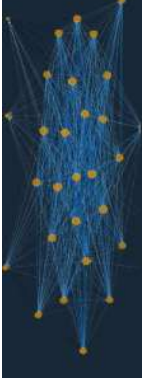
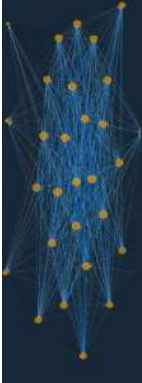


Deep Learning: Hyperparameter Tuning



Rückblick

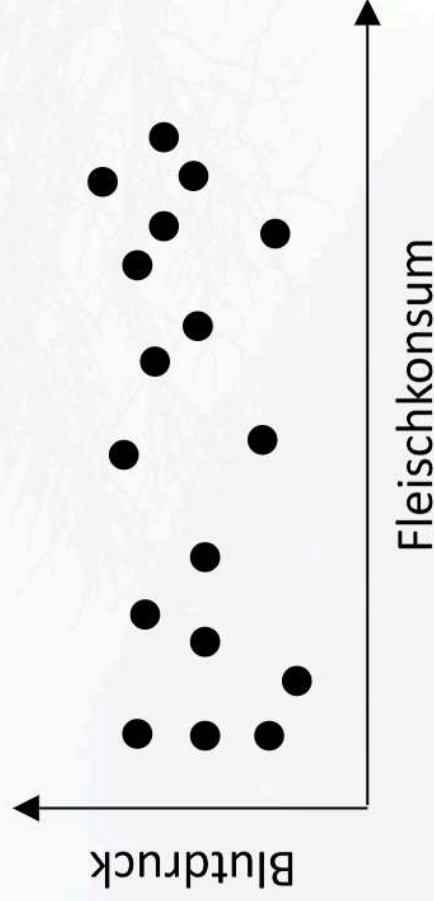
- Welche Möglichkeiten gibt es neuronale Netze in Keras zu erzeugen?
- Was sind Dense-Layer?
- Wir haben 10 Merkmale am Eingang und 100 Hidden-Neuronen in der 1. Schicht. Wie groß ist die Gewichtsmatrix? Wie groß ist der Bias-Vektor?
- Wie viele Output-Knoten gibt es typischerweise bei Regression und bei Klassifikation?
- Was ist eine typische Loss-Funktion für Regression?
- Was ist eine typische Aktivierungsfunktion in der Output-Schicht für Regression?
- Was ist eine geeignete Metrik für Regression?
- Wann werden Labels onehotencoded?
- Was ist eine typische Aktivierungsfunktion in der Output-Schicht für Klassifikation?
- Was ist die typische Loss-Funktion für Klassifikation?
- Was ist eine geeignete Metrik für Klassifikation?
- Wie erkennt man Overfitting?
- Wie erkennt man Underfitting?



Underfitting und Overfitting

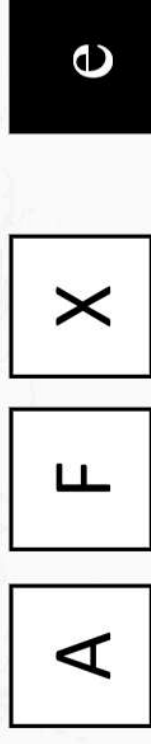
Underfitting:

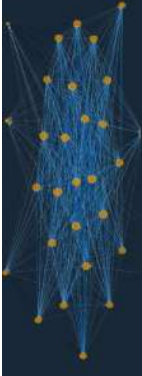
- Großer Fehler bei Trainingsdaten
- Großer Fehler bei Testdaten
- Modell ist zu einfach
- Merkmale sind nicht aussagekräftig
- Starke Messunsicherheit



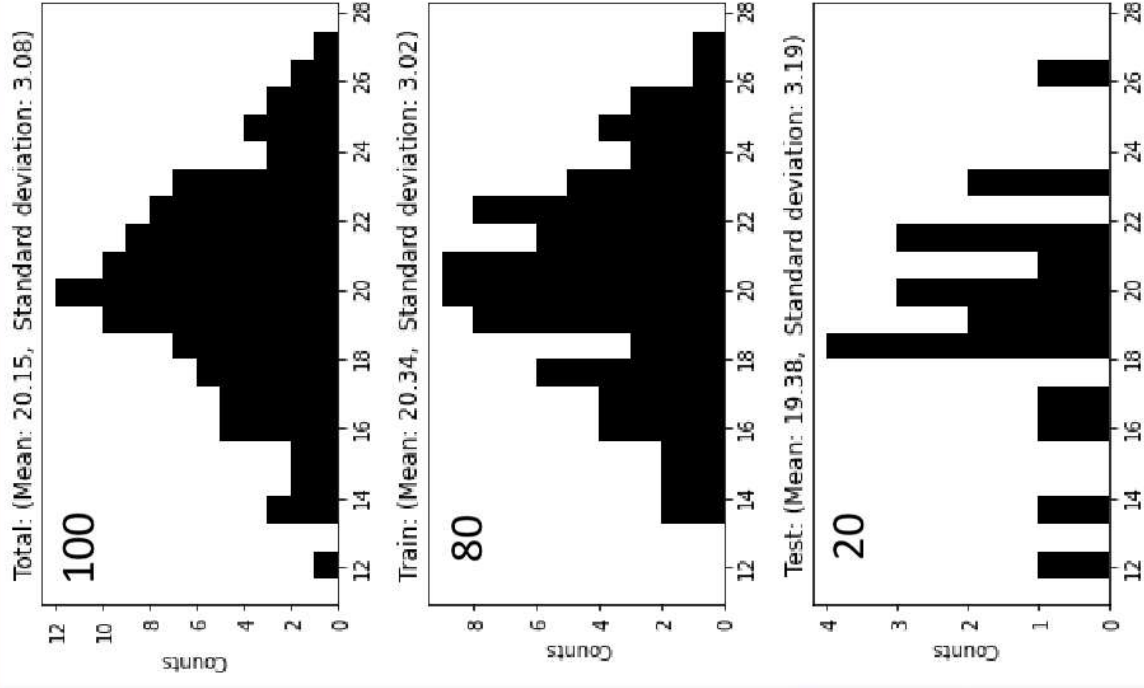
Overfitting:

- Geringer Fehler bei Trainingsdaten
- Großer Fehler bei Testdaten
- Modell ist zu komplex
- Zu wenige Trainingsdaten
- Zu hohe Variabilität
- Trainings- und Testdaten sind zu unterschiedlich





Gesetz der großen Zahlen



↓ Wenige Daten:

Trainings- und
Testdaten

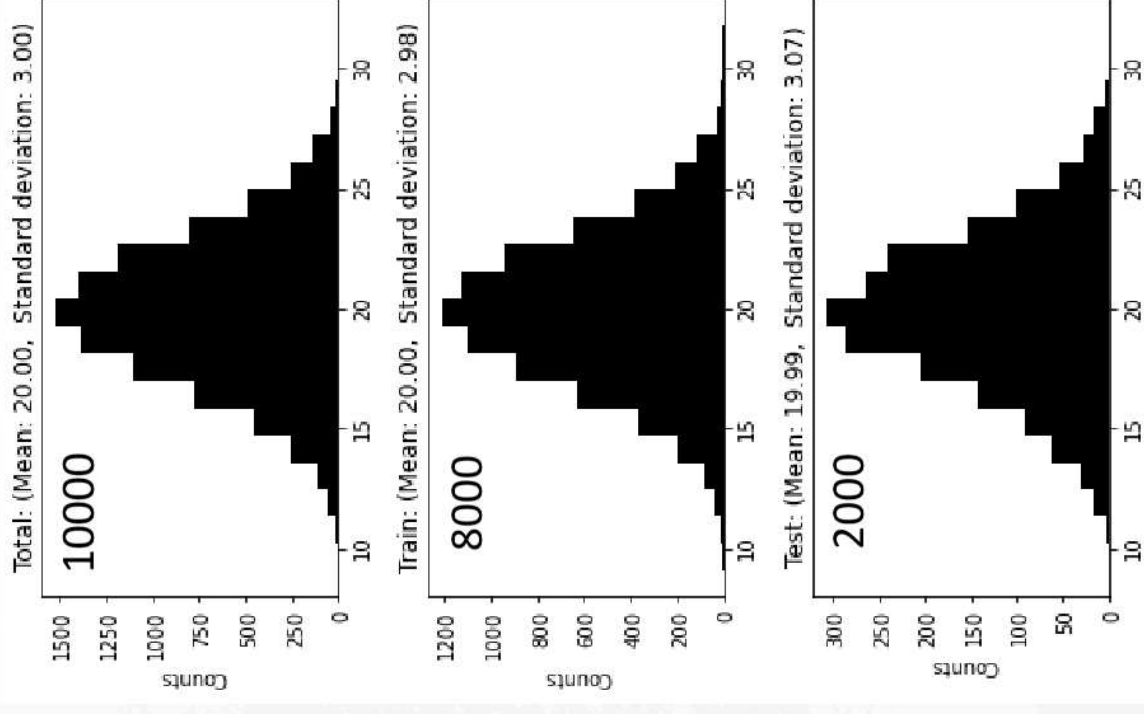
sind statistisch sehr
unterschiedlich

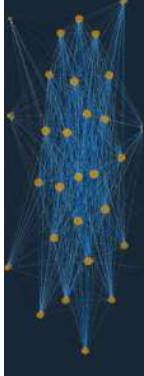


Viele Daten:

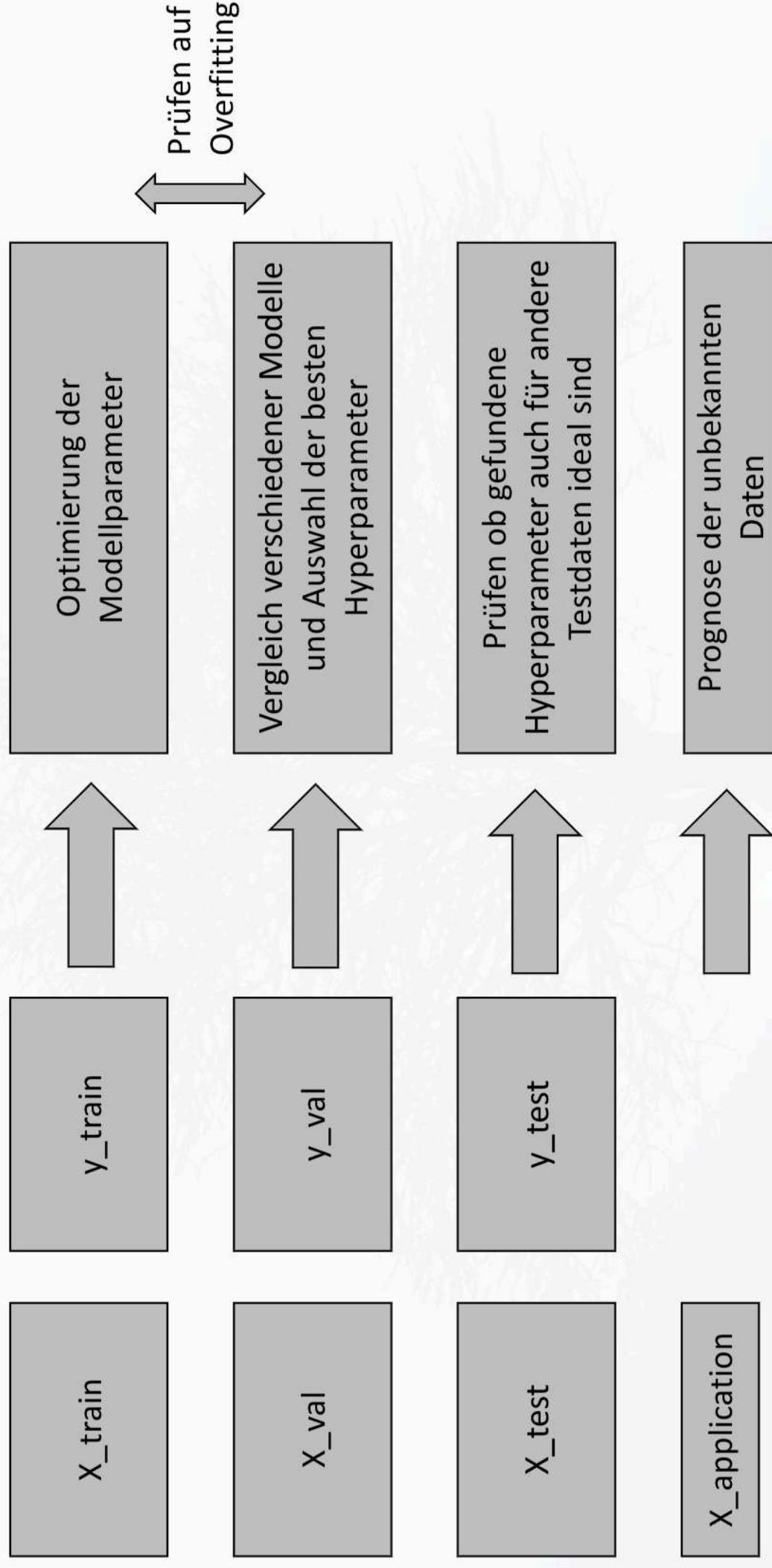
Trainings- und
Testdaten

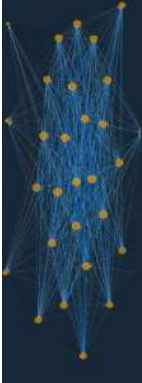
sind statistisch sehr
ähnlich





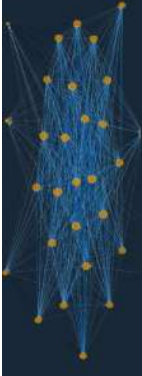
Trainings-, Test- und Validierungsdaten





Aufteilung in Trainings- und Testdaten

- Die Testdaten (und Validierungsdaten) sollen nicht in die Modellparameter einfließen und äquivalent zu den Anwendungsdaten sein
- Wenn Daten nochmal in einzelnen Unterklassen gegliedert sind, sollten die Unterklassen in den Trainings- und Testdaten jeweils komplett verwendet werden
 - Beispiel: Wir wollen für neue Personen das Geschlecht aus einem Bild erkennen
 - Es gibt im Datensatz nur wenige Personen die aber jeweils mit mehreren Bildern vertreten sind
 - Alle Bilder von einer Person sollten dann entweder zum Training oder Testen verwendet werden
 - Dadurch ist sichergestellt, dass die Testdaten den Anwendungsdaten entsprechen und wirklich geprüft wird, ob neue Personen identifiziert werden können
- Wenn es sehr viele Duplikate gibt (z. B. durch Oversampling), sollten die Duplikate nicht in den Trainings- und Testdaten gleichzeitig vorkommen
- Stratifizierung: Um Klassen gleichmäßig auf Trainings- und Testdaten zu verteilen insbesondere wenn es nur wenige Datenpunkte pro Klasse gibt
- Cross-Validation: Die Daten werden in unterschiedliche Trainings- und Validierungspakete aufgeteilt um jeweils separate Modelle zu fitten. Die Modelle werden mit der mittleren Validierungsmetrik beurteilt, um statistische Effekte insbesondere bei kleinen Datensätzen auszugleichen



Beispiel zu Underfitting und Overfitting

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split, ShuffleSplit, cross_val_score

np.random.seed(0)
X = (np.arange(100)-35)/15
y = 0.9*X - 2 * (X ** 2) + 0.5 * (X ** 3) + 0.9*np.random.normal(-3, 3, 100)
X = X[:, np.newaxis]

polynomial_features = PolynomialFeatures(degree=15, include_bias=False)
X_poly = polynomial_features.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(
    X_poly, y, test_size=0.3, random_state=0)

model = LinearRegression()
model.fit(X_train, y_train)
print(model.coef_)

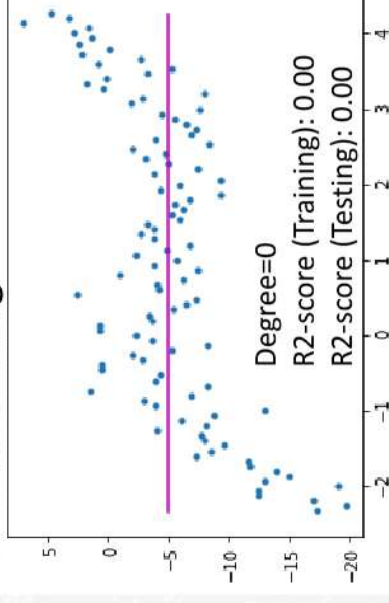
y_pred_train = model.predict(X_train)
y_pred_test = model.predict(X_test)
y_pred = model.predict(X_poly)

r2_train = r2_score(y_train, y_pred_train)
print('R2-score (Training):', r2_train)

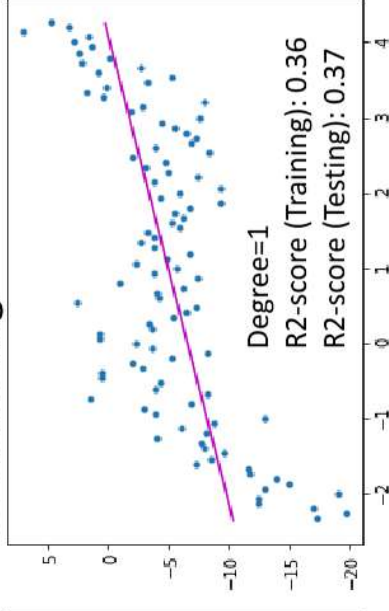
r2_test = r2_score(y_test, y_pred_test)
print('R2-score (Testing):', r2_test)

plt.scatter(X, y, s=10)
plt.plot(X, y_pred, color='m')
plt.show()
```

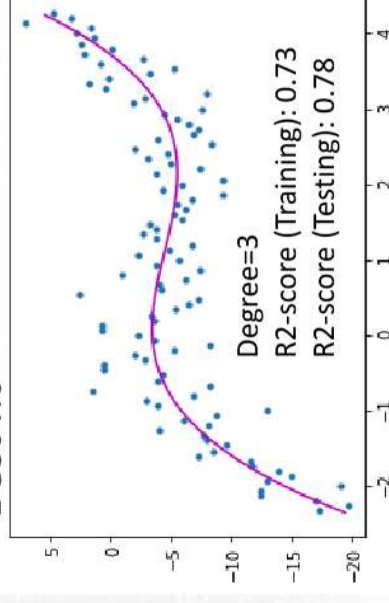
Underfitting



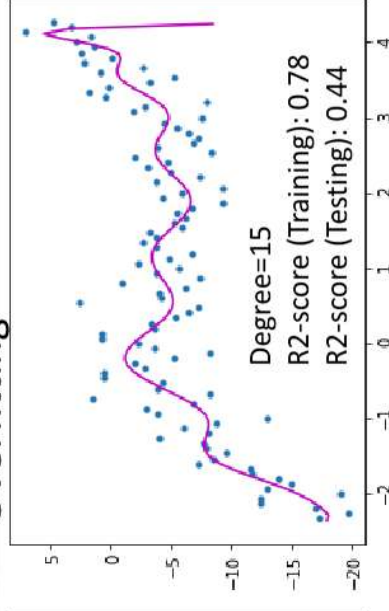
Underfitting

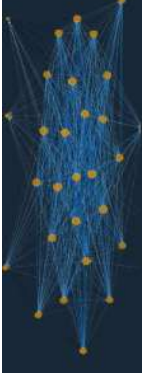


Best fit



Overfitting





Hyperparameter optimieren

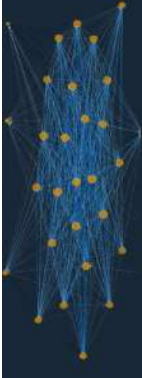
- Gruppe A: l1-, l2-Regularisierung, Buch S. 428-429, S. 187 – 193 (allgemein)
- Gruppe B: Drop-Out, Buch S. 429-432
- Gruppe C: Early-Stopping, Buch S. 373, 193-194 (allgemein)
- Gruppe D: Optimierung der Netz-Größe, Buch S. 378-385

Inhalte: Theorie des Verfahrens

Einfaches Beispiel zur Anwendung der Funktion

Implementierung in „Overfitting moons.py“ und Optimierung der Hyperparameter

Dauer: 13:45 Uhr, anschließend Präsentation



Aus Machine Learning: L2-Regularisierung

https://scikit-learn.org/stable/modules/linear_model.html#ridge-regression

Ridge-Regression implementiert eine L2-Regularisierung mit dem Strafterm $l2\|w\|^2$

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import Ridge
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split
```

```
np.random.seed(0)
X = (np.arange(100)-35)/15
y = 0.9*X - 2 * (X ** 2) + 0.5 * (X ** 3) + 0.9*np.random.normal(-3, 3, 100)
```

```
X = X[:, np.newaxis]
polynomial_features = PolynomialFeatures(degree=18)
X_poly = polynomial_features.fit_transform(X)
```

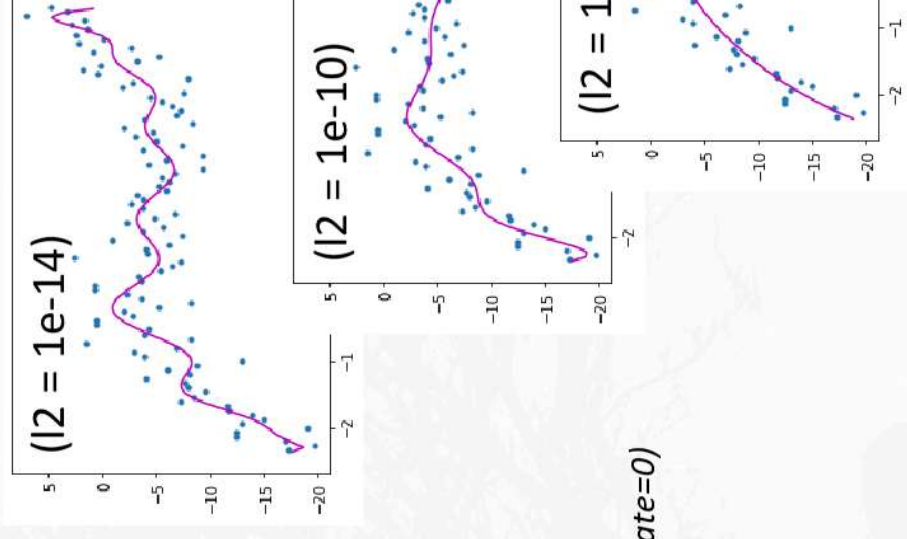
```
X_train, X_test, y_train, y_test = train_test_split(X_poly, y, test_size=0.3, random_state=0)
```

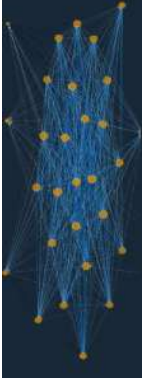
```
model = Ridge(alpha=1e-6, normalize=True)
model.fit(X_train, y_train)
y_pred = model.predict(X_poly)
```

```
plt.scatter(X, y, s=10)
plt.plot(X, y_pred, color='m')
plt.show()
```

$$loss = \sum_{i=1..N} (y_i - (wX_i + b))^2 + l2\|w\|^2$$

Je größer $l2$,
umso stärker die
Regularisierung





L1-, L2-Regularisierung

- In Keras ist Regularisierung auf Gewichtungen, Bias und nach der Aktivierung möglich

- `tf.keras.regularizers.l1(l1=0.01)`

$$loss = loss + l1|w|$$

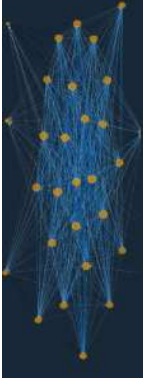
- `tf.keras.regularizers.l2(l2=0.01)`

$$loss = loss + l2\|w\|^2$$

- `tf.keras.regularizers.l1l2(l1=0.01, l2=0.01)`

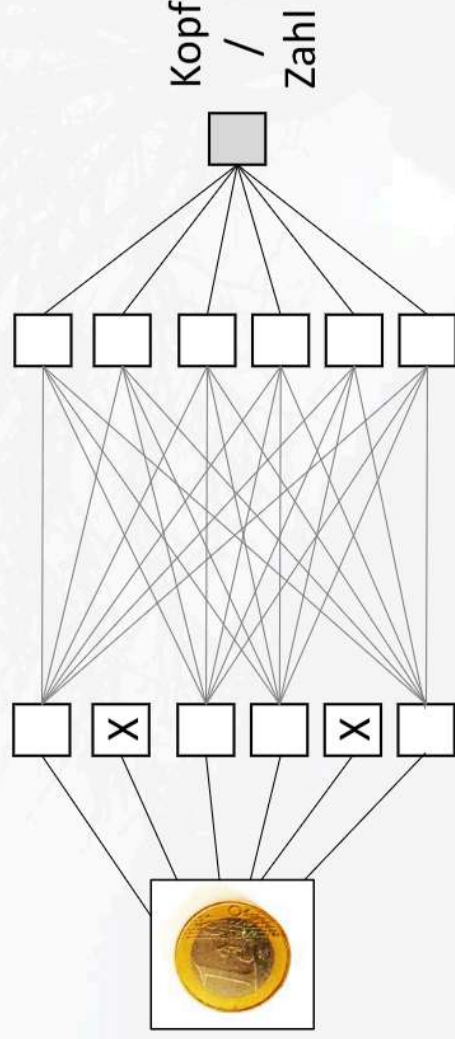
$$loss = loss + l1|w| + l2\|w\|^2$$


- L1 wirkt stärker auf kleine Werte, L2 wirkt stärker auf große Werte
- Z. B. Wert=0.05 $l1=0.01 \rightarrow 0.0005$, Wert=0.05 $l2=0.01 \rightarrow 0.000025$
- Z. B. Wert=0.95 $l1=0.01 \rightarrow 0.0095$, Wert=0.95 $l2=0.01 \rightarrow 0.009025$

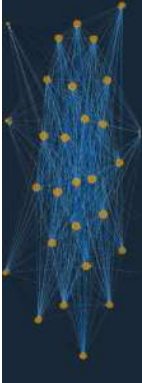


Dropout

- Zufällige Outputs der vorherigen Schicht (z. B. 20%) werden während des Trainings auf 0 gesetzt
- Bei jedem Trainingsschritt werden andere Outputs ausgeschaltet
- Die Werte nach dem Dropout werden um die mittlere Dropout-Rate erhöht, so dass der Mittelwert des Outputs statistisch unverändert bleibt
- Das neuronale Netz lernt wichtige Verbindungen redundant aufzubauen
- Z. B. um Kopf von Zahl zu unterscheiden sind nur einige Pixel relevant
- Bei der Prediction sind alle Outputs aktiv
- Dropout kann das Training verlangsamen



 Da Dropout nur auf die Trainingsdaten wirkt, können diese in den Lernkurven schlechter prognostiziert werden
 → mit evaluate vergleichen

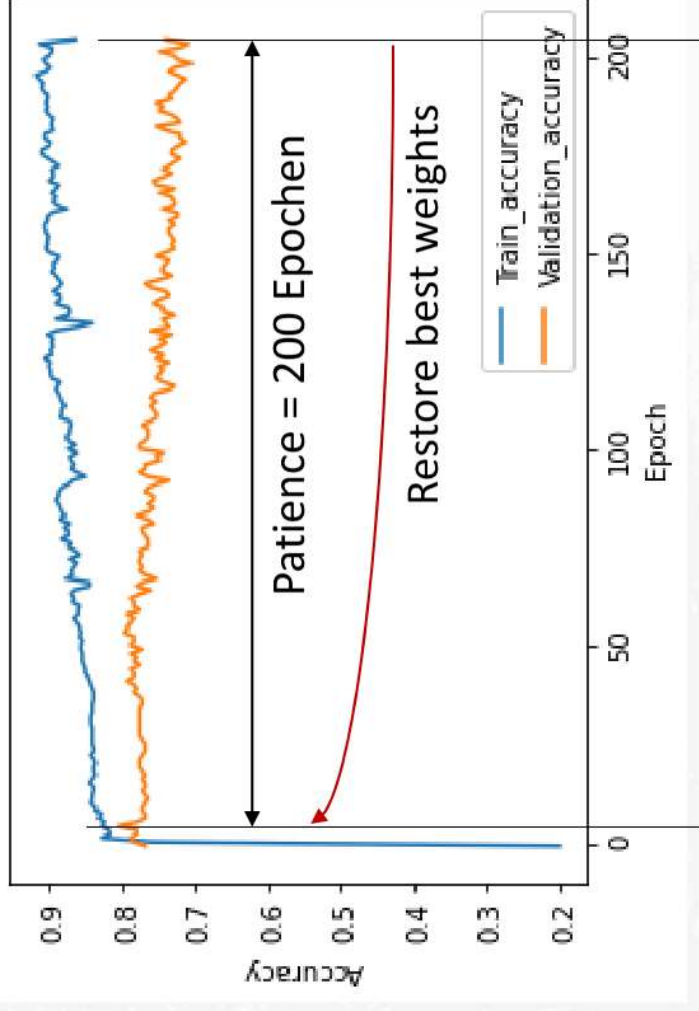


Early-Stopping

- Fitprozess abbrechen, sobald der Validationscore schlechter wird
- Patience: Anzahl Epochen ohne Verbesserung
- Restore_best_weights (nur wenn über early-stopping gestoppt wird)


```
early_stopping=keras.callbacks.EarlyStopping(
    monitor="val_accuracy",
    min_delta=0,
    patience=200,
    verbose=0,
    mode="auto",
    baseline=None,
    restore_best_weights=True,
    start_from_epoch=0)
```

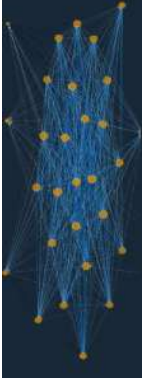
```
history=model.fit(X_train, y_train,,
    epochs=1000, verbose=True,
    validation_data=(X_test, y_test),
    callbacks=[early_stopping])
```



Beste Anzahl an Trainingsepochen

Stopp


 Es erfolgt eine genaue Anpassung
 an die Validierungsdaten
 → mit Testdaten beurteilen



Netzparameter

- Um Overfitting zu vermeiden, sollte die Anzahl der Schichten und Anzahl der Neuronen pro Schicht so gering wie möglich sein
- Eine Optimierung ist z. B. durch den `keras_tuner` (separate Bibliothek) möglich

```
def build_model(hp):
    n_layers = hp.Int('n_layers', min_value=2, max_value=6, step=1)
    n_neurons = hp.Int('n_neurons', min_value=10, max_value=100, step=10)
    model = keras.models.Sequential()
    model.add(keras.layers.InputLayer(input_shape=[2]))

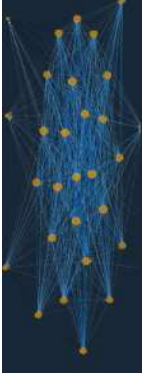
    for i in range(n_layers):
        model.add(keras.layers.Dense(n_neurons, activation="relu"))

    model.add(keras.layers.Dense(1, activation='sigmoid'))
    optimizer = keras.optimizers.Adam(learning_rate=0.01)
    model.compile(loss="binary_crossentropy", metrics=['accuracy'], optimizer='Adam')
    return model

# Die Hyperparameter nach wenigen Epochen vergleichen
tuner = kt.Hyperband(build_model, objective='val_accuracy', max_epochs=1000, factor=3, directory='my_dir',
                    project_name='overfitting_moons_keras_tuner')

tuner.search(x_train, y_train, epochs=1000, validation_data=(x_test, y_test), callbacks=[early_stopping])
```


Übungsvorschläge



- Wie erkennt man Overfitting und Underfitting?
- Welche Methoden gibt es um Overfitting zu vermeiden?
- Beispiel 'Overfitting moons.py' nachvollziehen
 - Neuronales Netz verkleinern (Z. B. eine Schicht mit 200 Neuronen)
 - Drop-out optimieren
 - Regularisierung optimieren
 - Patience für Early-Stopping wählen
 - Automatische Hyperparameter-Optimierung mit keras-tuner