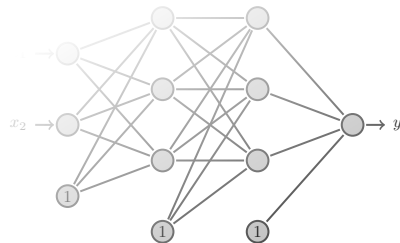
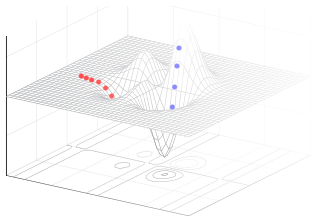


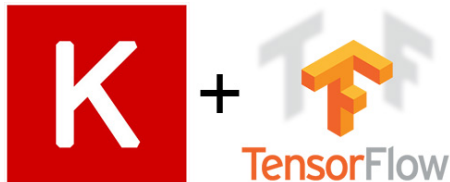
# Dichte Neuronale Netze mit Keras

Prof. Dr. Jörg Frochte

Maschinelles Lernen

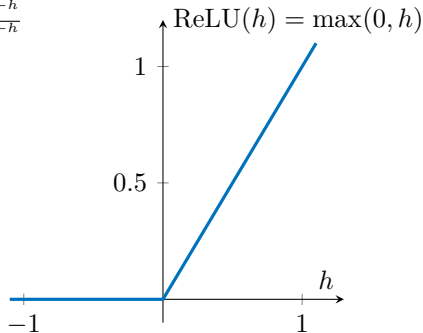
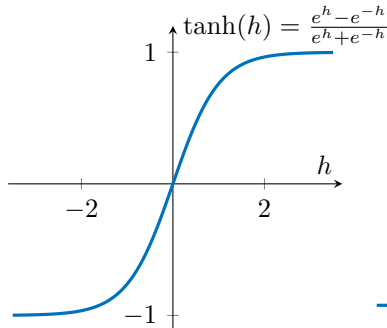
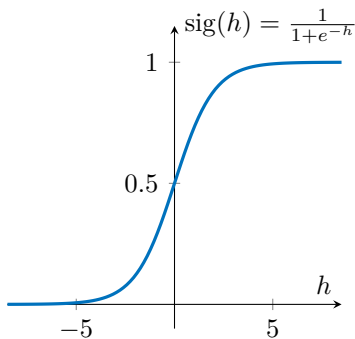


- Wir beginnen damit, dichte Netze, also die Architektur, die wir schon kennengelernt haben, mit **Keras** zu erstellen.
- **Keras**: eine quelloffene Python-Bibliothek für neuronale Netzwerke.
- Keras ist dabei primär ein Frontend mit dem Ziel, intuitiv und im positiven Sinne abstrakt zu sein. Als Backend kommen verschiedene Technologien in Frage, u. a. **TensorFlow**.
- Durch die Verwendung von Keras sind wir in der Lage, auch so genannte Deep Neuronal Networks zu bearbeiten.
- Mit handgeschriebenem Code kommt man hier schnell an Grenzen, u. a. bzgl. der Optimierung zum Bestimmen der Gewichte.
- Daneben erfolgt eine weitergehende Parallelisierung und ggf. die Unterstützung von GPUs bei der Berechnung.



# Aktivierungsfunktionen

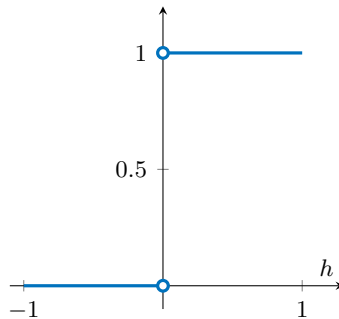
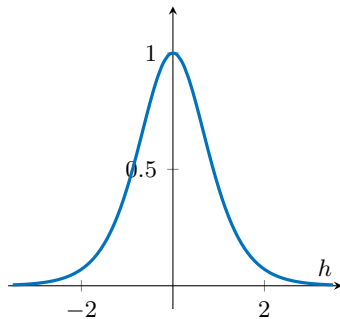
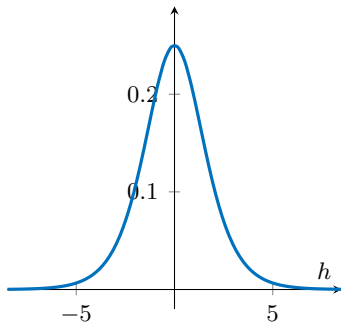
- Durch die Verwendung von Keras vergrößert sich unsere Auswahl an Aktivierungsfunktionen deutlich
- Wir nehmen hier zunächst nur ReLU dazu und verschieben andere auf einen späteren Foliensatz



# Ableitungen der Aktivierungsfunktionen

- In unserem handgeschriebenen Code hätte die unstetige Ableitung ggf. zu besonderen Herausforderungen geführt

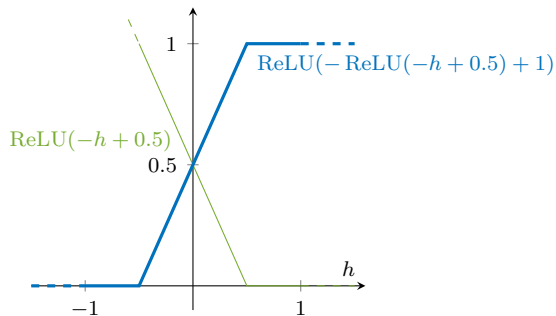
$$\text{sig}'(h) = \text{sig}(h) (1 - \text{sig}(h)) \quad \text{tanh}'(h) = (1 + \text{tanh}(h)) (1 - \text{tanh}(h)) \quad \text{ReLU}'(h) = H(h)$$



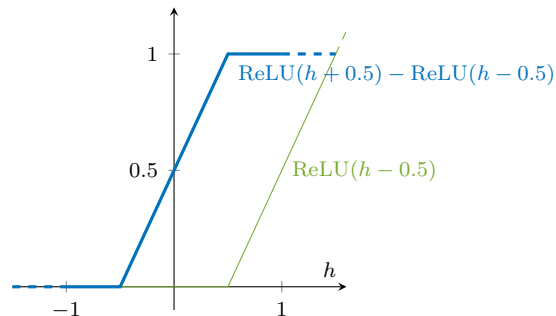
# Rectified-Linear-Unit-Ansatzfunktionen

- ReLU hilft gegen das Problem der Sättigung bzw. des verschwindenden Gradienten.
- sig und tanh sind differenzierbar und trennen Mengen oft mit weniger Neuronen.
- Das liegt daran, dass zwei ReLU-Ansatzfunktionen benötigt werden um ein ähnliches Verhalten wie bei den anderen Aktivierungsfunktionen zu erreichen.

Hinternanderausführung

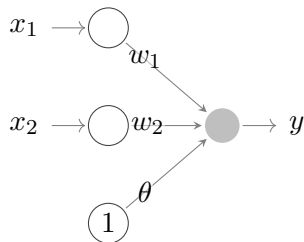


Nebeneinanderausführung



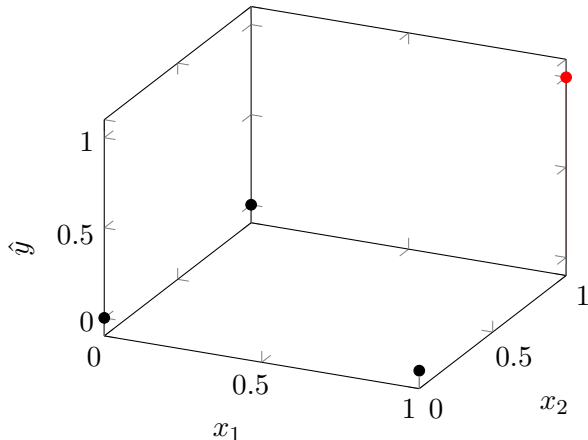
## Beispiel: AND lernen

Wir schauen uns nun das Training von einer AND-Funktion mit einem einzigen Neuron an.



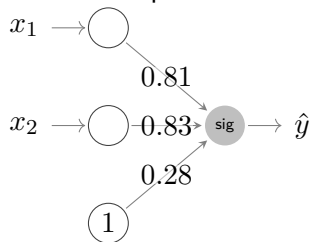
$x_1$	$x_2$	$y$
0	0	0
0	1	0
1	0	0
1	1	1

- Gewichte werden “zufällig” initialisiert.
- Gewichte werden nach jedem Sample geupdatet (“echtes SGD”).
- Nach den vier Samples ist jeweils eine Epoche vorbei und wir betrachten die Änderung.



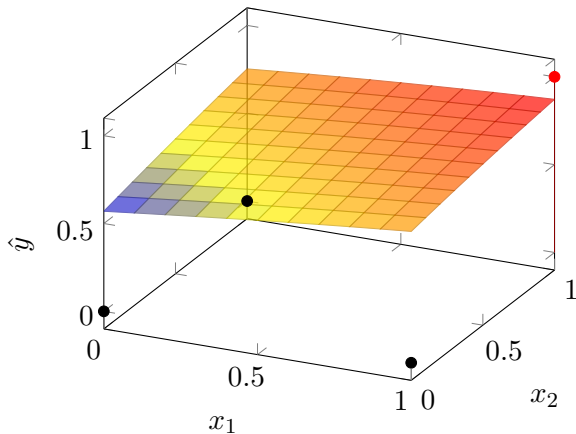
# Beispiel: AND lernen mit Sigmoid

Epoch: 0



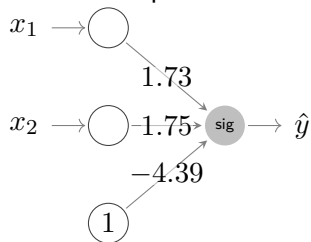
$x_1$	$x_2$	$\hat{y}$
0	0	0.57
0	1	0.752
1	0	0.748
1	1	0.872

Predictions:

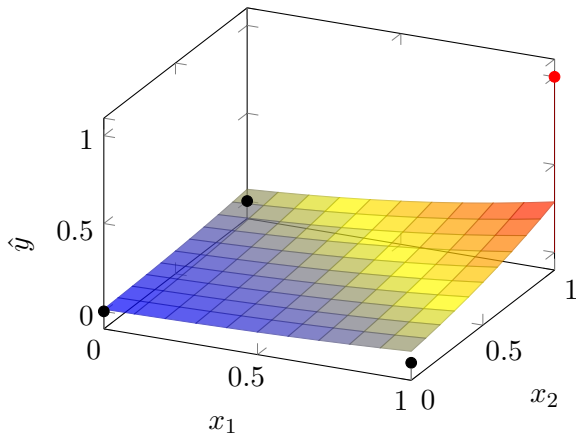


# Beispiel: AND lernen mit Sigmoid

Epoch: 1



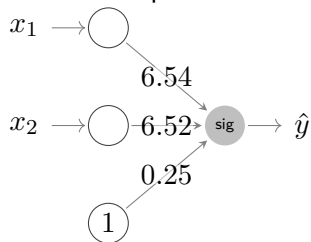
	$x_1$	$x_2$	$\hat{y}$
Predictions:	0	0	0.012
	0	1	0.067
	1	0	0.065
	1	1	0.288



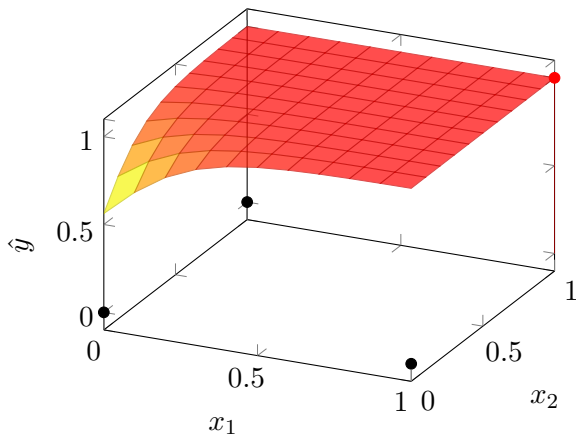


# Beispiel: AND lernen mit Sigmoid

Epoch: 2

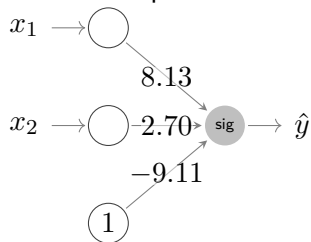


	$x_1$	$x_2$	$\hat{y}$
Predictions:	0	0	0.563
	0	1	0.999
	1	0	0.999
	1	1	1

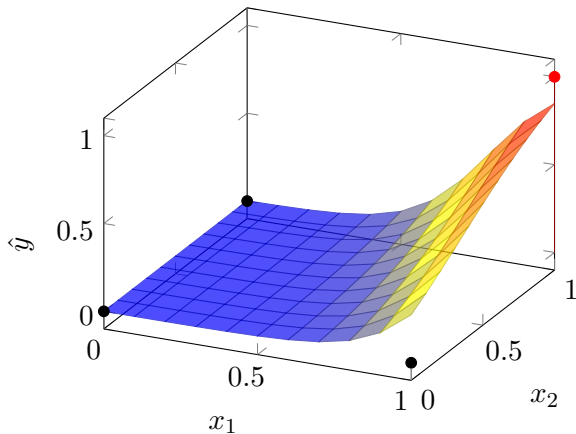


# Beispiel: AND lernen mit Sigmoid

Epoch: 3

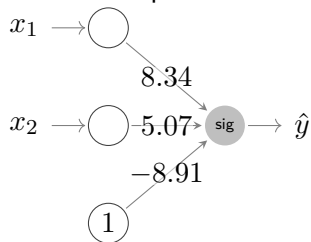


	$x_1$	$x_2$	$\hat{y}$
Predictions:	0	0	0
	0	1	0.002
	1	0	0.273
	1	1	0.848

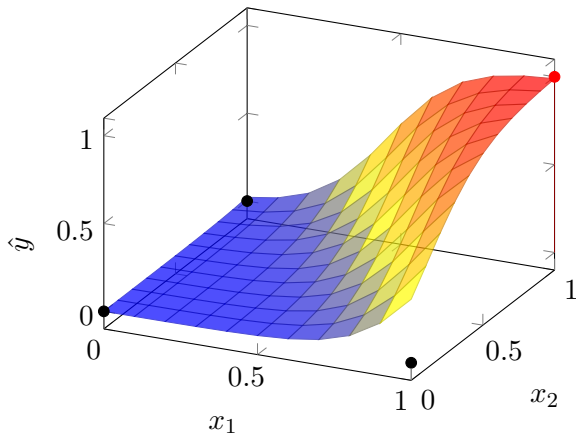


# Beispiel: AND lernen mit Sigmoid

Epoch: 4

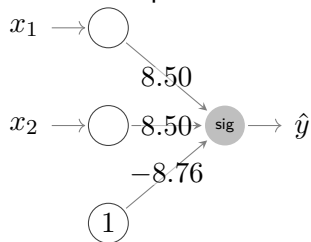


	$x_1$	$x_2$	$\hat{y}$
Predictions:	0	0	0
	0	1	0.021
	1	0	0.362
	1	1	0.989



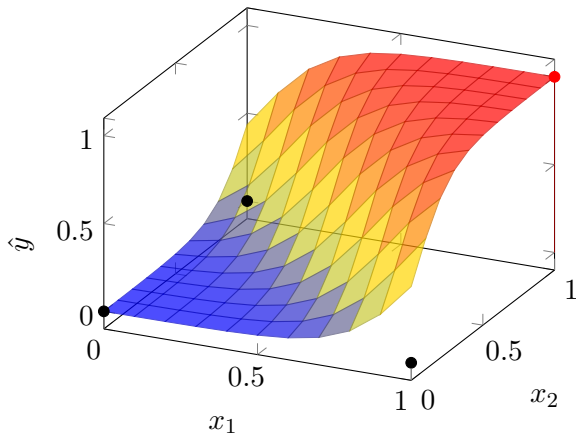
# Beispiel: AND lernen mit Sigmoid

Epoch: 5



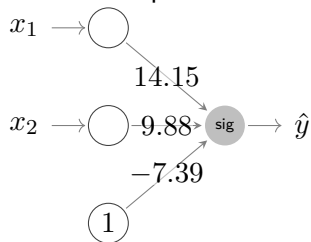
$x_1$	$x_2$	$\hat{y}$
0	0	0
0	1	0.435
1	0	0.434
1	1	1

Predictions:

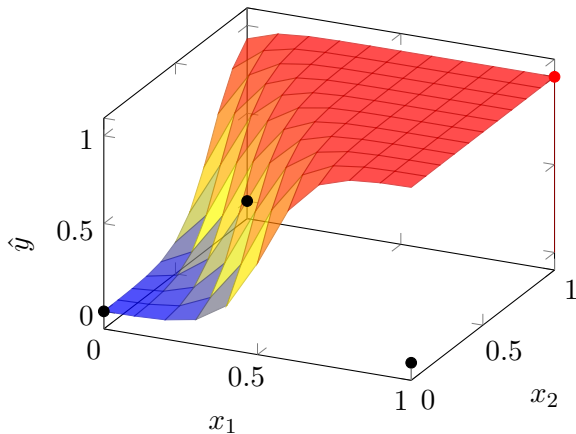


# Beispiel: AND lernen mit Sigmoid

Epoch: 6

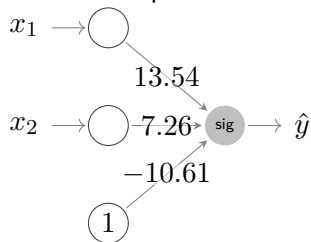


	$x_1$	$x_2$	$\hat{y}$
Predictions:	0	0	0
	0	1	0.923
	1	0	0.999
	1	1	1

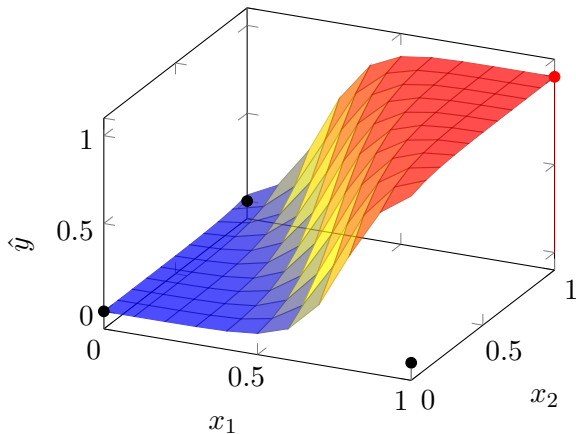


# Beispiel: AND lernen mit Sigmoid

Epoch: 7

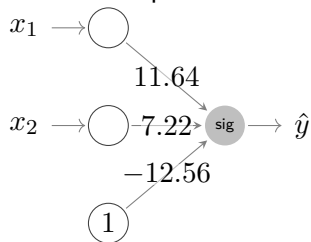


	$x_1$	$x_2$	$\hat{y}$
Predictions:	0	0	0
	0	1	0.034
	1	0	0.949
	1	1	1

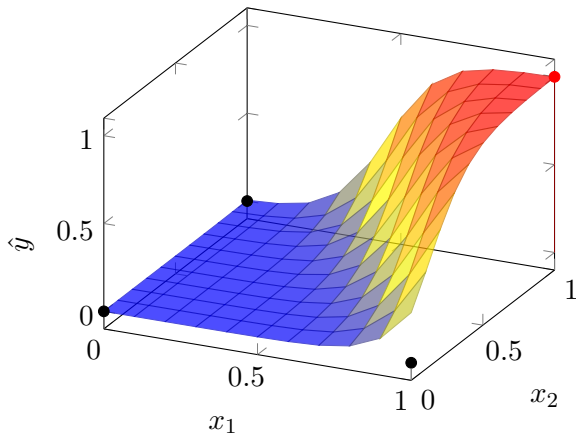


# Beispiel: AND lernen mit Sigmoid

Epoch: 8

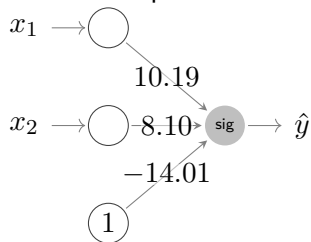


	$x_1$	$x_2$	$\hat{y}$
Predictions:	0	0	0
	0	1	0.005
	1	0	0.286
	1	1	0.998

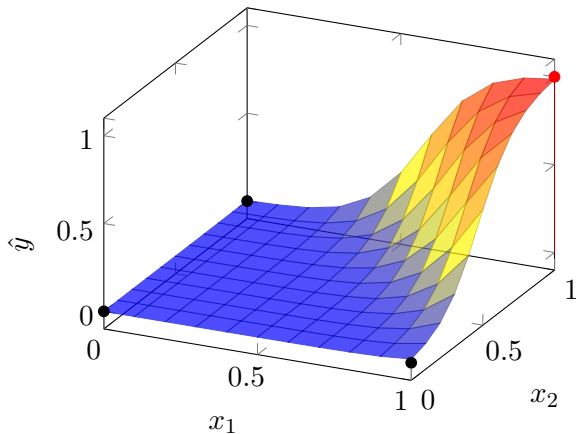


# Beispiel: AND lernen mit Sigmoid

Epoch: 9



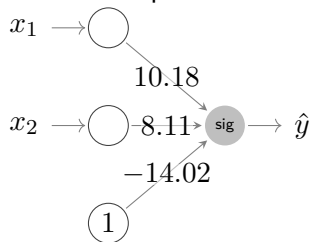
	$x_1$	$x_2$	$\hat{y}$
Predictions:	0	0	0
	0	1	0.003
	1	0	0.022
	1	1	0.986



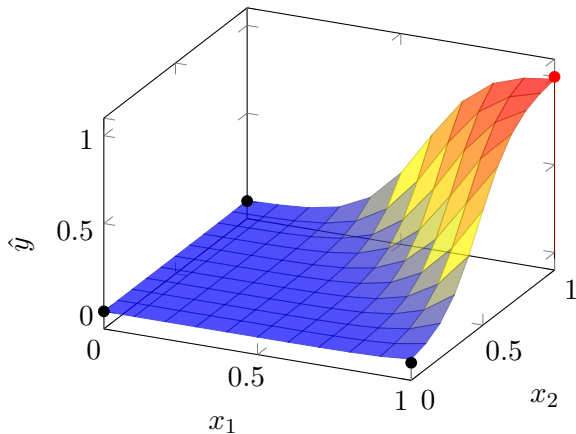


# Beispiel: AND lernen mit Sigmoid

Epoch: 10

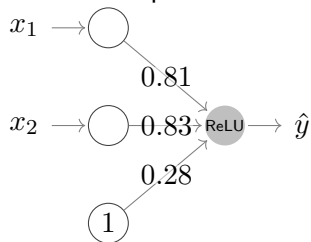


	$x_1$	$x_2$	$\hat{y}$
Predictions:	0	0	0
	0	1	0.003
	1	0	0.021
	1	1	0.986



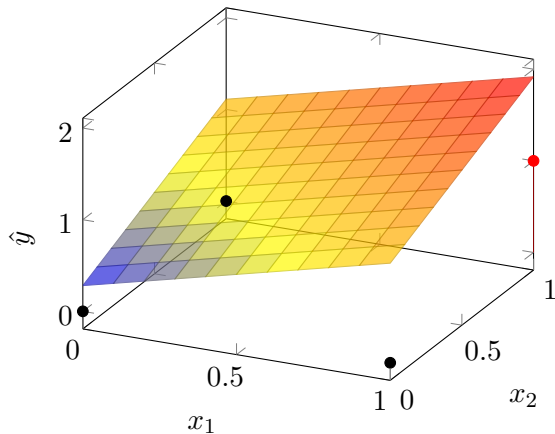
# Beispiel: AND lernen mit ReLU

Epoch: 0



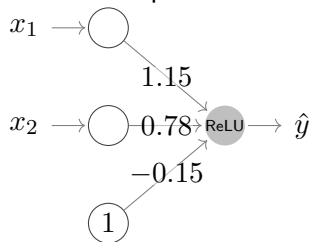
$x_1$	$x_2$	$\hat{y}$
0	0	0.281
0	1	1.111
1	0	1.087
1	1	1.917

Predictions:

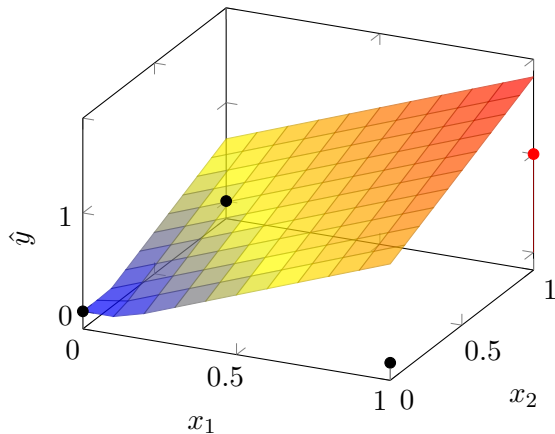


# Beispiel: AND lernen mit ReLU

Epoch: 1

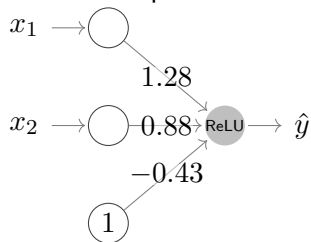


	$x_1$	$x_2$	$\hat{y}$
Predictions:	0	0	0
	0	1	0.632
	1	0	1.005
	1	1	1.785



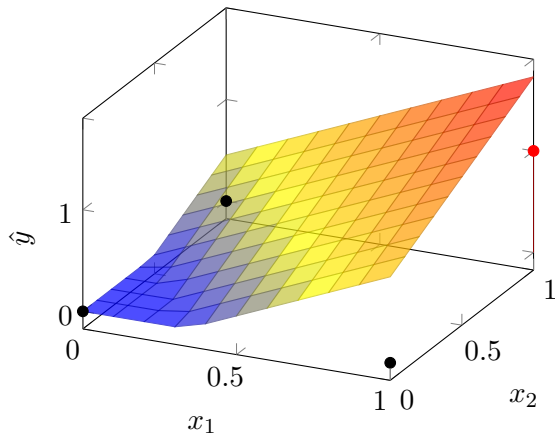
# Beispiel: AND lernen mit ReLU

Epoch: 2



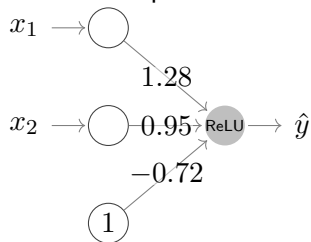
$x_1$	$x_2$	$\hat{y}$
0	0	0
0	1	0.453
1	0	0.846
1	1	1.729

Predictions:



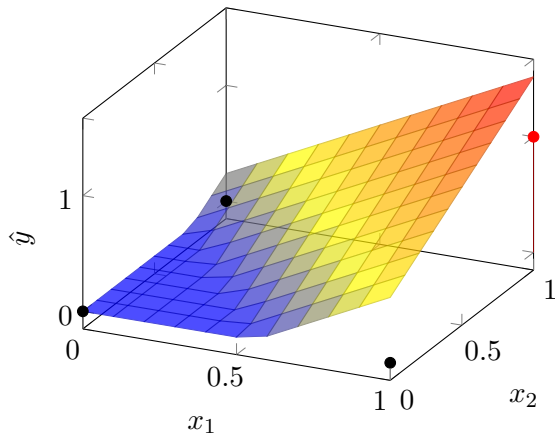
# Beispiel: AND lernen mit ReLU

Epoch: 3



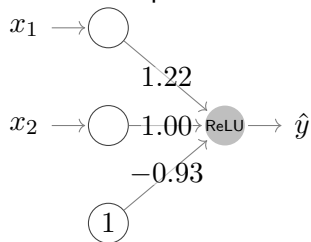
$x_1$	$x_2$	$\hat{y}$
0	0	0
0	1	0.238
1	0	0.565
1	1	1.518

Predictions:

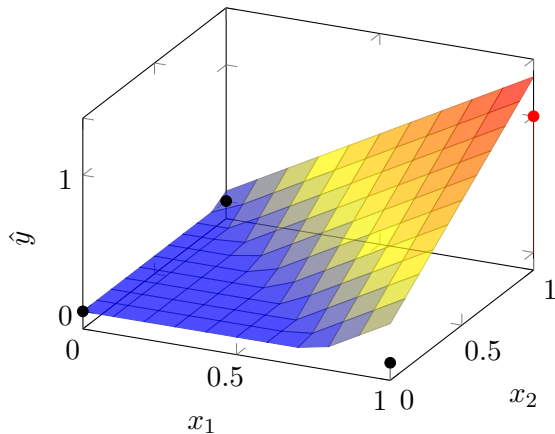


# Beispiel: AND lernen mit ReLU

Epoch: 4

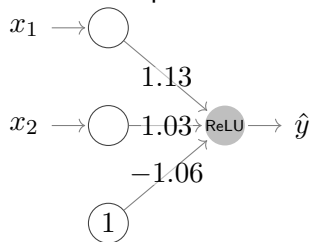


	$x_1$	$x_2$	$\hat{y}$
Predictions:	0	0	0
	0	1	0.072
	1	0	0.287
	1	1	1.29



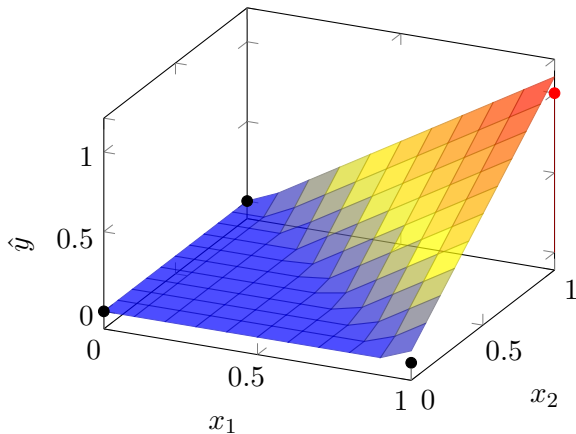
# Beispiel: AND lernen mit ReLU

Epoch: 5



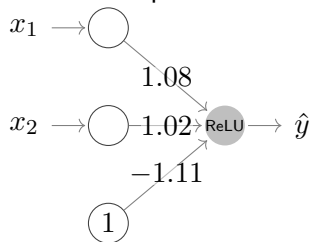
$x_1$	$x_2$	$\hat{y}$
0	0	0
0	1	0
1	0	0.075
1	1	1.102

Predictions:



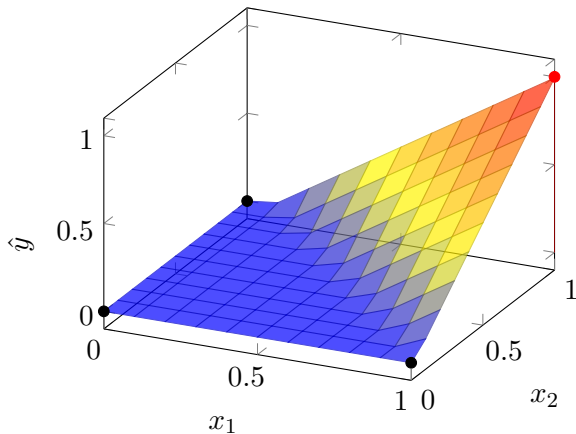
# Beispiel: AND lernen mit ReLU

Epoch: 6



$x_1$	$x_2$	$\hat{y}$
0	0	0
0	1	0
1	0	0
1	1	0.994

Predictions:



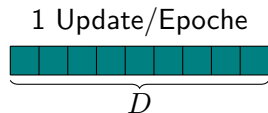


# Von Full-Batch-Learning zu stochastischen Gradientenabstieg

## Full-Batch-Learning (normaler Gradientenabstieg):

- Gradient der Loss-Funktion des gesamten Datensatzes

$$\nabla \frac{1}{|D|} \sum_{(\mathbf{x}_d, y_d) \in D} (y_d - \hat{y}(\mathbf{x}_d, \mathbf{W}))^2$$



## Mini-Batch SGD (Stochastic Gradient Descent), oft einfach "SGD":

- Gradient der Loss-Funktion eines Teils des Datensatzes

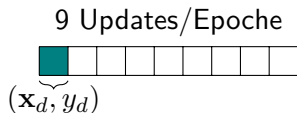
$$\nabla \frac{1}{|D_i|} \sum_{(\mathbf{x}_d, y_d) \in D_i} (y_d - \hat{y}(\mathbf{x}_d, \mathbf{W}))^2$$



## SGD (Stochastic Gradient Descent):

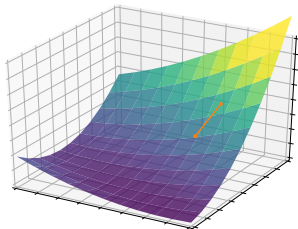
- Gradient der Loss-Funktion eines einzelnen Samples

$$\nabla (y_d - \hat{y}(\mathbf{x}_d, \mathbf{W}))^2$$

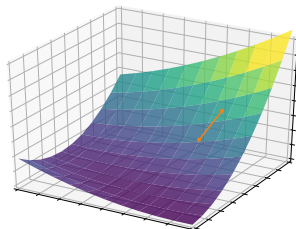


# Stochastischer Gradientenabstieg visualisiert

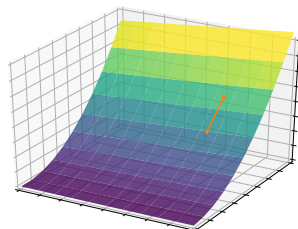
1 Update/Epoche



3 Updates/Epoche



9 Updates/Epoche

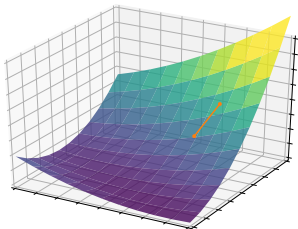
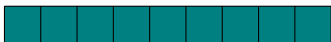


⇒ Mini-Batch SGD vereint die Vorteile:

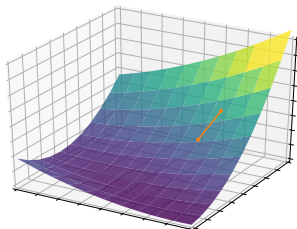
- Stabilität der Mini-Batch-Loss-Funktion besser als die der Sample-Loss-Funktion.
- Parallelisierbarkeit jeweils eines Mini-Batches.
- Häufige Updates sorgen oft für schnellen Fortschritt, können aber auch weniger stabil sein.

# Stochastischer Gradientenabstieg visualisiert

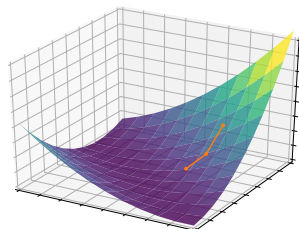
1 Update/Epoche



3 Updates/Epoche



9 Updates/Epoche

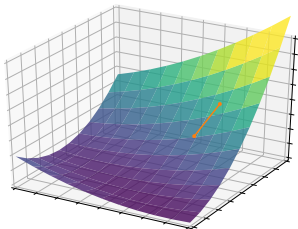
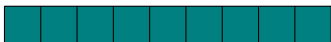


⇒ Mini-Batch SGD vereint die Vorteile:

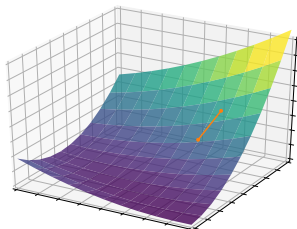
- Stabilität der Mini-Batch-Loss-Funktion besser als die der Sample-Loss-Funktion.
- Parallelisierbarkeit jeweils eines Mini-Batches.
- Häufige Updates sorgen oft für schnellen Fortschritt, können aber auch weniger stabil sein.

# Stochastischer Gradientenabstieg visualisiert

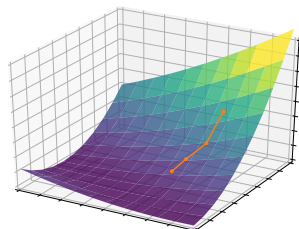
1 Update/Epoche



3 Updates/Epoche



9 Updates/Epoche

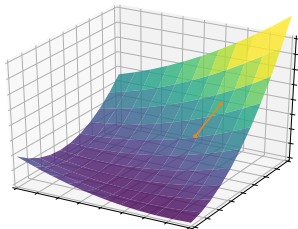


⇒ Mini-Batch SGD vereint die Vorteile:

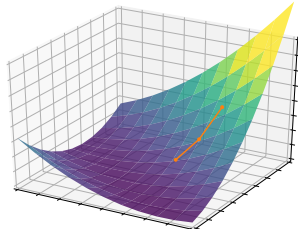
- Stabilität der Mini-Batch-Loss-Funktion besser als die der Sample-Loss-Funktion.
- Parallelisierbarkeit jeweils eines Mini-Batches.
- Häufige Updates sorgen oft für schnellen Fortschritt, können aber auch weniger stabil sein.

# Stochastischer Gradientenabstieg visualisiert

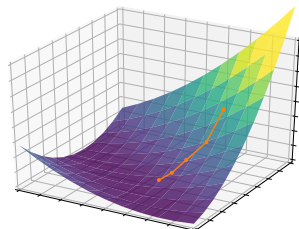
1 Update/Epoche



3 Updates/Epoche



9 Updates/Epoche

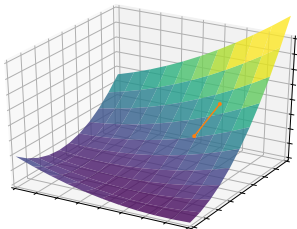
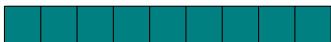


⇒ Mini-Batch SGD vereint die Vorteile:

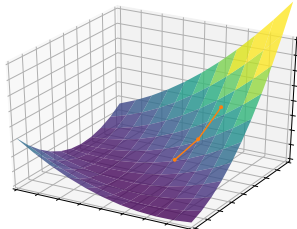
- Stabilität der Mini-Batch-Loss-Funktion besser als die der Sample-Loss-Funktion.
- Parallelisierbarkeit jeweils eines Mini-Batches.
- Häufige Updates sorgen oft für schnellen Fortschritt, können aber auch weniger stabil sein.

# Stochastischer Gradientenabstieg visualisiert

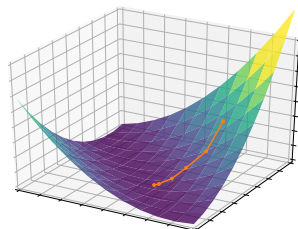
1 Update/Epoche



3 Updates/Epoche



9 Updates/Epoche

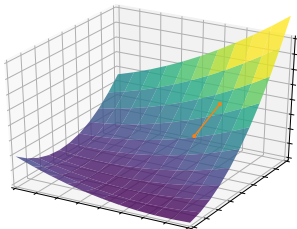
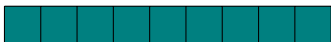


⇒ Mini-Batch SGD vereint die Vorteile:

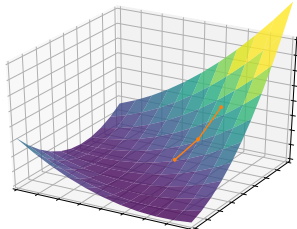
- Stabilität der Mini-Batch-Loss-Funktion besser als die der Sample-Loss-Funktion.
- Parallelisierbarkeit jeweils eines Mini-Batches.
- Häufige Updates sorgen oft für schnellen Fortschritt, können aber auch weniger stabil sein.

# Stochastischer Gradientenabstieg visualisiert

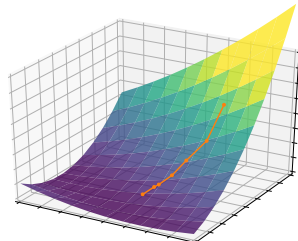
1 Update/Epoche



3 Updates/Epoche



9 Updates/Epoche

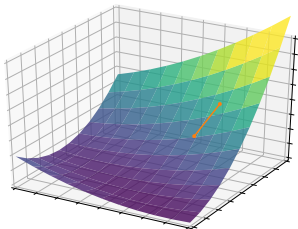
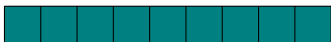


⇒ Mini-Batch SGD vereint die Vorteile:

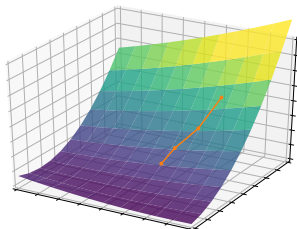
- Stabilität der Mini-Batch-Loss-Funktion besser als die der Sample-Loss-Funktion.
- Parallelisierbarkeit jeweils eines Mini-Batches.
- Häufige Updates sorgen oft für schnellen Fortschritt, können aber auch weniger stabil sein.

# Stochastischer Gradientenabstieg visualisiert

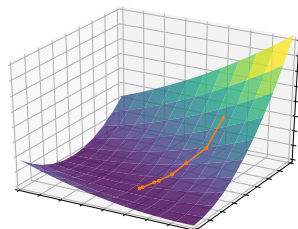
1 Update/Epoche



3 Updates/Epoche



9 Updates/Epoche



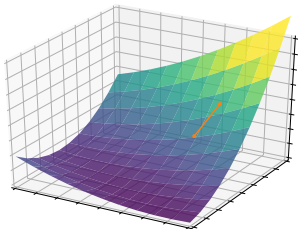
⇒ Mini-Batch SGD vereint die Vorteile:

- Stabilität der Mini-Batch-Loss-Funktion besser als die der Sample-Loss-Funktion.
- Parallelisierbarkeit jeweils eines Mini-Batches.
- Häufige Updates sorgen oft für schnellen Fortschritt, können aber auch weniger stabil sein.

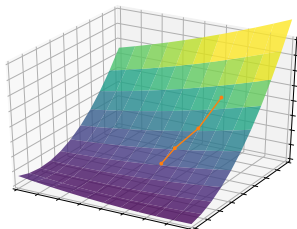
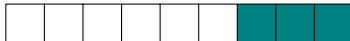


# Stochastischer Gradientenabstieg visualisiert

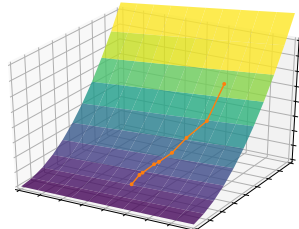
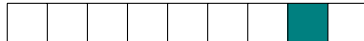
1 Update/Epoche



3 Updates/Epoche



9 Updates/Epoche

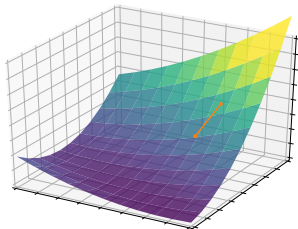


⇒ Mini-Batch SGD vereint die Vorteile:

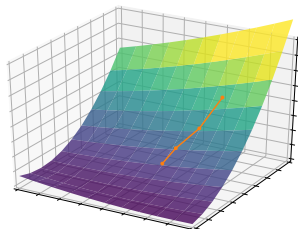
- Stabilität der Mini-Batch-Loss-Funktion besser als die der Sample-Loss-Funktion.
- Parallelisierbarkeit jeweils eines Mini-Batches.
- Häufige Updates sorgen oft für schnellen Fortschritt, können aber auch weniger stabil sein.

# Stochastischer Gradientenabstieg visualisiert

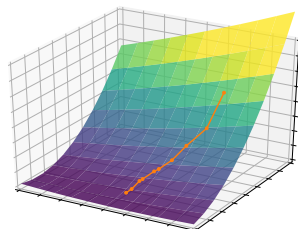
1 Update/Epoche



3 Updates/Epoche



9 Updates/Epoche

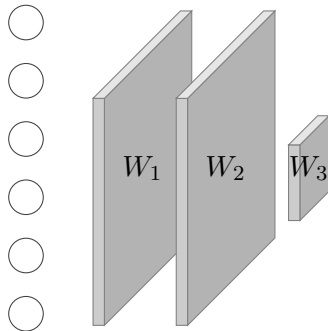


⇒ Mini-Batch SGD vereint die Vorteile:

- Stabilität der Mini-Batch-Loss-Funktion besser als die der Sample-Loss-Funktion.
- Parallelisierbarkeit jeweils eines Mini-Batches.
- Häufige Updates sorgen oft für schnellen Fortschritt, können aber auch weniger stabil sein.

# Keras Sequential model

- Wir beschränken uns auf das **Keras Sequential model**.
- Beim *Sequential model* wird zunächst einmal das Objekt angelegt und dann das neuronale Netz Layer um Layer aufgebaut.
- Hierbei muss man sich vor Augen halten, dass Keras den Input-Layer nicht zählt.
- Er wird implizit erzeugt, indem der erste dem Model hinzugefügte Layer die Information über die Input-Dimension als Parameter enthält.
- Um die Syntax und Grundlagen zu verstehen, gehen wir entlang eines Beispiels vor.
- Wir nutzen das **Boston Housing Dataset**.



*Aufbau des Sequential models*

# Vollverbundene Schicht

- Zunächst betrachten wir nur klassische Netze, bei denen alle Neuronen der Vorgängerschicht mit allen Neuronen der nachfolgenden verbunden sind.

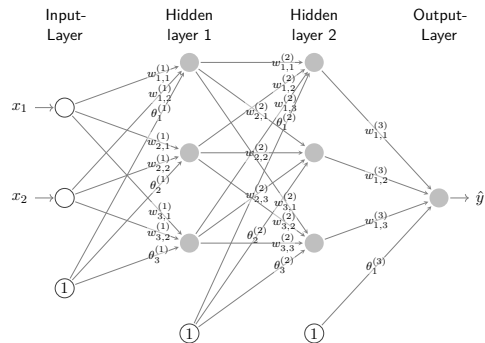
- Diese Art von Schichten nennt man **Fully-connected Layer**.

- Keras verwendet, um diesen Umstand zu benennen, die kürzere Bezeichnung Dense.

- Um diese zu verwenden, binden Sie Folgendes ein:

`from tensorflow.keras.layers import Dense`

- Besteht ein Netz ausschließlich aus solchen Layern, verwendet man oft in Abgrenzung zum Convolutional Neural Network, das wir später besprechen werden, den Begriff **Fully-connected Neural Network**.



- Keras nutzt als Default einen **Vektor von Bias-Neuronen**, den wir so in unserer eigenen Implementierung nicht hatten.
- Der Output einer *Dense*-Schicht wird wie folgt berechnet:

$$\text{output} = a(\text{input} \cdot W^{\top} + b)$$


- $a$  ist die Aktivierungsfunktion, z. B. eine Sigmoid-Funktion.
- Die Matrix der Gewichte ist dabei relativ zu unserer vorangegangenen Implementierung transponiert.
- Um den besseren Vergleich mit den vorangegangenen Kapiteln zu haben, schalten wir diese Bias-Neuronen später ab. Generell sind diese jedoch oft hilfreich.

## Mini-Umsetzung in Keras

```
1 import numpy as np
2 from tensorflow.keras.models import Sequential
3 from tensorflow.keras.layers import Dense
4 from tensorflow.keras.datasets import boston_housing
5
6 (XTrain, YTrain), (XTest, YTest) = boston_housing.load_data()
7 myANN = Sequential()
8 myANN.add(Dense(80, activation='tanh', input_dim=XTrain.shape[1]))
9 myANN.add(Dense(50, activation='tanh'))
10 myANN.add(Dense(1, activation='linear')) # output
11 myANN.compile(loss='mean_squared_error', optimizer='adam')
12 myANN.fit(XTrain, YTrain, epochs=1000, verbose=True)
13
14 yp = myANN.predict(XTest)
15 diff = yp.squeeze() - YTest
```

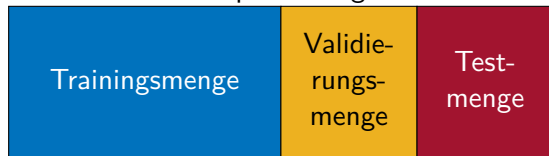
Mean abs error: 3.649, Max abs error: 27.2, Min abs error: 0.01802

## Ein paar praktische Hinweise

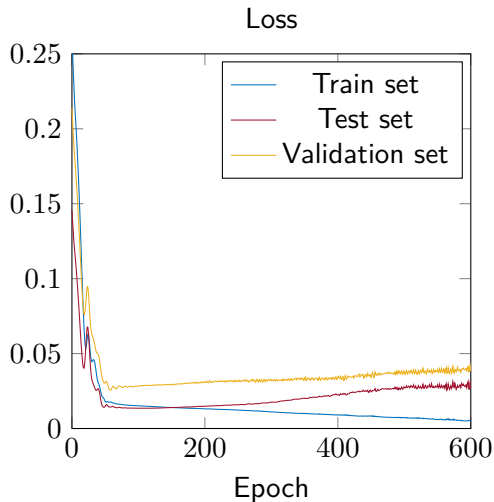
- Ein Trainingsvorgang kann lange dauern. Man kann das Training aber **abbrechen** ■ und das Netz behält seinen aktuellen Trainingszustand. Anschließend kann also ganz normal `ann.predict(...)` ausgeführt werden.
- Man kann ein Netz **weiter trainieren** und muss nicht wieder von neu beginnen! Man kann nach einem Training (oder einem Abbruch) einfach erneut `ann.fit(...)` aufrufen – in Spyder lässt sich eine Zeile mit  ausführen.
- Netze lassen sich **speichern** und **laden**. Es gibt zwei Möglichkeiten:
  - Zum Speichern des Modells kann man `ann.save('filepath.h5')` verwenden. Die Datei enthält dann die Gewichte, die Architektur, den Zustand des Optimierers, etc. Zum Laden reicht dann `ann = tensorflow.keras.models.load_model('filepath.h5')` aus.
  - Wenn man nur die Gewichte (ohne Architektur etc.) eines Modells speichern will, kann man `ann.save_weights('filepath.h5')` verwenden. Zum Laden muss man das Netz neu erstellen, aber anstatt einen Aufruf von `ann.fit(...)` verwendet man `ann.load_weights('filepath.h5')`.

# Validierungsmenge und Early Stopping

- Im Laufe des Trainings kann das Netz übertrainieren bzw. overfitten.
- Da wir die Testmenge erst am Ende benutzen wollen brauchen wir eine weitere Menge, welche nicht zur Optimierung benutzt wird.



- Durch die **Validierungsmenge** kann man das Training rechtzeitig abbrechen **early stopping** oder nachträglich die Gewichte mit dem kleinsten Loss übernehmen.





# Early Stopping in Keras

- Keras stellt Möglichkeiten bereit ein Early Stopping zu realisieren, jedoch ist es nicht das Default-Verhalten.
- Der Schlüssel zur Umsetzung sind in Keras dabei die **Callbacks**. Diese legen wir vor dem Aufruf der `fit`-Methode an und übergeben diese dann.
- Fangen wir mit dem *early stopping* an.

```
from tensorflow.keras.callbacks import EarlyStopping  
earlystop = EarlyStopping(monitor='val_loss', patience=150,  
                           restore_best_weights=True)
```

- `patience=150` bedeutet, dass das Training abbricht, wenn die Fehlerfunktion für mehr als 150 Schritte das bisherige Minimum nicht unterschreitet.
- `restore_best_weights=True` stellt nach dem Stopping die besten Gewichte wieder her anstatt die letzten zu behalten.
- Der `EarlyStopping`-Callback betrachtet standardmäßig den Validation-Loss und erwartet entsprechend, dass auch eine Validierungsmenge in `fit` übergeben wird.

# Early Stopping in Keras

- Alle Callbacks, die wir für das Training übergeben wollen, werden in einer Liste gespeichert.

```
from tensorflow.keras.callbacks import EarlyStopping
earlystop = EarlyStopping(monitor='val_loss', patience=150,
                          restore_best_weights=True)
callbacksList = [earlystop]
```

- Zusammen mit den Validierungsdaten wird diese nun für das Training übergeben.

```
hist = myANN.fit(XTr, YTr, epochs=1000, validation_data=(XVal, YVal),
                 callbacks=callbacksList, verbose=False)
```

- Der Rückgabewert `hist` erlaubt es uns nun, wie gewohnt Einblick in die Fehlerentwicklung über die einzelnen Lernzyklen zu nehmen.

- Der Zugriff erfolgt über eine Dictionary-Datenstruktur von Python, z. B. wie folgt:

```
lossMonitor = np.array(hist.history['loss'])
valLossMonitor = np.array(hist.history['val_loss'])
```

# Boston Housing mit Early Stopping

```
1 import numpy as np
2 from tensorflow import keras
3 from tensorflow.keras.models import Sequential
4 from tensorflow.keras.layers import Dense
5 from tensorflow.keras.datasets import boston_housing
6
7 (XTrain, YTrain), (XTest, YTest) = boston_housing.load_data()
8 trainIdx = np.random.choice(XTrain.shape[0], int(XTrain.shape[0]*0.80), replace=False)
9 XTr      = XTrain[trainIdx,:]
10 YTr      = YTrain[trainIdx]
11 valIdx   = np.delete(np.arange(0, len(YTrain) ), trainIdx)
12 XVal     = XTrain[valIdx,:]
13 YVal     = YTrain[valIdx]
```

# Boston Housing mit Early Stopping

```
14
15 myANN = Sequential()
16 myANN.add(Dense(10, activation='relu', input_dim=XTr.shape[1]))
17 myANN.add(Dense(10, activation='relu'))
18 myANN.add(Dense(1, activation='linear'))
19 myANN.compile(loss='mean_squared_error', optimizer='adam')
20
21 earlystop = keras.callbacks.EarlyStopping(monitor='val_loss', patience=150,
22                                           verbose=False, restore_best_weights=True)
23 cbList = [earlystop]
24 hist = myANN.fit(XTr, YTr, epochs=1000, validation_data=(XVal, YVal), callbacks=cbList)
25
26 yP = myANN.predict(XTest)      # yP.shape == (102, 1)
```

Mean abs error: 3.844, Max abs error: 16.78, Min abs error: 0.02706