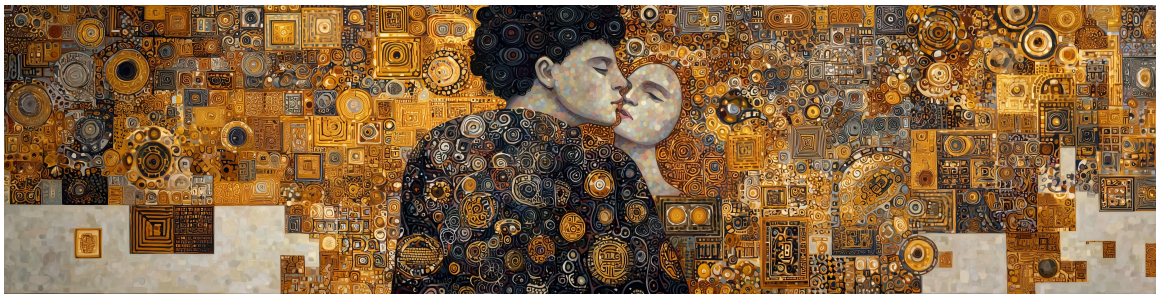
Künstliche Neuronale Netzwerke

Joern Ploennigs · AI4SC



> Emotions are enmeshed in the neural networks of reason.
>
> — António R. Damásio

Künstliche Neuronale Netzwerke haben sich in den letzten Jahren zu den wichtigsten Verfahren im Bereich des maschinellen Lernens und der künstlichen Intelligenz entwickelt. Dabei gehören zu den ältesten und grundlegendsten Algorithmen im Bereich des maschinellen Lernens und der Mustererkennung. Die ersten Grundlagen wurden bereits in den 1940er Jahren entwickelt, als ein Neurophysiologe zusammen mit einem Mathematiker ein Modell für das menschliche Denken vorschlug [1], das später u.a. von Rosenblatt zum Perzeptron weiterentwickelt wurde [2].

Sie sind inspiriert von den biologischen neuronalen Netzwerken des menschlichen Gehirns und versuchen dieses mathematisch nachzuempfinden und somit auch ähnliche Lernerfolge zu erzielen. Sie bestehen aus vielen miteinander verbundenen „Neuronen“, die in Schichten organisiert sind und zur Verarbeitung und Analyse komplexer Datenmuster verwendet werden.

Sie werden heutzutage in vielen Anwendungsfällen genutzt insbesondere bei der Verarbeitung unstrukturierter Daten wie Bilder, Texte oder Sprache. Im Bauwesen werden sie eingesetzt zur automatisierten Rissanalyse in Betonbauwerken mittels Bildverarbeitung, zur Suche nach anomalen Vibrationen oder Dehnungsgeräuschen durch Schallemissionsverfahren oder zur Prognose von Feuchte-/Temperaturverläufen in Bauteilen über neuronale Zeitreihenmodelle.

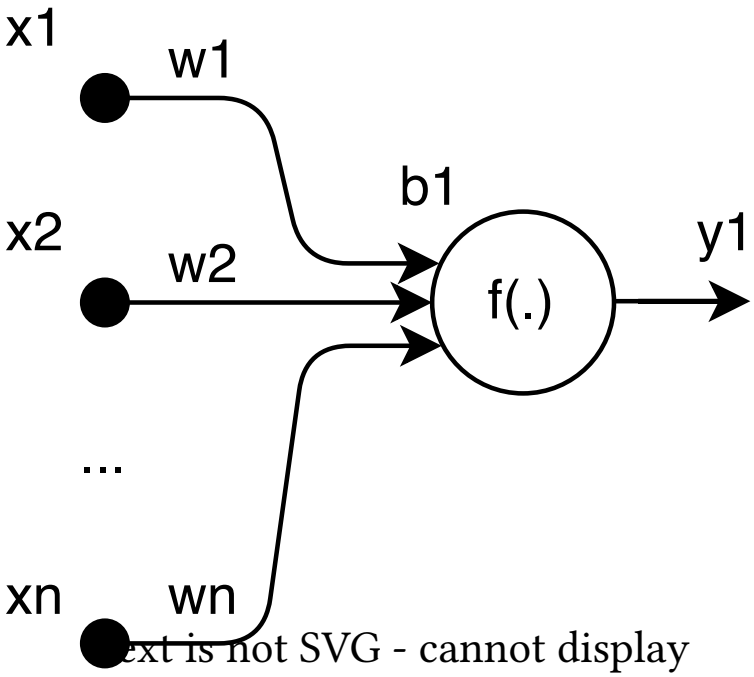
Grundlagen

Neuronen

Neuronen sind die grundlegenden Bausteine eines neuronalen Netzwerks. Ein Neuron empfängt Eingaben, verarbeitet sie und gibt eine Ausgabe weiter. Die Verarbeitung erfolgt durch eine Aktivierungsfunktion $f(\cdot)$, die den linearen Eingang in eine nichtlineare Ausgabe umwandelt.

Das einfachste Neuron ist ein Perceptron [2]. Seine Grundstruktur besteht aus mehreren Komponenten, die in ähnlicher Form auch in modernen Neuronen vorkommen:

- **Eingaben (Inputs, $x_1, x_2, ..., x_n$):** Signale oder Datenpunkte, die in das Neuron eingespeist werden.
- **Gewichte (Weights, $w_1, w_2, ..., w_n$):** Faktoren, die die Bedeutung jedes Eingangs steuern.
- **Bias (Bias, b):** Ein zusätzlicher Parameter, der den Schwellenwert für die Aktivierung anpasst.
- **Aktivierungsfunktion (Activation Function, $f(\cdot)$):** Eine Funktion, die den Summenwert der gewichteten Eingaben und des Bias in eine Ausgabe umwandelt.



Die Eingaben $x_1, x_2, ..., x_n$ sind die Signale oder Datenpunkte, die in das Neuron eingespeist werden. Gewichte $w_1, w_2, ..., w_n$ sind Faktoren, die die Bedeutung jedes Eingangs steuern. Der Bias b ist ein zusätzlicher Parameter, der den Schwellenwert für die Aktivierung anpasst. Schließlich gibt es die Aktivierungsfunktion $f(\cdot)$, die den Summenwert der gewichteten Eingaben und des Bias in eine Ausgabe umwandelt.

Die Ausgabe eines Neurons wird durch die folgende Formel berechnet:

$$y = f\left(b + \sum_{i=1}^{n} w_i x_i\right)$$

Damit gleicht das Neuron in dieser Formel einem *Generalisierten Linearem Modell (GLM)*. Dies trifft aber nur für das einfachste neuronale Netzwerk, das Perceptron, zu. Modernere KNN bestehen aus komplexeren Modellen, die komplexe nichtlineare Zusammenhänge modellieren können. Die Aktivierungsfunktion spielt dabei eine entscheidende Rolle, da sie die Nichtlinearität in das Modell einführt und es dem Netzwerk ermöglicht, komplexe Muster in den Daten zu lernen.

Bei GLM zielt die Transformationsfunktion $f(\cdot)$ darauf ab, die Verteilung der Ausgangsvariable zu verändern. Die meisten Aktivierungsfunktionen bei KNN zielen darauf ab, das Ausgangssignal entweder klarer zu differenzieren (aktiv/inaktiv) oder nichtlinear zu transformieren, um komplexe Zusammenhänge zu lernen. Einige gängige Aktivierungsfunktionen sind:

- **Sigmoid:** Diese Funktion kennen wir von der logistischen Regression und GLM. Sie transformiert den Eingang in einen Wert zwischen 0 und 1, was sie besonders geeignet für die Ausgabe von Wahrscheinlichkeiten macht.

$$f(x) = (1 + e^{-x})^{-1}.$$

- **Tanh (Hyperbolic Tangent):** Diese Funktion transformiert den Eingang in einen Wert zwischen −1 und 1 und wurde historisch in früheren neuronalen Netzen verwendet. Sie kann in Aufgaben nützlich sein, bei denen sowohl positive als auch negative Werte relevant sind.

$$f(x) = \tanh(x).$$

- **ReLU (Rectified Linear Unit):** Sie ist eine einfache und häufig verwendete Funktion, die den Eingang direkt zurückgibt, wenn er positiv ist, und ansonsten Null. ReLU ist bekannt dafür, neuronale Netze schneller und effizienter zu trainieren.

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

- **Softmax:** Die Softmax-Funktion normalisiert die Ausgabe eines Neurons auf eine Wahrscheinlichkeitsverteilung, wobei die Summe aller Ausgänge 1 ergibt. Sie wird häufig in mehrstufigen Klassifizierungsaufgaben eingesetzt, wo sie die Wahrscheinlichkeiten aller Klassen gleichzeitig ausgibt.

$$f(x_i) = e^{x_i} \left(\sum_{j=1}^{K} e^{x_j} \right)^{-1}.$$

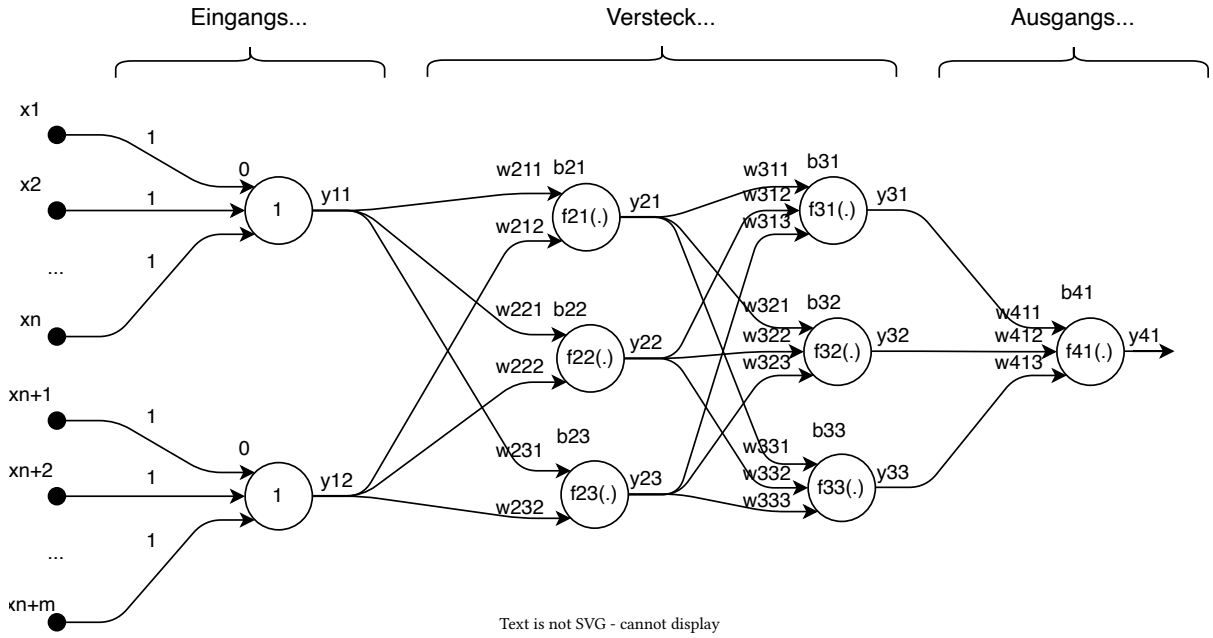
Die Wahl der richtigen Aktivierungsfunktion hängt von der spezifischen Aufgabe und der Architektur des neuronalen Netzes ab. Generell ist ReLU aufgrund seiner Einfachheit und Effizienz eine beliebte Wahl, während Sigmoid und Tanh in Situationen nützlich sein können, wo negative Werte relevant sind oder eine interpretierbare Ausgabe gewünscht ist. Softmax wird typischerweise für mehrstufige Klassifizierungsaufgaben verwendet.

Schichten

Die Fähigkeit komplexe Zusammenhänge zu erlernen, ergibt sich vor allem daraus, dass Neuronale Netzwerke aus mehreren Schichten von Neuronen bestehen, die in einer bestimmten *Architektur* angeordnet sind.

- *Eingabeschicht* (*Input Layer*) nimmt die Rohdaten entgegen und transformiert sie für die nachfolgenden Schichten .
- *Verborgene Schichten* (*Hidden Layers*) enthalten Neuronen, die die eingehenden Daten verarbeiten und transformieren. Bei modernen *Tiefen Neuronalen Netzwerken* (*DNN - Deep Neural Networks*) ist die Anzahl der Schichten hoch (3-100 Schichten).

- *Ausgabeschicht* (*Output Layer*) transformieren die Ausgaben des Netzwerk in die Ergebnisse. Die Anzahl der Neuronen in der Ausgabeschicht hängt von der Art des zu lösenden Problems ab.



Die erste Schicht ist die *Eingabeschicht* (*Input Layer*) und nimmt die Rohdaten des Problems entgegen und leitet sie an die nachfolgenden Schichten weiter. Diese Schicht enthält keine Aktivierungsfunktionen und dient lediglich als Schnittstelle zur Datenaufnahme.

Danach folgen mehrerer *verborgene Schichten* (*Hidden Layers*). Diese Schichten enthalten Neuronen, die die eingehenden Daten verarbeiten und transformieren. Die Anzahl der verborgenen Schichten sowie die Anzahl der Neuronen in jeder Schicht können je nach Komplexität des Netzwerks variieren. Bei modernen *Tiefen Neuronalen Netzwerken* (*DNN - Deep Neural Networks*) ist die Anzahl der Schichten hoch (3-100 Schichten).

Schließlich gibt es die *Ausgabeschicht* (*Output Layer*), die das Ergebnis der Verarbeitung durch das Netzwerk liefert. Die Anzahl der Neuronen in der Ausgabeschicht hängt von der Art des zu lösenden Problems ab. Für binäre Klassifikationsprobleme enthält die Ausgabeschicht oft nur ein Neuron, während für Mehrklassenklassifikationsprobleme mehrere Neuronen erforderlich sind, um die verschiedenen Klassen zu repräsentieren.

Trainingsprozess

Der Grund, weshalb Neuronale Netzwerke fast 60 Jahre lang nicht genutzt wurden, lag in Skalierbarkeit des Trainings. Das oben abgebildete Beispiel zeigt bereits, die hohe Anzahl an 25 Parametern (Gewichte und Bias) für ein kleines Netzwerk mit vier Schichten. Leistungsfähige Netze haben bis zu Milliarden an Parametern.

Aufgrund der Komplexität können die Parameter nicht direkt berechnet werden und werden deshalb normalerweise iterativ angenähert. Dieser Prozess umfasst typischerweise folgende Schritte:

1. **Zufällige Initialisierung** aller Parameter w und b mit Zufallszahlen.
2. **Vorwärtsausbreitung (Forward Propagation):** Die Eingabedaten werden durch das Netzwerk geleitet und die Ausgabe geschätzt. Jedes Neuron berechnet seine Aktivierungsfunktion basierend auf den gewichteten Eingaben und dem Bias mit Hilfe der oben angegeben Formel.
3. **Berechnung des Fehlers (Loss Calculation):** Der Fehler oder Verlust wird durch eine Verlustfunktion berechnet, die die Differenz zwischen der vorhergesagten Ausgabe und der tatsächlichen Ausgabe misst. Die passende Verlustfunktionen richtet sich dabei nach dem Datentyp der Ausgangsvariable. Typische Verlustfunktionen kennen wir bereits:
   - *Mean Square Error* für numerische Daten (siehe Lineare Regression).

$$E^{\mathrm{MSE}} = \frac{1}{n}\sum_{j=1}^{n}\left(\hat{y}_j - y_j\right)^2$$

   - *Kreuzentropie (LogLoss)* für binäre oder kategorische Daten (siehe Logistische Regression).

$$E^{\mathrm{LL}} = -\frac{1}{n}\sum_{j=1}^{n}\left[y_j \log\left(\hat{y}_j\right) + \left(1-y_j\right)\log\left(1-\hat{y}_j\right)\right]$$

4. **Rückwärtsausbreitung (Backpropagation):** Der Fehler wird durch das Netzwerk rückwärts propagiert, um die Gradienten der Verlustfunktion bezüglich der Gewichte und Biases zu berechnen. Dabei wird im Wesentlichen der Fehler rückwärts im Netzwerk verteilt, so dass Neuronen, die einen großen Anteil an der Vorhersage haben (also einen hohen Eingangswert und eine steile Aktivierungsfunktion an dieser Stelle), ein größerer Anteil am Fehler zugeteilt wird. Der Gradient $\nabla E\left(w_{ij}\right)$ des Gewichtes w_{ij} kann mit der Kettenregel berechnet werden:

$$\nabla E\left(w_{ij}\right) = \frac{\partial E}{\partial w_{ij}}$$

$$\nabla E(b_i) = \frac{\partial E}{\partial b_i}$$

   Die entsprechende partielle Ableitung richtet sich prinzipiell nach der Aktivierungsfunktion und wird in modernen Bibliotheken meist analytisch mit der Kettenregel sowie automatischer Differentiation berechnet.
5. **Gewichtsaktualisierung (Weight Update):** Die Gewichte und Biases werden mithilfe eines Optimierungsalgorithmus wie Gradient Descent oder Adam angepasst, um den Fehler zu minimieren. Beim Gradient Descent geschieht dies durch Anwendung der berechneten Gradienten auf die Gewichte und Biases mit Hilfe der Lernrate α:

$$w_{ij} = w_{ij} - \alpha^{*}\nabla E\left(w_{ij}\right)$$

$$b_i = b_i - \alpha^{*}\nabla E(b_i)$$

6. **Abbruchkriterium**: Wiederholung der Schritte 2.-5. bis ein Konvergenzkriteriums erfüllt wurde oder eine maximale Anzahl an Iterationen erreicht wird.

Der Trainingsprozess wird in der Regel über mehrere **Epochen** wiederholt, wobei jede Epoche eine Durchführung des gesamten Trainingsdatensatzes darstellt. Dies ermöglicht es dem Netzwerk, die Gewichte iterativ zu verbessern und seine Leistung auf neuen Daten zu generalisieren. Ein entscheidender Grund für den heutigen Erfolg von Neuronalen Netzwerken ist, dass diese Berechnung der Gewichte hoch parallelisierbar ist und somit sehr gut auf modernen Graphikkarten parallelisiert werden können.

Ein häufiges Problem beim Training neuronaler Netzwerke ist Overfitting, bei dem das Netzwerk die Trainingsdaten zu gut lernt und auf neuen Daten schlecht generalisiert. Um Overfitting zu vermeiden und die Leistung des Netzwerks zu verbessern, werden verschiedene Techniken der Regularisierung eingesetzt:

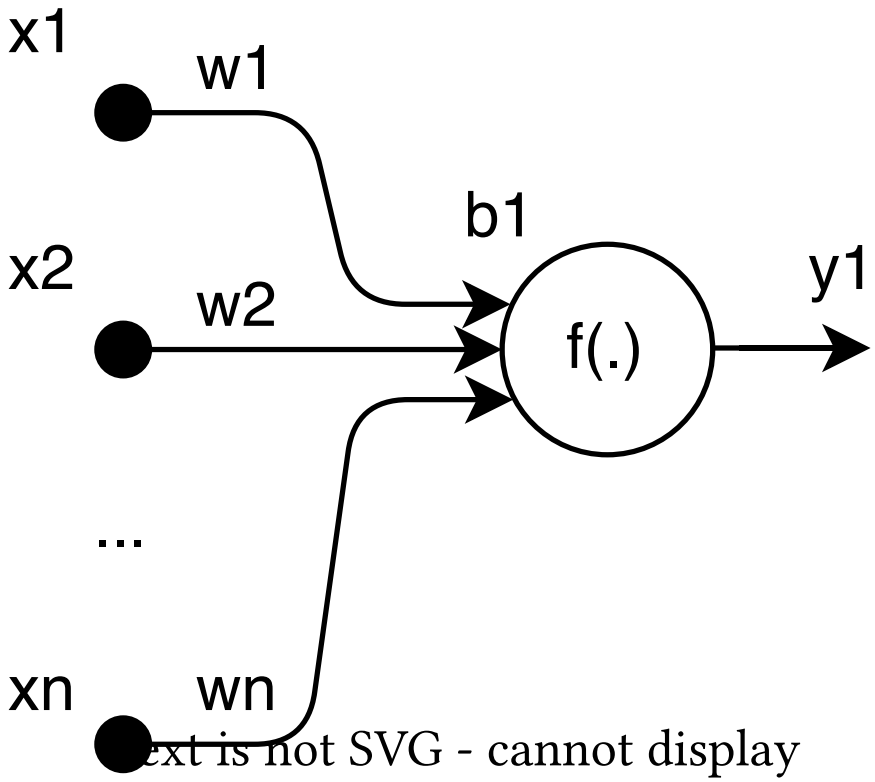
- **Regularisierung (z.B. L1, L2):** Hinzufügen eines Regularisierungsterms zur Verlustfunktion, um die Komplexität des Modells zu kontrollieren.
- **Dropout:** Zufälliges Ausschalten von Neuronen während des Trainings, um die Abhängigkeit von bestimmten Neuronen zu reduzieren.
- **Datenaugmentation:** Erweiterung des Trainingsdatensatzes durch zufällige Transformationen der Daten.

Architekturen

Vorwärtsgerichtete Netze

Ein weiterer wichtiger Faktor für den Erfolg von Neuronalen Netzwerken ist die Fähigkeit, Neuronen in unterschiedlichen Architekturen zu verbinden und sie dadurch zu befähigen, bestimmte Datenstrukturen besser zu lernen.

Die einfachsten Formen neuronaler Netzwerke sind *Vorwärtsgerichtete Neuronale Netze* (*FNN - Feedforward Neural Networks*). Bei ihnen fließen die Daten in eine Richtung von der Eingabeschicht zur Ausgabeschicht, ohne rückwärts gerichtete Datenflüsse. Sie entsprechen den oben dargestellten Netzwerken. Sie sind besonders geeignet bei einfachen Problemen, wo keine Autokorrelation oder komplexe Muster vorherrschen.

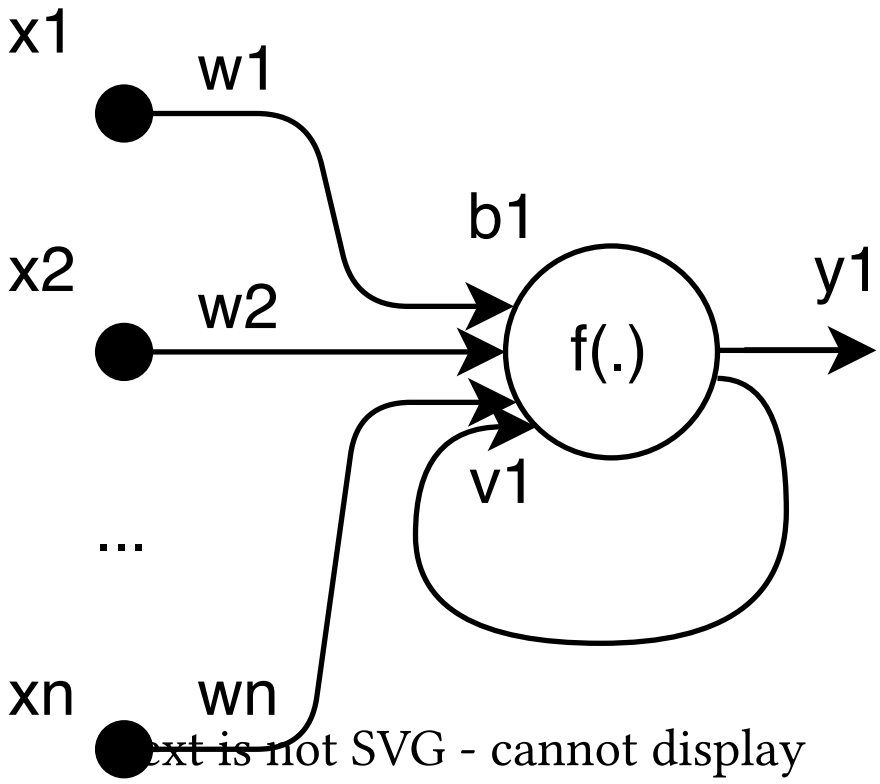


Rekurrente Netze

Rekurrente oder *rückgekoppelte Neuronale Netze* (*RNN - Recurrent Neural Networks*) sind neuronalen Netzen, die sich von Feedforward-Netzen dadurch unterscheiden, dass sie Verbindungen zwischen Neuronen einer Schicht zu Neuronen derselben oder vorheriger Schichten ermöglichen. Diese Art der Verschaltung ähnelt der bevorzugten Struktur neuronaler Netze im Gehirn, insbesondere im Neocortex. RNNs sind ideal für sequenzielle Daten wie Zeitreihen oder Text, da sie über Schleifen verfügen, die Informationen über Zeitpunkte (oder Wortfolgen/Sätze) hinweg speichern. Dadurch lässt sich z.B. gut die Autoregression in Zeitreihen oder komplette Sätze erlernen.

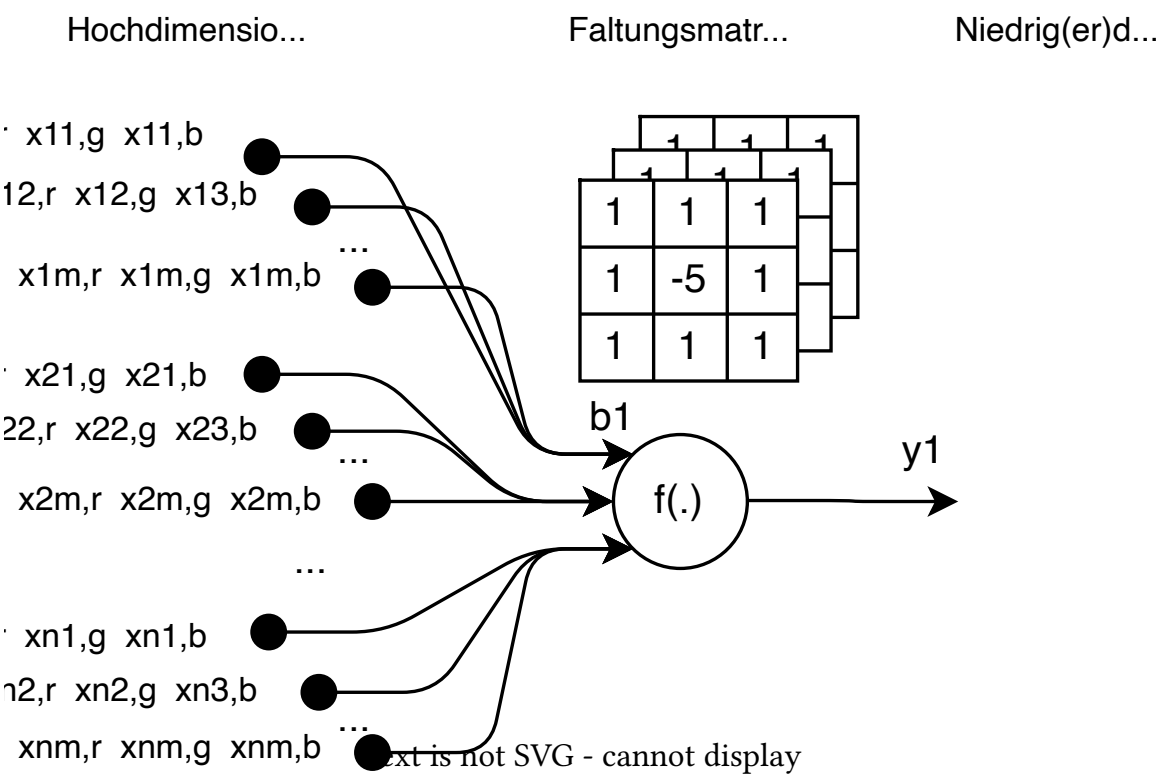
$$h_t = f\left(b + \sum_{i=1}^{n} w_i x_{i,t} + v\, h_{t-1}\right)$$

Hierbei ist h_t der versteckte Zustand (*hidden state*) zum aktuellen Zeitpunkt t und h_{t-1} der versteckte Zustand des vorherigen Zeitpunkts, der über das Gewicht v zurückgeführt wird.



Faltungsnetze

Faltungsnetze (*CNN - Convolutional Neural Networks*) sind besonders effektiv bei der Verarbeitung von höherdimensionalen Daten wie Bilddaten oder geospatialer Daten. Sie verwenden Faltungsoperationen, um Merkmale (Kanten, Reliefs, Muster, etc.) aus Eingabebildern zu extrahieren oder zu erlernen und gleichzeitig einen höherendimensionalen Eingang auf einen niedrigdimensionaleren Ausgang zu reduzieren, der diese Merkmale encodiert. Dadurch repräsentiert der Ausgang nur die Relevanz des extrahierten oder gelernten Merkmals und es muss nicht das Gesamtbild gelernt werden. Durch das Schichten mehrerer versteckter Faltungsschichten hintereinander werden so zuerst Merkmale extrahiert und dann Muster daraus im Bild gelernt (z.B. die Form eines Hundes). Auf die Faltung und die Bildverarbeitung werden wir im Nachfolgekurs „Künstliche Intelligenz und Foundation Modelle“ noch vertieft eingehen.



Neuronale Netzwerke in Python

Neuronale Netzwerke in `sklearn`

Es gibt verschiedene Bibliotheken für Neuronale Netzwerke in Python. Wir betrachten zuerst `sklearn`, da wir dort Neuronale Netzwerke einfach mit der bekannten API nutzen können.

Wir demonstrieren neuronale Netzwerke anhand einer erweiterten Version des Yin-Yang-Beispiels aus der Klassifikation indem wir die Anzahl der Farben auf 4 Kategorien und die Verwirbelung erhöhen. Wieder besteht die Aufgabe darin, die richtige Farbe basierend auf dem Punkt vorherzusagen, also eine Klassifikationsaufgabe. Wir erstellen die Daten wie folgt:

```
import numpy as np # Import von NumPy
import pandas as pd # Import von Pandas
import plotly.express as px # Import von Plotly
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split

np.random.seed(33)
N = 1000  # Anzahl der Punkte
X1 = np.random.normal(size=N)
X2 = np.random.normal(size=N)
X = np.column_stack((X1, X2))
```

```
alpha = np.arctan2(X2, X1)
r = np.sqrt(X1**2 + X2**2)

## Teile the sum of a sin and cosine into 5 intervals
category = pd.cut(np.sin(3*alpha + 2*r) + np.cos(3*alpha + 2*r),
           bins=[-1.5, -1.1, -0.6, 0.6, 1.1, 1.5],
           labels=[0, 1, 2, 3, 4])
y = category.astype(int)

# Teile in Test und Trainings-Datensatz
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

```
# Darstellung des Datensatzes
fig=px.scatter(x=X2, y=X1, color=y, width=600, height=600)
fig.update_traces(marker_line=dict(width=.3, color="black"))
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

Das Bild zeigt ein komplexes Muster von Spiralarmen in fünf verschiedenen Farben. Dreiarmige gelbe und blauen Arme entsprechen den größten und kleinsten Werten, während sechsarmige Arme in verschiedenen Rot- und Violetttönen den dazwischen liegenden Werten entsprechen. Diese Entscheidungsgrenzen sind sehr schwer mit einfachen Modellen wie logistischer Regression, SVM oder sogar Bäumen zu erfassen, da die Spiralarme stark nicht-konvexe und ineinander verschlungene Klassenregionen bilden: Bäume können solche Regionen nur grob durch viele achsenparallele Splits annähern, und ein SVM mit einem einzelnen festen Kernel findet keine Transformation, die alle Spiralarme gleichzeitig sauber trennt. Prüfen wir das noch einmal mit einer kleinen Visualisierungsfunktion, die wir auch in der Klassifikation genutzt haben.

```
mcolors=['blue', 'purple', 'violet', 'orange', 'yellow']

def DBPlot(m, X, y, nGrid = 300):
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max, nGrid),
                           np.linspace(x2_min, x2_max, nGrid))
    XX = np.column_stack((xx1.ravel(), xx2.ravel()))
    hatyy = m.predict(XX).reshape(xx1.shape)
    fig = px.imshow(hatyy, width=600, height=600)
    fig.add_scatter(x=X[:,0]/(x1_max-x1_min)*nGrid+nGrid/2, y=X[:,1]/(x2_max-
x2_min)*nGrid+nGrid/2, mode="markers",
                    marker=dict(color=[mcolors[i] for i in y]),
marker_line=dict(width=.3, color="black"))
    fig.update_coloraxes(showscale=False)
```

```
    fig.update_layout(showlegend=False)
    fig.show()
```

Testen wir einen Random Forest, so erhalten wir zwar ein gutes Modell, dass allerdings overfittet und schlecht generalisiert.

```
from sklearn.ensemble import RandomForestClassifier

m = RandomForestClassifier()
_ = m.fit(X_train, y_train)
m.score(X_test, y_test)
```

```
0.76
```

```
confusion_matrix(y_test, m.predict(X_test))
```

```
array([[51,  1,  2,  0,  0],
       [11, 18,  5,  0,  0],
       [ 2,  8, 50, 10,  0],
       [ 0,  1,  8, 24,  5],
       [ 0,  0,  4,  3, 47]])
```

```
DBPlot(m, X_test, y_test)
```

```
Unable to display output for mime type(s): text/html
```

Ein SVM mit radialem Kernel erkennt die Struktur besser, aber hat einen deutlich höheren Fehler.

```
from sklearn.svm import SVC

m = SVC(kernel="rbf", gamma=1)
_ = m.fit(X_train, y_train)
m.score(X_test, y_test)
```

```
0.544
```

```
confusion_matrix(y_test, m.predict(X_test))
```

```
array([[43,  0, 11,  0,  0],
       [19,  0, 15,  0,  0],
       [11,  0, 52,  0,  7],
```

```
       [ 1,  0, 11,  0, 26],
       [ 3,  0, 10,  0, 41]])
```

```
DBPlot(m, X_test, y_test)
```

```
Unable to display output for mime type(s): text/html
```

Allerdings können neuronale Netzwerke (und KNN) deutlich besser damit umgehen.

sklearn implementiert einfache Feed-Forward-Neuronale Netzwerke, so genannte *Multi-Layer Perceptrons*. Dies sind einfache dichte Feed-Forward-Netzwerke mit einer beliebigen Anzahl von versteckten Schichten. Auch wenn sie strukturell einfache neuronaler Netzwerke sind, sind sie dennoch leistungsfähig genug für viele Aufgaben.

Wie bei anderen fortgeschrittenen Modellen bietet *sklearn* zwei unterschiedlichen Klassen für Klassifikationsaufgaben `MLPClassifier` und für Regressionsaufgaben `MLPRegressor`. Die grundlegende Verwendung dieser Modelle ist identisch zu den anderen *sklearn*-Modellen.

Die wichtigsten Argumente für den `MLPClassifier` sind:

```
MLPClassifier(hidden_layer_sizes, activation, max_iter, alpha)
```

Von diesen ist **hidden_layer_sizes** der zentralste. Dies beschreibt die Anzahl der versteckten Schichten (die Größe der Eingabe- und Ausgabeschicht wird automatisch aus den Daten bestimmt). Es handelt sich um ein Tupel, das die Anzahl der Knoten für jede versteckte Schicht angibt, daher gibt die Länge des Tupels auch die Anzahl der versteckten Schichten an. Zum Beispiel bedeutet `hidden_layer_sizes = (32, 16)` zwei versteckte Schichten, die erste mit 32 und die folgende mit 16 Knoten. **activation** beschreibt die Aktivierungsfunktion. Es empfiehlt sich meist es bei der Standardfunktion `relu` zu lassen, außer es gibt die oben beschriebenen Gründe für andere Aktivierungsfunktionen. **alpha** ist der L2-Regularisierungsparameter und **max_iter** gibt die maximale Anzahl von Iterationen an, bevor die Optimierung stoppt. In moderneren Bibliotheken wie Keras oder PyTorch wird die L2-Regularisierung meist nicht global, sondern explizit pro Schicht angegeben, z.B. mit `kernel_regularizer=regularizers.L2(alpha)` in Keras.

Lassen Sie uns nun die Verwendung von `MLPClassifier` an den Yin-Yang Beispiel demonstrieren. Beginnen wir mit einem einfachen Perzeptron mit einer einzigen versteckten Schicht von 20 Knoten. Daher verwenden wir `hidden_layer_sizes=(20,)`, ein Tupel mit nur einer Zahl. Das Anpassen des Modells und die Vorhersage sind größtenteils ähnlich wie bei den anderen *sklearn*-Funktionen, daher diskutieren wir dies hier nicht. Wir erhöhen auch die Anzahl der Iterationen, da die Standardanzahl von 200 in diesem Fall zu gering ist:

```
from sklearn.neural_network import MLPClassifier

m = MLPClassifier(hidden_layer_sizes = (20,), max_iter=10000)
```

```
_ = m.fit(X_train, y_train)
m.score(X_test, y_test)
```

```
0.588
```

```
confusion_matrix(y_test, m.predict(X_test))
```

```
array([[46,  1,  6,  0,  1],
       [18,  2, 13,  0,  1],
       [11,  1, 53,  0,  5],
       [ 1,  0, 16,  0, 21],
       [ 0,  0,  6,  2, 46]])
```

```
DBPlot(m, X_test, y_test)
```

```
Unable to display output for mime type(s): text/html
```

Dieses einfache Modell hat zuerst auch nur eine niedrige Genauigkeit. Das liegt daran, dass es zu einfach ist, nur eine einzelne kleine versteckte Schicht reicht nicht aus, um das komplexe Spiralmuster gut zu modellieren. Was sich gut in der Visualisierung erkennen lässt. Wir können dort sehen, dass das Modell die Idee korrekt einfängt: Spiralen in verschiedenen Farben, aber die Form der Spiralen ist nicht genau genug.

Lassen Sie uns das Modell mit einem leistungsstärkeren Netzwerk wiederholen:

```
m = MLPClassifier(hidden_layer_sizes = (256, 128, 64), max_iter=10000)
_ = m.fit(X_train, y_train)
m.score(X_test, y_test)
```

```
0.912
```

```
confusion_matrix(y_test, m.predict(X_test))
```

```
array([[52,  1,  1,  0,  0],
       [ 6, 26,  2,  0,  0],
       [ 1,  2, 66,  1,  0],
       [ 0,  0,  4, 34,  0],
       [ 0,  0,  0,  4, 50]])
```

```
DBPlot(m, X_test, y_test)
```

```
Unable to display output for mime type(s): text/html
```

Jetzt sind die Ergebnisse sehr gut mit einer Genauigkeit von ca. 95 %. Die visuelle Inspektion bestätigt, dass das leistungsstärkere neuronale Netzwerk die Gesamtstruktur des Modells gut erfassen kann.

Die angepassten Netzwerkmodelle haben verschiedene Methoden und Attribute, z.B. gibt `coefs_` die Modellgewichte aus (als Liste von Gewichtsmatrizen, eine für jede Schicht), und `intercepts_` gibt die Modell-Biaswerte aus (als Liste von Bias-Vektoren, einer für jede Schicht). Zum Beispiel enthält das oben angepasste Modell den akkumilierten Bias-Vektor für die erste versteckte Schicht:

```
np.sum([b.size for b in m.intercepts_])
```

```
np.int64(453)
```

Während *sklearn* einen einfachen Zugang zu neuronalen Netzwerkmodellen bietet, sind diese Modelle erheblich eingeschränkt. *sklearn* ist damit gut für kleinere Aufgaben und zum schnellen Prototyping geeignet, aber für tiefere Netzwerke, größere Datenmengen und GPU-Beschleunigung braucht man spezialisierte Frameworks wie *tensorflow* oder *pytorch*.

Neuronale Netzwerke in Keras

Tensorflow and Keras

Tensorflow ist eine Bibliothek, die Berechnungsgraphen implementiert, die automatisch Gradienten berechnen können. Sie ermöglicht auch Berechnungen auf der GPU durchzuführen, was potenziell zu erheblichen Geschwindigkeitsverbesserungen gegenüber CPU-Berechnungen führen kann. In vielerlei Hinsicht ähnelt sie *numpy*, wo man Matrizen und Tensoren erstellen und manipulieren kann. Allerdings ist sie aufgrund des Berechnungsgraphen-Ansatzes und der GPU-bezogenen Überlegungen viel schwieriger zu verwenden.

Keras ist ein TensorFlow-Front-End, ein Submodul, das einen viel benutzerfreundlicheren Zugang zum Aufbau neuronaler Netzwerke bietet. Die Funktionalität umfasst eine Vielzahl von Netzwerkschichten, bei denen man die entsprechenden Parameter anpassen kann. Beim Aufbau des Netzwerks kann man einfach die Schichten hintereinander hinzufügen, die Verbindung zwischen den Schichten wird von Keras selbst übernommen. Eine gute Quelle für die Keras-Dokumentation ist die API-Referenzdokumentation.

Tensorflow kann schwierig und frustrierend zu installieren sein. Normalerweise funktioniert es gut mit `conda install tensorflow` oder `pip install tensorflow`. Manchmal können jedoch Probleme auftreten und es kann schwierig sein, die Probleme zu finden und zu beheben. Insbesondere installiert pip normalerweise die neueste Version, auch wenn sie mit dem Rest Ihrer Pakete nicht kompatibel ist. Es installiert auch Abhängigkeiten und kann bestimmte Pakete aktualisieren, was die Python-Installation beeinträchtigen kann. Um Probleme mit dem Rest Ihres

Systems zu vermeiden, empfehlen wir dringend, es in einer virtuellen Umgebung wie Anaconda zu installieren.

Beispielnetzwerk in *keras*

Lassen Sie uns das Farbspiralbeispiel aus dem letzten Abschnitt von *sklearn* neu implementieren, diesmal jedoch mit *keras*. Es wird einige bemerkenswerte Unterschiede geben:

Der Aufbau des Netzwerks selbst ist in Keras anders. Keras berechnet nicht die Größe der Eingabe- und Ausgangsschichten aus den Daten. Beide müssen vom Benutzer angegeben werden. Schließlich sagt Keras nur Wahrscheinlichkeiten für alle Kategorien voraus, daher müssen wir Code hinzufügen, der die Spalte (Kategorie) mit der höchsten Wahrscheinlichkeit findet.

Das Modell erzeugen

Zuerst der wichtigste Schritt: **Aufbau** und **Kompilierung** des Modells. Dies unterscheidet sich sehr von der Vorgehensweise in sklearn. Lassen Sie uns ein sequentielles Modell mit dichten Schichten aufbauen, die gleiche Architektur, die wir mit *sklearn* erstellt haben:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input

# sequential (not recursive) model (one input, one output)
model=Sequential()
model.add(Input(shape=(2,)))
model.add(Dense(512, activation="relu"))
model.add(Dense(256, activation="relu"))
model.add(Dense(64, activation="relu"))
nCategories = len(np.unique(category))
model.add(Dense(nCategories, activation="softmax"))
```

Wir beginnen mit dem Importieren der Funktionalität, die wir von `tensorflow.keras` benötigen. Danach erstellen wir ein leeres sequenzielles Modell (d.h. ein Modell, das keine Schleifen und rückwärts fließende Daten enthält) und fangen an, Schichten hinzuzufügen. Wir fügen 3 dichte Schichten (Dense) hinzu. Die erste Schicht enthält 512 Knoten, die zweite 256 und die letzte Schicht enthält 64 Knoten. Alle diese Knoten werden mit der *relu* Funktion aktiviert.

Das erste wichtige neue Feature ist die explizite Definition der Eingabeform, hier mit `Input(shape=(2,))` vor der ersten dichten Schicht. Dies sagt Keras, welche Art von Eingaben zu erwarten sind. Derzeit ist es nur ein Tupel `(2,)`, da unsere *X*-Matrix nur 2 Spalten enthält. Also `(2,)` ist nur die Form einer einzelnen Zeile von *X*, einer einzelnen Instanz der Eingabedaten. Sie können die richtige Form mit `X[0].shape` finden. Aber die Eingabe muss hier nicht nur ein Vektor sein. Zum Beispiel kann es bei Bildern ein 3-D-Tensor mit der Form `(Breite, Höhe, #Farbkanäle)` sein. Die folgenden Schichten können diese Form dann automatisch ableiten.

Wir müssen auch eine explizite Ausgabeschicht hinzufügen. Da es sich hier um Klassifizierung handelt, benötigen wir so viele Ausgabeknoten wie Kategorien - wir können diese Anzahl berechnen als

```
nCategories = len(np.unique(category))
```

Jeder Ausgabeknoten wird die Wahrscheinlichkeit vorhersagen, dass die Eingabe in die entsprechende Kategorie fällt. Wir können die *softmax* (multi-nomiale Logit)-Aktivierung verwenden, um sicherzustellen, dass die Ergebnisse gültige Wahrscheinlichkeiten sind. Beachten Sie, dass im Falle von nur zwei Kategorien die Softmax-Aktivierung äquivalent zur gewöhnlichen logistischen Regression ist. Aber im Gegensatz zur gewöhnlichen logistischen Regression haben wir eine Reihe anderer Schichten vor der letzten logistischen Schicht.

Dabei ist es hilfreich, zwischen drei Begriffen zu unterscheiden: *Logits* sind die rohen Ausgaben vor der letzten Aktivierungsfunktion, *Wahrscheinlichkeiten* sind die durch Sigmoid oder Softmax transformierten Werte, und *Klassenlabels* erhält man erst durch eine anschließende Entscheidung wie Schwellenwert oder `argmax`.

Das richtige Festlegen der Eingabeformen und Ausgabeknoten ist eine der Hauptursachen für Frustration, wenn man anfängt, mit *keras* zu arbeiten. Die Fehlermeldungen sind lang und nicht besonders hilfreich, und es ist schwer zu verstehen, was das Problem ist. Hier ist eine Checkliste, die durchgearbeitet werden kann, wenn etwas nicht funktioniert:

- Ist die Eingabeform explizit definiert, z.B. mit `Input(shape=...)` am Anfang des Modells?
- Stellt diese Form korrekt eine einzelne Instanz der Eingabedaten dar?
- Haben Sie die richtige Anzahl von Knoten in der Softmax-aktivierten Ausgangsschicht?

Training des Modells

Die nächste Aufgabe besteht darin, das Modell zu kompilieren und anzupassen. Keras-Modelle müssen kompiliert werden - was wir bisher eingerichtet haben, ist nur eine Beschreibung des Modells, nicht das tatsächliche Modell, das für TensorFlow-Tensoren eingerichtet ist und möglicherweise für die GPU-Ausführung. Wir können das Modell wie folgt kompilieren:

Für ein sauberes Training trennt man die Daten typischerweise in Trainings-, Validierungs- und Testdaten. Die Trainingsdaten werden zum Lernen der Gewichte verwendet, die Validierungsdaten zur Kontrolle während des Trainings und zur Wahl von Hyperparametern, und die Testdaten erst ganz am Ende für eine unabhängige Bewertung. Eine häufige Technik ist dabei *Early Stopping*: Man beendet das Training, wenn sich der Fehler auf den Validierungsdaten nicht weiter verbessert, um Overfitting zu vermeiden.

```
model.compile(loss='sparse_categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
print(model.summary())
```

```
Model: "sequential"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ dense (Dense)                   │ (None, 512)            │         1,536 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_1 (Dense)                 │ (None, 256)            │       131,328 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_2 (Dense)                 │ (None, 64)             │        16,448 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_3 (Dense)                 │ (None, 5)              │           325 │
└─────────────────────────────────┴────────────────────────┴───────────────┘
```

```
Total params: 149,637 (584.52 KB)
```

```
Trainable params: 149,637 (584.52 KB)
```

```
Non-trainable params: 0 (0.00 B)
```

```
None
```

Die drei wichtigsten Argumente sind

- `loss` beschreibt die Modellverlustfunktion, `sparse_categorical_crossentropy`, im Wesentlichen die Log-Likelihood, ist geeignet für Kategorisierungsaufgaben. Der genaue Typ hängt auch davon ab, wie das Ergebnis genau codiert ist.
- `optimizer` ist der Optimierer, der für den stochastischen Gradientenabstieg verwendet werden soll. `adam` und `rmsprop` sind gute Optionen, aber es gibt auch andere Möglichkeiten.
- `metrics` ist eine Metrik, die während der Optimierung ausgewertet und gedruckt wird und einige Rückmeldungen darüber bietet, wie die Auswertung verläuft.

Die letzte Zeile hier druckt die Modellzusammenfassung aus, eine praktische Übersicht darüber, was wir gemacht haben:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 512)               1536
_________________________________________________________________
dense_1 (Dense)              (None, 256)               131328
_________________________________________________________________
dense_2 (Dense)              (None, 64)                16448
_________________________________________________________________
```

```
dense_3 (Dense)              (None, 5)                 325
=================================================================
Total params: 149,637
Trainable params: 149,637
Non-trainable params: 0
```

Dieses Netzwerk enthält fast 150.000 Parameter, von denen alle trainierbar sind.

Nach erfolgreicher Kompilierung können wir das Modell anpassen:

```
history = model.fit(X_train, y_train, epochs=200, validation_split=0.2)
```

```
Epoch 1/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 10s 583ms/step - accuracy: 0.1875 - loss:
1.592719/19 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.2533 - loss: 1.5699
- val_accuracy: 0.2600 - val_loss: 1.5469
Epoch 2/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.3750 - loss: 1.474819/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.3017 - loss: 1.5437 -
val_accuracy: 0.3067 - val_loss: 1.5446
Epoch 3/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.3125 - loss: 1.507619/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.3050 - loss: 1.5315 -
val_accuracy: 0.3533 - val_loss: 1.5437
Epoch 4/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.3438 - loss: 1.533719/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.3483 - loss: 1.5109 -
val_accuracy: 0.3333 - val_loss: 1.5391
Epoch 5/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.3438 - loss: 1.461019/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.3467 - loss: 1.4979 -
val_accuracy: 0.3600 - val_loss: 1.5253
Epoch 6/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.2812 - loss: 1.461019/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.3633 - loss: 1.4799 -
val_accuracy: 0.4000 - val_loss: 1.5272
Epoch 7/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.4375 - loss: 1.513119/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.3817 - loss: 1.4590 -
val_accuracy: 0.4000 - val_loss: 1.5145
Epoch 8/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.3438 - loss: 1.443419/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.3717 - loss: 1.4460 -
val_accuracy: 0.3933 - val_loss: 1.5148
Epoch 9/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.5000 - loss: 1.335419/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4000 - loss: 1.4221 -
```

```
val_accuracy: 0.3733 - val_loss: 1.5434
Epoch 10/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.4375 - loss: 1.363819/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.3900 - loss: 1.4091 -
val_accuracy: 0.4067 - val_loss: 1.4779
Epoch 11/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.5000 - loss: 1.341119/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4317 - loss: 1.3820 -
val_accuracy: 0.4067 - val_loss: 1.4795
Epoch 12/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.5000 - loss: 1.281919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4267 - loss: 1.3592 -
val_accuracy: 0.4267 - val_loss: 1.4773
Epoch 13/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.3438 - loss: 1.330519/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4383 - loss: 1.3296 -
val_accuracy: 0.4600 - val_loss: 1.4253
Epoch 14/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.4375 - loss: 1.254319/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4300 - loss: 1.3043 -
val_accuracy: 0.4400 - val_loss: 1.4127
Epoch 15/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.4375 - loss: 1.209419/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4467 - loss: 1.2964 -
val_accuracy: 0.4667 - val_loss: 1.3783
Epoch 16/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.2812 - loss: 1.414319/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4683 - loss: 1.2531 -
val_accuracy: 0.5000 - val_loss: 1.3650
Epoch 17/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.6562 - loss: 1.156919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4933 - loss: 1.2199 -
val_accuracy: 0.5333 - val_loss: 1.3263
Epoch 18/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.4062 - loss: 1.243119/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4850 - loss: 1.1818 -
val_accuracy: 0.5200 - val_loss: 1.2931
Epoch 19/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.4688 - loss: 1.204219/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5200 - loss: 1.1706 -
val_accuracy: 0.5467 - val_loss: 1.2598
Epoch 20/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.4062 - loss: 1.157819/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5483 - loss: 1.1217 -
val_accuracy: 0.5800 - val_loss: 1.2292
Epoch 21/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.5625 - loss: 1.160319/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5683 - loss: 1.0677 -
```

```
val_accuracy: 0.6267 - val_loss: 1.1530
Epoch 22/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.5938 - loss: 1.028119/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6300 - loss: 1.0230 -
val_accuracy: 0.6667 - val_loss: 1.0997
Epoch 23/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.6875 - loss: 1.027619/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6233 - loss: 0.9963 -
val_accuracy: 0.6600 - val_loss: 1.1150
Epoch 24/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.7188 - loss: 0.904219/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6083 - loss: 0.9781 -
val_accuracy: 0.6533 - val_loss: 1.0736
Epoch 25/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.6250 - loss: 1.181319/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6683 - loss: 0.9354 -
val_accuracy: 0.6400 - val_loss: 0.9936
Epoch 26/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.6875 - loss: 0.914519/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6550 - loss: 0.8891 -
val_accuracy: 0.6733 - val_loss: 0.9962
Epoch 27/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.6875 - loss: 0.781519/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6717 - loss: 0.8503 -
val_accuracy: 0.6867 - val_loss: 0.9378
Epoch 28/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.7188 - loss: 0.768219/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6950 - loss: 0.8294 -
val_accuracy: 0.6867 - val_loss: 0.8915
Epoch 29/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.7188 - loss: 0.793719/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6700 - loss: 0.8202 -
val_accuracy: 0.7667 - val_loss: 0.8525
Epoch 30/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.7188 - loss: 0.835919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7083 - loss: 0.7712 -
val_accuracy: 0.7200 - val_loss: 0.8100
Epoch 31/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8438 - loss: 0.677819/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7300 - loss: 0.7167 -
val_accuracy: 0.7133 - val_loss: 0.7884
Epoch 32/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.554019/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7683 - loss: 0.6834 -
val_accuracy: 0.7267 - val_loss: 0.7383
Epoch 33/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8125 - loss: 0.545819/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7450 - loss: 0.6746 -
```

```
val_accuracy: 0.7133 - val_loss: 0.7544
Epoch 34/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.6875 - loss: 0.645519/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7267 - loss: 0.6718 -
val_accuracy: 0.7067 - val_loss: 0.6971
Epoch 35/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.7500 - loss: 0.699719/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7633 - loss: 0.6242 -
val_accuracy: 0.7533 - val_loss: 0.6930
Epoch 36/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.7812 - loss: 0.574019/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7633 - loss: 0.6093 -
val_accuracy: 0.7200 - val_loss: 0.7188
Epoch 37/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.5938 - loss: 0.770019/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7350 - loss: 0.6378 -
val_accuracy: 0.7867 - val_loss: 0.6145
Epoch 38/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.6875 - loss: 0.649019/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8017 - loss: 0.5897 -
val_accuracy: 0.7733 - val_loss: 0.6230
Epoch 39/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8438 - loss: 0.474519/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8083 - loss: 0.5285 -
val_accuracy: 0.7867 - val_loss: 0.6108
Epoch 40/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.415919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8217 - loss: 0.5123 -
val_accuracy: 0.8067 - val_loss: 0.5728
Epoch 41/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.463419/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8200 - loss: 0.5000 -
val_accuracy: 0.7800 - val_loss: 0.6421
Epoch 42/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.417019/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8400 - loss: 0.4859 -
val_accuracy: 0.7667 - val_loss: 0.6185
Epoch 43/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.409419/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8200 - loss: 0.4801 -
val_accuracy: 0.8333 - val_loss: 0.5697
Epoch 44/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.438119/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8567 - loss: 0.4527 -
val_accuracy: 0.7667 - val_loss: 0.5851
Epoch 45/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.7812 - loss: 0.527619/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8350 - loss: 0.4698 -
```

```
val_accuracy: 0.8133 - val_loss: 0.6555
Epoch 46/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.314319/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8633 - loss: 0.4500 -
val_accuracy: 0.8067 - val_loss: 0.5494
Epoch 47/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.422619/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8450 - loss: 0.4571 -
val_accuracy: 0.8133 - val_loss: 0.5627
Epoch 48/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.8125 - loss: 0.428819/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8333 - loss: 0.4235 -
val_accuracy: 0.8067 - val_loss: 0.5585
Epoch 49/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.7500 - loss: 0.436619/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8433 - loss: 0.4121 -
val_accuracy: 0.8133 - val_loss: 0.5059
Epoch 50/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.392819/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8833 - loss: 0.3988 -
val_accuracy: 0.8200 - val_loss: 0.4948
Epoch 51/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8125 - loss: 0.485919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8667 - loss: 0.3808 -
val_accuracy: 0.8867 - val_loss: 0.4826
Epoch 52/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.350719/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8850 - loss: 0.3640 -
val_accuracy: 0.8333 - val_loss: 0.4777
Epoch 53/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.7812 - loss: 0.485419/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8833 - loss: 0.3563 -
val_accuracy: 0.8667 - val_loss: 0.4746
Epoch 54/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.353619/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8950 - loss: 0.3453 -
val_accuracy: 0.8333 - val_loss: 0.4796
Epoch 55/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.320519/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8883 - loss: 0.3508 -
val_accuracy: 0.8267 - val_loss: 0.4743
Epoch 56/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.331819/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8833 - loss: 0.3523 -
val_accuracy: 0.7933 - val_loss: 0.5350
Epoch 57/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.279219/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8800 - loss: 0.3466 -
```

```
val_accuracy: 0.8133 - val_loss: 0.5003
Epoch 58/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.284919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8917 - loss: 0.3287 -
val_accuracy: 0.8400 - val_loss: 0.4761
Epoch 59/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9375 - loss: 0.285719/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8917 - loss: 0.3089 -
val_accuracy: 0.7867 - val_loss: 0.4825
Epoch 60/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.7812 - loss: 0.445019/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8733 - loss: 0.3559 -
val_accuracy: 0.8667 - val_loss: 0.4830
Epoch 61/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.311519/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9033 - loss: 0.3209 -
val_accuracy: 0.8667 - val_loss: 0.4465
Epoch 62/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.276919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8917 - loss: 0.3156 -
val_accuracy: 0.8467 - val_loss: 0.4554
Epoch 63/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8125 - loss: 0.383919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9050 - loss: 0.2893 -
val_accuracy: 0.8733 - val_loss: 0.4089
Epoch 64/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.388119/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9083 - loss: 0.2892 -
val_accuracy: 0.8333 - val_loss: 0.4592
Epoch 65/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.247819/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8900 - loss: 0.2930 -
val_accuracy: 0.8333 - val_loss: 0.5332
Epoch 66/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.246319/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8917 - loss: 0.3182 -
val_accuracy: 0.8400 - val_loss: 0.4897
Epoch 67/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.210319/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9100 - loss: 0.2710 -
val_accuracy: 0.8733 - val_loss: 0.4637
Epoch 68/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.232619/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9117 - loss: 0.2826 -
val_accuracy: 0.8467 - val_loss: 0.4464
Epoch 69/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.208419/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9283 - loss: 0.2719 -
```

```
val_accuracy: 0.8333 - val_loss: 0.5431
Epoch 70/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.243719/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8950 - loss: 0.3104 -
val_accuracy: 0.8333 - val_loss: 0.4581
Epoch 71/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9062 - loss: 0.416319/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9100 - loss: 0.3025 -
val_accuracy: 0.8400 - val_loss: 0.5125
Epoch 72/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.382319/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8933 - loss: 0.2979 -
val_accuracy: 0.8133 - val_loss: 0.5513
Epoch 73/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.232719/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9183 - loss: 0.2578 -
val_accuracy: 0.8533 - val_loss: 0.4442
Epoch 74/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.148819/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9233 - loss: 0.2574 -
val_accuracy: 0.8533 - val_loss: 0.4712
Epoch 75/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.273819/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8950 - loss: 0.3066 -
val_accuracy: 0.8133 - val_loss: 0.5130
Epoch 76/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.233619/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8917 - loss: 0.3032 -
val_accuracy: 0.8800 - val_loss: 0.4324
Epoch 77/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.8438 - loss: 0.358119/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9033 - loss: 0.2684 -
val_accuracy: 0.8733 - val_loss: 0.4181
Epoch 78/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9062 - loss: 0.294819/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9250 - loss: 0.2405 -
val_accuracy: 0.8600 - val_loss: 0.4813
Epoch 79/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.326719/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9367 - loss: 0.2335 -
val_accuracy: 0.8733 - val_loss: 0.4116
Epoch 80/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.269919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9283 - loss: 0.2307 -
val_accuracy: 0.8533 - val_loss: 0.4615
Epoch 81/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.355919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9183 - loss: 0.2524 -
```

```
val_accuracy: 0.8733 - val_loss: 0.3947
Epoch 82/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 1.0000 - loss: 0.172319/19
━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9383 - loss: 0.2184 -
val_accuracy: 0.8400 - val_loss: 0.4951
Epoch 83/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.9688 - loss: 0.196819/19
━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9333 - loss: 0.2288 -
val_accuracy: 0.8867 - val_loss: 0.4338
Epoch 84/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 1.0000 - loss: 0.140519/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9350 - loss: 0.2219 -
val_accuracy: 0.8800 - val_loss: 0.3876
Epoch 85/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.214019/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9400 - loss: 0.2020 -
val_accuracy: 0.8733 - val_loss: 0.4799
Epoch 86/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.198119/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9050 - loss: 0.2775 -
val_accuracy: 0.8000 - val_loss: 0.5430
Epoch 87/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8125 - loss: 0.348919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8950 - loss: 0.2697 -
val_accuracy: 0.8333 - val_loss: 0.5434
Epoch 88/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.7500 - loss: 0.604919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9033 - loss: 0.2554 -
val_accuracy: 0.8467 - val_loss: 0.4325
Epoch 89/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.185519/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9250 - loss: 0.2296 -
val_accuracy: 0.8933 - val_loss: 0.3537
Epoch 90/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.207819/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9417 - loss: 0.2099 -
val_accuracy: 0.8733 - val_loss: 0.4025
Epoch 91/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.208619/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9250 - loss: 0.2183 -
val_accuracy: 0.8267 - val_loss: 0.4683
Epoch 92/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.242619/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9367 - loss: 0.2086 -
val_accuracy: 0.8600 - val_loss: 0.4121
Epoch 93/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.146319/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9417 - loss: 0.1891 -
```

```
val_accuracy: 0.8667 - val_loss: 0.4299
Epoch 94/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 1.0000 - loss: 0.134419/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9550 - loss: 0.1853 -
val_accuracy: 0.8467 - val_loss: 0.4267
Epoch 95/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9375 - loss: 0.215019/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9400 - loss: 0.1896 -
val_accuracy: 0.8600 - val_loss: 0.4010
Epoch 96/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.216219/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9217 - loss: 0.2072 -
val_accuracy: 0.8600 - val_loss: 0.4504
Epoch 97/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9688 - loss: 0.147519/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9417 - loss: 0.1847 -
val_accuracy: 0.8733 - val_loss: 0.3853
Epoch 98/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.142419/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9350 - loss: 0.1987 -
val_accuracy: 0.8533 - val_loss: 0.4987
Epoch 99/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.187219/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9233 - loss: 0.2280 -
val_accuracy: 0.8733 - val_loss: 0.4450
Epoch 100/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.193819/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9133 - loss: 0.2247 -
val_accuracy: 0.8400 - val_loss: 0.3943
Epoch 101/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9375 - loss: 0.161719/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9150 - loss: 0.2155 -
val_accuracy: 0.8733 - val_loss: 0.4429
Epoch 102/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9688 - loss: 0.175219/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9483 - loss: 0.1742 -
val_accuracy: 0.8867 - val_loss: 0.4251
Epoch 103/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.176519/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9533 - loss: 0.1801 -
val_accuracy: 0.8533 - val_loss: 0.4420
Epoch 104/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.138919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9433 - loss: 0.1817 -
val_accuracy: 0.8733 - val_loss: 0.4493
Epoch 105/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.222419/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9483 - loss: 0.1736 -
```

```
val_accuracy: 0.9067 - val_loss: 0.3821
Epoch 106/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.126419/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9383 - loss: 0.1717 -
val_accuracy: 0.8733 - val_loss: 0.3973
Epoch 107/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.133719/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9383 - loss: 0.1933 -
val_accuracy: 0.8733 - val_loss: 0.4007
Epoch 108/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.165619/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9250 - loss: 0.2082 -
val_accuracy: 0.8933 - val_loss: 0.4456
Epoch 109/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.132319/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9383 - loss: 0.1810 -
val_accuracy: 0.8867 - val_loss: 0.4247
Epoch 110/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.190819/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9250 - loss: 0.2036 -
val_accuracy: 0.8467 - val_loss: 0.5161
Epoch 111/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.218419/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9317 - loss: 0.1907 -
val_accuracy: 0.8800 - val_loss: 0.4658
Epoch 112/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.250019/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9450 - loss: 0.1831 -
val_accuracy: 0.8867 - val_loss: 0.4105
Epoch 113/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.090419/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9400 - loss: 0.1663 -
val_accuracy: 0.8933 - val_loss: 0.4011
Epoch 114/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.093619/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9483 - loss: 0.1865 -
val_accuracy: 0.8867 - val_loss: 0.4166
Epoch 115/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9062 - loss: 0.348819/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9100 - loss: 0.2188 -
val_accuracy: 0.8533 - val_loss: 0.4608
Epoch 116/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.099519/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9417 - loss: 0.1857 -
val_accuracy: 0.8667 - val_loss: 0.4236
Epoch 117/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.187019/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9367 - loss: 0.1868 -
```

```
val_accuracy: 0.8933 - val_loss: 0.4690
Epoch 118/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.238519/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9433 - loss: 0.1743 -
val_accuracy: 0.8867 - val_loss: 0.3896
Epoch 119/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.140819/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9450 - loss: 0.1661 -
val_accuracy: 0.8867 - val_loss: 0.4106
Epoch 120/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.123419/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9533 - loss: 0.1604 -
val_accuracy: 0.8800 - val_loss: 0.4453
Epoch 121/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.097519/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9217 - loss: 0.2305 -
val_accuracy: 0.8733 - val_loss: 0.3962
Epoch 122/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.162319/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9417 - loss: 0.1728 -
val_accuracy: 0.8600 - val_loss: 0.5055
Epoch 123/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.271219/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9383 - loss: 0.2035 -
val_accuracy: 0.8867 - val_loss: 0.4023
Epoch 124/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.120519/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9550 - loss: 0.1608 -
val_accuracy: 0.8533 - val_loss: 0.5308
Epoch 125/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.110919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9417 - loss: 0.1814 -
val_accuracy: 0.8733 - val_loss: 0.4598
Epoch 126/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.058019/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9500 - loss: 0.1719 -
val_accuracy: 0.8800 - val_loss: 0.4028
Epoch 127/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.269919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9050 - loss: 0.2561 -
val_accuracy: 0.8200 - val_loss: 0.5437
Epoch 128/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.420719/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9350 - loss: 0.1886 -
val_accuracy: 0.8467 - val_loss: 0.4752
Epoch 129/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.166119/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9233 - loss: 0.2162 -
```

```
val_accuracy: 0.8733 - val_loss: 0.4949
Epoch 130/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.166719/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9267 - loss: 0.1835 -
val_accuracy: 0.8733 - val_loss: 0.4121
Epoch 131/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.084719/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9400 - loss: 0.1651 -
val_accuracy: 0.8733 - val_loss: 0.4164
Epoch 132/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.107519/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9300 - loss: 0.1859 -
val_accuracy: 0.8800 - val_loss: 0.4326
Epoch 133/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.164419/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9367 - loss: 0.1755 -
val_accuracy: 0.8400 - val_loss: 0.4784
Epoch 134/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9062 - loss: 0.147019/19
━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9333 - loss: 0.1841 -
val_accuracy: 0.8267 - val_loss: 0.4896
Epoch 135/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9688 - loss: 0.094219/19
━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9567 - loss: 0.1510 -
val_accuracy: 0.8667 - val_loss: 0.5083
Epoch 136/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9062 - loss: 0.235219/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9317 - loss: 0.1761 -
val_accuracy: 0.8600 - val_loss: 0.4667
Epoch 137/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.122819/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9550 - loss: 0.1512 -
val_accuracy: 0.8867 - val_loss: 0.4390
Epoch 138/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.087719/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9383 - loss: 0.1631 -
val_accuracy: 0.9200 - val_loss: 0.3643
Epoch 139/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.130219/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9583 - loss: 0.1317 -
val_accuracy: 0.8800 - val_loss: 0.4274
Epoch 140/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.125919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9517 - loss: 0.1362 -
val_accuracy: 0.8933 - val_loss: 0.4511
Epoch 141/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.244819/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9617 - loss: 0.1380 -
```

```
val_accuracy: 0.8667 - val_loss: 0.4244
Epoch 142/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.066119/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9633 - loss: 0.1307 -
val_accuracy: 0.9000 - val_loss: 0.3787
Epoch 143/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.093219/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9600 - loss: 0.1187 -
val_accuracy: 0.8600 - val_loss: 0.4324
Epoch 144/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.072219/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9717 - loss: 0.1192 -
val_accuracy: 0.8933 - val_loss: 0.4255
Epoch 145/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.044919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9517 - loss: 0.1383 -
val_accuracy: 0.8800 - val_loss: 0.4528
Epoch 146/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.102719/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9667 - loss: 0.1268 -
val_accuracy: 0.8933 - val_loss: 0.3799
Epoch 147/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.171819/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9633 - loss: 0.1224 -
val_accuracy: 0.8533 - val_loss: 0.4732
Epoch 148/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.042419/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9567 - loss: 0.1243 -
val_accuracy: 0.8733 - val_loss: 0.4572
Epoch 149/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.133919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9617 - loss: 0.1412 -
val_accuracy: 0.8600 - val_loss: 0.4270
Epoch 150/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.172319/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9533 - loss: 0.1380 -
val_accuracy: 0.8867 - val_loss: 0.4391
Epoch 151/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.274719/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9400 - loss: 0.1715 -
val_accuracy: 0.8533 - val_loss: 0.4869
Epoch 152/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.153919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9500 - loss: 0.1398 -
val_accuracy: 0.8800 - val_loss: 0.4449
Epoch 153/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9375 - loss: 0.123519/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9467 - loss: 0.1674 -
```

```
val_accuracy: 0.8333 - val_loss: 0.5403
Epoch 154/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.439919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9417 - loss: 0.1791 -
val_accuracy: 0.8867 - val_loss: 0.4637
Epoch 155/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.220519/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9450 - loss: 0.1600 -
val_accuracy: 0.8067 - val_loss: 0.5327
Epoch 156/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.037119/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9450 - loss: 0.1395 -
val_accuracy: 0.8933 - val_loss: 0.4022
Epoch 157/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9062 - loss: 0.152119/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9417 - loss: 0.1525 -
val_accuracy: 0.9133 - val_loss: 0.3665
Epoch 158/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.117519/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9667 - loss: 0.1153 -
val_accuracy: 0.8667 - val_loss: 0.4473
Epoch 159/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.170919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9450 - loss: 0.1333 -
val_accuracy: 0.8733 - val_loss: 0.4353
Epoch 160/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9688 - loss: 0.128419/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9550 - loss: 0.1466 -
val_accuracy: 0.8933 - val_loss: 0.4680
Epoch 161/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.165419/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9650 - loss: 0.1203 -
val_accuracy: 0.8933 - val_loss: 0.4487
Epoch 162/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.107319/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9600 - loss: 0.1205 -
val_accuracy: 0.8733 - val_loss: 0.4596
Epoch 163/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9688 - loss: 0.090419/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9400 - loss: 0.1397 -
val_accuracy: 0.8800 - val_loss: 0.4827
Epoch 164/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.152219/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9600 - loss: 0.1280 -
val_accuracy: 0.8667 - val_loss: 0.4423
Epoch 165/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9688 - loss: 0.099719/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9700 - loss: 0.1124 -
```

```
val_accuracy: 0.8667 - val_loss: 0.5117
Epoch 166/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.095919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9533 - loss: 0.1312 -
val_accuracy: 0.9000 - val_loss: 0.4309
Epoch 167/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.127119/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9450 - loss: 0.1467 -
val_accuracy: 0.8733 - val_loss: 0.4750
Epoch 168/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.158619/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9533 - loss: 0.1526 -
val_accuracy: 0.8933 - val_loss: 0.4125
Epoch 169/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9062 - loss: 0.250819/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9433 - loss: 0.1593 -
val_accuracy: 0.9000 - val_loss: 0.4091
Epoch 170/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.8750 - loss: 0.158419/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9333 - loss: 0.1614 -
val_accuracy: 0.8733 - val_loss: 0.4464
Epoch 171/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9688 - loss: 0.108019/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9417 - loss: 0.1735 -
val_accuracy: 0.8400 - val_loss: 0.4972
Epoch 172/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.108219/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9233 - loss: 0.2002 -
val_accuracy: 0.9000 - val_loss: 0.4056
Epoch 173/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.114519/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9250 - loss: 0.1995 -
val_accuracy: 0.9067 - val_loss: 0.3936
Epoch 174/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.089819/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9267 - loss: 0.1736 -
val_accuracy: 0.8467 - val_loss: 0.5520
Epoch 175/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.124219/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9417 - loss: 0.1431 -
val_accuracy: 0.8533 - val_loss: 0.5770
Epoch 176/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.092419/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9333 - loss: 0.1795 -
val_accuracy: 0.8667 - val_loss: 0.4839
Epoch 177/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.258219/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9333 - loss: 0.1647 -
```

```
val_accuracy: 0.8667 - val_loss: 0.5355
Epoch 178/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.276319/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9233 - loss: 0.2118 -
val_accuracy: 0.8600 - val_loss: 0.4965
Epoch 179/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8750 - loss: 0.151019/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9217 - loss: 0.1798 -
val_accuracy: 0.8467 - val_loss: 0.5299
Epoch 180/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.103919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9433 - loss: 0.1397 -
val_accuracy: 0.8600 - val_loss: 0.4085
Epoch 181/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.096219/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9550 - loss: 0.1185 -
val_accuracy: 0.8733 - val_loss: 0.4492
Epoch 182/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.076319/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9733 - loss: 0.0980 -
val_accuracy: 0.9067 - val_loss: 0.4310
Epoch 183/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9688 - loss: 0.099919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9583 - loss: 0.1134 -
val_accuracy: 0.8800 - val_loss: 0.4450
Epoch 184/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.083319/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9567 - loss: 0.1177 -
val_accuracy: 0.8533 - val_loss: 0.4477
Epoch 185/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.221819/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9700 - loss: 0.1126 -
val_accuracy: 0.8867 - val_loss: 0.4565
Epoch 186/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.073619/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9650 - loss: 0.1060 -
val_accuracy: 0.8600 - val_loss: 0.4703
Epoch 187/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.086719/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9683 - loss: 0.1057 -
val_accuracy: 0.8733 - val_loss: 0.4558
Epoch 188/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.103919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9683 - loss: 0.0998 -
val_accuracy: 0.8867 - val_loss: 0.4834
Epoch 189/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.052519/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9550 - loss: 0.1265 -
```

```
val_accuracy: 0.8600 - val_loss: 0.4442
Epoch 190/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.138719/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9517 - loss: 0.1247 -
val_accuracy: 0.8667 - val_loss: 0.4965
Epoch 191/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9375 - loss: 0.101719/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9550 - loss: 0.1473 -
val_accuracy: 0.8467 - val_loss: 0.6307
Epoch 192/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9062 - loss: 0.256919/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9617 - loss: 0.1330 -
val_accuracy: 0.8667 - val_loss: 0.4915
Epoch 193/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.128519/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9600 - loss: 0.1335 -
val_accuracy: 0.9067 - val_loss: 0.4544
Epoch 194/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.080719/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9650 - loss: 0.1022 -
val_accuracy: 0.9067 - val_loss: 0.3927
Epoch 195/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.105319/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9667 - loss: 0.1047 -
val_accuracy: 0.8667 - val_loss: 0.4578
Epoch 196/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.028519/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9717 - loss: 0.1013 -
val_accuracy: 0.9333 - val_loss: 0.3818
Epoch 197/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.088319/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9650 - loss: 0.1125 -
val_accuracy: 0.9000 - val_loss: 0.4368
Epoch 198/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.107419/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9550 - loss: 0.1359 -
val_accuracy: 0.8667 - val_loss: 0.5120
Epoch 199/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss: 0.034619/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9650 - loss: 0.1047 -
val_accuracy: 0.8600 - val_loss: 0.4815
Epoch 200/200
 1/19 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9688 - loss: 0.090319/19
━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9650 - loss: 0.0951 -
val_accuracy: 0.9200 - val_loss: 0.4188
```

In diesem Beispiel ist `X` die Designmatrix, `y` der Ergebnisvektor, und das Argument `epochs` gibt an, wie viele Epochen der Optimierer durchlaufen soll. *Keras* gibt den Fortschritt beim Optimieren an, es könnte so aussehen

```
Epoch 1/200
25/25 [==============================] - 0s 2ms/step - loss: 1.5984 -
accuracy: 0.2438
Epoch 2/200
25/25 [==============================] - 0s 2ms/step - loss: 1.5709 -
accuracy: 0.2763
Epoch 3/200
25/25 [==============================] - 0s 3ms/step - loss: 1.5550 -
accuracy: 0.3013
```

Hierbei kann man die aktuelle Epoche, den Batch (für stochastischen Gradientenabstieg) und die aktuelle Genauigkeit der Trainingsdaten sehen. In diesem Beispiel werden alle Epochen bis zum Abschluss ausgeführt, daher sehen wir `32/32` für Batches, aber normalerweise kann man die Batch-Nummer sehen, die sich während des Trainings fortschreitet. Dieses Beispiel funktioniert sehr schnell, Keras meldet 2 ms pro Schritt (Batch) und die Gesamtzeit für die Epoche ist zu gering, um gemeldet zu werden. Aber eine einzelne Epoche kann bei komplexeren Modellen und mehr Daten viele Minuten dauern.

> ⚠ Achtung: Batchqualität vs. Modellqualität
>
> Die während des Trainings angezeigte Modellqualität (hier Genauigkeit) ist eine laufende Trainingsmetrik und kann zwischen Batches und Epochen schwanken. Sie gibt nur eine grobe Richtlinie für die tatsächliche Modellgüte und sollte durch eine getrennte Auswertung auf Validierungs- oder Testdaten ergänzt werden.

> ⚠ Achtung: Modelle fertig trainieren
>
> Keras ermöglicht es Ihnen, Vorhersagen mit einem Modell zu treffen, das nicht angepasst ist (im Gegensatz zu scikit-learn, wo das einen Fehler verursacht). Die Ergebnisse werden bestenfalls mittelmäßig aussehen.

Vorhersagen und Plotten

Wenn die Anpassung abgeschlossen ist, können wir das Modell für **Vorhersagen** verwenden. Die Vorhersage funktioniert ähnlich wie in sklearn, nur dass die `predict`-Methode die Wahrscheinlichkeit vorhersagt, nicht die Kategorie (analog zu sklearn's `predict_proba`).

```
phat = model.predict(X_test)
```

```
1/8 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
```

In diesem Beispiel handelt es sich um eine Matrix mit 5 Spalten, wobei jede Spalte die Wahrscheinlichkeit darstellt, dass der Datenpunkt zur entsprechenden Kategorie gehört. Beispielsweise könnten Zeilen von `phat` so aussehen:

```
phat[:5]
```

```
array([[5.03318499e-29, 1.93861095e-12, 2.26488590e-01, 7.72773981e-01,
        7.37518945e-04],
       [4.17263433e-02, 9.57733691e-01, 5.39998990e-04, 8.77686011e-17,
        2.63804809e-24],
       [9.99989986e-01, 1.00534035e-05, 2.61536062e-12, 1.66430813e-28,
        9.19641017e-36],
       [9.99215961e-01, 7.83961150e-04, 2.45871279e-09, 8.93183847e-23,
        1.31478928e-28],
       [9.99997973e-01, 1.97123086e-06, 8.40218821e-15, 2.47281602e-33,
        0.00000000e+00]], dtype=float32)
```

Wir sehen, dass für jedes Test-Fall eine Wahrscheinlichkeit für jede Kategorie angegeben ist. Als nächstes wollen wir mit `np.argmax(phat, axis=-1)` die Spalte (Kategorie) finden, mit der höchsten Wahrscheinlichkeit. Es findet einfach die Position der größten Elemente im Array entlang der letzten Achse (`axis=-1`), d.h. Spalten. Für jede Zeile finden wir also die entsprechende Spaltennummer. Beachten Sie, dass `np.argmax` die Spalten ab 0 zählt, nicht ab 1:

```
yhat = np.argmax(phat, axis=-1)
yhat[:5]
```

```
array([3, 1, 0, 0, 0])
```

Endlich können wir die Verwirrungsmatrix mit `pd.crosstab` berechnen und die Genauigkeit berechnen:

```
from sklearn.metrics import confusion_matrix

print("confusion matrix:\n", confusion_matrix(y_test, yhat))
print("Accuracy (on test data):", np.mean(y_test == yhat))
```

```
confusion matrix:
 [[52  2  0  0  0]
 [ 3 29  1  0  1]
 [ 1  6 60  3  0]
 [ 0  0  1 35  2]
```

```
 [ 0  0  0  6 48]]
Accuracy (on test data): 0.896
```

In diesem Beispiel machen wir Vorhersagen auf Testdaten, aber wir können natürlich auch einen anderen Datensatz für Vorhersagen auswählen. Da der vorhergesagte Wert eine Wahrscheinlichkeitsmatrix mit 5 Spalten sein wird, berechnen wir `yhat` als die Spaltennummer, die die größte Wahrscheinlichkeit für jede Zeile enthält.

Wie man sehen kann, ist die Verwirrungsmatrix fast ausschließlich auf der Hauptdiagonalen besetzt, und die Genauigkeit ist hoch.

Schließlich müssen wir, wenn wir ein ähnliches Diagramm wie oben erstellen möchten, die `DBPlot`-Funktion anpassen, um zu berücksichtigen, dass Keras-Modelle nur Wahrscheinlichkeiten vorhersagen.

```
def DBPlot(m, X, y, nGrid = 300):
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max, nGrid),
                           np.linspace(x2_min, x2_max, nGrid))
    XX = np.column_stack((xx1.ravel(), xx2.ravel()))
    ## predict probability
    phat = m.predict(XX, verbose=0)
    ## find the column that corresponds to the maximum probability
    hatyy = np.argmax(phat, axis=-1).reshape(xx1.shape)
    fig = px.imshow(hatyy, width=600, height=600)
    fig.add_scatter(x=X[:,0]/(x1_max-x1_min)*nGrid+nGrid/2, y=X[:,1]/(x2_max-
x2_min)*nGrid+nGrid/2, mode="markers",
                    marker=dict(color=[mcolors[i] for i in y]),
marker_line=dict(width=.3, color="black"))
    fig.update_coloraxes(showscale=False)
    fig.update_layout(showlegend=False)
    fig.show()
```

```
DBPlot(model, X_test, y_test)
```

```
Unable to display output for mime type(s): text/html
```

Noch einmal, die Funktion ist fast identisch mit der *sklearn* Version, außer der Zeile, die `np.argmax(phat, axis=-1)` enthält, die die vorhergesagten Wahrscheinlichkeiten in Kategorien umwandelt.

Referenzen

Bibliographie

[1] W. S. McCulloch und W. Pitts, „A logical calculus of the ideas immanent in nervous activity“, *The bulletin of mathematical biophysics*, Bd. 5, Nr. 4, S. 115–133, 1943.

[2] F. Rosenblatt, „The perceptron: a probabilistic model for information storage and organization in the brain.“, *Psychological review*, Bd. 65, Nr. 6, S. 386, 1958.