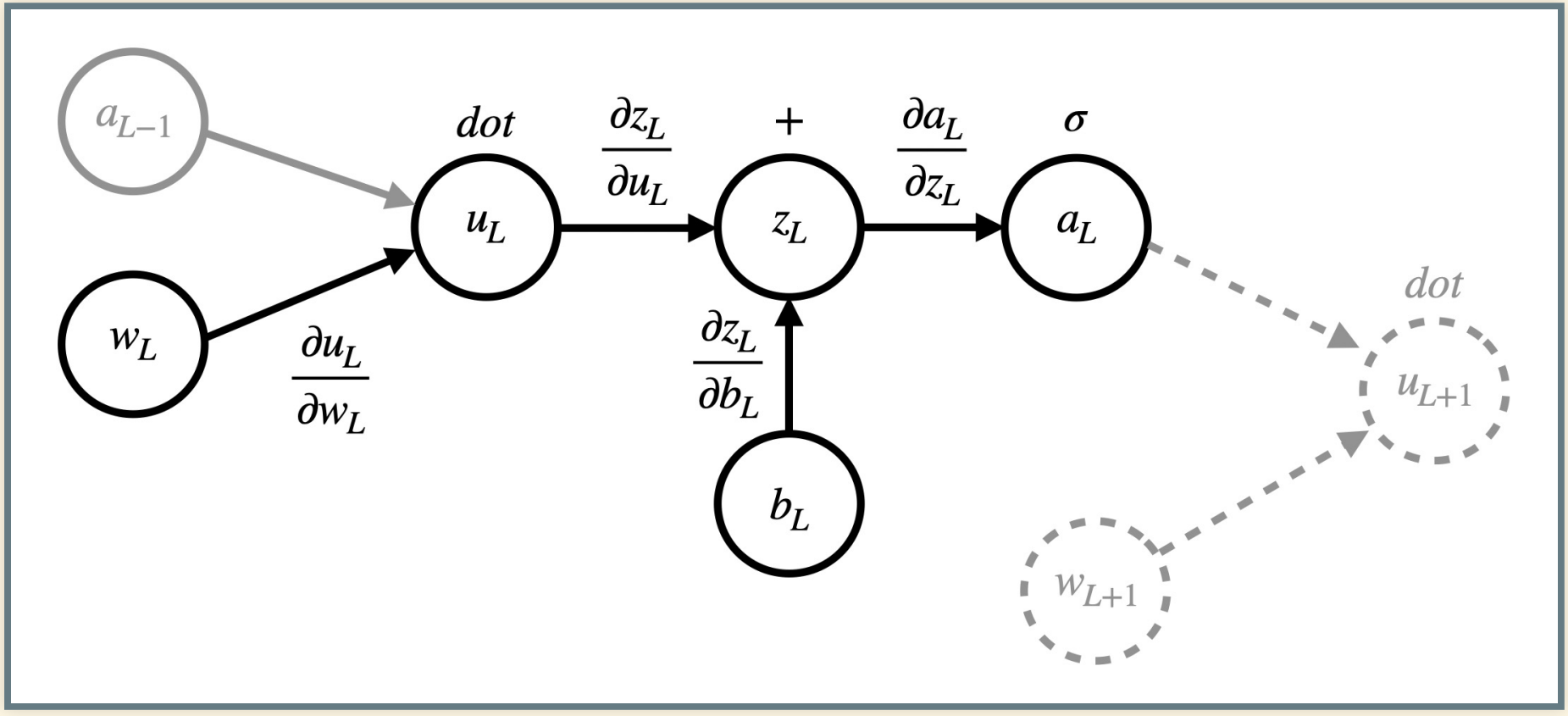
$$\nabla_{b^k} J = g$$

SCHRITT 6

$$\nabla a^{k-1} J = w^k g$$

WIEDERHOLEN VON SCHRITT 3 - 5 DES NÄCHSTEN LAYERS ($L-1$)

GRAPH


a_{L-1}
w_L
dot
u_L
$\frac{\partial u_L}{\partial w_L}$
$\frac{\partial z_L}{\partial u_L}$
+
z_L
$\frac{\partial z_L}{\partial b_L}$
b_L
$\frac{\partial a_L}{\partial z_L}$
σ
a_L
dot
u_{L+1}
w_{L+1}

PRAKTISCHES BEISPIEL IN PYTHON

Basic Backpropagation Implementation

MINI BATCH TRAINING

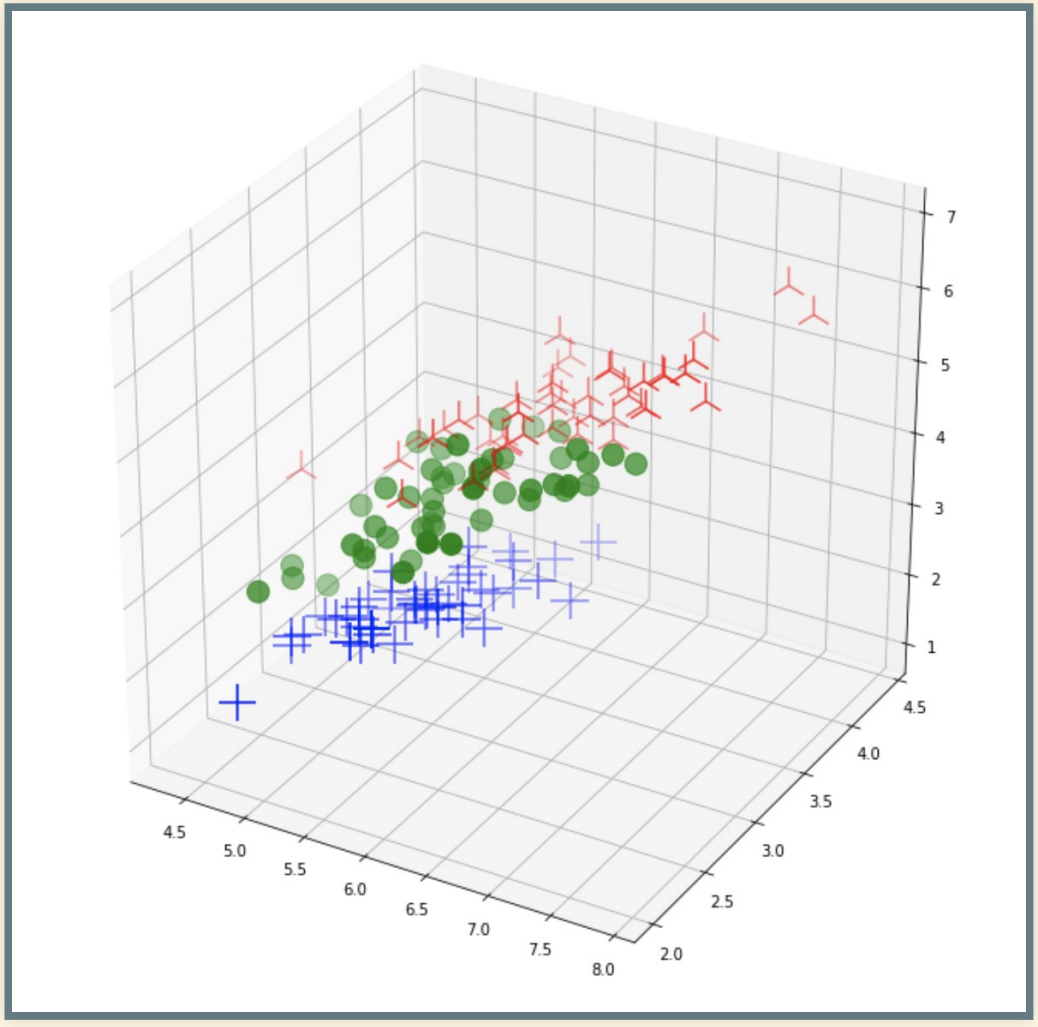
In der Praxis werden keine Vektoren als Input Daten benutzt, sondern Matrizen (siehe XOR Beispiel).

$$Input = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

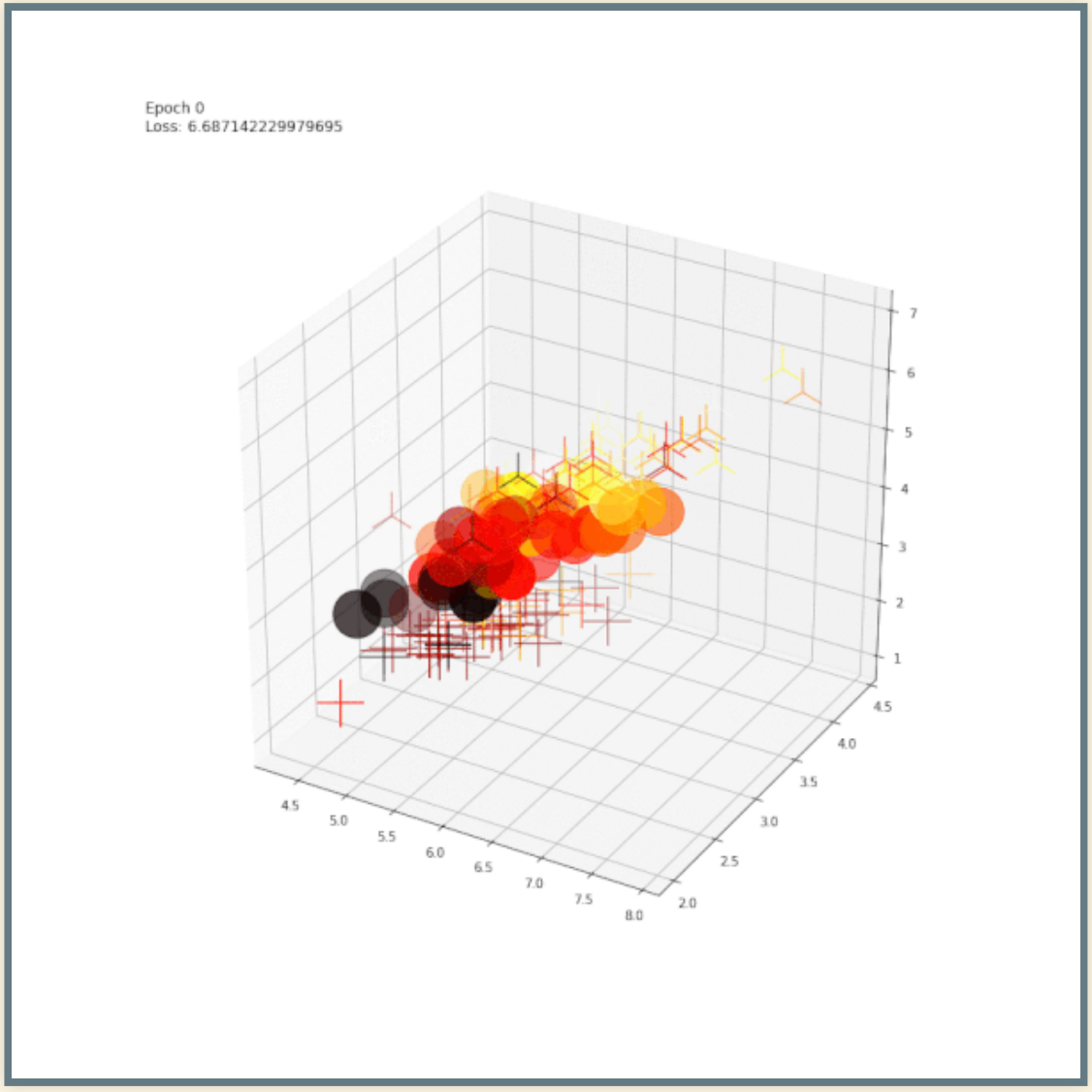
Wir erhalten nun auch mehrere Gradienten in Form einer Matrix. Wir können nun den Durchschnitt der Gradienten nutzen um unsere Weights anzupassen.

IRIS DATASET

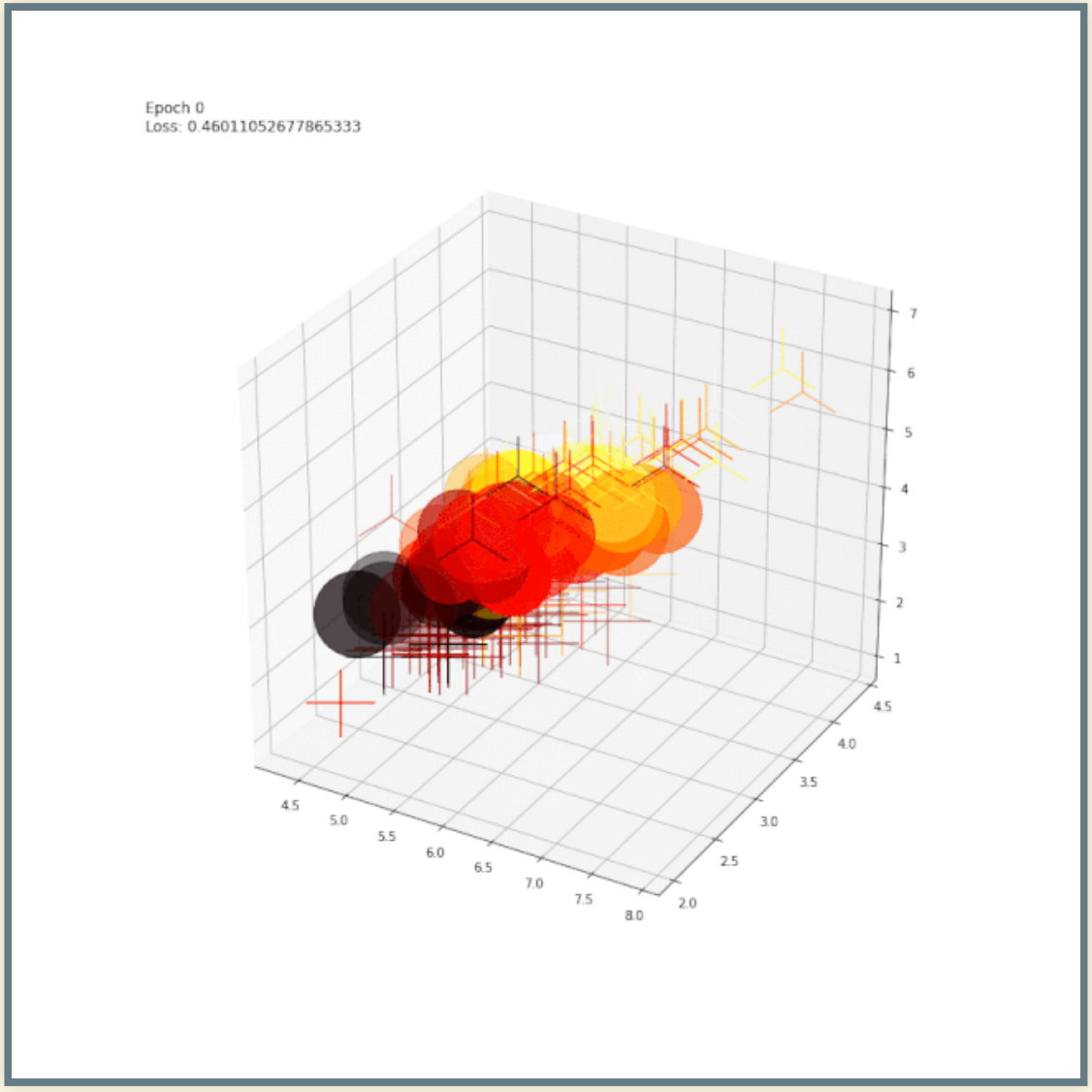
Mini Batch Backpropagation Implementation



LOW LEARNING RATE



HIGH LEARNING RATE



KOMPLEXITÄT

Wichtig

Folgende Komplexitäten beziehen sich ausschließlich auf den Backpropagation Algorithmus.

LAUFZEITKOMPLEXITÄT

Backpropagation besitzt die gleiche Laufzeitkomplexität wie Forward-propagation.

$$O(w)$$

- w Anzahl der Weights in neuronalem Netz.

SPEICHERKOMPLEXITÄT

$$O(mh)$$

- m Anzahl an Elementen in Batch
- h Anzahl der Hidden-Units

GENERAL BACKPROPAGATION

Bisher haben wir uns nur mit Backpropagation in Zusammenhang mit neuronalen Netzwerken beschäftigt.
Backpropagation kann aber auch generell für andere Anwendungen eingesetzt werden.

SYMBOL TO NUMBER / SYMBOL TO SYMBOL

Es existieren zwei verschiedene Möglichkeiten die Berechnungen der Gradients durchzuführen.

- Symbol to Number
- Symbol to Symbol

SYMBOL TO NUMBER

Die Input Variablen werden durch Zahlenwerte ersetzt und daraufhin (wie besprochen) alle nötigen Gradienten berechnet.

SYMBOL TO SYMBOL

Beim der Symbol to Symbol Herangehensweise wird zuerst der Graph mit allen Ableitungen mit der Hilfe von symbolischen Werten konstruiert.
Später wird dann der Graph mit der Hilfe eines eigenen Algorithmus ausgewertet.

Vorteil

Ableitungen eines höheren Grads können berechnet werden, indem man den Backpropagation Algorithmus auf einen bereits abgeleiteten Graphen ausführt.

OPERATIONEN

Wir betrachten einen computational Graph, jede Node in dem Graph repräsentiert eine Variable in Form eines Tensors.

FUNKTIONEN

- `get_operation()`
- `get_consumers()`
  Gibt alle Variablen/Operationen zurück, die 'Kinder' von sich selber sind.
- `get_inputs()`
  Gibt alle Variablen/Operationen zurück, die 'Eltern' von sich selber sind.
- `bprop()`
  Muss bei jeder Operation implementiert werden.

ALGORITHMUS

Benötigt ist:

- die Menge aller Variablen T, deren Gradienten wir berechnen müssen
- den Graphen G
- die Variable z, die wir differenzieren wollen

ÄUSSERE FUNKTION

Wir definieren G' als alle Variablen, die Vorfahren von z und Nachfahren von T sind.

In `grad_table` können wir Variablen Gradients zuweisen.

`grad_table[z] = 1` (da $\frac{\partial z}{\partial z} = 1$)

LOOP ÜBER ALLE VARIABLEN, DEREN GRADIENTEN WIR BERECHNEN MÜSSEN

In jedem Loop rufen wir die Funktion `build_grad` auf.

```
for v in T:
    build_grad(v, G, G_1, grad_table)
return [grad_table[v] for v in T]
```

build_grad(v, G, G_1, grad_table)

```
def build_grad(v, G, G_1, grad_table):
    if v in grad_table: return grad_table[v]
    i = 1
    for c in get_consumers(v, G_1):
        op = get_operation(c)
        d = build_grad(c, G, G_1, grad_table)
        g[i] = op.bprop(get_inputs(c, G_1), v, d)
        i += 1
    g = sum(g)
    grad_table[v] = g
    return g
```

`bprop` FUNKTION

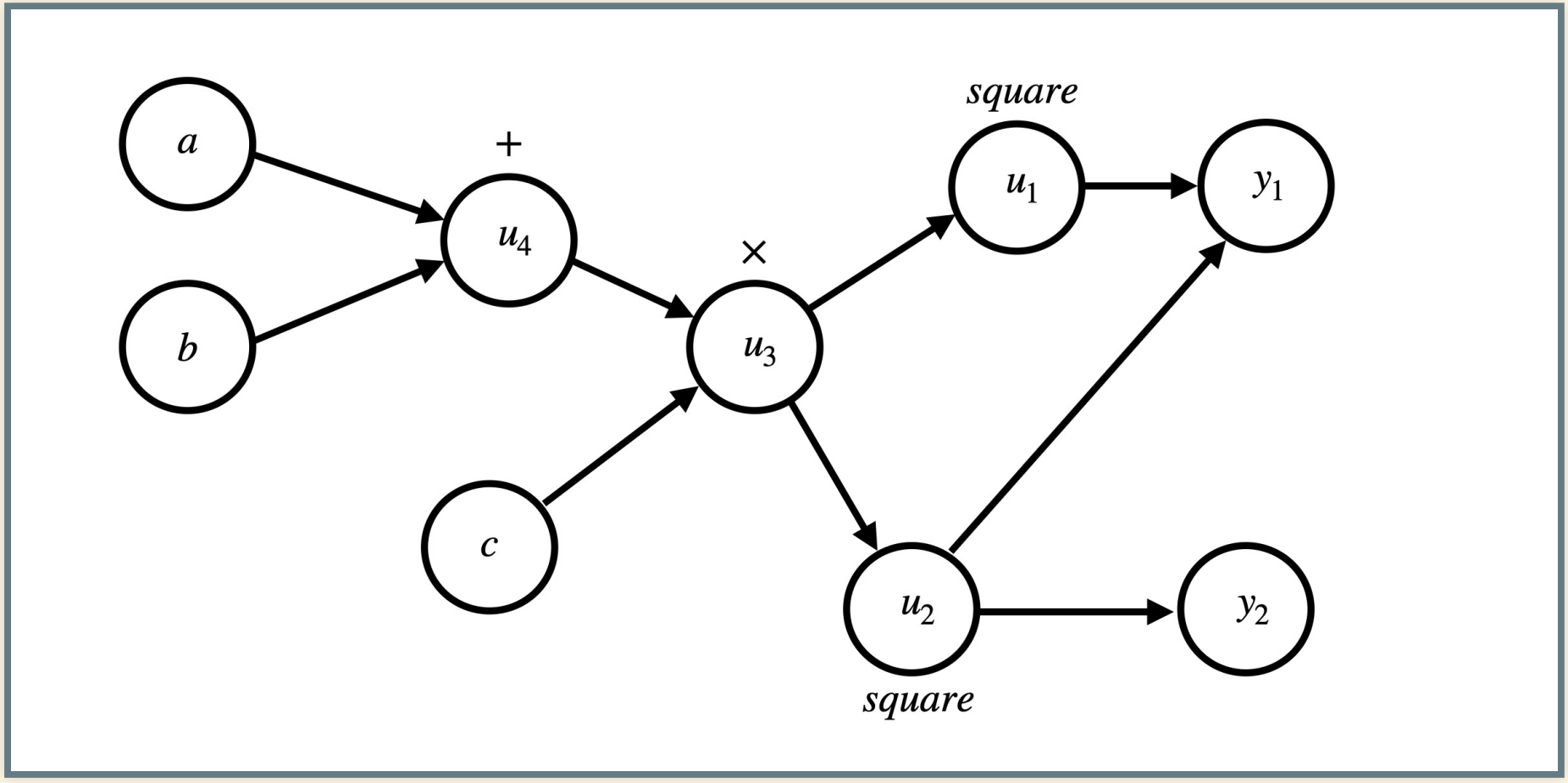
`op.bprop(inputs, X, G)`

`inputs`: Liste an Inputs, die wir der Operation zur Verfügung stellen
`X`: Input, dessen Ableitung wir berechnen wollen
`G`: Gradient des Outputs der Operation

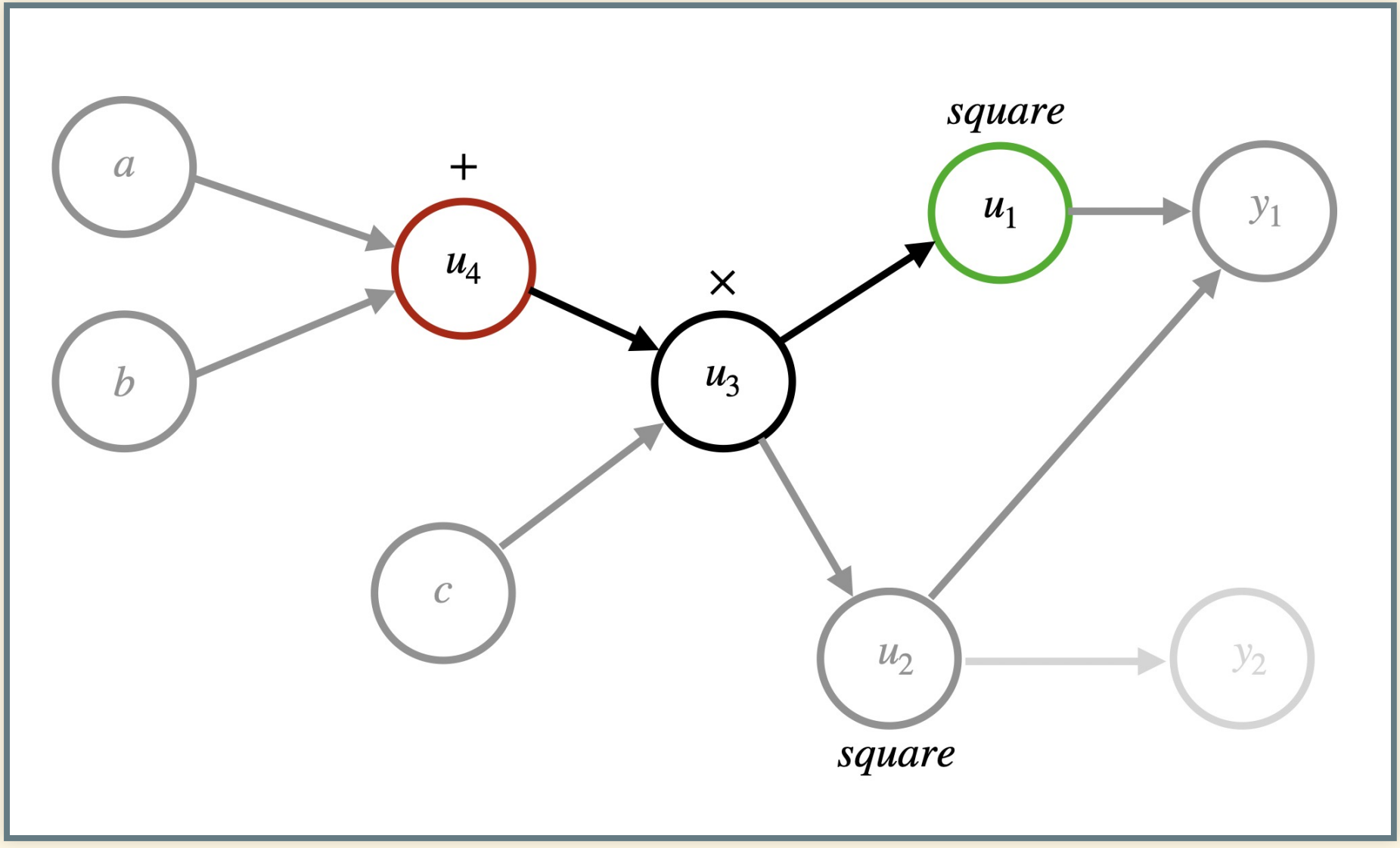
BEISPIEL

GRAPH

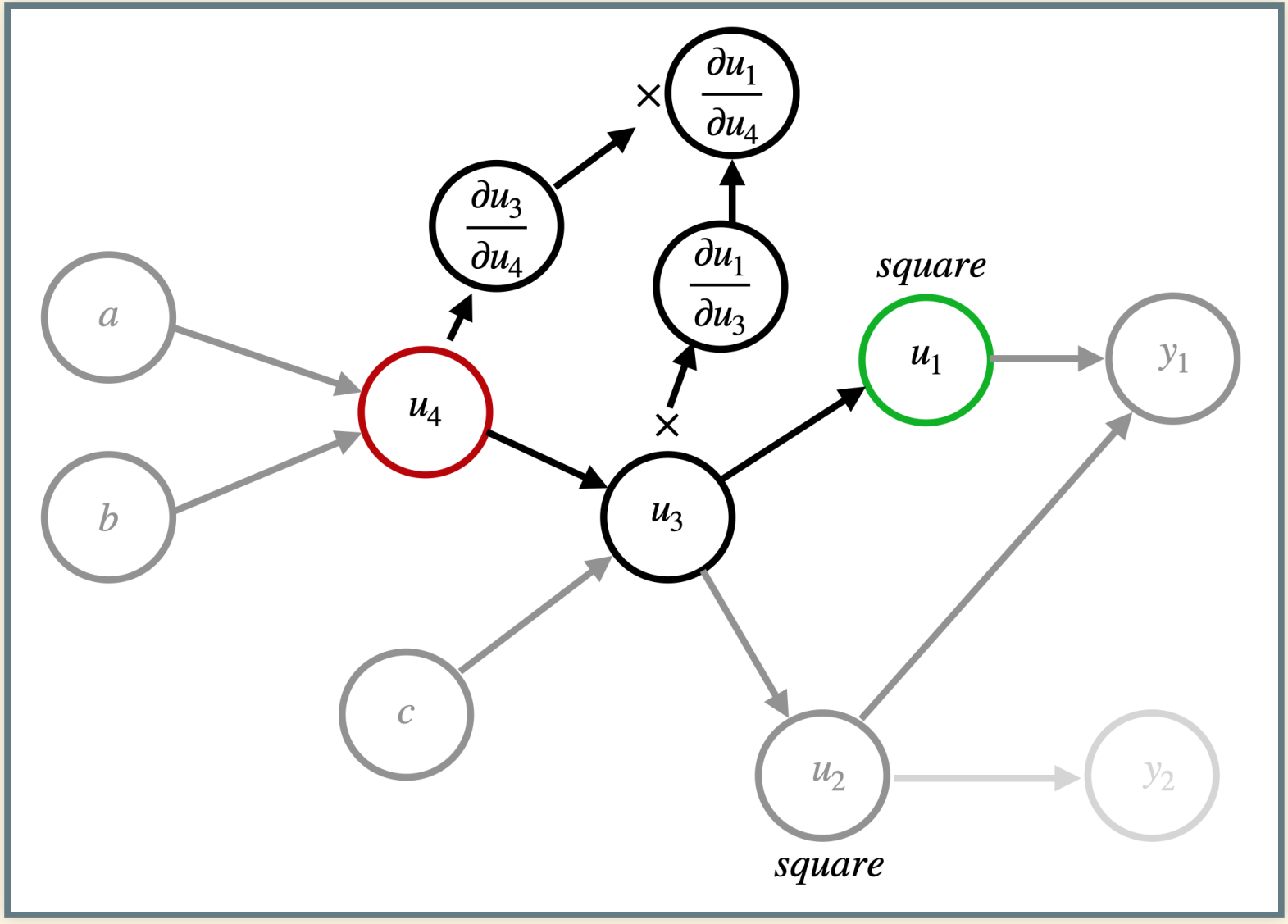
Wir wollen $\frac{\partial u_1}{\partial u_4}$ bestimmen.



BESTIMMEN DER ABLEITUNG $\frac{\partial u_1}{\partial u_4}$



EINTRAGEN ALLER ABLEITUNGEN IN GRAPH



GENERALISIERBARKEIT

Dadurch ist der Backpropagation Algorithmus sehr allgemein anwendbar.

Jede Operation ist für seine eigene Differenzierung verantwortlich und benötigt keine weiteren Informationen.

HISTORISCHES

- Kettenregel stammt aus dem 17ten Jahrhundert (Leibniz, 1676).
- Lineare neurale Netzwerke Mitte des 20ten Jahrhunderts.
- Erfolgreiche Experimente mit Back-Propagation (1986)

POPULARITÄT VON NEURONALEN NETZEN

Klassische machine learning Algorithmen wurden in den 90er Jahren mehr genutzt als neuronale Netzwerke.

Durch die hohe Speicheranforderung wurden NN ab ca. 2006 immer vermehrter eingesetzt und bilden heute einen fundamentalen Baustein von maschinellem Lernen.

BACKPROPAGATION & GRADIENT DESCENT

Beide treibenden Algorithmen von neuronalen Netzwerken haben sich seit den 80er Jahren nicht wesentlich verändert.

Bessere Resultate sind besser Hardware und Dataset Optimierung zu verdanken.

QUELLEN

- Deep Learning (Ian Goodfellow, Yoshua Bengio & Aaron Courville)
- Back-Propagation is very simple. Who made it Complicated? (Prakash Jay)
- Wikipedia: https://en.wikipedia.org/wiki/Backpropagation
- Wikipedia: https://en.wikipedia.org/wiki/Delta_rule