

O'REILLY®

Vorwort von
Andrew Ng



PyTorch für KI und ML

Das Praxisbuch für generative KI
und Machine Learning

Laurence Moroney
Übersetzung von Jørgen W. Lang

Daten mit PyTorch verwenden

In den ersten drei Kapiteln dieses Buchs haben Sie Modelle mit verschiedenen Daten trainiert – vom Fashion-MNIST-Datensatz, der bequem über eine API zugänglich war, bis zu den bildbasierten Datensätzen *Horses or Humans* sowie *Dogs vs. Cats*, die als ZIP-Dateien vorlagen und zunächst heruntergeladen und vorbereitet werden mussten. Sie wissen also, dass es verschiedene Wege gibt, an Trainingsdaten für ein Modell zu kommen.

Für viele öffentliche Datensätze müssen Sie sich aber zunächst eine Reihe spezieller Fähigkeiten aneignen, bevor Sie sich überhaupt um Ihre Modellarchitektur kümmern können. Die Idee hinter den Anwendungsbereichen und Werkzeugen im `torch.utils.data.DataSets`-Namensraum besteht darin, Datensätze auf eine leicht verwendbare Weise bereitzustellen. Dabei werden die vorbereiteten Daten über PyTorch-freundliche APIs zugänglich gemacht.

Einen kurzen Einblick in diese Vorgehensweise haben Sie bekommen, als wir in Kapitel 2 mit dem Fashion-MNIST-Datensatz gearbeitet haben. Zur Erinnerung hier noch einmal der Code zum Laden der Daten:

```
train_dataset = datasets.FashionMNIST(root='./data', train=True,
                                     download=True, transform=transform)
```

Für diesen Datensatz haben wir außerdem einen Import aus der `torchvision`-Bibliothek durchgeführt, um das `datasets`-Objekt für Fashion MNIST zu erhalten.

```
from torchvision import datasets
```

Da es sich hier um einen Datensatz für Computer Vision handelt, ist es naheliegend, dass er sich in der `torchvision`-Bibliothek befindet.

Darüber hinaus verfügt PyTorch über viele andere Datensätze mit ganz unterschiedlichen Datentypen, die auf die gleiche Weise geladen werden können, zum Beispiel:

Computer Vision

Hierzu gehört der gerade erwähnte Fashion-MNIST-Datensatz aus der `torchvision`-Bibliothek. Diese enthält eine Reihe von »Bildklassifizierungsdatensätzen« für Szenarien wie Bilderkennung, Segmentierung, optischen Fluss, Stereoabgleich, Bildpaarzuordnung, Bildbeschreibung, Videoklassifizierung, Videovorhersage und mehr.

Text

In der torchtext-Bibliothek finden Sie gängige Textdatensätze. Es würde zu viel Platz benötigen, sie alle hier aufzulisten. Es gibt Datensätze für Textklassifizierung, Sprachmodellierung, maschinelle Übersetzung, Sequenz-Tagging, Frage-Antwort-Systeme und nicht überwachtes Lernen und nicht überwachtes Lernen (*Unsupervised Learning*). Weitere Details finden Sie in der PyTorch-Dokumentation (<https://oreil.ly/aFamN>). Neben den Datensätzen selbst enthält diese Bibliothek viele Hilfsfunktionen für den Einsatz bei der Textbearbeitung.

Audio

Die torchaudio-Bibliothek enthält Datensätze, die in Szenarien für maschinelles Lernen (*Machine Learning*) von Klängen und Sprache genutzt werden können. Auch hierzu finden Sie die Details in der PyTorch-Dokumentation (<https://oreil.ly/tvDe4>).

Alle diese Datensätze sind Unterklassen von `torch.utils.data.Dataset`. Es lohnt sich also, sich diese Bibliothek anzusehen und zu verstehen. Das wird Ihnen nicht nur bei der Einbindung bereits vorhandener Datensätze helfen, sondern auch bei der Erstellung eigener Sammlungen, die Sie mit anderen teilen möchten.

Einstieg in Datasets

`torch.utils.data.Dataset` ist eine abstrakte Klasse, die einen Datensatz repräsentiert. Um einen eigenen Datensatz zu erstellen, müssen Sie eine Unterklasse anlegen, die folgende Methoden implementiert:

```
__len__(self)
```

Diese Methode sollte die Gesamtzahl der Elemente eines Datensatzes zurückgeben.

```
__getitem__(self, index)
```

Diese Methode sollte ein einzelnes Element Ihres Datensatzes am angegebenen `index` zurückgeben. Vor der Weitergabe an das Modell wird dieses Element transformiert.

Hier ein Beispiel:

```
from torch.utils.data import Dataset

class CustomDataset(Dataset):
    def __init__(self, data, transforms=None):
        self.data = data
        self.transforms = transforms

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

    def __getitem__(self, idx):
        sample = self.data[idx]
        if self.transforms:
            sample = self.transforms(sample)
        return sample
```

Und das ist im Grunde auch schon alles. Die Daten selbst liegen als Array namens `data()` vor. Wenn Sie jetzt einen Datensatz anlegen wollen, in dem zwischen einem x - und einem y -Wert eine lineare Beziehung besteht, wie in Kapitel 1, wie würden Sie dann vorgehen?

Beginnen wir mit ein paar einfachen synthetischen Daten:

```
# Synthetische Daten erstellen
torch.manual_seed(0) # Für Reproduzierbarkeit
x = torch.arange(0, 100, dtype=torch.float32)
y = 2 * x - 1
```

Dann könnten wir diese Daten wie folgt in einen Datensatz umwandeln:

```
class CustomDataset(Dataset):
    def __init__(self, x, y):
        """
        Initialize the dataset with x and y values.
        Arguments:
        x (torch.Tensor): The input features.
        y (torch.Tensor): The output labels.
        """
        self.x = x
        self.y = y

    def __len__(self):
        """
        Return the total number of samples in the dataset.
        """
        return len(self.x)

    def __getitem__(self, idx):
        """
        Fetch the sample at index `idx` from the dataset.
        Arguments:
        idx (int): The index of the sample to retrieve.
        """
        return self.x[idx], self.y[idx]
```

Um den Datensatz zu verwenden, legen wir einfach eine Instanz der Klasse an, initialisieren sie mit unseren x - und y -Werten, übergeben sie einem DataLoader und zählen sie auf:

```
# Instanz von CustomDataset anlegen
dataset = CustomDataset(x, y)

# DataLoader für Batching und Mischen der Daten benutzen
data_loader = DataLoader(dataset, batch_size=10, shuffle=True)

# Über den DataLoader iterieren
for batch_idx, (inputs, labels) in enumerate(data_loader):
    print(f"Batch {batch_idx+1}")
    print("Inputs:", inputs)
    print("Labels:", labels)
    # Zu Demonstrationszwecken nach dem ersten Batch abbrechen
    if batch_idx == 0:
        break
```

Mit diesen Grundlagen können Sie die Dataset-Klassen, die über die verschiedenen in diesem Kapitel erwähnten Bibliotheken zur Verfügung gestellt wurden, selbst weiter erforschen. Da sie entweder auf diesen Klassen aufbauen oder sie erweitern, sollten die APIs Ihnen bekannt vorkommen.

Die FashionMNIST-Klasse erforschen

Weiter oben im Buch haben wir uns bereits mit der Klasse `FashionMNIST` beschäftigt, über die Sie auf den Fashion-MNIST-Datensatz zugreifen können. Sie enthält Trainingsdaten mit 60.000 Beispielen für zehn verschiedene Arten von Kleidungsstücken sowie einen Satz an 10.000 Beispielen mit Testdaten. Jedes Beispiel ist ein 28×28 Pixel großes Graustufenbild.

Bei diesem Datensatz benutzen Sie für Training und Testing/Validierung die *gleiche* Klasse. Die Daten, die Sie erhalten, hängen vom Wert des übergebenen `train`-Parameters ab. Hier ein Beispiel:

```
# Den Fashion-MNIST-Trainingsdatensatz anlegen
fashion_mnist_train = datasets.FashionMNIST(root='./data',
                                             train=True, download=True, transform=transform)
```

Wenn Sie `train=True` setzen, übernimmt der Code, der die `init`-Methode der Klasse überschreibt, 60.000 Einträge und gibt sie an den Aufrufer zurück. Über weitere Parameter wird angegeben, in welchem Wurzelverzeichnis die Daten gespeichert werden sollen (`root`) und ob die Daten heruntergeladen werden sollen oder nicht (`download`). Außerdem gibt es, wie beim Herunterladen von Daten üblich, einen `transform`-Parameter. Wie in der vorherigen Basisklasse steht dieser optional allen Datensätzen zur Verfügung. Eine Transformation wird nur durchgeführt, wenn dieser Parameter gesetzt wurde.

Generische Dataset-Klassen

Manchmal müssen Sie Daten verwenden, die nicht über Dataset-Klassen wie `FashionMNIST` zur Verfügung stehen. Dennoch möchten Sie sämtliche Vorteile der `torch.utils.data`-Umgebung nutzen einschließlich der Möglichkeit, Daten zu transformieren und aufzutrennen, und aller weiteren nützlichen Funktionen der Klasse `DataLoader`, wie wir später in diesem Kapitel noch sehen werden. Für diesen Zweck stellt `torch.utils.data` eine Reihe allgemeiner (>generischer<) Datensatzklassen zur Verfügung.

ImageFolder

In Kapitel 3 haben wir die Datensätze *Horses or Humans*, *Rock, Paper, Scissors* sowie *Dogs vs. Cats* verwendet, die nicht direkt über eine Klasse zugänglich waren, sondern als ZIP-Dateien, die die Bilder enthielten. Nachdem wir sie heruntergeladen und in Unterverzeichnisse für die verschiedenen Bildtypen gespeichert haben, konnten wir die generische Klasse `ImageFolder` als Datensatz nutzen.

In diesem Fall werden die Bilder mithilfe des `DataLoader` abhängig von der dort definierten Batch-Größe und anderen Regeln aus dem Verzeichnis gestreamt. Dabei wurden die Labels von den Verzeichnisnamen abgeleitet. Als Klassenindizes dienten entsprechend die Labels in alphabetischer Reihenfolge. *Horses* war also Klasse 0 und *Humans* Klasse 1. Das sollten Sie beim Erstellen und Debuggen im Hinterkopf behalten, denn diese Reihenfolge kann verwirrend sein!

Im Englischen sagt man üblicherweise *Rock, Paper, Scissors* für »Stein, Schere, Papier«. In dieser (englischen) Reihenfolge würde man erwarten, dass *Rock* den Index 0, *Paper* den Index 1 und *Scissors* den Index 2 erhält. In alphabetischer Reihenfolge wird *Paper* aber zu Klasse 0, *Rock* zu Klasse 1 und *Scissors* zu Klasse 2!



Das können Sie mit einem eigenen Index umgehen, wie hier gezeigt:

```
custom_class_to_idx = {'rabbit': 0, 'dog': 1,
                       'cat': 2}

dataset = ImageFolder(
    root='data/animals',
    target_transform=
        lambda x: custom_class_to_idx[
            dataset.classes[x]]
)
dataset.class_to_idx = custom_class_to_idx
print(dataset.class_to_idx)
```

DatasetFolder

Eigentlich ist `ImageFolder` eine speziell für Bilder angepasste Unterklasse der generischen Klasse `DatasetFolder`. Diese ist nicht auf Bilddaten beschränkt, sondern kann für alle möglichen Datentypen eingesetzt werden. Mit ihr lassen sich Labels auch aus Verzeichnisnamen ableiten. Angenommen, Sie haben Dateien, die Texte in verschiedenen Kategorien enthalten, die in folgender Verzeichnisstruktur liegen:

```
root/sarcasm/document1.txt
root/sarcasm/document2.txt
root/sarcasm/document3.txt
root/factual/factdoc1.rtf
root/factual/factdoc2.doc
```

Sie könnten also einen `DatasetFolder` einsetzen, um die Dokumente entsprechend den korrekten Labels zu streamen. Und weil diese Klasse dokumentenbasiert ist, können Sie auch eine Transformation (`transform`) anwenden, um Daten aus dieser Datei abzuleiten!

FakeData

`FakeData` ist eine weitere nützliche `Dataset`-Klasse, die, wie der Name schon sagt, künstliche Zufallsdaten erzeugt. Beim Schreiben dieses Buchs funktionierte das allerdings nur für Bilddaten. Dieses `Dataset` ist außerdem nützlich, um mit verschiedenen Architekturen zu experimentieren oder die Leistung Ihres Systems zu messen, auch wenn Sie keine Daten zur Hand haben.

Dabei können Sie `FakeData` genauso verwenden wie die anderen `Datasets` aus diesem Buch. Mit folgendem Code können Sie `FakeData` beispielsweise nutzen, um einen Satz künstlicher Daten für das `MobileNet`-Modell zu erzeugen. Es erwartet Farbbilder im Format 224×224 :

```
import torch
from torchvision.datasets import FakeData
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
```

```

# Transformationen definieren (bei Bedarf)
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# FakeData erzeugen
fake_dataset = FakeData(size=100, image_size=(3, 224, 224),
                        num_classes=10, transform=transform)

# DataLoader
data_loader = DataLoader(fake_dataset, batch_size=10, shuffle=True)

```

Das würde 100 Bilder (die nur Rauschen enthalten) in der gewünschten Größe erzeugen und diese auf zehn Kategorien verteilen. Danach können wir sie wie jeden anderen Datensatz in einem `DataLoader` weiter nutzen.

Obwohl `FakeData` nur Bilddaten unterstützt, können Sie relativ leicht auch Ihre eigenen Daten verwenden. Hierfür nutzen Sie ein `CustomDataset`, wie weiter oben im Kapitel beschrieben, das Platzhalterdaten in anderen Formaten (zum Beispiel als numerische oder als sequenzielle Daten) erzeugen kann.

Custom Splits verwenden

Bis jetzt waren die Daten, die Sie für die Erstellung von Modellen genutzt haben, bereits in Trainings- und Testdaten aufgeteilt. Bei Fashion MNIST gab es beispielsweise 60.000 Trainings- und 10.000 Testdatensätze. Was machen Sie aber, wenn die Daten nach eigenen Kriterien aufgeteilt werden sollen?

Indem Sie den `datasets`-Namensraum nutzen, können Sie hierfür eine einfache und intuitive API nutzen.

Beim Laden der Klasse `FashionMNIST` haben Sie über den `train`-Parameter angegeben, ob die Trainingsdaten (60.000 Einträge) oder die Testdaten (10.000 Einträge) geladen werden sollen.

Wenn Sie `train` dagegen nicht definieren, werden alle Daten auf einmal geladen:

```

# Den gesamten Fashion-MNIST-Datensatz laden
# (Trainings *und* Testdaten gemeinsam)
dataset = datasets.FashionMNIST(root='./data',
                               download=True, transform=transform)

```

Zur Definition Ihrer eigenen Aufteilung (*Split*) finden Sie im `torch.utils.data`-Namensraum die Funktion `random_split`. Um beispielsweise einen eigenen Validierungsdatsatz zu definieren, den `FashionMNIST` von sich aus nicht bereitstellt, können Sie die insgesamt 70.000 Elemente per `random_split` in drei Datensätze aufteilen. Hier ist der Code, der 70 % der Daten für das Training, 15 % zum Testen und 15 % als Validierungsdaten reserviert:

```

from torch.utils.data import random_split

total_count = len(dataset)
train_count = int(0.7 * total_count)
val_count = int(0.15 * total_count)

```

```
# Sicherstellen, dass alle Daten verwendet werden
test_count = total_count - train_count - val_count

train_dataset, val_dataset, test_dataset =
    random_split(dataset, [train_count, val_count, test_count])
```

Wie Sie sehen, ist dieser Prozess recht unkompliziert. Die Zahl der Datensätze liegt in `total_count` (»Gesamtzahl«). Davon berechnen wir 70 % (0,7 multipliziert mit der Gesamtzahl) für die Trainingsdaten und 15 % für die Validierungsdaten. Bei solchen Berechnungen kann es zu Rundungsfehlern kommen, wodurch einige Einträge ausgelassen werden. Anstatt also für die Testdaten ebenfalls 15 % anzugeben, können Sie die tatsächliche Zahl der Einträge für Test- und Validierungsdaten (in `val_count`) einfach von der Gesamtzahl (in `total_count`) subtrahieren. Das Ergebnis wird in `test_count` gespeichert. So wird sichergestellt, dass keine Daten verschwendet werden.

Auf diese Weise können Sie sehr einfach neue und unterschiedliche Splits Ihres Datensatzes erstellen. Das schafft eine neue Möglichkeit, die Genauigkeit Ihrer Modelle beim und nach dem Training zu bewerten.

Liefert ein Teil des Datensatzes beim Training eine hohe Genauigkeit, während ein anderer zu einem schlechteren Wert führt, ist das ein Zeichen für eine Überanpassung. Wenn Sie dagegen mehrere unterschiedliche Splits Ihrer Daten ausprobieren und die Ergebnisse für Training und Validierung konsistent bleiben, dann ist dies ein Indiz für eine stabile und funktionierende Architektur.

Daher empfehle ich Ihnen dringend, beim Training von Modellen eigene Splits zu verwenden, weil sie Ihnen helfen, einige Fehler zu vermeiden, die schnell übersehen werden.

Beim Einsatz von Custom Splits sollten Sie bedenken, dass der Name `random_split` nicht bedeutet, dass die Einträge in Ihrem Datensatz hierdurch *gemischt* oder zufällig angeordnet werden. Der Datensatz wird nur an zufälligen Stellen *aufgetrennt*, damit Sie jedes Mal ein anderes Slice erhalten. Wollen Sie den Datensatz tatsächlich mischen, können Sie das im DataLoader erledigen, wie wir im folgenden Abschnitt zeigen.

Der ETL-Prozess zur Datenverwaltung im Machine Learning

Das Grundmuster für das Training von ML-Modellen heißt *Extract, Transfer, Load* (Extrahieren, Transformieren, Laden, kurz: ETL). Bisher haben wir uns in diesem Buch eher kleine Modelle erstellt, die auf einem einzelnen Computer laufen können. Die gleiche Technologie können wir aber auch anwenden, um Training in großem Maßstab mit riesigen Datensätzen und über mehrere Rechner verteilt durchzuführen.

Der ETL-Prozess besteht aus drei Phasen, die ihm seinen Namen geben:

Extraktion

Die Rohdaten werden aus ihrer Quelle geladen und so vorbereitet, dass sie transformiert werden können.

Transformation

Bei der Transformation werden die Daten so angepasst oder verbessert, dass sie für das Training verwendet werden können. Hierzu gehören Dinge wie die Einteilung in Batches, Image Augmentation, die Abbildung auf Features und ähnliche Logik.

Laden

Die Daten werden zum Training an das neuronale Netzwerk übergeben.

Sehen Sie sich hierzu noch einmal den Code aus Kapitel 3 an, mit dem wir den *Horses or Humans*-Classifier trainiert haben. Am Anfang des Codes befand sich dieser Abschnitt:

```
# Transformationen definieren
train_transform = transforms.Compose([
    transforms.Resize((150,150)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(20),
    transforms.RandomAffine(
        degrees=0,           # Keine Drehung
        translate=(0.2, 0.2), # Um 20% vertikal und horizontal
                             # verschieben
        scale=(0.8, 1.2),    # Um 20% ein- oder auszoomen
        shear=20,           # Scheren um bis zu 20 Grad
    ),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
])

# Datensätze laden
train_dataset = datasets.ImageFolder(root=training_dir,
                                     transform=train_transform)
val_dataset = datasets.ImageFolder(root=validation_dir,
                                   transform=train_transform)

# DataLoader
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=True)
```

Das ist das ETL-Muster in Codeform!

Am Anfang des Codes werden die Transformationen (das »T«) definiert. Der aktive Code beginnt aber erst nach dem Kommentar `# Datensätze laden`. Wenn Sie genau hinsehen, erkennen Sie, dass hier der `ImageFolder` verwendet wird, um die Daten von ihrem Speicherort auf der Festplatte zu *extrahieren*.

Bei der Extraktion der Daten werden dann die definierten `transforms` angewendet.

Auf den Kommentar `# DataLoader` folgt das Laden der Daten mithilfe des `train_loader` (Training) und des `val_loader` (Validierung). Streng genommen werden die Daten erst während der Trainingsschleife geladen, wenn sie aus den Loadern gelesen werden.

Durch diesen Prozess sind Ihre Daten-Pipelines weniger anfällig für Änderungen an den Daten und dem zugrunde liegenden Schema. Wenn Sie diesen Ansatz nutzen, wird grundsätzlich die gleiche Struktur verwendet, unabhängig davon, ob alle Daten auf einmal in den Speicher passen oder so groß sind, dass sie nicht mehr von einem einzelnen Rechner verarbeitet werden können. Und natürlich ist nach der Transformation der Daten auch der Pro-

zess des Ladens der Daten konsistent, und zwar unabhängig davon, welches Backend für das Training verwendet wird.

Dabei hat die Art und Weise, wie Sie die Daten laden, einen großen Einfluss auf die Trainingsgeschwindigkeit. Das wollen wir uns im folgenden Abschnitt genauer ansehen.

Die Ladephase optimieren

Jetzt wollen wir uns mit dem ETL-Prozess für das Training eines Modells befassen. Dabei gehen wir davon aus, dass Extraktion und Transformation auf einem beliebigen Prozessor inklusive einer CPU möglich sind. Tatsächlich sind GPUs und TPUs für Aufgaben wie das Herunterladen, Entpacken und die Verarbeitung Eintrag für Eintrag nicht gedacht. Das heißt, dieser Code wird ohnehin auf der CPU ausgeführt. Beim Training kann der Einsatz einer GPU oder TPU dagegen große Vorteile bringen, auf die Sie möglichst nicht verzichten sollten. Das heißt, wenn eine GPU oder TPU verfügbar ist, sollten Sie die Arbeit am besten zwischen CPU und GPU/TPU aufteilen. Dabei werden Extraktion und Transformation auf der CPU und das Laden auf der GPU/TPU ausgeführt.

Der Code in diesem Buch verwendet häufig die Schreibweise `.to(device)`. Soll das Training oder die Inferenz auf einem Beschleuniger (*Accelerator*) ausgeführt werden, sehen Sie beispielsweise Code wie `.to("cuda")`. Für die Extraktions- und Transformationsphase wird dieser Code dagegen nicht benutzt, weil er eine Verschwendung von GPU-Ressourcen bedeuten würde.

Angenommen, Sie arbeiten an einem so großen Datensatz, dass die Daten in Batches verarbeitet werden müssen (Extraktion und Transformation), wie in Abbildung 4.1 gezeigt. Während der Vorbereitung des ersten Batch kommt GPU bzw. TPU nicht zum Einsatz. Ist er bereit, kann er für das Training an die GPU/TPU übergeben werden. In dieser Zeit ist wiederum die CPU im »Leerlauf«, bis das Training abgeschlossen ist und der zweite Batch vorbereitet werden kann. Das ist eine Menge Leerlauf. Hier gibt es also noch einiges zu verbessern.

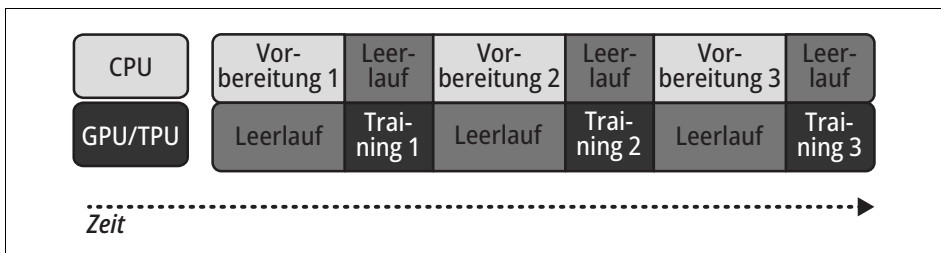


Abbildung 4.1: Nutzung von CPU oder GPU/TPU

Die logische Lösung besteht darin, die Aufgaben parallel abzuarbeiten, sodass Vorbereitung und Training gleichzeitig ausgeführt werden können. Dieses Verfahren wird als *Pipelining* bezeichnet, wie in Abbildung 4.2 gezeigt.

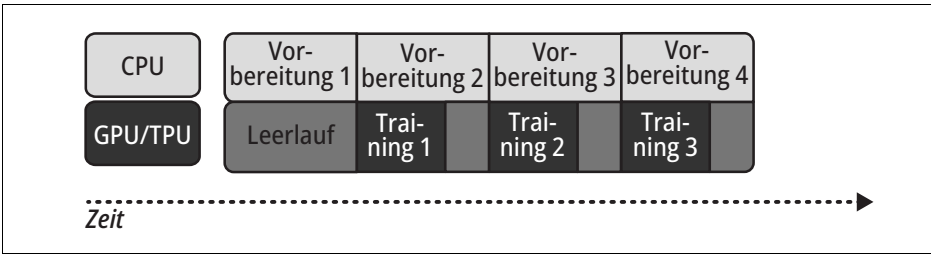


Abbildung 4.2: Pipelining

Hier befindet sich die GPU/TPU im »Leerlauf«, während die CPU den ersten Batch vorbe-reitet. Ist der erste Batch bereit, kann die GPU/TPU mit dem Training beginnen. Parallel dazu bereitet die CPU aber schon den zweiten Batch vor. Natürlich braucht es Zeit, Batch $n - 1$ zu trainieren, während die Vorbereitung für Batch n nicht immer genauso lang dauert. Ist das Training kürzer, gibt es auch weiterhin Leerlaufpausen auf der GPU/TPU. Dauert das Training dagegen länger, kann es trotzdem zu Leerlauf auf der CPU kommen. Das lässt sich durch die Wahl der richtigen Batch-Größe optimieren. Dabei ist GPU/TPU-Zeit sehr wahr-scheinlich teurer. Das heißt, hier sollten Sie Leerlaufzeiten nach Möglichkeit vermeiden.

Dies ist einer der Gründe, warum wir selbst für kleine Beispiele wie MNIST mit Batches ar-beiten: Das Pipelining-Modell funktioniert unabhängig von der Größe Ihres Datensatzes, und auf diese Weise nutzen Sie grundsätzlich ein konsistentes ETL-Muster.

Die DataLoader-Klasse verwenden

Auch wenn wir die Klasse `DataLoader` schon oft gesehen haben, lohnt sich ein genauerer Blick hinter die Kulissen, damit Sie Ihre ML-Arbeitsabläufe möglichst effizient gestalten können. Sie stellt folgende Funktionalitäten bereit:

Batching

Sie könnten denken, dass die Daten elementweise nacheinander an die folgenden Schichten weitergereicht werden. Das ist *grundsätzlich* möglich, aber einige Optimizer wie der stoch-astische Gradientenabstieg funktionieren deutlich besser, wenn die Eingaben in Batches (in denen mehrere Einträge zusammengefasst sind) übergeben werden, weil dann genauer ge-rechnet werden kann. Das Batching kann das Training gerade in größeren Szenarien be-schleunigen, in denen die GPU mit festen Speichergrößen verwendet wird. Am effizientes-ten ist es, Daten-Batches einzusetzen, die diesen Speicher möglichst vollständig ausnutzen. Beim Einsatz eines `DataLoader` müssen Sie hierfür einfach nur einen Parameter setzen.

Daten mischen

Das Mischen (*Shuffling*) der Daten ist besonders wichtig, vor allem wenn Sie das Batching verwenden. Nehmen wir zum Beispiel ein Szenario wie Fashion MNIST.

Wie Sie wissen, enthält Fashion MNIST 60.000 Einträge aus zehn Kategorien. Die Daten sind nicht gemischt. Die Batches haben eine Größe von jeweils 1.000 Einträgen. Dabei ist der gesamte erste Batch vollständig der Kategorie 0 zugeordnet, der zweite der Kategorie 1 und so weiter. Dadurch kann das Modell möglicherweise nicht gut lernen, weil jeder Batch für eine bestimmte Kategorie »voreingenommen« ist. Die Fähigkeit Ihres Modells zu verallgemeinern, steigt jedoch, wenn die Daten vor der Verteilung auf die Batches gemischt wurden, wodurch Labels und andere Merkmale der Daten zufällig verteilt sind.

Daten parallel laden

Oftmals kann das Laden, besonders von komplexen Daten, in ein Modell sehr zeitaufwendig sein. Der DataLoader ermöglicht eine Parallelisierung des Ladevorgangs, indem er Pythons Multiprocessing-Modell nutzt, was zu einer erheblichen Geschwindigkeitssteigerung führen kann.

Wie zuvor sollten Sie überlegen, das Laden und die Transformation der Daten sowie das eigentliche Lernen auf zwei separate Prozesse aufzuteilen. Damit vermeiden Sie Situationen, in denen das Modell im »Leerlauf« auf die Daten für das Training warten muss oder sich Tonnen von Daten im Speicher befinden, aber das Modell nicht darauf zugreifen kann. Das parallele Laden von Daten kann Abhilfe schaffen, sofern es gut auf das jeweilige Szenario abgestimmt ist. Für ein möglichst effizientes Training lohnt es sich, das benutzerdefinierte Daten-Sampling zu lernen, auf das wir im folgenden Abschnitt eingehen.

Benutzerdefiniertes Daten-Sampling

Zusätzlich zum Mischen der Daten, um eine zufällige Reihenfolge zu erhalten, können Sie auch eigene Regeln festlegen, nach denen die Daten geladen werden. Hierbei dient `torch.utils.data.Sampler` als Basisklasse. Diesen Prozess detailliert zu beschreiben, würde allerdings den Rahmen dieses Buchs sprengen. Sie finden eine Reihe ausgezeichnete Beispiele online.

ETL parallelisieren, um die Trainingsleistung zu steigern

Mit der `DataLoader`-Klasse ist die Einrichtung der Parallelisierung ein Kinderspiel. Sie müssen einfach dem Parameter `num_workers` einen passenden Wert zuweisen. Das folgende Beispiel zeigt Schritt für Schritt, wie Sie ein Modell mit dem CIFAR10-Datensatz trainieren und hierfür die Parallelisierung nutzen können.

Zunächst sehen wir uns die Extraktions- und Transformationsschritte an:

```
import torchvision.transforms as transforms
from torchvision.datasets import CIFAR10

# Transformationen definieren
transform = transforms.Compose([
    transforms.ToTensor(),
```

```

        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

# CIFAR10-Datensatz laden
dataset = CIFAR10(root='./data', train=True, download=True,
transform=transform)

```

Anschließend definieren wir für die Ladephase einen entsprechenden DataLoader:

```

from torch.utils.data import DataLoader

# DataLoader mit mehreren Worker-Prozessen
data_loader = DataLoader(dataset, batch_size=64, shuffle=True,
                        num_workers=4)

```

Im obigen Codebeispiel sorgt der Parameter `num_workers=4` für die Erstellung von vier Worker-Subprozessen, um die Daten gleichzeitig zu laden. Je nach verwendeter Hardware sowie Anzahl der Prozessorkerne und deren Geschwindigkeit können Sie mit der Anzahl der Worker-Prozesse experimentieren, um eventuelle Engstellen bei der Verarbeitung zu verringern.

Das Praktische an diesem Ansatz ist die vollständige Verkapselung des ETL-Prozesses. Obwohl die Daten parallel geladen werden, müssen Sie die Trainingsschleife Ihres Modells also nicht anpassen. Unten sehen Sie den Code eines einfachen CIFAR-Modells, das diese Daten nutzt:

```

import torch

# Einrichtung von Dummy-Modell und Optimizer
model = torch.nn.Sequential(
    torch.nn.Linear(3 * 32 * 32, 500),
    torch.nn.ReLU(),
    torch.nn.Linear(500, 10)
)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
criterion = torch.nn.CrossEntropyLoss()

# Trainingsschleife
def train(model, data_loader):
    model.train()
    for batch_idx, (inputs, targets) in enumerate(data_loader):
        # Eingaben umformen, damit sie auf die vom
        # Modell erwarteten Ausgaben passen
        inputs = inputs.view(inputs.size(0), -1)

        # Daten weiterreichen (Forward Pass)
        outputs = model(inputs)
        loss = criterion(outputs, targets)

        # Backpropagation (Backward Pass) und Optimierung
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch_idx % 100 == 0:
            print(f"Train Epoch: {batch_idx} Loss: {loss.item()}")

train(model, data_loader)

```

Die Parallelisierung ist ein weiteres Werkzeug, das Sie für das Training Ihrer Modelle nutzen können. Das ist keine allgemeingültige Lösung, aber ein guter Ansatz, wenn Sie merken, dass sich das Training verlangsamt. Leicht könnte man denken, die Schuld für ein langsames Training trüge die Netzwerkarchitektur, aber Sie werden überrascht sein, wie viel Zeit verschwendet wird, weil der Forward Pass auf neue Daten wartet! Durch diese Art der Parallelverarbeitung können Sie das Training möglicherweise deutlich beschleunigen.

Zusammenfassung

In diesem Kapitel haben wir uns die Arbeit mit Daten in PyTorch genauer angesehen und Ihnen die Klassen `Dataset` und `DataLoader` vorgestellt. Wir haben gezeigt, wie Sie durch eine einheitliche API und ein gemeinsames Format die Menge des zu schreibenden Codes für den Datenzugriff deutlich verringern können. Sie haben außerdem gesehen, wie Sie den ETL-Prozess nutzen können, der das Kernstück vieler häufig benutzter Entwurfsmuster für das Training von PyTorch-Modellen bildet. Außerdem haben wir uns damit befasst, wie Parallelisierung von Extraktion, Transformation und dem Laden von Daten die Leistung des Trainings verbessern kann.

Nachdem Sie Gelegenheit hatten, sich den Prozess anzusehen, sollten Sie versuchen, Ihren eigenen Datensatz zu erstellen! Vielleicht verwenden Sie hierfür Ihr eigenes Fotoalbum, bestimmte Testdaten oder einfach zufällig generiertes Rauschen, wie wir es hier getan haben.

Im nächsten Kapitel wenden wir das bisher Gelernte auf Probleme der Verarbeitung natürlicher Sprachen an.

Einführung in die Verarbeitung natürlicher Sprache

Die Verarbeitung natürlicher Sprache (*Natural Language Processing*, NLP) ist ein KI-Verfahren, das sich mit dem Verständnis von Sprache beschäftigt. Hierzu gehören Programmiertechniken, die Sprache verstehen, Inhalte klassifizieren und sogar neue sprachliche Inhalte erzeugen können. Sie bilden auch die Grundlage für große Sprachmodelle (*Large Language Models*, LLM) wie ChatGPT, Gemini und Claude. Wir werden uns in späteren Kapiteln mit LLMs beschäftigen. Zuerst kümmern wir uns in den folgenden Kapiteln aber um die Grundlagen von NLP, um Sie auf das vorzubereiten, was danach kommt.

Es gibt mittlerweile eine Vielzahl von Diensten, die NLP für die Erstellung von Applikationen wie Chatbots nutzen. Dieses Thema würde aber den Rahmen dieses Buchs sprengen. Stattdessen sehen wir uns die Grundlagen von NLP an und wie man Sprache so modellieren kann, dass Sie damit ein neuronales Netzwerk trainieren können, das Texte verstehen und klassifizieren kann. Weiter unten im Buch lernen Sie außerdem, wie Sie die Vorhersageelemente eines ML-Modells nutzen können, um Gedichte zu schreiben. Das ist nicht nur unterhaltsam, sondern dient auch als Vorstufe zum Lernen von Transformer-basierten Modellen, der Grundlage generativer KI!

Wir beginnen dieses Kapitel mit einem Blick darauf, wie Sie Sprache in Zahlen zerlegen können, die anschließend in neuronalen Netzwerken weiterverarbeitet werden können.

Sprache in Zahlen codieren

Im Prinzip basieren Computer auf Zahlen. Um Sprache verarbeiten zu können, brauchen Sie also eine Möglichkeit, sie entsprechend umzuwandeln. Dieser Prozess heißt *Encodierung* (*Encoding*).

Sie können Sprache auf vielerlei Weise encodieren. Am häufigsten wird nach Buchstaben encodiert, wie Sie das auch sonst tun würden, wenn Strings in Ihrem Programm gespeichert werden. Im Speicher liegt jedoch nicht der Buchstabe *a*, sondern eine Encodierung davon. Das kann ein ASCII- oder Unicode-Wert sein oder auch etwas ganz anderes. Um das Wort *listen* (»zuhören«) mit ASCII zu encodieren, nutzen Sie beispielsweise die Codes 76, 73, 83, 84, 69 und 78. Das ist praktisch, weil Sie das Wort jetzt als Zahlen darstellen können. Wenn Sie nun aber das Wort *silent* (»still«) betrachten (ein Anagramm von *listen*), sehen Sie, dass es aus denselben Zahlen besteht. Trotz der unterschiedlichen Reihenfolge würde dies die Erstellung eines Modells deutlich erschweren.

Eine bessere Alternative besteht in der Verwendung von Zahlen, um anstelle der enthaltenen Buchstaben ganze Wörter zu encodieren. In diesem Fall könnte *silent* die Zahl x und *listen* die Zahl y erhalten, ohne dass sich beide überschneiden.

Und jetzt stellen Sie sich vor, einen Satz wie »I love my dog« auf diese Weise zu encodieren, zum Beispiel mit den Zahlen [1, 2, 3, 4]. Wollten Sie nun »I love my cat« encodieren, könnten Sie [1, 2, 3, 5] schreiben. Inzwischen können Sie schon sehen, dass die Sätze eine ähnliche Bedeutung haben, weil sie numerisch ähnlich sind. Anders gesagt: [1, 2, 3, 4] hat große Ähnlichkeit mit [1, 2, 3, 5].

Die Zahlen, die wir für die Darstellung von Wörtern benutzen, nennt man *Tokens*. Daher wird dieses Verfahren auch als *Tokenisierung* (*Tokenization*) bezeichnet.

Einstieg in die Tokenisierung

Im PyTorch-Ökosystem gibt es eine große Zahl an Bibliotheken für die Tokenisierung. Sie übernehmen Wörter und wandeln sie in Tokens um. In Codebeispielen kommt dabei häufig `torchtext` als Tokenizer zum Einsatz. Dieser gilt allerdings seit 2023 als veraltet. Daher sollten Sie bei seiner Verwendung vorsichtig sein, besonders weil PyTorch beständig weiterentwickelt wird, `torchtext` aber nicht. Alternativ können Sie einen eigenen oder einen vortrainierten Tokenizer verwenden – oder (überraschenderweise) diejenigen aus dem Keras-Umfeld.

Einen eigenen Tokenizer verwenden

Unten sehen Sie ein Codebeispiel, in dem ich einen benutzerdefinierten Tokenizer erstellt habe, um die Wörter aus einem kleinen Korpus (zwei Sätze) in Tokens umzuwandeln:

```
import torch

sentences = [
    'Today is a sunny day',
    'Today is a rainy day'
]

# Tokenisierungsfunktion
def tokenize(text):
    return text.lower().split()

# Vokabular aufbauen
def build_vocab(sentences):
    vocab = {}
    for sentence in sentences:
        tokens = tokenize(sentence)
        for token in tokens:
            if token not in vocab:
                vocab[token] = len(vocab) + 1
    return vocab

# Index für Vokabular erstellen
vocab = build_vocab(sentences)

print("Vocabulary Index:", vocab)
```



Das Wort *Korpus* wird üblicherweise benutzt, um einen Satz an Textelementen zu beschreiben, die Sie für das Training einsetzen. Wörtlich steht es für den *Textkörper*, den Sie verwenden, um das Modell zu trainieren und dafür Tokenizer zu erstellen.

Die Ausgabe des obigen Codes sieht so aus:

```
Vocabulary Index: {'today': 1, 'is': 2, 'a': 3, 'sunny': 4, 'day': 5, 'rainy': 6}
```

Wie Sie sehen, hat der Tokenizer einfach eine Liste meiner Wörter erstellt. Jedes Mal, wenn er ein neues einmaliges Wort gefunden hat, wurde dieses zur Liste hinzugefügt. So erhalten wir für den ersten Satz, »Today is a sunny day«, fünf Tokens für fünf Wörter: »today«, »is«, »a«, »sunny« und »day«. Der zweite Satz hat die *meisten* Wörter mit dem ersten gemeinsam, wobei »rainy« die Ausnahme bildet. Daher bekommt es ein eigenes Token.

Für einen sehr großen Korpus ist dieser Prozess verständlicherweise sehr langsam.

Einen vortrainierten Tokenizer von Hugging Face verwenden

Aus diesem Grund werde ich stattdessen die `transformers`-Bibliothek von Hugging Face und die darin enthaltenen vortrainierten Tokenizer einsetzen. Da `transformers` viele Sprachmodelle unterstützt, die Tokenizer für die Arbeit mit einem Textkorpus benötigen, steht Ihnen der Tokenizer, der mit Millionen von Wörtern trainiert wurde, kostenlos zur Verfügung. Er hat eine größere Abdeckung, als Sie vermutlich erreichen können, und er ist kostenlos und einfach zu benutzen!

Wenn Sie diese Bibliothek noch nicht haben, können Sie sie mit diesem Befehl installieren:

```
!pip install transformers
```

Die Verwendung von `transformers` wollen wir jetzt an einem kleinen Beispiel demonstrieren:

```
from transformers import BertTokenizerFast

sentences = [
    'Today is a sunny day',
    'Today is a rainy day'
]

# Tokenizer initialisieren
tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')

# Sätze tokenisieren und encodieren
encoded_inputs = tokenizer(sentences, padding=True, truncation=True,
                           return_tensors='pt')

# Um die Tokens für jede Eingabe sehen zu können
# (hilfreich, um die Ausgaben zu verstehen)
tokens = [tokenizer.convert_ids_to_tokens(ids)
          for ids in encoded_inputs["input_ids"]]

# Einen Wortindex erstellen
```

```
# (vergleichbar mit 'word_index' aus Keras' Tokenizer)
word_index = tokenizer.get_vocab()

print("Tokens:", tokens)
print("Token IDs:", encoded_inputs['input_ids'])
print("Word Index:", dict(list(word_index.items())[:10]))
# Zur Übersicht nur die ersten 10 Einträge zeigen
```

Die Ausgabe sieht dann so aus:

```
Tokens: [['[CLS]', 'today', 'is', 'a', 'sunny', 'day', '[SEP]'],
          ['[CLS]', 'today', 'is', 'a', 'rainy', 'day', '[SEP]']]

Token IDs: tensor([
  [ 101, 2651, 2003, 1037, 11559, 2154, 102],
  [ 101, 2651, 2003, 1037, 16373, 2154, 102]])

Word Index: {'protestant': 8330, 'initial': 3988, '##pt': 13876,
             'charters': 23010, '243': 22884, 'ref': 25416,
             '##dies': 18389, '##uchi': 15217, 'sainte': 16947,
             'annette': 22521}
```

Das wollen wir uns jetzt im Detail ansehen. Wir beginnen mit dem Import von `BertTokenizerFast` aus der `transformers`-Bibliothek. Dieser kann mit einer Vielzahl vortrainierter Tokenizer initialisiert werden. Wir haben uns hier für `'bert-base-uncased'` entschieden. Jetzt fragen Sie sich wahrscheinlich, was das denn bloß sein soll! Die Grundidee war, einen vortrainierten Tokenizer zu einzusetzen. Diese sind an das Modell gekoppelt, an dem sie trainiert wurden. BERT (*Bidirectional Encoder Representations from Transformers*) ist ein Modell, das von Google an einem großen Korpus mit einem Vokabular von 30.000 Wörtern trainiert wurde. Modelle wie dieses finden Sie im Hugging-Face-Model-Hub (<https://huggingface.co/docs/hub/models-the-hub>). Und wenn Sie sich tief genug in die Informationen zu einem Modell hineinarbeiten, finden Sie häufig den Code, mit dem der Transformer seinen Tokenizer definiert. Ein Beispiel sehen Sie auf dieser von mir genutzten Seite (<https://oreil.ly/Ok7L9>). Obwohl ich das Modell selbst nicht verwendet habe, kann ich dessen Tokenizer übernehmen, anstatt selbst einen zu entwickeln.

In diesem Beispiel erstellen wir ein `tokenizer`-Objekt und geben an, wie viele Wörter er tokenisieren kann. Dies ist die maximale Anzahl an Tokens, die aus dem Wortkorpus erzeugt werden können. Wir haben es hier mit einem sehr kleinen Korpus zu tun, der nur sechs einmalige Wörter enthält. Daher bleiben wir problemlos unter dem Maximalwert von 100.

Sobald ich den Tokenizer habe, kann ich ihm den Text übergeben:

```
# Sätze tokenisieren und encodieren
encoded_inputs = tokenizer(sentences, padding=True, truncation=True,
                          return_tensors='pt')
```

Auf die Parameter `padding` und `truncation` werden wir etwas später in diesem Kapitel zu sprechen kommen. Im Moment ist vor allem der Parameter `return_tensors='pt'` wichtig. Dies ist eine Arbeitserleichterung für PyTorch-Entwickler, weil die Werte als `torch.Tensor`-Objekte zurückgegeben werden, die wir leicht weiterverarbeiten können.

Das BERT-Modell nutzt auf Basis der ursprünglichen Tokenisierung eine Reihe zusätzlicher Repräsentationsebenen, zum Beispiel `attention_masking`. Das heißt, anstelle der »rohen« Tokens werden für jedes Wort IDs vergeben. Für Sie ist vor allem wichtig, dass Sie die Tokens bei Bedarf auf die unten gezeigte Weise selbst extrahieren müssen, weil sie vom BERT-Tokenizer als `input_ids` encodiert wurden.

```
# Um die Tokens für jede Eingabe sehen zu können
# (hilfreich, um die Ausgaben zu verstehen)
tokens = [tokenizer.convert_ids_to_tokens(ids)
          for ids in encoded_inputs["input_ids"]]
```

Danach können Sie ohne große Probleme die folgende Token-Sammlung ausgeben:

```
Tokens: [['[CLS]', 'today', 'is', 'a', 'sunny', 'day', '[SEP]'],
         ['[CLS]', 'today', 'is', 'a', 'rainy', 'day', '[SEP]']]
```

Jetzt wollen Sie vermutlich wissen, was es mit `[CLS]` und `[SEP]` auf sich hat. BERT wurde darauf trainiert, Sätze zu erwarten, die mit einem *Classifier* (`[CLS]`) beginnen und mit einem *Separator* (`[SEP]`) enden. Diese beiden Ausdrücke sind als Werte 101 und 102 tokenisiert. Wenn Sie die Token-Werte für Ihren Satz ausgeben, sieht das entsprechend so aus:

```
Token IDs: tensor([
  [ 101, 2651, 2003, 1037, 11559, 2154, 102],
  [ 101, 2651, 2003, 1037, 16373, 2154, 102]])
```

Daran können Sie erkennen, dass *today* in BERT als Token 2651, *is* als Token 2003 usw. encodiert ist.

Es kommt also sehr darauf an, wie Sie an die Sache herangehen wollen. Für das Lernen mit kleinen Datensätzen wird ein eigener Tokenizer ausreichen. Sobald Sie aber mit größeren Datensätzen arbeiten, ist ein vortrainierter Tokenizer wahrscheinlich besser geeignet. Das bedeutet aber auch einen gewissen Mehraufwand. Daher werde ich in diesem Kapitel mit eigenem Code für die Tokenisierung und das Preprocessing des Texts arbeiten und auf Dinge wie den BERT-Tokenizer verzichten.

Sobald die Wörter in Ihren Sätzen tokenisiert sind, besteht der nächste Schritt darin, die Sätze in Listen mit Zahlen umzuwandeln. Dabei dienen die Wörter als Schlüssel und die Zahlen als Werte. Dieser Prozess wird als *Sequenzierung* (*Sequencing*) bezeichnet.

Sätze in Sequenzen umwandeln

Nachdem Sie gesehen haben, wie Wörter zu Zahlen tokenisiert werden, besteht der nächste Schritt darin, die Sätze in Zahlenfolgen (Sequenzen) zu encodieren, wie hier gezeigt:

```
def text_to_sequence(text, vocab):
    return [vocab.get(token, 0) for token in tokenize(text)]
# 0 für unbekannte Wörter
```

Dadurch erhalten Sie die Sequenzen, die für unsere zwei Sätze stehen. Hier zur Erinnerung noch einmal der Wortindex:

```
Vocabulary Index: {'today': 1, 'is': 2, 'a': 3, 'sunny': 4, 'day': 5,
                  'rainy': 6}
```

Die Ausgabe dazu sieht so aus:

```
[1, 2, 3, 4, 5]
[1, 2, 3, 6, 5]
```

Wenn Sie die Wörter durch die Zahlen austauschen, sehen Sie, dass die Sätze tatsächlich einen Sinn ergeben.

Und jetzt überlegen Sie, was passiert, wenn Sie ein neuronales Netzwerk mit einem Datensatz trainieren. Typischerweise decken die Trainingsdaten nicht alle Ihre Bedürfnisse ab. Dennoch hoffen Sie, dass die Abdeckung so groß wie möglich ist. Im Fall von NLP enthalten Ihre Trainingsdaten vielleicht Tausende Wörter, die in verschiedenen Kontexten benutzt werden können. Dabei ist es aber so gut wie unmöglich, jedes Wort in jedem möglichen Kontext vorzuhalten. Und was passiert wohl, wenn Sie Ihrem neuronalen Netzwerk neuen bisher unbekanntem Text geben, der noch nicht gesehene Wörter enthält? Sie können es sich schon denken: Das Netzwerk kommt durcheinander, weil es schlicht keinen Kontext für diese Wörter hat. Das Ergebnis ist, dass jede Vorhersage negativ beeinflusst wird.

Verwendung von Tokens, die außerhalb des Vokabulars liegen

Ein Lösungsansatz für dieses Problem liegt in der Verwendung von Tokens, die außerhalb des Vokabulars liegen, sogenannten *Out-of-Vocabulary-Tokens* (OOV). Sie können Ihrem neuronalen Netzwerk dabei helfen, den Kontext von bisher noch nicht gesehenem Text zu verstehen. Im nächsten Beispiel versuchen wir, die folgenden Sätze mit dem bisherigen Korpus zu verarbeiten:

```
test_data = [
    'Today is a snowy day',
    'Will it be rainy tomorrow?'
]
```

Dabei fügen wir diese Eingaben nicht dem Korpus des vorhandenen Texts hinzu (den wir uns als Trainingsdaten vorstellen können). Wie würde ein vortrainiertes Netzwerk diesen wohl sehen? Angenommen, Sie tokenisieren ihn mit den Wörtern, die Sie im bereits vorhandenen Tokenizer benutzt haben:

```
for test_sentence in test_data:
    test_seq = text_to_sequence(test_sentence, vocab)
    print(test_seq)
```

Dann sähen die Ergebnisse folgendermaßen aus:

```
[1, 2, 3, 0, 5]
[0, 0, 0, 6, 0]
```

Wenn wir die Tokens wieder in Wörter umwandeln, lauten die Ergebnisse »today is a <UNB> day« und »<UNB> <UNB> <UNB> rainy <UNB>.«

Hier nutze ich das Tag <UNB> (für *unbekannt*) für das Token 0. Im weiter oben gezeigten `text_to_sequence`-Code haben wir festgelegt, dass Wörtern, die sich nicht im Wörterbuch befinden, das Token 0 zugewiesen werden soll. Sie können aber auch einen komplett anderen Wert verwenden, zum Beispiel »Wurbelpunst«.

Padding und Truncation verstehen

Beim Training von neuronalen Netzwerken müssen normalerweise alle Daten die gleiche Form haben. Weiter oben im Buch haben Sie beispielsweise Bilder so formatiert, dass alle die gleiche Höhe und Breite aufweisen. Bei Text entsteht ein ähnliches Problem – sobald die Wörter tokenisiert und die Sätze in Sequenzen umgewandelt sind, können diese unterschiedliche Längen haben. Damit alle die gleiche Größe und Form bekommen, können Sie *Padding* (»Auffüllen«) verwenden.

Bisher bestanden alle Sätze, die wir zusammengesetzt haben, aus fünf Wörtern, sodass die Sequenzen entsprechend aus je fünf Tokens bestanden. Was würde wohl passieren, wenn einige Sätze länger wären als andere, zum Beispiel fünf, acht oder zehn Wörter hätten? Damit ein neuronales Netzwerk diese Sätze verarbeiten kann, müssen alle die gleiche Länge haben! Eine Möglichkeit wäre, kürzere Sätze auf zehn Wörter zu verlängern, eine andere, sie auf fünf Wörter zu verkürzen und die Bits der längeren Sätze einfach abzuschneiden. Und vielleicht gibt es auch noch andere Wege, das Problem zu lösen.

Um das Padding zu erforschen, erweitern wir unseren Korpus um einige deutlich längere Sätze:

```
sentences = [  
    'Today is a sunny day',  
    'Today is a rainy day',  
    'Is it sunny today?',  
    'I really enjoyed walking in the snow today'  
]
```

Nach dem Sequenzieren haben die resultierenden Zahlenlisten unterschiedliche Längen. Wenn Sie keine erneute Tokenisierung durchgeführt haben, enthalten die letzten beiden Sätze außerdem eine Reihe von Nullen:

```
[1, 2, 3, 4, 5]  
[1, 2, 3, 6, 5]  
[2, 0, 4, 0]  
[0, 0, 0, 0, 0, 0, 0, 1]
```

Vergessen Sie also nicht den folgenden Aufruf:

```
vocab = build_vocab(sentences)
```

Damit enthält der Tokenizer die nötigen Tokens für die neuen Wörter, und die Ausgabe sieht nun so aus:

```
[1, 2, 3, 4, 5]  
[1, 2, 3, 6, 5]  
[2, 7, 4, 8]  
[9, 10, 11, 12, 13, 14, 15, 1]
```

Beim Training der neuronalen Netzwerke weiter oben im Buch mussten die an die Eingabeschicht übergebenen Bilder konsistente Größen und Formen haben. Das Gleiche gilt in großen Teilen auch für NLP. (Es gibt eine Ausnahme namens *Ragged Tensors*, die wir in diesem Kapitel nicht weiter behandeln.) Wir brauchen, wie gesagt, eine Möglichkeit, die Sätze auf die gleiche Länge zu bringen.

Hier ist eine einfache Padding-Funktion:

```
def pad_sequences(sequences, maxlen):
    return [seq + [0] * (maxlen - len(seq)) if len(seq) < maxlen
            else seq[:maxlen] for seq in sequences]
```

Diese Funktion formt jedes Array der Sequenz so um, dass alle die gleiche Länge haben wie das längste. Um unsere Sätze nach der Sequenzierung aufzufüllen, können Sie beispielsweise diesen Code verwenden:

```
for sentence in sentences:
    seq = text_to_sequence(sentence, vocab)
    padded_seq = pad_sequences([seq], maxlen=10) # Beispiel für
                                                # maximale Länge
    print(padded_seq)
```

Anschließend sieht die Ausgabe so aus:

```
[[1, 2, 3, 4, 5, 0, 0, 0, 0, 0]]
[[1, 2, 3, 6, 5, 0, 0, 0, 0, 0]]
[[2, 7, 4, 8, 0, 0, 0, 0, 0, 0]]
[[9, 10, 11, 12, 13, 14, 15, 1, 0, 0]]
```

Jetzt hat jede Sequenz eine Länge von 10, die durch den Parameter `maxlen` definiert wird. Dies ist eine recht einfache Implementierung, die für ernsthafte Anwendungen vermutlich erweitert werden muss. So sollten Sie überlegen, was passiert, wenn eine Sequenz länger ist als die in `maxlen` angegebene maximale Länge. Im Moment wird einfach alles, was über `maxlen` hinausgeht, abgeschnitten. Eventuell ist ein anderes Verhalten hier sinnvoller. An dieser Stelle kommt die *Truncation* (»Verkürzung«) ins Spiel. Hierbei wird die Sequenz auf die gewünschte Länge gekürzt. Typischerweise findet die Truncation auf der rechten Seite statt. In manchen Fällen (etwa bei Sprachen, die von rechts nach links geschrieben werden) ist eine Kürzung ausgehend vom Anfang der Sequenz eventuell geeigneter.

Bedenken Sie auch, dass vorgefertigte Tokenizer wie der weiter oben gezeigte BERT einen großen Teil dieser Funktionalität schon mitbringt. Vergessen Sie also nicht, zu experimentieren.

Stoppwörter entfernen und Text aufräumen

In diesem Abschnitt sehen wir uns ein paar Datensätze aus der realen Welt an. Dabei werden Sie feststellen, dass es oft Text gibt, den Sie da *nicht* haben wollen. Außerdem wollen Sie möglicherweise sogenannte *Stoppwörter* ausfiltern, also Begriffe wie »the«, »and« und »but«, die zu häufig vorkommen und die den Sinngehalt des Texts nicht steigern. Außerdem enthält Text oft HTML-Tags. Auch hierfür ist es gut, eine saubere Methode zu ihrer Entfernung zu besitzen. Weitere Dinge, die Sie vielleicht ausfiltern möchten, sind Beleidigungen oder Schimpfwörter, Interpunktionszeichen oder Namen. Später untersuchen wir einen Datensatz mit Tweets, die oft eine Benutzer-ID enthalten, die ebenfalls ausgefiltert werden sollte.

Zwar unterscheiden sich die Aufgaben je nachdem, welchen Korpus Sie verwenden, trotzdem gibt es drei Dinge, die Sie programmatisch erledigen können, um Ihren Text aufzuräumen.

HTML-Tags entfernen

Beispielsweise können Sie HTML-Tags aus Ihrem Text entfernen. Glücklicherweise gibt es eine Python-Bibliothek namens BeautifulSoup, mit der das ganz einfach funktioniert. Wenn Ihre Sätze beispielsweise HTML-Tags wie `
` enthalten, können Sie diese mit folgendem Code ausfiltern:

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(sentence)
sentence = soup.get_text()
```

Stoppwörter entfernen

Zusätzlich können Sie die Stoppwörter ausfiltern. Häufig legt man hierfür eine Liste unerwünschter Wörter an, die anschließend in einem Preprocessing-Schritt aus Ihren Sätzen entfernt werden. Hier ein verkürztes Beispiel für eine solche Liste:

```
stopwords = ["a", "about", "above", ... "yours", "yourself",
"yourselves"]
```

Eine vollständige Liste englischer Stoppwörter finden Sie in einigen der Onlinebeispiele zu diesem Kapitel (<https://github.com/lmoroney/PyTorch-Book-Files>).

Wenn Sie anschließend über die Sätze iterieren, können Sie Code wie den folgenden nutzen, um die Stoppwörter daraus zu entfernen:

```
words = sentence.split()
filtered_sentence = ""
for word in words:
    if word not in stopwords:
        filtered_sentence = filtered_sentence + word + " "
sentences.append(filtered_sentence)
```

Interpunktionszeichen entfernen

Außerdem kann es sinnvoll sein, Interpunktionszeichen auszufiltern, denn diese können bei der Entfernung von Stoppwörtern Verwirrung stiften. Der obige Code sucht beispielsweise nach Wörtern, die von Leerzeichen umgeben sind. Ein Stoppwort am Satzende oder vor einem Komma würde also »übersehen«.

Dieses Problem lässt sich einfach mit den Umwandlungsfunktionen aus Pythons string-Bibliothek lösen. Dabei müssen Sie allerdings vorsichtig sein, denn es kann sich negativ auf die NLP-Analyse – besonders die Sentiment-Analyse (»Stimmungsanalyse«) – auswirken.

Die Bibliothek beinhaltet eine Konstante namens `string.punctuation`, die eine Liste häufig verwendeter Interpunktionszeichen enthält. Um diese von einem Wort zu entfernen, können Sie wie hier gezeigt vorgehen:

```
import string
table = str.maketrans('', '', string.punctuation)
words = sentence.split()
filtered_sentence = ""
for word in words:
```

Vorwort	15
Einleitung	17
Teil I Mit PyTorch ML-Modelle erstellen	
<hr/>	
1 Einführung in PyTorch	25
Was ist Machine Learning?	25
Grenzen der traditionellen Programmierung	27
Vom Programmieren zum Lernen	29
Was ist PyTorch?	31
PyTorch verwenden	33
Installation von PyTorch auf der Kommandozeile	33
PyTorch in PyCharm verwenden	34
PyTorch in Google Colab verwenden	36
Einstieg ins Machine Learning	38
Sehen, was das Netzwerk gelernt hat	44
Zusammenfassung	45
2 Einführung in Computer Vision	47
Die Funktionsweise der Computer Vision	47
Der Fashion-MNIST-Datensatz	48
Neuronen für Computer Vision	50
Das neuronale Netzwerk entwerfen	51
Der vollständige Code	53
Das Netzwerk trainieren	58
Die Modellausgaben genauer untersuchen	61
Überanpassung	63
Early Stopping (»vorzeitiger Abbruch«)	64
Zusammenfassung	65

3	Über die Grundlagen hinaus: Merkmale in Bildern erkennen	67
	Faltung	68
	Pooling	69
	Convolutional Neural Networks implementieren	71
	Das Convolutional Neural Network untersuchen	74
	Ein CNN für die Unterscheidung zwischen Pferden und Menschen	76
	Der Datensatz »Horses or Humans«	76
	Mit den Daten umgehen	77
	CNN-Architektur für »Horses or Humans«	79
	Validierung für den Datensatz »Horses or Humans«	82
	Die »Horses or Humans«-Bilder testen	84
	Image Augmentation	86
	Transferlernen	90
	Mehrfachklassifizierung	95
	Dropout-Regularisierung	98
	Zusammenfassung	101
4	Daten mit PyTorch verwenden	103
	Einstieg in Datasets	104
	Die FashionMNIST-Klasse erforschen	106
	Generische Dataset-Klassen	106
	ImageFolder	106
	DatasetFolder	107
	FakeData	107
	Custom Splits verwenden	108
	Der ETL-Prozess zur Datenverwaltung im Machine Learning	109
	Die Ladephase optimieren	111
	Die DataLoader-Klasse verwenden	112
	Batching	112
	Daten mischen	112
	Daten parallel laden	113
	Benutzerdefiniertes Daten-Sampling	113
	ETL parallelisieren, um die Trainingsleistung zu steigern	113
	Zusammenfassung	115
5	Einführung in die Verarbeitung natürlicher Sprache	117
	Sprache in Zahlen codieren	117
	Einstieg in die Tokenisierung	118
	Sätze in Sequenzen umwandeln	121
	Stoppwörter entfernen und Text aufräumen	124
	HTML-Tags entfernen	125
	Stoppwörter entfernen	125
	Interpunktionszeichen entfernen	125

Mit echten Datenquellen arbeiten	126
Textdatensätze erhalten	126
Text aus CSV-Dateien lesen	130
Text aus JSON-Dateien auslesen	133
Zusammenfassung	135
6 Stimmungen mit Embeddings programmierbar machen	137
Bedeutung aus Wörtern ableiten	137
Ein einfaches Beispiel: Positives und Negatives	137
Tiefer eintauchen: Vektoren	138
Embedding in PyTorch	139
Embeddings für die Erstellung eines Sarkasmus-Detektors nutzen	140
Überanpassung in Sprachmodellen verringern	143
Die Einzelteile zusammensetzen	155
Das Modell für die Klassifizierung eines Satzes nutzen	156
Embeddings visualisieren	158
Vortrainierte Embeddings einsetzen	161
Zusammenfassung	162
7 Rekurrente neuronale Netzwerke für das Natural Language Processing	165
Grundlagen der Rekurrenz	165
Rekurrenz für die Verarbeitung natürlicher Sprachen erweitern	168
Erstellung eines Text-Classifiers mit RNNs	170
LSTMs stapeln (Stacking)	173
Vortrainierte Embeddings mit RNNs verwenden	181
Zusammenfassung	186
8 Mit Machine Learning Texte erzeugen	187
Sequenzen in Eingabesequenzen umwandeln	188
Das Modell erstellen	193
Text erzeugen	195
Das nächste Wort vorhersagen	196
Vorhersagen kombinieren, um Text zu erzeugen	197
Den Datensatz erweitern	200
Die Modellarchitektur verbessern	202
Embedding-Dimensionen	202
Die LSTMs initialisieren	202
Variable Lernrate	203
Die Daten verbessern	204
Zeichenbasierte Encodierung	206
Zusammenfassung	208

9	Sequenz- und Zeitreihendaten verstehen	209
	Häufige Attribute von Zeitreihen	210
	Trend	210
	Saisonalität	211
	Autokorrelation	211
	Rauschen	212
	Techniken für die Vorhersage von Zeitreihen	213
	Naive Vorhersage zur Ermittlung einer Grundlinie	213
	Vorhersagegenauigkeit messen	215
	Weniger naive Vorhersagen: Verwendung eines gleitenden Mittelwerts (Moving Average)	215
	Verbesserung der Moving-Average-Analyse	216
	Zusammenfassung	217
10	Erstellung von ML-Modellen für die Vorhersage von Sequenzen	219
	Ein »Windowed Dataset« erzeugen	220
	Eine »Sliding Window«-Version des Zeitreihendatensatzes erstellen	223
	Erstellung und Training eines DNN für die Vorhersage aus Sequenzdaten	226
	Die Ergebnisse des DNN auswerten	228
	Die Lernrate anpassen	231
	Zusammenfassung	231
11	Faltungsbasierte und rekurrente Verfahren für Sequenzmodelle	233
	Faltung für Sequenzdaten	233
	Faltungen programmieren	234
	Mit den Conv1-D-Hyperparametern experimentieren	239
	NASA-Wetterdaten verwenden	242
	GIS-Daten in Python importieren	243
	Verwendung von RNNs für die Sequenzmodellierung	246
	Einen größeren Datensatz untersuchen	248
	Andere rekurrente Verfahren einsetzen	251
	Dropouts verwenden	252
	Bidirektionale RNNs verwenden	254
	Zusammenfassung	256

Teil II Generative KI einsetzen

12	Konzepte der Inferenz	259
	Tensoren	259
	Bilddaten	260
	Textdaten	262
	Tensoren als Ausgabe eines Modells	264
	Zusammenfassung	266

13	PyTorch-Modelle für den produktiven Betrieb bereitstellen	267
	Einführung in TorchServe	268
	TorchServe einrichten	270
	Die Umgebung vorbereiten	270
	Die config.properties-Datei anlegen	270
	Das Modell definieren	271
	Die Handler-Datei anlegen	272
	Das Modellarchiv anlegen	274
	Den Server starten	275
	Inferenz testen	277
	Weitere Schritte	279
	Flask als Server nutzen	279
	Eine Umgebung für Flask einrichten	280
	Einen Flask-Server in Python erstellen	280
	Zusammenfassung	281
14	Modelle von Drittanbietern und zentrale Modellverzeichnisse	283
	Der Hugging-Face-Hub	284
	Den Hugging-Face-Hub benutzen	285
	Ein Modell von Hugging Face nutzen	290
	Der PyTorch-Hub	292
	Die PyTorch-Vision-Modelle verwenden	292
	Verarbeitung natürlicher Sprachen (NLP)	295
	Andere Modelle	295
	Zusammenfassung	295
15	Transformer und Transformers	297
	Das Konzept der Transformer verstehen	297
	Encoder-Architekturen	298
	Die Decoder-Architektur	305
	Die Encoder-Decoder-Architektur	311
	Die Transformers-API	313
	Einstieg in Transformers	314
	Grundkonzepte	315
	Pipelines	315
	Tokenizer	317
	Zusammenfassung	321
16	LLMs mit eigenen Daten verwenden	323
	Feintuning eines LLM	323
	Einrichtung und Abhängigkeiten	324
	Die Daten laden und untersuchen	325
	Modell und Tokenizer initialisieren	325

Preprocessing der Daten	326
Die Daten zusammenführen	326
Metriken definieren	327
Das Training konfigurieren	327
Den Trainer initialisieren	328
Training und Auswertung	329
Das Modell speichern und testen	330
Ein LLM per Prompt-Tuning optimieren	331
Die Daten vorbereiten	332
Die DataLoader anlegen	333
Das Modell definieren	333
Das Modell trainieren	336
Auswertung während des Trainings	338
Trainingskennzahlen ausgeben	339
Die Prompt-Embeddings speichern	340
Inferenz mit dem Modell durchführen	340
Zusammenfassung	343
17 LLMs mit Ollama bereitstellen	345
Installation und Einstieg in die Arbeit mit Ollama	346
Ollama als Server betreiben	349
Applikationsentwicklung mit einem Ollama-LLM	351
Das Szenario	352
Ein Python-Skript als Machbarkeitsstudie	353
Eine Web-App für Ollama erstellen	356
Die Datei app.js	357
Die Datei index.html	359
Zusammenfassung	360
18 Einführung in RAG	363
Was ist RAG?	365
Einstieg in die Arbeit mit RAG	366
Ähnlichkeit verstehen	367
Die Datenbank anlegen	368
Eine Ähnlichkeitssuche durchführen	370
Die Einzelteile zusammensetzen	371
RAG-Inhalte mit einem LLM nutzen	372
Gehostete Modelle nutzen	376
Zusammenfassung	377

19 Einsatz generativer Modelle mit Hugging-Face-Diffusers	379
Was sind Diffusion-Modelle?	379
Die Hugging-Face-Diffusers-Bibliothek	382
Bild-zu-Bild mit Diffusers	385
Inpainting mit Diffusers	387
Zusammenfassung	390
20 Generative Bildmodelle mit LoRA und Diffusers optimieren	391
LoRA mit Diffusers trainieren	392
Diffusers laden	392
Daten zur Feinabstimmung von LoRA laden	393
Ein Modell mit Diffusers optimieren	396
Das Modell veröffentlichen	398
Mit optimierter LoRA ein Bild erzeugen	400
Zusammenfassung	403
Index	405

A

- Abfragevektor (Q)
 - Cross-Attention 313
 - Heads 299
 - maskierte Self-Attention 307
 - Self-Attention-Schicht 299
- Adam, Optimizer 56, 57, 81
 - Amsgrad als alternative Implementierung 143
 - Lernrate, Parameter anpassen für 143, 175, 231
 - Zeitreihen-DNN 227
 - Lernrate anpassen 231
- Ähnlichkeit 367
- Aktivierungsfunktionen 52
 - ReLU 52
 - Feedforward-Netzwerk 300
- Amsgrad-Optimizer 143
- »An Empirical Exploration of Recurrent Network Architectures« (Jozefowicz et al.) 202
- APIs
 - Custom Splits für Datensätze 108
 - Diffusers-Bibliothek *siehe* Diffusers-Bibliothek (Hugging Face)
 - Hugging-Face-API für Anmeldung 382
 - RAG API von LangChain 366
 - Transformers-API von Hugging Face 290, 313
 - siehe auch* also transformers library (HuggingFace)
- Architektursuche, neuronal 239
 - Suchraum 240
- »A Theoretically Grounded Application of Dropout in Recurrent Neural Networks« (Gal und Ghahramani) 253

- »Attention Is All You Need« (Vaswani et al.) 297
- Audiodatensätze in torchaudio-Bibliothek 104
- Auffüllung 123
- Autokorrelation, in Zeitreihendaten 211
- AutoTokenizer-Klasse 317
- Average-Pooling 70

B

- Backpropagation (Rückwärtspropagation) 42, 56
 - Vektoren für Embedding-Schicht 140
- Batches, Daten aufteilen in
 - DataLoader 112
 - Pipelining 111
- BCELoss (binäre Kreuzentropie), Verlustfunktion 81
- BeautifulSoup für Entfernung von HTML-Tags aus Text 131
- Bedeutungen von Wörtern 137
 - Embeddings 139
 - Embedding-Größe, als vierte Wurzel der Vokabulargröße 149
 - Embeddings visualisieren 158
 - Ergebnisse, Vermeidung von Überanpassung 155
 - Sarkasmus-Detektor 140
 - Sarkasmus-Detektor, RNNs für 170
 - Sätze mit Sarkasmus-Detektor klassifizieren 156
 - Überanpassung 142
 - Überanpassung, reduziert 143
 - vortrainiert, notwendige Tokenizer-Aktualisierung 161
 - vortrainierte Embeddings 161

- CNN, untersuchen mit torchsummary-Bibliothek 74
 - CNN implementieren 71
 - Dropout-Regularisierung 98
 - Faltungen 68, 90
 - Faltungen mit Pooling 69
 - Image-Augmentation-Verfahren 86
 - Pferde oder Menschen 76
 - Pferde oder Menschen, einzelnes Ausgabeneuron 80, 95
 - Pooling 69
 - Testbilder, gleiche Größe wie Trainingsbilder 85
 - über 67
 - Neuronen für Computer Vision 50
 - Parameter 50
 - »sehen« lernen 51
 - online verfügbarer Code für »Horses or Humans« und »Dogs vs. Cats« 95
 - torchvision-Bibliothek 103
 - Transferlernen 90
 - »Dogs vs. Cats«-Datensatz (Kaggle) 93
 - Google Inception, an ImageNet trainiert 90
 - Inception-Modell anpassen 92
 - über 32, 89
 - Überanpassung 63
 - über 47
 - config.properties-Datei für TorchServe 270
 - Convolutional Neural Networks (CNNs)
 - Faltungen 68, 90
 - Fashion-MNIST-Daten, Vergleich mit »Horses or Humans« 79
 - Implementierung 71
 - Pferde oder Menschen 76
 - CNN-Architektur 79, 90
 - Daten vorbereiten 77
 - mit Bildern testen 84
 - Validierung beim Training 82
 - Pooling 69
 - Transferlernen, Architektur für 91
 - untersuchen mit torchsummary-Bibliothek 74
 - Zeitreihendaten und Sequenzen 233
 - 1-D-faltungsbasierte Schicht 235, 239
 - Architektursuche, neuronal 239
 - Faltungen programmieren 234
 - »Sliding Windows«-Datensatz, Online-codebeispiele 234
 - über 67
 - CPU (Central Processing Unit) 32
 - Extraktion und Transformation als Teil des ETL-Prozesses 111
 - Verwendung von, PyTorch konfigurieren für 34
 - Cross-Attention 311
 - CSV-Dateien, als Quelle für Textdaten 130
 - NASA, monatliche Wetterdaten, Download 242
 - über 131
 - CUDA (cu) 37
 - model.to(»cuda«) für Beschleuniger 111
 - PyTorch aktualisieren 38
- ## D
- DataLoader 112
 - Batching 112
 - Daten laden, Parallelisierung 113
 - num_workers, Parameter 113
 - Pipelining 111
 - Daten mischen 112
 - für Erstellung zufälliger Daten-Stichproben 113
 - vorherige Entfernung von Validierungs- und Testdaten 222
 - Daten-Sampling, benutzerdefiniert 113
 - torch.utils.data.Sampler, Basisklasse 113
 - einen Datensatz verwenden 105
 - FakeData 107
 - Fashion-MNIST-Daten 53
 - »Horses or Humans«-Datensatz 79, 86
 - ImageFolder als 96, 106, 110
 - Unterverzeichnisse, Zuweisung als Bild-Labels 77, 106
 - DatasetFolder 107
 - Daten aus Datei extrahieren per transform 107
 - Verzeichnisse, Nutzung für Labels 107
 - Datenfenster 204, 219
 - Daten mischen mit DataLoader 112
 - vorherige Entfernung von Validierungs- und Testdaten 222
 - Daten parallel laden, mit DataLoader 113
 - num_workers-Parameter 113
 - Daten-Sampling, benutzerdefiniert 113
 - torch.utils.data.Sampler, Basisklasse 113
 - Datensätze
 - als Teil von PyTorch 33, 103
 - Bilddaten 260
 - komprimiert 260

- Variationen zwischen Pixelwerten 261
- Bilder auf Pixabay.com 84
- CIFAR10 113
- computergenerierte Bilder für Training 77, 86
- Custom Splits 108
 - random_split, Funktionsweise 109
- DataLoader 105
- Datenfenster 219
- Datenschutz mithilfe von Ollama 351
- »Dogs vs. Cats« von Kaggle 93
- Einstieg 104
 - eigene Datensätze, mit linearer Beziehung 104
 - eigener Datensatz, Erstellung 104
- Erstellung mit PyTorch-Datentyp Tensor-Dataset 222
- ETL-Prozess zur Handhabung 109
 - Ladephase optimieren 111
- Fashion MNIST, Vergleich mit »Horses or Humans« 79
- Fashion-MNIST-Datensatz 48
 - Beispiel für Datensatzinhalt 48
 - DataLoader 58
 - in eindimensionales Array »verflachen« 52
 - mit Labels versehen 77
 - train-Parameter für Training, Vergleich mit Test-/Validierungsdaten 106
- generische Klassen 106
 - DatasetFolder 107
 - FakeData 107
 - ImageFolder 106
 - über 106
- ImageNet-Datensatz für Google Inception 91
- IMDb-Filmkritiken, Textdatensatz 126
- KNMI Climate Explorer 248
- Misato, digitale Influencerin 393
- NASA-Wetterdaten 242
 - Daten in Python einlesen 243
- Pferde und Menschen 76
 - computergenerierte Bilder 77, 86
 - Daten vorbereiten 77
 - Unterverzeichnisse anstelle von Labels 77
 - Validierungsdatensatz 82
- PyTorch-Datensatzwerkzeuge 103
- Rock, Paper, Scissors 95
- Roman, Volltext 353

- Tensor, Datentyp für Ein- und Ausgabe von Modellen 259
- torchaudio-Bibliothek 104
- torchtext-Bibliothek 104
- torch.utils.data.Dataset-Bibliothek 103, 104
 - eigene Datensätze, mit linearer Beziehung 104
 - eigener Datensatz, Erstellung 104
- torchvision-Bibliothek 103
- Training, Validierung und Testing 82
- Training und Testing 55
- transform-Parameter 55, 106
- Verarbeitung natürlicher Sprachen (NLP) 126
 - Datenfenster, für Textdaten 204
 - JSON-Dateien 133
 - News Headlines Dataset for Sarcasm Detection 133, 140
 - Onlinebuchkapitel 125
 - Sentiment Analysis in Text 130
 - Shakespeare, Gesamtwerk 207
 - Text aus CSV-Dateien 130
 - Textdatensätze 126
- DB Browser für SQLite 370
- Decoder-Architektur 305
 - Addition und Normalisierung 309
 - Encoder-Decoder-Architektur 311
 - Feedforward-Schicht 309
 - gierig, Vergleich mit Top-k-Decodierung 310
 - maskierte Multihed-Attention 307
 - Residualverbindung 308
 - Token und Positionscodierung 306
- Deep Learning
 - Tensoren für numerische Daten 259
 - Unsicherheit in online verfügbaren Informationen zu Deep Learning 253
- »Deep Learning Specialization« (Coursera von Ng) 227
- Deep Neural Networks (DNNs)
 - Zeitreihendaten 226
 - Ergebnisse auswerten 228
- »Delving Deep into Rectifiers« (He et al.) 203
- Differenzbildung 216
- Diffusers-Bibliothek (Hugging Face)
 - Diffusionsmodelle, Funktionsweise 379
 - generative Bildmodelle optimieren
 - Bilderzeugung 400
 - Daten erhalten 393
 - Diffusers erhalten 392
 - Feinabstimmung des Modells 396

- Modell veröffentlichen 398
 - über 391
 - Installation 382
 - LoRA, mit Diffusers trainiert 392
 - Bilderzeugung 400
 - über 391
 - Modelle auf Hugging-Face-Website 383
 - Modell von Hugging Face verwenden 291
 - Onlineinformation zu Pipeline 383
 - Stable-Diffusion-Modelle 380
 - Stable Diffusion 3.5 382, 383
 - über Diffusion 379
 - Verwendung 382
 - Bild-zu-Bild 384
 - Inpainting 387
 - Zugriffsbeschränkungen 382
 - über 284
 - »Dogs vs. Cats«-Datensatz (Kaggle) 93
 - Onlinecodebeispiele 95
 - »Dropout: A Simple Way to Prevent Neural Networks from Overfitting« (Srivastav) 98
 - Dropout-Regularisierung 98, 151
 - Dropout-Implementierung in PyTorch 100
 - LSTM-Stacking 178
 - rekurrentes Dropout für RNNs 252
 - weitere Informationen online 253
 - Überanpassung in Zeitreihen 252
- ## E
- eigenes feinjustiertes generatives Bildmodell veröffentlichen 398
 - Embedding Projector 158
 - Embeddings 139
 - Embedding-Größe, als vierte Wurzel der Vokabulargröße 149
 - Embeddings visualisieren 158
 - OpenAIEmbeddings-Klasse 367
 - OPENAI_API_KEY-Umgebungsvariable 369
 - PDF, Umwandlung in Embeddings 368
 - Sarkasmus-Detektor 140
 - Ergebnisse, Vermeidung von Überanpassung 155
 - Pooling 141
 - RNNs für Erstellung von Text-Classifizier 170
 - Sätze mit Modell klassifizieren 156
 - Überanpassung 142
 - Self-Attention-Schicht in Transformer 299
 - Textdaten 262
 - Überanpassung, reduziert 143
 - Dropout 151
 - Embedding-Dimensionen 149
 - Lernrate anpassen 143
 - Modellarchitektur 150
 - Regularisierung 153
 - Satzlänge 154
 - über Überanpassung 143
 - Vokabulargröße 145
 - Vektoren von Decoder in Transformer 306
 - vortrainierte Embeddings 161
 - OPENAI_API_KEY-Umgebungsvariable 369
 - OpenAIEmbeddings-Klasse 367
 - PDF, Umwandlung in Embeddings 368
 - rekurrente neuronale Netzwerke mit 181
 - Tokenizer, Aktualisierung für Übereinstimmung mit Regeln 161
 - Wortbedeutungen 137
 - über 137
 - Embeddings visualisieren 158
 - Encoder-Architekturen 298
 - Feedforward-Netzwerkschicht 300
 - Schichten normalisieren 302
 - Self-Attention-Schicht 299
 - wiederholte Encoder-Schichten 304
 - Encoder-Decoder-Architektur 311
 - Epoche, Trainingsschleife 58
 - Erkennung von Merkmalen (Features)
 - Dropout-Regularisierung 98
 - Dropout-Implementierung in PyTorch 100
 - Faltungen 68
 - mit Pooling 69
 - über Faltung 67, 90
 - Image-Augmentation-Verfahren 86
 - Implementierung 71
 - CNN, untersuchen mit torchsummary-Bibliothek 74
 - CNNs 67
 - Mehrfachklassifizierung 95
 - Onlinebilder zum Testen 98
 - »Stein, Schere, Papier«, Datensatz 95
 - Pferde oder Menschen 76
 - CNN-Architektur 79, 90
 - Datensatz 76
 - Daten vorbereiten 77
 - falsche Klassifizierung 85, 89
 - mit Bildern testen 84
 - Onlinecodebeispiele 95

- Validierung beim Training 82
- Pooling 69
- Transferlernen 90
 - »Dogs vs. Cats«-Datensatz (Kaggle) 93
 - Google Inception, an ImageNet trainiert 90
- Inception-Modell, Anpassung 92
 - über 32, 89
- über 67
- ETL *siehe* Extraktion, Transformation, Laden (ETL)
- EulerAncestralDiscreteScheduler 400
- Extraktion, Transformation, Laden (ETL) 109
 - Ladephase optimieren 111
 - Parallelisierung zur Verbesserung von Trainingsleistung 113

F

- fairseq-Modelle 295
- FakeData 107
 - nur Bilddaten 107
- Faltungen 68
 - Implementierung 71
 - Pooling mit 69
 - über 67, 90
- Fashion MNIST-Datensatz
 - transform-Parameter 106
- Fashion-MNIST-Datensatz 48
 - Beispiel für Datensatzinhalt 48
 - Computer Vision, Herausforderungen 47
 - Convolutional Neural Networks (CNN) 72
 - DataLoader 58, 103
 - faltungs-basiertes neuronales Netzwerk
 - untersuchen mit torchsummary-Bibliothek 74
 - in eindimensionales Array »verflachen« 52
 - Modellausgaben untersuchen 61
 - Pixelwerte zwischen 0 und 255 48, 50
 - Normalisierung 55
 - Training bis zur gewünschten Genauigkeit 64
 - train-Parameter für Training, Vergleich mit Test-/Validierungsdaten 106
 - transform-Parameter 55
 - Überanpassung 63
 - vollständiger Code für Modelltraining 53
 - Fashion-MNIST-Datensatz geladen 54
- Feature-Maps 81
 - aktiviert 81
- Feedforward-Netzwerk-(FFN-)Schicht
- Decoder 309
- Encoder 300
- Feintuning für generative Bildmodelle 391
 - Daten erhalten 393
 - Diffusers erhalten 392
 - Feinabstimmung des Modells 396
 - LoRA, mit Diffusers trainiert 392
 - Bilderzeugung 400
 - über 391
 - Modell veröffentlichen 398
 - über 391
- Fibonacci-Folge 166
- Flask 279

G

- Gal, Yarin 253
- Gamma-Werte 303
- Gated Recurrent Units (GRUs) 251
- gemma2:2b unter Ollama 346
- Genauigkeit
 - Fashion-MNIST-Datensatz 58
 - Training bis zur gewünschten Genauigkeit 64
 - Geschwindigkeit oder Genauigkeit 52
 - Lernprozess 41
- generative KI
 - Anpassung von Bildmodellen
 - Bilderzeugung 400
 - Daten erhalten 393
 - Diffusers erhalten 392
 - Feinabstimmung des Modells 396
 - Modell veröffentlichen 398
 - über 391
 - Diffusers-Bibliothek von Hugging Face
 - Bild-zu-Bild 384
 - Diffusionsmodelle, Funktionsweise 379
 - Inpainting 387
 - Installation 382
 - Modelle auf Hugging-Face-Website 383
 - Modell von Hugging Face verwenden 291
 - Onlineinformation zu Pipeline 383
 - Stable Diffusion 3.5 382, 383
 - über Diffusion 379
 - Verwendung 382
 - über 284
 - LoRA, mit Diffusers trainiert 392
 - Bilderzeugung 400
 - über 391

- Modelle zur Bilderzeugung, autoregressiv 382
- Retrieval-Augmented Generation (RAG)
 - Definition 365
 - Einstieg in RAG 366
 - RAG-Inhalte, Verwendung mit einem LLM 372
 - über 363
- Stable-Diffusion-Modelle 380
 - Diffusionsmodelle, Funktionsweise 379
 - Onlineinformationen 292
 - Stable Diffusion 3.5 382, 383
 - über Diffusion 379
- Textgenerator
 - Datensätze erweitern 200
 - Daten verbessern 204
 - Eingabesequenzen und Labels 188
 - Modellarchitektur verbessern 202
 - Modell erstellen 193
 - nächstes Wort vorhersagen 196
 - Seed Text 188, 196
 - Sunspring, Onlinebeispiel 200
 - Texterzeugung 195
 - variable Lernrate (VLR) 203
 - Vorhersagen kombinieren, um Text zu erzeugen 197
 - Wortsalat, Erzeugung von 199
 - zeichenbasierte Encodierung 206
 - über 187
- Transformer
 - »Attention Is All You Need« (Vaswani et al.) 297
 - Decoder-Architektur 305
 - Encoder-Architekturen 298
 - Encoder-Decoder-Architektur 311
 - über 187
- Transformers-Bibliothek von Hugging Face
 - API 313
 - Installation 314
 - Modell von Hugging Face verwenden 290
 - Pipelines 315
 - Token für Zugriff auf Modelle 314
 - Tokenizer 317
 - vortrainierte Tokenizer 119
 - über 284, 297
- über 187
- Gewichte
 - Faltungen 68
 - Regularisierung, zur Vermeidung von Überanpassung 153
 - von Neuronen gelernt 44
 - Gewichte anzeigen in PyTorch 44
- Gewichtsverfall (Weight Decay) 154
- Ghahramani, Zoubin 253
- »gierige« Decodierung 310
- GISS *siehe* Goddard Institute for Space Studies (GISS), Wetterdaten
- GitHub
 - Code für Vergleich von Pferden und Menschen bzw. Hunden und Katzen 95
 - Handler für MNIST-Bilder, Beispiel 279
 - »Horses or Humans«-Colab-Notebook testen 84
 - Kaiming-Initialisierung von linearen Schichten, Code für 203
 - RNNs für Sequenzmodellierungscode 247
 - Roman, Volltext 353
 - Shakespeare, Texterzeugung 207
 - »Sliding Windows«-Technik für Faltung von Sequenzen 234
- gleitender Mittelwert, für Zeitreihen-Vorhersage 215
 - Differenzbildung zur Verbesserung von 216
- Global Vectors for Word Representation (GloVe), vortrainierte Embeddings 161, 181
- Goddard Institute for Space Studies (GISS), Wetterdaten 242
 - CSV-Download von monatlichen Daten 242
 - Daten in Python einlesen 243
 - RNNs für Sequenzmodellierung 246
- Google Colab
 - Aktualisierung von PyTorch 38
 - »Horses or Humans«-Notebook testen 84
 - Hugging Face, Zugangstoken 288, 382
 - PyTorch, Einführung 36
 - PyTorch aktualisieren 38
 - Shakespeare, Texterzeugung 207
 - verfügbare Beschleuniger 200
 - über 36
- Google Inception
 - entwickelt für n Neuronen, um n Kategorien auszugeben 93
 - Training an ImageNet-Datensatz 91, 93
 - Transferlernen 90
- Google TensorFlow 260
 - im Kontext von Machine Learning 41
- GPT-Modell 187
 - ChatOpenAI-Klasse 376

- RAG-Inhalt mit gehostetem LLM 376
- GPU (Graphics Processing Unit) 32
 - CUDA-Nvidia-Bibliothek 37
 - für aufwendige Berechnungen in ETL-Prozess 111
- Google Colab GPU, Backend 36
- .to(device) für Beschleuniger 111
- torch.tensor, optimiert für 260
- Gewichtsverfall (»Weight Decay«) 154

H

- He, Kaiming 203
- Heads für Self-Attention 299
- He-Initialisierung von linearen Schichten 203
- »Hello World«, PyTorch-Code 38
- Hilfsfunktion für Wortfrequenz, Code 145
- HTML-Tags, aus Text entfernen 125
 - mit BeautifulSoup 131
- Hugging-Face-Hub 284
 - API für Anmeldung 382
 - Diffusers-Bibliothek 284
 - Bild-zu-Bild 384
 - Diffusionsmodelle, Funktionsweise 379
 - generative Bildmodelle optimieren 391
 - Inpainting 387
 - Installation 382
 - Modelle auf Hugging-Face-Website 383
 - Modell von Hugging Face verwenden 291
 - Onlineinformation zu Pipeline 383
 - Stable Diffusion 3.5 382, 383
 - Stable-Diffusion-Modelle 380
 - Stable-Diffusion-Modelle *siehe auch* fine-generative Bildmodelle anpassen 380
 - Verwendung 382
 - eigenes generatives Bildmodell veröffentlichen 398
 - Erlaubnis zur Nutzung von Modellen 287, 314, 382
 - Misato (digitale Influencerin), Datensatz 393
 - Modelle von, Verwendung 290
 - nutzen 285
 - Scheduler 400
 - Tokens 285, 314
 - Transformers-Bibliothek 284, 297
 - API 313
 - Installation 314
 - Pipelines 315

- Tokenizer 317
- Tokens für Zugriff auf Modelle 314
- vortrainierte Tokenizer 119
- Zugangstoken, Verwendung in Code 289
- Zugangstoken in Google Colab 288, 382
- Zugangstoken in Python-Code 314
- Zugangstoken in Umgebungsvariable angeben 314
- Hyperparameter 52
 - Anpassung 52
 - Parameter, Vergleich mit 52

I

- Image-Augmentation-Verfahren 86
- ImageFolder 106
 - als DataLoader 96, 106, 110
 - eigener Index 107
 - Unterklasse von DatasetFolder 107
- ImageNet-Datensatz für Google Inception 91
- Inception (Google) *siehe* Google Inception
- Inferenz
 - Bilddaten 260
 - komprimiert 260
 - Variationen zwischen Pixelwerten 261
 - Fashion-MNIST-Datensatz 58
 - Prompt-Tuning von LLMs 340
 - ResNet, Modell 293
 - Tensor als Datentyp für Ein- und Ausgabe 259
 - in/aus Modell 264
 - über Tensoren 259
 - Textdaten 262
 - TorchServe, Inferenz testen 277
 - über 32, 44
- Inpainting mit Diffusers 387
- Interpunktionszeichen, aus Text entfernen 125

J

- Jozefowicz, Rafal 202
- JSON
 - Arrays 133
 - in Objekten enthalten 133
 - Datenquellen, für Verarbeitung natürlicher Sprachen 133
 - News Headlines Dataset for Sarcasm Detection 133
 - Stanford Question Answering Dataset 133
 - »Gedächtnis« der RAG-Konversation 374
 - JSON-Dateien lesen 134

- Code für Textbereinigung 134
- Syntaxgrundlagen 133
- K**
- Kaggle »Dogs vs. Cats«-Datensatz 93
 - Onlinecodebeispiele 95
- Kaiming-Initialisierung von linearen Schichten 203
- Kale, Satyen 143
- Kleidung *siehe* Fashion MNIST-Datensatz
- KNMI Climate Explorer, Datensatz 248
- Korpus 119
- Kosinus-Ähnlichkeit 367
- Kumar, Sanjiv 143
- künstliches Verstehen 363, 364
- K-Vektor *siehe* Schlüssel-Vektor (K)
- L**
- LangChain-RAG-API 366
 - ChatOpenAI-Klasse 376
- Large Language Models (LLMs) *siehe* LLMs (Large Language Models)
- Lernprozess
 - Fashion-MNIST-Datensatz
 - Code für Verlustberechnung 57
 - Testdaten, Inferenz und Genauigkeit 58
 - im Kontext von Machine Learning 41
- Lernrate (LR)
 - Anpassung für Zeitreihen-DNN 231
 - Hyperparameter für Optimizer 143
 - Stacked LSTMs optimieren 175
 - Überanpassung reduzieren durch Anpassung von 143
 - variable Lernrate (VLR) 203
- lineare Schichten
 - Decoder 309
 - einzelne Schicht 40
 - Kaiming-Initialisierung 203
 - mehrere Schichten 51
 - Parameter (1,1) 40
- Linearität oder Nichtlinearität 300
- Linux Foundation 31
- llama3.2, Ausführung mit Ollama 347
- LLMs, Feintuning 323
 - Daten laden und untersuchen 325
 - Daten zusammenführen 326
 - Einrichtung und Abhängigkeiten 324
 - Kennzahlen definieren 327
 - Modell speichern und testen 330
 - Modell und Tokenizer initialisieren 325
 - Preprocessing der Daten 326
 - Trainer initialisieren 328
 - Training konfigurieren 327
 - Training und Auswertung 329
- LLMs, Halluzinationen von 200
 - Modell nicht mit privaten Daten trainiert 363
- LLMs (Large Language Models)
 - Feintuning 323
 - Daten laden und untersuchen 325
 - Daten zusammenführen 326
 - Einrichtung und Abhängigkeiten 324
 - Kennzahlen definieren 327
 - Modell speichern und testen 330
 - Modell und Tokenizer initialisieren 325
 - Preprocessing der Daten 326
 - Trainer initialisieren 328
 - Training konfigurieren 327
 - Training und Auswertung 329
- Halluzinationen 200
 - Modell nicht mit privaten Daten trainiert 363
- Ollama, Plattform
 - App erstellen, die Ollama nutzt 351
 - Einstieg 346
 - gemma2:2b ausführen 346
 - llama3.2, Ausführung 347
 - Servermodus 349
 - über 345
- Prompt-Tuning 331
 - Auswertung beim Training 338
 - das Modell definieren 333
 - das Modell trainieren 336
 - DataLoader anlegen 333
 - Daten vorbereiten 332
 - Inferenz 340
 - Prompt-Embeddings speichern 340
 - Soft-Prompts 331
 - Trainingskennzahlen, Bericht 339
 - über Prompt-Tuning 331
- RAG-Inhalt, verwendet mit 372
 - »Gedächtnis« der Konversation 374
 - gehostete Modelle 376
 - Modelle, per Ollama installiert 375
- und Verarbeitung natürlicher Sprachen (NLP) 117
- Logits 56
- Long Short-Term Memory (LSTM) 168
 - Bidirektionalität 169, 194
 - Onlineinformationen 168
 - Stacking (»stapeln«) 173

- Verwendung durch Text-Classifer 170
 - Dropout in Stacked LSTMs 178
 - Lernrate zur Verbesserung von Stacked LSTMs 175
 - Stacking von LSTMs 173
- Verwendung in Textgenerator 193
 - Modellarchitektur verbessern 202
- Zeitreihendaten 251
- LoRA (Low-Rank Adaptation)
 - Modelle von Drittanbietern 283
 - Training mit Diffusers 392
 - Bilderzeugung 400
 - Daten erhalten 393
 - Diffusers erhalten 392
 - Feinabstimmung des Modells 396
 - Modell veröffentlichen 398
 - über 391
- LR (Lernrate) *siehe* Lernrate (LR)
- LSTM *siehe* Long Short-Term Memory (LSTM)
- Lua 31

M

- Machine Learning (ML)
 - Beschreibung
 - Grenzen traditioneller Programmierung 27
 - TensorFlow 30
 - traditionelle Programmierung 25
 - Weiterentwicklung der Programmierung in Richtung Lernen 29
 - Einstieg 38
 - Lernprozess 41
 - Optimizer 41
 - PyTorch, Code für den Anfang 38
 - sehen, was das Netzwerk gelernt hat 44
 - Vorhersagen 44
 - Machine Learning Operations (MLOps) 267
 - MAE *siehe* mittlerer absoluter Fehler (MAE)
 - Manning, Christopher 181
 - MAR-Dateien (Modellarchitektur) 269
 - maskierte Self-Attention 307
 - Max-Pooling 69
 - Mehrfachklassifizierung 95
 - »Stein, Schere, Papier«, Datensatz 95
 - Onlinebilder zum Testen 98
 - Min-Pooling 70
 - Misato (digitale Influencerin), Datensatz 393
 - Misra, Rishabh 133
 - mittlerer absoluter Fehler (MAE)
 - Genauigkeit von Zeitreihenvorhersage messen 230
 - mittlerer absoluter Fehler (MAE) für Zeitreihen 215
 - mittlerer quadratischer Fehler (MSE) für Zeitreihen 215
 - MLOps 267
 - ML *siehe* Machine Learning (ML)
 - MNIST-Fashion-Datensatz (Modified National Institute of Standards and Technology) 48
 - Beispiel für Datensatzinhalt 48
 - Computer Vision, Herausforderungen 47
 - Convolutional Neural Network (CNN) untersuchen mit torchsummary-Bibliothek 74
 - DataLoader 58
 - faltungsbasiertes neuronales Netzwerk 72
 - Image-Handler, Onlineinformationen 279
 - in eindimensionales Array »verflachen« 52
 - Modellausgaben untersuchen 61
 - Pixelwerte zwischen 0 und 255 48, 50
 - Normalisierung 55
 - Training bis zur gewünschten Genauigkeit 64
 - train-Parameter für Training, Vergleich mit Test-/Validierungsdaten 106
 - transform-Parameter 55, 106
 - Überanpassung 63
 - vollständiger Code für Modelltraining 53
 - Fashion-MNIST-Datensatz geladen 54
 - Mobilgeräte 32, 33
 - Modelle
 - Architektur untersuchen 92
 - beschneiden 92
 - einfrühen 92
 - Tensor, Datentyp für Ein- und Ausgabe 259
 - in/aus Modell 264
 - über Training 32
 - und Inferenz 32
 - Vorhersagen 44
 - vortrainiert *siehe* vortrainierte Modelle
 - Modelle und Sammlungen von Drittanbietern
 - Hugging-Face-Hub 284
 - Hugging-Face-Hub *siehe auch* Hugging Face Hub 284
 - PyTorch-Hub 292
 - Verarbeitung natürlicher Sprachen (NLP) 295
 - Vision-Modelle, verwenden 292
 - weitere Modelltypen 295

über 283
Modelle zur Bilderzeugung, autoregressiv 382
model.to(device) für Beschleuniger 111
Moore'sches Gesetz 209
MSELoss, Verlustfunktion 227
MSE *siehe* mittlerer quadratischer Fehler (MSE)
multivariate Zeitreihen 210

N

NASA-Wetterdaten 242
 CSV-Download von monatlichen Daten
 242
 Daten in Python einlesen 243
 RNNs für Sequenzmodellierung 246
neuronale Netzwerke
 Architektursuche, neuronal 239
 Suchraum 240
 Dropout-Regularisierung zur Vermeidung
 von Überanpassung 98
 Dropout-Implementierung in PyTorch
 100
 Einstieg in ML 39
 Neuronen 39
Fashion-MNIST-Datensatz
 Code für Verlustberechnung 57
 neuronales Netzwerk entwerfen 51, 55
 Testdaten, Inferenz und Genauigkeit 58
Implementierung 71
 CNN-Architektur für »Horses or Humans«-Datensatz 79, 90
 CNN-Architektur für Transferlernen 91
 CNNs 67
 untersuchen mit torchsummary-Bibliothek 74
Lernprozess 41
normalisierte Werte 55
 Umwandlung von 0-255-Wertebereich
 55
RNNs *siehe* rekurrente neuronale Netzwerke (RNNs)
Neuronen
 binäre Klassifizierung, mit einzelner Neuron 80
 Computer Vision 50
 Parameter 50
 »sehen« lernen 51
 Einstieg in ML 39
 einzelnes Neuron 40
 Gewicht und Bias lernen 44
 PyTorch zeigt Gewichte und Bias an 44

gelernter Bias 44
 PyTorch, Anzeige von Bias 44
Geschwindigkeit oder Genauigkeit 52
rekurrente Neuronen 165
Überspezialisierung als Grund für Überanpassung 98
»News Headlines Dataset for Sarcasm Detection«, Datensatz (Misra) 133
Ng, Andrew 227
Nichtlinearität oder Linearität 300
NLP *siehe* Verarbeitung natürlicher Sprachen (NLP)
nn.CrossEntropyLoss, Verlustfunktion 95
node.js-App mit Anbindung an Ollama 356
 index.html-Datei 359
 node.js, Onlineanleitung 356
normalisierte Werte für neuronale Netzwerke
 55
 Onlineinformationen 55
 Umwandlung von 0-255-Wertebereich 55
NumPy 19

O

Ollama
 App erstellen, die Ollama nutzt 351
 app.js-Datei 357
 index.html-Datei 359
 Python-basierte Machbarkeitsstudie 353
 Szenario 352
 Web-App erstellen 356
Datenschutz 351
Einstieg 346
gemma2.2b ausführen 346
llama3.2, Ausführung 347
RAG-Inhalte, Verwendung in LLMs 375
Servermodus 349
über 345
Onlinere Ressourcen
 »Attention Is All You Need« (Vaswani et al.) 297
Audiodatensätze 104
Bild aus Textinformation erzeugen 292
Bilder auf Pixabay.com 84
Bildgenerator-Modelle auf Hugging Face 383
Codebeispiele für dieses Buch 19
 »Horses or Humans«-Colab-Notebook testen 84
 »Pferde oder Menschen« und »Hunde oder Katzen« 95

- Python-Bibliotheken 81
- RNNs für Sequenzmodellierungscode 247
 - »Sliding Windows«-Technik für Faltung von Sequenzen 234
- config.properties-Datei für TorchServe 270
- Handler für MNIST-Bilder, Beispiel 279
- Kaiming-Initialisierung von linearen Schichten 203
- Long Short-Term Memory (LSTM), Informationen zu 168
- LSTM, interne Neuronentypen, Informationen 202
- Misato, digitale Influencerin 393
- mit Subtoken-Tokenizer erzeugtes Stargate-Drehbuch, Video 318
- NASA-Wetterdaten 242
 - »News Headlines Dataset for Sarcasm Detection«, Datensatz 133
- node.js, Anleitung 356
- normalisierte Daten für neuronale Netzwerke, Information zu 55
- O'Reilly-Website 19
- Ollama 346
- Pipeline-Informationen für Diffusers-API 383
- Python-Installation und Informationen 33
- rekurrente Dropouts, Informationen zu 253
- Roman, Volltext 353
- Scheduler 400
- Shakespeare, Gesamtwerk 207
 - Google Colab, PyTorch-basiert 207
 - Google Colab, TensorFlow-basiert 207
 - »Stein, Schere, Papier«, Bilder für Training 98
- Sunspring, Beispiel für generierten Text 200
- Text aus Buchkapiteln 125
- Textdatensätze 104
- TorchServe 268
- venv 270
- »On the Convergence of Adam and Beyond« (Reddi, Kale und Kumar) 143
- »On the difficulty of training recurrent neural networks« (Pascanu et al.) 202
- OPENAI_API_KEY-Umgebungsvariable 369
- OpenAIEmbeddings-Klasse 367
 - OPENAI_API_KEY 369
 - PDF, Umwandlung in Embeddings 368
- Optimizer
 - Adam 56, 57, 81
 - Amsgrad als Alternative 143

- Lernrate anpassen 143, 231
- Zeitreihen-DNN 227
 - im Kontext von Machine Learning 41
- Lernrate als Hyperparameter 143
- stochastischer Gradientenabstieg 41
- Out-of-Vocabulary-Problem, Umgang mit, durch Hugging-Face-Transformer 317

P

- Parameter
 - Hyperparameter versus 52
 - Neuronen für Computer Vision 50
- Pascanu, Razvan 202
- PDF, laden und in Embeddings umwandeln 368
- Pennington, Jeffrey 181
- Pferde und Menschen, Unterscheidung 76
 - CNN-Architektur 79, 90
 - computergenerierte Bilder für Training 86
 - Datensatz 76
 - computergenerierte Bilder 77, 86
 - Daten vorbereiten 77
 - einzelnes Ausgabeneuron, Grund für binäre Klassifizierung 80, 95
 - Extrahieren, Transformieren, Laden (ETL), Prozess 110
 - mit Bildern testen 84
 - Bilder auf Pixabay.com 84
 - Colab-Notebook auf GitHub 84
 - falsche Klassifizierung 85, 89
 - Image-Augmentation-Verfahren 86
 - Onlinecodebeispiele 95
 - Validierung beim Training 82
- Pferde und Menschen, Unterschiede 76
- pipeline-Klasse in Transformers-API von Hugging Face 290, 315
- Pipelines aus Diffusers-Bibliothek von Hugging Face 291
 - Bild-zu-Bild mit StableDiffusion3Img2Img-Pipeline 385
- Pipelining während Ladephase von ETL-Prozess 111
- Pixel filtern *siehe* Faltung
- Pooling 69
 - Average-Pooling 70
 - Embeddings in NLP 141
 - Faltung mit 69
 - Implementierung 71
 - Max-Pooling 69

- Min-Pooling 70
 - Pools 70
 - Positionscodierung von Transformern 306
 - private Daten
 - Halluzinationen durch 363
 - Ollama, Schutz für 351
 - Prompt 305
 - negativer Prompt für Stable Diffusion 3.5 383
 - Prompt-Tuning von LLMs 331
 - Auswertung beim Training 338
 - das Modell definieren 333
 - das Modell trainieren 336
 - DataLoader anlegen 333
 - Daten vorbereiten 332
 - Inferenz 340
 - Prompt-Embeddings speichern 340
 - Soft-Prompts 331
 - Trainingskennzahlen, Bericht 339
 - über Prompt-Tuning 331
 - PyCharm
 - PyTorch, Ausführung in 34
 - PyTorch, Installation in virtueller Umgebung 34
 - Python
 - Array-Schreibweise, nötig zum Verständnis 19
 - Bibliotheken in Repository zu diesem Buch 81
 - Installations-URL 33
 - json-Bibliothek 134
 - online lernen 33
 - PyTorch, Installation 33
 - PyTorch
 - Aktualisierung in Google Colab 38
 - Ausführung in PyCharm 34
 - Beschreibung 31
 - Bezeichnung als »torch« 31
 - Daten verstehen mithilfe von Tensoren 41
 - eingebaute/beiliegende Datensätze 33
 - Einstieg in ML 38
 - PyTorch, Code für den Anfang 38
 - sehen, was das Netzwerk gelernt hat 44
 - für CPU konfigurieren 34
 - Installation
 - Google Colab 36
 - PyCharm, virtualisierte Umgebung für 34
 - model.to(device) für Beschleuniger 111
 - Optimizer 41
 - Version anzeigen 33, 37
 - PyTorch, Entwicklung durch Meta AI 31
 - PyTorch-Hub 292
 - Verarbeitung natürlicher Sprachen (NLP) 295
 - Vision-Modelle, verwenden 292
 - torchvision, Installation 292
 - torchvision, Version ausgeben 293
 - weitere Modelltypen 295
 - PyTorch-Modelle, Bereitstellung
 - Flask 279
 - TorchServe 268
 - Einrichtung 270
 - über 267
- ## R
- RAG (Retrieval-Augmented Generation)
 - Definition 365
 - Einstieg in RAG 366
 - Ähnlichkeitssuche mit Vektorspeicher 370
 - API von LangChain 366
 - Chroma, Vektorspeicherdatenbank 366
 - Datenbank anlegen 368
 - Einzelteile zusammensetzen 371
 - Erklärung von Ähnlichkeit 367
 - OPENAI_API_KEY-Umgebungsvariable 369
 - OpenAIEmbeddings-Klasse 367
 - RAG-Inhalte, Verwendung mit einem LLM 372
 - »Gedächtnis« der Konversation 374
 - gehostete Modelle 376
 - Modelle, per Ollama installiert 375
 - über 363
 - Rauschen, in Zeitreihendaten 212
 - Reddi, Sashank 143
 - Referenzdaten
 - Ground Truth 53
 - Regularisierung, zur Vermeidung von Überanpassung 153
 - L1-Regularisierung 153
 - L2-Regularisierung 154
 - Rekurrente neuronale Netzwerke (RNNs) 165
 - bidirectional 254
 - Long Short-Term Memory (LSTM) 168
 - Bidirektionalität 169
 - rekurrente Schicht, gleiche Größe wie Embedding 170
 - rekurrentes Dropout für 252
 - Rekurrenz 165

- Rekurrenz, erweitert für Sprache 168
- Sequenzmodellierung
 - bidirektionale RNNs 254
 - KNMI Climate Explorer, Datensatz 248
 - NASA-Wetterdaten 246
- Textklassifizierung mit 170
 - Dropout in Stacked LSTMs 178
 - Lernrate zur Verbesserung von Stacked LSTMs 175
 - Stacking von LSTMs 173
- vortrainierte Embeddings mit 181
- über 165
- rekurrentes Dropout 252
 - weitere Informationen online 253
- ReLU (Rectified Linear Unit), Aktivierungsfunktion 52
 - Feedforward-Netzwerk 300
- ReLU (Rectified Linear Unit) *siehe* ReLU (Rectified Linear Unit), Aktivierungsfunktion 45
- Residualverbindung 308
- ResNet, Modell 293
 - PyTorch-Hub 293
- Retrieval Augmented Generation (RAG) *siehe* RAG (Retrieval Augmented Generation)
- RNNs *siehe* rekurrente neuronale Netzwerke (RNNs)
- Roman, Volltext online verfügbar 353

S

- Saisonalität von Zeitreihendaten 211
- Sammlungen und Verzeichnisse (Hubs)
 - Hugging-Face-Hub 284
 - Hugging-Face-Hub *siehe auch* Hugging Face Hub 284
- PyTorch-Hub 292
 - torchvision, Installation 292
 - torchvision, Version ausgeben 293
 - Verarbeitung natürlicher Sprachen (NLP) 295
 - Vision-Modelle, verwenden 292
 - weitere Modelltypen 295
- über 283
- Sarkasmus-Detektor
 - Embeddings 140
 - positive und negative Werte für Wörter 137
 - RNNs für Erstellung von Text-Classifer 170
 - Sarkasmus-Datensatz 133, 140

- Sätze
 - Eingabesequenzen und Labels aus 188
 - maximale Länge festlegen 154
 - Sequenzen aus 121
 - Textdaten 262
- Scheduler 400
- Schichten
 - 2-D-Daten zu 1-D-Array verflachen 52
 - Einstieg in ML 39
 - Feedforward-Netzwerkschicht
 - Decoder 309
 - Encoder 300
 - linear
 - Decoder 309
 - einzelne Schicht 40
 - Kaiming-Initialisierung 203
 - mehrere Schichten 51
 - Parameter (1,1) 40
 - Sequential, Definition
 - einzelne Schicht 40
 - mehrere Schichten 51
 - Softmax-Schicht
 - Decoder 309, 310
 - Logits 56
 - Self-Attention-Schicht in Transformer 299
 - verborgene Schichten 52
 - Schicht-Normalisierung von Encoder 302
 - Schlüsselvektor (K)
 - Cross-Attention 313
 - Heads 299
 - maskierte Self-Attention 307
 - Self-Attention-Schicht 299
- SciPy
 - ascent-Bild, Beispiel für Max-Pooling 70
 - ascent-Bild für Faltungen 68
- Seed Text für Textgeneratoren 188, 196
- Self-Attention 299
 - Cross-Attention von Encoder-Decoder 311
 - Heads 299
 - maskierte Self-Attention von Decoder 307
 - Self-Attention von Encoder 299
- SentencePiece-Tokenizer 320
- Sentiment-Analyse 137
 - Einfluss von Interpunktion auf Erkennung 125
 - Sentiment-Analyse in Textdatensatz 130
- Sentiment-Analyse in Textdatensatz 130
- Sequential
 - einzelne Schicht, Definition 40
 - Linear-Schichten, Definition 40, 51

- mehrere Schichten, Definition 51
 - Sequenzdaten *siehe* Zeitreihen
 - Sequenzen aus Sätzen
 - Eingabesequenzen und Labels 188
 - Out-of-Vocabulary-Token 122
 - Sprache in Zahlen encodieren 121
 - Sequenz-zu-Sequenz-Architektur 311
 - Shakespeare, Gesamtwerk online 207
 - Google Colab 207
 - sigmoid-Funktion 80
 - Socher, Richard 181
 - Softmax-Schicht
 - Decoder 309, 310
 - Logits 56
 - Self-Attention-Schicht in Transformer 299
 - Soft-Prompts 331
 - Sparseness 149
 - Sprache in Zahlen encodieren 117
 - reale Datenquellen 126
 - JSON-Dateien 133
 - Sentiment-Analyse in Textdatensatz 130
 - Text aus CSV-Dateien 130
 - Textdatensätze 126
 - Sätze in Sequenzen umwandeln 121
 - Auffüllung 123
 - Out-of-Vocabulary-Token 122
 - Text bereinigen 124
 - Code für Textbereinigung in JSON-Datei 134
 - HTML-Tags, Entfernung mit BeautifulSoup 131
 - HTML-Tags entfernen 125
 - Interpunktionszeichen, Entfernung 125
 - Stoppwörter entfernen 125
 - Textdaten 262
 - Token 118
 - Tokenisierung 118
 - eigene Tokenizer 118
 - vortrainierte Tokenizer 119
 - und Embeddings 137
 - zeichenbasierte Encodierung 206
 - Sprache *siehe* Verarbeitung natürlicher Sprachen (NLP)
 - SQLite als Vektorspeicherdatenbank 369
 - SQuAD (Stanford Question Answering Dataset) 133
 - Srivastav, Nitish 98
 - StableDiffusion3Img2ImgPipeline 384
 - Stable-Diffusion-Modelle 380
 - Stable Diffusion 3.5 382
 - negativer Prompt 383
 - StableDiffusion3Pipeline-Klasse 383
 - Stacking, LSTMs 173
 - Stanford Global Vectors for Word Representation (GloVe), vortrainierte Embeddings 161, 181
 - Stanford Question Answering Dataset (SQuAD) 133
 - Stargate-Drehbuch, Video von Tischlesung 318
 - »Stein, Schere, Papier«, Datensatz 95
 - Dropout-Regularisierung, Implementierung 100
 - Onlinebilder zum Testen 98
 - Stochastischer Gradientenabstieg (SGD), Optimizer 41
 - besser bei Eingabe von Batches 112
 - Stoppwörter aus Text entfernen 125
- ## T
- TensorDataset, Datentyp (PyTorch) 222
 - Tensoren 55, 259
 - Fähigkeit, verschiedene Wertetypen zu enthalten 260
 - im Kontext von Machine Learning 41
 - Tensor als Datentyp für Ein- und Ausgabe 259
 - in/aus Modell 264
 - TensorFlow *siehe* Google TensorFlow
 - Tensor Processing Unit (TPU) 32
 - Testing
 - »Horses or Humans«-Modell für Bilder 84
 - Bilder auf Pixabay.com 84
 - Colab-Notebook auf GitHub 84
 - Inferenz und Genauigkeit 58
 - Überanpassung 63
 - Trainingsbilder mit gleicher Größe wie Testbilder 85
 - Vergleich mit Validierung 82
 - Text-Classifer mit RNNs 170
 - rekurrente Schicht, gleiche Größe wie Embedding 170
 - Stacking von LSTMs 173
 - Dropout in Stacked LSTMs 178
 - Lernrate zur Verbesserung von Stacked LSTMs 175
 - Textdaten 262
 - Textdatensätze, für Verarbeitung natürlicher Sprachen 126
 - Datenfenster 204
 - IMDb-Filmkritiken 126
 - Text aus CSV-Dateien 130

- torchtext, veraltet 118, 126
- torchtext-Bibliothek für Textdatensätze 104
- Textgenerator
 - Datensätze erweitern 200
 - Daten verbessern 204
 - Eingabesequenzen und Labels 188
 - generative KI 187
 - Modellarchitektur verbessern 202
 - Embedding-Dimensionen 202
 - LSTMs initialisieren 202
 - variable Lernrate (VLR) 203
 - Modell erstellen 193
 - Seed Text 188, 196
 - Texterzeugung 195
 - nächstes Wort vorhersagen 196
 - Sunspring, Onlinebeispiel 200
 - Vorhersagen zusammenführen 197
 - Wortsalat, Erzeugung von 199
 - über Transformer 187
 - zeichenbasierte Encodierung 206
 - über 187
- .to(device) für Beschleuniger 111
- Tokenisierung 118
 - eigene Tokenizer 118
 - Textdaten 262
 - vortrainierte Tokenizer 119
 - BertTokenizerFast 119
- Tokenizer
 - Byte-Pair-Encoding 320
 - Hugging-Face-Transformers 317
 - SentencePiece 320
 - Stargate-Drehbuch, Subtoken-Tokenizer 318
 - Subword-Tokenisierung, häufigste Vorgehensweise 317
 - WordPiece tokenizer 318
- Tokens 118
 - Prompt 305
 - Self-Attention 299
- Top-k-Decodierung 310
- Torch 31
- torchaudio-Bibliothek 104
- TorchServe 31, 268
 - Einrichtung 270
 - Add-ons und Beispiele 279
 - config.properties, Datei 270
 - das Modell definieren 271
 - die Umgebung vorbereiten 270
 - Handler-Datei 272
 - Inferenz testen 277
 - Modellarchiv 274
 - Server starten 275
 - TorchServe installieren 270
 - Handler für MNIST-Bilder, Beispiel auf GitHub 279
- torchsummary-Bibliothek für Untersuchung von CNN 74
- torch.tensor, Optimierung für Ausführung auf GPUs 260
- torchtext, veraltet 118, 126
- torchtext-Bibliothek 104
- torch.utils.data.Dataset-Bibliothek 104
 - DataLoader 105
 - eigene, mit linearer Beziehung 104
 - eigener Datensatz, Erstellung 104
- torch.utils.data-Namensraum
 - Custom Splits für Datensätze 108
- torch.utils.data.Sampler, Basisklasse für Einbindung eigener Daten 113
- torchvision-Bibliothek 103
 - Installation 292
 - Version anzeigen 293
- torchvision.models-Bibliothek 32
 - Inception V3 91
- TPU (Tensor Processing Unit) 32
 - für aufwendige Berechnungen in ETL-Prozess 111
 - Google Colab, TPU-Backend 36
- Training
 - Computerchips für 32
 - computergenerierte Bilder für 77, 86
 - Early Stopping 241
 - Epoche, Trainingsschleife 58
 - Fashion-MNIST-Datensatz
 - Code für Verlustberechnung 57
 - Testdaten, Inferenz und Genauigkeit 58
 - Training bis zur gewünschten Genauigkeit 64
 - Lernprozess 41
 - Parallelisierung von ETL zur Verbesserung der Trainingsleistung 113
 - Testbilder, gleiche Größe wie Trainingsbilder 85
 - Überanpassung 52, 63
 - Validierung während des Trainings, Testing danach 82
 - verteiltes Training 31
 - PyTorch-Bibliotheken 32
 - über 32
- Training vorzeitig abbrechen (Early Stopping) 241
- Transferlernen 90

- CNN-Architektur 91
- Inception V3 (Google) 90
 - das Modell anpassen 92
 - Training an ImageNet-Datensatz 91
 - über 32, 89, 283
- Transformer
 - »Attention Is All You Need« (Vaswani et al.) 297
 - Decoder-Architektur 305
 - Encoder-Architekturen 298
 - Encoder-Decoder-Architektur 311
 - über 187
- Transformers-API 313
- Transformers-Bibliothek (Hugging Face)
 - API 313
 - Installation 314
 - Modell von Hugging Face verwenden 290
 - Pipelines 315
 - Token für Zugriff auf Modelle 314
 - Tokenizer 317
 - vortrainierte Tokenizer 119
 - über 284, 297
- Trends in Zeitreihendaten 210
- Truncation (»Verkürzung«) 124

U

- Überanpassung 52, 63
 - Dropout, Hilfe bei Vermeidung 98, 151
 - Dropout-Implementierung in PyTorch 100
 - rekurrentes Dropout für RNNs 252
 - Zeitreihendaten 252
 - verringern, in Sprachmodellen 143
 - Dropout 151
 - Embedding-Dimensionen 149
 - Embeddings 142
 - Lernrate anpassen 143
 - Modellarchitektur 150
 - Regularisierung 153
 - Satzlänge 154
 - über Überanpassung 143
 - Vokabulargröße 145
- Umgebungsvariablen
 - Hugging Face, Zugangstoken 314
 - OPENAI_API_KEY 369
- univariate Zeitreihen 210
- Unsicherheit in online verfügbaren Informationen zu Deep Learning 253

V

- Validierung
 - »Horses or Humans«-Datensatz 82
 - Vergleich mit Testing 82
- Vaswani, Ashish 297
- Vektor, für Wert (V)
 - Cross-Attention 313
 - Heads 299
 - maskierte Self-Attention 307
 - Self-Attention-Schicht 299
- Vektoren für Wortbedeutungen
 - Cross-Attention 313
 - Embeddings 139, 140
 - Decoder für Transformer 306
 - Embedding-Dimensionen 149
 - Embedding Projector zur Visualisierung von Embeddings 158
 - PDF umwandeln in 368
 - Kosinus-Ähnlichkeit 367
 - Self-Attention-Vektoren für Tokens 299
 - Heads 299
 - Sparseness 149
 - Textdaten 262
 - über 138
- venv, virtuelle Umgebung 270
- Verarbeitung natürlicher Sprachen (NLP)
 - Embeddings 139
 - Embedding-Größe, als vierte Wurzel der Vokabulargröße 149
 - Embeddings visualisieren 158
 - Ergebnisse, Vermeidung von Überanpassung 155
 - Pooling 141
 - Sarkasmus-Detektor 140
 - Sarkasmus-Detektor, RNNs für 170
 - Sätze mit Sarkasmus-Detektor klassifizieren 156
 - Überanpassung 142
 - Überanpassung, reduziert 143
 - vortrainiert, notwendige Tokenizer-Aktualisierung 161
 - vortrainierte Embeddings 161
 - vortrainierte Embeddings mit RNNs 181
 - über 137, 139
- PyTorch-Hub 295
- reale Datenquellen 126
 - JSON-Dateien 133
 - Sentiment-Analyse in Textdatensatz 130
 - Text aus CSV-Dateien 130

- Textdatensätze 126
 - Rekurrente neuronale Netzwerke (RNNs)
 - 165
 - Long Short-Term Memory (LSTM) 168
 - Rekurrenz 165
 - Rekurrenz, erweitert für Sprache 168
 - Textklassifizierung mit 170
 - vortrainierte Embeddings mit 181
 - über 165
 - Sentiment-Analyse 137
 - Einfluss von Interpunktion auf Erkennung 125
 - Sprache in Zahlen encodieren 117
 - Auffüllung 123
 - Out-of-Vocabulary-Token 122
 - Sätze in Sequenzen umwandeln 121
 - Token 118
 - Tokenisierung 118
 - über 117
 - Text bereinigen
 - Code für Textbereinigung in JSON-Datei 134
 - HTML-Tags, Entfernung mit BeautifulSoup 131
 - HTML-Tags entfernen 125
 - Interpunktionszeichen, Entfernung 125
 - Stoppwörter entfernen 125
 - über 124
 - Textdaten 262
 - Texterzeugung *siehe* Textgenerator
 - Wörter
 - Bedeutung 137
 - Sentiment-Analyse, positiv und negativ 137
 - Vektoren 138, 139, 140
 - zeichenbasierte Encodierung 206
 - über 117
 - verborgene Schichten 52
 - Überanpassung, Verringerung durch Anpassung von 150
 - Verlustfunktion
 - BCELoss 81
 - Computer Vision 51, 56
 - Code für Verlustberechnung 57
 - Testdaten, Inferenz und Genauigkeit 58
 - im Kontext von Machine Learning 40
 - loss.backward-Funktion 42, 56
 - MSELoss für Regressionsprobleme 227
 - nn.CrossEntropyLoss 95
 - »Verschwindende Gradienten«, Problem 309
 - verteiltes Training 31
 - PyTorch-Bibliotheken 32
 - virtuelle Umgebung, venv 270
 - Vokabulareffizienz 317
 - Vokabulargröße und Verringerung von Überanpassung 145
 - Vorhersagen 44
 - Fashion-MNIST-Modell 57, 59
 - Mehrfachklassifizierung 97
 - vortrainierte Embeddings 161
 - OpenAIEmbeddings-Klasse 367
 - PDF, Umwandlung in Embeddings 368
 - OpenAIFEmbeddings-Klasse
 - OPENAI_API_KEY-Umgebungsvariable 369
 - rekurrente neuronale Netzwerke mit 181
 - Tokenizer, Aktualisierung für Übereinstimmung mit Regeln 161
 - vortrainierte Modelle
 - einfrieren 92
 - Modellarchitektur untersuchen 92
 - torchvision.models-Bibliothek 32
 - Inception V3 91
 - Transferlernen 90, 283
 - das Modell anpassen 92
 - über 32, 89
 - über 283
 - V-Vektor *siehe* Wert-Vektor (V)
- ## W
- Web-App auf Basis von Ollama erstellen 356
 - Wörter
 - Bedeutung 137
 - Encoder-Architekturen verstehen 298
 - Sentiment-Analyse, positiv und negativ 137
 - Vektoren 138, 139, 140
 - Embeddings 139
 - Embedding-Größe, als vierte Wurzel der Vokabulargröße 149
 - Embeddings visualisieren 158
 - Ergebnisse, Vermeidung von Überanpassung 155
 - Sarkasmus-Detektor 140
 - Sarkasmus-Detektor, RNNs für 170
 - Sätze mit Sarkasmus-Detektor klassifizieren 156
 - Überanpassung 142
 - Überanpassung, reduziert 143
 - vortrainiert, notwendige Tokenizer-Aktualisierung 161

- vortrainierte Embeddings 161
 - vortrainierte Embeddings mit RNNs
 - 181
 - über 139
 - Hilfsfunktion für Wortfrequenz, Code 145
 - nächstes Wort vorhersagen 196
 - Sunspring, Onlinebeispiel 200
 - Vorhersagen kombinieren, um Text zu erzeugen 197
 - Wortsalat, Erzeugung von 199
 - Stimmungen (Sentiment) erkennen 137
 - Textdaten 262
 - Vokabulargröße und Verringerung von Überanpassung 145
- Y**
- YOLO (You Only Look Once), Modell 292
- Z**
- zeichenbasierte Encodierung 206
- Zeitreihendaten
 - Attribute
 - Autokorrelation 211
 - Rauschen 212
 - Saisonalität 211
 - Trend 210
 - Aufteilung, Erhaltung vollständiger saisonaler Abschnitte 214, 225
 - Dropout als Hilfe gegen Überanpassung 252
 - Faltung für 233
 - 1-D-faltungsbasierte Schicht 235, 239
 - Architektursuche, neuronal 239
 - Faltungen programmieren 234
 - Gated Recurrent Units (GRU) 251
 - KNMI Climate Explorer, Datensatz 248
 - Long Short-Term Memory (LSTM)-Schichten 251
 - multivariate Zeitreihen 210
 - NASA-Wetterdaten 242
 - CSV-Download von monatlichen Daten 242
 - Daten in Python einlesen 243
 - RNNs für Sequenzmodellierung 246
 - RNNs für Sequenzmodellierung
 - bidirektionale RNNs 254
 - KNMI Climate Explorer, Datensatz 248
 - NASA-Wetterdaten 246
 - synthetisch, Erzeugung 213
 - Datenfenster 223
 - Trainingsdaten, Erstellung aus Zeitreihendatensatz 223
 - univariate Zeitreihen 210
 - Vorhersage mit ML-Modellen
 - Datenfenster 219
 - Datenfenster-Datensatz aus synthetischer Zeitreihe 223
 - Ergebnisse auswerten 228
 - Lernrate anpassen 231
 - vollständig verbundenes (Deep) neuronales Netzwerk 226
 - Vorhersagen
 - gleitender Mittelwert, als weniger naives Vorgehen 215
 - gleitender Mittelwert, Verbesserung durch 216
 - naive Vorhersage zur Ermittlung von Grundlinie 213
 - Vorhersagegenauigkeit messen 215 über 209
 - zufällige Stichproben per Shuffling 113

PyTorch für KI und ML

Sie möchten mehr über die Programmierung von KI- und Machine-Learning-Anwendungen erfahren, wissen aber nicht, wo Sie starten sollen? Mit diesem codeorientierten Guide zu PyTorch können Sie sofort KI-Projekte für den realen Einsatz erstellen, denn er legt den Fokus auf die konkrete Praxis. Bestsellerautor Laurence Moroney erklärt die zugrunde liegenden KI-Konzepte dabei verständlich und ohne komplizierte Mathematik.

Von Computer Vision und Natural Language Processing (NLP) über Sequenzmodellierung bis hin zu generativer KI mit Hugging Face Transformers vermittelt Ihnen dieses Buch die jobbezogenen Skills, die in der KI-Entwicklung derzeit am meisten gefragt sind. Darüber hinaus erfahren Sie, wie Sie Ihre Modelle sicher im Web und in der Cloud bereitstellen und produktiv einsetzen.

- **Von ML-Basics bis zu GenAI:** Erkunden Sie die ganze Bandbreite der KI-Programmierung
- **Code-first-Ansatz:** Lernen Sie mithilfe von Code statt durch Mathematik
- **Mit PyTorch arbeiten:** Erstellen Sie KI-Modelle für Computer Vision, Zeitreihenanalysen, NLP und Sequenzmodellierung
- **Hugging-Face-Bibliotheken nutzen:** Vertiefen Sie Ihre Kenntnisse über generative KI-Techniken durch Transformer- und Diffusionsmodelle
- **LLM-Tuning und RAG:** Optimieren Sie LLMs mit Prompt- und Feintuning sowie mit LoRA und setzen Sie RAG-Systeme ein

»Ein Buch, das man lesen sollte, wenn man als Full-Stack-KI-Praktiker arbeiten möchte.«

– Dr. Pin-Yu Chen
IBM Research

»Verständlich, konkret und konsequent auf PyTorch ausgerichtet.«

– Dominic Monn
CEO, MentorCruise.com

»Ich wünsche Ihnen beim Erlernen von PyTorch viel Erfolg. Mit Laurence als Lehrer erwarten Sie große Abenteuer.«

– Andrew Ng
Gründer, DeepLearning.AI

Laurence Moroney ist ein preisgekrönter Researcher und Bestsellerautor von über 20 Büchern. Als KI-Experte mit mehr als 30 Jahren Erfahrung in der Software- und Machine-Learning-Branche engagiert er sich mit großem Enthusiasmus dafür, Softwareentwickler*innen KI und ML verständlich zu vermitteln.



www.dpunkt.de

Euro 44,90 (D)
ISBN 978-3-96009-285-8

Gedruckt in Deutschland
Mineralölfreie Druckfarben
Zertifiziertes Papier