Bert Gollnick

PyTorch

KI-Modelle trainieren, tunen und einsetzen

+ Deep Learning von der Datenaufbereitung über Training und Finetuning bis zum Deployment
+ Vielfältige Architekturen wie Autoencoder, RNNs, LLMs und RAG-Systeme
+ Inkl. PyTorch Lightning, TensorBoard, LangChain, FastAPI u.v.m.

> Mit lokal lauffähigen Beispielprojekten zum Download

Rheinwerk Computing

Kapitel 3
Unser erstes PyTorch-Modell

»Jede hinreichend fortschrittliche Technologie ist von Magie nicht zu unterscheiden.«
– Arthur C. Clarke, Autor und Physiker

Für mich fühlte sich das erste *Deep-Learning*-Modell, das ich trainiert habe, genauso an – wie Magie. Möglicherweise wird es Ihnen ähnlich gehen.

In diesem Kapitel legen wir die Grundlage für alle weiteren Kapitel. Sie werden lernen, wie PyTorch-Modelle trainiert werden.

Zunächst werden wir ein Modell von Grund auf trainieren und hierbei nahezu jeden Schritt »von Hand« implementieren. Das wird Ihnen helfen, ein besseres Verständnis für die Arbeitsweise von PyTorch zu erlangen.

Danach werden wir sukzessiv mehr Features von PyTorch nutzen, um unsere Skripte so modular wie möglich zu definieren. Im Idealfall soll unser Skript am Ende so modular sein, dass ein anderer Datensatz verwendet werden kann und das Modell trainiert wird, ohne dass an diversen Stellen im Code Anpassungen notwendig sind.

Auf dem Weg werden wir uns in Abschnitt 3.1 zunächst mit dem Datensatz vertraut machen und diesen vorbereiten. In Abschnitt 3.2 trainieren wir unser Modell, das wir im Anschluss verbessern.

Im Anschluss werden Sie in Abschnitt 3.3 lernen, wie eine Modellklasse definiert wird. Das Konzept der *Batches* wird in Abschnitt 3.4 eingeführt. Der Datensatz wird mittels *Dataset*- und *DataLoader*-Klasse abstrahiert. Dieser Thematik widmen wir uns in Abschnitt 3.5.

Da wir nicht immer wieder das Modell neu trainieren wollen, erkläre ich in Abschnitt 3.6, wie Modelle und deren *Modellgewichte* gespeichert und geladen werden können.

Zum Abschluss dieses Kapitels widmen wir uns in Abschnitt 3.7 dem *Data Sampling*. Hierbei geht es letztlich darum, sicherzustellen, dass das Modell zu generalisieren lernt, damit es nicht nur die Trainingsdaten gut vorhersagen kann, sondern im besten Fall jegliche Daten, die es noch nie zuvor gesehen hat.

Ganz allgemein kann gesagt werden, dass es in diesem Kapitel weniger um Regression geht, sondern um das gesamte »Drumherum« des Modelltrainings.

Aber bevor wir in unser erstes Modelltraining einsteigen, müssen wir uns zunächst mit dem Datensatz vertraut machen, für den das Modell trainiert werden soll.

3.1 Datenvorbereitung

Wir arbeiten mit einem Datensatz von *Kaggle* (*www.kaggle.com*). Kaggle ist eine Online-Community mit speziellem Fokus auf Datenanalysten und Data Scientists. Mit dieser Plattform ist es möglich, Datensätze zu erforschen, Analysen durchzuführen und von anderen zu lernen, die bereits mit den Daten gearbeitet haben. Es ist eine sehr wertvolle Quelle von Wissen.

Im Speziellen werden wir mit dem *Social-Anxiety-Datensatz* (*https://www.kaggle.com/datasets/natezhang123/social-anxiety-dataset*) arbeiten. Er enthält mehr als 10.000 Stichproben von Personen, die unterschiedlich starke soziale Ängste aufweisen. Das Angst-Level folgt einer Punkteskala von 1 bis 10. Das ist die Zielgröße (oder auch *abhängige Größe* genannt), die vom Modell letztlich vorhergesagt werden soll.

Abbildung 3.1 zeigt einen Ausschnitt des Datensatzes.

	Age	Gender	Occupation	Sleep Hours	Physical Activity (hrs/week)	Caffeine Intake (mg/day)	Alcohol Consumption (drinks/week)	Smoking	Family History of Anxiety	Stress Level (1-10)	Heart Rate (bpm)
0	29	Female	Artist	6.0	2.7	181	10	Yes	No	10	114
1	46	Other	Nurse	6.2	5.7	200	8	Yes	Yes	1	62
2	64	Male	Other	5.0	3.7	117	4	No	Yes	1	91
3	20	Female	Scientist	5.8	2.8	360	6	Yes	No	4	86
4	49	Female	Other	8.2	2.3	247	4	Yes	No	1	98

Abbildung 3.1 Ein Ausschnitt des Social-Anxiety-Datensatzes

3.1.1 Feature-Typen

Erklärt werden kann die Zielgröße von verschiedenen unabhängigen Variablen. Hierzu gehören demografische Merkmale wie Alter, Geschlecht oder Beruf. Weitere Merkmale sind den Bereichen »allgemeine Gesundheit«, »mentale Indikatoren« sowie »mentale Gesundheit« zuzuordnen.

Unabhängige und abhängige Features

Die Begriffe *unabhängige* und *abhängige Features* beziehen sich auf die Rollen, die Variablen (Spalten) in einem Datensatz spielen. Das Konzept kommt vor allem beim überwachten (supervised) Lernen zum Einsatz.

- Unabhängige Features (engl. Independent Features) werden auch Eingabevariablen, Prädiktoren oder Merkmale genannt. Das sind die Eingaben für ML-Modelle. Man geht davon aus, dass diese Features die Ursachen oder beeinflussende Faktoren für die abhängige Variable sind.
- Abhängige Features (engl. *Dependent Features*) werden in der Statistik auch Zielvariablen, Ausgabevariablen oder Label genannt. Das ist der Ausgabewert, der letztlich vom Modell vorhergesagt wird.

Das ML-Modell lernt somit den Zusammenhang bzw. die Muster, die zwischen den unabhängigen und der abhängigen Variable bestehen. Auf der Basis können, nachdem das Modell trainiert wurde, zukünftige Werte der abhängigen Variable basierend auf neuen Werten der unabhängigen Features vorhergesagt werden.

Am Beginn jedes Skripts laden wir alle benötigten Pakete und Klassen. Der Datensatz kommt wie zuvor erwähnt von Kaggle und wird direkt mittels des hauseigenen Pakets `kagglehub` importiert. Die Daten werden als `pandas`-Dataframe importiert. Das Paket `numpy` benötigen wir, um später die Daten von einem Dataframe in ein `numpy`-Array umzuwandeln.

Auf das Thema Skalierung der Daten gehe ich später noch ein. An dieser Stelle laden wir den `StandardScaler` aus dem Paket `sklearn`. Das Paket `os` nutzen wir immer, wenn wir mit Funktionen des Betriebssystems interagieren wollen.

Zur Visualisierung der Daten und Ergebnisse verwenden wir `seaborn` und `matplotlib`:

```
#%% packages
import numpy as np
import pandas as pd
import kagglehub
import os
from sklearn.preprocessing import StandardScaler
import seaborn as sns
import matplotlib.pyplot as plt
```

Listing 3.1 Datenvorbereitung – Paketimport

Kaggle bietet uns einen einfachen Weg, um die Daten mithilfe des Pakets `kagglehub` in Python zu importieren. Wir müssen nur den Datensatz über dessen ID laden. Während des Ladens wird der Datensatz auf die Festplatte kopiert und der Ordner zurückgeliefert. In dem Ordner ist die Datei gespeichert und kann nun direkt über die Funktion `pd.read_csv()` geladen werden. Nun haben wir die Daten erfolgreich geladen und den Dataframe `anxiety` erstellt:

```
#%% Download latest version
path = kagglehub.dataset_download("natezhang123/social-anxiety-dataset")
print("Path to dataset files:", path)
#%% data import
anxiety_file = os.path.join(path, 'enhanced_anxiety_dataset.csv')
anxiety = pd.read_csv(anxiety_file)
```

```
Path to dataset files: C:\Users\BertGollnick\.cache\kagglehub\datasets\
natezhang123\social-anxiety-dataset\versions\2
```

Listing 3.2 Datenvorbereitung – Paket- und Datenimport (Quelle: 030_FirstModel_Regression\DataPrep.py)

Schauen wir uns nun an, welche Spalten der Datensatz aufweist und wie viele Zeilen und Spalten er hat:

```
#%% check data
print(f"anxiety.columns: {anxiety.columns}")
print(f"anxiety.shape: {anxiety.shape}")
anxiety.columns: Index(['Age', 'Gender', 'Occupation', 'Sleep Hours',
       'Physical Activity (hrs/week)', 'Caffeine Intake (mg/day)',
       'Alcohol Consumption (drinks/week)', 'Smoking',
       'Family History of Anxiety', 'Stress Level (1-10)',
       'Heart Rate (bpm)', 'Breathing Rate (breaths/min)',
       'Sweating Level (1-5)', 'Dizziness', 'Medication',
       'Therapy Sessions (per month)', 'Recent Major Life Event',
       'Diet Quality (1-10)', 'Anxiety Level (1-10)'],
      dtype='object')
anxiety.shape: (11000, 19)
```

Insgesamt umfasst der Datensatz 11.000 Stichproben und 19 Features. Einige davon beinhalten keine numerischen Informationen, sondern Texte.

Das ist zum Beispiel bei dem Feature `Smoking` der Fall, das die zwei Zustände `Yes` und `No` aufweist.

3.1.2 Datentypen

Vergegenwärtigen wir uns an dieser Stelle, welche Typen von Daten es gibt.

Typen von Daten

Es werden im Allgemeinen zwei Haupttypen bei Daten unterschieden: *numerische Daten* und *kategorische Daten*:

- Numerische Daten werden auch quantitative Daten oder metrische Daten genannt. Diese Daten bestehen aus Zahlen, die gemessen werden können.
- Kategorische Daten werden auch qualitative oder nominale Daten genannt. Sie beschreiben Qualitäten oder Kategorien. Sie können nicht im herkömmlichen Sinne gemessen oder gezählt werden. Typische Beispiele sind das Geschlecht oder der Beruf. Man kann hier noch weiter unterteilen in nominale Daten, die ungeordnet sind (z. B. Lieblingsfarben) und ordinale Daten. Letztere sind Kategorien mit einer natürlichen Reihenfolge. Ein ganz übliches Beispiel sind Bildungsabschlüsse.

Da PyTorch nur numerische Daten verarbeiten kann, müssen alle Features, die kategorische Informationen enthalten, in numerische Informationen umgewandelt werden. Das wird mittels One-Hot Encoding erreicht.

3.1.3 One-Hot Encoding

One-Hot Encoding ist eine spezielle Technik, die im *Machine Learning* (ML) eingesetzt wird, um kategorische Daten in ein numerisches Format umzuwandeln. Nur so können die Daten von Algorithmen verarbeitet werden.

Wie funktioniert One-Hot Encoding?

Das zugrunde liegende Konzept können wir uns anhand eines Beispiels verdeutlichen. Stellen Sie sich vor, dass in einem Datensatz über Personen die Spalte *Lieblingsfarbe* erfasst wurde:

Person	Lieblingsfarbe
Bob	Gelb
Stuart	Grün
Kevin	Rot
Gru	Grün

Beim One-Hot Encoding werden alle erfassten eindeutigen Werte als einzelne Spalte dargestellt. Nach der Anwendung von One-Hot Encoding wird die Spalte *Lieblingsfarbe* in so viele Spalten umgewandelt, wie es eindeutige Ausprägungen gibt. In unserem Beispiel gibt es drei eindeutige Ausprägungen [Gelb, Grün, Rot]. Aus diesen entstehen die Spalten *Lieblingsfarbe_gelb*, *Lieblingsfarbe_grün* und *Lieblingsfarbe_rot*.

Diese Spalten enthalten nur binäre Informationen – also 1, wenn es der Lieblingsfarbe entspricht, und 0, falls nicht. Für jede Person wird dann eine 1 in die Spalte eingetragen, die der Lieblingsfarbe entspricht.

Die oben dargestellte Tabelle sieht nach dem One-Hot Encoding dann wie folgt aus:

Person	Lieblingsfarbe_gelb	Lieblingsfarbe_grün	Lieblingsfarbe_rot
Bob	1	0	0
Stuart	0	1	0
Kevin	0	1	0
Gru	0	1	0

Man kann sogar eine Spalte weglassen, ohne Informationen zu verlieren, weil sich diese Spalte dann implizit aus den anderen Spalten ergibt. Konkret funktioniert das, wenn es nur die Farben Gelb, Grün und Rot gibt und jede Person genau eine Lieblingsfarbe besitzt.

Vorteile von One-Hot Encoding

In dieser Form sind die Informationen nun numerisch dargestellt und eignen sich daher für die meisten ML-Algorithmen. Ein weiterer Vorteil ist, dass es keine implizite Ordnung gibt. Stellen Sie sich vor, die ursprünglichen Farben wären numerisch kodiert gewesen, zum Beispiel Gelb = 1, Grün = 2, Rot = 3. Dann hätte die ursprüngliche Form bereits formell die Anforderungen von ML-Algorithmen erfüllt, da die Information numerisch kodiert worden wäre. Aber der Algorithmus hätte implizit eine Ordnung der Farben »angenommen«, und zwar in der Form, dass Grün doppelt so viel zählt wie Gelb und Rot dreimal so viel wie Gelb, was keinen Sinn macht. Solche Probleme können mit One-Hot Encoding umgangen werden.

Nachteile von One-Hot Encoding

Ein eindeutiger Nachteil ist, dass die Anzahl der Dimensionen zunimmt. Gerade wenn es viele verschiedene Ausprägungen gibt, schlägt sich das in einer großen Anzahl neuer Features nieder.

Damit verbunden ist eine erhöhte Trainingszeit des Modells sowie der sogenannte *Fluch der Dimensionalität*[1]. Damit sind Probleme gemeint, die auftreten, wenn die Anzahl der Features im Vergleich zur Anzahl der Datenpunkte groß ist.

Wenden wir nun diese neu erlernte Technik auf unsere Daten an. Dankenswerterweise haben die Entwickler des `pandas`-Pakets uns die Arbeit hier sehr leicht gemacht, sodass wir die One-Hot-Kodierung mit der Methode `pd.get_dummies` erstellen können.

Listing 3.3 verdeutlicht, wie das One-Hot Encoding implementiert wird. Neben dem Datensatz `anxiety` werden einige weitere Parameter übergeben. Der Parameter `drop_`

1 Der Begriff »Fluch der Dimensionaliät« wurde vom Mathematiker Richard E. Bellman geprägt. Er verwendete den Begriff erstmals in den 1950er-Jahren.

`first` sorgt dafür, dass die erste kodierte Spalte weggelassen wird und man Dummy-Variablen erhält:

```
anxiety_dummies = pd.get_dummies(anxiety, drop_first=True, dtype=int)
anxiety_dummies.head()
#%% df shape
anxiety_dummies.shape
```

```
(11000, 31)
```

Listing 3.3 Datenvorbereitung – One-Hot Encoding (Quelle: 030_FirstModel_Regression\DataPrep.py)

Durch die Anwendung dieser Technik hat sich die Anzahl der Spalten von 19 auf 31 erhöht. Nun können wir zum besseren Verständnis einen Blick auf einen Zusammenhang der Daten werfen.

3.1.4 Explorative Datenanalyse

Insbesondere schauen wir uns an, wie sich das Schlafverhalten auf die Angststörung auswirkt.

Der entsprechende Code ist in Listing 3.4 dargestellt:

```
sns.regplot(x='Sleep Hours', y='Anxiety Level (1-10)', data=anxiety_
dummies, color='blue', line_kws={'color': 'red'})
# add a title
plt.title('Sleep Hours vs Anxiety Level')
# add x title
plt.xlabel('Sleep Hours')
# add y title
plt.ylabel('Anxiety Level')
```

Listing 3.4 Datenvorbereitung – Datenvisualisierung (Quelle: 030_FirstModel_Regression\DataPrep.py)

Es ergibt sich das in Abbildung 3.2 gezeigte Bild für den Zusammenhang. Die Datenpunkte werden als Punktdiagramm dargestellt. Zusätzlich ist die lineare Korrelation zwischen den beiden Größen als Linie verdeutlicht.

Der Zusammenhang ist ziemlich eindeutig: Mit schlechterem Schlaf (geringe Schlafdauer) steigt das Angst-Level.

Das ist nur ein möglicher Zusammenhang. Wir haben insgesamt 30 unabhängige Features, die wir uns ansehen könnten.

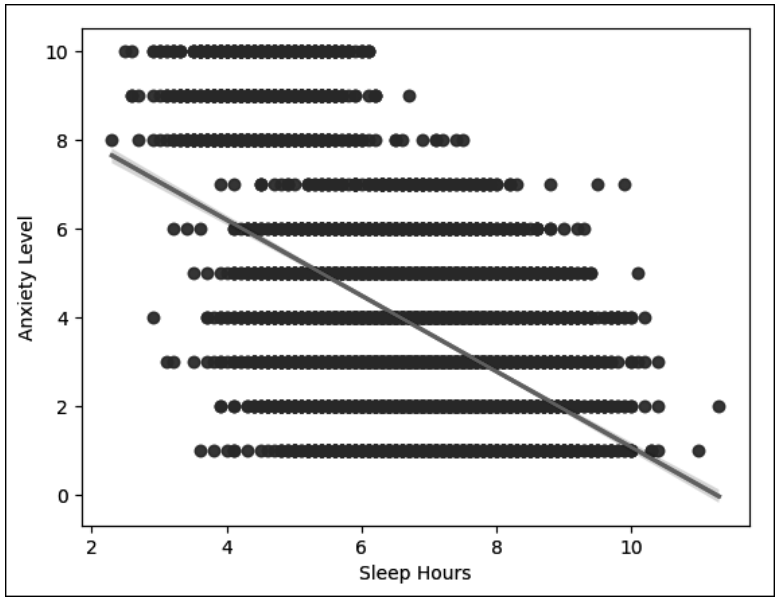


Abbildung 3.2 Zusammenhang zwischen Schlaf und Angststörung

Um schnell einen Überblick zu erhalten, kann man die Korrelation zwischen den unabhängigen Features und der Zielgröße ermitteln und in einer *Korrelationsmatrix* als Heatmap darstellen. Eine *Heatmap* ist eine Diagrammform, bei der die kategorischen Informationen als Farbwerte codiert werden. Hierbei wird die lineare Korrelation zwischen allen Größen ermittelt und kann anschließend als Farbwert visualisiert werden.

Listing 3.5 verdeutlicht, wie die Korrelationen ermittelt werden. Der besseren Übersichtlichkeit halber werden nur numerische Features analysiert. Der gefilterte Pandas-Dataframe `numerical_features` besitzt die Methode `corr()`. Mit ihr kann die lineare Korrelation zwischen allen Features ermittelt werden. Bei N Spalten ergibt sich hieraus eine Korrelationsmatrix `corr` mit den Dimensionen NxN.

```
#%% check correlation
# Select only numerical features for correlation analysis
numerical_features = anxiety.select_dtypes(include=['int64', 'float64'])
corr = numerical_features.corr()
```

Listing 3.5 Datenvorbereitung – Ermittlung der Korrelation

Listing 3.6 zeigt, wie diese Korrelationen nun mit `sns.heatmap` visualisiert werden können. Da die Matrix symmetrisch ist, reicht es aus, das obere oder untere Dreieck zu betrachten. Das können Sie über eine Maskierung `mask` implementieren, die dann

als Parameter der Heatmap übergeben wird. Diese Maske besteht aus NxN Boolean-Werten und gibt an, welche Werte somit dargestellt werden sollen:

```
# Create mask for upper triangle
mask = np.triu(np.ones_like(corr, dtype=bool))

# Plot correlation heatmap
sns.heatmap(corr, annot=False, cmap='coolwarm', vmin=-1, vmax=1, mask=mask)
plt.title('Correlation Heatmap (Numerical Features Only)', fontsize=10)
plt.xticks(rotation=45, ha='right', fontsize=8)
plt.yticks(rotation=0, ha='right', fontsize=8)
plt.tight_layout()
plt.show()
```

Listing 3.6 Datenvorbereitung – Visualisierung der Korrelationen

Unsere Visualisierung der numerischen Features sehen Sie in Abbildung 3.3. Hierbei reicht die Farbcodierung von –1 über 0 bis zu +1.

Abbildung 3.3 Korrelation der numerischen Features

Ein Korrelationskoeffizient von +1 stellt den maximal positiven Zusammenhang dar. Das kann man so interpretieren, dass ein steigender Wert des einen Features mit einem steigenden Wert des anderen Features *einhergeht*. Man kann hier nicht sagen,

dass der steigende Werte des einen Features den steigenden Wert des anderen Features *bedingt* oder *zur Folge hat*. Das würde heißen, dass eine Kausalität zwischen den beiden Größen herrscht. Es heißt erst mal nur, dass es einen Zusammenhang gibt. Ob dieser Zusammenhang kausal ist, kann auf dieser Basis jedoch nicht gesagt werden.

Umgekehrt gilt, dass ein Korrelationskoeffizient von –1 einen perfekt negativen Zusammenhang darstellt. Das heißt, dass ein steigender Wert des einen Features mit einem fallenden Wert des anderen Features einhergeht.

Wir sind vor allem an den Korrelationen zwischen unserer Zielgröße `Anxiety Level (1-10)` und den beschreibenden Features interessiert. Diese sind in der letzten Zeile von Abbildung 3.3 dargestellt. An der Stelle wird deutlich, dass das *Anxiety Level* stark mit *Sleep Hours* und *Stress Level* korreliert ist.

Bis zu diesem Punkt haben wir die Daten in einem `pandas`-Dataframe gespeichert.

Wir müssen nun zwei Dinge bearbeiten: Erstens müssen wir die Daten in unabhängige und abhängige Features trennen, und zweitens müssen wir die Daten in `numpy`-Arrays umwandeln, also in reine Zahlenmatrizen.

Beide Schritte werden im folgenden Listing 3.7 vereint. Die unabhängigen Features werden im Objekt `X` und die abhängigen im Objekt `y` gespeichert. Diese Begrifflichkeit stammt aus der Mathematik. Das steht im Widerspruch zu Namenskonventionen in Python – vor allem das große X, aber da die Begriffe so verbreitet sind, folge ich an dieser Stelle der statistischen Konvention.

Die unabhängigen Features entsprechen allen Features des `anxiety_dummies`-Datensatzes, außer der Spalte mit der Zielgröße. Im Gegensatz dazu steht das unabhängige Feature `y`, in dem nur die Zielgröße gespeichert wird.

Letztlich überprüfen wir die Ausgabe, indem wir uns die Größen der Objekte veranschaulichen:

```
#%% convert data to numpy array
X = np.array(anxiety_dummies.drop(
    columns=['Anxiety Level (1-10)']),
    dtype=np.float32)
y = np.array(anxiety_dummies[['Anxiety Level (1-10)']],
    dtype=np.float32)
print(f"X shape: {X.shape}, y shape: {y.shape}")
```

```
X shape: (11000, 30), y shape: (11000, 1)
```

Listing 3.7 Datenvorbereitung – Umwandlung der Daten in numpy-Arrays (Quelle: 030_FirstModel_Regression\DataPrep.py)

Von den 31 ursprünglichen Spalten sind nun 30 im Objekt `X` und eine im Objekt `y` übernommen worden.

3.1.5 Skalierung

Im nächsten Schritt geht es um die Skalierung der Daten. Hier schauen wir uns zunächst an, warum dieser Schritt überhaupt notwendig ist.

Daten-Skalierung

Die Skalierung der Daten spielt beim Training vieler Modelle eine entscheidende Rolle. Warum ist das so? Rohdaten, die in ihren Werten sehr stark variieren, können beim Training zu Problemen führen. Große Werte können dazu führen, dass Gradienten während des Backpropagation-Prozesses »explodieren«. Dadurch würde das Training instabil werden und sogar ganz scheitern.

Umgekehrt könnten sehr kleine Werte zu verschwindenden Gradienten (siehe Abschnitt 2.7.2) führen, wodurch das Lernen ebenso instabil werden könnte.

Die Skalierung der Daten wird mit dem Ziel durchgeführt, die Werte der Eingabefeatures in einen ähnlichen Wertebereich zu transformieren. Hierfür gibt es verschiedene Arten:

- Eine geläufige Art ist die *Min-Max-Skalierung*. Dabei werden die Daten üblicherweise in den Wertebereich 0 bis 1 skaliert.
- Ein anderer Ansatz ist die *Standardisierung*. Dabei werden die Daten so transformiert, dass sie um einen Mittelwert von 0 schwanken und eine Standardabweichung von 1 aufweisen. Wichtig ist hierbei auch, dass die Skalierung konsistent ist, um vergleichbare Ergebnisse zu erzielen.

Die Parameter der Skalierung (Mittelwerte und Standardabweichungen) sollten ausschließlich auf den Trainingsdaten berechnet werden und erst dann auf den Validierungs- und Testdatensatz angewandt werden. So kann *Data Leakage* vermieden werden. Auf diese Aspekte zum Thema Datenaufteilung komme ich in Abschnitt 3.7, »Data Sampling«, zu sprechen.

Die Skalierung, in unserem Falle die *Standardisierung*, kann mithilfe der Klasse `StandardScaler` durchgeführt werden. Zunächst wird eine Instanz der Klasse erstellt, und im Anschluss werden die Daten der Methode `fit_transform` übergeben, die die Parameter ermittelt und die Standardisierung durchgeführt wird. Das finale Objekt `X` beinhaltet die standardisierten Daten.

```
#%% normalize data
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

Damit haben wir unsere Daten hinreichend vorbereitet und sind in der Lage, unser erstes Modell zu trainieren.

3.2 Modell-Erstellung

In unserem ersten Modell werden wir noch viele Details selbst implementieren, da Ihnen das beim Verstehen des Modells helfen wird. So werden wir zum Beispiel die Vorhersagen des Modells mittels Matrixmultiplikation ermitteln, die Modellparameter selbst implementieren und die Anpassung der Modellparameter eigenständig vornehmen.

Die hierbei trainierten Modellparameter, *Slopes* und *Offsets*, bezeichnen die zwei wichtigsten lernbaren Parameter innerhalb eines Neurons oder einer linearen Transformation.

Später werden wir diese Aufgaben mehr und mehr dem Framework PyTorch übergeben. Würden wir das aber von vornherein machen, blieben viele Aspekte des Modelltrainings Blackboxes, die Sie nicht ganz verstehen würden.

Letztlich trainieren wir ein Modell, um `y` (das Angstlevel) auf Basis einer Vielzahl von unabhängigen Features vorherzusagen:

$$y = w_1 \cdot X_1 + w_2 \cdot X_2 + ... + w_{30} \cdot X_{30} + b$$

3.2.1 Datenimport

Wir beginnen wie gehabt mit dem Import der Pakete. Da wir direkt auf der Datenvorbereitung des vorherigen Abschnittes aufbauen, importieren wir die unabhängigen Features `X` und das abhängige Feature `y` direkt aus dem Skript `Dataprep`. Wir laden auch `numpy` und `torch` für die Erstellung von Tensoren. Für die Visualisierung werden `seaborn` und `matplotlib` geladen. Letztlich nutzen wir den R^2-Wert zur Evaluierung des Modells und laden daher die Funktion `r2_score` von `sklearn`:

```
#%% packages
from DataPrep import X, y
import torch
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import r2_score
```

Listing 3.8 Unser erstes Modell – Paketimport (Quelle: 030_FirstModel_Regression\00_LinRegFromScratch.py)

PyTorch arbeitet nur mit Tensoren, sodass wir zunächst die numpy-Arrays mit `torch.from_numpy` in Tensoren umwandeln:

```
#%% convert to tensor
X_tensor = torch.from_numpy(X.astype(np.float32))
y_tensor = torch.from_numpy(y.astype(np.float32))  # Ensure y is float32
```

3.2.2 Modelltraining

Jetzt haben wir die Daten so weit in Form gebracht, dass wir loslegen können. Unser Regressionsmodell wird letztlich durch einen *Bias*-Parameter sowie einen Steigungsparameter (*Slope* oder *Weight*) beschrieben. Für jedes Feature gibt es einen Slope-Parameter und insgesamt einen Bias-Parameter.

Diese Terme `w` (für *weight*) und `b` (*bias*) werden zunächst initialisiert. Das können wir mit `torch.zeros` implementieren. Wichtig ist hierbei, dass der Parameter `requires_grad` auf `True` gesetzt werden. Nur so kann das automatische Rückwärtsrechnen (Backpropagation) und das Trainieren des Modells ermöglicht werden.

```
#%% training
# Initialize weights with smaller values to prevent exploding gradients
w = torch.zeros(X.shape[1], 1, requires_grad=True, dtype=torch.float32)
b = torch.zeros(1, requires_grad=True, dtype=torch.float32)
print(f"w shape: {w.shape}, b shape: {b.shape}")
w shape: torch.Size([30, 1]), b shape: torch.Size([1])
```

Der Trainingsprozess wird über einige Parameter beeinflusst. Die wichtigsten sind die Anzahl der *Epochen* sowie die *Learning Rate*.

Bevor wir das Training starten, schauen wir uns diese beiden wichtigen Parameter noch einmal genauer an. Beginnen wir mit den Epochen.

Epoche

Unser Trainingsdatensatz hat 11.000 Samples. Diese werden üblicherweise in kleineren Häppchen dem Modell übergeben. Diese Häppchen nennt man *Batches*, und zu dem Konzept kommen wir noch. Nachdem alle Samples einmal verwendet wurden, um die Gewichte des Modells anzupassen, ist die *Epoche* abgeschlossen.

Der Prozess wiederholt sich, sodass das Modell dieselben Daten viele Male »sieht«, um aus ihnen zu lernen.

Typischerweise trainiert man ein Modell über mehrere Epochen, wobei mit jeder Epoche die Muster in den Daten immer besser erfasst werden.

Nachdem Sie nun das Konzept der Epochen kennen, erkläre ich noch, was es mit der Learning Rate auf sich hat.

Learning Rate

Stellen Sie sich das Modelltraining wie die Suche nach dem tiefsten Punkt in einem unbekannten Tal vor. Einem Wanderer, der vom Berg herabsteigt, wurden die Augen verbunden und er muss sich langsam vorantasten. Er kann nun entscheiden, ob seine Schritte eher groß oder klein sein sollen. Die Schrittlänge entspricht der Learning Rate.

Bei großen Schrittlängen (hohen *Learning Rates*) deckt unser Wanderer schnell ein großes Gebiet ab. Er könnte aber auch schnell am tiefsten Punkt vorbeigehen und schon wieder den gegenüberliegenden Berg hinaufsteigen.

Umgekehrt könnte er sehr kleine Schritte (geringe Learning Rates) wählen, um sich sehr vorsichtig fortzubewegen. In diesem Fall ist die Wahrscheinlichkeit hoch, dass er genau den tiefsten Punkt findet, aber es könnte bis dahin relativ lange dauern. Es könnte aber auch sein, dass er in einem lokalen Minimum stecken bleibt.

Die Learning Rate definiert somit, wie zügig oder vorsichtig der tastende Wanderer das Tal des Fehlers erkundet, um den optimalen Punkt zu finden.

Jetzt kann es aber wirklich losgehen! Wir definieren diese beiden Parameter.

```
EPOCHS = 100
LEARNING_RATE = 0.01
```

Nun kommen wir zum eigentlichen Kern des Modelltrainings: zu der Trainingsschleife, die in Listing 3.9 implementiert wird.

Die Daten werden dem Modell 100-mal gezeigt. Das wird mit einer `for`-Schleife über die `EPOCHS` implementiert. Wie gut das Modell lernt, können Sie begutachten, indem Sie die Verluste studieren. Diese werden in jeder Epoche extrahiert und der Liste `loss_list` hinzugefügt.

Innerhalb der Schleife werden die immer gleichen Schritte durchlaufen:

1. Im *Forward-Pass* werden die Vorhersagen erstellt. Hierbei werden die unabhängigen Features mit den Modellgewichten multipliziert sowie die Aktivierungsfunktionen angewandt.
2. Diese Vorhersagen werden mit den richtigen Ergebnissen verglichen, und dabei wird der Verlust berechnet. Hierfür gibt es verschiedene *Verlustfunktionen*, wie Sie in Kapitel 2 gelernt haben. Für Regressionsmodelle ist der *Mean Squared Error*-Verlust (*MSE*-Loss) eine gute Wahl.
3. Nun können die Gradienten berechnet werden. Hierfür führen wir `loss.backward()` aus, und sämtliche Gradienten werden ermittelt.
4. Diese Gradienten werden jetzt genutzt, um die Modellgewichte zu aktualisieren. Dabei wird die *Lernrate* mit den Gradienten multipliziert und diese Korrektur vom bisherigen Modellgewicht abgezogen.
5. Bevor die nächste Epoche startet, müssen die Gradienten auf 0 zurückgesetzt werden, da sie sich sonst aufsummieren würden und das Ergebnis verfälscht werden würde.
6. Der Verlustwert der aktuellen Epoche wird der Gesamtliste aller Verluste hinzugefügt.

7. Zum Überprüfen des Modelltrainings wird jeweils die aktuelle Epoche sowie der aktuelle Verlust ausgegeben.

```
loss_list = []
for epoch in range(EPOCHS):
    # 1. Forward pass
    y_predict = torch.matmul(X_tensor, w) + b

    # 2. Calculate loss (MSE)
    loss = torch.nn.functional.mse_loss(y_predict, y_tensor)

    # 3. Backward pass
    loss.backward()

    # 4. Update weights and biases
    with torch.no_grad():
        w -= LEARNING_RATE * w.grad
        b -= LEARNING_RATE * b.grad
        # 5. Zero gradients after using them
        w.grad.zero_()
        b.grad.zero_()

    # 6. Store loss for plotting
    loss_list.append(loss.item())

    # 7. Print loss for this epoch
    print(f"Epoch {epoch}, Loss: {loss.item():.4f}")
```

```
Epoch 0, Loss: 19.9446
Epoch 1, Loss: 19.0938
...
Epoch 98, Loss: 1.5858
Epoch 99, Loss: 1.5732
```

Listing 3.9 Unser erstes Modell – Trainingsschleife (Quelle: 030_FirstModel_Regression\00_LinRegFromScratch.py)

Wir können hier bereits in der Ausgabe beobachten, wie das Training voranschreitet und die Verluste immer geringer werden.

3.2.3 Modell-Evaluierung

Das können wir jetzt aber auch noch einmal in einer Grafik visualisieren (siehe Abbildung 3.4). Der dazugehörige Code ist in Listing 3.10 dargestellt. Die Verluste, die in der

Liste `loss_list` gespeichert sind, werden über der Anzahl der Epochen `EPOCHS` als Liniendiagramm gezeigt. Hierfür nutzen wir `seaborn` mit der Funktion `sns.lineplot`:

```
#%% plot loss
sns.lineplot(x=range(EPOCHS), y=loss_list)
plt.title('Loss over Epochs')
plt.xlabel('Epoch [-]')
plt.ylabel('Loss [-]')
```

Listing 3.10 Unser erstes Modell – Visualisierung der Verluste (Quelle: 030_FirstModel_Regression\00_LinRegFromScratch.py)

Abbildung 3.4 zeigt das Ergebnis des Modelltrainings.

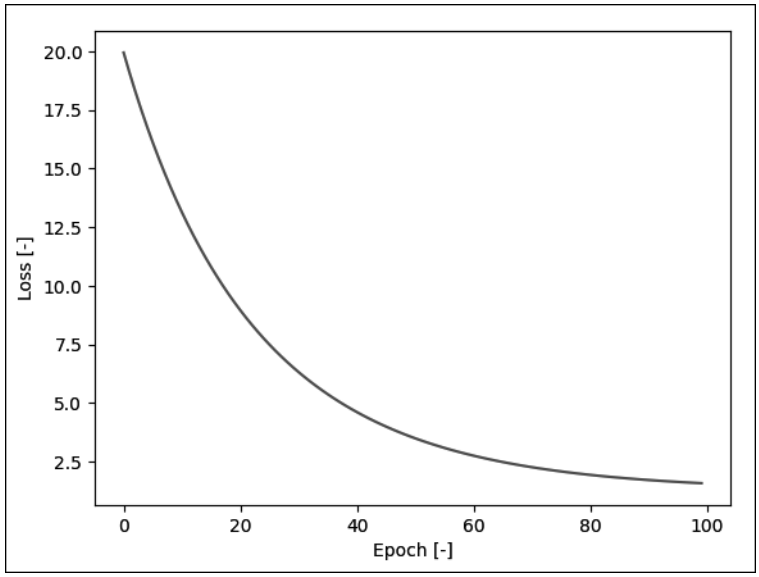


Abbildung 3.4 Unser erstes Modell – Verluste und Epochen

Der Verlust sinkt kontinuierlich mit jeder weiteren Epoche. Man kann hier aber auch schon erkennen, dass das Modell zwar immer weniger Verluste aufweist, dass sich die Verluste aber asymptotisch einer Grenze annähern. Der Frage nach der optimalen Trainingsdauer werden wir uns an späterer Stelle noch einmal zuwenden.

Selten sieht das Bild so »sauber« aus wie hier. Üblicherweise gibt es mehr Fluktuation, also auch Epochen, in denen die Verluste kurzzeitig wieder leicht steigen, bevor sie dem längerfristigen Trend der fallenden Verluste folgen.

Schauen wir uns als Nächstes die Modellgewichte (die Steigungswerte `w` und den Offsetwert `b`) näher an:

```
#%% check results
print(f"Weights: {w.detach().numpy().flatten()}, Bias: {b.item()}")
```

```
Weights: [-0.09916524 -0.5292779  -0.16298231  0.34272027  0.06862636
...
Bias: 3.408252716064453
```

3.2.4 Modell-Inferenz

Letztlich könnten wir in diesem einfachen Fall diese Werte nutzen und die Berechnung auf Basis der Formel zur Berechnung der Regression durchführen.

Wir erreichen das, indem wir die unabhängigen Features mit den Modellgewichten multiplizieren. Wichtig ist hierbei, dass die Berechnung innerhalb des Scopes von `torch.no_grad()` durchgeführt wird. Damit wird verhindert, dass Gradienten berechnet werden. Wir befinden uns hier in der *Modell-Inferenz*, also beim Testen des Modells – im Gegensatz zum Modelltraining. Während der Modell-Inferenz möchten wir keine Operationen ausführen, die das Netzwerk beeinflussen könnten, aber nicht Teil des Trainingsprozesses sein sollen. Ein positiver Nebeneffekt ist, dass hiermit auch Ressourcen wie Speicher und Rechenzeit geschont werden können. Es wird sichergestellt, dass bestimmte Operationen nicht fälschlicherweise in die Gradientenberechnung einfließen.

```
# %%
with torch.no_grad():
    y_pred = (torch.matmul(X_tensor, w) + b).detach().numpy().flatten()
```

Die Vorhersage `y_pred` haben wir berechnet und sie zusammen mit den tatsächlichen Werten `y` dargestellt. Listing 3.11 zeigt den entsprechenden Code. Wir nutzen die Funktion `sns.regplot()`, um ein Streuungsdiagramm mit überlagerter Regressionsgerade zu erstellen. Die Datenpunkte sind blau dargestellt mit einem Transparenzwert von 0.1. Letzterer sorgt dafür, dass in Bereichen, in denen viele Werte übereinanderliegen, die Punkte in einem intensiveren Blau dargestellt sind, und dass in Bereichen mit sehr wenigen Punkten die Punkte eher in einem schwachen Blau zu sehen sind. Zusätzlich ist die Regressionsgerade als rote Linie zu sehen.

```
# %% visualise correlation
sns.regplot(x=y_pred, y=y, color='red',
            scatter_kws={'s': 10,
                         'color': 'blue',
                         'alpha': 0.1})
plt.title('Predicted Anxiety Level vs Actual Anxiety Level')
plt.xlabel('Predicted Anxiety Level [-]')
plt.ylabel('Actual Anxiety Level [-]')
```

Listing 3.11 Unser erstes Modell – Visualisierung der Korrelationen (Quelle: 030_FirstModel_Regression\00_LinRegFromScratch.py)

Das Ergebnis ist ein Korrelationsdiagramm, das Sie in Abbildung 3.5 sehen. Hierbei ist das tatsächliche Angst-Level über dem vorhergesagten Angst-Level geplottet.

Die Korrelation ist positiv und im Mittel wird ein Angst-Level von 5 auch als 5 vorhergesagt. Aber natürlich gibt es Streuung in den Daten, sodass in manchen Fällen dann auch Werte zwischen 1 und 7 vorhergesagt werden.

Mit dieser Darstellung können Sie sich einen guten Überblick darüber verschaffen, in welchen Bereichen das Modell gut funktioniert und in welchen vielleicht noch Verbesserungen notwendig wären.

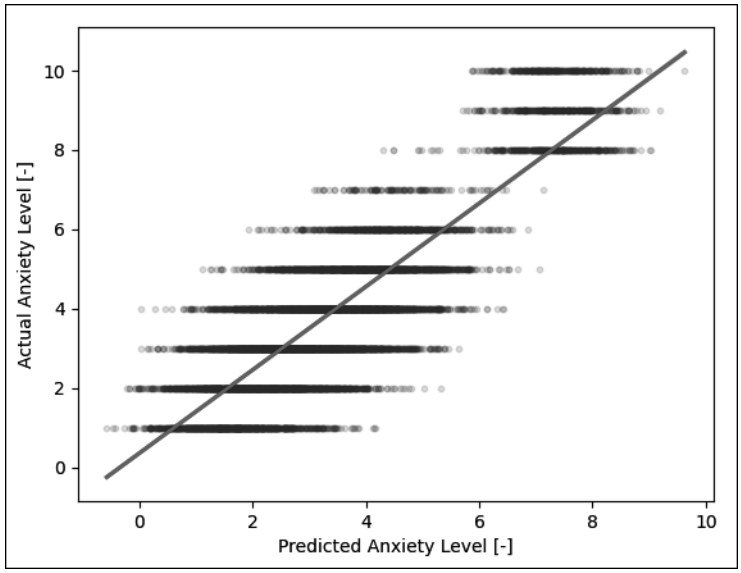


Abbildung 3.5 Unser erstes Modell – vorhergesagtes und tatsächliches Angst-Level

Häufig will man aber verschiedene Modelle miteinander vergleichen. Das geht einfacher, wenn man die Modellgüte auf einen einzigen Zahlenwert verdichtet.

Im Bereich der Regressionsmodelle ist der R^2-Wert ein häufig genutztes Maß.

Der R^2-Wert

Der R^2-Wert wird auch *Bestimmheitsmaß* oder *Determinationskoeffizient* genannt. Er ist eine statistische Kennzahl, die angibt, wie gut die unabhängigen Features in einem Regressionsmodell die Varianz der abhängigen Variablen erklären.

Der Wertebereich liegt in der Regel immer zwischen 0 und 1 bzw. zwischen 0 % und 100 %. Die Extremwerte sind dabei wie folgt zu verstehen:

- R^2=0: Das trainierte Modell kann die abhängige Variable überhaupt nicht erklären. Es gibt keinen *linearen* Zusammenhang zwischen den unabhängigen Variablen

und der abhängigen Variable. Wichtig ist hierbei das Wort *linearer* Zusammenhang. Es kann durchaus sein, dass ein Zusammenhang zwischen den Daten besteht, dieser aber einfach nicht-linear ist.

- R^2=1: Das Modell ist perfekt in der Lage, die gesamte Variabilität der abhängigen Variable zu erklären. Das ist in der Praxis so gut wie nie der Fall, da es immer Messungenauigkeiten oder zufällige Fehler gibt – oder weitere unabhängige Variablen, die nicht im Modell berücksichtigt wurden.

Im Gegenteil ist es so, dass Sie einem sehr hohen Wert mit Vorsicht begegnen sollten, da er auf ein Overfitting des Modells hinweisen kann.

Wenn der R^2-Wert beispielsweise bei 0.75 liegt, kann das so gedeutet werden, dass 75 % der Streuung der abhängigen Variable durch die im Modell enthaltenen unabhängigen Variablen erklärt werden kann. Die restlichen 25 % der Streuung sind auf andere, nicht im Modell berücksichtigte Faktoren oder zufällige Fehler zurückzuführen.

Allgemein gilt, dass ein höherer R^2-Wert eine bessere Anpassung des Modells an die Daten widerspiegelt.

Extrem wichtig ist an dieser Stelle, dass ein hoher R^2-Wert nicht zwangsläufig bedeutet, dass es einen *kausalen Zusammenhang* gibt. Es muss demnach nicht automatisch gelten, dass die unabhängigen Variablen kausal die abhängige Variable beeinflussen. Es zeigt lediglich einen statistischen Zusammenhang.

Die Berechnung erfolgt mit der Funktion `r2_score` von `sklearn`. Wir übergeben der Funktion die echten sowie die vorhergesagten Werte und erhalten einen einzigen Zahlenwert:

```
#%% Calculate R-squared
r2 = r2_score(y_true=y,
              y_pred=y_pred)
print(f"R-squared: {r2:.2f}")
```

```
R-squared: 0.65
```

Unser erstes Modell erreicht einen R^2-Wert von 0.65. Diesen Wert könnten wir nun als Vergleichsmaß heranziehen, um dieses Modell mit anderen Modellen zu vergleichen.

Ob ein R^2-Wert als gut oder schlecht einzuschätzen ist, hängt stark von seinem Kontext ab. In bestimmten Fällen ist ein R^2-Wert von 0.98 als schlecht zu erachten und in anderen Fällen ein R^2-Wert von 0.4 bereits als sehr gut.

Zunächst sind wir aber mit dem Ergebnis zufrieden und wollen unser Modelltraining weiter verbessern, indem wir Fähigkeiten von PyTorch nutzen, die unseren Code modularer und damit vielseitiger einsetzbar machen.

Im nächsten Abschnitt werden Sie lernen, wie Sie das Modell in einer eigenen Klasse definieren können und wie der Optimierer in das Training eingebettet werden kann.

3.3 Modellklasse und Optimierer

Das Modell in einer eigenen Klasse zu definieren, macht es sehr viel einfacher, den Code später anzupassen. Für die spätere Nutzung des Modells ist es auch wichtig, dass es abgespeichert werden und zu einem späteren Zeitpunkt sehr einfach geladen werden kann. Um diese Vorteile nutzen zu können, hilft es, das Modell in einer separaten Klasse zu speichern.

Zunächst importieren wir wieder die erforderlichen Pakete so, wie in Listing 3.12 gezeigt:

```
#%% packages
from DataPrep import X, y
import torch
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import r2_score
```

Listing 3.12 Modellklasse – Paketimport (Quelle: 030_FirstModel_Regression\10_ModelClass.py)

Es gilt auch als guter Stil, die konstanten Trainingsparameter, die sogenannten *Hyperparameter*, gebündelt am Anfang des Skripts zu definieren. Nachdem die Pakete geladen wurden, definieren Sie direkt die Hyperparameter.

Hyperparameter

Hyperparameter sind Konstanten, also Variablen, die während des Programmablaufs nicht verändert werden. Sie werden vor dem Trainingsprozess festgelegt und dazu genutzt, diesen zu beeinflussen. Sie haben großen Einfluss auf das Modelltraining und werden vom Modellentwickler (uns) manuell, auf Basis bestimmter Optimierungsverfahren oder Best Practices (Erfahrungswerten) festgelegt.

Die Hyperparameter steuern das Verhalten des Lernprozesses und haben einen enormen Einfluss auf die spätere Leistungsfähigkeit und Performance des Modells.

Die am häufigsten verwendeten Hyperparameter sind die Lernrate (Learning Rate), die Batch-Größe (Batch Size) sowie die Anzahl der Epochen.

Die Parameter hängen stark von den genutzten Daten und dem gewählten Modell ab. Daher ist die Hyperparameter-Optimierung ein wichtiger Schritt während des Modelltrainings.

```
#%% Hyperparameters
EPOCHS = 100
LEARNING_RATE = 0.1
```

Die Daten müssen in Tensoren umgewandelt werden, da alle Objekte während des Trainings als Tensoren vorliegen müssen. Hierbei ist es ebenfalls wichtig, den Datentyp gegebenenfalls anzupassen. Üblicherweise werden wir hier `float32` nutzen:

```
#%% convert to tensor
X_tensor = torch.from_numpy(X)
y_tensor = torch.from_numpy(y.astype(np.float32))
```

An dieser Stelle kommen wir zur *Modellklasse*. Das ist eine von uns definierte Klasse, die von der Klasse `torch.nn.Module` abstammt.

Die Modellklasse stellt zusammen mit dem Optimierer den Bauplan für ein Modell dar. Auf dieser Basis erstellen wir später eine *Modellinstanz*, die dann während des Trainings genutzt wird.

Die Modellklasse muss zwei Methoden aufweisen: `__init__()` und `forward()`:

- Die `__init__`-Funktion wird während der Erstellung einer Modellinstanz einmalig aufgerufen. Idealerweise wollen wir eine Modellklasse erstellen, die flexibel nutzbar ist und sich somit an den Datensatz bzw. weitere Parameter anpassen kann. Aus diesem Grund übergeben wir hier der `__init__()`-Methode die Parameter `input_size` und `output_size`, die es uns ermöglichen, das Modell an die Anzahl der unabhängigen Features des Datensatzes sowie an die Anzahl der vorherzusagenden Features anzupassen. Üblicherweise werden hier auch die Netzwerkschichten erstellt, die anschließend verwendet werden sollen. In unserem Beispiel verwenden wir eine lineare Schicht, die direkt die Eingabefeatures mit der nächsten Schicht – hier der Ausgabeschicht – verbindet.
- In der `forward()`-Methode wird beschrieben, wie das Netzwerk aufgebaut ist und wie die Daten durch das Netzwerk fließen. Neben dem klassenspezifischen Parameter `self` erhält diese Funktion auch den Parameter `x`. Manchmal wird dieser Parameter auch `input` oder `data` genannt. Hierbei steht `x` für das, was ins Modell hineingegeben wird. Das sind die Daten, die das Modell verarbeiten wird. Im Folgenden wird dann innerhalb der Funktion beschrieben, wie diese Daten weiterverarbeitet werden, also wie sie von Schicht zu Schicht weitergereicht werden. In diesem einfachen Beispiel nutzen wir nur eine lineare Schicht mit `self.linear(x)`. Das Ergebnis überschreibt dann den Wert von `x` und wird als Ergebnis der Funktion zurückgegeben.

```
#%% Model class
class LinearRegression(torch.nn.Module):
    def __init__(self, input_size, output_size):
        super(LinearRegression, self ).__init__()
        self.linear = torch.nn.Linear(input_size, output_size)
```

```
    def forward(self, x):
        x = self.linear(x)
        return x
```

Listing 3.13 Modellklasse – Definition (Quelle: 030_FirstModel_Regression\10_ModelClass.py)

Nachdem die Modellklasse erstellt wurde, können wir nun eine Instanz der Klasse erstellen. Hierfür übergeben wir die Dimensionen, die sich aus unserem Datensatz ergeben. Die Anzahl der unabhängigen Features lässt sich über die Anzahl der Spalten des Datensatzes mit `X.shape[1]` ermitteln. Da nur eine Zielgröße vorhergesagt werden soll, beträgt die `output_size` 1:

```
#%% Model instance
model = LinearRegression(input_size=X.shape[1],
                         output_size=1)
```

Neben der Modellinstanz benötigen wir als weitere wichtige Elemente des Netzwerktrainings noch den Optimierer und die Verlustfunktion.

Als *Optimierer* wird an dieser Stelle `Adam` verwendet. Die gängigste *Verlustfunktion* bei Regressionsaufgaben ist der *Mean Squared Error*-(MSE-)Verlust, und sie wird über `torch.nn.MSELoss()` aufgerufen:

```
#%% Optimizer and Loss Function
optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)
loss_fn = torch.nn.MSELoss()
```

Jetzt haben wir alles vorbereitet und können das Training beginnen. Listing 3.14 zeigt, wie wir zunächst die Verluste `loss_list` als leere Liste initialisieren. Während der Iteration über die Epochen durchlaufen wir dieselben Schritte wie zuvor:

1. Im Forward Pass werden die Vorhersagen `y_predict` erstellt.
2. Die Verluste `loss` werden auf Basis der Vorhersagen `y_predict` und der tatsächlichen Werte `y_tensor` ermittelt. Wichtig ist hierbei immer, dass die Dimensionen der beiden Objekte exakt gleich sind.
3. Im Backward Pass können mittels `loss.backward()` alle Gradienten bestimmt werden.
4. Letztlich werden die Modellgewichte über `optimizer.step()` aktualisiert.
5. Um zu vermeiden, dass die Gradientenberechnung verfälscht wird, müssen sie auf null zurückgesetzt werden. Das geschieht mit `optimizer.zero_grad()`.
6. (optional) Zur späteren Auswertung, wie das Modelltraining sich kontinuierlich verbessert, werden die Verluste der aktuellen Epoche `loss.item()` zur Gesamtliste

der Verluste hinzugefügt. Um den tatsächlich berechneten Wert zwischenzuspeichern, wird die Methode `item()` aufgerufen. Diese liefert keinen Tensor, sondern einen numerischen Wert zurück.

```
loss_list = []
for epoch in range(EPOCHS):
    # 1. Forward pass
    y_predict = model(X_tensor)

    # 2. Calculate loss (MSE)
    loss = loss_fn(y_predict.squeeze(), y_tensor)

    # 3. Backward pass
    loss.backward()

    # 4. Update weights and biases
    optimizer.step()

    # 5. zero gradients
    optimizer.zero_grad()

    # 6. Store loss for plotting
    loss_list.append(loss.item())

    # 7. Print loss for this epoch
    print(f"Epoch {epoch}, Loss: {loss.item():.4f}")
```

Listing 3.14 Modellklasse – Trainingsschleife (Quelle: 030_FirstModel_Regression\10_ModelClass.py)

Nachdem das Modell trainiert wurde, können wir uns über die Verluste ein Bild davon machen, wie gut das Modelltraining ablief. Hierzu werden so, wie in Listing 3.15 dargestellt, die Verluste über den Epochen geplottet:

```
#%% plot loss
sns.lineplot(x=range(EPOCHS), y=loss_list)
plt.title('Loss over Epochs')
plt.xlabel('Epoch [-]')
plt.ylabel('Loss [-]')
```

Listing 3.15 Modellklasse – Verluste visualisieren (Quelle: 030_FirstModel_Regression\10_ModelClass.py)

Das Ergebnis des Trainings ist in Abbildung 3.6 zu sehen. Die Verluste nehmen zunächst mit zunehmenden Epochen stark ab, um dann ab circa Epoche 40 abzuflachen. Das Modell kann immer noch dazulernen, aber es profitiert immer weniger von weiteren Durchläufen, was an dem Abflachen der Verluste zu erkennen ist.

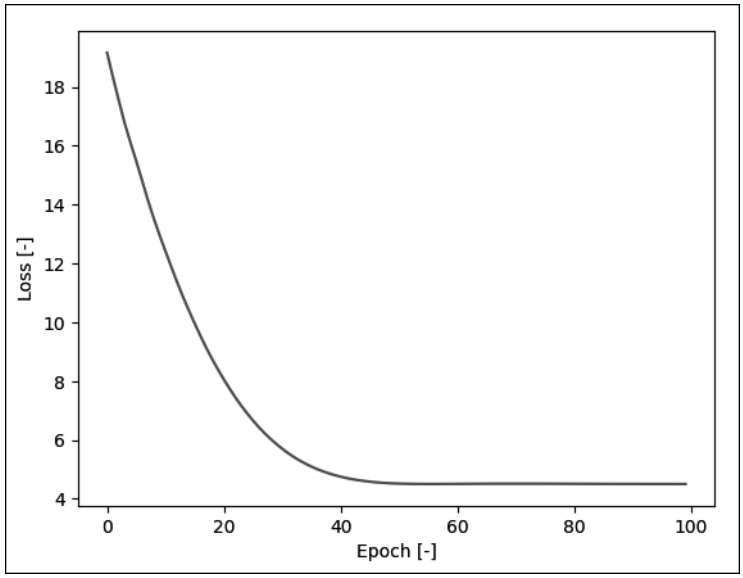


Abbildung 3.6 Modellklasse und Optimierer – Entwicklung der Verluste

Nachdem wir nun die Modell-Klasse und den Optimierer eingeführt haben, können wir unser System noch weiter verbessern. Eine Verbesserung betrifft das Übergeben der Trainingsdaten an das Modell während des Trainings. Wenn die Daten nicht komplett, sondern in kleineren Häppchen übergeben werden, nennt man diese *Batches*, und damit beschäftigen wir uns ausführlich im folgenden Abschnitt.

3.4 Batches

Batches sind ein integraler Bestandteil jedes Trainings und können auch die Performance des Trainings beeinflussen.

Sie sind außerdem auch Hyperparameter für das Training. Da auch sie das Ergebnis beeinflussen können, werden oftmals verschiedene Batchgrößen und deren Einfluss untersucht.

Ich werde im Folgenden erklären, was Batches sind, welche Vorteile sie bieten und welche Batchgrößen man verwenden sollte. In Abschnitt 3.4.3 werden Sie an einem praktischen Beispiel sehen, wie man Batches implementiert.

3.4.1 Was sind Batches?

Zunächst sollten wir klären, was Batches genau sind. Das Konzept wird grundlegend in Abbildung 3.7 verdeutlicht.

Bisher haben wir in jeder Epoche den gesamten Datensatz dem Modell übergeben. Beim Training mit Batches wird hingegen der gesamte Datensatz in kleinere »Häppchen« unterteilt und diese einzelnen Häppchen werden dem Modell übergeben.

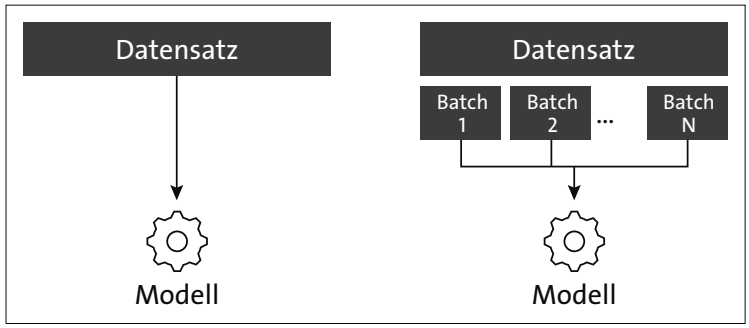


Abbildung 3.7 Batches: links – Situation ohne Batches, rechts – Aufteilung des Datensatzes in einzelne Batches

Schauen wir uns im nächsten Abschnitt an, warum man Batches verwendet.

3.4.2 Vorteile von Batches

Warum wird das gemacht? Dafür gibt es einige Gründe, die ich im Kasten näher beleuchte.

Vorteile durch Batches

Die Implementierung von Batches macht die Trainingsschleife etwas komplizierter, bringt aber etliche Vorteile mit sich:

- **Speicher:** Oftmals gibt es technische Gründe, die die Nutzung von Batches erfordern. So ist es vielleicht aufgrund der Größe des Datensatzes nicht mehr möglich, den gesamten Datensatz in einem Rutsch zu verarbeiten, weil er nicht komplett in den Arbeits- oder Grafikkartenspeicher passt. Das ist zum Beispiel häufig im Bereich von großen Bild- oder Video-Datensätzen der Fall.
- **Parallelisierung:** Das Training kann beschleunigt werden, indem Batches verwendet und diese dann parallel auf moderner Hardware wie GPUs ausgeführt werden.
- **Verbessertes Lernen:** Beim Training kommt es darauf an, dass das Modell lernt, zu generalisieren. Das heißt, dass es nicht nur die »bekannten« Trainingsdaten gut vorhersagen kann, sondern auch für gänzlich unbekannte Daten gute Vorhersagen trifft. Hierbei können kleinere und mittlere Batchgrößen helfen, sodass das Modell

weniger schnell überangepasst (*overfitted*) reagiert. Die stärkeren Schwankungen in den Gradienten können das Modell aus lokalen Minima herausholen und so zu einer besseren Generalisierungsfähigkeit auf unbekannten Daten führen.

Gehen wir davon aus, dass wir Batches nutzen wollen. Die sich anschließende Frage lautet: Welche Batchgröße sollten wir verwenden?

3.4.3 Die optimale Batchgröße

Es gibt keine universelle beste Batchgröße, da sie stark von Parametern wie dem verwendeten Datensatz, dem Modell oder den zur Verfügung stehenden Hardwareressourcen abhängt.

Dennoch gibt es einige Best-Practice Empfehlungen:

- Üblicherweise werden Batchgröße als Potenz von 2 verwendet, also z. B. 16, 32, 64 usw.
- Als Faustregel kann gelten, dass man mit moderaten Batchgrößen wie 32 oder 64 startet und sie schrittweise erhöht, wenn es die Rechenressourcen zulassen und dadurch die Modellperformance verbessert werden kann.
- Wenn das Modell bereits komplexer ist, kann das kleinere Batchgrößen erfordern.
- Es gibt Zusammenhänge zwischen Batchgröße und Lernrate: Größere Batchgrößen können von höheren Lernraten profitieren. Wichtig kann es sein, beide Parameter zu optimieren.
- Im Bereich Computer-Vision, wie für das Klassifizieren von Bildern, werden häufig Batchgrößen zwischen 32 und 512 verwendet.
- Bei der Sprachverarbeitung werden Batchgrößen von 8 bis 256 verwendet. Gerade bei den häufig genutzten Transformer-Modellen werden eher kleinere Batches bevorzugt, da die Modelle selbst einen hohen Speicherbedarf aufweisen.
- Am Ende muss man meist mehrere Batchgrößen testen, um die für den jeweiligen Fall optimale Batchgröße zu ermitteln.

Im nächsten Abschnitt sehen wir uns an, wie Batches praktisch in das Training eingebettet werden können.

3.4.4 Coding: Implementierung von Batches

Schauen wir uns in der Praxis an, wie das implementiert werden kann. In Listing 3.16 werden die Pakete sowie die vorbereiteten Daten `X`, `y` aus unserem früheren Skript `DataPrep` geladen und danach in die Tensoren `X_tensor` und `y_tensor` umgewandelt:

```
#%% packages
import numpy as np
from sklearn.preprocessing import StandardScaler
import seaborn as sns
import matplotlib.pyplot as plt
import kagglehub
import torch
from sklearn.metrics import r2_score
from DataPrep import X, y

#%% convert to tensor
X_tensor = torch.from_numpy(X.astype(np.float32))
y_tensor = torch.from_numpy(y.astype(np.float32))  # Ensure y is float32
```

Listing 3.16 Nutzung von Batches – Datenvorbereitung (Quelle: 030_FirstModel_Regression\20_Batches.py)

Das Training wird über mehrere Hyperparameter beeinflusst. Hierzu gehören `EPOCHS`, `LEARNING_RATE` sowie die neu hinzugekommene `BATCH_SIZE`:

```
#%% Hyperparameters
EPOCHS = 100
LEARNING_RATE = 0.01
BATCH_SIZE = 512
```

Im folgenden Listing 3.17 werden die Modellklasse, der Optimierer sowie die Verlustfunktion definiert. Hier gibt es keine Anpassungen im Vergleich zu früheren Skripten:

```
#%% Model class
class LinearRegression(torch.nn.Module):
    def __init__(self, input_size, output_size):
        super(LinearRegression, self).__init__()
        self.linear = torch.nn.Linear(input_size, output_size)

    def forward(self, x):
        x = self.linear(x)
        return x

#%% Model instance
model = LinearRegression(X.shape[1], 1)

#%% Loss function
criterion = torch.nn.MSELoss()
```

```
#%% Optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)
```

Listing 3.17 Nutzung von Batches – Modellklasse, Verlustfunktion und Optimierer (Quelle: 030_FirstModel_Regression\20_Batches.py)

In Listing 3.18 ist die Trainingsschleife mit den erforderlichen Anpassungen zur Verarbeitung von Batches dargestellt. Das Entscheidende ist, dass in der Epochenschleife nun eine zusätzliche Schleife für die Verarbeitung der einzelnen Batches zu finden ist (1). In gleichbleibenden Schritten von der Größe der `BATCH_SIZE` wird durch den Datensatz iteriert.

Die weitere Anpassung (2) betrifft die Daten. Wir müssen für die jeweiligen Batchpositionen die entsprechenden unabhängigen Features `X_batch` und abhängigen Features `y_batch` extrahieren.

Ab hier folgt das Training dem gewohnten Muster.

```
loss_list = []
for epoch in range(EPOCHS):
    epoch_loss = 0
    for i in range(0, len(X_tensor), BATCH_SIZE):   # (1)
        # get batch (2)
        X_batch = X_tensor[i:i+BATCH_SIZE]
        y_batch = y_tensor[i:i+BATCH_SIZE].unsqueeze(1)

        # forward pass
        y_predict = model(X_batch)

        # calculate loss
        loss = criterion(y_predict, y_batch)

        # backward pass
        loss.backward()

        # update weights and biases
        optimizer.step()

        # zero gradients
        optimizer.zero_grad()

        # Store loss for plotting
        epoch_loss += loss.item()
```

```
    # Print loss for this epoch
    print(f"Epoch {epoch}, Loss: {epoch_loss/len(X_tensor):.4f}")
    loss_list.append(epoch_loss)
```

Listing 3.18 Nutzung von Batches – Trainingsschleife (Quelle: 030_FirstModel_Regression\20_Batches.py)

Wir haben damit erfolgreich das Konzept der Batches in das Training integriert. Aber es gibt noch Verbesserungspotenzial. So greifen wir innerhalb der Trainingsschleife direkt auf die Daten zu und müssen uns »händisch« um das Iterieren in Batches kümmern.

An dieser Stelle greifen wir auf weitere Hilfsfunktionen zurück: auf `Dataset` und `DataLoader`. Diesen Konzepten werden wir uns im nächsten Abschnitt widmen.

3.5 Dataset und DataLoader

In diesem Abschnitt lernen Sie weitere Hilfsfunktionen zum Training kennen: `Dataset` und `DataLoader`. Lassen Sie mich zunächst erklären, was das genau ist. Dann schauen wir uns an, welche Vorteile mit diesen Hilfsfunktionen verbunden sind, und betrachten anschließend, wie diese Konzepte implementiert werden.

3.5.1 Was sind Dataset und DataLoader?

`Dataset` und `DataLoader` sind PyTorch-Klassen, die als Abstraktionen der Daten genutzt werden.

Mithilfe `Dataset` erhalten Sie eine Schnittstelle, um auf einzelne Daten des Datensatzes zuzugreifen. Damit erreichen Sie, dass die Logik zum Laden, Preprocessing (Vorverarbeiten) und Abrufen der Daten gekapselt wird.

Damit ist es möglich, die Logik des Datenzugriffs von dem Modelltraining zu trennen. Und durch diese Trennung der Logik werden unsere Skripte deutlich flexibler und lassen sich einfacher an neue Daten anpassen.

3.5.2 Die Vorteile von Dataset und DataLoader

Wie schon erwähnt, sind der konsistente Zugriff auf die Daten und die Trennung der Daten von der Modelllogik als Hauptvorteile dieses Konzeptes anzusehen.

Darüber hinaus ist es hiermit möglich, spezifische Datenstrukturen leicht anzupassen. Zur Anpassung können auch Transformationen der Daten zählen wie die *Data Augmentation*. Darunter versteht man, die Anzahl und Vielfalt der Trainingsdaten künstlich zu erhöhen. Wir kommen in Kapitel 5, »Computer-Vision«, auf dieses Thema zurück.

Werfen wir nun einen Blick darauf, wie diese Klassen implementiert werden.

3.5.3 Coding: Implementierung mit Dataset und DataLoader

Wir erweitern das bisher erstellte Skript. Die Klassen `Dataset` und `DataLoader` werden über `torch` bereitgestellt. Die weiteren Pakete sind bekannt und werden in Listing 3.19 aufgeführt:

```
#%% packages
import numpy as np
import pandas as pd
import os
import torch
from torch.utils.data import Dataset, DataLoader
from sklearn.metrics import r2_score
from DataPrep import X, y
import seaborn as sns

#%% Hyperparameters
EPOCHS = 50
LEARNING_RATE = 0.1
BATCH_SIZE = 512
```

Listing 3.19 Dataset und DataLoader – Datenvorbereitung (Quelle: 030_FirstModel_Regression\30_DatasetDataLoaders.py)

Wir implementieren in Listing 3.20 zuerst die eigene Dataset-Klasse `AnxietyDataset`. Diese erbt von der Klasse `Dataset`. In dieser Klasse müssen drei Methoden definiert werden:

- **`__init__()`**: Dieser Konstruktor der Klasse wird aufgerufen, wenn eine neue Instanz der Klasse erstellt wird. Man nutzt sie, um Dateipfade und Metadaten zu laden, die für einen späteren Zugriff benötigt werden. Hier werden auch Transformationen auf die Daten angewandt.
- **`__len__()`**: Mit dieser Methode wird die Gesamtanzahl an Daten ermittelt. Der `DataLoader` nutzt diese Methode, um die Anzahl der Datenpunkte im `Dataset` zu ermitteln.
- **`__getitem__()`**: Diese Methode ist die wichtigste der drei Methoden. Sie wird aufgerufen, um einzelne Datenpunkte anhand eines Indexes abzurufen.

Sehen Sie sich das praktische Beispiel aus Listing 3.20 an. In der `__init__`-Methode werden die unabhängigen Features `X` und das abhängige Feature `y` übergeben und die entsprechenden Properties der Klasse erstellt. Auf diese kann dann über `self.X` und `self.y` zugegriffen werden. An dieser Stelle wird der Datentyp von `y` mittels `astype()` in `float` umgewandelt.

Die __len__-Methode wird genutzt, um über die len-Funktion die Anzahl der Datensätze zu ermitteln.

Die __getitem__-Funktion benötigt neben dem klasseneigenen Objekt self einen Index idx, der genutzt wird, um auf einzelne Datenpunkte zuzugreifen. Diese Methode gibt das unabhängige und abhängige Feature an der Position des Indexes zurück.

```
#%% Dataset class
class AnxietyDataset(Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y.astype(np.float32)

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```

Listing 3.20 Dataset und DataLoader – Dataset-Klasse (Quelle: 030_FirstModel_Regression\30_DatasetDataLoaders.py)

Dataset und DataLoader kommen immer paarweise vor. DataLoader ist hierbei der Partner des Dataset, der dafür sorgt, dass das Modell immer optimal mit fertigen Daten versorgt wird.

Der DataLoader kümmert sich um das Batching. Darüber hinaus kann man leicht die Daten vor jeder Trainings-Epoche mischen. Dadurch kann man verhindern, dass das Modell die Daten immer in der gleichen Reihenfolge sieht und sich somit an bestimmte Muster anpasst, die vielleicht nur durch die Reihenfolge der Daten entstehen. Hierdurch kann das Modell robuster werden und eine bessere Generalisierbarkeit erreichen.

Außerdem kann der DataLoader so konfiguriert werden, dass die Daten im Hintergrund geladen werden. Damit kann sichergestellt werden, dass die schnelle GPU immer ausgelastet ist. Das Laden und Preprocessing der Daten (z. B. Augmentierung, Normalisierung, Batch-Zusammenstellung) findet meist auf der langsameren CPU statt und ist damit oft langsamer als die eigentliche Modellberechnung. Auf diese Weise kann die CPU zum Flaschenhals werden.

Viele dieser Schritte mussten wir bisher manuell in den Trainingscode integrieren und können sie nun einfach an den DataLoader delegieren.

Die eigentliche Erstellung der Modellinstanzen unseres Datensatzes ist mit wenig Aufwand erledigt.

Die Datensatzinstanz `dataset` wird mit unserer zuvor definierten Klasse `AnxietyDataset` erstellt, wie Sie im Folgenden sehen.

Die `dataloader`-Instanz erhalten Sie mithilfe der Klasse `DataLoader`, die als wichtigsten Parameter `dataset` erhält. Weitere wichtige Parameter sind die Batchgröße `BATCH_SIZE` sowie der Parameter `shuffle`, der ein Durchmischen der Daten ermöglicht:

```
#%% DataLoader
dataset = AnxietyDataset(X, y)
dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)
```

Listing 3.21 zeigt die Modellklasse, die Verlustfunktion und den Optimierer. Hier gibt es keine Überraschungen oder Anpassungen gegenüber früheren Implementierungen:

```
#%% Model class
class LinearRegression(torch.nn.Module):
    def __init__(self, input_size, output_size):
        super(LinearRegression, self).__init__()
        self.linear = torch.nn.Linear(input_size, output_size)

    def forward(self, x):
        x = self.linear(x)
        return x

#%% Model instance
model = LinearRegression(X.shape[1], 1)

#%% Loss function
loss_fun = torch.nn.MSELoss()

#%% Optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)
```

Listing 3.21 Dataset und DataLoader – Modellklasse, Verlustfunktion und Optimierer (Quelle: 030_FirstModel_Regression\30_DatasetDataLoaders.py)

In Listing 3.22 sehen Sie die Trainingsschleife und einen der großen Vorteile der Nutzung des Ansatzes mit `Dataset` und `DataLoader`: Es gibt nun keinen direkten Zugriff mehr auf den ursprünglichen Datensatz. Stattdessen werden `X_batch` und `y_batch` direkt aus dem `dataloader` extrahiert.

```
#%%
loss_list = []
for epoch in range(EPOCHS):
    epoch_loss = 0
    for i, (X_batch, y_batch) in enumerate(dataloader):
        # forward pass
        y_predict = model(X_batch)

        # calculate loss
        loss = loss_fun(y_predict, y_batch.reshape(-1, 1))

        # backward pass
        loss.backward()

        # update weights and biases
        optimizer.step()

        # zero gradients
        optimizer.zero_grad()

        # Store loss for plotting
        epoch_loss += loss.item()

    # Print loss for this epoch
    print(f"Epoch {epoch}, Loss: {epoch_loss}")
    loss_list.append(epoch_loss)
```

Listing 3.22 Dataset und DataLoader – Modelltraining (Quelle: 030_FirstModel_Regression\30_DatasetDataLoaders.py)

Dieser Ansatz ist dahingehend vorteilhaft, dass nun bei Anpassungen am Datensatz keinerlei Änderungen in der Trainingsschleife erforderlich sind. Der hier vorliegende Code mit der äußeren Schleife für die Epochen, der inneren Schleife für die Batches sowie den einzelnen Elementen für das Modelltraining wird uns in dieser Form sehr oft wiederbegegnen.

Damit haben Sie die verschiedenen Elemente des Modelltrainings kennengelernt. Was uns noch fehlt, ist die Fähigkeit, Modelle zu speichern und zu laden.

Wir wollen nicht jedes Mal ein Modell neu trainieren müssen, bevor wir es nutzen können. Stattdessen wollen wir trainierte Netzwerke mit wenig Aufwand laden und zum Einsatz bringen.

3.6 Modelle speichern und laden

Um das trainierte Modell zu speichern, müssen wir uns zunächst mit dem Prozess des Speicherns vertraut machen. Es werden nämlich nur die Modellgewichte gespeichert.

Abbildung 3.8 verdeutlicht den Prozess anhand einer Analogie. Stellen Sie sich vor, dass Sie ein Haus gebaut haben und dieses nun an einem anderen Ort wieder aufbauen wollen. Sie können das Haus nicht komplett an seinen neuen Standort verfrachten. Stattdessen müssen Sie das Haus Stein für Stein auseinandernehmen und detailliert in einem Bauplan beschreiben, wie es wieder aufgebaut werden muss.

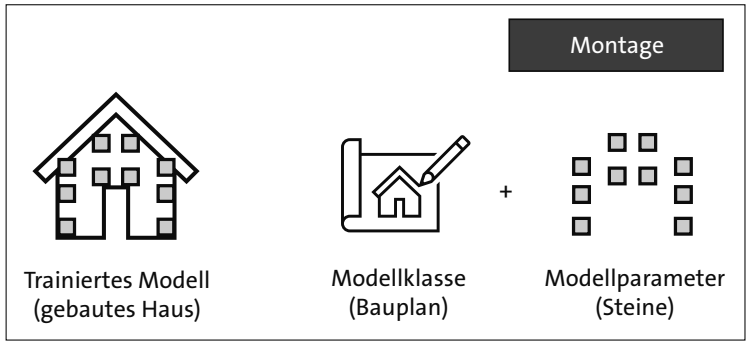


Abbildung 3.8 Vorgehen beim Speichern und Laden eines Modells

Ganz ähnlich geschieht es beim Speichern und Laden eines Modells. Der Bauplan ist bekannt – das ist die zuvor erstellte Modellklasse. Den Steinen in unserer Analogie entsprechen die Modellparameter. Beide werden separat gespeichert und beim Laden wieder miteinander verbunden.

3.6.1 Modellparameter speichern

Auf die Modellparameter haben wir Zugriff über die Methode `state_dict()` des Modells. Wir können es uns direkt ausgeben lassen. Hierbei handelt es sich um ein geordnetes Dictionary, in dem Tupel den jeweiligen Schichtnamen sowie die dazugehörigen Modellgewichte widerspiegeln:

```
model.state_dict()
OrderedDict([('linear.weight',
              tensor([[-1.3479e-01, -5.1887e-01, -1.8175e-01, ... ]])),
             ('linear.bias', tensor([3.9327]))])
```

Dieses *State Dictionary* wird nun mittels `torch.save()` in einer Datei gespeichert. Üblicherweise wird beim Speichern einer solchen Gewichtsdatei die Dateiendung *.pt* oder *.pth* verwendet.

Der Code zeigt, wie die Modellgewichte in der Datei *Model1.pth* gespeichert werden:

```
#%% save model weights
torch.save(model.state_dict(), 'models/Model1.pth')
```

Nachdem Sie jetzt wissen, wie Modelle gespeichert werden, schauen wir uns an, wie wir Modelle wiederherstellen können.

3.6.2 Modell laden

Dieser Schritt ist zweistufig. Im ersten Schritt wird eine Instanz der Modellklasse erstellt und anschließend werden die Modellgewichte ins Modell geladen.

Die Modellklasse kann in einer separaten Datei gespeichert werden. Listing 3.23 zeigt, wie die eigene Modellklasse definiert wird:

```
#%% packages
import torch

#%% Model class
class LinearRegression(torch.nn.Module):
    def __init__(self, input_size, output_size):
        super(LinearRegression, self).__init__()
        self.linear = torch.nn.Linear(input_size, output_size)

    def forward(self, x):
        x = self.linear(x)
        return x
```

Listing 3.23 Modellkasse (Quelle: 030_FirstModel_Regression\models\Model1.py)

Kommen wir nun zum eigentlichen Skript, das darauf aufbaut. In Listing 3.24 laden wir erst einmal alle Pakete. Unsere Modellklasse ist in ein Skript *models/Model1.py* ausgelagert, sodass wir von dort die Modellklasse `LinearRegression` laden können. Anschließend erstellen wir eine Instanz des Modells:

```
import torch
from models.Model1 import LinearRegression
import seaborn as sns
import matplotlib.pyplot as plt
# %% create model instance
model = LinearRegression(37, 1)
```

Listing 3.24 Modell laden – Pakete und Modellinstanz (Quelle: 030_FirstModel_Regression\40_ModelLoading.py)

Wir wollen überprüfen, dass die Modellgewichte erfolgreich geladen wurden. Dazu erstellen wir eine Funktion `show_model_parameters`, die die Modellgewichte als Histogramm darstellt:

```
#%% function to show model parameter distribution
def show_model_parameters(model):
    params = []
    for param in model.parameters():
        params.append(param.detach().numpy().flatten())
    g = sns.histplot(params, kde=True)
    # add title
    g.set_title('Model Parameter Distribution')
    g.set_xlabel('Parameter Value')
    g.set_ylabel('Frequency')
    return g
```

Listing 3.25 Modell laden – Funktion zur Visualisierung (Quelle: 030_FirstModel_Regression\40_ModelLoading.py)

Diese Funktion wird jetzt in Abbildung 3.9 genutzt, um die Modellgewichte direkt nach der Instanziierung des Modells und dann nach dem Laden der Modellgewichte zu zeigen:

```
show_model_parameters(model)
#%% load model weights
model.load_state_dict(torch.load('models/Model1.pth'))
<All keys matched successfully>
show_model_parameters(model)
```

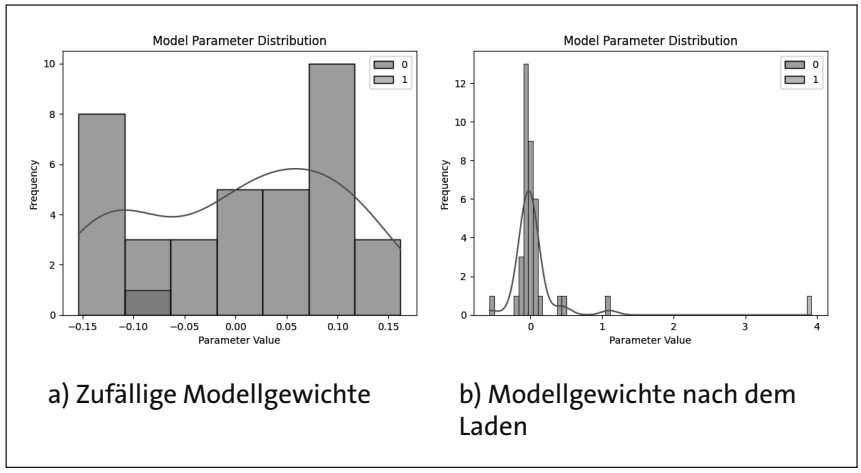


Abbildung 3.9 Modellgewichte (links: zufällige Gewichte; rechts: Gewichte nach dem Laden)

Links ist der Zustand direkt nach dem Erstellen der Modellinstanz zu sehen. Die Gewichte liegen alle rund um 0. Im rechten Bild sieht man, dass die Modellgewichte einen viel größeren Bereich abdecken und dass der Biaswert (Balken ganz rechts) bei nahezu 4 liegt. Das Laden der Modellparameter hat also funktioniert.

Damit kommen wir nun zur nächsten Erweiterung unseres Trainings, und Sie werden lernen, wie und warum die Daten aufgeteilt werden.

3.7 Data Sampling

In diesem Abschnitt beschäftigen wir uns mit dem nächsten Konzept, das wichtig für das Modelltraining ist, dem *Data Sampling*.

Sie lernen zunächst, was genau unter diesem Begriff zu verstehen ist und warum man diesen Ansatz benötigt. Danach lernen Sie, das Data Sampling zu implementieren.

3.7.1 Was ist Data Sampling?

Data Sampling ist der Prozess der Auswahl von Datenpunkten aus einem größeren Datensatz. Anstatt das Modell auf Basis des kompletten Datensatzes zu trainieren, wird nur ein Teil für das Training verwendet und ein anderer Teil für die Validierung der Daten. Das ist das Konzept des *Train Test Splits* und wird in Abbildung 3.10 dargestellt.

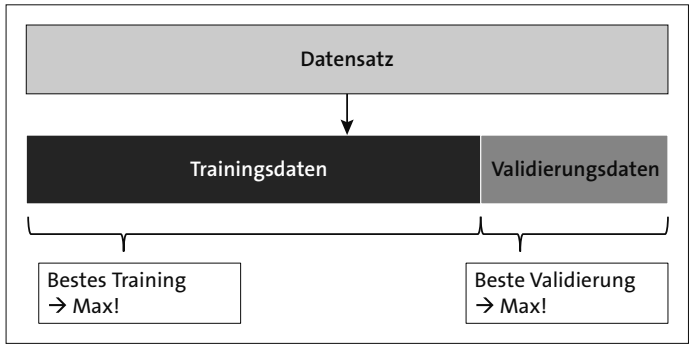


Abbildung 3.10 Konzept des »Train Test Splits«

Die ursprünglichen Daten werden in zwei (oder in einigen Fällen auch drei) separate Teile unterteilt:

- **Trainingsdatensatz (Train Dataset):** Der größte Teil der Daten wird verwendet, um das Modell zu trainieren.
- **Validierungsdatensatz (Validation Dataset):** Die *Validierungsdaten*, in der Regel der kleinere Teil der Daten, dienen dazu, die Leistungsfähigkeit des trainierten Modells mit Daten zu bewerten, die es noch nicht gesehen hat. So können Sie über-

prüfen, wie gut es generalisiert. Das Modell soll ja nicht nur auf den Trainingsdaten gut funktionieren, sondern vor allem auf neuen, noch unbekannten Daten.

- **Testdatensatz (Test Dataset):** Manchmal gibt es auch einen dritten Datensatz, den *Testdatensatz*. Dieser Datensatz wird erst ganz am Ende verwendet, um die finale, unvoreingenommene Leistung des trainierten und validierten Modells zu bewerten. Diese Daten darf das Modell während des Trainings und der Validierung niemals »sehen«.

Abbildung 3.10 zeigt auch, dass es vorteilhaft ist, den Trainingsdatensatz so groß wie möglich zu machen. Problematisch wird es, weil dasselbe für den Test- (oder auch Validierungsdatensatz) gilt. Wie kann dieser Zielkonflikt überwunden und das optimale Verhältnis gefunden werden? Damit befasst sich die folgende Infobox.

Optimales Aufteilungsverhältnis von Trainings- und Validierungsdaten

Es gibt keine allgemeingültige »perfekte« Aufteilung, da sie von verschiedenen Faktoren abhängt.

Die genaue Ausgestaltung hängt davon ab, ob es eine Zweiteilung (Trainings- und Validierungsdaten) oder eine Dreiteilung mit Trainings-/Validierungs-/Testdaten gibt.

Als groben Anhaltspunkt können Sie eine Aufteilung von 80 % Trainings- und 20 % Validierungsdaten annehmen. Falls eine Dreiteilung verwendet wird, können Sie 70 % für Trainingsdaten, 15 % für Validierungsdaten und 15 % für Testdaten vorsehen.

Im Folgenden gehen wir von einer Zweiteilung aus.

Einige Faktoren beeinflussen die Aufteilung:

- **Größe des Datensatzes:** Ein wichtiger Parameter ist die Größe des Datensatzes. Je größer der Datensatz ist, desto prozentual größer kann der Trainingsdatensatz sein, da die absolute Anzahl der Datenpunkte im Validierungsdatensatz immer noch groß genug ist, um statistisch signifikante Aussagen abzuleiten.
- **Modellkomplexität:** Je komplexer ein Modell ist, desto mehr Trainingsdaten benötigt es tendenziell, um ein Overfitting zu vermeiden. Hier sind Validierungsdaten sehr wichtig, um die Generalisierungsfähigkeit zu überwachen.
- **Zeitreihendaten:** Eine Besonderheit stellen Zeitreihendaten dar, da sie nicht zufällig aufgeteilt werden dürfen, um die zeitliche Reihenfolge zu erhalten. Diesem Thema widmen wir uns in Kapitel 9.
- **Unausgewogene Daten:** Bei unausgewogenen Daten (Imbalanced Datasets) handelt es sich um Daten bei Klassifizierungsproblemen, in denen bestimmte Klassen stark unterrepräsentiert sind. In diesen Fällen müssen Sie darauf achten, die Klassen in den verschiedenen Datensätzen ähnlich verteilt sind.

Die im nächsten Abschnitt vorgestellte *Kreuzvalidierung* (*Cross-Validation*) eröffnet Ihnen eine Möglichkeit, auf das Data Sampling zu verzichten.

3.7.2 Kreuzvalidierung

Insbesondere bei kleinen Datensätzen ist die Kreuzvalidierung eine sehr empfehlenswerte Technik. Dieses Verfahren wird hauptsächlich zur Ermittlung einer stabilen Modell-Performance verwendet und erweitert das zuvor beschriebene Data Sampling:

1. Der gesamte Datensatz wird in gleich große Teile (*Folds*) mit der Größe K aufgeteilt.
2. Das Modell wird K-mal trainiert, wobei in jeder Iteration ein anderer Fold als Validierungsdatensatz verwendet wird und die restlichen K-1 Folds als Trainingsdatensatz dienen.
3. Die Metriken zur Beurteilung der Modellperformance werden für jedes Modell ermittelt.
4. Die ermittelten Metriken werden gemittelt, um eine robuste Einschätzung der gesamten Modellleistung zu erhalten.

Der Prozess der Aufteilung der Daten ist in Abbildung 3.11 verdeutlicht.

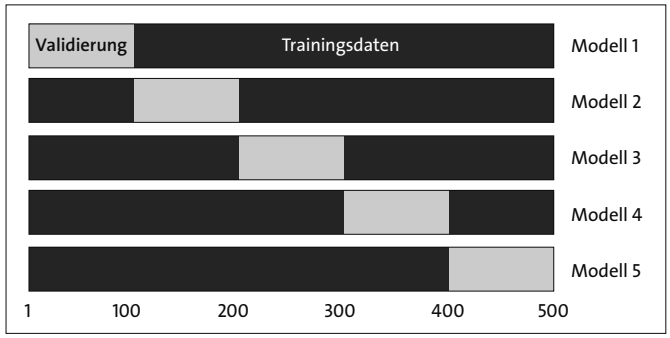


Abbildung 3.11 Kreuzvalidierung

Üblicherweise werden die Validierungsdaten nicht »am Stück« festgelegt, sondern zufällig ausgewählt. Sie wurden hier nur vereinfacht so dargestellt, um das Prinzip zu erklären.

Übliche Werte für die Anzahl an Folds sind 5 oder 10. Das Verfahren hat den Vorteil, dass jeder Datenpunkt sowohl zum Trainieren als auch zum Validieren verwendet wird. Ein Nachteil ist natürlich, dass der Rechenaufwand stark ansteigt. Wo zuvor nur ein Modell trainiert und validiert wurde, muss es nun K-mal trainiert und validiert werden.

Aus diesem letzteren Grund wird die Kreuzvalidierung in diesem Buch nicht weiter eingesetzt. Aber es ist wichtig, dieses Konzept zu kennen, um es im Zweifelsfall einsetzen zu können.

3.7.3 Warum braucht man das?

Man will also die Fähigkeit des Modells zur Generalisierung sicherstellen. Mit anderen Worten: *Overfitting (Überanpassung)* soll vermieden werden.

Von Overfitting spricht man, wenn ein Modell die Trainingsdaten zu genau lernt. Eigentlich soll es ja nur die grundlegenden Zusammenhänge verstehen. Bei einem Overfitting lernt es auch das Rauschen und andere spezifische Eigenheiten auswendig.

Abbildung 3.12 zeigt Beispiele für ein unterangepasstes und ein überangepasstes Modell.

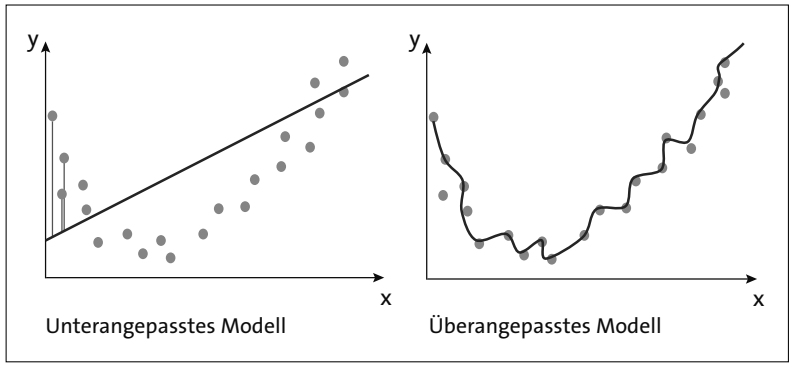


Abbildung 3.12 Unterangepasstes und überangepasstes Modell

In Abbildung 3.12 sehen Sie links ein Beispiel für eine *Unteranpassung*. Der Einfachheit halber handelt es sich um ein Regressionsproblem mit einer unabhängigen Variable X und der abhängigen Variable y. Die Punktewolke ist leicht verrauscht, folgt aber grob einem quadratischen Trend.

Um das linke Teilbild zu erzeugen, wurde ein lineares Modell trainiert, das den Zusammenhang über eine Gerade widerspiegelt. Die Fehler der einzelnen Punkte sind als vertikale Linien verdeutlicht. Die Fehler sind relativ hoch.

Auf dem rechten Teilbild ist das andere Extrem dargestellt. Hier ist das Modell so an die Trainingsdaten angepasst, dass versucht wird, wirklich jeden einzelnen Punkt exakt nachzuvollziehen. Das sieht bei den Trainingsdaten sehr gut aus und der Fehler zwischen dem Modell (Linie) und dem jeweiligen Datenpunkt ist minimal, aber in der Praxis zeigt sich, dass das Modell schlecht mit Daten umgehen kann, die es vorher nie gesehenen hat, und somit sehr schlecht generalisiert.

Man kann sich das wie einen Schüler vorstellen, der Antworten einfach auswendig lernt, ohne die zugrunde liegenden Zusammenhänge zu verstehen. Der Schüler kann alle alten Fragen perfekt beantworten. Aber sobald eine Frage leicht abgewandelt wird, scheitert er.

Ein gutes Modell findet den Mittelweg zwischen Unter- und Überanpassung.

3.7.4 Coding: Aufteilung in Trainings- und Validierungsdaten

Wir erweitern unseren bisherigen Code, um nun auch Trainings- und Validierungsdaten zu berücksichtigen.

Listing 3.26 zeigt alle benötigten Pakete, und es werden die Daten aus dem `DataPrep`-Skript importiert:

```
#%% packages
import numpy as np
import pandas as pd
import os
import matplotlib.pyplot as plt
import torch
from torch.utils.data import Dataset, DataLoader
from DataPrep import X, y
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score
import seaborn as sns
```

Listing 3.26 Train-Test-Split – Pakete und Datenimport (Quelle: 030_FirstModel_Regression\50_DataSplitting.py)

Die Hyperparameter, die Sie in Listing 3.27 sehen, umfassen die maximale Anzahl an Epochen `EPOCHS`, die `LEARNING_RATE` sowie die `BATCH_SIZE`:

```
#%% Hyperparameters
EPOCHS = 20
LEARNING_RATE = 0.1
BATCH_SIZE = 512
```

Listing 3.27 Train-Test-Split – Hyperparameter (Quelle: 030_FirstModel_Regression\50_DataSplitting.py)

Nun kommen wir zu dem besprochenen Ansatz des *Datensplits*. Wir teilen in Listing 3.28 die Daten in Trainingsdaten und Validierungsdaten auf. Hierzu wird die Funktion `train_test_split` verwendet. In unserem Fall werden 80 % für die Trainingsdaten und 20 % für die Testdaten verwendet:

```
#%% split data
X_train, X_val, y_train, y_val = train_test_split(X, y,
    test_size=0.2,
    random_state=42)
```

Listing 3.28 Train-Test-Split – Datensplit (Quelle: 030_FirstModel_Regression\50_DataSplitting.py)

Es ist üblich und fast immer ratsam, die Daten zu skalieren. Listing 3.29 zeigt, wie wir hierbei vorgehen. Die unabhängigen Trainingsfeatures `X_train` werden skaliert. In dem Schritt werden auch die Skalierungsparameter ermittelt und im Objekt `scaler` gespeichert.

Ein interessantes Detail ist hierbei, dass man die Validierungsdaten `X_val` auf Basis der Skalierungsparameter der Trainingsdaten anpasst. Der Hintergrund ist, dass man davon ausgehen muss, dass nur die Trainingsdaten und deren Parameter bekannt sind. Würde man die Skalierung der Validierungsdaten auf den eigenen Verteilungswerten vornehmen, hieße das, dass man implizit Informationen über die Validierungsdaten besitzt, die man praktisch nicht haben sollte. Der Einfluss mag in der Regel klein sein, aber ich empfehle Ihnen, dieser Best Practice zu folgen. Sie erhalten die angepassten Validierungsdaten, indem Sie die `transform`-Methode des `scalers` auf die Daten anwenden:

```
#%% scale data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_val = scaler.transform(X_val)
```

Listing 3.29 Train-Test-Split – Skalierung der Daten (Quelle: 030_FirstModel_Regression\50_DataSplitting.py)

An der in Listing 3.30 gezeigten `Dataset`-Klasse gibt es keine Änderungen gegenüber früheren Abschnitten:

```
#%% Dataset class
class AnxietyDataset(Dataset):
    def __init__(self, X, y):
        self.X = torch.from_numpy(X.astype(np.float32))
        self.y = torch.from_numpy(y.astype(np.float32))

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```

Listing 3.30 Train-Test-Split – die Dataset-Klasse (Quelle: 030_FirstModel_Regression\50_DataSplitting.py)

Spannend wird es jetzt wieder beim `DataLoader`. Da es zwei Datensätze gibt, werden auch zwei `DataLoader` (`train_dataloader` und `val_dataloader`) instanziiert:

```
#%% DataLoader
train_dataset = AnxietyDataset(X_train, y_train)
train_dataloader = DataLoader(train_dataset,
                              batch_size=BATCH_SIZE,
                              shuffle=True)

val_dataset = AnxietyDataset(X_val, y_val)
val_dataloader = DataLoader(val_dataset,
                            batch_size=BATCH_SIZE,
                            shuffle=False)
```

Listing 3.31 Train-Test-Split – DataLoader (Quelle: 030_FirstModel_Regression\50_DataSplitting.py)

In Listing 3.32 wird die Modellklasse und damit die Instanz `model` erstellt:

```
#%% Model class
class LinearRegression(torch.nn.Module):
    def __init__(self, input_size, output_size):
        super(LinearRegression, self).__init__()
        self.linear = torch.nn.Linear(input_size, output_size)

    def forward(self, x):
        x = self.linear(x)
        return x

#%% Model instance
model = LinearRegression(input_size=train_dataset.X.shape[1],
                         output_size=1)
```

Listing 3.32 Train-Test-Split – Modellklasse und Modellinstanz (Quelle: 030_FirstModel_Regression\50_DataSplitting.py)

Wie Sie in Listing 3.33 sehen, verwenden wir die Verlustfunktion `MSELoss`, und der Optimizer basiert auf `Adam`:

```
#%% Loss function
loss_fun = torch.nn.MSELoss()

#%% Optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)
```

Listing 3.33 Train-Test-Split – Verlustfunktion und Optimierer (Quelle: 030_FirstModel_Regression\50_DataSplitting.py)

Nun kommen wir zum Training des Modells. Die entscheidenden Unterschiede zu früheren Trainings sind:

1. Den zwei verschiedenen Datensätzen wird Rechnung getragen, indem ihre Verluste separat getrackt werden. Dafür werden die Listen `epoch_loss_train` und `epoch_loss_val` erstellt.
2. Am Ende jeder Epoche werden die Verluste der Validierungsdaten ermittelt. Dafür wird der Scope `torch.no_grad()` verwendet. In diesem Scope wird mit einem Forward-Pass die Vorhersage berechnet und anschließend der Verlustwert ermittelt.

```
#%%
# (1) Empty lists initialized
loss_train_list, loss_val_list = [], []
for epoch in range(EPOCHS):
    epoch_loss_train = 0
    epoch_loss_val = 0
    for i, (X_train_batch, y_train_batch) in enumerate(train_dataloader):
        # get batch

        # forward pass
        y_pred_train = model(X_train_batch)

        # calculate loss
        loss_train = loss_fun(y_pred_train, y_train_batch.reshape(-
1, 1)).mean()

        # backward pass
        loss_train.backward()

        # update weights and biases
        optimizer.step()

        # zero gradients
        optimizer.zero_grad()

        # Store loss for plotting
        epoch_loss_train += loss_train.item()

    # (2) evaluate on test set
    with torch.no_grad():
        for X_val_batch, y_val_batch in val_dataloader:
            y_pred_val = model(X_val_batch)
            loss_val = loss_fun(y_pred_val,
                                y_val_batch.reshape(-1, 1)).mean()
            epoch_loss_val += loss_val.item()
```

```
    # Store the losses for plotting
    loss_train_list.append(epoch_loss_train / len(train_dataloader))
    loss_val_list.append(epoch_loss_val / len(val_dataloader))

    # Print loss for this epoch
    print(f"Epoch {epoch}, Train Loss: {epoch_loss_train}, Test Loss: {loss_
val.item()}")
```

Listing 3.34 Train-Test-Split – Modelltraining (Quelle: 030_FirstModel_Regression\50_DataSplitting.py)

In Listing 3.35 sehen Sie nun die Visualisierung der Verluste auf Basis der Trainings- und Validierungsverluste. Der Großteil des Codes dient zur Skalierung der Daten, sodass die Verluste jeweils auf den Bereich 0 bis 1 beschränkt werden. Andernfalls könnte es sein, dass die Verluste so unterschiedlich hoch sind, dass sie schlecht zu erkennen sind.

```
#%% plot loss
# Convert to numpy arrays
loss_train_arr = np.array(loss_train_list)
loss_val_arr = np.array(loss_val_list)

# Train loss: scale independently
train_min = loss_train_arr.min()
train_max = loss_train_arr.max()
train_range = train_max - train_min if train_max > train_min else 1
loss_train_scaled = (loss_train_arr - train_min) / train_range

# Val loss: scale independently
val_min = loss_val_arr.min()
val_max = loss_val_arr.max()
val_range = val_max - val_min if val_max > val_min else 1
loss_val_scaled = (loss_val_arr - val_min) / val_range

sns.lineplot(x=range(EPOCHS), y=loss_train_scaled, color='blue', label=
'Train')
sns.lineplot(x=range(EPOCHS), y=loss_val_scaled, color='red', label=
'Validation')
plt.title('Losses over Epochs: Train (blue) vs. Validation (red)')
plt.xlabel('Epoch [-]')
plt.ylabel('Loss [-]')
plt.legend()
plt.show()
```

Listing 3.35 Train-Test-Split – Trainings- und Validierungsverluste (Quelle: 030_First-Model_Regression\50_DataSplitting.py)

Das Ergebnis des Aufwands ist in Abbildung 3.13 zu sehen. Die Trainingsverluste (blau) und die Validierungsverluste (rot) folgen demselben Trend. Zunächst nehmen sie sehr stark ab, um dann asymptotisch gegen null zu konvergieren.

Dass beide Kurven nahezu deckungsgleich sind, ist eher unüblich und liegt an dem konkreten Datensatz, mit dem das Modell trainiert wurde.

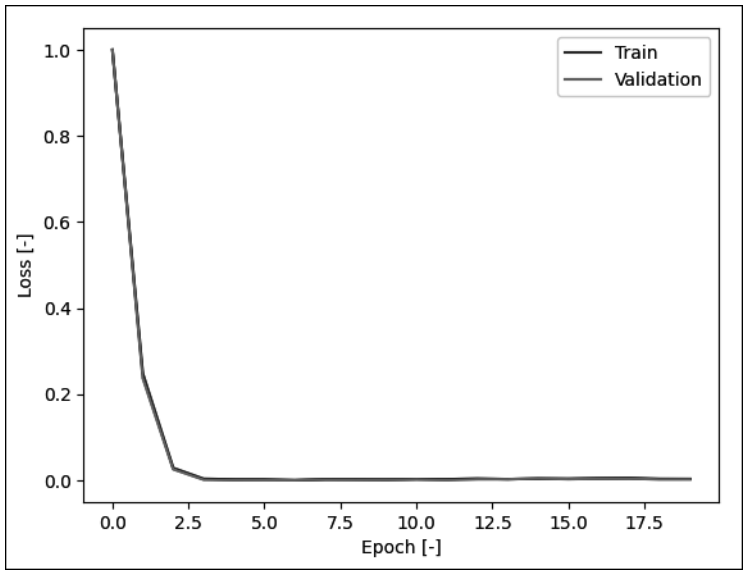


Abbildung 3.13 Train-Test-Split – Trainings- und Validierungsverluste

Damit haben wir das Ende dieses Abschnitts erreicht. Sie wissen nun, was Data Sampling ist und wie Sie es implementieren können.

3.8 Zusammenfassung

Dieses Kapitel lieferte einen Einstieg in das Training von Deep-Learning-Modellen mit PyTorch.

Nachdem Sie in Abschnitt 3.1 gelernt haben, wie die Daten vorzubereiten sind, begann das eigentliche Training in Abschnitt 3.2 mit dem Datenimport sowie dem Modelltraining und der Modellevaluierung.

Dieses erste Modelltraining hat ein funktionsfähiges Modell geliefert, aber noch reichlich Platz für Verbesserungen gelassen. Denen haben wir uns in den Nachfolgeabschnitten gewidmet.

Dazu habe ich zunächst in Abschnitt 3.3 die Modellklasse und den Optimierer eingeführt. Anschließend haben Sie in Abschnitt 3.4 gelernt, was Batches sind und warum und wie sie implementiert werden.

Eine weitere Abstraktion habe ich in Abschnitt 3.5 eingeführt, in dem Sie Dataset und DataLoader kennenlernten. Mit diesem Konzept können Sie die Daten vom eigentlichen Modelltraining separieren, was Ihren Code modularer macht und somit einfacher zu erweitern und zu pflegen.

Da wir üblicherweise Modelle trainieren, um sie danach einzusetzen, mussten Sie lernen, wie Modelle gespeichert und anschließend wieder geladen werden können. Das war das Thema von Abschnitt 3.6.

Zu guter Letzt wurden die Daten in Abschnitt 3.7 in Trainings- und Validierungsdaten aufgeteilt. Dieser Schritt, der Data Sampling genannt wird, stellt sicher, dass das Modell zu generalisieren lernt, sodass es später nicht nur gut mit den Trainingsdaten, sondern auch mit unbekannten Daten funktioniert.

Mit diesem Wissen sind Sie jetzt dafür gerüstet, weitere Modellarchitekturen kennenzulernen. Die Konzepte aus diesem Kapitel werden Sie durch den Rest des Buches begleiten.

Kapitel 10
Sprachmodelle

»Die Grenzen meiner Sprache bedeuten die Grenzen meiner Welt.«
– Ludwig Wittgenstein, Philosoph

Unsere Sprache ist nicht nur ein Werkzeug zur Kommunikation, sondern auch das Medium, durch das wir unser Denken, unsere Wahrnehmung und unser Verständnis strukturieren. Wenn uns die Worte fehlen, um etwas zu beschreiben, verengt sich unsere Sichtweise. Wo wir Sprache erweitern, eröffnen sich automatisch neue Möglichkeiten, die Welt zu begreifen.

Genau an dieser Stelle setzen *große Sprachmodelle* an. Sie werden den *Foundation Models* zugerechnet. Dabei handelt es sich um vielseitige Basismodelle, die auf riesigen Datensätzen trainiert wurden. Sie dienen als Fundament, das für eine Vielzahl verschiedener Aufgaben angepasst werden kann.

Sprachmodelle können mehr, als nur Texte erzeugen. Dadurch, dass sie Muster in Sprache sichtbar machen, erweitern sie unser Ausdrucksspektrum.

Die Frage ist also nicht nur, wie Sprachmodelle funktionieren, sondern auch, wie sie unsere Welt verändern.

Große Sprachmodelle (*Large Language Models*, *LLMs*) haben sich als transformative Technologie etabliert, die das maschinelle Verstehen und Generieren von Texten grundlegend neu definiert.

Im Kern handelt es sich bei LLMs um komplexe neuronale Netze, die auf unvorstellbar großen Mengen an Textdaten trainiert wurden. Dadurch sind die Modelle in der Lage, komplexe sprachliche Muster und Zusammenhänge zu erkennen. Das verleiht ihnen die Fähigkeit, in sich stimmige und inhaltlich passende Texte zu erstellen, Fragen zu beantworten, Übersetzungen durchzuführen und kreative Inhalte zu erstellen.

Die bemerkenswerten Fähigkeiten von LLMs beruhen auf Fortschritten in der Modellarchitektur, insbesondere der Transformer-Architektur. Diese spezielle Art von Netzwerk ermöglicht es, langfristige Abhängigkeiten in Texten effizient zu verarbeiten.

In diesem Kapitel beginnen wir in Abschnitt 10.1 gleich mit einem Sprung ins kalte Wasser: Dort lernen Sie, wie Sie Sprachmodelle direkt mit Python nutzen können.

Anschließend gehe ich in Abschnitt 10.2 auf Modellparameter ein, die Ihnen helfen, die Modellantwort zu beeinflussen. Sie werden lernen, was es mit Parametern wie der *Temperatur*, *Top-p* oder *Top-K* auf sich hat.

Heutzutage gibt es eine gewaltige Auswahl an Sprachmodellen. Welche Parameter man zur Auswahl des richtigen Modells heranziehen kann, beleuchte ich in Abschnitt 10.3.

Um komplexe Workflows abzubilden, empfiehlt es sich, sogenannte *Chains* zu verwenden, da sie helfen, den Code zu modularisieren und sehr viel Flexibilität bieten. Chains sind die grundlegenden Bausteine, um LLMs mit anderen Komponenten wie Datenquellen oder Werkzeugen zu verbinden. Den Chains widmen wir uns in Abschnitt 10.6. Da sie aber auf Prompt Templates (siehe Abschnitt 10.5) und Messages (siehe Abschnitt 10.4) beruhen, müssen Sie diese Konzepte zuvor kennenlernen.

Nachdem Sie sich mit Chains vertraut gemacht haben, lernen Sie in Abschnitt 10.7 eine bestimmte Form von Chain kennen, die strukturierte Ergebnisse (Outputs) zurückliefert. Das kann ungemein hilfreich sein, wenn das Modellergebnis in einer Datenbank abgespeichert werden oder als Eingabe für einen folgenden Prozess dienen soll.

Zum Abschluss des Kapitels werfen wir einen Blick in die zugrunde liegende Architektur von Sprachmodellen. Ohne die Transformer-Architektur wären heutige Sprachmodelle nicht denkbar. Daher widmen wir uns in einem technischen Deep Dive in Abschnitt 10.8 diesem bahnbrechenden Netzwerktyp.

Legen wir nun aber direkt los, und sehen wir uns an, wie Sie ein Sprachmodell direkt aus Python heraus verwenden können.

10.1 Nutzung von LLMs mit Python

Aufgrund der Komplexität und der technischen Anforderungen beim Training von Sprachmodellen steigen wir auf einer höheren Abstraktionsebene ein. Sie werden bereits trainierte Sprachmodelle nutzen und vor allem lernen, wie Sie sie effizient nutzen können.

Abbildung 10.1 zeigt eine Reihe von beliebten Sprachmodellen und Modellfamilien. Zu den leistungsfähigsten Modellen gehören die Flaggschiffmodelle von OpenAI (z. B. GPT-5), Google (Gemini 2.5 Pro), Anthropic (Claude Opus 4.1) und X-AI (Grok 4). Erwähnenswert ist hier Mistral mit LeChat als europäischer Anbieter, der auch DSGVO[1]-konform arbeitet. Einige Modellanbieter gewähren nur Zugriff auf ihre *proprietären Modelle* über eine API. Im Unterschied dazu gibt es auch *Open-Weight-* oder *Open-Source*-Modelle. Mehr dazu später.

1 DSGVO steht für »Datenschutz-Grundverordnung«. Das ist ein EU-Gesetz, das die Verarbeitung personenbezogener Daten durch Unternehmen und Behörden europaweit vereinheitlicht und die Rechte der Bürger auf den Schutz ihrer Daten stärkt.

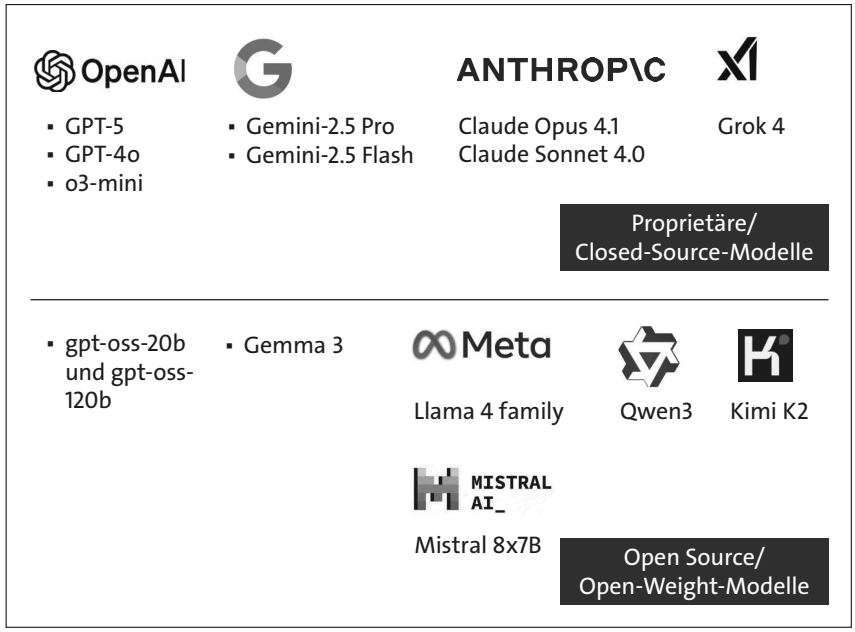


Abbildung 10.1 Beliebte Sprachmodelle und Modellfamilien

Für viele Menschen ist das Thema »Sprachmodell« synonym mit *ChatGPT*. Das liegt daran, dass *OpenAI* mit dem Release von ChatGPT Ende 2022 den Stein ins Rollen gebracht hat und dass seitdem die Modellreihe GPT die berühmteste Modellreihe bei den Sprachmodellen ist.

Das sind sehr leistungsstarke Modelle, für die Sie aber bezahlen müssen. Da es aber auch andere extrem fähige LLMs gibt, die als Open Source bereitgestellt werden, zeige ich Ihnen, wie Sie LLMs kostenlos über *Groq*[1] nutzen können. Sie werden sehen, dass die Interaktion mit den Modellen dank des Python-Frameworks *LangChain* sehr einfach ist. Nach diesen beiden Beispielen werden Sie in der Lage sein, sich mit jedem anderen LLM-Anbieter zu verbinden.

10.1.1 Coding: Nutzung von OpenAI

Um OpenAI nutzen zu können, benötigen Sie einen *API-Schlüssel*. Diese Modelle sind nicht kostenlos, also werden Sie für ihre Nutzung zur Kasse gebeten. Die Preise sinken ständig im Laufe der Zeit, und Sie können die aktuellen Preise unter *https://openai.com/api/pricing/* nachschauen.

1 Ein entscheidender Unterschied besteht zwischen *Grok* und *Groq*: Bei Groq handelt es sich um einen Hardwareanbieter, der den Zugriff auf Open-Weight-LLMs bereitstellt, während es sich bei Grok um ein Sprachmodell von X-AI handelt.

Einen API-Schlüssel einrichten

Zuerst müssen Sie einen API-Schlüssel einrichten. Wenn Sie einen Webdienst nutzen möchten, benötigen Sie normalerweise einen Benutzernamen und ein Passwort, um auf den Dienst zuzugreifen. Wenn Sie einen Dienst jedoch programmgesteuert nutzen möchten, benötigen Sie einen *API-Schlüssel*. Ein API-Schlüssel ist also wie eine Kombination aus Benutzername und Passwort.

Wie können Sie einen API-Schlüssel bekommen, wenn Sie noch keinen haben? Folgen Sie hierzu diesen Schritten:

1. Navigieren Sie zu *https://platform.openai.com/*.
2. Erstellen Sie einen Account.
3. Aktivieren Sie die Abrechnung und laden etwas Geld auf ihr Konto.
4. Gehen Sie zum Bereich der API-Schlüssel und erstellen Sie einen neuen API-Schlüssel. Der Name, den Sie im Web-Frontend angeben, ist nur für die spätere Wiedererkennung relevant. Praktisch benötigen Sie nur den Schlüssel.
5. Kopieren Sie den Schlüssel in die Zwischenablage.
6. Fügen Sie ihn in eine Datei namens *.env* ein. Sie sollte so aussehen: `sk-proj...` Mehr dazu erfahren Sie im nächsten Abschnitt.

Umgebungsvariablen

Es ist generell eine gute Praxis, Code von Anmeldeinformationen zu trennen. Daher speichern Sie den API-Schlüssel in einer separaten Datei. Ein gängiger Ansatz ist, ihn in einer Datei namens *.env* zu speichern und diese im Arbeitsordner abzulegen. In dieser Datei speichern Sie den API-Schlüssel und möglicherweise viele weitere Schlüssel, falls nötig. Listing 10.1 zeigt, wie eine Umgebungsdatei aussehen sollte:

```
OPENAI_API_KEY = sk-proj...
```

Listing 10.1 Beispielinhalt einer ».env«-Datei

Die API-Schlüssel werden als Umgebungsvariablen behandelt. Umgebungsvariablen sind typischerweise Variablen, die von Ihrem Betriebssystem verwendet werden. Unsere Variable heißt `OPENAI_API_KEY`. Sie hat einen Wert, der auf der rechten Seite des Gleichheitszeichens definiert werden muss. Es ist wichtig, im Codeskript denselben Schlüsselnamen zu verwenden.

Wenn Sie zögern, Ihre Bankdaten im Internet anzugeben, überspringen Sie diese Lektion und gehen zur nächsten, um direkt mit Groq zu arbeiten, das kostenlosen Zugang zu den Modellen bietet. Verwechseln Sie Groq nicht mit *Grok*: Groq ist ein KI-Startup, das sich auf die Entwicklung von Chips für schnelle Inferenz von LLMs konzentriert, während Grok ein LLM ist, das als Initiative von Elon Musk ins Leben gerufen wurde.

Coding-Skript

Lassen Sie uns mit unserem Coding-Skript anfangen. Sie finden es im Materialordner unter *100_LLM/10_model_chat_openai.py*. Hier ist es eine gute Praxis, alle benötigten Pakete und Funktionen am Anfang der Datei zu platzieren. Lassen Sie uns durchgehen, was wir hier brauchen.

Das Paket `os` wird benötigt, um die Umgebungsvariablen abzurufen und zu laden. Alle großen Modellanbieter bieten Pakete zur Integration in LangChain an. Hier verwenden wir also `langchain_openai`. Das Paket `dotenv` ist erforderlich, um mit der Datei der Umgebungsvariablen zu arbeiten. Seine Funktion `load_dotenv()` lädt den Inhalt der *.env*-Datei und stellt ihn als Umgebungsvariablen zur Verfügung:

```
#%% packages
import os
from langchain_openai import ChatOpenAI
from dotenv import load_dotenv
load_dotenv('.env')
```

Sie können ganz einfach überprüfen, ob der API-Schlüssel verfügbar ist, indem Sie `os.getenv('OPENAI_API_KEY')` ausführen. Dadurch sollten Sie den API-Schlüssel auf dem Bildschirm angezeigt bekommen.

Jetzt erstellen wir eine Instanz des Modells, das wir verwenden werden, indem wir die `ChatOpenAI`-Klasse nutzen. Dafür brauchen wir einen Modellnamen. Wir wählen hier `gpt-4o-mini`. Ein weiterer wichtiger Parameter ist die Temperatur (mehr zu Modellparametern lesen Sie in Abschnitt 10.2). Dieser Parameter steuert die Kreativität des Modells. Und Sie müssen den API-Schlüssel übergeben, um sich zu authentifizieren und es OpenAI zu ermöglichen, die Kosten basierend auf Ihrer Nutzung zu berechnen.

```
MODEL_NAME = 'gpt-4o-mini'
model = ChatOpenAI(model_name=MODEL_NAME,
 temperature=0.5,
 api_key=os.getenv('OPENAI_API_KEY'))
```

Das `model`-Objekt hat eine sehr wichtige Methode: `invoke()`. Damit können Sie das Modell basierend auf bestimmten Parametern ausführen. In unserem ersten Beispiel bitten wir das Modell um Informationen zu »Was ist LangChain?«. Das Ergebnis wird in einem Objekt gespeichert. Das Ergebnis ist ein Objekt des Typs `AIMessage`. Wir können herausfinden, welche Informationen wir vom Modellaufruf erhalten haben, indem wir uns die Ausgabe seiner `model_dump()`-Methode anschauen. Die Modellausführung und das erzeugte Ergebnis sehen Sie in Listing 10.2:

```
res = model.invoke("What is a LangChain?")
res.model_dump()
{'content': 'LangChain is ...',
```

```
 'additional_kwargs': {'refusal': None},
 'response_metadata': {'token_usage': {'completion_tokens': 290,
   'prompt_tokens': 13,
   'total_tokens': 303,
   'completion_tokens_details': {'accepted_prediction_tokens': 0,
    'audio_tokens': 0,
    'reasoning_tokens': 0,
    'rejected_prediction_tokens': 0},
   'prompt_tokens_details': {'audio_tokens': 0, 'cached_tokens': 0}},
  'model_name': 'gpt-4o-mini-2024-07-18',
  'system_fingerprint': 'fp_560af6e559',
  'id': 'chatcmpl-C5oA3wS5xLKygiq66IWHdOWbFigRC',
  'service_tier': 'default',
  'finish_reason': 'stop',
  'logprobs': None},
 'type': 'ai',
 'name': None,
 'id': 'run--95a59d13-84a9-4d1b-838a-38fa7b8b0d92-0',
 'example': False,
 'tool_calls': [],
 'invalid_tool_calls': [],
 'usage_metadata': {'input_tokens': 13,
  'output_tokens': 290,
  'total_tokens': 303,
  'input_token_details': {'audio': 0, 'cache_read': 0},
  'output_token_details': {'audio': 0, 'reasoning': 0}}}
```

Listing 10.2 OpenAI-Nutzung

Es gibt eine Menge Informationen, die vom Modell zurückkommen. Lassen Sie uns mit dem Wichtigsten anfangen: dem Inhalt `content`. Diese Eigenschaft enthält den tatsächlichen Modellausgabe-Prompt. Von den anderen Eigenschaften möchte ich nur die `response_metadata` erwähnen. Diese enthält Informationen zur Token-Nutzung. Sie werden für Eingabe-Tokens und Ausgabe-Tokens zur Kasse gebeten. Hier können Sie sehen, wie viele Tokens in der Anfrage verwendet wurden.

Sie können sich mit den verschiedenen Modellen aus der Modellfamilie von OpenAI vertraut machen, indem Sie die Modellübersicht studieren, die Sie unter *https://platform.openai.com/docs/models/overview* finden. Einige wichtige Funktionen sind unten im Kasten aufgeführt.

Sie sind natürlich nicht auf die OpenAI-Modellfamilie beschränkt. Sie können mit vielen anderen LLMs arbeiten. Jetzt werden Sie entdecken, wie man mit Open-Source-LLMs arbeitet, die Sie kostenlos über Groq ausführen können.

Die OpenAI-Modellfamilie

OpenAI hat eine Modellfamilie (*https://platform.openai.com/docs/models*) geschaffen, die aus mehreren Modellen besteht, die für verschiedene Aufgaben geeignet sind:

- **Sprachmodelle** wie die GPT-Familie (z. B. GPT-5) können Text verarbeiten und erzeugen, und einige von ihnen können auch Bilder erzeugen.
- **Text-zu-Bild-Generierung:** GPT Image 1 und DALL-E 3 sind Modelle, die Bilder generieren und bearbeiten können.
- **Text-zu-Sprache (TTS):** Mehrere Modelle (wie z. B. GPT-4o mini TTS) können Text in natürliche, gesprochene Audios umwandeln.
- **Realtime-Modelle:** Mit Realtime-Modellen können Sie Text und Audio-Ein- und -Ausgaben in Echtzeit erstellen.
- **Text-Embeddings**: *Embeddings* sind numerische Darstellungen von Text. Solche Embeddings sind das Fundament der Verarbeitung natürlicher Sprache.

10.1.2 Coding: Nutzung von Groq

Groq ist ein Unternehmen, das KI-Hardware entwickelt, die schnelle Inferenz ermöglicht. Für Entwickler bietet Groq Zugang zu LLMs, insbesondere zu Open-Source-LLMs. Sie können den Service kostenlos nutzen, müssen sich aber mit einem API-Schlüssel authentifizieren. Der erste Schritt ist also, zu *https://console.groq.com/* zu gehen, ein Konto einzurichten und einen API-Schlüssel zu erstellen, den Sie in Ihrem Code verwenden können.

Bitte kopieren Sie diesen API-Schlüssel und speichern Sie ihn in einer Datei namens *.env* im Arbeitsordner. Der Inhalt der Datei sollte so aussehen:

```
GROQ_API_KEY = gsk_...
```

Listing 10.3 Ausschnitt aus der ».env«-Datei

Das Skript, das Sie unter *100_LLM/20_model_chat_groq.py* finden, beginnt damit, die relevanten Pakete zu laden. Das Hauptpaket ist `langchain_groq` – es ist die Schnittstelle, um die Modelle aus der Groq-Modellfamilie zu verwenden. Die Pakete `os` und `dotenv` werden verwendet, um Umgebungsvariablen einzurichten und abzurufen, die den Groq-API-Schlüssel enthalten.

```
#%% packages
import os
from langchain_groq import ChatGroq
from dotenv import load_dotenv
load_dotenv('.env')
```

Wir müssen nun ein Modell auswählen. Details zu spezifischen Modellen finden Sie in der Übersicht über Groq-Modelle unter *https://console.groq.com/docs/models*.

Hier wählen wir jetzt ein Modell aus der *Llama*-Familie. Das ist ein Open-Source-Modell, genauer gesagt ein *Open-Weight-Modell*. Das bedeutet: Das Modell wird der Öffentlichkeit zur kostenlosen Nutzung zur Verfügung gestellt, allerdings sind nicht alle Details zu den verwendeten Datensätzen und dem Trainingsprozess öffentlich zugänglich.

Nachdem wir uns für ein Modell entschieden haben, können wir eine Instanz der `ChatGroq`-Klasse erstellen. Bei dieser Instanziierung übergeben wir den Namen des Modells als Parameter. Wir müssen auch den API-Schlüssel übergeben, den wir zuvor erstellt haben. Diese beiden Parameter sind Pflicht. Unter vielen anderen verfügbaren Parametern setzen wir nur den Temperaturparameter, der die Kreativität des Modells steuert. (Mehr über die Modellparameter erfahren Sie in Abschnitt 10.2.) Damit haben wir alles bereit, um mit dem LLM zu interagieren:

```
MODEL_NAME = 'llama-3.3-70b-versatile'
model = ChatGroq(model_name=MODEL_NAME,
 temperature=0.5,
 api_key=os.getenv('GROQ_API_KEY'))
```

Wir fragen das Modell »What is a Huggingface?« über die `invoke()`-Methode:

```
# %% Run the model
res = model.invoke("What is a Huggingface?")
```

Mit der in Listing 10.4 gezeigten `model_dump()`-Methode bekommen wir einen Überblick über die Ausgabe des Modells:

```
# %% find out what is in the result
res.model_dump()
{'content': 'Hugging Face is a popular open-source library and platform for
natural language processing (NLP) and machine learning (ML) ...',
 'additional_kwargs': {},
 'response_metadata': {'token_usage': {'completion_tokens': 314,
   'prompt_tokens': 42,
   'total_tokens': 356,
   'completion_time': 1.032163328,
   'prompt_time': 0.010863083,
   'queue_time': 0.085254578,
   'total_time': 1.043026411},
  'model_name': 'llama-3.3-70b-versatile',
  'system_fingerprint': 'fp_2ddfbb0da0',
  'service_tier': 'on_demand',
```

```
  'finish_reason': 'stop',
  'logprobs': None},
 'type': 'ai',
 'name': None,
 'id': 'run--982093cc-280a-49cc-9568-a9c9e6d37943-0',
 'example': False,
 'tool_calls': [],
 'invalid_tool_calls': [],
 'usage_metadata': {'input_tokens': 42,
  'output_tokens': 314,
  'total_tokens': 356}}
```

Listing 10.4 Groq-Modellausführung und -ergebnis

Der wichtigste Output hier ist wieder der Inhalt. Typischerweise greifen wir auf ihn direkt über seine Eigenschaft zu:

```
# %% only print content
print(res.content)
```

```
Hugging Face is a popular open-source library and platform for natural language
processing (NLP) and machine learning (ML) tasks. It was founded in 2016 by
Julien Chaumond, Clement Delangue, and Thomas Wolf. Hugging Face is known for its
Transformers library, which provides pre-trained models and a simple interface
for using and fine-tuning transformer-based models for various NLP tasks, such as
text classification, language translation, question answering, and more.
…
```

Listing 10.5 Groq-Modellergebnis – nur der Inhalt
(Quelle: 03_LLMs/10_model_chat_groq.py)

Wir haben erfolgreich ein Modell von Groq aufgerufen. Lassen Sie uns jetzt anschauen, wo wir mehr Informationen zu den verfügbaren Modellen finden können.

Groq-Modell-Übersicht

Sie können alle von Groq bereitgestellten Modelle unter *https://console.groq.com/docs/models* finden.

Für jedes Modell sind die Informationen zur Modell-ID angegeben. Das ist der String, den Sie in Ihrem Skript verwenden müssen. Außerdem wird der Entwickler angezeigt sowie ein begrenzender Faktor. Bei LLMs ist das das Kontextfenster, und die maximale Anzahl an Tokens wird ebenfalls angezeigt. Mehr dazu finden Sie im nächsten Infokasten. Wenn Sie tiefer in das Modell eintauchen möchten, können Sie sich die Modellkarte anschauen und werden zur Entwicklerseite des Modells weitergeleitet.

Eine Besonderheit sind die *Whisper*-Modelle, die ein Sprach-zu-Text-Modell bereitstellen. Das bedeutet, Sie können eine MP3-Datei (bis zu einer bestimmten Größe) hochladen und erhalten die dazugehörige Transkription.

Die meisten verfügbaren Modelle sind LLMs. Sie sind alle Open-Source- oder Open-Weight-Modelle.

Prominente Modelle hier sind die Mitglieder der *Llama*-Familie (von Meta), *Gemma*-Modelle (von Google), *DeepSeek*-Modelle, *Kimi* (von Moonshot AI) oder *Qwen* (von Alibaba).

Im Modellüberblick haben Sie vielleicht das Kontextfenster als einen der wichtigsten Parameter gesehen. Deshalb lege ich in dem folgenden Infokasten den Fokus auf diesen Parameter.

Kontextfenster

Der *Kontextfenster* bezieht sich auf die maximale Anzahl von Eingabe-Tokens, die ein Modell auf einmal verarbeiten kann. Das ist ein wichtiger Aspekt, denn die Fähigkeit des Modells, relevante Ausgaben zu generieren, hängt davon ab, welche Informationen es in einem einzigen Prompt behalten und nutzen kann. Jedes LLM hat eine feste Grenze, wie viele Tokens es gleichzeitig in seinem Kontextfenster verarbeiten kann.

Was ist aber ein *Token* genau? Ein LLM zerlegt den Eingabetext in kleinere Einheiten, die Tokens genannt werden. So ein Token kann ein Wort sein, nur ein Teil eines Wortes oder sogar nur ein Satzzeichen. Beispiel: Der Satz »Sprachmodelle sind sehr leistungsfähig.« wird in die Token *Sp*, *rach*, *model*, *le*, *sind*, *sehr*, *leistungs*, *fähig* und . unterteilt.

Ein großes Kontextfenster ermöglicht es dem Modell, mehr Informationen zu verarbeiten, was seine Fähigkeit verbessert, längere Texte zu verstehen oder sich lange Gespräche zu »merken«. Wenn die Größe des Kontextfensters zunimmt, steigen aber auch die benötigten Rechenressourcen, um den Text zu verarbeiten. Außerdem erhöht sich die *Latenz* des Modells – das heißt, es dauert länger, bis das Modell eine Antwort liefert.

Wenn das Kontextfenster überschritten wird, »vergisst« das Modell entweder ältere Tokens oder der LLM-Aufruf kann sogar nicht verarbeitet werden. Das hängt von der Implementierung des Pakets ab.

10.1.3 Multimodale Modelle

In diesem Kapitel werden wir hauptsächlich mit Texteingaben und -ausgaben arbeiten und traditionelle große Sprachmodelle verwenden. Aber die Nachfrage nach Modellen, die komplexere und vielfältigere Informationsformen verstehen und mit ihnen interagieren können, ist gestiegen. Deshalb wurden *multimodale Modelle* ent-

wickelt. Diese Modelle sind darauf trainiert, mehrere Arten oder Modalitäten von Eingabe- und Ausgabeformaten zu verstehen und zu generieren. *Modalitäten* sind typischerweise neben Text auch Bilder, Audio und Video.

Im Gegensatz zu traditionellen LLMs, die in einer einzigen Modalität (Text) arbeiten, können diese multimodalen Modelle Informationen in verschiedenen Formaten verarbeiten. Dafür kombinieren sie Fortschritte in der Verarbeitung natürlicher Sprache (NLP, *Natural Language Processing*) mit Innovationen in der Computer-Vision und der Audioverarbeitung.

Dadurch können diese Modelle

- Bilder in Textform analysieren und beschreiben,
- Bilder basierend auf Textbeschreibungen generieren,
- Audiobeiträge transkribieren und
- Audiobeiträge interpretieren und darauf basierende Antworten geben.

Lassen Sie uns ein paar multimodale Modelle verwenden, damit Sie lernen, wie man mit ihnen arbeitet.

10.1.4 Coding: Multimodale Modelle

Wir wollen herausfinden, wie wir ein Bild als Eingabe für ein Modell verwenden können. Dann werden wir mit dem LLM interagieren, um herauszufinden, ob es versteht, was es im Bild »sieht«.

Abbildung 10.2 stellt ein Flussdiagramm dar, das den Prozess des Trainings eines tiefen neuronalen Netzwerks beschreibt, und wir werden dieses Bild an das multimodale Modell weitergeben.

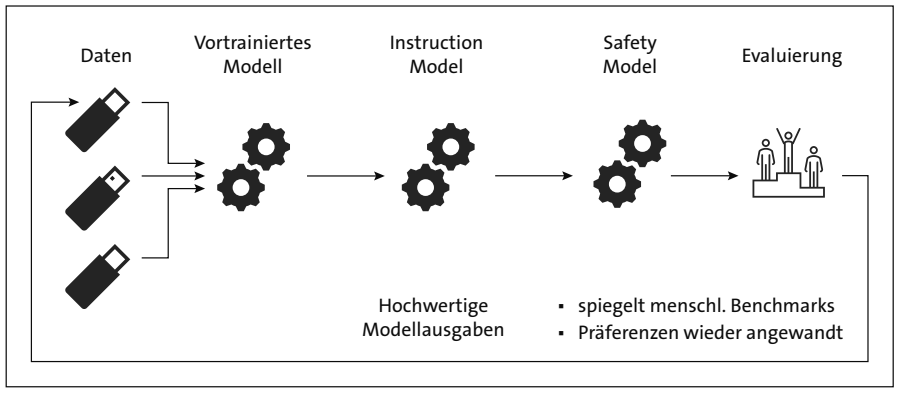


Abbildung 10.2 Trainingsprozess eines Deep-Learning-Netzwerks

Sie finden den Code für dieses Skript unter *100_LLM\30_multimodal.py*. Er basiert hauptsächlich auf der Dokumentation von Groq (*https://console.groq.com/docs/vision*).

In Listing 10.6 beginnen wir mit dem Import der benötigten Pakete:

```
#%% packages
from groq import Groq
from dotenv import load_dotenv, find_dotenv
load_dotenv(find_dotenv(usecwd=True))
import base64
```

Listing 10.6 Multimodales Modell – erforderliche Pakete

Es ist eine gute Praxis, die Konstanten am Anfang des Skripts zu definieren. Hier legen wir fest, welches Modell wir wählen, wo sich das Bild befindet und was die Eingabe des Nutzers ist:

```
MODEL = "meta-llama/llama-4-maverick-17b-128e-instruct"
IMAGE_PATH = "TrainingProcess.png"
USER_PROMPT = "What is shown in this image?
 Answer in a paragraph and in German."
```

Da wir mit einem lokalen Bild arbeiten und es an die API von Groq gesendet werden muss, muss das Bild geladen und in ein Format umgewandelt werden, damit es als Text-String gesendet werden kann. Für diese Funktionalität definieren wir in Listing 10.7 eine Funktion namens `encode_image`. Sie lädt das Bild und wandelt es direkt in das `base64`-Format um. (*Base64* ist ein Binär-zu-Text-Codierungsverfahren, das binäre Daten in ein ASCII-String-Format umwandelt.)

```
#%% Function to encode the image
def encode_image(image_path):
  with open(image_path, "rb") as image_file:
    return base64.b64encode(image_file.read()).decode('utf-8')

base64_image = encode_image(IMAGE_PATH)
```

Listing 10.7 Multimodales Modell – Funktion zur Codierung des Bildes

Jetzt können wir eine `Groq`-Instanz einrichten. Das ist die native Implementierung des `groq`-Pakets, also hat die Chat-Anfrage ein anderes Format im Vergleich zur LangChain-Interaktion mit Modellen. Aber Sie können viele Elemente erkennen, wie zum Beispiel die Nachrichten. Im `messages`-Objekt definieren wir eine Benutzer-Nachricht. Darin übergeben wir ein Dictionary mit Textinhalt – dem Benutzer-Prompt sowie dem Bildinhalt – dem Bild, mit dem wir interagieren wollen:

```
#%% Getting the base64 string
client = Groq()

chat_completion = client.chat.completions.create(
    messages=[
        {
            "role": "user",
            "content": [
                {"type": "text", "text": USER_PROMPT},
                {
                    "type": "image_url",
                    "image_url": {
                        "url": f"{base64_image}",
                    },
                },
            ],
        }
    ],
    model=MODEL,
)
```

Wir haben eine Antwort im Objekt `chat_completion` erhalten, die wir auf dem Bildschirm ausgeben können:

```
#%% analyze the output
print(chat_completion.choices[0].message.content)
```

Das Bild zeigt einen Prozess zur Entwicklung eines Modells, das auf künstlicher Intelligenz basiert. Der Prozess beginnt mit der Datensammlung, die in Form von drei USB-Sticks dargestellt wird, und führt über verschiedene Stufen wie Pre-Trained Model, Instruction Model und Safety Model schließlich zur Evaluation. Jeder Schritt wird durch eine Zahnradgrafik symbolisiert, die die Verarbeitung und Verfeinerung des Modells darstellt.

Listing 10.8 Multimodales Modell – Modellantwort

Das Modell kann wertvolle Antworten liefern. Es versteht, was es sieht. Versuchen Sie doch, die Benutzeranfrage zu ändern, um zu überprüfen, ob das Modell detailliertere Fragen zum Bild beantworten kann.

10.1.5 Coding: Lokales Betreiben von LLMs

Bisher haben wir große Sprachmodelle über API-Aufrufe von Software-as-a-Service-(SaaS-)Anbietern genutzt. Manchmal möchten Sie ein Modell jedoch lokal ausführen.

Das kann notwendig sein, wenn die Privatsphäre wichtig ist und Sie es vermeiden möchten, vertrauliche Informationen über das Internet zu übertragen.

In solchen Fällen können Sie ein Modell auf Ihrem lokalen Computer betreiben. Idealerweise haben Sie hierfür eine leistungsstarke GPU, die ein ordentlich großes Modell ausführen kann. Aber auch ein kleines Modell kann auf Ihrer CPU laufen.

Eine leistungsstarke Plattform, die diesen Prozess sehr einfach macht, ist *Ollama*. Mit Ollama können Sie ein LLM auf Ihrem Laptop oder Desktop-Rechner betreiben, ohne dass Sie eine Internetverbindung benötigen.

Ein alternativer Anbieter ist *LM Studio* (*https://lmstudio.ai/*).

Die lokale Nutzung von Sprachmodellen bietet Privatsphäre und volle Kontrolle, indem sie es Ihnen ermöglicht, direkt auf Ihrer eigenen Hardware mit einem LLM zu interagieren.

Zunächst müssen Sie die Ollama-Software lokal auf Ihrem Rechner installieren. Dafür besuchen Sie, wie in Abbildung 10.3 gezeigt, bitte *https://ollama.com/*.

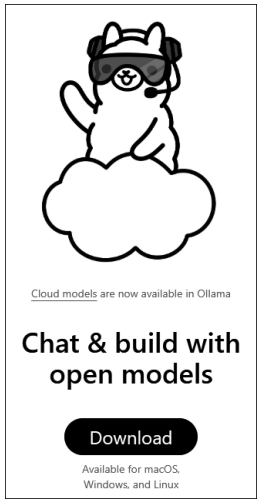


Abbildung 10.3 Download der Ollama-Software

Hier können Sie die Software herunterladen, die zu Ihrem Betriebssystem passt. Sie wird von Ollama für macOS, Linux und Windows angeboten.

Nachdem Sie das gemacht haben, müssen Sie herausfinden, welches Modell für Ihre Hardware und Projektanforderungen geeignet ist. Auf *https://ollama.com/library* finden Sie eine Liste der verfügbaren Modelle.

In diesem Abschnitt werden wir mit `gemma3` arbeiten. Abbildung 10.4 zeigt die `gemma3`-Modellklasse.

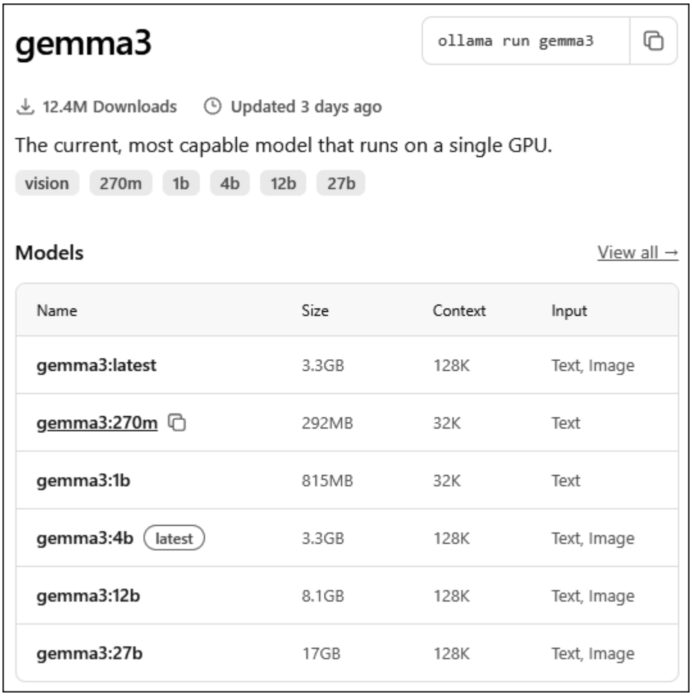


Abbildung 10.4 Ollama – die verwendete Modellklasse »gemma3«

gemma3 wird von Google bereitgestellt. Es ist ein Modell mit relativ wenigen Modellparametern, das aber dennoch sehr leistungsstark ist. Es gibt mehrere verschiedene Varianten: von einem winzigen *270m*- bis hin zu einem großen 27b-Modell. Die Zahlen und Buchstaben beziehen sich auf die Anzahl der Parameter. das heißt, *270m* steht für 270 Millionen Parameter bzw. *27b* für 27 Milliarden (engl. *billons*) Parameter. Wenn Sie auf den Namen des Modells klicken, finden Sie weitere Informationen, z. B. die Dateigröße und den tatsächlichen Namen des Modells.

Eine Besonderheit bei dieser Modellklasse ist, dass sie auch multimodal sein kann. Werfen Sie hierzu einen Blick auf die letzte Spalte. Dort wird ersichtlich, dass Modelle ab 4b sowohl Texte als auch Bilder verarbeiten können.

Wir werden nun das Modell mit dem Namen `gemma3:4b` verwenden.

Listing 10.9 zeigt, wie Sie ein Modell herunterladen können, indem Sie es über Ollama abrufen. Bitte führen Sie in Ihrem Terminal Folgendes aus:

```
ollama pull gemma3:4b
pulling manifest
pulling aeda25e63ebd: 100% |██████████| 3.3 GB
pulling e0a42594d802: 100% |██████████| 358 B
pulling dd084c7d92a3: 100% |██████████| 8.4 KB
```

```
pulling 3116c5225075: 100% ▕████████▏ 77 B
pulling b6ae5839783f: 100% ▕████████▏ 489 B
verifying sha256 digest
writing manifest
success
```

Listing 10.9 Ollama – Modelldownload

Das Modell wurde auf Ihre Festplatte heruntergeladen und ist jetzt verfügbar. Sie können das überprüfen mit:

```
ollama list
NAME           ID           SIZE      MODIFIED
gemma3:4b  a2af6cc3eb7f   3.3 GB   2 minutes ago
```

Der letzte Schritt auf Betriebssystemebene ist, das Python-Paket `langchain-ollama` hinzuzufügen. Sie können es über

```
uv add langchain-ollama
```

hinzufügen oder mittels `pip`:

```
pip install langchain-ollama
```

Damit sind die Vorbereitungen abgeschlossen. Jetzt können wir direkt aus einem Python-Skript mit dem lokalen Modell interagieren.

In unserem Python-Skript, das Sie unter *100_LLMs\40_ollama.py* finden, müssen zuerst die Pakete importiert werden. Wir werden wie im vorherigen Abschnitt das Modell multimodal nutzen und benötigen das Paket `base64` für die Codierung des Bildes. Auf das Modell wird über `ChatOllama` zugegriffen. Die Nutzeranfrage wird als *HumanMessage* übergeben (mehr zu Messages folgt in Abschnitt 10.4).

```
import base64
from langchain_ollama import ChatOllama
from langchain_core.messages import HumanMessage
```

Nun definieren wir zunächst einige Variablen, auf die wir später zugreifen werden. Der Modellname `MODEL_NAME` bezieht sich auf *gemma3*. Die eigentliche Anfrage besteht aus einer Frage im `USER_PROMPT` sowie aus dem Bild, das wir mit seinem Pfad `IMAGE_PATH` übergeben:

```
MODEL_NAME = "gemma3:4b"
USER_PROMPT = "Was zeigt dieses Bild?
 Antworte in einem Absatz und auf Deutsch."
IMAGE_PATH = "TrainingProcess.png"
```

Das Bild kann nicht als Pfad übergeben werden, sondern muss mit `base64` codiert werden. Dafür benutzen wir die Hilfsfunktion `encode_image`, der wir den Pfad zum Bild übergeben und von der wir das codierte Bild zurückerhalten:

```
def encode_image(image_path: str) -> str:
    with open(image_path, "rb") as image_file:
        return base64.b64encode(image_file.read()).decode("utf-8")

base64_image = encode_image(IMAGE_PATH)
```

Listing 10.10 Ollama – Bildcodierung

Die Interaktion mit dem Modell geschieht über eine Modellinstanz `model`, die wir mit `ChatOllama` erstellen:

```
model = ChatOllama(model=MODEL_NAME, temperature=0.2)
```

Die Anfrage, die aus einem Text und einem Bild besteht, muss über ein Message-Objekt übergeben werden. Das Objekt erstellen wir mit `HumanMessage` und es besteht aus dem Text sowie dem Bild:

```
message = HumanMessage(
    content=[
        {"type": "text", "text": USER_PROMPT},
        {"type": "image_url", "image_url": f"{base64_
image}"},
    ]
)
```

Listing 10.11 Ollama – Message-Objekt

Es ist alles vorbereitet, sodass wir jetzt die Daten an das Modell übergeben können. Dazu verwenden wir wieder `invoke`:

```
res = model.invoke([message])
```

Das Schöne an den Modellintegrationen über LangChain ist, dass sie alle dem gleichen Schema folgen. Auf die Modellantwort können wir daher wie in vorherigen Abschnitten über die Eigenschaft `content` zugreifen:

```
res.content
'Das Bild stellt einen Prozess zur Bewertung von Sprachmodellen dar. Es
beginnt mit den Daten, die in ein vortrainiertes Modell eingespeist werden.
Dieses Modell erzeugt dann hochwertige Ausgaben, die auf menschliche
Präferenzen abgestimmt sind. Abschließend werden diese Ausgaben anhand von
Benchmarks bewertet, um die Qualität und Angemessenheit des Modells zu
```

```
beurteilen. Es handelt sich um einen iterativen Prozess, der darauf abzielt,
Modelle zu entwickeln, die nicht nur technisch leistungsfähig, sondern auch
den menschlichen Werten und Erwartungen entsprechen.'
```

Ist das nicht großartig? Sie können ein LLM verwenden, sogar mit vertraulichen Informationen, ohne irgendwelche Daten über das Internet preiszugeben.

Fragen Sie sich nun auch, wie Sie das Verhalten des Modells steuern, ja sogar finetunen können? Dazu müssen Sie sich zunächst mit den Grundlagen und der Theorie befassen. Im folgenden Abschnitt werden Sie die wichtigsten Modellparameter kennenlernen.

10.2 Modellparameter

Es gibt einige sehr wichtige Parameter, die Sie anpassen können, um die Ausgaben zu steuern, die ein Modell erzeugt. Parameter wie *Temperatur*, *Top-p* und *Top-k* spielen eine wichtige Rolle, und mit ihrer Hilfe können Sie die Kreativität, Zufälligkeit und den Fokus der erzeugten Ausgaben steuern.

Modelltemperatur

Mit der *Modelltemperatur* können Sie die Zufälligkeit der Ergebnisse steuern. Typische Werte sind 0 (niedrige Temperatur) und 1 oder sogar darüber (hohe Temperatur):

- **Niedrige Temperaturen** halten das Modell sehr fokussiert: Sie bekommen eher deterministische Ergebnisse, was bedeutet, dass Sie immer wieder die gleiche Antwort erhalten. Das Modell bevorzugt extrem wahrscheinliche Tokens.
- **Hohe Temperaturen** hingegen erhöhen die Zufälligkeit bei der Token-Auswahl. Es wird eine breitere Verteilung von Tokens ausgewählt, was kreativere oder unerwartete Ausgaben ermöglicht. Temperaturen sollten normalerweise den Wert 1 nicht überschreiten, da dies zu chaotischen und inkohärenten Ausgaben führen kann.

Lassen Sie mich das an einem Beispiel verdeutlichen: Stellen Sie sich vor, Sie besitzen eine Eisdiele. Wenn die (Umgebungs-)Temperatur niedrig ist, kommen weniger Kunden, und es könnte eine bessere Geschäftsentscheidung sein, nur die beliebtesten Sorten anzubieten (siehe Abbildung 10.5). Aber wenn die Temperatur steigt, steigt die Nachfrage, und es ist eine gute Entscheidung, auch exotischere Sorten anzubieten.

Die Temperatur ist direkt mit der *Wahrscheinlichkeitsverteilung* der Tokens verbunden. Lassen Sie mich erklären, wie die Temperatur die Wahrscheinlichkeitsverteilung anhand eines künstlichen Beispiels beeinflusst.

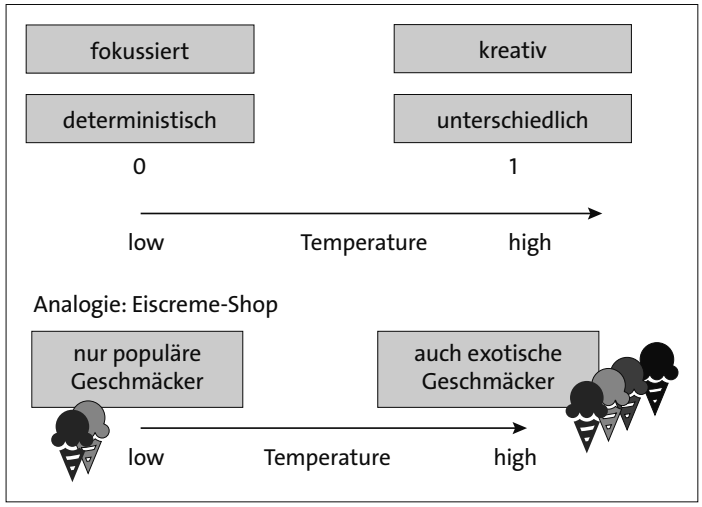


Abbildung 10.5 Modellparameter – Temperatur am Beispiel einer Eisdiele erklärt

Sie haben den Prompt »Bert mag <MASK>.«, und das Modell hat die Aufgabe, das fehlende Wort auszufüllen. Es gibt eine riesige Anzahl möglicher Wörter. Zur Vereinfachung verwenden und zeigen wir nur drei Wörter: »lesen«, »laufen« und »programmieren«.

Das Modell hat eine zugrunde liegende Wahrscheinlichkeit für diese Wörter, die auf seinen Trainingsdaten basiert. Bei sehr niedrigen Temperaturen verstärkt das Modell die Unterschiede zwischen den Wahrscheinlichkeiten. Bei sehr hohen Temperaturen verschwinden diese Unterschiede und alle Wörter haben die gleiche Wahrscheinlichkeit.

Abbildung 10.6 zeigt die Beispiel-Wahrscheinlichkeitsverteilungen für eine gegebene Benutzeranfrage und verschiedene Temperaturwerte.

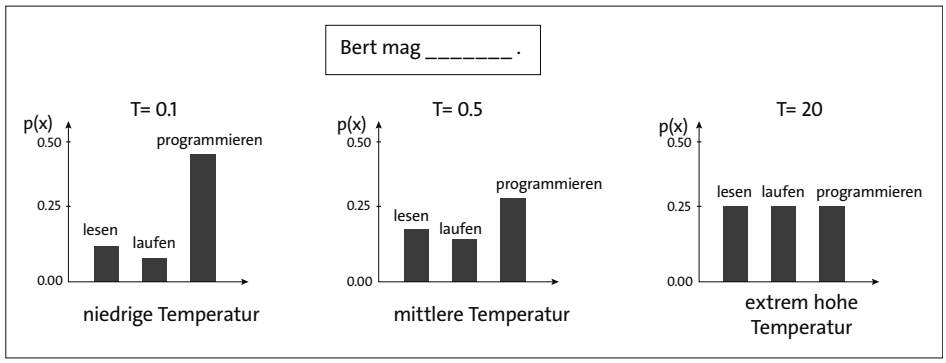


Abbildung 10.6 Modellparameter – Temperatur und Wahrscheinlichkeitsverteilung

Niedrige Temperaturen von zum Beispiel 0.1 sind im Diagramm auf der linken Seite dargestellt. Das mittlere Diagramm zeigt den Einfluss einer mittleren Temperatur auf

die Wahrscheinlichkeitsverteilung, während das Diagramm auf der rechten Seite eine extrem hohe Temperatur darstellt.

Können Sie sehen, wie die Unterschiede mit steigender Temperatur kleiner werden? Genau das bewirkt dieser Parameter. Jetzt schauen wir uns zwei weitere Parameter an, die zusammen mit der Temperatur wirken: *Top-p* und *Top-k*.

Top-p und Top-k

Top-p (auch *Nucleus Sampling* genannt) steuert die Wahrscheinlichkeit, das nächste Token zu berücksichtigen. Das geschieht, indem die Anzahl der möglichen Tokens, aus denen das Modell wählen kann, dynamisch angepasst wird. Nehmen wir ein Beispiel mit `top-p = 0.9`. In diesem Fall berücksichtigt das Modell eine kumulierte Wahrscheinlichkeit von Tokens, die sich auf 90 % summiert, und wählt die kleinste Menge von Tokens innerhalb dieser Grenzen.

Dieser Ansatz balanciert deterministische und kreative Ausgaben. Wenn Sie `top-p = 1` einstellen, gibt es keine Filterung und effektiv werden alle möglichen Tokens berücksichtigt. Wenn Sie einen kleinen Wert wie `top-p < 0.5` definieren, tendieren die Ausgaben des Modells dazu, fokussierter und vorhersehbarer zu sein, da nur die besten Tokens berücksichtigt werden.

Das *Top-k Sampling* steuert, wie viele der wahrscheinlichsten Tokens berücksichtigt werden, wenn das nächste Wort generiert wird. Wenn ein Wert von `top-k = 1` gewählt wird, wird nur das wahrscheinlichste Token ausgewählt.

So erhalten Sie ein völlig deterministisches Ergebnis. Wenn `top-k = 50` ist, sampelt das Modell aus den 50 wahrscheinlichsten Tokens für jeden Schritt. Das erhöht die Vielfalt und ermöglicht kreativere und abwechslungsreichere Ausgaben. Top-k legt eine feste Anzahl von Tokens fest, aus denen gewählt werden kann, unabhängig von der kumulierten Wahrscheinlichkeit, die durch die Top-k-Tokens dargestellt wird.

Abbildung 10.7 zeigt ein Beispiel für Top-p- und Top-k-Parameter.

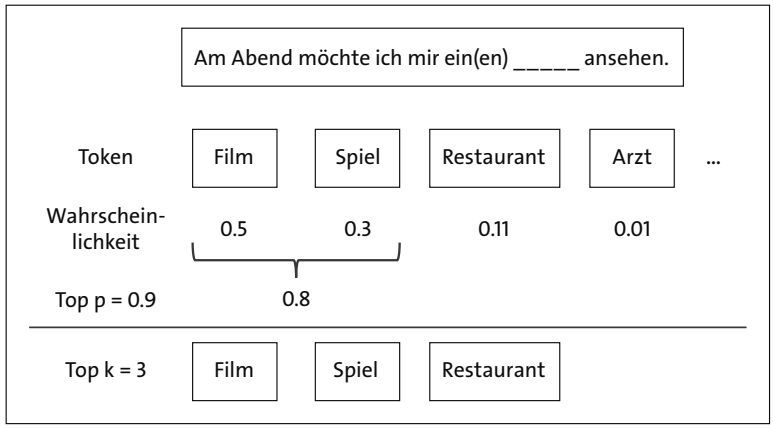


Abbildung 10.7 Modellparameter – Top-p- und Top-k-Parameter

Der Beispielbenutzer-Prompt lautet: »Am Abend möchte ich mir ein(en) <MASK> ansehen«, wobei die Lücke (oder <MASK>) ausgefüllt werden soll. Es gibt wieder mehrere möglichen Tokens.

Basierend auf einer bestimmten Temperatur können die Wahrscheinlichkeiten dieser Tokens berechnet werden. Die Tokens werden in absteigender Reihenfolge dargestellt. Alle Wahrscheinlichkeiten, die zusammen weniger als Top-p ergeben, werden berücksichtigt. In diesem Fall haben »Film« und »Spiel« eine Wahrscheinlichkeit von 80 %. Wenn das nächste Token »Restaurant« hinzugefügt wird, würde die aggregierte Wahrscheinlichkeit 91 % betragen und damit über dem Top-p-Wert liegen.

Top-k ist in diesem Beispiel einfach zu verstehen. Hier ist Top-k auf 3 gesetzt, sodass die drei wahrscheinlichsten Tokens ausgewählt werden, aus denen die endgültige Vorhersage zufällig gewählt wird.

Empfehlungen

Das Gleichgewicht zwischen diesen Parametern hängt vom jeweiligen Einsatzgebiet oder der zu lösenden Aufgabe ab:

- Beim kreativen Schreiben sollten Sie eine höhere Temperatur (0,8 bis 1,0) wählen, kombiniert mit einem moderaten Top-p (0,9 bis 1,0) und Top-k (50 bis 100), um eine Vielzahl kreativer Ausgaben zu erkunden.
- Bei der Code-Generierung möchten Sie eher zuverlässige Code-Schnipsel erhalten und wählen daher eine niedrige Temperatur (0,1 bis 0,3) mit kleinem Top-k (10 bis 20) und Top-p (0,7 bis 0,9), um syntaktisch korrekte Ausgaben sicherzustellen.
- Im Kundenservice oder bei Chatbot-Anwendungen muss die Modellausgabe zuverlässig, fokussiert und konsistent sein. Eine niedrige Temperatur von 0,2 bis 0,4 sorgt für vorhersehbare Antworten. Ein Top-p von 0,7 bis 0,9 ermöglicht es dem Modell, sehr wahrscheinliche Tokens auszuwählen, behält aber etwas Flexibilität, um die Interaktion natürlich zu gestalten. Sie möchten ja robotermäßige Antworten vermeiden. Top-k kann im Bereich von 20 bis 50 liegen; damit bleibt das Modell auf relevante Antworten fokussiert.

In diesem Abschnitt haben Sie gesehen, über welche Parameter die Modellantwort beeinflusst werden kann. Schauen wir uns nun an, auf welche Parameter wir achten sollten, um das richtige Modell auszuwählen.

10.3 Modellauswahl

Im vorherigen Abschnitt haben wir unsere ersten Interaktionen mit verschiedenen LLMs gemacht, und in 10.1 hatte Ihnen eine Auswahl der Modelle vorgestellt. Nun fragen Sie sich vielleicht, wie Sie das »richtige« Modell für Ihre Aufgabe auswählen sollten.

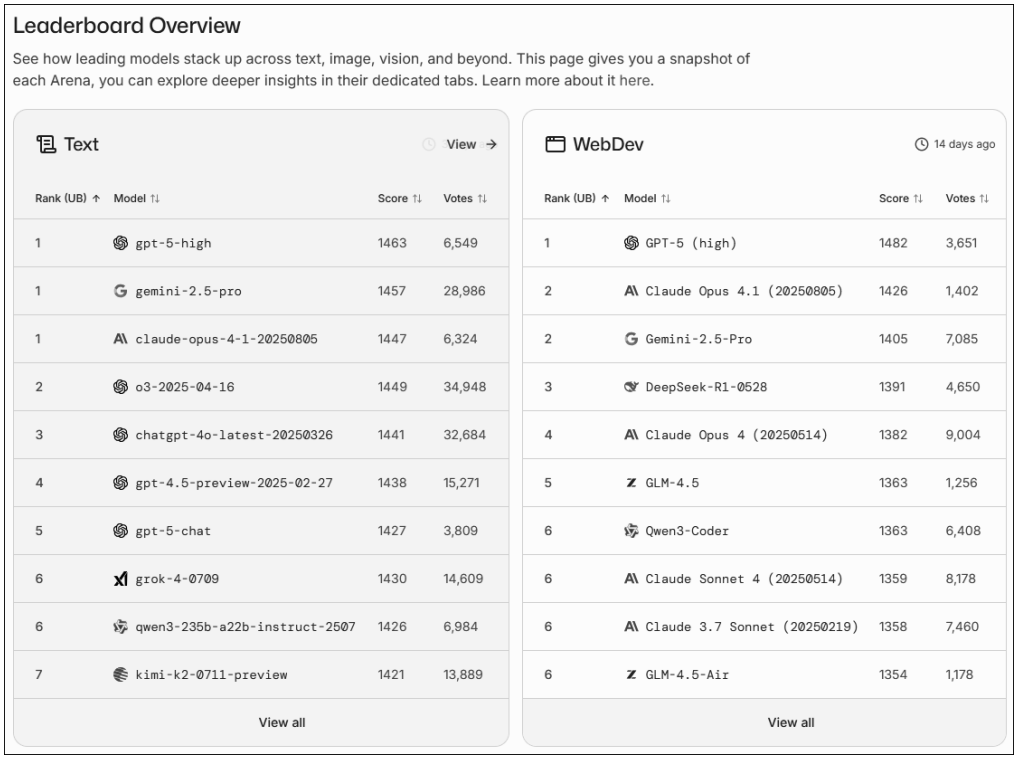
Je nach Ihrem Projekt gibt es harte und weiche Kriterien, die zu berücksichtigen sind:

- Wenn Sie lange Eingabeaufforderungen verarbeiten möchten, ist das *Kontextfenster* (siehe den Infokasten in Abschnitt 10.1.2, »Coding: Nutzung von Groq«) ein entscheidender Faktor.
- Wenn Sie mit einem Modell interagieren möchten, das aktuelle Entwicklungen und Trends berücksichtigt, könnte das Modell-*Cutoff-Date* extrem wichtig sein.
- Weitere wichtige Parameter sind die *Kosten*, die *Latenz* und die *Leistung*.

Lassen Sie uns die Leistung des Modells betrachten und einen genaueren Blick darauf werfen.

10.3.1 Leistungsfähigkeit

Sie können die Leistung verschiedener Modelle in der *LMArena* (*https://lmarena.ai/?leaderboard*) überprüfen. Sie erhalten dann ein Ergebnis, das so wie in Abbildung 10.8 aussieht.



Leaderboard Overview

See how leading models stack up across text, image, vision, and beyond. This page gives you a snapshot of each Arena, you can explore deeper insights in their dedicated tabs. Learn more about it here.

Text — View →

Rank (UB) ↑	Model ↑↓	Score ↑↓	Votes ↑↓
1	gpt-5-high	1463	6,549
1	gemini-2.5-pro	1457	28,986
1	claude-opus-4-1-20250805	1447	6,324
2	o3-2025-04-16	1449	34,948
3	chatgpt-4o-latest-20250326	1441	32,684
4	gpt-4.5-preview-2025-02-27	1438	15,271
5	gpt-5-chat	1427	3,809
6	grok-4-0709	1430	14,609
6	qwen3-235b-a22b-instruct-2507	1426	6,984
7	kimi-k2-0711-preview	1421	13,889

View all

WebDev — 14 days ago

Rank (UB) ↑	Model ↑↓	Score ↑↓	Votes ↑↓
1	GPT-5 (high)	1482	3,651
2	Claude Opus 4.1 (20250805)	1426	1,402
2	Gemini-2.5-Pro	1405	7,085
3	DeepSeek-R1-0528	1391	4,650
4	Claude Opus 4 (20250514)	1382	9,004
5	GLM-4.5	1363	1,256
6	Qwen3-Coder	1363	6,408
6	Claude Sonnet 4 (20250514)	1359	8,178
6	Claude 3.7 Sonnet (20250219)	1358	7,460
6	GLM-4.5-Air	1354	1,178

View all

Abbildung 10.8 Das »LMArena Leaderboard« (Snapshot vom 18. August 2025, Quelle: https://lmarena.ai/leaderboard)

Die Modelle sind nach dem *Arena-Score* sortiert. Aber wie wird dieser Arena-Score bewertet? Er heißt nicht umsonst »Arena« (*https://lmarena.ai/*): In der Arena interagiert der Nutzer mit zwei Modellen – Modell A und Modell B.

Der Nutzer kann einen Prompt definieren und erhält die Antworten von den beiden Modellen, die er dann bewerten muss, um herauszufinden, welches Modell besser abschneidet. So haben wir ein doppelblindes Testsetting, das als Goldstandard in der Bewertung von Testergebnissen gilt. In Abbildung 10.8 sehen Sie, dass mehrere Modelle den gleichen Rang teilen. Das liegt daran, dass das *95-%-Konfidenzintervall* berücksichtigt wird. Die Ränge ändern sich oft, also wird Ihr Ranking wahrscheinlich ganz anders aussehen, je mehr Zeit vergeht.

Oben werden die Hauptkategorien Text und WebDev angezeigt. Aber Sie können auch andere Kategorien auswählen – wie Vision, Text-to-Image, CoPilot oder Search – und das Ranking überprüfen.

Aber die Leistungsfähigkeit ist nicht der einzige relevante Faktor.

10.3.2 Der Wissensstand des Modells

Jedes Modell hat einen »Wissensstichtag« (*Knowledge Cutoff Date*). Das bedeutet, dass die Daten, mit denen das Modell trainiert wurde, an einem bestimmten Datum finalisiert wurden. Deshalb ist es wichtig, das Cutoff-Datum zu kennen: Wenn Sie ein Modell nach Informationen fragen, wie zum Beispiel nach einem Ereignis oder einem anderen Fakt, kann das Modell nicht wissen, ob es nach dem Cutoff-Datum passiert ist.

Für Chatbots wird dieser Parameter immer weniger relevant, da diese Modelle immer häufiger die Fähigkeit haben, im Internet nach aktuellen Informationen zu suchen. Aber für Sie als Entwickler oder Entwicklerin von KI-Systemen könnte das ein wichtiger Faktor sein, den Sie berücksichtigen wollen.

10.3.3 On-Premises vs. Cloud-Hosting

Ein weiterer wichtiger Aspekt bei der Modellauswahl ist der Datenschutz. Wenn Sie mit vertraulichen Informationen arbeiten, möchten Sie oder Ihre Kunden vielleicht nicht, dass die Daten das Unternehmensnetzwerk verlassen. Außerdem ist es wichtig zu wissen, mit welchen Daten das Modell trainiert wurde und ob es DSGVO-kompatibel ist.

Wenn Sie unter Berücksichtigung dieser Parameter ein lokales Modell gewählt haben, können Sie es risikolos im eigenen Netzwerk verwenden und können davon ausgehen, dass Ihre Daten Ihr Netzwerk nicht verlassen.

10.3.4 Open-Source-, Open-Weight- und proprietäre Modelle

Es gibt *proprietäre Modelle*, die den Nutzern über Webanwendungen oder APIs zur Verfügung gestellt werden. Ein bekannter Vertreter dieser Klasse ist Anthropic. Anthropic bietet üblicherweise seine Modelle auf diese Weise an.

Google und OpenAI handhaben das anders: Ihre Modelle werden entweder als proprietäre Modelle über APIs bereitgestellt, z. B. Gemini oder GPT 5. Aber andere Modellklassen wie Gemma oder GPT-OSS werden als *Open-Weight-Modelle* angeboten.

Um ganz korrekt zu sein, sollten wir zwischen *Open Source* und *Open Weight* unterscheiden: Wirkliche Open-Source-Modelle werden mit allen Details wie Modellarchitektur oder verwendeten Trainingsdaten bereitgestellt. Das ist meist nicht der Fall. Der Anbieter veröffentlicht das trainierte Modell mit seinen Gewichten für die Öffentlichkeit, aber spezifische Details zu den zugrunde liegenden Daten und Trainingsdetails bleiben geheim. In so einem Fall spricht man von einem *Open-Weight-Modell*.

Ein bekanntes Beispiel aus dieser Gruppe ist Meta mit seiner Llama-Modellfamilie. Diese Modelle sind kostenlos nutzbar, aber das Unternehmen hält die Details der Trainingsdaten geheim.

10.3.5 Kosten

Die Kosten für die Nutzung eines LLM-Dienstes können ein entscheidender Faktor bei der Auswahl des Modells sein. Typischerweise werden proprietäre Modelle auf Token-Basis abgerechnet. Um genau zu sein: Es wird zwischen Eingabe-Tokens und Ausgabe-Tokens unterschieden. Eingabe-Tokens sind normalerweise günstiger als Ausgabe-Tokens. Die aktuellen Preise für OpenAI-Modelle finden Sie unter *https://openai.com/api/pricing/* und für Anthropic unter *https://www.anthropic.com/pricing#anthropic-api*.

Sie sollten eine Abschätzung vornehmen, wie viele API-Anfragen und wie viele Tokens verarbeitet werden. Basierend darauf können Sie eine Schätzung Ihrer Gesamtkosten erstellen.

10.3.6 Kontextfenster

Ihr Projekt könnte die Verarbeitung von sehr langen Dokumenten beinhalten, und es könnte notwendig sein, so viele Informationen wie möglich an das Modell weiterzugeben. Daher ist das Kontextfenster ein entscheidender Faktor für die beste Wahl des Modells.

Wenn Sie sich zum Beispiel die Modelle auf Groq (*https://console.groq.com/docs/models*) anschauen, finden Sie Modelle mit eher kleinen Kontextfenstern wie *LlaVa 1.5 7B* mit einem Kontextfenster von 4.096 Tokens oder aber *Llama 3.3 70B Versatile* mit einem extrem großen Kontextfenster von 128.000 Tokens.

10.3.7 Latenz

Einige Anwendungsfälle erfordern sehr schnelle Modellantworten. Es gibt eine Abhängigkeit von der Bereitstellung eines Modells bzw. wie lange es bis zur Bereitstellung der Antwort dauert (*Time to First Token*).

Wenn *Latenz* keine Rolle spielt, könnten Sie sogar ein Open-Source-Modell auf einer CPU laufen lassen. In anderen Fällen könnte die Latenz jedoch der wichtigste Faktor sein, z. B. wenn Sie ein LLM mit Sprachgenerierung koppeln möchten, um Echtzeit-Chats zu ermöglichen. In so einer Situation kann ein LLM leicht zum Flaschenhals werden und die Benutzererfahrung beeinträchtigen, weil es keine »natürliche« Konversation gibt, wenn der Gesprächspartner lange Antwortzeiten hat.

10.4 Nachrichtentypen

In Abschnitt 10.1 haben Sie Ihre ersten Schritte mit LLMs gemacht. Wir haben die Modellobjekte aufgerufen, eine einfache Nachricht gesendet und eine Antwort erhalten. In einem realistischeren Chat gibt es verschiedene Arten von Nachrichten. Jede Nachricht hat eine bestimmte Rolle und einen bestimmten Inhalt. Wir schauen uns die häufigsten Nachrichtentypen an.

10.4.1 Benutzereingabe (User- bzw. Human Message)

Dieser Nachrichtentyp bezieht sich auf die menschliche Nachricht und stellt die Eingabe des Nutzers dar. Die Effektivität einer LLM-Antwort hängt von der Klarheit der Nutzer-Nachricht ab. Ein ganzer Arbeitsbereich namens *Prompt-Engineering* beschäftigt sich im Grunde genommen damit, die Nutzer-Nachricht zu optimieren.

10.4.2 Systemnachricht

Neben der Benutzereingabe kann eine *Systemnachricht* definiert werden. Diese legt fest, wie das Modell sich verhalten und arbeiten soll, ähnlich wie in einem Rollenspiel.

- Wenn Sie zum Beispiel einen allgemeinen Assistenten einrichten möchten, könnte eine typische Systemnachricht so aussehen:

  ```
  You are a helpful AI assistant designed to provide accurate, concise, and polite
  responses. Always ensure that your answers are clear and informative.
  ```

- Falls Sie aber möchten, dass Ihr Modell sich wie ein technischer Support-Assistent verhält, könnten Sie das Modell mit der folgenden Systemnachricht anweisen:

  ```
  You are a technical support AI assistant specializing in troubleshooting and ex-
  plaining software-related issues. Respond with clear, step-by-step instructions,
  avoiding technical jargon whenever possible.
  ```

Mit der Systemnachricht definieren Sie also die Rolle des Modells, seinen Ton und spezifische Ziele, bevor die Interaktion mit dem Nutzer beginnt. Die Systemnachricht ist entscheidend, um die Grenzen und Erwartungen des Modells festzulegen. Sie hilft dabei, es so zu lenken, dass es sich im Einklang mit den Anforderungen des Nutzers verhält.

Systemnachrichten haben jedoch ihre Grenzen. Während sie das anfängliche Verhalten prägen können, können sie nicht durchgehend die strikte Einhaltung während des Gesprächs durchsetzen. Das bedeutet, dass Modelle in ihrem Ton oder Verhalten »abdriften« können, wenn sie mit unvorhergesehenen Eingaben des Nutzers konfrontiert werden.

Außerdem können Systemnachrichten allein keine feingranulare Kontrolle über die Genauigkeit der Inhalte oder ethische Überlegungen durchsetzen, ohne ergänzende Leitplanken oder Moderation.

10.4.3 Assistant

Der Nachrichtentyp `assistant` entspricht der Antwort des Modells. Die Haupteigenschaft ist der Inhalt, der die Ausgabe des Modells enthält.

Außerdem gibt es eine Eigenschaft namens `response_metadata`. Diese Eigenschaft enthält einige modellspezifische Ausgaben. Typischerweise werden hier die Token-Nutzung sowie die Dauer der Abfrage angezeigt.

Damit kennen Sie die verfügbaren Nachrichtentypen. Lassen Sie uns nun herausfinden, wie sie in Prompts verwendet werden können. LangChain bietet eine sehr flexible Schnittstelle, um Prompts einzurichten: die Prompt-Templates.

10.5 Prompt-Templates

Bevor wir das LLM aufrufen und eine Anfrage senden, richten wir einen Prompt auf eine einheitliche und strukturierte Weise ein, indem wir die Prompt-Vorlagen (*Prompt-Templates*) des Frameworks *LangChain* verwenden. So können wir das Modell anleiten, wie es handeln soll. Außerdem kann das Modell mithilfe der Prompt-Vorlagen den Nutzer und dessen Absichten besser verstehen.

10.5.1 Coding: ChatPromptTemplates

Die flexibelste Möglichkeit ist die Verwendung von `ChatPromptTemplates`, was es Ihnen erlaubt, eine Liste von Nachrichten zu übergeben. In Listing 10.12 sehen Sie ein Beispiel für den Code, um eine Prompt-Vorlage einzurichten.

Wir beginnen mit einem einfachen Beispiel, das die Idee veranschaulicht. Zuerst müssen wir die Klasse `ChatPromptTemplate` importieren. Im nächsten Schritt erstellen wir eine Instanz dieser Klasse, indem wir die Methode `from_messages()` aufrufen. Diese Nachrichten sind eine Liste von Tupeln. Jedes Tupel hat die Form *`("Nachrichtentyp", "Inhalt")`*. So definieren wir eine Systemnachricht, die dem Modell sagt, wie es sich verhalten soll, gefolgt von einer menschlichen Nachricht, die die eigentliche Benutzeranfrage enthält. Wichtig ist hier, wie wir Variablen definieren, die als Platzhalter eingerichtet und später befüllt werden. In unserem Beispiel sind die Variablen in den Benutzer- oder Menschennachrichten in geschweifte Klammern `{}` gesetzt. Wir richten `input` und `target_language` als Variablen ein.

Obwohl es ein bisschen wie ein Python-f-String aussieht, ist es nicht dasselbe: In einem f-String werden vordefinierte Variablen übergeben und durch die String-Darstellung der Variablen ersetzt. Hier haben wir jedoch keine Variable `input` oder `target_language` im Voraus vordefiniert.

Im letzten Schritt rufen wir die Prompt-Vorlage auf, und in diesem Schritt werden die Variablen durch den tatsächlichen Inhalt ersetzt. Dafür müssen wir einfach die `invoke()`-Methode des `prompt_template`-Objekts aufrufen. Als Parameter wird ein Dictionary übergeben, das Schlüssel verwendet, die den Variablen entsprechen; und die Werte entsprechen dem Inhalt, der verwendet werden soll.

Schließlich wird, da wir den Invoke-Prompt nicht in einer neuen Variablen speichern, die Ausgabe einfach im Terminal angezeigt. Die Prompt-Vorlage wurde in ein `ChatPromptValue`-Objekt umgewandelt, das eine `SystemMessage` und `HumanMessage` enthält:

```
#%% packages
from langchain_core.prompts import ChatPromptTemplate

#%% set up prompt template
prompt_template = ChatPromptTemplate.from_messages([
    ("system", "You are an AI assistant that translates English into another language."),
    ("user", "Translate this sentence: '{input}' into {target_language}"),
])

#%% invoke prompt template
prompt_template.invoke({"input": "I love programming.", "target_language": "German"})
ChatPromptValue(messages=[
  SystemMessage(content='You are an AI assistant that translates English into another
language.'),
  HumanMessage(content="Translate this sentence: 'I love programming.' into German")])
```

Listing 10.12 Prompt-Template – Nutzung

Was ist der Zweck dieses Ansatzes? Mit der Prompt-Vorlage haben wir eine flexible erste Komponente, die an ein Modell übergeben werden kann, um eine Antwort zu erhalten. Der Ansatz »Prompt an das LLM schicken« ist eine einfache Abfolge von Schritten.

Wir werden uns im nächsten Abschnitt mit LangChain-Ketten beschäftigen.

Aber vorher lassen Sie uns die Weisheit der Menge nutzen, um einen guten Prompt zu entwickeln. LangChain hat ein Ökosystem geschaffen, das es Nutzern ermöglicht, Prompts mit dem *LangChain Hub* zu teilen und zu erkunden.

10.5.2 Coding: Verbesserung eines Prompts mit dem LangChain Hub

Sie finden den *LangChain Hub* unter *https://smith.langchain.com/hub*. Dort können Sie sich Prompts anschauen, die von anderen für verschiedene Zwecke erstellt wurden. Von dort holen wir uns Hilfe bei der Erstellung eines Prompts.

Wenn Sie nach »prompt maker« suchen, werden Sie den Prompt `hardkothari/prompt-maker` finden. Dieser Prompt wurde erstellt, um einen detaillierteren Prompt zu generieren. In unserem Beispiel werden Sie herausfinden, wie das funktioniert.

Der Code in Listing 10.13 entspricht der Datei *100_LLM\60_prompt_hub.py*.

Wir müssen die benötigten Pakete laden. Der Neuling hier ist `hub` aus dem `langchain`-Paket:

```
from langchain import hub
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers import StrOutputParser
from dotenv import load_dotenv
load_dotenv('.env')
from pprint import pprint
```

Listing 10.13 Prompt Hub – erforderliche Pakete

Um die Erstellung von Prompts zu nutzen, müssen wir die `pull`-Methode von `hub` aufrufen:

```
#%% fetch prompt
prompt = hub.pull("hardkothari/prompt-maker")
```

Es gibt einige Eingangsvariablen, die über die Eigenschaft `input_variables` zugänglich sind:

```
#%% get input variables
prompt.input_variables
['lazy_prompt', 'task']
```

Nun erstellen wir einen verbesserten Prompt. Wir müssen nur ein Modell einrichten und es in einer Kette ausführen. Das geht schon auf das Wissen des nächsten Abschnitts ein, haben Sie also noch etwas Geduld: Wir werden uns mit Ketten ausführlicher beschäftigen. Nehmen Sie das für den Moment bitte einfach so hin:

```
# %% model instance
model = ChatOpenAI(model="gpt-4o-mini",
                   temperature=0)
# %% chain
chain = prompt | model | StrOutputParser()
```

Listing 10.14 Prompt Hub – Setup der Chain

Wir rufen die Kette in Listing 10.15 auf und übergeben die relevanten Parameter `lazy_prompt` und `task`, um einen verbesserten Prompt zu erhalten:

```
# %% invoke chain
lazy_prompt = "summer, vacation, beach"
task = "Shakespeare poem"
improved_prompt = chain.invoke({"lazy_prompt": lazy_prompt, "task": task})
print(improved_prompt)
```

```
As a skilled poet in the style of William Shakespeare, compose a sonnet that
captures the essence of summer, vacation, and the beach. ### Instructions:
Your poem should reflect the beauty and joy of a summer getaway, using rich
imagery and evocative language typical of Shakespearean verse. Aim for 14
lines, adhering to the traditional iambic pentameter and ABABCDCDEFEFGG rhyme
scheme. ### Context: Incorporate themes of nature, leisure, and the fleeting
nature of time, while evoking a sense of nostalgia and warmth. Use metaphors
and similes to enhance the emotional depth of the poem, and consider including
references to the sun, sea, and the carefree spirit of summer. ### Desired
Outcome: The final piece should resonate with readers, transporting them to a
sun-drenched beach, filled with laughter and the gentle sound of waves, while
also reflecting on the transient beauty of such moments.
```

Listing 10.15 Prompt Hub – Chain-Ausführung

Dieser verbesserte Prompt beschreibt die Rolle des Modells viel detaillierter und gibt umfassende Anweisungen zur Aufgabe sowie eine Beschreibung des gewünschten Ergebnisses. Lassen Sie uns in Listing 10.16 das Modell mit dem verbesserten Prompt ausführen:

```
res = model.invoke(improved_prompt)
print(res.content)
```

```
Upon the golden sands where sunbeams play,
```

```
The azure waves do kiss the shore with glee,
In summer's arms, we cast our cares away,
As laughter dances on the breeze so free.

The sun, a sovereign in the heavens high,
Doth paint the sky with hues of rose and gold,
While seagulls wheel and cry, as if to vie
For joy that in this fleeting time we hold.

O, let us linger in this warm embrace,
Where time, like grains of sand, slips through our hands,
Each moment savored, sweet as summer's grace,
A tapestry of dreams upon the strands.

Yet, as the twilight calls the day to rest,
We cherish memories, our hearts possessed.
```

Listing 10.16 Prompt Hub – Chain-Ergebnis für den verbesserten Prompt

Ich überlasse es Ihnen, das Modell nur basierend auf dem `lazy_prompt` und der Aufgabe `task` auszuführen und die beiden Ergebnisse zu vergleichen. Im Skript finden Sie meine Beispiellösung.

Damit haben Sie ein kleines Beispiel gesehen, wie eine Kette verschiedene Bausteine kombiniert. Dieses Konzept ist sehr mächtig aufgrund seiner Struktur, wie Sie im nächsten Abschnitt sehen werden.

10.6 Chains

Verkettungen (*Chains*) sind ein so wichtiges Konzept, dass sie Teil des Pakets namens `LangChain` sind. Da der englische Begriff geläufiger ist, werde ich im Folgenden das Wort »Chain« anstelle von »Verkettung« nutzen. Eine Chain bezieht sich auf eine Abfolge von Prozessschritten, die miteinander verbunden sind, um eine Aufgabe zu erfüllen. Typischerweise bestehen sie aus mehreren Komponenten.

Wir beginnen mit der kleinsten und einfachsten Kette.

10.6.1 Eine einfache sequenzielle Chain

Die einfachste Chain könnte eine »Prompt zu LLM«-Chain sein, wie sie in Abbildung 10.9 dargestellt ist: Eine Benutzereingabe wird an eine Prompt-Vorlage weitergegeben. Die Prompt-Vorlage selbst gibt ihre Ausgabe an einen LLM-Schritt weiter. Und schließlich erzeugt der LLM-Schritt eine Modellausgabe.

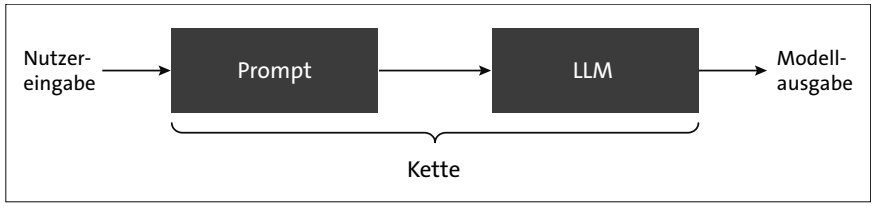


Abbildung 10.9 Einfache LangChain-Verkettung

Aber Sie sind nicht auf eine sequenzielle Chain beschränkt. Sie können auch komplexere Strukturen wie parallel laufende Chains oder Router-Chains nutzen. Diese Konzepte sind in Abbildung 10.10 dargestellt.

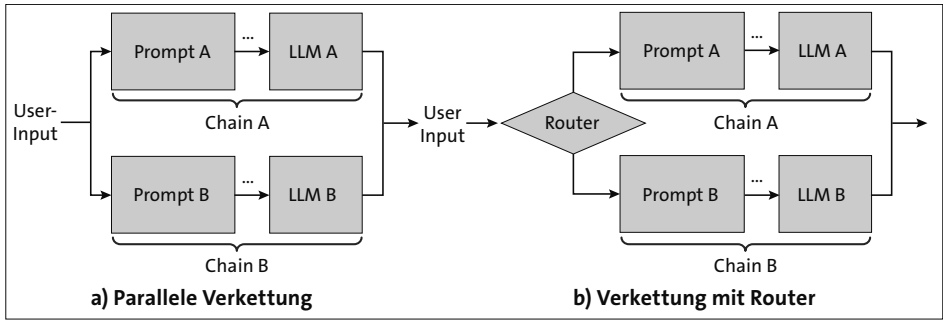


Abbildung 10.10 Komplexere Verkettungen (links: parallel, rechts: Router)

10.6.2 Coding: Eine einfache sequenzielle Chain

Wir werden eine Chain einrichten, die aus einem Prompt-Template besteht. Das Prompt-Template wird an ein LLM übergeben. Nachdem das Modell eine Ausgabe erstellt hat, wird die Ausgabe an einen `StrOutputParser` weitergegeben. Den entsprechenden Code finden Sie in *100_LLM/70_simple_chain.py*.

In Listing 10.17 beginnen wir damit, relevante Pakete und API-Schlüssel zu importieren. Als LLM-Provider verwenden wir OpenAI. Der API-Schlüssel wird über `dotenv` geladen. Bitte stellen Sie sicher, dass eine *.env*-Datei im Arbeitsordner gespeichert ist, die einen Eintrag für `OPENAI_API_KEY` enthält.

```
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from dotenv import load_dotenv
from langchain_core.output_parsers import StrOutputParser
load_dotenv('.env')
```

Listing 10.17 Sequenzielle Chain – erforderliche Pakete

Wir erweitern in Listing 10.18 die Prompt-Vorlage aus Abschnitt 10.5 und erstellen ein `ChatPromptTemplate`, das auf einer System- und User-Message basiert. Die Aufgabe besteht darin, einen Eingabetext `input` in eine Zielsprache `target_language` zu übersetzen:

```
#%% set up prompt template
prompt_template = ChatPromptTemplate.from_messages([
    ("system", "You are an AI assistant that translates English into another
       language."),
    ("user", "Translate this sentence: '{input}' into {target_language}"),
])
```

Listing 10.18 Sequenzielle Chain – Prompt-Template-Erstellung

Die nächste Komponente, die wir anwenden, ist ein LLM. Wir verwenden `GPT-4o-mini` und erstellen die Modellinstanz `model` mit `ChatOpenAI`:

```
model = ChatOpenAI(model="gpt-4o-mini",
                   temperature=0)
```

Das Verbinden der Chain-Elemente kann nicht einfacher sein. Wir müssen nur den Pipe-Operator | anwenden. Die Komponenten werden durch | getrennt. In unserem Beispiel ist der Prompt die erste Komponente in der Kette, gefolgt vom Modell, und anschließend wird die Modellausgabe an den `StrOutputParser` übergeben, der die Modellausgabe in den wahrscheinlichsten String parst:

```
# %% chain
chain = prompt_template | model | StrOutputParser()
```

Alles ist vorbereitet, damit wir die Kette mit Eingabeparametern aufrufen können. Dadurch erhalten wir das endgültige Ergebnis:

```
# %% invoke chain
res = chain.invoke({"input": "I love programming.",
  "target_language": "German"})
res
```

Jetzt, da Sie wissen, wie Sie eine einfache sequenzielle Chain verwenden, könnten Sie sich höhere Ziele setzen und deutlich komplexere Konstrukte wie Router oder parellele Chains aufbauen. Das überlasse ich Ihnen zum Selbststudium. Stattdessen will ich auf eine Fähigkeit der Sprachmodelle eingehen, die extrem wertvoll ist: auf die strukturierten Ausgaben.

10.7 Strukturierte Outputs

10.7.1 Was sind strukturierte Outputs?

Sprachmodelle sind sehr gut darin, freien Text zu generieren, und das ist toll für kreative Geschichten, E-Mails oder auch für die Code-Generierung. Manchmal ist jedoch ein bestimmtes Format erforderlich, etwa für die Datenerfassung, die Automatisierung von Prozessen oder für die Integration in andere Systemen – also eigentlich immer dann, wenn das Sprachmodell nicht am Ende des Prozesses steht, sondern die Ausgabe des Sprachmodells von anderen Tools oder Systemen weiterverwendet werden soll. Genau an dieser Stelle kommen *strukturierte Outputs* ins Spiel.

Ein strukturierter Output ist eine Modellantwort, die nicht als freier Text, sondern in einem bestimmten, maschinenlesbaren Format ausgegeben wird. Die gängigsten Formate sind JSON oder XML, aber auch CSV oder einfache Listen.

Die großen Vorteile von strukturierten Outputs sind:

- die **nahtlose Kommunikation** zwischen Sprachmodellen und anderen Software-Systemen
- Mit strukturierten Outputs lassen sich **Workflows automatisieren**. Zum Beispiel könnte ein LLM Produktdetails aus einem freien Text extrahieren und sie dann in ein E-Commerce-System einspeisen.
- Durch strukturierte Outputs lassen sich **Mehrdeutigkeiten vermeiden**, die bei freiem Text auftreten könnten. Da das Modell gezwungen wird, sich an ein klar definiertes Schema zu halten, werden die Ergebnisse konsistenter.

Schauen wir uns an, wie wir strukturierte Outputs implementieren können.

10.7.2 Coding: Strukturierte Outputs

Waren Sie schon einmal in der Situation, dass Ihnen zwar die Handlung eines Films einfällt, aber Sie sich beim besten Willen nicht an den Titel oder die Darsteller erinnern können? Für dieses Problem bauen wir uns nun eine Chain, wie sie in Abbildung 10.11 zu sehen ist.

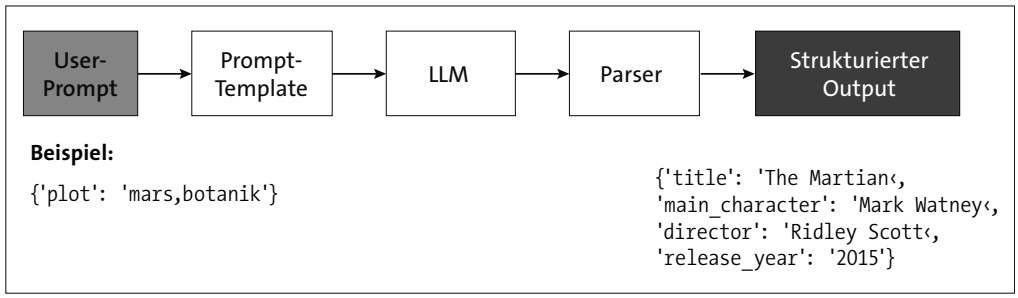


Abbildung 10.11 Chain für strukturierten Output

Der Nutzer übergibt die Rahmenhandlung an die Chain, die aus einem Prompt-Template, einem Modell und einem Output-Parser besteht. Als Ergebnis erhalten wir den strukturierten Output im JSON-Format.

Ein solcher Workflow lässt sich einfach implementieren. Dafür nutzen wir den Code aus *100_LLM\80_structured_outputs.py*.

Wir laden zunächst in Listing 10.19 die benötigten Pakete. Die entscheidende neue Funktionalität kommt über den Parser, der das Ergebnis des Modells nachbearbeitet. Dazu nutzen wir `PydanticOutputParser`. Hiermit in Verbindung steht das Paket `pydantic`, aus dem wir die Klasse `BaseModel` laden:

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_groq import ChatGroq
from dotenv import load_dotenv, find_dotenv
load_dotenv(find_dotenv(usecwd=True))
from langchain_core.output_parsers import PydanticOutputParser
from pydantic import BaseModel
```

Listing 10.19 Strukturierte Outputs – erforderliche Pakete

Bei der Ausgabe erwarten wir ein ganz bestimmtes Format, das wir über eine eigene Klasse `MyMovieOutput` definieren. Diese Klasse erbt von `BaseModel`, und in dieser Klasse definieren wir die Keys und den Datentyp der Values des JSON-Objektes, das uns zurückgegeben werden soll. So wird beispielsweise festgelegt, dass der Titel `title` als String erwartet wird:

```
#%% pydantic model
class MyMovieOutput(BaseModel):
    title: str
    main_character: str
    director: str
    release_year: str
```

Dieses Ausgabeformat wird nun dem `PydanticOutputParser` übergeben. Damit können wir dem Modell klare Anweisungen hinsichtlich des Outputs geben:

```
# %% prompt
parser = PydanticOutputParser(pydantic_object=MyMovieOutput)
```

In den Nachrichten wird das Modellverhalten über den `system`-Prompt festgelegt. Hier werden auch die Formatanweisungen übergeben. Im `user`-Prompt wird dann die eigentliche Benutzeranfrage hinterlegt:

```
messages = [
    ("system", "Du bist ein Filmexperte. {format_instructions}"),
    ("user", "Handlung: {plot}")
]
```

Kommen wir jetzt zum ersten Modul unserer Chain – dem `prompt_template`. Basierend auf den `messages` wird das Prompt-Template erstellt. Neu ist hier die Methode `partial()`, die den Zweck hat, dem Prompt-Template bereits den Parameter `format_instructions` zu übergeben. Wir müssen die Formatanweisungen nicht selbst schreiben, sondern können direkt auf die Anweisungen des `parser` zurückgreifen:

```
prompt_template = ChatPromptTemplate.from_messages(messages).partial(
    format_instructions=parser.get_format_instructions()
)
```

Die Modellinstanz wird ganz klassisch mit `ChatGroq` erzeugt. Sinnvoll ist es, dem Modell eine niedrige Temperatur mitzugeben.

Temperatur bei strukturierten Outputs

Ein wichtiger Hinweis an dieser Stelle: Bei strukturierten Outputs soll das Modell wenig kreativ sein, sondern eher deterministisch arbeiten. Daher sollte die Temperatur gering eingestellt werden, zum Beispiel auf 0 bis 0.3.

```
MODEL_NAME = "meta-llama/llama-4-scout-17b-16e-instruct"
model = ChatGroq(model=MODEL_NAME, temperature=0.2)
```

Jetzt können wir die `chain` erstellen, die aus einer Sequenz aus `prompt_template`, `model` und `parser` besteht:

```
chain = prompt_template | model | parser
```

Damit ist alles für unseren ersten Test vorbereitet. Die Eingabe wird im Objekt `chain_inputs` als Dictionary definiert und der `chain` mittels `invoke` übergeben:

```
chain_inputs = {"plot": "mars, botanik"}
res = chain.invoke(chain_inputs)
```

Das Ergebnis können wir in Listing 10.20 über die Methode `model_dump()` abgreifen:

```
res.model_dump()
```

```
{'title': 'The Martian',
 'main_character': 'Mark Watney',
 'director': 'Ridley Scott',
 'release_year': '2015'}
```

Listing 10.20 Strukturierte Outputs – Modellergebnis

Das Ergebnis ist wie gewünscht ein JSON-Objekt mit den genannten Keys und den korrekten Werten.

Es gäbe noch unendlich mehr zum Thema »Sprachmodelle und allgemeine generative KI« zu berichten. An dieser Stelle möchte ich daher auf mein Buch »Generative KI mit Python« verweisen, das sich diesen Themen sehr ausführlich widmet.

Das Thema der großen Sprachmodelle ist technologisch untrennbar mit der Transformer-Architektur verbunden. Im folgenden Abschnitt wagen wir einen technischen Deep Dive in diese Technologie.

10.8 Deep Dive: Wie funktionieren Transformer?

Abbildung 10.12 verdeutlicht den grundsätzlichen Aufbau eines *Transformer*-Modells. Ein Transformer-Modell besteht aus einigen fundamentalen Bausteinen, die Sie besser verstehen müssen.

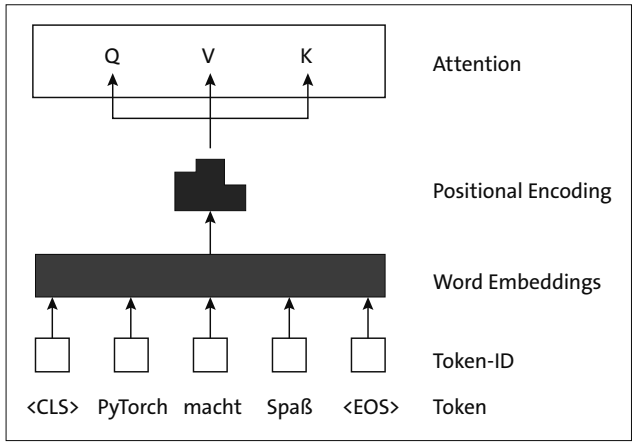


Abbildung 10.12 Aufbau eines Transformer-Modells

Auf der untersten Ebene steht der Text, der dem Modell übergeben wird. In unserem Beispiel wird der Satz »PyTorch macht Spaß« an das Modell übergeben. Dabei wird der Text zunächst tokenisiert, also in einzelne Teile zerlegt (mehr dazu erfahren Sie in Abschnitt 10.8.1).

Der Einfachheit halber nehmen wir an dieser Stelle an, dass der Text in einzelne Wörter zerlegt wurde. Die Wörter werden durch ein erstes Netzwerkmodul geleitet, das *Word Embeddings* ermittelt. Diese Word Embeddings sind Vektoren, die für jedes Wort die inhaltliche Bedeutung widerspiegeln. Weitere Details hierzu lesen Sie in Abschnitt 10.8.2.

Der nächste Schritt ist das *Positional Encoding*. Stellen Sie sich den Satz »Die Katze fängt die Maus« vor. Dass die Reihenfolge der Wörter eine Rolle spielt, zeigt der Satz: »Die Maus fängt die Katze«. Es werden exakt die gleichen Wörter verwendet, aber die

Bedeutung hat sich durch die geänderte Position der Wörter komplett verändert. Mit dem Positional Encoding bekommt das Modell die Information, welches Wort an welcher Stelle zu finden ist.

Im nächsten Schritt werden die Daten in ein Modul geführt, das sich *Attention* nennt. Das ist das eigentliche Kernstück des Modells. Hier lernt das Modell die innere Struktur des Satzes und welches Wort mit welchem anderen Wort zusammenhängt. Mit diesem Konzept werden wir uns in Abschnitt 10.8.4 näher beschäftigen.

10.8.1 Tokenisierung

Tokenisierung bedeutet einfach nur, dass der Text in kleinere Einheiten, sogenannte *Tokens*, zerlegt wird. Die Tokens können einzelne Wörter, Satzteile oder sogar einzelne Zeichen sein. Das hängt von der verwendeten Tokenisierungsmethode ab. Bei heutzutage üblichen Sprachmodellen kommt eine *Subword-Tokenisierung* zum Einsatz.

Jedes Token wird dann einem eindeutigen numerischen Wert, der sogenannten *Token-ID*, zugewiesen. Diese ID ist so eine Art Wörterbucheintrag, wobei dieses Wörterbuch eine Zuordnung zwischen menschlichen Wörtern und zugeordneten Zahlenwerten ermöglicht.

Abbildung 10.13 zeigt die Funktionsweise des Tokenizers. Er verarbeitet eine Texteingabe des Nutzers, extrahiert die einzelnen Tokens und liefert die Tokens mit ihren dazugehörigen Token-IDs zurück.

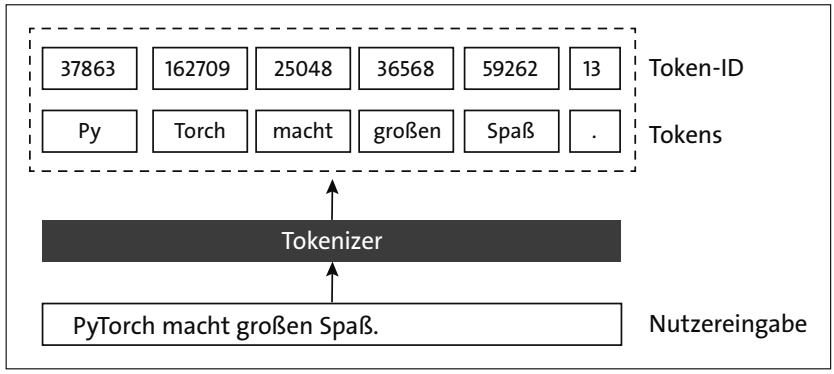


Abbildung 10.13 Tokenisierung

Im Beispiel wurde der Tokenizer von GPT-4o und GPT-4o mini verwendet. Sie können das selbst mit anderen Eingaben testen, wenn Sie den Tokenizer von OpenAI unter *https://platform.openai.com/tokenizer* verwenden.

Wie Sie sehen, beruht dieser Tokenizer auf Subword Tokenization. Das ist daran zu erkennen, dass das Wort »PyTorch« in die beiden Tokens »Py« und »Torch« zerlegt

wurde. Außerdem ist es wichtig zu wissen, dass auch für Sonderzeichen separate Tokens erstellt werden. In unserem Beispiel wurde der Punkt als eigenes Token codiert.

In der Praxis wird immer von »Tokens« gesprochen, wobei für uns eher das Konzept »Wort« verständlicher ist. Es ist für mich einfach, die Wörter in diesem Kapitel zählen zu lassen, aber wie vielen Tokens entspricht das? Damit befassen wir uns im folgenden Infokasten.

Umrechnung von Wörtern in Tokens

Die Umrechnung von Wörtern in Tokens ist keine exakte Wissenschaft, da sie von mehreren Faktoren abhängt.

Es gibt aber gute Heuristiken, die man heranziehen kann. Sie unterscheiden sich von Sprache zu Sprache:

- Im Englischen ist die gängige Faustformel, dass 1 Token ungefähr ¾ eines Wortes entspricht. Vereinfacht gesagt bedeutet das, dass 75 Wörter durch 100 Tokens beschrieben werden können.
- Im Deutschen ist das Verhältnis ungünstiger. Eine Faustformel lautet, dass 1 Wort im Durchschnitt in 2.1 Token umgesetzt wird. Das liegt an Eigenheiten der deutschen Sprache, zum Beispiel daran, dass viele Wörter zusammengesetzte Wörter sind. Es liegt aber auch daran, dass die meisten Sprachmodelle sich am englischen Sprachschatz orientieren, was dazu führt, dass viele gängige englische Wörter als einzelne Tokens behandelt werden.

Es gibt nicht nur ein solches »Wörterbuch«. Stattdessen müssen Sie als Entwickler bzw. Entwicklerin darauf achten, dass der zum Modell passende Tokenizer verwendet wird. Häufig wird dieser Schritt vor dem Nutzer verborgen, sodass er sich nicht darum kümmern muss, aber das ist nicht immer der Fall.

Da Deep-Learning-Modelle nur mit Tensoren und allgemeiner mit numerischen Werten arbeiten können, findet dieser Schritt bei allen Sprachmodellen statt. Die Nutzereingabe wird tokenisiert, vom Modell verarbeitet, und die Modellausgabe wird wieder in menschliche Sprache zurückgewandelt.

Kommen wir nun zum nächsten wichtigen Aspekt des Transformer-Modells: den Word Embeddings.

10.8.2 Word Embeddings

Wenn Sie an Ihren Mathematikunterricht zurückdenken, erinnern Sie sich vielleicht, was ein Vektor ist – die Linie mit einem Pfeil, die Sie in ein Koordinatensystem zeichnen mussten. Damals mussten Sie diese Linien zeichnen, die durch zwei Zahlen dargestellt werden konnten.

Ein ganz einfaches Beispiel sehen Sie in Abbildung 10.14. Dieses Diagramm zeigt Lebewesen und ihre Position in einem zweidimensionalen Diagramm, in dem die beiden Dimensionen die Anzahl der Beine und die Körpergröße darstellen. Indem wir die Informationen auf diese Weise darstellen, lernen wir etwas über die Welt und die semantische Bedeutung von Wörtern, zum Beispiel, dass sich Katzen und Hunde in Bezug auf diese beiden Eigenschaften ziemlich ähnlich sind.

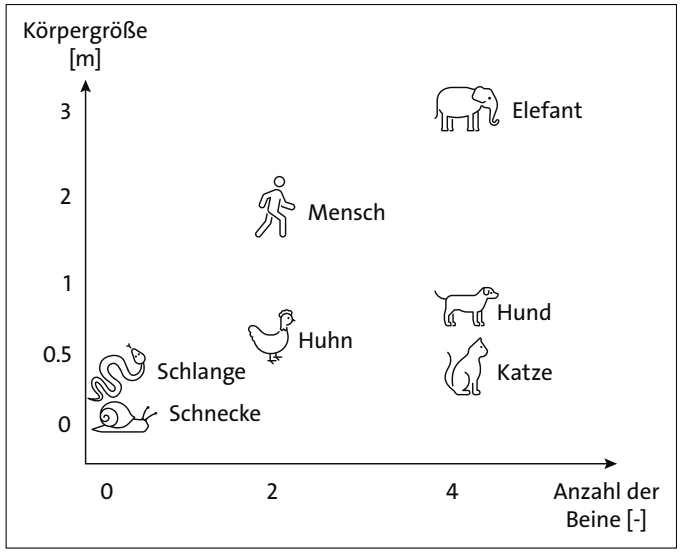


Abbildung 10.14 Beispiel für einen vereinfachten Vektorraum

Wir Menschen können uns einen Punkt in einem zweidimensionalen Raum vorstellen (wie im Beispiel gezeigt) oder in einem dreidimensionalen Raum. Stellen Sie sich vor, wir fügen eine dritte Dimension wie *Intelligenz* in dieses Diagramm ein.

Aber wir können uns keinen 1536- oder 3072-dimensionalen Raum vorstellen. Computer können das problemlos. Und das ist hilfreich, denn die semantische Bedeutung von Wörtern, Bildern, Geräuschen oder jeder anderen Art von Informationen kann in einer höheren Dimension dargestellt werden. Der entscheidende Aspekt hier ist, dass ähnliche Konzepte näher beieinander liegen als Konzepte, die sehr unterschiedlich sind. Wie in unserem Beispiel, in dem wir sehen können, dass Katzen und Hunde oder Schlange und Schnecke vergleichbar sind – je nach den Eigenschaften, die wir ausgewählt haben. Mit jeder zusätzlichen Dimension lernt der Computeralgorithmus, die Bedeutung eines Wortes besser zu verstehen.

Wir verwenden Sprachen wie Englisch oder Deutsch, um zu kommunizieren und ein Konzept zu klären. Computer arbeiten nicht direkt mit unseren Sprachen, sondern mit ihrem Äquivalent – numerischen, sogenannten *Einbettungsvektoren* (*Embedding Vectors*).

In unserem Beispiel wird für den Computeralgorithmus z. B. ein Hund durch [4, 1] definiert, eine Katze durch [4, 0.5] und ein Mensch durch [4, 2]. Eine Beispielzuordnung ist in Tabelle 10.1 dargestellt.

Menschliches Konzept	Einbettungsvektor
Hund	[4, 1]
Katze	[4, 0.5]
Mensch	[4, 2]
Elefant	[4, 3]
Schlange	[0, 0.5]
Schnecke	[0, 0.1]

Tabelle 10.1 Menschliche Konzepte und ihre Computer-Entsprechung

Und der Prozess, menschliche Texte in Vektoren zu übersetzen, wird *Embedding* genannt. Es gibt verschiedene Arten der Embeddings. Es können einzelne Wörter, aber auch ganze Textpassagen durch jeweils einen Embedding-Vektor repräsentiert werden.

Zum jetzigen Zeitpunkt versteht das Transformer-Modell bereits einzelne Wörter, aber was passiert, wenn sich die Position der Wörter verändert? Hier kommt das Positional Encoding ins Spiel.

10.8.3 Positional Encoding

Eine entscheidende Technik, um die Reihenfolge der Wörter zu erhalten, ist das *Positional Encoding*.

In der Praxis wird den Word Embeddings ein weiterer Vektor hinzugefügt, der die Informationen über die absolute oder relative Position des Tokens in der Eingabesequenz beinhaltet.

Die mathematische Implementierung verwendet trigonometrische Funktionen wie Sinus- oder Kosinus. Dies ermöglicht es dem Modell, unabhängig von der Länge der Eingabesequenz konsistente Positionsmuster zu erkennen und die Positionsinformationen auch auf Sequenzen zu übertragen, die viel länger sind als die im Training verwendeten.

10.8.4 Attention

Attention wird hier am Beispiel der *Self-Attention* erläutert. Die Self-Attention berechnet die Ähnlichkeit der Word Embeddings zwischen allen Wörtern und sich selbst.

Abbildung 10.15 zeigt den Prozess am Beispiel des Satzes »The man ate the pizza because it smelled delicious«. Wenn wir uns hier nur auf das Wort »it« konzentrieren – worauf bezieht es sich? Mit einem gewissen Grundverständnis von Sprache könnte man einem Algorithmus beibringen, dass es sich auf ein Substantiv bezieht. Aber in diesem Fall könnte es sich auf »man« oder »pizza« beziehen. Genau hier kommt das Konzept der Self-Attention zum Tragen. Damit ist es einem Modell möglich, den Zusammenhang zwischen Wörtern zu »verstehen«. Wenn es auf genügend Daten trainiert wurde, sollte es dem Modell deutlich werden, dass sich »it« sehr viel wahrscheinlicher auf »pizza« bezieht.

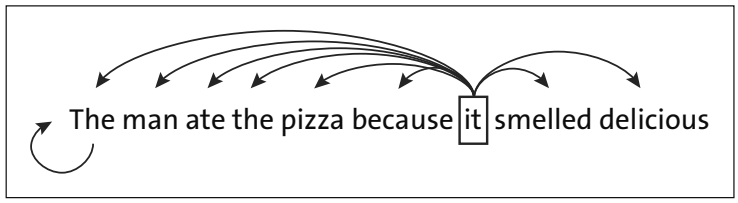


Abbildung 10.15 Der Self-Attention-Prozess

Diese Funktionsweise schauen wir uns an einem praktischen Beispiel an und überprüfen sie anhand des Skriptes unter *100_LLM\self_attention.py*.

Wir laden zu Beginn des Skriptes in Listing 10.21 die erforderlichen Pakete. Zur Verarbeitung der Modelle brauchen wir `torch` bzw. `AutoTokenizer` und `AutoModel`. Die Ergebnisse werden mit `matplotlib` und `seaborn` visualisiert:

```
#%% packages
import torch
from transformers import AutoTokenizer, AutoModel
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

Listing 10.21 Self-Attention – Paketimport

In Listing 10.22 übergeben wir den Beispieltext `sample_sentence` an das Modell, nachdem zuvor der Tokenizer erstellt wurde. Wie schon erwähnt muss der Tokenizer zum Modell passen. Hier werden das Modell (und der Tokenizer) `bert-base-uncased` verwendet. Die `inputs` sind die Token-IDs, die mit dem Tokenizer erstellt wurden. Die Word Embeddings ergeben sich hingegen aus der letzten versteckten Schicht namens `output.last_hidden_state`:

```
#%% test
sample_sentence = "the man ate the pizza because it smelled delicious"
```

```
#%% Get word encodings and attention weights from BERT
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
model = AutoModel.from_pretrained("bert-base-uncased", output_attentions=True)
inputs = tokenizer(sample_sentence, return_tensors="pt",
  padding=True, truncation=True)

with torch.no_grad():
    outputs = model(**inputs)
word_encodings = outputs.last_hidden_state
tokens = tokenizer.convert_ids_to_tokens(inputs['input_ids'][0])
```

Listing 10.22 Self-Attention – Erstellung der Word Embeddings und Tokens

Nun können die Attention-Gewichte ebenso aus den Modellergebnissen extrahiert werden. Die durchschnittliche Attention `avg_attention` wird ermittelt:

```
#%% Get attention weights from all layers and heads
attention_weights = outputs.attentions

last_layer_attention = attention_weights[-1][0]

avg_attention = last_layer_attention.mean(dim=0)
```

Abbildung 10.16 zeigt das Ergebnis des Modells. Die Zahlen repräsentieren die Attention-Gewichte ausgehend vom Wort »it«.

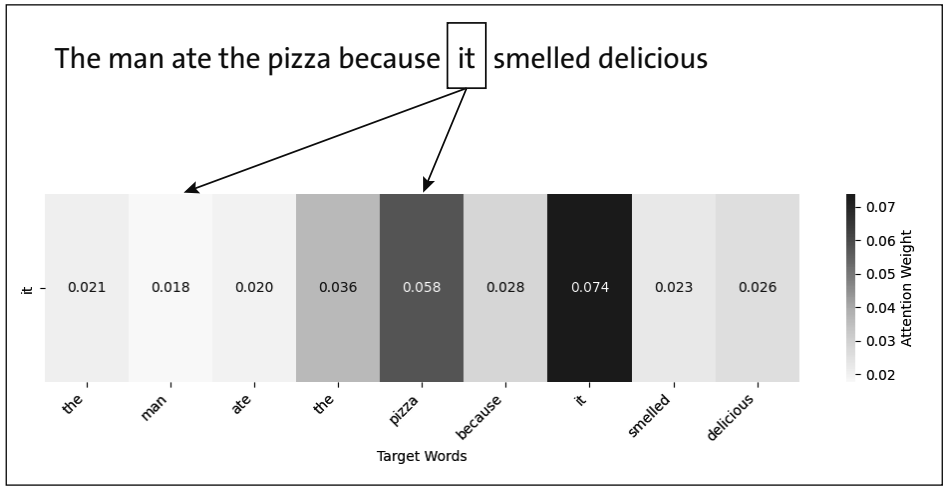


Abbildung 10.16 Self-Attention von »it« in Bezug auf »man« und »pizza«

Es wird deutlich, dass sich »it« am ehesten auf »pizza« bezieht.

In einem realen Transformer gibt es nicht nur ein Attention-Modul (man spricht in der Regel von einem Attention-*Head*), sondern *Multi-Head Attention*.

Bei der Multi-Head Attention fokussiert sich jeder Head auf verschiedene Aspekte von Zusammenhängen zwischen den Wörtern. So kann es zum Beispiel sein, dass sich ein Head auf den Zusammenhang zwischen Verb und Substantiven oder ein anderer Head auf Adjektive und Substantive fokussiert.

10.9 Zusammenfassung

In diesem Kapitel haben Sie die Grundlagen von Sprachmodellen kennengelernt: von ihrer einfachen Nutzung mit Python über die Bedeutung von Modellparametern bis hin zu verschiedenen Kriterien der Modellauswahl – darunter Leistung, Cutoff-Date, Hosting- und Deployment-Optionen.

Wir haben uns mit Nachrichtentypen, Prompt-Templates und Chains befasst, um effektive Interaktionen zu gestalten.

Ein besonderer Fokus lag auf strukturierten Outputs, die es uns ermöglichen, spezifische und vorhersehbare Ergebnisse zu erhalten.

Abschließend erhielten Sie einen tieferen Einblick in die Transformer-Architektur, der Ihnen ein besseres Verständnis der Funktionsweise dieser mächtigen Modelle vermittelt hat. Mit diesem Wissen sind Sie nun bestens gerüstet, um Sprachmodelle in Ihren eigenen Projekten zu verwenden.

Auf einen Blick

Inhalt

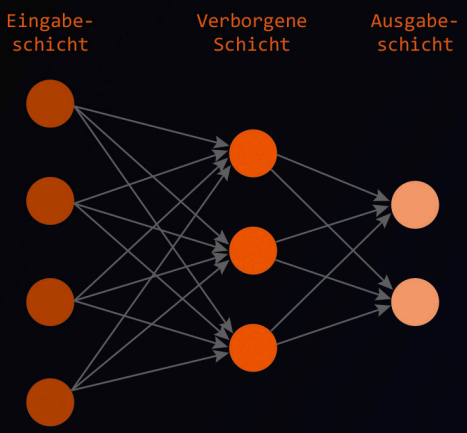
PyTorch

Professionelle Programmierung von KI-Modellen

PyTorch ist das meistgenutzte Deep-Learning-Framework in der Praxis. Lernen Sie mithilfe spezifischer Aufgaben aus realen Projektszenarien, wie Sie PyTorch richtig einsetzen und KI-Modelle professionell trainieren, optimieren und produktiv einsetzen können.

Bert Gollnick zeigt Ihnen in diesem Buch alle einschlägigen Verfahren inklusive Python-Implementierung: von linearer Regression über große Sprachmodelle bis zur Kombination mehrerer Verfahren. Profitieren Sie von umfassenden Codebeispielen und entdecken Sie das Zusammenspiel mit PyTorch Lightning, HuggingFace und weiteren Tools.

+ Installation und Grundkonzepte
+ Deep Learning und Datenaufbereitung
+ Immer das richtige Vorgehen für Klassifizierung, Zeitreihen-Vorhersage, Computer Vision u. v. m.
+ Große Sprachmodelle mit HuggingFace
+ Verwendung vortrainierter Modelle
+ PyTorch Lightning
+ Modell-Evaluierung
+ Der vollständige ML-Workflow: TensorBoard, FlowML, WandB, Heroku und mehr



Bert Gollnick ist Senior Data Scientist mit 17 Jahren Erfahrung in der Industrie und einem Schwerpunkt auf künstlicher Intelligenz. Er unterrichtet Data Science und Machine Learning, insbesondere generative KI und Verarbeitung natürlicher Sprache (NLP).

Rheinwerk Computing

Gedruckt in Deutschland
Mineralölfreie Druckfarben
Zertifiziertes Papier

€ 39,90 [D] € 41,10 [A]

Programmierung/KI
ISBN 978-3-367-11129-9