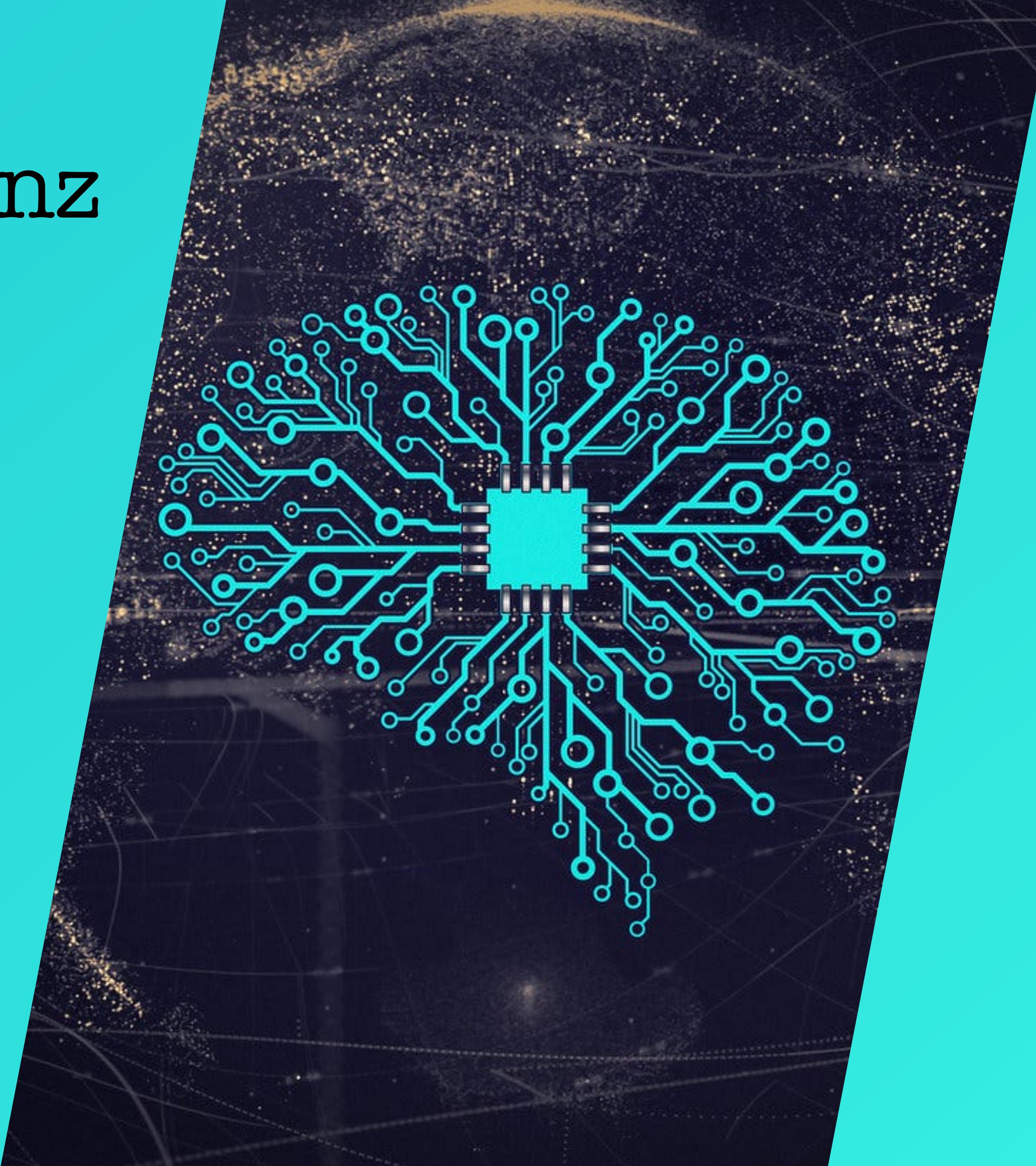


# Künstliche Intelligenz selbst gemacht

Neuronale Netze in Python entwickeln



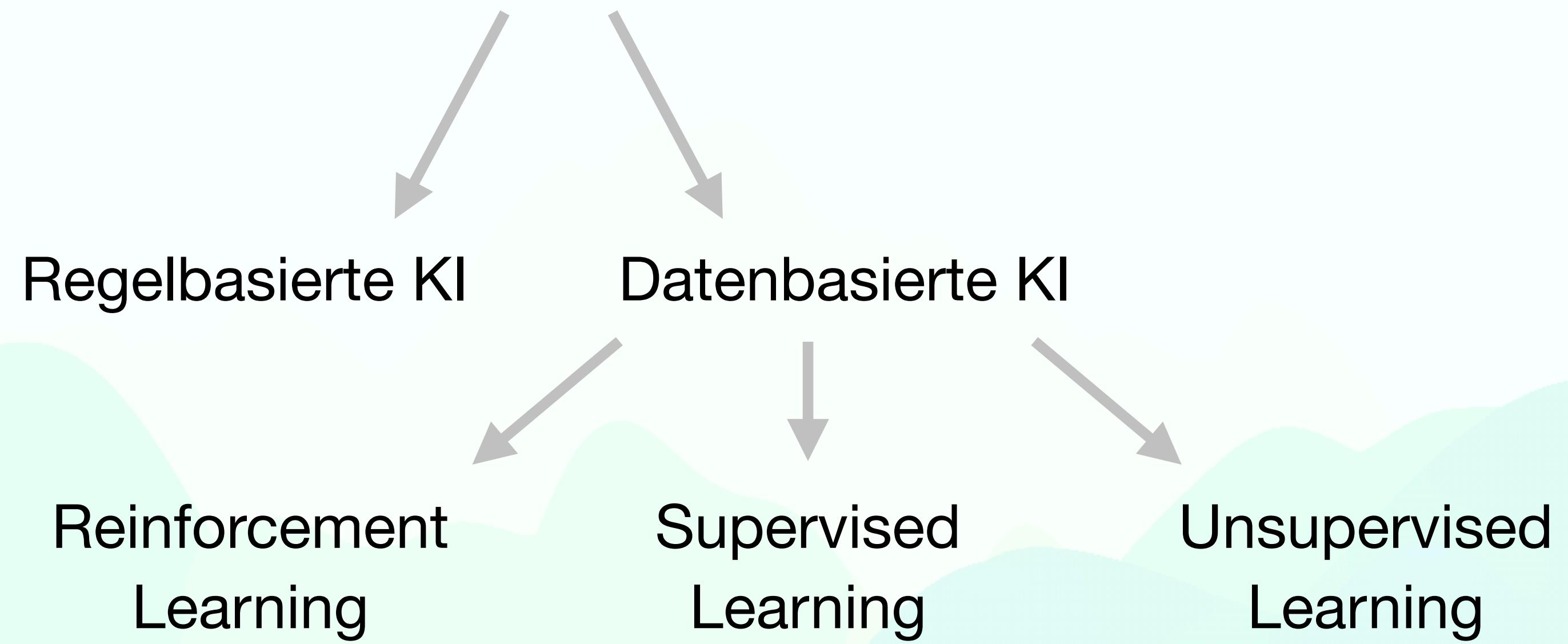
# „Künstliche Intelligenz“

# „Künstliche Intelligenz“

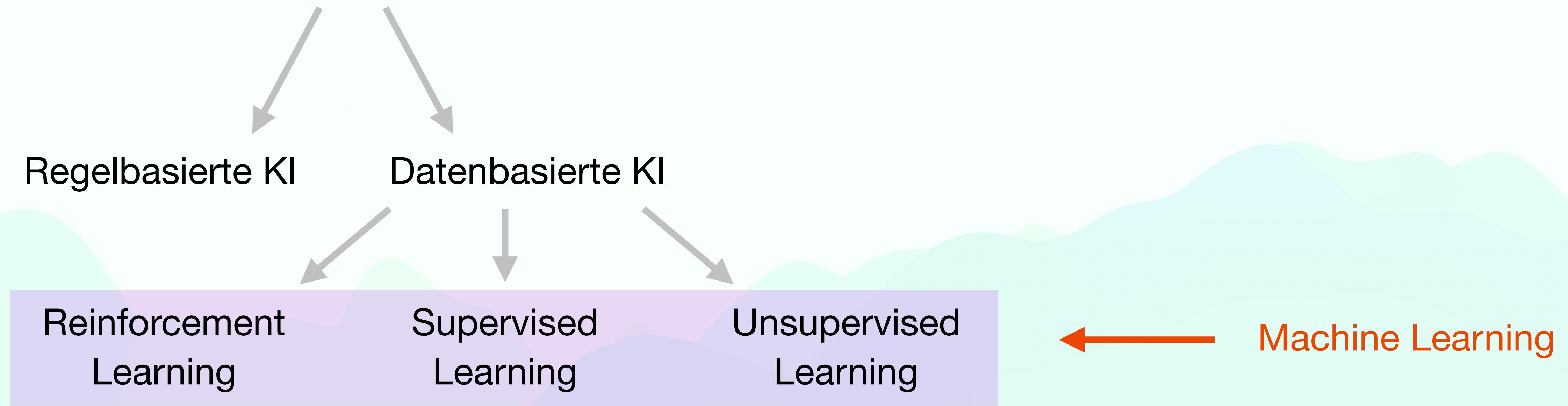
Regelbasierte KI

Datenbasierte KI

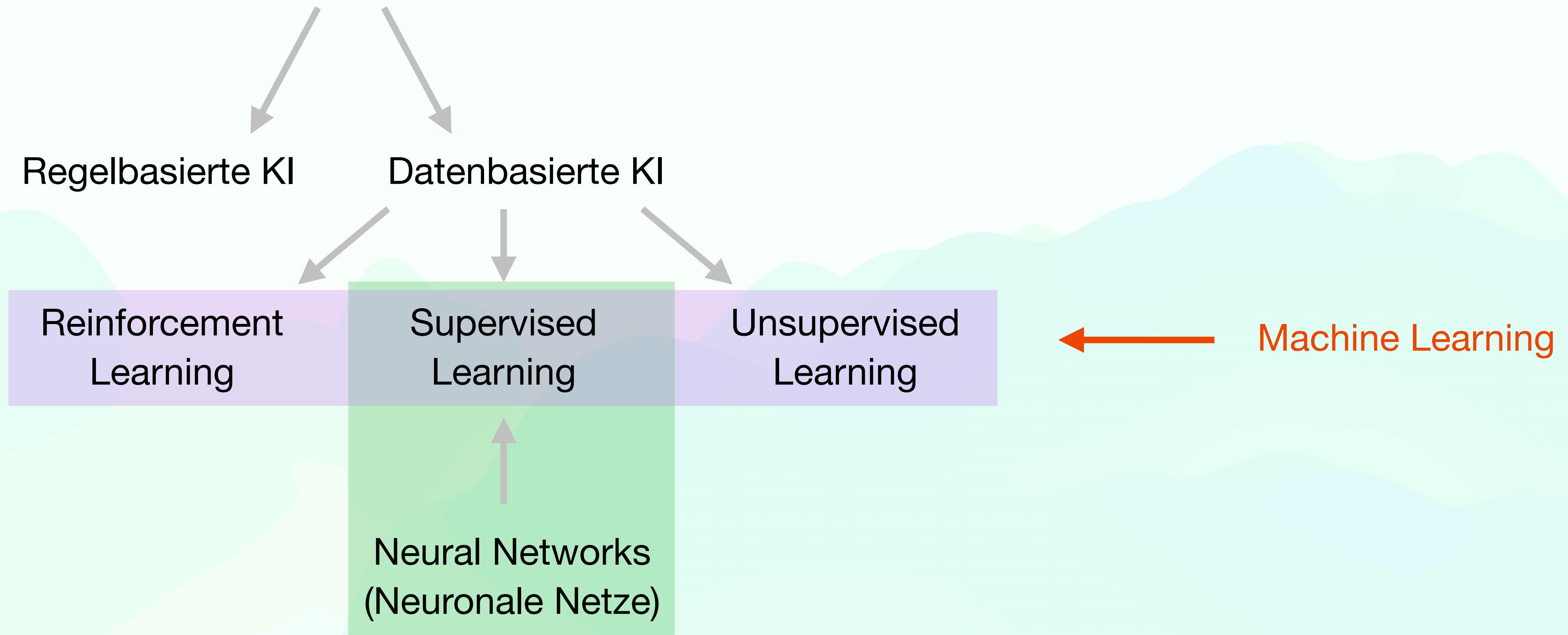
# „Künstliche Intelligenz“



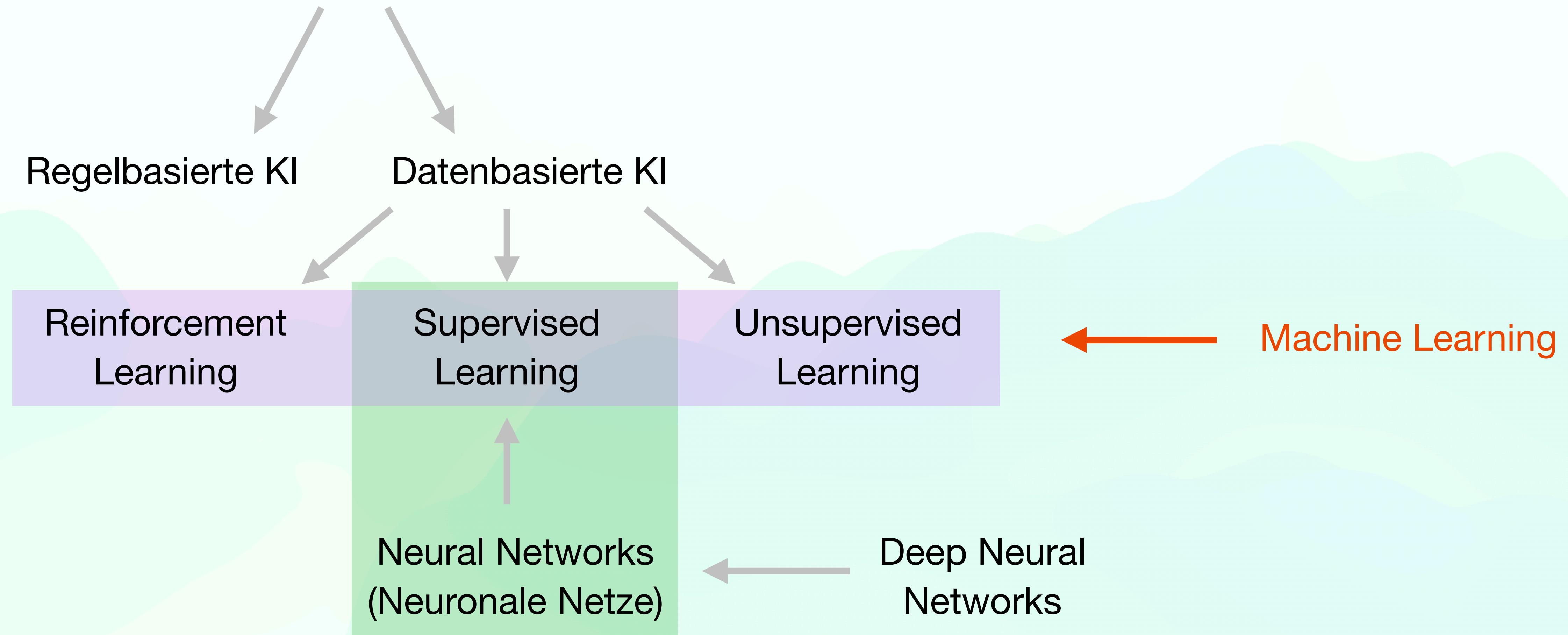
# „Künstliche Intelligenz“



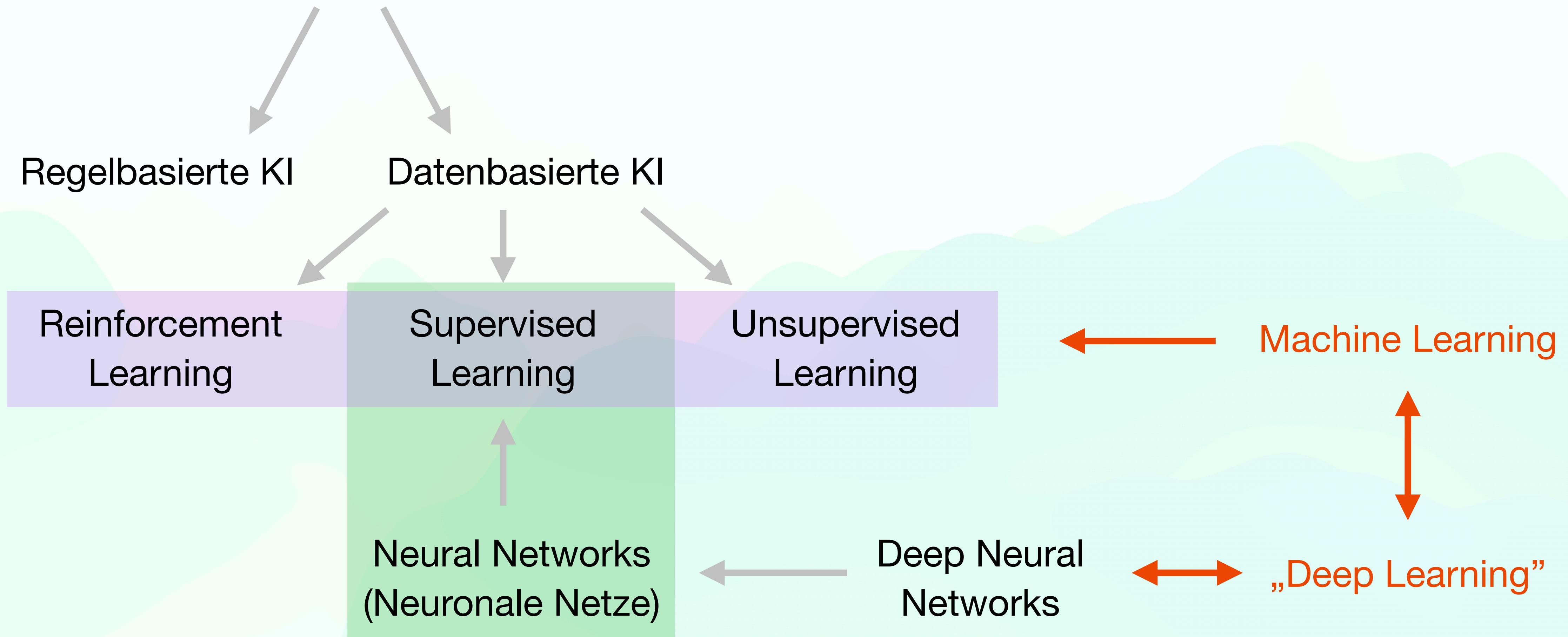
# „Künstliche Intelligenz“



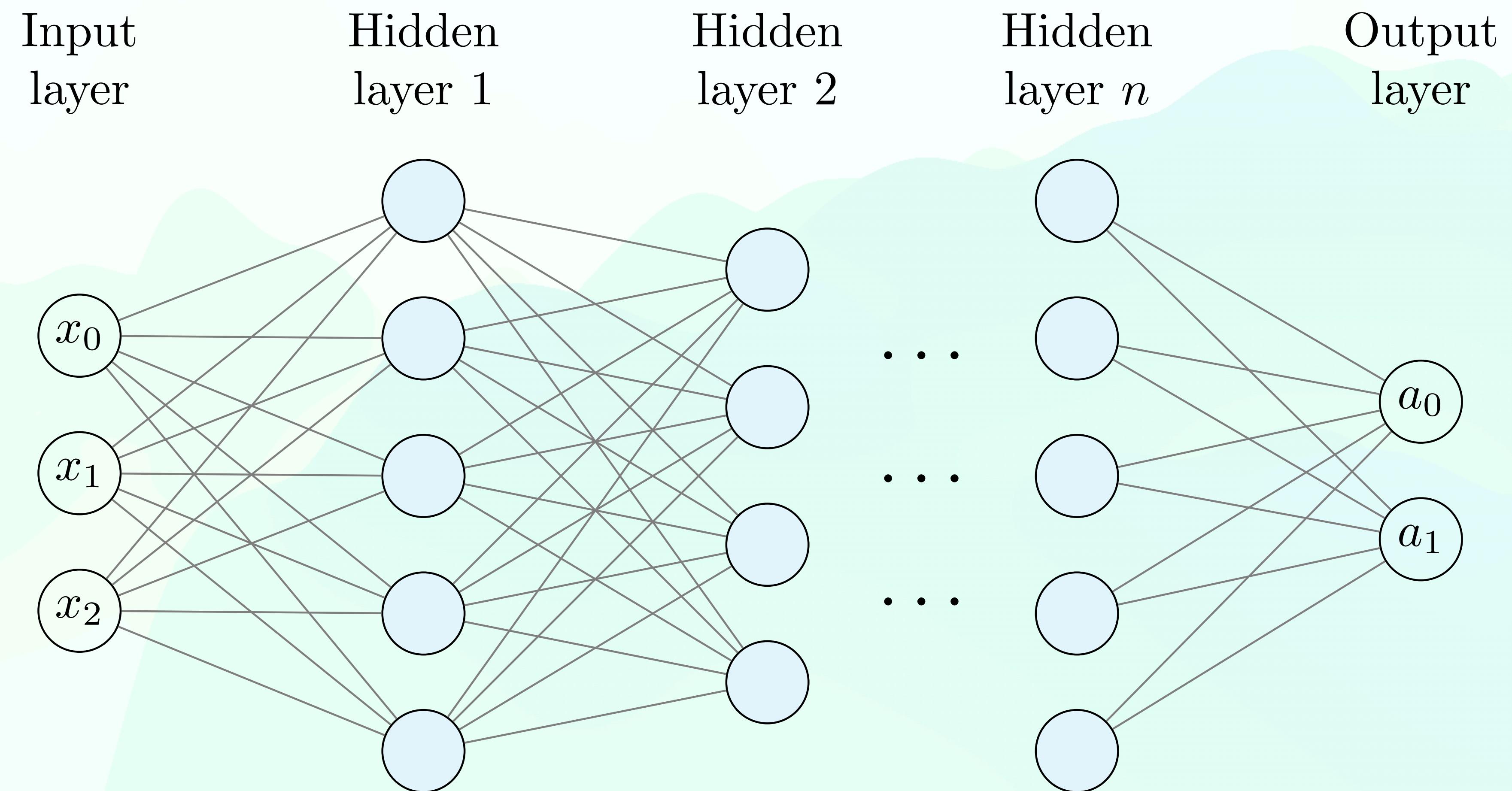
# „Künstliche Intelligenz“



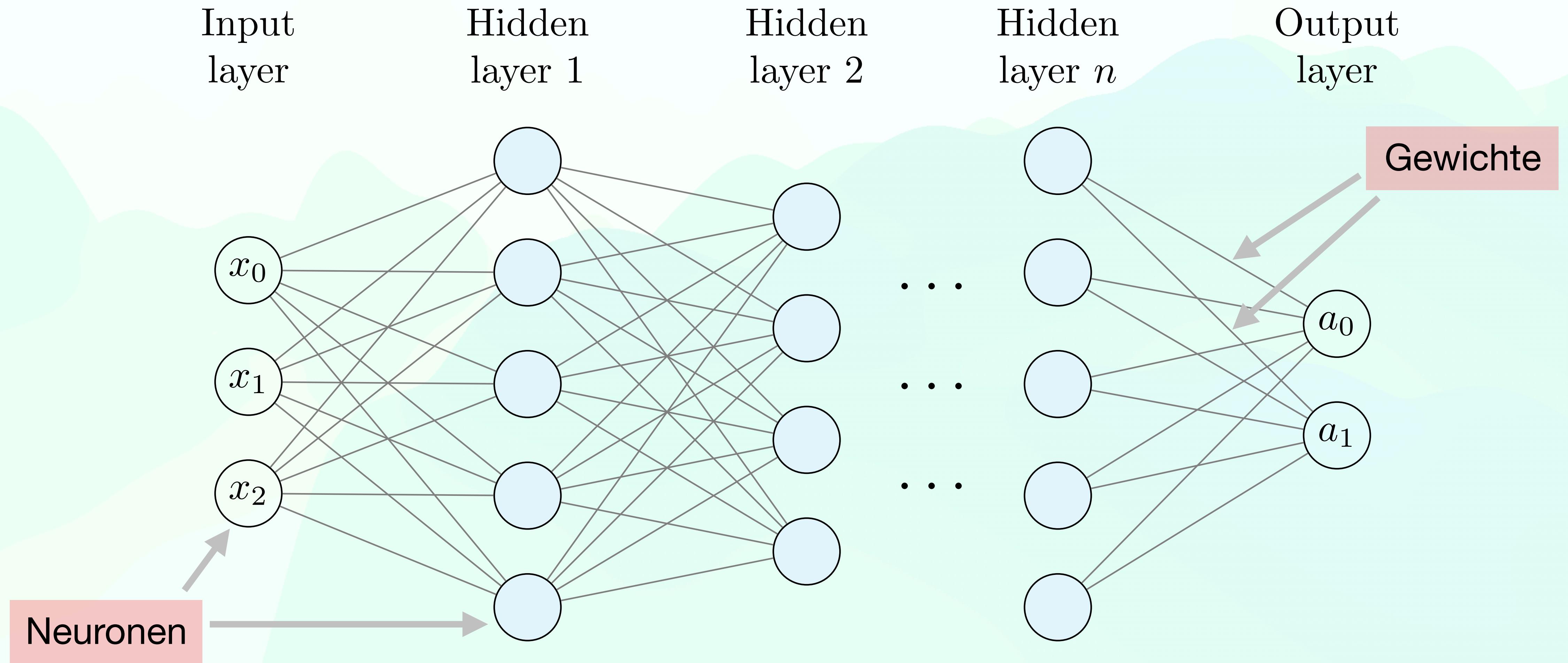
# „Künstliche Intelligenz“



# Überblick



# Überblick



# Überblick

Grundidee des **Supervised Learning** in einem neuronalen Netz:

# Überblick

Grundidee des **Supervised Learning** in einem neuronalen Netz:

- Gegeben: Ein Datensatz von vielen Trainingsbeispielen, jeweils mit Input und erwartetem Output

# Überblick

Grundidee des **Supervised Learning** in einem neuronalen Netz:

- Gegeben: Ein Datensatz von vielen Trainingsbeispielen, jeweils mit Input und erwartetem Output
- **Feedforward-Step:** Befülle den Input-Layer mit den Inputs eines Trainingsbeispiels und lass das Netzwerk einen Output vorhersagen (**Prediction**)

# Überblick

Grundidee des **Supervised Learning** in einem neuronalen Netz:

- Gegeben: Ein Datensatz von vielen Trainingsbeispielen, jeweils mit Input und erwartetem Output
- **Feedforward-Step:** Befülle den Input-Layer mit den Inputs eines Trainingsbeispiels und lass das Netzwerk einen Output vorhersagen (**Prediction**)
- **Cost Function/Loss Function:** Untersuche, wie groß die Abweichung zwischen erwartetem Output und vorhergesagtem Output ist

# Überblick

Grundidee des **Supervised Learning** in einem neuronalen Netz:

- Gegeben: Ein Datensatz von vielen Trainingsbeispielen, jeweils mit Input und erwartetem Output
- **Feedforward-Step:** Befülle den Input-Layer mit den Inputs eines Trainingsbeispiels und lass das Netzwerk einen Output vorhersagen (**Prediction**)
- **Cost Function/Loss Function:** Untersuche, wie groß die Abweichung zwischen erwartetem Output und vorhergesagtem Output ist
- **Backpropagation:**
  - Berechne, in welche Richtung (Gradient) die Gewichte angepasst werden müssen, um den Wert der Kostenfunktion zu minimieren (**Gradient Descent**)
  - Wiederhole all diese Schritte für alle weiteren Trainingsbeispiele
  - Bilde den Durchschnitt der Gradienten aller Trainingsbeispiele und passe die Gewichte entsprechend an

# Überblick

„Lernen“ in einem neuronalen Netz heißt, die **Gewichte** im Netz so **anzupassen**, dass der Fehler (die „**Kosten**“) im Durchschnitt über alle Trainingsbeispiele **minimiert** wird.

# „Hello world” in Python



```
if __name__ == '__main__':
    print("Hello world")
```

# „Hello world“ in Python



```
if __name__ == '__main__':
    print("Hello world")
```

Sorgt dafür, dass der nachfolgende Code nur aufgerufen wird, wenn unser Script als eigenständiges Programm aufgerufen wird.

# „Hello world“ in Python



```
if __name__ == '__main__':
    print("Hello world")
```

Ruft die **Funktion** `print` auf, die einen Text auf der Konsole ausgeben kann.

Als **Parameter** wird der **String** „Hello world“ übergeben.

# „Hello world“ in Python



```
if __name__ == '__main__':
    print("Hello world")
```

In Python ganz wichtig: Einrückung beachten!

# Überblick

- Wir betrachten **Fully Connected Neural Networks**:  
Jedes Neuron aus Layer  $l - 1$  ist mit jedem Neuron aus Layer  $l$  verbunden

# Überblick

- Wir betrachten **Fully Connected Neural Networks**:  
Jedes Neuron aus Layer  $l - 1$  ist mit jedem Neuron aus Layer  $l$  verbunden
- Grundsätzlich zwei mögliche Problemklassen, die mit NN gelöst werden können:
  - **Klassifikation:** *Zuordnung eines Inputs zu einer Kategorie*  
Beispiele: Handelt es sich bei einem Bild (Input) um das einer Katze (Kategorie 1), eines Hundes (Kategorie 2) oder eines Wellensittichs (Kategorie 3)?  
Ist eine E-Mail (Input) Spam (Kategorie 1) oder kein Spam (Kategorie 2)?

# Überblick

- Wir betrachten **Fully Connected Neural Networks**:  
Jedes Neuron aus Layer  $l - 1$  ist mit jedem Neuron aus Layer  $l$  verbunden
- Grundsätzlich zwei mögliche Problemklassen, die mit NN gelöst werden können:
  - **Klassifikation**: *Zuordnung eines Inputs zu einer Kategorie*  
Beispiele: Handelt es sich bei einem Bild (Input) um das einer Katze (Kategorie 1), eines Hundes (Kategorie 2) oder eines Wellensittichs (Kategorie 3)?  
Ist eine E-Mail (Input) Spam (Kategorie 1) oder kein Spam (Kategorie 2)?
  - **Regression**: *Vorhersage eines numerischen Outputs basierend auf dem Input*  
Beispiele: Bei gegebener Quadratmeterzahl, Lage und Sanierungszustand einer Wohnung (Inputs), wie hoch ist der Mietpreis (Output)?  
Wie schnell wird ein Mensch die 100 Meter laufen (Output), wenn von ihm Körpergröße, -gewicht, Geschlecht und Häufigkeit der sportlichen Aktivität (Inputs) bekannt sind?

# Überblick

$x_i$   $i$ -ter Eintrag im Inputvektor des Trainingsdatensatzes

$a_j^{(l)}$  Aktivierung des  $j$ -ten Neurons im Layer  $l$

$b^{(l)}$  Bias im Layer  $l$  zum Links-Rechts-Shift der Aktivierungsfunktion

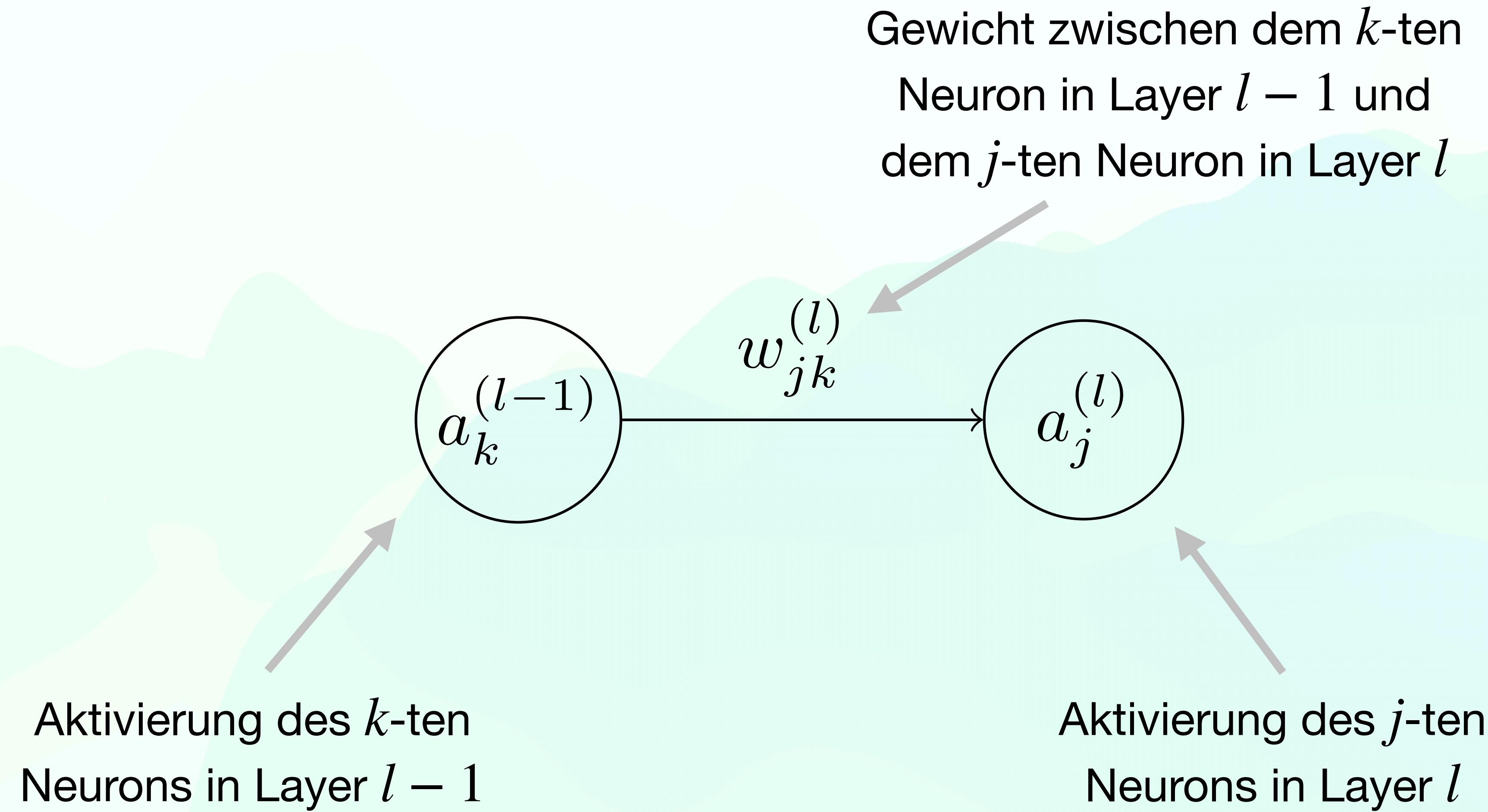
$w_{jk}^{(l)}$  Gewicht vom  $k$ -ten Neuron im Layer  $l - 1$  zum  $j$ -ten Neuron in Layer  $l$

$z_j^{(l)}$  Gewichtete Summe aller eingehenden Neuronenaktivierungen im  $j$ -ten Neuron im Layer  $l$

$\sigma^{(l)}$  Aktivierungsfunktion im Layer  $l$

$y_i$   $i$ -ter Eintrag im erwarteten Outputvektor des Trainingsdatensatzes

# Neuronen und Gewichte



# Neuronen und Gewichte

- Aktivierung eines Neurons  $a$  ist ein numerischer Wert

# Neuronen und Gewichte

- Aktivierung eines Neurons  $a$  ist ein numerischer Wert
  - **Im Input-Layer:** Ergibt sich direkt aus dem Input
    - Input muss entsprechend aufbereitet sein
    - Meist normalisiert auf Werte zwischen 0 und 1

# Neuronen und Gewichte

- Aktivierung eines Neurons  $a$  ist ein numerischer Wert
  - **Im Input-Layer:** Ergibt sich direkt aus dem Input
    - Input muss entsprechend aufbereitet sein
    - Meist normalisiert auf Werte zwischen 0 und 1
  - **In den Hidden Layers und im Output-Layer:** Ergibt sich aus gewichteter Summe  $z$  der Eingangs-Neuronen und Aktivierungsfunktion  $\sigma$ 
    - Output-Layer: bei Klassifikation zwischen 0 und 1 (interpretierbar als Wahrscheinlichkeiten), bei Regression beliebige Werte möglich

# Neuronen und Gewichte

- Aktivierung eines Neurons  $a$  ist ein numerischer Wert
  - **Im Input-Layer:** Ergibt sich direkt aus dem Input
    - Input muss entsprechend aufbereitet sein
    - Meist normalisiert auf Werte zwischen 0 und 1
  - **In den Hidden Layers und im Output-Layer:** Ergibt sich aus gewichteter Summe  $z$  der Eingangs-Neuronen und Aktivierungsfunktion  $\sigma$ 
    - Output-Layer: bei Klassifikation zwischen 0 und 1 (interpretierbar als Wahrscheinlichkeiten), bei Regression beliebige Werte möglich
- Gewicht  $w$  zwischen zwei Neuronen gibt an, wie groß der Einfluss der Aktivierung des ersten Neurons auf die des zweiten ist

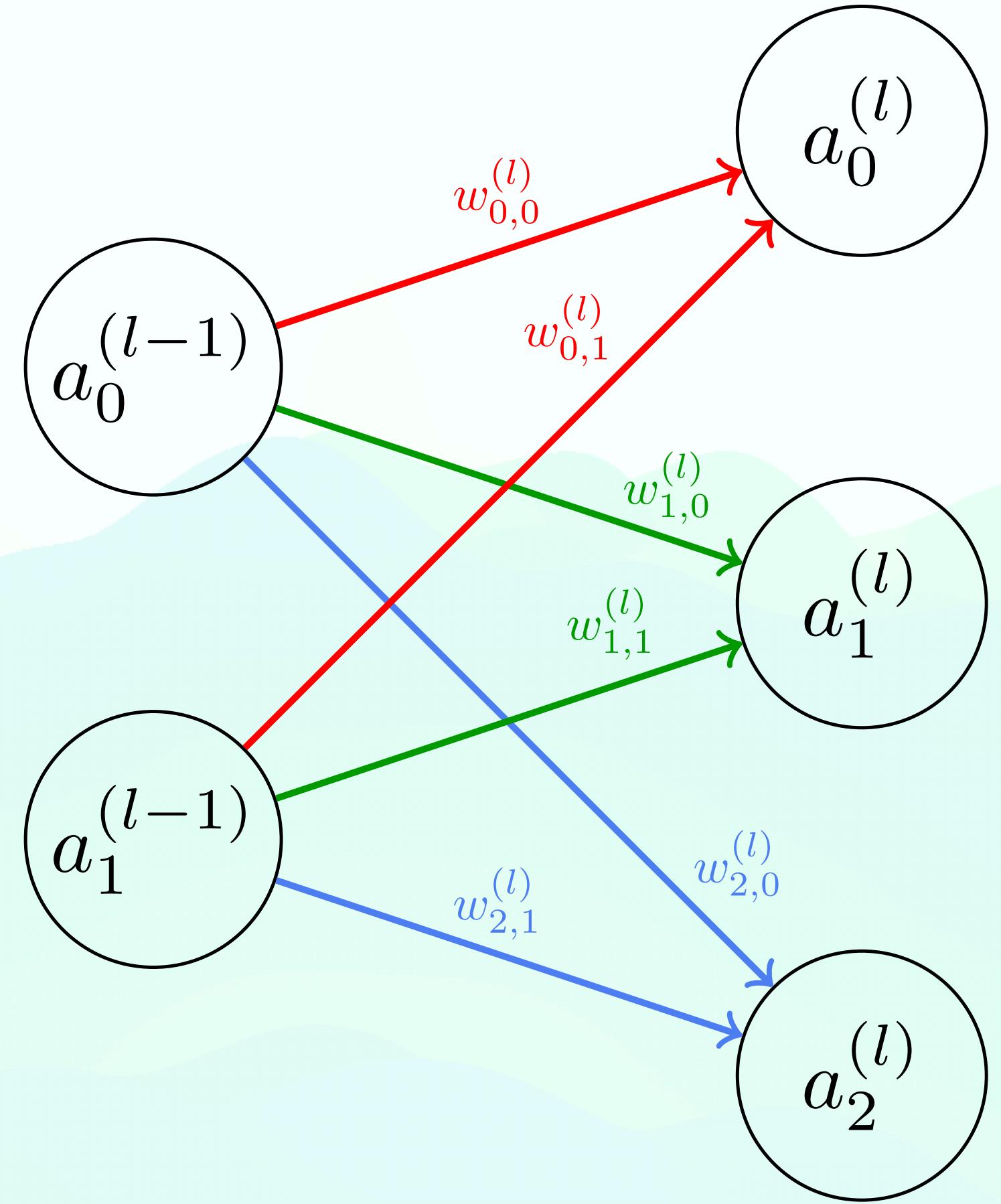
# Neuronen und Gewichte

- Gewichtete Summe  $z$  in einem Neuron:
  - Summe der Einflüsse aller Neuronen aus dem vorherigen Layer und dem Bias  $b$
  - Stärke dieses Einflusses bemisst sich am Gewicht zwischen den beiden Neuronen

# Neuronen und Gewichte

- Gewichtete Summe  $z$  in einem Neuron:
  - Summe der Einflüsse aller Neuronen aus dem vorherigen Layer und dem Bias  $b$
  - Stärke dieses Einflusses bemisst sich am Gewicht zwischen den beiden Neuronen

$$z_j^{(l)} = w_{j,0}^{(l)} a_0^{(l-1)} + w_{j,1}^{(l)} a_1^{(l-1)} + \dots + w_{j,n}^{(l)} a_n^{(l-1)} + b^{(l)}$$



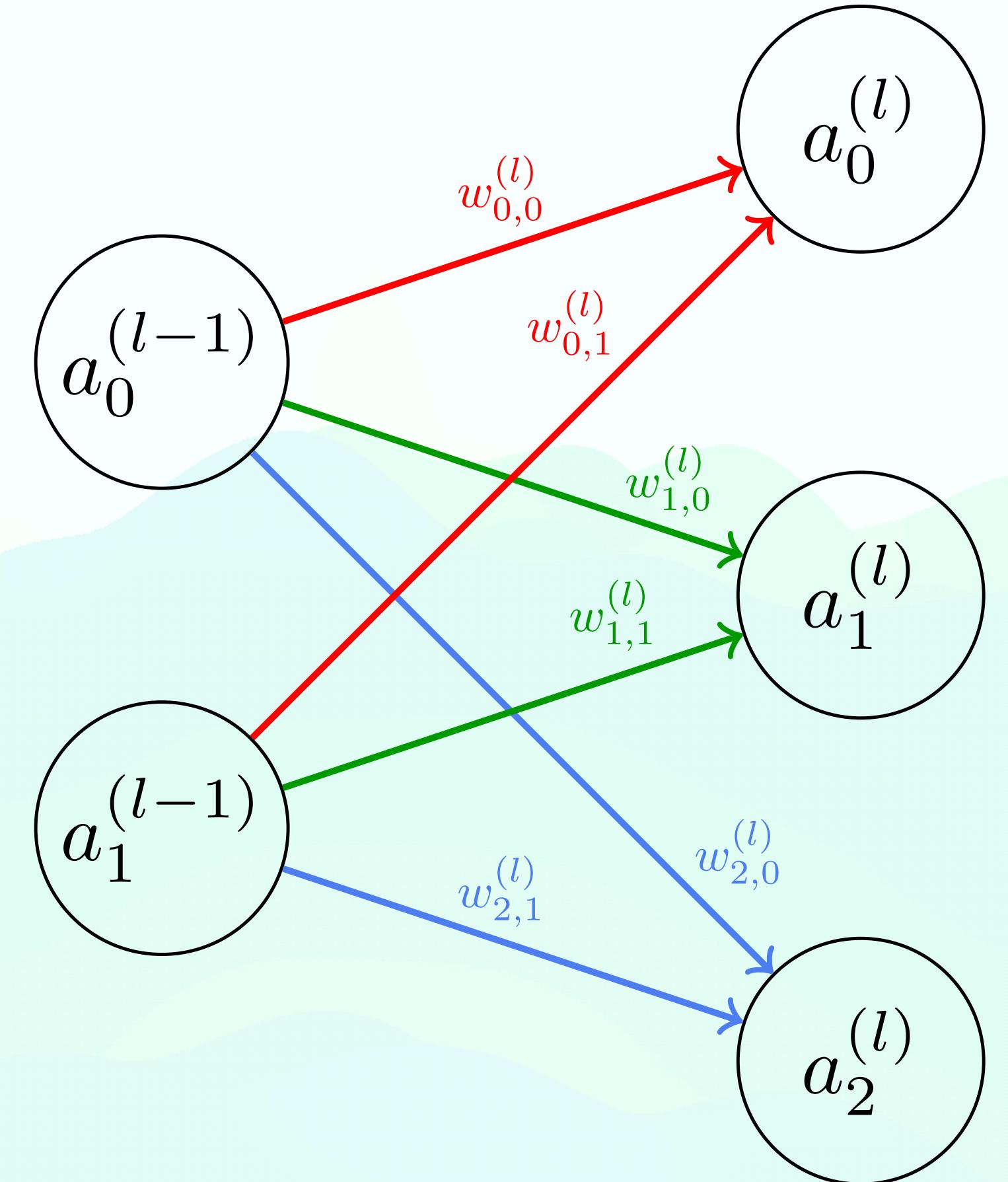
# Neuronen und Gewichte

- Gewichtete Summe  $z$  in einem Neuron:
  - Summe der Einflüsse aller Neuronen aus dem vorherigen Layer und dem Bias  $b$
  - Stärke dieses Einflusses bemisst sich am Gewicht zwischen den beiden Neuronen

$$z_j^{(l)} = w_{j,0}^{(l)} a_0^{(l-1)} + w_{j,1}^{(l)} a_1^{(l-1)} + \dots + w_{j,n}^{(l)} a_n^{(l-1)} + b^{(l)}$$

- Aktivierungsfunktion  $\sigma(z)$  in einem Neuron:
  - Berechnet die Aktivierung des Neurons basierend auf der gewichteten Summe  $z$

$$a_j^{(l)} = \sigma(z_j^{(l)})$$



# Neuronen und Gewichte

- **Trick:** Für den Bias fügen wir in jedem Layer (außer dem Output-Layer) einfach ein eigenes Bias-Neuron hinzu

# Neuronen und Gewichte

- **Trick:** Für den Bias fügen wir in jedem Layer (außer dem Output-Layer) einfach ein eigenes Bias-Neuron hinzu
- Bias-Neuronen: Feste Aktivierung von 1, keine eingehenden Gewichte
- Das Netz lernt dann die ausgehenden Gewichte der Bias-Neuronen auf die gleiche Weise wie die Gewichte zwischen allen anderen Neuronen

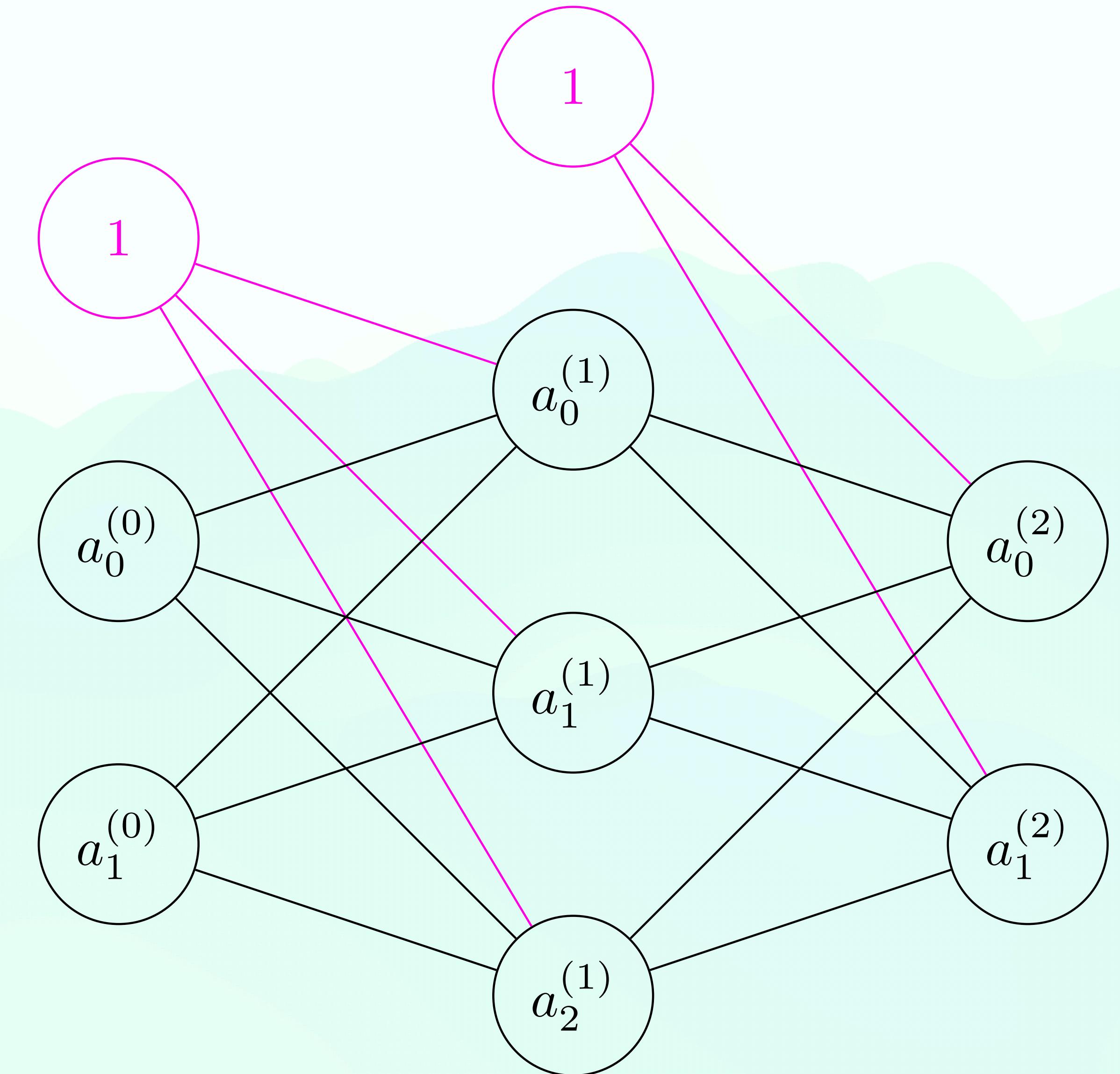
# Neuronen und Gewichte

- **Trick:** Für den Bias fügen wir in jedem Layer (außer dem Output-Layer) einfach ein eigenes Bias-Neuron hinzu
- Bias-Neuronen: Feste Aktivierung von 1, keine eingehenden Gewichte
- Das Netz lernt dann die ausgehenden Gewichte der Bias-Neuronen auf die gleiche Weise wie die Gewichte zwischen allen anderen Neuronen
- Damit entfällt für die Berechnung der gewichteten Summe der eigene Term für  $b^{(l)}$ :

$$z_j^{(l)} = w_{j,0}^{(l)} a_0^{(l-1)} + w_{j,1}^{(l)} a_1^{(l-1)} + \dots + w_{j,n}^{(l)} a_n^{(l-1)} = \sum_{k=0}^{n^{(l-1)}-1} w_{jk}^{(l)} a_k^{(l-1)}$$

# Neuronen und Gewichte

- Keine eingehenden Gewichte in **Biases**
- Daher keine Berechnung von gewichteten Summen und Aktivierungen in den Bias-Neuronen



# Klasse für neuronales Netz anlegen



```
class NeuralNetwork( ):
    def __init__(self, structure):
        self.structure = structure
        self.__init_layers()

    def __init_layers(self):
        pass
```

# Klasse für neuronales Netz anlegen



Deklariert eine **Klasse** namens NeuralNetwork. Sie enthält den „Bauplan“ für ein NN-**Objekt**, also alle seine **Eigenschaften** und **Methoden**.

```
class NeuralNetwork( ):
    def __init__(self, structure):
        self.structure = structure
        self.__init_layers()

    def __init_layers(self):
        pass
```

# Klasse für neuronales Netz anlegen



```
class NeuralNetwork( ):
    def __init__(self, structure):
        self.structure = structure
        self.__init_layers()

    def __init_layers(self):
        pass
```

Der **Konstruktor** der Klasse. Er wird beim Erzeugen des NN-Objekts aufgerufen, und ihm muss ein **Parameter** namens **structure** übergeben werden. Dieser Parameter soll später die Anzahl der **Layer** und ihrer **Neuronen** bestimmen.

# Klasse für neuronales Netz anlegen



```
class NeuralNetwork():
    def __init__(self, structure):
        self.structure = structure
        self.__init_layers()

    def __init_layers(self):
        pass
```

Im Parameter `self` bekommt jede Objektmethode in Python eine Referenz auf die eigene Objektinstanz übergeben. Darüber kann die Methode auf die Instanz zugreifen und darauf z. B. andere Methoden des Objekts aufrufen oder Eigenschaften des Objekts Werte zuweisen.

# Klasse für neuronales Netz anlegen



```
class NeuralNetwork():
    def __init__(self, structure):
        self.structure = structure
        self.__init_layers()

    def __init_layers(self):
        pass
```

Im Konstruktor wird die **Eigenschaft** `structure` angelegt und ihr der Wert des übergebenen, gleichnamigen **Parameters** zugewiesen.

# Klasse für neuronales Netz anlegen



```
class NeuralNetwork():
    def __init__(self, structure):
        self.structure = structure
        self.__init_layers()

    def __init_layers(self):
        pass
```

Danach rufen wir eine **Methode** namens `__init_layers` auf. Ein Methoden- oder Funktionsaufruf besteht immer aus dem Namen der Methode oder Funktion, gefolgt von der Parameterliste in runden Klammern. In diesem Fall übergeben wir der Methode keine Parameter.

# Klasse für neuronales Netz anlegen



```
class NeuralNetwork( ):
    def __init__(self, structure):
        self.structure = structure
        self.__init_layers()
```

```
def __init_layers(self):
    pass
```

Die **Methode `__init_layers`** wird deklariert.  
Die beiden Unterstriche am Beginn des Namens  
zeigen eine **private** Methode an. Sie kann nicht  
von außerhalb der Klasse aufgerufen werden.

# Klasse für neuronales Netz anlegen



```
class NeuralNetwork( ):
    def __init__(self, structure):
        self.structure = structure
        self.__init_layers()
```

```
def __init_layers(self):
    pass
```

Das `pass` zeigt an, dass der Inhalt dieser Methode vorerst leer ist. Es hat keine inhaltliche Funktion, aber es wäre ein Syntaxfehler, den Rumpf der Methode leer zu lassen.  
Wir fügen den Inhalt der Methode später hinzu.

# Objektinstanz für neuronales Netz erzeuaen



```
if __name__ == '__main__':
    nn = NeuralNetwork(structure=[3, 4, 2])
```

# Objektinstanz für neuronales Netz erzeugen



```
if __name__ == '__main__':
    nn = NeuralNetwork(structure=[3, 4, 2])
```

Erzeugt ein neues **Objekt** der `NeuralNetwork`-Klasse und weist es der **Variable** `nn` zu. Als `structure`-Parameter übergeben wir eine **Liste** mit drei Zahlen.

# Objektinstanz für neuronales Netz erzeugen



3 Neuronen im Input-Layer, 4 Neuronen im Hidden Layer, 2 Neuronen im Output-Layer

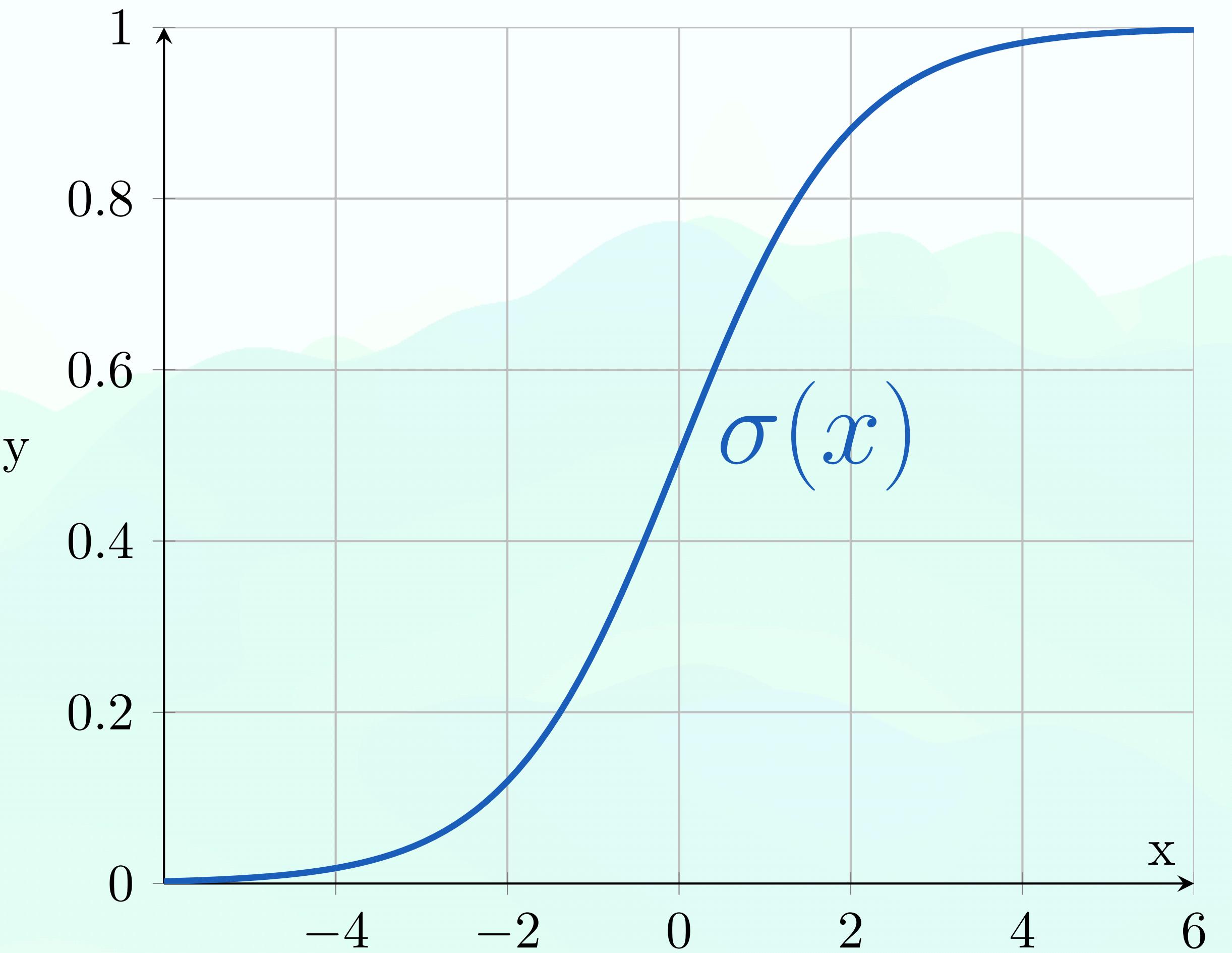
```
if __name__ == '__main__':
    nn = NeuralNetwork(structure=[3, 4, 2])
```

# Aktivierungsfunktionen

- Klassisch: Die Sigmoid-Funktion

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

- Abbildung einer reellen Zahl auf den Wertebereich 0 bis 1

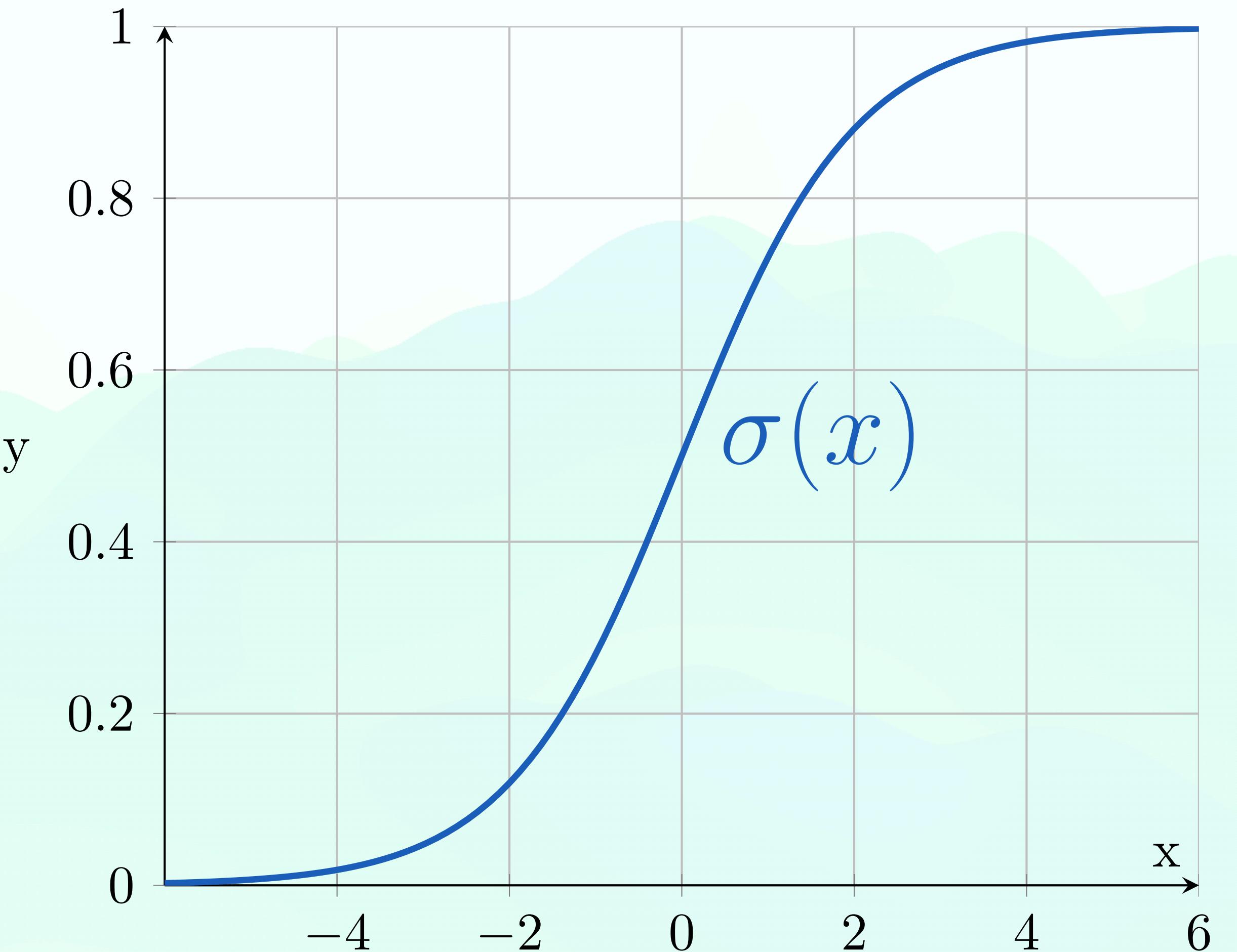


# Aktivierungsfunktionen

- Klassisch: Die Sigmoid-Funktion

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

- Abbildung einer reellen Zahl auf den Wertebereich 0 bis 1
- Problem: **Vanishing Gradients**
  - Nur sehr schmaler Bereich, in dem sich die Werte signifikant von 0 und 1 unterscheiden
  - Daher in den „Außenbereichen“ nur langsames Lernen möglich

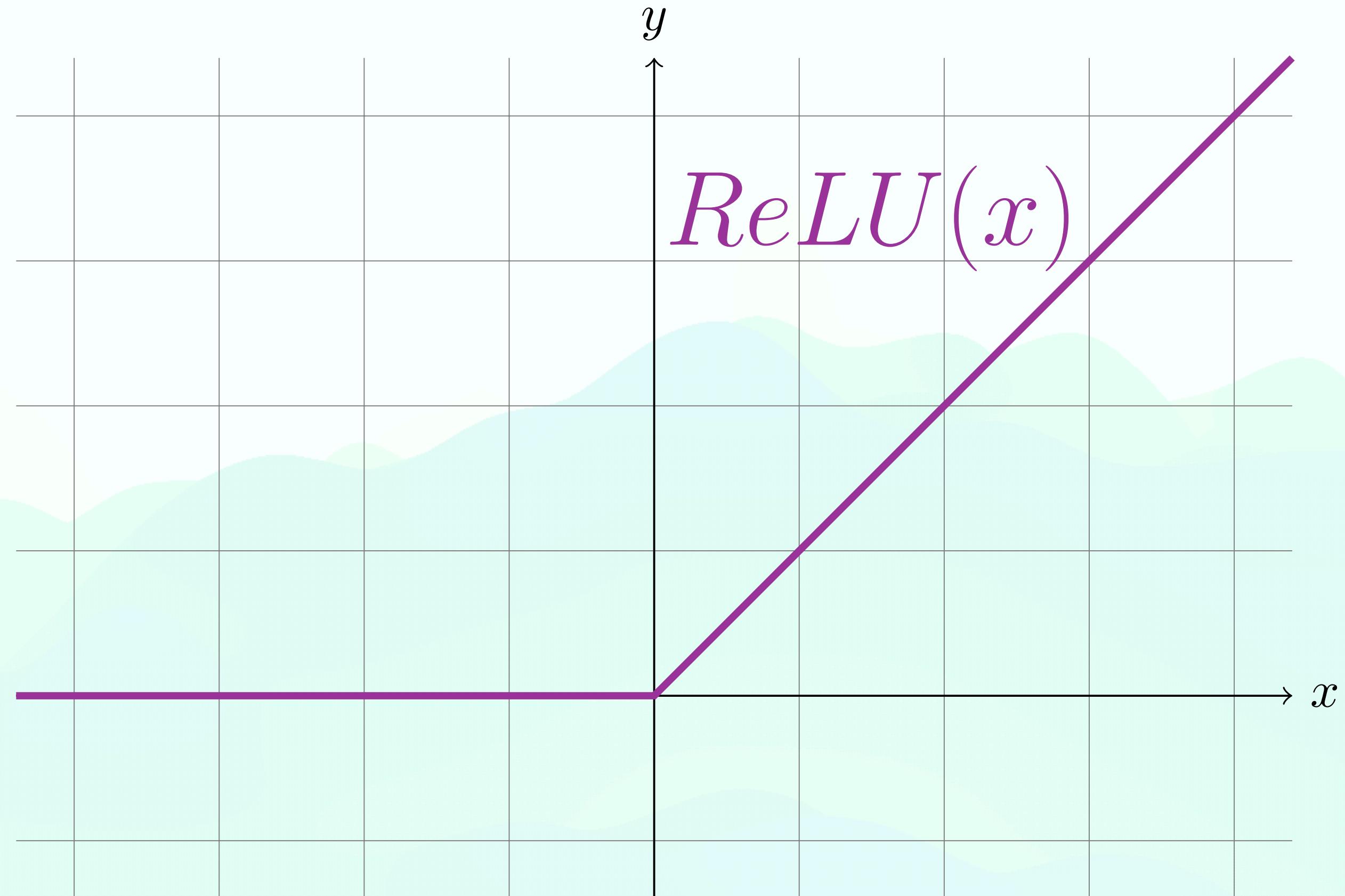


# Aktivierungsfunktionen

- ReLU (Rectified Linear Unit)

$$ReLU(x) = \max(0, x) = \begin{cases} x & \text{wenn } x > 0 \\ 0 & \text{sonst} \end{cases}$$

- Produziert Werte im Bereich  $y \geq 0$



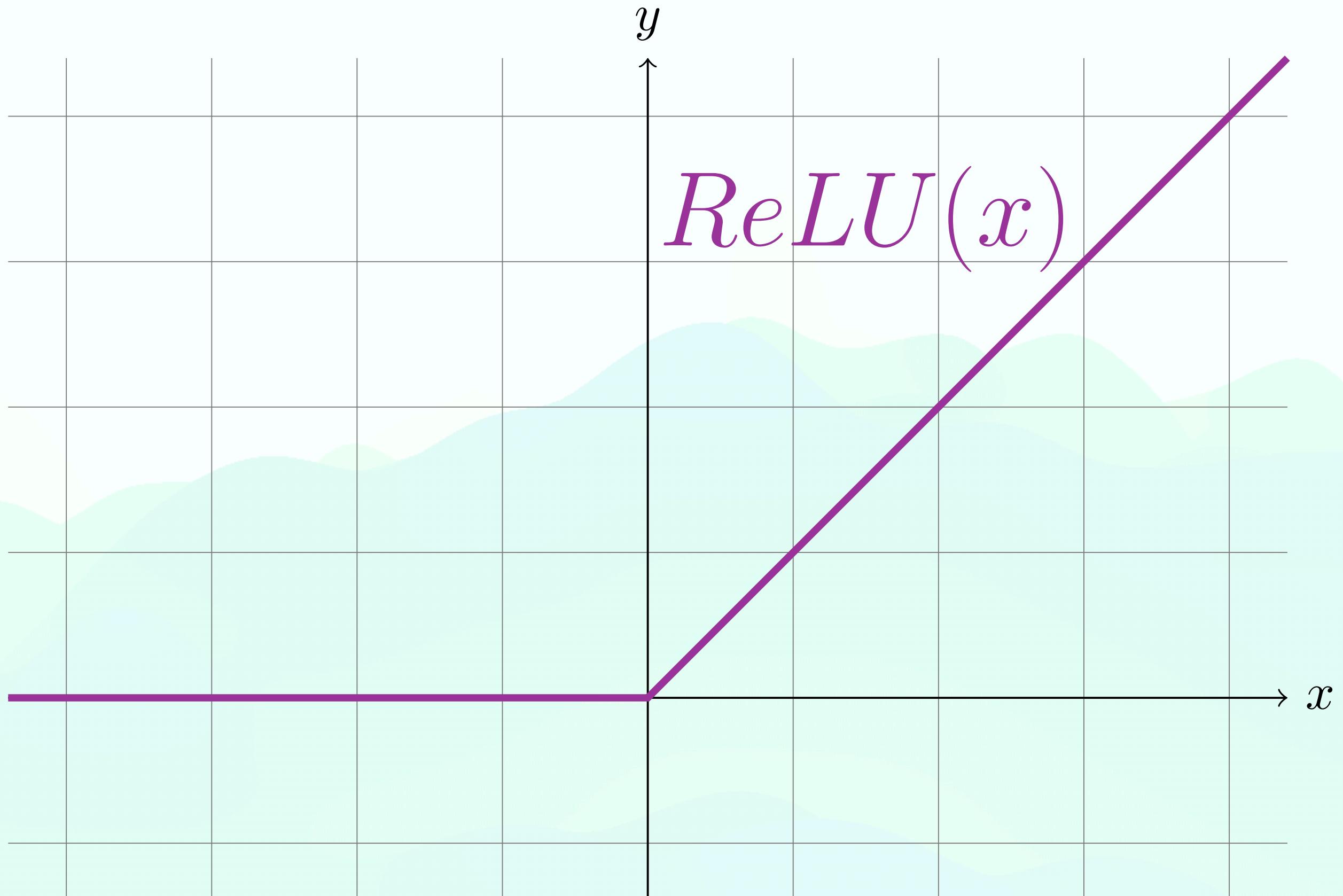
# Aktivierungsfunktionen

- ReLU (**Rectified Linear Unit**)

$$ReLU(x) = \max(0, x) = \begin{cases} x & \text{wenn } x > 0 \\ 0 & \text{sonst} \end{cases}$$

- Produziert Werte im Bereich  $y \geq 0$
- Schnelles Lernen durch konstanten Gradienten (kein **Vanishing Gradient**)
- Effiziente Backpropagation durch triviale Ableitung:

$$ReLU'(x) = \begin{cases} 1 & \text{wenn } x > 0 \\ 0 & \text{sonst} \end{cases}$$



# Aktivierungsfunktionen anlegen

```
● ● ●  
import numpy  
  
def relu(z):  
    return numpy.maximum(z, 0)  
  
def sigmoid(z):  
    return 1 / (1 + numpy.exp(-z))  
  
def identity(z):  
    return z
```

# Aktivierungsfunktionen anlegen



```
import numpy
```

```
def relu(z):  
    return numpy.maximum(z, 0)
```

```
def sigmoid(z):  
    return 1 / (1 + numpy.exp(-z))
```

```
def identity(z):  
    return z
```

Wir importieren das über PIP installierte Package **numpy**, das uns viele mathematische Hilfsfunktionen anbietet. Erst durch den **import** können wir es auch in unserem Code benutzen.

# Aktivierungsfunktionen anlegen



```
import numpy

def relu(z):
    return numpy.maximum(z, 0)

def sigmoid(z):
    return 1 / (1 + numpy.exp(-z))

def identity(z):
    return z
```

Implementierung der ReLU-Funktion. Sie bekommt den **Parameter**  $z$  übergeben und gibt mit dem Schlüsselwort **return** das Maximum aus  $z$  und 0 zurück.

# Aktivierungsfunktionen anlegen

```
import numpy

def relu(z):
    return numpy.maximum(z, 0)

def sigmoid(z):
    return 1 / (1 + numpy.exp(-z))

def identity(z):
    return z
```

Auch die Funktion sigmoid erwartet einen Parameter  $z$ . Mithilfe der Funktion `numpy.exp` berechnen wir die Potenz  $e^{-z}$  und geben das Ergebnis mit `return` zurück.

# Aktivierungsfunktionen anlegen

```
import numpy

def relu(z):
    return numpy.maximum(z, 0)

def sigmoid(z):
    return 1 / (1 + numpy.exp(-z))

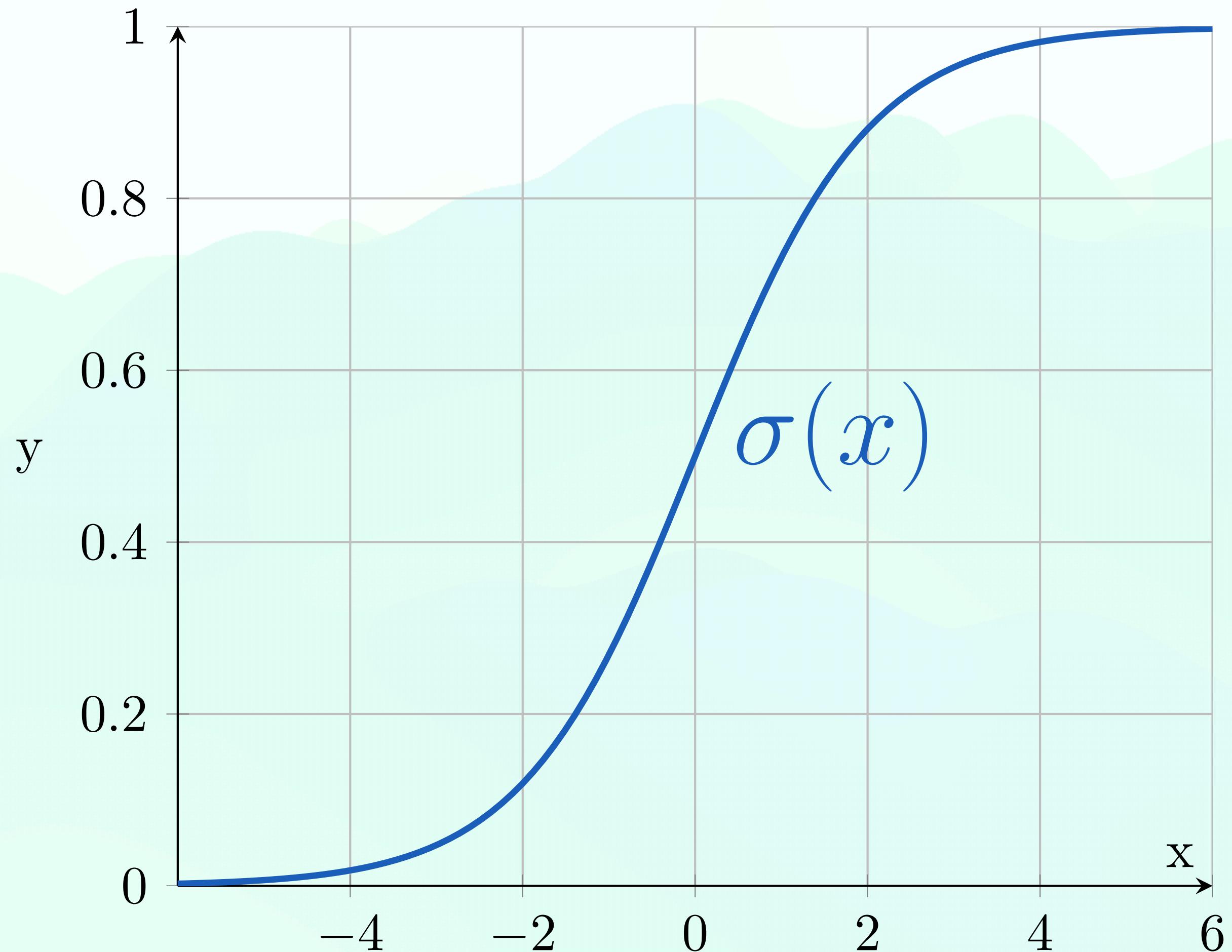
def identity(z):
    return z
```

Wir implementieren auch noch die Identitätsfunktion, die den übergebenen Wert einfach unverändert zurückgibt. Auch diese Aktivierungsfunktion kann für Testzwecke interessant sein.

# Bias

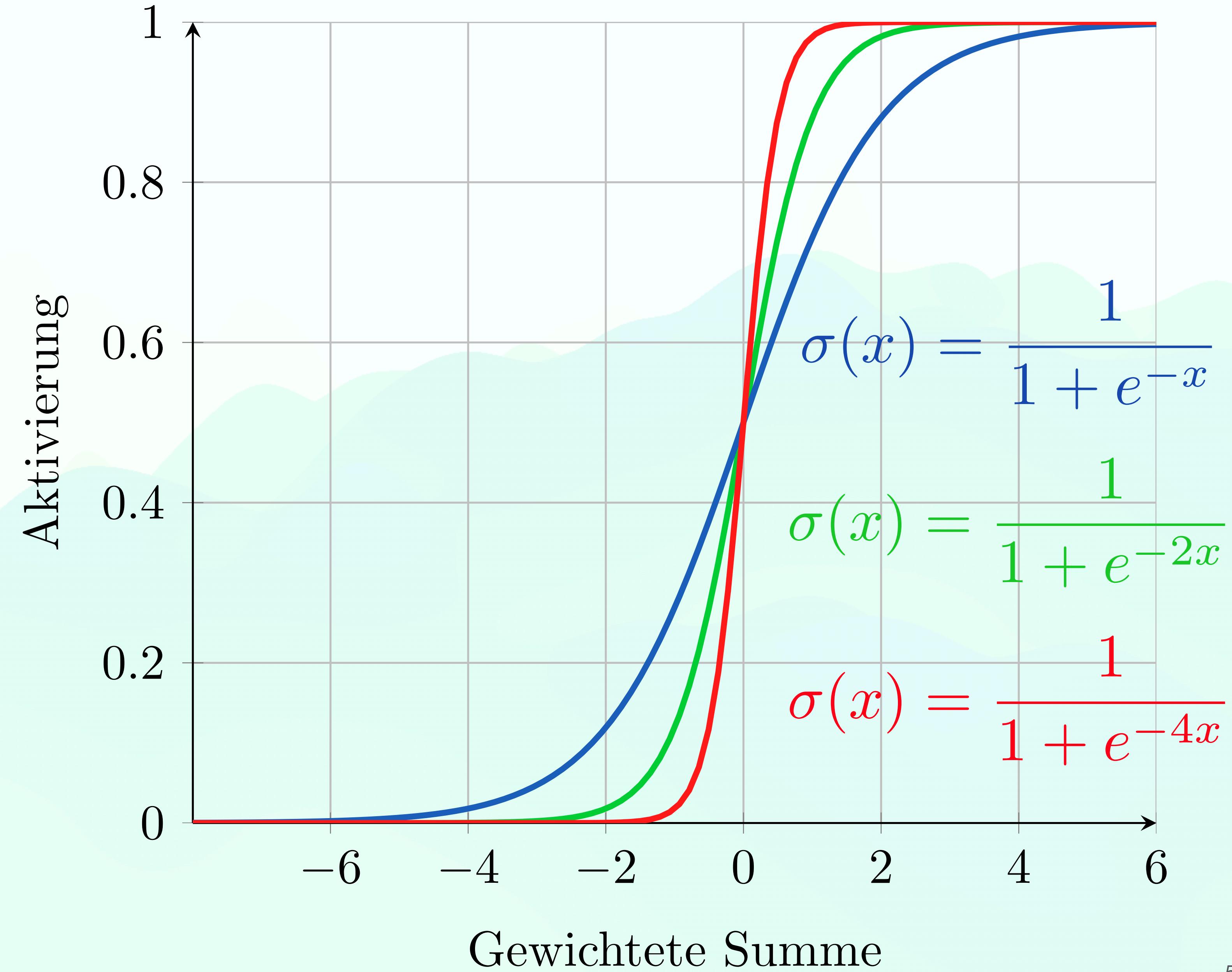
- Zur Erinnerung: An der  $x$ -Achse steht die gewichtete Summe, an der  $y$ -Achse die Aktivierung
- Nur um  $x = 0$  herum verändert sich die Aktivierung signifikant
- Analog bei ReLU: Erst ab einer gewichteten Summe von  $z > 0$  beginnt die Aktivierung
- In vielen Anwendungsfällen soll die Aktivierung aber früher oder später beginnen
- Wir brauchen eine Möglichkeit, die Kurve der Aktivierungsfunktion zu **verschieben**

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



# Bias

- Multiplikation des Inputs  $x$  mit einem Faktor (z. B. einem Gewicht  $w$ ) macht die Kurve nur **steiler**

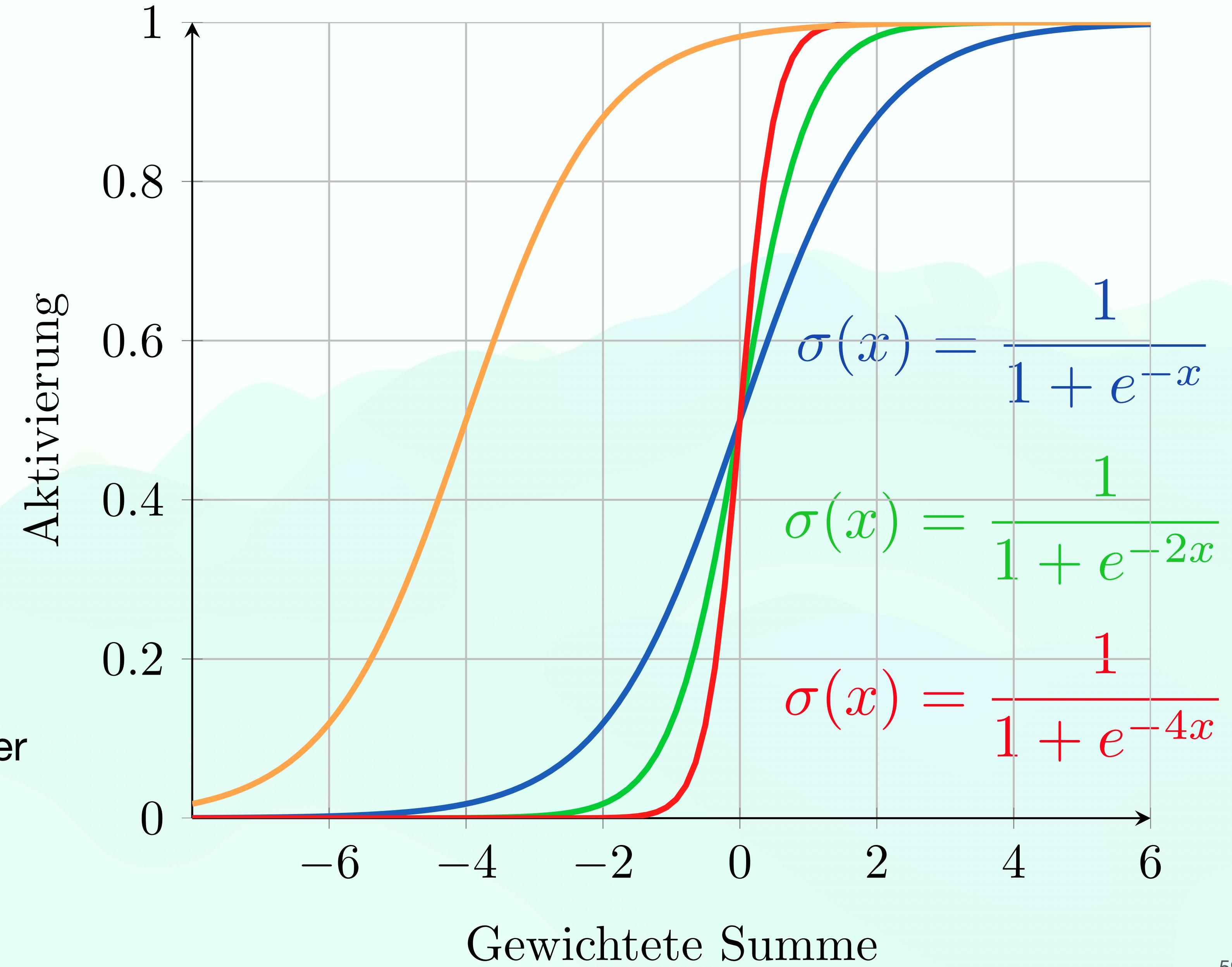


# Bias

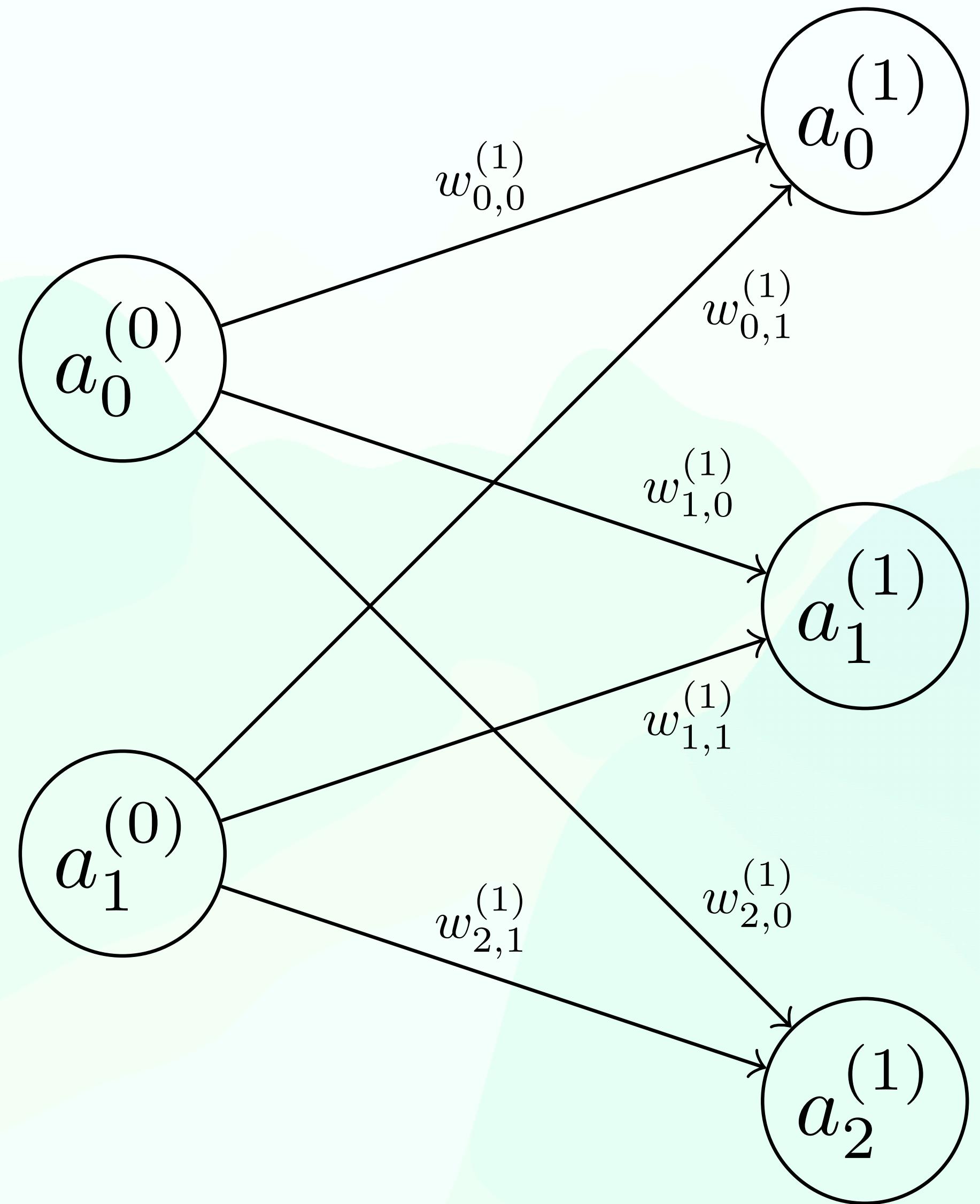
- Multiplikation des Inputs  $x$  mit einem Faktor (z. B. einem Gewicht  $w$ ) macht die Kurve nur **steiler** oder **flacher**

$$\sigma(x) = \frac{1}{1 + e^{-x-4}}$$


Nur Addition (oder Subtraktion) der gewichteten Summe mit einer Konstante kann die Kurve in  $x$ -Richtung **verschieben**

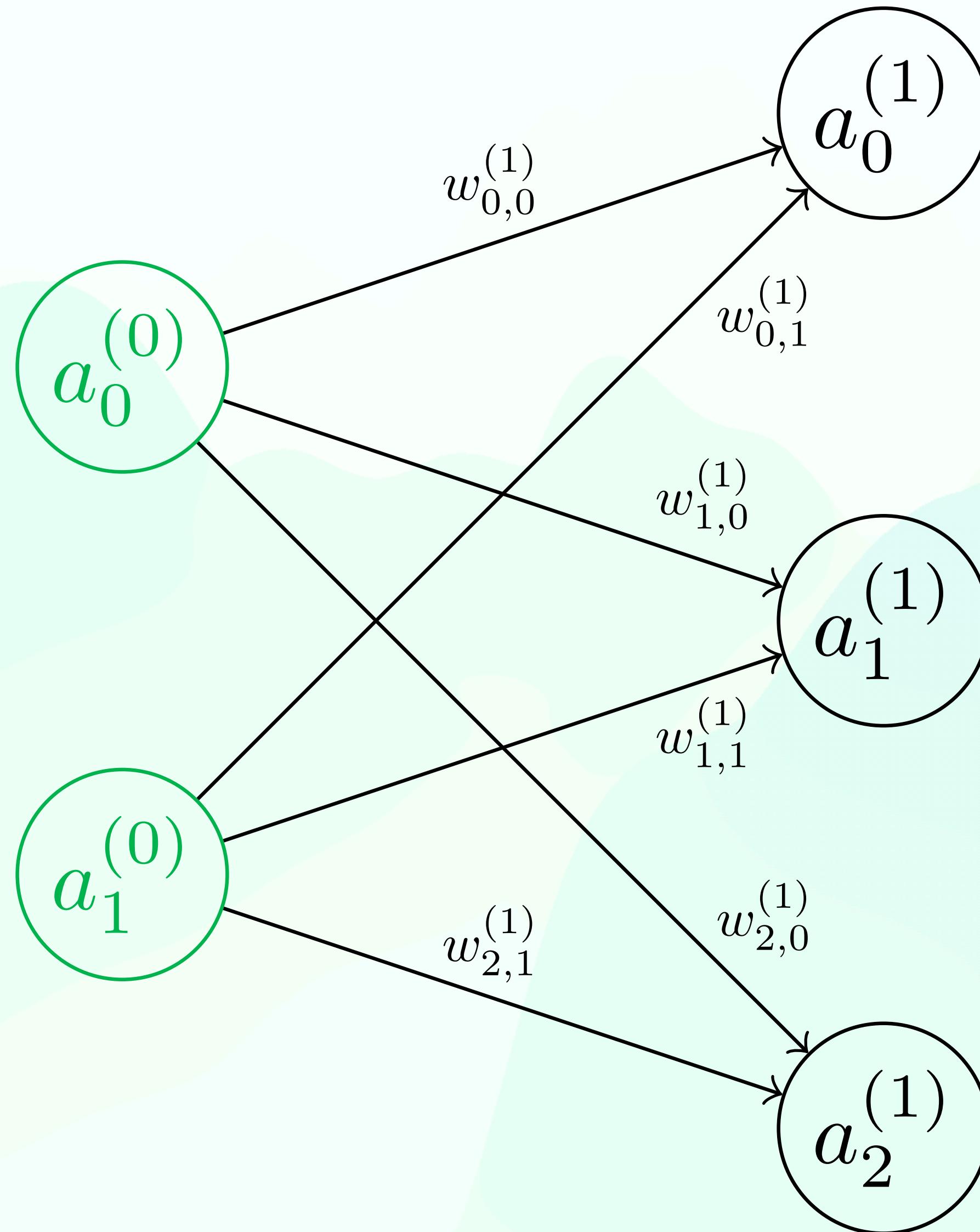


# Notation



Für Notation und Implementierung in Python enorme  
Hilfe: Vektor- und Matrixschreibweise

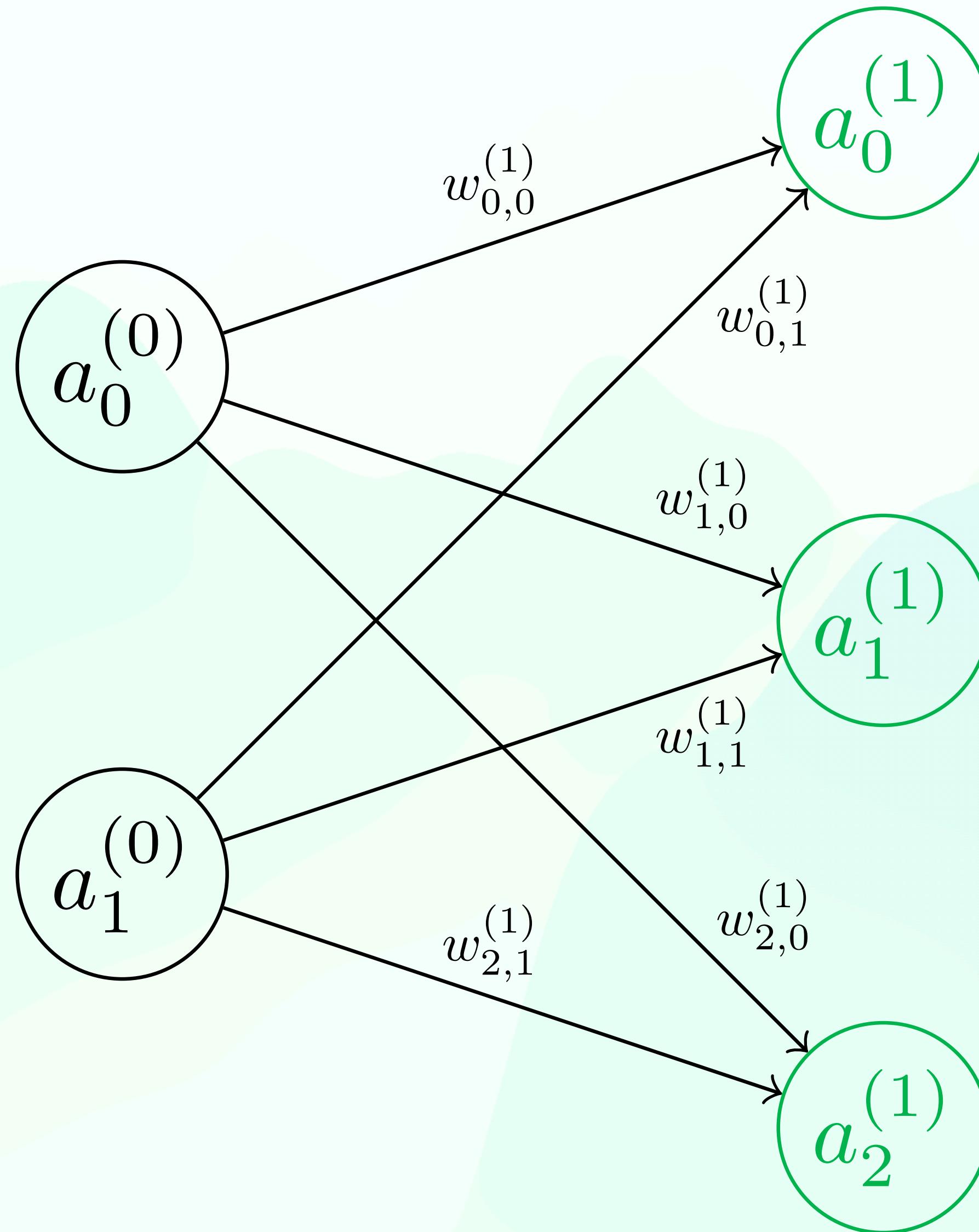
# Notation – Aktivierungsvektoren



Für Notation und Implementierung in Python enorme  
Hilfe: Vektor- und Matrixschreibweise

$$A^{(0)} = \begin{pmatrix} a_0^{(0)} \\ a_1^{(0)} \end{pmatrix}$$

# Notation – Aktivierungsvektoren



Für Notation und Implementierung in Python enorme  
Hilfe: Vektor- und Matrixschreibweise

$$A^{(1)} = \begin{pmatrix} a_0^{(1)} \\ a_1^{(1)} \\ a_2^{(1)} \end{pmatrix}$$

# Neuronenlayer initialisieren



```
def __init_layers(self):
    self.activations = []
    self.weighted_sums = []

    for l in range(len(self.structure)):
        is_output_layer = l == len(self.structure) - 1
        n_activations = self.structure[l] if is_output_layer else self.structure[l] + 1
        n_weighted_sums = self.structure[l]
        layer_weighted_sums = numpy.zeros((n_weighted_sums, 1))
        layer_activations = numpy.ones((n_activations, 1))

        self.weighted_sums.append(layer_weighted_sums)
        self.activations.append(layer_activations)
```

# Neuronenlayer initialisieren



```
def __init_layers(self):
    self.activations = []
    self.weighted_sums = []

    for l in range(len(self.structure)):
        is_output_layer = l == len(self.structure) - 1
        n_activations = self.structure[l] if is_output_layer else self.structure[l] + 1
        n_weighted_sums = self.structure[l]
        layer_weighted_sums = numpy.zeros((n_weighted_sums, 1))
        layer_activations = numpy.ones((n_activations, 1))

        self.weighted_sums.append(layer_weighted_sums)
        self.activations.append(layer_activations)
```

Wir erzeugen jeweils Listen für die Aktivierungen und für die gewichteten Summen der einzelnen Layer. Durch das `self.`-Präfix legen wir sie als **Objekteigenschaften** und nicht nur als lokale **Variable** der **Methode** an.

# Neuronenlayer initialisieren



```
def __init_layers(self):
    self.activations = []
    self.weighted_sums = []

    for l in range(len(self.structure)):
        is_output_layer = l == len(self.structure) - 1
        n_activations = self.structure[l] if is_output_layer else self.structure[l] + 1
        n_weighted_sums = self.structure[l]
        layer_weighted_sums = numpy.zeros((n_weighted_sums, 1))
        layer_activations = numpy.ones((n_activations, 1))

        self.weighted_sums.append(layer_weighted_sums)
        self.activations.append(layer_activations)
```

Die **for-Schleife** führt den nachfolgenden Code wiederholt aus, und zwar so oft, wie es der Aufruf der `range`-Funktion angibt.

# Neuronenlayer initialisieren



```
def __init_layers(self):
    self.activations = []
    self.weighted_sums = []

    for l in range(len(self.structure)):
        is_output_layer = l == len(self.structure) - 1
        n_activations = self.structure[l] if is_output_layer else self.structure[l] + 1
        n_weighted_sums = self.structure[l]
        layer_weighted_sums = numpy.zeros((n_weighted_sums, 1))
        layer_activations = numpy.ones((n_activations, 1))

        self.weighted_sums.append(layer_weighted_sums)
        self.activations.append(layer_activations)
```

In diesem Fall richtet sich die Anzahl der Wiederholungen nach der Anzahl der Layer, die in der Eigenschaft **structure** vorgegeben ist.

**structure** ist eine **Liste**, und mit **len** erhalten wir ihre Länge (Anzahl der Elemente).

# Neuronenlayer initialisieren



```
def __init_layers(self):
    self.activations = []
    self.weighted_sums = []

    for l in range(len(self.structure)):
        is_output_layer = l == len(self.structure) - 1
        n_activations = self.structure[l] if is_output_layer else self.structure[l] + 1
        n_weighted_sums = self.structure[l]
        layer_weighted_sums = numpy.zeros((n_weighted_sums, 1))
        layer_activations = numpy.ones((n_activations, 1))

        self.weighted_sums.append(layer_weighted_sums)
        self.activations.append(layer_activations)
```

Die **Laufvariable** `l` fängt bei 0 an und zählt bei jedem Schleifendurchlauf um 1 hoch. Ihr Maximalwert ist die **Anzahl der Layer minus 1**, danach bricht die Schleife ab.

# Neuronenlayer initialisieren



```
def __init_layers(self):
    self.activations = []
    self.weighted_sums = []

    for l in range(len(self.structure)):
        is_output_layer = l == len(self.structure) - 1
        n_activations = self.structure[l] if is_output_layer else self.structure[l] + 1
        n_weighted_sums = self.structure[l]
        layer_weighted_sums = numpy.zeros((n_weighted_sums, 1))
        layer_activations = numpy.ones((n_activations, 1))

        self.weighted_sums.append(layer_weighted_sums)
        self.activations.append(layer_activations)
```

Listen in Python haben **0-basierte Indizes**. Die Liste [17, 12, 23] hat an Index 0 den Wert 17 und an Index 2 den Wert 23. Das müssen wir bei jedem Zugriff bedenken.

# Neuronenlayer initialisieren



```
def __init_layers(self):
    self.activations = []
    self.weighted_sums = []

    for l in range(len(self.structure)):
        is_output_layer = l == len(self.structure) - 1
        n_activations = self.structure[l] if is_output_layer else self.structure[l] + 1
        n_weighted_sums = self.structure[l]
        layer_weighted_sums = numpy.zeros((n_weighted_sums, 1))
        layer_activations = numpy.ones((n_activations, 1))

        self.weighted_sums.append(layer_weighted_sums)
        self.activations.append(layer_activations)
```

Das Ziel unserer `for`-Schleife ist, die einzelnen Layer des Netzes mit der jeweils korrekten Anzahl an Neuronen anzulegen.

# Neuronenlayer initialisieren



```
def __init_layers(self):
    self.activations = []
    self.weighted_sums = []

    for l in range(len(self.structure)):
        is_output_layer = l == len(self.structure) - 1
        n_activations = self.structure[l] if is_output_layer else self.structure[l] + 1
        n_weighted_sums = self.structure[l]
        layer_weighted_sums = numpy.zeros((n_weighted_sums, 1))
        layer_activations = numpy.ones((n_activations, 1))

        self.weighted_sums.append(layer_weighted_sums)
        self.activations.append(layer_activations)
```

Zunächst ermitteln wir, ob `l` gerade den Index des Output-Layers hat, da dieser eine Sonderbehandlung benötigt. Da Listenindizes 0-basiert sind, wäre der letzte Index die **Länge minus 1**.

# Neuronenlayer initialisieren



```
def __init_layers(self):  
    self.activations = []  
    self.weighted_sums = []  
  
    for l in range(len(self.structure)):  
        is_output_layer = l == len(self.structure) - 1  
        n_activations = self.structure[l] if is_output_layer else self.structure[l] + 1  
        n_weighted_sums = self.structure[l]  
        layer_weighted_sums = numpy.zeros((n_weighted_sums, 1))  
        layer_activations = numpy.ones((n_activations, 1))  
  
        self.weighted_sums.append(layer_weighted_sums)  
        self.activations.append(layer_activations)
```

Das Ergebnis weisen wir der **Variable is\_output\_layer** zu. Sie wird den Wert **True** erhalten, wenn wir im Output-Layer sind, und den Wert **False**, wenn nicht.

# Neuronenlayer initialisieren



```
def __init_layers(self):
    self.activations = []
    self.weighted_sums = []

    for l in range(len(self.structure)):
        is_output_layer = l == len(self.structure) - 1
        n_activations = self.structure[l] if is_output_layer else self.structure[l] + 1
        n_weighted_sums = self.structure[l]
        layer_weighted_sums = numpy.zeros((n_weighted_sums, 1))
        layer_activations = numpy.ones((n_activations, 1))

        self.weighted_sums.append(layer_weighted_sums)
        self.activations.append(layer_activations)
```

Man beachte den Unterschied: Ein doppeltes Gleichheitszeichen `==` führt einen **booleschen Vergleich** zwischen zwei Werten durch, dessen Ergebnis `True` oder `False` ist. Das einfache Gleichheitszeichen `=` weist einer Variable einen Wert zu. Es wird immer zuerst der Ausdruck rechts vom `=` ausgewertet und danach zugewiesen.

# Neuronenlayer initialisieren



```
def __init_layers(self):
    self.activations = []
    self.weighted_sums = []

    for l in range(len(self.structure)):
        is_output_layer = l == len(self.structure) - 1
        n_activations = self.structure[l] if is_output_layer else self.structure[l] + 1
        n_weighted_sums = self.structure[l]
        layer_weighted_sums = numpy.zeros((n_weighted_sums, 1))
        layer_activations = numpy.ones((n_activations, 1))

        self.weighted_sums.append(layer_weighted_sums)
        self.activations.append(layer_activations)
```

Wir legen die Variable `n_activations` für die Anzahl der Neuronen des aktuellen Layers an und weisen ihr die Anzahl zu, die wir aus dem entsprechenden Index der Liste in `structure` auslesen. Der Zugriff auf Elemente der Liste erfolgt durch den Index in eckigen Klammern.

# Neuronenlayer initialisieren



```
def __init_layers(self):
    self.activations = []
    self.weighted_sums = []

    for l in range(len(self.structure)):
        is_output_layer = l == len(self.structure) - 1
        n_activations = self.structure[l] if is_output_layer else self.structure[l] + 1
        n_weighted_sums = self.structure[l]
        layer_weighted_sums = numpy.zeros((n_weighted_sums, 1))
        layer_activations = numpy.ones((n_activations, 1))

        self.weighted_sums.append(layer_weighted_sums)
        self.activations.append(layer_activations)
```

Dabei verwenden wir die ausgelesene Anzahl für den Output-Layer direkt.

# Neuronenlayer initialisieren



```
def __init_layers(self):
    self.activations = []
    self.weighted_sums = []

    for l in range(len(self.structure)):
        is_output_layer = l == len(self.structure) - 1
        n_activations = self.structure[l] if is_output_layer else self.structure[l] + 1
        n_weighted_sums = self.structure[l]
        layer_weighted_sums = numpy.zeros((n_weighted_sums, 1))
        layer_activations = numpy.ones((n_activations, 1))

        self.weighted_sums.append(layer_weighted_sums)
        self.activations.append(layer_activations)
```

Für alle anderen Layer addieren wir 1 zu der ausgelesenen Zahl, da im Input- und Hidden Layer je ein Bias-Neuron zusätzlich angelegt werden soll.

# Neuronenlayer initialisieren



```
def __init_layers(self):
    self.activations = []
    self.weighted_sums = []

    for l in range(len(self.structure)):
        is_output_layer = l == len(self.structure) - 1
        n_activations = self.structure[l] if is_output_layer else self.structure[l] + 1
        n_weighted_sums = self.structure[l]
        layer_weighted_sums = numpy.zeros((n_weighted_sums, 1))
        layer_activations = numpy.ones((n_activations, 1))

        self.weighted_sums.append(layer_weighted_sums)
        self.activations.append(layer_activations)
```

Die Variable `n_weighted_sums` soll die Anzahl der gewichteten Summen des Layers enthalten. In den Bias-Neuronen soll es keine gewichteten Summen geben, da diese unbeeinflusst vom vorherigen Layer sein sollen. Daher können wir die Anzahl unmittelbar aus der `structure`-Liste auslesen.

# Neuronenlayer initialisieren



```
def __init_layers(self):
    self.activations = []
    self.weighted_sums = []

    for l in range(len(self.structure)):
        is_output_layer = l == len(self.structure) - 1
        n_activations = self.structure[l] if is_output_layer else self.structure[l] + 1
        n_weighted_sums = self.structure[l]
        layer_weighted_sums = numpy.zeros((n_weighted_sums, 1))
        layer_activations = numpy.ones((n_activations, 1))

        self.weighted_sums.append(layer_weighted_sums)
        self.activations.append(layer_activations)
```

Dann erzeugen wir mit der Funktion `numpy.zeros` für die gewichteten Summen des Layers eine Matrix, die mit **0**en vorbefüllt wird. Die Matrix soll **n\_weighted\_sums Zeilen** und **eine Spalte** haben (also ein Spaltenvektor sein).

# Neuronenlayer initialisieren



```
def __init_layers(self):
    self.activations = []
    self.weighted_sums = []

    for l in range(len(self.structure)):
        is_output_layer = l == len(self.structure) - 1
        n_activations = self.structure[l] if is_output_layer else self.structure[l] + 1
        n_weighted_sums = self.structure[l]
        layer_weighted_sums = numpy.zeros((n_weighted_sums, 1))
        layer_activations = numpy.ones((n_activations, 1))

        self.weighted_sums.append(layer_weighted_sums)
        self.activations.append(layer_activations)
```

Für die Aktivierungen gehen wir ähnlich vor. Hier befüllen wir die Matrix aber mithilfe von `numpy.ones` mit Einsen vor. Für das Bias-Neuron bleibt diese Aktivierung dauerhaft erhalten, alle anderen werden im Rahmen der Berechnungen überschrieben.

# Neuronenlayer initialisieren



```
def __init_layers(self):
    self.activations = []
    self.weighted_sums = []

    for l in range(len(self.structure)):
        is_output_layer = l == len(self.structure) - 1
        n_activations = self.structure[l] if is_output_layer else self.structure[l] + 1
        n_weighted_sums = self.structure[l]
        layer_weighted_sums = numpy.zeros((n_weighted_sums, 1))
        layer_activations = numpy.ones((n_activations, 1))

        self.weighted_sums.append(layer_weighted_sums)
        self.activations.append(layer_activations)
```

Man beachte jeweils die doppelten Klammern. Wir übergeben den numpy-Funktionen nicht zwei Parameter, sondern nur einen: Nämlich ein **Tupel**, das die Dimensionen der Matrix angibt, in diesem Fall im Format (**Zeilen, Spalten**).

# Neuronenlayer initialisieren



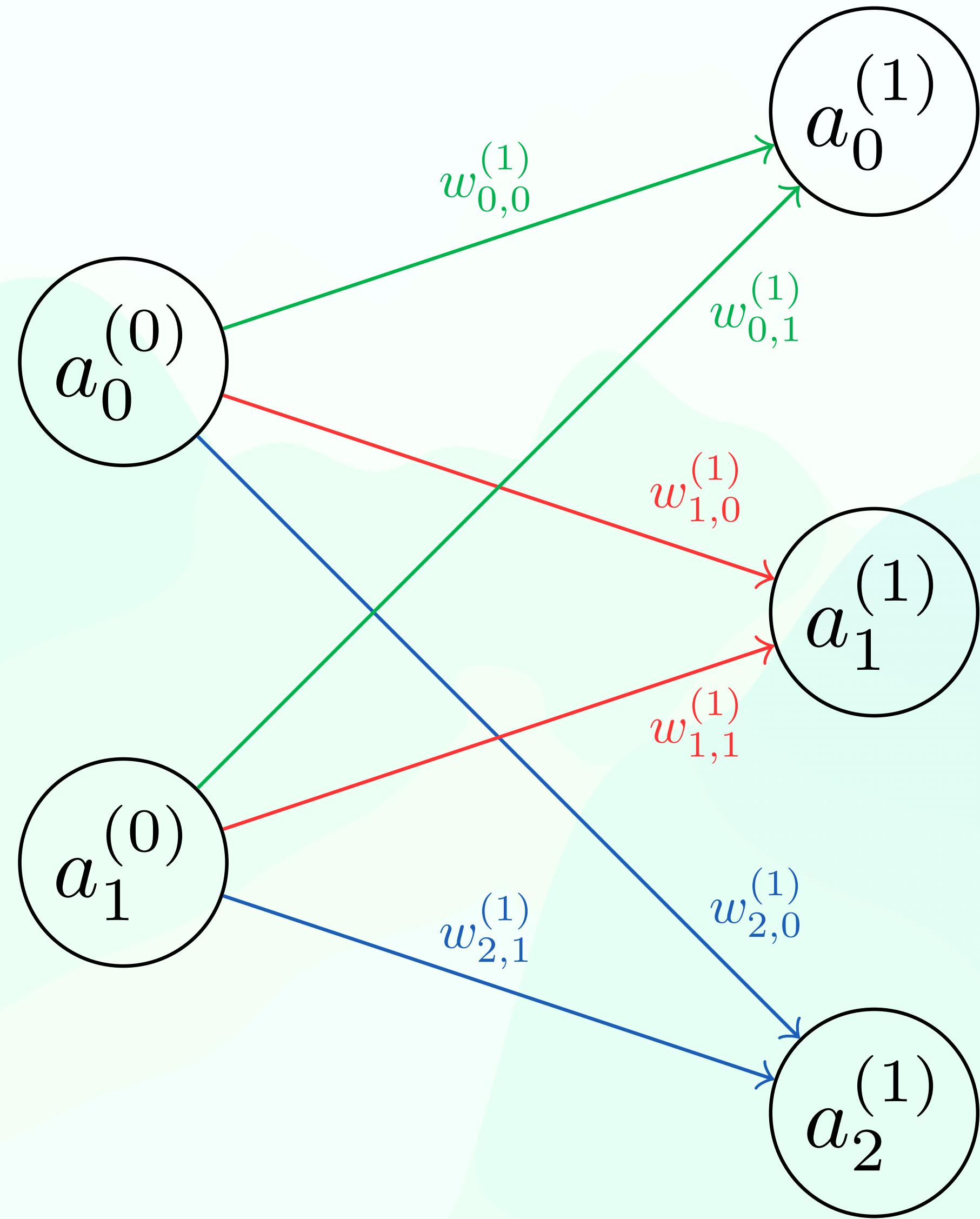
```
def __init_layers(self):
    self.activations = []
    self.weighted_sums = []

    for l in range(len(self.structure)):
        is_output_layer = l == len(self.structure) - 1
        n_activations = self.structure[l] if is_output_layer else self.structure[l] + 1
        n_weighted_sums = self.structure[l]
        layer_weighted_sums = numpy.zeros((n_weighted_sums, 1))
        layer_activations = numpy.ones((n_activations, 1))

        self.weighted_sums.append(layer_weighted_sums)
        self.activations.append(layer_activations)
```

Zum Schluss fügen wir die Matrizen (Vektoren) für den aktuellen Layer jeweils mit der `append`-Methode den Listen hinzu. Auf diese Weise bauen wir Layer für Layer die Struktur des neuronalen Netzes auf.

# Notation – Gewichtsmatrix



Für Notation und Implementierung in Python enorme  
Hilfe: Vektor- und Matrixschreibweise

$$W^{(1)} = \begin{pmatrix} w_{0,0}^{(1)} & w_{0,1}^{(1)} \\ w_{1,0}^{(1)} & w_{1,1}^{(1)} \\ w_{2,0}^{(1)} & w_{2,1}^{(1)} \end{pmatrix}$$

# Gewichte initialisieren



```
def __init_weights(self):
    self.weights = []

    for l in range(len(self.activations)):
        if l == 0:
            self.weights.append( [] )
        else:
            layer_weights = numpy.random.rand(
                len(self.weighted_sums[l]),
                len(self.activations[l-1]),
            ) * 2 - 1

        self.weights.append(layer_weights)
```

# Gewichte initialisieren



```
def __init_weights(self):
    self.weights = []

    for l in range(len(self.activations)):
        if l == 0:
            self.weights.append( [] )
        else:
            layer_weights = numpy.random.rand(
                len(self.weighted_sums[l]),
                len(self.activations[l-1]),
            ) * 2 - 1

        self.weights.append(layer_weights)
```

Analog zu `__init_layers` deklarieren wir auch die Methode `__init_weights` zur Initialisierung der Gewichte in unserem Netz.

# Gewichte initialisieren



```
def __init_weights(self):
    self.weights = []

    for l in range(len(self.activations)):
        if l == 0:
            self.weights.append( [] )
        else:
            layer_weights = numpy.random.rand(
                len(self.weighted_sums[l]),
                len(self.activations[l-1]),
            ) * 2 - 1

        self.weights.append(layer_weights)
```

Wir legen zunächst wieder eine leere Liste in der Eigenschaft `weights` an. Die einzelnen Einträge werden am Ende die Gewichtsmatrizen der einzelnen Layer sein.

# Gewichte initialisieren



```
def __init_weights(self):
    self.weights = []

    for l in range(len(self.activations)):
        if l == 0:
            self.weights.append( [] )
        else:
            layer_weights = numpy.random.rand(
                len(self.weighted_sums[l]),
                len(self.activations[l-1]),
            ) * 2 - 1

        self.weights.append(layer_weights)
```

Wir iterieren wieder durch die Layer, deren Anzahl uns durch die Länge der Liste in `self.activations` vorgegeben ist. Wieder wird `l` den jeweils aktuellen Index des Layers beinhalten.

# Gewichte initialisieren



```
def __init_weights(self):
    self.weights = []

    for l in range(len(self.activations)):
        if l == 0:
            self.weights.append( [] )
        else:
            layer_weights = numpy.random.rand(
                len(self.weighted_sums[l]),
                len(self.activations[l-1]),
            ) * 2 - 1

        self.weights.append(layer_weights)
```

Der Input-Layer hat den Index  $0$  und keine Gewichte, die zu ihm führen. Um eine konsistente Indizierung zu ermöglichen, fügen wir für den Index  $0$  trotzdem einen Eintrag zur Liste hinzu. Er wird später ignoriert.

Damit bekommt die Gewichtsmatrix vom Input- zum ersten Hidden Layer den Index  $1$ , wie es auch in unseren Gleichungen vorgegeben ist.

# Gewichte initialisieren



```
def __init_weights(self):
    self.weights = []

    for l in range(len(self.activations)):
        if l == 0:
            self.weights.append( [] )
        else:
            layer_weights = numpy.random.rand(
                len(self.weighted_sums[l]),
                len(self.activations[l-1]),
            ) * 2 - 1

            self.weights.append(layer_weights)
```

Für alle weiteren Indizes erzeugen wir echte Gewichtsmatrizen, diesmal mit der Funktion `numpy.random.rand`, die eine Matrix mit Zufallszahlen zwischen 0 und 1 erzeugt.

# Gewichte initialisieren



```
def __init_weights(self):
    self.weights = []

    for l in range(len(self.activations)):
        if l == 0:
            self.weights.append( [] )
        else:
            layer_weights = numpy.random.rand(
                len(self.weighted_sums[l]),
                len(self.activations[l-1]),
            ) * 2 - 1

        self.weights.append(layer_weights)
```

Der erste Parameter gibt die Anzahl der **Zeilen** der Matrix an. Es sollen so viele Zeilen wie **gewichtete Summen im aktuellen Layer** sein — also eine weniger, als es Neuronen gibt. Zum Bias-Neuron sollen keine Gewichte führen.

# Gewichte initialisieren



```
def __init_weights(self):
    self.weights = []

    for l in range(len(self.activations)):
        if l == 0:
            self.weights.append( [] )
        else:
            layer_weights = numpy.random.rand(
                len(self.weighted_sums[l]),
                len(self.activations[l-1]),
            ) * 2 - 1

        self.weights.append(layer_weights)
```

Der zweite Parameter ist die Anzahl der **Spalten**. Sie ist so groß wie die Anzahl der **Aktivierungen im vorherigen Layer**, denn alle Neuronen des vorherigen Layers sollen in die gewichteten Summen des nächsten Layers einfließen, inkl. des Bias-Neurons.

# Gewichte initialisieren



```
def __init_weights(self):
    self.weights = []

    for l in range(len(self.activations)):
        if l == 0:
            self.weights.append( [] )
        else:
            layer_weights = numpy.random.rand(
                len(self.weighted_sums[l]),
                len(self.activations[l-1]),
            ) * 2 - 1

        self.weights.append(layer_weights)
```

Unsere Gewichte sollen initial im Bereich zwischen -1 und 1 liegen. Dazu müssen wir die Größe des Intervalls [0, 1] verdoppeln und um 1 nach unten verschieben.

Praktischerweise können wir numpy-Matrizen auch mit Skalaren multiplizieren und addieren.

# Gewichte initialisieren



```
def __init_weights(self):
    self.weights = []

    for l in range(len(self.activations)):
        if l == 0:
            self.weights.append( [] )
        else:
            layer_weights = numpy.random.rand(
                len(self.weighted_sums[l]),
                len(self.activations[l-1]),
            ) * 2 - 1

    self.weights.append(layer_weights)
```

Zuletzt fügen wir die Gewichtsmatrix des aktuellen Layers zum Netz hinzu.

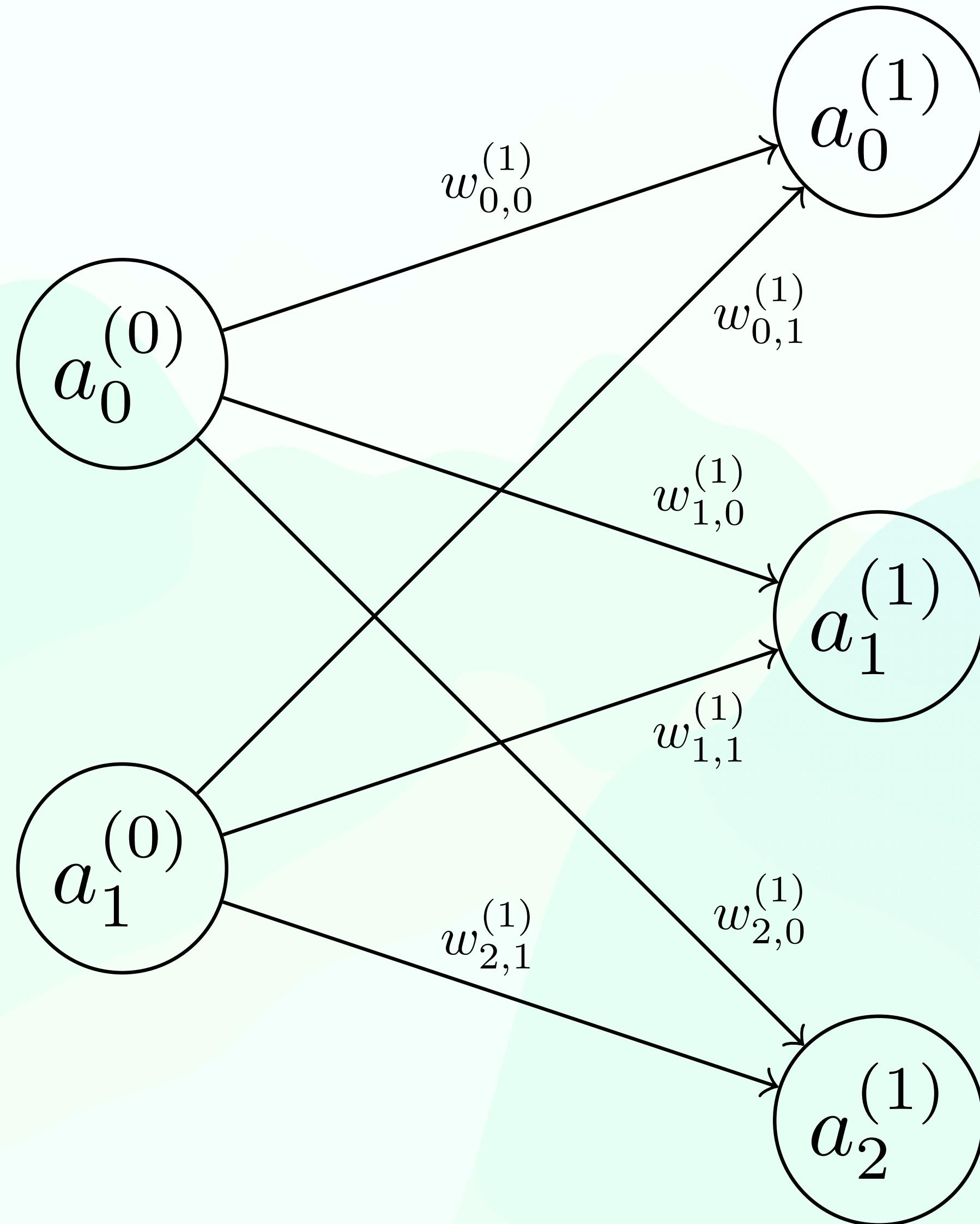
# Gewichte initialisieren



```
def __init__(self, structure):
    self.structure = structure
    self.__init_layers()
    self.__init_weights()
```

Natürlich muss unsere Methode zur Initialisierung der Gewichte auch im Konstruktor aufgerufen werden.

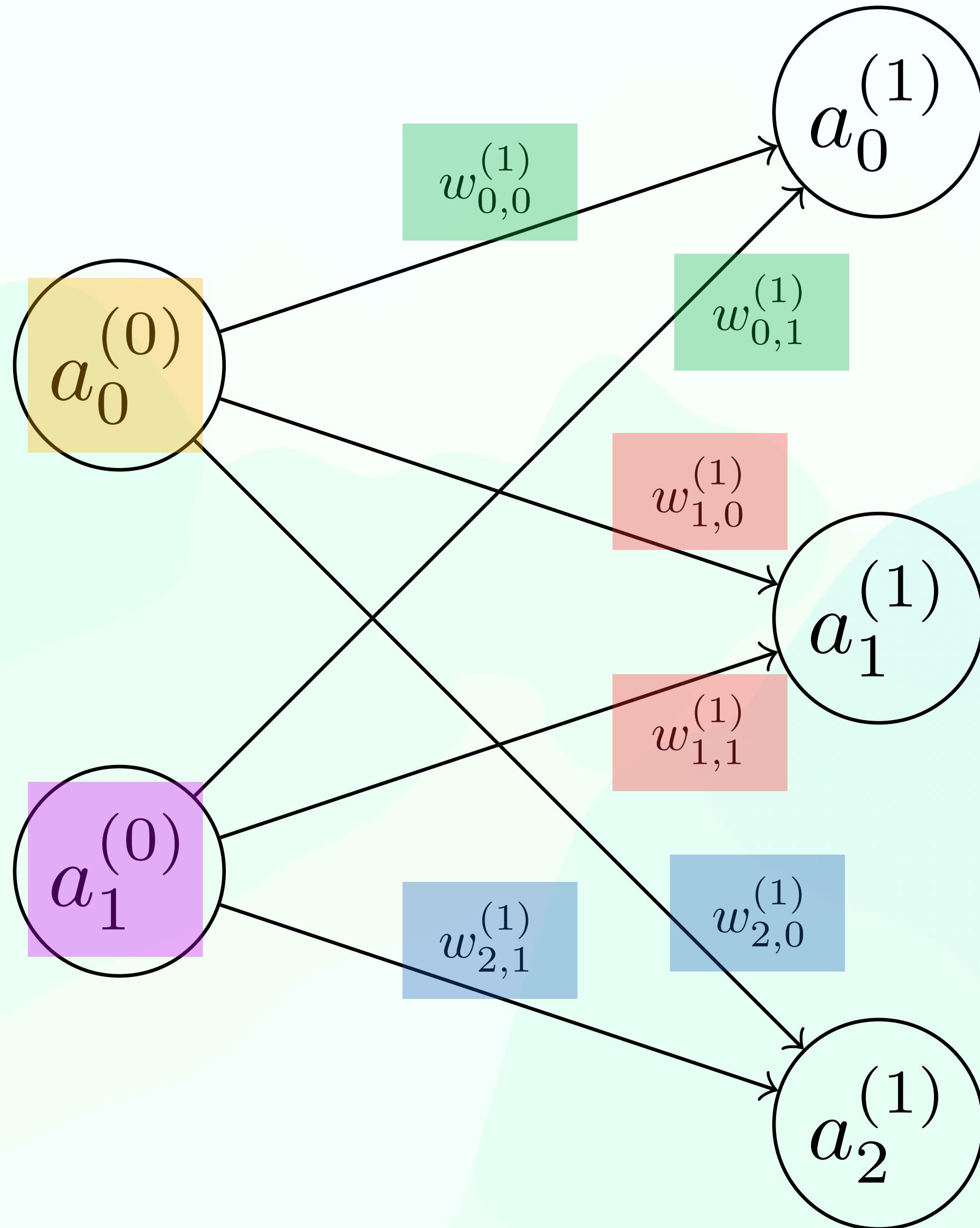
# Notation – Aktivierungsfunktion



Für Notation und Implementierung in Python enorme  
Hilfe: Vektor- und Matrixschreibweise

$$\begin{aligned} A^{(1)} &= \sigma^{(1)} (Z^{(1)}) \\ &= \sigma^{(1)} (W^{(1)} A^{(0)}) \\ &= \sigma^{(1)} \left[ \begin{pmatrix} w_{0,0}^{(1)} & w_{0,1}^{(1)} \\ w_{1,0}^{(1)} & w_{1,1}^{(1)} \\ w_{2,0}^{(1)} & w_{2,1}^{(1)} \end{pmatrix} \begin{pmatrix} a_0^{(0)} \\ a_1^{(0)} \end{pmatrix} \right] \end{aligned}$$

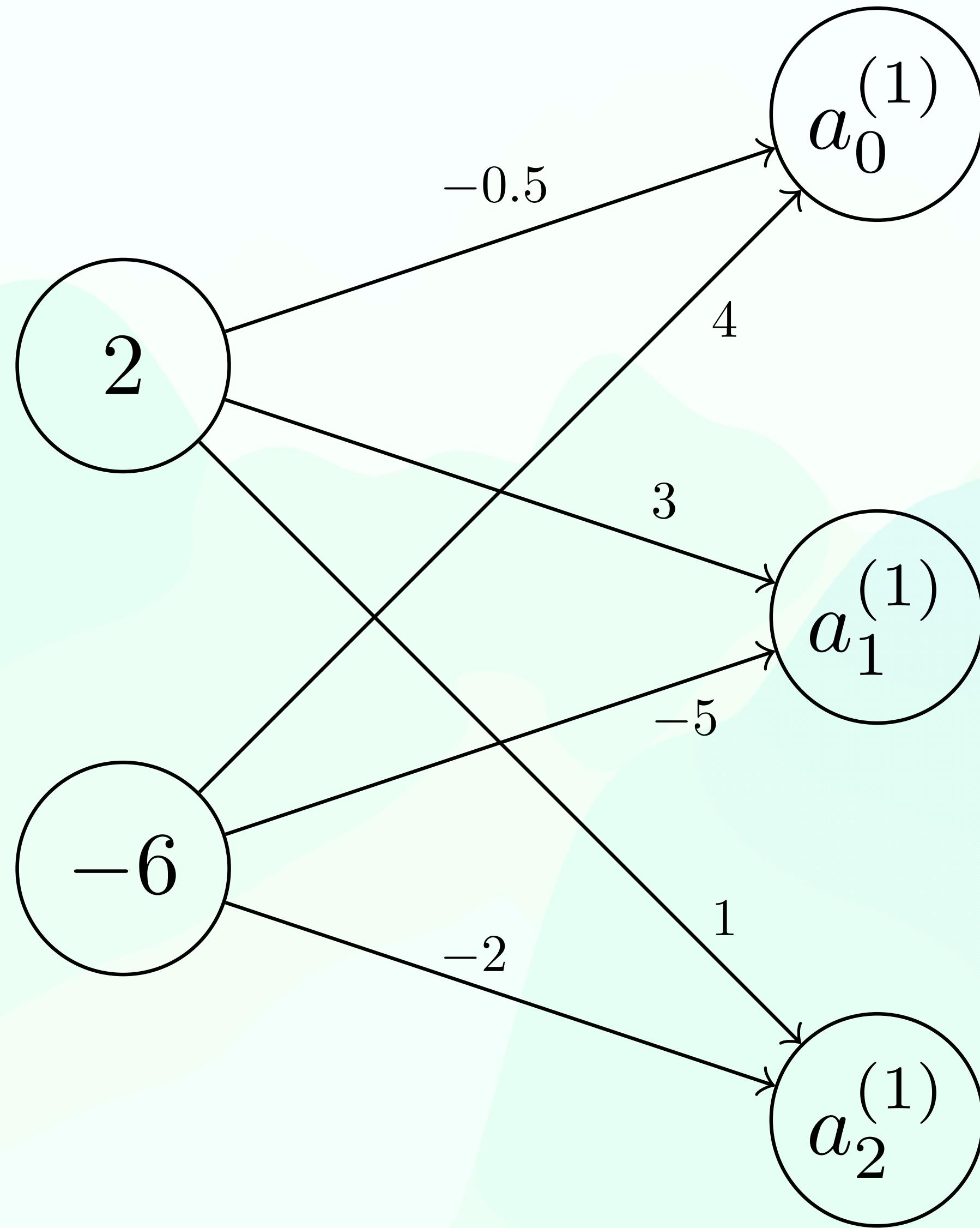
# Matrixmultiplikation



- Linke Matrix muss genauso viele **Spalten** haben wie die rechte Matrix **Zeilen** hat
- Ergebnis hat so viele **Zeilen** wie die linke Matrix und so viele **Spalten** wie die rechte
- Matrixmultiplikation ist **nicht** kommutativ!

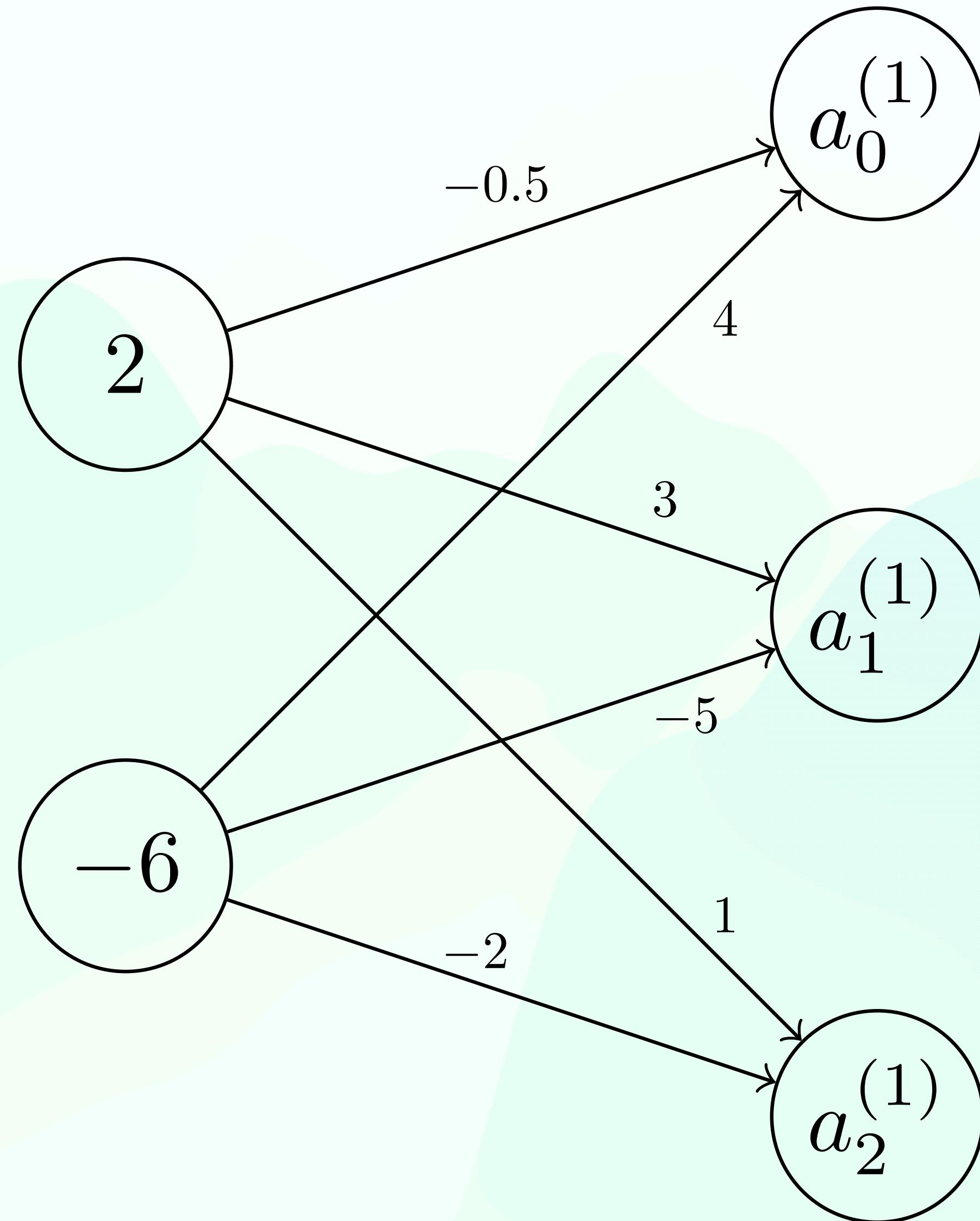
$$\begin{aligned}
 \begin{pmatrix} z_0^{(1)} \\ z_1^{(1)} \\ z_2^{(1)} \end{pmatrix} &= \begin{pmatrix} w_{0,0}^{(1)} & w_{0,1}^{(1)} \\ w_{1,0}^{(1)} & w_{1,1}^{(1)} \\ w_{2,0}^{(1)} & w_{2,1}^{(1)} \end{pmatrix} \begin{pmatrix} a_0^{(0)} \\ a_1^{(0)} \end{pmatrix} = \\
 &= \begin{pmatrix} w_{0,0}^{(1)} a_0^{(0)} + w_{0,1}^{(1)} a_1^{(0)} \\ w_{1,0}^{(1)} a_0^{(0)} + w_{1,1}^{(1)} a_1^{(0)} \\ w_{2,0}^{(1)} a_0^{(0)} + w_{2,1}^{(1)} a_1^{(0)} \end{pmatrix}
 \end{aligned}$$

# Matrixmultiplikation – Beispiel



$$\begin{pmatrix} z_0^{(1)} \\ z_1^{(1)} \\ z_2^{(1)} \end{pmatrix} = \begin{pmatrix} -0.5 & 4 \\ 3 & -5 \\ 1 & -2 \end{pmatrix} \begin{pmatrix} 2 \\ -6 \end{pmatrix} =$$
$$= \begin{pmatrix} -0.5 \cdot 2 + 4 \cdot (-6) \\ 3 \cdot 2 + (-5) \cdot (-6) \\ 1 \cdot 2 + (-2) \cdot (-6) \end{pmatrix} = \begin{pmatrix} -25 \\ 36 \\ 14 \end{pmatrix}$$

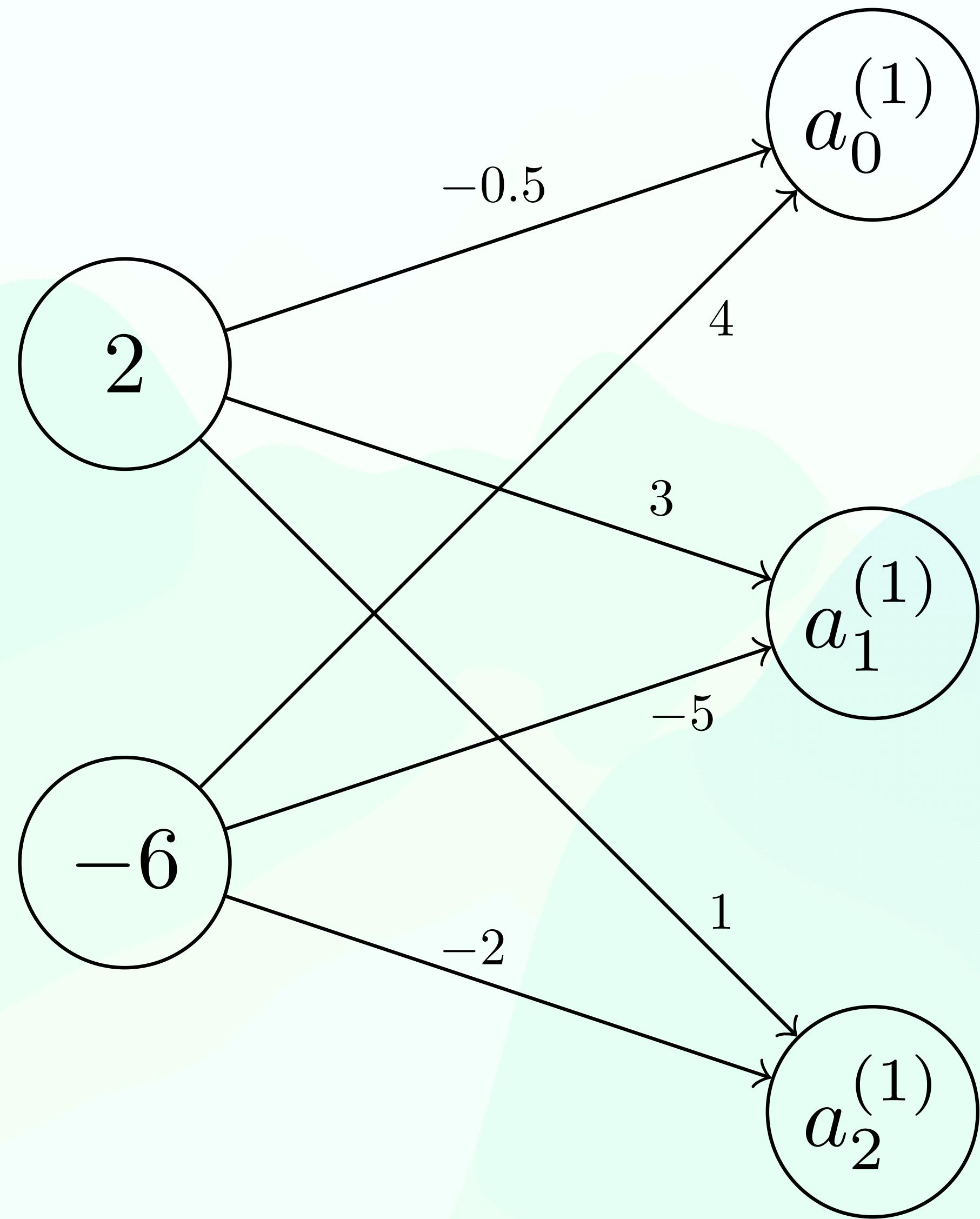
# Matrixmultiplikation – Beispiel



$$\begin{pmatrix} z_0^{(1)} \\ z_1^{(1)} \\ z_2^{(1)} \end{pmatrix} = \begin{pmatrix} -0.5 & 4 \\ 3 & -5 \\ 1 & -2 \end{pmatrix} \begin{pmatrix} 2 \\ -6 \end{pmatrix} =$$
$$= \begin{pmatrix} -0.5 \cdot 2 + 4 \cdot (-6) \\ 3 \cdot 2 + (-5) \cdot (-6) \\ 1 \cdot 2 + (-2) \cdot (-6) \end{pmatrix} = \begin{pmatrix} -25 \\ 36 \\ 14 \end{pmatrix}$$

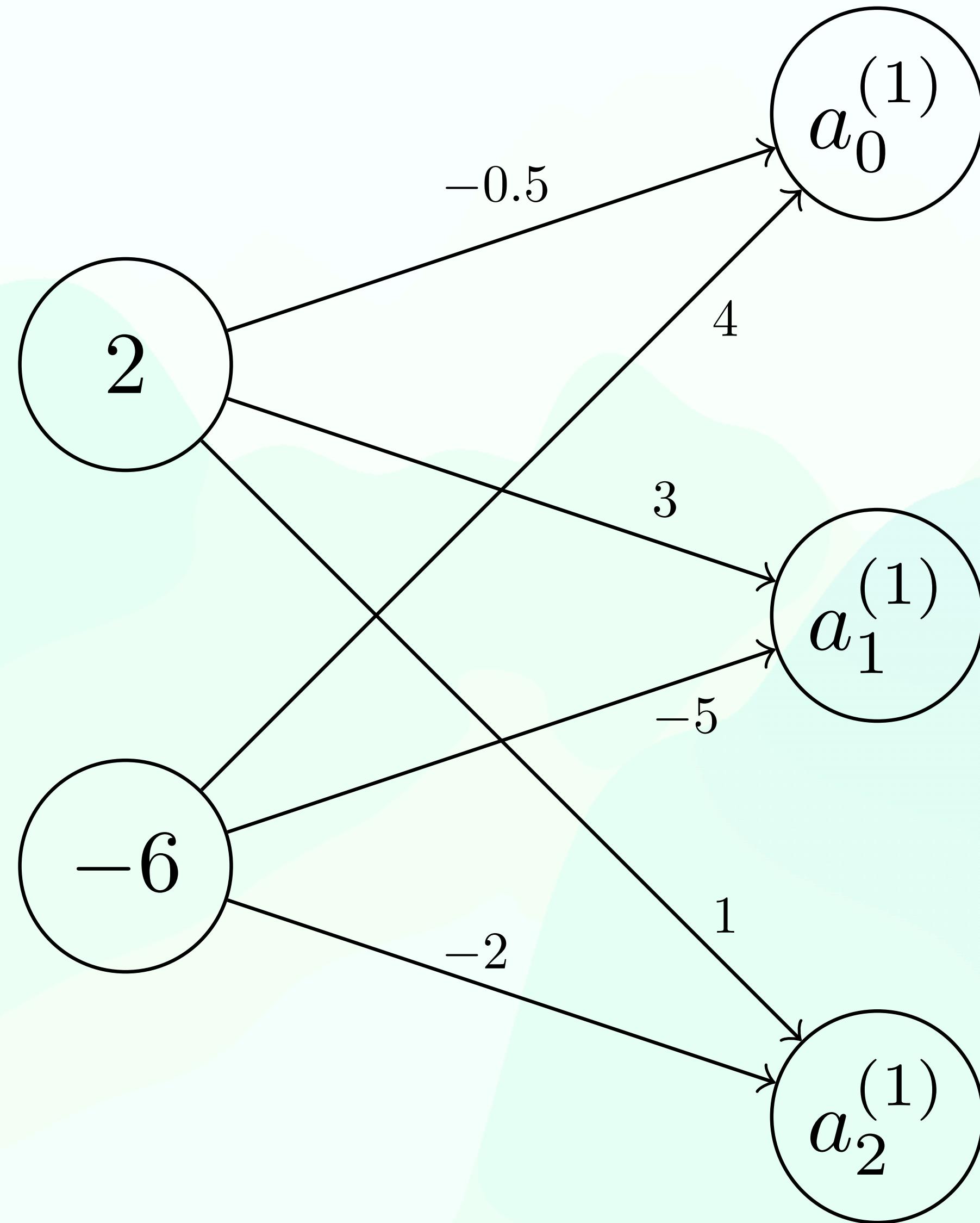
- Ergebnis der Multiplikation der **Gewichtsmatrix** mit dem **Aktivierungsvektor** ist die **gewichtete Summe  $z$**

# Feedforward-Schritt



- Aktivierung ergibt sich durch Aufruf der Aktivierungsfunktion  $\sigma$  mit gewichteter Summe  $z$

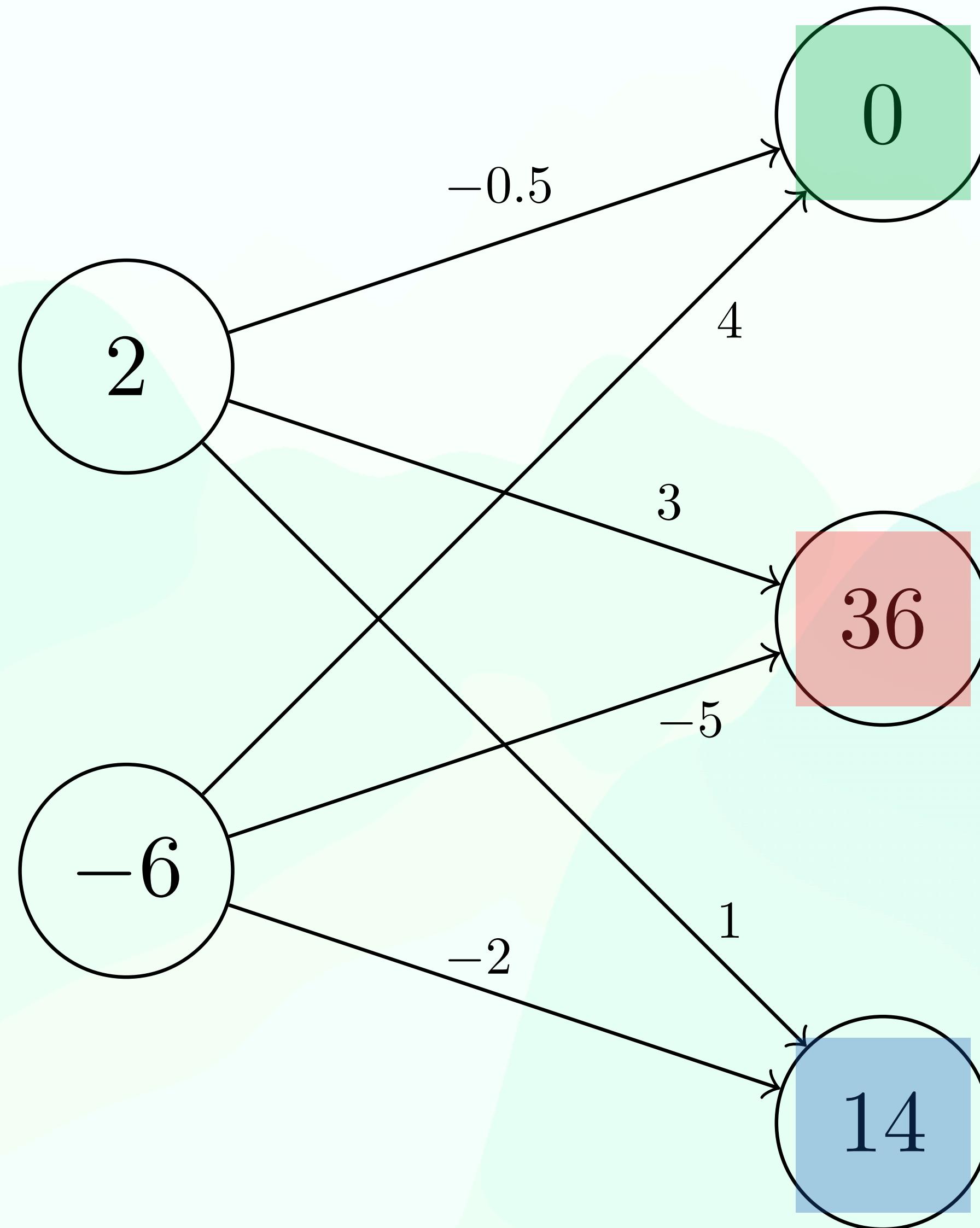
# Feedforward-Schritt



- Aktivierung ergibt sich durch Aufruf der Aktivierungsfunktion  $\sigma$  mit gewichteter Summe  $z$
- Wir verwenden hier  $\sigma = ReLU$

$$\begin{pmatrix} a_0^{(1)} \\ a_1^{(1)} \\ a_2^{(1)} \end{pmatrix} = ReLU \left[ \begin{pmatrix} z_0^{(1)} \\ z_1^{(1)} \\ z_2^{(1)} \end{pmatrix} \right]$$
$$= ReLU \left[ \begin{pmatrix} -25 \\ 36 \\ 14 \end{pmatrix} \right] = \begin{pmatrix} 0 \\ 36 \\ 14 \end{pmatrix}$$

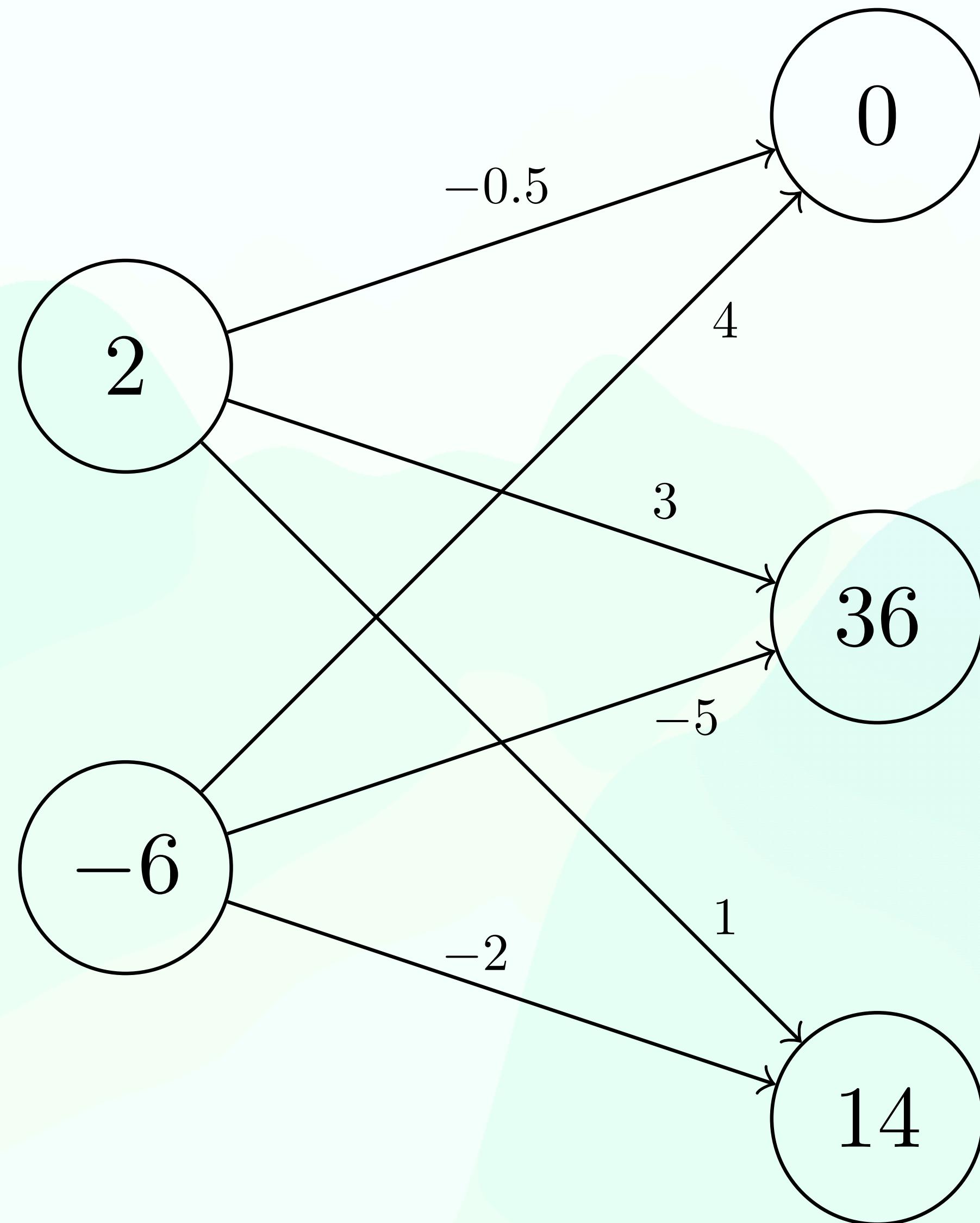
# Feedforward-Schritt



- Aktivierung ergibt sich durch Aufruf der Aktivierungsfunktion  $\sigma$  mit gewichteter Summe  $z$
- Wir verwenden hier  $\sigma = \text{ReLU}$

$$\begin{pmatrix} a_0^{(1)} \\ a_1^{(1)} \\ a_2^{(1)} \end{pmatrix} = \text{ReLU} \left[ \begin{pmatrix} z_0^{(1)} \\ z_1^{(1)} \\ z_2^{(1)} \end{pmatrix} \right]$$
$$= \text{ReLU} \left[ \begin{pmatrix} -25 \\ 36 \\ 14 \end{pmatrix} \right] = \begin{pmatrix} 0 \\ 36 \\ 14 \end{pmatrix}$$

# Feedforward-Schritt



- Diese Berechnungsschritte werden jeweils von einem Layer zum nächsten wiederholt, bis am Output-Layer die **Prediction** vorliegt

# Erweiterung des Konstruktors um Aktivierungsfunktionen



```
def __init__(  
    self,  
    structure,  
    output_activation_func=relu,  
    hidden_activation_func=sigmoid,  
):  
    self.structure = structure  
    self.output_activation_func = output_activation_func  
    self.hidden_activation_func = hidden_activation_func  
    self.__init_layers()  
    self.__init_weights()
```

# Erweiterung des Konstruktors um Aktivierungsfunktionen



```
def __init__(  
    self,  
    structure,  
    output_activation_func=relu,  
    hidden_activation_func=sigmoid,  
):  
    self.structure = structure  
    self.output_activation_func = output_activation_func  
    self.hidden_activation_func = hidden_activation_func  
    self.__init_layers()  
    self.__init_weights()
```

Zunächst fügen wir unserem Konstruktor zwei weitere Parameter für die Aktivierungsfunktionen in Output- und Hidden Layer hinzu.

Diesmal sind die Parameter allerdings **optional**: Übergeben wir bei der Objekterzeugung nichts, würden als **Default**-Werte die ReLU-Funktion im Output-Layer und die Sigmoid-Funktion im Hidden Layer verwendet werden.

# Erweiterung des Konstruktors um Aktivierungsfunktionen



```
def __init__(  
    self,  
    structure,  
    output_activation_func=relu,  
    hidden_activation_func=sigmoid,  
):  
    self.structure = structure  
    self.output_activation_func = output_activation_func  
    self.hidden_activation_func = hidden_activation_func  
    self.__init_layers()  
    self.__init_weights()
```

Die übergebenen Parameter (bzw. deren Defaultwerte) weisen wir gleichnamigen Eigenschaften des NeuralNetwork-Objekts zu.

# Prediction mit Feedforward-Algorithmus erzeugen



```
def predict(self, X):
    if len(X) != len(self.activations[0]) - 1:
        raise Exception("Falsche Anzahl an Input-Werten übergeben")

    self.activations[0][1:] = X

    for l in range(1, len(self.activations)):
        self.weighted_sums[l] = numpy.matmul(self.weights[l], self.activations[l-1])

        if l == len(self.activations) - 1:
            self.activations[l] = self.output_activation_func(self.weighted_sums[l])
        else:
            self.activations[l][1:] = self.hidden_activation_func(self.weighted_sums[l])

    return self.activations[-1]
```

# Prediction mit Feedforward-Algorithmus erzeugen



```
def predict(self, X):
    if len(X) != len(self.activations[0]) - 1:
        raise Exception("Falsche Anzahl an Input-Werten übergeben")

    self.activations[0][1:] = X

    for l in range(1, len(self.activations)):
        self.weighted_sums[l] = numpy.matmul(self.weights[l], self.activations[l-1])

        if l == len(self.activations) - 1:
            self.activations[l] = self.output_activation_func(self.weighted_sums[l])
        else:
            self.activations[l][1:] = self.hidden_activation_func(self.weighted_sums[l])

    return self.activations[-1]
```

Wir deklarieren die Methode `predict`, diesmal nicht als privat, denn sie soll von außerhalb der Klasse aufrufbar sein. Ihr soll als Parameter ein Vektor `X` mit den Inputs übergeben werden, für die eine Vorhersage berechnet werden soll.

# Prediction mit Feedforward-Algorithmus erzeugen



```
def predict(self, X):
    if len(X) != len(self.activations[0]) - 1:
        raise Exception("Falsche Anzahl an Input-Werten übergeben")

    self.activations[0][1:] = X

    for l in range(1, len(self.activations)):
        self.weighted_sums[l] = numpy.matmul(self.weights[l], self.activations[l-1])

        if l == len(self.activations) - 1:
            self.activations[l] = self.output_activation_func(self.weighted_sums[l])
        else:
            self.activations[l][1:] = self.hidden_activation_func(self.weighted_sums[l])

    return self.activations[-1]
```

Zuerst prüfen wir, ob die Anzahl der übergebenen Inputs gleich der Anzahl der Input-Neuronen abzüglich einem Bias-Neuron ist.

Falls nicht, erzeugen wir eine Fehlermeldung mit einer `Exception`.

# Prediction mit Feedforward-Algorithmus erzeugen



```
def predict(self, X):
    if len(X) != len(self.activations[0]) - 1:
        raise Exception("Falsche Anzahl an Input-Werten übergeben")

    self.activations[0][1:] = X

    for l in range(1, len(self.activations)):
        self.weighted_sums[l] = numpy.matmul(self.weights[l], self.activations[l-1])

        if l == len(self.activations) - 1:
            self.activations[l] = self.output_activation_func(self.weighted_sums[l])
        else:
            self.activations[l][1:] = self.hidden_activation_func(self.weighted_sums[l])

    return self.activations[-1]
```

Ist der Test bestanden, belegen wir den ersten Layer unseres Netzes (den mit dem Index 0) mit den übergebenen Input-Aktivierungen. Dabei müssen wir allerdings das Bias-Neuron unangetastet lassen.

# Prediction mit Feedforward-Algorithmus erzeugen



```
def predict(self, X):
    if len(X) != len(self.activations[0]) - 1:
        raise Exception("Falsche Anzahl an Input-Werten übergeben")

    self.activations[0][1:] = X

    for l in range(1, len(self.activations)):
        self.weighted_sums[l] = numpy.matmul(self.weights[l], self.activations[l-1])

        if l == len(self.activations) - 1:
            self.activations[l] = self.output_activation_func(self.weighted_sums[l])
        else:
            self.activations[l][1:] = self.hidden_activation_func(self.weighted_sums[l])

    return self.activations[-1]
```

Dazu verwenden wir die Slice-Syntax von numpy.  
Wir überschreiben im Aktivierungsvektor nur die Einträge ab dem Index 1.

# Prediction mit Feedforward-Algorithmus erzeugen



```
def predict(self, X):
    if len(X) != len(self.activations[0]) - 1:
        raise Exception("Falsche Anzahl an Input-Werten übergeben")

    self.activations[0][1:] = X

    for l in range(1, len(self.activations)):
        self.weighted_sums[l] = numpy.matmul(self.weights[l], self.activations[l-1])

        if l == len(self.activations) - 1:
            self.activations[l] = self.output_activation_func(self.weighted_sums[l])
        else:
            self.activations[l][1:] = self.hidden_activation_func(self.weighted_sums[l])

    return self.activations[-1]
```

Für die weiteren Layer benutzen wir wieder eine `for`-Schleife, beginnend beim Index 1 (dem des ersten Hidden Layers). Wieder gibt uns die Laufvariable `l` den Index des jeweils aktuellen Layers an.

# Prediction mit Feedforward-Algorithmus erzeugen



```
def predict(self, X):
    if len(X) != len(self.activations[0]) - 1:
        raise Exception("Falsche Anzahl an Input-Werten übergeben")
    self.activations[0][1:] = X

    for l in range(1, len(self.activations)):
        self.weighted_sums[l] = numpy.matmul(self.weights[l], self.activations[l-1])

        if l == len(self.activations) - 1:
            self.activations[l] = self.output_activation_func(self.weighted_sums[l])
        else:
            self.activations[l][1:] = self.hidden_activation_func(self.weighted_sums[l])

    return self.activations[-1]
```

Zunächst berechnen wir im aktuellen Layer  $l$  die gewichteten Summen als Produkte seiner Gewichtsmatrix und des Aktivierungsvektors des vorherigen Layers  $l-1$ . Die Matrixmultiplikation wird dabei von `numpy.matmul` erledigt.

# Prediction mit Feedforward-Algorithmus erzeugen



```
def predict(self, X):
    if len(X) != len(self.activations[0]) - 1:
        raise Exception("Falsche Anzahl an Input-Werten übergeben")

    self.activations[0][1:] = X

    for l in range(1, len(self.activations)):
        self.weighted_sums[l] = numpy.matmul(self.weights[l], self.activations[l-1])

        if l == len(self.activations) - 1:
            self.activations[l] = self.output_activation_func(self.weighted_sums[l])
        else:
            self.activations[l][1:] = self.hidden_activation_func(self.weighted_sums[l])

    return self.activations[-1]
```

Anschließend berechnen wir die Aktivierungen. Für den Output-Layer (identifizierbar am Index  $l$ ) rufen wir dafür die beim Erzeugen des Netzes übergebene Output-Aktivierungsfunktion mit dem soeben berechneten Vektor der gewichteten Summen auf.

# Prediction mit Feedforward-Algorithmus erzeugen



```
def predict(self, X):
    if len(X) != len(self.activations[0]) - 1:
        raise Exception("Falsche Anzahl an Input-Werten übergeben")

    self.activations[0][1:] = X

    for l in range(1, len(self.activations)):
        self.weighted_sums[l] = numpy.matmul(self.weights[l], self.activations[l-1])

        if l == len(self.activations) - 1:
            self.activations[l] = self.output_activation_func(self.weighted_sums[l])
        else:
            self.activations[l][1:] = self.hidden_activation_func(self.weighted_sums[l])

    return self.activations[-1]
```

Auf dieselbe Weise berechnen wir die Aktivierungen der Hidden Layer für alle anderen Indizes  $l$  und rufen dabei die andere übergebene Aktivierungsfunktion auf.

# Prediction mit Feedforward-Algorithmus erzeugen



```
def predict(self, X):
    if len(X) != len(self.activations[0]) - 1:
        raise Exception("Falsche Anzahl an Input-Werten übergeben")

    self.activations[0][1:] = X

    for l in range(1, len(self.activations)):
        self.weighted_sums[l] = numpy.matmul(self.weights[l], self.activations[l-1])

        if l == len(self.activations) - 1:
            self.activations[l] = self.output_activation_func(self.weighted_sums[l])
        else:
            self.activations[l][1:] = self.hidden_activation_func(self.weighted_sums[l])

    return self.activations[-1]
```

Auch hier müssen wir darauf achten, die Aktivierung des Bias-Neurons des Hidden Layers nicht zu überschreiben und verwenden wieder die Slice-Syntax.

# Prediction mit Feedforward-Algorithmus erzeugen



```
def predict(self, X):
    if len(X) != len(self.activations[0]) - 1:
        raise Exception("Falsche Anzahl an Input-Werten übergeben")
    self.activations[0][1:] = X

    for l in range(1, len(self.activations)):
        self.weighted_sums[l] = numpy.matmul(self.weights[l], self.activations[l-1])

        if l == len(self.activations) - 1:
            self.activations[l] = self.output_activation_func(self.weighted_sums[l])
        else:
            self.activations[l][1:] = self.hidden_activation_func(self.weighted_sums[l])

    return self.activations[-1]
```

Nachdem die Schleife auf diese Weise die Aktivierungen in allen Layern berechnet hat, geben wir die **Aktivierungen des Output-Layers** zurück. Sie enthalten die **Prediction** unseres neuronalen Netzes.

# Prediction mit Feedforward-Algorithmus erzeugen



```
def predict(self, X):
    if len(X) != len(self.activations[0]) - 1:
        raise Exception("Falsche Anzahl an Input-Werten übergeben")

    self.activations[0][1:] = X

    for l in range(1, len(self.activations)):
        self.weighted_sums[l] = numpy.matmul(self.weights[l], self.activations[l-1])

        if l == len(self.activations) - 1:
            self.activations[l] = self.output_activation_func(self.weighted_sums[l])
        else:
            self.activations[l][1:] = self.hidden_activation_func(self.weighted_sums[l])

    return self.activations[-1]
```

Um das letzte Element der Liste `activations` zu erhalten, verwenden wir den Index `-1`.

# Erster Test des Prediction-Algorithmus



```
if __name__ == '__main__':
    nn = NeuralNetwork(structure=[3, 4, 2])
    X = numpy.array([[0.7], [0.1], [0.5]])
    prediction = nn.predict(X)
    print(prediction)
```

# Erster Test des Prediction-Algorithmus



```
if __name__ == '__main__':
    nn = NeuralNetwork(structure=[3, 4, 2])
    X = numpy.array([[0.7], [0.1], [0.5]])
    prediction = nn.predict(X)
    print(prediction)
```

Wir legen einen Vektor mit irgendwelchen Input-Daten an. Wichtig ist nur, dass er genau so viele Elemente enthält, wie es der erste Eintrag des Parameters `structure` vorgibt — denn er bestimmt die Zahl der Input-Neuronen.

# Erster Test des Prediction-Algorithmus



```
if __name__ == '__main__':
    nn = NeuralNetwork(structure=[3, 4, 2])
    X = numpy.array([[0.7], [0.1], [0.5]])
    prediction = nn.predict(X)
    print(prediction)
```

Für unseren Input-Vektor  $X$  lassen wir eine Vorhersage berechnen und speichern sie in der Variable `prediction`.

# Erster Test des Prediction-Algorithmus



```
if __name__ == '__main__':
    nn = NeuralNetwork(structure=[3, 4, 2])
    X = numpy.array([[0.7], [0.1], [0.5]])
    prediction = nn.predict(X)
    print(prediction)
```

Wir geben mit einem einfachen `print` die Vorhersage auf der Konsole aus. Sie ist ein zweielementiger Vektor, da `structure` genau zwei Output-Neuronen vorsieht. Da die Gewichte in unserem Netz mit Zufallszahlen vorbelegt werden, hängt auch dieses Ergebnis vom Zufall ab.

# Kostenfunktion

- Gibt ein Maß für die Genauigkeit der Prediction an
- Gegeben: berechnete ( $A^{(l)}$ ) und erwartete ( $Y$ ) Aktivierung der Neuronen im Output-Layer

# Kostenfunktion

- Gibt ein Maß für die Genauigkeit der Prediction an
- Gegeben: berechnete ( $A^{(l)}$ ) und erwartete ( $Y$ ) Aktivierung der Neuronen im Output-Layer
- Kostenfunktion **Squared Error Loss (SEL)** berechnet das Quadrat der Differenz:

$$C_i = (A^{(l)} - Y)^2 = \left[ \begin{pmatrix} a_0^{(l)} \\ a_1^{(l)} \\ \vdots \\ a_n^{(l)} \end{pmatrix} - \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix} \right]^2$$

# Kostenfunktion – Beispiel

- Seien  $A^{(l)}$  die tatsächliche und  $Y$  die erwartete Aktivierung des Output-Layers:

$$A^{(l)} = \begin{pmatrix} 0.9 \\ 0.3 \\ 0.6 \end{pmatrix}$$

$$Y = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

# Kostenfunktion – Beispiel

- Seien  $A^{(l)}$  die tatsächliche und  $Y$  die erwartete Aktivierung des Output-Layers:

$$A^{(l)} = \begin{pmatrix} 0.9 \\ 0.3 \\ 0.6 \end{pmatrix} \quad Y = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

- Dann betragen die Kosten nach **Squared Error Loss**:

$$C = \left[ \begin{pmatrix} 0.9 \\ 0.3 \\ 0.6 \end{pmatrix} - \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \right]^2 = \begin{pmatrix} -0.1 \\ 0.3 \\ -0.4 \end{pmatrix}^2 = \begin{pmatrix} 0.01 \\ 0.09 \\ 0.16 \end{pmatrix}$$

# Kostenfunktion – Beispiel

- Seien  $A^{(l)}$  die tatsächliche und  $Y$  die erwartete Aktivierung des Output-Layers:

$$A^{(l)} = \begin{pmatrix} 0.9 \\ 0.3 \\ 0.6 \end{pmatrix} \quad Y = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

- Dann betragen die Kosten nach **Squared Error Loss**:

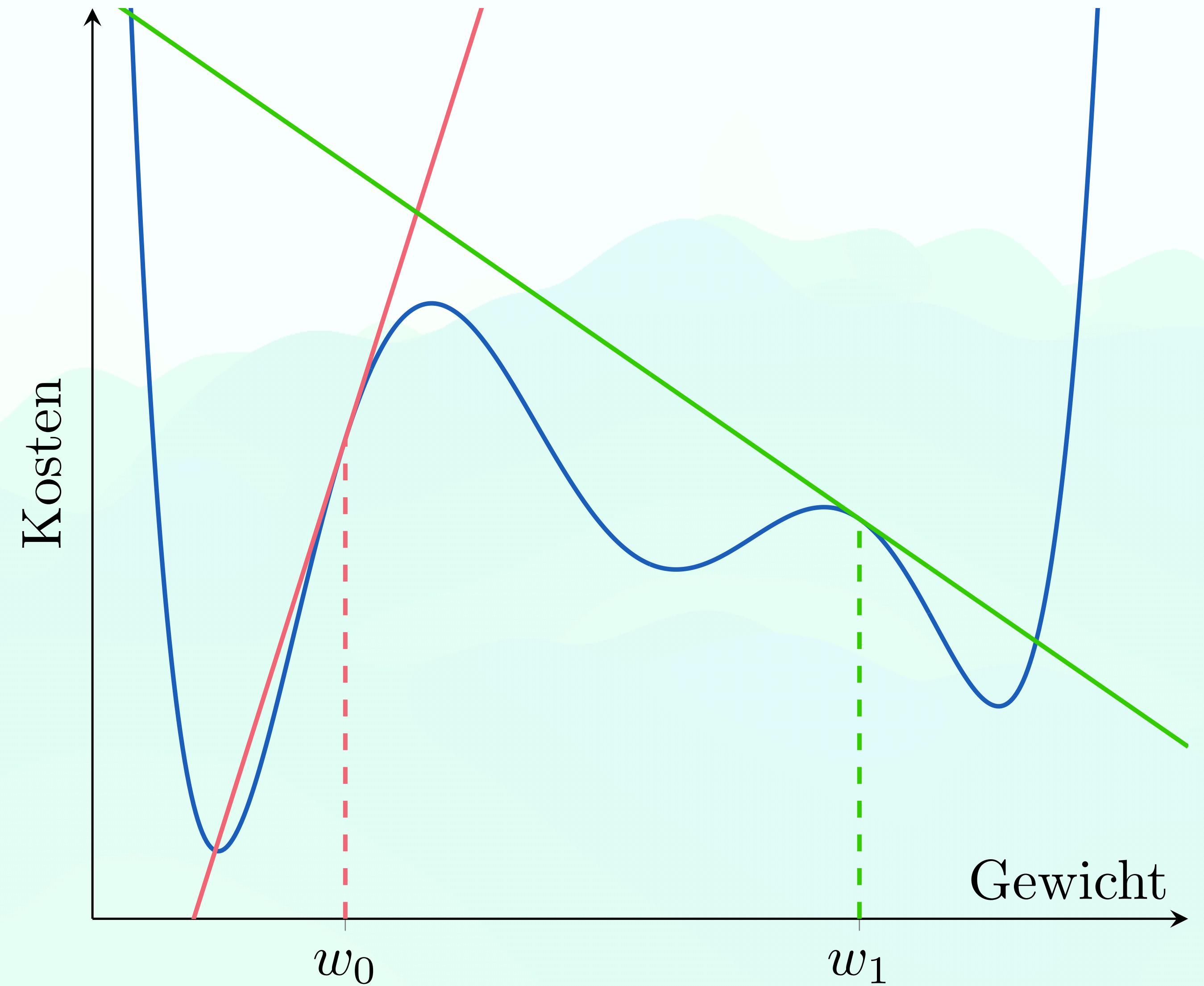
$$C = \left[ \begin{pmatrix} 0.9 \\ 0.3 \\ 0.6 \end{pmatrix} - \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \right]^2 = \begin{pmatrix} -0.1 \\ 0.3 \\ -0.4 \end{pmatrix}^2 = \begin{pmatrix} 0.01 \\ 0.09 \\ 0.16 \end{pmatrix}$$

Im ersten Neuron am niedrigsten,  
da Abweichung am geringsten

Im dritten Neuron am höchsten,  
da Abweichung am größten

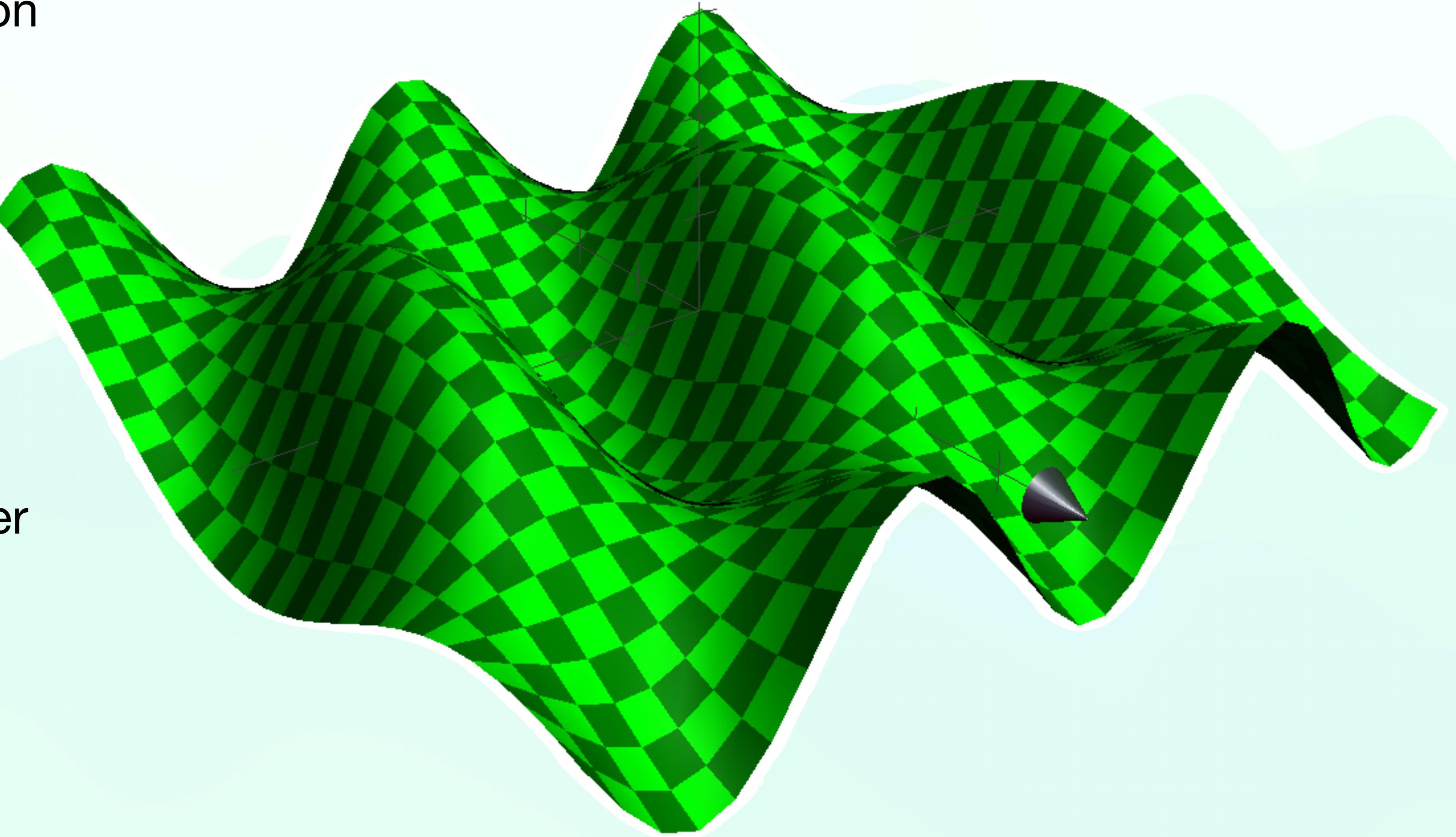
# Kostenfunktion

- Kostenfunktion soll minimiert werden
- **Gradient Descent:** Folge dem Weg des steilsten Abstiegs
- Je steiler der Gradient, desto größer die Schritte
- Neuronale Netze als **Universal Function Approximator**



# Kostenfunktion

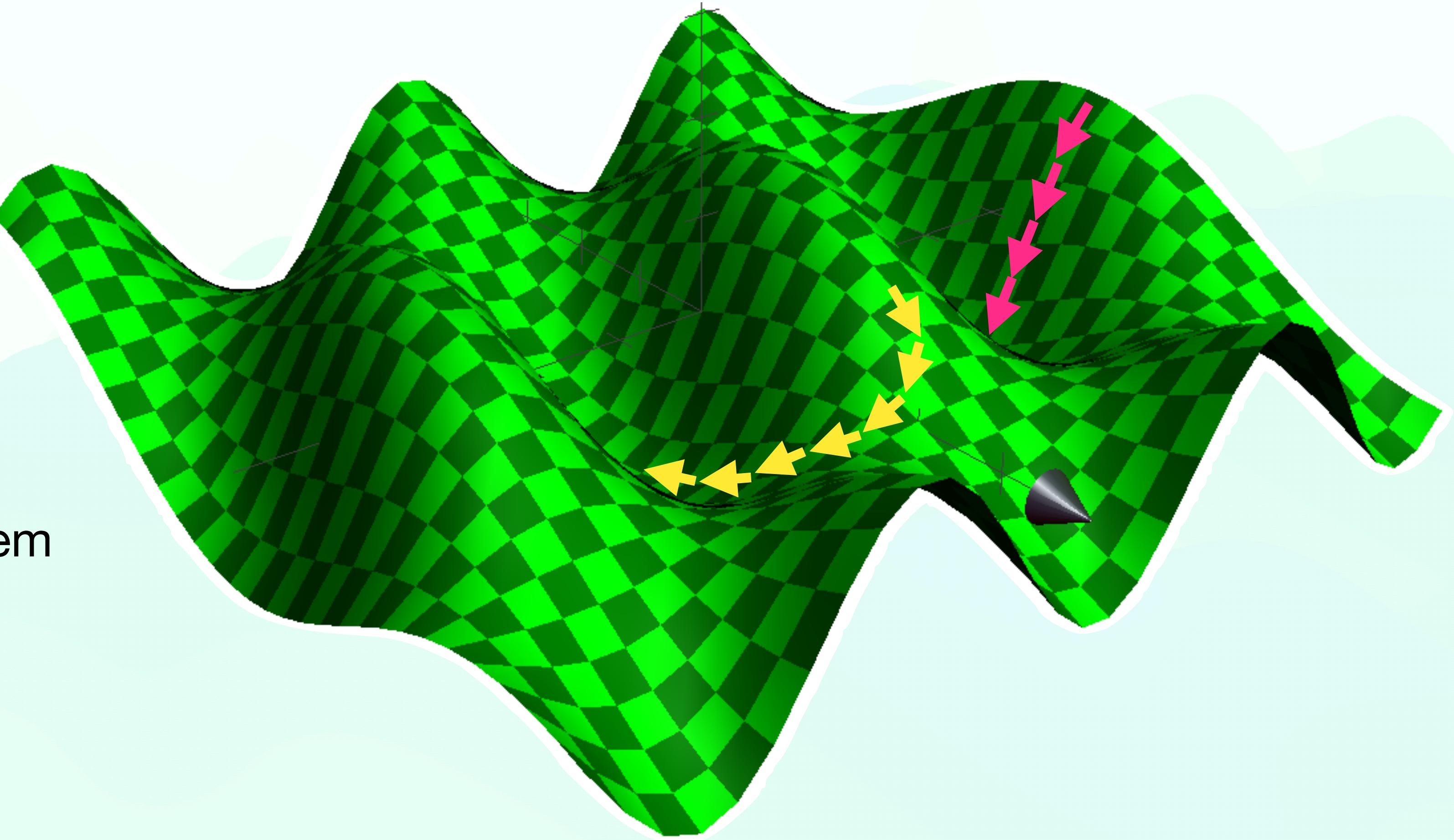
- In der Praxis ist die Kostenfunktion multidimensional
- Keine Garantie, dass Gradient Descent das **globale Minimum** findet
- Forschung zeigt, dass bei gut strukturierten Trainingsdaten die **lokalen Minima** von etwa gleicher Qualität sind [1]



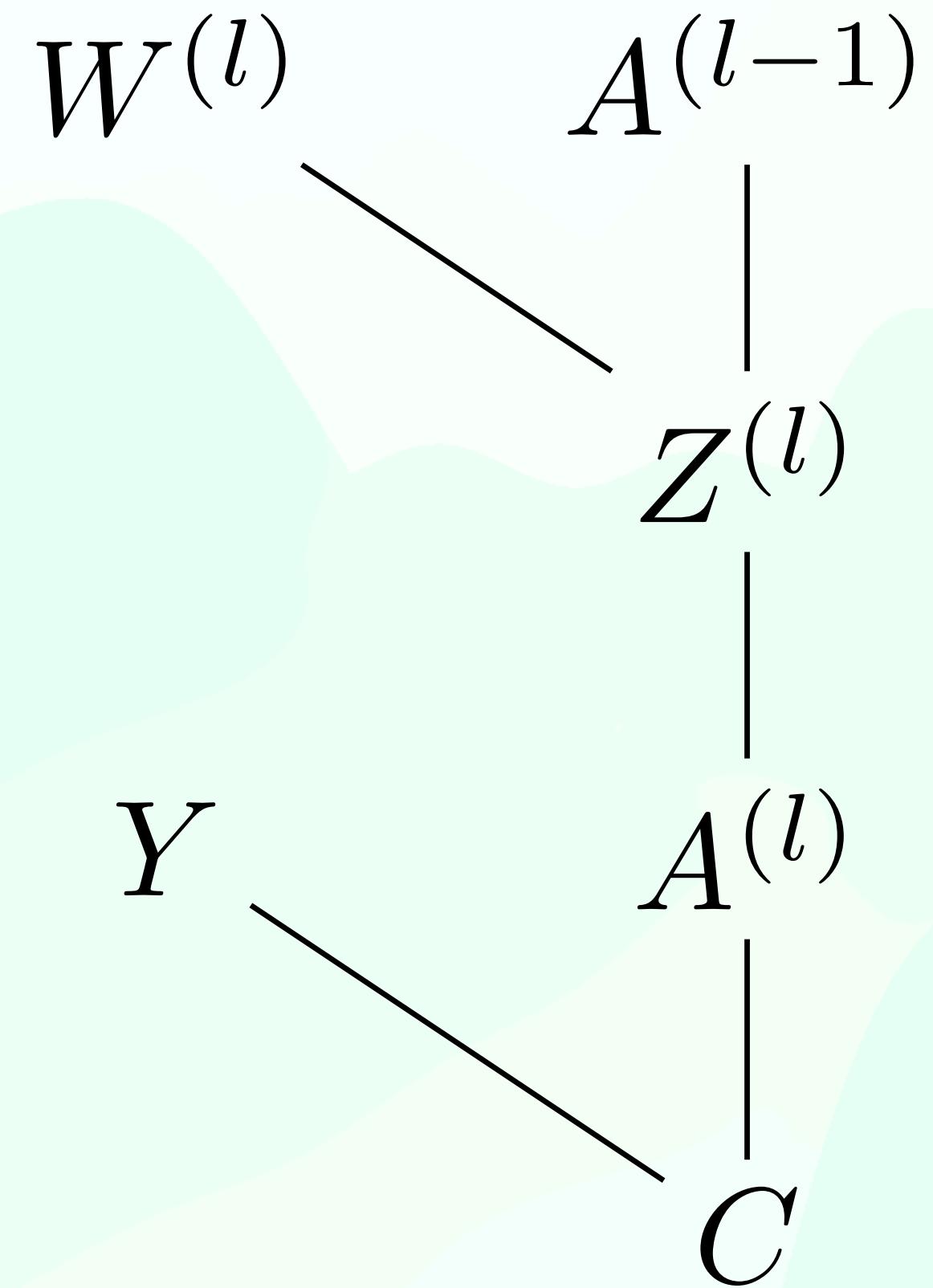
[1] <https://arxiv.org/pdf/1412.0233>

# Gradient Descent

- Berechne Gradientenvektor  $\nabla C$ , der die Richtung des steilsten Anstiegs der Kostenfunktion angibt
- Mache einen Schritt in Richtung  $-\nabla C$
- Wiederhole beides so lange, bis ausreichende Konvergenz zu einem lokalen Minimum vorliegt



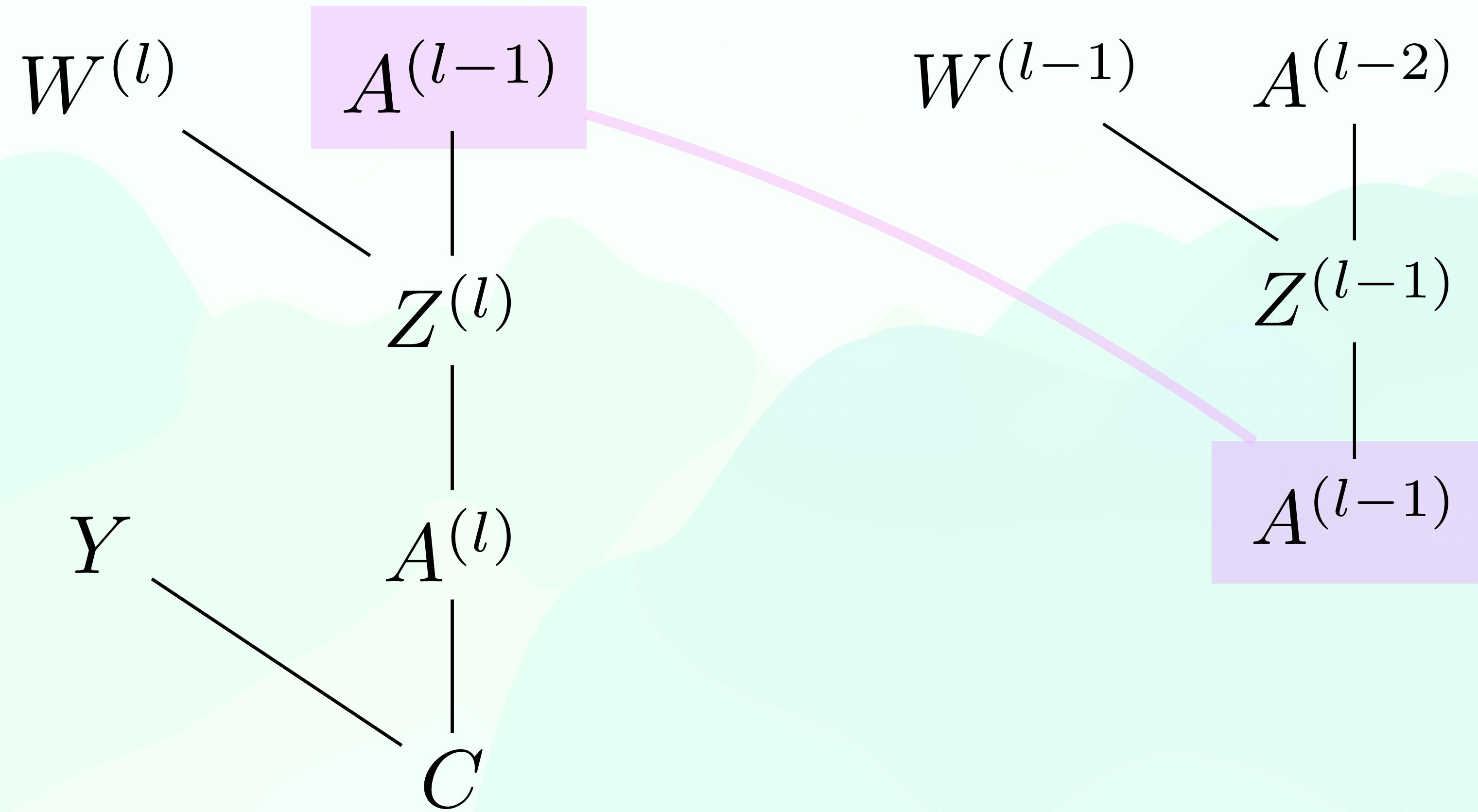
# Ableitung der Kostenfunktion



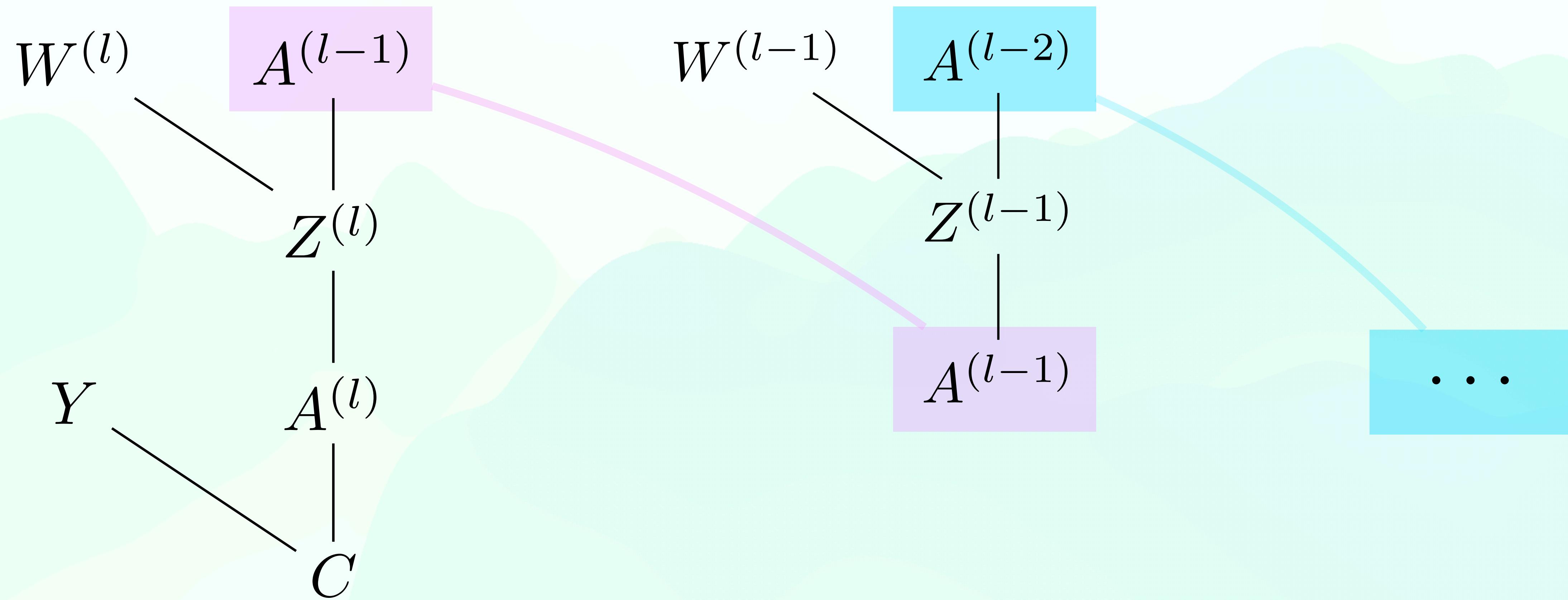
# Ableitung der Kostenfunktion



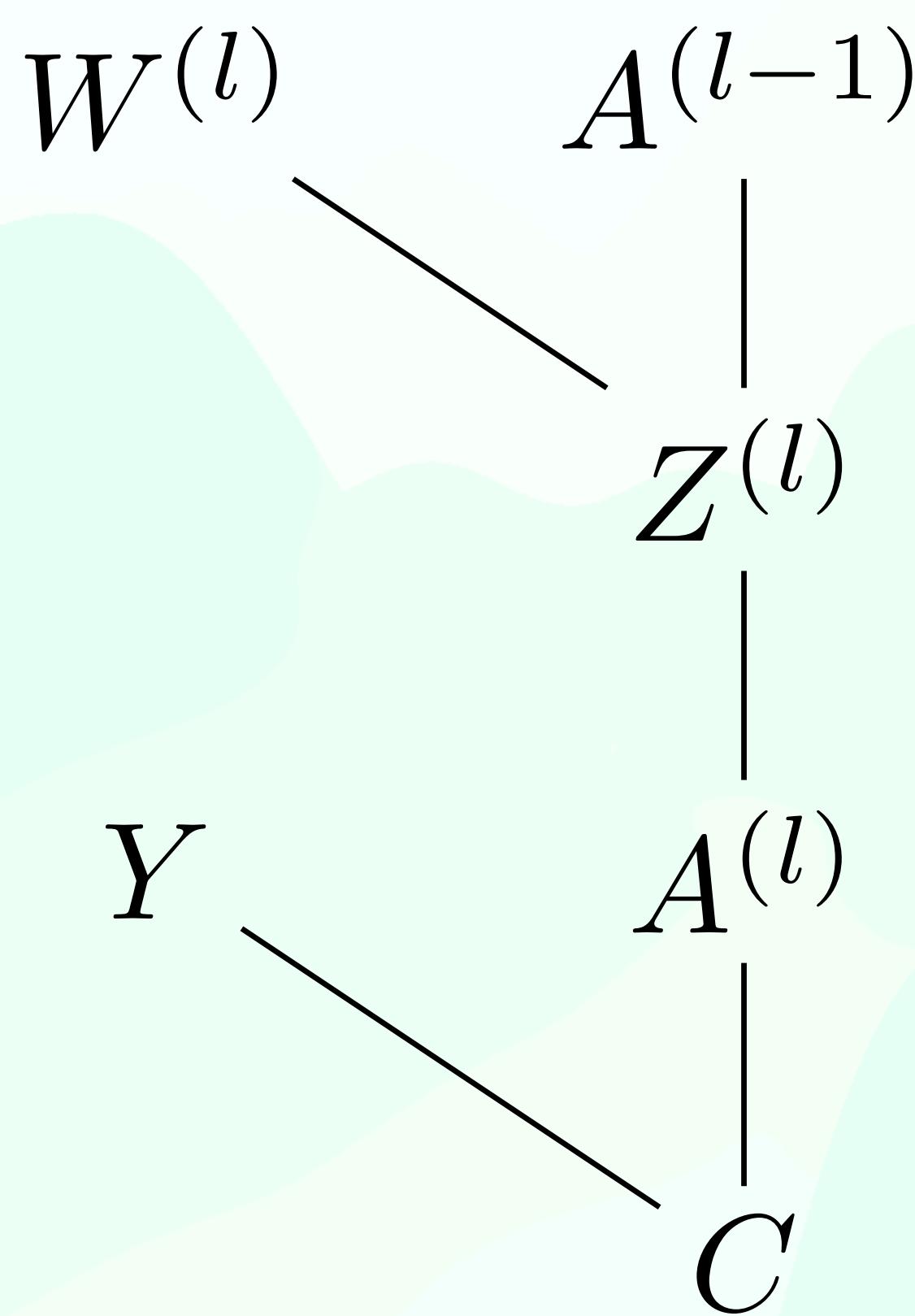
# Ableitung der Kostenfunktion



# Ableitung der Kostenfunktion

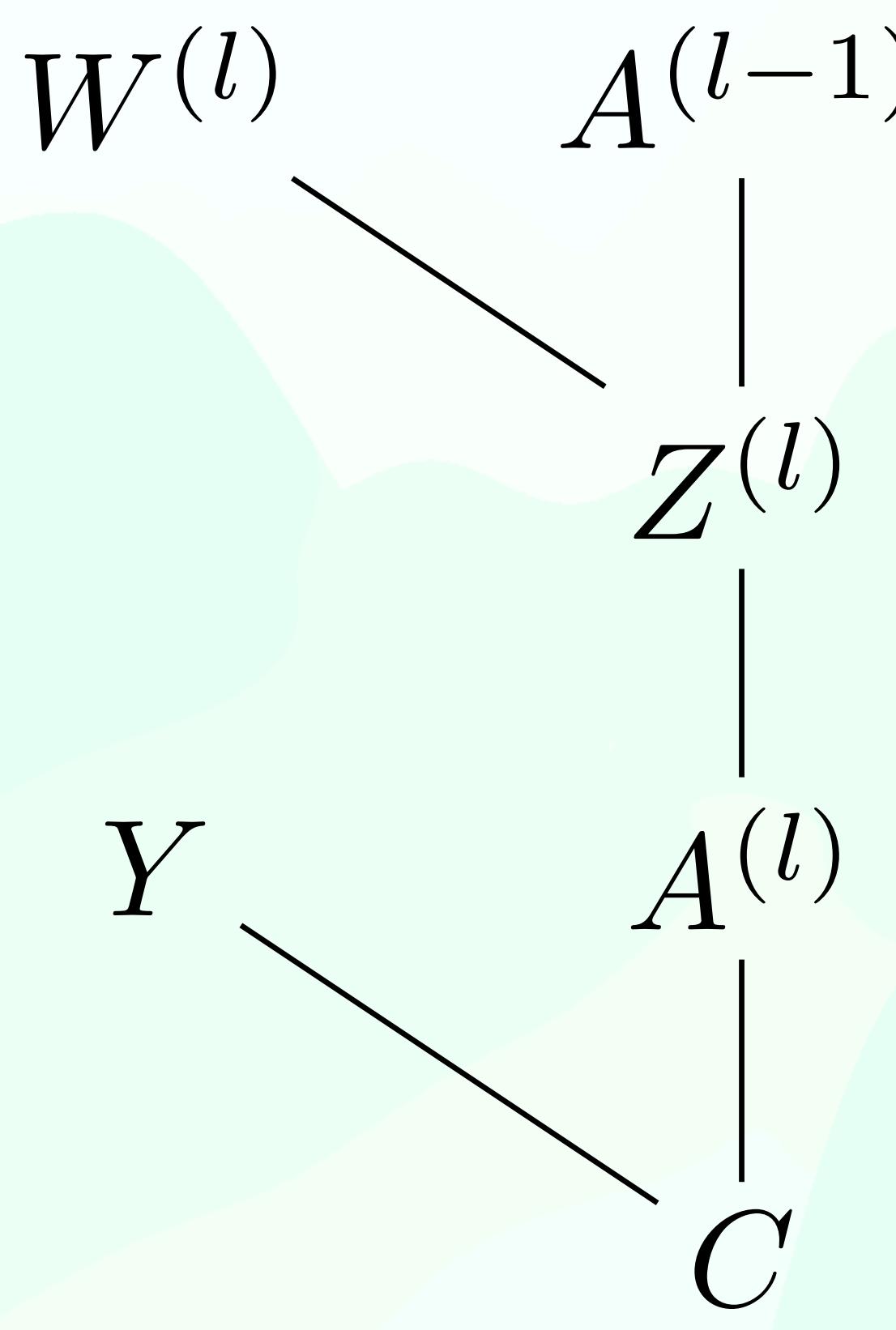


# Ableitung der Kostenfunktion



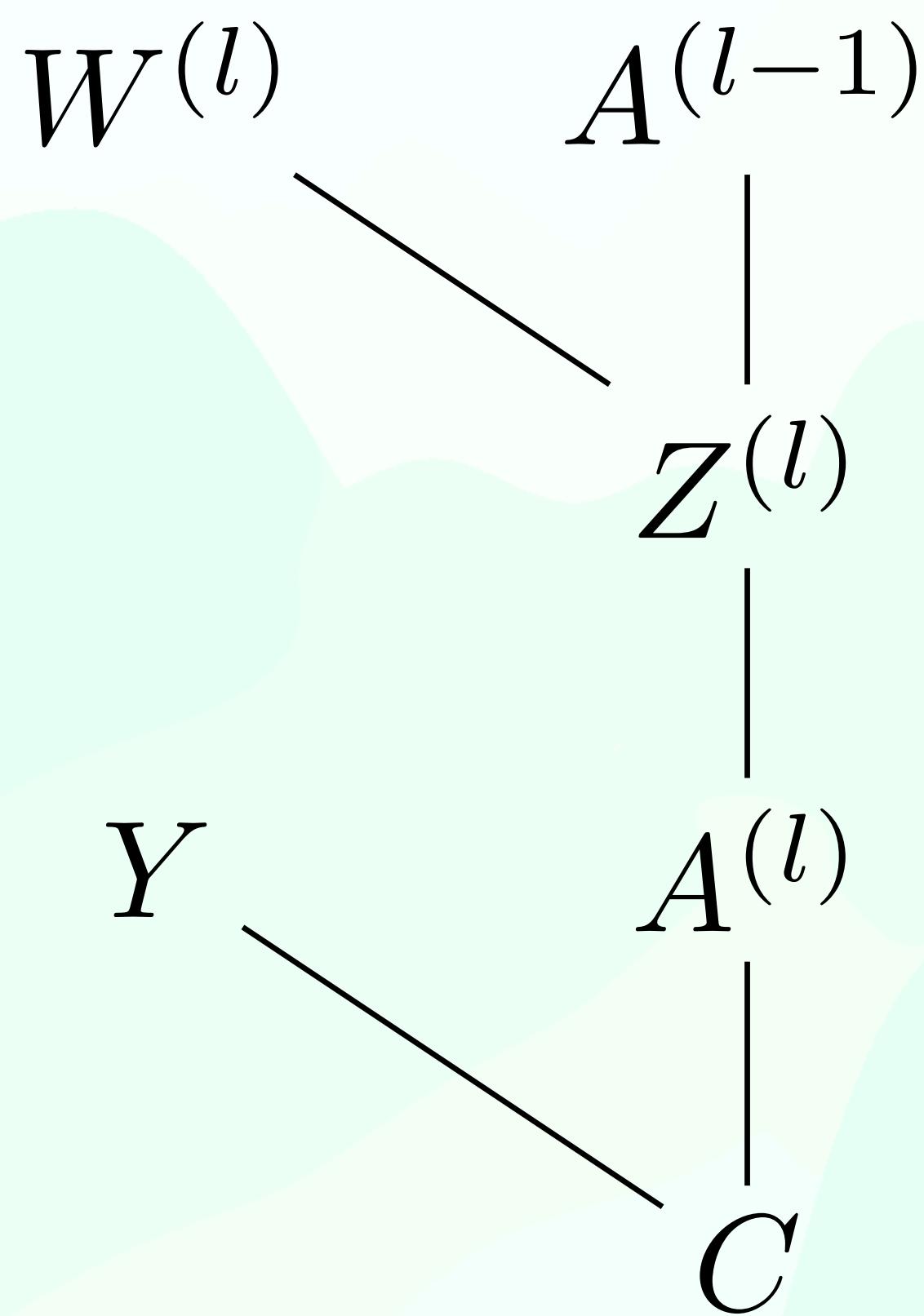
- Aktivierungen des Layers  $l - 1$  können nicht direkt beeinflusst werden, sondern nur durch Anpassung der Gewichte im Layer  $l - 1$

# Ableitung der Kostenfunktion



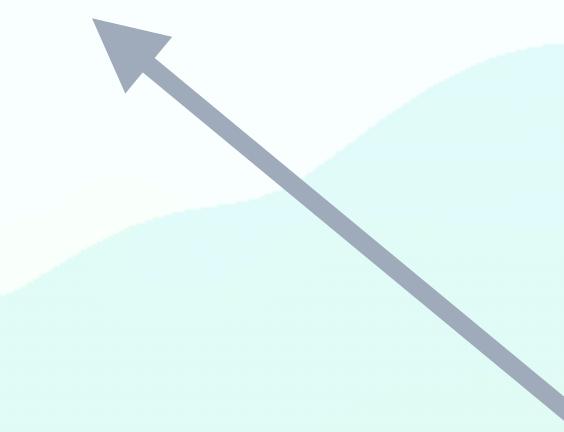
- Aktivierungen des Layers  $l - 1$  können nicht direkt beeinflusst werden, sondern nur durch Anpassung der Gewichte im Layer  $l - 1$
- Idee der **Backpropagation**:
  - Berechne, in welche Richtung die Gewichte zum Output-Layer angepasst werden müssen, um Kosten zu verringern
  - Benutze diese Information, um diese Richtung auch für die Gewichte des vorherigen Layers zu berechnen
  - Wiederhole diese Schritte von Layer zu Layer „rückwärts“ durch das Netz bis zu den Gewichten vom Input-Layer zum ersten Hidden-Layer

# Ableitung der Kostenfunktion



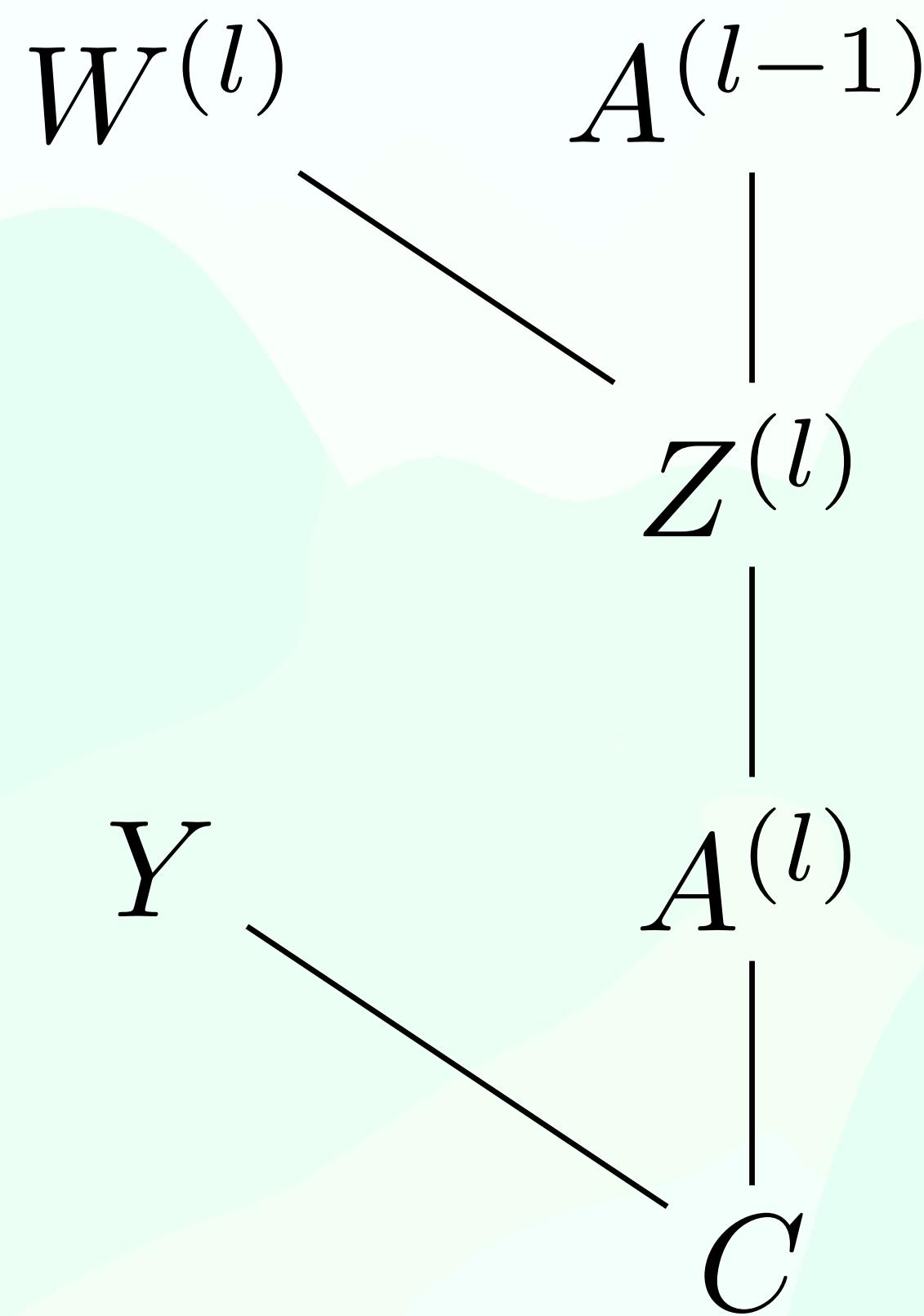
Für jedes  $w_{jk}^{(l)}$  gesucht:

$$\frac{\partial C}{\partial w_{jk}^{(l)}}$$



Welchen Einfluss auf  $C$  hat  
eine winzige Änderung in  $w_{jk}^{(l)}$ ?

# Ableitung der Kostenfunktion



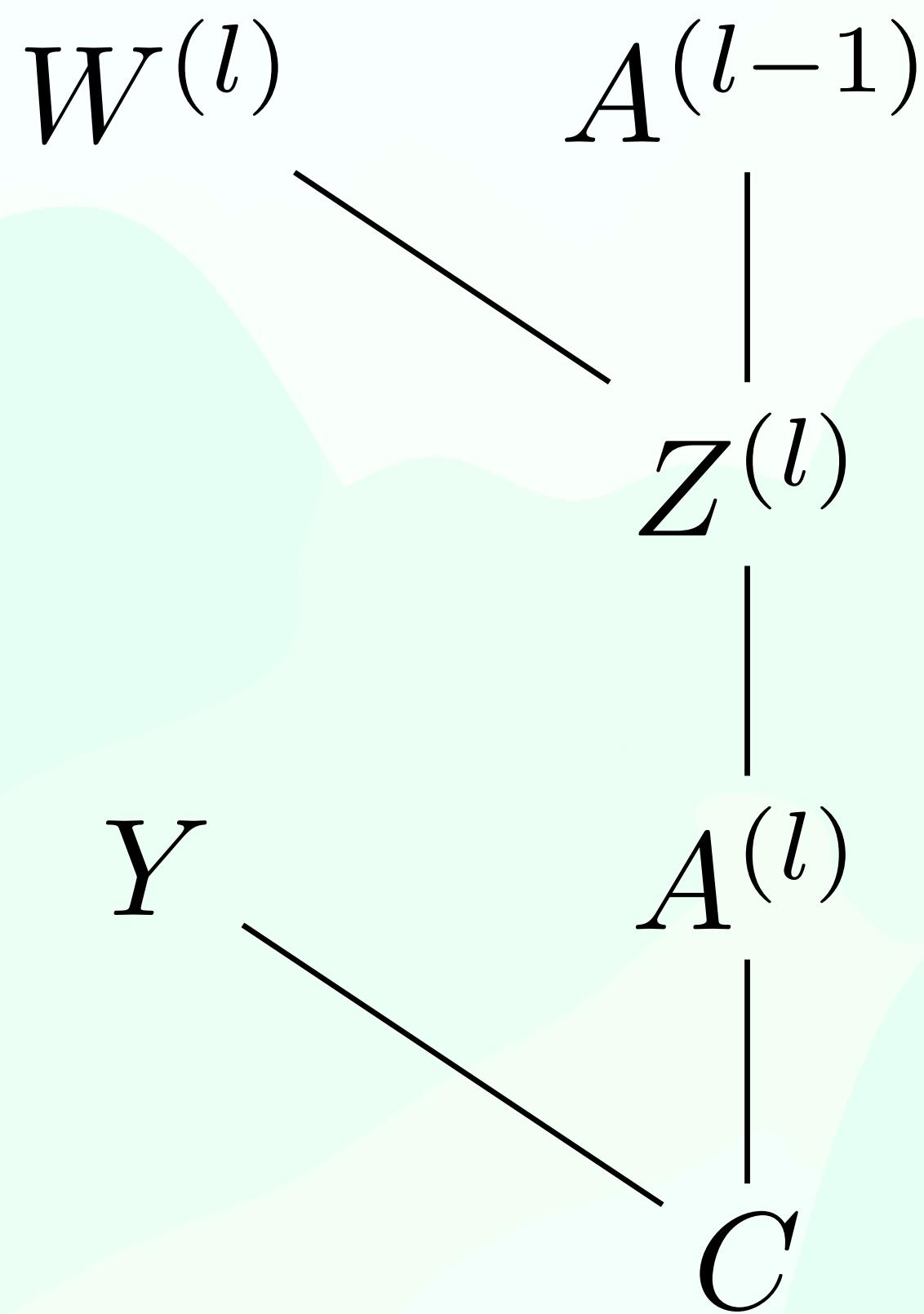
Für jedes  $w_{jk}^{(l)}$  gesucht:

$$\frac{\partial C}{\partial w_{jk}^{(l)}}$$

Die Ableitung von  $C$  nach  $w_{jk}^{(l)}$

Welchen Einfluss auf  $C$  hat  
eine winzige Änderung in  $w_{jk}^{(l)}$ ?

# Ableitung der Kostenfunktion



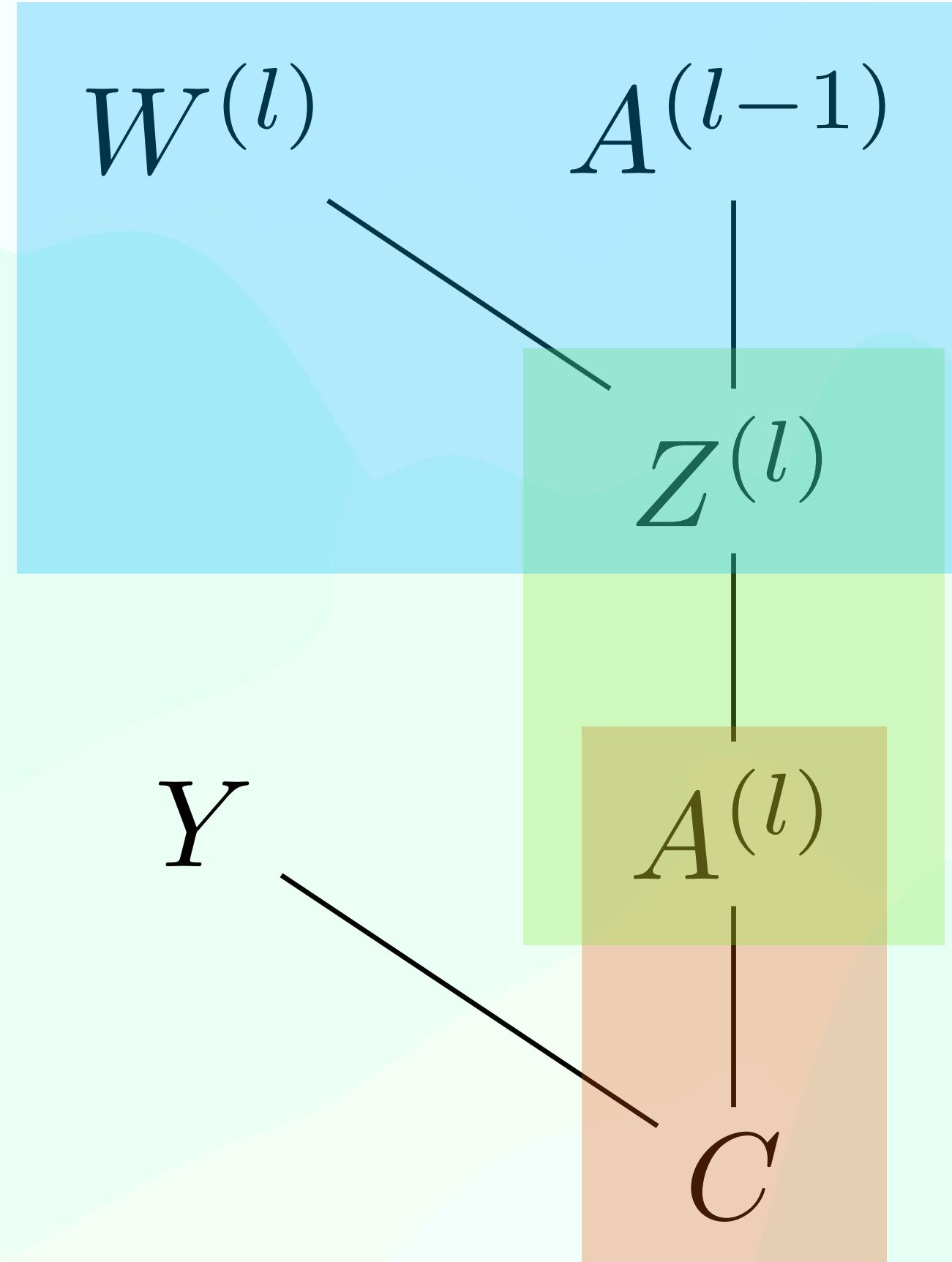
Für jedes  $w_{jk}^{(l)}$  gesucht:  $\frac{\partial C}{\partial w_{jk}^{(l)}}$

Aus Kettenregel der Differentialrechnung folgt:

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$

1

# Ableitung der Kostenfunktion



Für jedes  $w_{jk}^{(l)}$  gesucht:  $\frac{\partial C}{\partial w_{jk}^{(l)}}$

Aus Kettenregel der Differentialrechnung folgt:

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$

1

The equation is shown with three colored boxes: orange for  $\frac{\partial C}{\partial a_j^{(l)}}$ , green for  $\frac{\partial a_j^{(l)}}{\partial z_j^{(l)}}$ , and blue for  $\frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$ . Three grey arrows point from the right side of each box to the left side of the next box in sequence.

Alle Ableitungen entlang der  
Kette werden multipliziert

# Ableitung der Kostenfunktion

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$

Welchen Einfluss auf  $C$  hat  
eine winzige Änderung in  $a_j^{(l)}$ ?

Welchen Einfluss auf  $a_j^{(l)}$  hat  
eine winzige Änderung in  $z_j^{(l)}$ ?

Welchen Einfluss auf  $z_j^{(l)}$  hat  
eine winzige Änderung in  $w_{jk}^{(l)}$ ?

# Ableitung der Kostenfunktion

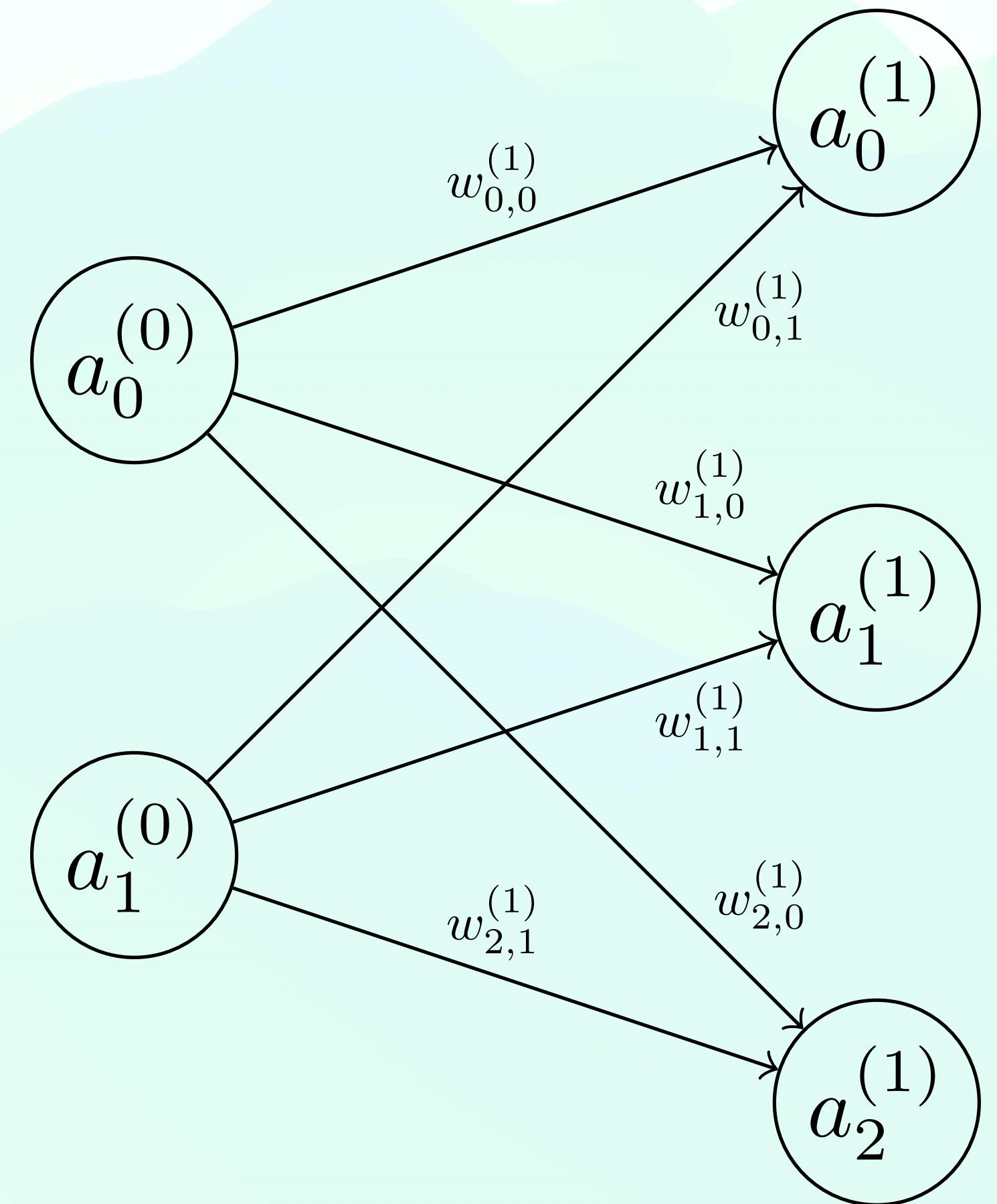
$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$

# Ableitung der Kostenfunktion

- Sei  $n^{(l-1)}$  die Anzahl der Neuronen in Layer  $l - 1$ , dann gilt:

$$z_j^{(l)} = \sum_{m=0}^{n^{(l-1)}-1} w_{jm}^{(l)} a_m^{(l-1)}$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$



# Ableitung der Kostenfunktion

- Sei  $n^{(l-1)}$  die Anzahl der Neuronen in Layer  $l - 1$ , dann gilt:

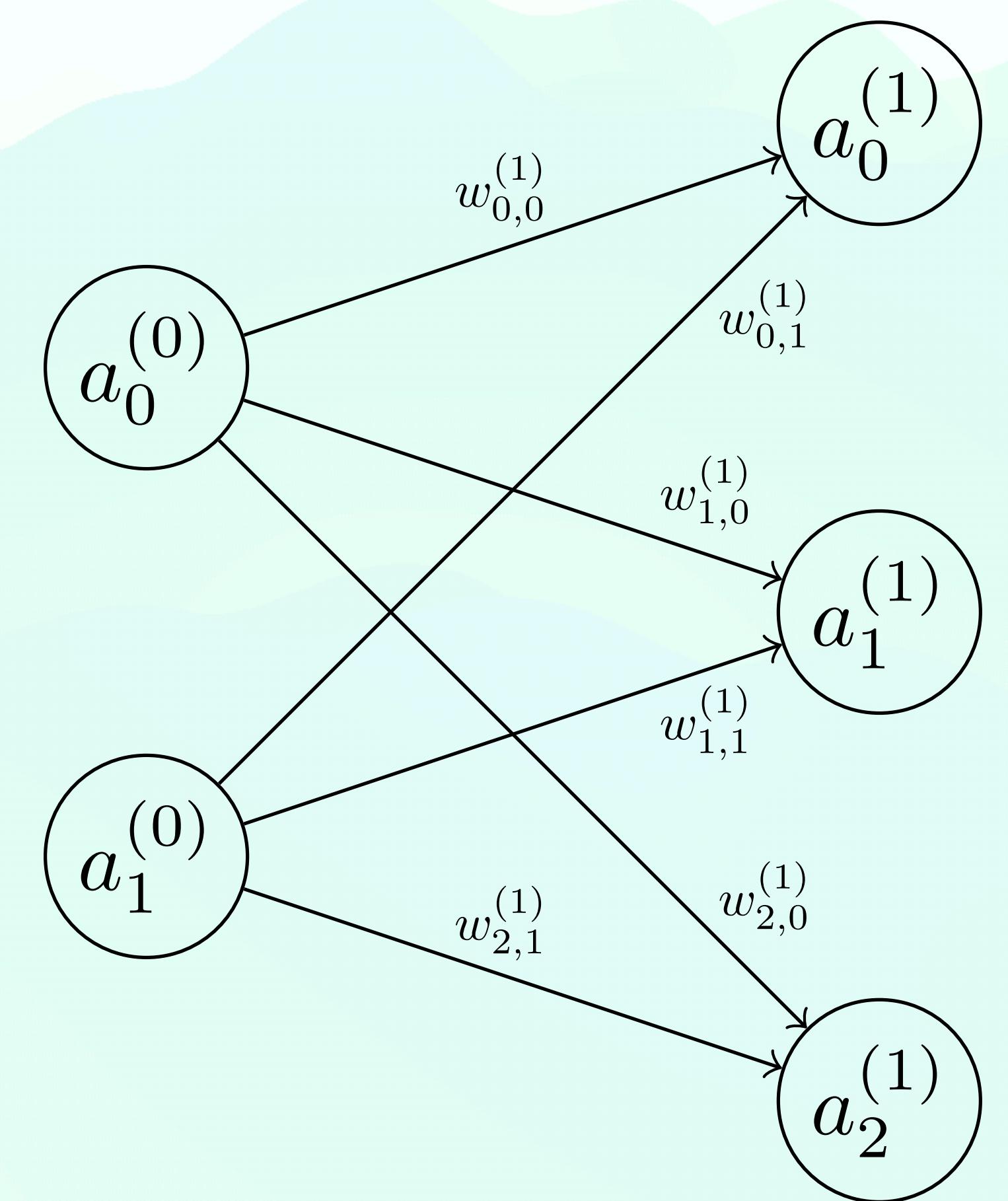
$$z_j^{(l)} = \sum_{m=0}^{n^{(l-1)}-1} w_{jm}^{(l)} a_m^{(l-1)}$$

- Nur ein Term in dieser Summe hängt von  $w_{jk}^{(l)}$  ab, alle anderen Gewichte sind im Sinne dieses Differentials Konstanten.
- Daher gilt:

$$\frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \frac{\partial w_{jk}^{(l)} a_k^{(l-1)}}{\partial w_{jk}^{(l)}} = a_k^{(l-1)}$$

2

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$



# Ableitung der Kostenfunktion

Mit anderen Worten:

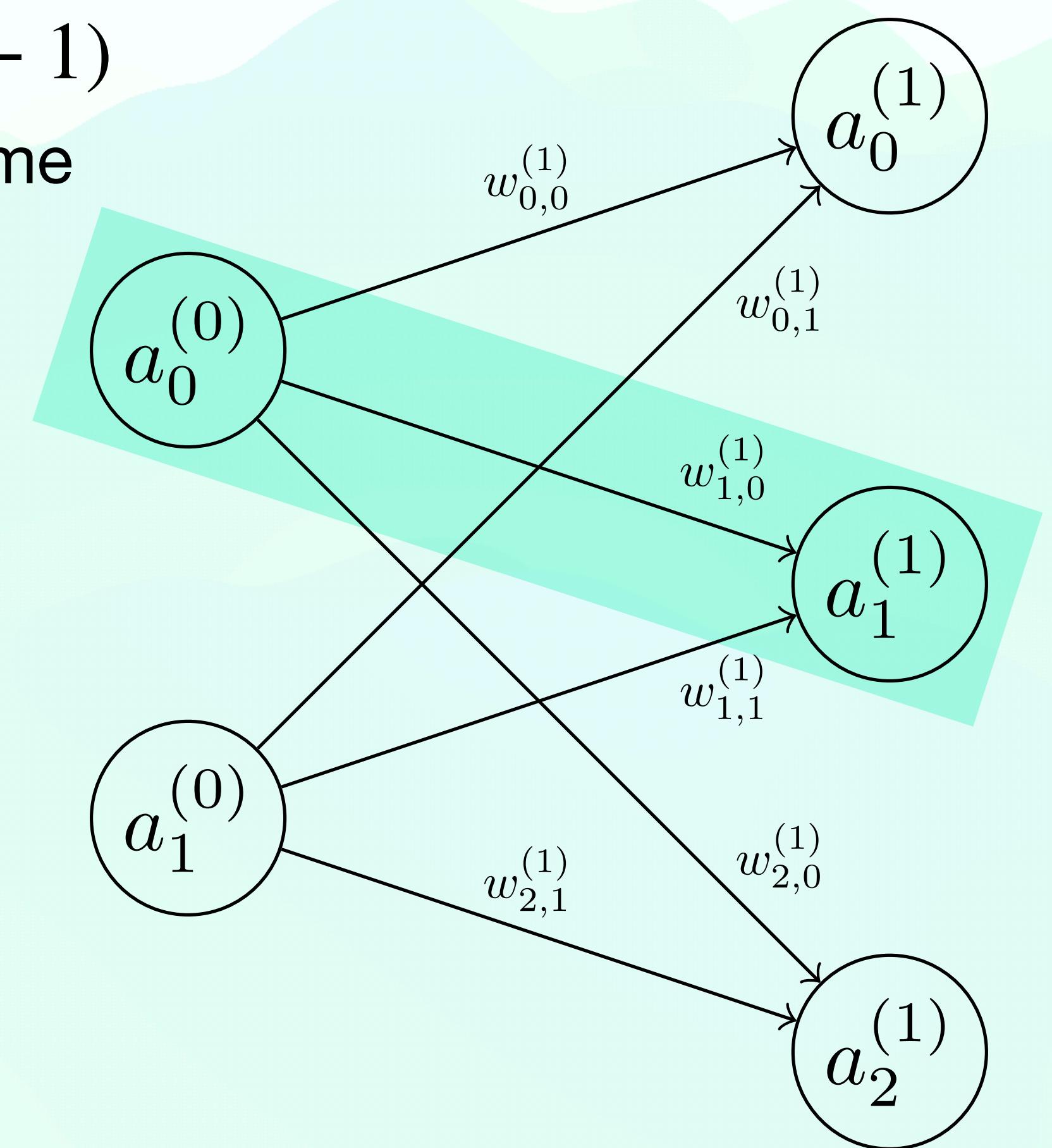
Eine Änderung des Gewichts vom  $k$ -ten Neuron des Layers  $(l - 1)$  zum  $j$ -ten Neuron des Layers  $l$  beeinflusst die gewichtete Summe im  $j$ -ten Neuron in Layer  $l$  genau in der Höhe der Aktivierung des  $k$ -ten Neurons in Layer  $l - 1$ .



$$\frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \frac{\partial w_{jk}^{(l)} a_k^{(l-1)}}{\partial w_{jk}^{(l)}} = a_k^{(l-1)}$$

2

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$



# Ableitung der Kostenfunktion

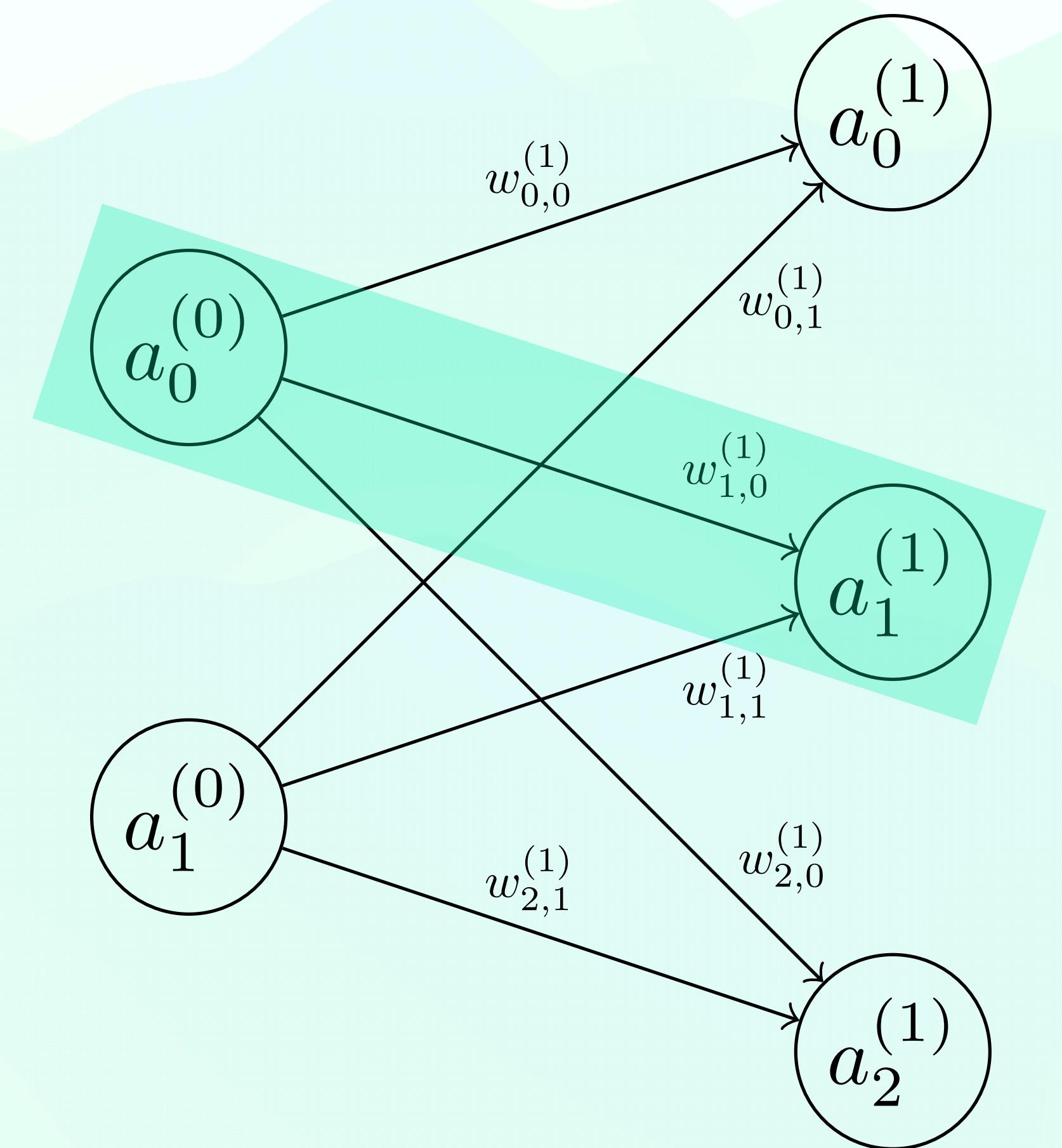
Noch einfacher:

Der Einfluss eines bestimmten **Gewichts** auf die **gewichtete Summe** ist exakt so groß wie die **Aktivierung** des Neurons, von dem es kommt.

$$\frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \frac{\partial w_{jk}^{(l)} a_k^{(l-1)}}{\partial w_{jk}^{(l)}} = a_k^{(l-1)}$$

2

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$



# Ableitung der Kostenfunktion

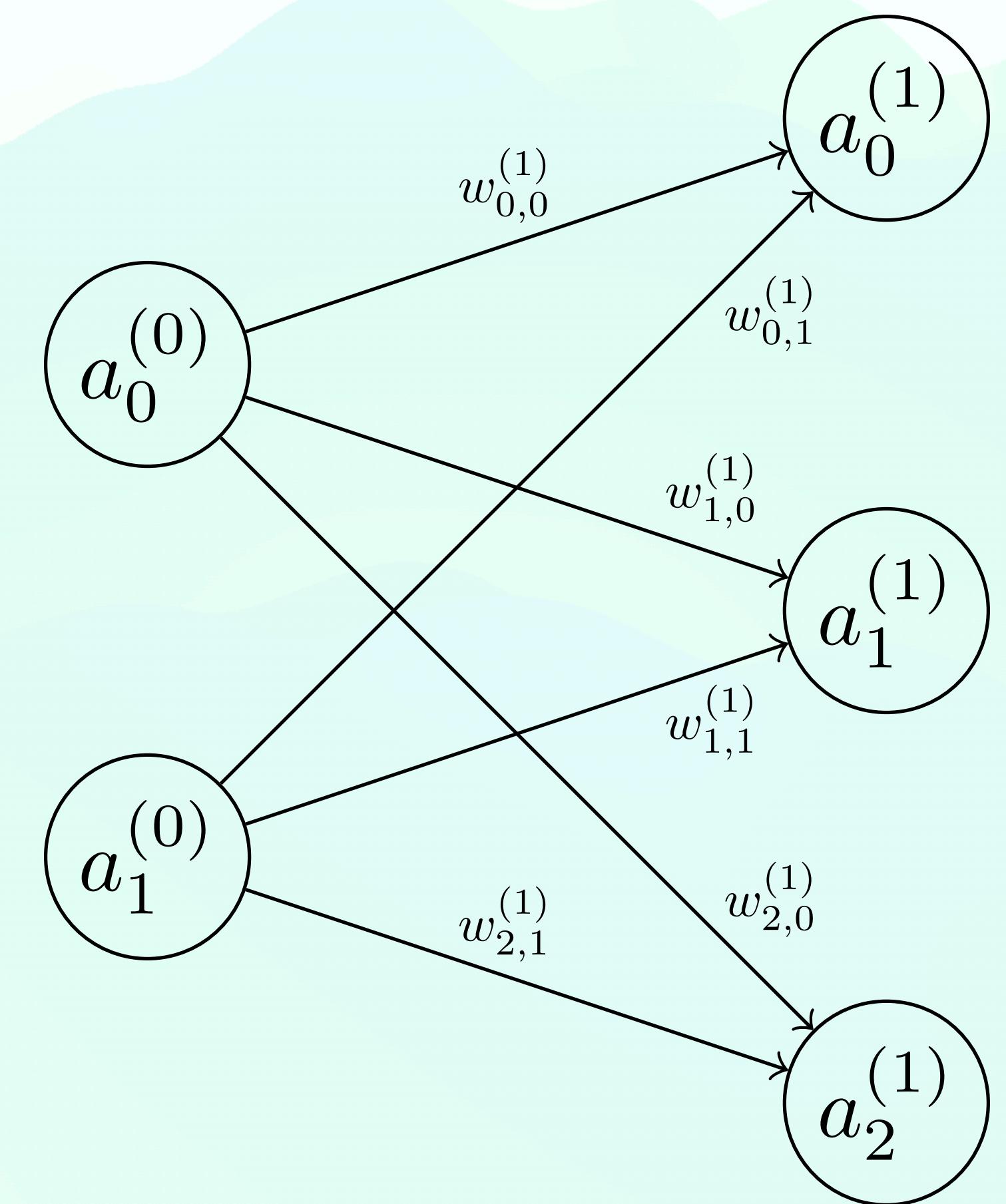
$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$

# Ableitung der Kostenfunktion

- Aktivierung ergibt sich aus Aufruf der Aktivierungsfunktion mit der gewichteten Summe:

$$a_j^{(l)} = \sigma^{(l)}(z_j^{(l)})$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$



# Ableitung der Kostenfunktion

- Aktivierung ergibt sich aus Aufruf der Aktivierungsfunktion mit der gewichteten Summe:

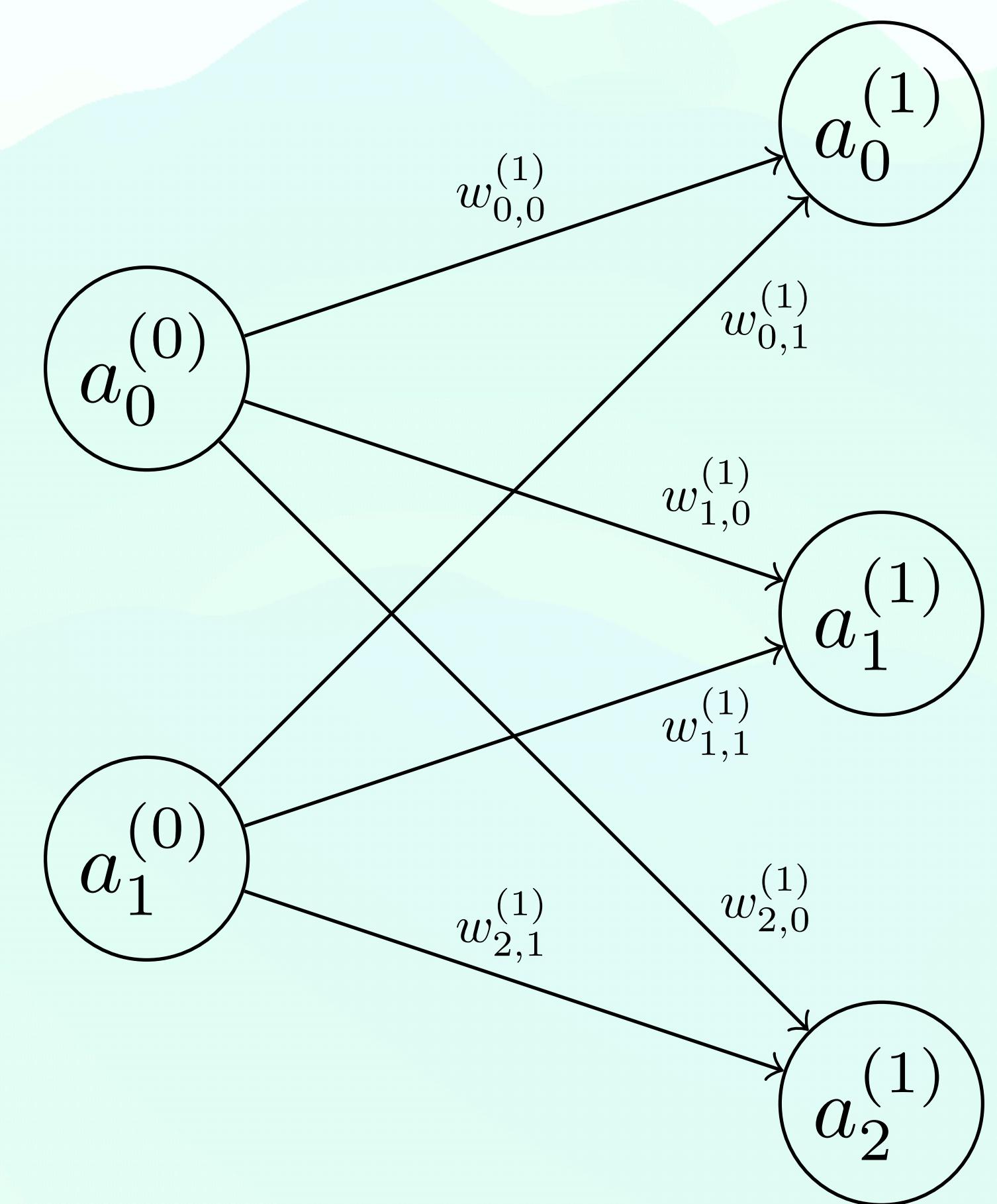
$$a_j^{(l)} = \sigma^{(l)}(z_j^{(l)})$$

- Daher gilt:

$$\frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} = \sigma^{(l)'}(z_j^{(l)})$$

3

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$



# Ableitung der Kostenfunktion

- Aktivierung ergibt sich aus Aufruf der Aktivierungsfunktion mit der gewichteten Summe:

$$a_j^{(l)} = \sigma^{(l)}(z_j^{(l)})$$

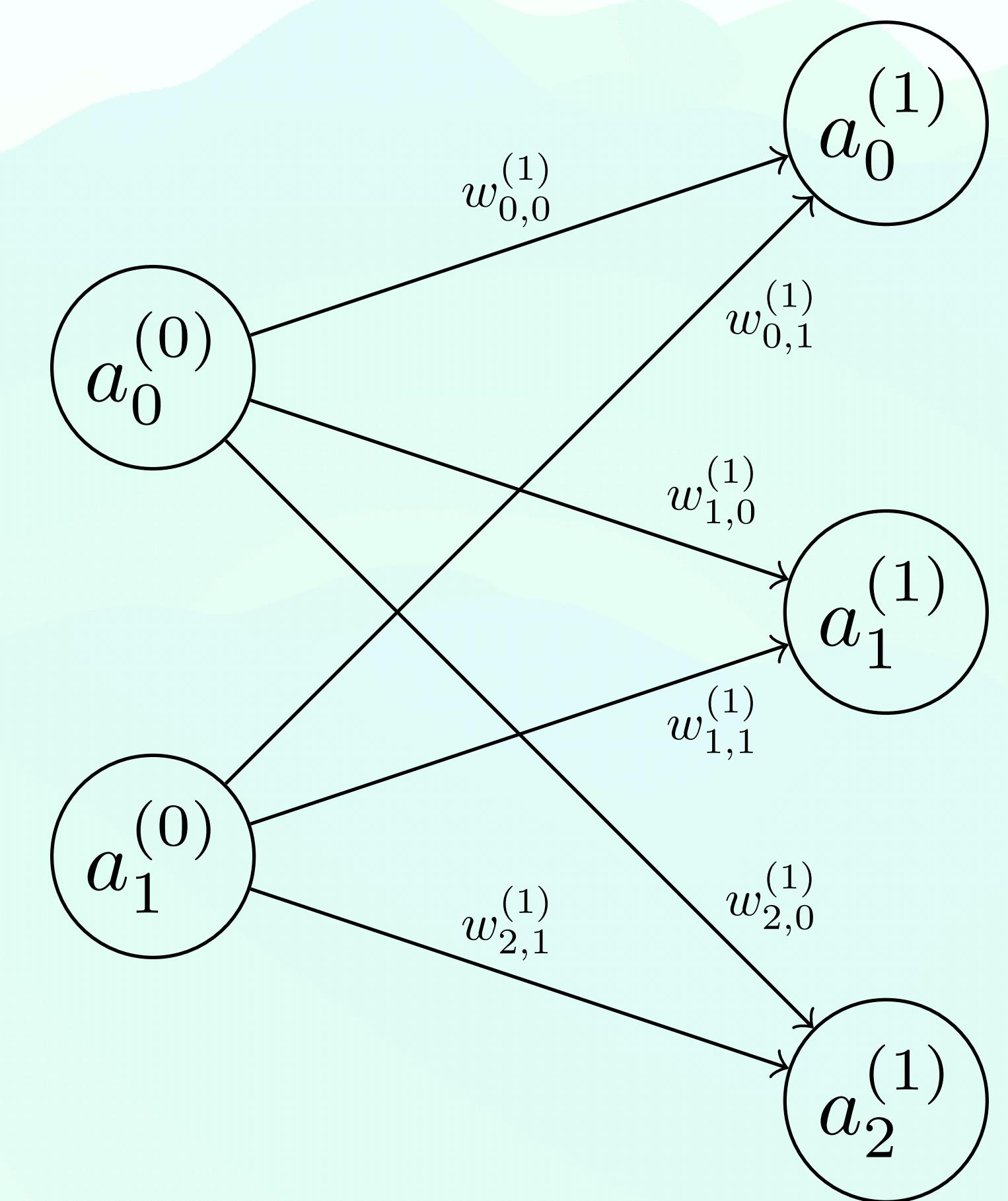
- Daher gilt:

$$\frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} = \sigma^{(l)'}(z_j^{(l)})$$

3

- Wichtig bei der Wahl der Aktivierungsfunktion zu beachten: Berechnung sollte performant möglich sein und keinen **Vanishing Gradient** haben

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$



# Ableitung der Kostenfunktion

$$\frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} = \sigma^{(l)'}(z_j^{(l)})$$

3

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$

- ReLU und Sigmoid sind besonders einfach zu differenzieren:

$f(x)$	$f'(x)$
$ReLU(x) = \begin{cases} x & \text{wenn } x > 0 \\ 0 & \text{sonst} \end{cases}$	$ReLU'(x) = \begin{cases} 1 & \text{wenn } x > 0 \\ 0 & \text{sonst} \end{cases}$
$\sigma(x) = \frac{1}{1 + e^{-x}}$	$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$

# Ableitungen der Aktivierungsfunktionen



```
def relu(z, derivative=False):
    if derivative:
        return numpy.heaviside(z, 1)
    return numpy.maximum(z, 0)

def sigmoid(z, derivative=False):
    if derivative:
        return sigmoid(z) * (1 - sigmoid(z))
    return 1 / (1 + numpy.exp(-z))

def identity(z, derivative=False):
    if derivative:
        return numpy.ones_like(z)
    return z
```

# Ableitungen der Aktivierungsfunktionen



```
def relu(z, derivative=False):
    if derivative:
        return numpy.heaviside(z, 1)
    return numpy.maximum(z, 0)

def sigmoid(z, derivative=False):
    if derivative:
        return sigmoid(z) * (1 - sigmoid(z))
    return 1 / (1 + numpy.exp(-z))

def identity(z, derivative=False):
    if derivative:
        return numpy.ones_like(z)
    return z
```

Wir erweitern die Aktivierungsfunktionen jeweils um die Möglichkeit, auch ihre eigene **Ableitung** für  $z$  zu berechnen. Um die Ableitung zu erhalten, können wir dann künftig für den Parameter `derivative` den Wert `True` übergeben. Standardmäßig steht dieser Parameter aber auf `False`, ohne Parameter berechnet die Funktion also weiterhin die Aktivierung.

# Ableitungen der Aktivierungsfunktionen



```
def relu(z, derivative=False):
    if derivative:
        return numpy.heaviside(z, 1)
    return numpy.maximum(z, 0)

def sigmoid(z, derivative=False):
    if derivative:
        return sigmoid(z) * (1 - sigmoid(z))
    return 1 / (1 + numpy.exp(-z))

def identity(z, derivative=False):
    if derivative:
        return numpy.ones_like(z)
    return z
```

Im Falle der ReLU-Funktion benutzen wir für den Fall, dass die Ableitung berechnet werden soll, die Funktion `numpy.heaviside`. Sie nimmt eine Matrix entgegen und gibt eine neue Matrix gleichen Formats zurück, in der alle Elemente 1 sind, die in der originalen Matrix größer als 0 waren, und alle Elemente 0 sind, die in der originalen Matrix kleiner als 0 waren. Dies entspricht genau der Ableitung von ReLU.

Der zweite Parameter 1 sagt aus, dass Werte gleich 0 in der Ergebnismatrix den Wert 1 bekommen sollen.

# Ableitungen der Aktivierungsfunktionen



```
def relu(z, derivative=False):
    if derivative:
        return numpy.heaviside(z, 1)
    return numpy.maximum(z, 0)

def sigmoid(z, derivative=False):
    if derivative:
        return sigmoid(z) * (1 - sigmoid(z))
    return 1 / (1 + numpy.exp(-z))

def identity(z, derivative=False):
    if derivative:
        return numpy.ones_like(z)
    return z
```

Für die Sigmoid-Funktion berechnen wir die Ableitung nach unserer **rekursiven** Berechnungsvorschrift.

# Ableitungen der Aktivierungsfunktionen



```
def relu(z, derivative=False):
    if derivative:
        return numpy.heaviside(z, 1)
    return numpy.maximum(z, 0)

def sigmoid(z, derivative=False):
    if derivative:
        return sigmoid(z) * (1 - sigmoid(z))
    return 1 / (1 + numpy.exp(-z))

def identity(z, derivative=False):
    if derivative:
        return numpy.ones_like(z)
    return z
```

Die Ableitung der Identitätsfunktion  $f(x) = x$  hat triviale Weise für jedes  $x$  den Wert 1.

# Ableitung der Kostenfunktion

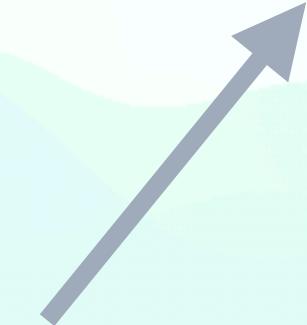
$$\frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} = \sigma^{(l)'}(z_j^{(l)})$$

3

Das bedeutet im Falle von ReLU als Aktivierungsfunktion:  
Da der Wert der Ableitung immer entweder 0 oder 1 ist,  
wird entweder das gesamte Produkt 0, oder es vereinfacht  
sich auf die beiden anderen Differentialterme.

$$ReLU'(x) = \begin{cases} 1 & \text{wenn } x > 0 \\ 0 & \text{sonst} \end{cases}$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$



# Ableitung der Kostenfunktion

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$

# Ableitung der Kostenfunktion

- Kosten nach Squared Error Loss:

$$C = (A^{(l)} - Y)^2$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$

# Ableitung der Kostenfunktion

- Kosten nach Squared Error Loss:

$$C = (A^{(l)} - Y)^2$$

- Wir betrachten die einzelne Neuronenaktivierung  $a_j^{(l)}$ , und sie fließt nur durch den Term  $(a_j^{(l)} - y_j)^2$  in die Kosten ein. Für diesen gilt die normale Polynomableitung:

$$\frac{\partial C}{\partial a_j^{(l)}} = 2 \left( a_j^{(l)} - y_j \right)$$

4

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$

# Ableitung der Kostenfunktion

- Kosten nach Squared Error Loss:

$$C = (A^{(l)} - Y)^2$$

- Wir betrachten die einzelne Neuronenaktivierung  $a_j^{(l)}$ , und sie fließt nur durch den Term  $(a_j^{(l)} - y_j)^2$  in die Kosten ein. Für diesen gilt die normale Polynomableitung:

$$\frac{\partial C}{\partial a_j^{(l)}} = 2 \left( a_j^{(l)} - y_j \right)$$

4

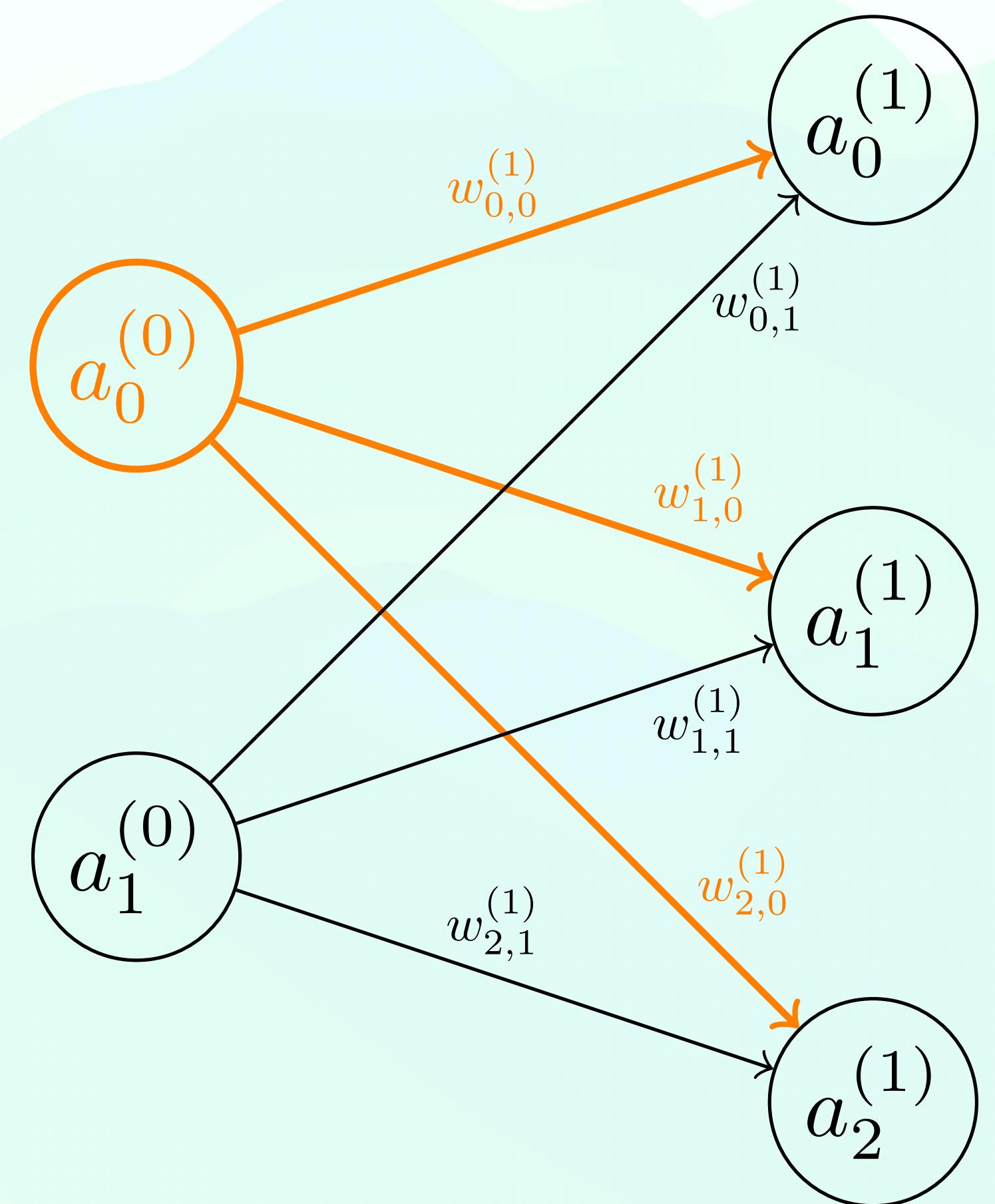
- Diese Berechnung ist natürlich nur im **Output-Layer** möglich, da nur hier die Kosten direkt von den erwarteten Outputs  $y_j$  abhängen!

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$

# Ableitung der Kostenfunktion

- Die Aktivierung eines **inneren Neurons** beeinflusst die Kosten im nächsten Layer durch **alle ausgehenden Gewichte**

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$

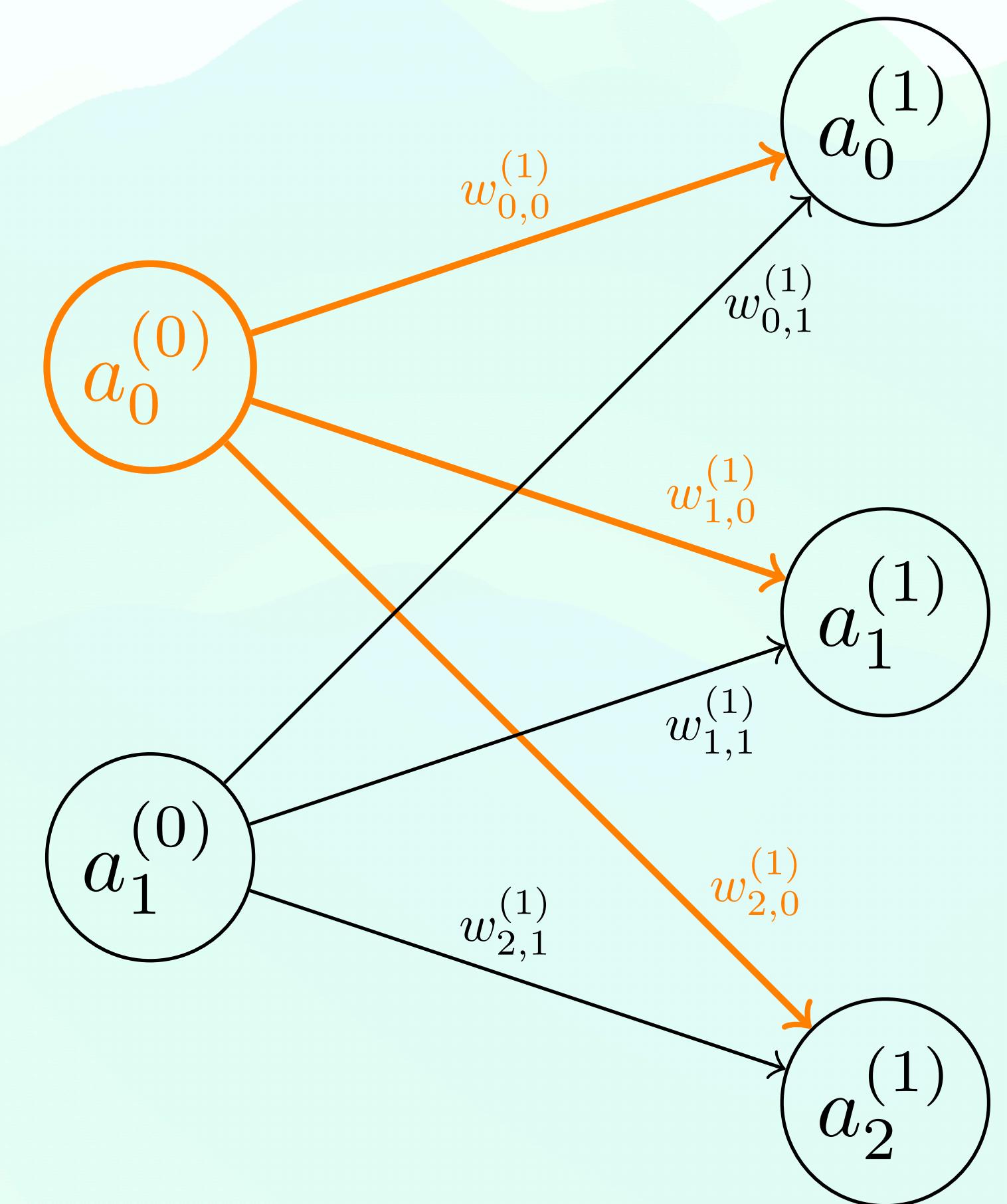


# Ableitung der Kostenfunktion

- Die Aktivierung eines **inneren Neurons** beeinflusst die Kosten im nächsten Layer durch **alle ausgehenden Gewichte**
- Daher müssen alle diese Einflüsse addiert werden:

$$\frac{\partial C}{\partial a_j^{(l)}} = \sum_{i=0}^{n^{(l+1)}-1} \left( \frac{\partial C}{\partial a_i^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}} \frac{\partial z_i^{(l+1)}}{\partial a_j^{(l)}} \right)$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$



# Ableitung der Kostenfunktion

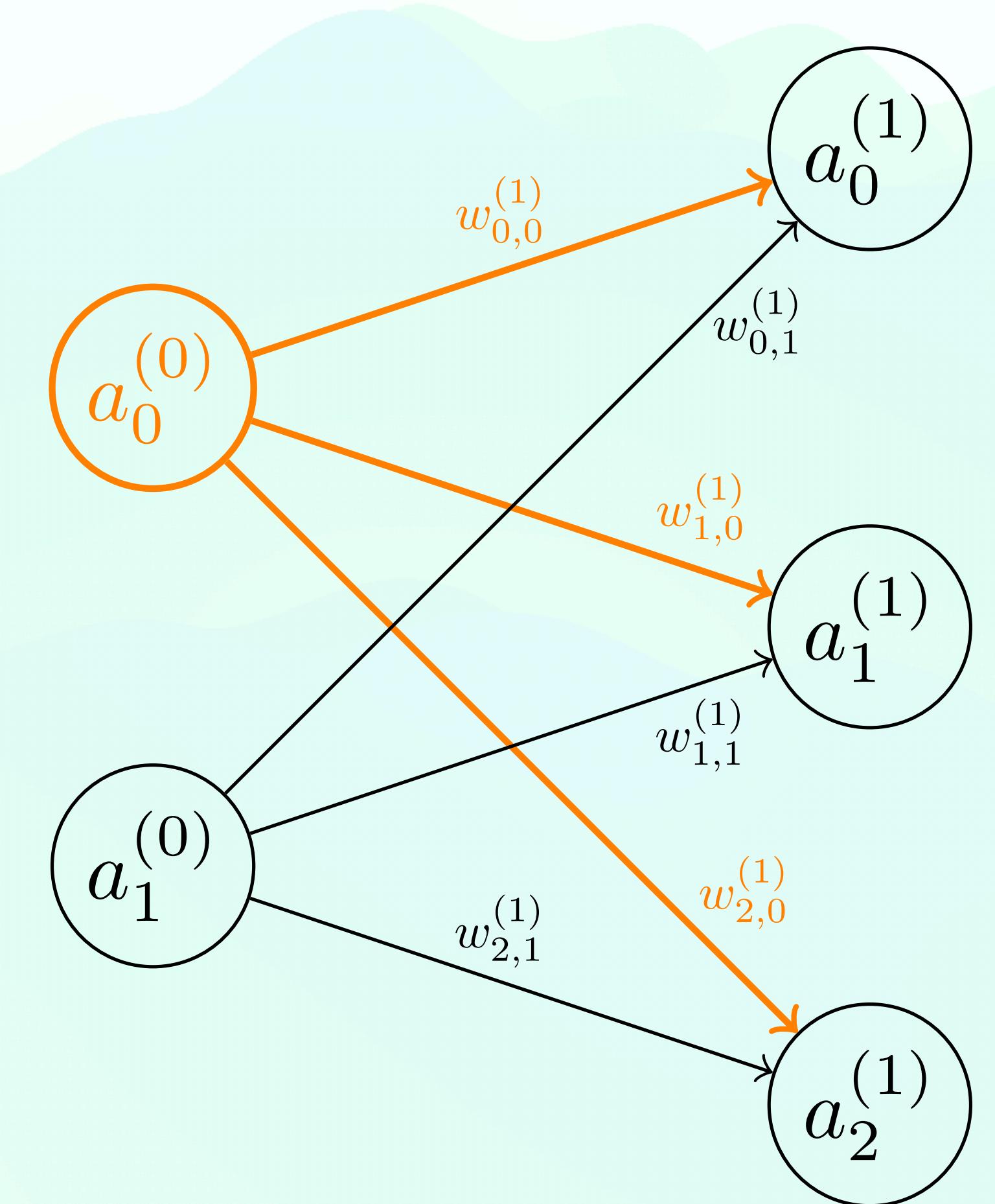
- Die Aktivierung eines **inneren Neurons** beeinflusst die Kosten im nächsten Layer durch **alle ausgehenden Gewichte**
- Daher müssen alle diese Einflüsse addiert werden:

$$\frac{\partial C}{\partial a_j^{(l)}} = \sum_{i=0}^{n^{(l+1)}-1} \left( \frac{\partial C}{\partial a_i^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}} \frac{\partial z_i^{(l+1)}}{\partial a_j^{(l)}} \right)$$



Erneut die Kettenregel: Jede „Kette“, also jeder Pfad, durch den  $a_j^{(l)}$  die Kosten beeinflusst, wird in sich multipliziert, und die einzelnen Ketten dann addiert

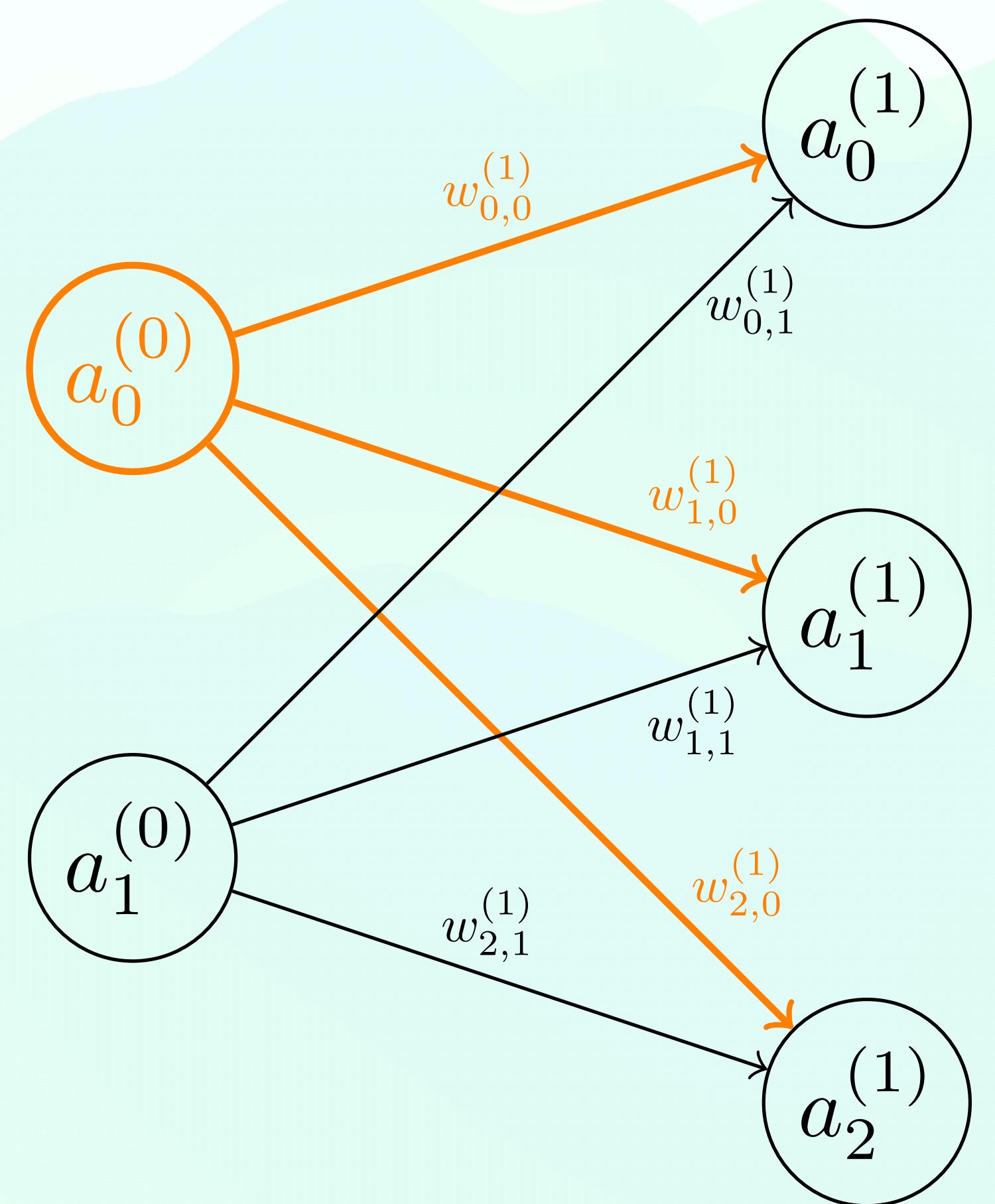
$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$



# Ableitung der Kostenfunktion

$$\frac{\partial C}{\partial a_j^{(l)}} = \sum_{i=0}^{n^{(l+1)}-1} \left( \frac{\partial C}{\partial a_i^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}} \frac{\partial z_i^{(l+1)}}{\partial a_j^{(l)}} \right)$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$



# Ableitung der Kostenfunktion

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$

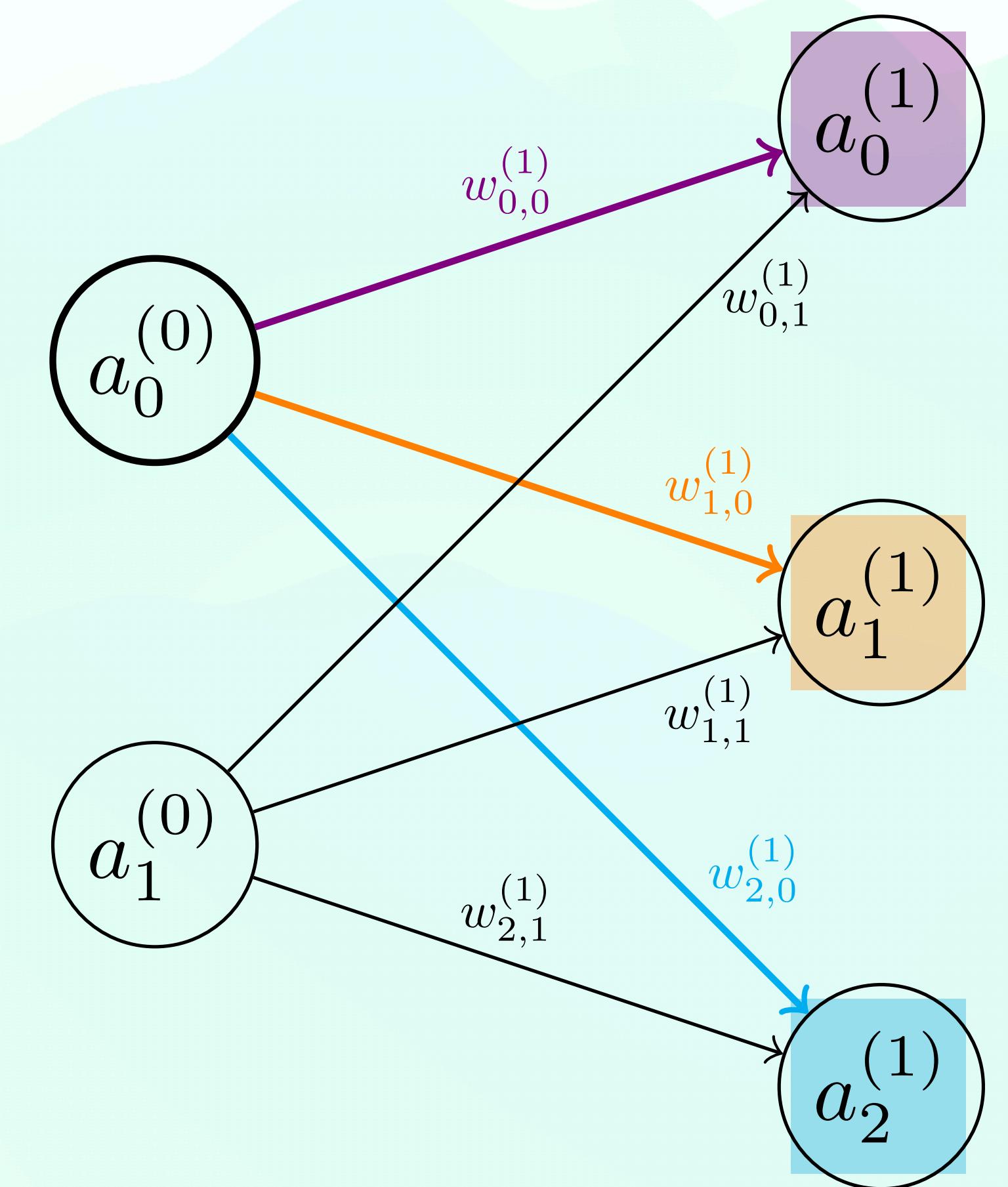
$$\frac{\partial C}{\partial a_j^{(l)}} = \sum_{i=0}^{n^{(l+1)}-1} \left( \frac{\partial C}{\partial a_i^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}} \frac{\partial z_i^{(l+1)}}{\partial a_j^{(l)}} \right)$$

- Für den Ausschnitt aus dem Beispielnetz rechts bedeutet das:

$$\frac{\partial C}{\partial a_0^{(0)}} = \frac{\partial C}{\partial a_0^{(1)}} \frac{\partial a_0^{(1)}}{\partial z_0^{(1)}} \frac{\partial z_0^{(1)}}{\partial a_0^{(0)}} +$$

$$+ \frac{\partial C}{\partial a_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial a_0^{(0)}} +$$

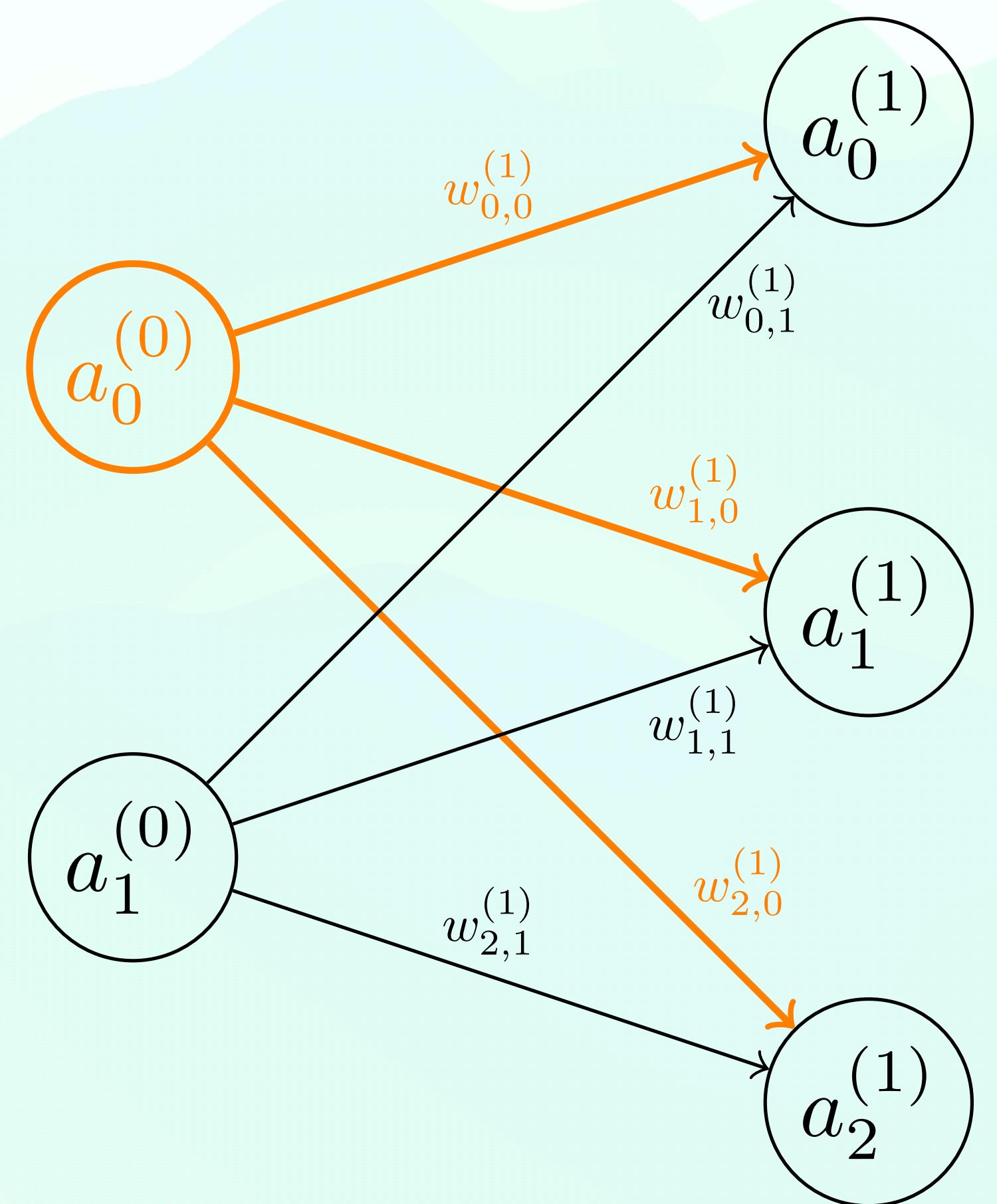
$$+ \frac{\partial C}{\partial a_2^{(1)}} \frac{\partial a_2^{(1)}}{\partial z_2^{(1)}} \frac{\partial z_2^{(1)}}{\partial a_0^{(0)}}$$



# Ableitung der Kostenfunktion

$$\frac{\partial C}{\partial a_j^{(l)}} = \sum_{i=0}^{n^{(l+1)}-1} \left( \frac{\partial C}{\partial a_i^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}} \frac{\partial z_i^{(l+1)}}{\partial a_j^{(l)}} \right)$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$

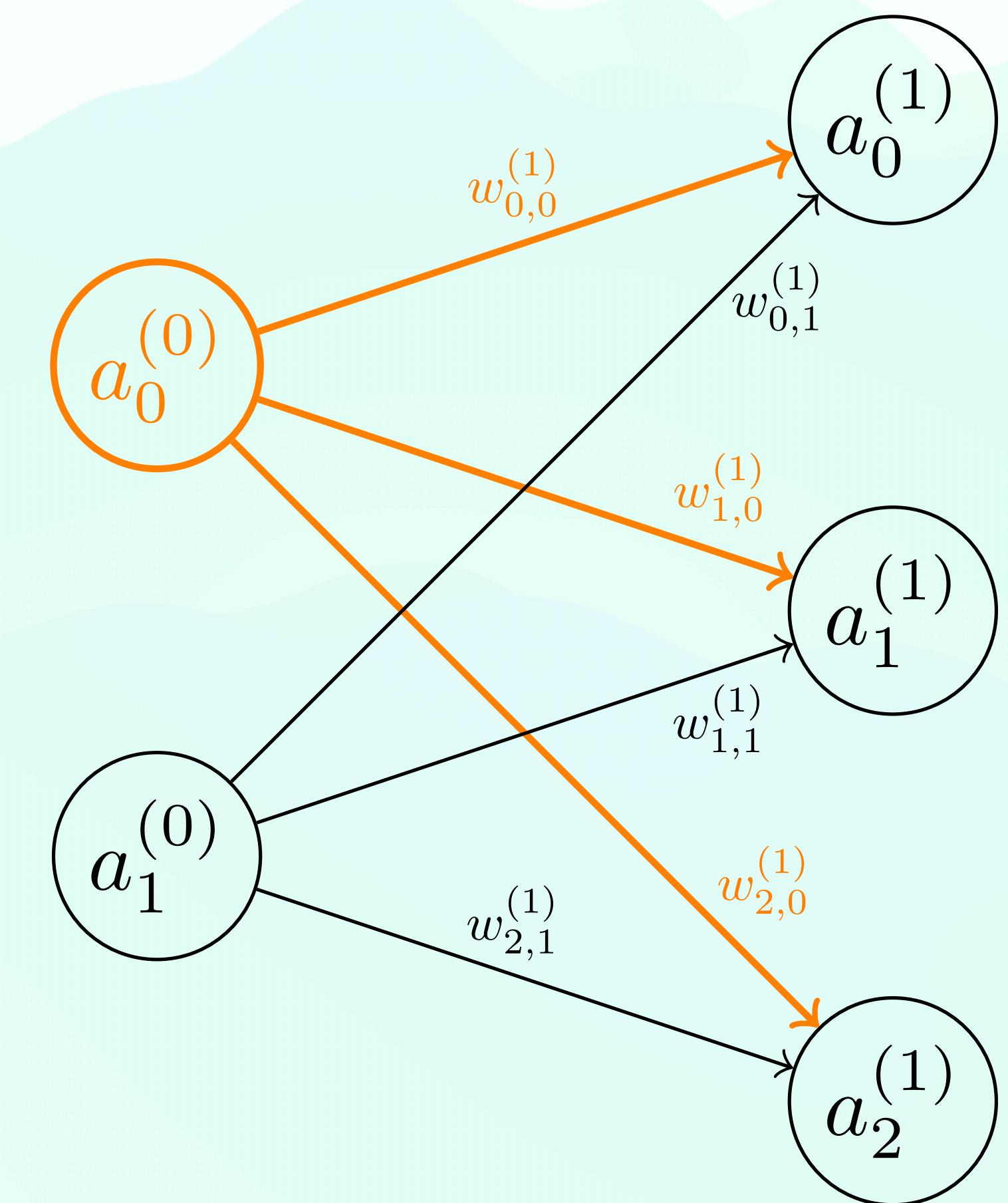


# Ableitung der Kostenfunktion

$$\frac{\partial C}{\partial a_j^{(l)}} = \sum_{i=0}^{n^{(l+1)}-1} \left( \frac{\partial C}{\partial a_i^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}} \frac{\partial z_i^{(l+1)}}{\partial a_j^{(l)}} \right)$$

Der Einfluss der einzelnen **Neuronenaktivierung** auf die **gewichtete Summe** ist auch hier genau so groß, wie das **Gewicht** zwischen beiden betroffenen Neuronen:  $w_{ij}^{(l+1)}$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$



# Ableitung der Kostenfunktion

$$\frac{\partial C}{\partial a_j^{(l)}} = \sum_{i=0}^{n^{(l+1)}-1} \left( \frac{\partial C}{\partial a_i^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}} \frac{\partial z_i^{(l+1)}}{\partial a_j^{(l)}} \right)$$

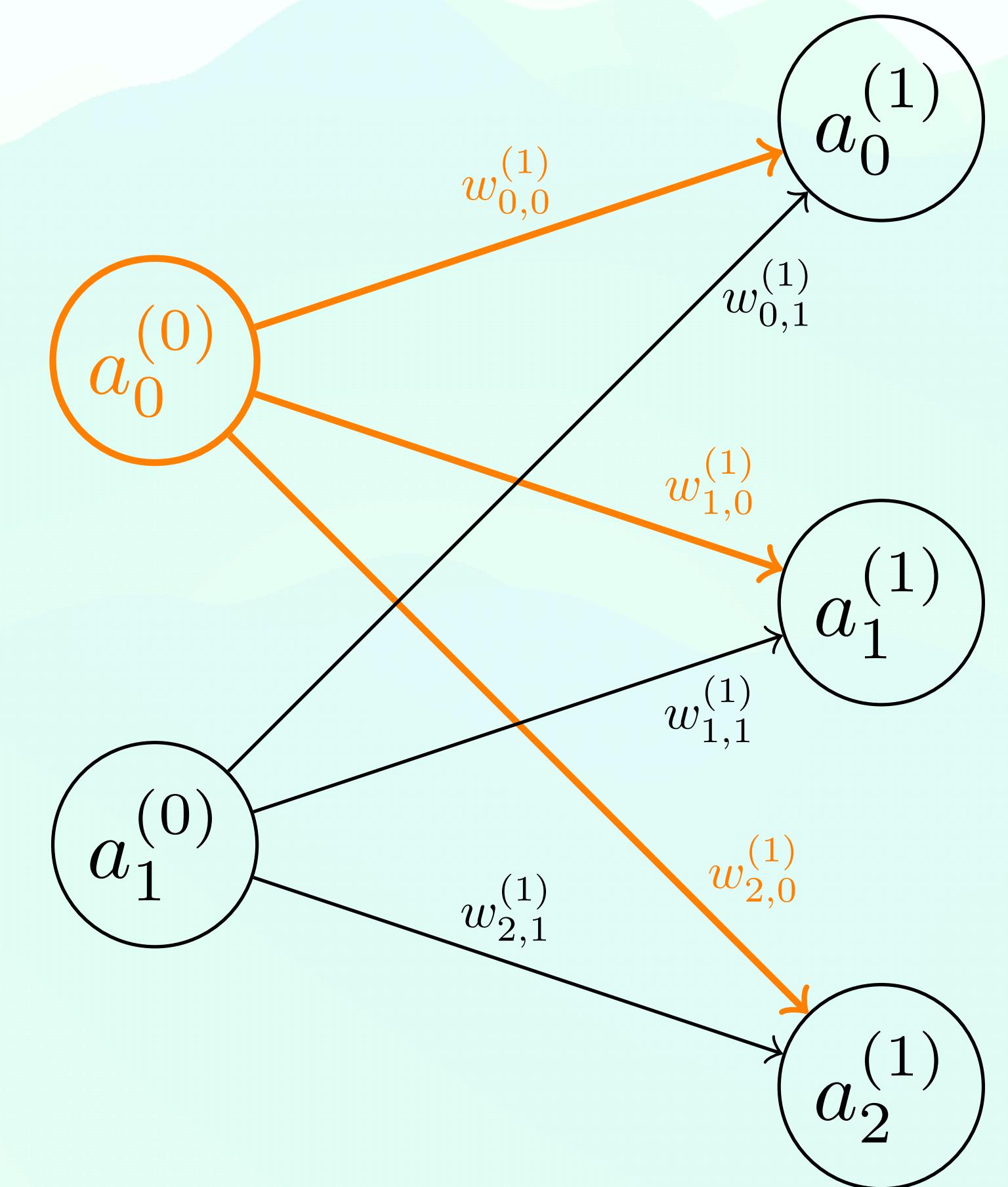
Der Einfluss der einzelnen **Neuronenaktivierung** auf die **gewichtete Summe** ist auch hier genau so groß, wie das **Gewicht** zwischen beiden betroffenen Neuronen:  $w_{ij}^{(l+1)}$

Daher können wir vereinfachen:

$$\frac{\partial C}{\partial a_j^{(l)}} = \sum_{i=0}^{n^{(l+1)}-1} \left( \frac{\partial C}{\partial a_i^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}} w_{ij}^{(l+1)} \right)$$

5

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$

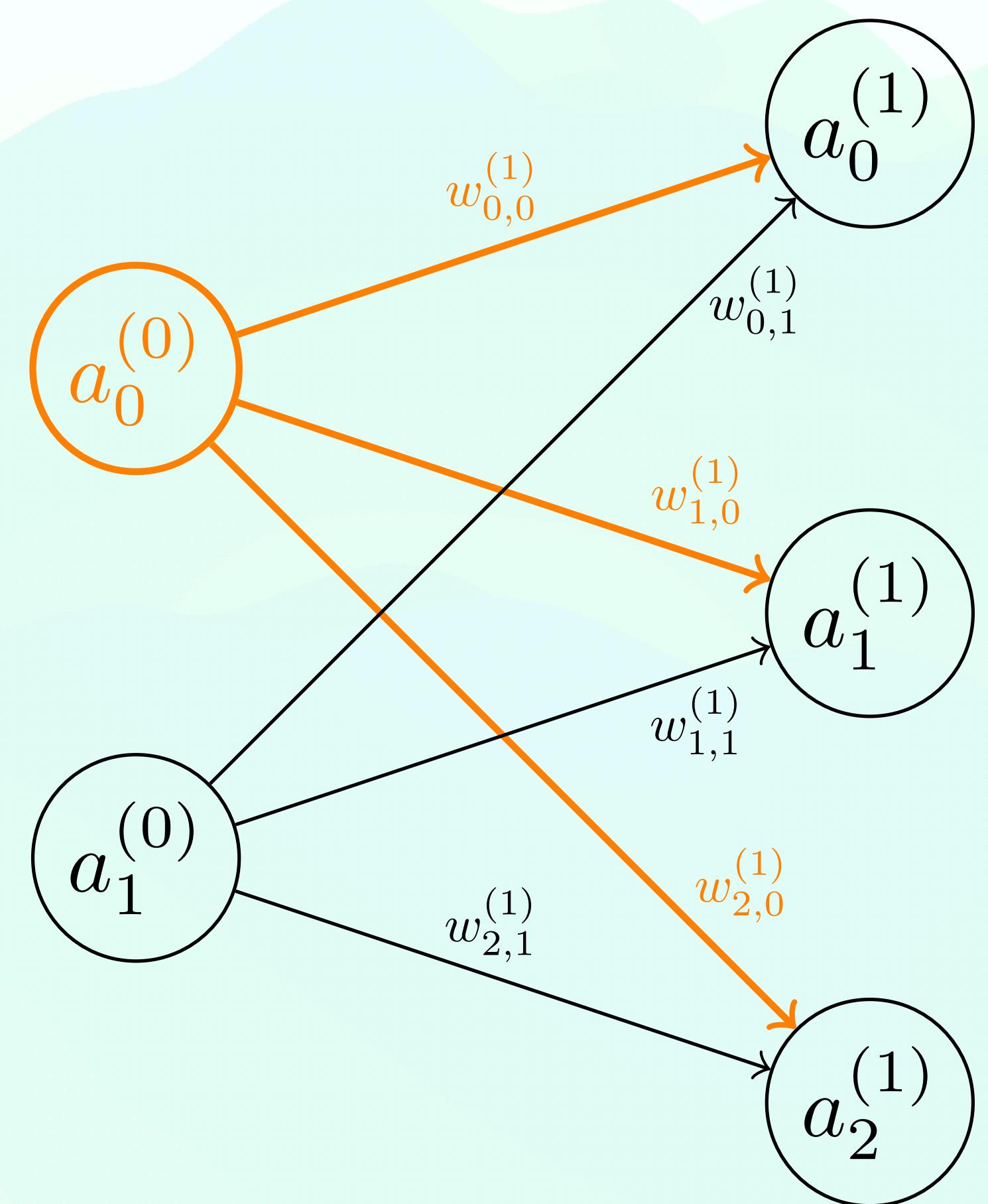


# Ableitung der Kostenfunktion

$$\frac{\partial C}{\partial a_j^{(l)}} = \sum_{i=0}^{n^{(l+1)}-1} \left( \frac{\partial C}{\partial a_i^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}} w_{ij}^{(l+1)} \right)$$

5

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$



# Ableitung der Kostenfunktion

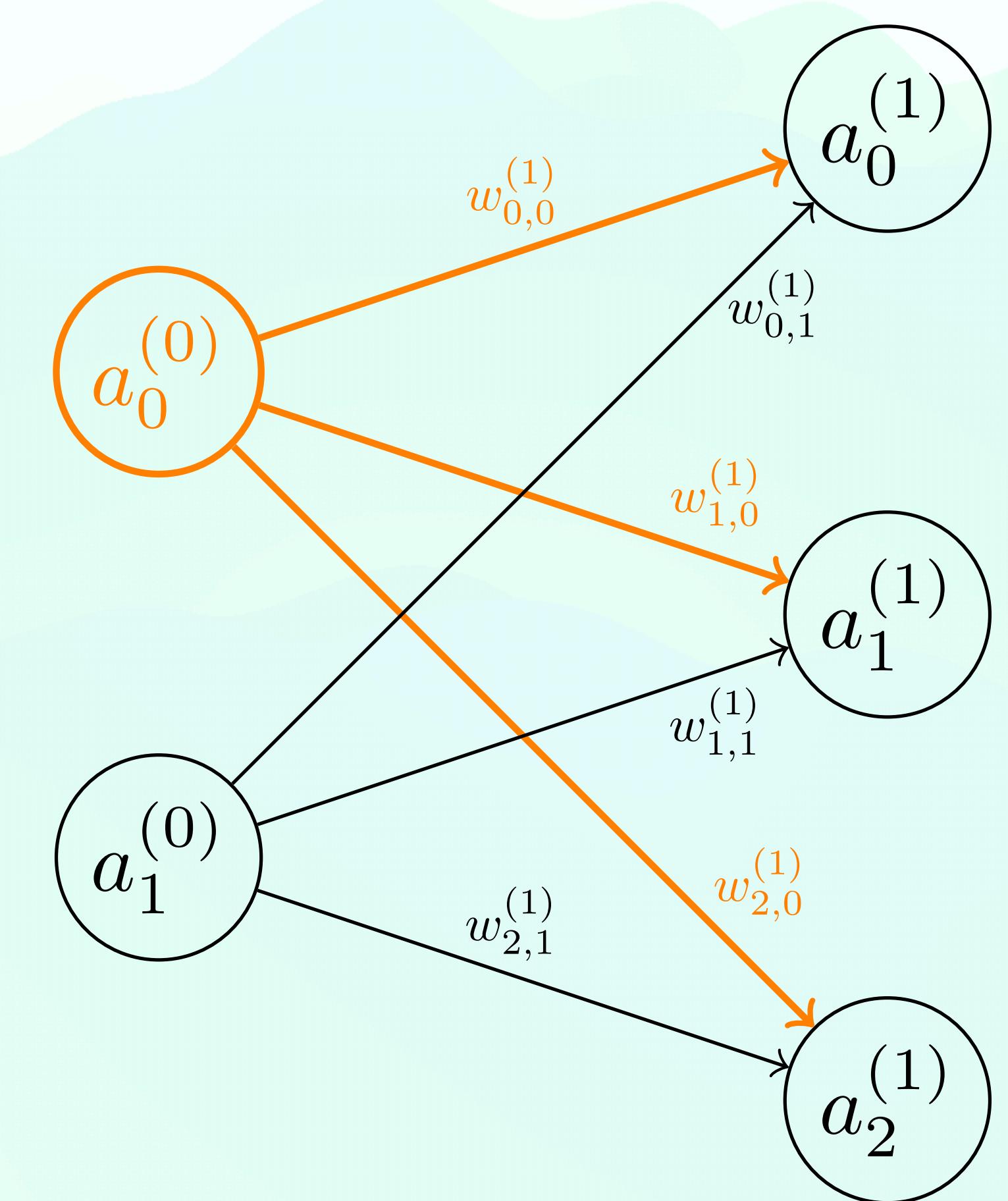
$$\frac{\partial C}{\partial a_j^{(l)}} = \sum_{i=0}^{n^{(l+1)}-1} \left( \frac{\partial C}{\partial a_i^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}} w_{ij}^{(l+1)} \right)$$

5

Der erhaltene Ausdruck ist **rekursiv**:

Um den Einfluss einer Aktivierung im **weiter innen liegenden Layer  $l$**  auf die Kosten zu ermitteln, muss zuerst der Einfluss der Aktivierungen des **nachfolgenden Layers  $l + 1$**  bekannt sein.

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$



# Ableitung der Kostenfunktion

$$\frac{\partial C}{\partial a_j^{(l)}} = \sum_{i=0}^{n^{(l+1)}-1} \left( \frac{\partial C}{\partial a_i^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}} w_{ij}^{(l+1)} \right)$$

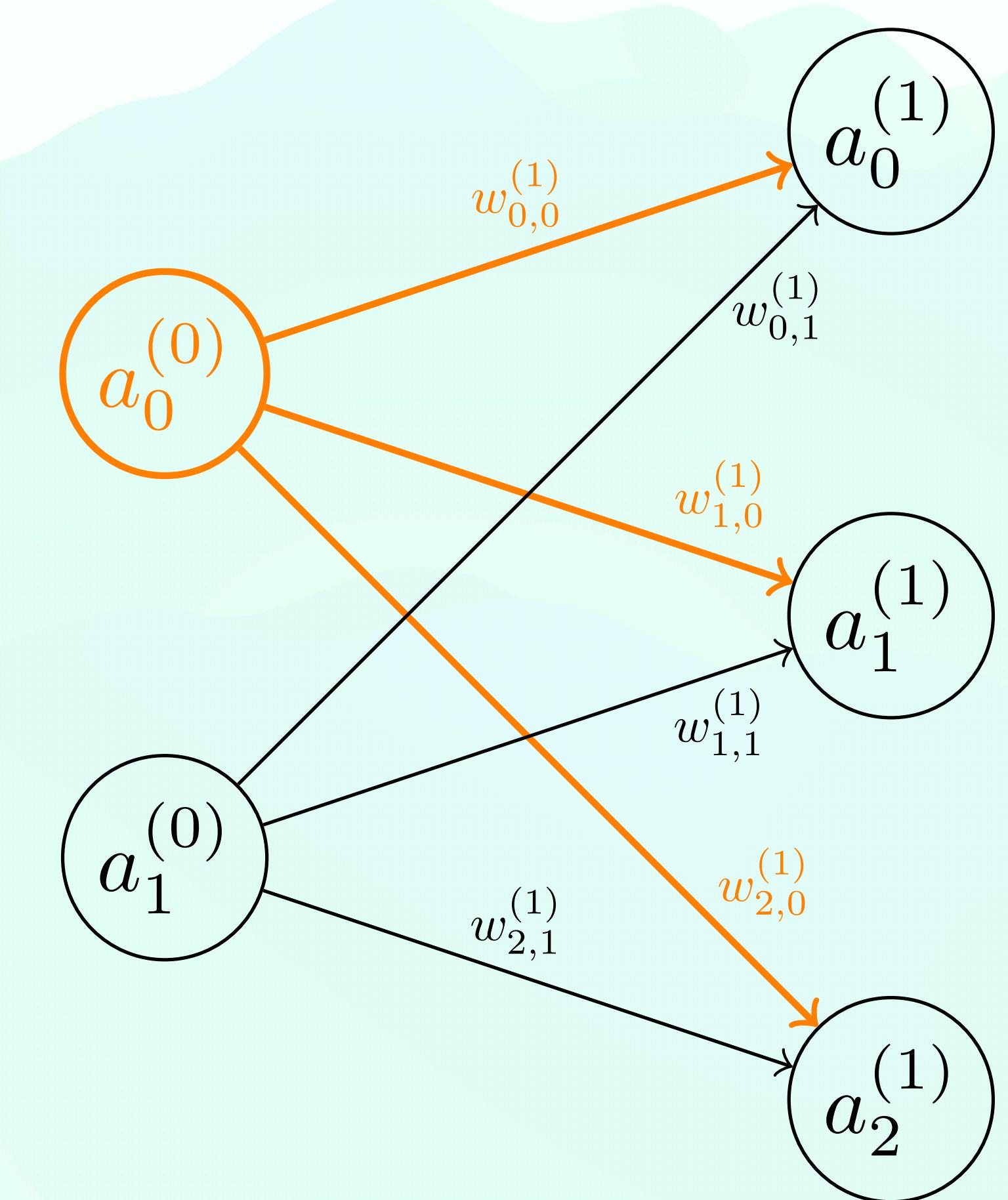
5

Der erhaltene Ausdruck ist **rekursiv**:

Um den Einfluss einer Aktivierung im **weiter innen liegenden Layer  $l$**  auf die Kosten zu ermitteln, muss zuerst der Einfluss der Aktivierungen des **nachfolgenden Layers  $l + 1$**  bekannt sein.

Wenn  $l + 1$  der Output-Layer ist, ist das der Rekursionsabbruch und es greift Gleichung 4.

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$



# Ableitung der Kostenfunktion

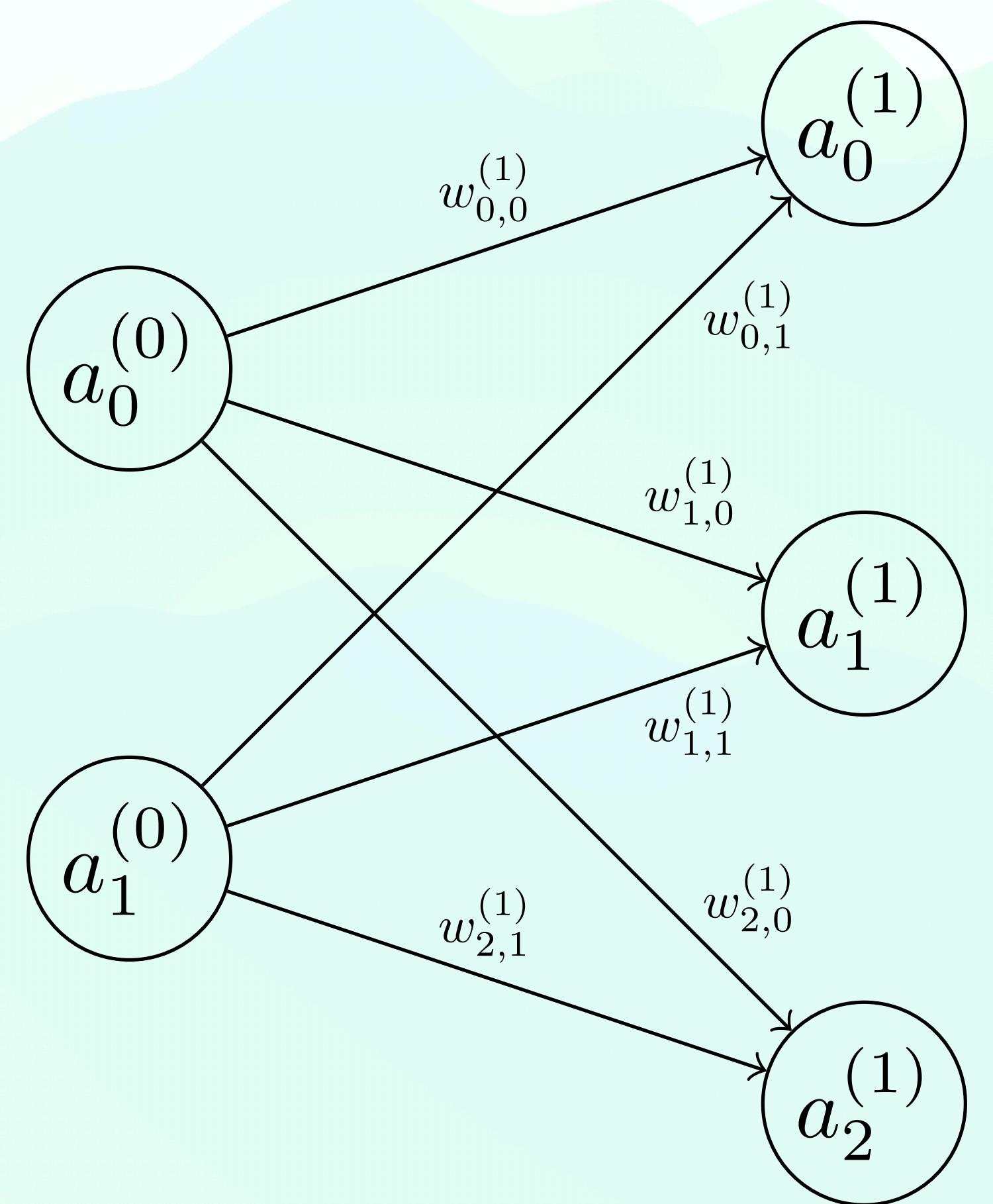
- Zur Erinnerung:

$$\frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \frac{\partial w_{jk}^{(l)} a_k^{(l-1)}}{\partial w_{jk}^{(l)}} = a_k^{(l-1)}$$

2

1

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$



# Ableitung der Kostenfunktion

- Zur Erinnerung:

$$\frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \frac{\partial w_{jk}^{(l)} a_k^{(l-1)}}{\partial w_{jk}^{(l)}} = a_k^{(l-1)}$$

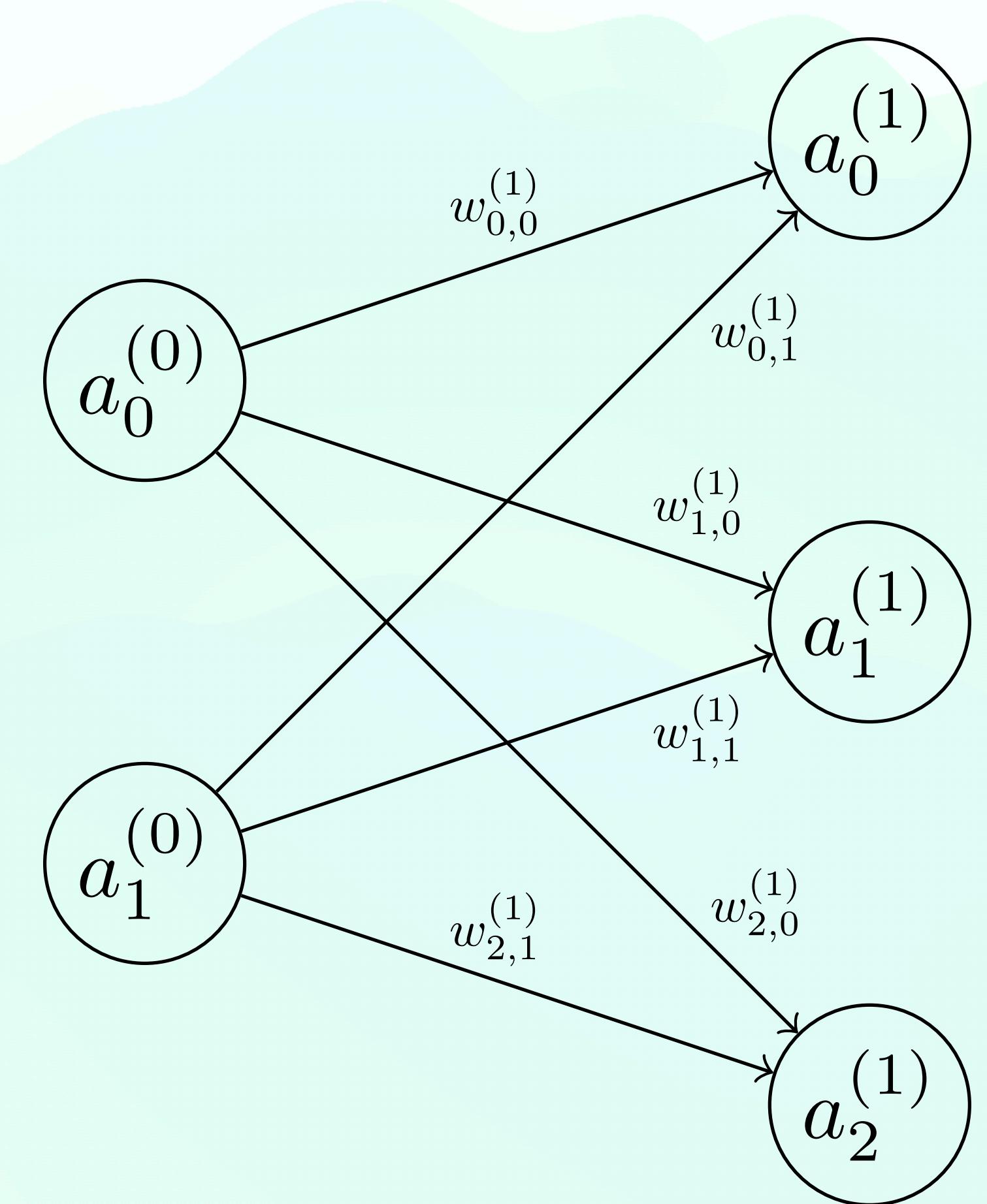
2

- Einsetzen von 2 in 1 ergibt:

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \cdot a_k^{(l-1)}$$

1

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$



# Ableitung der Kostenfunktion

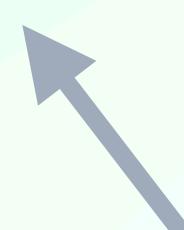
- Zur Erinnerung:

$$\frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \frac{\partial w_{jk}^{(l)} a_k^{(l-1)}}{\partial w_{jk}^{(l)}} = a_k^{(l-1)}$$

2

- Einsetzen von 2 in 1 ergibt:

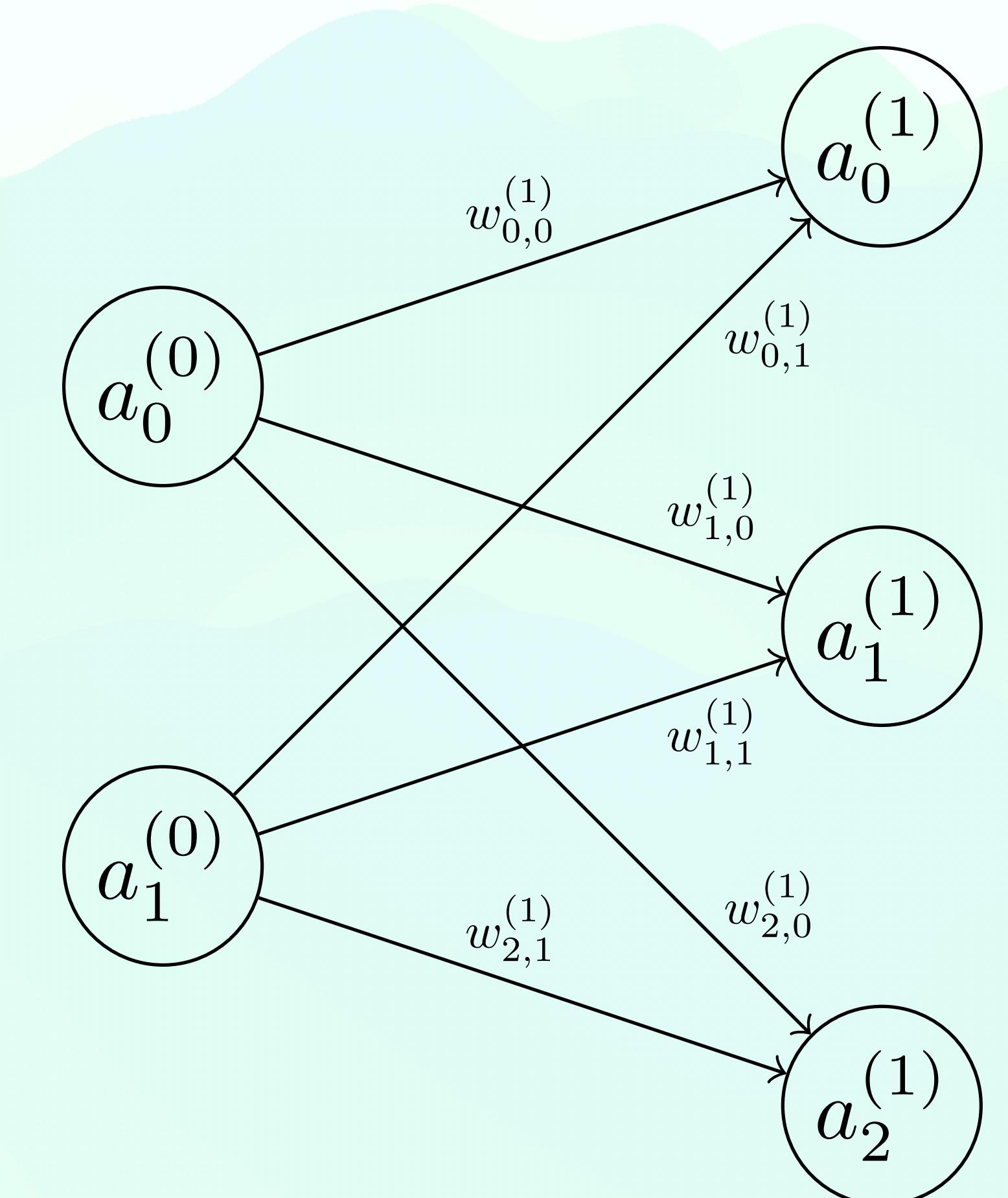
$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \cdot a_k^{(l-1)}$$



Zur Erinnerung: Das ist unsere gesuchte Komponente  
des Gradienten-Vektors  $\nabla C$  für das Gewicht  $w_{jk}^{(l)}$

1

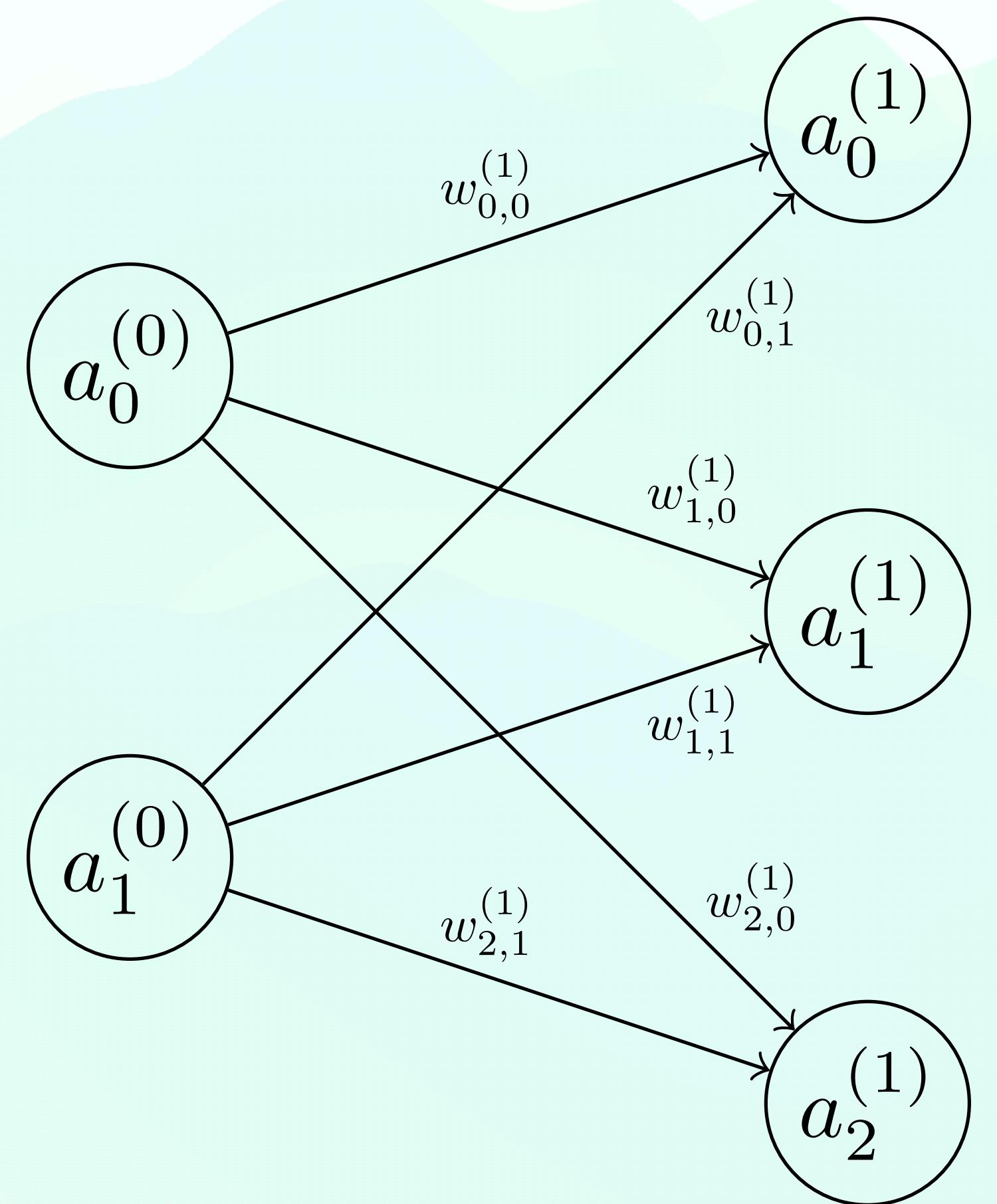
$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$



# Ableitung der Kostenfunktion

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \cdot a_k^{(l-1)}$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$

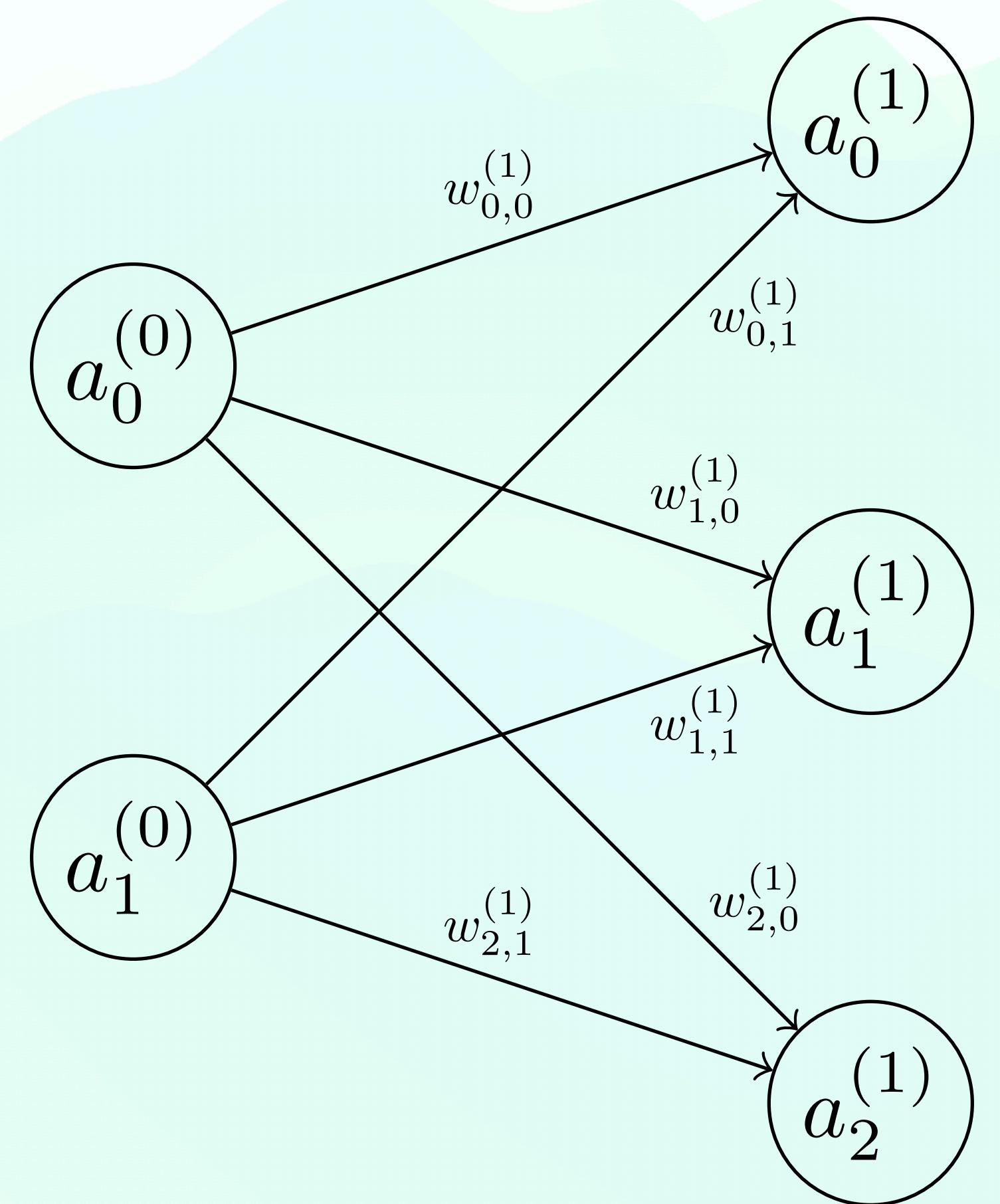


# Ableitung der Kostenfunktion

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \cdot a_k^{(l-1)}$$

- Wir definieren:  $\delta_j^{(l)} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}}$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$



# Ableitung der Kostenfunktion

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \cdot a_k^{(l-1)}$$

- Wir definieren:

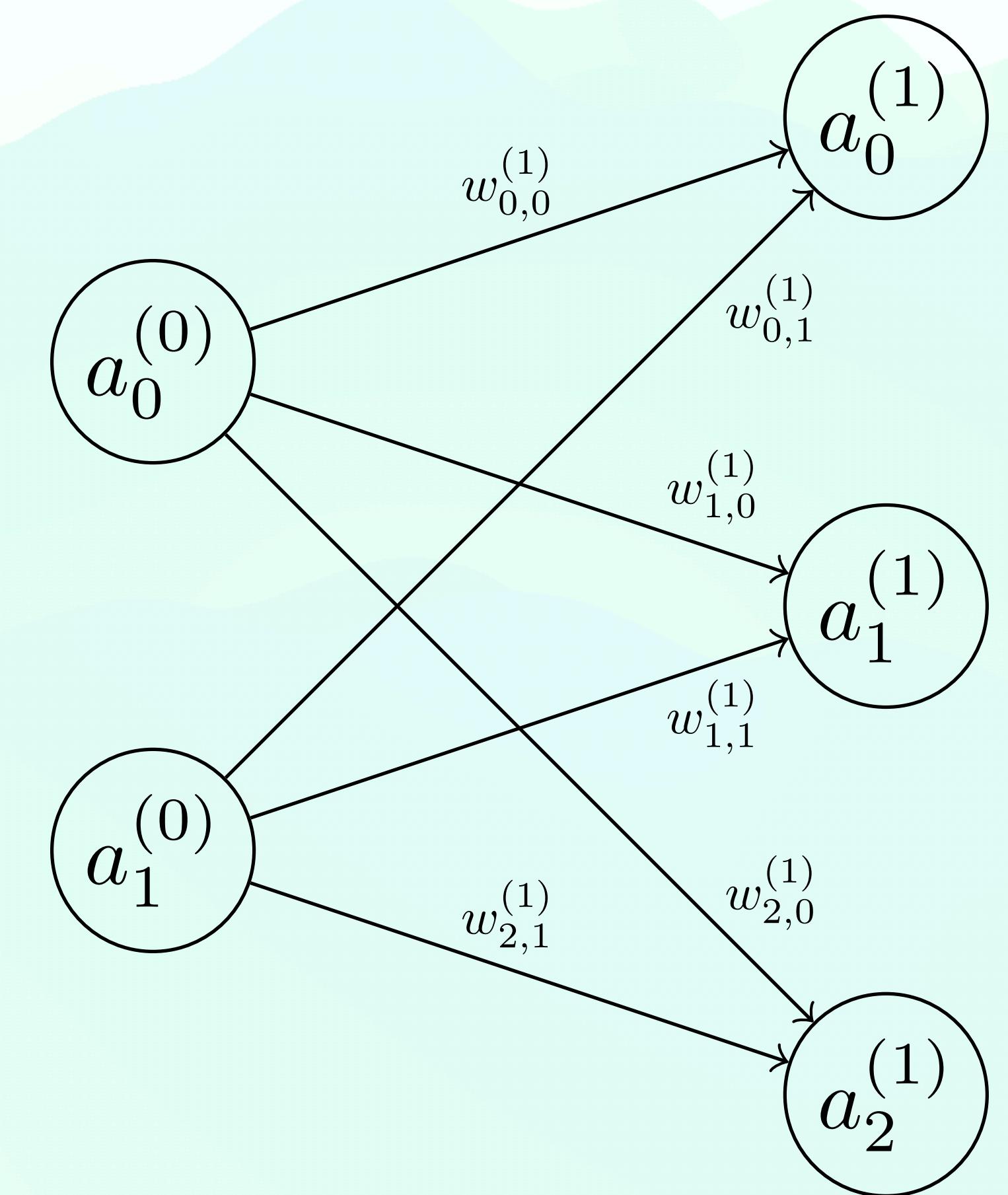
$$\delta_j^{(l)} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}}$$

- Und damit:

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \cdot a_k^{(l-1)} = \delta_j^{(l)} \cdot a_k^{(l-1)}$$

6

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$



# Ableitung der Kostenfunktion

$$\delta_j^{(l)} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}}$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$

# Ableitung der Kostenfunktion

$$\delta_j^{(l)} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}}$$

- Durch Einsetzen von 3, 4 und 5 erhalten wir:

$$\delta_j^{(l)} = \begin{cases} 2 \left( a_j^{(l)} - y_j \right) \cdot \sigma^{(l)'}(z_j^{(l)}) & \text{für den Output-Layer} \\ \left( \sum_{i=0}^{n^{(l+1)}-1} \frac{\partial C}{\partial a_i^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}} w_{ij}^{(l+1)} \right) \cdot \sigma^{(l)'}(z_j^{(l)}) & \text{für innere Layer} \end{cases}$$

# Ableitung der Kostenfunktion

$$\delta_j^{(l)} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}}$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$

- Durch Einsetzen von 3, 4 und 5 erhalten wir:

$$\delta_j^{(l)} = \begin{cases} 2 \left( a_j^{(l)} - y_j \right) \cdot \sigma^{(l)'}(z_j^{(l)}) & \text{für den Output-Layer} \\ \left( \sum_{i=0}^{n^{(l+1)}-1} \frac{\partial C}{\partial a_i^{(l+1)}} \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}} w_{ij}^{(l+1)} \right) \cdot \sigma^{(l)'}(z_j^{(l)}) & \text{für innere Layer} \end{cases}$$

Hier sehen wir die **Rekursion** und die Ähnlichkeit zu  $\delta_j^{(l)}$ :

Dieser Ausdruck entspricht  $\delta_i^{(l+1)}$

# Ableitung der Kostenfunktion

$$\delta_j^{(l)} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}}$$

$$\delta_j^{(l)} = \begin{cases} 2 \left( a_j^{(l)} - y_j \right) \cdot \sigma^{(l)'}(z_j^{(l)}) & \text{für den Output-Layer} \\ \left( \sum_{i=0}^{n^{(l+1)}-1} \delta_j^{(l+1)} w_{ij}^{(l+1)} \right) \cdot \sigma^{(l)'}(z_j^{(l)}) & \text{für innere Layer} \end{cases}$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$

7

# Ableitung der Kostenfunktion

$$\delta_j^{(l)} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}}$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$

Wir sehen also, dass die partielle Ableitung im Layer  $l$  von der partiellen Ableitung im nächsten Layer  $l + 1$  abhängt. Wir fangen im Output-Layer an und arbeiten uns von hinten nach vorne: **Backpropagation**

$$\delta_j^{(l)} = \begin{cases} 2 \left( a_j^{(l)} - y_j \right) \cdot \sigma^{(l)'}(z_j^{(l)}) & \text{für den Output-Layer} \\ \left( \sum_{i=0}^{n^{(l+1)}-1} \delta_j^{(l+1)} w_{ij}^{(l+1)} \right) \cdot \sigma^{(l)'}(z_j^{(l)}) & \text{für innere Layer} \end{cases}$$

7

# Ableitung der Kostenfunktion

$$\delta_j^{(l)} = \begin{cases} 2 \left( a_j^{(l)} - y_j \right) \cdot \sigma^{(l)'}(z_j^{(l)}) & \text{für den Output-Layer} \\ \left( \sum_{i=0}^{n^{(l+1)}-1} \delta_j^{(l+1)} w_{ij}^{(l+1)} \right) \cdot \sigma^{(l)'}(z_j^{(l)}) & \text{für innere Layer} \end{cases}$$

7

# Ableitung der Kostenfunktion

$$\delta_j^{(l)} = \begin{cases} 2 \left( a_j^{(l)} - y_j \right) \cdot \sigma^{(l)'}(z_j^{(l)}) & \text{für den Output-Layer} \\ \left( \sum_{i=0}^{n^{(l+1)}-1} \delta_j^{(l+1)} w_{ij}^{(l+1)} \right) \cdot \sigma^{(l)'}(z_j^{(l)}) & \text{für innere Layer} \end{cases}$$

- Laut Gleichung 6 gilt zur Berechnung der Komponente des Gradienten-Vektors:

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} \cdot a_k^{(l-1)}$$

7

# Ableitung der Kostenfunktion

$$\delta_j^{(l)} = \begin{cases} 2 \left( a_j^{(l)} - y_j \right) \cdot \sigma^{(l)'}(z_j^{(l)}) & \text{für den Output-Layer} \\ \left( \sum_{i=0}^{n^{(l+1)}-1} \delta_j^{(l+1)} w_{ij}^{(l+1)} \right) \cdot \sigma^{(l)'}(z_j^{(l)}) & \text{für innere Layer} \end{cases}$$

7

- Laut Gleichung 6 gilt zur Berechnung der Komponente des Gradienten-Vektors:

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} \cdot a_k^{(l-1)}$$

- Wir haben also nun alles, was nötig ist, um für jedes Gewicht in unserem neuronalen Netz den Anteil an der Gradientenrichtung zur Verringerung der Kosten zu bestimmen!

# Anpassung der Gewichte

- Der Betrag, um den wir das Gewicht  $w_{jk}^{(l)}$  anpassen müssen, berechnet sich wie folgt:

$$\Delta w_{jk}^{(l)} = -\eta \cdot \frac{\partial C}{\partial w_{jk}^{(l)}} = -\eta \cdot \delta_j^{(l)} \cdot a_k^{(l-1)}$$

8

# Anpassung der Gewichte

- Der Betrag, um den wir das Gewicht  $w_{jk}^{(l)}$  anpassen müssen, berechnet sich wie folgt:

$$\Delta w_{jk}^{(l)} = -\eta \cdot \frac{\partial C}{\partial w_{jk}^{(l)}} = -\eta \cdot \delta_j^{(l)} \cdot a_k^{(l-1)}$$

8

Der Gradient würde uns die Richtung des steilsten **Anstiegs** anzeigen, deswegen drehen wir das Vorzeichen um: Uns interessiert der **Gradientenabstieg**.

Wir wollen die Kostenfunktion **minimieren**.

# Anpassung der Gewichte

- Der Betrag, um den wir das Gewicht  $w_{jk}^{(l)}$  anpassen müssen, berechnet sich wie folgt:

$$\Delta w_{jk}^{(l)} = -\eta \cdot \frac{\partial C}{\partial w_{jk}^{(l)}} = -\eta \cdot \delta_j^{(l)} \cdot a_k^{(l-1)}$$

8

Die Lernrate  $\eta$  (gesprochen: „Eta“) gibt an, wie groß oder klein die Gewichtsanpassungsschritte sein sollen, also wie schnell oder langsam unser Netz lernen soll.

# Anpassung der Gewichte

- Der Betrag, um den wir das Gewicht  $w_{jk}^{(l)}$  anpassen müssen, berechnet sich wie folgt:

$$\Delta w_{jk}^{(l)} = -\eta \cdot \frac{\partial C}{\partial w_{jk}^{(l)}} = -\eta \cdot \delta_j^{(l)} \cdot a_k^{(l-1)}$$

8

Die Lernrate  $\eta$  (gesprochen: „Eta“) gibt an, wie groß oder klein die Gewichtsanpassungsschritte sein sollen, also wie schnell oder langsam unser Netz lernen soll.

Eine zu hohe Lernrate könnte die Anpassung übers Ziel hinaus schießen lassen und das lokale Minimum der Kostenfunktion verfehlten.

# Anpassung der Gewichte

- Der Betrag, um den wir das Gewicht  $w_{jk}^{(l)}$  anpassen müssen, berechnet sich wie folgt:

$$\Delta w_{jk}^{(l)} = -\eta \cdot \frac{\partial C}{\partial w_{jk}^{(l)}} = -\eta \cdot \delta_j^{(l)} \cdot a_k^{(l-1)}$$

8

Der Einfluss, den eine kleine Änderung unseres Gewichts auf die Kosten hätte – also die Richtung, in die sich die Kostenfunktion bei einer kleinen Änderung des Gewichts entwickeln würde.

# Anpassung der Gewichte

- Der Betrag, um den wir das Gewicht  $w_{jk}^{(l)}$  anpassen müssen, berechnet sich wie folgt:

$$\Delta w_{jk}^{(l)} = -\eta \cdot \frac{\partial C}{\partial w_{jk}^{(l)}} = -\eta \cdot \delta_j^{(l)} \cdot a_k^{(l-1)}$$

8

Dieses Delta muss auf den bisherigen Wert des Gewichts addiert werden, um die Kosten zu senken.

# Anpassung der Gewichte

- Der Betrag, um den wir das Gewicht  $w_{jk}^{(l)}$  anpassen müssen, berechnet sich wie folgt:

$$\Delta w_{jk}^{(l)} = -\eta \cdot \frac{\partial C}{\partial w_{jk}^{(l)}} = -\eta \cdot \delta_j^{(l)} \cdot a_k^{(l-1)}$$

8

Dieses Delta muss auf den bisherigen Wert des Gewichts addiert werden, um die Kosten zu senken.

Aber: Jedes Trainingsbeispiel erzeugt ein eigenes  $\Delta w_{jk}^{(l)}$ . Wir bilden den **Mittelwert aus allen** und addieren diesen auf das Gewicht.

# Anpassung der Gewichte

- Der Betrag, um den wir das Gewicht  $w_{jk}^{(l)}$  anpassen müssen, berechnet sich wie folgt:

$$\Delta w_{jk}^{(l)} = -\eta \cdot \frac{\partial C}{\partial w_{jk}^{(l)}} = -\eta \cdot \delta_j^{(l)} \cdot a_k^{(l-1)}$$

8

Dieses Delta muss auf den bisherigen Wert des Gewichts addiert werden, um die Kosten zu senken.

Aber: Jedes Trainingsbeispiel erzeugt ein eigenes  $\Delta w_{jk}^{(l)}$ . Wir bilden den **Mittelwert aus allen** und addieren diesen auf das Gewicht.

Und das natürlich für alle Gewichte im neuronalen Netz ...

# Backpropagation-Algorithmus



```
def __init__(  
    self,  
    structure,  
    n_iterations=1000,  
    output_activation_func=relu,  
    hidden_activation_func=sigmoid,  
):  
    self.structure = structure  
    self.n_iterations = n_iterations  
    self.output_activation_func = output_activation_func  
    self.hidden_activation_func = hidden_activation_func  
    self.__init_layers()  
    self.__init_weights()
```

Wir erweitern zunächst unseren Konstruktor um einen Parameter `n_iterations` für die Anzahl der Iterationen, die während des Trainings durchgeführt werden sollen. Als Default-Wert legen wir `1000` fest.

# Backpropagation-Algorithmus



```
def train(self, training_data):
    for _ in range(self.n_iterations):
        for X, Y in training_data:
            if len(Y) != len(self.activations[-1]):
                raise Exception("Falsche Anzahl an erwarteten Output-Werten übergeben")

            prediction = self.predict(X)
            diff = prediction - Y
            output_layer_derivative = 2 * diff
```

Dann deklarieren wir neben `predict` eine zweite öffentliche Methode namens `train`, die im Parameter `training_data` einen Trainingsdatensatz entgegennimmt und mit ihm das Netz trainieren wird.

# Backpropagation-Algorithmus



```
def train(self, training_data):
    for _ in range(self.n_iterations):
        for X, Y in training_data:
            if len(Y) != len(self.activations[-1]):
                raise Exception("Falsche Anzahl an erwarteten Output-Werten übergeben")

            prediction = self.predict(X)
            diff = prediction - Y
            output_layer_derivative = 2 * diff
```

Der Trainingsdatensatz soll dabei eine Liste von **Tupeln** sein, die jeweils einen Satz an Inputs und die dazugehörigen erwarteten Outputs enthalten. Sowohl Inputs als auch Outputs sollen dabei als numpy-Spaltenvektoren gegeben sein.

# Backpropagation-Algorithmus



```
def train(self, training_data):
    for _ in range(self.n_iterations):
        for X, Y in training_data:
            if len(Y) != len(self.activations[-1]):
                raise Exception("Falsche Anzahl an erwarteten Output-Werten übergeben")

            prediction = self.predict(X)
            diff = prediction - Y
            output_layer_derivative = 2 * diff
```

Zunächst soll eine `for`-Schleife dafür sorgen, dass alle folgenden Trainings-schritte entsprechend der übergebenen Zahl an Iterationen wiederholt werden.

# Backpropagation-Algorithmus



```
def train(self, training_data):
    for _ in range(self.n_iterations):
        for X, Y in training_data:
            if len(Y) != len(self.activations[-1]):
                raise Exception("Falsche Anzahl an erwarteten Output-Werten übergeben")

            prediction = self.predict(X)
            diff = prediction - Y
            output_layer_derivative = 2 * diff
```

Eine weitere `for`-Schleife iteriert durch jedes einzelne Tupel unserer `training_data`, wobei jeweils der aktuelle Input-Satz in `X` und der erwartete Output-Satz in `Y` abgelegt wird.

# Backpropagation-Algorithmus



```
def train(self, training_data):
    for _ in range(self.n_iterations):
        for X, Y in training_data:
            if len(Y) != len(self.activations[-1]):
                raise Exception("Falsche Anzahl an erwarteten Output-Werten übergeben")
```

```
prediction = self.predict(X)
diff = prediction - Y
output_layer_derivative = 2 * diff
```

Wenn die Anzahl der erwarteten Outputs im aktuellen Trainingsbeispiel nicht mit der Anzahl der Neuronen im Output-Layer übereinstimmt, erzeugen wir wieder eine Exception.

# Backpropagation-Algorithmus



```
def train(self, training_data):
    for _ in range(self.n_iterations):
        for X, Y in training_data:
            if len(Y) != len(self.activations[-1]):
                raise Exception("Falsche Anzahl an erwarteten Output-Werten übergeben")
```

```
prediction = self.predict(X)
diff = prediction - Y
output_layer_derivative = 2 * diff
```

Dann erzeugen wir eine Vorhersage für die aktuellen Inputs mithilfe der `predict`-Methode, die wir zuvor implementiert haben.

# Backpropagation-Algorithmus



```
def train(self, training_data):
    for _ in range(self.n_iterations):
        for X, Y in training_data:
            if len(Y) != len(self.activations[-1]):
                raise Exception("Falsche Anzahl an erwarteten Output-Werten übergeben")

            prediction = self.predict(X)
            diff = prediction - Y
            output_layer_derivative = 2 * diff
```

Die Abweichung zwischen vorhergesagtem und erwartetem Output legen wir in der Variable `diff` ab. Sowohl `prediction` als auch `Y` sind in diesem Fall Vektoren, trotzdem können wir ganz einfach den Minus-Operator `-` verwenden.

# Backpropagation-Algorithmus

$$\frac{\partial C}{\partial a_j^{(l)}} = 2 \left( a_j^{(l)} - y_j \right)$$

4



```
def train(self, training_data):
    for _ in range(self.n_iterations):
        for X, Y in training_data:
            if len(Y) != len(self.activations[-1]):
                raise Exception("Falsche Anzahl an erwarteten Output-Werten übergeben")

            prediction = self.predict(X)
            diff = prediction - Y
            output_layer_derivative = 2 * diff
```

Nach Gleichung 4 ist die Ableitung der Kosten nach den Aktivierungen im Output-Layer gleich dem Doppelten dieser Differenz. Wir legen sie zur späteren Verwendung in einer Variable ab.

# Backpropagation-Algorithmus



```
def train(self, training_data):
    partial_deltas = []
    for l in range(len(self.activations)):
        partial_deltas.append(numpy.zeros((len(self.weighted_sums[l]), 1)))

    delta_W = []
    for l in range(len(self.activations)):
        if l == 0:
            delta_W.append([])
        else:
            delta_W.append(numpy.zeros((
                len(self.weighted_sums[l]),
                len(self.activations[l-1]),
            )))

    for _ in range(self.n_iterations):
        # ...
```

Den folgenden Block müssen wir noch vor unserer Iterations-Schleife einfügen. In ihm werden noch einige für die Backpropagation benötigte Variablen angelegt.

# Backpropagation-Algorithmus



```
def train(self, training_data):
    partial_deltas = []
    for l in range(len(self.activations)):
        partial_deltas.append(numpy.zeros((len(self.weighted_sums[l]), 1)))

    delta_W = []
    for l in range(len(self.activations)):
        if l == 0:
            delta_W.append([])
        else:
            delta_W.append(numpy.zeros((
                len(self.weighted_sums[l]),
                len(self.activations[l-1]),
            )))

    for _ in range(self.n_iterations):
        # ...
```

Zunächst deklarieren wir eine Liste für die partiellen Ableitungen  $\delta_j$  der einzelnen Layer.

# Backpropagation-Algorithmus



```
def train(self, training_data):
    partial_deltas = []
    for l in range(len(self.activations)):
        partial_deltas.append(numpy.zeros((len(self.weighted_sums[l]), 1)))

    delta_W = []
    for l in range(len(self.activations)):
        if l == 0:
            delta_W.append([])
        else:
            delta_W.append(numpy.zeros((
                len(self.weighted_sums[l]),
                len(self.activations[l-1]),
            )))
    for _ in range(self.n_iterations):
        # ...
```

Mit einer Schleife iterieren wir durch alle Layer und fügen für jeden einen mit Nullen vorbefüllten Vektor hinzu, der genauso viele Elemente enthält, wie der jeweilige Layer Neuronen hat — abzüglich des Bias-Neurons, das sich nicht auf vorherige Layer auswirken soll. Daher benutzen wir die Anzahl der gewichteten Summen im Layer  $l$ , nicht die Anzahl der Aktivierungen.

# Backpropagation-Algorithmus



```
def train(self, training_data):
    partial_deltas = []
    for l in range(len(self.activations)):
        partial_deltas.append(numpy.zeros((len(self.weighted_sums[l]), 1)))

    delta_W = []
    for l in range(len(self.activations)):
        if l == 0:
            delta_W.append([])
        else:
            delta_W.append(numpy.zeros((
                len(self.weighted_sums[l]),
                len(self.activations[l-1]),
            )))
    for _ in range(self.n_iterations):
        # ...
```

Auch für die errechneten Gewichtsanpassungen  $\Delta w$  legen wir eine Liste an. Jedes Element der Liste soll am Ende für einen Layer stehen und ist analog zu den Gewichten selbst eine Matrix.

# Backpropagation-Algorithmus



```
def train(self, training_data):
    partial_deltas = []
    for l in range(len(self.activations)):
        partial_deltas.append(numpy.zeros((len(self.weighted_sums[l]), 1)))

    delta_W = []
    for l in range(len(self.activations)):
        if l == 0:
            delta_W.append([])
        else:
            delta_W.append(numpy.zeros((
                len(self.weighted_sums[l]),
                len(self.activations[l-1]),
            )))
    for _ in range(self.n_iterations):
        # ...
```

Dementsprechend iterieren wir wie gehabt durch die Layer.

# Backpropagation-Algorithmus



```
def train(self, training_data):
    partial_deltas = []
    for l in range(len(self.activations)):
        partial_deltas.append(numpy.zeros((len(self.weighted_sums[l]), 1)))

    delta_W = []
    for l in range(len(self.activations)):
        if l == 0:
            delta_W.append([])
        else:
            delta_W.append(numpy.zeros((
                len(self.weighted_sums[l]),
                len(self.activations[l-1]),
            )))
    for _ in range(self.n_iterations):
        # ...
```

Wie bei den Gewichtsmatrizen auch legen wir für den Layer-Index 0 ein später nicht verwendetes Element in der Liste ab. Damit sorgen wir für konsistente Indizes, indem z. B. die  $\Delta w$  vom Input-Layer zum ersten Hidden Layer den Index 1 bekommen.

# Backpropagation-Algorithmus



```
def train(self, training_data):
    partial_deltas = []
    for l in range(len(self.activations)):
        partial_deltas.append(numpy.zeros((len(self.weighted_sums[l]), 1)))

    delta_W = []
    for l in range(len(self.activations)):
        if l == 0:
            delta_W.append([])
        else:
            delta_W.append(numpy.zeros((
                len(self.weighted_sums[l]),
                len(self.activations[l-1]),
            )))
    for _ in range(self.n_iterations):
        # ...
```

Für alle anderen Layer legen wir Matrizen an, die dieselben Dimensionen wie die entsprechenden Gewichtsmatrizen haben. `numpy.zeros` sorgt wie gewohnt dafür, dass alle Elemente der Matrizen mit 0 vorbelegt sind.

# Backpropagation-Algorithmus



```
def __init__(  
    self,  
    structure,  
    n_iterations=1000,  
    eta=0.01,  
    output_activation_func=relu,  
    hidden_activation_func=sigmoid,  
):  
    self.structure = structure  
    self.n_iterations = n_iterations  
    self.eta = eta  
    self.output_activation_func = output_activation_func  
    self.hidden_activation_func = hidden_activation_func  
    self.__init_layers()  
    self.__init_weights()
```

Bevor wir die Backpropagation weiter implementieren können, müssen wir unserem neuronalen Netz noch einen letzten Parameter hinzufügen: die Lernrate  $\eta$ , genannt `eta`. Wir lassen sie uns im Konstruktor **optional** übergeben, geben ihr ansonsten den Default-Wert von `0.01`, und speichern sie in einer gleichnamigen Objekt-Eigenschaft.

# Backpropagation-Algorithmus



```
# ...
output_layer_derivative = 2 * diff

for l in range(len(self.activations))-1, 0, -1):
    if l == len(self.activations) - 1:
        partial_deltas[l] = output_layer_derivative * \
            self.output_activation_func(self.weighted_sums[l], derivative=True)
    else:
        partial_deltas[l] = \
            numpy.matmul(self.weights[l+1][:, 1:].T, partial_deltas[l+1]) * \
            self.hidden_activation_func(self.weighted_sums[l], derivative=True)

    delta_W[l] += -self.eta * numpy.matmul(partial_deltas[l], self.activations[l-1].T)
```

Jetzt springen wir wieder zurück in unsere Trainingsschleife und fügen den folgenden Block ein, der die eigentliche Backpropagation durchführt und dafür unsere gerade angelegten Variablen `partial_deltas` und `delta_W` verwendet.

# Backpropagation-Algorithmus



```
# ...
output_layer_derivative = 2 * diff

for l in range(len(self.activations)-1, 0, -1):
    if l == len(self.activations) - 1:
        partial_deltas[l] = output_layer_derivative * \
            self.output_activation_func(self.weighted_sums[l], derivative=True)
    else:
        partial_deltas[l] = \
            numpy.matmul(self.weights[l+1][:, 1:].T, partial_deltas[l+1]) * \
            self.hidden_activation_func(self.weighted_sums[l], derivative=True)

    delta_W[l] += -self.eta * numpy.matmul(partial_deltas[l], self.activations[l-1].T)
```

Da wir uns bei der Backpropagation von **hinten nach vorne** durch das Netz bewegen müssen, lassen wir unsere **for**-Schleife diesmal rückwärts laufen. Der `range`-Funktion müssen wir dafür drei Parameter übergeben: Zuerst als Startindex den des letzten Layers (Länge der Layer-Liste minus 1), dann eine `0` für den Wert für `l`, bei dem die Schleife abbricht, und zuletzt die Schrittänge von `-1`, die dafür sorgt, dass `l` bei jedem Schleifendurchlauf um eins herunterzählt.

# Backpropagation-Algorithmus



```
# ...
output_layer_derivative = 2 * diff

for l in range(len(self.activations)-1, 0, -1):
    if l == len(self.activations) - 1:
        partial_deltas[l] = output_layer_derivative * \
            self.output_activation_func(self.weighted_sums[l], derivative=True)
    else:
        partial_deltas[l] = \
            numpy.matmul(self.weights[l+1][:, 1:].T, partial_deltas[l+1]) * \
            self.hidden_activation_func(self.weighted_sums[l], derivative=True)

delta_W[l] += -self.eta * numpy.matmul(partial_deltas[l], self.activations[l-1].T)
```

Zuerst wird  $l$  den Index des Output-Layers haben, und da sich in ihm die Berechnung von den anderen Layern unterscheidet, bauen wir eine Fallunterscheidung ein.

# Backpropagation-Algorithmus

$$\delta_j^{(l)} = 2 \left( a_j^{(l)} - y_j \right) \cdot \sigma^{(l)'}(z_j^{(l)})$$

7

Für den Output-Layer berechnet sich die partielle Ableitung  $\delta_j$  wie im ersten Fall von Gleichung 7 angegeben. Da wir die Ableitung der Aktivierungsfunktion benötigen, rufen wir sie mit dem Parameter `derivative=True` auf.

```
● ● ●  
# ...  
output_layer_derivative = 2 * diff  
  
for l in range(len(self.activations)-1, 0, -1):  
    if l == len(self.activations) - 1:  
        partial_deltas[l] = output_layer_derivative * \  
            self.output_activation_func(self.weighted_sums[l], derivative=True)  
    else:  
        partial_deltas[l] = \  
            numpy.matmul(self.weights[l+1][:, 1:].T, partial_deltas[l+1]) * \  
            self.hidden_activation_func(self.weighted_sums[l], derivative=True)  
  
    delta_w[l] += -self.eta * numpy.matmul(partial_deltas[l], self.activations[l-1].T)
```

# Backpropagation-Algorithmus

$$\delta_j^{(l)} = 2 \left( a_j^{(l)} - y_j \right) \cdot \sigma^{(l)'}(z_j^{(l)})$$

7



```
# ...
output_layer_derivative = 2 * diff

for l in range(len(self.activations)-1, 0, -1):
    if l == len(self.activations) - 1:
        partial_deltas[l] = output_layer_derivative * \
            self.output_activation_func(self.weighted_sums[l], derivative=True)
    else:
        partial_deltas[l] = \
            numpy.matmul(self.weights[l+1][:, 1:].T, partial_deltas[l+1]) * \
            self.hidden_activation_func(self.weighted_sums[l], derivative=True)

delta_W[l] += -self.eta * numpy.matmul(partial_deltas[l], self.activations[l-1].T)
```

Ein \ am Ende der Zeile bedeutet, dass aus Platzgründen ein Zeilenumbruch gesetzt und der Ausdruck in der nächsten Zeile fortgeführt wird.

# Backpropagation-Algorithmus

$$\delta_j^{(l)} = \left( \sum_{i=0}^{n^{(l+1)}-1} \delta_i^{(l+1)} w_{ij}^{(l+1)} \right) \cdot \sigma^{(l)'}(z_j^{(l)})$$

7



```
# ...
output_layer_derivative = 2 * diff

for l in range(len(self.activations))-1, 0, -1):
    if l == len(self.activations) - 1:
        partial_deltas[l] = output_layer_derivative * \
            self.output_activation_func(self.weighted_sums[l], derivative=True)
    else:
        partial_deltas[l] = \
            numpy.matmul(self.weights[l+1][:, 1:].T, partial_deltas[l+1]) * \
            self.hidden_activation_func(self.weighted_sums[l], derivative=True)

    delta_w[l] += -self.eta * numpy.matmul(partial_deltas[l], self.activations[l-1].T)
```

Für die Hidden Layer richten wir uns nach dem zweiten Fall in Gleichung 7. Der zweite Faktor ist fast identisch, auch hier rufen wir die Aktivierungsfunktion (diesmal die für die Hidden Layer) mit den gewichteten Summen und **derivative=True** als Parameter auf.

# Backpropagation-Algorithmus

$$\delta_j^{(l)} = \left( \sum_{i=0}^{n^{(l+1)}-1} \delta_i^{(l+1)} w_{ij}^{(l+1)} \right) \cdot \sigma^{(l)'}(z_j^{(l)})$$

7



```
# ...
output_layer_derivative = 2 * diff

for l in range(len(self.activations)-1, 0, -1):
    if l == len(self.activations) - 1:
        partial_deltas[l] = output_layer_derivative * \
            self.output_activation_func(self.weighted_sums[l], derivative=True)
    else:
        partial_deltas[l] = \
            numpy.matmul(self.weights[l+1][:, 1:].T, partial_deltas[l+1]) * \
            self.hidden_activation_func(self.weighted_sums[l], derivative=True)

    delta_w[l] += -self.eta * numpy.matmul(partial_deltas[l], self.activations[l-1].T)
```

Die große Summe im ersten Faktor führen wir auf eine **Matrixmultiplikation** der Gewichte zum nächsten Layer und der partiellen Ableitungen des nächsten Layers zurück.

# Backpropagation-Algorithmus

$$\delta_j^{(l)} = \left( \sum_{i=0}^{n^{(l+1)}-1} \delta_i^{(l+1)} w_{ij}^{(l+1)} \right) \cdot \sigma^{(l)'}(z_j^{(l)})$$

7



```
# ...
output_layer_derivative = 2 * diff

for l in range(len(self.activations))-1, 0, -1):
    if l == len(self.activations) - 1:
        partial_deltas[l] = output_layer_derivative * \
            self.output_activation_func(self.weighted_sums[l], derivative=True)
    else:
        partial_deltas[l] = \
            numpy.matmul(self.weights[l+1][:, 1:].T, partial_deltas[l+1]) * \
            self.hidden_activation_func(self.weighted_sums[l], derivative=True)

    delta_w[l] += -self.eta * numpy.matmul(partial_deltas[l], self.activations[l-1].T)
```

Mit der Slice-Syntax schneiden wir dafür die erste Spalte der Gewichtsmatrix heraus, also die von den Bias-Neuronen ausgehenden Gewichte. Die Bias-Neuronen sollen sich nicht auf die vorherigen Layer auswirken und daher keinen Eintrag in `partial_deltas` erzeugen.

# Backpropagation-Algorithmus

$$\delta_j^{(l)} = \left( \sum_{i=0}^{n^{(l+1)}-1} \delta_i^{(l+1)} w_{ij}^{(l+1)} \right) \cdot \sigma^{(l)'}(z_j^{(l)})$$

7



```
# ...
output_layer_derivative = 2 * diff

for l in range(len(self.activations)-1, 0, -1):
    if l == len(self.activations) - 1:
        partial_deltas[l] = output_layer_derivative * \
            self.output_activation_func(self.weighted_sums[l], derivative=True)
    else:
        partial_deltas[l] = \
            numpy.matmul(self.weights[l+1][:, 1:].T, partial_deltas[l+1]) * \
            self.hidden_activation_func(self.weighted_sums[l], derivative=True)

delta_w[l] += -self.eta * numpy.matmul(partial_deltas[l], self.activations[l-1].T)
```

Von der auf diese Weise zurechtgeschnittenen Matrix benötigen wir die transponierte Version (siehe nächste Folie), die wir von numpy mit `.T` erhalten.

# Matrixtransposition

- Eine Matrix zu **transponieren** heißt, ihre **Zeilen** und **Spalten** die Rollen tauschen zu lassen.

# Matrixtransposition

- Eine Matrix zu **transponieren** heißt, ihre **Zeilen** und **Spalten** die Rollen tauschen zu lassen.
- Sei eine Matrix  $W$  mit  $m$  Zeilen und  $n$  Spalten gegeben durch:

$$W = \begin{pmatrix} w_{1,1} & \dots & w_{1,n} \\ \vdots & \ddots & \vdots \\ w_{m,1} & \dots & w_{m,n} \end{pmatrix}$$

# Matrixtransposition

- Eine Matrix zu **transponieren** heißt, ihre **Zeilen** und **Spalten** die Rollen tauschen zu lassen.
- Sei eine Matrix  $W$  mit  $m$  Zeilen und  $n$  Spalten gegeben durch:

$$W = \begin{pmatrix} w_{1,1} & \dots & w_{1,n} \\ \vdots & \ddots & \vdots \\ w_{m,1} & \dots & w_{m,n} \end{pmatrix}$$

- Dann ist ihre transponierte Matrix  $W^T$  definiert als:

$$W^T = \begin{pmatrix} w_{1,1} & \dots & w_{m,1} \\ \vdots & \ddots & \vdots \\ w_{1,n} & \dots & w_{m,n} \end{pmatrix}$$

# Matrixtransposition – Beispiel

$$W = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$



$$W^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

# Matrixtransposition – Beispiel

$$W = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$



$$W^T = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{pmatrix}$$

2 Zeilen, 3 Spalten

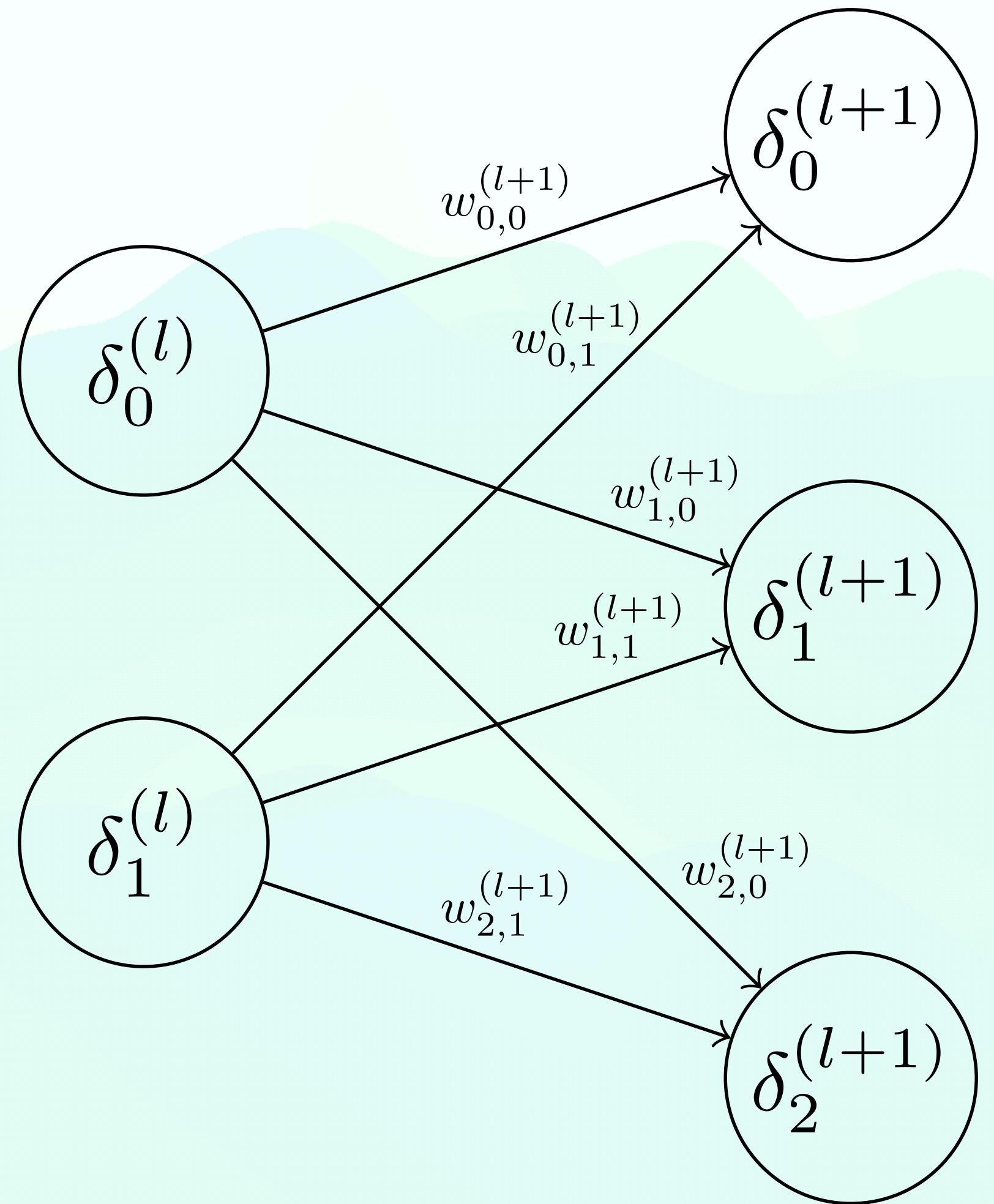
3 Zeilen, 2 Spalten

# Backpropagation als Matrixmultiplikation

$$\delta_j^{(l)} = \left( \sum_{i=0}^{n^{(l+1)}-1} \delta_i^{(l+1)} w_{ij}^{(l+1)} \right) \cdot \sigma^{(l)'}(z_j^{(l)})$$

7

Es werden die  $\delta_i$  und die  $w_{ij}$  von Layer  $l + 1$  miteinander multipliziert.



# Backpropagation als Matrixmultiplikation

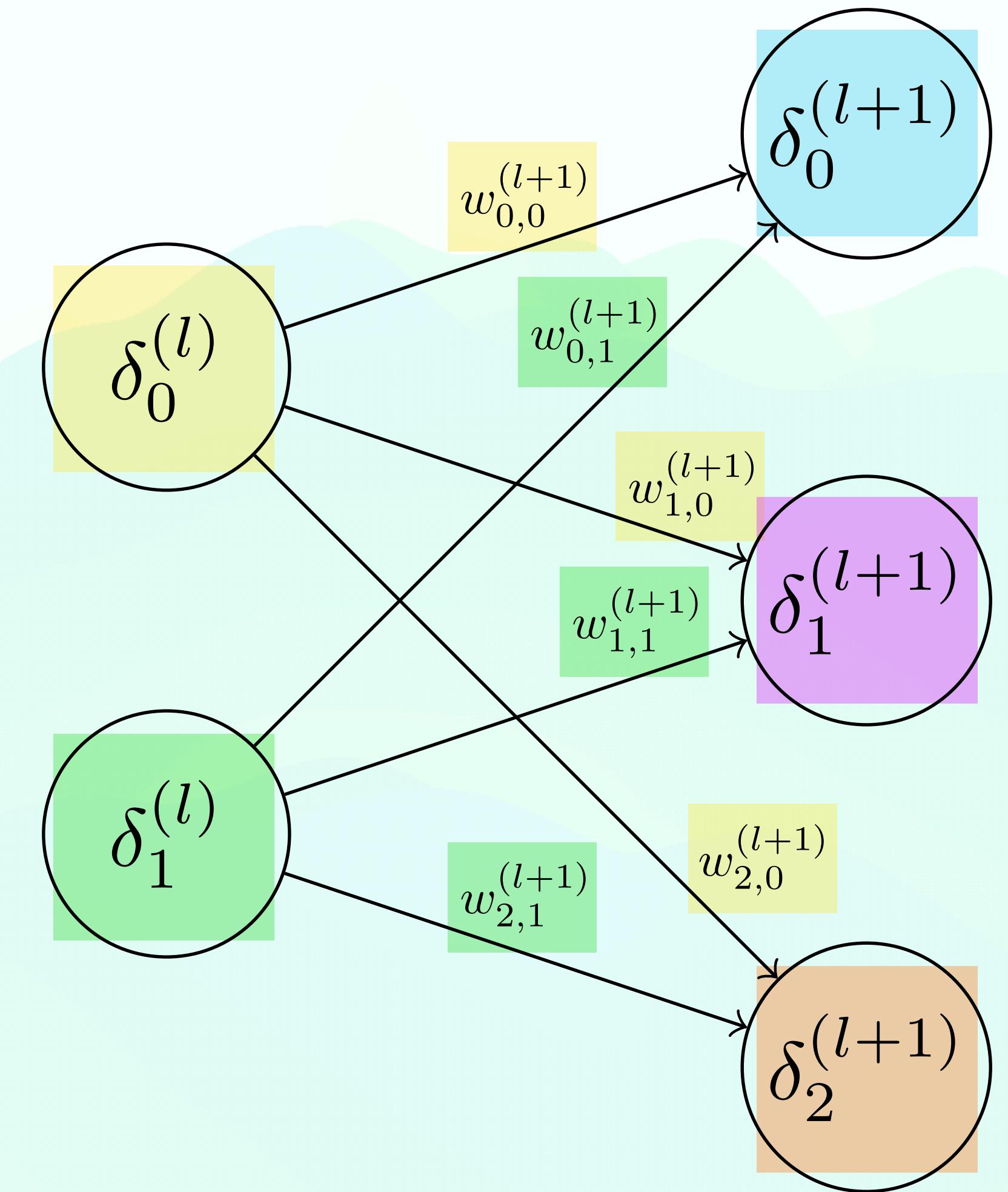
$$\delta_j^{(l)} = \left( \sum_{i=0}^{n^{(l+1)}-1} \delta_i^{(l+1)} w_{ij}^{(l+1)} \right) \cdot \sigma^{(l)'}(z_j^{(l)})$$

7

Es werden die  $\delta_i$  und die  $w_{ij}$  von Layer  $l + 1$  miteinander multipliziert.

Für die einzelnen  $\delta_j^{(l)}$  ergeben sich also folgende Summen:

$\delta_j^{(l)}$	Summe für $\delta_j^{(l)}$
$\delta_0^{(l)}$	$\delta_0^{(l+1)} w_{0,0}^{(l+1)} + \delta_1^{(l+1)} w_{1,0}^{(l+1)} + \delta_2^{(l+1)} w_{2,0}^{(l+1)}$
$\delta_1^{(l)}$	$\delta_0^{(l+1)} w_{0,1}^{(l+1)} + \delta_1^{(l+1)} w_{1,1}^{(l+1)} + \delta_2^{(l+1)} w_{2,1}^{(l+1)}$



# Backpropagation als Matrixmultiplikation

$\delta_j^{(l)}$	Summe für $\delta_j^{(l)}$
$\delta_0^{(l)}$	$\delta_0^{(l+1)}w_{0,0}^{(l+1)} + \delta_1^{(l+1)}w_{1,0}^{(l+1)} + \delta_2^{(l+1)}w_{2,0}^{(l+1)}$
$\delta_1^{(l)}$	$\delta_0^{(l+1)}w_{0,1}^{(l+1)} + \delta_1^{(l+1)}w_{1,1}^{(l+1)} + \delta_2^{(l+1)}w_{2,1}^{(l+1)}$

Die beiden zu multiplizierenden Matrizen dafür sind:

$$\vec{\delta}^{(l+1)} = \begin{pmatrix} \delta_0^{(l+1)} \\ \delta_1^{(l+1)} \\ \delta_2^{(l+1)} \end{pmatrix}$$

$$W^{(l+1)} = \begin{pmatrix} w_{0,0}^{(l+1)} & w_{0,1}^{(l+1)} \\ w_{1,0}^{(l+1)} & w_{1,1}^{(l+1)} \\ w_{2,0}^{(l+1)} & w_{2,1}^{(l+1)} \end{pmatrix}$$

# Backpropagation als Matrixmultiplikation

$\delta_j^{(l)}$	Summe für $\delta_j^{(l)}$
$\delta_0^{(l)}$	$\delta_0^{(l+1)}w_{0,0}^{(l+1)} + \delta_1^{(l+1)}w_{1,0}^{(l+1)} + \delta_2^{(l+1)}w_{2,0}^{(l+1)}$
$\delta_1^{(l)}$	$\delta_0^{(l+1)}w_{0,1}^{(l+1)} + \delta_1^{(l+1)}w_{1,1}^{(l+1)} + \delta_2^{(l+1)}w_{2,1}^{(l+1)}$

Doch für die Matrixmultiplikation muss die erste Matrix so viele **Spalten** haben wie die zweite **Zeilen** hat.

Die beiden zu multiplizierenden Matrizen dafür sind:

$$\vec{\delta}^{(l+1)} = \begin{pmatrix} \delta_0^{(l+1)} \\ \delta_1^{(l+1)} \\ \delta_2^{(l+1)} \end{pmatrix}$$

$$W^{(l+1)} = \begin{pmatrix} w_{0,0}^{(l+1)} & w_{0,1}^{(l+1)} \\ w_{1,0}^{(l+1)} & w_{1,1}^{(l+1)} \\ w_{2,0}^{(l+1)} & w_{2,1}^{(l+1)} \end{pmatrix}$$

# Backpropagation als Matrixmultiplikation

$\delta_j^{(l)}$	Summe für $\delta_j^{(l)}$
$\delta_0^{(l)}$	$\delta_0^{(l+1)}w_{0,0}^{(l+1)} + \delta_1^{(l+1)}w_{1,0}^{(l+1)} + \delta_2^{(l+1)}w_{2,0}^{(l+1)}$
$\delta_1^{(l)}$	$\delta_0^{(l+1)}w_{0,1}^{(l+1)} + \delta_1^{(l+1)}w_{1,1}^{(l+1)} + \delta_2^{(l+1)}w_{2,1}^{(l+1)}$

Doch für die Matrixmultiplikation muss die erste Matrix so viele **Spalten** haben wie die zweite **Zeilen** hat.

Wenn wir  $W^{(l+1)}$  transponieren, erhalten wir eine Matrix mit zwei Zeilen und **drei Spalten**, und können diese transponierte Matrix dann mit  $\vec{\delta}^{(l+1)}$  (**drei Zeilen**) multiplizieren.

Die beiden zu multiplizierenden Matrizen dafür sind:

$$\vec{\delta}^{(l+1)} = \begin{pmatrix} \delta_0^{(l+1)} \\ \delta_1^{(l+1)} \\ \delta_2^{(l+1)} \end{pmatrix}$$

$$W^{(l+1)} = \begin{pmatrix} w_{0,0}^{(l+1)} & w_{0,1}^{(l+1)} \\ w_{1,0}^{(l+1)} & w_{1,1}^{(l+1)} \\ w_{2,0}^{(l+1)} & w_{2,1}^{(l+1)} \end{pmatrix}$$

# Backpropagation als Matrixmultiplikation

$\delta_j^{(l)}$	Summe für $\delta_j^{(l)}$
$\delta_0^{(l)}$	$\delta_0^{(l+1)}w_{0,0}^{(l+1)} + \delta_1^{(l+1)}w_{1,0}^{(l+1)} + \delta_2^{(l+1)}w_{2,0}^{(l+1)}$
$\delta_1^{(l)}$	$\delta_0^{(l+1)}w_{0,1}^{(l+1)} + \delta_1^{(l+1)}w_{1,1}^{(l+1)} + \delta_2^{(l+1)}w_{2,1}^{(l+1)}$

Doch für die Matrixmultiplikation muss die erste Matrix so viele **Spalten** haben wie die zweite **Zeilen** hat.

Wenn wir  $W^{(l+1)}$  transponieren, erhalten wir eine Matrix mit zwei Zeilen und **drei Spalten**, und können diese transponierte Matrix dann mit  $\vec{\delta}^{(l+1)}$  (**drei Zeilen**) multiplizieren.

Das Ergebnis ist eine Matrix mit **einer Spalte** und **zwei Zeilen**.

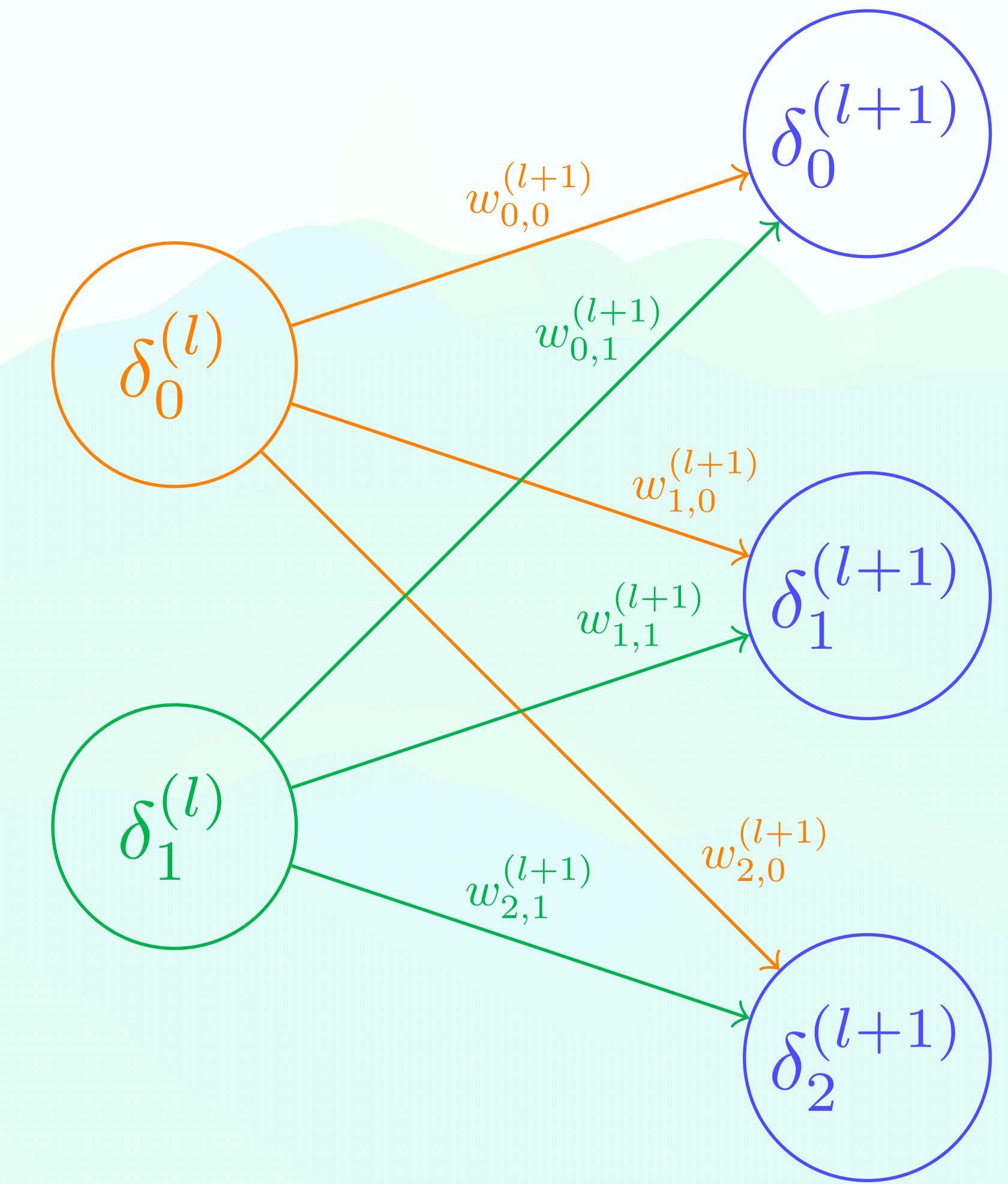
Die beiden zu multiplizierenden Matrizen dafür sind:

$$\vec{\delta}^{(l+1)} = \begin{pmatrix} \delta_0^{(l+1)} \\ \delta_1^{(l+1)} \\ \delta_2^{(l+1)} \end{pmatrix}$$

$$W^{(l+1)} = \begin{pmatrix} w_{0,0}^{(l+1)} & w_{0,1}^{(l+1)} \\ w_{1,0}^{(l+1)} & w_{1,1}^{(l+1)} \\ w_{2,0}^{(l+1)} & w_{2,1}^{(l+1)} \end{pmatrix}$$

# Backpropagation als Matrixmultiplikation

$$\begin{aligned}
 & \left( W^{(l+1)} \right)^T \times \vec{\delta}^{(l+1)} = \\
 & \begin{pmatrix} w_{0,0}^{(l+1)} & w_{1,0}^{(l+1)} & w_{2,0}^{(l+1)} \\ w_{0,1}^{(l+1)} & w_{1,1}^{(l+1)} & w_{2,1}^{(l+1)} \end{pmatrix} \times \begin{pmatrix} \delta_0^{(l+1)} \\ \delta_1^{(l+1)} \\ \delta_2^{(l+1)} \end{pmatrix} = \\
 & \left( \begin{matrix} w_{0,0}^{(l+1)} \delta_0^{(l+1)} \\ w_{0,1}^{(l+1)} \delta_0^{(l+1)} \end{matrix} + \begin{matrix} w_{1,0}^{(l+1)} \delta_1^{(l+1)} \\ w_{1,1}^{(l+1)} \delta_1^{(l+1)} \end{matrix} + \begin{matrix} w_{2,0}^{(l+1)} \delta_2^{(l+1)} \\ w_{2,1}^{(l+1)} \delta_2^{(l+1)} \end{matrix} \right) = \begin{pmatrix} \delta_0^{(l)} \\ \delta_1^{(l)} \end{pmatrix}
 \end{aligned}$$



# Backpropagation-Algorithmus

$$\Delta w_{jk}^{(l)} = -\eta \cdot \delta_j^{(l)} \cdot a_k^{(l-1)}$$

8



```
# ...
output_layer_derivative = 2 * diff

for l in range(len(self.activations)-1, 0, -1):
    if l == len(self.activations) - 1:
        partial_deltas[l] = output_layer_derivative * \
            self.output_activation_func(self.weighted_sums[l], derivative=True)
    else:
        partial_deltas[l] = \
            numpy.matmul(self.weights[l+1][:, 1:].T, partial_deltas[l+1]) * \
            self.hidden_activation_func(self.weighted_sums[l], derivative=True)

    delta_W[l] += -self.eta * numpy.matmul(partial_deltas[l], self.activations[l-1].T)
```

Im letzten Schritt berechnen wir nach Gleichung 8 die Gewichtsanpassungen  $\Delta w$  für den aktuellen Layer  $l$ , also den ermittelten Gradienten zur Minimierung der Kostenfunktion.

# Backpropagation-Algorithmus

$$\Delta w_{jk}^{(l)} = -\eta \cdot \delta_j^{(l)} \cdot a_k^{(l-1)}$$

8



```
# ...
output_layer_derivative = 2 * diff

for l in range(len(self.activations)-1, 0, -1):
    if l == len(self.activations) - 1:
        partial_deltas[l] = output_layer_derivative * \
            self.output_activation_func(self.weighted_sums[l], derivative=True)
    else:
        partial_deltas[l] = \
            numpy.matmul(self.weights[l+1][:, 1:].T, partial_deltas[l+1]) * \
            self.hidden_activation_func(self.weighted_sums[l], derivative=True)

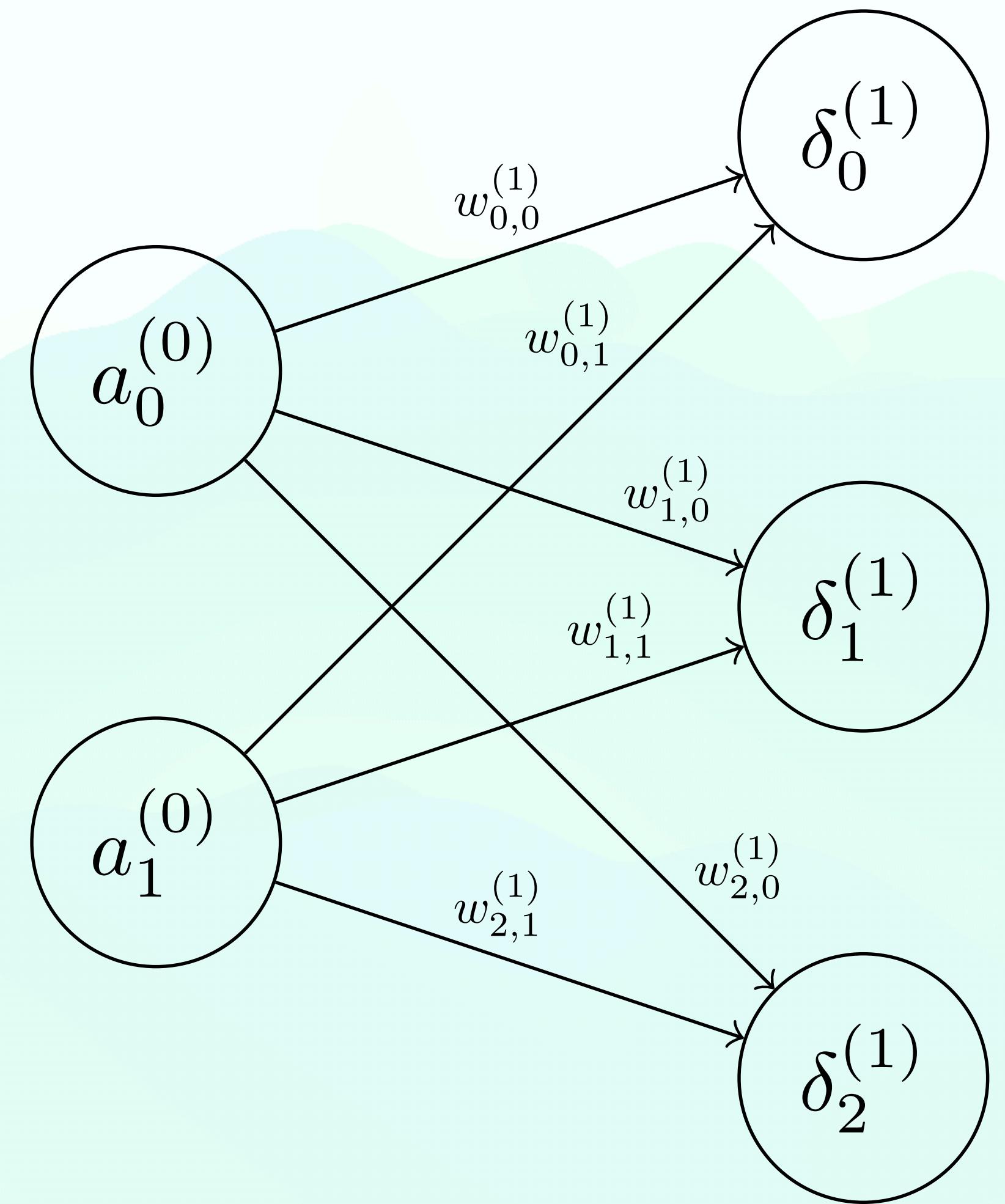
    delta_w[l] += -self.eta * numpy.matmul(partial_deltas[l], self.activations[l-1].T)
```

Um die Multiplikationen der partiellen Ableitungen des Layers  $l$  und der Aktivierungen des Layers  $l-1$  nicht einzeln ausführen zu müssen, führen wir auch diese Operation auf eine Matrixmultiplikation zurück.

# Anpassung der Gewichte als Matrixmultiplikation

$$\Delta w_{jk}^{(l)} = -\eta \cdot \delta_j^{(l)} \cdot a_k^{(l-1)}$$

8



# Anpassung der Gewichte als Matrixmultiplikation

$$\Delta w_{jk}^{(l)} = -\eta \cdot \delta_j^{(l)} \cdot a_k^{(l-1)}$$

8

Das heißt für das Beispielnetz rechts:

$$\Delta w_{0,0}^{(1)} = -\eta \cdot \delta_0^{(1)} \cdot a_0^{(0)}$$

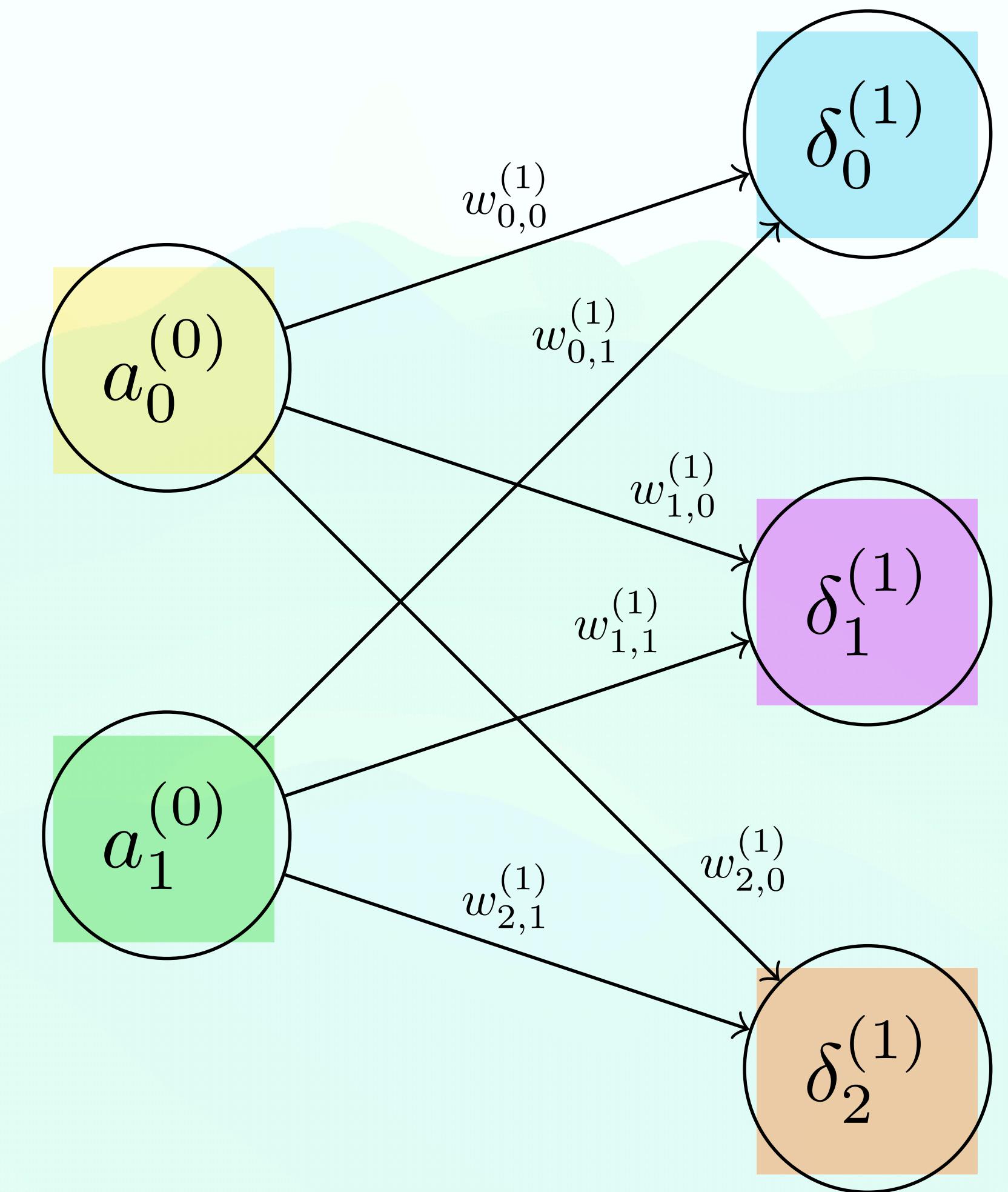
$$\Delta w_{0,1}^{(1)} = -\eta \cdot \delta_0^{(1)} \cdot a_1^{(0)}$$

$$\Delta w_{1,0}^{(1)} = -\eta \cdot \delta_1^{(1)} \cdot a_0^{(0)}$$

$$\Delta w_{1,1}^{(1)} = -\eta \cdot \delta_1^{(1)} \cdot a_1^{(0)}$$

$$\Delta w_{2,0}^{(1)} = -\eta \cdot \delta_2^{(1)} \cdot a_0^{(0)}$$

$$\Delta w_{2,1}^{(1)} = -\eta \cdot \delta_2^{(1)} \cdot a_1^{(0)}$$



# Anpassung der Gewichte als Matrixmultiplikation

$$\Delta w_{jk}^{(l)} = -\eta \cdot \delta_j^{(l)} \cdot a_k^{(l-1)}$$

8

Das heißt für das Beispielnetz rechts:

$$\Delta w_{0,0}^{(1)} = -\eta \cdot \delta_0^{(1)} \cdot a_0^{(0)}$$

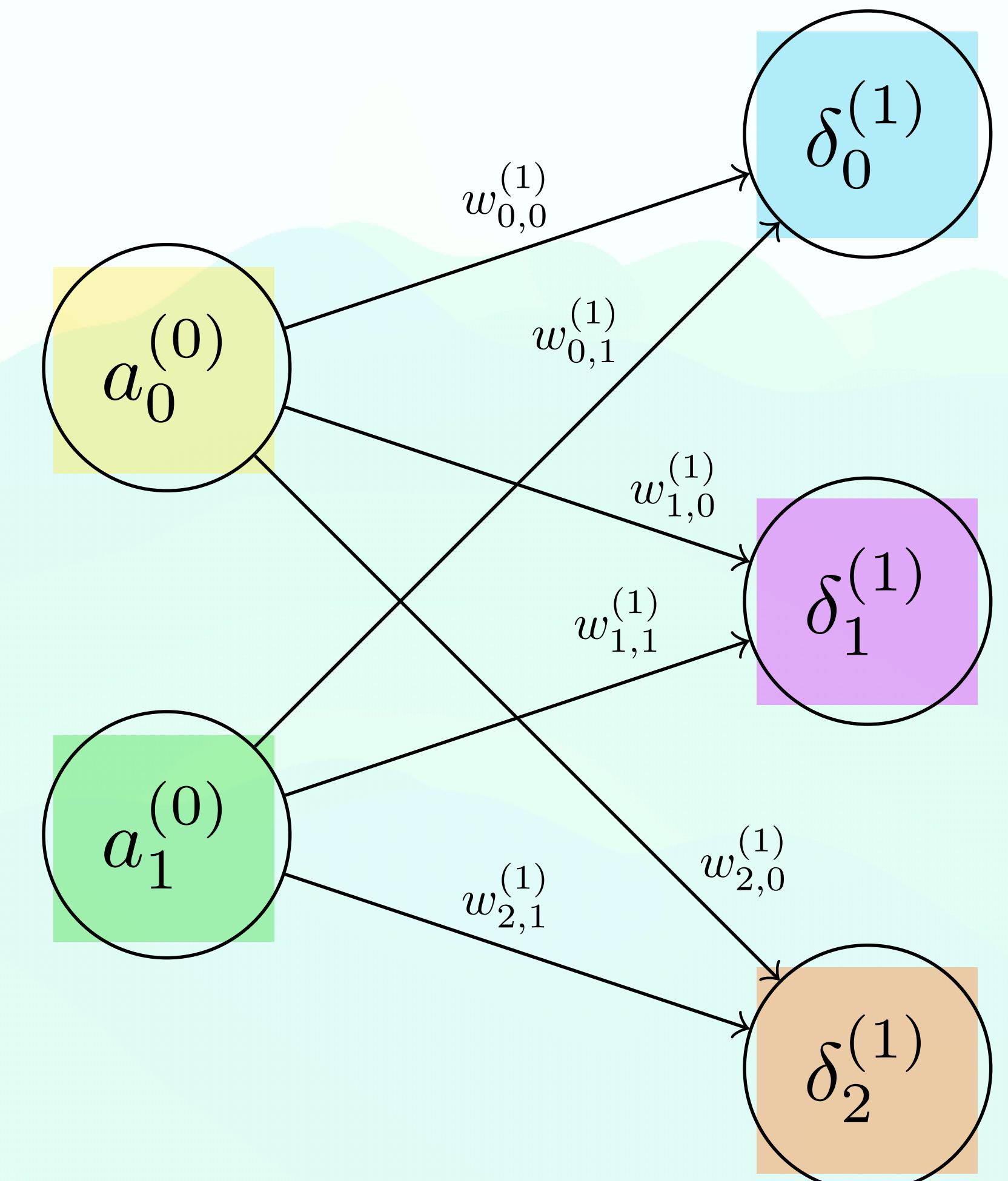
$$\Delta w_{1,0}^{(1)} = -\eta \cdot \delta_1^{(1)} \cdot a_0^{(0)}$$

$$\Delta w_{2,0}^{(1)} = -\eta \cdot \delta_2^{(1)} \cdot a_0^{(0)}$$

$$\Delta w_{0,1}^{(1)} = -\eta \cdot \delta_0^{(1)} \cdot a_1^{(0)}$$

$$\Delta w_{1,1}^{(1)} = -\eta \cdot \delta_1^{(1)} \cdot a_1^{(0)}$$

$$\Delta w_{2,1}^{(1)} = -\eta \cdot \delta_2^{(1)} \cdot a_1^{(0)}$$



$\Delta W^{(1)}$  soll eine Matrix vom gleichen Format wie  $W^{(1)}$  sein, also **drei Zeilen und zwei Spalten** haben

# Anpassung der Gewichte als Matrixmultiplikation

$$\Delta w_{0,0}^{(1)} = -\eta \cdot \delta_0^{(1)} \cdot a_0^{(0)}$$

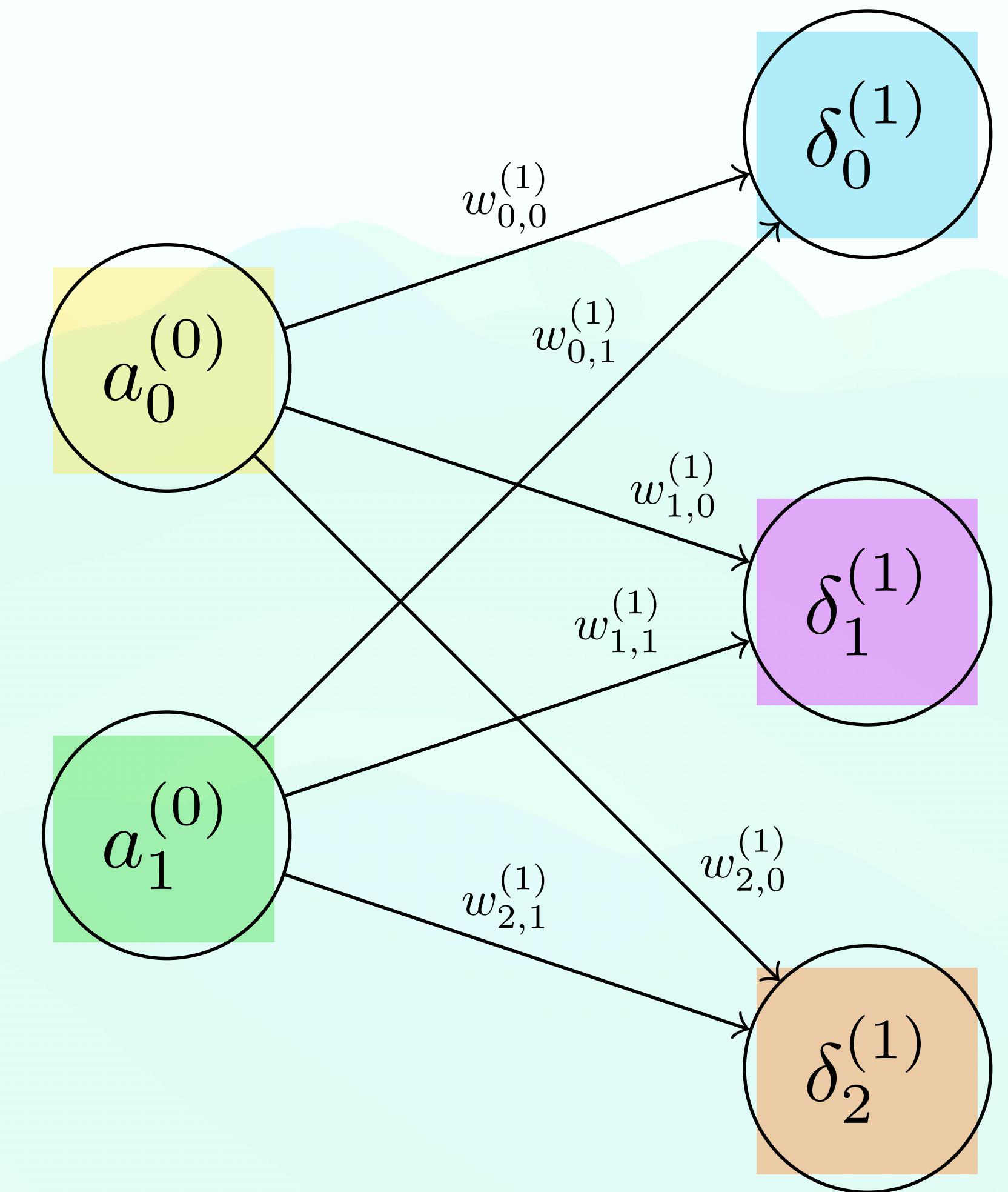
$$\Delta w_{1,0}^{(1)} = -\eta \cdot \delta_1^{(1)} \cdot a_0^{(0)}$$

$$\Delta w_{2,0}^{(1)} = -\eta \cdot \delta_2^{(1)} \cdot a_0^{(0)}$$

$$\Delta w_{0,1}^{(1)} = -\eta \cdot \delta_0^{(1)} \cdot a_1^{(0)}$$

$$\Delta w_{1,1}^{(1)} = -\eta \cdot \delta_1^{(1)} \cdot a_1^{(0)}$$

$$\Delta w_{2,1}^{(1)} = -\eta \cdot \delta_2^{(1)} \cdot a_1^{(0)}$$



# Anpassung der Gewichte als Matrixmultiplikation

$$\Delta w_{0,0}^{(1)} = -\eta \cdot \delta_0^{(1)} \cdot a_0^{(0)}$$

$$\Delta w_{0,1}^{(1)} = -\eta \cdot \delta_0^{(1)} \cdot a_1^{(0)}$$

$$\Delta w_{1,0}^{(1)} = -\eta \cdot \delta_1^{(1)} \cdot a_0^{(0)}$$

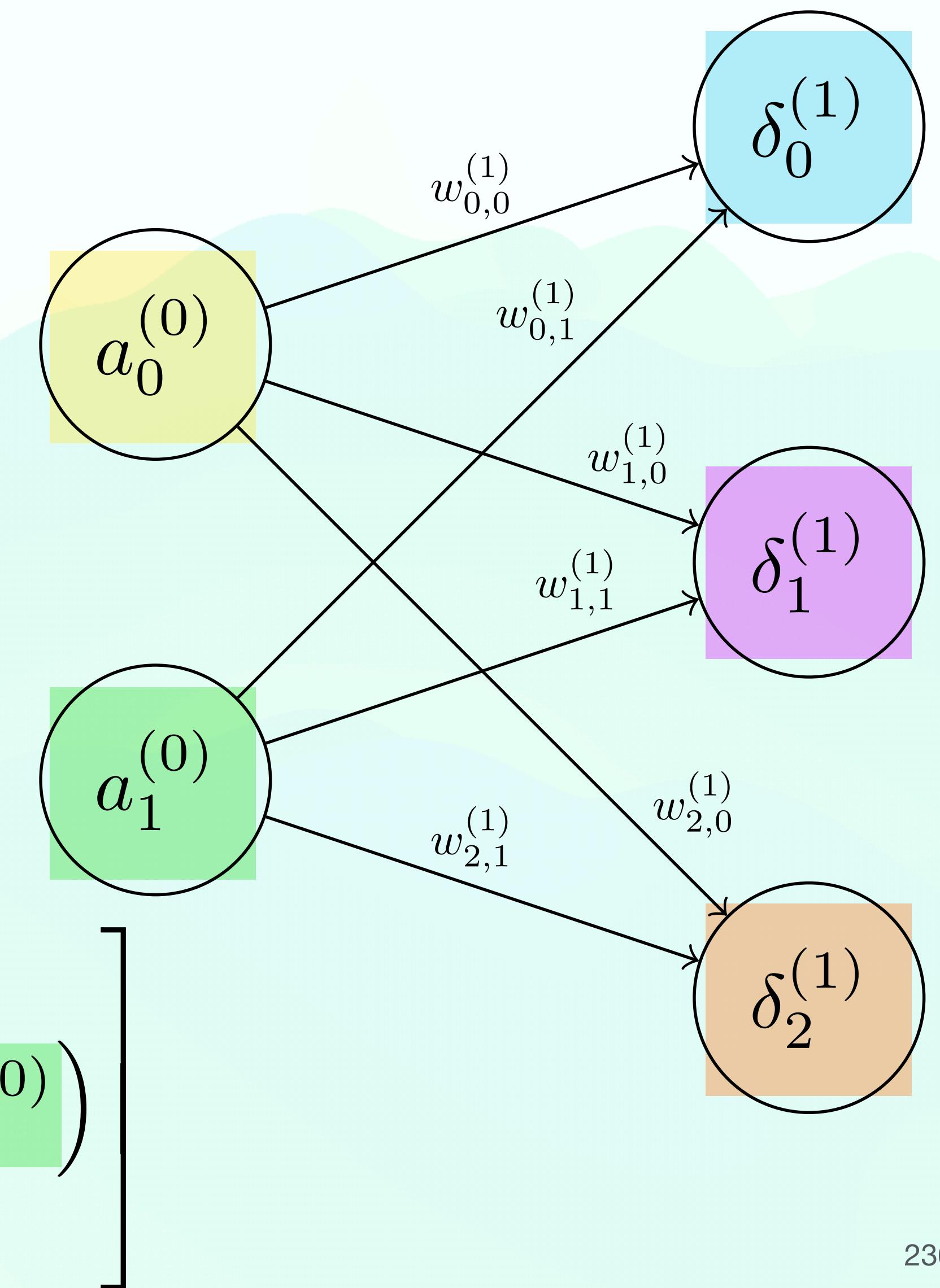
$$\Delta w_{1,1}^{(1)} = -\eta \cdot \delta_1^{(1)} \cdot a_1^{(0)}$$

$$\Delta w_{2,0}^{(1)} = -\eta \cdot \delta_2^{(1)} \cdot a_0^{(0)}$$

$$\Delta w_{2,1}^{(1)} = -\eta \cdot \delta_2^{(1)} \cdot a_1^{(0)}$$

Das erreichen wir, indem wir den Vektor mit den Deltas (**eine Spalte, drei Zeilen**) mit dem transponierten Vektor der Aktivierungen (**zwei Spalten, eine Zeile**) multiplizieren:

$$\Delta W^{(1)} = -\eta \cdot \left( \delta^{(1)} A^{(l-1)^T} \right) = -\eta \cdot \left[ \begin{pmatrix} \delta_0^{(1)} \\ \delta_1^{(1)} \\ \delta_2^{(1)} \end{pmatrix} \left( \begin{matrix} a_0^{(0)} & a_1^{(0)} \end{matrix} \right) \right]$$



# Backpropagation-Algorithmus

$$\Delta w_{jk}^{(l)} = -\eta \cdot \delta_j^{(l)} \cdot a_k^{(l-1)}$$

8



```
# ...
output_layer_derivative = 2 * diff

for l in range(len(self.activations)-1, 0, -1):
    if l == len(self.activations) - 1:
        partial_deltas[l] = output_layer_derivative * \
            self.output_activation_func(self.weighted_sums[l], derivative=True)
    else:
        partial_deltas[l] = \
            numpy.matmul(self.weights[l+1][:, 1:].T, partial_deltas[l+1]) * \
            self.hidden_activation_func(self.weighted_sums[l], derivative=True)

    delta_w[l] += -self.eta * numpy.matmul(partial_deltas[l], self.activations[l-1].T)
```

Das Ergebnis wird dann mit der Lernrate  $\eta$  multipliziert und bekommt ein negatives Vorzeichen, da wir uns in die Richtung des steilsten Gradientenabstiegs bewegen wollen.

# Backpropagation-Algorithmus

$$\Delta w_{jk}^{(l)} = -\eta \cdot \delta_j^{(l)} \cdot a_k^{(l-1)}$$

8



```
# ...
output_layer_derivative = 2 * diff

for l in range(len(self.activations)-1, 0, -1):
    if l == len(self.activations) - 1:
        partial_deltas[l] = output_layer_derivative * \
            self.output_activation_func(self.weighted_sums[l], derivative=True)
    else:
        partial_deltas[l] = \
            numpy.matmul(self.weights[l+1][:, 1:].T, partial_deltas[l+1]) * \
            self.hidden_activation_func(self.weighted_sums[l], derivative=True)

    delta_W[l] += -self.eta * numpy.matmul(partial_deltas[l], self.activations[l-1].T)
```

Ganz wichtig zu beachten: Wir wollen unsere gewünschten Gewichtsanpassungen der einzelnen Trainingsbeispiele zunächst nur aufsummieren und bilden später den Durchschnitt. Dafür benutzen wir die Python-Syntax `+=`. Wir weisen den Wert damit nicht direkt zu, sondern addieren ihn auf den bisherigen Wert auf.

# Backpropagation-Algorithmus



```
for _ in range(self.n_iterations):
    for X, Y in training_data:
        # ...

    for l in range(1, len(delta_W)):
        self.weights[l] += delta_W[l] / len(training_data)
        delta_W[l][:] = 0
```

# Backpropagation-Algorithmus



```
for _ in range(self.n_iterations):
    for X, Y in training_data:
        # ...

        for l in range(1, len(delta_W)):
            self.weights[l] += delta_W[l] / len(training_data)
            delta_W[l][:] = 0
```

Zum Schluss führen wir den eigentlichen Lernschritt durch und passen die Gewichte an. Dafür legen wir zunächst eine weitere **for**-Schleife durch alle Layer des Netzes an, und zwar nach dem Ende der Schleife, die durch alle Trainingsdatensätze iteriert ist. Beide Schleifen müssen also auf derselben Einrückungsebene liegen.

# Backpropagation-Algorithmus



```
for _ in range(self.n_iterations):
    for X, Y in training_data:
        # ...

    for l in range(1, len(delta_W)):
        self.weights[l] += delta_W[l] / len(training_data)
        delta_W[l][:] = 0
```

Wir nehmen unsere Matrix mit den einzelnen, für jedes Trainingsbeispiel aufsummierten  $\Delta w$  im aktuellen Layer  $l$  und teilen sie durch die Anzahl der Trainingsbeispiele, um den Mittelwert zu berechnen. Eine Matrix geteilt durch eine Zahl ergibt in numpy praktischerweise wieder eine Matrix, in der jedes Element durch diese Zahl geteilt wurde.

# Backpropagation-Algorithmus



```
for _ in range(self.n_iterations):
    for X, Y in training_data:
        # ...

    for l in range(1, len(delta_W)):
        self.weights[l] += delta_W[l] / len(training_data)
        delta_W[l][:] = 0
```

Zu beachten ist auch hier wieder die Syntax  $+=$ .  
Wir wollen  $\Delta w$  auf die bestehenden Gewichte  
addieren, und sie nicht einfach überschreiben.

# Backpropagation-Algorithmus



```
for _ in range(self.n_iterations):
    for X, Y in training_data:
        # ...

        for l in range(1, len(delta_W)):
            self.weights[l] += delta_W[l] / len(training_data)
            delta_W[l][:] = 0
```

Bevor die nächste Iteration der äußersten Schleife beginnt, müssen die  $\Delta w$  wieder auf  $0$  zurückgesetzt werden, da der hier aufsummierte Wert nur für die aktuelle Trainingsiteration gilt.

# Backpropagation-Algorithmus



```
for _ in range(self.n_iterations):
    for X, Y in training_data:
        # ...

    for l in range(1, len(delta_W)):
        self.weights[l] += delta_W[l] / len(training_data)
        delta_W[l][:] = 0
```

Wir verwenden auch hier die Slice-Syntax. Es soll nicht einfach `delta_W[l]` den Wert 0 erhalten, sondern jedes einzelne Element darin. Dafür sorgen wir mit dem `[ : ]`.

# Damit ist unser neuronales Netz einsatzbereit!

Jetzt müssen wir nur noch eine Aufgabe dafür finden ...

# Das neuronale Netz trainieren

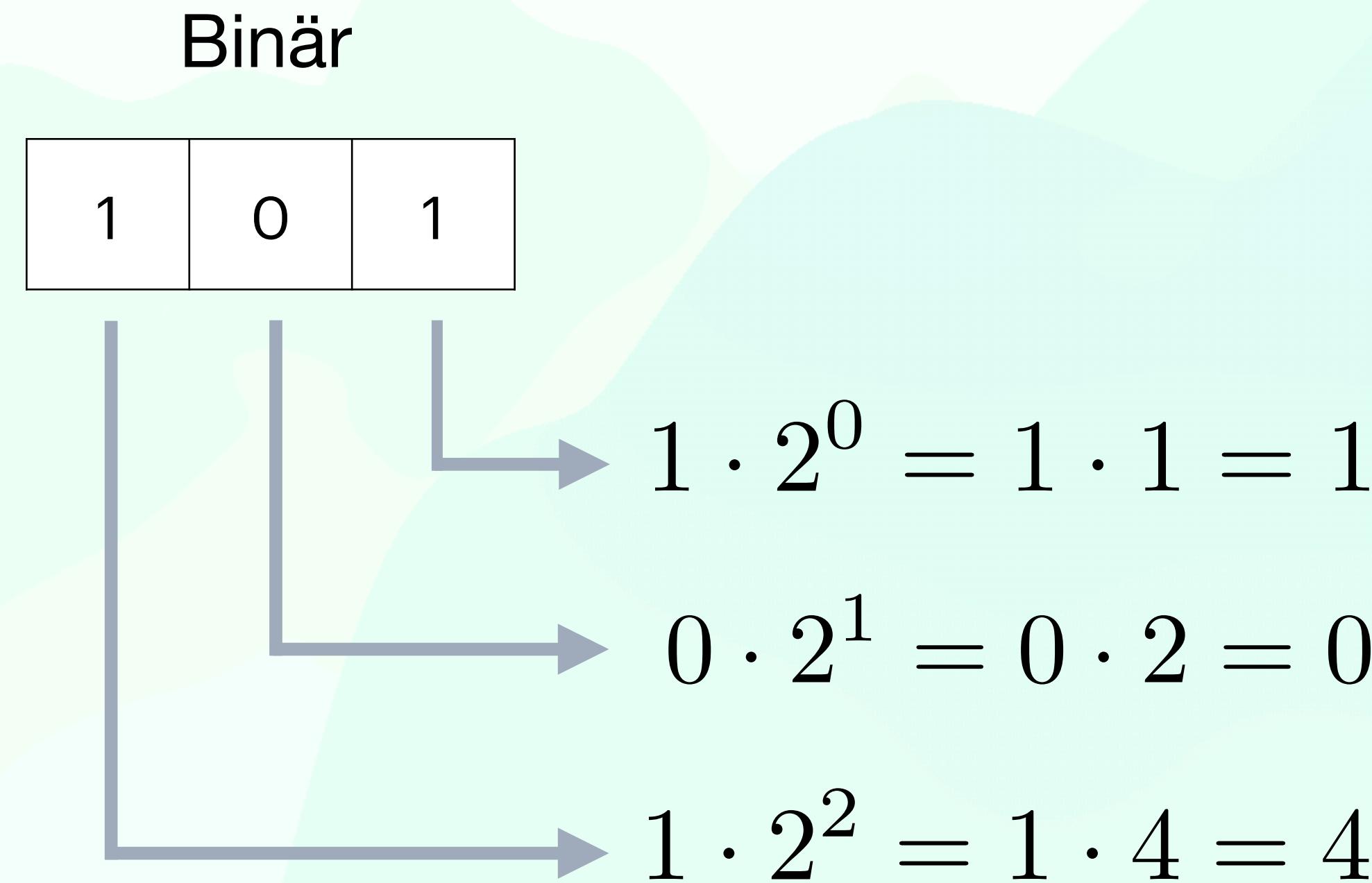
- Unser neuronales Netz soll lernen, dreistellige Binärzahlen in Dezimalzahlen umzuwandeln

# Das neuronale Netz trainieren

- Unser neuronales Netz soll lernen, dreistellige Binärzahlen in Dezimalzahlen umzuwandeln
- Binärzahlen bestehen nur aus den Ziffern 0 und 1, diese können direkt als Input-Aktivierungen verwendet werden

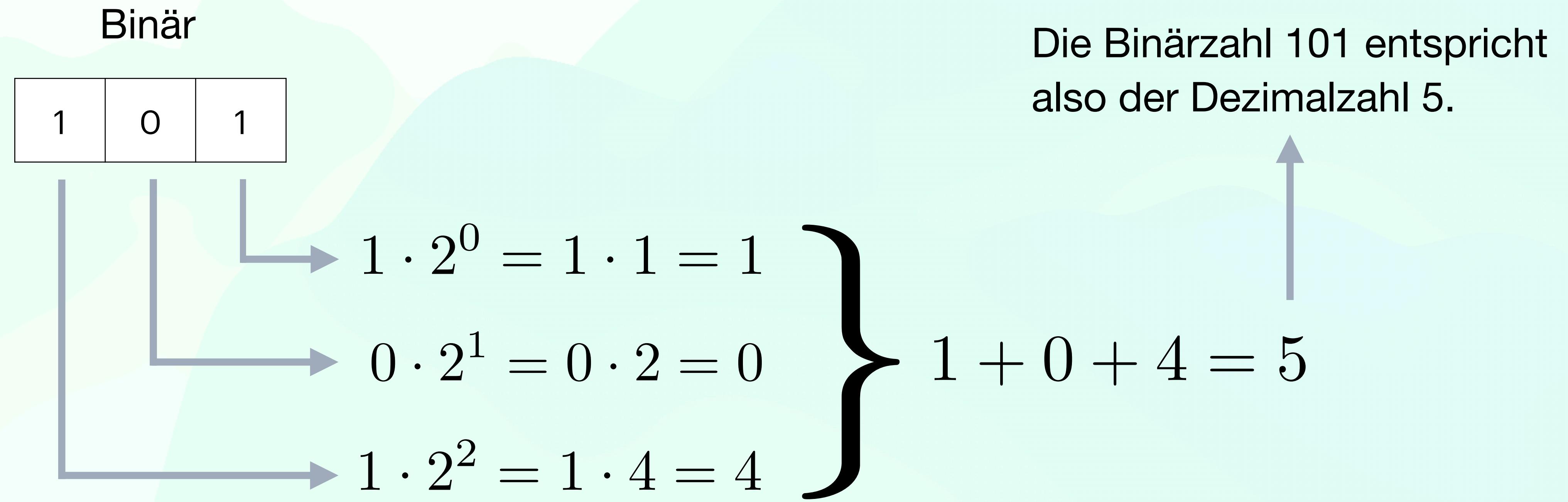
# Das neuronale Netz trainieren

- Unser neuronales Netz soll lernen, dreistellige Binärzahlen in Dezimalzahlen umzuwandeln
- Binärzahlen bestehen nur aus den Ziffern 0 und 1, diese können direkt als Input-Aktivierungen verwendet werden



# Das neuronale Netz trainieren

- Unser neuronales Netz soll lernen, dreistellige Binärzahlen in Dezimalzahlen umzuwandeln
- Binärzahlen bestehen nur aus den Ziffern 0 und 1, diese können direkt als Input-Aktivierungen verwendet werden



# Das neuronale Netz trainieren

- Auf dieselbe Weise können wir alle acht dreistelligen Binärzahlen in ihre Dezimaldarstellung umwandeln:

Binär	Dezimal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

# Das neuronale Netz trainieren

- Auf dieselbe Weise können wir alle acht dreistelligen Binärzahlen in ihre Dezimaldarstellung umwandeln:

Binär	Dezimal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Dies wird der Trainingsdatensatz für unser neuronales Netz sein.

**Aber:** Eine Zahl werden wir im Training bewusst weglassen – in der Hoffnung, dass unser Netz sie trotzdem lernt!

# Das neuronale Netz trainieren

- Auf dieselbe Weise können wir alle acht dreistelligen Binärzahlen in ihre Dezimaldarstellung umwandeln:

Binär	Dezimal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Dies wird der Trainingsdatensatz für unser neuronales Netz sein.

**Aber:** Eine Zahl werden wir im Training bewusst weglassen – in der Hoffnung, dass unser Netz sie trotzdem lernt!

Es handelt sich um ein **Regressionsproblem**. Wir brauchen im **Input-Layer** drei Neuronen und im **Output-Layer eins**.

Mit der Anzahl der Hidden Layer und ihrer Neuronen experimentieren wir.

# Binärumwandlung trainieren



```
if __name__ == '__main__':
    nn = NeuralNetwork(
        structure=[3, 4, 1],
        eta=0.01,
        n_iterations=3000,
        output_activation_func=relu,
        hidden_activation_func=relu
    )

    training_data = [
        (numpy.array([[0], [0], [0]]), numpy.array([[0]])),
        (numpy.array([[0], [0], [1]]), numpy.array([[1]])),
        (numpy.array([[0], [1], [0]]), numpy.array([[2]])),
        (numpy.array([[0], [1], [1]]), numpy.array([[3]])),
        (numpy.array([[1], [0], [0]]), numpy.array([[4]])),
        (numpy.array([[1], [1], [0]]), numpy.array([[6]])),
        (numpy.array([[1], [1], [1]]), numpy.array([[7]])),
    ]
    nn.train(training_data)
    input = numpy.array([[1], [0], [1]])
    prediction = nn.predict(input)

    print("Vorhersage für unbekannten Input", input, ":", prediction[0, 0])

    for X, _ in training_data:
        output = nn.predict(X)
        print("Vorhersage für bekannten Input", X, ":", output[0, 0])
```

Wir ersetzen unseren bestehenden Hauptprogrammblock durch die Konfiguration und das Training eines `NeuralNetwork`-Objekts mit unserem Binärzahlen-Datensatz.

# Binärumwandlung trainieren

```
● ● ●

if __name__ == '__main__':
    nn = NeuralNetwork(
        structure=[3, 4, 1],
        eta=0.01,
        n_iterations=3000,
        output_activation_func=relu,
        hidden_activation_func=relu
    )

    training_data = [
        (numpy.array([[0], [0], [0]]), numpy.array([[0]])),
        (numpy.array([[0], [0], [1]]), numpy.array([[1]])),
        (numpy.array([[0], [1], [0]]), numpy.array([[2]])),
        (numpy.array([[0], [1], [1]]), numpy.array([[3]])),
        (numpy.array([[1], [0], [0]]), numpy.array([[4]])),
        (numpy.array([[1], [1], [0]]), numpy.array([[6]])),
        (numpy.array([[1], [1], [1]]), numpy.array([[7]])),
    ]
    nn.train(training_data)
    input = numpy.array([[1], [0], [1]])
    prediction = nn.predict(input)

    print("Vorhersage für unbekannten Input", input, ":", prediction[0, 0])

    for X, _ in training_data:
        output = nn.predict(X)
        print("Vorhersage für bekannten Input", X, ":", output[0, 0])
```

Unser Netzwerk erzeugen wir anfangs mit nur einem Hidden Layer mit vier Neuronen. Sowohl Hidden als auch Output-Layer sollen als Aktivierungsfunktionen jeweils ReLU verwenden. Als Lernrate verwenden wir 0,01 und führen 3.000 Trainings-Iterationen durch.

# Binärumwandlung trainieren

```
● ● ●

if __name__ == '__main__':
    nn = NeuralNetwork(
        structure=[3, 4, 1],
        eta=0.01,
        n_iterations=3000,
        output_activation_func=relu,
        hidden_activation_func=relu
    )

    training_data = [
        (numpy.array([[0], [0], [0]]), numpy.array([[0]])),
        (numpy.array([[0], [0], [1]]), numpy.array([[1]])),
        (numpy.array([[0], [1], [0]]), numpy.array([[2]])),
        (numpy.array([[0], [1], [1]]), numpy.array([[3]])),
        (numpy.array([[1], [0], [0]]), numpy.array([[4]])),
        (numpy.array([[1], [1], [0]]), numpy.array([[6]])),
        (numpy.array([[1], [1], [1]]), numpy.array([[7]])),
    ]
    nn.train(training_data)
    input = numpy.array([[1], [0], [1]])
    prediction = nn.predict(input)

    print("Vorhersage für unbekannten Input", input, ":", prediction[0, 0])

    for X, _ in training_data:
        output = nn.predict(X)
        print("Vorhersage für bekannten Input", X, ":", output[0, 0])
```

Dann definieren wir unseren Trainingsdatensatz. Er ist eine Liste von Tupeln, die jeweils aus Input- und erwarteten Output-Aktivierungen besteht. Die Binärdarstellung der Zahl 5 (101) lassen wir dabei weg.

# Binärumwandlung trainieren

```
● ● ●

if __name__ == '__main__':
    nn = NeuralNetwork(
        structure=[3, 4, 1],
        eta=0.01,
        n_iterations=3000,
        output_activation_func=relu,
        hidden_activation_func=relu
    )

    training_data = [
        (numpy.array([[0], [0], [0]]), numpy.array([[0]])),
        (numpy.array([[0], [0], [1]]), numpy.array([[1]])),
        (numpy.array([[0], [1], [0]]), numpy.array([[2]])),
        (numpy.array([[0], [1], [1]]), numpy.array([[3]])),
        (numpy.array([[1], [0], [0]]), numpy.array([[4]])),
        (numpy.array([[1], [1], [0]]), numpy.array([[6]])),
        (numpy.array([[1], [1], [1]]), numpy.array([[7]])),
    ]
    nn.train(training_data)
    input = numpy.array([[1], [0], [1]])
    prediction = nn.predict(input)

    print("Vorhersage für unbekannten Input", input, ":", prediction[0, 0])

    for X, _ in training_data:
        output = nn.predict(X)
        print("Vorhersage für bekannten Input", X, ":", output[0, 0])
```

Wir müssen die Listen jeweils zweidimensional übergeben, damit numpy eindeutige Informationen über die Form der Vektoren hat. `numpy.array` erzeugt eine Matrix und erwartet die Elemente zeilenweise. Unser Netzwerk erwartet die Input- und Output-Vektoren als Zeilenvektoren.

# Binärumwandlung trainieren

```
● ● ●

if __name__ == '__main__':
    nn = NeuralNetwork(
        structure=[3, 4, 1],
        eta=0.01,
        n_iterations=3000,
        output_activation_func=relu,
        hidden_activation_func=relu
    )

    training_data = [
        (numpy.array([[0], [0], [0]]), numpy.array([[0]])),
        (numpy.array([[0], [0], [1]]), numpy.array([[1]])),
        (numpy.array([[0], [1], [0]]), numpy.array([[2]])),
        (numpy.array([[0], [1], [1]]), numpy.array([[3]])),
        (numpy.array([[1], [0], [0]]), numpy.array([[4]])),
        (numpy.array([[1], [1], [0]]), numpy.array([[6]])),
        (numpy.array([[1], [1], [1]]), numpy.array([[7]])),
    ]
    nn.train(training_data)
    input = numpy.array([[1], [0], [1]])
    prediction = nn.predict(input)

    print("Vorhersage für unbekannten Input", input, ":", prediction[0, 0])

    for X, _ in training_data:
        output = nn.predict(X)
        print("Vorhersage für bekannten Input", X, ":", output[0, 0])
```

Mit dem Trainingsdatensatz können wir unsere soeben implementierte **fit**-Methode aufrufen und damit das eigentliche Training durchführen.

# Binärumwandlung trainieren

```
● ● ●

if __name__ == '__main__':
    nn = NeuralNetwork(
        structure=[3, 4, 1],
        eta=0.01,
        n_iterations=3000,
        output_activation_func=relu,
        hidden_activation_func=relu
    )

    training_data = [
        (numpy.array([[0], [0], [0]]), numpy.array([[0]])),
        (numpy.array([[0], [0], [1]]), numpy.array([[1]])),
        (numpy.array([[0], [1], [0]]), numpy.array([[2]])),
        (numpy.array([[0], [1], [1]]), numpy.array([[3]])),
        (numpy.array([[1], [0], [0]]), numpy.array([[4]])),
        (numpy.array([[1], [1], [0]]), numpy.array([[6]])),
        (numpy.array([[1], [1], [1]]), numpy.array([[7]])),
    ]
    nn.train(training_data)
    input = numpy.array([[1], [0], [1]])
    prediction = nn.predict(input)

    print("Vorhersage für unbekannten Input", input, ":", prediction[0, 0])

    for X, _ in training_data:
        output = nn.predict(X)
        print("Vorhersage für bekannten Input", X, ":", output[0, 0])
```

Wir erzeugen einen Input-Datensatz  
für unsere in den Trainingsdaten  
fehlende Binärkombination ...

# Binärumwandlung trainieren

```
● ● ●
```

```
if __name__ == '__main__':
    nn = NeuralNetwork(
        structure=[3, 4, 1],
        eta=0.01,
        n_iterations=3000,
        output_activation_func=relu,
        hidden_activation_func=relu
    )
```

```
training_data = [
    (numpy.array([[0], [0], [0]]), numpy.array([[0]])),
    (numpy.array([[0], [0], [1]]), numpy.array([[1]])),
    (numpy.array([[0], [1], [0]]), numpy.array([[2]])),
    (numpy.array([[0], [1], [1]]), numpy.array([[3]])),
    (numpy.array([[1], [0], [0]]), numpy.array([[4]])),
    (numpy.array([[1], [1], [0]]), numpy.array([[6]])),
    (numpy.array([[1], [1], [1]]), numpy.array([[7]])),
]
```

```
nn.train(training_data)
input = numpy.array([[1], [0], [1]])
prediction = nn.predict(input)
```

```
print("Vorhersage für unbekannten Input", input, ":", prediction[0, 0])
```

```
for X, _ in training_data:
    output = nn.predict(X)
    print("Vorhersage für bekannten Input", X, ":", output[0, 0])
```

... und lassen unser frisch  
trainiertes Netz eine  
Vorhersage dafür treffen.

# Binärumwandlung trainieren



```
if __name__ == '__main__':
    nn = NeuralNetwork(
        structure=[3, 4, 1],
        eta=0.01,
        n_iterations=3000,
        output_activation_func=relu,
        hidden_activation_func=relu
    )

    training_data = [
        (numpy.array([[0], [0], [0]]), numpy.array([[0]])),
        (numpy.array([[0], [0], [1]]), numpy.array([[1]])),
        (numpy.array([[0], [1], [0]]), numpy.array([[2]])),
        (numpy.array([[0], [1], [1]]), numpy.array([[3]])),
        (numpy.array([[1], [0], [0]]), numpy.array([[4]])),
        (numpy.array([[1], [1], [0]]), numpy.array([[6]])),
        (numpy.array([[1], [1], [1]]), numpy.array([[7]])),
    ]
    nn.train(training_data)
    input = numpy.array([[1], [0], [1]])
    prediction = nn.predict(input)

    print("Vorhersage für unbekannten Input", input, ":", prediction[0, 0])

    for X, _ in training_data:
        output = nn.predict(X)
        print("Vorhersage für bekannten Input", X, ":", output[0, 0])
```

Diese Vorhersage geben wir auf der Konsole aus.

# Binärumwandlung trainieren

```
● ● ●

if __name__ == '__main__':
    nn = NeuralNetwork(
        structure=[3, 4, 1],
        eta=0.01,
        n_iterations=3000,
        output_activation_func=relu,
        hidden_activation_func=relu
    )

    training_data = [
        (numpy.array([[0], [0], [0]]), numpy.array([[0]])),
        (numpy.array([[0], [0], [1]]), numpy.array([[1]])),
        (numpy.array([[0], [1], [0]]), numpy.array([[2]])),
        (numpy.array([[0], [1], [1]]), numpy.array([[3]])),
        (numpy.array([[1], [0], [0]]), numpy.array([[4]])),
        (numpy.array([[1], [1], [0]]), numpy.array([[6]])),
        (numpy.array([[1], [1], [1]]), numpy.array([[7]])),
    ]
    nn.train(training_data)
    input = numpy.array([[1], [0], [1]])
    prediction = nn.predict(input)

    print("Vorhersage für unbekannten Input", input, ":", prediction[0, 0])

    for X, _ in training_data:
        output = nn.predict(X)
        print("Vorhersage für bekannten Input", X, ":", output[0, 0])
```

Auf dieselbe Weise erzeugen wir auch für die dem Netz bereits bekannten Input-Kombinationen Vorhersagen und geben sie auf der Konsole aus.

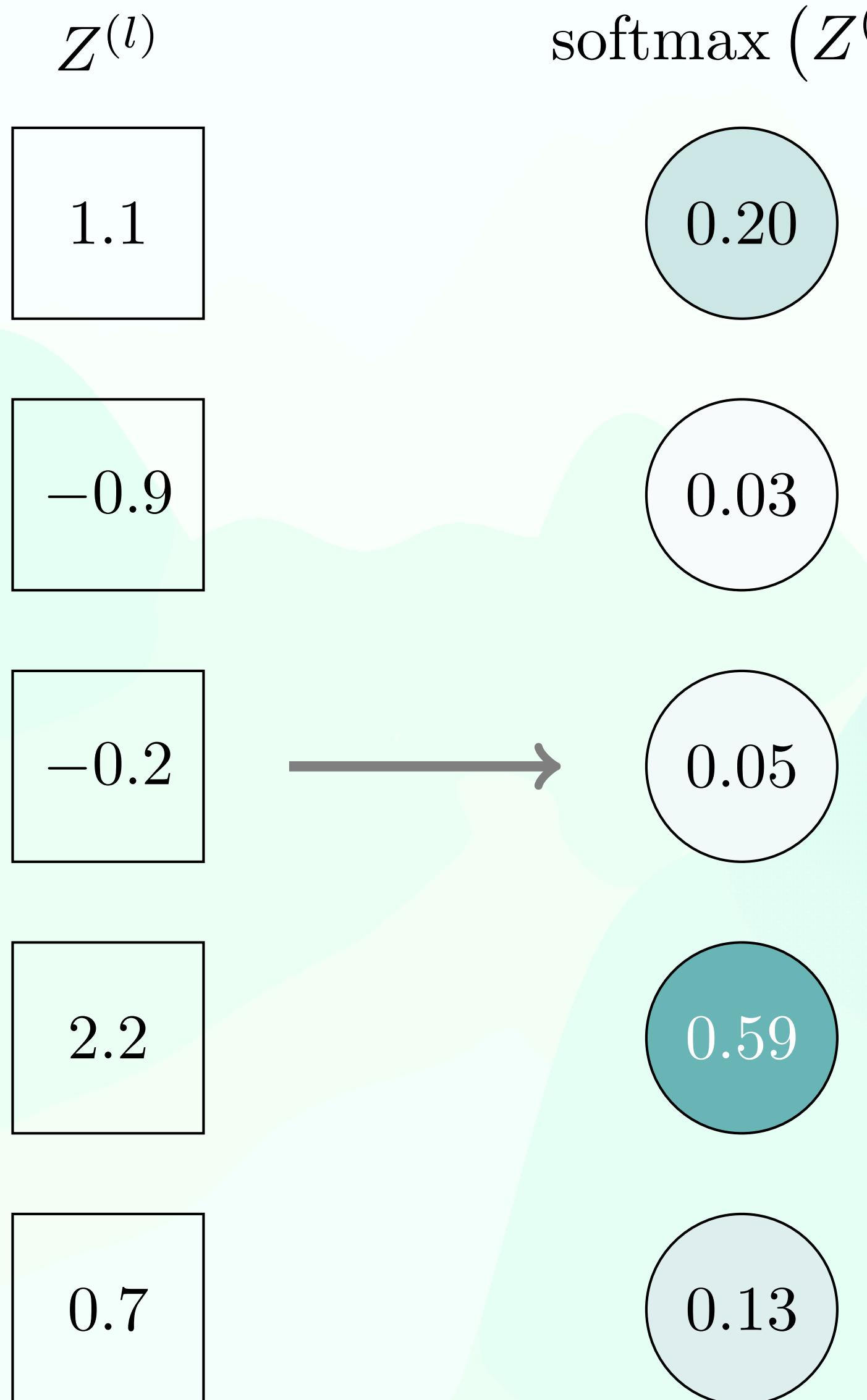
# Softmax für Klassifikationsnetzwerke

- Hilfreich für Klassifikation: Outputs als Wahrscheinlichkeiten interpretierbar
- Sigmoid liefert zwar Werte zwischen 0 und 1, aber ihre Summe ergibt nicht 1
- Lösung: **Softmax** als Aktivierungsfunktion im Output-Layer
- Sei  $n^{(l)}$  die Anzahl der Neuronen in Layer  $l$ :

$$a_i^{(l)} = \text{softmax}\left(z_i^{(l)}\right) = \frac{e^{z_i^{(l)}}}{\sum_{j=0}^{n^{(l)}-1} e^{z_j^{(l)}}}$$

- Softmax hängt also von allen gewichteten Summen des Layers ab!

# Softmax für Klassifikationsnetzwerke



$$\text{softmax}\left(z_i^{(l)}\right) = \frac{e^{z_i^{(l)}}}{\sum_{j=0}^{n^{(l)}-1} e^{z_j^{(l)}}}$$

Output-Aktivierungen ergeben in Summe 1, können damit wie Wahrscheinlichkeiten behandelt werden.

Größenrelationen der gewichteten Summen bleiben erhalten:

$$z_k^{(l)} > z_m^{(l)} \iff \text{softmax}\left(z_k^{(l)}\right) > \text{softmax}\left(z_m^{(l)}\right)$$

# Cross-Entropy Loss

- Passende Kostenfunktion zu Softmax: **Cross-Entropy Loss**
- Sei  $Y$  ein **One-Hot-kodierter** Vektor der erwarteten Output-Aktivierungen, dann betragen die Kosten nach Cross-Entropy Loss:

$$C = - \sum_{i=0}^{n^{(l)}-1} y_i \cdot \ln \left( a_i^{(l)} \right)$$

# Cross-Entropy Loss

- Passende Kostenfunktion zu Softmax: **Cross-Entropy Loss**
- Sei  $Y$  ein **One-Hot-kodierter** Vektor der erwarteten Output-Aktivierungen, dann betragen die Kosten nach Cross-Entropy Loss:

$$C = - \sum_{i=0}^{n^{(l)}-1} y_i \cdot \ln \left( a_i^{(l)} \right)$$

- Beispiel:

$$Y = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

„One-Hot“, alle anderen Einträge sind 0

# Cross-Entropy Loss

- Passende Kostenfunktion zu Softmax: **Cross-Entropy Loss**
- Sei  $Y$  ein **One-Hot-kodierter** Vektor der erwarteten Output-Aktivierungen, dann betragen die Kosten nach Cross-Entropy Loss:

$$C = - \sum_{i=0}^{n^{(l)}-1} y_i \cdot \ln(a_i^{(l)})$$

Nur die eine zugehörige Aktivierung im Output-Layer fließt in die Kosten ein!

- Beispiel:

$$Y = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

„One-Hot“, alle anderen Einträge sind 0

# Cross-Entropy Loss - Beispiel

- Gegeben:

$$Y = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad A^{(l)} = \begin{pmatrix} 0.1 \\ 0.7 \\ 0.2 \end{pmatrix}$$

$$C = - \sum_{i=0}^{n^{(l)}-1} y_i \cdot \ln \left( a_i^{(l)} \right)$$

# Cross-Entropy Loss - Beispiel

- Gegeben:

$$Y = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad A^{(l)} = \begin{pmatrix} 0.1 \\ 0.7 \\ 0.2 \end{pmatrix}$$

- Dann betragen die Kosten nach Cross-Entropy Loss:

$$C = - (0 \cdot \ln(0.1) + 1 \cdot \ln(0.7) + 0 \cdot \ln(0.2))$$

$$C = - \sum_{i=0}^{n^{(l)}-1} y_i \cdot \ln(a_i^{(l)})$$

# Cross-Entropy Loss - Beispiel

- Gegeben:

$$Y = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad A^{(l)} = \begin{pmatrix} 0.1 \\ 0.7 \\ 0.2 \end{pmatrix}$$

- Dann betragen die Kosten nach Cross-Entropy Loss:

$$\begin{aligned} C &= - (0 \cdot \ln(0.1) + 1 \cdot \ln(0.7) + 0 \cdot \ln(0.2)) \\ &= - \ln(0.7) \end{aligned}$$

$$C = - \sum_{i=0}^{n^{(l)}-1} y_i \cdot \ln(a_i^{(l)})$$

# Cross-Entropy Loss - Beispiel

- Gegeben:

$$Y = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad A^{(l)} = \begin{pmatrix} 0.1 \\ 0.7 \\ 0.2 \end{pmatrix}$$

- Dann betragen die Kosten nach Cross-Entropy Loss:

$$\begin{aligned} C &= - (0 \cdot \ln(0.1) + 1 \cdot \ln(0.7) + 0 \cdot \ln(0.2)) \\ &= - \ln(0.7) \\ &\approx 0.3567 \end{aligned}$$

$$C = - \sum_{i=0}^{n^{(l)}-1} y_i \cdot \ln(a_i^{(l)})$$

# Empfehlungen

# Empfehlungen

Exzellente Videoserie mit tollen Animationen: <http://bit.ly/4148KeC>

								Average over all training data ...	
$w_0$	-0.08	+0.02	-0.02	+0.11	-0.05	-0.14	...	→ -0.08	
$w_1$	-0.11	+0.11	+0.07	+0.02	+0.09	+0.05	...	→ +0.12	
$w_2$	-0.07	-0.04	-0.01	+0.02	+0.13	-0.15	...	→ -0.06	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
$w_{13,001}$	+0.13	+0.08	-0.06	-0.09	-0.02	+0.04	...	→ +0.04	

Neural networks

3Blue1Brown - 1/8

From the ground up | 18:40

But what is a neural network? | Deep learning chapter 1

3Blue1Brown

How machines learn | 20:33

Gradient descent, how neural networks learn | DL2

3Blue1Brown

Backpropagation | 12:47

Backpropagation, step-by-step | DL3

3Blue1Brown

Backpropagation calculus | 10:18

Backpropagation calculus | DL4

3Blue1Brown

# Empfehlungen

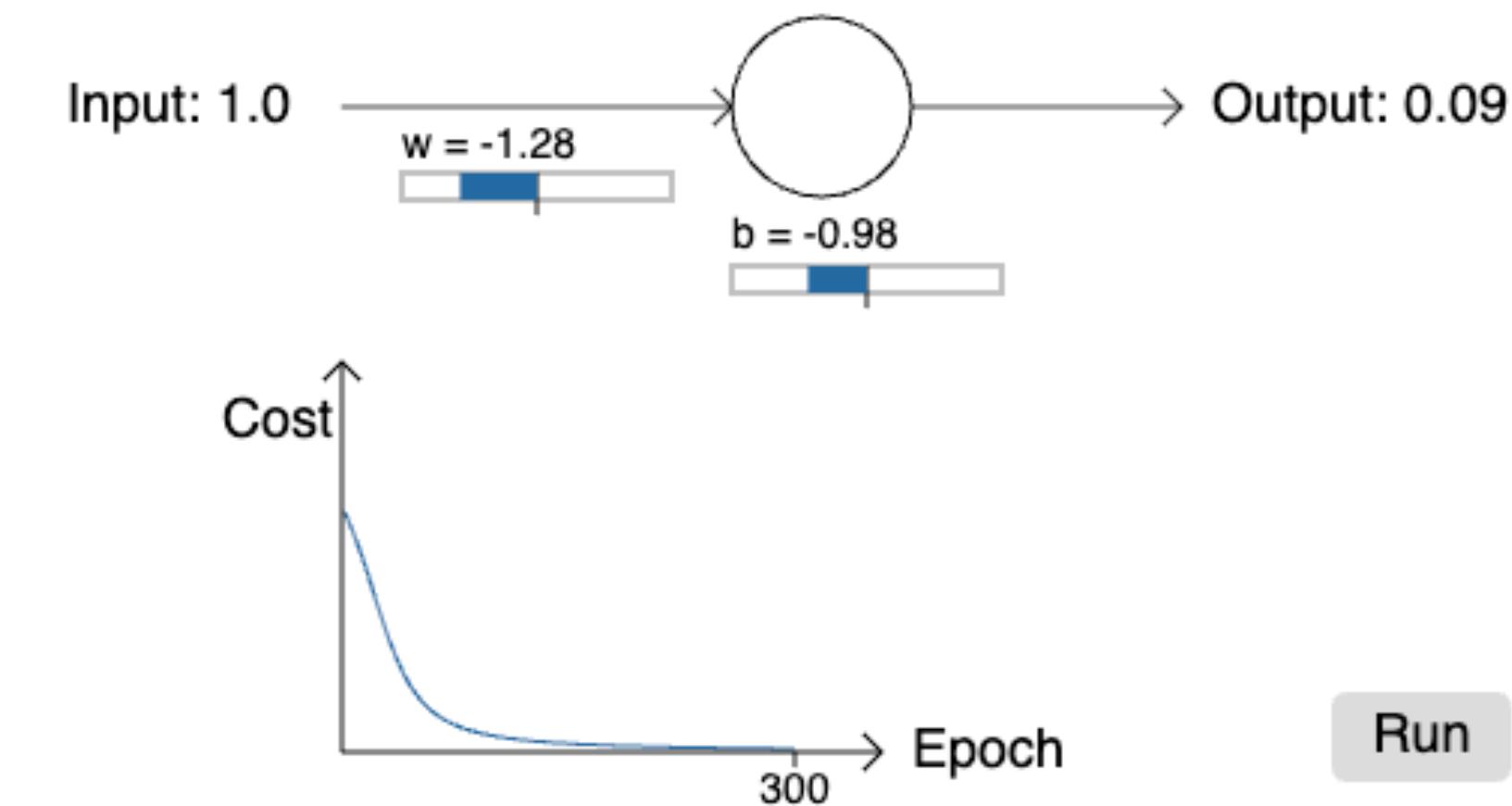
## Neural Networks and Deep Learning

Michael Nielsen

Kostenloses E-Book

<http://neuralnetworksanddeeplearning.com/>

be needed before our neuron gets near the desired output, 0.0. Click on "Run" in the bottom right corner below to see how the neuron learns an output much closer to 0.0. Note that this isn't a pre-recorded animation, your browser is actually computing the gradient, then using the gradient to update the weight and bias, and displaying the result. The learning rate is  $\eta = 0.15$ , which turns out to be slow enough that we can follow what's happening, but fast enough that we can get substantial learning in just a few seconds. The cost is the quadratic cost function,  $C$ , introduced back in Chapter 1. I'll remind you of the exact form of the cost function shortly, so there's no need to go and dig up the definition. Note that you can run the animation multiple times by clicking on "Run" again.



# Empfehlungen

## Neuronale Netze selbst programmieren

Tariq Rashid

dpunkt.verlag/O'Reilly 2017

ISBN: 9781 492 064 046

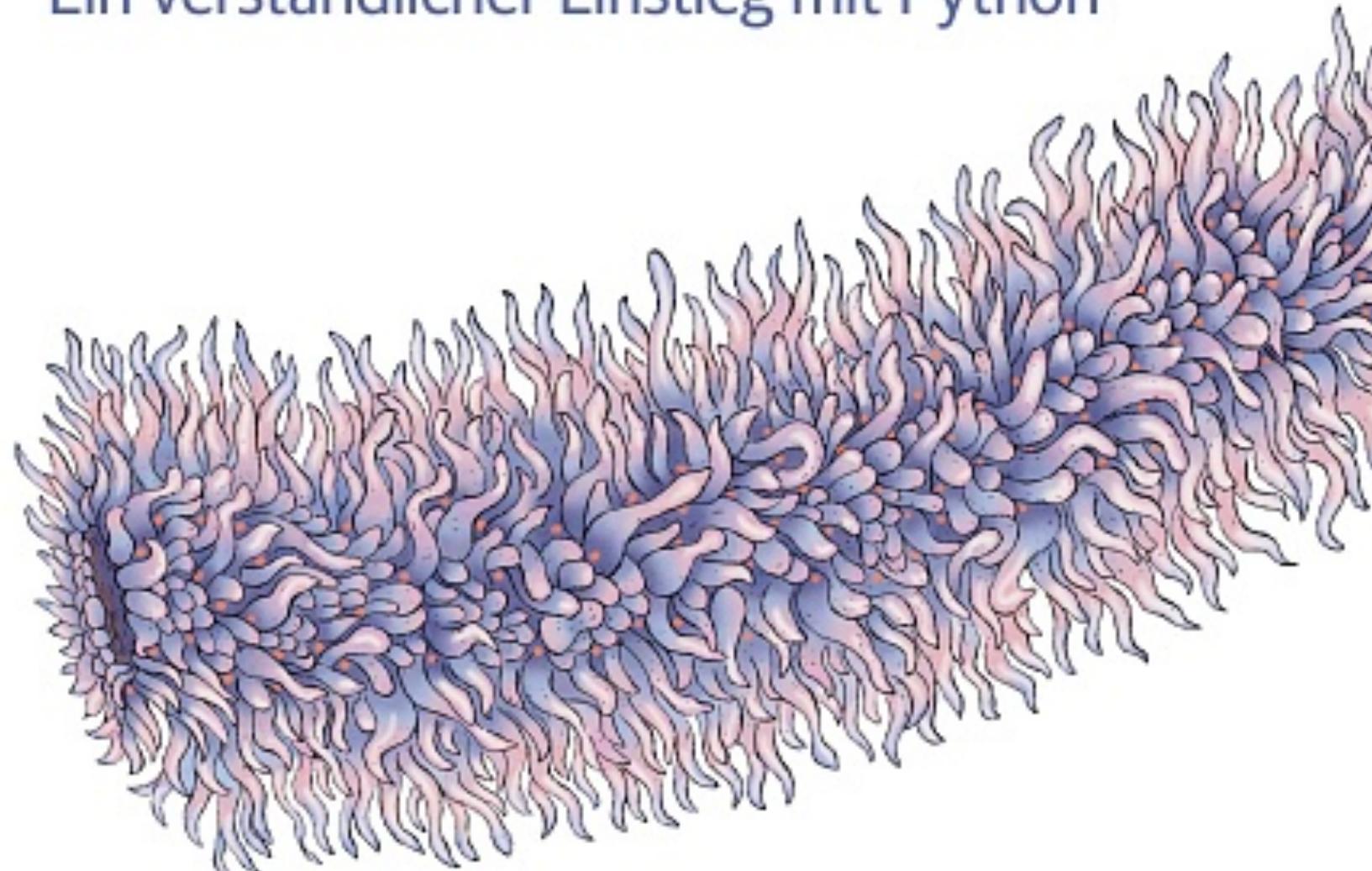
<https://bit.ly/3VuWvV5>

O'REILLY®

2. Auflage  
aktualisiert und  
erweitert

# Neuronale Netze selbst programmieren

Ein verständlicher Einstieg mit Python



Tariq Rashid

Übersetzung von Frank Langenau

# Empfehlungen – mit Einschränkungen

## Neuronale Netze programmieren mit Python

Jochen Steinwendner/Roland Schwaiger

Rheinwerk Computing 2020

ISBN: 9783 8362 7450 0

<https://bit.ly/4fHcFIY>

