

# Vorabfragen

- Ein neuronales Netz soll das XOR-Problem lösen. Warum könnte ein einzelnes Neuron dafür nicht ausreichen, und was müsste man am Modell ändern?
- Beim Training eines neuronalen Netzes wird eine Kostenfunktion minimiert. Was könnte passieren, wenn man zu lange trainiert, obwohl der Trainingsfehler weiter sinkt?
- Ein neuronales Netz erzielt sehr gute Ergebnisse auf Trainingsdaten, aber schlechte auf neuen Daten. Welche möglichen Ursachen und Gegenmaßnahmen fallen Ihnen ein?



# Deep Learning, Machine Learning und Künstliche Intelligenz – SS 26

Gelesen von Prof. Dr. Daniel Gaida

26.05.2026

Prof. Dr. Daniel Gaida

Professor für Cyber-Physische Systeme

Fakultät für Informatik und Ingenieurwissenschaften - Institut für Informatik

Seite 2

**Technology**  
**Arts Sciences**  
**TH Köln**

# Lernziele von heute und Fragen zur Überprüfung der Lernziele

Die Studierenden können ein Deep Learning Projekt selbstständig durchführen, indem sie

- für eine gegebene Anwendung ein geeignetes künstliches neuronales Netz auswählen können,
- dieses in keras implementieren und trainieren
- und bei schlechten Validierungsergebnissen Lösungsstrategien zur Verbesserung der Ergebnisse anwenden können,

um später eigene Deep Learning Projekte zur Bilderkennung, Zeitreihenanalyse, Textanalyse und -generierung, ... umsetzen zu können.

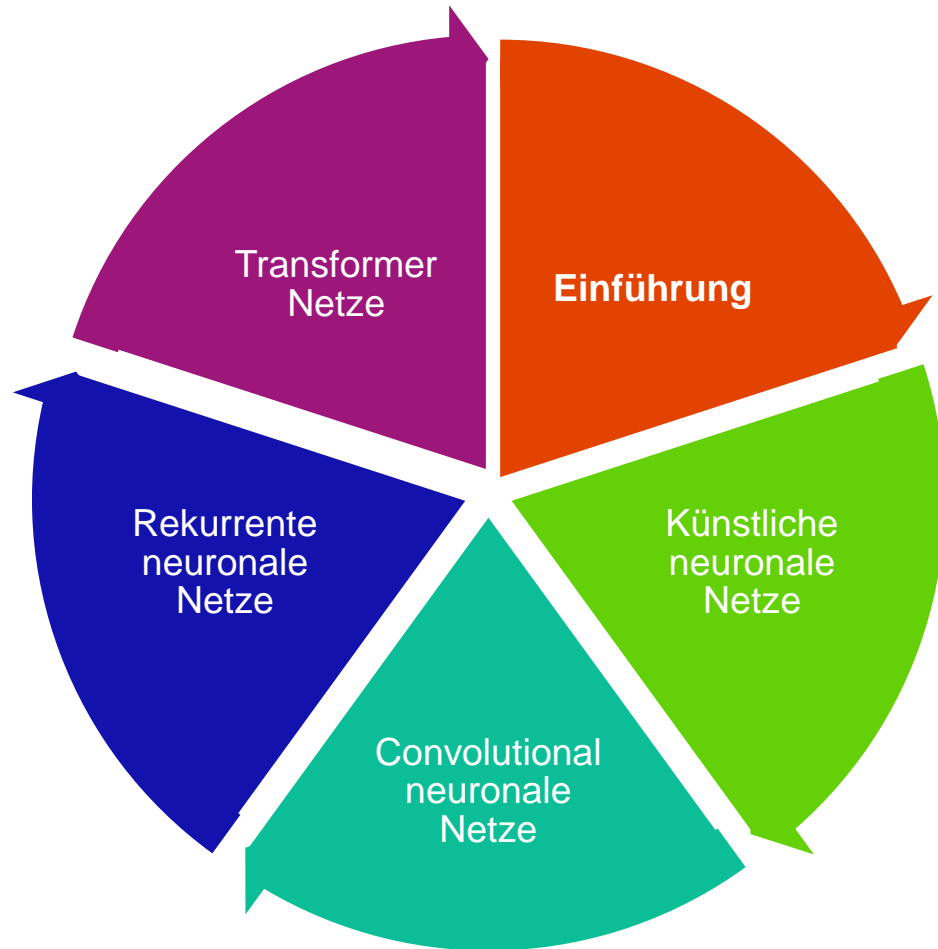
Überprüfung des Lernziels: S. fortlaufende Übung in diesem Lernraum II.

# Lernraum II: Deep Learning

- Künstliche neuronale Netze
  - Grundlagen
  - Training
  - Implementierung
  - Lösungsstrategien, um gute Ergebnisse zu erzielen
- Convolutional neuronale Netze
  - Bilderkennung
- Rekurrente neuronale Netze
  - Zeitreihen, Text
- Transformer Netze
  - Text, Bilderkennung
- Anwendungen
  - Entwickeln eines Chatbots
  - Objektdetektion und -segmentierung

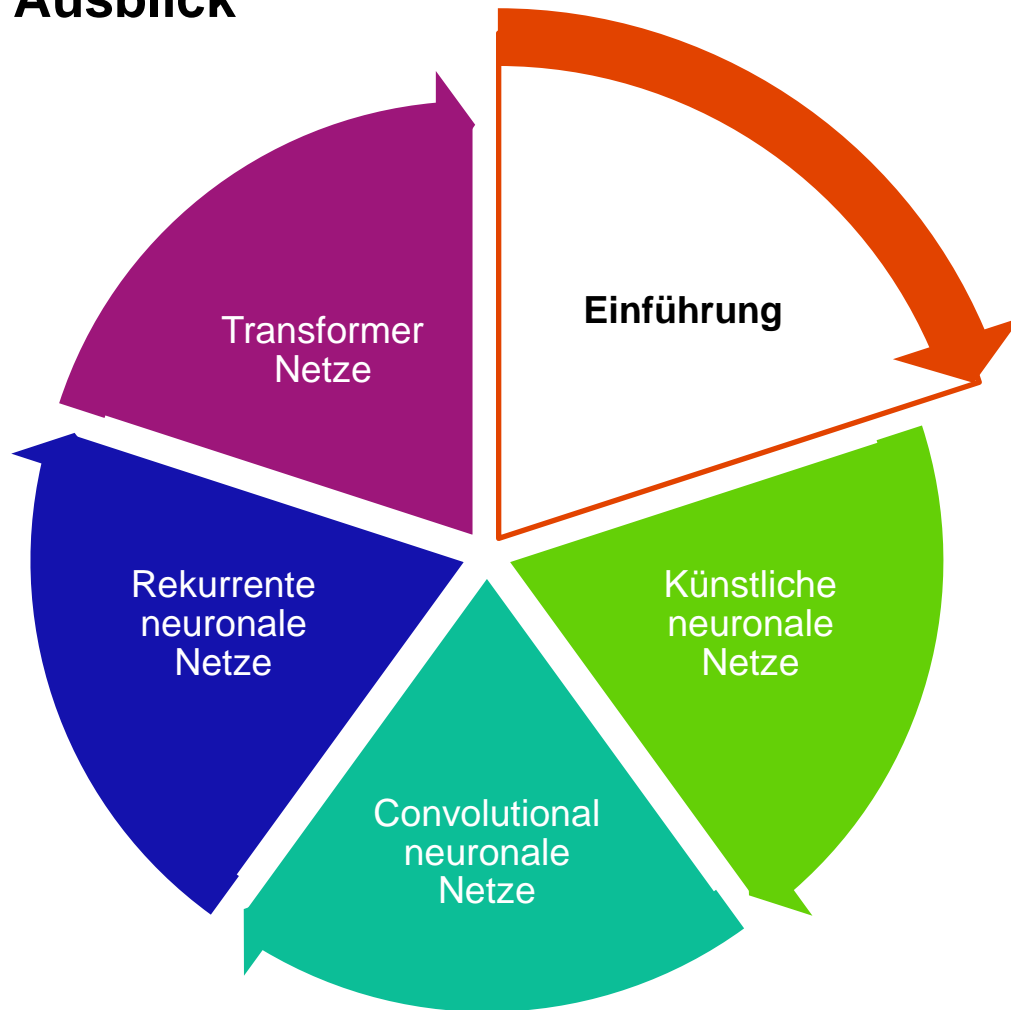
# Deep Learning

## Ausblick



# Deep Learning

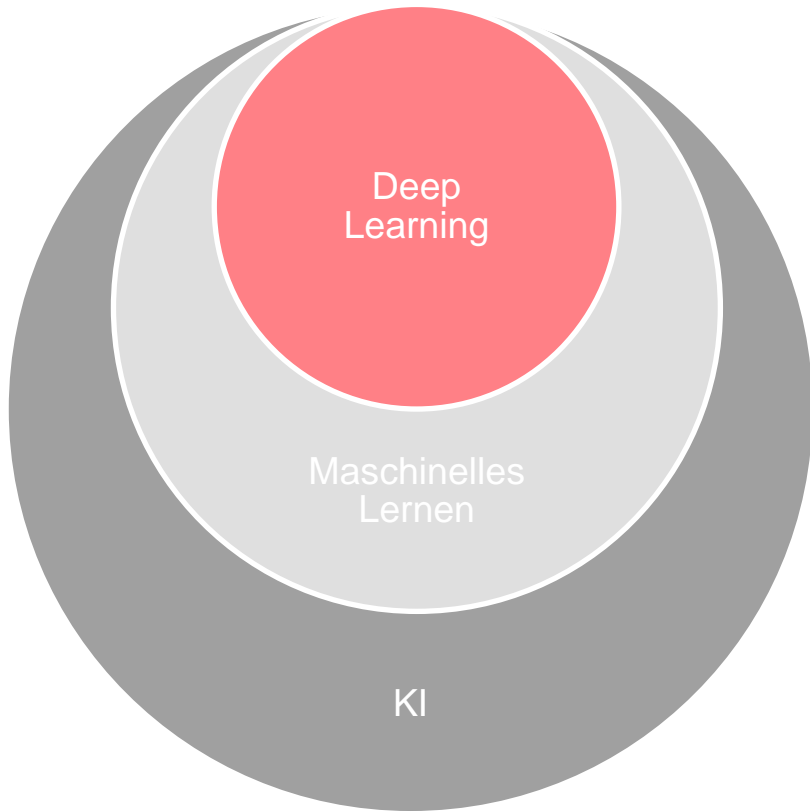
## Ausblick



### Einführung

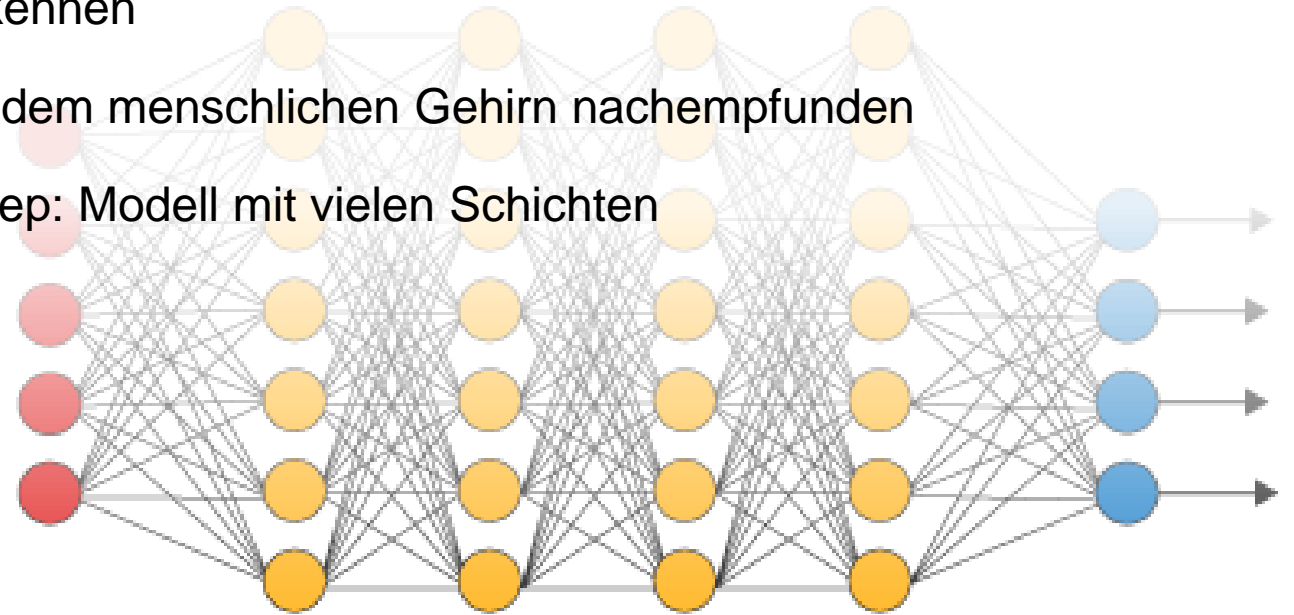
- Begriffserklärung
- Deep Learning
- Warum Deep Learning?

# KI – Maschinelles Lernen – Deep Learning



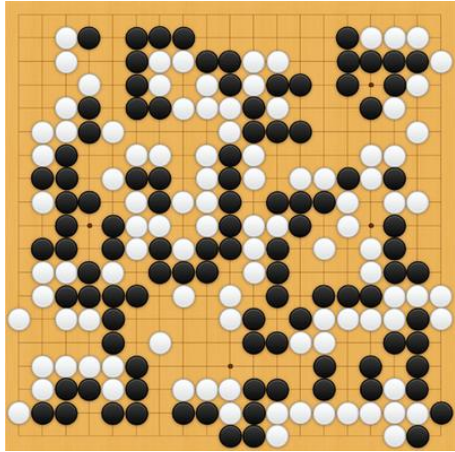
## Deep Learning

- Verwendet neuronale Netze, um Muster in Daten zu erkennen
- Ist dem menschlichen Gehirn nachempfunden
- Deep: Modell mit vielen Schichten



# Welche Möglichkeiten ergeben sich durch Deep Learning?

## Brettspiel Go



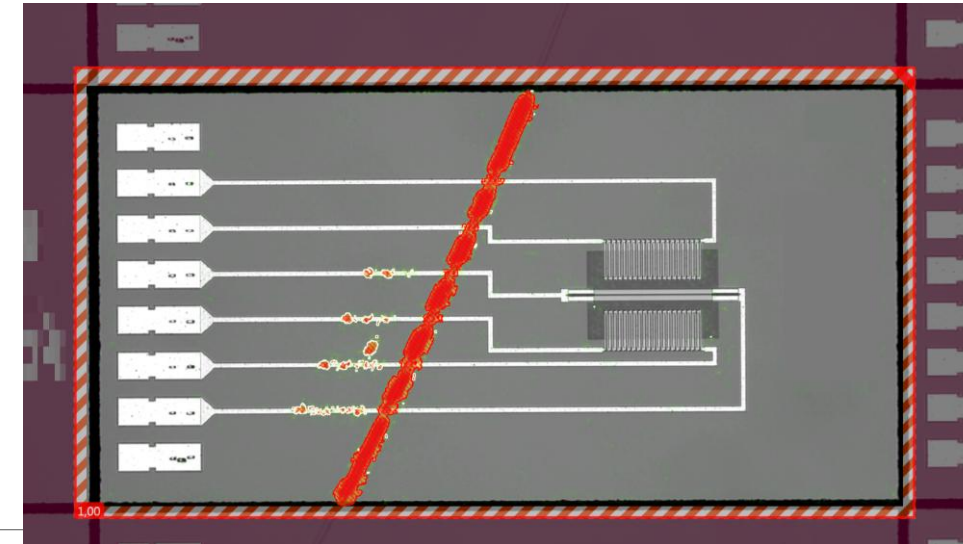
## Spracherkennung



## Übersetzen und Generieren von Texten



## Automatische Optische Inspektion



## Bilderkennung und -generierung



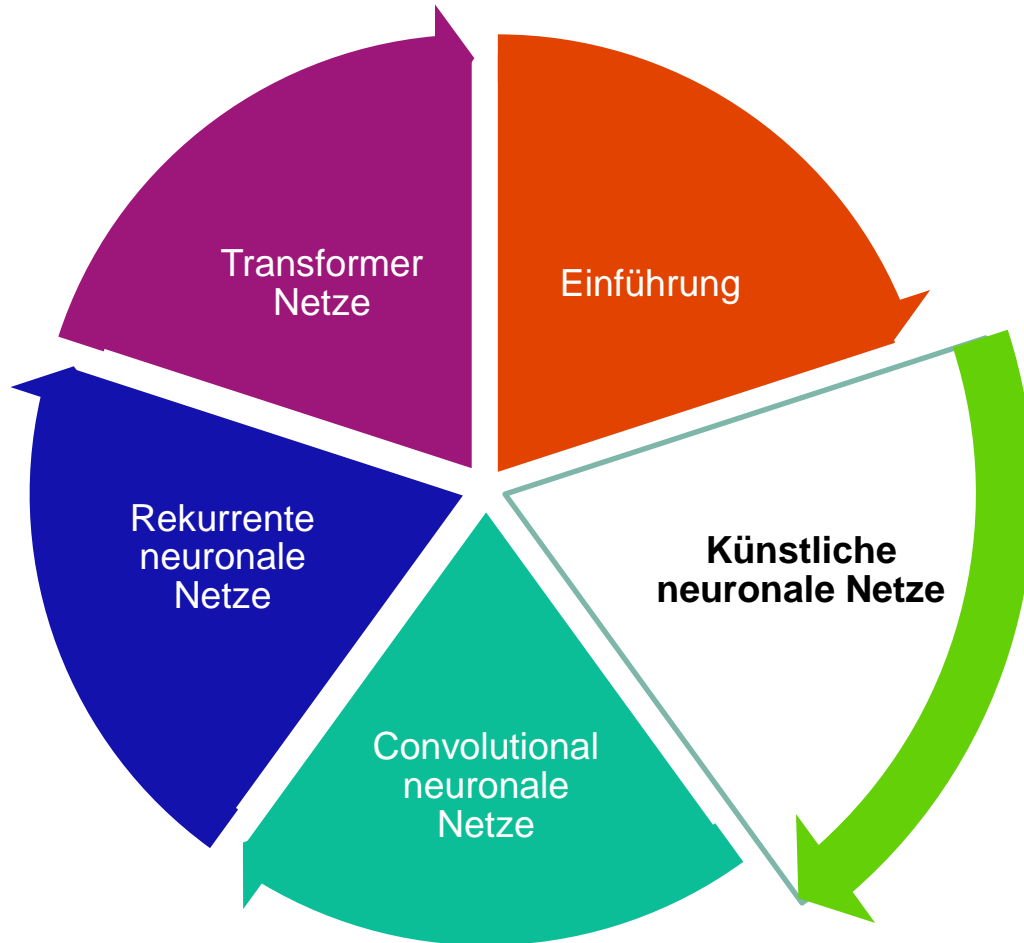
26.05.2026

Seite 8

Quelle: Patrick S., Hahn-Schickard

# Deep Learning

## Ausblick

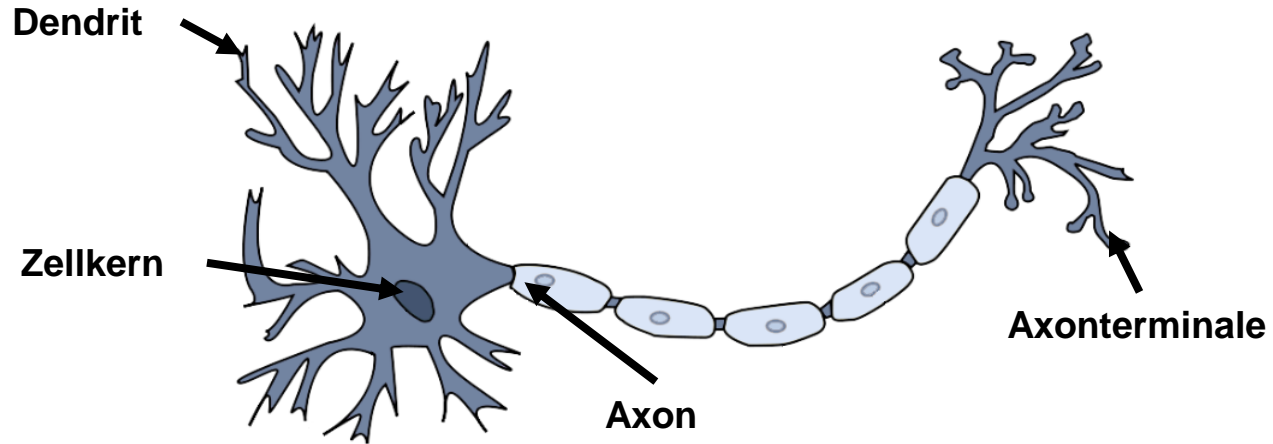


## Künstliche neuronale Netze: Part 1

- Einführung
  - Das biologische Neuron
  - Das künstliche Neuron
  - Das XOR Problem
- Training neuronaler Netze

# Künstliche Neuronale Netze - KNN

## Das künstliche Neuron

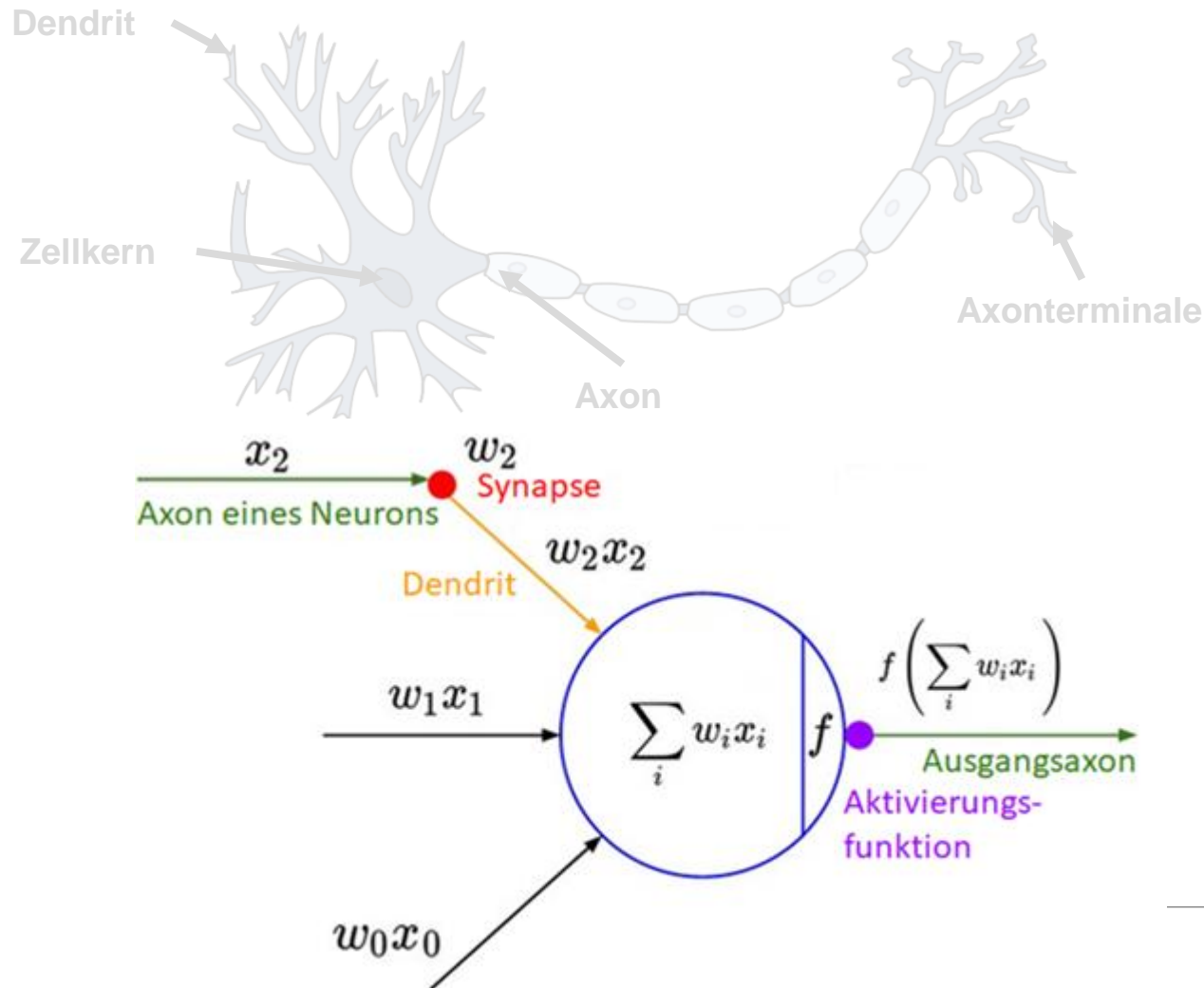


## Das biologische Neuron

- Elektrisch erregbare Zelle (Aufnahme, Verarbeitung und Weiterleitung von Informationen)
- Grundeinheit eines biologischen Gehirns
- Überträgt elektrische Signale von den Dendriten entlang des Axons zu den Terminalen
- Signale werden von Neuron zu Neuron übergeben
- Ausgangssignal entsteht nur, wenn die Eingabe ein Schwellwert überschreitet:
  - Das Neuron feuert
- Auf diese Weise nehmen wir Licht, Töne, Druck, Wärme, usw. wahr

# Künstliche Neuronale Netze - KNN

## Das künstliche Neuron

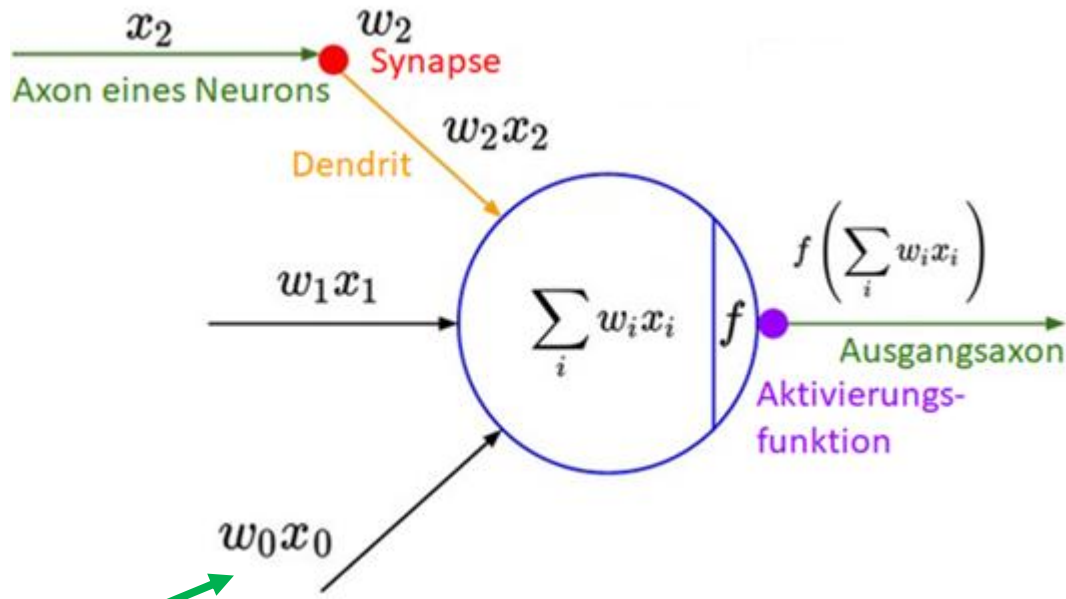


## Das künstliche Neuron - Perzeptron

- Grundeinheit eines neuronalen Netzes
- Ein- und Ausgänge sind Zahlen (anstatt binäre Ein/Aus Werte)
- Jede Eingangsverbindung ist mit einem Gewicht verbunden
- Berechnung der gewichteten Summe aller Eingänge
- Anwenden einer Aktivierungsfunktion
- Verknüpfung dieser Bausteine bilden künstliche Neuronale Netze

# Künstliche Neuronale Netze - KNN

## Das künstliche Neuron



$w_0$  wird als Bias bezeichnet.  $x_0$  ist kein Eingang des Neurons

Aktivierungsfunktion (Stufenfunktion, Signumfunktion):

$$f(z) = f(\mathbf{w}^T \cdot \mathbf{x}) = \begin{cases} 0 & \text{falls } \mathbf{w}^T \cdot \mathbf{x} < 0 \\ 1 & \text{sonst} \end{cases}$$

Vorwärtspropagation

- Gewichtete Summe:

$$z = \sum_{i=0}^n w_i \cdot x_i = \mathbf{w}^T \cdot \mathbf{x}$$

mit  $\mathbf{w} = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{pmatrix}$   $\mathbf{x} = \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{pmatrix}$

Bias ist im Gewichtsvektor enthalten

# Künstliche Neuronale Netze - KNN

## Das künstliche Neuron – Bias

Bias  $w_0$ :

- Analogie: y-Achsenabschnitt (hier: n)
- Bsp. für n = 1:
  - $z = w_0 + w_1 \cdot x_1$

$$\boldsymbol{w} = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{pmatrix} \quad \boldsymbol{x} = \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{pmatrix}$$

Bias ist im Gewichtsvektor enthalten

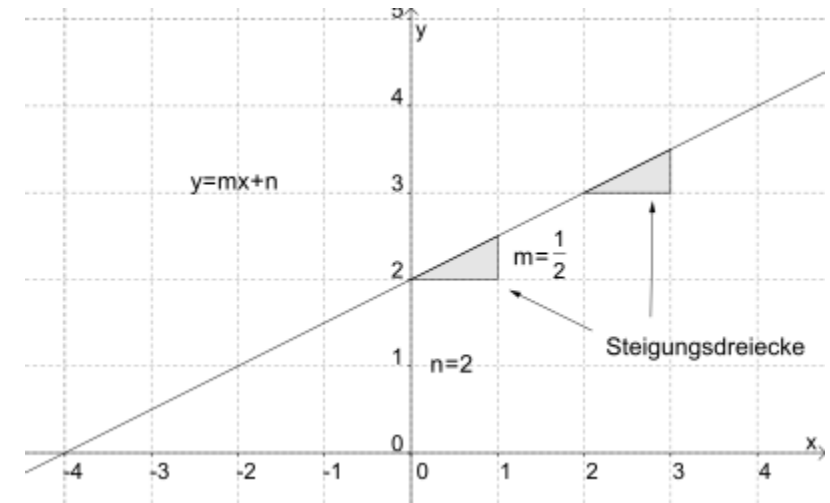


Bild: [https://de.wikipedia.org/wiki/Lineare\\_Funktion](https://de.wikipedia.org/wiki/Lineare_Funktion)

# Künstliche Neuronale Netze - KNN

## Das künstliche Neuron – Stufenfunktion

### Aktivierungsfunktion

- Stufenfunktion, Signumfunktion:

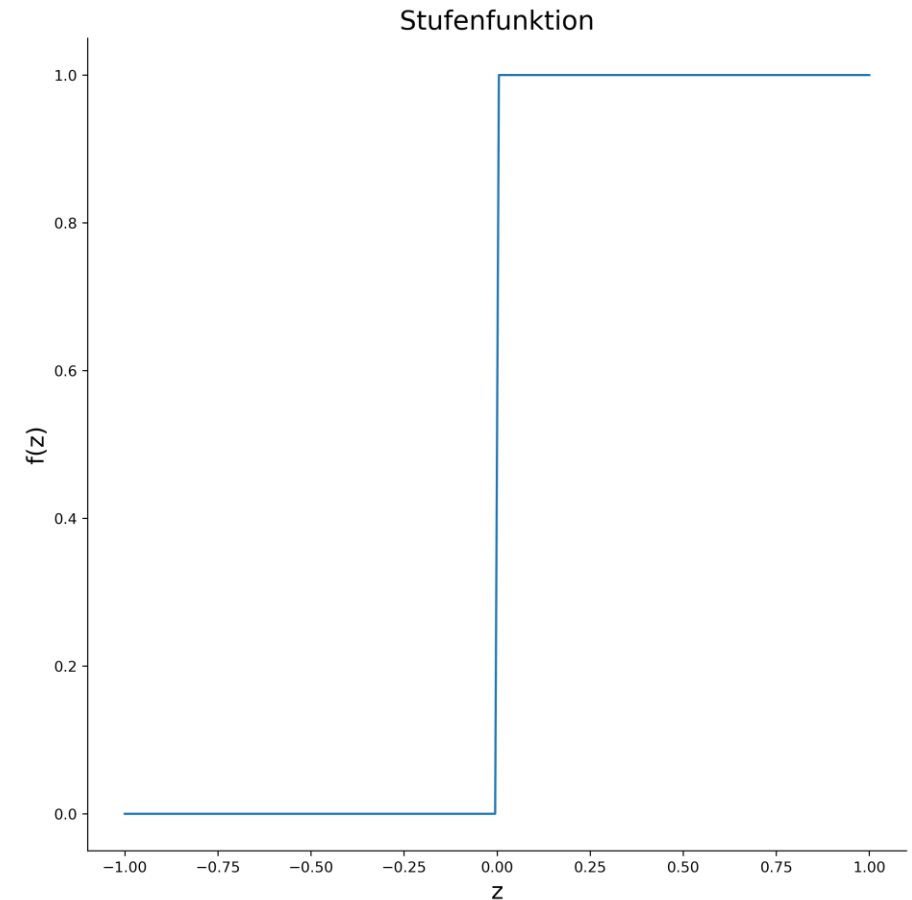
$$f(z) = f(\mathbf{w}^T \cdot \mathbf{x}) = \begin{cases} 0 & \text{falls } \mathbf{w}^T \cdot \mathbf{x} < 0 \\ 1 & \text{sonst} \end{cases}$$

- Bias  $w_0$  verschiebt den Schwellwert der Aktivierungsfunktion

Bsp. für  $n = 1$ :

$$z = w_0 + w_1 \cdot x_1 \geq 0$$

$$w_1 \cdot x_1 \geq -w_0$$



# Künstliche Neuronale Netze - KNN

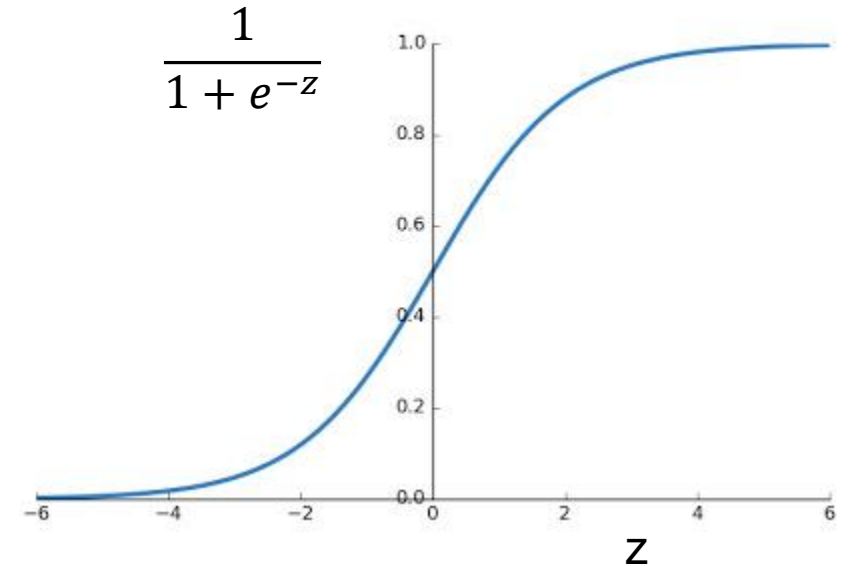
## Das künstliche Neuron – Sigmoid-Funktion

Aktivierungsfunktion

- Sigmoid-Funktion (logistisches Wachstum):

$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}}$$

- Vorteil: differenzierbar



# Künstliche Neuronale Netze - KNN

## Das künstliche Neuron – Kostenfunktion

### Kostenfunktion

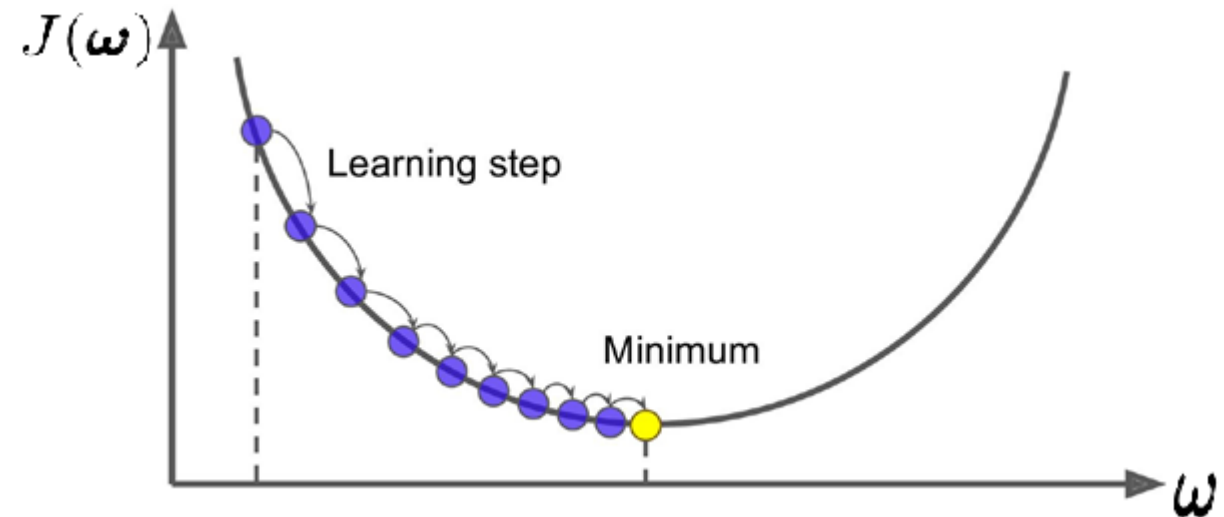
- Ziel des Neurons sei bspw. Eingangsdaten korrekt zu klassifizieren
- Kostenfunktion misst den Fehler zwischen der prognostizierten und der tatsächlichen Klasse
- Wir möchten diesen Fehler minimieren

# Künstliche Neuronale Netze - KNN

## Das künstliche Neuron – Gradientenabstieg

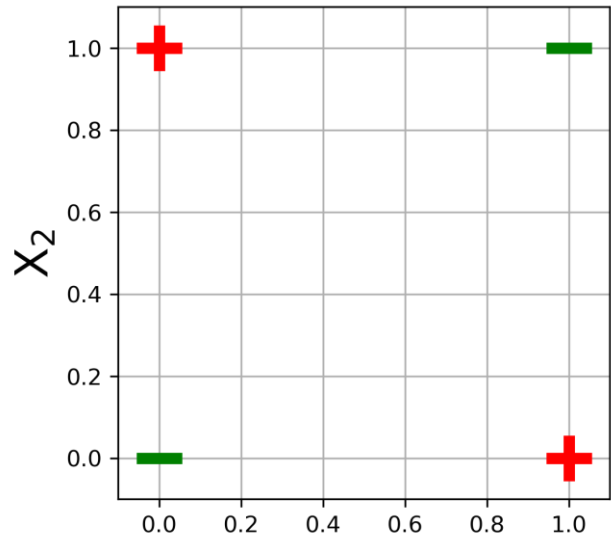
### Gradientenabstieg

- Iterativer Algorithmus, der angibt ob Gewichte ein bisschen kleiner oder größer gemacht werden müssen, um bessere Vorhersagen zu erzielen.
- Gewichte werden schrittweise so verändert, dass man in Richtung des steilsten Abstiegs der Kostenfunktion geht



# Künstliche Neuronale Netze - KNN

## Das XOR-Problem

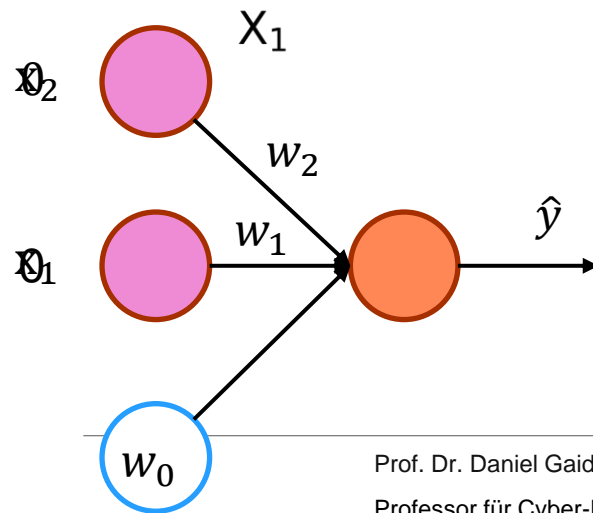


### Das XOR-Problem

- Beispiel für ein nicht-linear separierbares Problem

Daten erstellen

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \quad y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}$$



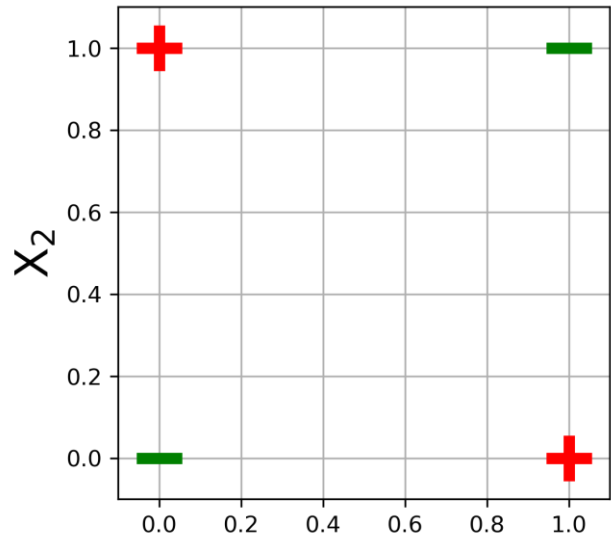
Eingabeschicht

Neuronen in der Eingabeschicht leiten Eingangsdaten weiter

- 2 Neuronen in der Eingabeschicht, da Eingangsdaten 2-dimensional
- 3. Dimension ist nur der Bias

# Künstliche Neuronale Netze - KNN

## Das XOR-Problem



### Das XOR-Problem

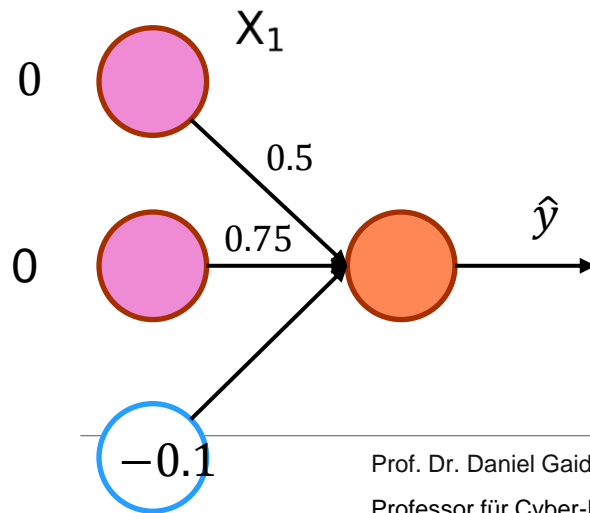
- Beispiel für ein nicht-linear separierbares Problem

Daten erstellen

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \quad y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}$$

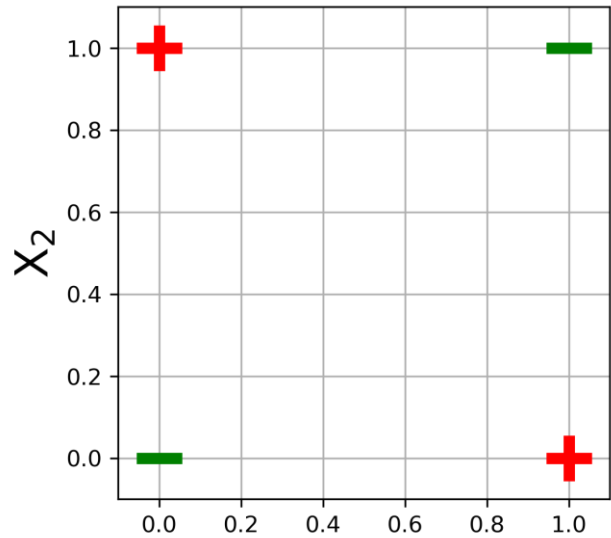
### 1. Gewichtsinitialisierung

$$w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} -0.1 \\ 0.75 \\ 0.5 \end{bmatrix}$$



# Künstliche Neuronale Netze - KNN

## Das XOR-Problem



### Das XOR-Problem

- Beispiel für ein nicht-linear separierbares Problem

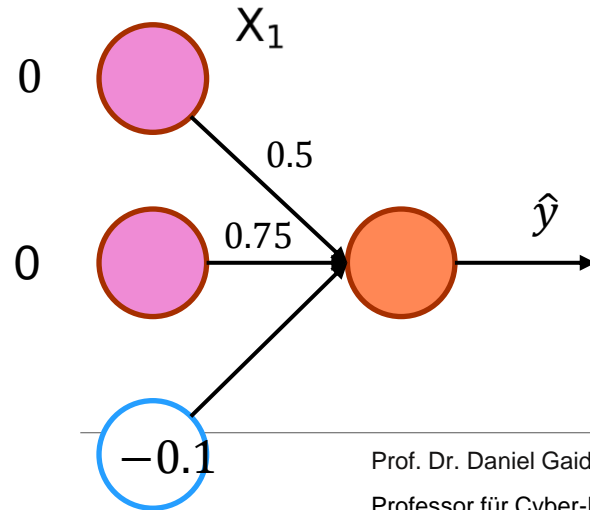
Daten erstellen

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

$$y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

$$w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}$$

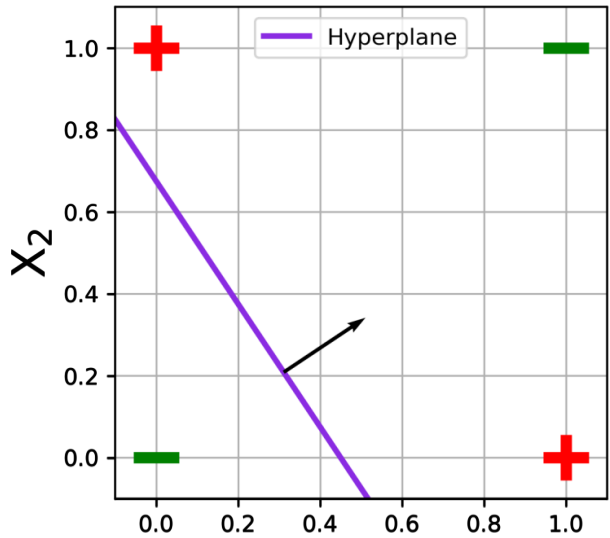
### 2. Forward Propagation



$$z = \mathbf{w}^T \cdot \mathbf{x} = [-0.1 \quad 0.75 \quad 0.5] \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = -0.1$$

# Künstliche Neuronale Netze - KNN

## Das XOR-Problem



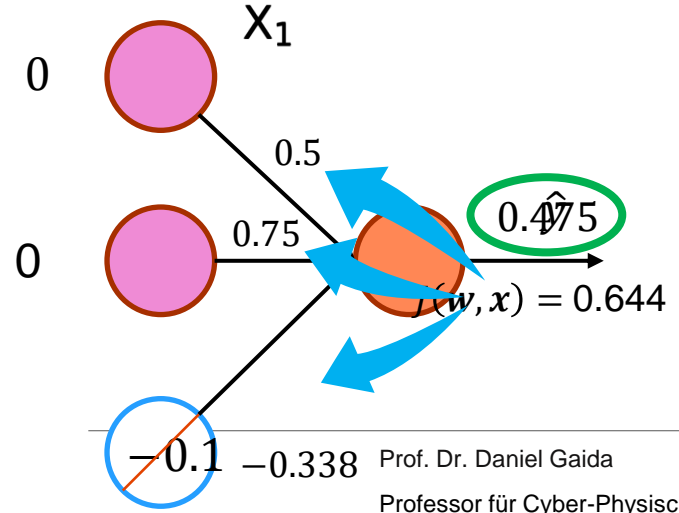
### Das XOR-Problem

- Beispiel für ein nicht-linear separierbares Problem



$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \quad y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}$$

3. Die Kostenfunktion (Gibner (Biquadrates Entropy) Step,  $\eta = \frac{1}{2}$ )

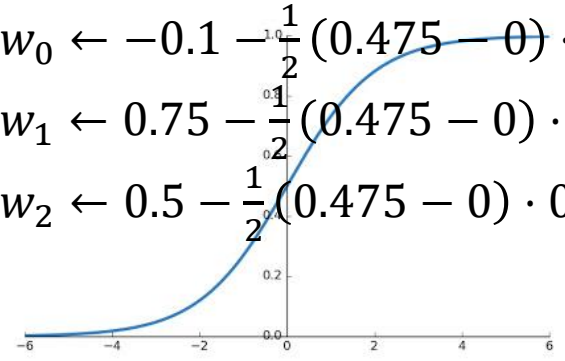


$$J(w, x) = -[y \log(\hat{y}) + (1-y) \log(1-\hat{y})]$$

$$\frac{\partial J}{\partial w_i} = \frac{1}{\eta} \frac{1}{1 + e^{-z}} - \frac{1}{\eta} \frac{1}{e^{-z} + 1} = \frac{1}{\eta} (\hat{y} - y) x_i$$

$$J(w, x) = -1 \log(0.475) \approx 0.644$$

- $w_0 \leftarrow -0.1 - \frac{1}{2} (0.475 - 0) \cdot 1 = -0.338$
- $w_1 \leftarrow 0.75 - \frac{1}{2} (0.475 - 0) \cdot 0 = 0.75$
- $w_2 \leftarrow 0.5 - \frac{1}{2} (0.475 - 0) \cdot 0 = 0.5$



# Künstliche Neuronale Netze - KNN

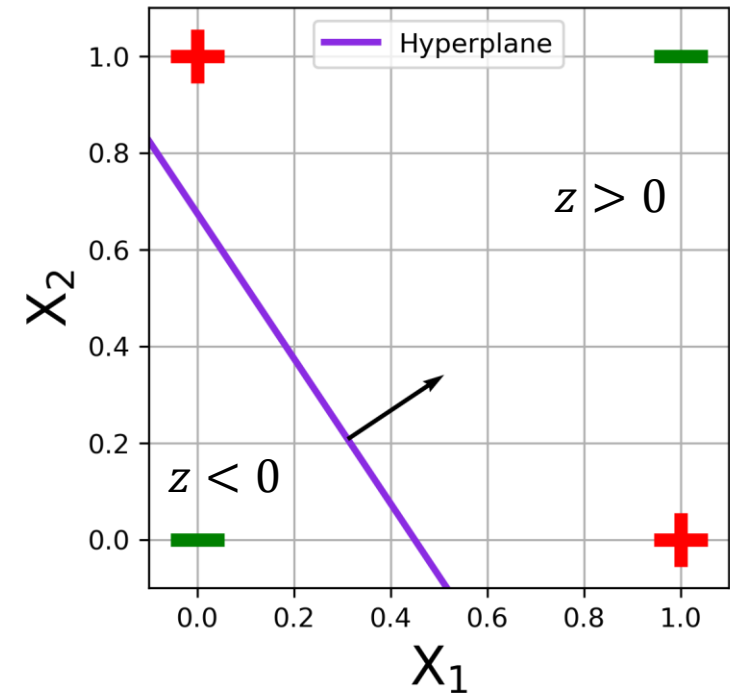
## Warum ist der Gewichtsvektor eine Hyperebene?

Ein Neuron führt eine lineare Transformation durch:

- Berechnet:  $z = w_1x_1 + w_2x_2 + w_0$
- Gewichte ( $w_1, w_2$ ) bestimmen die Neigungsrichtung.
- Bias ( $w_0$ ) verschiebt die Grenze.

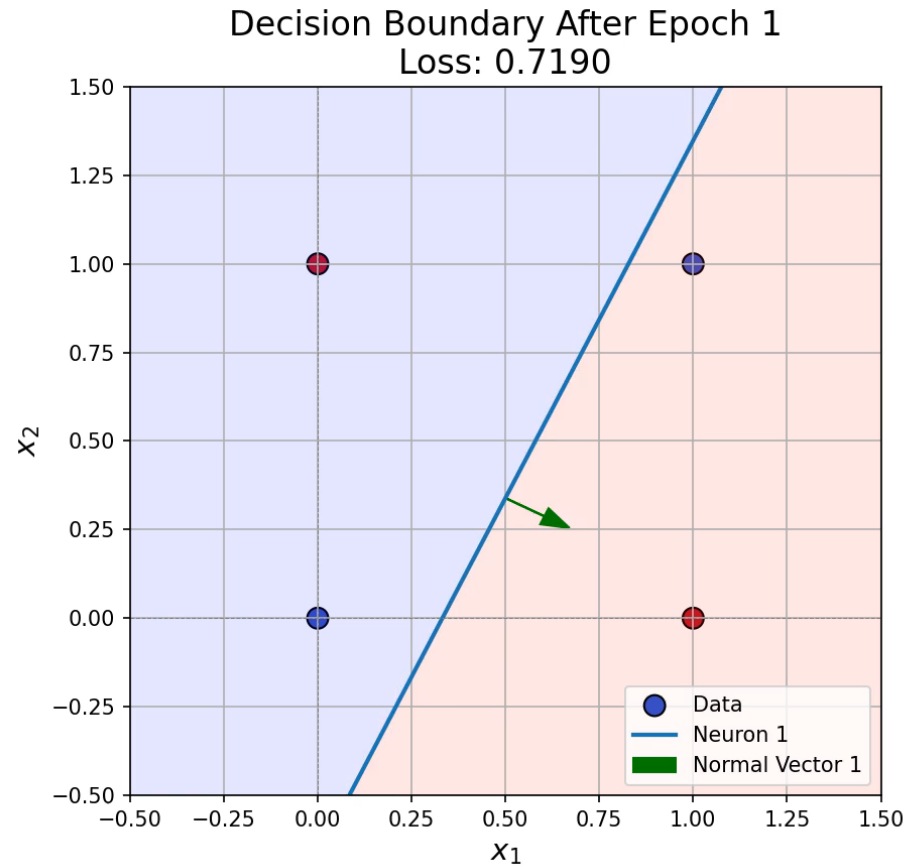
Entscheidungsgrenze = Nullstelle der Funktion

- Setze  $z = 0 \rightarrow z = w_1x_1 + w_2x_2 + w_0 = 0$
- $w_2x_2 = -w_1x_1 - w_0$
- Ergibt eine Gerade in 2D, eine Ebene in 3D, eine Hyperebene in höheren Dimensionen.

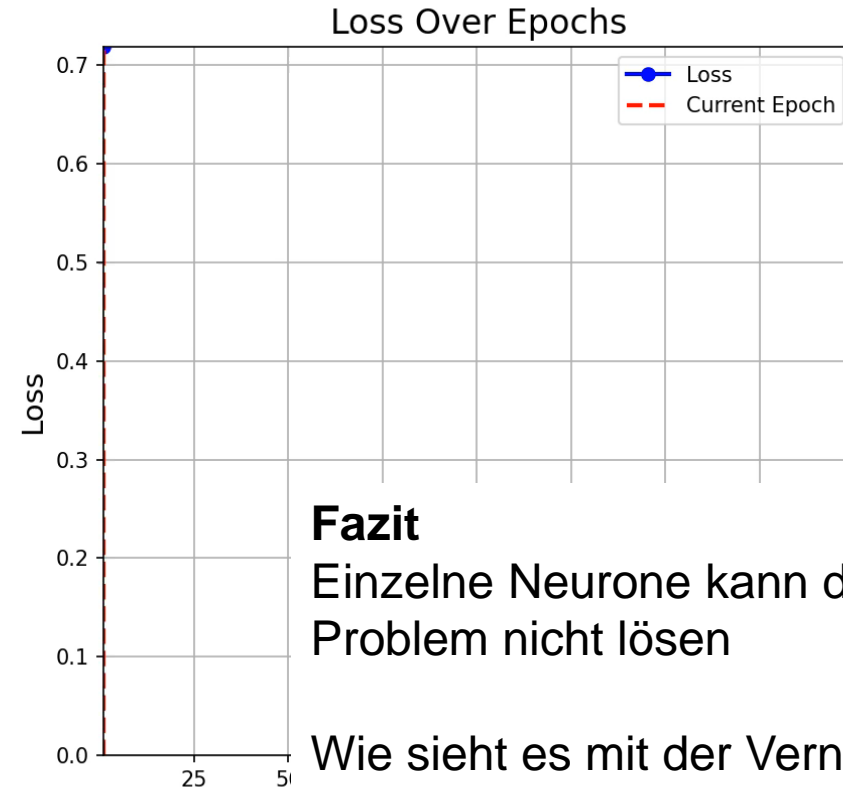


# Künstliche Neuronale Netze - KNN

## Das XOR-Problem – Ein Neuron



### Kostenfunktion



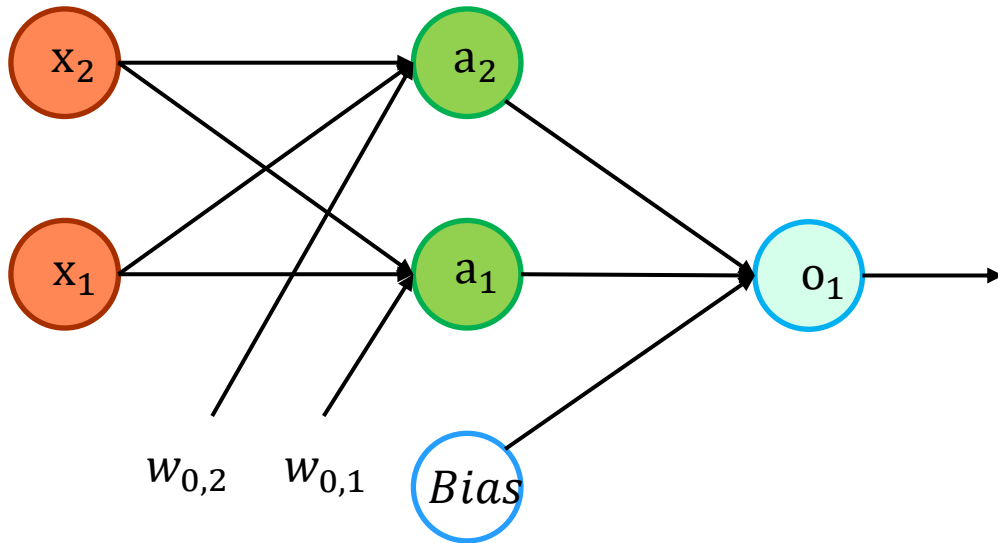
### Fazit

Einzelne Neurone kann das XOR-Problem nicht lösen

Wie sieht es mit der Vernetzung mehrerer künstlicher Neuronen aus?

# Künstliche Neuronale Netze - KNN

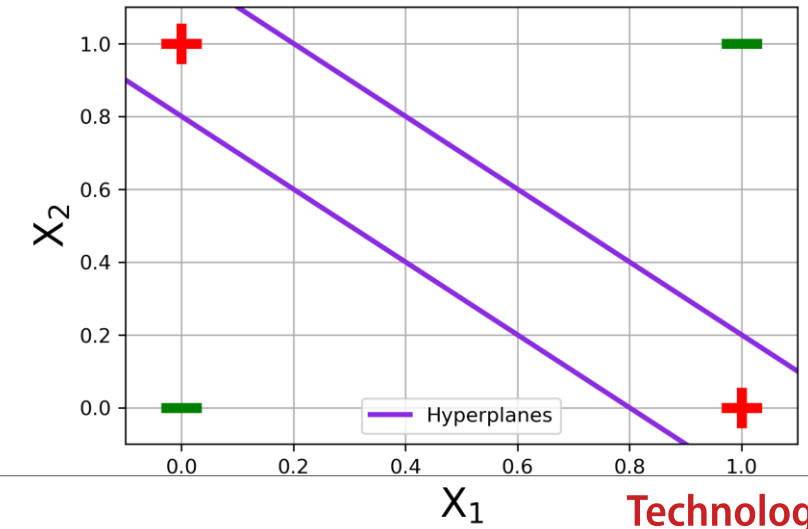
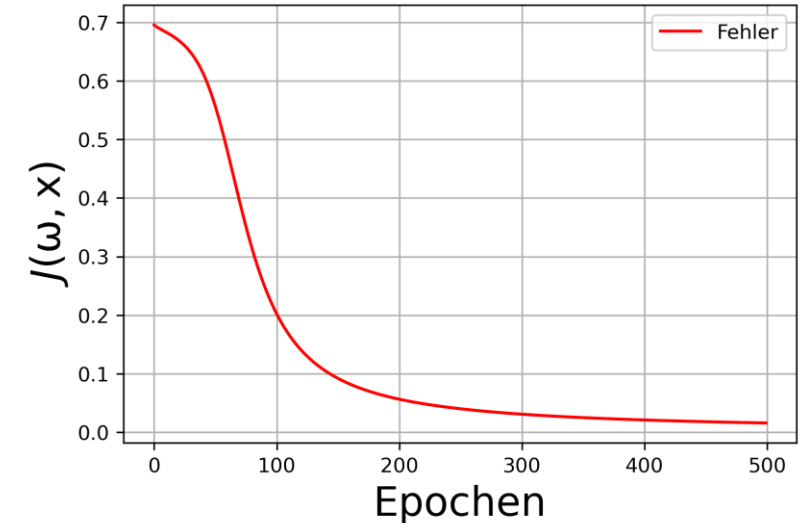
## Das XOR-Problem



Eingangsschicht      Verdeckte Schicht      Ausgangsschicht

Drei Neuronen:  $a_1, a_2, o_1$

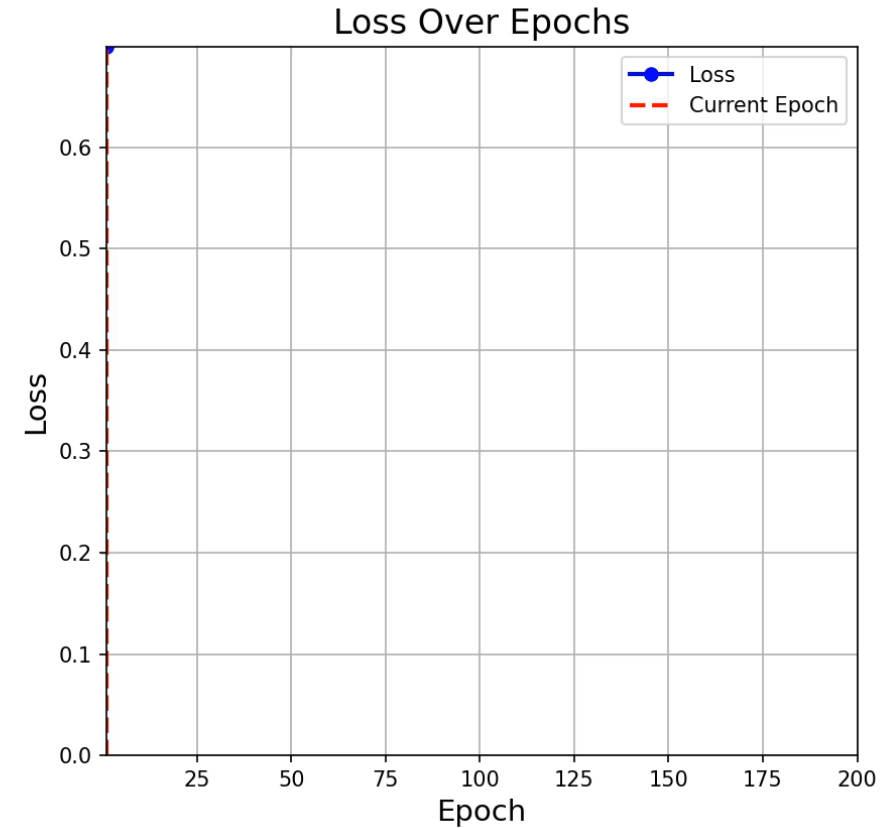
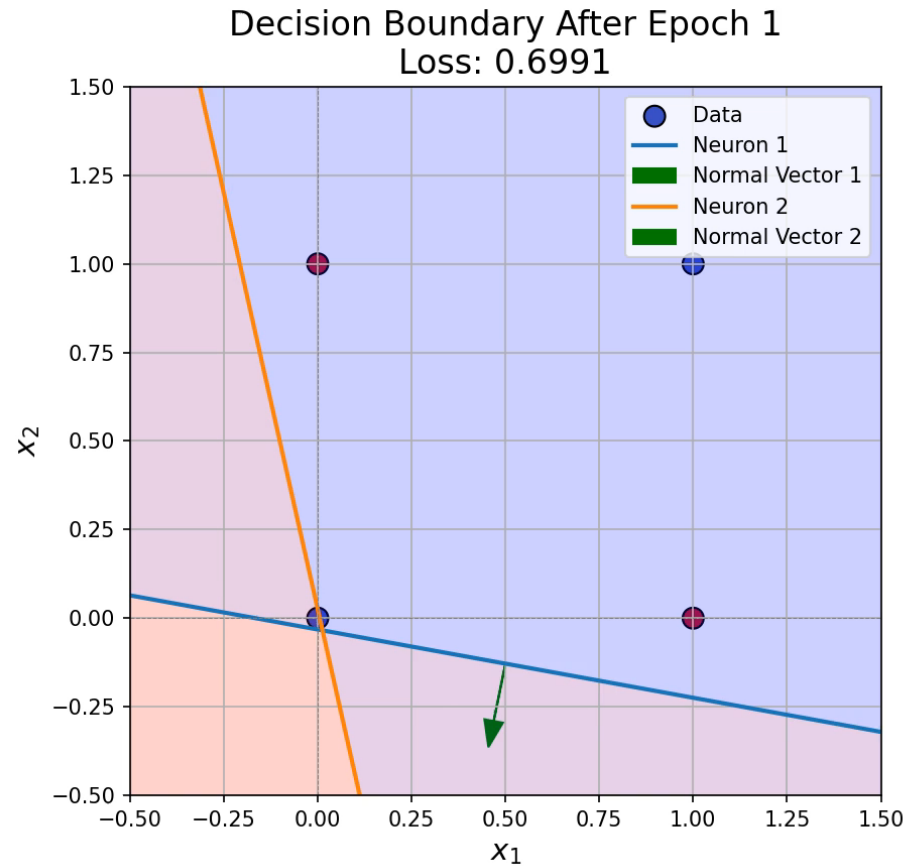
Erste verdeckte Schicht ( $a_1, a_2$ ) definiert zwei Hyperebenen



Aktivierung: sigmoid,  $\eta = 0.5$

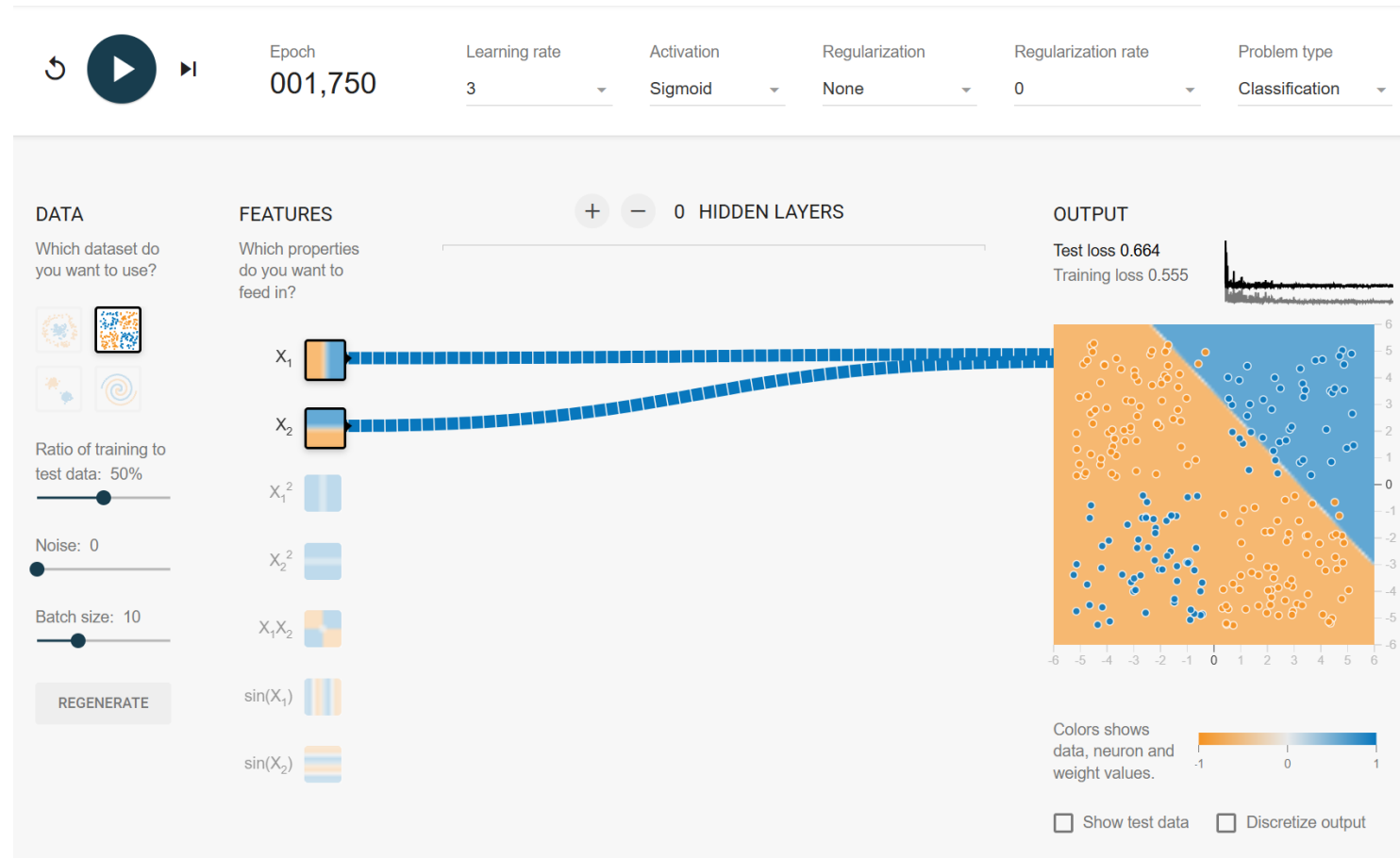
# Künstliche Neuronale Netze - KNN

## Das XOR-Problem



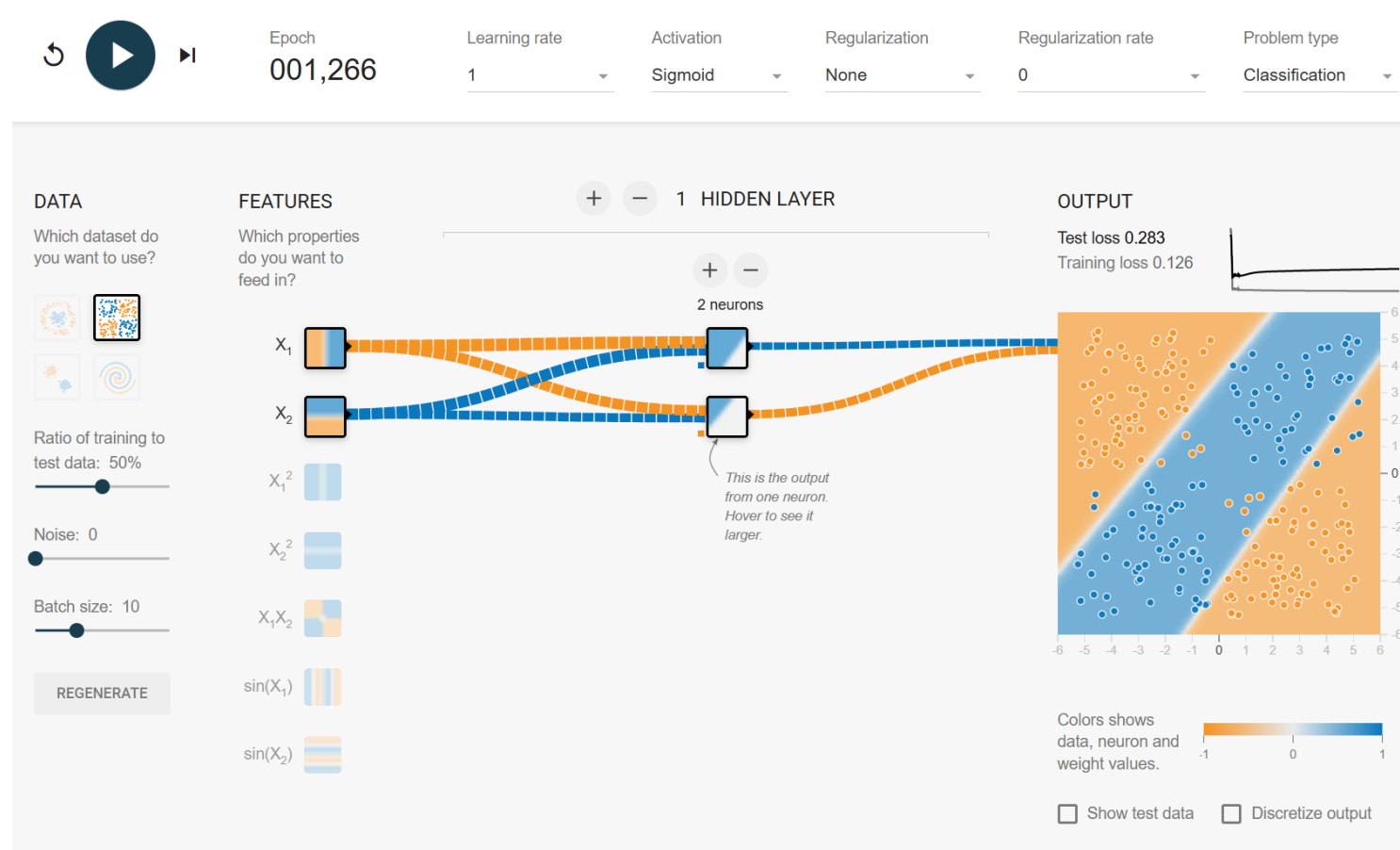
# Künstliche Neuronale Netze - KNN

Übung: Experimentieren Sie mit Google Playground: <https://playground.tensorflow.org/>



# Künstliche Neuronale Netze - KNN

Übung: Experimentieren Sie mit Google Playground: <https://playground.tensorflow.org/>



# Künstliche Neuronale Netze - KNN

## Das XOR-Problem – Training eines neuronalen Netzes

Training eines neuronalen Netzes:

1. Gewichtsinitialisierung

Für jeden Datenpunkt:

2. Forward Propagation
3. Aktivierung
4. Kostenfunktion (berechnen des Prognosefehlers)
5. Gewichtsaktualisierung (Gradient Descent Step)

Viele Begriffe eingeführt:

**Forward Propagation, Aktivierungsfunktion, Kostenfunktion, Gradient Descent**

# Künstliche Neuronale Netze - KNN

## Fragen I

Das Perzeptron (ein künstliches Neuron) feuert immer dann, wenn

- Alle Eingänge größer gleich 0 sind
- Falls mindestens ein Eingang kleiner 0 ist
- Summe der gewichteten Eingabe größer gleich 0 ist

Ein Perzeptron mit zwei Eingängen exkl. Bias löst folgende Probleme

- Linear separierbares Problem in einer zweidimensionalen Ebene
- Linear separierbares Problem in einer dreidimensionalen Ebene
- Nicht-linear separierbares Problem in einer zweidimensionalen Ebene (XOR-Problem)
- Keines der genannten Probleme

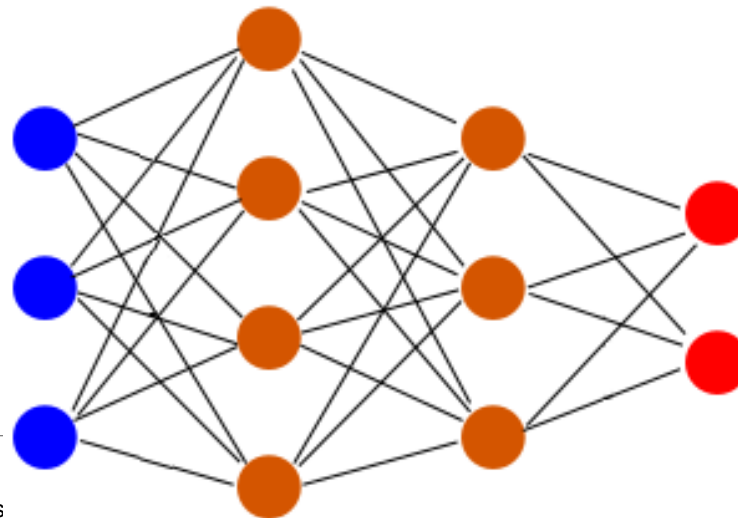
# Künstliche Neuronale Netze - KNN

## Problemstellung herkömmlicher Computer

Problem	Computer	Mensch
Tausende große Zahlen schnell multiplizieren	Leicht	Schwer
Gesichter auf einem Foto einer Menschenmenge suchen	Schwer	Leicht

→ KI: neue Algorithmen finden, die auf neuartige Weise versuchen, derartige Probleme zu lösen

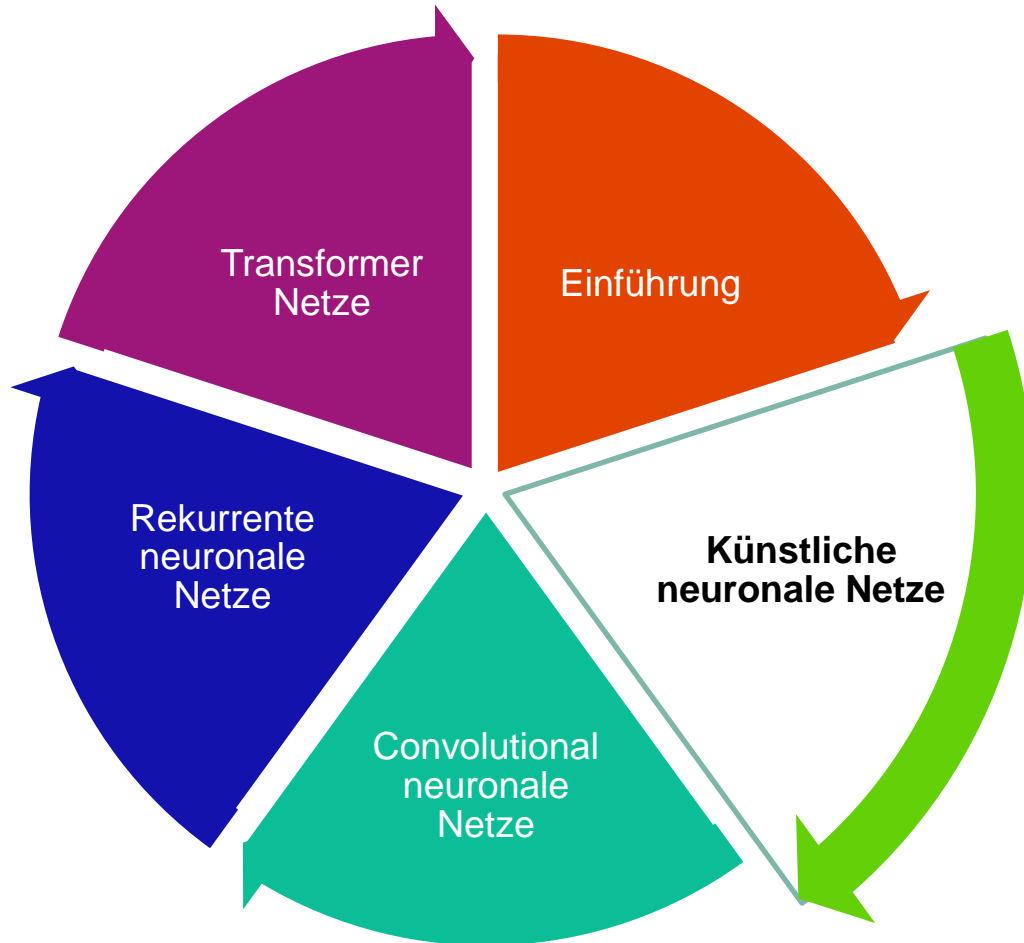
### Künstliches neuronales Netz



- Übernimmt eine Eingabe und liefert eine Ausgabe
- Inspiriert durch das menschliche Gehirn
- Kann Muster erkennen
- Muss trainiert werden
- KNNs: Herzstück des Deep Learnings

# Deep Learning

## Ausblick



### Künstliche neuronale Netze: Part 1

- Einführung
- Training neuronaler Netze
  - Optimierungsalgorithmus
  - Aktivierungsfunktionen
  - Kostenfunktion
  - Performancemetriken

# Deep Learning - Zutaten

## Modellarchitektur

- Anzahl Schichten
- Aktivierungsfunktion

## Kostenfunktion

Definiert was ein gutes neuronales Netz ist

## Optimierungsalgorithmus

Minimiert die Kostenfunktion, indem es Gewichte des NNs variiert

## Training des NNs

Minimierung der Kostenfunktion

# Neuronale Netze programmieren in keras

## Das XOR-Beispiel

```
from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Dense  
from tensorflow.keras.optimizers import SGD
```

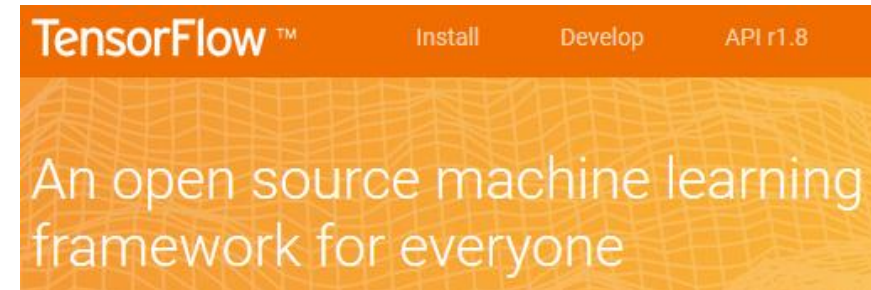
```
(x_train, y_train), (x_test, y_test) = load_data() Daten importieren
```

```
model = Sequential()  
model.add(Dense(units=2, activation='sigmoid', input_shape=(2,)))  
model.add(Dense(units=1, activation='sigmoid'))
```

```
model.compile(loss='binary_crossentropy', optimizer=SGD())
```

```
model.fit(x_train, y_train, epochs=500, verbose=0)
```

```
classes = (model.predict(x_test) > 0.5).astype("int32")  
print("Predictions:", classes)
```



Keras: The Python Deep Learning library



# Neuronale Netze programmieren in keras

## Das XOR-Beispiel

```
import numpy as np
```

```
# Dummy-Daten (XOR-Problem)  
def load_data():  
    x_train = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  
    y_train = np.array([[0], [1], [1], [0]])  
    return (x_train, y_train), (x_train, y_train)
```

Daten importieren

```
(x_train, y_train), (x_test, y_test) = load_data()
```

$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Hier absoluter Sonderfall, dass Training = Testdaten sind.

# Neuronale Netze programmieren in keras

## Das XOR-Beispiel

```
from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Dense  
from tensorflow.keras.optimizers import SGD
```

```
(x_train, y_train), (x_test, y_test) = load_data()
```

```
model = Sequential()  
model.add(Dense(units=2, activation='sigmoid', input_shape=(2,)))  
model.add(Dense(units=1, activation='sigmoid'))
```

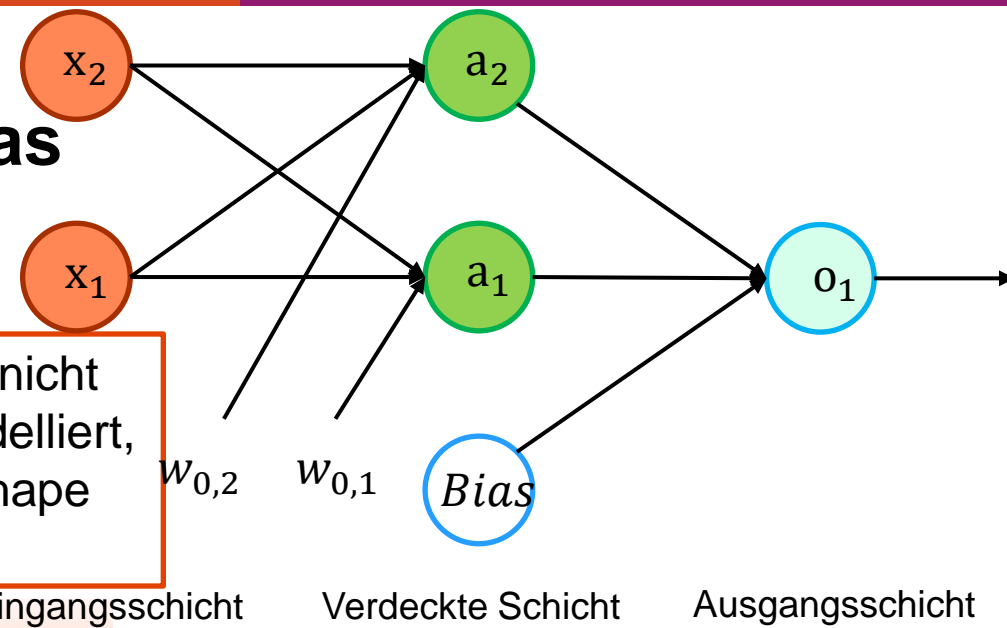
Modell erstellen

```
model.compile(loss='binary_crossentropy', optimizer=SGD())
```

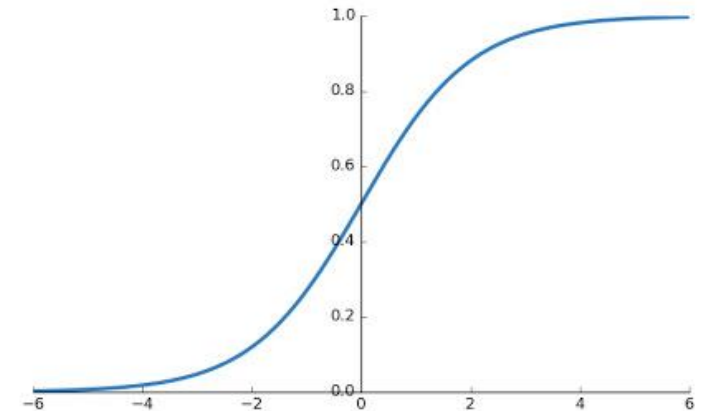
```
model.fit(x_train, y_train, epochs=500, verbose=0)
```

```
classes = (model.predict(x_test) > 0.5).astype("int32")  
print("Predictions:", classes)
```

SGD: Stochastic Gradient Descent



Eingangsschicht wird nicht als eigener Layer modelliert, sondern über input\_shape definiert



# Neuronale Netze programmieren in keras

## Das XOR-Beispiel

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import SGD
```

```
(x_train, y_train), (x_test, y_test) = load_data()
```

```
model = Sequential()
model.add(Dense(units=2, activation='sigmoid', input_shape=(2,)))
model.add(Dense(units=1, activation='sigmoid'))
```

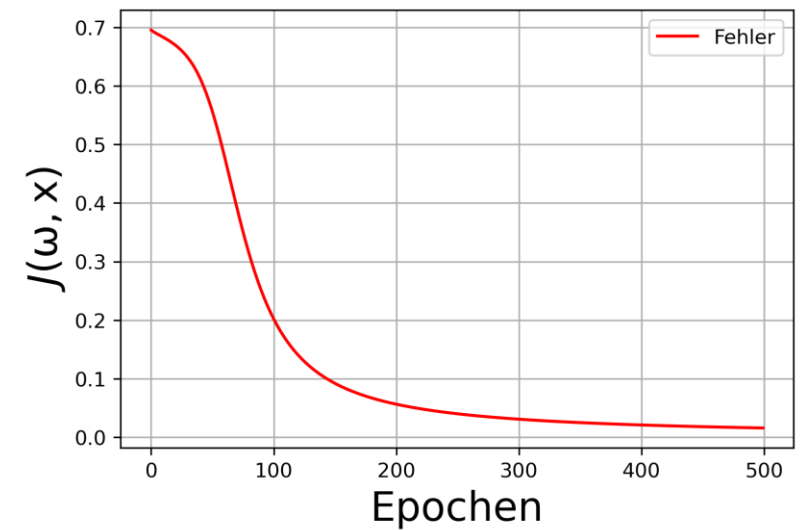
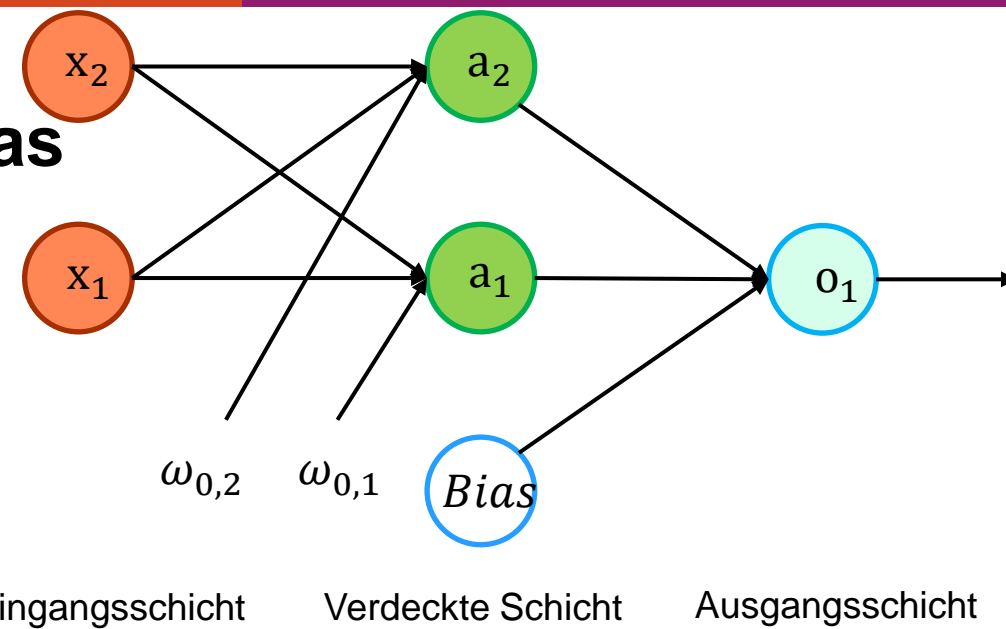
```
model.compile(loss='binary_crossentropy', optimizer=SGD())
```

```
model.fit(x_train, y_train, epochs=500, verbose=0)
```

```
classes = (model.predict(x_test) > 0.5).astype("int32")
print("Predictions:", classes)
```

Modell trainieren

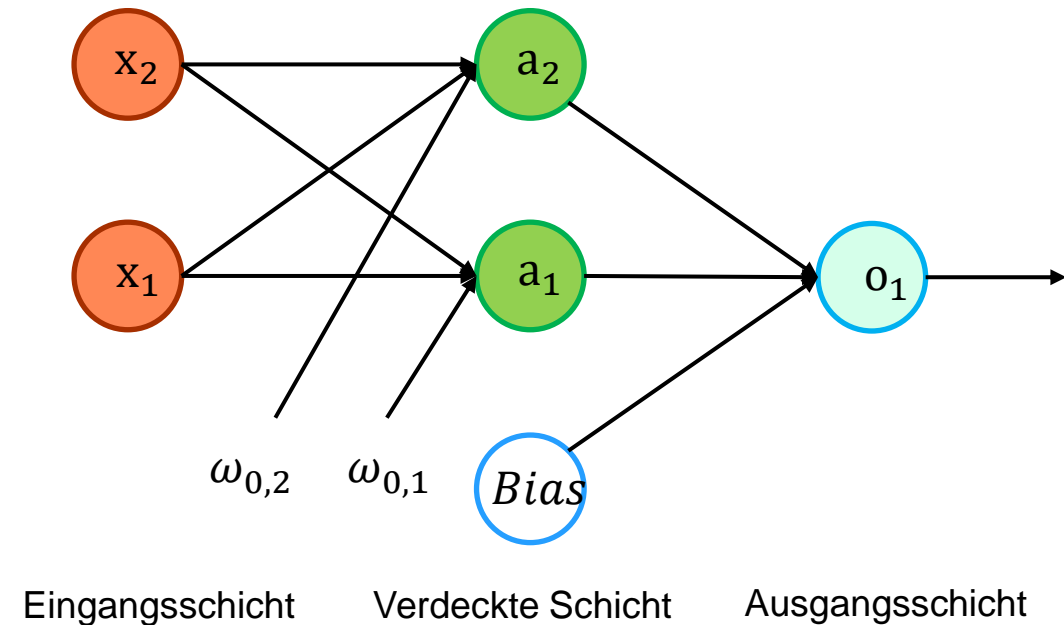
Modell testen



# Neuronale Netze programmieren in keras

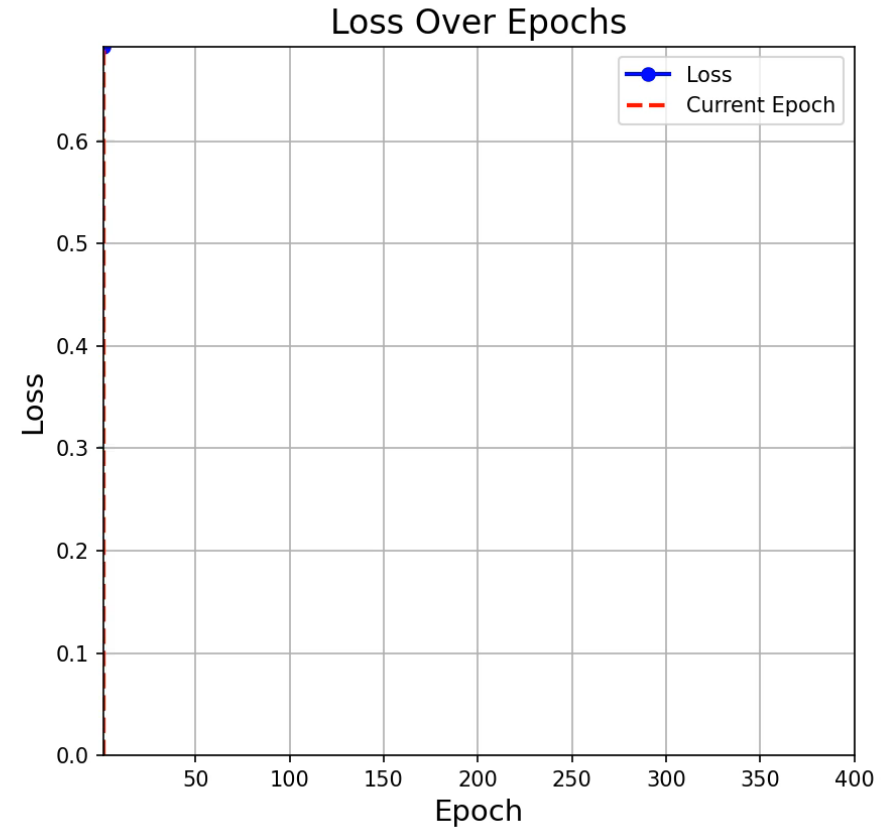
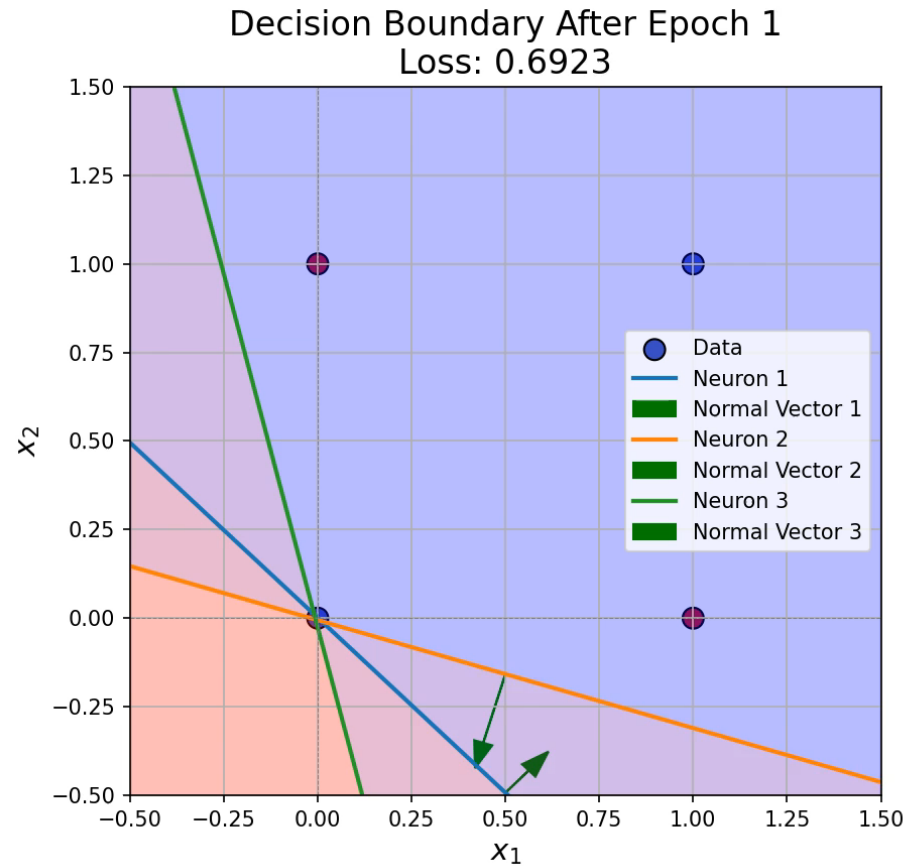
## Das XOR-Beispiel - Übung

1. Gegeben Code mit einem Neuron ([01\\_xor\\_neuralnet1neuron.ipynb](#)), erweitern Sie den Code, sodass Sie drei Neurone haben.
  - 2 in verdeckter Schicht und 1 in Ausgangsschicht
2. Nutzen Sie auch mal drei Neurone in der verdeckten Schicht.
3. Fügen Sie eine zweite verdeckte Schicht hinzu.
4. Vergleichen Sie den Verlauf des Trainingsloss über die Anzahl Epochen für die drei Lösungen



# Künstliche Neuronale Netze - KNN

## Das XOR-Problem – 3 Neurone im Hidden Layer



# Künstliche Neuronale Netze - KNN

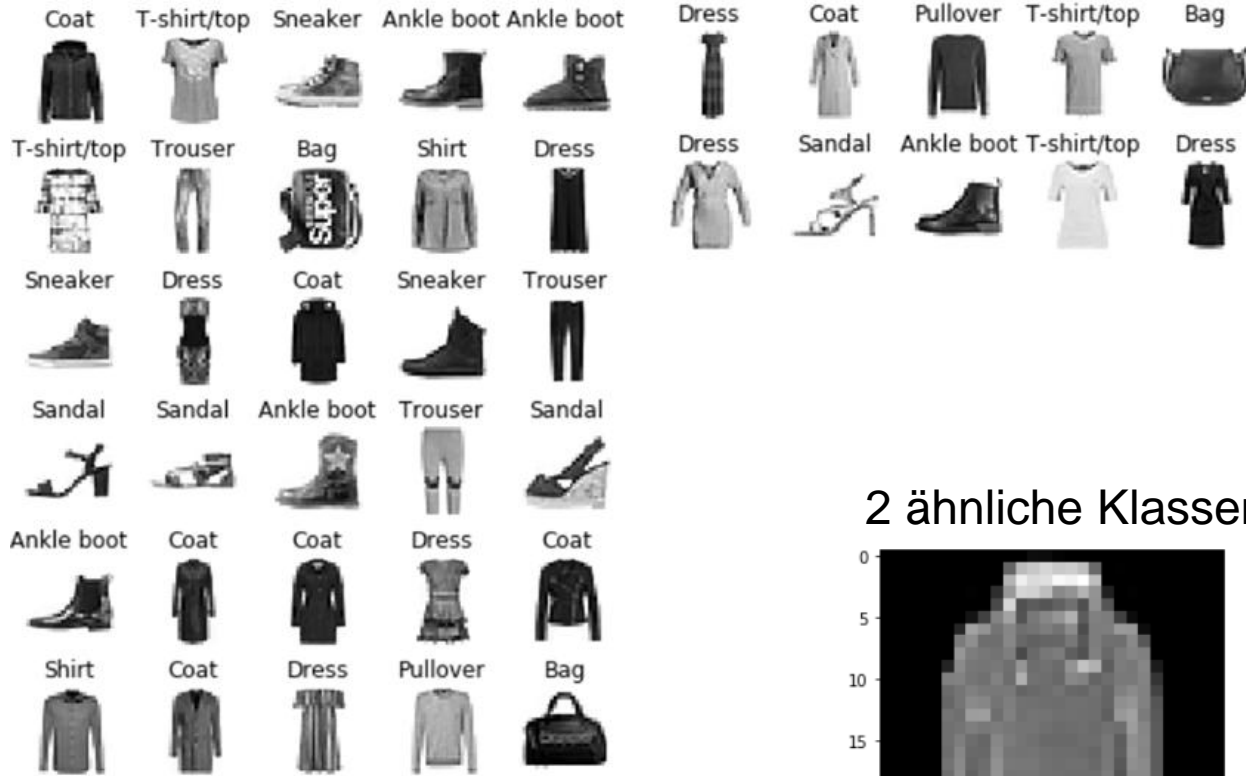
## Einfaches Beispiel

### Binäres Klassifikationsproblem

- Welchen Einfluss hat die Anzahl Schichten auf die Güte des neuronalen Netzes?
- Welchen Einfluss hat die Aktivierungsfunktion auf die Güte des neuronalen Netzes?
- Welchen Einfluss hat der Optimierungsalgorithmus, um möglichst schnell ein gutes neuronales Netz zu erhalten?

# Künstliche Neuronale Netze - KNN

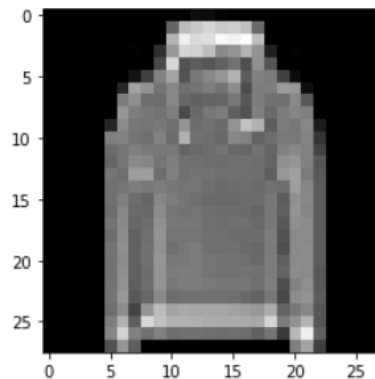
## Beispiel Bildklassifizierer: Fashion MNIST



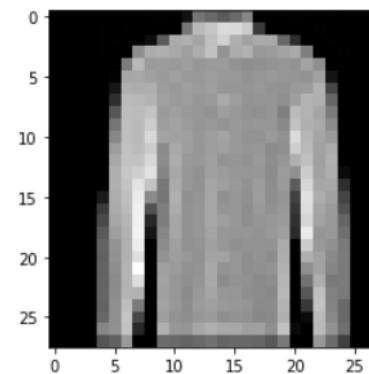
### Fashion-MNIST: Zalando-Artikel

- 28x28 Graustufenbilder
- 60.000 Bilder
- 10 Klassen:
  - T-Shirt/Top, Hose, Pullover, Kleid, Mantel, Sandale, Hemd, Sneaker, Tasche, Stiefelette

### 2 ähnliche Klassen für die binäre Klassifizierung



**Pullover**  
5507 Bilder



**Hemd**  
5496 Bilder

# Neuronale Netze programmieren in keras

## Code für das Fashion MNIST Beispiel

```
from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Dense, Flatten  
from tensorflow.keras.optimizers import SGD
```

```
(x_train, y_train), (x_test, y_test)= load_data()
```

```
model = Sequential()
```

Flatten wandelt Matrix in Vektor um

```
model.add(Flatten(input_shape=[28,28]))  
model.add(Dense(units=1, activation='sigmoid'))
```

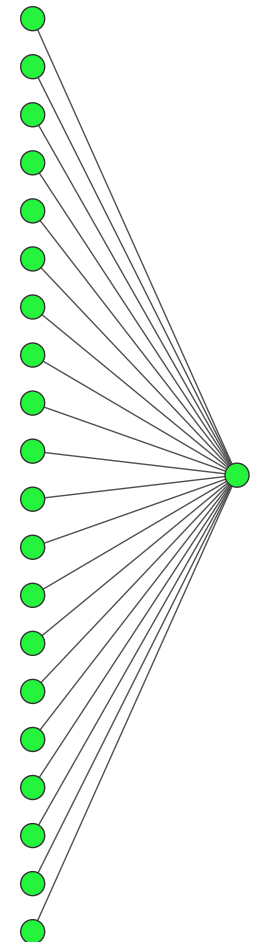
Modell erstellen

```
model.compile(loss='binary_crossentropy', optimizer=SGD())
```

```
model.fit(x_train, y_train, epochs=100)
```

Modell trainieren

Eingabeschicht:  
 $28 \times 28 = 784$   
Neuronen



# Neuronale Netze programmieren in keras

## Code für das Fashion MNIST Beispiel

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import SGD
```

```
(x_train, y_train), (x_test, y_test) = load_data()
```

```
model = Sequential()
```

```
model.add(Flatten(input_shape=[28,28]))
model.add(Dense(units=1, activation='sigmoid'))
```

```
model.compile(loss='binary_crossentropy', optimizer=SGD())
```

```
model.fit(x_train, y_train, epochs=100)
```

```
model.summary()
```

Model: "sequential\_1"

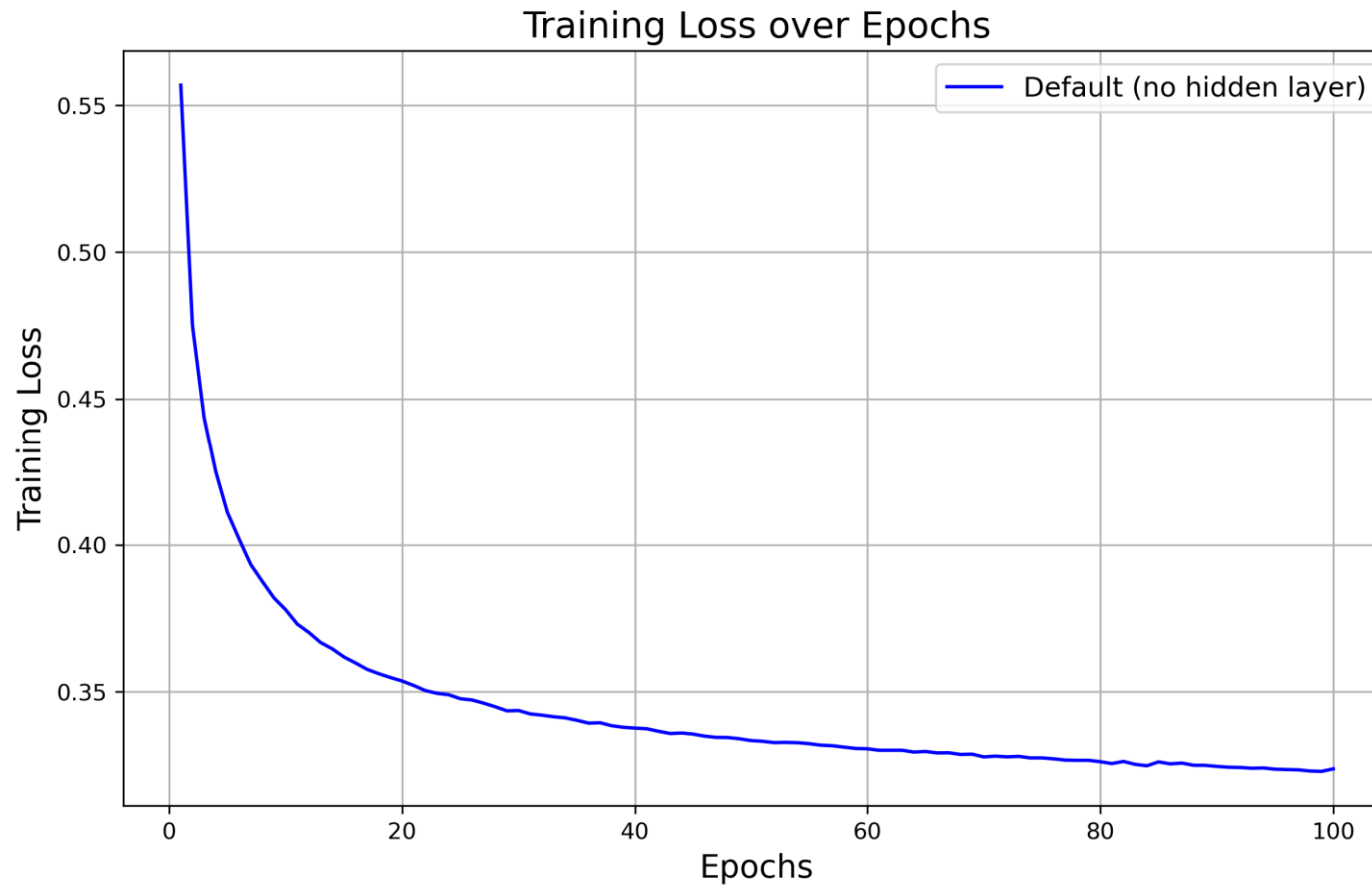
Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 784)	0
dense_1 (Dense)	(None, 1)	785
Total params: 785		
Trainable params: 785		
Non-trainable params: 0		

Modell erstellen

Modell trainieren

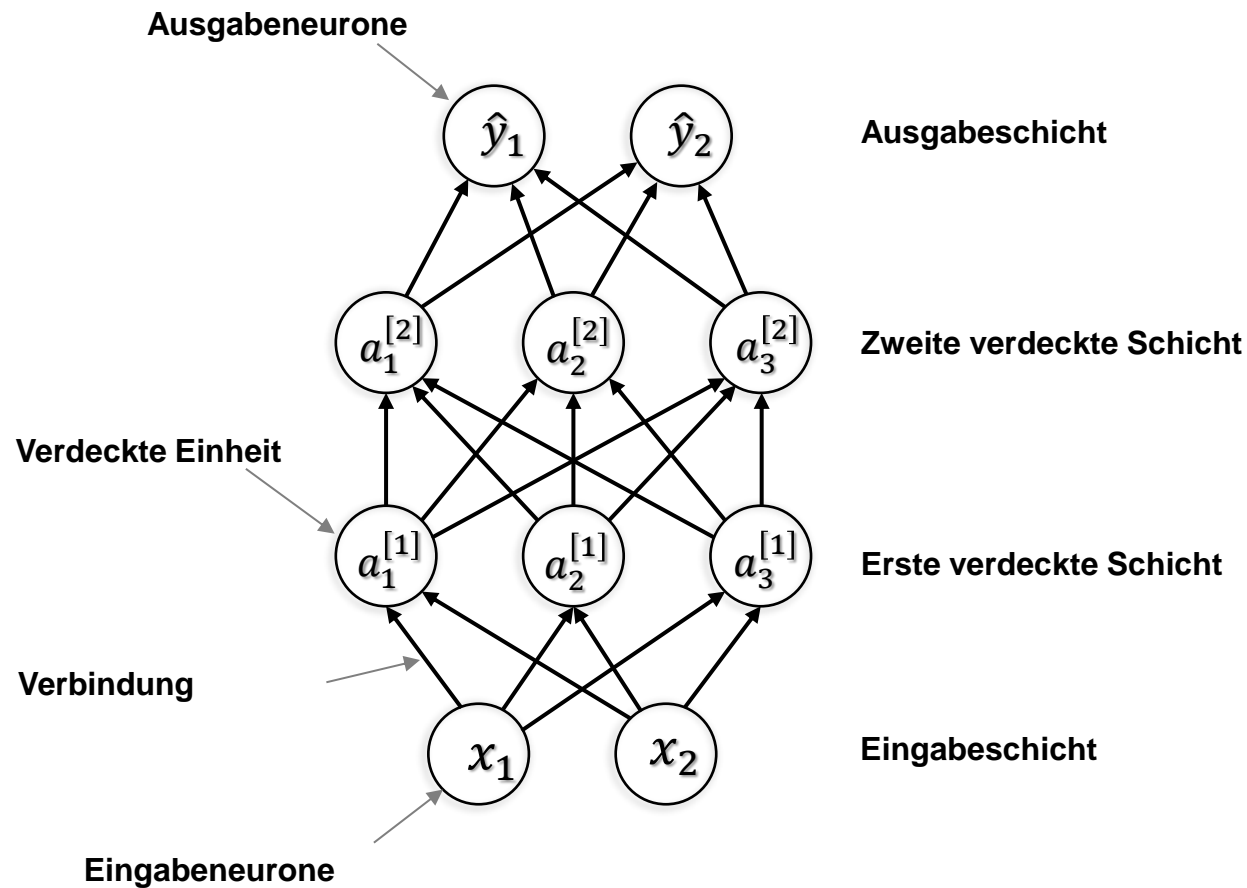
# Künstliche Neuronale Netze - KNN

## Beispiel Bildklassifizierer: Trainingsergebnisse von neuronalem Netz mit 1 Layer



# Künstliche Neuronale Netze - KNN

## Vorwärtspropagation eines Mehrschichtigen neuronalen Netzes



Vorwärtspropagation (mit  $N$  verdeckten Schichten):

$$a_j^{[1]} = \sigma^{[1]} \left( \sum_i w_{ji}^{[1]} \cdot x_i \right)$$

$$a_j^{[\ell]} = \sigma^{[\ell]} \left( \sum_i w_{ji}^{[\ell]} \cdot a_i^{[\ell-1]} \right), \ell = 2, \dots, N$$

$$\hat{y}_j = \sigma^{[N+1]} \left( \sum_i w_{ji}^{[N+1]} \cdot a_j^{[N]} \right)$$

$a_j^{[\ell]}$ : Aktivierungsfunktion des  $j$ -ten Neurons der  $\ell$ -ten verdeckten Schicht

$w_{ji}^{[\ell]}$ : Gewicht des  $i$ -ten Eingangs des  $j$ -ten Neurons der  $\ell$ -ten verdeckten Schicht

$\hat{y}_j$ : Vorhersage des  $j$ -ten Ausgabeneurons

# Künstliche Neuronale Netze - KNN

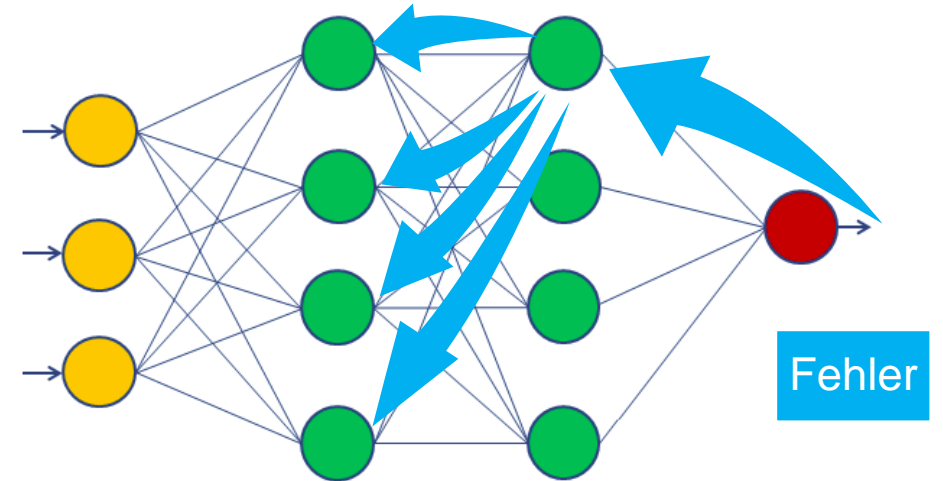
## Trainieren neuronaler Netze

**Trainieren:** Minimierung der Kostenfunktion  $J(\mathbf{w})$ , die Fehler zwischen Vorhersage und Label berechnet

### Backpropagation

- Rückführung des Fehlers durch das gesamte Netz
- Anteilige Aufteilung des Fehlers auf alle Neuronen

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial w_1} \\ \frac{\partial J(\mathbf{w})}{\partial w_2} \\ \frac{\partial J(\mathbf{w})}{\partial w_3} \end{pmatrix}$$



# Künstliche Neuronale Netze - KNN

## Trainieren neuronaler Netze

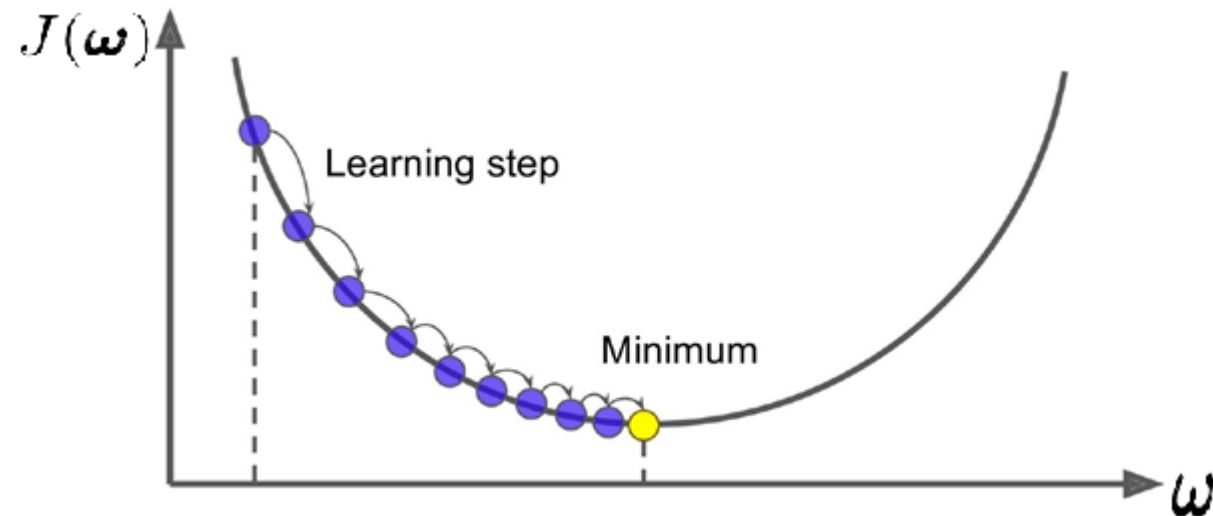
Minimierung der Kostenfunktion durch

**Gradientenabstieg** (Optimierungsalgorithmus)

- Aktualisierung der Parameter (Gewichte, Bias)
- Entgegen der Richtung des Gradienten der Kostenfunktion
- Schrittweite  $\eta$  wird als Lernrate bezeichnet

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} J(\mathbf{w}) \quad \nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial w_1} \\ \frac{\partial J(\mathbf{w})}{\partial w_2} \\ \frac{\partial J(\mathbf{w})}{\partial w_3} \end{pmatrix}$$

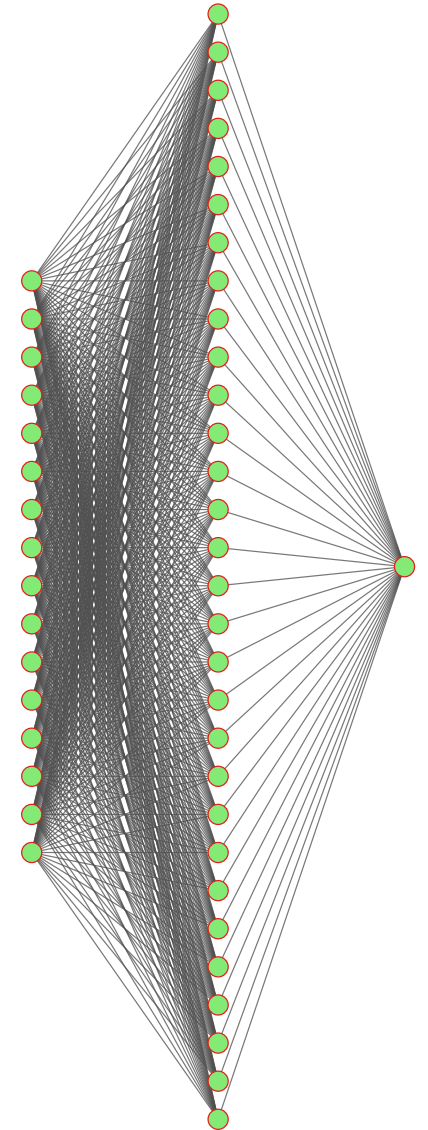
Gradient zeigt in Richtung des steilsten Anstiegs



# Künstliche Neuronale Netze - KNN

## Beispiel Bildklassifizierer: Vergleich von zwei Aktivierungsfunktionen

Mehrlagiges Netz mit einer Eingangsschicht (Flatten), einer verdeckten Schicht (Dense und 300 Neuronen) und einer Ausgangsschicht (Dense, Sigmoid):



# Künstliche Neuronale Netze - KNN

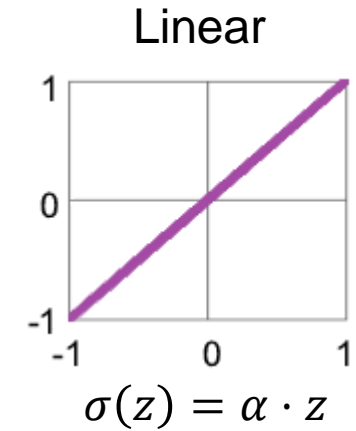
## Beispiel Bildklassifizierer: Vergleich von zwei Aktivierungsfunktionen

Mehrlagiges Netz mit einer Eingangsschicht (Flatten), einer verdeckten Schicht (Dense und 300 Neuronen) und einer Ausgangsschicht (Dense, Sigmoid):

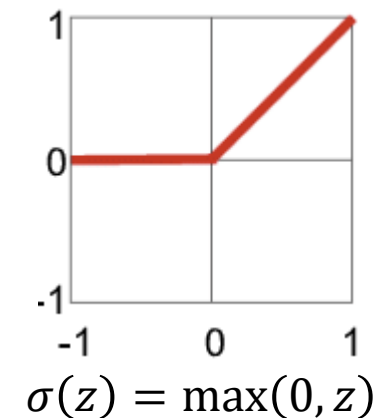
- verdeckte Schicht mit linearer Aktivierung  
Einführung des **Hyperparameters**  $\alpha$
- verdeckte Schicht mit ReLU (Rectifier Linear Unit) Aktivierung

```
model_linear.add(Flatten(input_shape=[28, 28]))
model_linear.add(Dense(units=300, activation='linear'))
model_linear.add(Dense(units=1, activation='sigmoid'))
```

```
model_relu.add(Flatten(input_shape=[28, 28]))
model_relu.add(Dense(units=300, activation='relu'))
model_relu.add(Dense(units=1, activation='sigmoid'))
```



Rectified Linear Unit (ReLU)



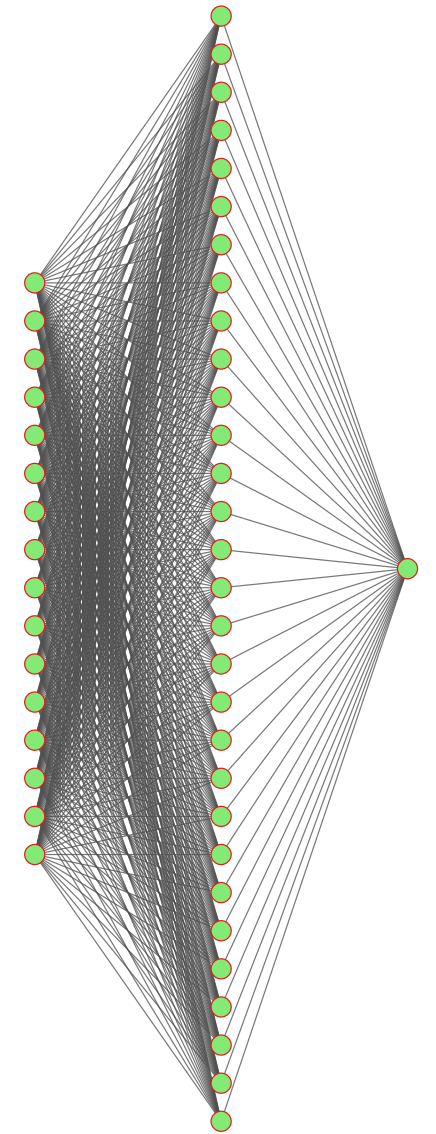
# Künstliche Neuronale Netze - KNN

## Beispiel Bildklassifizierer: Vergleich von zwei Aktivierungsfunktionen

```
model_relu.summary()
```

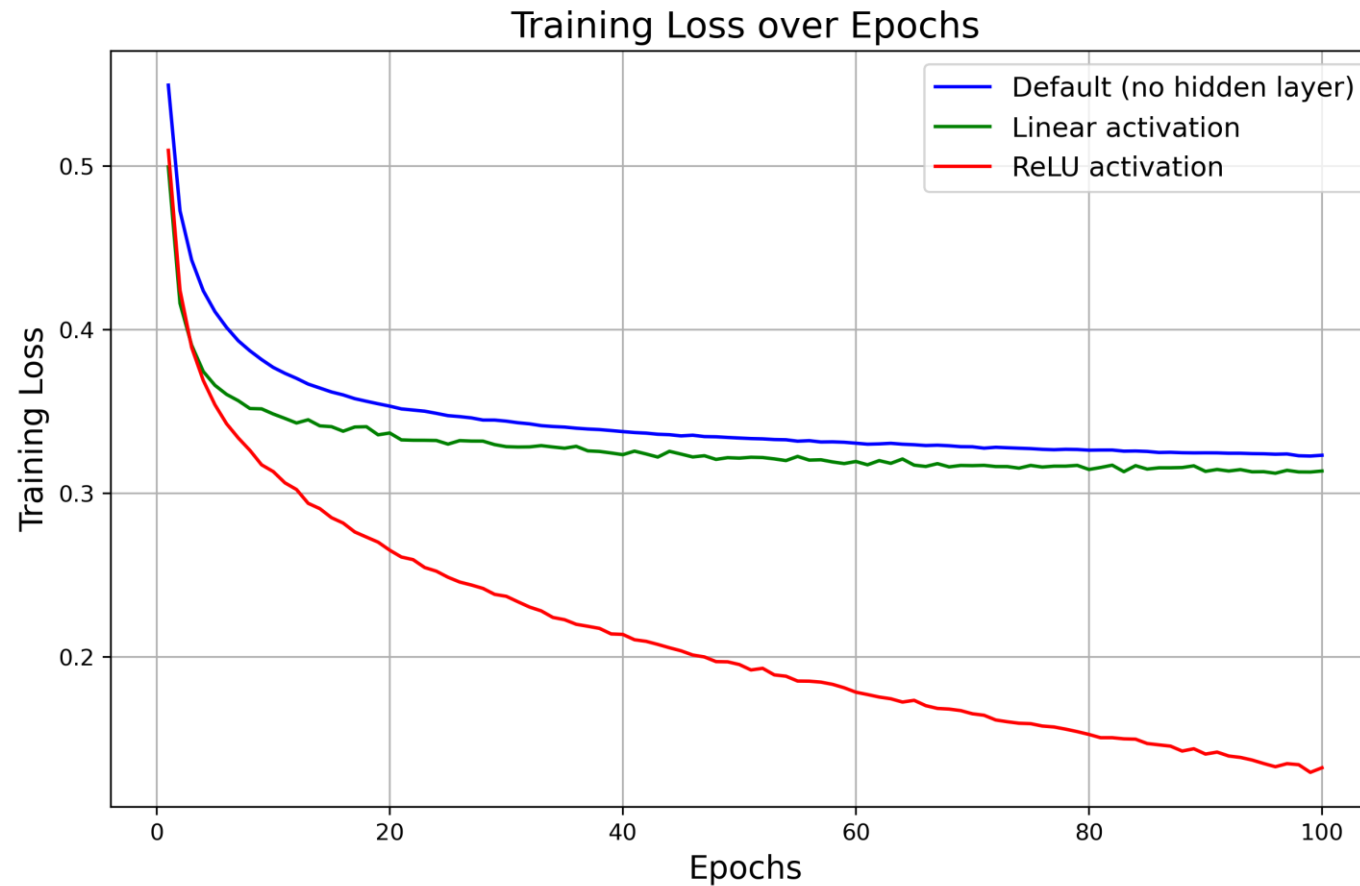
Layer (type)	Output Shape	Param #
flatten_2 (Flatten)	(None, 784)	0
dense_3 (Dense)	(None, 300)	235500
dense_4 (Dense)	(None, 1)	301

=====  
Total params: 235,801  
Trainable params: 235,801  
Non-trainable params: 0  
=====  
 $300 * 784 + 300 = 235500$



# Künstliche Neuronale Netze - KNN

## Beispiel Bildklassifizierer: Vergleich von zwei Aktivierungsfunktionen

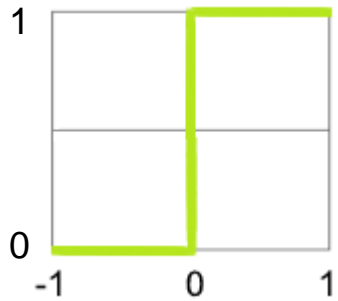


Eine weitere Schicht bringt nicht viel, wenn die Aktivierungsfunktion in dieser linear ist

# Künstliche Neuronale Netze - KNN

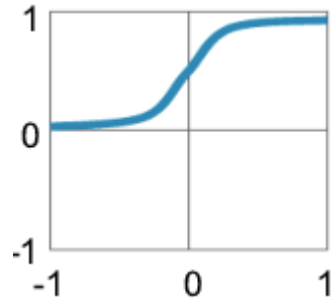
## Aktivierungsfunktionen

Binäre Stufenfunktion



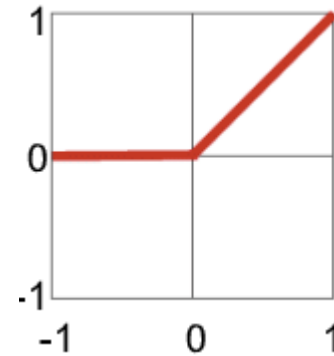
$$f(z) = \begin{cases} 0 & \text{falls } z < 0 \\ 1 & \text{sonst} \end{cases}$$

Sigmoid



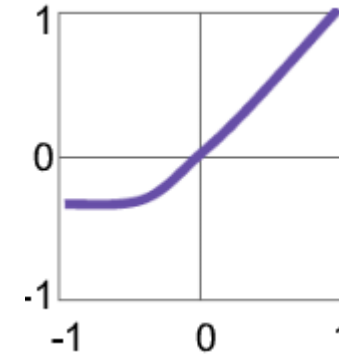
$$f(z) = \frac{1}{1 + e^{-z}}$$

Rectified Linear Unit (ReLU)



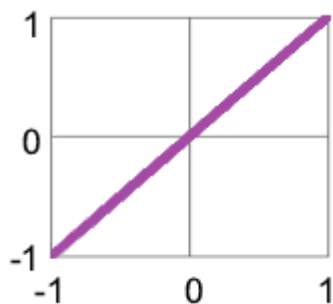
$$f(z) = \max(0, z)$$

Scaled Exp. LU (SELU)



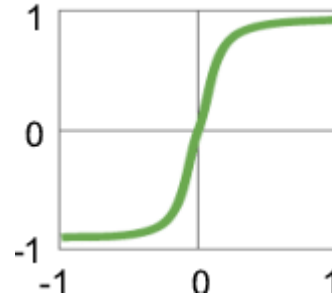
$$f(z) = \lambda \begin{cases} \alpha(e^z - 1) & \text{falls } z < 0 \\ z & \text{sonst} \end{cases}$$

Linear



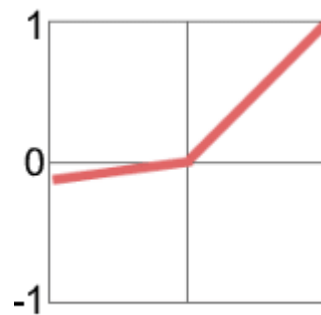
$$f(z) = \alpha z$$

Tangens hyperbolicus



$$f(z) = \tanh(z)$$

Leaky ReLU



$$f(z) = \max(\alpha z, z)$$

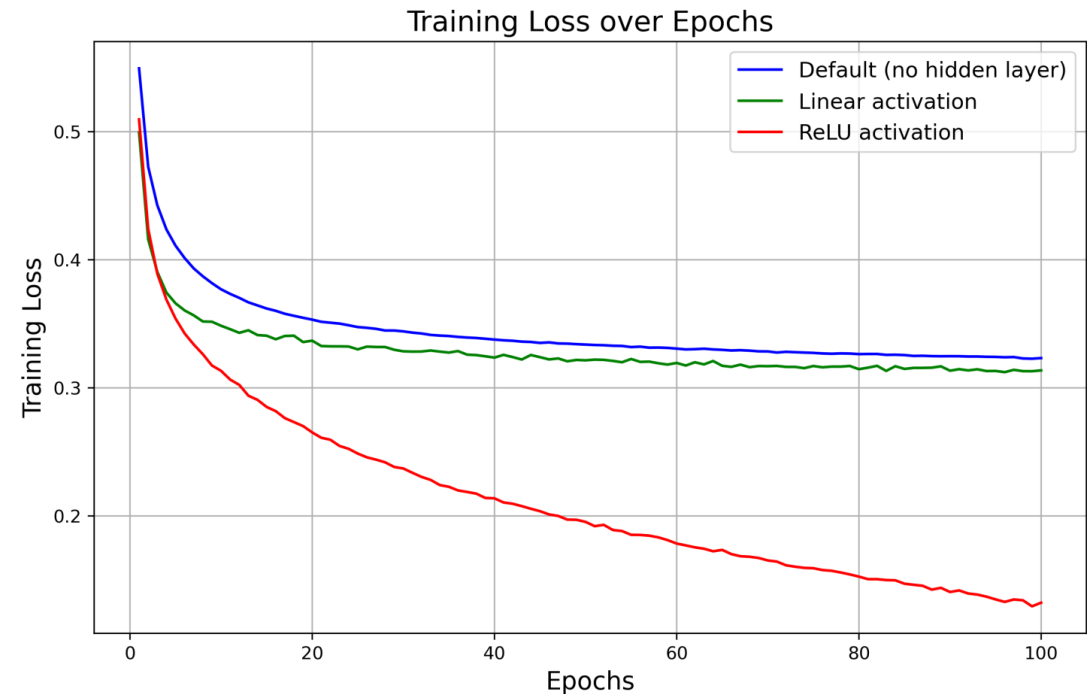
Softmax

$$f(z_k) = \text{softmax}(z_k) = \frac{e^{z_k}}{\sum_c e^{z_c}}$$

# Künstliche Neuronale Netze - KNN

## Übung Bildklassifizierer: Vergleich von Aktivierungsfunktionen

- Probieren Sie verschiedene Aktivierungsfunktionen aus und schauen Sie sich den Trainingsloss über die Epochen an.
- SELU, ReLU, leaky ReLU, ...  
[https://www.tensorflow.org/api\\_docs/python/tf/keras/activations](https://www.tensorflow.org/api_docs/python/tf/keras/activations)
- Nutzen Sie:  
[02\\_fashion\\_mnist\\_binary.ipynb](#)
- Tipp: Nutzen Sie eine GPU in Google Colab



# Deep Learning - Zutaten

## Modellarchitektur

- Anzahl Schichten
- Aktivierungsfunktion

## Kostenfunktion

Definiert was ein gutes neuronales Netz ist

## Optimierungsalgorithmus

Minimiert die Kostenfunktion, indem es Gewichte des NNs variiert

## Training des NNs

Minimierung der Kostenfunktion

# Künstliche Neuronale Netze - KNN

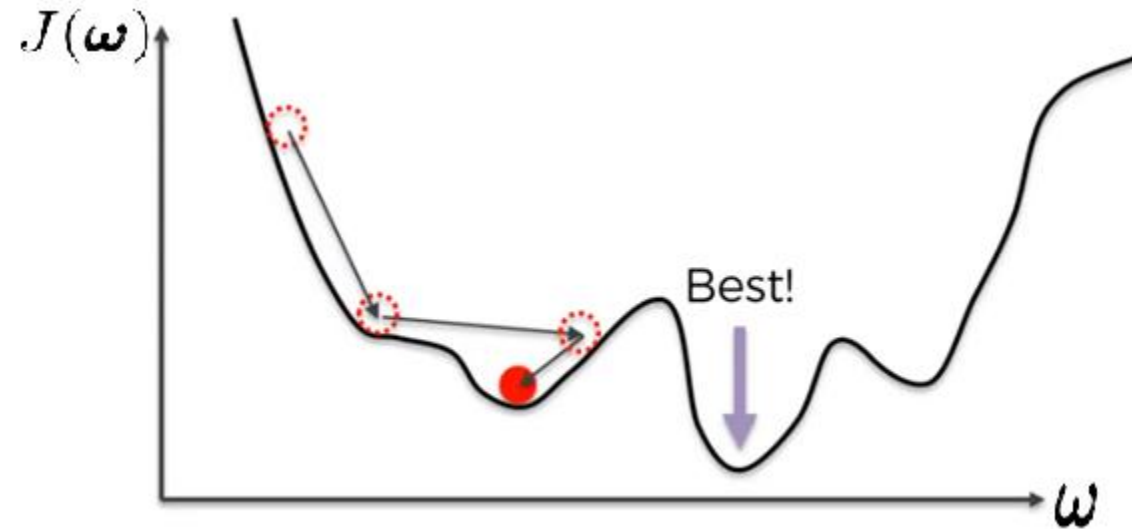
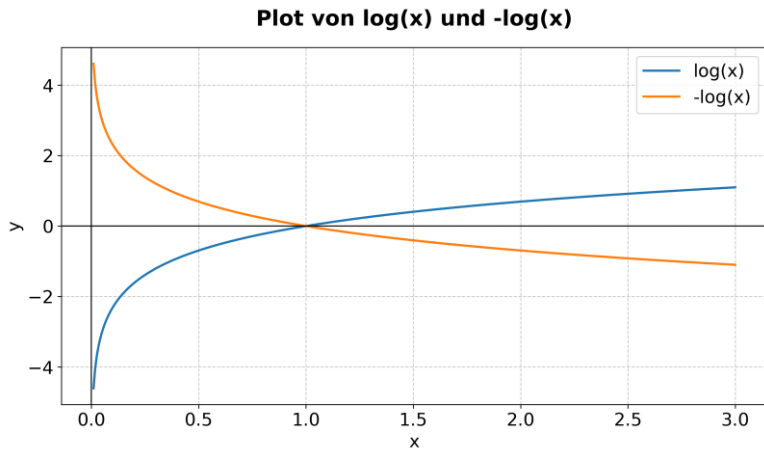
## Gradient Descent minimiert Kostenfunktion

Kostenfunktion für binäres Klassifikationsproblem:

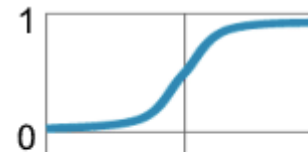
- Binäre Kreuzentropie (binary cross entropy)

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \frac{\partial J(\mathbf{w}, \mathbf{x})}{\partial \mathbf{w}} = \mathbf{w} - \eta \cdot \nabla_{\mathbf{w}} J(\mathbf{w}, \mathbf{x})$$

```
model.compile(loss='binary_crossentropy', optimizer=SGD())
```



$$J(\mathbf{w}; \mathbf{X}) = -\frac{1}{N} \sum_{i=0}^{N-1} \left( y^{(i)} \log(\hat{p}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)}) \right) \quad \text{mit} \quad \left( \hat{p} = \sigma(z) = \frac{1}{1 + e^{-z}} \right)$$



Sigmoid

# Künstliche Neuronale Netze - KNN

## Gradient Descent

### Vanilla Gradient Descent (Batch Gradient Descent)

- Mittlerer Gradient aus allen Trainingsinstanzen
- Nachteil: nur konvexe Funktionen; Vorteil: schnell

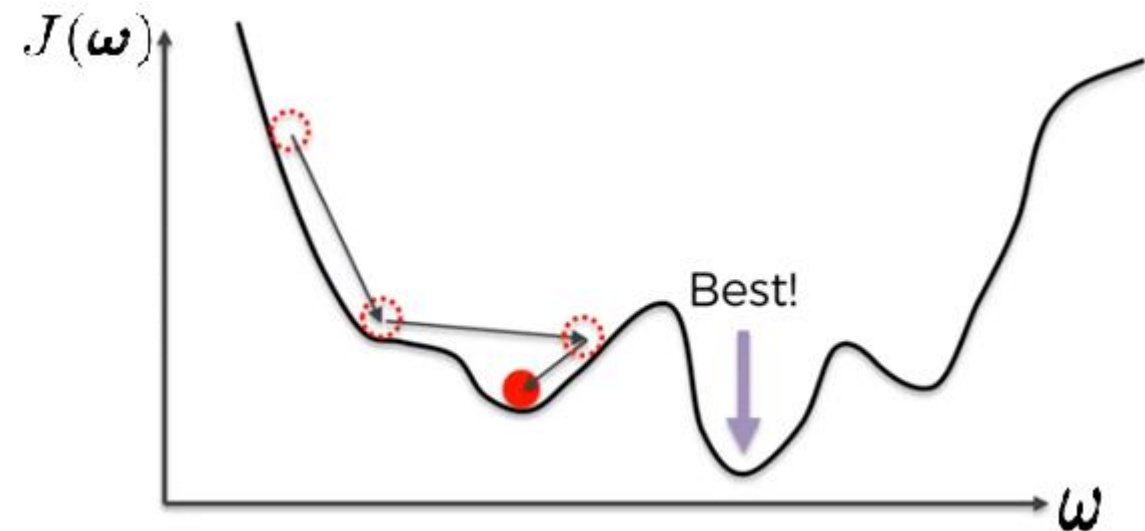
### Stochastic Gradient Descent (SGD)

- Berechnet Gradient für jede Trainingsinstanz einzeln; Nachteil: sehr langsam
- Flucht aus lokalen Minima möglich

### Mini-Batch Gradient Descent

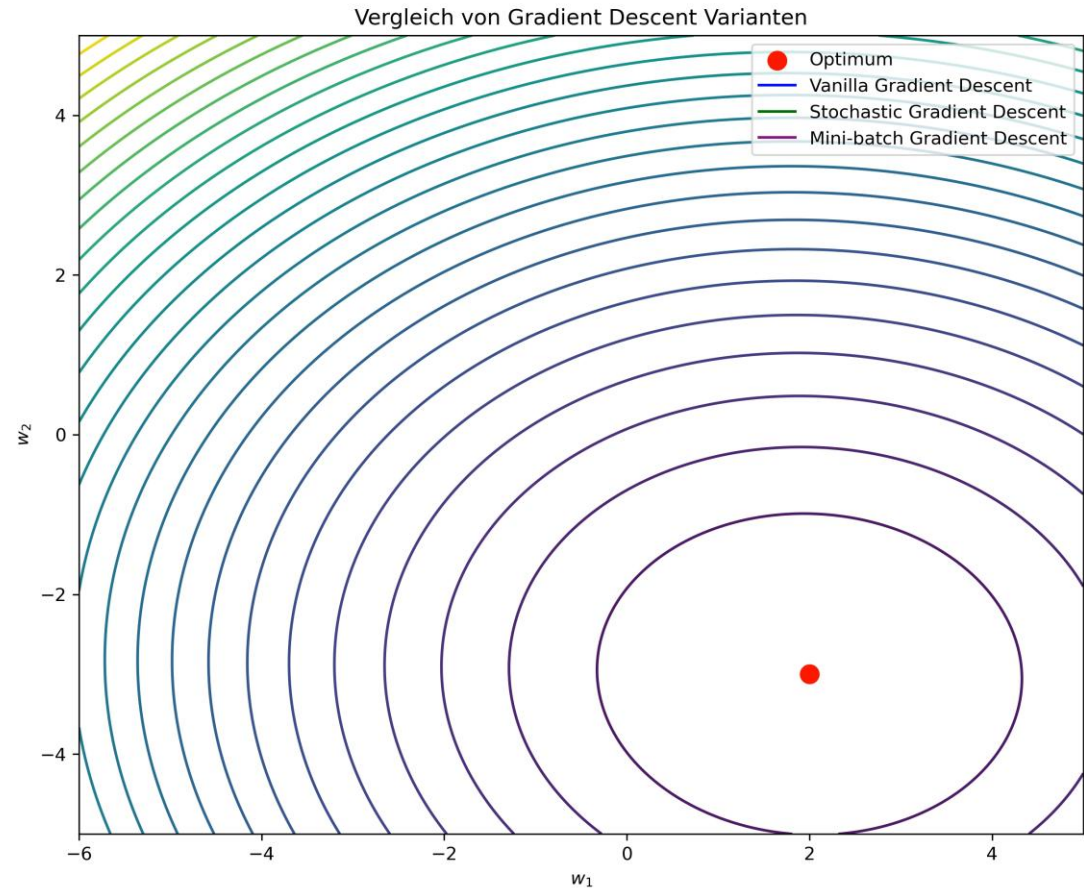
- Nutzt Mini-Batches
- Mini-Batch: Teil des Trainingsdatensatzes
- Hat Vorteile beider vorherigen Methoden

- $\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \frac{\partial J(\mathbf{w}, \mathbf{x})}{\partial \mathbf{w}} = \mathbf{w} - \eta \cdot \nabla_{\mathbf{w}} J(\mathbf{w}, \mathbf{x})$
- $\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \nabla_{\mathbf{w}} J(\mathbf{w}, \mathbf{x}^{(i)}; y^{(i)})$
- $\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \nabla_{\mathbf{w}} J(\mathbf{w}, \mathbf{x}^{(i:i+n)}; y^{(i:i+n)})$



# Künstliche Neuronale Netze - KNN

## Gradient Descent



# Künstliche Neuronale Netze - KNN

## Beispiel Bildklassifizierer: Vergleich von drei Optimierungsalgorithmen

Mehrlagiges Netz mit einer Eingangsschicht (Flatten), einer verdeckten Schicht (Dense) und einer Ausgangsschicht (Dense):

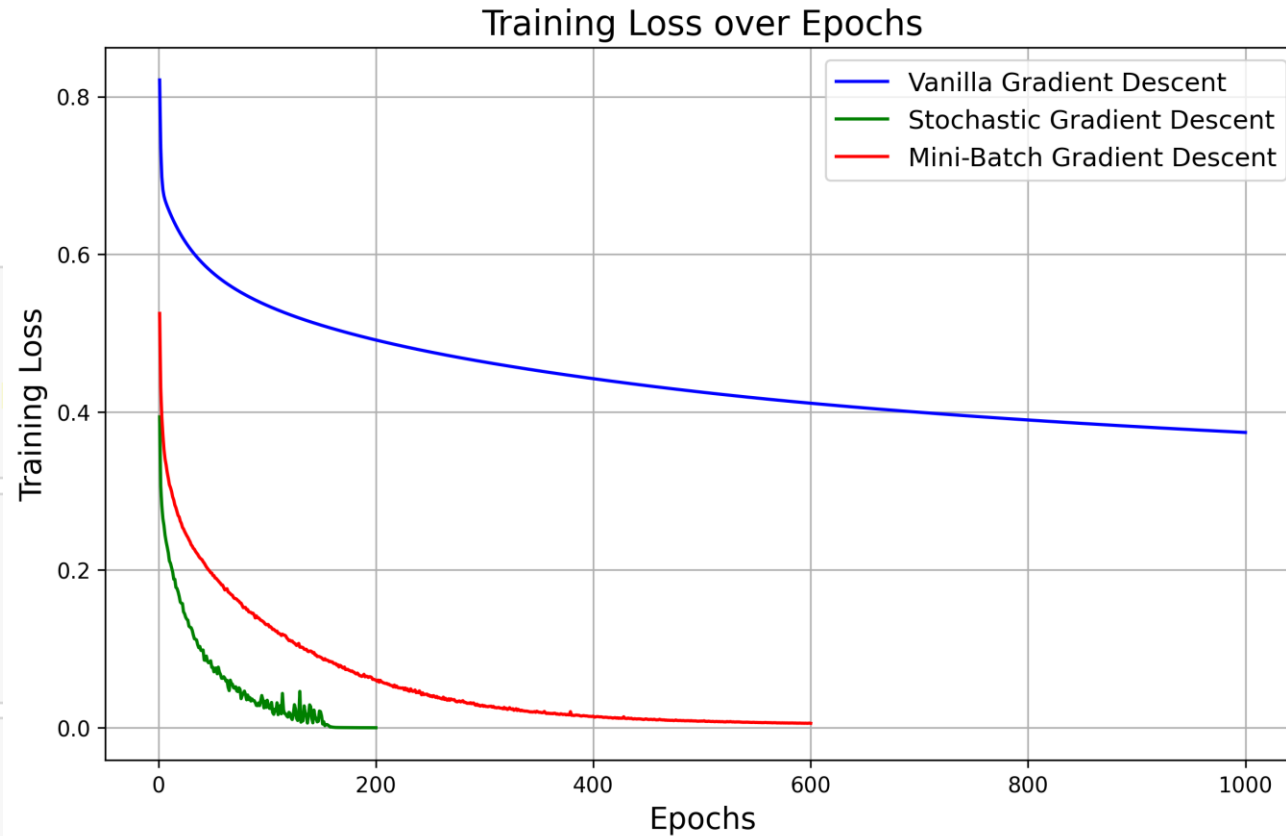
- a. Vanilla Gradient Descent (1000, 35.14 s)
- b. Stochastic Gradient Descent (200, 1713.81 s)
- c. Mini-batch Gradient Descent (600, 216.22 s)

```
history_VGD = model_VGD.fit(X_train_two,
                             y_train_two,
                             batch_size=X_train_two.shape
                             epochs=1000)
```

```
history_SGD = model_SGD.fit(X_train_two,
                             y_train_two,
                             batch_size=1,
                             epochs=200)
```

```
history_BGD = model_BGD.fit(X_train_two,
                             y_train_two,
                             epochs=600)
```

Standardwert: 32



# Künstliche Neuronale Netze - KNN

## Einfaches Beispiel - Erkenntnisse

### Binäres Klassifikationsproblem

- Welchen Einfluss hat die Anzahl Schichten auf die Güte des neuronalen Netzes?
  - Höhere Anzahl Schichten = Mehr Gewichte = Mehr Freiheitsgrade
  - → kleinerer finaler Loss auf den Trainingsdaten
- Welchen Einfluss hat die Aktivierungsfunktion auf die Güte des neuronalen Netzes?
  - Die Aktivierungsfunktion ist das einzig Nichtlineare im neuronalen Netz
  - Aktivierungsfunktion hat auch einen erheblichen Einfluss auf die Konvergenz des Gradientenabstiegs (dazu später mehr)
- Welchen Einfluss hat der Optimierungsalgorithmus, um möglichst schnell ein gutes neuronales Netz zu erhalten?
  - Mini-Batch Gradient Descent beschleunigt das Training sehr
  - Später werden wir weitere Optimierungsalgorithmen kennenlernen

# Künstliche Neuronale Netze - KNN

## Multinomiales Klassifikationsproblem: Fashion MNIST



### Fashion-MNIST: Zalando-Artikel

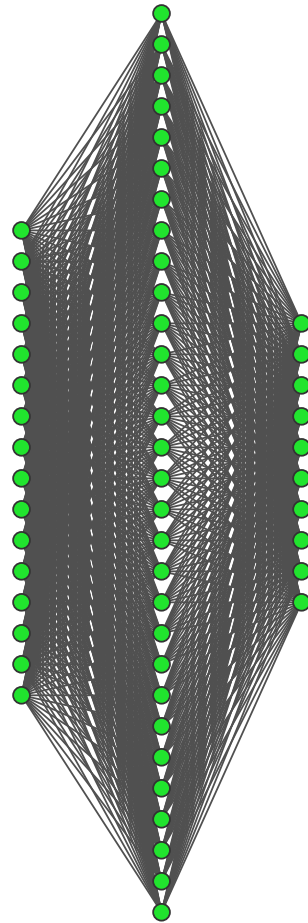
- 28x28 Graustufenbilder
- 60.000 Bilder
- 10 Klassen:
  - T-Shirt/Top, Hose, Pullover, Kleid, Mantel, Sandale, Hemd, Sneaker, Tasche, Stiefelette

### Benötigte Änderungen:

- Anzahl Neuronen der letzten Schicht
- Aktivierungsfunktion der letzten Schicht
- Kostenfunktion

# Künstliche Neuronale Netze - KNN

## Multinomiales Klassifikationsproblem: Fashion MNIST

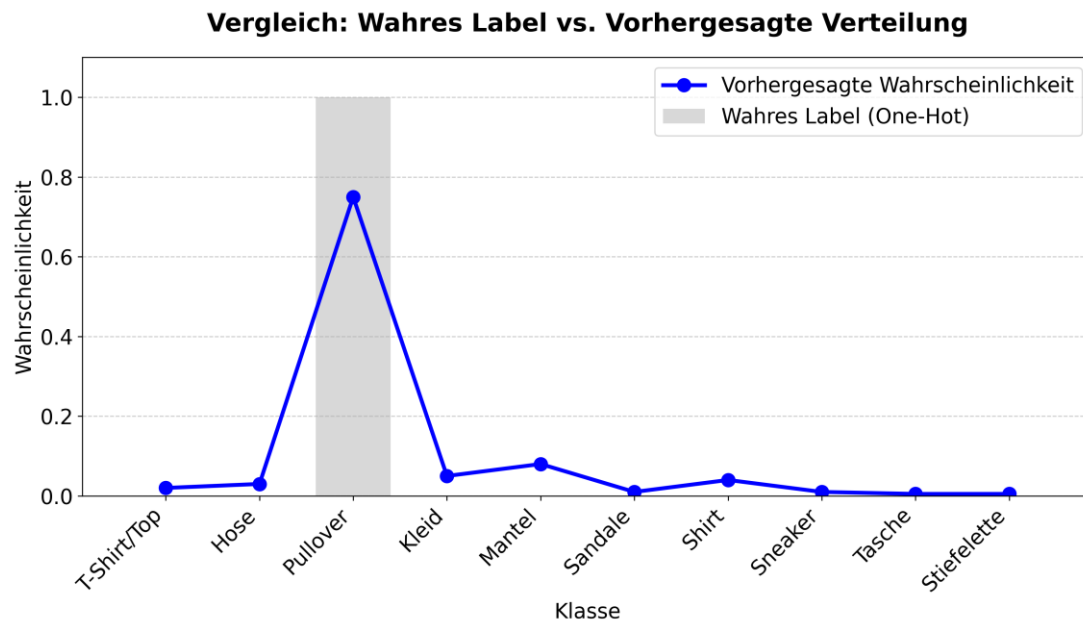


- 10 Klassen → 10 Neuronen in Ausgabeschicht; jedes Neuron modelliert eine Klasse
- Ziel: Jedes Neuron gibt Wahrscheinlichkeit für ihre Klasse zurück; die Ausgabeschicht eine Wahrscheinlichkeitsverteilung
- D.h., Aktivierungsfunktion muss gewichtete Summe in Zahlen zwischen 0 und 1 transformieren, die in Summe (über alle Neuronen) 1 ergeben.
- Softmax Aktivierungsfunktion macht das:

$$\begin{bmatrix} 1.3 \\ 5.1 \\ 2.2 \\ 0.7 \\ 1.1 \end{bmatrix} \rightarrow \frac{e^{z_k}}{\sum_{j=0}^K e^{z_j}} \rightarrow \begin{bmatrix} 0.02 \\ 0.90 \\ 0.05 \\ 0.01 \\ 0.02 \end{bmatrix}$$

# Künstliche Neuronale Netze - KNN

## Multinomiales Klassifikationsproblem: Fashion MNIST



- Kostenfunktion vergleicht Labels mit zurückgegebener Wahrscheinlichkeitsverteilung
- Dafür müssen Labels auch als Wahrscheinlichkeitsverteilung modelliert werden
- One-Hot-Encoding
- Bsp.:
  - 3 Klassen → 3 One-Hot-Vektoren
    - [1, 0, 0]
    - [0, 1, 0]
    - [0, 0, 1]

# Künstliche Neuronale Netze - KNN

## Trainieren neuronaler Netze – Kostenfunktion $J(\mathbf{w})$

**Kostenfunktion**  $J(\mathbf{w}; \mathbf{X})$ : misst die Abweichung zwischen den Ausgängen  $\hat{\mathbf{y}}^{(i)} = h(\mathbf{x}^{(i)})$  und den entsprechenden Sollwerten  $\mathbf{y}^{(i)}$

Regressionsproblem:

- Mittlere Abweichung (*mean squared error*):

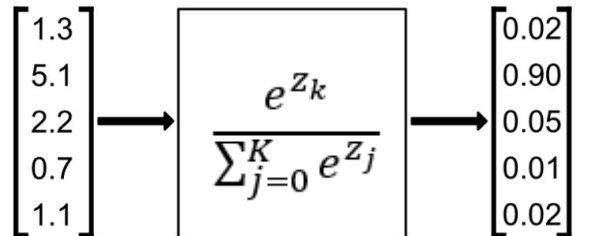
$$J(\mathbf{w}; \mathbf{X}) = \frac{1}{N} \sum_{i=0}^{N-1} (\hat{\mathbf{y}}^{(i)} - \mathbf{y}^{(i)})^2$$

# Künstliche Neuronale Netze - KNN

## Trainieren neuronaler Netze – Kostenfunktion $J(\mathbf{w})$

**Kostenfunktion**  $J(\mathbf{w}; \mathbf{X})$ : misst die Abweichung zwischen den Ausgängen  $\hat{\mathbf{y}}^{(i)} = h(\mathbf{x}^{(i)})$  und den entsprechenden Sollwerten  $\mathbf{y}^{(i)}$

- Kreuzentropie / kategorische Kreuzentropie (*cross entropy / categorical cross entropy*):
  - Softmax transformiert K-dimensionalen Vektor  $\mathbf{z}$  in den Wertebereich (0,1) mit Summe 1

$$J(\mathbf{w}; \mathbf{X}) = -\frac{1}{N} \sum_{i=0}^{N-1} \sum_{k=0}^{K-1} y_k^{(i)} \log(\hat{p}_k^{(i)}) \quad \text{mit} \quad \left( \hat{p}_k = \sigma(\mathbf{z})_k = \frac{e^{z_k}}{\sum_{j=0}^{K-1} e^{z_j}} \right)$$


- Binäre Kreuzentropie (2 Klassen:  $K=2$ ) (*binary cross entropy*):

$$J(\mathbf{w}; \mathbf{X}) = -\frac{1}{N} \sum_{i=0}^{N-1} \left( y^{(i)} \log(\hat{p}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)}) \right) \quad \text{mit} \quad \left( \hat{p} = \sigma(z) = \frac{1}{1 + e^{-z}} \right)$$

# Neuronale Netze programmieren in keras

## Code für das Fashion MNIST Beispiel

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import SGD
```

```
(x_train, y_train), (x_test, y_test) = load_data()
```

Daten importieren

```
model = Sequential()
```

```
model.add(Flatten(input_shape=[28,28]))
model.add(Dense(units=300, activation='relu'))
model.add(Dense(units=10, activation='softmax'))
```

Modell erstellen

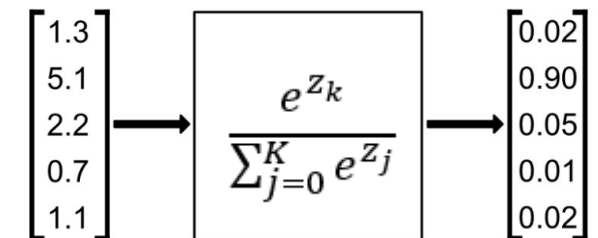
```
model.compile(loss='sparse_categorical_crossentropy', optimizer=SGD())
```

```
model.fit(x_train, y_train, epochs=100)
```

Modell trainieren

Softmax Aktivierungsfunktion:

Garantiert, dass die Ausgabe des neuronalen Netzes ein normalisierter K-dimensionaler Vektor ist (hier K=10), d.h. ein Wahrscheinlichkeitsvektor ist.



# Neuronale Netze programmieren in keras

## Code für das Fashion MNIST Beispiel

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import SGD
```

```
(x_train, y_train), (x_test, y_test) = load_data()
```

Daten importieren

```
model = Sequential()
```

```
model.add(Flatten(input_shape=[28,28]))
model.add(Dense(units=300, activation='relu'))
model.add(Dense(units=10, activation='softmax'))
```

Modell erstellen

```
model.compile(loss='sparse_categorical_crossentropy', optimizer=SGD())
```

```
model.fit(x_train, y_train, epochs=100)
```

Modell trainieren

*sparse\_categorical\_crossentropy* und *categorical\_crossentropy*

Beispiel: 5 Samples mit 3 Klassen

- Klassen schließen sich gegenseitig aus

*sparse\_categorical\_crossentropy*:

$$y = \begin{pmatrix} 1 \\ 0 \\ 2 \\ 1 \\ 0 \end{pmatrix}$$

*categorical\_crossentropy*:

$$y = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

# Neuronale Netze programmieren in keras

## Übung: Fashion MNIST Beispiel

1. Trainieren Sie das neuronale Netz unter Nutzung der „*sparse\_categorical\_crossentropy*“.  
Nutzen Sie: [03 fashion\\_mnist\\_10\\_loss\\_fun.ipynb](#)
2. Ändern Sie die „loss function“ zu „*categorical\_crossentropy*“ und passen Sie die Labels  $\mathbf{y}$  so an, dass Sie die neue loss function nutzen können.
  - Siehe „load\_data\_full“ in `utils_fminst.py`
3. Trainieren Sie das neuronale Netz mit der neuen loss function. Vergleichen Sie die Ergebnisse mit denen aus dem ersten Experiment.

Tipp: Reduzieren Sie die Anzahl Epochen auf 5 und/oder nutzen Sie eine GPU. Editieren Sie nicht die `utils_fminst.py` sondern passen Sie im Notebook `y_train/y_test` an. Nach jeder Änderung in `utils_fminst.py` müssten Sie sonst die Sitzung neu starten.

*sparse\_categorical\_crossentropy* und *categorical\_crossentropy*

Beispiel: 5 Samples mit 3 Klassen

- Klassen schließen sich gegenseitig aus

*sparse\_categorical\_crossentropy*:

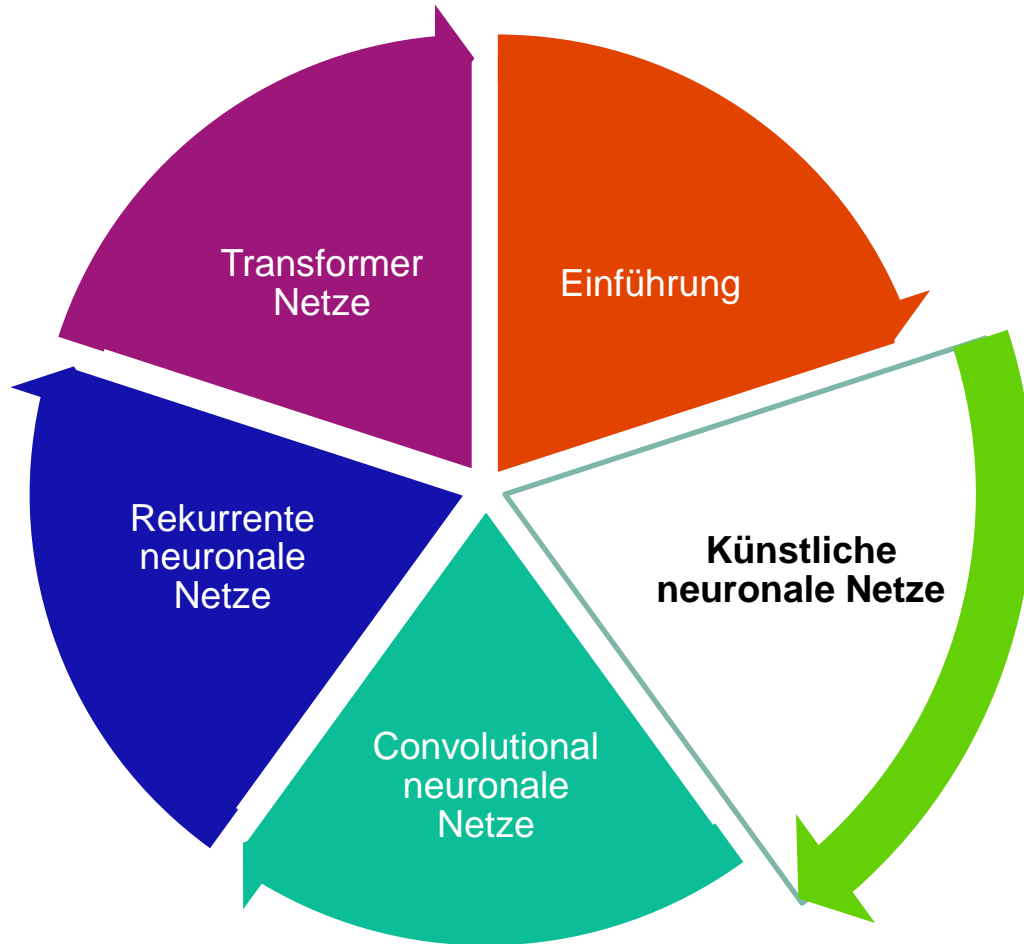
$$\mathbf{y} = \begin{pmatrix} 1 \\ 0 \\ 2 \\ 1 \\ 0 \end{pmatrix}$$

*categorical\_crossentropy*:

$$\mathbf{y} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

# Deep Learning

## Ausblick

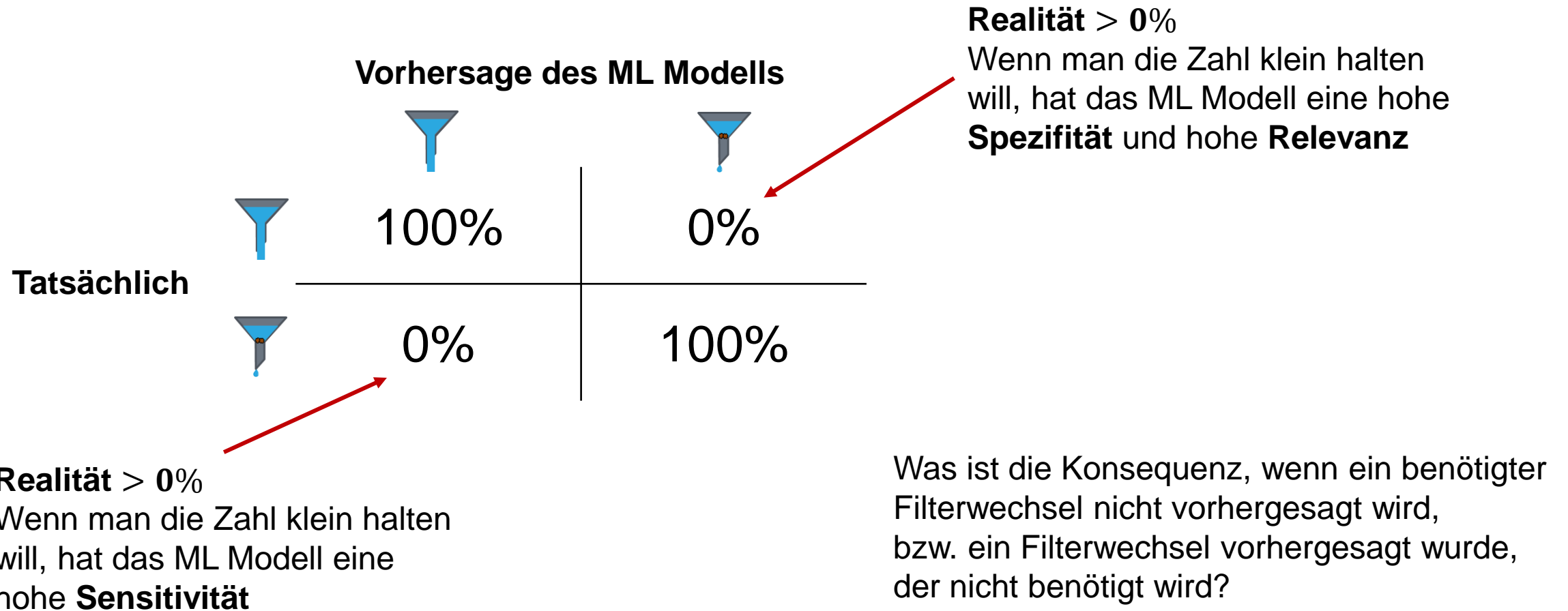


### Künstliche neuronale Netze: Part 1

- Einführung
- Training neuronaler Netze
  - Optimierungsalgorithmus
  - Aktivierungsfunktionen
  - Kostenfunktion
- **Performancemetriken**

# Künstliche Neuronale Netze - KNN

## Performancemetriken – Wie gut ist mein neuronales Netz?



# Künstliche Neuronale Netze - KNN

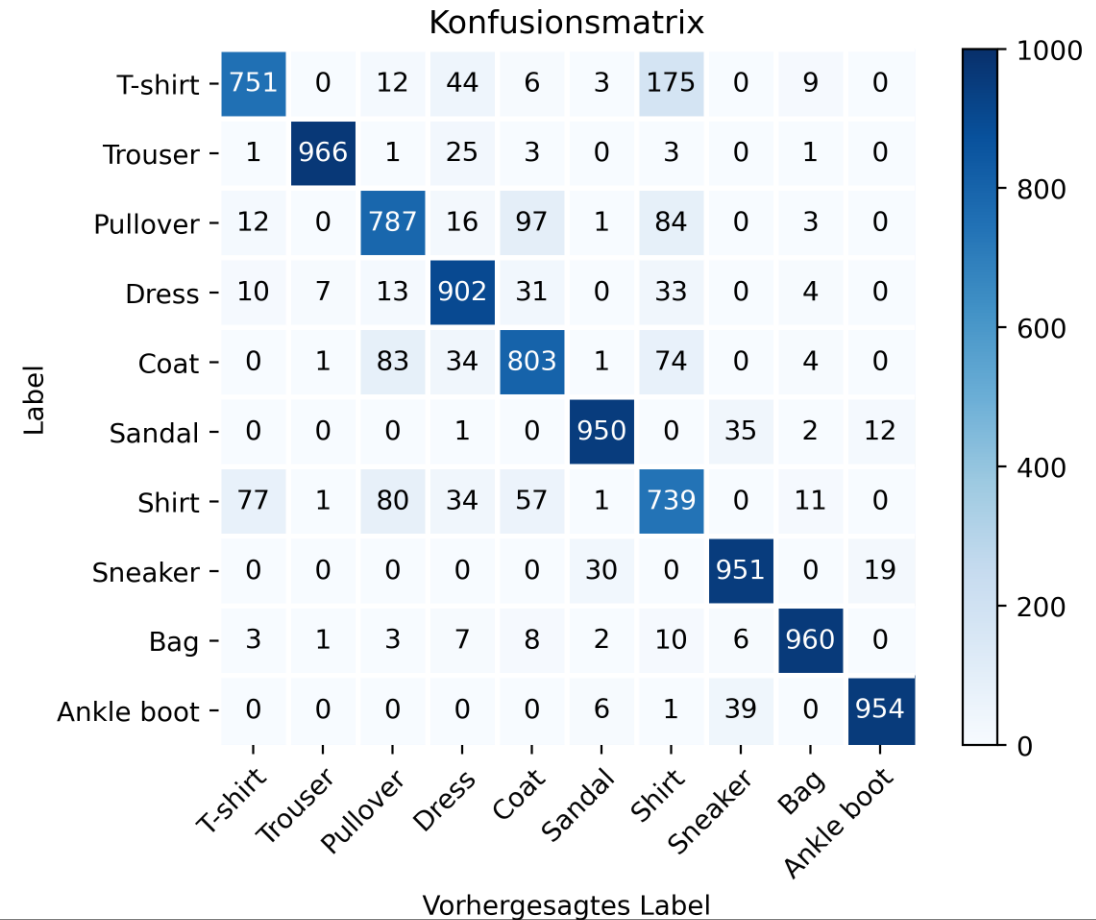
## Beispiel Bildklassifizierer: Konfusionsmatrix

```
y_pred = model.predict(x_test)
y_pred_classes = np.argmax(y_pred, axis=1)
```

```
# Berechnung der Confusion Matrix
cm = confusion_matrix(y_test, y_pred_classes)
```

```
array([[751, 0, 12, 44, 6, 3, 175, 0, 9, 0],
       [ 1, 966, 1, 25, 3, 0, 3, 0, 1, 0],
       [12, 0, 787, 16, 97, 1, 84, 0, 3, 0],
       [10, 7, 13, 902, 31, 0, 33, 0, 4, 0],
       [ 0, 1, 83, 34, 803, 1, 74, 0, 4, 0],
       [ 0, 0, 0, 1, 0, 950, 0, 35, 2, 12],
       [77, 1, 80, 34, 57, 1, 739, 0, 11, 0],
       [ 0, 0, 0, 0, 0, 30, 0, 951, 0, 19],
       [ 3, 1, 3, 7, 8, 2, 10, 6, 960, 0],
       [ 0, 0, 0, 0, 0, 6, 1, 39, 0, 954]])
```

s.: [04\\_fashion\\_mnist\\_10\\_confusion\\_matrix.ipynb](#)



# Künstliche Neuronale Netze - KNN

## Fragen II

Was ist wahr in Bezug auf die Rückwärtspropagation?

- Der Fehler in der Ausgabe wird nur rückwärts propagiert, um Gewichtsaktualisierungen festzulegen
- Ein Lernalgorithmus für neuronale Netze mit nur einer Feedforward-Schicht
- Lokale Minima und langsame Konvergenz können die Rückwärtspropagation einschränken

Basiert die Rückpropagation auf einem Gradientenabstieg entlang der Fehleroberfläche?

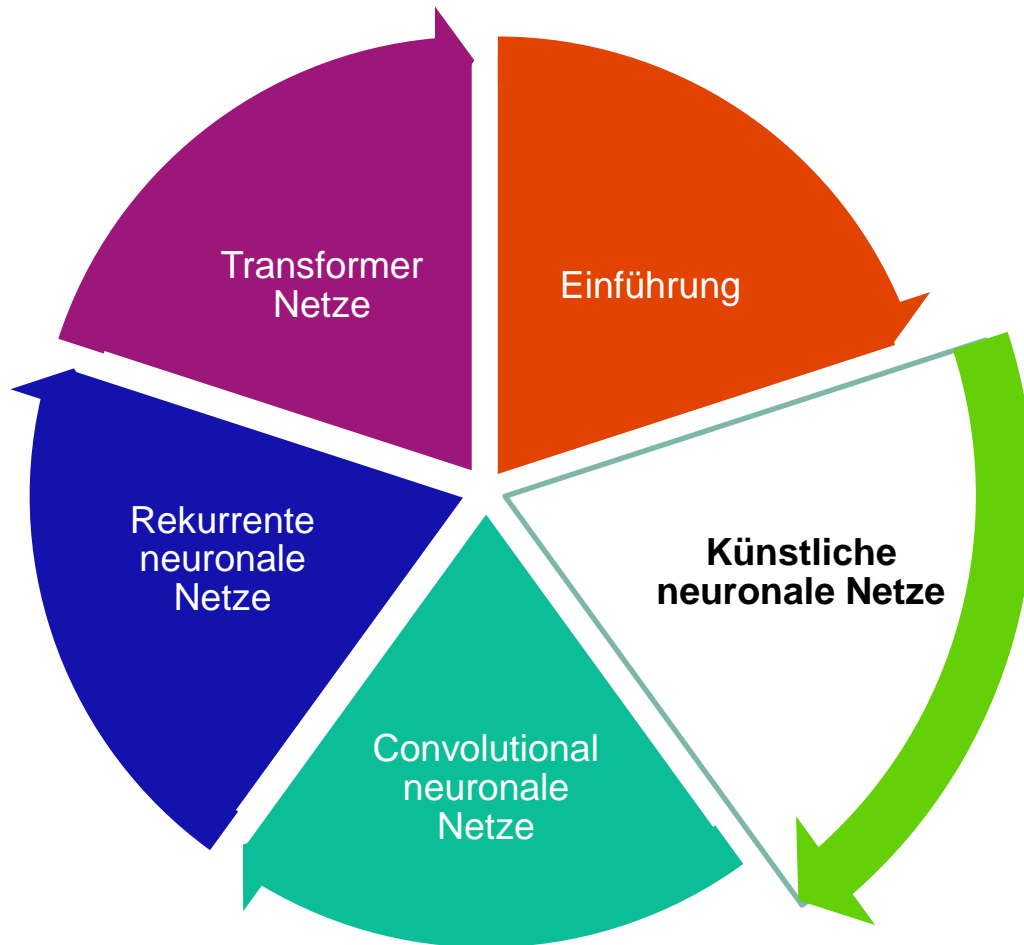
# Künstliche Neuronale Netze - KNN

## Zusammenfassung

- Wir können neuronale Netze für einfache binäre und multinomiale Klassifikationsprobleme in keras programmieren und trainieren.
  - Modell-Architektur: Anzahl Schichten, Anzahl Neuronen pro Schicht, Aktivierungsfunktion
  - Training: Backpropagation, Gradientenabstieg (Mini-Batch)
  
- Offene Fragen:
  - Woher wissen wir wie gut das neuronale Netz auf Testdaten performen wird?
  - Wie können wir Overfitting auf Trainingsdaten vermeiden?

# Deep Learning

## Ausblick



### Künstliche neuronale Netze: Part 2

- Trainings-, Test- und Validierungsdaten
- Regularisierung
  - **Early Stopping**
  - $l_1$  &  $l_2$ -Regularisierung
  - **Dropout**
  - Gewichtsinitialisierung
  - Batch Normalisierung
- Optimierungsalgorithmen

# Künstliche Neuronale Netze - KNN

## Trainings-, Validierungs- und Testdaten

Split des Datensatzes in Trainings-, Validierungs- und Testdaten:

- z.B. mit einem Verhältnis von 75:10:15 (Es müssen disjunkte Mengen sein)
- Trainiere das neuronale Netz mit den Trainingsdaten
- Validiere das neuronale Netz mit den Validierungsdaten (Kreuzvalidierung auch möglich)
- Endkontrolle auf den Testdaten (Test darf nur einmal erfolgen)

```
X_valid, X_train = X_train_full[:5000], X_train_full[5000:]  
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
```

Was heißt validieren?

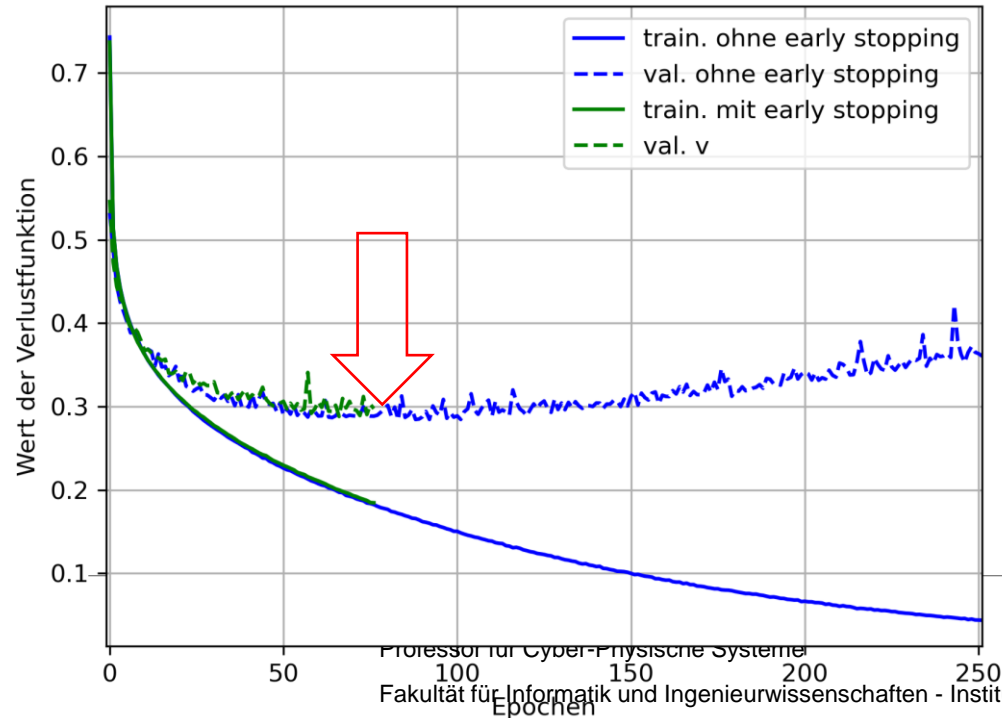
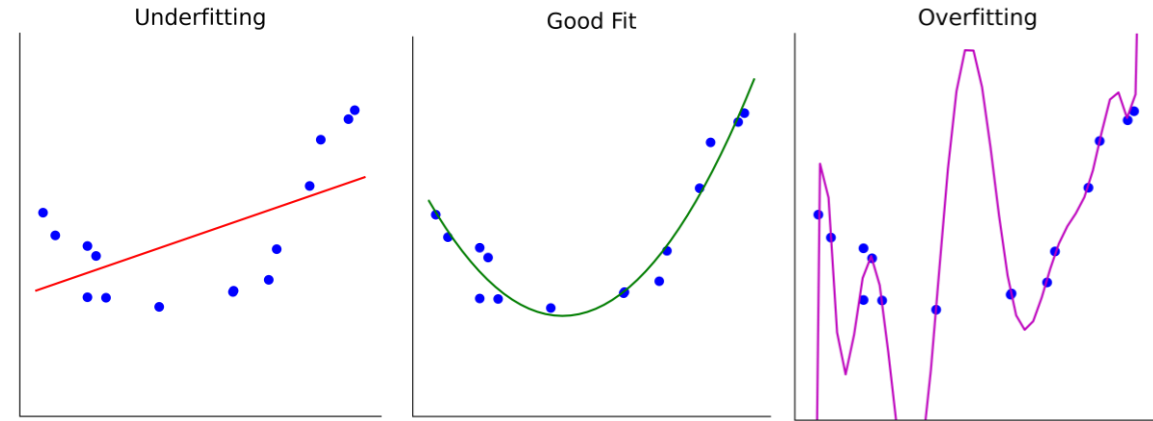
- Neuronale Netze haben Hyperparameter (Anzahl Schichten und Neuronen, Initialisierung der Gewichte, Aktivierungsfunktion, Datenvorverarbeitung, ...), welche eingestellt werden müssen
- Wann sollte ich das iterative Training beenden? (s. nächste Folie)

# Künstliche Neuronale Netze - KNN

## Regularisierung

Problemstellung:  
**Overfitting** auf Trainingsdaten → **Regularisierung**

- Early Stopping  
Ermittelt das beste Modell anhand des Validierungsfehlers



```
history_early_stopping = model_early_stopping.fit(  
    X_train,  
    y_train,  
    epochs=400,  
    callbacks=[tf.keras.callbacks.EarlyStopping(  
        monitor='val_loss',  
        min_delta=1e-3,  
        patience=10,  
        restore_best_weights=True)],  
    validation_data=(X_valid, y_valid))
```

Quelle: Patrick S., Hahn-Schickard

# Künstliche Neuronale Netze - KNN

## Regularisierung

$\ell_1$  &  $\ell_2$  - Regularisierung

- Erweitern der Kostenfunktion um ein Regularisierungsterm  $\Omega(\mathbf{W})$

$$\ell_2 : \Omega(\mathbf{W}) = \frac{\alpha}{2} \|\mathbf{W}\|_2^2 = \frac{\alpha}{2} \sum_i \sum_j w_{ij}^2$$

$$\ell_1 : \Omega(\mathbf{W}) = \alpha \|\mathbf{W}\|_1 = \alpha \sum_i \sum_j |w_{ij}|$$

Beispiel: binary cross entropy mit  $\ell_2$ -Regularisierung:

$$J_{\text{reg}}(\mathbf{w}, \mathbf{X}) = J(\mathbf{w}, \mathbf{X}) + \Omega(\mathbf{w}) = -\frac{1}{N} \sum_{i=0}^N (y^{(i)} \log(\hat{p}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)})) + \frac{\alpha}{2} \frac{1}{N} \sum_l \sum_i \sum_j (w_{ij}^{[l]})^2$$

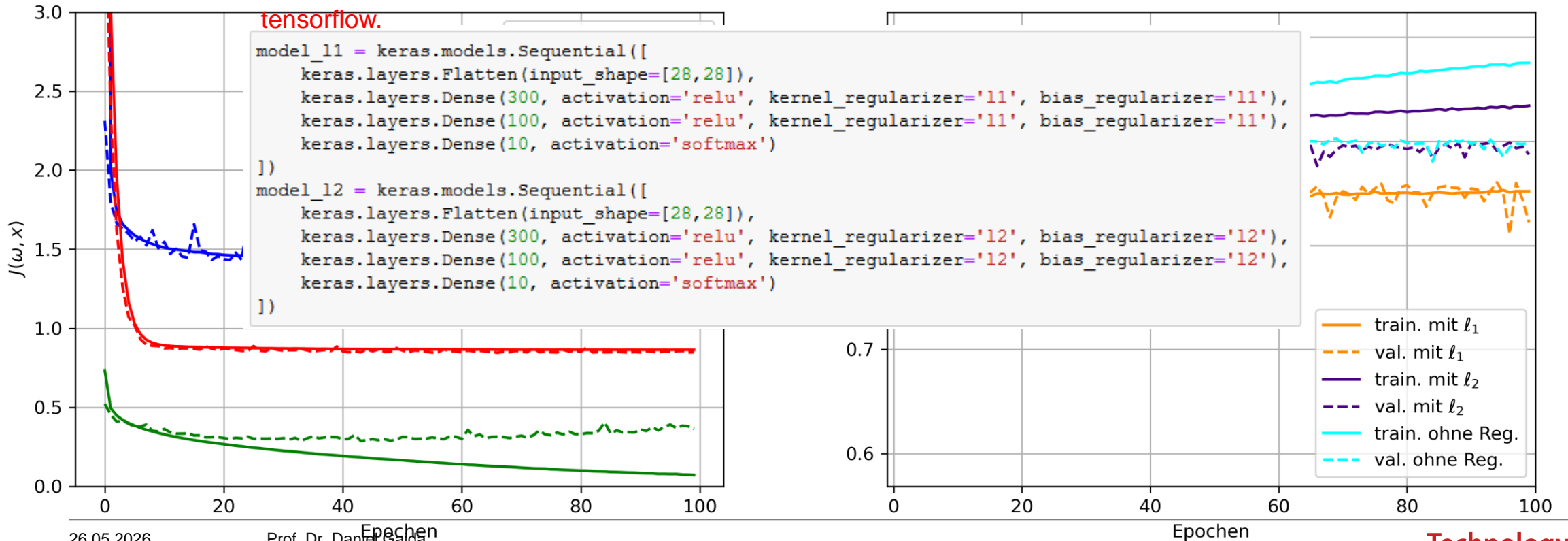
# Künstliche Neuronale Netze - KNN

## Beispiel Bildklassifizierer: Regularisierung

Vergleich zweier Modelle, eines mit Regularisierung eines ohne

Verlustfunktion

Metrik



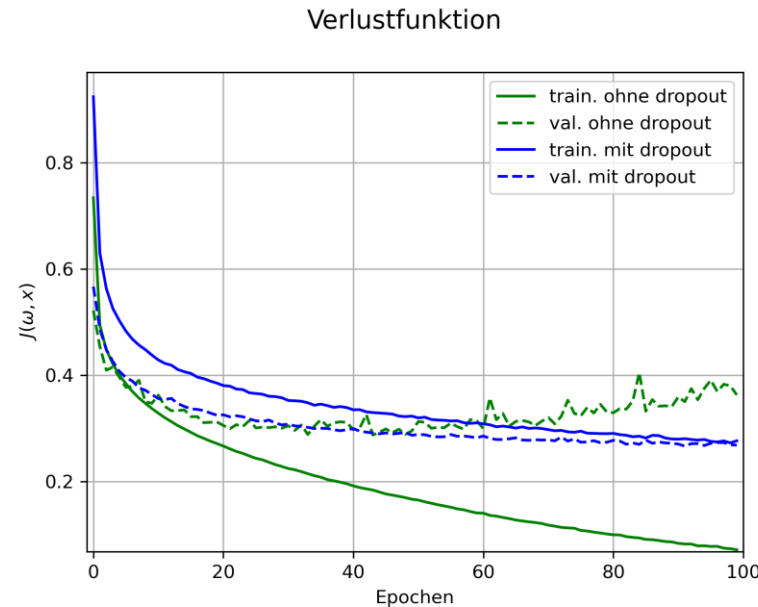
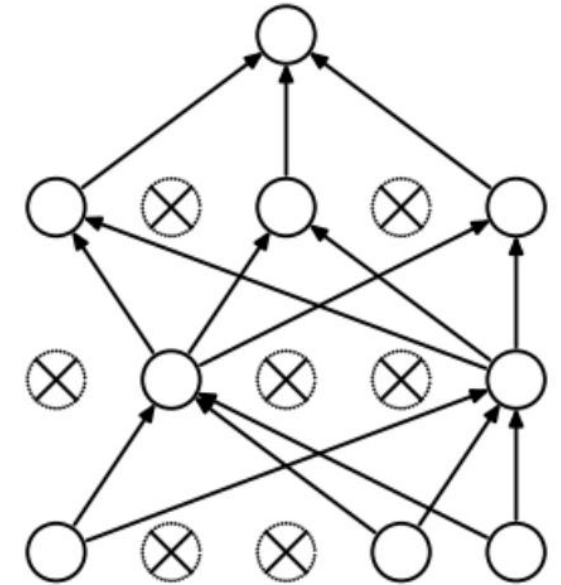
# Künstliche Neuronale Netze - KNN

## Regularisierung

**Dropout:** Zufälliges deaktivieren von Neuronen während des Trainings.  
 Reduziert Komplexität des neuronalen Netzes.

```
from tensorflow.keras.layers import Dropout
```

```
model = Sequential()
model.add(Flatten(input_shape=[28, 28]))
model.add(Dense(units=300, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(units=100, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(units=10, activation='softmax'))
```



# Künstliche Neuronale Netze - KNN

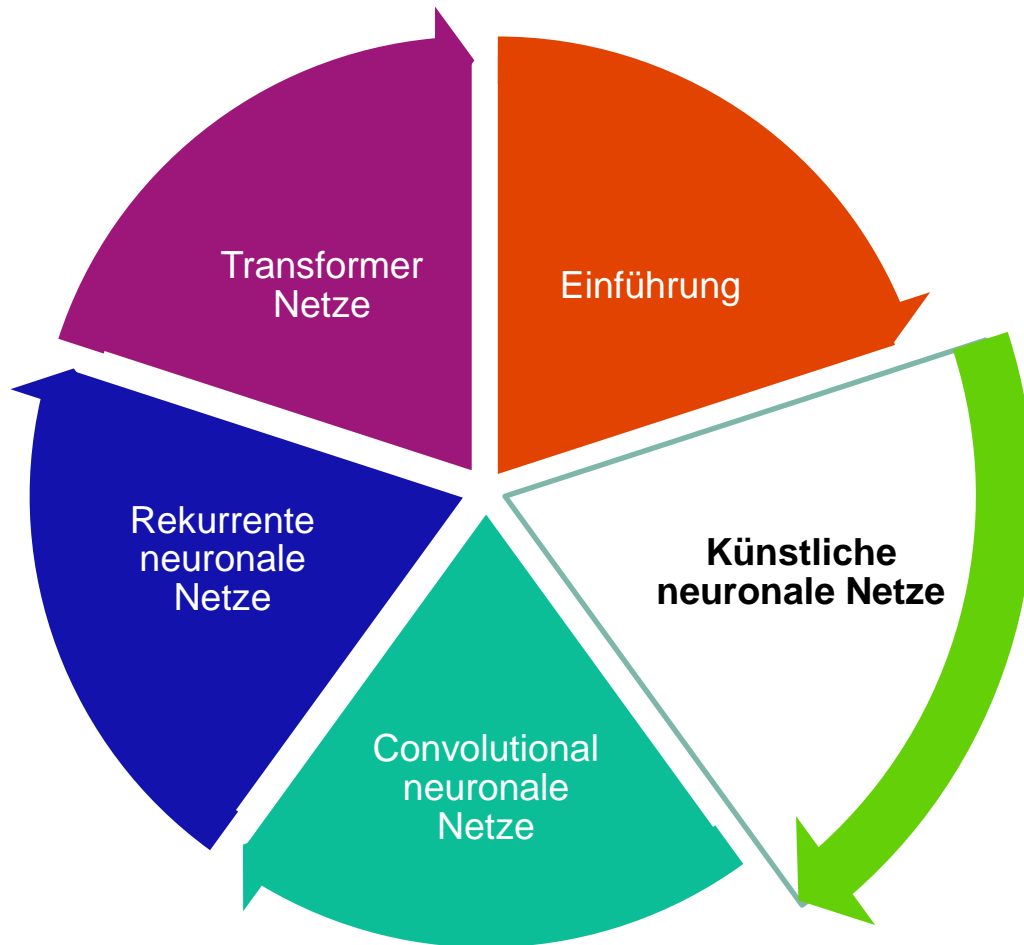
## Regularisierung: Übung

Experimentieren Sie mit verschiedenen Regularisierungsmethoden:

- Early Stopping
- l1, l2 Regularisierung - [05\\_fashion\\_mnist\\_10\\_l1\\_l2\\_regularizers.ipynb](#)
- Dropout - [06\\_fashion\\_mnist\\_10\\_dropout.ipynb](#)
  
- Integrieren Sie Early Stopping in die beiden anderen Skripte.
  - Experimentieren Sie mit den Argumenten `min_delta` und `patience`
  
- Wie unterscheiden sich die Ergebnisse?
  
- Wichtig: Nach jeder Änderung in `utils_fmnist.py` müssen Sie die Sitzung neu starten.

# Deep Learning

## Ausblick



### Künstliche neuronale Netze: Part 2

- Trainings-, Test- und Validierungsdaten
- Regularisierung
  - Early Stopping
  - $l_1$  &  $l_2$ -Regularisierung
  - Dropout
  - **Gewichtsinitialisierung**
  - **Batch Normalisierung**
- Optimierungsalgorithmen

# Deep Learning - Zutaten

## Modellarchitektur

- Anzahl Schichten
- Aktivierungsfunktion

## Kostenfunktion

Definiert was ein gutes neuronales Netz ist

## Optimierungsalgorithmus

Minimiert die Kostenfunktion, indem es Gewichte des NNs variiert

## Training des NNs

Minimierung der Kostenfunktion

# Künstliche Neuronale Netze - KNN

## Optimierungsalgorithmen I

- (Vanilla) Gradientenabstieg (Mini-Batch)
  - Gradientenabstieg mit Momentum
    - Berücksichtigt vorherige Gradienten (Momentvektor),  $0 < \beta < 1$
    - Schnellere Konvergenz und weniger Oszillation
  - Nesterov Accelerated Gradient (NAG)
    - Messung des Gradienten an  $\mathbf{w} + \beta \mathbf{m}$  statt  $\mathbf{w}$
    - $\mathbf{m}$  zeigt gewöhnlich in Richtung Optimum
    - Fast immer schneller als Vanilla Momentum Optimierung
    - Reduziert Oszillationen
  - RMSProp (Root Mean Square Propagation)
    - Gradient wird durch exponentiell gewichteten Mittelwert der quadrierten Gradienten skaliert
- $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} J(\mathbf{w})$
  - $\mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\mathbf{w}} J(\mathbf{w})$
  - $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{m}$
  - $\mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\mathbf{w}} J(\mathbf{w} + \beta \mathbf{m})$
  - $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{m}$
  - $\mathbf{s} \leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_{\mathbf{w}} J(\mathbf{w}) \otimes \nabla_{\mathbf{w}} J(\mathbf{w})$
  - $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} J(\mathbf{w}) \oslash \sqrt{\mathbf{s} + \epsilon}$

# Künstliche Neuronale Netze - KNN

## Optimierungsalgorithmen I

- RMSProp
- Gradient wird durch exponentiell gewichteten gleitenden Mittelwert der quadrierten Gradienten skaliert

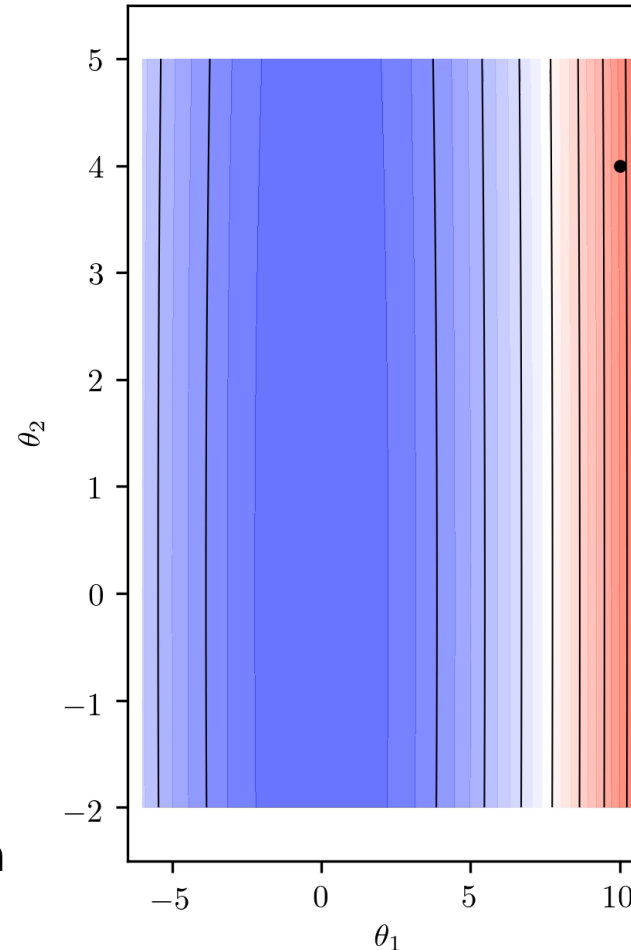
- $\mathbf{s} \leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_{\mathbf{w}} J(\mathbf{w}) \otimes \nabla_{\mathbf{w}} J(\mathbf{w})$

- $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} J(\mathbf{w}) \oslash \sqrt{\mathbf{s} + \epsilon}$

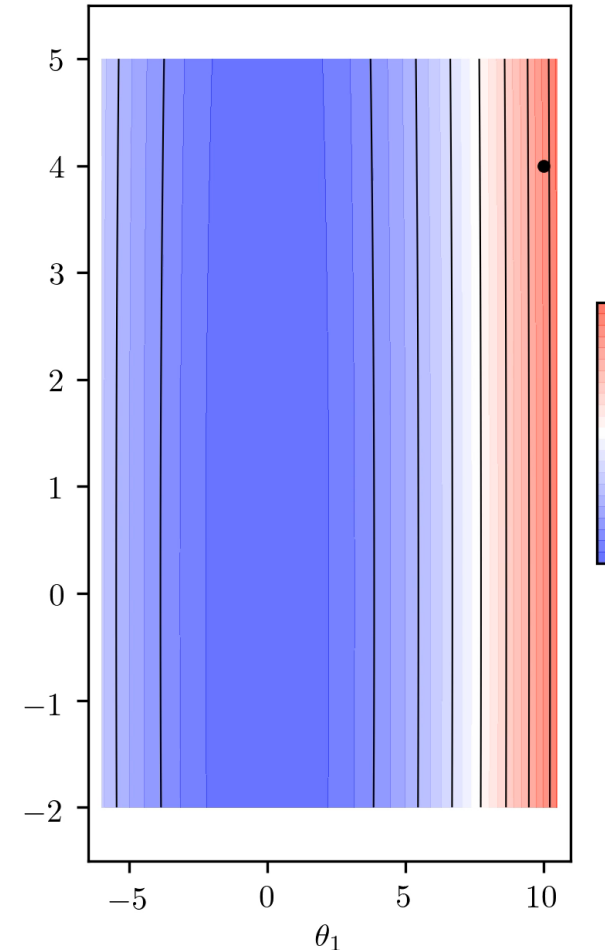


Komponentenweise Multiplikation und Division

Alg.: RMSProp,  $\eta$ : 0.40,  $\beta_1$ : 0.9  
epoch: 0,  $J(\theta)$ : 100.6,  $\Delta\theta_k$ : 1.79



Alg.: GD,  $\eta$ : 0.40,  
epoch: 0,  $J(\theta)$ : 100.6,  $\Delta\theta_k$ : 8.00



# Künstliche Neuronale Netze - KNN

## Optimierungsalgorithmen II

### 1 Adam Algorithmus

Der Adam Algorithmus wird in Algorithmus 1 dargestellt. Die Berechnung des 1. und des 2. Moments sind aus dem Momentum Optimierung Algorithmus bzw. des RMSProp Algorithmus bekannt.

---

#### Algorithm 1 Adam Algorithmus

---

**Require:**  $\eta > 0$ : Schrittweite

**Require:**  $\beta_1, \beta_2 \in [0, 1)$ : Exponentielle Abbauraten für 1. und 2. Moment

**Require:**  $J(\omega)$ : Zielfunktion

**Require:**  $\omega_0$ : Startvektor

$\mathbf{m}_0 \leftarrow 0$  (1. Moment auf Null initialisieren)

$\mathbf{v}_0 \leftarrow 0$  (2. Moment auf Null initialisieren)

$t \leftarrow 0$  (Zeitschritt auf Null initialisieren)

**while**  $\omega_t$  not converged **do**

$t \leftarrow t + 1$  ▷ Aktualisiere Zeitschritt

$\mathbf{g}_t \leftarrow \nabla_{\omega} J(\omega_{t-1})$  ▷ Erhalte Gradienten in Schritt t

$\mathbf{m}_t \leftarrow \beta_1 \cdot \mathbf{m}_{t-1} + (1 - \beta_1) \cdot \mathbf{g}_t$  ▷ Aktualisiere 1. Moment

$\mathbf{v}_t \leftarrow \beta_2 \cdot \mathbf{v}_{t-1} + (1 - \beta_2) \cdot \mathbf{g}_t \odot \mathbf{g}_t$  ▷ Aktualisiere 2. Moment

$\widehat{\mathbf{m}}_t \leftarrow \mathbf{m}_t / (1 - \beta_1^t)$  ▷ Bias im 1. Moment korrigieren

$\widehat{\mathbf{v}}_t \leftarrow \mathbf{v}_t / (1 - \beta_2^t)$  ▷ Bias im 2. Moment korrigieren

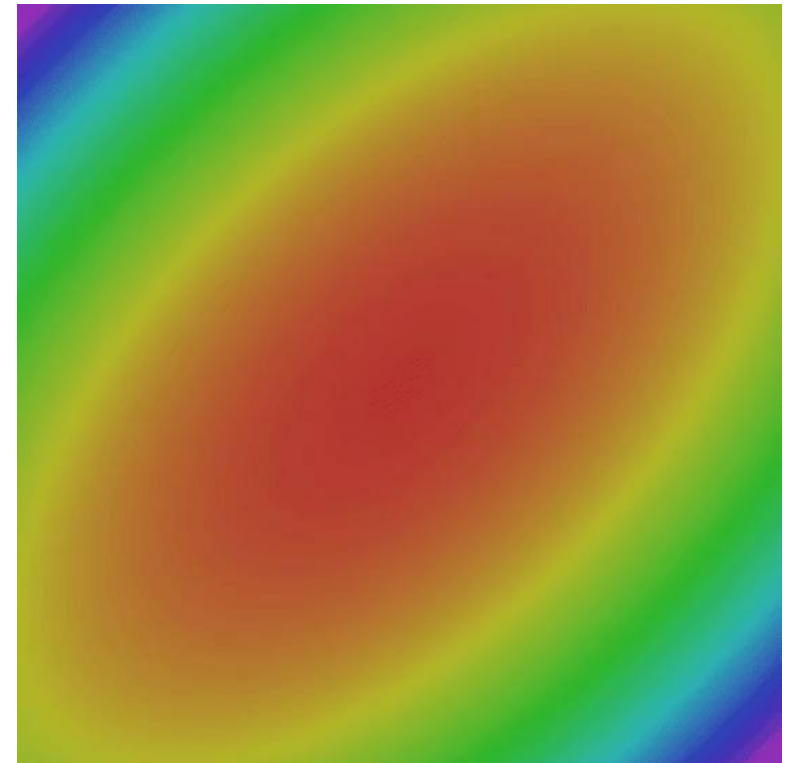
$\omega_t \leftarrow \omega_{t-1} - \eta \cdot \widehat{\mathbf{m}}_t \odot (\sqrt{\widehat{\mathbf{v}}_t} + \epsilon)$  ▷ Aktualisiere die Parameter

**end while**

**return**  $\omega_t$  ▷ Rückgabe der Endparameter

---

— SGD — RMSProp  
 — SGD+Momentum — Adam



# Künstliche Neuronale Netze - KNN

## Optimierungsalgorithmen in Keras

```
# Adadelta
...
optimizer="Adadelta"
optimizer=keras.optimizers.Adadelta()
...

# Adagrad
...
optimizer="Adagrad"
optimizer=keras.optimizers.Adagrad()
...

# Adam
...
optimizer="Adam"
optimizer=keras.optimizers.Adam()
...
```

```
# Nadam
...
optimizer="Nadam"
optimizer=keras.optimizers.Nadam()
...

# RMSprop
...
optimizer="RMSprop"
optimizer=keras.optimizers.RMSprop()
...

# SGD
...
optimizer="SGD"
optimizer=keras.optimizers.SGD()
...
```

# Künstliche Neuronale Netze - KNN

## Trainingsverfahren Zusammenfassung

### Trainingsverfahren

1. Gewichte initialisieren (zufällig oder nach einem Schema)
2. Propagiere Trainingsdaten vorwärts durch das Netz
3. Berechne den Fehler zwischen der Zielausgabe und der Prädiktion
4. Propagiere den Fehler rückwärts und berechne den anteiligen Fehler aller Neuronen
5. Aktualisiere die Gewichte nach einem Optimierungsverfahren (Gradientenabstieg)
6. Ist der Validierungsfehler größer als ein gewünschter Betrag, springe zu Punkt 2.  
Ansonsten beende das Training

# Künstliche Neuronale Netze - KNN

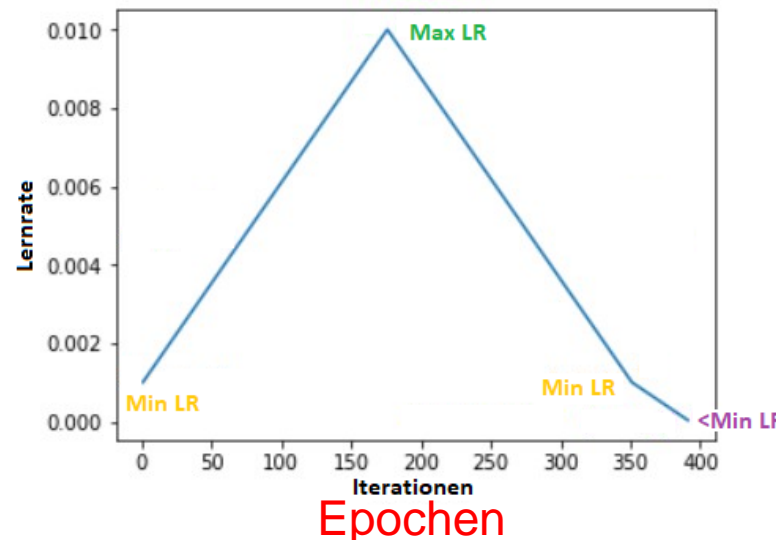
Quelle: Géron, Aurélien. Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems. O'Reilly Media, 2019.

## Praxisleitfaden

Hyperparameter	Standardwert
Gewichtsinitialisierung	He Normal
Aktivierungsfunktion	ReLU; Swish
Normalisierung	Batchnormalisierung bei Deep
Regularisierung	Early Stopping (+ $\ell_2$ falls nötig)
Optimierer	NAG oder AdamW
Lernrate	Ein-Zyklus-Politik (1Cycle policy)*

### Ein-Zyklus-Politik

- Maximale Lernrate: LR range test
- Minimale Lernrate:  $\frac{1}{5}$  oder  $\frac{1}{10}$  der maximalen Lernrate
- Zykluslänge: < Totalanzahl an Epochen
- Letzte Iterationen:  $LR < \frac{1}{10}$  oder  $\frac{1}{100}$



# Künstliche Neuronale Netze - KNN

## Fragen III

Welche Aussagen über lineare Aktivierungsfunktionen sind wahr?

- Nur binäre Ausgaben sind möglich
- Ermöglicht nicht-lineare Zuordnung zwischen Ein- und Ausgängen
- Gradientenabstieg ist nicht möglich, da die Ableitung eine Konstante ist

# Künstliche Neuronale Netze - KNN

## Bildklassifizierer mit Keras

MNIST Fashion Datensatz

Vollständiges Beispiel:

[07\\_fashion\\_mnist\\_classification.ipynb](#)



# Nächste Vorlesung und Fragen

- Convolutional Neuronale Netze
- Fragen?

# Künstliche Neuronale Netze - KNN

## Notationen

- $N$ : Anzahl an (Trainings-)Instanzen
- $p$ : Dimension der Eingabe (Anzahl der Features)
- $K$ : Dimension der Ausgabe (oder Anzahl der Klassen)
- $L$ : Anzahl der Schichten eines neuronalen Netzes
- $\mathbf{x}^{(i)} \in \mathbb{R}^p / \mathbf{y}^{(i)} \in \mathbb{R}^K$  : der  $i$ -te Eingabevektor / Ausgabevektor dargestellt als Zeilenvektor
- $x_j^{(i)} / y_k^{(i)}$  : Das  $j$ -te Element des  $i$ -te Eingabevektors / Ausgabevektor (Skalar)
- $\mathbf{X} \in \mathbb{R}^{N \times p} / \mathbf{Y} \in \mathbb{R}^{N \times K} : \mathbf{X} = \begin{pmatrix} (\mathbf{x}^{(1)})^T \\ \vdots \\ (\mathbf{x}^{(N)})^T \end{pmatrix} = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_p^{(1)} \\ \vdots & \vdots & & \vdots \\ x_1^{(N)} & x_2^{(N)} & \dots & x_p^{(N)} \end{pmatrix}$  Eingabematrix
- $\omega_{ji}^{[\ell]}$ : Gewicht des  $j$ -ten Eingangs des  $i$ -ten Neurons der  $\ell$ -ten Schicht (alternativ  $\theta$ )
- $\mathbf{W}^{[\ell]} \in \mathbb{R}^{(\text{Anzahl Neuronen der Schicht } \ell) \times (\text{Anzahl Neuronen der Schicht } \ell-1)}$ : Gewichtsmatrix der Schicht  $\ell$ .
- $\hat{\mathbf{y}}^{(i)} \in \mathbb{R}^K$ ,  $\hat{\mathbf{y}}^{(i)} = h(\mathbf{x}^{(i)})$ : Ist der vorhergesagte Ausgabevektor (Estimator)  
 $h$ : ist die als Hypothese bezeichnete Vorhersagefunktion Ihres Systems
- $\sigma^{[\ell]}(\cdot)$ : Aktivierungsfunktion der  $\ell$ -ten Schicht, für die Stufenfunktion schreiben wir  $\sigma(\cdot) = f(\cdot)$

26.05.2026

Prof. Dr. Daniel Gaida  
Professor für Cyber-Physische Systeme