PyTorch
für KI und ML

Das Praxisbuch für generative KI und Machine Learning

DIE LESEPROBE

KAPITEL 4

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 Validierungsdatensatz 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.

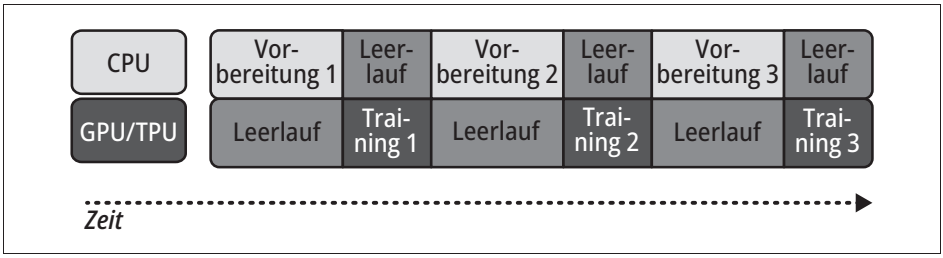

Abbildung 4.1: Nutzung von CPU oder GPU/TPU

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.

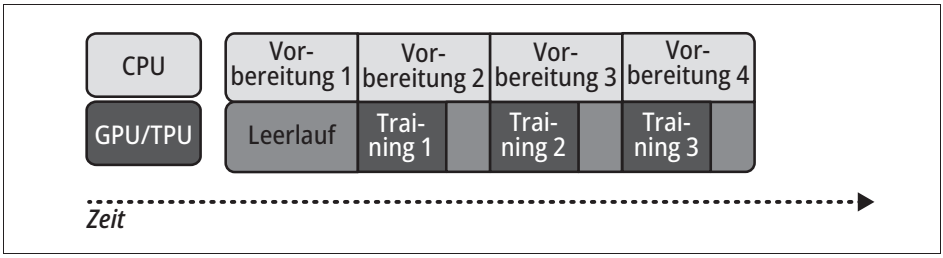

Abbildung 4.2: Pipelining

Hier befindet sich die GPU/TPU im »Leerlauf«, während die CPU den ersten Batch vorbereitet. 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 wahrscheinlich 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 arbeiten: 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 stochastische Gradientenabstieg funktionieren deutlich besser, wenn die Eingaben in Batches (in denen mehrere Einträge zusammengefasst sind) übergeben werden, weil dann genauer gerechnet werden kann. Das Batching kann das Training gerade in größeren Szenarien beschleunigen, in denen die GPU mit festen Speichergrößen verwendet wird. Am effizientesten 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 ausgezeichneter 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 träge 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.

KAPITEL 5

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 Text*körper*, 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 unbekannten 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 »Wurbelspunst«.

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 `<br>` 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:
```