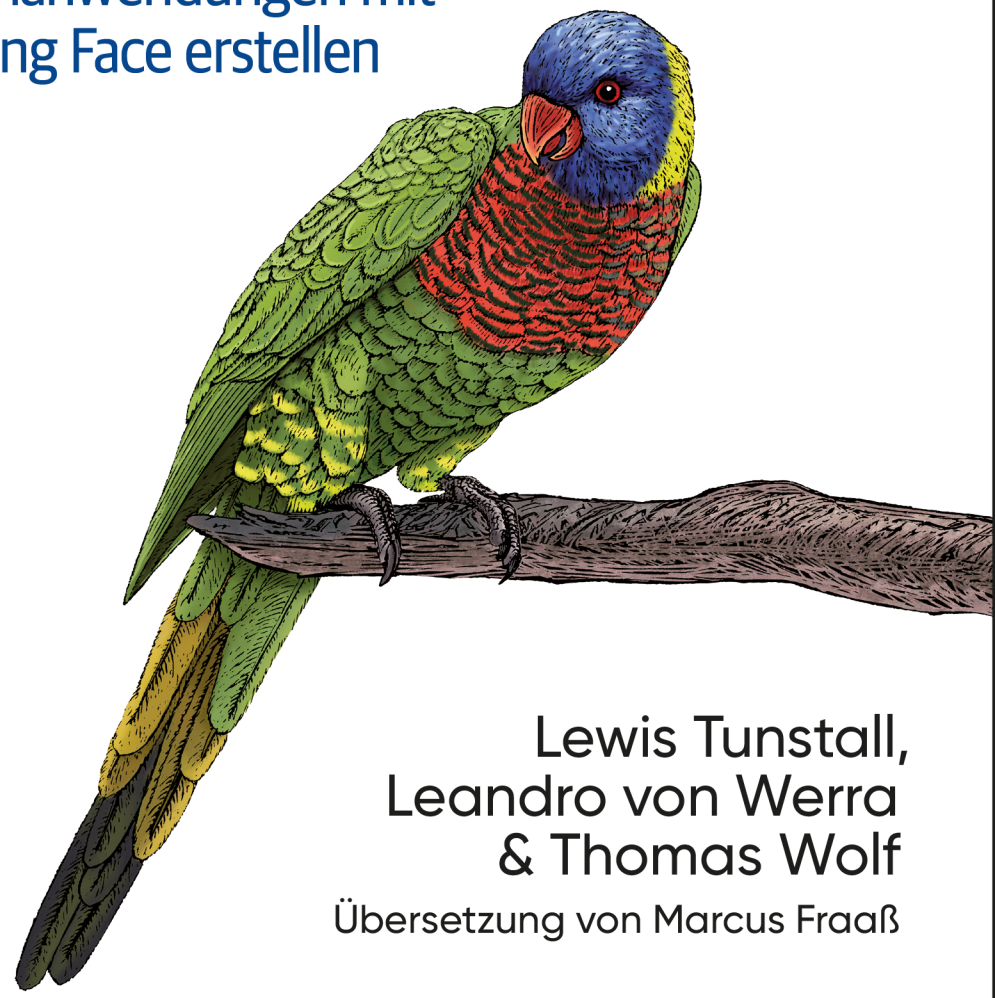
O'REILLY®

Deutsche Ausgabe

Natural Language Processing mit Transformern

Sprachanwendungen mit Hugging Face erstellen



Lewis Tunstall,
Leandro von Werra
& Thomas Wolf

Übersetzung von Marcus Fraaß

Coypright und Urheberrechte:
Die durch die dpunkt.verlag GmbH vertriebenen digitalen Inhalte sind urheberrechtlich geschützt. Der Nutzer verpflichtet sich, die Urheberrechte anzuerkennen und einzuhalten. Es werden keine Urheber-, Nutzungs- und sonstigen Schutzrechte an den Inhalten auf den Nutzer übertragen. Der Nutzer ist nur berechtigt, den abgerufenen Inhalt zu eigenen Zwecken zu nutzen. Er ist nicht berechtigt, den Inhalt im Internet, in Intranets, in Extranets oder sonst wie Dritten zur Verwertung zur Verfügung zu stellen. Eine öffentliche Wiedergabe oder sonstige Weiterveröffentlichung und eine gewerbliche Vervielfältigung der Inhalte wird ausdrücklich ausgeschlossen. Der Nutzer darf Urheberrechtsvermerke, Markenzeichen und andere Rechtsvorbehalte im abgerufenen Inhalt nicht entfernen.

Natural Language Processing mit Transformern

Sprachanwendungen mit Hugging Face erstellen

Lewis Tunstall, Leandro von Werra, Thomas Wolf
Vorwort von Aurélien Géron

Deutsche Übersetzung von Marcus Fraaß

O'REILLY®

Lewis Tunstall, Leandro von Werra, Thomas Wolf

Lektorat: Alexandra Follenius
Übersetzung: Marcus Fraaß
Copy-Editing: Claudia Lötschert, *www.richtiger-text.de*
Satz: III-satz, *www.drei-satz.de*
Herstellung: Stefanie Weidner
Umschlaggestaltung: Karen Montgomery, Michael Oréal, *www.oreal.de*
Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

ISBN:
Print 978-3-96009-202-5
PDF 978-3-96010-712-5
ePub 978-3-96010-713-2
mobi 978-3-96010-714-9

1. Auflage 2023
Translation Copyright für die deutschsprachige Ausgabe © 2023 dpunkt.verlag GmbH
Wieblinger Weg 17
69123 Heidelberg

Authorized German translation of the English edition of *Natural Language Processing with Transformers, Revised Edition* ISBN 9781098136796 © 2022 by Lewis Tunstall, Leandro von Werra, and Thomas Wolf. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Dieses Buch erscheint in Kooperation mit O'Reilly Media, Inc. unter dem Imprint »O'REILLY«. O'REILLY ist ein Markenzeichen und eine eingetragene Marke von O'Reilly Media, Inc. und wird mit Einwilligung des Eigentümers verwendet.

Hinweis:
Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.

PEFC zertifiziert
Das Papier für dieses Buch stammt aus nachhaltig bewirtschafteten Wäldern und kontrollierten Quellen.
PEFC
PEFC/04-31-0810 www.pefc.de

Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: *kommentar@oreilly.de*.

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.
Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.
Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autoren noch Verlag noch Übersetzer können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

5 4 3 2 1 0

Inhalt

Vorwort 11

Einführung 15

1 Hallo Transformer 23
Das Encoder-Decoder-Framework 24
Der Attention-Mechanismus 26
Einsatz von Transfer Learning im NLP 28
Die Transformers-Bibliothek von Hugging Face: die Lücke schließen 32
Die Anwendungsmöglichkeiten von Transformern im Überblick 33
Textklassifizierung 33
Named Entity Recognition 34
Question Answering 35
Automatische Textzusammenfassung (Summarization) 36
Maschinelle Übersetzung (Translation) 37
Textgenerierung 37
Das Ökosystem von Hugging Face 38
Der Hugging Face Hub 39
Die Tokenizers-Bibliothek von Hugging Face 41
Die Datasets-Bibliothek von Hugging Face 41
Die Accelerate-Bibliothek von Hugging Face 42
Die größten Herausforderungen im Zusammenhang mit Transformer-Modellen 42
Zusammenfassung 44

2 Textklassifizierung 45
Der Datensatz 46
Ein erster Blick auf die Datasets-Bibliothek von Hugging Face 47
Dataset-Objekte in DataFrames überführen 50
Ein Blick auf die Verteilung der Kategorien 51
Wie lang sind unsere Tweets? 52

Vom Text zu Tokens 54
Tokenisierung auf der Ebene von Zeichen (Character Tokenization) 54
Tokenisierung auf der Ebene von Wörtern (Word Tokenization) 57
Tokenisierung auf der Ebene von Teilwörtern (Subword Tokenization) 58
Den gesamten Datensatz tokenisieren 60
Trainieren eines Textklassifikators 62
Transformer-Modelle als Feature-Extraktoren 64
Feintuning von Transformer-Modellen 72
Zusammenfassung 81

3 Die Anatomie von Transformer-Modellen 83
Die Transformer-Architektur 83
Der Encoder 86
Self-Attention 87
Die Feed-Forward-Schicht 98
Layer Normalization integrieren 99
Positional-Embeddings 100
Einen Head zur Klassifizierung hinzufügen 103
Der Decoder 104
Transformer-Modelle im Überblick 106
Die drei Entwicklungsstränge von Transformer-Modellen 107
Rein Encoder-basierte Transformer-Modelle 108
Rein Decoder-basierte Transformer-Modelle 110
Encoder-Decoder-basierte Transformer-Modelle 112
Zusammenfassung 114

4 Multilinguale Named Entity Recognition 115
Der Datensatz 116
Multilinguale Transformer-Modelle 120
Ein genauerer Blick auf die Tokenisierung 122
Die Tokenizer-Pipeline 122
Der SentencePiece-Tokenizer 124
Transformer-Modelle für die Named Entity Recognition 125
Der Aufbau der Model-Klasse der Transformers-Bibliothek 127
Bodies und Heads 127
Ein selbst definiertes Modell zur Klassifizierung von Tokens erstellen 128
Ein selbst definiertes Modell laden 130
Tokenisierung von Texten für die Named Entity Recognition 132
Qualitätsmaße 135

Feintuning eines XLM-RoBERTa-Modells 136
Fehleranalyse 138
Sprachenübergreifender Transfer 146
Wann ist ein Zero-Shot-Transfer sinnvoll? 147
Modelle für mehrere Sprachen gleichzeitig feintunen 149
Interaktion mit den Modell-Widgets 152
Zusammenfassung 153

5 Textgenerierung 155
Die Herausforderungen bei der Generierung von kohärenten Texten 157
Greedy-Search-Decodierung 159
Beam-Search-Decodierung 163
Sampling-Verfahren 167
Top-k- und Nucleus-Sampling 169
Welcher Ansatz zur Decodierung ist der beste? 173
Zusammenfassung 173

6 Automatische Textzusammenfassung (Summarization) 175
Der CNN/DailyMail-Datensatz 176
Pipelines für die automatische Textzusammenfassung 177
Ein einfacher Ansatz zur Textzusammenfassung 178
GPT-2 178
T5 178
BART 179
PEGASUS 180
Verschiedene Zusammenfassungen vergleichen 181
Evaluierung der Qualität von generierten Texten 182
BLEU 183
ROUGE 187
Evaluierung des PEGASUS-Modells auf dem CNN/DailyMail-Datensatz 190
Trainieren eines Modells zur Generierung von Zusammenfassungen . . . 192
Das PEGASUS-Modell auf dem SAMSum-Datensatz evaluieren . . . 193
Das PEGASUS-Modell feintunen 194
Zusammenfassungen von Dialogen erstellen 198
Zusammenfassung 199

7 Question Answering 201
Aufbau eines rezensionsbasierten QA-Systems 202
Der Datensatz 203
Antworten aus einem Text extrahieren 209
Die Haystack-Bibliothek zum Aufbau einer QA-Pipeline verwenden 217

Verbesserung unserer QA-Pipeline . . . 226
Den Retriever evaluieren . . . 226
Den Reader evaluieren . . . 233
Domain Adaptation . . . 235
Die gesamte QA-Pipeline evaluieren . . . 240
Jenseits des extraktiven QA . . . 241
Zusammenfassung . . . 243

8 Effizientere Transformer-Modelle für die Produktion . . . 247
Die Intentionserkennung als Fallstudie . . . 248
Eine Benchmark-Klasse zur Beurteilung der Performance erstellen . . . 250
Verkleinerung von Modellen mithilfe der Knowledge Distillation . . . 255
Knowledge Distillation im Rahmen des Feintunings . . . 256
Knowledge Distillation im Rahmen des Pretrainings . . . 258
Eine Trainer-Klasse für die Knowledge Distillation erstellen . . . 259
Ein geeignetes Modell als Ausgangspunkt für das Schüler-Modell wählen . . . 260
Geeignete Hyperparameter mit Optuna finden . . . 265
Unser destilliertes Modell im Vergleich . . . 267
Beschleunigung von Modellen mithilfe der Quantisierung . . . 268
Das quantisierte Modell im Vergleich . . . 275
Optimierung der Inferenz mit ONNX und der ONNX Runtime . . . 276
Erhöhung der Sparsität von Modellen mithilfe von Weight Pruning . . . 282
Sparsität tiefer neuronaler Netze . . . 283
Weight-Pruning-Methoden . . . 283
Zusammenfassung . . . 287

9 Ansätze bei wenigen bis keinen Labels . . . 289
Erstellung eines GitHub-Issues-Tagger . . . 291
Die Daten beschaffen . . . 292
Die Daten vorbereiten . . . 292
Trainingsdatensätze erstellen . . . 297
Unterschiedlich große Trainingsdatensätze erstellen . . . 299
Implementierung eines naiven Bayes-Klassifikators als Baseline . . . 300
Ansätze, wenn keine gelabelten Daten vorliegen . . . 303
Ansätze, wenn nur wenige gelabelte Daten zur Verfügung stehen . . . 313
Datenaugmentierung . . . 313
Embeddings als Nachschlagetabelle verwenden . . . 316
Ein standardmäßiges Transformer-Modell feintunen . . . 327
In-Context- und Few-Shot-Learning auf Basis von Prompts . . . 330
Ungelabelte Daten nutzbar machen . . . 331
Ein Sprachmodell feintunen . . . 332
Einen Klassifikator feintunen . . . 335

Fortgeschrittene Methoden . . . 337
Zusammenfassung . . . 339

10 Transformer-Modelle von Grund auf trainieren . . . 341
Große Datensätze und wie sie beschafft werden können . . . 342
Herausforderungen beim Aufbau eines großen Korpus . . . 343
Einen eigenen Codedatensatz erstellen . . . 346
Mit großen Datensätzen arbeiten . . . 349
Datensätze zum Hugging Face Hub hinzufügen . . . 352
Erstellung eines Tokenizers . . . 354
Das Tokenizer-Modell . . . 355
Die Leistung eines Tokenizers beurteilen . . . 356
Ein Tokenizer für die Programmiersprache Python . . . 357
Einen Tokenizer trainieren . . . 362
Einen selbst erstellten Tokenizer auf dem Hub speichern . . . 366
Ein Modell von Grund auf trainieren . . . 367
Verschiedene Pretraining-Objectives im Überblick . . . 367
Das Modell initialisieren . . . 370
Den Dataloader implementieren . . . 371
Die Trainingsschleife einrichten . . . 374
Der Trainingslauf . . . 382
Ergebnisse und Analyse . . . 383
Zusammenfassung . . . 388

11 Künftige Herausforderungen . . . 389
Skalierung von Transformer-Modellen . . . 389
Skalierungsgesetze . . . 391
Herausforderungen bei der Skalierung . . . 393
Attention Please! – Den Attention-Mechanismus effizienter gestalten . . . 395
Sparse-Attention . . . 396
Linearisierte Attention . . . 398
Jenseits von Textdaten . . . 399
Computer Vision . . . 400
Tabellen . . . 403
Multimodale Transformer . . . 406
Speech-to-Text . . . 406
Computer Vision und Text . . . 409
Wie geht es weiter? . . . 415

Index . . . 417

Vorwort

Während Sie diese Zeilen lesen, geschieht ein Wunder: Die Schnörkel auf dieser Seite formen sich zu Wörtern, Konzepten und Emotionen, während sie sich ihren Weg durch Ihren Kortex bahnen. Meine Gedanken vom November 2021 sind nun erfolgreich in Ihr Gehirn eingedrungen. Sollte es ihnen gelingen, Ihre Aufmerksamkeit zu erregen und lange genug in dieser rauen und hart umkämpften Umgebung zu überleben, haben sie vielleicht sogar die Chance, sich weiter zu verbreiten, wenn Sie diese Gedanken mit anderen teilen. Dank der Sprache sind Gedanken zu übertragbaren und hochansteckenden Gehirnbakterien geworden – und ein Impfstoff ist nicht in Sicht.

Glücklicherweise sind die meisten Gehirnbakterien harmlos,[1] und einige sind sogar überaus nützlich. Tatsächlich formen diese menschlichen Gehirnbakterien zwei unserer wertvollsten Schätze: Wissen und Kultur. So wie wir ohne gesunde Darmbakterien nicht richtig verdauen können, können wir ohne gesunde Gehirnbakterien nicht richtig denken. Die meisten Ihrer Gedanken stammen gar nicht von Ihnen: Sie sind in vielen anderen Gehirnen entstanden, gewachsen und haben sich entwickelt, bevor sie Sie infiziert haben. Wenn wir also intelligente Maschinen erschaffen möchten, müssen wir einen Weg finden, auch sie zu infizieren.

Die gute Nachricht ist, dass sich in den letzten Jahren ein weiteres Wunder ereignet hat: Dank mehrerer Durchbrüche auf dem Gebiet des Deep Learning wurden leistungsfähige Sprachmodelle hervorgebracht. Da Sie dieses Buch lesen, sind Ihnen wahrscheinlich schon einige erstaunliche Ausführungen dieser Sprachmodelle begegnet. So z.B. GPT-3, das nach einer kurzen Texteingabe (einem sogenannten Prompt) wie »ein Frosch trifft ein Krokodil« eine ganze Geschichte niederschreiben kann. Obwohl es noch nicht ganz Shakespeare ist, ist es manchmal schwer zu glauben, dass diese Texte von einem künstlichen neuronalen Netz geschrieben wurden. Tatsächlich hilft mir das Copilot-System von GitHub beim Schreiben dieser Zeilen: Sie werden nie erfahren, wie viel ich davon wirklich selbst verfasst habe.

1 Tipps zur Gehirnhygiene finden Sie in CGP Greys hervorragendem Video über Memes (*https://youtu.be/rE3j_RHkqJc*).

Die Revolution geht weit über die Generierung von Texten hinaus. Sie umfasst den gesamten Bereich der maschinellen Verarbeitung natürlicher Sprache (engl. Natural Language Pocessing, NLP) – im Deutschen auch als Maschinelle Sprachverarbeitung oder Computerlinguistik (CL) bezeichnet –, von der Textklassifizierung bis zur automatischen Zusammenfassung von Texten, maschinellen Übersetzung, Beantwortung von Fragen, Chatbots, dem Verstehen natürlicher Sprache (engl. Natural Language Understanding, NLU) und mehr. Wo immer Sprache, gesprochener oder geschriebener Text vorkommt, gibt es eine Anwendung im NLP. Sie können bereits Ihr Telefon nach dem morgigen Wetter fragen, mit einem virtuellen Helpdesk-Assistenten chatten, um ein Problem zu beheben, oder aussagekräftige Ergebnisse von Suchmaschinen erhalten, die Ihre Anfrage wirklich zu verstehen scheinen. Doch diese Technologie ist so neu, dass uns das Beste wahrscheinlich erst noch bevorsteht.

Wie die meisten Fortschritte in der Wissenschaft beruht auch die jüngste Revolution im Bereich des NLP auf der harten Arbeit von Hunderten von unbesungenen Helden. Für den Erfolg sind allerdings drei wesentliche Faktoren ausschlaggebend:

- Der *Transformer* ist eine Architektur für neuronale Netze, die im Jahr 2017 in einer bahnbrechenden Arbeit mit dem Titel »Attention Is All You Need« (*https://arxiv.org/abs/1706.03762*) von einem Team von Forschern von Google vorgeschlagen wurde. In nur wenigen Jahren hat sie sich durchgesetzt und die vorherigen Architekturen, die in der Regel auf rekurrenten neuronalen Netzen (engl. Recurrent Neural Networks, RNNs) basieren, verdrängt. Die Transformer-Architektur eignet sich hervorragend zur Erfassung von Mustern in langen Datensequenzen und zur Bewältigung riesiger Datensätze – so gut, dass ihr Einsatz inzwischen weit über den Bereich des NLP hinausgeht und beispielsweise auch in der Verarbeitung von Bildern (engl. Image Processing) Anwendung findet.
- Im Rahmen der meisten Projekte können Sie auf keinen großen Datensatz, mit dem Sie ein Modell von Grund auf trainieren können, zugreifen. Glücklicherweise ist es oftmals möglich, ein Modell herunterzuladen, das bereits auf einem generischen Datensatz *vortrainiert* wurde: Sie müssen es dann nur noch auf Ihrem eigenen (bedeutend kleineren) Datensatz feintunen. Modelle vorzutrainieren, ist seit Anfang der 2010er-Jahre in der Bildverarbeitung gang und gäbe, im NLP beschränkte es sich jedoch auf kontextlose Worteinbettungen (d.h. dichtbesetzte [engl. dense] Vektordarstellungen einzelner Wörter). So hatte zum Beispiel das englische Wort »bear« die gleiche vortrainierte Einbettung (engl. Embedding) im Zusammenhang mit der Nutzung von »teddy bear«, also dem Plüschbären, und »to bear«, was so viel wie aushalten bzw. ertragen bedeutet. Im Jahr 2018 wurden dann in mehreren Veröffentlichungen vollwertige Sprachmodelle vorgeschlagen, die für eine Vielzahl von NLP-Aufgaben vortrainiert und feingetunt werden können. Dadurch änderte sich das gesamte Vorgehen grundlegend.

- Sogenannte *Model Hubs* wie der von Hugging Face sind ebenfalls »Game-Changer«. Anfangs wurden fertig vortrainierte Modelle einfach irgendwo veröffentlicht, sodass es nicht einfach war, das geeignete Modell zu finden, das man benötigte. Murphys Gesetz sorgte dafür, dass PyTorch-Benutzer nur TensorFlow-Modelle ausfindig machen konnten, und umgekehrt. Und wenn man ein Modell gefunden hatte, war es nicht immer einfach, herauszufinden, wie man es feintunen konnte. Hier kommt die Transformers-Bibliothek von Hugging Face ins Spiel: Sie ist quelloffen, unterstützt sowohl TensorFlow als auch PyTorch und erlaubt es, ein hochmodernes, vortrainiertes Modell vom Hugging Face Hub herunterzuladen, es für Ihre Aufgabe zu konfigurieren, es auf Ihrem Datensatz feinzutunen und es zu evaluieren. Die Bibliothek findet zunehmend mehr Verwendung: Im vierten Quartal 2021 wurde sie von mehr als fünftausend Unternehmen und Einrichtungen genutzt und über vier Millionen Mal pro Monat mit dem Paketverwaltungsprogramm `pip` installiert. Darüber hinaus erweitern sich die Bibliothek und ihr Ökosystem über NLP hinaus, sodass inzwischen auch Bildverarbeitungsmodelle (engl. Image Processing) verfügbar sind. Ebenso können Sie zahlreiche Datensätze vom Hub herunterladen, mit denen Sie Ihre Modelle trainieren oder evaluieren können.

Was kann man sich also noch wünschen? Nun, dieses Buch! Es wurde von den Open-Source-Entwicklern von Hugging Face verfasst – einschließlich des Begründers der Transformers-Bibliothek – und das merkt man: Die Breite und Tiefe der Informationen, die Sie auf diesen Seiten finden, ist erstaunlich. Es deckt von der Transformer-Architektur selbst bis hin zur Transformers-Bibliothek und dem gesamten Ökosystem, das sie umgibt, alles ab. Besonders gut gefallen hat mir der praxisnahe Ansatz: Während Sie das Buch durcharbeiten, können Sie gleichzeitig den gesamten Code, der in Jupyter Notebooks vorliegt, direkt nachvollziehen und ausführen. Alle Codebeispiele sind direkt auf den Punkt gebracht und einfach zu verstehen. Die Autoren bringen langjährige Erfahrung im Trainieren sehr großer Transformer-Modelle mit und liefern eine Fülle von Tipps und Tricks, mit denen Sie alles effizient zum Laufen bringen. Und nicht zuletzt ist ihr Schreibstil geradlinig und lebendig: Es liest sich wie ein Roman.

Kurzum, ich habe dieses Buch sehr gerne gelesen und ich bin sicher, Sie werden ebenfalls Gefallen daran finden. Jeder, der an der Entwicklung von Produkten mit modernsten Sprachverarbeitungsfunktionen interessiert ist, sollte es lesen. Es ist randvoll mit all den nützlichen Gehirnbakterien!

Aurélien Géron

November 2021, Auckland (Neuseeland)

Einführung

Seit ihrer Einführung im Jahr 2017 haben sich Transformer-Modelle zum De-facto-Standard für die Bewältigung einer Vielzahl von Aufgaben im Bereich der natürlichen Sprachverarbeitung (engl. Natural Language Processing, NLP) sowohl in der Wissenschaft als auch in der Industrie entwickelt. Ohne dass Sie es bemerkt haben, haben Sie heute wahrscheinlich bereits mit einem Transformer interagiert: Google verwendet heutzutage das BERT-Modell, um die Suchanfragen der Nutzer besser zu verstehen und so die Suchmaschine zu verbessern. Auch die Modelle der GPT-Familie von OpenAI haben in den Mainstream-Medien wiederholt für Schlagzeilen gesorgt, weil sie in der Lage sind, wie von Menschen hervorgebrachte Texte und Bilder zu generieren.[1] Mithilfe dieser Transformer-basierten Modelle werden Anwendungen wie GitHub's Copilot (*https://copilot.github.com*) betrieben, die, wie in Abbildung 1-1 gezeigt, einen bloßen Kommentar in Quellcode umwandeln können, mit dem automatisch ein neuronales Netz (engl. Neural Network) für Sie erstellt wird!

Weshalb also haben Transformer das Gebiet fast über Nacht verändert? Wie bei vielen großen wissenschaftlichen Durchbrüchen handelte es sich um die Synthese mehrerer Ideen, wie *Attention*, *Transfer Learning* und der *Skalierung neuronaler Netze*, die zu dieser Zeit in der Forschungsgemeinschaft kursierten.

Aber wie nützlich sie auch sein mögen – um in der Industrie Fuß zu fassen, braucht jede ausgefallene neue Methode Werkzeuge, die sie zugänglich machen. Die 🤗[2] Transformers-Bibliothek (*https://oreil.ly/Z79jF*) und das sie umgebende Ökosystem sind genau darauf ausgerichtet und erleichtern Praktikern, Modelle zu verwenden, zu trainieren und sie mit anderen zu teilen. Dies hat die Verbreitung von Transformer-Modellen stark begünstigt, und die Bibliothek wird heute von über fünftausend Unternehmen und Einrichtungen genutzt. In diesem Buch zeigen wir Ihnen, wie Sie diese Modelle für praktische Anwendungen trainieren und optimieren können.

1 NLP-Forscher neigen dazu, die von ihnen entwickelten Modelle nach Figuren aus der *Sesamstraße* zu benennen. Was diese Akronyme bedeuten, erklären wir in Kapitel 1.

2 Sie erkennen es bestimmt, dies ist das Emoji *Hugging Face*.

```
1   # Create a convolutional neural network to classify MNIST images in PyTorch.
2   class ConvNet(nn.Module):
        def __init__(self):
            super(ConvNet, self).__init__()
            self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
            self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
            self.conv2_drop = nn.Dropout2d()
            self.fc1 = nn.Linear(320, 50)
            self.fc2 = nn.Linear(50, 10)

        def forward(self, x):
            x = F.relu(F.max_pool2d(self.conv1(x), 2))
            x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
            x = x.view(-1, 320)
            x = F.relu(self.fc1(x))
            x = F.dropout(x, training=self.training)
            x = self.fc2(x)
            return F.log_softmax(x, dim=1)
```

Abbildung 1-1: Ein Beispiel für GitHub's Copilot-System, das infolge einer kurzen Beschreibung der Aufgabe einen Vorschlag für die gesamte Klasse liefert (alles, was auf class folgt, wurde automatisch generiert)

An wen richtet sich dieses Buch?

Dieses Buch richtet sich an Data Scientists und Machine Learning Engineers, die vielleicht schon von den jüngsten Durchbrüchen mit Transformern gehört haben, denen aber ein detaillierter Leitfaden fehlt, um diese Modelle an ihre eigenen Anwendungsfälle anzupassen. Das Buch ist nicht als Einführung in das Machine Learning zu verstehen. Wir gehen davon aus, dass Sie mit der Programmierung in Python vertraut sind und ein grundlegendes Verständnis von Deep-Learning-Frameworks wie PyTorch (*https://pytorch.org*) oder TensorFlow (*https://www.tensorflow.org*) haben. Wir gehen auch davon aus, dass Sie einige praktische Erfahrungen mit dem Trainieren von Modellen auf GPUs besitzen. Obwohl sich das Buch auf die PyTorch-API der 🤗 Transformers-Bibliothek konzentriert, zeigen wir Ihnen in Kapitel 2, wie Sie alle Beispiele in TensorFlow überführen können.

Die folgenden Ressourcen bieten Ihnen eine gute Grundlage für die in diesem Buch behandelten Themen. Wir gehen davon aus, dass Ihr Kenntnisstand in etwa auf deren Niveau liegt:

- *Praxiseinstieg Machine Learning mit Scikit-Learn, Keras und TensorFlow* von Aurélien Géron (O'Reilly)
- *Deep Learning for Coders with fastai and PyTorch* von Jeremy Howard und Sylvain Gugger (O'Reilly)
- *Natural Language Processing mit PyTorch* von Delip Rao und Brian McMahan (O'Reilly)
- Der Onlinekurs von Hugging Face (*https://oreil.ly/n3MaR*) des Open-Source-Teams von Hugging Face, auch auf Deutsch unter *https://huggingface.co/course/de/*

Was Sie lernen werden

Das Ziel dieses Buchs ist es, Sie in die Lage zu versetzen, Ihre eigenen Sprachanwendungen zu erstellen. Zu diesem Zweck konzentriert es sich auf praktische Anwendungsfälle und geht nur dort auf die theoretischen Aspekte ein, wo es notwendig ist. Der Ansatz des Buchs ist praxisorientiert, und wir empfehlen Ihnen dringend, die Codebeispiele selbst auszuprobieren.

Das Buch deckt alle wichtigen Anwendungen von Transformern im NLP ab, wobei jedes Kapitel (mit wenigen Ausnahmen) einer bestimmten Aufgabenstellung, verbunden mit einem realistischen Anwendungsfall und Datensatz, gewidmet ist. In jedem Kapitel werden außerdem einige zusätzliche Konzepte vorgestellt. Hier ist ein Überblick über die behandelten Aufgabenstellungen (engl. Tasks) und Themen:

- Kapitel 1, *Hallo Transformer*, stellt Transformer vor und ordnet sie in den Kontext ein. Außerdem wird eine Einführung in das Hugging-Face-Ökosystem gegeben.
- Kapitel 2, *Textklassifizierung*, konzentriert sich auf die Sentiment- bzw. Stimmungsanalyse – engl. Sentiment Analysis – (ein gängiges Textklassifizierungsproblem) und stellt die `Trainer`-Klasse vor.
- Kapitel 3, *Die Anatomie von Transformer-Modellen*, geht näher auf die Transformer-Architektur ein, um Sie auf die folgenden Kapitel vorzubereiten.
- Kapitel 4, *Multilinguale Named Entity Recognition*, konzentriert sich auf die Identifizierung von Entitäten bzw. Eigennamen in verschiedensprachigen Texten (eine Problemstellung im Rahmen der Klassifizierung von Tokens).
- Kapitel 5, *Textgenerierung*, untersucht die Fähigkeit von Transformer-Modellen, Text zu generieren, und stellt Decodierungsstrategien und Maße zur Beurteilung der Qualität vor.
- Kapitel 6, *Automatische Textzusammenfassung (Summarization)*, befasst sich mit der komplexen Sequence-to-Sequence-Aufgabe der Textzusammenfassung und erläutert die für diese Aufgabe verwendeten Maße.
- Kapitel 7, *Question Answering*, konzentriert sich auf den Aufbau eines rezensionsbasierten Fragebeantwortungssystems und stellt das Retrieval mit Haystack vor.
- Kapitel 8, *Effizientere Transformer-Modelle für die Produktion*, befasst sich mit der Leistungsfähigkeit der Modelle. Wir werden die Aufgabe der Intentionserkennung – engl. Intent Detection – (eine Art von Sequenzklassifzierungsproblem) betrachten und Techniken wie Knowledge Distillation, Quantisierung und Pruning untersuchen.
- Kapitel 9, *Ansätze bei wenig bis gar keinen zur Verfügung stehenden gelabelten Daten*, zeigt Möglichkeiten zur Verbesserung der Modellleistung auf, wenn keine großen Mengen an gelabelten Daten zur Verfügung stehen. Wir werden

einen GitHub Issues Tagger erstellen und Techniken wie Zero-Shot-Klassifikation und Datenerweiterung (engl. Data Augmentation) untersuchen.

- Kapitel 10, *Transformer-Modelle von Grund auf trainieren*, zeigt Ihnen, wie Sie ein Modell für die automatische Vervollständigung von Python-Quellcode von Grund auf erstellen und trainieren können. Wir befassen uns mit dem Streaming von Datensätzen und dem Training von Modellen in großem Maßstab und erstellen unseren eigenen Tokenizer.
- Kapitel 11, *Künftige Herausforderungen*, untersucht die Herausforderungen, mit denen Transformer konfrontiert sind, und einige der spannenden neuen Richtungen, die die Forschung in diesem Bereich einschlägt.

Die 🤗 Transformers-Bibliothek bietet mehrere Abstraktionsebenen für die Verwendung und das Training von Transformer-Modellen. Wir beginnen mit den benutzerfreundlichen Pipelines, die es uns ermöglichen, Textbeispiele durch die Modelle zu leiten und die Vorhersagen mit nur wenigen Codezeilen zu ermitteln. Anschließend befassen wir uns mit Tokenizern, Modellklassen und der `Trainer`-Klasse, mit der wir Modelle für unsere eigenen Anwendungsfälle trainieren können. Später werden wir Ihnen zeigen, wie Sie die `Trainer`-Klasse durch die 🤗 Accelerate-Bibliothek ersetzen können, die uns die volle Kontrolle über die Trainingsschleife gibt und es uns ermöglicht, große Transformer-Modelle komplett von Grund auf zu trainieren! Jedes Kapitel ist weitgehend in sich abgeschlossen, wobei der Schwierigkeitsgrad der Aufgaben in den späteren Kapiteln zunimmt. Aus diesem Grund empfehlen wir, mit den Kapiteln 1 und 2 zu beginnen, bevor Sie sich dem Thema zuwenden, das Sie am meisten interessiert.

Neben der 🤗 Transformers- und der 🤗 Accelerate-Bibliothek werden wir auch ausgiebig von der 🤗 Datasets-Bibliothek Gebrauch machen, die sich nahtlos in andere Bibliotheken integrieren lässt. Die 🤗 Datasets-Bibliothek bietet ähnliche Funktionen für die Datenverarbeitung wie Pandas, ist jedoch von Grund auf für die Verarbeitung großer Datenmengen und Machine Learning (bzw. maschinelles Lernen) konzipiert.

Mit diesen Tools haben Sie alles, was Sie benötigen, um fast jede Herausforderung im Bereich des NLP zu meistern!

Software- und Hardwareanforderungen

Aufgrund des praxisorientierten Ansatzes dieses Buchs empfehlen wir Ihnen dringend, die Codebeispiele auszuführen, während Sie die einzelnen Kapitel lesen. Da wir es mit Transformern zu tun haben, benötigen Sie Zugang zu einem Computer mit einer NVIDIA-GPU, um diese Modelle trainieren zu können. Glücklicherweise gibt es online mehrere kostenlose Optionen, die Sie nutzen können, u. a.:

- Google Colaboratory (*https://oreil.ly/jyXgA*)
- Kaggle Notebooks (*https://oreil.ly/RnMP3*)
- Paperspace Gradient Notebooks (*https://oreil.ly/mZEKy*)

Um die Beispiele ausführen zu können, müssen Sie die Installationsanleitung befolgen, die wir im GitHub-Repository des Buchs bereitstellen. Sie finden die Anleitung und die Codebeispiele unter *https://github.com/nlp-with-transformers/notebooks*.

Wir haben die meisten Kapitel mit NVIDIA Tesla P100 GPUs entwickelt, die über 16 GB an Speicher verfügen. Einige der freien Plattformen bieten GPUs mit einem geringeren Speicher an, sodass Sie beim Trainieren der Modelle möglicherweise die Batchgröße verringern müssen.

In diesem Buch verwendete Konventionen

Die folgenden typografischen Konventionen werden in diesem Buch verwendet:

Kursiv
: Kennzeichnet neue Begriffe, URLs, E-Mail-Adressen, Dateinamen und Dateiendungen.

`Konstante Zeichenbreite`
: Wird für Programmlistings und für Programmelemente in Textabschnitten wie Namen von Variablen und Funktionen, Datenbanken, Datentypen, Umgebungsvariablen, Anweisungen und Schlüsselwörter verwendet.

`Konstante Zeichenbreite, fett`
: Kennzeichnet Befehle oder anderen Text, den der Nutzer wörtlich eingeben sollte.

`Konstante Zeichenbreite, kursiv`
: Kennzeichnet Text, den der Nutzer je nach Kontext durch entsprechende Werte ersetzen sollte.

Tipp
Dieses Symbol steht für einen Tipp oder eine Empfehlung.

Hinweis
Dieses Symbol steht für einen allgemeinen Hinweis.

Warnung
Dieses Symbol warnt oder mahnt zur Vorsicht.

Verwenden von Codebeispielen

Zusätzliche Materialien (Codebeispiele, Übungen usw.) können Sie unter *https://github.com/nlp-with-transformers/notebooks* herunterladen.

Wir haben eine Webseite für dieses Buch, auf der wir Errata, Beispiele und zusätzliche Informationen veröffentlichen. Sie können diese Seite unter *https://www.oreilly.com/library/view/natural-language-processing/9781098136789/* aufrufen.

Dieses Buch dient dazu, Ihnen bei der Erledigung Ihrer Arbeit zu helfen. Im Allgemeinen dürfen Sie die Codebeispiele aus diesem Buch in Ihren eigenen Programmen und der dazugehörigen Dokumentation verwenden. Sie müssen uns dazu nicht um Erlaubnis bitten, solange Sie nicht einen beträchtlichen Teil des Codes reproduzieren. Beispielsweise benötigen Sie keine Erlaubnis, um ein Programm zu schreiben, in dem mehrere Codefragmente aus diesem Buch vorkommen. Wollen Sie dagegen eine CD-ROM mit Beispielen aus Büchern von O'Reilly verkaufen oder verbreiten, benötigen Sie eine Erlaubnis. Eine Frage zu beantworten, indem Sie aus diesem Buch zitieren und ein Codebeispiel wiedergeben, benötigt keine Erlaubnis. Eine beträchtliche Menge Beispielcode aus diesem Buch in die Dokumentation Ihres Produkts aufzunehmen, bedarf hingegen unserer ausdrücklichen Zustimmung.

Wir freuen uns über Zitate, verlangen diese aber nicht. Ein Zitat enthält Titel, Autor, Verlag und ISBN. Beispiel: »*Natural Language Processing mit Transformern* von Lewis Tunstall, Leandro von Werra und Thomas Wolf (O'Reilly). Copyright 2023 dpunkt.verlag, ISBN 978-3-96009-202-5.«

Wenn Sie glauben, dass Ihre Verwendung von Codebeispielen über die übliche Nutzung hinausgeht oder außerhalb der oben vorgestellten Nutzungsbedingungen liegt, kontaktieren Sie uns bitte unter *kommentar@oreilly.de*.

Danksagungen

Das Schreiben eines Buchs über einen der sich am schnellsten entwickelnden Bereiche des maschinellen Lernens wäre ohne die Hilfe vieler Menschen nicht möglich gewesen. Wir danken dem wunderbaren O'Reilly-Team und insbesondere Melissa Potter, Rebecca Novack und Katherine Tozer für ihre Unterstützung und Beratung. Das Buch hat auch von großartigen Fachgutachtern profitiert, die unzählige Stunden damit verbracht haben, uns unschätzbares Feedback zu geben. Besonders dankbar sind wir Luca Perozzi, Hamel Husain, Shabie Iqbal, Umberto Lupo, Malte Pietsch, Timo Möller und Aurélien Géron für ihre ausführlichen Rezensionen. Wir danken Branden Chan von deepset (*https://www.deepset.ai*) für seine Hilfe bei der Erweiterung der Haystack-Bibliothek zur Unterstützung des Anwendungsfalls in Kapitel 7. Die wunderschönen Illustrationen in diesem Buch verdanken wir der fantastischen Christa Lanz (*https://christalanz.ch*) – vielen Dank, dass Sie dieses Buch zu etwas ganz Besonderem gemacht haben. Wir hatten auch das Glück, die Unterstützung des gesamten Hugging-Face-Teams zu erhalten. Vielen Dank an

Quentin Lhoest für die Beantwortung zahlloser Fragen zur 🤗 Datasets-Bibliothek, an Lysandre Debut für seine Hilfe bei allem, was mit dem Hugging Face Hub zu tun hat, an Sylvain Gugger für seine Hilfe im Zusammenhang mit der 🤗 Accelerate-Bibliothek und an Joe Davison für seine Inspiration zu Kapitel 9 in Bezug auf Zero-Shot-Learning. Wir danken auch Sidd Karamcheti und dem gesamten Mistral-Team (*https://oreil.ly/aOYLt*) für die Stabilitätsverbesserungen für GPT-2, die Kapitel 10 möglich machten. Dieses Buch wurde vollständig in Jupyter Notebooks verfasst, und wir danken Jeremy Howard und Sylvain Gugger für die Entwicklung von wunderbaren Tools wie fastdoc (*https://oreil.ly/yVCfT*), die dies ermöglicht haben.

Lewis

Sofia, ich danke dir für deine ständige Unterstützung und Ermutigung – sonst würde es dieses Buch nicht geben. Nach der langen Zeit des Schreibens können wir endlich wieder unsere Wochenenden genießen!

Leandro

Janine, ich danke dir für deine Geduld und deine ermutigende Unterstützung während dieses langen Jahrs mit vielen langen Nächten und arbeitsreichen Wochenenden.

Thomas

Ich möchte vor allem Lewis und Leandro dafür danken, dass sie die Idee zu diesem Buch hatten und sich dafür stark gemacht haben, es in einem so schönen und zugänglichen Format zu veröffentlichen. Ich möchte auch dem gesamten Team von Hugging Face dafür danken, dass es an die Mission von KI als gemeinschaftliche Leistung glaubt, und der gesamten NLP-/KI-Community dafür, dass sie die Bibliotheken und die Forschung, die wir in diesem Buch beschreiben, gemeinsam mit uns aufgebaut und genutzt hat.

Mehr als das, was wir aufbauen, ist die Reise selbst, die wir unternehmen, was wirklich zählt. Wir haben das Privileg, heute diesen Weg mit Tausenden von Community-Mitgliedern und Lesern wie Ihnen zu gehen. Wir danken Ihnen allen aus tiefstem Herzen.

KAPITEL 1

Hallo Transformer

Im Jahr 2017 veröffentlichten Forscher von Google einen Artikel, der eine neuartige neuronale Netzwerkarchitektur für die Modellierung von Sequenzen vorschlug.[1] Diese als *Transformer* bezeichnete Architektur übertraf rekurrente neuronale Netze (engl. Recurrent Neural Networks, RNNs) bei maschinellen Übersetzungsaufgaben sowohl in Bezug auf die Übersetzungsqualität als auch auf die Trainingskosten.

Gleichzeitig wurde mit einem effektiven Transfer-Learning-Verfahren namens ULMFiT gezeigt, dass durch das Training von LSTM-Netzwerken (Long Short-Term Memory) auf einem sehr großen und vielfältigen Korpus hochmoderne Textklassifikatoren erstellt werden können, die nur wenige gelabelte Daten erfordern.[2]

Diese Fortschritte bildeten die Grundlage für zwei der heute bekanntesten Transformer-Modelle: zum einen dem Generative Pretrained Transformer (GPT)[3] und zum anderen Bidirectional Encoder Representations from Transformers (BERT)[4]. Durch die Kombination der Transformer-Architektur mit unüberwachtem Lernen (engl. Unsupervised Learning) machten diese Modelle das Training aufgabenspezifischer Architekturen, die von Grund auf trainiert werden müssen, überflüssig und brachen fast jede Benchmark im NLP-Bereich mit deutlichem Abstand. Seit der Veröffentlichung von GPT und BERT ist eine beträchtliche Sammlung (im Englischen als Model Zoo bezeichnet) von Transformer-Modellen entwickelt worden. Die chronologische Entwicklung mit den wichtigsten Fortschritten können Sie in Abbildung 1-1 nachvollziehen.

1 A. Vaswani et al., »Attention Is All You Need« (*https://arxiv.org/abs/1706.03762*), (2017). Dieser Titel war so einprägsam, dass nicht weniger als 50 Folgeartikel (*https://oreil.ly/wT8Ih*) »all you need« in ihren Titeln enthalten!

2 J. Howard und S. Ruder, »Universal Language Model Fine-Tuning for Text Classification« (*https://arxiv.org/abs/1801.06146*), (2018).

3 A. Radford et al., »Improving Language Understanding by Generative Pre-Training« (*https://openai.com/blog/language-unsupervised*), (2018).

4 J. Devlin et al., »BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding« (*https://arxiv.org/abs/1810.04805*), (2018).

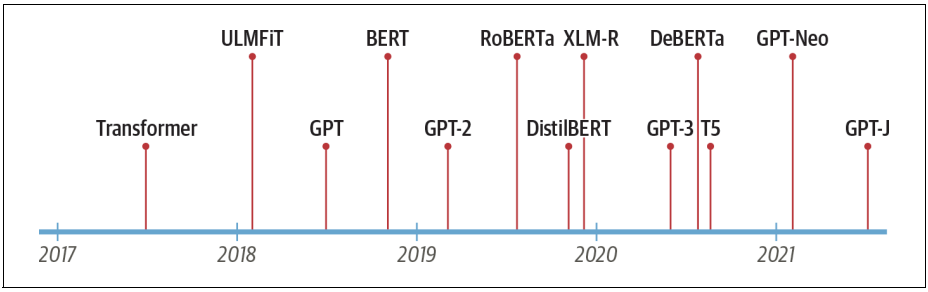


Abbildung 1-1: Chronologische Entwicklung von Transformer-Modellen

Doch greifen wir nicht zu weit vor. Um zu verstehen, was das Neue an Transformer-Modellen ist, müssen wir zunächst einige Begrifflichkeiten erläutern:

- das Encoder-Decoder-Framework
- den Attention-Mechanismus
- Transfer Learning

In diesem Kapitel stellen wir Ihnen die grundlegenden Konzepte vor, die für die weite Verbreitung von Transformern verantwortlich sind, und werfen einen Blick auf die Aufgaben, für die sich diese besonders eignen.

Lassen Sie uns zunächst das Encoder-Decoder-Framework und die Architekturen erkunden, die der Entwicklung von Transformer-Modellen vorausgingen.

Das Encoder-Decoder-Framework

Im Bereich des Natural Language Processing (NLP) galten vor den Transformern rekurrente Architekturen wie LSTMs als State of the Art. Diese Architekturen enthalten eine Rückkopplungsschleife in den Netzwerkverbindungen, die es ermöglicht, Informationen von einem Schritt zum nächsten weiterzugeben – dadurch eignen sie sich ideal für die Modellierung sequenzieller Daten wie Texte. Wie auf der linken Seite von Abbildung 1-2 dargestellt, empfängt ein RNN eine Eingabe bzw. einen Input (z.B. ein Wort oder ein Zeichen), leitet sie durch das Netzwerk und gibt einen Vektor aus, den sogenannten *Hidden State* bzw. *verborgenen Zustand* (auch verdeckter Zustand genannt). Gleichzeitig gibt das Modell über die Rückkopplungsschleife einige Informationen an sich selbst zurück, die es dann im nächsten Schritt verwenden kann. Das wird noch deutlicher, wenn wir die Schleife weiter »aufdröseln« bzw. in zeitlicher Hinsicht desaggregieren, wie auf der rechten Seite von Abbildung 1-2 gezeigt: Das RNN gibt in jedem Schritt Informationen über seinen Zustand an die nächste Operation in der Sequenz weiter. Auf diese Weise kann ein RNN die Informationen aus den vorangegangenen Schritten berücksichtigen und sie für seine Vorhersagen nutzen.

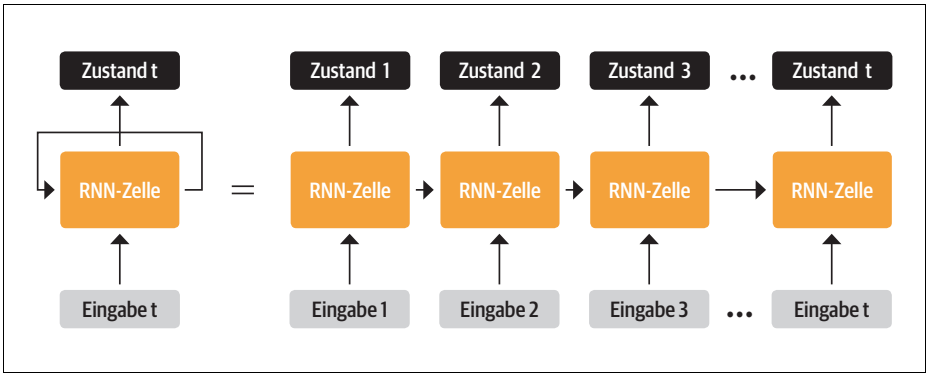


Abbildung 1-2: Zeitliche Desaggregation eines RNN

Diese Architekturen wurden (und werden) häufig für Aufgaben im Bereich des NLP, der Sprachverarbeitung (engl. Speech Processing) und für Zeitreihenanalysen verwendet. Eine wunderbare Übersicht, die Ihnen ein Bild davon vermittelt, wozu sie in der Lage sind, finden Sie in dem von Andrej Karpathy verfassten Blogbeitrag »The Unreasonable Effectiveness of Recurrent Neural Networks« (*https://oreil.ly/Q55o0*).

Ein Bereich, in dem RNNs eine wichtige Rolle spielten, war die Entwicklung von maschinellen Übersetzungssystemen. Bei diesen geht es darum, eine Folge von Wörtern aus einer Sprache in eine andere zu überführen. Diese Art von Aufgabe wird normalerweise mit einer *Encoder-Decoder-* bzw. einer *Sequence-to-Sequence-*Architektur[5] bewältigt, die sich gut für Situationen eignet, in denen sowohl die Eingabe (engl. Input) als auch die Ausgabe (engl. Output) Sequenzen beliebiger Länge darstellen. Die Aufgabe des Encoders ist es, die Informationen aus der Eingabesequenz in eine numerische Darstellung zu codieren, die oft als *letzter verborgener Zustand* (engl. Last Hidden State) bezeichnet wird. Dieser Zustand wird anschließend an den Decoder weitergegeben, der die Ausgabesequenz erzeugt.

Im Allgemeinen können die Encoder- und Decoder-Komponenten jede Art von neuronaler Netzwerkarchitektur sein, mit der sich Sequenzen modellieren lassen. In dem Beispiel in Abbildung Abbildung 1-3 werden z.B. mehrere RNNs verwendet, um für den englischen Satz »Transformers are great!« (der als verborgener Zustandsvektor codiert und dann decodiert wird) die deutsche Übersetzung »Transformer sind grossartig!« zu erzeugen. Die eingegebenen Wörter werden nacheinander durch den Encoder geleitet und die ausgegebenen Wörter – jeweils eines nach dem anderen, d.h. von oben nach unten – erzeugt.

5 I. Sutskever, O. Vinyals, and Q.V. Le, »Sequence to Sequence Learning with Neural Networks« (*https://arxiv.org/abs/1409.3215*), (2014).

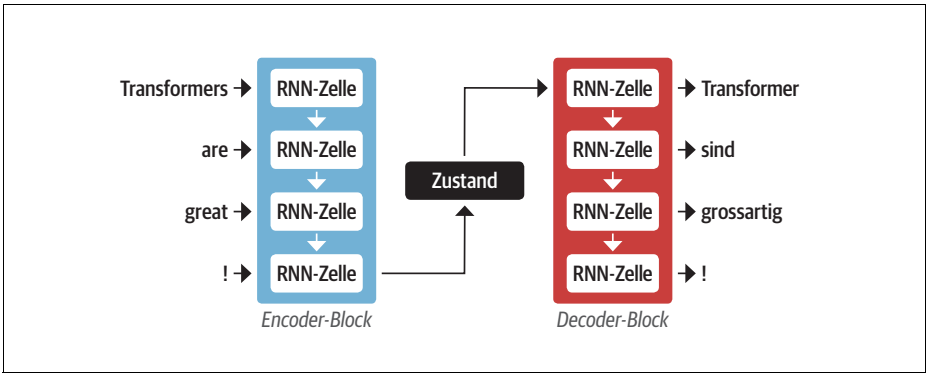


Abbildung 1-3: Eine Encoder-Decoder-Architektur mit mehreren RNNs (in der Regel gibt es bedeutend mehr rekurrente Schichten bzw. Layer als hier dargestellt)

Obwohl sie in ihrer Einfachheit elegant ist, besteht eine Schwäche dieser Architektur darin, dass der verborgene Endzustand des Encoders einen *Informationsengpass* darstellt: Er muss die Bedeutung der gesamten Eingabesequenz repräsentieren bzw. abbilden, da dies alles ist, worauf der Decoder bei der Erzeugung der Ausgabe zugreifen kann. Das ist vor allem bei langen Sequenzen eine Herausforderung, da Informationen, die am Anfang der Sequenz enthalten sind, bei der Komprimierung auf eine einzelne, starre Darstellung verloren gehen können.

Glücklicherweise gibt es einen Weg, diesen Engpass zu umgehen, indem der Decoder Zugriff auf alle verborgenen Zustände des Encoders erhält. Der grundlegende Ansatz für diese Lösung heißt *Attention*[6], der eine Schlüsselkomponente in vielen modernen neuronalen Netzwerkarchitekturen darstellt. Wenn wir nachvollziehen, wie der Attention-Mechanismus in RNNs implementiert wurde, können wir einen der wichtigsten Bausteine der Transformer-Architektur besser verstehen. Werfen wir einen genaueren Blick darauf.

Der Attention-Mechanismus

Die Hauptidee hinter Attention ist, dass der Encoder nicht nur einen einzigen verborgenen Zustand für die Eingabesequenz erzeugt, sondern bei jedem Schritt einen verborgenen Zustand (engl. Hidden State) ausgibt, auf den der Decoder zugreifen kann. Würden jedoch alle Zustände gleichzeitig verwendet, würde der Decoder eine sehr große Eingabe erhalten, sodass ein Mechanismus erforderlich ist, um Prioritäten bei der Verwendung der Zustände zu setzen. Hier kommt Attention ins Spiel: Der Decoder kann jedem der Encoderzustände bei jedem Decodierschritt eine andere Gewichtung zuweisen bzw. »Aufmerksamkeit« schenken. Dieser Prozess wird in Abbildung 1-4 veranschaulicht, wo gezeigt wird, welche Rolle die Attention für die Vorhersage des zweiten Tokens in der Ausgabesequenz einnimmt.

6 D. Bahdanau, K. Cho, and Y. Bengio, »Neural Machine Translation by Jointly Learning to Align and Translate« (*https://arxiv.org/abs/1409.0473*), (2014).

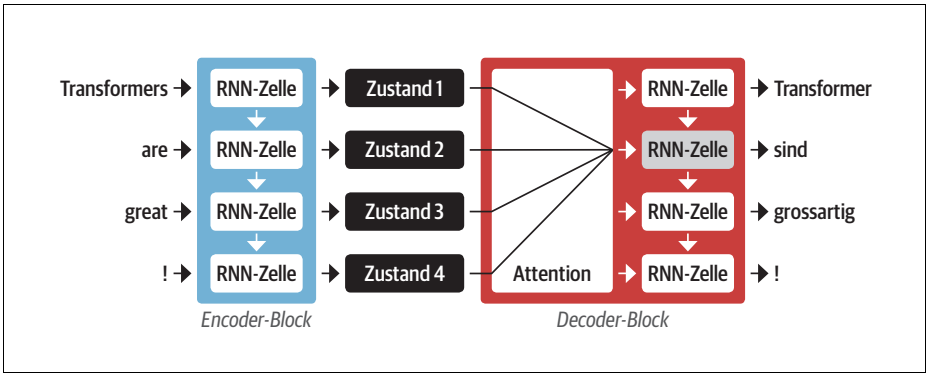


Abbildung 1-4: Eine Encoder-Decoder-Architektur mit Attention-Mechanismus für mehrere RNNs

Indem sie sich darauf konzentrieren, welche der eingegebenen Tokens zu jedem Zeitpunkt (also Schritt) am relevantesten sind, sind diese auf Attention basierenden Modelle in der Lage, nicht-triviale Zuordnungen zwischen den Wörtern in einer generierten Übersetzung und denen in einem Ausgangssatz zu lernen. In Abbildung 1-5 werden zum Beispiel die Attention-Gewichte für ein Modell, das Übersetzungen vom Englischen ins Französische vornimmt, dargestellt, wobei jedes Pixel einem Gewicht entspricht. Die Abbildung zeigt, wie der Decoder in der Lage ist, die Wörter »zone« und »Area«, die in den beiden Sprachen unterschiedlich angeordnet sind, korrekt zuzuordnen.

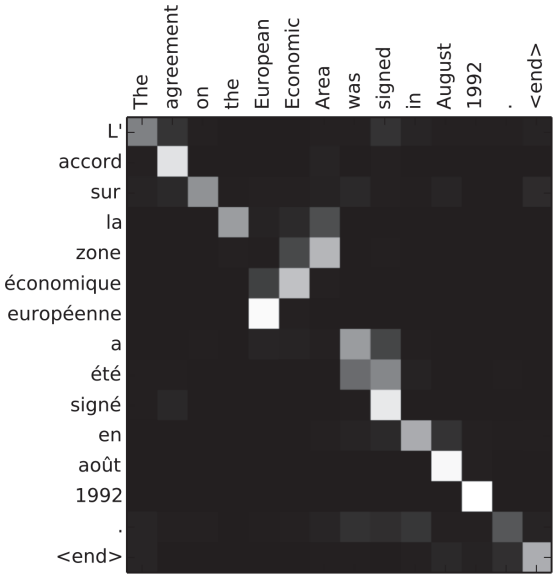


Abbildung 1-5: Zuordnung der Wörter zwischen einem englischen Ausgangstext und der generierten Übersetzung ins Französische eines Encoder-Decoder-Modells auf Basis von RNNs (mit freundlicher Genehmigung von Dzmitry Bahdanau)

Obwohl infolge der Verwendung von Attention bedeutend bessere Übersetzungen erzielt werden konnten, gab es bei der Verwendung von rekurrenten Modellen als Encoder und Decoder einen großen Nachteil: Die Berechnungen sind von Natur aus sequenziell und können nicht für die gesamte Eingabesequenz parallelisiert werden.

Mit dem Transformer wurde ein neues Modellierungsparadigma eingeführt: Bei dieser Architektur wird auf die Rekursion verzichtet und stattdessen vollständig auf eine besondere Form von Attention, der sogenannten *Self-Attention*, zurückgegriffen. Was Self-Attention genau ist, werden wir noch in Kapitel 3 erläutern. Die Grundidee hierbei ist, dass die Attention auf alle Zustände *derselben Schicht* des neuronalen Netzes operieren kann. Dies können Sie in Abbildung 1-6 nachvollziehen, wo sowohl der Encoder als auch der Decoder ihre eigenen Self-Attention-Mechanismen haben, deren Ausgaben in neuronale Feed-Forward-Netze (engl. Feed-Forward Neural Networks) eingespeist werden. Diese Architektur kann bedeutend schneller trainiert werden als rekurrente Modelle und ebnete vielen der jüngsten Durchbrüche im NLP den Weg.

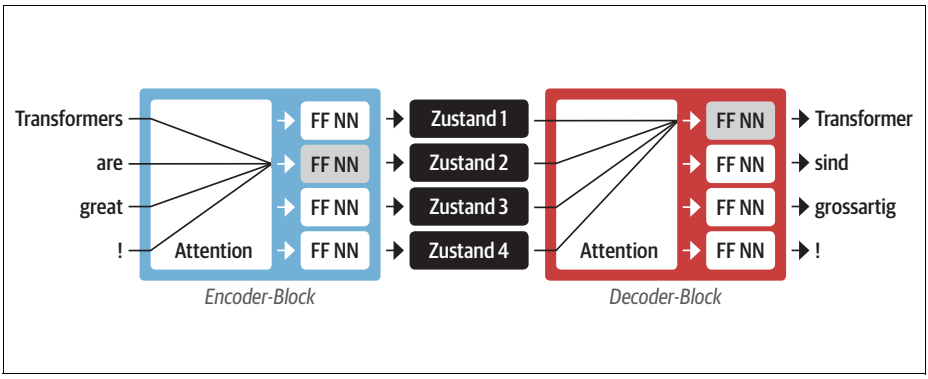


Abbildung 1-6: Encoder-Decoder-Architektur des ursprünglichen Transformers

In dem ursprünglichen Artikel zu Transformer-Modellen wurde das Übersetzungsmodell anhand eines großen Korpus von Satzpaaren, die in verschiedenen Sprachen vorliegen, von Grund auf trainiert. In vielen Fällen, in denen NLP in der Praxis zum Einsatz kommt, gibt es jedoch keine große Menge an gelabelten Textdaten, mit denen wir unsere Modelle trainieren könnten. Um die Transformer-Revolution vollends in Gang zu setzen, fehlte daher noch ein letzter Baustein: Transfer Learning.

Einsatz von Transfer Learning im NLP

Im Bereich der Computer Vision (computerbasiertes Sehen) ist es heutzutage üblich, ein Convolutional Neural Network (CNN, auch neuronales Konvolutionsnetz genannt) wie ResNet mithilfe von Transfer Learning für eine Aufgabe zu trainieren und es dann an eine neue Aufgabe anzupassen bzw. es für diese feinzutunen (engl.

fine-tune). Auf diese Weise kann das Netz das Wissen nutzen, das es bei der ursprünglichen Aufgabe gelernt hat. Architektonisch bedeutet dies, dass das Modell in einen *Body* (bzw. Körper) und einen *Head* (bzw. Kopf) aufgeteilt wird, wobei der Head ein aufgabenspezifisches Netz darstellt. Während des Trainings lernt der Body mittels seiner Gewichte allgemeine Merkmale (engl. Features) der Ausgangsdomäne. Diese Gewichte werden verwendet, um ein neues Modell, das für die neue Aufgabe bestimmt ist, zu initialisieren.[7] Im Vergleich zum traditionellen überwachten Lernen (engl. Supervised Learning) führt dieser Ansatz in der Regel zu qualitativ hochwertigen Modellen, die bedeutend effizienter für eine Vielzahl von nachgelagerten Aufgaben (engl. Downstream Tasks) und mit weit weniger gelabelten Daten trainiert werden können. Einen Vergleich der beiden Ansätze finden Sie in Abbildung 1-7.

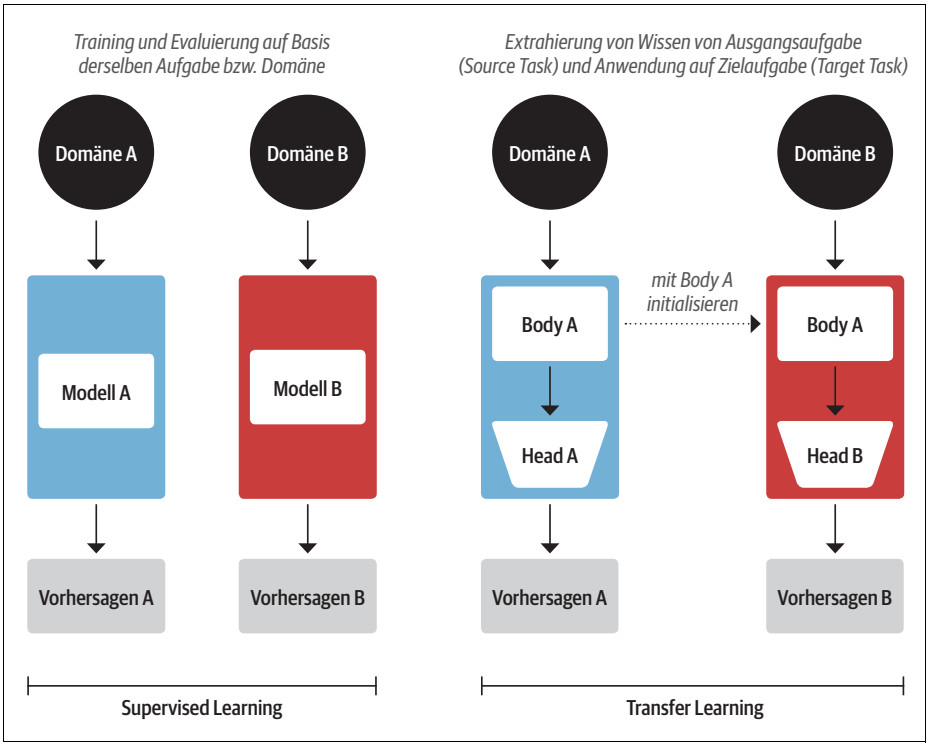


Abbildung 1-7: Vergleich von traditionellem Supervised Learning (links) und Transfer Learning (rechts)

In der Computer Vision werden die Modelle zunächst auf großen Datensätzen wie ImageNet (*https://image-net.org*) trainiert, die Millionen von Bildern enthalten. Dieser Vorgang wird als *Pretraining* (bzw. Vortraining) bezeichnet und dient vor allem dazu, den Modellen die grundlegenden Merkmale von Bildern, wie Kanten oder Farben, beizubringen. Diese vortrainierten Modelle können dann für eine

7 Gewichte sind die lernbaren Parameter eines neuronalen Netzes.

nachgelagerte Aufgabe, wie z. B. die Klassifizierung von Blumenarten, mit einer relativ kleinen Anzahl von gelabelten Beispielen (in der Regel einige Hundert pro Kategorie) feingetunt werden. Solche (im Rahmen des Feintunings) optimierten Modelle erreichen in der Regel eine höhere Treffergenauigkeit (engl. Accuracy) als überwachte Modelle, die mit der gleichen Menge an gelabelten Daten von Grund auf trainiert wurden.

Obwohl sich das Transfer Learning in der Computer Vision durchgesetzt hat, war lange Zeit nicht klar, wie das entsprechende Pretraining beim NLP gestaltet sein sollte. Dementsprechend benötigten NLP-Anwendungen in der Regel große Mengen an gelabelten Daten, um eine hohe Leistungsfähigkeit zu erreichen. Allerdings war selbst dann die Leistung nicht vergleichbar mit dem, was im Bereich Computer Vision erreicht werden konnte.

In den Jahren 2017 und 2018 wurden neue Ansätze von verschiedenen Forschungsgruppen vorgeschlagen, die schließlich dazu führten, dass Transfer Learning auch für das NLP nutzbar gemacht werden konnte. Es begann damit, dass Forscher bei OpenAI erkannten, dass eine starke Leistung bei einer Sentiment-Klassifizierungsaufgabe erzielt wurde, indem sie Features verwendeten, die im Rahmen eines unüberwachten Pretrainings gewonnen wurden.[8] Im Anschluss folgte ULMFiT, mit dem ein allgemeiner Rahmen für die Verwendung vortrainierter LSTM-Modelle für verschiedene Aufgaben geschaffen wurde.[9]

Wie in Abbildung 1-8 dargestellt, umfasst ULMFiT drei Hauptschritte:

Pretraining

Das anfängliche Ziel des Trainings ist ganz einfach: das nächste Wort auf der Grundlage der vorherigen Wörter vorherzusagen. Diese Aufgabe wird als *Sprachmodellierung* (engl. Language Modeling) bezeichnet. Die Eleganz dieses Ansatzes liegt darin, dass keine gelabelten Daten benötigt werden und auf reichlich vorhandenen Text aus Quellen wie Wikipedia zurückgegriffen werden kann.[10]

Domänenadaption (engl. Domain Adaptation)

Sobald das Sprachmodell auf ein großes Korpus vortrainiert wurde, wird es im nächsten Schritt auf ein domänenspezifisches Korpus übertragen (z. B. von Wikipedia auf das IMDb-Korpus, das Filmrezensionen enthält, wie in Abbildung 1-8 dargestellt). Zwar wird auch in diesem Schritt auf die Sprachmodellierung zurückgegriffen, allerdings muss das Modell in diesem Zusammenhang das nächste Wort im Zielkorpus vorhersagen.

8 A. Radford, R. Jozefowicz und I. Sutskever, »Learning to Generate Reviews and Discovering Sentiment« (*https://arxiv.org/abs/1704.01444*), (2017).

9 Eine verwandte Veröffentlichung zu dieser Zeit war ELMo (Embeddings from Language Models), mit der gezeigt wurde, wie durch ein Pretraining von LSTMs qualitativ hochwertige Worteinbettungen – aus dem Englischen auch häufig als Word Embeddings bezeichnet – für nachgelagerte Aufgaben erzeugt werden können.

10 Dies gilt eher für das Englische als für die meisten anderen Sprachen der Welt, für die es schwierig sein kann, ein großes Korpus digitalisierter Texte zu erhalten. Die Suche nach Möglichkeiten, diese Lücke zu schließen, ist ein aktiver Bereich der NLP-Forschung und -Bemühungen.

Feintuning (engl. Fine-tuning)

In diesem Schritt wird das Sprachmodell mit einer (zusätzlichen) Klassifizierungsschicht bzw. einem Klassifizierungslayer für die Zielaufgabe (engl. Target Task) feingetunt (z.B. zur Klassifizierung des Sentiments bzw. der Polarität von Filmrezensionen in Abbildung 1-8).

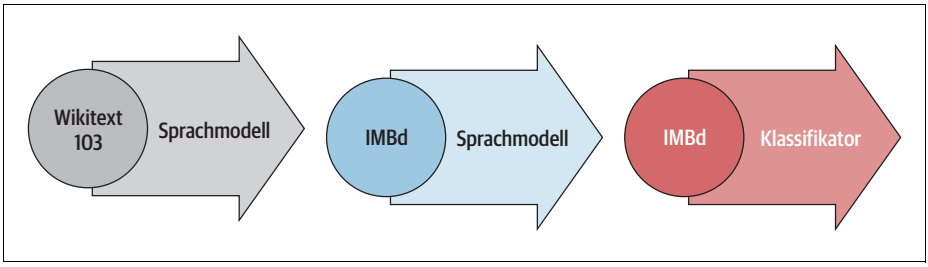


Abbildung 1-8: Das Vorgehen bei ULMFiT (mit freundlicher Genehmigung von Jeremy Howard)

Die Einführung von ULMFiT bot somit einen praktikablen Rahmen für das Pretraining und die Anwendung von Transfer Learning im Bereich des NLP und lieferte damit das fehlende Puzzleteil, um Transformern zum Durchbruch zu verhelfen. Im Jahr 2018 wurden zwei Transformer-Modelle veröffentlicht, bei denen Self-Attention mit Transfer Learning kombiniert wurde:

GPT

Verwendet nur den Decoder-Teil der Transformer-Architektur und denselben Sprachmodellierungsansatz wie ULMFiT. GPT wurde mithilfe des BookCorpus-Datensatzes[11] vortrainiert, der aus 7.000 unveröffentlichten Büchern aus verschiedenen Genres wie Abenteuer, Fantasy und Romantik besteht.

BERT

Verwendet den Encoder-Teil der Transformer-Architektur und eine spezielle Form der Sprachmodellierung, die als *Masked Language Modeling* bezeichnet wird. Das Ziel dieses Ansatzes besteht darin, in einem Text Wörter vorherzusagen, die zufällig maskiert bzw. verdeckt wurden. Bei einem Satz wie »Ich habe auf meine `[MASK]` geschaut und gesehen, dass sich `[MASK]` verspätet hat.« muss das Modell jeweils die wahrscheinlichsten Kandidaten für die maskierten Wörter, die durch `[MASK]` gekennzeichnet sind, vorhersagen. BERT wurde auf dem BookCorpus-Datensatz und der englischsprachigen Wikipedia vortrainiert.

Sowohl GPT als auch BERT haben den Stand der Technik (State of the Art) in einer Reihe von NLP-Benchmarks neu gesetzt und das Zeitalter der Transformer-Modelle eingeläutet.

Da jedoch verschiedene Forschungseinrichtungen ihre Modelle in inkompatiblen Frameworks (PyTorch oder TensorFlow) veröffentlichten, war es für NLP-Praktike-

11 Y. Zhu et al., »Aligning Books and Movies: Towards Story-Like Visual Explanations by Watching Movies and Reading Books« (*https://arxiv.org/abs/1506.06724*), (2015).

rinnen und -Praktiker nicht immer einfach, diese Modelle auf ihre eigenen Anwendungen zu übertragen. Mit der Veröffentlichung der 🤗 Transformers-Bibliothek (*https://oreil.ly/Z79jF*) wurde nach und nach eine einheitliche Programmierschnittstelle (API) für mehr als 50 Architekturen entwickelt. Dank dieser Bibliothek wurde die Forschung im Bereich von Transformer-Modellen explosionsartig vorangetrieben, und die Praktikerinnen und Praktiker im Bereich NLP konnten diese Modelle innerhalb kurzer Zeit auf einfache Weise in viele reale Anwendungen integrieren. Lassen Sie uns einen Blick darauf werfen!

Die Transformers-Bibliothek von Hugging Face: die Lücke schließen

Eine neue Machine-Learning-Architektur auf eine neue Aufgabe anzuwenden, kann ein komplexes Unterfangen sein und umfasst in der Regel die folgenden Schritte:

1. Die Modellarchitektur als Code zu implementieren, typischerweise auf Basis von PyTorch oder TensorFlow.
2. Die vortrainierten Gewichte (falls verfügbar) von einem Server zu laden.
3. Die Eingaben (engl. Inputs) vorzuverarbeiten (engl. Preprocessing), sie durch das Modell laufen zu lassen und je nach Aufgabe, die verfolgt wird, nachzuverarbeiten (engl. Postprocessing).
4. Die Objekte, mit denen sich die Daten laden lassen, sogenannte Dataloader, zu implementieren und die Verlustfunktionen und den Optimizer bzw. den Optimierungsalgorithmus, die zum Trainieren des Modells verwendet werden, zu bestimmen.

Je nach Modell und Aufgabe sind jeweils verschiedenartige Schritte erforderlich. Wenn Forschungsgruppen einen neuen Artikel veröffentlichen, geben sie üblicherweise (aber nicht immer!) auch den Code zusammen mit der Modellgewichtung frei. Der Code ist jedoch selten standardisiert, und es erfordert oft tagelange Entwicklungsarbeit, ehe er so angepasst ist, dass er für neue Anwendungsfälle verwendet werden kann.

Genau an diesem Punkt kommt die Transformers-Bibliothek den NLP-Anwenderinnen und -Anwendern zu Hilfe! Sie bietet eine standardisierte Schnittstelle zu einer breiten Palette von Transformer-Modellen sowie Code und Tools, mit denen sich die Modelle auf neue Anwendungsfälle übertragen lassen. Die Bibliothek unterstützt derzeit drei wichtige Deep-Learning-Frameworks (PyTorch, TensorFlow und JAX) und ermöglicht es Ihnen, ohne Weiteres zwischen ihnen zu wechseln. Darüber hinaus bietet sie aufgabenspezifische Heads, sodass Sie Transformer für nachgelagerte Aufgaben wie die Textklassifizierung, die Eigennamenerkennung (engl. Named Entity Recognition, NER) oder die Beantwortung von Fragen (engl. Question Answering) problemlos feintunen können.[12] Dadurch verkürzt sich die

12 Anm. d. Übersetzers: Um eine bessere Nachvollziehbarkeit und Übertragbarkeit in die Praxis zu gewährleisten, verwenden wir in diesem Buch vornehmlich die englischen Begriffe.

Zeit, die Sie zum Trainieren und Testen einer Handvoll von Modellen benötigen, von einer Woche auf einen einzigen Nachmittag!

Im nächsten Abschnitt werden wir Ihnen zeigen, dass Sie einige der häufigsten NLP-Anwendungsfälle, die Ihnen in der Praxis begegnen werden, mithilfe der Transformers-Bibliothek mit nur wenigen Zeilen Code angehen können.

Die Anwendungsmöglichkeiten von Transformern im Überblick

Am Anfang einer jeden NLP-Aufgabe liegt ein Text vor, wie z. B. das folgende, frei erfundene Kundenfeedback zu einer bestimmten Onlinebestellung:

```
text = """Dear Amazon, last week I ordered an Optimus Prime action figure
from your online store in Germany. Unfortunately, when I opened the package,
I discovered to my horror that I had been sent an action figure of Megatron
instead! As a lifelong enemy of the Decepticons, I hope you can understand my
dilemma. To resolve the issue, I demand an exchange of Megatron for the
Optimus Prime figure I ordered. Enclosed are copies of my records concerning
this purchase. I expect to hear from you soon. Sincerely, Bumblebee."""
```

Je nach Ihrer Anwendung kann der Text, mit dem Sie arbeiten, ein juristischer Vertrag, eine Produktbeschreibung oder etwas ganz anderes sein. Im Falle von Kundenfeedback möchten Sie wahrscheinlich wissen, ob das Feedback positiv oder negativ ist. Diese Aufgabe wird *Sentimentanalyse* bzw. Stimmungsanalyse genannt und ist Teil des umfassenderen Bereichs der *Textklassifizierung* (engl. Text Classification), den wir in Kapitel 2 beleuchten werden. Schauen wir uns zunächst einmal an, was nötig ist, um die Stimmungslage bzw. das Sentiment unseres Texts mithilfe der 🤗 Transformers-Bibliothek zu ermitteln.

Textklassifizierung

Wie wir in den folgenden Kapiteln sehen werden, verfügt die 🤗 Transformers-Bibliothek über eine abgestufte API, die es Ihnen ermöglicht, mit der Bibliothek auf verschiedenen Abstraktionsebenen zu interagieren. In diesem Kapitel beginnen wir mit *Pipelines*, mit denen alle Schritte abstrahiert werden, die notwendig sind, um einen Rohtext in eine Reihe von Vorhersagen auf Basis eines feingetunten Modells umzuwandeln.

In der 🤗 Transformers-Bibliothek instanziieren wir eine Pipeline, indem wir die Funktion `pipeline()` aufrufen und den Namen der Aufgabe angeben, die für uns von Interesse ist:

```
from transformers import pipeline

classifier = pipeline("text-classification")
```

Wenn Sie diesen Code zum ersten Mal ausführen, werden einige Fortschrittsbalken angezeigt, da die Pipeline die Modellgewichtung automatisch vom Hugging

Face Hub (*https://oreil.ly/zLK11*) herunterlädt. Wenn Sie die Pipeline ein zweites Mal instanziieren, stellt die Bibliothek fest, dass Sie die Gewichtung bereits heruntergeladen haben, und verwendet stattdessen die im Cache gespeicherte Version. Die `text-classification`-Pipeline verwendet standardmäßig ein Modell, das für die Sentimentanalyse entwickelt wurde, unterstützt aber auch Multiklassen- (bzw. mehrkategoriale) und Multilabel-Klassifizierung.

Nachdem wir unsere Pipeline erstellt haben, können wir nun einige Vorhersagen treffen! Jede Pipeline nimmt eine Textzeichenkette bzw. Strings (oder eine Liste von Strings) als Eingabe entgegen und gibt eine Liste von Vorhersagen zurück. Jede Vorhersage entspricht einem Python-Dictionary, weshalb wir auf Pandas zurückgreifen können, um sie als `DataFrame` darzustellen:

```
import pandas as pd

outputs = classifier(text)
pd.DataFrame(outputs)
```

	Label	Score
0	NEGATIVE	0.901546

In diesem Fall geht das Modell mit hoher Wahrscheinlichkeit davon aus, dass der Text ein negatives Stimmungsbild vermittelt. Wenn Sie sich vor Augen führen, dass es sich um eine Beschwerde eines verärgerten Kunden handelt, erscheint dies durchaus eine plausible Einschätzung zu sein! Beachten Sie, dass die Pipeline im Rahmen von Aufgaben der Sentimentanalyse nur eines der beiden verwendeten Labels, `POSITIVE` oder `NEGATIVE`, zurückgibt, da beide Scores bzw. Werte in Summe eins ergeben und der jeweils andere Wert dementsprechend durch die Berechnung von `1-Score` ermittelt werden kann.

Werfen wir nun einen Blick auf eine andere häufig anzutreffende Aufgabenstellung: die Identifizierung von Eigennamen bzw. benannten Entitäten in Texten.

Named Entity Recognition

Die Vorhersage des Stimmungsbilds des Kundenfeedbacks ist ein guter erster Schritt. Oft möchten Sie jedoch erfahren, ob sich das Feedback auf einen bestimmten Artikel oder eine bestimmte Dienstleistung bezieht. Im NLP werden Objekte der realen Welt wie Produkte, Orte und Personen als *Named Entities*, also Eigennamen bzw. benannte Entitäten, bezeichnet, und das Extrahieren dieser Objekte aus Text wird *Eigennamenerkennung* bzw. *Named Entity Recognition* (NER) genannt. Um eine NER durchführen zu können, müssen wir die entsprechende Pipeline laden und sie mit unserer Kundenrezension füttern:

```
ner_tagger = pipeline("ner", aggregation_strategy="simple")
outputs = ner_tagger(text)
pd.DataFrame(outputs)
```

	entity_group	score	word	start	end
0	ORG	0.879010	Amazon	5	11
1	MISC	0.990859	Optimus Prime	36	49
2	LOC	0.999755	Germany	90	97
3	MISC	0.556569	Mega	208	212
4	PER	0.590256	##tron	212	216
5	ORG	0.669692	Decept	253	259
6	MISC	0.498350	##icons	259	264
7	MISC	0.775361	Megatron	350	358
8	MISC	0.987854	Optimus Prime	367	380
9	PER	0.812096	Bumblebee	502	511

Wie Sie sehen, hat die Pipeline alle Entitäten erkannt und jeder von ihnen eine Kategorie wie `ORG` (Organisation), `LOC` (Ort, engl. Location) oder `PER` (Person) zugewiesen. Hier haben wir das Argument `aggregation_strategy` verwendet, um die Wörter entsprechend den Vorhersagen des Modells zu gruppieren. Die Entität »Optimus Prime« besteht zum Beispiel aus zwei Wörtern, wird aber einer einzigen Kategorie zugeordnet: `MISC` (Verschiedenes, engl. Miscellaneous). Anhand der Scores können wir beurteilen, mit welcher Konfidenz das Modell die identifizierten Entitäten bzw. Eigennamen einschätzt. Wir sehen, dass es bei »Decepticons« und dem ersten Vorkommen von »Megatron« am unsichersten war, da es beide nicht als eine einzelne Entität zusammenfassen konnte.

Sehen Sie in der vorherigen Tabelle diese seltsamen Hash-Symbole (`#`) in der Spalte `word`? Diese werden vom *Tokenizer* des Modells erzeugt, der Wörter in einzelne Einheiten, sogenannte *Tokens*, zerlegt. In Kapitel 2 werden Sie noch mehr über die Tokenisierung erfahren.

Alle Eigennamen (bzw. benannte Entitäten) in einem Text zu identifizieren, ist hilfreich, doch manchmal würden wir gerne gezieltere Fragen stellen. Hierfür können wir auf das *Question Answering* (QA) bzw. die *automatische Fragenbeantwortung* zurückgreifen.

Question Answering

Beim Question Answering übermitteln wir dem Modell eine Textpassage, die wir als *Kontext* (engl. Context) bezeichnen, zusammen mit einer Frage, deren Antwort wir gerne extrahieren möchten. Das Modell gibt dann die Passage des Texts zurück, die der Antwort entspricht. Sehen wir uns einmal an, was wir erhalten, wenn wir eine konkrete Frage bezüglich unseres Kundenfeedbacks stellen:

```
reader = pipeline("question-answering")
question = "What does the customer want?"
outputs = reader(question=question, context=text)
pd.DataFrame([outputs])
```

	score	start	end	answer
0	0.631291	335	358	an exchange of Megatron

Wie wir sehen, hat die Pipeline neben der Antwort auch die Ganzzahlen `start` und `end` zurückgegeben, die den Indexen der Zeichen entsprechen, in denen die Antwort gefunden wurde (genau wie beim NER-Tagging). Dieser eingegrenzte Bereich wird als Antwortspanne (engl. Answer Span) bezeichnet, die den Beginn und das Ende der Textpassage, in der die Antwort enthalten ist, kennzeichnet. Es gibt verschiedene Arten des Question Answering, auf die wir noch in Kapitel 7 eingehen werden. Diese spezielle Art wird als *extraktives Question Answering* bezeichnet, da die Antwort direkt aus dem Text extrahiert wird.

Mit diesem Ansatz können Sie schnell relevante Informationen aus dem Feedback eines Kunden erkennen und extrahieren. Aber was ist, wenn Sie einen Berg von langatmigen Beschwerden erhalten und nicht die Zeit haben, sie alle zu lesen? Mal sehen, ob ein Modell, das Zusammenfassungen generiert, helfen kann!

Automatische Textzusammenfassung (Summarization)

Das Ziel der automatischen Textzusammenfassung (engl. Summarization) ist es, aus einem langen Text eine Kurzfassung zu erstellen, die alle relevanten Fakten enthält. Dies ist eine wesentlich kompliziertere Aufgabe als die vorherigen, da das Modell einen zusammenhängenden Text *generieren* muss. Wie Sie es bereits von den vorherigen Aufgaben kennen, können wir eine Zusammenfassungspipeline wie folgt instanziieren:

```
summarizer = pipeline("summarization")
outputs = summarizer(text, max_length=45, clean_up_tokenization_spaces=True)
print(outputs[0]['summary_text'])
```

```
 Bumblebee ordered an Optimus Prime action figure from your online store in German
y. Unfortunately, when I opened the package, I discovered to my horror that I had
been sent an action figure of Megatron instead.
```

Die ausgegebene Zusammenfassung ist gar nicht mal so schlecht! Obwohl Teile des ursprünglichen Texts lediglich kopiert wurden, war das Modell in der Lage, das Wesentliche des Problems zu erfassen und korrekt zu erkennen, dass »Bumblebee« (der am Ende des Texts vorkam) der Verfasser der Beschwerde ist. In diesem Beispiel können Sie auch sehen, dass wir einige Schlüsselwortargumente wie `max_length` und `clean_up_tokenization_spaces` an die Pipeline übergeben haben. Diese ermöglichen es uns, die Ergebnisse weiter zu optimieren.

Was aber, wenn Sie Feedback in einer Sprache erhalten, die Sie nicht verstehen? Dann können Sie Google Translate verwenden oder auch Ihr eigenes Transformer-Modell, das Ihnen die Übersetzung abnimmt!

Maschinelle Übersetzung (Translation)

Wie die automatische Textzusammenfassung ist auch die maschinelle Übersetzung (engl. Translation) eine Aufgabe, bei der die Ausgabe einem generierten Text entspricht. Verwenden wir nun eine Übersetzungspipeline, um einen englischen Text ins Deutsche zu übersetzen:

```
translator = pipeline("translation_en_to_de",
                      model="Helsinki-NLP/opus-mt-en-de")
outputs = translator(text, clean_up_tokenization_spaces=True, min_length=100)
print(outputs[0]['translation_text'])
```

```
Sehr geehrter Amazon, letzte Woche habe ich eine Optimus Prime Action Figur aus
Ihrem Online-Shop in Deutschland bestellt. Leider, als ich das Paket öffnete,
entdeckte ich zu meinem Entsetzen, dass ich stattdessen eine Action Figur von
Megatron geschickt worden war! Als lebenslanger Feind der Decepticons, Ich
hoffe, Sie können mein Dilemma verstehen. Um das Problem zu lösen, Ich fordere
einen Austausch von Megatron für die Optimus Prime Figur habe ich bestellt.
Anbei sind Kopien meiner Aufzeichnungen über diesen Kauf. Ich erwarte, bald von
Ihnen zu hören. Aufrichtig, Bumblebee.
```

Das Modell hat auch hier eine sehr gute Übersetzung hervorgebracht, bei der die förmlichen Pronomen der deutschen Sprache – wie »Ihrem« und »Sie« – korrekt verwendet werden. Außerdem haben wir hierdurch gezeigt, wie Sie das standardmäßig vorgegebene (engl. default) Modell in der Pipeline ersetzen, um das für Ihre Anwendung am besten geeignete Modell auswählen zu können – und auf dem Hugging Face Hub können Sie Modelle für Tausende von Sprachpaaren finden. Nehmen wir noch eine letzte Anwendung unter die Lupe, bevor wir wieder einen Schritt zurücktreten, um uns das gesamte Ökosystem von Hugging Face anzusehen.

Textgenerierung

Nehmen wir an, Sie möchten in der Lage sein, schneller auf Kundenfeedback zu antworten und dazu auf eine Autovervollständigungsfunktion zurückgreifen. Mit einem Textgenerierungsmodell (engl. Text Generation Model) könnten Sie dies wie folgt umsetzen:

```
generator = pipeline("text-generation")
response = "Dear Bumblebee, I am sorry to hear that your order was mixed up."
prompt = text + "\n\nCustomer service response:\n" + response
outputs = generator(prompt, max_length=200)
print(outputs[0]['generated_text'])
```

```
Dear Amazon, last week I ordered an Optimus Prime action figure from your online
store in Germany. Unfortunately, when I opened the package, I discovered to my
horror that I had been sent an action figure of Megatron instead! As a lifelong
enemy of the Decepticons, I hope you can understand my dilemma. To resolve the
issue, I demand an exchange of Megatron for the Optimus Prime figure I ordered.
Enclosed are copies of my records concerning this purchase. I expect to hear
from you soon. Sincerely, Bumblebee.
```

```
Customer service response:
Dear Bumblebee, I am sorry to hear that your order was mixed up. The order was
completely mislabeled, which is very common in our online store, but I can
appreciate it because it was my understanding from this site and our customer
service of the previous day that your order was not made correct in our mind and
that we are in a process of resolving this matter. We can assure you that your
order
```

Nun gut. Die generierte Vervollständigung würden wir wahrscheinlich nicht dazu verwenden, Bumblebee zu beruhigen. Die Idee dahinter sollte jedoch klar geworden sein.

Nachdem Sie nun ein paar spannende Anwendungen von Transformer-Modellen kennengelernt haben, fragen Sie sich vielleicht, wo diese trainiert werden. Alle Modelle, die wir in diesem Kapitel verwendet haben, sind frei verfügbar und bereits für die jeweilige Aufgabe feingetunt. Normalerweise möchten Sie die Modelle jedoch für Ihre eigenen Daten feintunen. In den folgenden Kapiteln werden Sie lernen, wie Sie das anstellen können.

Doch ein Modell zu trainieren, macht nur einen kleinen Teil eines jeden NLP-Projekts aus. Die Daten effizient zu verarbeiten, Ergebnisse mit Kollegen zu teilen und Ihre Arbeit reproduzierbar zu machen, sind ebenfalls wichtige Komponenten. Glücklicherweise ist die 🤗 Transformers-Bibliothek von einem großen Ökosystem nützlicher Tools umgeben, die einen Großteil des modernen Machine-Learning-Workflows unterstützen. Sehen wir uns dieses Ökosystem etwas genauer an.

Das Ökosystem von Hugging Face

Was mit der 🤗 Transformers-Bibliothek begann, hat sich schnell zu einem ganzen Ökosystem entwickelt, das aus vielen Bibliotheken und Tools besteht, mit denen Sie Ihre NLP- und Machine-Learning-Projekte schneller umsetzen können. Das Ökosystem von Hugging Face besteht hauptsächlich aus zwei Teilen: einer ganzen Reihe von Bibliotheken und dem Hub, wie in Abbildung 1-9 gezeigt. Die Bibliotheken stellen den Code zur Verfügung, während der Hub die Pretraining-Modellgewichte, Datensätze, Skripte für die Berechnung der Maße zur Evaluierung und mehr bereitstellt. In diesem Abschnitt werfen wir einen kurzen Blick auf die verschiedenen Komponenten. Wir überspringen die 🤗 Transformers-Bibliothek, da wir sie bereits vorgestellt haben und im Laufe des Buchs noch sehr viel mehr über sie erfahren werden.

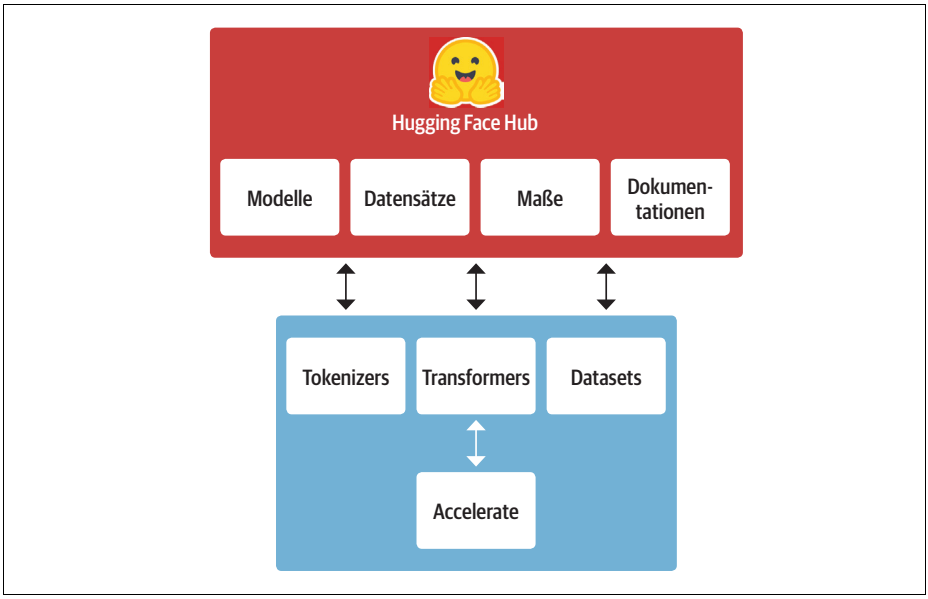


Abbildung 1-9: Ein Überblick über das Ökosystem von Hugging Face

Der Hugging Face Hub

Wie bereits erwähnt, ist Transfer Learning einer der Schlüsselfaktoren für den Erfolg von Transformer-Modellen, da es dadurch möglich ist, vortrainierte Modelle für neue Aufgaben wiederzuverwenden. Daher ist es von entscheidender Bedeutung, dass Sie vortrainierte Modelle schnell laden und Experimente mit ihnen durchführen können.

Der Hugging Face Hub enthält über 20.000 frei verfügbare Modelle. Wie Sie in Abbildung 1-10 sehen, gibt es Filter für Aufgaben (»Tasks«), Frameworks, Datensätze (»Datasets«) und vieles mehr. Diese sollen Ihnen helfen, sich im Hub zurechtzufinden und schnell vielversprechende Modellkandidaten zu finden. Wie Sie bereits bei der Verwendung von Pipelines feststellen konnten, ist das Laden eines vielversprechenden Modells dann buchstäblich nur eine Zeile Code entfernt. Das erleichtert es Ihnen, mit einer Vielzahl von Modellen zu experimentieren, und gibt Ihnen die Möglichkeit, sich auf die domänenspezifischen Aspekte Ihres Projekts zu konzentrieren.

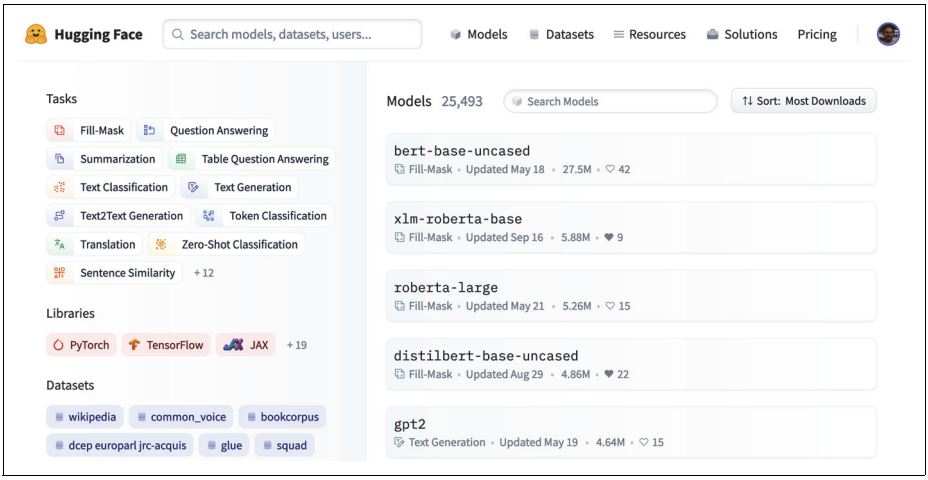


Abbildung 1-10: Die »Models«-Seite des Hugging Face Hub, mit Filterfunktionen auf der linken Seite und einer Liste von Modellen auf der rechten Seite

Zusätzlich zu den Modellgewichtungen bzw. -gewichten enthält der Hub auch Datensätze und Skripte zur Berechnung der Maße zur Evaluierung, mit denen Sie die veröffentlichten Ergebnisse reproduzieren oder zusätzliche Daten für Ihre Anwendung nutzen können.

Der Hub bietet auch sogenannte *Model-Cards* und *Dataset-Cards*, in denen allgemeine Aspekte zu den Modellen und Datensätzen dokumentiert sind und die Ihnen eine fundierte Entscheidung darüber ermöglichen sollen, ob sie die richtige Wahl für Sie darstellen. Eines der besten Features des Hub ist, dass Sie jedes Modell direkt über die verschiedenen aufgabenspezifischen interaktiven Widgets ausprobieren können (siehe Abbildung 1-11).

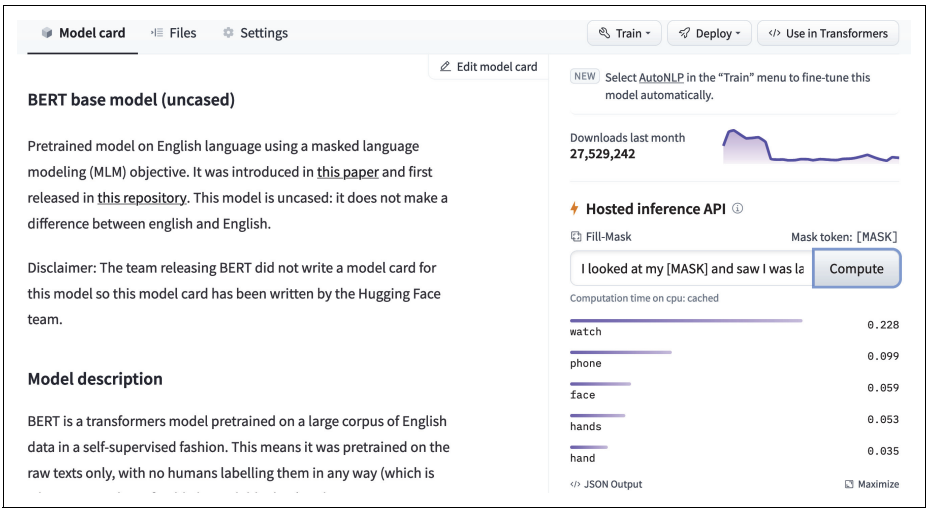


Abbildung 1-11: Ein Beispiel für eine Model-Card aus dem Hugging Face Hub: Das Inference-Widget, mit dem Sie mit dem Modell interagieren können, sehen Sie rechts.

Kommen wir nun zur 🤗 Tokenizers-Bibliothek.

PyTorch (*https://oreil.ly/AyTYC*) und TensorFlow (*https://oreil.ly/JOKgq*) bieten auch eigene Hubs, deren Nutzung sich insbesondere dann anbietet, wenn ein bestimmtes Modell oder ein bestimmter Datensatz nicht über den Hugging Face Hub verfügbar ist.

Die Tokenizers-Bibliothek von Hugging Face

Bei jedem der bisher gezeigten Pipeline-Beispiele wurde im Hintergrund ein Schritt zur Tokenisierung durchgeführt, bei dem der Rohtext in kleinere Teile bzw. Einheiten, sogenannte Tokens, zerlegt worden ist. Wie das im Einzelnen funktioniert, werden wir in Kapitel 2 erfahren. Für den Moment reicht es, zu wissen, dass Tokens Wörter, Teile von Wörtern oder einfach Zeichen wie Satzzeichen sein können. Transformer-Modelle werden auf Basis numerischer Darstellungen bzw. Repräsentationen dieser Tokens trainiert. Daher ist es für das gesamte NLP-Projekt von großer Bedeutung, diesen Schritt erfolgreich zu meistern!

Die 🤗 Tokenizers-Bibliothek (*https://oreil.ly/Z79jF*) bietet viele verschiedene Strategien zur Tokenisierung und kann Text dank ihres Rust-Backends extrem schnell in Tokens überführen.[13] Sie übernimmt auch alle Vor- und Nachverarbeitungsschritte (engl. Pre/Postprocessing), wie die Normalisierung (engl. Normalization) der Eingaben (engl. Inputs) und die Umwandlung der Ausgaben (engl. Outputs) des Modells in das gewünschte Format. Die Tokenizer der 🤗 Tokenizers-Bibliothek können Sie auf die gleiche Weise laden wie die Modellgewichte vortrainierter Modelle mit der 🤗 Transformers-Bibliothek.

Um Modelle trainieren und auch evaluieren zu können, benötigen wir einen Datensatz und Maße, mit denen wir die Güte- bzw. Qualität des Modells beurteilen können.[14] Werfen wir also einen Blick auf die 🤗 Datasets-Bibliothek, die hierfür vorgesehen ist.

Die Datasets-Bibliothek von Hugging Face

Datensätze zu laden, zu verarbeiten und zu speichern, kann recht mühsam sein, vor allem wenn die Datensätze so groß werden, dass sie nicht mehr in den Arbeitsspeicher Ihres Notebooks passen. Darüber hinaus müssen Sie in der Regel verschiedene Skripte implementieren, mit denen Sie die Daten herunterladen und in ein Standardformat überführen können.

Die 🤗 Datasets-Bibliothek (*https://oreil.ly/959YT*) vereinfacht diesen Prozess, indem sie eine Standardschnittstelle für Tausende von Datensätzen bietet, die auf

13 Rust (*https://rust-lang.org*) ist eine hochgradig leistungsstarke Programmiersprache.

14 Anm. d. Übers.: Zum Zeitpunkt der Übersetzung des Buchs gibt es inzwischen zusätzlich noch die 🤗 Evaluate-Bibliothek (*https://github.com/huggingface/evaluate*), wodurch sich das Vorgehen im Rahmen der Evaluierung leicht verändert hat. Statt z. B. `datasets.load_metric("sacrebleu")` kann inzwischen `evaluate.load("sacrebleu")` verwendet werden. Der im Buch beschriebene Ansatz über die 🤗 Datasets-Bibliothek (*https://oreil.ly/959YT*) ist in den aktuellen Versionen dennoch möglich.

dem Hub (*https://oreil.ly/Rdhcu*) zu finden sind. Außerdem bietet sie eine intelligente Zwischenspeicherung (engl. Caching), sodass die Vorverarbeitung nicht jedes Mal, wenn Sie Ihren Code erneut ausführen, durchgeführt werden muss. Ferner hilft sie dabei, die mit einem begrenzten Arbeitsspeicher verbundenen Einschränkungen zu überwinden, indem sie einen speziellen Mechanismus namens *Memory Mapping* nutzt, bei dem der Inhalt einer Datei im virtuellen Speicher gespeichert und es mehreren Prozessen ermöglicht wird, Dateien auf effizientere Weise zu modifizieren. Die Bibliothek ist außerdem mit gängigen Frameworks wie Pandas und NumPy kompatibel, sodass Sie nicht auf Ihre bevorzugten Tools zur Datenverarbeitung verzichten müssen.

Allerdings sind hochwertige Datensätze und leistungsstarke Modelle wertlos, wenn Sie deren Leistung nicht zuverlässig beurteilen können. Leider gibt es bei den klassischen NLP-Maßen bzw. -Metriken viele verschiedene Implementierungen, die leicht variieren und irreführende Ergebnissen erzielen können. Indem es die Skripte für viele Maße bereitstellt, trägt die 🤗 Datasets-Bibliothek dazu bei, dass Experimente besser reproduzierbar und die Ergebnisse vertrauenswürdiger sind.

Mit der 🤗 Transformers-, der 🤗 Tokenizers- und der 🤗 Datasets-Bibliothek haben wir nun alle notwendigen Voraussetzungen, um unsere eigenen Transformer-Modelle trainieren zu können! Wie wir jedoch noch in Kapitel 10 sehen werden, gibt es Situationen, in denen wir die Trainingsschleife noch stärker modifizieren müssen, als dies die `Trainer`-Klasse zulässt. Hier kommt die letzte Bibliothek des Ökosystems ins Spiel: 🤗 Accelerate.

Die Accelerate-Bibliothek von Hugging Face

Wenn Sie jemals Ihr eigenes Trainingsskript in PyTorch schreiben mussten, dann haben Sie sich wahrscheinlich schon einmal den Kopf darüber zerbrochen, wie Sie den Code, der auf Ihrem Laptop läuft, auf das Cluster Ihres Unternehmens bringen. Die 🤗 Accelerate-Bibliothek (*https://oreil.ly/iRfDe*) fügt Ihren normalen Trainingsschleifen eine zusätzliche Abstraktionsebene hinzu, die sich um die gesamte benutzerdefinierte Logik kümmert, die für die Trainingsinfrastruktur erforderlich ist. Dies vereinfacht eine gegebenenfalls erforderliche Änderung der Infrastruktur und beschleunigt (»accelerates«) somit Ihre Arbeitsabläufe.

Soweit zu den wesentlichen Komponenten des Open-Source-Ökosystems von Hugging Face. Doch bevor wir dieses Kapitel abschließen, werfen wir noch einen Blick auf einige der typischen Herausforderungen, die beim Deployment von Transformer-Modellen in der Praxis auftreten.

Die größten Herausforderungen im Zusammenhang mit Transformer-Modellen

In diesem Kapitel haben Sie einen ersten Eindruck davon bekommen, wie vielfältig die NLP-Aufgaben sind, die mit Transformer-Modellen bewältigt werden können.

Wenn man die Schlagzeilen in den Medien liest, klingt es manchmal so, als seien sie zu allem fähig. Doch so nützlich sie auch sein mögen, Transformer sind keineswegs der Weisheit letzter Schluss. Im Folgenden finden Sie einige mit ihnen einhergehende Herausforderungen, die wir im Laufe des Buchs noch genauer beleuchten werden:

Sprache
: Die Forschung im Bereich des NLP wird überwiegend in englischer Sprache betrieben. Zwar gibt es einige Modelle für andere Sprachen, doch insbesondere für Sprachen, die selten sind oder für die nur wenige Ressourcen zur Verfügung stehen, ist es schwieriger, vortrainierte Modelle zu finden. In Kapitel 4 untersuchen wir mehrsprachige Transformer-Modelle und ihre Fähigkeit, einen sprachübergreifenden Zero-Shot-Transfer durchzuführen.

Datenverfügbarkeit
: Obwohl wir Transfer Learning anwenden können und so die Menge an gelabelten Trainingsdaten, die unsere Modelle benötigen, drastisch verringern können, sind dies im Vergleich zu der Menge, die ein Mensch benötigt, um die Aufgabe zu bewältigen, nach wie vor viele. Wie Sie Situationen meistern können, bei denen Sie nur auf wenige oder gar keine gelabelten Daten zurückgreifen können, erfahren Sie in Kapitel 9.

Mit langen Texten bzw. Dokumenten arbeiten
: Der Self-Attention-Mechanismus funktioniert sehr gut bei Texten, die nur aus einem Absatz bestehen. Bei längeren Texten, z. B. ganzen Dokumenten, ist er jedoch sehr rechenintensiv. Ansätze, die diesem Problem entgegenwirken, finden Sie in Kapitel 11.

Intransparenz (engl. Opacity)
: Wie andere Deep-Learning-Modelle auch, sind Transformer weitgehend intransparent. Es ist schwer oder gar unmöglich, herauszufinden, »warum« ein Modell eine bestimmte Vorhersage getroffen hat. Dies ist eine besonders große Herausforderung, wenn Modelle dazu eingesetzt werden, schwerwiegende Entscheidungen zu treffen. In den Kapiteln Kapitel 2 und Kapitel 4 werden wir einige Möglichkeiten erkunden, wie sich Fehler (engl. Errors) bei Vorhersagen von Transformer-Modellen untersuchen lassen.

Bias bzw. Voreingenommenheit
: Transformer-Modelle werden in erster Linie auf der Grundlage von Textdaten vortrainiert, die aus dem Internet stammen. Dadurch werden alle Vorurteile (engl. Bias), die sich in den Daten widerspiegeln, von den Modellen übernommen. Sicherzustellen, dass diese weder rassistisch noch sexistisch oder anderweitig voreingenommen sind, ist eine anspruchsvolle Aufgabe. Auf einige dieser Probleme gehen wir in Kapitel 10 näher ein.

Diese Herausforderungen mögen zwar entmutigend wirken, doch viele davon können überwunden werden. Wir werden diese Themen nicht nur in den genannten Kapiteln, sondern auch in fast allen anderen Kapiteln ansprechen.

Zusammenfassung

Hoffentlich sind Sie jetzt gespannt darauf, wie Sie diese vielseitigen Modelle trainieren und in Ihre eigenen Anwendungen integrieren können! Sie haben in diesem Kapitel gesehen, dass Sie mit nur wenigen Zeilen Code hochmoderne Modelle zur Klassifizierung, zur Erkennung von Eigennamen (Named Entity Recognition), zur Beantwortung von Fragen (Question Answering), zur Übersetzung und zur Zusammenfassung verwenden können. Doch das ist lediglich die »Spitze des Eisbergs«.

In den folgenden Kapiteln lernen Sie, wie Sie Transformer-Modelle für eine Vielzahl von Anwendungsfällen, wie z.B. für den Aufbau eines Textklassifikators oder eines kompakteren Modells für die Produktion oder sogar für das Training eines Sprachmodells von Grund auf, anpassen können. Dabei werden wir einen praktischen Ansatz verfolgen, d.h., für jedes behandelte Konzept gibt es begleitenden Code, den Sie auf Google Colab oder Ihrem eigenen, mit einer GPU ausgestatteten Rechner ausführen können.

Nachdem wir nun mit den grundlegenden Konzepten hinter Transformern vertraut sind, ist es an der Zeit, dass wir unsere erste Anwendung in Angriff nehmen: die Textklassifizierung, das Thema des nächsten Kapitels.

KAPITEL 2

Textklassifizierung

Die Textklassifizierung ist eine der am häufigsten angewandten Aufgaben im Bereich des NLP. Sie kann für eine Vielzahl von Anwendungen eingesetzt werden, z.B. um Kundenfeedback in Kategorien einzuteilen oder Support-Tickets entsprechend ihrer jeweiligen Sprache weiterzuleiten. Die Chancen stehen gut, dass der Spamfilter Ihres E-Mail-Programms Ihren Posteingang mithilfe der Textklassifizierung vor einer Flut von unerwünschtem Spam schützt!

Eine andere häufig verwendete Art der Textklassifizierung ist die Sentimentanalyse, die (wie wir in Kapitel 1 gesehen haben) darauf abzielt, die Polarität eines bestimmten Texts, d.h. die Stimmungslage, die aus diesem hervorgeht, zu ermitteln. Ein Unternehmen wie Tesla könnte beispielsweise Twitter-Beiträge wie den in Abbildung 2-1 analysieren, um herauszufinden, ob die neuen Autodächer des Unternehmens Anklang finden oder nicht.

Abbildung 2-1: Die Analyse von Twitter-Inhalten kann nützliches Kundenfeedback liefern (mit freundlicher Genehmigung von Aditya Veluri).

Stellen Sie sich nun vor, Sie sind ein Data Scientist, der ein System entwickeln muss, das automatisch emotionale Zustände wie »Wut« (engl. Anger) oder »Freude« (engl. Joy) erkennen kann, die von den Nutzern über das Produkt Ihres Unternehmens auf Twitter geäußert werden. In diesem Kapitel gehen wir diese Aufgabe mit einer Variante von BERT namens DistilBERT an.[1] Der Hauptvorteil dieses Modells besteht darin, dass es eine vergleichbare Leistung wie BERT erzielt, jedoch bedeutend kleiner und effizienter ist. Dadurch sind wir in der Lage, einen Klassifikator in nur wenigen Minuten zu trainieren. Falls Sie ein größeres BERT-Modell trainieren möchten, können Sie einfach den Checkpoint, den Sie für das vortrainierte Modell laden, anpassen. Ein *Checkpoint* entspricht einem Satz an Gewichten (auch Gewichtung genannt), die in einer bestimmten Transformer-Architektur geladen werden.

In diesem Zusammenhang werden wir auch zum ersten Mal mit drei der Kernbibliotheken aus dem Hugging-Face-Ökosystem arbeiten: 🤗 Datasets, 🤗 Tokenizers und 🤗 Transformers. Diese Bibliotheken gestatten es uns, schnell vom Rohtext zu einem für das Feintuning geeigneten Modell zu gelangen, das zur Auswertung neuer Tweets (»Inferenz«) verwendet werden kann (siehe Abbildung 2-2). Also, ganz im Sinne von Optimus Prime[2]: Rein ins Vergnügen, »transformieren und los geht's!«.

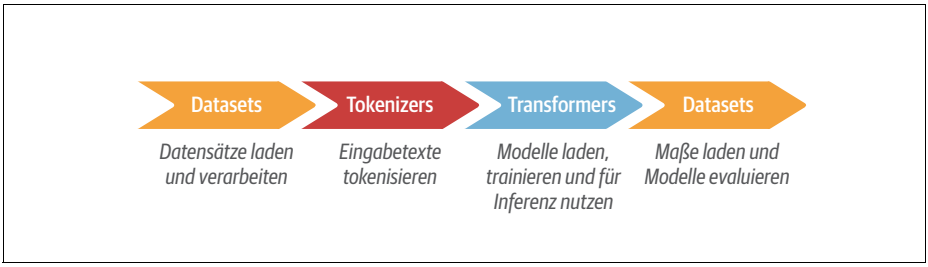


Abbildung 2-2: Eine typische Pipeline für das Training von Transformer-Modellen, bei der die Bibliotheken 🤗 Datasets, 🤗 Tokenizers und 🤗 Transformers verwendet werden

Der Datensatz

Um unseren Emotionsdetektor zu entwickeln, verwenden wir einen großartigen Datensatz, auf den in einem Artikel zurückgegriffen wurde, in dem untersucht wird, wie Emotionen in englischsprachigen Twitter-Nachrichten zum Ausdruck gebracht werden.[3] Im Gegensatz zu anderen Datensätzen, die im Rahmen der Sentimentanalyse genutzt werden und nur »positive« und »negative« Polaritäten bein-

1 V. Sanh et al., »DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter« (*https://arxiv.org/abs/1910.01108*), (2019).

2 Optimus Prime ist der Anführer einer Roboterrasse in der beliebten Transformers-Fernsehserie für Kinder (und Junggebliebene!)

3 E. Saravia et al., »CARER: Contextualized Affect Representations for Emotion Recognition,« *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing* (Okt–Nov 2018): 3687–3697, *http://dx.doi.org/10.18653/v1/D18-1404*.

halten, enthält dieser Datensatz sechs grundlegende Emotionen: Wut (engl. Anger), Liebe (engl. Love), Angst (engl. Fear), Freude (engl. Joy), Trauer (engl. Sadness) sowie Überraschung (engl. Surprise). Unsere Aufgabe besteht darin, ein Modell zu trainieren, das einen gegebenen Tweet in eine dieser Emotionen einordnen bzw. klassifizieren kann.

Ein erster Blick auf die Datasets-Bibliothek von Hugging Face

Die Daten können wir mithilfe der 🤗 Datasets-Bibliothek vom Hugging Face Hub (*https://oreil.ly/959YT*) herunterladen. Um zu sehen, welche Datensätze auf dem Hub verfügbar sind, können wir die Funktion `list_datasets()` aufrufen:

```
from datasets import list_datasets

all_datasets = list_datasets()
print(f"Derzeit sind {len(all_datasets)} Datensätze auf dem Hub verfügbar")
print(f"Die ersten 10 sind: {all_datasets[:10]}")
```

```
Derzeit sind 1753 Datensätze auf dem Hub verfügbar
Die ersten 10 sind: ['acronym_identification', 'ade_corpus_v2', 'adversarial_qa',
'aeslc', 'afrikaans_ner_corpus', 'ag_news', 'ai2_arc', 'air_dialogue',
'ajgt_twitter_ar', 'allegro_reviews']
```

Wie Sie sehen, wurde jedem Datensatz ein Name gegeben. Laden wir nun also den Datensatz `emotion` mithilfe der Funktion `load_dataset()`:

```
from datasets import load_dataset

emotions = load_dataset("emotion")
```

Wenn wir unser Objekt `emotions` in Augenschein nehmen:

```
emotions
```

```
DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 16000
    })
    validation: Dataset({
        features: ['text', 'label'],
        num_rows: 2000
    })
    test: Dataset({
        features: ['text', 'label'],
        num_rows: 2000
    })
})
```

sehen wir, dass es sich um ein Python-Dictionary handelt, wobei jeder Schlüssel (engl. Key) einem anderen der bereits zuvor aufgeteilten Teildatensätze (engl. Splits) des Datensatzes entspricht. Um auf einen der Teildatensätze zuzugreifen, können wir die übliche Dictionary-Syntax verwenden:

```
train_ds = emotions["train"]
train_ds
```

```
Dataset({
    features: ['text', 'label'],
    num_rows: 16000
})
```

und erhalten eine Instanz der `Dataset`-Klasse. Das `Dataset`-Objekt ist eine der wichtigsten Datenstrukturen in der 🤗 Datasets-Bibliothek. Im weiteren Verlauf dieses Buchs werden wir viele seiner Funktionen kennenlernen. Zunächst einmal verhält es sich wie ein gewöhnliches Python-Array bzw. eine Liste. Die Länge des Objekts können wir daher wie folgt abfragen:

```
len(train_ds)
```

```
16000
```

oder auf ein einzelnes Beispiel (bzw. Beobachtung) über seinen Index zugreifen:

```
train_ds[0]
```

```
{'label': 0, 'text': 'i didnt feel humiliated'}
```

Wie Sie sehen, werden die einzelnen Zeilen als Dictionarys dargestellt, wobei die Schlüssel den Spaltennamen entsprechen:

```
train_ds.column_names
```

```
['text', 'label']
```

und die Werte jeweils aus dem Tweet und der Emotion bestehen. Dieses Format ist der Tatsache geschuldet, dass die 🤗 Datasets-Bibliothek auf *Apache Arrow* (*https://arrow.apache.org*) basiert, das ein typisiertes spaltenorientiertes Format verwendet, das im Vergleich zu nativem Python effizienter in Bezug auf den Speicherbedarf ist. Welche Datentypen im Hintergrund verwendet werden, können Sie feststellen, indem Sie auf das Attribut `features` eines `Dataset`-Objekts zugreifen:

```
print(train_ds.features)
```

```
{'text': Value(dtype='string', id=None), 'label': ClassLabel(num_classes=6,
names=['sadness', 'joy', 'love', 'anger', 'fear', 'surprise'], names_file=None,
id=None)}
```

In diesem Fall ist der Datentyp der `text`-Spalte `string`, während die `label`-Spalte ein spezielles `ClassLabel`-Objekt ist, das Informationen zu den Kategorienamen (bzw. Klassennamen) und deren Zuordnung zu Ganzzahlen enthält. Zudem können wir auf mehrere Zeilen zugreifen, indem wir den Slice-Operator verwenden:

```
print(train_ds[:5])
```

```
{'text': ['i didnt feel humiliated', 'i can go from feeling so hopeless to so
damned hopeful just from being around someone who cares and is awake', 'im
grabbing a minute to post i feel greedy wrong', 'i am ever feeling nostalgic
about the fireplace i will know that it is still on the property', 'i am feeling
grouchy'], 'label': [0, 0, 3, 2, 3]}
```

Beachten Sie, dass in diesem Fall die Werte des Dictionarys nun Listen und keine einzelnen Elemente sind. Wir können auch die vollständige Spalte über den entsprechenden Spaltennamen abrufen:

```
print(train_ds["text"][:5])
```

```
['i didnt feel humiliated', 'i can go from feeling so hopeless to so damned
hopeful just from being around someone who cares and is awake', 'im grabbing a
minute to post i feel greedy wrong', 'i am ever feeling nostalgic about the
fireplace i will know that it is still on the property', 'i am feeling grouchy']
```

Nachdem wir nun gesehen haben, wie Daten mithilfe 🤗 der Datasets-Bibliothek geladen und inspiziert werden können, sollten wir noch ein wenig dem Inhalt unserer Tweets auf den Zahn fühlen.

Was ist, wenn sich der Datensatz, den ich verwenden möchte, nicht auf dem Hub befindet?

Im Rahmen der meisten Beispiele in diesem Buch verwenden wir den Hugging Face Hub, um die Datensätze herunterzuladen. In vielen Fällen werden Sie jedoch mit Daten arbeiten, die entweder auf Ihrem Laptop oder auf einem Remote-Server Ihres Unternehmens gespeichert sind. Die 🤗 Datasets-Bibliothek bietet mehrere Skripte, mit denen Datensätze geladen werden können – sowohl lokale als auch auf Remote-Servern gespeicherte. Beispiele für die gängigsten Datenformate finden Sie in Tabelle 2-1.

Tabelle 2-1: Wie Sie Datensätze verschiedener Formate laden

Datenformat	Skript zum Laden	Beispiel
CSV	`csv`	`load_dataset("csv", data_files="my_file.csv")`
Text	`text`	`load_dataset("text", data_files="my_file.txt")`
JSON	`json`	`load_dataset("json", data_files="my_file.jsonl")`

Wie Sie erkennen können, müssen wir für jedes Datenformat nur das Skript, mit dem der Datensatz in dem entsprechenden Format geladen werden kann, an die Funktion `load_dataset()` übergeben sowie das Argument `data_files`, das den Pfad oder die URL zu einer oder mehreren Dateien angibt. Die Quelldateien für den `emotion`-Datensatz werden zum Beispiel auf Dropbox gehostet. Dadurch können Sie statt des gesamten Datensatzes zunächst auch nur einen der zuvor aufgeteilten Teildatensätze herunterladen und verwenden:

```
dataset_url = "https://www.dropbox.com/s/1pzkadrvffbqw6o/train.txt"
!wget {dataset_url}
```

Wenn Sie sich wundern, warum in dem vorangehenden Shell-Befehl ein `!` vorkommt – das liegt daran, dass wir die Befehle in einem Jupyter Notebook ausführen. Lassen Sie das Präfix einfach weg, wenn Sie den Datensatz innerhalb eines Terminals herunterladen und entpacken möchten. Werfen wir nun einen Blick auf die erste Zeile der Datei *train.txt*:

```
!head -n 1 train.txt
i didnt feel humiliated;sadness
```

Wie Sie sehen, sind keine Spaltenüberschriften vorhanden, und die einzelnen Tweets und Emotionen sind durch ein Semikolon voneinander getrennt. Nichtsdestotrotz ähnelt dies dem Aufbau einer CSV-Datei, weshalb wir das Skript `csv` verwenden und über das Argument `data_files` auf die Datei *train.txt* verweisen können, um den Datensatz lokal zu laden:

```
emotions_local = load_dataset("csv", data_files="train.txt", sep=";",
                              names=["text", "label"])
```

Hier haben wir auch die Art des Trennzeichens und die Namen der Spalten angegeben. Eine noch einfachere Methode ist es, über das Argument `data_files` auf die URL selbst zu verweisen,

```
dataset_url = "https://www.dropbox.com/s/1pzkadrvffbqw6o/train.txt?dl=1"
emotions_remote = load_dataset("csv", data_files=dataset_url, sep=";",
                               names=["text", "label"])
```

wodurch der Datensatz automatisch für Sie heruntergeladen und zwischengespeichert wird. Wie Sie sehen, ist die Funktion `load_dataset()` sehr vielseitig. Wir empfehlen Ihnen, einen Blick in die Dokumentation der 🤗 Datasets-Bibliothek (*https://oreil.ly/Jodu4*) zu werfen, um einen umfassenderen Überblick zu erhalten.

Dataset-Objekte in DataFrames überführen

Obwohl die 🤗 Datasets-Bibliothek viele Low-Level-Funktionen zur Aufteilung und Manipulierung unserer Daten bietet, ist es oftmals praktisch, ein `Dataset`-Objekt in einen Pandas-`DataFrame` zu konvertieren, damit wir auf die High-Level-APIs zur Datenvisualisierung zugreifen können. Zur Konvertierung bietet die 🤗 Datasets-Bibliothek eine Methode namens `set_format()`, die es uns erlaubt, das *Ausgabeformat* (engl. Output Format) des `Dataset`-Objekts zu ändern. Beachten Sie, dass das zugrunde liegende *Datenformat* (eine Arrow-Tabelle) dadurch nicht verändert wird und Sie – falls erforderlich – zu einem späteren Zeitpunkt auf ein anderes Format wechseln können:

```
import pandas as pd

emotions.set_format(type="pandas")
df = emotions["train"][:]
df.head()
```

	text	label
0	i didnt feel humiliated	0
1	i can go from feeling so hopeless to so damned...	0
2	im grabbing a minute to post i feel greedy wrong	3
3	i am ever feeling nostalgic about the fireplac...	2
4	i am feeling grouchy	3

Wie Sie feststellen können, wurden die Spaltenüberschriften beibehalten, und die ersten paar Zeilen stimmen mit den Daten überein, die wir zuvor betrachtet haben. Allerdings werden die Labels nun als Ganzzahlen dargestellt. Dementsprechend sollten wir die Methode `int2str()` auf die Feature-Spalte `label` anwenden, um eine neue Spalte in unserem `DataFrame` zu erstellen, die die entsprechenden Namen der Labels enthält:

```
def label_int2str(row):
    return emotions["train"].features["label"].int2str(row)

df["label_name"] = df["label"].apply(label_int2str)
df.head()
```

	text	label	label_name
0	i didnt feel humiliated	0	sadness
1	i can go from feeling so hopeless to so damned...	0	sadness
2	im grabbing a minute to post i feel greedy wrong	3	anger
3	i am ever feeling nostalgic about the fireplac...	2	love
4	i am feeling grouchy	3	anger

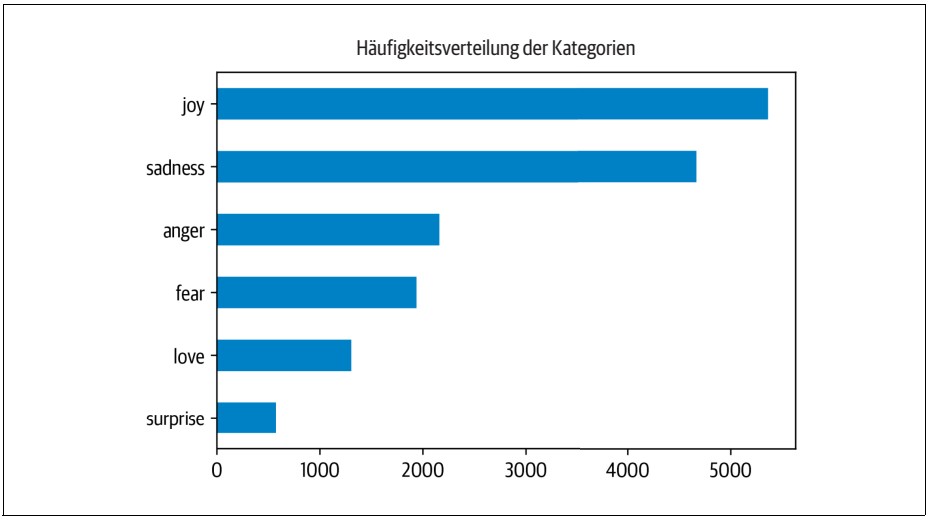
Ehe wir uns an den Aufbau eines Klassifikators machen, sollten wir den Datensatz noch etwas genauer inspizieren. Wie Andrej Karpathy in seinem berühmten Blogbeitrag »A Recipe for Training Neural Networks« (*https://oreil.ly/bNayo*) ausgeführt hat, ist es für das Training erstklassiger Modelle unerlässlich, »eins mit den Daten zu werden«!

Ein Blick auf die Verteilung der Kategorien

Wann immer Sie an Problemen im Bereich der Textklassifizierung arbeiten, ist es eine gute Idee, zu untersuchen, wie sich die Beispiele auf die einzelnen Kategorien (engl. Class Distribution) verteilen. Ein Datensatz, dessen Beispiele in Bezug auf die Kategorien ungleich verteilt sind, erfordert möglicherweise eine andere Vorgehensweise in Bezug auf die zum Training herangezogene Verlustfunktion und die zur Evaluierung verwendeten Qualitätsmaße als ein ausgewogener Datensatz.

Mithilfe von Pandas und Matplotlib können wir rasch die Verteilung der Kategorien (bzw. Klassen) visualisieren:

```
import matplotlib.pyplot as plt

df["label_name"].value_counts(ascending=True).plot.barh()
plt.title("Häufigkeitsverteilung der Kategorien")
plt.show()
```



In diesem Fall sehen wir, dass der Datensatz äußerst unausgewogen ist. Die Kategorien `joy` (Freude) und `sadness` (Trauer) kommen häufig vor, wohingegen die Kategorien `love` (Liebe) und `surprise` (Überraschung) etwa 5 bis 10 Mal seltener vertreten sind. Es gibt mehrere Möglichkeiten, wie Sie mit unausgewogenen Daten verfahren können, unter anderem können Sie:

- eine Zufallsstichprobe ziehen, wobei Sie die Kategorien, die seltener vorkommen (engl. Minority Class), relativ häufiger ziehen.
- eine Zufallsstichprobe ziehen, wobei Sie die Kategorien, die häufiger vorkommen (engl. Majority Class), relativ seltener ziehen.
- mehr gelabelte Daten für die unterrepräsentierten Kategorien sammeln bzw. erfassen.

Der Einfachheit halber arbeiten wir in diesem Kapitel mit den ursprünglichen Daten, d. h., wir belassen die Daten hinsichtlich der Kategorien unausgewogen. Wenn Sie mehr über diese Sampling-Verfahren (im Deutschen auch Stichprobenziehungsverfahren genannt) erfahren möchten, empfehlen wir Ihnen, mal einen Blick auf die Bibliothek Imbalanced-learn (*https://oreil.ly/5XBhb*) zu werfen. Stellen Sie jedoch sicher, dass Sie die Sampling-Verfahren nicht anwenden, *bevor* Sie Ihre Daten in einen Trainings- und einen Testdatensatz aufteilen, da es sonst (Anm. d. Übers.: infolge einer mehrfachen Berücksichtigung von Beispielen) zu einem erheblichen Leakage zwischen den Teildatensätzen (engl. Splits) kommt!

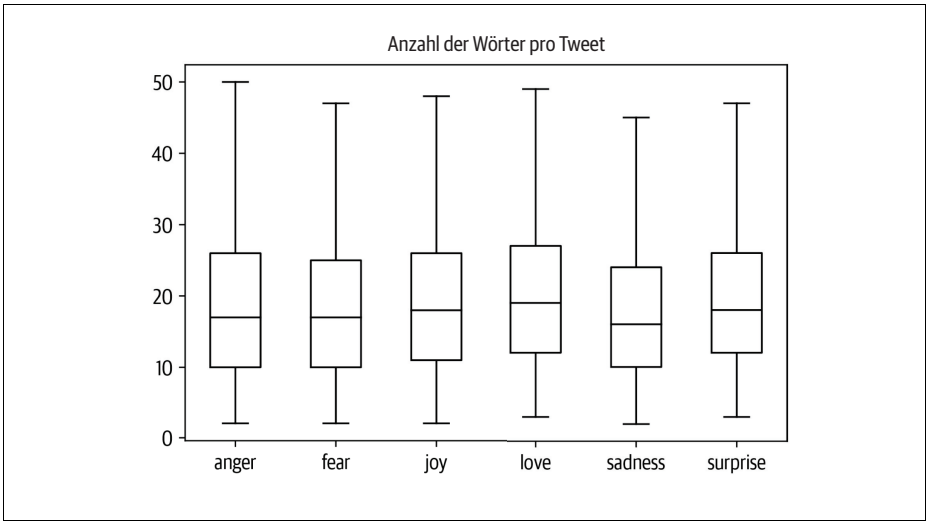
Nachdem wir uns nun die Kategorien angesehen haben, sollten wir noch einen Blick auf die Tweets selbst werfen.

Wie lang sind unsere Tweets?

Bei Transformer-Modellen darf die Eingabesequenz nur eine bestimmte Länge aufweisen, was als *maximale Kontextlänge* (engl. Maximum Context Size) bezeichnet

wird. Für Anwendungen, die DistilBERT verwenden, beträgt die maximale Kontextlänge 512 Tokens, was ein paar Absätzen Text entspricht. Wie wir im nächsten Abschnitt sehen werden, handelt es sich bei einem Token um ein einzelnes Stück Text. Für den Moment fassen wir ein Token als ein einzelnes Wort auf. Um eine grobe Einschätzung darüber zu erhalten, wie unterschiedlich lang die Tweets je nach der zum Ausdruck gebrachten Emotion ausfallen, können wir uns ansehen, wie die Anzahl an Wörtern pro Tweet hinsichtlich der Emotionen verteilt ist:

```
df["Anzahl der Wörter pro Tweet"] = df["text"].str.split().apply(len)
df.boxplot("Anzahl der Wörter pro Tweet", by="label_name", grid=False,
          showfliers=False, color="black")
plt.suptitle("")
plt.xlabel("")
plt.show()
```



Aus der Abbildung ist ersichtlich, dass – im Hinblick auf alle Emotionen – die meisten Tweets etwa 15 Wörter lang sind und die längsten Tweets deutlich unter der maximalen Kontextlänge von DistilBERT liegen. Texte, die länger als die maximale Kontextlänge eines Modells sind, müssen ansonsten gekürzt bzw. gestutzt (engl. truncated) werden. Hierbei können wichtige Informationen verloren gehen, was wiederum zu Leistungseinbußen führt. Im vorliegenden Fall stellt dies somit kein Problem dar.

Überlegen wir uns nun, wie wir diese Rohtexte in ein Format konvertieren können, das sich für die 🤗 Transformers-Bibliothek eignet. Wenn wir schon dabei sind, sollten wir auch das Ausgabeformat unseres Datensatzes zurücksetzen, da wir das `DataFrame`-Format nicht länger benötigen:

```
emotions.reset_format()
```

Vom Text zu Tokens

Transformer-Modelle wie DistilBERT akzeptieren keine reinen Strings als Eingabe. Stattdessen gehen sie davon aus, dass der Text *tokenisiert* und in Form numerischer Vektoren *codiert* wurde. Bei der Tokenisierung wird ein String in die im Modell verwendeten atomaren Einheiten zerlegt. Es gibt mehrere Tokenisierungsstrategien, die angewendet werden können. Normalerweise wird auf Basis eines Korpus gelernt, in welche Untereinheiten die Wörter optimalerweise unterteilt werden sollten. Bevor wir uns den Tokenizer ansehen, der für DistilBERT verwendet wird, ist es hilfreich, sich zwei Extremfälle vor Augen zu führen: die Tokenisierung auf der Ebene von *Zeichen* (engl. Character Tokenization) sowie auf der Ebene von *Wörtern* (engl. Word Tokenization).

Tokenisierung auf der Ebene von Zeichen (Character Tokenization)

Das einfachste Tokenisierungsschema besteht darin, jedes Zeichen einzeln in das Modell einzuspeisen. In Python handelt es sich bei `str`-Objekten im Grunde genommen um Arrays, weshalb wir eine Tokenisierung auf der Ebene von Zeichen quasi im Handumdrehen mit nur einer Zeile Code implementieren können:

```
text = "Tokenizing text is a core task of NLP."
tokenized_text = list(text)
print(tokenized_text)
```

```
['T', 'o', 'k', 'e', 'n', 'i', 'z', 'i', 'n', 'g', ' ', 't', 'e', 'x', 't', ' ',
'i', 's', ' ', 'a', ' ', 'c', 'o', 'r', 'e', ' ', 't', 'a', 's', 'k', ' ', 'o',
'f', ' ', 'N', 'L', 'P', '.']
```

Auch wenn dies ein guter Anfang ist, sind wir noch nicht am Ziel. Bei unserem Modell muss jedes Zeichen in eine Ganzzahl konvertiert werden – ein Vorgang, der im Englischen manchmal als *Numericalization* bezeichnet wird. Eine einfache Möglichkeit, dies zu erreichen, besteht darin, jedes der verschiedenen Tokens (in diesem Fall Zeichen) mit einer eindeutigen Ganzzahl zu codieren:

```
token2idx = {ch: idx for idx, ch in enumerate(sorted(set(tokenized_text)))}
print(token2idx)
```

```
{' ': 0, '.': 1, 'L': 2, 'N': 3, 'P': 4, 'T': 5, 'a': 6, 'c': 7, 'e': 8, 'f': 9,
'g': 10, 'i': 11, 'k': 12, 'n': 13, 'o': 14, 'r': 15, 's': 16, 't': 17, 'x': 18,
'z': 19}
```

Dadurch erhalten wir eine Zuordnung (engl. Mapping), bei der jedes in unserem Vokabular enthaltene Zeichen einer eindeutigen Ganzzahl entspricht. Mithilfe des `token2idx`-Dicionarys können wir nun den tokenisierten Text in eine Liste von Ganzzahlen überführen:

```
input_ids = [token2idx[token] for token in tokenized_text]
print(input_ids)
```

```
[5, 14, 12, 8, 13, 11, 19, 11, 13, 10, 0, 17, 8, 18, 17, 0, 11, 16, 0, 6, 0, 7,
14, 15, 8, 0, 17, 6, 16, 12, 0, 14, 9, 0, 3, 2, 4, 1]
```

Jedes Token wurde nun einem eindeutigen numerischen Identifikator (ID) zugeordnet (daher der Name `input_ids`). Der letzte Schritt besteht darin, das Array (bzw. die Liste) `input_ids` in einen zweidimensionalen Tensor zu konvertieren, der aus sogenannten 1-aus-n- bzw. One-Hot-codierten Vektoren besteht. One-Hot-codierte Vektoren werden im Machine Learning häufig verwendet, um kategoriale Daten zu codieren, die entweder ordinal- oder nominalskaliert vorliegen. Nehmen wir zum Beispiel an, wir möchten die Namen der Spielfilmfiguren aus der Fernsehserie *Transformers* codieren. Ein Ansatz wäre, jeden Namen einer eindeutigen ID zuzuordnen, und zwar wie folgt:

```
categorical_df = pd.DataFrame(
    {"Name": ["Bumblebee", "Optimus Prime", "Megatron"], "Label ID": [0,1,2]})
categorical_df
```

	Name	Label ID
0	Bumblebee	0
1	Optimus Prime	1
2	Megatron	2

Allerdings ergibt sich bei diesem Ansatz eine fiktive Reihenfolge zwischen den Namen. Da neuronale Netze *äußerst* gut darin sind, diese Art von Beziehungen zu lernen, erweist sich dies als problematisch. Stattdessen können wir für jede Kategorie eine neue Spalte erstellen und eine 1 zuweisen, wenn die Kategorie zutrifft, und ansonsten eine 0. In Pandas lässt sich dies wie folgt mithilfe der Funktion `get_dummies()` implementieren:

```
pd.get_dummies(categorical_df["Name"])
```

	Bumblebee	Megatron	Optimus Prime
0	1	0	0
1	0	0	1
2	0	1	0

Die Zeilen dieses `DataFrame` entsprechen den One-Hot-codierten Vektoren, die jeweils einen einzigen »heißen« (»hot«), also entscheidenden Eintrag, eine 1, und sonst überall nur Nullen aufweisen. Wenn wir nun unsere `input_ids` betrachten, stehen wir vor einem ähnlichen Problem: Die Elemente sind ordinal skaliert. Das bedeutet, dass das Addieren oder Subtrahieren von zwei IDs keine sinnvolle Operation darstellt, da das Ergebnis einer neuen ID entspricht, die ein bestimmtes anderes zufälliges Token darstellt.

Andererseits lässt sich das Ergebnis bei der Addition zweier One-Hot-Codierungen leicht interpretieren: Die beiden Einträge, die »heiß« sind, zeigen an, dass die entsprechenden Tokens gemeinsam vorkommen. In PyTorch können wir die One-Hot-Codierungen erstellen, indem wir `input_ids` in einen Tensor konvertieren und anschließend die Funktion `one_hot()` anwenden:

```
import torch
import torch.nn.functional as F

input_ids = torch.tensor(input_ids)
one_hot_encodings = F.one_hot(input_ids, num_classes=len(token2idx))
one_hot_encodings.shape
```

```
torch.Size([38, 20])
```

Für jedes der 38 Eingabe-Tokens (engl. Input Tokens) haben wir nun einen One-Hot-codierten Vektor mit 20 Dimensionen (bzw. einer Länge von 20), da unser Vokabular aus 20 verschiedenen Zeichen besteht.

Die Angabe von `num_classes` in der Funktion `one_hot()` ist wichtig, da die One-Hot-codierten Vektoren sonst kürzer als die Länge des Vokabulars sein können (und manuell mit Nullen aufgefüllt werden müssten). In TensorFlow heißt die entsprechende Funktion `tf.one_hot()`, wobei dem Argument `depth` die Rolle von `num_classes` zukommt.

Begutachten wir den ersten Vektor, um zu überprüfen, ob der Wert (d. h. der Index des Tensors), der für `input_ids[0]` hinterlegt ist, der Stelle entspricht, an der die 1 steht.

```
print(f"Token: {tokenized_text[0]}")
print(f"Tensor-Index: {input_ids[0]}")
print(f"One-hot: {one_hot_encodings[0]}")
```

```
Token: T
Tensor-Index: 5
One-hot: tensor([0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Anhand unseres einfachen Beispiels können wir sehen, dass bei der Tokenisierung auf Ebene von Zeichen jegliche Struktur im Text ignoriert und der gesamte String als eine Abfolge von Zeichen behandelt wird. Obgleich dies den Umgang mit Rechtschreibfehlern und seltenen Wörtern erleichtert, besteht der größte Nachteil darin, dass linguistische Strukturen wie Wörter aus den Daten *gelernt* werden müssen. Dies erfordert allerdings einen hohen Bedarf an Rechenleistung, Speicherplatz und Daten. Aus diesem Grund kommt die Tokenisierung auf der Ebene von Zeichen in der Praxis nur selten zum Einsatz. Stattdessen wird angestrebt, eine gewisse Struktur des Texts im Rahmen der Tokenisierung beizubehalten. Dies lässt sich mit der *Tokenisierung auf der Ebene von Wörtern* (engl. Word Tokenization) umsetzen. Sehen wir uns also an, wie sie funktioniert.

Tokenisierung auf der Ebene von Wörtern (Word Tokenization)

Anstatt den Text in Zeichen zu zerlegen, können wir ihn in Wörter zerlegen und jedes Wort einer Ganzzahl zuordnen. Wenn wir von Beginn an Wörter verwenden, kann das Modell den Schritt, Wörter aus Zeichen zu lernen, überspringen, wodurch die Komplexität des Trainingsvorgangs verringert wird.

Eine relativ simpel gehaltene Klasse von Tokenizern, die auf der Ebene von Wörtern operiert, tokenisiert Texte anhand von Leerzeichen. In Python können wir hierzu die Funktion `split()` direkt auf den Rohtext anwenden (so wie wir es bei der Bestimmung der Länge der Tweets gemacht haben):

```
tokenized_text = text.split()
print(tokenized_text)
```

```
['Tokenizing', 'text', 'is', 'a', 'core', 'task', 'of', 'NLP.']
```

Ab diesem Punkt können wir die gleichen Schritte vornehmen, die wir für den Tokenizer auf der Ebene von Zeichen vorgenommen haben, um jedes Wort einer eindeutigen ID zuzuordnen. Allerdings zeichnet sich bereits ein potenzielles Problem mit diesem Tokenisierungsschema ab: Interpunktionszeichen werden nicht berücksichtigt, wodurch beispielsweise `NLP.` als ein einzelnes Token behandelt wird. In Anbetracht der Tatsache, dass Wörter dekliniert oder konjugiert sein oder auch Rechtschreibfehler enthalten können, kann die Größe des Vokabulars schnell in den Bereich von Millionen anwachsen!

Einige Tokenizers, die auf der Ebene von Wörtern operieren, haben zusätzliche Regeln für die Interpunktion. Sie können auch Wortstämme bilden (engl. Stemming) oder Lemmatisierung (engl. Lemmatization) anwenden, wodurch Wörter auf ihren Wortstamm reduziert bzw. normalisiert werden (z. B. werden »great«, »greater« und »greatest« allesamt zu »great«), wodurch jedoch einige Informationen im Text verloren gehen.

Ein sehr umfangreiches Vokabular ist problematisch, da es erfordern würde, dass das neuronale Netz eine enorme Anzahl von Parametern umfassen muss. Zur besseren Veranschaulichung: Nehmen wir an, wir haben eine Million verschiedener Wörter und möchten die 1-Million-dimensionalen Eingabevektoren in der ersten Schicht (engl. Layer) unseres neuronalen Netzes auf 1000-dimensionale Vektoren komprimieren. Bei den meisten NLP-Architekturen ist dies ein Standardschritt, und die resultierende Gewichtungsmatrix dieser ersten Schicht würde 1 Million x 1 Tausend, also 1 Milliarde, Gewichte enthalten. Das ist bereits vergleichbar mit dem größten GPT-2-Modell[4], das insgesamt etwa 1,5 Milliarden Parameter aufweist.

4 GPT-2 ist der Nachfolger von GPT und erregte die Aufmerksamkeit der Öffentlichkeit durch seine beeindruckende Fähigkeit, realistische Texte zu generieren. Wir werden uns in Kapitel 6 ausführlich mit GPT-2 beschäftigen.

Selbstverständlich möchten wir vermeiden, derart verschwenderisch mit unseren Modellparametern umzugehen, denn Modelle zu trainieren, ist kostspielig. Zudem sind größere Modelle schwieriger zu warten. Ein gängiger Ansatz besteht darin, das Vokabular einzugrenzen und seltene Wörter zu verwerfen, indem z.B. nur die 100.000 häufigsten Wörter aus dem Korpus berücksichtigt werden. Wörter, die nicht Teil des Vokabulars sind, werden als »unbekannt« eingestuft und einem einheitlichen Token »UNK« zugeordnet. Das bedeutet, dass wir im Zuge der Tokenisierung auf der Ebene von Wörtern einige potenziell wichtige Informationen verlieren, da das Modell keine Informationen über Wörter vorliegen hat, die dem Token »UNK« zugeordnet sind.

Wäre es nicht schön, wenn es einen Kompromiss zwischen der Tokenisierung auf der Ebene von Zeichen und Wörtern gäbe, der alle Informationen aus den Eingabedaten *und* einen Teil der Struktur der Eingabedaten bewahrt? Diesen gibt es in der Tat mit der *Tokenisierung auf der Ebene von Teilwörtern* (engl. Subword Tokenization).

Tokenisierung auf der Ebene von Teilwörtern (Subword Tokenization)

Die Grundidee der Tokenisierung auf der Ebene von Teilwörtern besteht darin, die besten Aspekte der Tokenisierung auf der Ebene von Zeichen und der Ebene von Wörtern zu kombinieren. Einerseits möchten wir seltene Wörter in kleinere Einheiten zerlegen, damit das Modell mit komplexen Wörtern und Rechtschreibfehlern umgehen kann. Andererseits möchten wir häufige Wörter als unverwechselbare bzw. eindeutige Einheiten beibehalten, damit wir die Länge unserer Eingaben in einem überschaubaren Rahmen halten können. Das Hauptunterscheidungsmerkmal der Tokenisierung auf der Ebene von Teilwörtern (wie auch der auf Wortebene) ist, dass sie mittels einer Mischung aus statistischen Regeln und Algorithmen auf der Grundlage des im Rahmen des Pretrainings verwendeten Korpus *gelernt* wird.

Es gibt mehrere Algorithmen, die zur Tokenisierung auf der Ebene von Teilwörtern verwendet werden. Beginnen wir mit WordPiece[5], der von den für BERT und DistilBERT genutzten Tokenizern verwendet wird. Am besten können Sie die Funktionsweise des WordPiece-Algorithmus nachvollziehen, wenn Sie ihn in Aktion sehen. Die 🤗 Transformers-Bibliothek bietet eine nützliche Klasse namens `AutoTokenizer`, mit der Sie schnell den mit einem bestimmten vortrainierten Modell verknüpften Tokenizer laden können. Sie müssen lediglich die Methode `from_pretrained()` aufrufen und die Kennung des sich auf dem Hub befindlichen Modells oder einen lokalen Dateipfad angeben. Beginnen wir zunächst damit, den Tokenizer, der für das DistilBERT-Modell verwendet wird, zu laden:

5 M. Schuster und K. Nakajima, »Japanese and Korean Voice Search,« *2012 IEEE International Conference on Acoustics, Speech and Signal Processing* (2012): 5149-5152, *https://doi.org/10.1109/ICASSP.2012.6289079*.

```
from transformers import AutoTokenizer

model_ckpt = "distilbert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
```

Die `AutoTokenizer`-Klasse gehört zu einer größeren Gruppe »automatischer« Klassen (*https://oreil.ly/h4YPz*), deren Aufgabe es ist, die Konfiguration des Modells, die vortrainierte Gewichtung oder das Vokabular automatisch auf Basis des Namens des Checkpoints abzurufen. Dies erlaubt es Ihnen, schnell zwischen den Modellen zu wechseln. Wenn Sie jedoch die spezifische Klasse manuell laden möchten, können Sie dies ebenfalls tun. Zum Beispiel hätten wir den DistilBERT-Tokenizer auch wie folgt laden können:

```
from transformers import DistilBertTokenizer

distilbert_tokenizer = DistilBertTokenizer.from_pretrained(model_ckpt)
```

Wenn Sie die Methode `AutoTokenizer.from_pretrained()` zum ersten Mal ausführen, sehen Sie einen Fortschrittsbalken, der anzeigt, welche Parameter des vortrainierten Tokenizers vom Hugging Face Hub geladen werden. Wenn Sie den Code ein zweites Mal ausführen, wird der Tokenizer aus dem Cache geladen, der sich in der Regel im Verzeichnis *~/.cache/huggingface* befindet.

Sehen wir uns an, wie dieser Tokenizer funktioniert, indem wir ihm einen einfachen Beispieltext, »Tokenizing text is a core task of NLP.«, übergeben:

```
encoded_text = tokenizer(text)
print(encoded_text)
```

```
{'input_ids': [101, 19204, 6026, 3793, 2003, 1037, 4563, 4708, 1997, 17953,
2361, 1012, 102], 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

Wie auch bei der Tokenisierung auf der Ebene von Zeichen können wir erkennen, dass die Wörter eindeutigen Ganzzahlen im Feld `input_ids` zugeordnet wurden. Auf die Rolle des Felds `attention_mask` werden wir im nächsten Abschnitt eingehen. Nachdem wir nun die `input_ids` vorliegen haben, können wir sie wieder in Tokens konvertieren, indem wir auf die `convert_ids_to_tokens()`-Methode des Tokenizers zurückgreifen:

```
tokens = tokenizer.convert_ids_to_tokens(encoded_text.input_ids)
print(tokens)
```

```
['[CLS]', 'token', '##izing', 'text', 'is', 'a', 'core', 'task', 'of', 'nl',
'##p', '.', '[SEP]']
```

Wir können hier drei Dinge beobachten. Erstens wurde jeweils ein spezielles `[CLS]`- und `[SEP]`-Token am Anfang und am Ende der Sequenz hinzugefügt. Diese speziellen Tokens (engl. Special Tokens) sind von Modell zu Modell unterschiedlich, aber ihre Hauptaufgabe ist es, den Anfang und das Ende einer Sequenz zu kennzeichnen. Zweitens wurden die Tokens in Kleinbuchstaben umgewandelt, was eine Be-

sonderheit dieses Checkpoints darstellt. Drittens können wir feststellen, dass »tokenizing« und »NLP« in zwei Tokens aufgeteilt wurden, was sinnvoll erscheint, da sie nicht zusammengehören. Das Präfix `##` in `##izing` und `##p` bedeutet, dass das vorangehende Zeichen kein Leerzeichen ist. Jedes Token, das dieses Präfix aufweist, sollte also wieder mit dem vorherigen Token zusammengeführt werden, wenn Sie die Tokens zurück in eine Zeichenkette konvertieren. Die `AutoTokenizer`-Klasse verfügt über eine Methode namens `convert_tokens_to_string()`, die sich hierfür anbietet. Wenden wir sie auf unsere Tokens an:

```
print(tokenizer.convert_tokens_to_string(tokens))
```

```
[CLS] tokenizing text is a core task of nlp. [SEP]
```

Die `AutoTokenizer`-Klasse hat auch mehrere Attribute, die uns Informationen über den Tokenizer liefern. Zum Beispiel können wir die Größe des Vokabulars abfragen:

```
tokenizer.vocab_size
```

```
30522
```

und auch die maximale Kontextlänge (engl. Maximum Context Size) des entsprechenden Modells:

```
tokenizer.model_max_length
```

```
512
```

Ein weiteres interessantes Attribut, das Sie kennen sollten, sind die Namen der Felder, die das Modell in seinem Forward-Pass erwartet:

```
tokenizer.model_input_names
```

```
['input_ids', 'attention_mask']
```

Wir haben nun ein grundlegendes Verständnis darüber, wie ein einzelner String in Tokens umgewandelt wird. Sehen wir uns nun an, wie wir den gesamten Datensatz tokenisieren können.

Wenn Sie vortrainierte Modelle verwenden, ist es *äußerst* wichtig, darauf zu achten, dass Sie denselben Tokenizer verwenden, der auch beim Training verwendet wurde. Aus der Sicht des Modells ist ein Wechsel des Tokenizers wie eine Umstellung des Vokabulars. Wenn alle um Sie herum anfangen würden, zufällige Wörter wie »Haus« statt »Katze« zu verwenden, würden es Ihnen auch schwer fallen zu verstehen, was vor sich geht!

Den gesamten Datensatz tokenisieren

Um das gesamte Korpus zu tokenisieren, nutzen wir die Methode `map()` unseres `DatasetDict`-Objekts. Diese Methode wird uns im Laufe dieses Buchs noch häufiger begegnen, da sie eine bequeme Möglichkeit bietet, eine bestimmte Verarbei-

tungsfunktion (engl. Processing Function) auf alle Elemente eines Datensatzes gleichzeitig anzuwenden. Wie Sie gleich noch erfahren werden, kann die `map()`-Methode auch dazu verwendet werden, neue Zeilen und Spalten zu erstellen.

Zunächst benötigen wir eine Verarbeitungsfunktion, mit der wir unsere Beispiele tokenisieren können:

```
def tokenize(batch):
    return tokenizer(batch["text"], padding=True, truncation=True)
```

Mit dieser Funktion wird der Tokenizer auf ein Batch von Beispielen angewandt. Durch `padding=True` werden die Beispiele entsprechend der Länge des längsten Beispiels, das in einem Batch enthalten ist, mit Nullen aufgefüllt (engl. padded). Indem das Argument `truncation=True` gesetzt wird, werden die Beispiele auf die maximale Kontextlänge des Modells gekürzt bzw. gestutzt (d.h., es wird eine Trunkierung, engl. Truncation, vorgenommen). Übergeben wir nun ein Batch, das aus zwei Beispielen des Trainingsdatensatzes besteht, um zu sehen, wie die `tokenize()`-Funktion funktioniert:

```
print(tokenize(emotions["train"][:2]))
```

```
{'input_ids': [[101, 1045, 2134, 2102, 2514, 26608, 102, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0], [101, 1045, 2064, 2175, 2013, 3110, 2061, 20625, 2000,
2061, 9636, 17772, 2074, 2013, 2108, 2105, 2619, 2040, 14977, 1998, 2003, 8300,
102]], 'attention_mask': [[1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1]]}
```

Wie wir erkennen können, hat sich das Padding auf unser Ergebnis ausgewirkt: Da das erste Element von `input_ids` kürzer als das zweite ist, wurden Nullen zu diesem Element hinzugefügt, sodass nun beide die gleiche Länge aufweisen. Diese Nullen entsprechen im Vokabular dem `[PAD]`-Token. Die hier verwendeten speziellen Tokens umfassen auch die Tokens `[CLS]` und `[SEP]`, die wir bereits kennengelernt haben:

spezielles Token	[PAD]	[UNK]	[CLS]	[SEP]	[MASK]
ID des speziellen Tokens	0	100	101	102	103

Beachten Sie auch, dass der Tokenizer nicht nur die codierten Tweets als `input_ids` zurückgibt, sondern auch eine Liste mit `attention_mask`-Arrays. Der Grund dafür ist, dass wir nicht möchten, dass das Modell durch die zusätzlichen Padding-Tokens irregeführt wird: Die Attention-Mask erlaubt es dem Modell, die Teile der Eingabe zu ignorieren, die aufgefüllt wurden. In Abbildung 2-3 können Sie auf anschauliche Weise nachvollziehen, wie die `input_ids`- und die `attention_mask`-Arrays aufgefüllt werden.

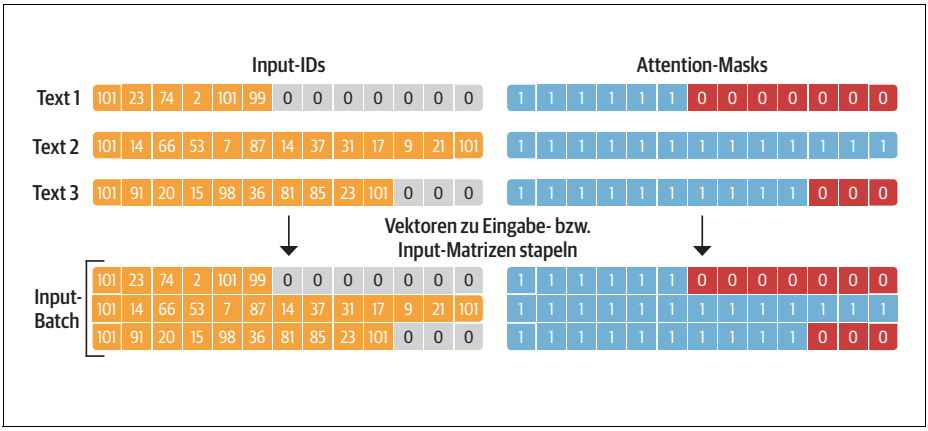


Abbildung 2-3: Für jedes Batch werden die Eingabesequenzen auf die Länge der im Batch enthaltenen längsten Sequenz aufgefüllt. Die Attention-Mask wird im Modell verwendet, damit die aufgefüllten Bereiche der Eingabe-Tensoren nicht berücksichtigt werden.

Da wir die Verarbeitungsfunktion bereits zur Hand haben, können wir sie nun mit nur einer einzigen Codezeile auf alle Teildatensätze (Splits) des Korpus anwenden:

```
emotions_encoded = emotions.map(tokenize, batched=True, batch_size=None)
```

Die Methode `map()` bearbeitet standardmäßig jedes Beispiel im Korpus einzeln. Wenn Sie also `batched=True` setzen, werden die Tweets als Batches codiert. Da wir `batch_size=None` gesetzt haben, wird unsere Funktion `tokenize()` auf den gesamten Datensatz, der als ein einzelnes Batch behandelt wird, angewandt. Dadurch wird sichergestellt, dass die Eingabe- bzw. Input-Tensoren und die Attention-Masks alle einheitlich dimensioniert (engl. Shape) sind. Wie wir sehen können, wurden dem Datensatz durch diese Operation zwei neue Spalten, `input_ids` und `attention_mask`, hinzugefügt:

```
print(emotions_encoded["train"].column_names)
```

```
['attention_mask', 'input_ids', 'label', 'text']
```

In späteren Kapiteln werden wir sehen, wie Data-Collator verwendet werden können, um die Tensoren in jedem Batch dynamisch aufzufüllen. Das Padding einheitlich durchzuführen, wird sich im nächsten Abschnitt als nützlich erweisen, wenn wir eine Feature-Matrix auf Basis des gesamten Korpus ermitteln.

Trainieren eines Textklassifikators

Wie in Kapitel 1 erläutert, wurden Modelle wie DistilBERT darauf vortrainiert, maskierte Wörter in einer Textsequenz vorherzusagen. Allerdings können wir diese Sprachmodelle nicht direkt für die Textklassifizierung verwenden, sondern müssen sie leicht modifizieren. Um zu verstehen, welche Modifikationen vonnöten sind, lohnt es sich, einen Blick auf die Architektur eines Encoder-basierten Modells wie DistilBERT, das in Abbildung 2-4 dargestellt ist, zu werfen.

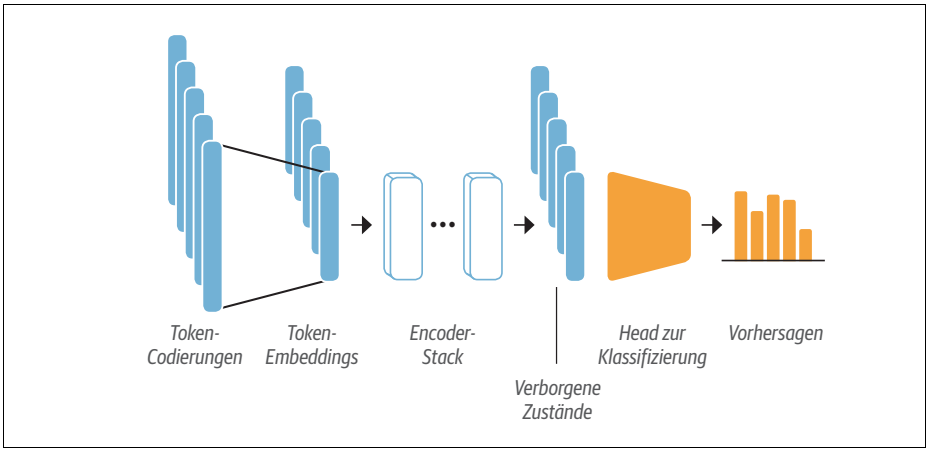


Abbildung 2-4: Die Architektur, die zur Klassifizierung von Sequenzen auf Basis eines Encoder-basierten Transformer-Modells verwendet wird. Sie besteht aus einem für das Pretraining des Modells verwendeten Body in Kombination mit einem benutzerdefinierten bzw. individuell gestalteten Head zur Klassifizierung.

Zunächst wird der Text in Tokens umgewandelt (tokenisiert) und als One-Hot-codierte Vektoren dargestellt, die *Token-Codierungen* (engl. Token Encodings) genannt werden. Die Größe des Tokenizer-Vokabulars bestimmt die Dimension der Token-Codierungen, die normalerweise aus 20.000 bis 200.000 verschiedenen Tokens besteht. Im nächsten Schritt werden diese Token-Codierungen in *Token-Embeddings* bzw. -Einbettungen überführt, d.h. in Vektoren, die in einem niedriger dimensionalen Raum liegen. Die Token-Embeddings werden dann durch die Schichten des Encoder-Blocks geleitet, um einen *verborgenen Zustand* (engl. Hidden State) für jedes Eingabe- bzw. Input-Token zu erhalten. Im Rahmen des Pretrainings wird eine Sprachmodellierung durchgeführt,[6] wobei jeder verborgene Zustand an eine Schicht weitergeleitet wird, die die maskierten Eingabe-Tokens vorhersagt. Für die Klassifizierungsaufgabe ersetzen wir die Sprachmodellierungsschicht durch eine Klassifizierungsschicht (engl. Classification Layer).

In Wirklichkeit wird bei PyTorch der Schritt der Erstellung von One-Hot-codierten Vektoren als Token-Codierungen übersprungen, da die Multiplikation einer Matrix mit einem One-Hot-codierten Vektor dasselbe ist, wie eine Spalte aus dieser Matrix auszuwählen. Dementsprechend wird nur die Spalte aus der Matrix benötigt, die der ID des Tokens entspricht. Diesem Ansatz werden wir noch in Kapitel 3 begegnen, wenn wir die Klasse `nn.Embedding` verwenden.

Wir haben zwei Möglichkeiten, ein solches Modell auf unseren Twitter-Datensatz zu trainieren:

6 Im Fall von DistilBERT geht es darum, die maskierten Tokens zu erraten.

Feature Extraction (Merkmalsextraktion)
: Wir verwenden die verborgenen Zustände als Features (auch Merkmale genannt) und trainieren einfach einen Klassifikator mit ihnen, ohne das vortrainierte Modell zu verändern.

Feintuning (engl. Fine-Tuning)
: Wir trainieren das gesamte Modell vollständig (»End-to-End«), wodurch auch die Parameter des vortrainierten Modells aktualisiert werden.

In den folgenden Abschnitten werden wir uns beide Vorgehensweisen in Bezug auf DistilBERT ansehen und die jeweils damit verbundenen Kompromisse beleuchten.

Transformer-Modelle als Feature-Extraktoren

Einen Transformer dazu zu verwenden, Merkmale bzw. Features zu extrahieren, ist ziemlich einfach. Wie in Abbildung 2-5 gezeigt, frieren (engl. freeze) wir die Gewichtung des Bodys während des Trainings ein und verwenden die verborgenen Zustände als Features für den Klassifikator. Der Vorteil dieses Ansatzes besteht darin, dass wir recht schnell ein kleines bzw. wenig komplexes Modell trainieren können. Ein derartiges Modell könnte eine neuronale Klassifizierungsschicht oder ein Verfahren sein, das nicht auf Gradienten basiert, wie z. B. ein Random Forest. Der Ansatz ist besonders praktisch, wenn Ihnen keine GPUs zur Verfügung stehen, da die verborgenen Zustände nur einmal vorberechnet werden müssen.

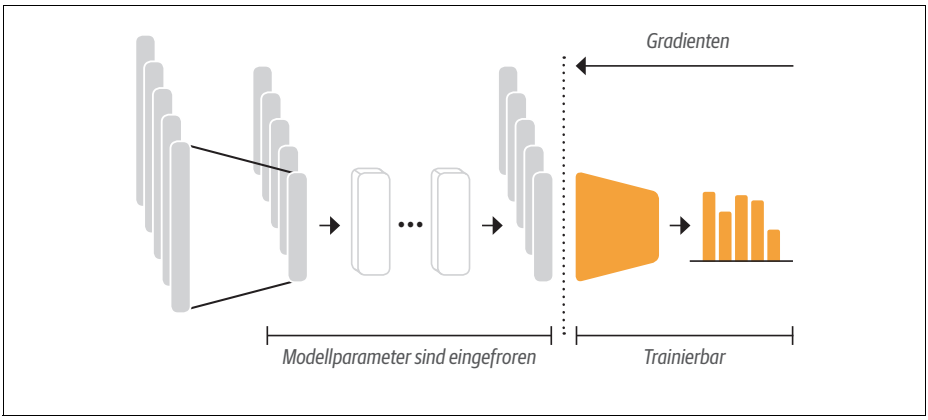


Abbildung 2-5: Beim Feature-basierten Ansatz wird das DistilBERT-Modell »eingefroren« und liefert lediglich Features für einen Klassifikator.

Vortrainierte Modelle verwenden

Wir werden eine weitere praktische »Auto«-Klasse der 🤗 Transformers-Bibliothek namens `AutoModel` verwenden. Ähnlich wie die `AutoTokenizer`-Klasse verfügt die `AutoModel`-Klasse über eine `from_pretrained()`-Methode, mit der die Gewichtung eines vortrainierten Modells geladen werden kann. Verwenden wir nun die Methode, um den Checkpoint für das DistilBERT-Modell zu laden:

```
from transformers import AutoModel

model_ckpt = "distilbert-base-uncased"
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = AutoModel.from_pretrained(model_ckpt).to(device)
```

In vorliegenden Fall haben wir PyTorch verwendet, um zu prüfen, ob eine GPU verfügbar ist oder nicht. Anschließend haben wir die PyTorch-Methode `nn.Module.to()` mit der Methode zum Laden des vortrainierten Modells (`from_pretrained()`) verkettet. Dadurch stellen wir sicher, dass das Modell auf der GPU läuft, sofern eine vorhanden ist. Wenn nicht, wird das Modell auf der CPU ausgeführt, was allerdings bedeutend mehr Zeit in Anspruch nehmen kann.

Die `AutoModel`-Klasse wandelt die Token-Codierungen in Embeddings um und lässt diese anschließend durch den Encoder-Block laufen, damit die verborgenen Zustände zurückgegeben werden. Sehen wir uns nun an, wie wir diese Zustände aus unserem Korpus ermitteln bzw. extrahieren können.

Interoperabilität zwischen Frameworks

Obwohl der Code in diesem Buch größtenteils in PyTorch geschrieben ist, bietet die 🤗 Transformers-Bibliothek eine gute Interoperabilität mit TensorFlow und JAX. Das bedeutet, dass Sie nur ein paar Zeilen Code anpassen müssen, um ein vortrainiertes Modell in Ihrem bevorzugten Deep-Learning-Framework zu laden. Zum Beispiel können wir das DistilBERT-Modell in TensorFlow laden, indem wir die `TFAutoModel`-Klasse wie folgt verwenden:

```
from transformers import TFAutoModel

tf_model = TFAutoModel.from_pretrained(model_ckpt)
```

Diese Interoperabilität ist besonders nützlich, wenn ein Modell nur in einem Framework veröffentlicht wurde, Sie es aber gerne in einem anderen verwenden möchten. Für das XLM-RoBERTa-Modell (*https://oreil.ly/OUMvG*), auf das wir in Kapitel 4 zurückgreifen werden, steht die Gewichtung beispielsweise nur für PyTorch zur Verfügung. Wenn Sie es wie eben gezeigt in TensorFlow laden wollen:

```
tf_xlmr = TFAutoModel.from_pretrained("xlm-roberta-base")
```

dann erhalten Sie einen Fehler. In diesen Fällen können Sie der Funktion `TfAutoModel.from_pretrained()` als Argument `from_pt=True` vorgeben, sodass die Bibliothek die Gewichtung für PyTorch automatisch für Sie herunterlädt und konvertiert:

```
tf_xlmr = TFAutoModel.from_pretrained("xlm-roberta-base", from_pt=True)
```

Wie Sie sehen können, ist es ein Leichtes, mit der 🤗 Transformers-Bibliothek zwischen Frameworks zu wechseln. In den meisten Fällen können Sie den Namen der Klassen einfach ein »TF"-Präfix voranstellen, um die entsprechenden Klassen für TensorFlow 2.0 zu erhalten. Immer wenn wir den String »pt« verwenden (z.B. im folgenden Abschnitt), der für PyTorch steht, können Sie ihn einfach durch »tf« ersetzen, was eine Abkürzung für TensorFlow darstellt.

Die letzten verborgenen Zustände extrahieren

Rufen wir zum Einstieg die letzten verborgenen Zustände für einen einzelnen String ab. Als Erstes müssen wir den String codieren und die Tokens in PyTorch-Tensoren umwandeln. Hierzu können Sie wie folgt dem Tokenizer das Argument `return_tensors="pt"` übergeben:

```
text = "this is a test"
inputs = tokenizer(text, return_tensors="pt")
print(f"Shape des Eingabe-Tensors: {inputs['input_ids'].size()}")
```

```
Shape des Eingabe-Tensors: torch.Size([1, 6])
```

Wie wir sehen können, hat der resultierende Tensor eine Dimensionierung, im Folgenden als Shape bezeichnet, von `[batch_size, n_tokens]`. Nachdem wir nun die Codierungen als einen Tensor vorliegen haben, müssen wir sie wie folgt auf demselben Device (d.h. CPU oder GPU) wie das Modell platzieren und die Eingabedaten übergeben:

```
inputs = {k:v.to(device) for k,v in inputs.items()}
with torch.no_grad():
    outputs = model(**inputs)
print(outputs)
```

```
BaseModelOutput(last_hidden_state=tensor([[[-0.1565, -0.1862,  0.0528,  ...,
-0.1188,  0.0662,  0.5470],
         [-0.3575, -0.6484, -0.0618,  ..., -0.3040,  0.3508,  0.5221],
         [-0.2772, -0.4459,  0.1818,  ..., -0.0948, -0.0076,  0.9958],
         [-0.2841, -0.3917,  0.3753,  ..., -0.2151, -0.1173,  1.0526],
         [ 0.2661, -0.5094, -0.3180,  ..., -0.4203,  0.0144, -0.2149],
         [ 0.9441,  0.0112, -0.4714,  ...,  0.1439, -0.7288, -0.1619]]],
       device='cuda:0'), hidden_states=None, attentions=None)
```

Hier haben wir den Kontextmanager `torch.no_grad()` verwendet, um eine automatische Berechnung des Gradienten zu unterbinden. Dies ist im Rahmen der Inferenz nützlich, da es den Speicherbedarf der Berechnungen verringert. Je nach Modellkonfiguration kann die Ausgabe mehrere Objekte enthalten, wie z.B. die verborgenen Zustände, die Verluste oder die Attention-Arrays, die in einer Klasse zusammengefasst sind, die einem `namedtuple` in Python ähnelt. In unserem Beispiel entspricht die Modellausgabe einer Instanz von `BaseModelOutput`, wobei wir einfach über den Namen auf ihre Attribute zugreifen können. Das vorliegende Modell liefert nur ein Attribut, nämlich den letzten verborgenen Zustand. Ermitteln wir zunächst, wie er dimensioniert ist:

```
outputs.last_hidden_state.size()
```

```
torch.Size([1, 6, 768])
```

Wenn wir den Tensor mit den verborgenen Zuständen (engl. Hidden State Tensor) begutachten, können wir feststellen, dass er ein Shape von `[batch_size, n_tokens, hidden_dim]` hat. Mit anderen Worten, es wird jeweils ein 768-dimensionaler Vektor für jedes der 6 eingegebenen Tokens zurückgegeben. Bei Klassifizierungsaufgaben ist es üblich, nur den verborgenen Zustand, der mit dem `[CLS]`-Token assozi-

iert ist, als Eingabe- bzw. Input-Feature zu verwenden. Da dieses Token in jeder Sequenz am Anfang erscheint, können wir es extrahieren, indem wir einfach in `outputs.last_hidden_state` den Index wie folgt angeben:

```
outputs.last_hidden_state[:,0].size()
```

```
torch.Size([1, 768])
```

Wir wissen jetzt, wie wir den letzten verborgenen Zustand für einen einzelnen String ermitteln können. Das Gleiche können wir nun für den gesamten Datensatz vornehmen, indem wir eine neue Spalte, `hidden_state`, erstellen, in der alle diese Vektoren gespeichert werden. Wie zuvor beim Tokenizer verwenden wir die `map()`-Methode des `DatasetDict`-Objekts, um alle verborgenen Zustände auf einmal zu extrahieren. Als Erstes müssen wir die vorherigen Schritte in einer Verarbeitungsfunktion zusammenfassen:

```
def extract_hidden_states(batch):
    # Modelleingaben auf der GPU platzieren
    inputs = {k:v.to(device) for k,v in batch.items()
              if k in tokenizer.model_input_names}
    # Letzte verborgene Zustände extrahieren
    with torch.no_grad():
        last_hidden_state = model(**inputs).last_hidden_state
    # Vektor für [CLS]-Token zurückgeben
    return {"hidden_state": last_hidden_state[:,0].cpu().numpy()}
```

Der einzige Unterschied zwischen dieser Funktion und der Vorgehensweise, die wir zuvor verfolgt haben, ist der letzte Schritt, bei dem wir den endgültigen verborgenen Zustand (bzw. Zustandsvektor) als NumPy-Array zurück auf die CPU platzieren. Die `map()`-Methode erfordert, dass die Verarbeitungsfunktion Python- bzw. NumPy-Objekte zurückgibt, wenn wir Eingaben verwenden, die als Batches vorliegen.

Da unser Modell Tensoren als Eingaben erwartet, müssen wir als Nächstes die Spalten `input_ids` und `attention_mask` in das `"torch"`-Format konvertieren:

```
emotions_encoded.set_format("torch",
                            columns=["input_ids", "attention_mask", "label"])
```

Anschließend können wir die verborgenen Zustände (bzw. Zustandsvektoren) für alle Teildatensätze auf einen Schlag extrahieren:

```
emotions_hidden = emotions_encoded.map(extract_hidden_states, batched=True)
```

Beachten Sie, dass wir in diesem Fall nicht `batch_size=None` gesetzt haben. Das bedeutet, dass stattdessen der vorgegebene Default-Wert `batch_size=1000` verwendet wird. Infolge der Anwendung der `extract_hidden_states()`-Funktion wurde unserem Datensatz erwartungsgemäß eine neue Spalte namens `hidden_state` hinzugefügt:

```
emotions_hidden["train"].column_names
```

```
['attention_mask', 'hidden_state', 'input_ids', 'label', 'text']
```

Nachdem wir nun die verborgenen Zustände der einzelnen Tweets ermittelt haben, besteht der nächste Schritt darin, einen Klassifikator auf Basis dieser Zustände zu trainieren. Zu diesem Zweck benötigen wir eine Feature-Matrix, auf die wir nun zu sprechen kommen.

Eine Feature-Matrix erstellen

Der vorverarbeitete Datensatz enthält nun alle Informationen, die wir benötigen, um einen Klassifikator trainieren zu können. Die verborgenen Zustände werden wir als Eingabe- bzw. Input-Features und die Labels als Zielvariable (engl. Target) verwenden. Die entsprechenden Arrays können wir wie folgt problemlos im vertrauten Scikit-learn-Format erstellen:

```
import numpy as np

X_train = np.array(emotions_hidden["train"]["hidden_state"])
X_valid = np.array(emotions_hidden["validation"]["hidden_state"])
y_train = np.array(emotions_hidden["train"]["label"])
y_valid = np.array(emotions_hidden["validation"]["label"])
X_train.shape, X_valid.shape
```

```
((16000, 768), (2000, 768))
```

Bevor wir ein Modell auf der Grundlage der verborgenen Zustände trainieren, ist es ratsam, kurz zu prüfen, ob sie die Emotionen, die wir klassifizieren möchten, auf eine brauchbare Weise abbilden. Im nächsten Abschnitt werden Sie erfahren, wie Sie dies mithilfe einer Visualisierung der Features schnell erreichen können.

Den Trainingsdatensatz visualisieren

Da es, gelinde gesagt, schwierig ist, die 768-dimensionalen verborgenen Zustände zu visualisieren, verwenden wir den leistungsfähigen UMAP-Algorithmus, mit dem wir die Vektoren auf zwei Dimensionen projizieren bzw. abbilden können.[7] Da der UMAP-Algorithmus am besten funktioniert, wenn die Features so skaliert sind, dass sie im Intervall [0,1] liegen, wenden wir zunächst einen `MinMaxScaler` an. Anschließend greifen wir auf die UMAP-Implementierung aus der `umap-learn`-Bibliothek zurück, um die verborgenen Zustände zu verdichten:

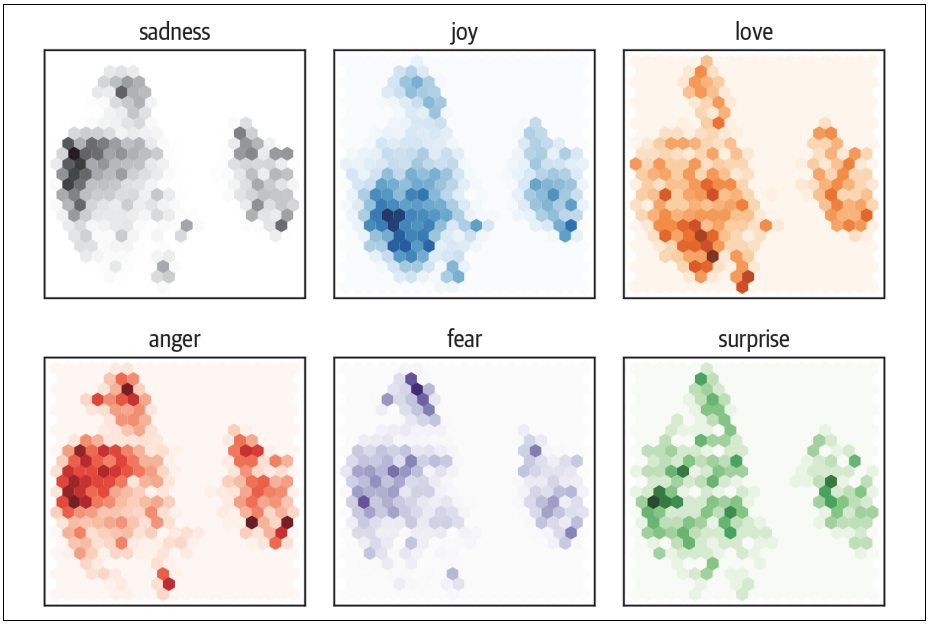
```
from umap import UMAP
from sklearn.preprocessing import MinMaxScaler

# Features so skalieren, dass sie im Intervall [0,1] liegen
X_scaled = MinMaxScaler().fit_transform(X_train)
# Initialisierung und Anpassung mit UMAP-Algorithmus
mapper = UMAP(n_components=2, metric="cosine").fit(X_scaled)
# DataFrame mit zweidimensionalem Embedding erstellen
df_emb = pd.DataFrame(mapper.embedding_, columns=["X", "Y"])
df_emb["label"] = y_train
df_emb.head()
```

7 L. McInnes, J. Healy und J. Melville, »UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction« (*https://arxiv.org/abs/1802.03426*), (2018).

	X	Y	label
0	4.358075	6.140816	0
1	-3.134567	5.329446	0
2	5.152230	2.732643	3
3	-2.519018	3.067250	2
4	-3.364520	3.356613	3

Als Ergebnis erhalten wir ein Array, das die gleiche Anzahl an Trainingsbeispielen, jedoch nur 2 Features anstelle der 768, mit denen wir begonnen haben, umfasst. Wir können die verdichteten Daten noch ein wenig genauer unter die Lupe nehmen und für jede Kategorie ein separates Diagramm erstellen, das jeweils aufzeigt, wie die Datenpunkte verteilt sind:

```
fig, axes = plt.subplots(2, 3, figsize=(7,5))
axes = axes.flatten()
cmaps = ["Greys", "Blues", "Oranges", "Reds", "Purples", "Greens"]
labels = emotions["train"].features["label"].names

for i, (label, cmap) in enumerate(zip(labels, cmaps)):
    df_emb_sub = df_emb.query(f"label == {i}")
    axes[i].hexbin(df_emb_sub["X"], df_emb_sub["Y"], cmap=cmap,
                   gridsize=20, linewidths=(0,))
    axes[i].set_title(label)
    axes[i].set_xticks([]), axes[i].set_yticks([])

plt.tight_layout()
plt.show()
```



Es handelt sich dabei lediglich um Projektionen auf einen niedrigerdimensionalen Raum. Nur weil sich einige Kategorien überschneiden, bedeutet das nicht, dass sie im ursprünglichen Raum nicht voneinander separierbar sind. Umgekehrt gilt: Wenn sie im projizierten Raum separierbar sind, sind sie es auch im ursprünglichen Raum.

Aus diesem Diagramm können wir einige klare Muster erkennen: Die negativen Gefühle wie `sadness` (Trauer), `anger` (Angst) und `fear` (Furcht) befinden sich alle in ähnlichen Bereichen mit leicht unterschiedlichen Verteilungen. Andererseits sind `joy` (Freude) und `love` (Liebe) gut von den negativen Gefühlen abgegrenzt und teilen sich ebenfalls einen ähnlichen Raum. Die Emotion `surprise` (Überraschung) ist hingegen über den ganzen Raum verstreut. Obwohl wir gehofft haben, dass sich die Kategorien in gewisser Weise voneinander abgrenzen, ist dies keineswegs garantiert, da das Modell nicht darauf trainiert wurde, den Unterschied zwischen diesen Emotionen zu kennen. Es hat sie nur implizit gelernt, indem es die in den Texten maskierten Wörter erraten hat.

Nachdem wir nun einen gewissen Einblick in die Features unseres Datensatzes gewonnen haben, können wir sie schließlich dafür verwenden, ein Modell zu trainieren.

Einen einfachen Klassifikator trainieren

Wie wir gesehen haben, unterscheiden sich die verborgenen Zustände zwischen den Emotionen ein wenig, wobei jedoch einige der Emotionen nicht klar voneinander zu trennen sind. Nehmen wir nun die verborgenen Zustände, um ein logistisches Regressionsmodell unter Verwendung von Scikit-learn zu trainieren. Ein solch einfaches Modell zu trainieren, geht schnell und erfordert keine GPU:

```
from sklearn.linear_model import LogisticRegression

# Wir erhöhen `max_iter`, um Konvergenz zu garantieren
lr_clf = LogisticRegression(max_iter=3000)
lr_clf.fit(X_train, y_train)
lr_clf.score(X_valid, y_valid)
```

```
0.633
```

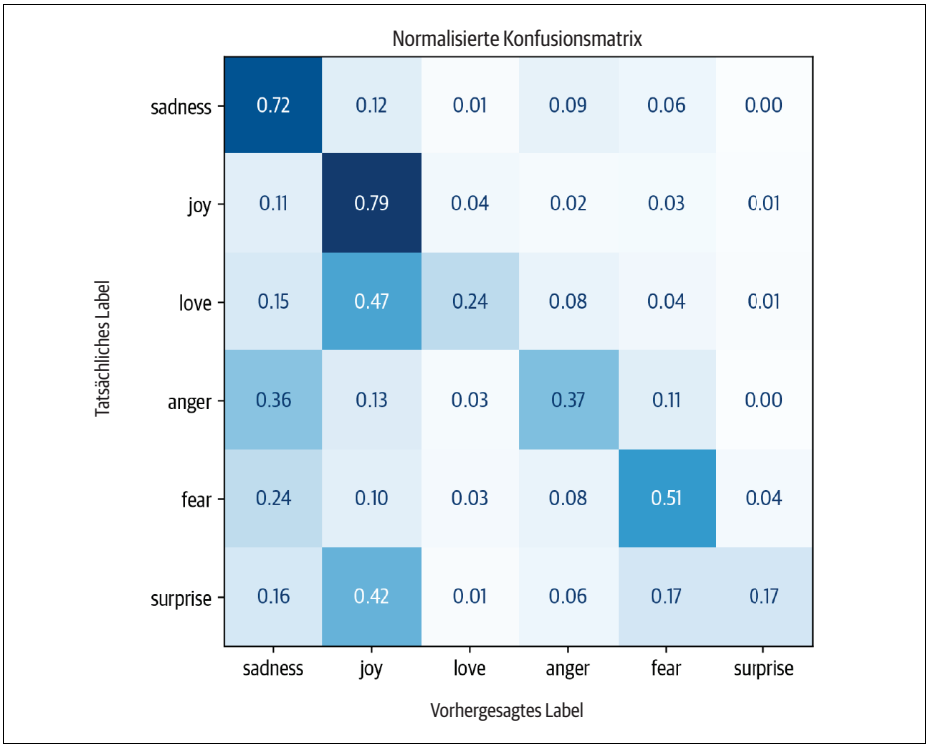
Wenn Sie sich die Treffergenauigkeit (engl. Accuracy) ansehen, könnten Sie den Eindruck gewinnen, dass unser Modell nur ein wenig besser ist als ein rein auf Zufall basierendes Modell. Allerdings haben wir einen Datensatz vorliegen, der mehr als zwei Kategorien umfasst (Multiklassendatensatz), und die Kategorien sind unausgewogen verteilt. Deshalb ist das Modell bedeutend besser, als es scheint. Um besser beurteilen zu können, wie gut unser Modell tatsächlich abschneidet, können wir es mit einem einfachen Referenzmodell (bzw. einer sogenannten Baseline) vergleichen. In Scikit-learn gibt es einen `DummyClassifier`, mit dem Sie einen Klassifikator mithilfe einfacher Heuristiken erstellen können, z. B. indem immer die Kategorie gewählt wird, die am häufigsten vorkommt (»Majority Class«), oder indem die Kategorien immer zufällig gezogen werden. Im vorliegenden Fall besteht die am besten funktionierende Heuristik darin, immer die am häufigsten vorkommende Kategorie zu wählen, was zu einer Treffergenauigkeit von etwa 35 % führt:

```
from sklearn.dummy import DummyClassifier

dummy_clf = DummyClassifier(strategy="most_frequent")
dummy_clf.fit(X_train, y_train)
dummy_clf.score(X_valid, y_valid)
```

```
0.352
```

Wie sich zeigt, ist unser einfacher Klassifikator, der auf DistilBERT-Embeddings basiert, deutlich besser als unser Baseline-Klassifikator. Wir können die Leistung des Modells noch eingehender untersuchen, indem wir uns die Konfusionsmatrix (auch Wahrheitsmatrix genannt) des Klassifikators ansehen, die uns Aufschluss darüber gibt, wie das Verhältnis zwischen den tatsächlichen (bzw. wahren) und den vorhergesagten Labels ausfällt:

```
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix

def plot_confusion_matrix(y_preds, y_true, labels):
    cm = confusion_matrix(y_true, y_preds, normalize="true")
    fig, ax = plt.subplots(figsize=(6, 6))
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=labels)
    disp.plot(cmap="Blues", values_format=".2f", ax=ax, colorbar=False)
    plt.title("Normalisierte Konfusionsmatrix")
    plt.show()

y_preds = lr_clf.predict(X_valid)
plot_confusion_matrix(y_preds, y_valid, labels)
```



Wie sich zeigt, werden `anger` (Angst) und `fear` (Furcht) am häufigsten mit `sadness` (Trauer) verwechselt, was sich mit der Einschätzung deckt, die wir bereits bei der Visualisierung der Embeddings angestellt haben. Ebenso werden `love` (Liebe) und `surprise` (Überraschung) häufig mit `joy` (Freude) verwechselt.

Im nächsten Abschnitt werden wir uns mit dem Ansatz des Feintunings beschäftigen, der zu einer höheren Genauigkeit bei der Klassifizierung führt. Allerdings sollte Ihnen bewusst sein, dass dieser Ansatz mehr Rechenressourcen wie z.B. GPUs erfordert, die in Ihrem Unternehmen möglicherweise nicht zur Verfügung stehen. In solchen Fällen kann ein Feature-basierter Ansatz einen guten Kompromiss zwischen traditionellem Machine Learning und Deep Learning darstellen.

Feintuning von Transformer-Modellen

Erkunden wir nun, was für das vollständige (»End-to-End«) Feintuning eines Transformer-Modells erforderlich ist. Beim Feintuning verwenden wir die verborgenen Zustände nicht als feste Features, sondern trainieren sie (siehe Abbildung 2-6). Dies erfordert, dass der Klassifizierungs-Head differenzierbar ist, weshalb bei diesem Ansatz in der Regel ein neuronales Netzwerk zur Klassifizierung verwendet wird.

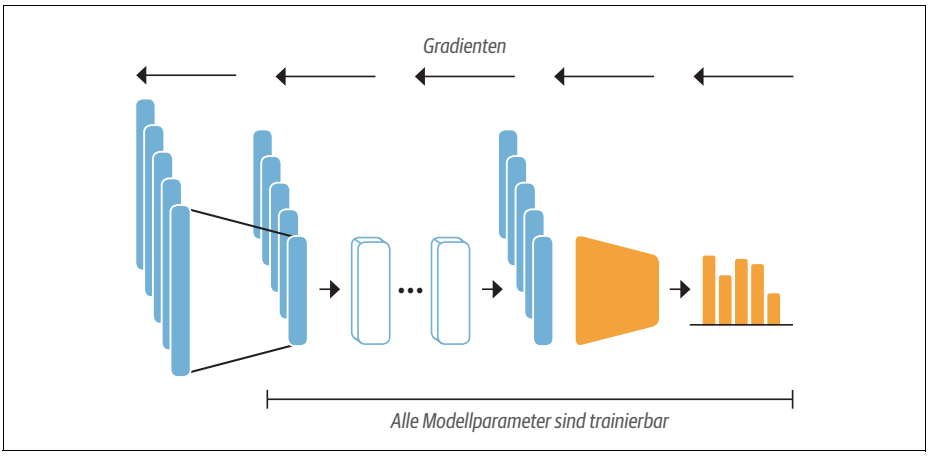


Abbildung 2-6: Im Rahmen des Feintunings wird das gesamte DistilBERT-Modell zusammen mit dem Klassifizierungs-Head trainiert.

Dadurch, dass wir die verborgenen Zustände trainieren, die als Eingaben für das Klassifizierungsmodell dienen, vermeiden wir, dass wir mit Eingaben arbeiten, die möglicherweise nicht sonderlich gut geeignet für die Klassifizierungsaufgabe sind. Stattdessen passen sich die anfänglichen verborgenen Zustände während des Trainings an, sodass sich der Verlust des Modells verringert und sich seine Leistung erhöht.

Wir werden auf die `Trainer`-Klasse aus der 🤗 Transformers-Bibliothek zurückgreifen, mit der wir die Trainingsschleife vereinfachen können. Werfen wir einen Blick

auf die einzelnen Komponenten, die wir benötigen, um eine Trainingsschleife einzurichten.

Ein vortrainiertes Modell laden

Als Erstes benötigen wir ein vortrainiertes DistilBERT-Modell, wie wir es auch beim Feature-basierten Ansatz verwendet haben. Die einzige kleine Änderung ist, dass wir das Modell `AutoModelForSequenceClassification` anstelle von `AutoModel` verwenden. Der Unterschied besteht darin, dass das Modell `AutoModelForSequenceClassification` zusätzlich zu den Ausgaben des vortrainierten Modells einen Klassifizierungs-Head hat, der sich leicht mit dem DistilBERT-Base-Modell trainieren lässt. Wir müssen nur angeben, wie viele Labels das Modell vorhersagen soll (in unserem Fall sechs), da dies bestimmt, wie viele Ausgaben bzw. Outputs der Klassifizierungs-Head hat:

```
from transformers import AutoModelForSequenceClassification

num_labels = 6
model = (AutoModelForSequenceClassification
         .from_pretrained(model_ckpt, num_labels=num_labels)
         .to(device))
```

Sie werden eine Warnmeldung erhalten, dass einige Teile des Modells zufällig initialisiert wurden. Das ist normal, da der Klassifizierungs-Head noch nicht trainiert wurde. Im nächsten Schritt legen wir die Maße bzw. Metriken fest, mit denen wir die Leistung unseres Modells während des Feintunings evaluieren werden.

Qualitätsmaße festlegen

Um die Qualitätsmaße (auch Gütemaße genannt) während des Trainings überwachen zu können, müssen wir eine Funktion, `compute_metrics()`, für den `Trainer` definieren. Diese Funktion gestalten wir so, dass sie ein `EvalPrediction`-Objekt (ein benanntes Tupel mit den Attributen `predictions` und `label_ids`) entgegennimmt und ein Dictionary, das die Namen der einzelnen Maße sowie die entsprechenden Werte enthält, zurückgibt. Im Rahmen unserer Anwendung werden wir das F_1-Maß (engl. F_1-Score) und die Treffergenauigkeit (engl. Accuracy) des Modells ermitteln:

```
from sklearn.metrics import accuracy_score, f1_score

def compute_metrics(pred):
    labels = pred.label_ids
    preds = pred.predictions.argmax(-1)
    f1 = f1_score(labels, preds, average="weighted")
    acc = accuracy_score(labels, preds)
    return {"accuracy": acc, "f1": f1}
```

Nachdem wir den Datensatz und die Berechnung der Maße vorbereitet haben, müssen wir uns nur noch um zwei letzte Dinge kümmern, bevor wir die `Trainer`-Klasse definieren:

1. Melden Sie sich bei Ihrem Konto auf dem Hugging Face Hub an. So können Sie das feingetunte Modell auf unser Konto im Hub hochladen und es mit der Community teilen.
2. Legen Sie alle Hyperparameter für den Trainingslauf fest.

Genau mit diesen Schritten befassen wir uns im nächsten Abschnitt.

Das Modell trainieren

Wenn Sie diesen Code in einem Jupyter Notebook ausführen möchten, können Sie die folgende Hilfsfunktion verwenden, um sich beim Hub anzumelden:

```
from huggingface_hub import notebook_login

notebook_login()
```

Daraufhin wird ein Widget angezeigt, in das Sie Ihren Benutzernamen und Ihr Passwort bzw. ein Zugangstoken mit Schreibberechtigung eingeben sollen. Alle erforderlichen Informationen zur Erstellung von Zugriffstokens finden Sie in der Dokumentation des Hubs (*https://oreil.ly/IRkN1*). Sollten Sie über das Terminal bzw. die Kommandozeile arbeiten, können Sie zur Anmeldung den folgenden Befehl ausführen:

```
$ huggingface-cli login
```

Um die Parameter für das Training zu spezifizieren, verwenden wir die `TrainingArguments`-Klasse. Diese Klasse speichert eine Vielzahl von Informationen und gibt Ihnen die Möglichkeit, das Training und die Evaluierung gezielt zu steuern. Das wichtigste zu spezifizierende Argument ist `output_dir`, mit dem Sie angeben, wo alle Artefakte, die beim Training erstellt werden, gespeichert werden. Hier ein Musterbeispiel dafür, wie Sie die `TrainingArguments`-Klasse verwenden können:

```
from transformers import Trainer, TrainingArguments

batch_size = 64
logging_steps = len(emotions_encoded["train"]) // batch_size
model_name = f"{model_ckpt}-finetuned-emotion"
training_args = TrainingArguments(output_dir=model_name,
                                  num_train_epochs=2,
                                  learning_rate=2e-5,
                                  per_device_train_batch_size=batch_size,
                                  per_device_eval_batch_size=batch_size,
                                  weight_decay=0.01,
                                  evaluation_strategy="epoch",
                                  disable_tqdm=False,
                                  logging_steps=logging_steps,
                                  push_to_hub=True,
                                  save_strategy="epoch",
                                  load_best_model_at_end=True,
                                  log_level="error")
```

Hier legen wir auch die Batchgröße (engl. Batch Size), die Lernrate (engl. Learning Rate) und die Anzahl der Epochen fest und geben an, dass nach Abschluss des Trainingslaufs das beste Modell geladen werden soll. Nachdem wir diesen letzten

Schritt vollzogen haben, können wir unser Modell mithilfe der `Trainer`-Klasse instanziieren und feintunen:

```
from transformers import Trainer

trainer = Trainer(model=model, args=training_args,
                  compute_metrics=compute_metrics,
                  train_dataset=emotions_encoded["train"],
                  eval_dataset=emotions_encoded["validation"],
                  tokenizer=tokenizer)
trainer.train();
```

Epoche	Trainingsverlust	Validierungsverlust	Treffergenauigkeit (Accuracy)	F1-Maß
1	0.840900	0.327445	0.896500	0.892285
2	0.255000	0.220472	0.922500	0.922550

Ein Blick auf die Logging-Daten offenbart, dass unser Modell in Bezug auf den Validierungsdatensatz einen Wert für das F_1-Maß von ca. 92 % erreicht – eine erhebliche Verbesserung gegenüber dem Feature-basierten Ansatz!

Indem wir die Konfusionsmatrix ermitteln, können wir einen besseren Einblick als über die beim Training ermittelten Qualitätsmaße gewinnen. Damit wir die Konfusionsmatrix visualisieren können, müssen wir zunächst ermitteln, welche Vorhersagen das Modell in Bezug auf den Validierungsdatensatz trifft. Die `predict()`-Methode der `Trainer`-Klasse gibt mehrere nützliche Objekte zurück, die wir zur Evaluierung einbeziehen können:

```
preds_output = trainer.predict(emotions_encoded["validation"])
```

Infolge des Aufrufs der `predict()`-Methode wird ein `PredictionOutput`-Objekt zurückgegeben, das jeweils ein Array mit den Vorhersagen (`predictions`) und den IDs der entsprechenden Labels (`label_ids`) umfasst, zusammen mit den Maßen, die wir für das `Trainer`-Objekt spezifiziert haben. Beispielsweise können Sie auf die für den Validierungsdatensatz ermittelten Maße wie folgt zugreifen:

```
preds_output.metrics
```

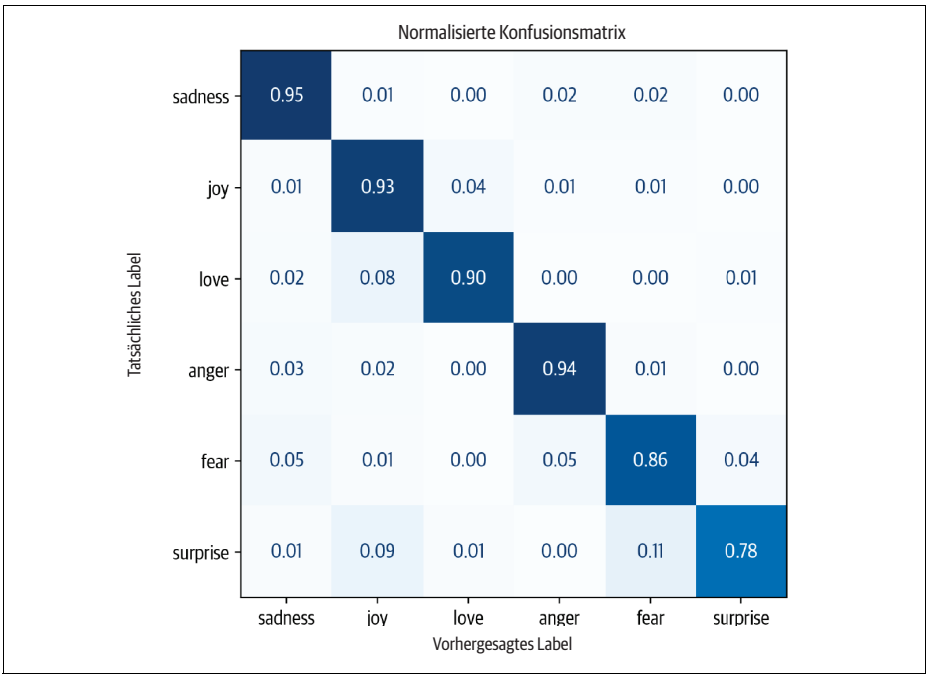
```
{'test_loss': 0.22047173976898193,
 'test_accuracy': 0.9225,
 'test_f1': 0.9225500751072866,
 'test_runtime': 1.6357,
 'test_samples_per_second': 1222.725,
 'test_steps_per_second': 19.564}
```

Um zu ermitteln, welchen Kategorien die noch unverarbeiteten, rohen Vorhersagen entsprechen, können wir die Vorhersagen mithilfe der `np.argmax()`-Funktion decodieren (Greedy-Ansatz). Dadurch erhalten wir die vorhergesagten Labels in dem gleichen Format wie zuvor beim Scikit-learn-Modell im Rahmen des Feature-basierten Ansatzes:

```
y_preds = np.argmax(preds_output.predictions, axis=1)
```

Da wir nun ermittelt haben, welchen Kategorien die Vorhersagen entsprechen, können wir – wie zuvor beim Feature-basierten Ansatz – die Konfusionsmatrix visualisieren:

```
plot_confusion_matrix(y_preds, y_valid, labels)
```



Das kommt der idealen Konfusionsmatrix, bei der alle Werte auf der Diagonalen 1 sind, bedeutend näher. Die Kategorie `love` (Liebe) wird immer noch häufig mit `joy` (Freude) verwechselt, was naheliegend ist. Auch die Kategorie `surprise` (Überraschung) wird häufig mit `joy` (Freude) oder mit `fear` (Angst) verwechselt. Insgesamt scheint die Leistung des Modells allerdings recht gut zu sein. Doch bevor wir das Thema abschließen, sollten wir uns noch eingehender mit den Arten von Fehlern beschäftigen, die unserem Modell unterlaufen können.

Feintuning mit Keras

Wenn Sie TensorFlow verwenden, können Sie Ihre Modelle auch mithilfe der Keras-API feintunen. Der Hauptunterschied zur PyTorch-API besteht darin, dass es keine `Trainer`-Klasse gibt, da Keras-Modelle bereits eine integrierte `fit()`-Methode bieten. Sehen wir uns die Vorgehensweise einmal an und laden wir zunächst das DistilBERT- als TensorFlow-Modell:

```
from transformers import TFAutoModelForSequenceClassification

tf_model = (TFAutoModelForSequenceClassification
            .from_pretrained(model_ckpt, num_labels=num_labels))
```

Als Nächstes konvertieren wir unsere Datensätze in das `tf.data.Dataset`-Format. Da unsere Eingaben bereits tokenisiert und aufgefüllt wurden, können wir die Konvertierung problemlos mithilfe der `to_tf_dataset()`-Methode vornehmen und sie auf `emotions_encoded` anwenden:

```
# Namen der Spalten, die in TensorFlow-Tensoren konvertiert werden sollen
tokenizer_columns = tokenizer.model_input_names

tf_train_dataset = emotions_encoded["train"].to_tf_dataset(
    columns=tokenizer_columns, label_cols=["label"], shuffle=True,
    batch_size=batch_size)
tf_eval_dataset = emotions_encoded["validation"].to_tf_dataset(
    columns=tokenizer_columns, label_cols=["label"], shuffle=False,
    batch_size=batch_size)
```

Dabei haben wir auch den Trainingsdatensatz gemischt (engl. shuffle) und die Batchgröße für den Trainings- und Validierungsdatensatz festgelegt. Zu guter Letzt muss das Modell noch kompiliert und trainiert werden:

```
import tensorflow as tf

tf_model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=5e-5),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=tf.metrics.SparseCategoricalAccuracy())

tf_model.fit(tf_train_dataset, validation_data=tf_eval_dataset, epochs=2)
```

Fehleranalyse

Bevor wir fortfahren, sollten wir die Vorhersagen unseres Modells noch ein wenig genauer untersuchen. Ein einfacher, aber erfolgversprechender Ansatz besteht darin, die Beispiele des Validierungsdatensatzes danach zu sortieren, wie hoch jeweils der Verlust des Modells ausgefallen ist. Wenn wir die Labels während des Forward-Pass bereitstellen, wird automatisch der Verlust berechnet und zurückgegeben. Die folgende Funktion gibt den Verlust zusammen mit dem vorhergesagten Label zurück:

```
from torch.nn.functional import cross_entropy

def forward_pass_with_label(batch):
    # Alle Eingabe-Tensoren auf demselben Device wie Modell platzieren
    inputs = {k:v.to(device) for k,v in batch.items()
              if k in tokenizer.model_input_names}

    with torch.no_grad():
        output = model(**inputs)
        pred_label = torch.argmax(output.logits, axis=-1)
        loss = cross_entropy(output.logits, batch["label"].to(device),
                             reduction="none")
    # Ausgaben auf der CPU platzieren, um Kompatibilität mit
    #  anderen Spalten des Datensatzes zu gewährleisten.
    return {"loss": loss.cpu().numpy(),
            "predicted_label": pred_label.cpu().numpy()}
```

Mithilfe der `map()`-Methode können wir die Funktion auf alle Beispiele des Validierungsdatensatzes anwenden und die Werte für den Verlust erhalten:

```
# Datensatz zurück in PyTorch-Tensoren konvertieren
emotions_encoded.set_format("torch",
                            columns=["input_ids", "attention_mask", "label"])
# Werte für den Verlust berechnen
emotions_encoded["validation"] = emotions_encoded["validation"].map(
    forward_pass_with_label, batched=True, batch_size=16)
```

Zum Schluss erstellen wir einen `DataFrame`, der jeweils die Texte, die Werte für den Verlust und sowohl die vorhergesagten als auch die tatsächlichen (d.h. wahren) Labels umfasst:

```
emotions_encoded.set_format("pandas")
cols = ["text", "label", "predicted_label", "loss"]
df_test = emotions_encoded["validation"][:][cols]
df_test["label"] = df_test["label"].apply(label_int2str)
df_test["predicted_label"] = (df_test["predicted_label"]
                              .apply(label_int2str))
```

Jetzt sind wir so weit, dass wir die Spalte `emotions_encoded` ganz einfach nach der Höhe des Verlusts in aufsteigender oder absteigender Reihenfolge sortieren können. Im Rahmen der Fehleranalyse verfolgen wir das Ziel, eines der beiden folgenden Probleme zu erkennen:

Unzutreffende Labels

Jeder Vorgang, bei dem Daten mit Labels versehen werden (auch Annotation genannt), kann mit Fehlern einhergehen. Diejenigen, die die Labels vergeben (sogenannte Annotatoren), können Fehler machen oder unterschiedlicher Meinung sein. Ebenso können Labels, die aus anderen Features abgeleitet wurden, unzutreffend sein. Wenn es einfach wäre, Daten automatisch zu labeln, bräuchten wir erst gar kein Vorhersagemodell. Folglich ist es ganz normal, dass es einige Beispiele gibt, deren Label(s) nicht korrekt ist. Mithilfe des von uns verfolgten Ansatzes können wir sie schnell ausfindig machen und korrigieren.

Unvollkommenheiten des Datensatzes

Die Datensätze, die in der Praxis anzutreffen sind, sind immer ein wenig unsauber. Wenn Sie mit Textdaten arbeiten, können Sonderzeichen oder gewisse Strings, die in den Eingaben enthalten sind, die Vorhersagen des Modells stark beeinflussen. Indem Sie die schlechtesten Vorhersagen des Modells (d.h. die mit dem höchsten Verlust) ausfindig machen, können Sie solche Faktoren identifizieren und die Daten anschließend bereinigen oder auch ähnliche Beispiele hinzufügen, die das Modell robuster machen.

Ermitteln wir nun zunächst die Beispiele in unseren Daten, die den höchsten Verlust aufweisen:

```
df_test.sort_values("loss", ascending=False).head(10)
```

text	label	predicted_label	loss
i feel that he was being overshadowed by the supporting characters	love	sadness	5.704531
i called myself pro life and voted for perry without knowing this information i would feel betrayed but moreover i would feel that i had betrayed god by supporting a man who mandated a barely year old vaccine for little girls putting them in danger to financially support people close to him	joy	sadness	5.484461
i guess i feel betrayed because i admired him so much and for someone to do this to his wife and kids just goes beyond the pale	joy	sadness	5.434768
i feel badly about reneging on my commitment to bring donuts to the faithful at holy family catholic church in columbus ohio	love	sadness	5.257482
i as representative of everything thats wrong with corporate america and feel that sending him to washington is a ludicrous idea	surprise	sadness	4.827708
i guess this is a memoir so it feels like that should be fine too except i dont know something about such a deep amount of self absorption made me feel uncomfortable	joy	fear	4.713047
i am going to several holiday parties and i can t wait to feel super awkward i am going to several holiday parties and i can t wait to feel super awkward a href http badplaydate	joy	sadness	4.704955
i felt ashamed of these feelings and was scared because i knew that something wrong with me and thought i might be gay	fear	sadness	4.656096
i guess we would naturally feel a sense of loneliness even the people who said unkind things to you might be missed	anger	sadness	4.593202
im lazy my characters fall into categories of smug and or blas people and their foils people who feel inconvenienced by smug and or blas people	joy	fear	4.311287

Wie sich zeigt, hat das Modell einige der Labels falsch vorhergesagt. Andererseits scheint es eine ganze Reihe von Beispielen zu geben, die an sich nicht eindeutig einer Kategorie zugeordnet werden können und die entweder falsch gelabelt wurden oder eine komplett neue Kategorie erfordern würden. Insbesondere Beispiele, die der Emotion bzw. dem Label `joy` (Freude) zugeordnet wurden, scheinen in einigen Fällen falsch gelabelt worden zu sein. Auf der Grundlage dieser Informationen können wir nun den Datensatz entsprechend anpassen. Dadurch verbessert sich die Qualität bzw. Genauigkeit des Modells oftmals in gleichem (oder sogar höherem) Maße, als wenn wir mehr Daten zur Verfügung hätten oder größere Modelle verwenden würden.

Wenn wir uns die Beispiele ansehen, bei denen der Verlust am niedrigsten ausfällt, stellen wir fest, dass das Modell die Kategorie `sadness` (Trauer) am zuverlässigsten vorherzusagen scheint. Deep-Learning-Modelle sind außergewöhnlich gut darin, Abkürzungen zu finden und auszunutzen, um zu einer Vorhersage zu gelangen. Aus diesem Grund lohnt es sich auch, sich die Beispiele anzusehen, bei denen das Modell am sichersten ist: Auf diese Weise können wir sicherstellen, dass das Modell bestimmte Merkmale des Texts nicht falsch verwertet. Werfen wir nun also

auch einen Blick auf die Vorhersagen, die mit dem geringsten Verlust des Modells einhergehen:

```
df_test.sort_values("loss", ascending=True).head(10)
```

text	label	predicted_label	loss
i feel try to tell me im ungrateful tell me im basically the worst daughter sister in the world	sadness	sadness	0.017331
im kinda relieve but at the same time i feel disheartened	sadness	sadness	0.017392
i and feel quite ungrateful for it but i m looking forward to summer and warmth and light nights	sadness	sadness	0.017400
i remember feeling disheartened one day when we were studying a poem really dissecting it verse by verse stanza by stanza	sadness	sadness	0.017461
i feel like an ungrateful asshole	sadness	sadness	0.017485
i leave the meeting feeling more than a little disheartened	sadness	sadness	0.017670
i am feeling a little disheartened	sadness	sadness	0.017685
i feel like i deserve to be broke with how frivolous i am	sadness	sadness	0.017888
i started this blog with pure intentions i must confess to starting to feel a little disheartened lately by the knowledge that there doesnt seem to be anybody reading it	sadness	sadness	0.017899
i feel so ungrateful to be wishing this pregnancy over now	sadness	sadness	0.017913

Abschließend können wir festhalten, dass zum einen die Kategorie `joy` (Freude) scheinbar manchmal falsch gelabelt wurde und dass das Modell zum anderen die Kategorie `sadness` (Trauer) mit der größten Konfidenz vorhersagt. Auf Basis dieser Informationen können wir unseren Datensatz gezielt verbessern und auch die Kategorie im Auge behalten, bei der das Modell sehr sicher zu sein scheint.

Bevor wir das trainierte Modell bereitstellen können, müssen wir es erst noch für die spätere Verwendung speichern. Wie im nächsten Abschnitt gezeigt, können wir dies mithilfe der 🤗 Transformers-Bibliothek in nur wenigen Schritten umsetzen.

Das Modell speichern und teilen

Die NLP-Community profitiert in hohem Maße davon, vortrainierte und feingetunte Modelle untereinander auszutauschen. Jeder kann seine Modelle über den Hugging Face Hub mit anderen teilen, und jedes von der Community erstellte Modell kann auf die gleiche Weise vom Hub heruntergeladen werden, wie wir es zuvor für das DistilBERT-Modell bewerkstelligt haben. Mit der `Trainer`-Klasse ist es denkbar einfach, ein Modell zu speichern und zu teilen:

```
trainer.push_to_hub(commit_message="Training completed!")
```

Jetzt können wir das feingetunte Modell dafür verwenden, Vorhersagen für neue Tweets zu treffen. Da wir unser Modell auf den Hub hochgeladen haben, können wir es mit der `pipeline()`-Funktion verwenden – genauso wie wir es in Kapitel 1 gehandhabt haben. Laden wir zunächst die Pipeline:

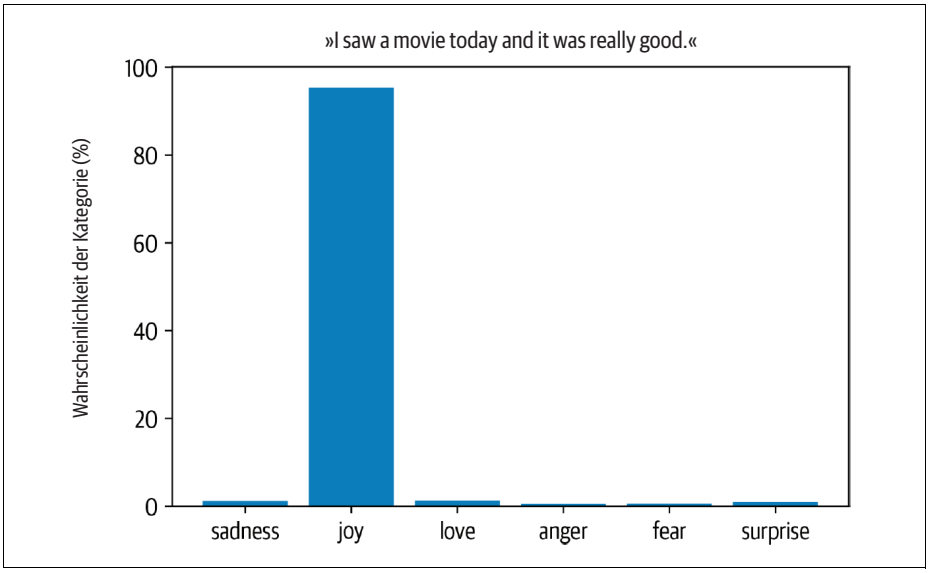
```
from transformers import pipeline

# Ersetzen Sie `transformersbook` durch Ihren Hub-Benutzernamen
model_id = "transformersbook/distilbert-base-uncased-finetuned-emotion"
classifier = pipeline("text-classification", model=model_id)
```

Jetzt können wir die Pipeline mit einem Beispiel-Tweet testen:

```
custom_tweet = "I saw a movie today and it was really good."
preds = classifier(custom_tweet, return_all_scores=True)
```

Zum Schluss können wir die für die einzelnen Kategorien ermittelten Wahrscheinlichkeiten in einem Balkendiagramm darstellen. Das Modell schätzt, dass die wahrscheinlichste Kategorie `joy` (Freude) ist. In Anbetracht des Tweets scheint dies durchaus plausibel zu sein:

```
preds_df = pd.DataFrame(preds[0])
plt.bar(labels, 100 * preds_df["score"], color='C0')
plt.title(f'"{custom_tweet}"')
plt.ylabel("Wahrscheinlichkeit der Kategorie (%)")
plt.show()
```



Zusammenfassung

Herzlichen Glückwunsch! Sie wissen jetzt, wie man ein Transformer-Modell trainiert, um in Tweets zum Ausdruck gebrachte Emotionen zu klassifizieren. Wir haben uns zwei sich ergänzenden Ansätzen gewidmet – einerseits auf Features und andererseits auf Feintuning basierend – und ihre Stärken und Schwächen beleuchtet.

Dies ist jedoch nur der erste Schritt beim Aufbau einer realen Anwendung, die auf Transformer-Modellen fußt, und es liegt noch eine Menge Arbeit vor uns. Es gibt

eine Reihe von Herausforderungen, auf die Sie auf Ihrer NLP-Reise wahrscheinlich stoßen werden:

Mein Chef möchte, dass mein Modell bereits gestern in Produktion gegangen ist!
In den meisten Anwendungen liegt Ihr Modell nicht einfach irgendwo herum und sammelt Staub – Sie wollen sicherstellen, dass es Vorhersagen liefert! Wenn ein Modell auf den Hub geladen wird, wird automatisch ein Endpunkt zum Treffen von Vorhersagen (engl. Inference Endpoint) erstellt, der mittels HTTP-Anfragen (engl. Requests) aufgerufen werden kann. Wenn Sie mehr darüber erfahren möchten, empfehlen wir Ihnen, einen Blick in die Dokumentation der Inference API (*https://oreil.ly/XACF5*) zu werfen.

Meine Anwender wünschen sich schnellere Vorhersagen!
Für dieses Problem haben wir bereits einen Lösungsansatz kennengelernt: die Verwendung des DistilBERT-Modells. In Kapitel 8 werden wir uns mit der Knowledge Distillation (dem Verfahren, mit dem das DistilBERT-Modell erstellt wurde) und anderen Tricks beschäftigen, mit denen Sie Ihre Transformer-Modelle schneller machen können.

Kann Ihr Modell auch für X verwendet werden?
Wie wir in diesem Kapitel bereits angedeutet haben, sind Transformer-Modelle äußerst vielseitig. Im weiteren Verlauf des Buchs werden wir uns mit einer Reihe von Aufgaben befassen – wie dem Question Answering und der Named Entity Recognition –, bei denen alle dieselbe grundlegende Architektur zum Einsatz kommt.

Keiner meiner Texte ist auf Englisch!
Inzwischen gibt es auch Transformer-Modelle, die mehrsprachig sind, und wir werden in Kapitel 4 auf sie zurückgreifen, um mehrere Sprachen auf einmal zu handhaben.

Ich habe keine Labels!
Wenn nur sehr wenige gelabelte Daten verfügbar sind, ist es unter Umständen nicht möglich, ein Feintuning vorzunehmen. In Kapitel 9 werden wir einige Techniken erkunden, um dieser Situation zu begegnen.

Nachdem wir nun gesehen haben, wie ein Transformer trainiert und mit anderen geteilt werden kann, werden wir uns im nächsten Kapitel damit beschäftigen, unser eigenes Transformer-Modell von Grund auf zu implementieren.

KAPITEL 3

Die Anatomie von Transformer-Modellen

In Kapitel 2 haben Sie erfahren, wie ein Transformer-Modell feingetunt und evaluiert wird. Werfen wir nun einen Blick darauf, wie diese Modelle eigentlich im Kern funktionieren. In diesem Kapitel lernen Sie die wichtigsten Bausteine von Transformer-Modellen kennen und wie Sie sie in PyTorch implementieren können. Darüber hinaus zeigen wir Ihnen, wie sich das Gleiche in TensorFlow umsetzen lässt. Wir werden uns zunächst darauf konzentrieren, den Attention-Mechanismus einzurichten, und nach und nach die einzelnen Komponenten hinzufügen, die notwendig sind, um einen (rein) Encoder-basierten Transformer zu implementieren. Zudem werden wir uns kurz mit den architektonischen Unterschieden zwischen dem Encoder- und dem Decoder-Modul befassen. Am Ende dieses Kapitels werden Sie in der Lage sein, selbst ein einfaches Transformer-Modell zu implementieren.

In der Regel benötigen Sie kein tiefgreifendes Fachwissen über die Transformer-Architektur, um die 🤗 Transformers-Bibliothek verwenden und Modelle für Ihren Anwendungsfall feintunen zu können. Allerdings kann es hilfreich sein, sich der Grenzen von Transformer-Modellen bewusst zu sein und auch zu wissen, wie Sie mit ihnen umgehen und die Modelle in neuen Domänen bzw. Anwendungsbereichen einsetzen können.

In diesem Kapitel wird auch eine Taxonomie der Transformer-Modelle vorgestellt, die Ihnen helfen soll, einen Einblick in die große Vielfalt von Modellen zu gewinnen, die in den letzten Jahren entstanden ist. Bevor wir uns in den Code vertiefen, sollten wir uns zunächst einen Überblick über die ursprüngliche Architektur verschaffen, mit der die Transformer-Revolution ihren Anfang nahm.

Die Transformer-Architektur

Wie wir in Kapitel 1 gesehen haben, basiert der ursprüngliche Transformer auf der *Encoder-Decoder*-Architektur, die üblicherweise für Aufgaben wie die maschinelle Übersetzung verwendet wird, bei der eine Folge von Wörtern von einer Sprache in eine andere übersetzt wird. Diese Architektur besteht aus zwei Komponenten:

Encoder (im Deutschen auch Codierer genannt)

Wandelt eine Eingabesequenz von Tokens in eine Sequenz von Embedding-Vektoren (bzw. Einbettungsvektoren) um, die oft als *verborgener Zustand* (engl. Hidden State) oder *Kontext* (engl. Context) bezeichnet werden.

Decoder (im Deutschen auch Decodierer genannt)

Verwendet den verborgenen Zustand des Encoders, um Schritt für Schritt eine Ausgabesequenz von Tokens zu erzeugen, d. h. ein Token nach dem anderen.

Wie in Abbildung 3-1 dargestellt, bestehen der Encoder und der Decoder ihrerseits selbst aus mehreren Bausteinen.

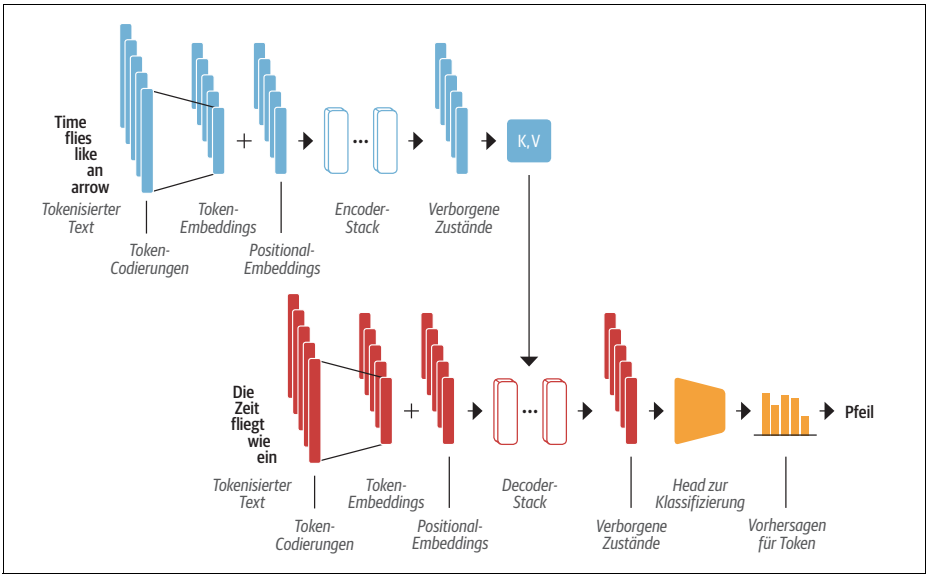


Abbildung 3-1: Encoder-Decoder-Architektur von Transformern, wobei der Encoder in der oberen Hälfte der Abbildung und der Decoder in der unteren Hälfte dargestellt ist

Wir werden uns die einzelnen Komponenten in Kürze noch genauer ansehen, können aber in Abbildung 3-1 bereits einige Dinge erkennen, die für die Transformer-Architektur charakteristisch sind:

- Der Eingabetext wird mithilfe der Techniken, die wir in Kapitel 2 kennengelernt haben, in Tokens umgewandelt (d. h. tokenisiert) und in sogenannte *Token-Embeddings* überführt. Da der Attention-Mechanismus die relativen Positionen der Tokens unberücksichtigt lässt, bedarf es einer Möglichkeit, zu berücksichtigen, an welcher Stelle die Tokens in der Texteingabe stehen, um so den sequenziellen Charakter des Texts modellieren zu können. Die Token-Embeddings werden daher mit *Positional-Embeddings* (»positionsbezogene Einbettungen«) kombiniert, die Informationen zur Position der einzelnen Tokens enthalten.
- Der Encoder besteht aus einer Reihe von »aufeinandergestapelten« bzw. aneinandergereihten (engl. Stacking) *Encoder-Schichten* bzw. »-Blöcken«, die Sie

sich analog zu den aneinandergereihten Konvolutionsschichten (engl. Convolutional Layer), auch Faltungsschichten genannt, im Bereich der Computer Vision vorstellen können. Das Gleiche gilt für den Decoder, der ebenfalls aus einer Reihe von aneinandergereihten *Decoder-Schichten* besteht.

- Die Ausgabe des Encoders wird in jede Decoder-Schicht eingespeist. Der Decoder sagt dann voraus, welches Token mit der größten Wahrscheinlichkeit an nächster Stelle steht. Die Ausgabe dieses Schritts wird dann wieder in den Decoder zurückgeführt, um das nächste Token zu erzeugen, usw., bis ein spezielles EOS-Token am Ende der Sequenz (engl. End of Sequence, EOS) erreicht wird. Führen wir uns das Beispiel aus Abbildung 3-1 vor Augen und gehen wir davon aus, dass der Decoder bereits »Die« und »Zeit« vorhergesagt hat. Nun erhält er diese beiden sowie alle Ausgaben des Encoders als Eingabe, um das nachfolgende Token, »fliegt«, vorherzusagen. Im nächsten Schritt erhält der Decoder »fliegt« als zusätzliche Eingabe. Dieser Vorgang wird so lange wiederholt, bis der Decoder das EOS-Token vorhersagt oder die maximale Sequenzlänge erreicht wurde.

Auch wenn die Transformer-Architektur ursprünglich für Sequence-to-Sequence-Aufgaben (d. h., sowohl die Eingabe als auch die Ausgabe des Modells stellen eine Sequenz dar) wie die maschinelle Übersetzung entwickelt wurde, wurden sowohl der Encoder- als auch der Decoder-Block bereits nach kurzer Zeit als eigenständige Modelle übernommen. Obwohl es Hunderte von verschiedenen Transformer-Modellen gibt, gehören die meisten von ihnen zu einer der drei folgenden verschiedenen Kategorien:

Rein Encoder-basiert (»Encoder-only«)
: Diese Modelle wandeln eine eingegebene Textsequenz in eine reichhaltige numerische Darstellung um, die sich gut für Aufgaben wie die Textklassifizierung oder die Eigennamenerkennung (engl. Named Entity Recognition, NER) eignet. Dieser Kategorie von Architekturen können wir BERT und seine abgewandelten Varianten, wie RoBERTa und DistilBERT, zuordnen. Die für ein bestimmtes Token in dieser Architektur berechnete Darstellung hängt sowohl von dem linken (vor dem Token) als auch von dem rechten (nach dem Token) Kontext ab. Dies wird oft als *bidirektionale Attention* bezeichnet.

Rein Decoder-basiert (»Decoder-only«)
: Diese Modelle vervollständigen eine vorgegebene Eingabesequenz (Prompt) wie »Danke für das Mittagessen, es hat ...« automatisch, indem sie schrittweise das nächste Wort vorhersagen, das jeweils am wahrscheinlichsten ist. In diese Kategorie fällt die Familie der GPT-Modelle. Die für ein bestimmtes Token in dieser Architektur berechnete Darstellung hängt nur von dem linken Kontext ab. Dies wird oft als *kausale* oder *autoregressive Attention* bezeichnet.

Encoder-Decoder-basiert
: Sie werden für die Modellierung komplexer Zuordnungen (engl. Mappings) von einer Textsequenz zu einer anderen verwendet und eignen sich für maschinelle Übersetzungen und Textzusammenfassungsaufgaben. Neben den Transformer-Modellen, die wir bereits kennengelernt haben, gehören auch

die BART- und T5-Modelle zu dieser Kategorie, die dadurch charakterisiert ist, das jeweils ein Encoder und ein Decoder kombiniert werden.

In Wirklichkeit ist die Trennlinie zwischen den Anwendungen für reine Decoder- und reine Encoder-basierte Architekturen ein wenig unscharf. So können beispielsweise reine Decoder-Modelle wie die der GPT-Familie für Aufgaben wie die maschinelle Übersetzung eingesetzt werden, die jedoch üblicherweise als Sequence-to-Sequence-Aufgaben betrachtet werden. Ebenso können rein Encoder-basierte Modelle wie BERT für Textzusammenfassungsaufgaben zum Einsatz kommen, die normalerweise Encoder-Decoder- oder rein Decoder-basierten Modellen zugeschrieben werden.[1]

Nachdem Sie nun ein grundlegendes Verständnis von der Transformer-Architektur haben, sollten wir uns das Innenleben des Encoders genauer ansehen.

Der Encoder

Wie wir bereits gesehen haben, besteht der Encoder des Transformers aus vielen aneinandergereihten Encoder-Schichten. Wie in Abbildung 3-2 dargestellt, erhält jede Encoder-Schicht eine Reihe von Embeddings und leitet sie durch die folgenden Teilschichten:

- eine Multi-Head-Attention-Schicht (»mehrköpfige Aufmerksamkeitsschicht«)
- eine vollständig verbundene Feed-Forward-Schicht (d.h. vorwärtsgerichtete Schicht), die auf jedes eingegebene Embedding (engl. Input Embedding) angewandt wird

Die ausgegebenen Embeddings (engl. Output Embeddings) jeder Encoder-Schicht haben die gleiche Dimension wie die Eingaben. Sie werden bald erfahren, dass die Hauptaufgabe des Encoder-Stacks darin besteht, die Input-Embeddings so zu »aktualisieren«, dass Darstellungen erzeugt werden, die gewisse kontextuelle Informationen in der Sequenz codieren. Zum Beispiel wird das Wort »Apple« so aktualisiert, dass es mit größerer Wahrscheinlichkeit dem Unternehmensbereich und nicht den Früchten zuzuordnen ist, wenn die Wörter »Keynote« oder »Phone« in dessen Kontext verwendet werden.

In allen Teilschichten werden zudem Skip-Verbindungen verwendet und eine Normalisierung (engl. Layer Normalization) vorgenommen. Dies sind Standardtricks, die dazu beitragen, tiefe neuronale Netze auf effektive Weise trainieren zu können. Doch um die Funktionsweise eines Transformers wirklich nachvollziehen zu können, müssen wir noch weiter ins Detail gehen. Beginnen wir mit dem wichtigsten Baustein: der Self-Attention-Schicht (»Selbstaufmerksamkeitsschicht«).

1 Y. Liu und M. Lapata, »Text Summarization with Pretrained Encoder« (*https://arxiv.org/abs/1908.08345*), (2019).

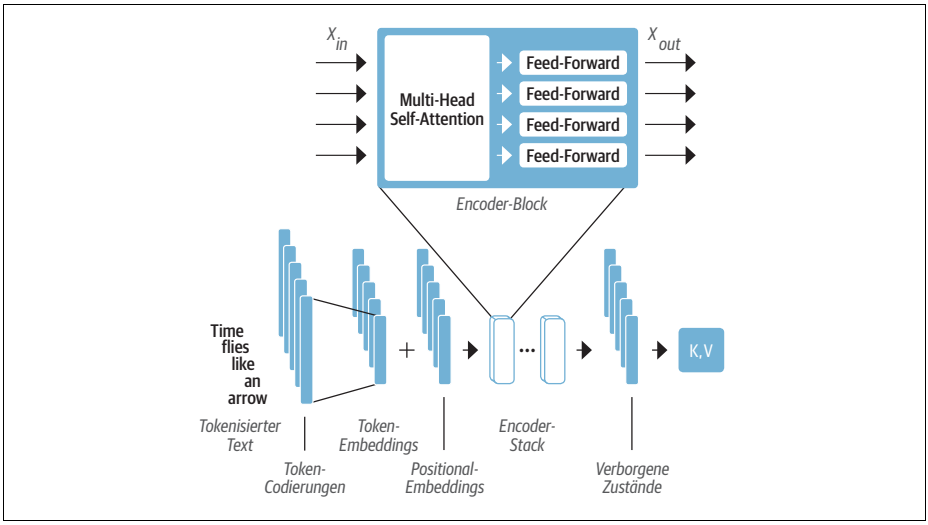


Abbildung 3-2: Die Encoder-Schicht unter der Lupe

Self-Attention

Wie wir in Kapitel 1 erläutert haben, ist Attention ein Mechanismus, der es neuronalen Netzen ermöglicht, jedem Element in einer Sequenz eine unterschiedliche Gewichtung zuzuweisen bzw. »Aufmerksamkeit« zu schenken. Bei Textsequenzen handelt es sich bei den Elementen um *Token-Embeddings*, wie wir sie im Kapitel 1 kennengelernt haben, bei denen jedes Token auf einen Vektor einer fest vorgegebenen Dimension abgebildet wird. In BERT wird beispielsweise jedes Token als ein 768-dimensionaler Vektor dargestellt. Der »Self« bzw. »Selbst«-Teil in Self-Attention bzw. »Selbstaufmerksamkeit« bezieht sich auf die Tatsache, dass diese Gewichte für alle verborgenen Zustände desselben Blocks berechnet werden – beispielsweise für alle verborgenen Zustände des Encoders. Im Gegensatz dazu wird beim Attention-Mechanismus, der bei rekurrenten Modellen zum Einsatz kommt, die Relevanz jedes verborgenen Zustands des Encoders für den verborgenen Zustand des Decoders zu einem bestimmten Decodierungsschritt berechnet.

Die Hauptidee hinter Self-Attention ist, dass wir statt eines festen Embeddings für jedes Token die gesamte Sequenz dazu heranziehen können, einen *gewichteten Durchschnitt* auf Basis der einzelnen Embeddings zu berechnen. Anders ausgedrückt: Gegeben einer Reihe von Token-Embeddings $x_1,\ldots,x_n$, erzeugt Self-Attention eine Reihe von neuen Embeddings $x'_1,\ldots,x'_n$, wobei jedes x'_i jeweils eine Linearkombination aller x_j darstellt:

$$x'_i = \sum_{j=1}^{n} w_{ji} x_j$$

Die Koeffizienten w_{ji} werden *Attention-Gewichte* (engl. Attention Weights) genannt und sind so normalisiert (bzw. normiert), dass $\sum_j w_{ji} = 1$. Um besser nach-

vollziehen zu können, warum die Durchschnittsbildung der Token-Embeddings eine gute Idee sein könnte, überlegen Sie, was Ihnen bei dem englischen Wort »flies« (auf Deutsch entweder das Substantiv »Fliegen« oder das Verb »fliegen«) in den Sinn kommt. Vielleicht denken Sie dabei ja an die lästigen Insekten. Doch wenn Sie mehr darüber wissen, in welchem Kontext es genutzt wird, wie z. B. in dem Satz »time flies like an arrow« (»die Zeit vergeht wie im Flug«), dann wäre für Sie ersichtlich, dass sich »flies« stattdessen auf das Verb (»to fly«) bezieht. In ähnlicher Weise können wir eine Darstellung für das Wort »flies« erstellen, bei der dieser Kontext berücksichtigt wird: indem wir die Token-Embeddings mit unterschiedlicher Gewichtung kombinieren, etwa indem wir den Token-Embeddings für »time« (»Zeit«) und »arrow« (»Pfeil« bzw. hier »im Flug«) ein höheres Gewicht w_{ji} zuweisen. Die auf diese Weise erzeugten Embeddings werden *kontextualisierte Embeddings* (engl. Contextualized Embeddings) genannt, die bereits vor Transformer-basierten Sprachmodellen wie ELMo eingeführt wurden.[2] Abbildung 3-3 veranschaulicht, wie je nach Kontext zwei verschiedene Darstellungen für das Wort »flies« mittels Self-Attention-Mechanismus erzeugt werden.

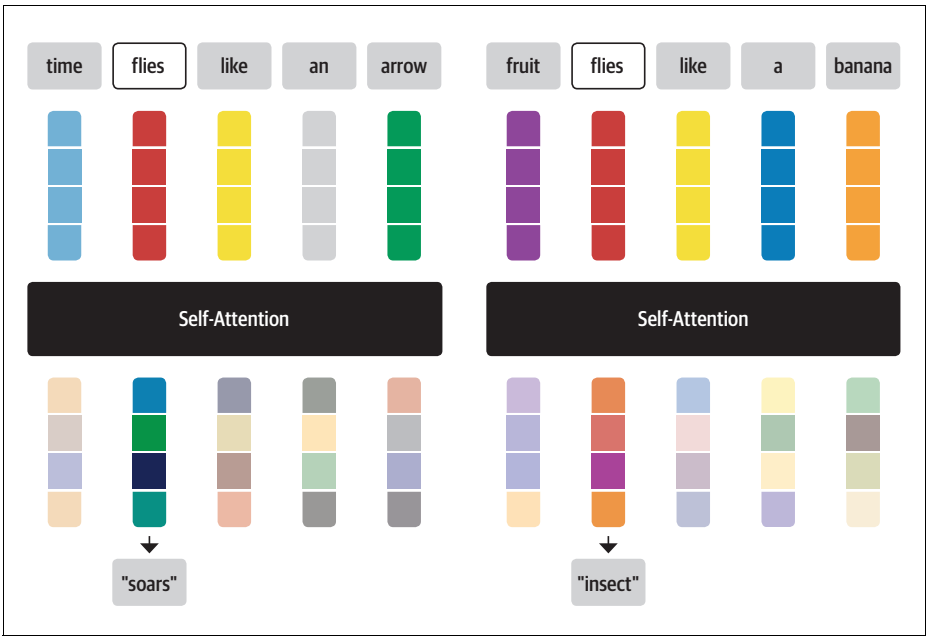


Abbildung 3-3: Die Abbildung zeigt, wie Self-Attention die reinen Token-Embeddings (oben) in kontextualisierte Embeddings (unten) überführt, d. h. in Darstellungen, die Informationen aus der gesamten Sequenz einbeziehen.

Widmen wir uns nun der Frage, wie wir die Attention-Gewichte berechnen können.

2 M. E. Peters et al., »Deep Contextualized Word Representations« (*https://arxiv.org/abs/1802.05365*), (2017).

Attention auf Basis des skalierten Skalarprodukts

Es gibt mehrere Möglichkeiten, wie eine Attention-Schicht implementiert werden kann. Die gebräuchlichste ist jedoch die Attention auf Basis des *skalierten Skalarprodukts* (engl. Scaled Dot-Product Attention), wie im Beitrag erläutert, in dem die Transformer-Architektur vorgestellt wurde.[3] Zur Umsetzung dieses Mechanismus sind vier grundlegende Schritte erforderlich:

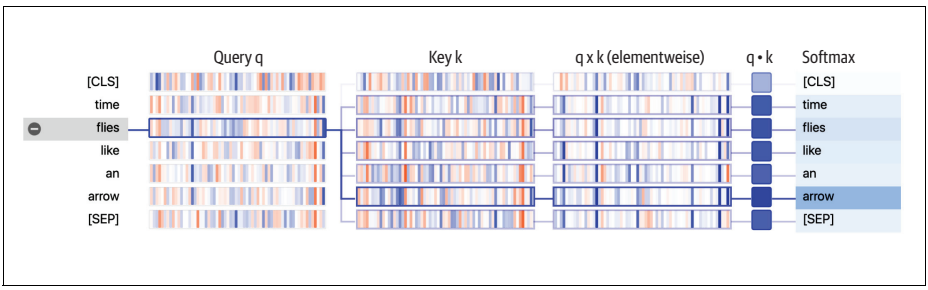
1. Abbildung jedes Token-Embedding auf drei Vektoren (d.h. eine Projektion), und zwar auf einen *Abfrage-* (engl. Query), einen *Schlüssel-* (engl. Key) und einen *Wertvektor* (engl. Value).
2. Berechnung der Attention-Scores. Mithilfe einer *Ähnlichkeitsfunktion* bestimmen wir, inwieweit der Abfrage- und der Schlüsselvektor miteinander in Beziehung stehen bzw. sich ähneln. Wie bereits der Name vermuten lässt, wird bei dieser Form von Attention als Ähnlichkeitsfunktion das skalierte Skalarprodukt verwendet, das effizient durch Matrixmultiplikation der Embeddings berechnet werden kann. Abfragen- und Schlüsselvektoren, die sich ähneln, weisen ein hohes Skalarprodukt (auch inneres Produkt oder Punktprodukt genannt) auf, während Abfragen- und Schlüsselvektoren, die nicht viel gemeinsam haben, wenig bis keine Ähnlichkeit aufweisen. Die Ergebnisse dieses Schritts werden als *Attention-Scores* (»Aufmerksamkeitswerte«) bezeichnet, und eine Sequenz mit n Eingabe- bzw. Input-Tokens ergibt eine $n \times n$-Matrix von Attention-Scores.
3. Berechnung der Attention-Gewichte. Skalarprodukte können im Allgemeinen beliebig große Zahlen hervorbringen, wodurch der Trainingsvorgang destabilisiert werden kann. Daher werden die Attention-Scores zunächst mit einem Skalierungsfaktor multipliziert, um ihre Varianz zu skalieren, und dann mithilfe einer Softmax-Funktion normalisiert, um sicherzustellen, dass die Summe aller Spaltenwerte 1 ergibt. Die resultierende n x n-Matrix umfasst alle Attention-Gewichte w_{ji}.
4. Aktualisierung der Token-Embeddings. Sobald die Attention-Gewichte berechnet wurden, multiplizieren wir sie mit dem Wertvektor, $v_1,\ldots,v_n$, um eine aktualisierte Darstellung für das Embedding, $x'_i = \sum_j w_{ji} v_j$, zu erhalten.

Mithilfe einer praktischen Bibliothek namens *BertViz* for Jupyter (*https://oreil.ly/eQK3I*) können wir visualisieren, wie die Attention-Gewichte berechnet werden. Sie bietet mehrere Funktionen, mit denen Sie verschiedene Aspekte des Attention-Mechanismus in Transformer-Modellen darstellen können. Zur Visualisierung der Attention-Gewichte können wir das Modul `neuron_view` verwenden, mit dem sich die Berechnung der Gewichte nachvollziehen lässt. Dadurch können Sie nachvollziehen, wie die Abfrage- und Schlüsselvektoren kombiniert werden, um den endgültigen Wert für das Gewicht zu erhalten. Da die BertViz-Bibliothek auf die Attention-Schicht des Modells zugreifen muss, müssen wir zunächst unseren Checkpoint für

3 A. Vaswani et al., »Attention Is All You Need« (*https://arxiv.org/abs/1706.03762*), (2017).

das BERT-Modell unter Verwendung der `model`-Klasse von BertViz instanziieren und dann die Funktion `show()` verwenden, um jeweils für eine bestimmte Encoder-Schicht und einen Attention-Head eine interaktive Visualisierung zu erzeugen. Beachten Sie, dass Sie auf das »+« auf der linken Seite klicken müssen, um die Visualisierung der Attention zu aktivieren:

```
from transformers import AutoTokenizer
from bertviz.transformers_neuron_view import BertModel
from bertviz.neuron_view import show

model_ckpt = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
model = BertModel.from_pretrained(model_ckpt)
text = "time flies like an arrow"
show(model, "bert", tokenizer, text, display_mode="light", layer=0, head=8)
```



In der Abbildung sind die Werte der Abfrage- und Schlüsselvektoren als vertikale Balken dargestellt, wobei die Intensität der einzelnen Balken jeweils die Höhe des Beitrags widerspiegelt. Die Verbindungslinien sind je nach Attention zwischen den Tokens unterschiedlich intensiv dargestellt. Wie Sie sehen können, weist der Abfragevektor für »flies« die größte Ähnlichkeit mit dem Schlüsselvektor für »arrow« auf.

Abfrage-, Schlüssel- und Wertvektoren auf verständliche Weise erklärt

Die Begriffe Abfrage-, Schlüssel- und Wertvektor mögen Ihnen beim ersten Mal etwas kryptisch erscheinen. Auch wenn ihre Namensgebung auf Informationsabfragesysteme (engl. Information Retrieval Systems) zurückzuführen ist, können wir ihre Bedeutung mit einer einfachen Analogie verdeutlichen. Stellen Sie sich vor, Sie sind im Supermarkt und kaufen alle Zutaten, die Sie für Ihr Abendessen benötigen. Jede der benötigten Zutaten aus dem Rezept Ihres Gerichts können Sie sich als eine Abfrage vorstellen. Während Sie die Regale durchsuchen, sehen Sie sich die Etiketten an (Schlüssel) und prüfen, ob sie mit einer Zutat auf Ihrer Liste übereinstimmen (Ähnlichkeitsfunktion). Wenn dies der Fall ist, nehmen Sie den Artikel (Wert) aus dem Regal.

In dieser Analogie erhalten Sie für jedes Etikett, das mit der Zutat übereinstimmt, nur einen Lebensmittelartikel. Self-Attention stellt eine abstraktere und »glattere« Version davon dar: *jedes* Etikett im Supermarkt stimmt in dem Maße mit der Zutat überein, wie die jeweiligen Schlüssel mit der Abfrage übereinstimmen. Wenn Sie also ein Dutzend Eier auf Ihrer Liste haben, kaufen Sie am Ende vielleicht 10 Eier, ein Omelett und einen Hühnerflügel.

Die einzelnen Schritte können Sie noch einmal anhand von Abbildung 3-4 nachvollziehen, in der die einzelnen Operationen, die zur Berechnung der Attention auf Basis des skalierten Skalarprodukts erforderlich sind, dargestellt werden.

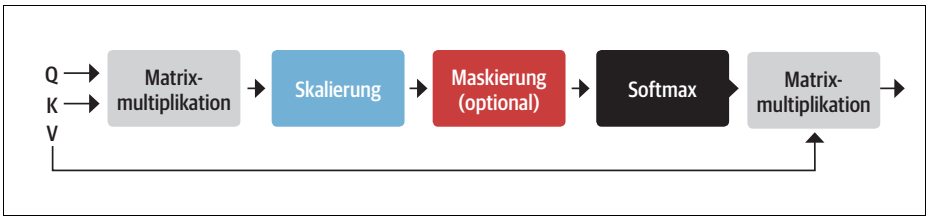


Abbildung 3-4: Die einzelnen Operationen, die zur Berechnung der Attention auf Basis des skalierten Skalarprodukts erforderlich sind

In diesem Kapitel greifen wir zur Implementierung der Transformer-Architektur auf PyTorch zurück. In TensorFlow sind die einzelnen Schritte jedoch analog. In Tabelle 3-1 werden die wichtigsten Funktionen beider Frameworks gegenübergestellt.

Tabelle 3-1: In diesem Kapitel verwendete Klassen und Methoden von PyTorch und TensorFlow (Keras)

PyTorch	TensorFlow (Keras)	Erstellt/implementiert
`nn.Linear`	`keras.layers.Dense`	Dichte, d. h. vollständig verbundene, neuronale Netzwerkschicht (engl. Dense Layer)
`nn.Module`	`keras.layers.Layer`	Grundbausteine von Modellen
`nn.Dropout`	`keras.layers.Dropout`	Dropout-Schicht
`nn.LayerNorm`	`keras.layers.LayerNormalization`	Layer Normalization bzw. Schichtnormalisierung
`nn.Embedding`	`keras.layers.Embedding`	Embedding-Schicht
`nn.GELU`	`keras.activations.gelu`	Gaussian-Error-Linear-Unit-Aktivierungsfunktion
`nn.bmm`	`tf.matmul`	Batch-Matrix-Matrixprodukt
`model.forward`	`model.call`	Forward-Pass bzw. Vorwärtsdurchlauf des Modells

Als Erstes müssen wir den Text in Tokens umwandeln (d.h. tokenisieren). Verwenden wir also unseren Tokenizer, um die IDs der Eingaben (Input-IDs) zu erhalten:

```
inputs = tokenizer(text, return_tensors="pt", add_special_tokens=False)
inputs.input_ids
```

```
tensor([[ 2051, 10029,  2066,  2019,  8612]])
```

Wie wir bereits in Kapitel 2 erfahren haben, wird jedes Token in einem Satz einer eindeutigen ID im Vokabular des Tokenizers zugeordnet bzw. auf diese abgebildet. Der Einfachheit halber haben wir auch dafür gesorgt, dass die Tokens `[CLS]` und `[SEP]` nicht berücksichtigt werden, indem wir `add_special_tokens=False` gesetzt haben. Als Nächstes müssen wir einige dichtbesetzte Einbettungen bzw. Dense-Embeddings erstellen. »Dichtbesetzt« bedeutet in diesem Zusammenhang, dass jeder Eintrag in den Embeddings einen Wert von ungleich null aufweist. Im Gegensatz dazu sind die One-Hot-Codierungen, die wir in Kapitel 2 kennengelernt haben, *dünnbesetzt* bzw. *schwachbesetzt* (engl. Sparse), da alle Einträge bis auf einen den Wert null aufweisen. In PyTorch können wir dies mithilfe einer `torch.nn.Embedding`-Schicht umsetzen, die als Nachschlagetabelle (engl. Lookup Table) für sämtliche Input-IDs dient:

```
from torch import nn
from transformers import AutoConfig

config = AutoConfig.from_pretrained(model_ckpt)
token_emb = nn.Embedding(config.vocab_size, config.hidden_size)
token_emb
```

```
Embedding(30522, 768)
```

Hier haben wir die `AutoConfig`-Klasse verwendet, um die mit dem `bert-base-uncased`-Checkpoint assoziierte *config.json*-Datei zu laden. In der 🤗 Transformers-Bibliothek ist jedem Checkpoint eine bestimmte Konfigurationsdatei zugeordnet, in der verschiedene Hyperparameterwerte wie `vocab_size` und `hidden_size` angegeben sind. In unserem Beispiel hat dies zur Folge, dass jede Input-ID auf einen der 30.522 Embedding-Vektoren abgebildet wird, die in `nn.Embedding` gespeichert sind, wobei jeder Vektor eine Länge (engl. Size) von 768 hat. Die `AutoConfig`-Klasse speichert auch zusätzliche Metadaten, wie z.B. die Namen der Labels, die dazu verwendet werden, die Vorhersagen des Modells in ein lesbares Format zu überführen.

Beachten Sie, dass die Token-Embeddings an dieser Stelle nicht davon abhängen, in welchem Kontext die Tokens jeweils genutzt werden. Das bedeutet, dass Homonyme (d.h. Wörter, die dieselbe Schreibweise, aber eine unterschiedliche Bedeutung haben) wie »flies« im vorherigen Beispiel, die gleiche Darstellung besitzen. Die Aufgabe der nachfolgenden Attention-Schichten besteht darin, diese Token-Embeddings in Darstellungen bzw. Repräsentationen zu überführen, bei denen die einzelnen Tokens voneinander unterscheidbar und mit dem jeweiligen Kontext verknüpft sind.

Nachdem wir nun unsere Nachschlagetabelle zur Hand haben, können wir die Embeddings erstellen, indem wir die Input-IDs einspeisen:

```
inputs_embeds = token_emb(inputs.input_ids)
inputs_embeds.size()
```

```
torch.Size([1, 5, 768])
```

Auf diese Weise erhalten wir einen Tensor mit einem Shape von `[batch_size, seq_len, hidden_dim]` – wie wir es bereits in Kapitel 2 gesehen haben. Wir gehen erst später auf die Positionscodierungen (engl. Positional Encodings) ein, sodass der nächste Schritt darin besteht, die Abfrage-, Schlüssel- und Wertvektoren zu erstellen und die Attention-Scores zu berechnen, wobei wir als Ähnlichkeitsfunktion das Skalarprodukt verwenden:

```
import torch
from math import sqrt

query = key = value = inputs_embeds
dim_k = key.size(-1)
scores = torch.bmm(query, key.transpose(1,2)) / sqrt(dim_k)
scores.size()
```

```
torch.Size([1, 5, 5])
```

Dadurch wurde eine 5 × 5-Matrix von Attention-Scores je Beispiel im Batch erstellt. Wir werden später sehen, dass die Abfrage-, Schlüssel- und Wertvektoren durch die Anwendung voneinander unabhängiger Gewichtungsmatrizen $W_{Q,K,V}$ auf die Embeddings erzeugt werden. Für den Moment haben wir sie jedoch der Einfachheit halber identisch gehalten. Bei der mit dem skalierten Skalarprodukt berechneten Attention werden die Skalarprodukte entsprechend der Länge der Embedding-Vektoren skaliert, um zu verhindern, dass während des Trainings zu viele große Zahlen entstehen, die die Softmax-Funktion, die als Nächstes angewendet wird, in den Sättigungsbereich bringen können.

Die Funktion `torch.bmm()` ermittelt ein *Batch-Matrix-Matrixprodukt* (engl. Batch Matrix-Matrix Product), das die Berechnung der Attention-Scores vereinfacht, wenn die Abfrage- und Schlüsselvektoren einen Shape von `[batch_size, seq_len, hidden_dim]` aufweisen. Wenn wir die Batch-Dimension ignorieren würden, könnten wir das Skalarprodukt zwischen jedem Abfrage- und Schlüsselvektor berechnen, indem wir den Schlüssel-Tensor einfach transponieren, sodass er einen Shape von `[hidden_dim, seq_len]` hat, und dann das Matrixprodukt bilden und alle Skalarprodukte in einer `[seq_len, seq_len]`-Matrix sammeln. Da wir dies für alle Sequenzen in einem Batch unabhängig voneinander durchführen möchten, verwenden wir `torch.bmm()`, das zwei Batches mit den zugehörigen Matrizen nimmt und jede Matrix des ersten Batches mit der entsprechenden Matrix im zweiten Batch multipliziert.

Wenden wir nun die Softmax-Funktion an:

```
import torch.nn.functional as F

weights = F.softmax(scores, dim=-1)
weights.sum(dim=-1)
```

```
tensor([[1., 1., 1., 1., 1.]], grad_fn=<SumBackward1>)
```

Der letzte Schritt besteht darin, die Attention-Gewichte mit dem Wertvektor zu multiplizieren:

```
attn_outputs = torch.bmm(weights, value)
attn_outputs.shape
```

```
torch.Size([1, 5, 768])
```

Das war's bereits! Wir haben alle notwendigen Schritte vollzogen, um selbst eine vereinfachte Form der Self-Attention zu implementieren! Beachten Sie, dass der gesamte Vorgang lediglich zwei Matrizenmultiplikationen und eine einmalige Anwendung der Softmax-Funktion umfasst hat. Halten wir also fest, dass Sie sich »Self-Attention« lediglich wie eine ausgefallene Form der Mittelwertbildung vorstellen müssen.

Fassen wir diese Schritte in einer Funktion zusammen, um sie später verwenden zu können:

```
def scaled_dot_product_attention(query, key, value):
    dim_k = query.size(-1)
    scores = torch.bmm(query, key.transpose(1, 2)) / sqrt(dim_k)
    weights = F.softmax(scores, dim=-1)
    return torch.bmm(weights, value)
```

Unser Attention-Mechanismus, bei dem wir identische Abfrage- und Schlüsselvektoren verwendet haben, wird identischen Wörtern im Kontext – und insbesondere dem jeweils aktuellen Wort selbst – einen sehr hohen Score zuweisen. Im Allgemeinen lässt sich die Bedeutung eines Worts jedoch besser durch komplementäre Wörter, die im Kontext verwendet werden, bestimmen als durch identische Wörter – zum Beispiel lässt sich die Bedeutung von »flies« besser bestimmen, wenn man die Information, dass es mit den Wörtern »time« und »arrow« verwendet wird, mit einbezieht, als wenn das Wort »flies« noch einmal Erwähnung findet. Allerdings stellt sich die Frage, wie wir ein solches Verhalten herbeiführen können?

Erlauben wir dem Modell nun, dass es verschiedene Vektoren für die Abfrage, den Schlüssel und den Wert eines Tokens erstellt, indem es drei verschiedene lineare Projektionen vornimmt, um unseren ursprünglichen Token-Vektor in drei verschiedene Räume zu projizieren.

Multi-Headed Attention

In unserem einfachen Beispiel haben wir die Embeddings lediglich »so wie sie sind« verwendet, um die Attention-Scores und -Gewichte zu berechnen. Das ist allerdings bei Weitem nicht die ganze Wahrheit. In der Praxis wendet die Self-Atten-

tion-Schicht drei voneinander unabhängige lineare Transformationen auf jedes Embedding an, um die Abfrage-, Schlüssel- und Wertvektoren zu erzeugen. Diese Transformationen projizieren die Embeddings. Jede Projektion ist mit einer eigenen Reihe von lernbaren Parametern versehen, sodass sich die Self-Attention-Schicht auf verschiedene semantische Aspekte der Sequenz konzentrieren kann.

Es erweist sich auch als vorteilhaft, *mehrere* Sätze linearer Projektionen zu haben, von denen jede einen sogenannten *Attention-Head* (»Aufmerksamkeitskopf«) darstellt. Die daraus resultierende *Multi-Head-Attention-Schicht* ist in Abbildung 3-5 dargestellt. Doch wozu benötigen wir mehr als einen Attention-Head? Der Grund ist, dass die Anwendung der Softmax-Funktion in einem Head dazu führt, dass sich dieser hauptsächlich auf einen Aspekt der Ähnlichkeit fokussiert. Mit mehreren Heads kann sich das Modell auf mehrere Aspekte gleichzeitig fokussieren. So kann sich beispielsweise ein Head auf die Interaktion zwischen Subjekt und Verb konzentrieren, während ein anderer Head nahe gelegene Adjektive findet. Diese Beziehungen werden natürlich nicht von Hand in das Modell eingearbeitet, sondern vollständig auf der Grundlage der Daten gelernt. Wenn Sie mit Computer-Vision-Modellen vertraut sind, erkennen Sie vielleicht die Ähnlichkeit mit Filtern in neuronalen Konvolutionsnetzen (engl. Convolutional Neural Networks), bei denen ein Filter für die Erkennung von Gesichtern zuständig sein kann und ein anderer die Räder von Autos in Bildern erfasst.

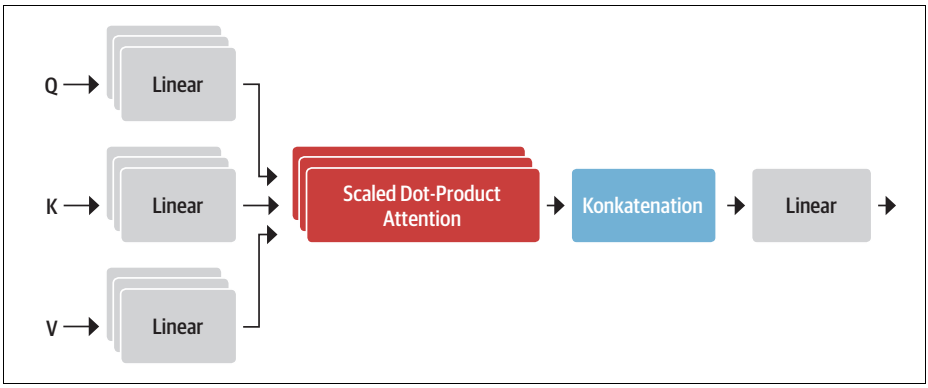


Abbildung 3-5: Multi-Head Attention

Nehmen wir nun die Implementierung dieser Schicht vor, indem wir zunächst einen einzelnen Attention-Head codieren:

```
class AttentionHead(nn.Module):
    def __init__(self, embed_dim, head_dim):
        super().__init__()
        self.q = nn.Linear(embed_dim, head_dim)
        self.k = nn.Linear(embed_dim, head_dim)
        self.v = nn.Linear(embed_dim, head_dim)

    def forward(self, hidden_state):
        attn_outputs = scaled_dot_product_attention(
            self.q(hidden_state), self.k(hidden_state), self.v(hidden_state))
        return attn_outputs
```

Hier haben wir drei voneinander unabhängige lineare Schichten initialisiert, die eine Matrixmultiplikation auf die Embedding-Vektoren vornehmen, um Tensoren zu erzeugen, die einen Shape von `[batch_size, seq_len, head_dim]` aufweisen, wobei `head_dim` der Anzahl der Dimensionen entspricht, in denen die Projektion vorgenommen wird. Obwohl `head_dim` nicht kleiner sein muss als die Anzahl der Embedding-Dimensionen der Tokens (`embed_dim`), wird `head_dim` in der Praxis so gewählt, dass `embed_dim` stets ein Vielfaches von `head_dim` darstellt, sodass die einzelnen Heads einheitlich und effizient berechnet werden können. BERT verfügt beispielsweise über 12 Attention-Heads. Dementsprechend hat jeder Head eine Dimension von 768 / 12 = 64.

Da wir nun einen einzelnen Attention-Head haben, können wir die Ausgaben der einzelnen Heads miteinander verketten (engl. concatenate), um eine Schicht mit mehreren Heads (Multi-Head-Attention-Schicht) zu implementieren:

```
class MultiHeadAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        embed_dim = config.hidden_size
        num_heads = config.num_attention_heads
        head_dim = embed_dim // num_heads
        self.heads = nn.ModuleList(
            [AttentionHead(embed_dim, head_dim) for _ in range(num_heads)]
        )
        self.output_linear = nn.Linear(embed_dim, embed_dim)

    def forward(self, hidden_state):
        x = torch.cat([h(hidden_state) for h in self.heads], dim=-1)
        x = self.output_linear(x)
        return x
```

Beachten Sie, dass die verkettete Ausgabe der Attention-Heads auch durch eine letzte lineare Schicht geleitet wird, um einen Ausgabetensor mit einem Shape von `[batch_size, seq_len, hidden_dim]` zu erzeugen, der für das nachgelagerte Feed-Forward-Netz geeignet ist. Werfen wir nun einen Blick auf unsere Multi-Head-Attention-Schicht, um zu überprüfen, ob deren Ausgaben dem erwarteten Shape der Eingaben für das nachgelagerte Feed-Forward-Netz entspricht. Bei der Initialisierung des `MultiHeadAttention`-Moduls übergeben wir die zuvor geladene Konfiguration des vortrainierten BERT-Modells. Damit stellen wir sicher, dass wir ein identisches Setting wie bei BERT verwenden:

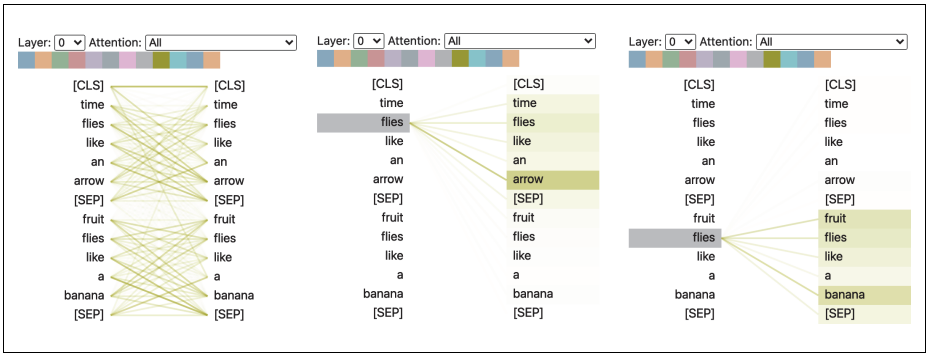
```
multihead_attn = MultiHeadAttention(config)
attn_output = multihead_attn(inputs_embeds)
attn_output.size()
```

```
torch.Size([1, 5, 768])
```

Es funktioniert! Zum Abschluss dieses Abschnitts über Attention greifen wir noch einmal auf die BertViz-Bibliothek zurück, um die Attention für zwei unterschiedliche Verwendungen des englischen Worts »flies« zu visualisieren. Hierzu können wir die Funktion `head_view()` aus der BertViz-Bibliothek verwenden. Zuvor müssen

wir lediglich noch die Attentions des vortrainierten Checkpoints berechnen und angegeben, an welcher Stelle die Grenze beider Sätze liegt:

```
from bertviz import head_view
from transformers import AutoModel

model = AutoModel.from_pretrained(model_ckpt, output_attentions=True)

sentence_a = "time flies like an arrow"
sentence_b = "fruit flies like a banana"

viz_inputs = tokenizer(sentence_a, sentence_b, return_tensors='pt')
attention = model(**viz_inputs).attentions
sentence_b_start = (viz_inputs.token_type_ids == 0).sum(dim=1)
tokens = tokenizer.convert_ids_to_tokens(viz_inputs.input_ids[0])

head_view(attention, tokens, sentence_b_start, heads=[8])
```



In der Visualisierung werden die Attention-Gewichte als Linien dargestellt, die jeweils das Token, dessen Embedding aktualisiert wird (links), mit allen Wörtern, auf die die Aufmerksamkeit (bzw. Attention) gerichtet ist (rechts), verbindet. Die Intensität der Linien spiegelt wider, wie unterschiedlich hoch die Attention-Gewichte ausfallen. Dunkle Linien stehen für Werte nahe 1 und blasse Linien für Werte nahe 0.

In diesem Beispiel besteht die Eingabe aus zwei Sätzen. Die Tokens `[CLS]` und `[SEP]` sind die speziellen Tokens des Tokenizers von BERT, auf die wir bereits in Kapitel 2 gestoßen sind. Eine Sache, die wir aus der Visualisierung erkennen können, ist, dass die Attention-Gewichte zwischen Wörtern, die zum selben Satz gehören, am höchsten ausfallen, was darauf hindeutet, dass BERT erkannt hat, dass es auf die Wörter im selben Satz achten sollte. Bei dem Wort »flies« können wir jedoch sehen, dass BERT im ersten Satz »arrow« und im zweiten Satz »fruit« und »banana« als wichtig identifiziert hat. Dank dieser Attention-Gewichtung kann das Modell also unterscheiden, ob »flies« als Verb oder als Substantiv verwendet wird – je nach Kontext, in dem es vorkommt.

Nachdem wir nun die Attention-Schicht erläutert haben, sollten wir uns die Implementierung des fehlenden Teils der Encoder-Schicht ansehen: positionsbezogene Feed-Forward-Netzwerke (engl. Position-Wise Feed-Forward Networks).

Die Feed-Forward-Schicht

Die vorwärtsgerichtete (engl. Feed-Forward) Teilschicht im Encoder und Decoder ist ein einfaches, zweischichtiges, vollständig verbundenes neuronales Netz, das jedoch mit einer Besonderheit versehen ist: Anstatt alle Embeddings als einen einzigen Vektor zu verarbeiten, verarbeitet es jedes Embedding *unabhängig* voneinander. Aus diesem Grund wird diese Schicht oft als *Position-Wise-Feed-Forward-Schicht* bezeichnet. Sie wird in der Regel von Personen mit einem Hintergrund in der Computer Vision auch als eindimensionale Faltung bzw. Konvolution (engl. Convolution) mit einer Kernel- bzw. Filtergröße von 1 bezeichnet (z. B. ist diese Bezeichnung in der Codebasis von OpenAI's GPT-Modell üblich). Als Aktivierungsfunktion wird am häufigsten eine GELU-Aktivierungsfunktion verwendet, und eine Faustregel aus der Literatur besagt, dass die Dimension der ersten Schicht (`hidden_size`) das Vierfache der Länge der Embeddings betragen sollte. Diesem Teil kann der größte Teil der Aufnahme- und Erinnerungsfähigkeit zugeschrieben werden. Wenn das Modell skaliert werden soll, wird meist dieser Teil skaliert. Die Teilschicht können wir wie folgt als einfaches `nn.Module` implementieren:

```
class FeedForward(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.linear_1 = nn.Linear(config.hidden_size, config.intermediate_size)
        self.linear_2 = nn.Linear(config.intermediate_size, config.hidden_size)
        self.gelu = nn.GELU()
        self.dropout = nn.Dropout(config.hidden_dropout_prob)

    def forward(self, x):
        x = self.linear_1(x)
        x = self.gelu(x)
        x = self.linear_2(x)
        x = self.dropout(x)
        return x
```

Beachten Sie, dass eine vorwärtsgerichtete Schicht wie `nn.Linear` normalerweise auf einen Tensor mit einem Shape von (`batch_size, input_dim`) angewandt wird, wobei sie auf alle Elemente der jeweiligen Batch-Dimension unabhängig operiert. Wenn wir also einen Tensor mit einem Shape von (`batch_size, seq_len, hidden_dim`) übergeben, wird diese Schicht auf alle Token-Embeddings des Batches und der Sequenz unabhängig voneinander angewandt, was genau das ist, was wir erreichen wollen. Probieren wir es aus, indem wir die Ausgaben der Attention-Schicht übergeben:

```
feed_forward = FeedForward(config)
ff_outputs = feed_forward(attn_outputs)
ff_outputs.size()
```

```
torch.Size([1, 5, 768])
```

Wir haben nun alle erforderlichen Komponenten, um eine voll funktionsfähige Encoder-Schicht für Transformer-Modelle zu erstellen. Die einzige Entscheidung, die wir noch treffen müssen, ist, an welcher Stelle wir die Skip-Verbindungen und die Layer Normalization platzieren. Sehen wir uns an, welche Auswirkungen dies auf die Modellarchitektur hat.

Layer Normalization integrieren

Wie bereits erwähnt, nutzt die Transformer-Architektur die *Layer Normalization* (sozusagen eine Schichtnormalisierung) und *Skip-Verbindungen* (engl. Skip Connections). Erstere normalisiert alle in einem Batch enthaltenen Eingaben so, dass sie einen Mittelwert von null und eine Varianz von eins aufweisen. Skip-Verbindungen leiten einen Tensor ohne Verarbeitung an die nächste Schicht des Modells weiter und addieren ihn zu dem verarbeiteten Tensor (Anm. d. Übersetzers: Die Schicht wird sozusagen »übersprungen«, vom Englischen »to skip«). Wenn es darum geht, wo die Schichtnormalisierung in den Encoder- oder Decoder-Schichten eines Transformers zu platzieren ist, werden in der Literatur vor allem zwei verschiedene Ansätze verfolgt:

Post Layer Normalization
: Dies ist die Anordnung, die in dem ursprünglichen Transformer-Forschungsbeitrag verwendet wurde. Die Normalisierung wird zwischen den Skip-Verbindungen platziert. Es ist schwierig, diese Anordnung von Grund auf zu trainieren, da die Gradienten divergieren können. Aus diesem Grund werden Sie oft ein Konzept antreffen, das als *Learning Rate Warm-up* bekannt ist, bei dem die Lernrate während des Trainings schrittweise – ausgehend von einem kleinen bis zu einem maximalen Wert – erhöht wird.

Pre Layer Normalization
: Dies ist die in der Literatur am häufigsten anzutreffende Anordnung. Bei ihr wird die Normalisierung innerhalb der Skip-Verbindungen platziert. Dieser Ansatz ist in der Regel während des Trainings bedeutend stabiler und erfordert keine »Warm-up«-Phase für die Lernrate.

Der Unterschied zwischen den beiden Anordnungen wird in Abbildung 3-6 veranschaulicht.

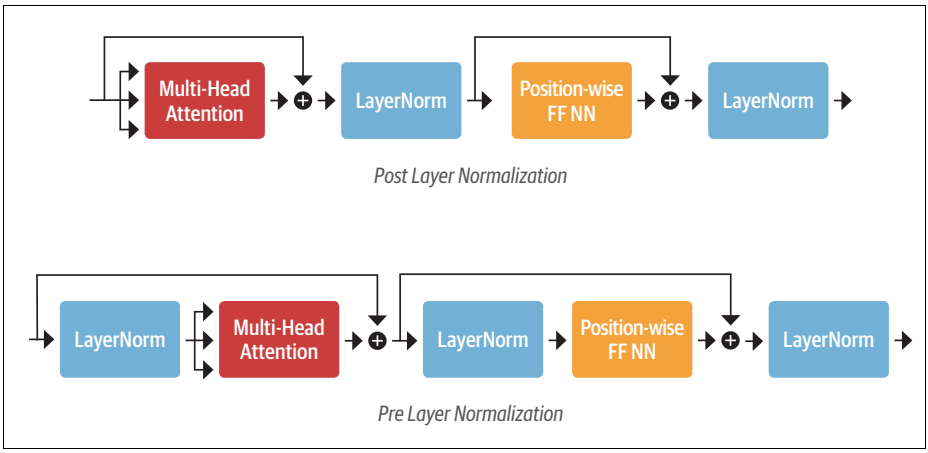


Abbildung 3-6: Verschiedene Anordnungen für die Layer Normalization in einer Encoder-Schicht eines Transformers

Wir verwenden die zweite Anordnungsmöglichkeit, sodass wir unsere Bausteine einfach wie folgt zusammenfügen können:

```
class TransformerEncoderLayer(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.layer_norm_1 = nn.LayerNorm(config.hidden_size)
        self.layer_norm_2 = nn.LayerNorm(config.hidden_size)
        self.attention = MultiHeadAttention(config)
        self.feed_forward = FeedForward(config)

    def forward(self, x):
        # Normalisierung der Schicht und dann Eingabe in
        #  Abfrage-, Schlüssel- und Wertvektor kopieren
        hidden_state = self.layer_norm_1(x)
        # Attention mit einer Skip-Verbindung
        x = x + self.attention(hidden_state)
        # Feed-Forward-Schicht mit einer Skip-Verbindung
        x = x + self.feed_forward(self.layer_norm_2(x))
        return x
```

Lassen Sie uns dies nun mit unseren Input-Embeddings ausprobieren:

```
encoder_layer = TransformerEncoderLayer(config)
inputs_embeds.shape, encoder_layer(inputs_embeds).size()
```

```
(torch.Size([1, 5, 768]), torch.Size([1, 5, 768]))
```

Wir haben jetzt unsere allererste Encoder-Schicht eines Transformers von Grund auf implementiert. Allerdings gibt es bei der Art und Weise, wie wir die Schichten des Encoders eingerichtet haben, einen Haken: Sie sind völlig unabhängig davon, an welcher Position bzw. Stelle die Tokens stehen. Da es sich bei der Multi-Head-Attention-Schicht im Grunde um eine ausgeklügelt gewichtete Summenbildung handelt, geht die Information über die Position der Tokens verloren.[4]

Zum Glück gibt es einen einfachen Trick, mit dem Sie Informationen zur Position mithilfe von Positional Embeddings einbinden können. Nehmen wir das mal unter die Lupe.

Positional-Embeddings

Positional-Embeddings basieren auf einer einfachen, aber sehr effektiven Idee: Die Token-Embeddings werden um ein in einem Vektor angelegtes positionsabhängiges Muster von Werten erweitert. Wenn das Muster für die jeweilige Position charakteristisch ist, können die Attention-Heads und Feed-Forward-Schichten in jedem Stack lernen, positionsbezogene Informationen in ihre Transformationen einzubeziehen.

4 In der Fachterminologie spricht man davon, dass die Self-Attention-Schicht und die Feed-Forward-Schicht *permutationsäquivariant* (engl. permutation equivariant) sind – wenn die Eingabe permutiert wird, dann wird die entsprechende Ausgabe der Schicht auf genau die gleiche Weise permutiert.

Es gibt mehrere Möglichkeiten, wie sich dies umsetzen lässt. Einer der beliebtesten Ansätze ist die Verwendung eines erlernbaren Musters, insbesondere dann, wenn der Datensatz für das Pretraining ausreichend groß ist. Dies funktioniert genauso wie die Token-Embeddings, wobei jedoch der Positionsindex anstelle der Token-ID als Eingabe verwendet wird. Bei diesem Ansatz wird während des Pretrainings gelernt, wie die Positionen der Tokens auf effiziente Weise codiert werden können.

Erstellen wir ein selbst definiertes `Embeddings`-Modul, das eine Token-Embedding-Schicht, die die `input_ids` auf einen dichtbesetzten verborgenen Zustand projiziert, mit dem Positional-Embedding kombiniert, das das Gleiche für die `position_ids` vornimmt. Das resultierende Embedding entspricht einfach der Summe der beiden Embeddings:

```
class Embeddings(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.token_embeddings = nn.Embedding(config.vocab_size,
                                             config.hidden_size)
        self.position_embeddings = nn.Embedding(config.max_position_embeddings,
                                                config.hidden_size)
        self.layer_norm = nn.LayerNorm(config.hidden_size, eps=1e-12)
        self.dropout = nn.Dropout()

    def forward(self, input_ids):
        # Positions-IDs für Eingabesequenz erstellen
        seq_length = input_ids.size(1)
        position_ids = torch.arange(seq_length, dtype=torch.long).unsqueeze(0)
        # Token- und Positional-Embeddings erstellen
        token_embeddings = self.token_embeddings(input_ids)
        position_embeddings = self.position_embeddings(position_ids)
        # Token- und Positional-Embedding kombinieren
        embeddings = token_embeddings + position_embeddings
        embeddings = self.layer_norm(embeddings)
        embeddings = self.dropout(embeddings)
        return embeddings

embedding_layer = Embeddings(config)
embedding_layer(inputs.input_ids).size()

torch.Size([1, 5, 768])
```

Wie wir sehen, erzeugt die Schicht nun ein einzelnes, dichtbesetztes Embedding für jedes Token.

Auch wenn erlernbare Positional-Embeddings einfach zu implementieren sind und häufig verwendet werden, gibt es einige Alternativen:

Absolute Positional-Embeddings

Transformer-Modelle können statische Muster verwenden, die aus modulierten Sinus- und Cosinussignalen bestehen, um die Positionen der Tokens zu codieren. Dies funktioniert besonders gut, wenn nicht allzu viele Daten zur Verfügung stehen.

Relative Positional-Embeddings

Obwohl die absoluten Positionen durchaus von Bedeutung sind, ließe sich argumentieren, dass die umgebenden Tokens bei der Berechnung eines Embedding am wichtigsten sind. Relative Positionsdarstellungen greifen diese Idee auf und codieren die relativen Positionen der Tokens. Dies lässt sich allerdings nicht dadurch bewerkstelligen, dass einfach eine neue Schicht für das relative Embedding eingefügt wird, denn das relative Embedding ändert sich für jedes Token, je nachdem, von welcher Stelle der Sequenz aus wir ihm unsere Aufmerksamkeit bzw. Attention widmen. Stattdessen wird der Attention-Mechanismus selbst mit zusätzlichen Termen angepasst, die die relativen Positionen der Tokens berücksichtigen. Modelle wie DeBERTa verwenden solche Darstellungen.[5]

Fügen wir nun all diese Komponenten zusammen und erstellen wir einen vollständigen Encoder eines Transformers. Hierzu führen wir die Embeddings mit den Schichten des Encoders zusammen:

```
class TransformerEncoder(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.embeddings = Embeddings(config)
        self.layers = nn.ModuleList([TransformerEncoderLayer(config)
                                     for _ in range(config.num_hidden_layers)])

    def forward(self, x):
        x = self.embeddings(x)
        for layer in self.layers:
            x = layer(x)
        return x
```

Überprüfen wir, ob die Ausgabe des Encoders den korrekten Shape hat:

```
encoder = TransformerEncoder(config)
encoder(inputs.input_ids).size()
```

```
torch.Size([1, 5, 768])
```

Wie wir erkennen können, erhalten wir für jedes Token im Batch einen verborgenen Zustand. Durch dieses Ausgabeformat ist die Architektur sehr flexibel, und wir können sie leicht für verschiedene Anwendungen, wie z. B. für die Vorhersage fehlender Tokens im Rahmen des Masked Language Modeling oder für die Vorhersage der Anfangs- und Endposition einer Antwort im Rahmen des Question Answering, anpassen. Im folgenden Abschnitt werden wir uns damit beschäftigen, wie wir einen Klassifikator wie den in Kapitel 2 verwendeten erstellen können.

5 Sogenannte Rotary Position Embeddings, bei denen die Idee absoluter und relativer Positionsdarstellungen kombiniert wird, erzielen bei vielen Aufgaben hervorragende Ergebnisse. GPT-Neo ist beispielsweise ein Modell mit Rotary Position Embeddings.

Einen Head zur Klassifizierung hinzufügen

Transformer-Modelle sind in der Regel in einen von der jeweiligen Aufgabe unabhängigen Body und einen aufgabenspezifischen Head unterteilt. Wir werden diesem prinzipiellen Aufbau in Kapitel 4 erneut begegnen, wenn wir uns dem Entwurfsmuster (Design Pattern) der 🤗 Transformers-Bibliothek zuwenden. Bislang haben wir lediglich den Body entwickelt. Wenn wir also einen Textklassifikator erstellen wollen, müssen wir diesem Body einen Head hinzufügen, der zur Klassifizierung dient. Für den Moment haben wir für jedes Token einen verborgenen Zustand vorliegen, benötigen allerdings nur eine einzelne Vorhersage. Es gibt mehrere Möglichkeiten, dies zu bewerkstelligen, und üblicherweise wird bei solchen Modellen das erste Token für die Vorhersage verwendet. Wir können eine Dropout- und eine lineare Schicht anhängen, damit wir eine Vorhersage zur Klassifizierung treffen können. Mit der folgenden Klasse können wir den bestehenden Encoder für die Klassifizierung von Sequenzen erweitern.

```
class TransformerForSequenceClassification(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.encoder = TransformerEncoder(config)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.classifier = nn.Linear(config.hidden_size, config.num_labels)

    def forward(self, x):
        # Verborgenen Zustand des [CLS]-Tokens in nächster Zeile auswählen
        x = self.encoder(x)[:, 0, :]
        x = self.dropout(x)
        x = self.classifier(x)
        return x
```

Bevor wir das Modell initialisieren, müssen wir festlegen, wie viele Kategorien wir vorhersagen (bzw. klassifizieren) möchten:

```
config.num_labels = 3
encoder_classifier = TransformerForSequenceClassification(config)
encoder_classifier(inputs.input_ids).size()
```

```
torch.Size([1, 3])
```

Das ist genau das, wonach wir gesucht haben. Für jedes Beispiel im Batch erhalten wir als Ausgabe die nicht-normalisierten Logits für jede Kategorie – wie bereits beim BERT-Modell, das wir in Kapitel 2 verwendet haben, um Emotionen in Tweets zu bestimmen.

Damit ist unsere Analyse des Encoders abgeschlossen, und wir wissen jetzt, wie wir ihn mit einem aufgabenspezifischen Head kombinieren können. Richten wir nun unsere Aufmerksamkeit (Wortspiel beabsichtigt!) auf den Decoder.

Der Decoder

Wie in Abbildung 3-7 dargestellt, besteht der Hauptunterschied zwischen dem Decoder und dem Encoder darin, dass der Decoder *zwei* Attention-Unterschichten bzw. -Sublayer hat:

Masked-Multi-Head-Self-Attention-Schicht
: Stellt sicher, dass die Tokens, die wir in jedem (Zeit-)Schritt erzeugen, nur auf den vorangegangenen Ausgaben und dem aktuell vorhergesagten Token basieren. Ohne dies könnte der Decoder während des Trainings betrügen, indem er einfach die Texte, die er eigentlich vorhersagen soll, kopiert. Dadurch, dass wir die Eingaben maskieren, stellen wir sicher, dass diese Aufgabe nicht so einfach zu bewältigen ist.

Encoder-Decoder-Attention-Schicht
: Führt Multi-Head-Attention über die Schlüssel- und -Wertvektoren, die vom Encoder-Stack ausgegeben werden, durch, wobei die Zwischendarstellungen des Decoders als Abfragevektoren fungieren.[6] Auf diese Weise lernt die Encoder-Decoder-Attention-Schicht, wie Tokens aus zwei verschiedenen Sequenzen, z. B. für zwei verschiedene Sprachen, in Beziehung gesetzt werden können. Der Decoder hat innerhalb jedes Blocks Zugriff auf die Schlüssel- und Wertvektoren des Encoders.

Gehen wir die Änderungen durch, die wir vornehmen müssen, um eine Form der Maskierung in unsere Self-Attention-Schicht aufzunehmen – die Implementierung der Encoder-Decoder-Attention-Schicht können Sie sich als Hausaufgabe vornehmen. Der Trick bei der Masked-Self-Attention besteht darin, eine *Maskierungsmatrix* (engl. Mask Matrix) einzuführen, die unterhalb (und auf) der Hauptdiagonale nur Einsen und darüber nur Nullen aufweist:

```
seq_len = inputs.input_ids.size(-1)
mask = torch.tril(torch.ones(seq_len, seq_len)).unsqueeze(0)
mask[0]
```

```
tensor([[1., 0., 0., 0., 0.],
        [1., 1., 0., 0., 0.],
        [1., 1., 1., 0., 0.],
        [1., 1., 1., 1., 0.],
        [1., 1., 1., 1., 1.]])
```

In diesem Beispiel haben wir die PyTorch-Funktion `tril()` verwendet, um die untere Dreiecksmatrix zu erstellen. Sobald wir diese Maskierungsmatrix zur Hand haben, müssen wir sie noch so anpassen, dass wir verhindern, dass die Attention-Heads zukünftige (d. h. nachfolgende) Tokens auslesen können. Hierzu verwenden wir die Funktion `Tensor.masked_fill()`, mit der wir alle Nullen durch einen Wert von minus unendlich ersetzen können:

6 Beachten Sie, dass im Gegensatz zur Self-Attention-Schicht die Schlüssel- und Abfragevektoren bei der Encoder-Decoder-Attention unterschiedliche Längen haben können. Das liegt daran, dass die Eingaben des Encoders und des Decoders in der Regel aus Sequenzen unterschiedlicher Länge bestehen. Daher ist die Matrix der Attention-Scores in dieser Schicht nicht quadratisch, sondern rechteckig.

```
scores.masked_fill(mask == 0, -float("inf"))

tensor([[[26.8082,    -inf,    -inf,    -inf,    -inf],
         [-0.6981, 26.9043,    -inf,    -inf,    -inf],
         [-2.3190,  1.2928, 27.8710,    -inf,    -inf],
         [-0.5897,  0.3497, -0.3807, 27.5488,    -inf],
         [ 0.5275,  2.0493, -0.4869,  1.6100, 29.0893]]],
       grad_fn=<MaskedFillBackward0>)
```

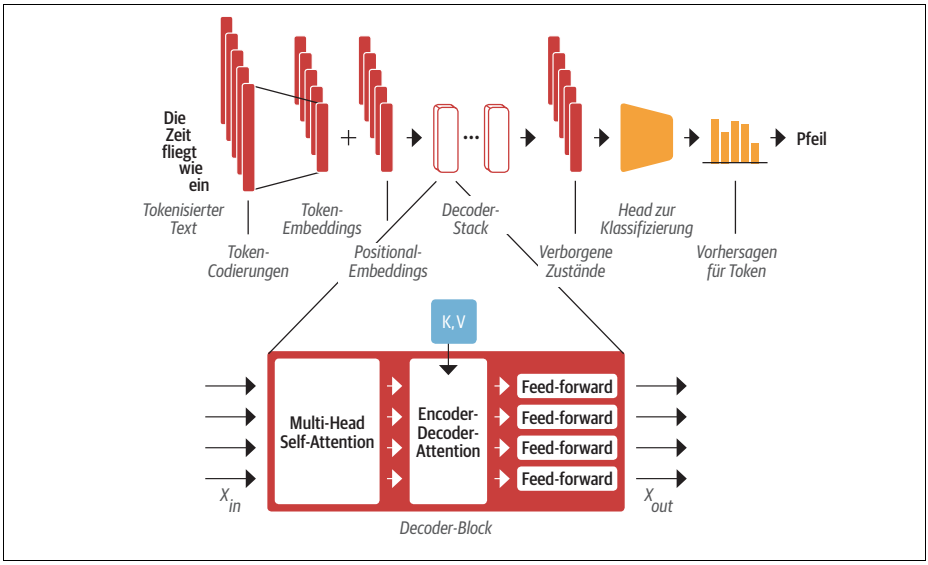


Abbildung 3-7: Nahansicht der Decoder-Schicht des Transformers

Indem wir die Werte oberhalb der Hauptdiagonale auf einen Wert von minus unendlich setzen, gewährleisten wir, dass die Attention-Gewichte alle gleich null sind, sobald wir die Softmax-Funktion auf die Scores anwenden, da $e^{-\infty} = 0$ (rufen Sie sich in Erinnerung, dass es sich bei der Softmax-Funktion um eine normalisierte Exponentialfunktion handelt). Um dieses Maskierungsverhalten zu erzielen, müssen wir lediglich eine kleine Änderung an unserer Funktion zur Berechnung der Attention auf Basis des skalierten Skalarprodukts, die wir zuvor in diesem Kapitel implementiert haben, vornehmen:

```
def scaled_dot_product_attention(query, key, value, mask=None):
    dim_k = query.size(-1)
    scores = torch.bmm(query, key.transpose(1, 2)) / sqrt(dim_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, float("-inf"))
    weights = F.softmax(scores, dim=-1)
    return weights.bmm(value)
```

Von diesem Punkt ausgehend ist es relativ einfach, die Decoder-Schicht aufzubauen. Für weitere Details verweisen wir den Leser auf die hervorragende minGPT-Implementierung (*https://oreil.ly/kwsOP*) von Andrej Karpathy.

Wir haben Ihnen in diesem Kapitel eine ganze Reihe an technischen Informationen gegeben, sodass Sie jetzt ein gutes Verständnis dafür haben sollten, wie jeder Teil

der Transformer-Architektur funktioniert. Bevor wir uns daran machen, Modelle für fortgeschrittenere Aufgaben als die Textklassifizierung zu erstellen, lassen Sie uns zum Abschluss des Kapitels einen Blick auf die Landschaft der verschiedenen Transformer-Modelle werfen und darauf, welche Unterschiede sie zueinander aufweisen.

Encoder-Decoder-Attention einfach erklärt

Versuchen wir, etwas mehr Klarheit in die Sache mit der vielleicht etwas rätselhaften Encoder-Decoder-Attention zu bringen. Stellen Sie sich vor, Sie (der Decoder) sitzen im Unterrichtssaal und schreiben eine Prüfung. Ihre Aufgabe ist es, das nächste Wort auf der Grundlage der vorherigen Wörter (d. h. der Eingaben des Decoders) vorherzusagen. Das hört sich zwar einfach an, ist aber für gewöhnlich unglaublich schwer (versuchen Sie es doch einmal selbst und sagen Sie die nächsten Wörter in einer Passage dieses Buchs voraus). Glücklicherweise hat Ihre Nachbarin (der Encoder) den vollständigen Text. Leider ist sie eine ausländische Austauschstudentin und der Text in ihrer Muttersprache gehalten. Als gewiefte(r) Student(in) finden Sie jedoch einen Weg, um dennoch zu schummeln. Sie zeichnen eine kleine Karikatur, die den Text illustriert, den Sie bereits vorliegen haben (die Abfrage) und geben sie Ihrer Nachbarin. Sie versucht herauszufinden, welche Passage zu dieser Beschreibung passt (der Schlüssel), zeichnet eine Karikatur, die das Wort nach dieser Passage beschreibt (der Wert), und gibt sie Ihnen zurück. Dank dieser Vorgehensweise bestehen Sie die Prüfung mit Bravour.

Transformer-Modelle im Überblick

Wie Sie in diesem Kapitel erfahren haben, gibt es drei grundlegende Architekturen für Transformer-Modelle: Encoder-, Decoder- und Encoder-Decoder-basierte Modelle. Der anfängliche Erfolg der frühen Transformer-Modelle löste eine wahre kambrische Explosion in der Entwicklung von Modellen aus: Forscher entwickelten Modelle auf Basis verschiedener Datensätze unterschiedlicher Größe und Art, verwendeten neue Ansätze im Rahmen des Pretrainings (sogenannte Pretraining-Objectives) und optimierten die Architektur, um die Leistung weiter zu verbessern. Obwohl die Anzahl an Modellen stetig zunimmt,[7] lassen sie sich weiterhin in diese drei grundlegenden Kategorien einteilen.

In diesem Abschnitt geben wir Ihnen einen kurzen Überblick über die wichtigsten Transformer-Modelle der einzelnen Kategorien bzw. Architekturen. Lassen Sie uns zunächst einen Blick auf den »Stammbaum« der Transformer werfen.

7 Anm. d. Übersetzers: Im Englischen spricht man bei einer (Online-)Modellsammlung übrigens von einem »Model Zoo«.

Die drei Entwicklungsstränge von Transformer-Modellen

Im Laufe der Zeit hat jede der drei grundlegenden Architekturen eine eigene Entwicklung durchgemacht. Dies wird klar in Abbildung 3-8 ersichtlich, die Ihnen einen Überblick über einige der bekanntesten Modelle und ihrer »Nachkommen« bietet.

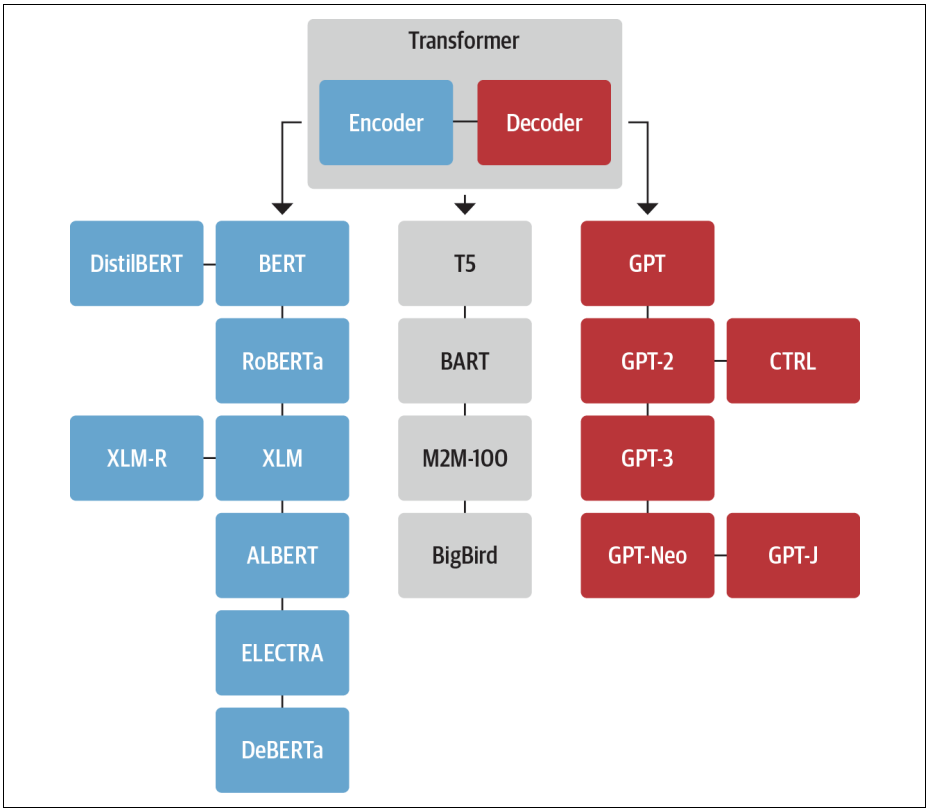


Abbildung 3-8: Ein Überblick über einige der bekanntesten Transformer-Architekturen

Mit über 50 verschiedenen Architekturen, die in der 🤗 Transformers-Bibliothek enthalten sind, bietet dieser »Stammbaum« keineswegs einen vollständigen Überblick über alle existierenden Architekturen: Er hebt lediglich einige der architektonischen Meilensteine hervor. Nachdem wir uns bisher in diesem Kapitel eingehend mit der ursprünglichen Transformer-Architektur befasst haben, sollten wir nun noch einige der wichtigsten »Nachkommen« bzw. Weiterentwicklungen genauer betrachten, wobei wir mit dem Encoder-Branch (bzw. -Zweig), also den rein Encoder-basierten Transformer-Modellen, beginnen.

Rein Encoder-basierte Transformer-Modelle

Das erste reine Encoder-basierte Modell, das auf der Transformer-Architektur beruhte, war BERT. Als es veröffentlicht wurde, übertraf es alle State-of-the-Art-Modelle in der beliebten GLUE-Benchmark[8], anhand der ermittelt wird, wie gut ein Modell im Bereich des Natural Language Understanding (NLU), also des Verstehens natürlicher Sprache, hinsichtlich mehrerer Aufgaben unterschiedlichen Schwierigkeitsgrads abschneidet. In der Folge wurde sowohl das Ziel, das im Rahmen des Pretrainings verfolgt wird, als auch die Architektur von BERT angepasst, um die Leistung weiter zu verbessern. In Forschung und Industrie dominieren bei NLU-Aufgaben wie der Textklassifizierung, der Named Entity Recognition und dem Question Answering nach wie vor rein Encoder-basierte Modelle. Werfen wir nun einen kurzen Blick auf das BERT-Modell und seine Varianten:

BERT

Bei BERT werden im Rahmen des Pretrainings zwei Ziele (Objectives) verfolgt: Einerseits werden maskierte Tokens in Texten vorhergesagt, und andererseits wird bestimmt, ob es wahrscheinlich ist, das eine bestimmte Textstelle auf eine andere folgt.[9] Die erste Aufgabe wird im Englischen *Masked Language Modeling* (MLM, »maskierte Sprachmodellierung«) und die zweite *Next Sentence Prediction* (NSP) genannt.

DistilBERT

Obwohl das BERT-Modell hervorragende Ergebnisse liefert, kann seine Größe es erschweren, es in Umgebungen zu deployen, in denen eine niedrige Latenzzeit erforderlich ist. Durch den Einsatz einer Technik, die während des Pretrainings angewandt und als Knowledge Distillation bekannt ist, erreicht DistilBERT 97 % der Leistung von BERT, obwohl es 40 % weniger Speicher benötigt und 60 % schneller ist.[10] Weitere Einzelheiten zur Knowledge Distillation erfahren Sie in Kapitel 8.

RoBERTa

Eine Studie im Anschluss an die Veröffentlichung von BERT ergab, dass seine Leistung weiter verbessert werden kann, indem das Pretraining anders ausgestaltet wird. RoBERTa wird länger, mit größeren Batches und mehr Trainingsdaten trainiert, und auf die NSP-Aufgabe wird verzichtet.[11] Zusammengenommen führen diese Änderungen zu einer erheblichen Verbesserung der Leistung im Vergleich zum ursprünglichen BERT-Modell.

8 A. Wang et al., »GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding« (*https://arxiv.org/abs/1804.07461*), (2018).

9 J. Devlin et al., »BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding« (*https://arxiv.org/abs/1810.04805*), (2018).

10 V. Sanh et al., »DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter« (*https://arxiv.org/abs/1910.01108*), (2019).

11 Y. Liu et al., »RoBERTa: A Robustly Optimized BERT Pretraining Approach« (*https://arxiv.org/abs/1907.11692*), (2019).

XLM

In dem Beitrag über das Cross-Lingual Language Model (XLM) wurden zum Aufbau mehrsprachiger Modelle mehrere verschiedene Pretraining-Objectives untersucht,[12] einschließlich dem Autoregressive Language Modeling (»autoregressive Sprachmodellierung«) und dem MLM, die bei Modellen wie GPT respektive BERT übernommen wurden. Darüber hinaus haben die Autoren des Forschungsbeitrags *Translation Language Modeling* (TLM) für das Pretraining von XLM eingeführt, das eine Erweiterung von MLM auf mehrsprachige Eingaben darstellt. Bei ihren Experimenten mit den beim Pretraining verfolgten Aufgaben erzielten sie sowohl bei multilingualen NLU-Benchmarks als auch bei Übersetzungsaufgaben Spitzenergebnisse.

XLM-RoBERTa

In Anlehnung an die Beiträge zu dem XLM- und RoBERTa-Modell geht das XLM-RoBERTa- bzw. XLM-R-Modell beim mehrsprachigen Pretraining noch einen Schritt weiter, indem die Menge an Trainingsdaten massiv erhöht wird.[13] Unter Verwendung des Common Crawl Corpus (*https://commoncrawl.org*) erstellten die Entwickler einen 2,5 Terabyte großen Textdatensatz. Auf diesem Datensatz haben sie dann einen Encoder mittels MLM trainiert. Da der Datensatz nur Daten ohne Paralleltexte (also keine Übersetzungen) enthält, wurde das TLM-Ziel, das beim XLM-Modell verfolgt wurde, nicht berücksichtigt. Dieser Ansatz übertrifft XLM und die mehrsprachigen BERT-Varianten bei Weitem, insbesondere bei Sprachen, für die nur wenige Ressourcen zur Verfügung stehen.

ALBERT

Das ALBERT-Modell führte drei Änderungen ein, die die Architektur des Encoders effizienter gestalten sollten.[14] Erstens entkoppelt es die Token-Embedding-Dimension von der verborgenen Dimension, sodass die Embedding-Dimension klein gehalten werden kann und dadurch Parameter eingespart werden können, insbesondere dann, wenn das Vokabular groß werden sollte. Zweitens teilen sich alle Schichten dieselben Parameter, wodurch die Anzahl effektiver Parameter noch weiter verringert wird. Drittens wird das NSP-Ziel durch eine Vorhersage der Satzreihenfolge ersetzt: Das Modell soll vorhersagen, ob die Reihenfolge von zwei aufeinanderfolgenden Sätzen vertauscht wurde oder nicht, anstatt vorherzusagen, ob sie überhaupt zusammengehören. Diese Änderungen ermöglichen es, noch größere Modelle mit einer geringeren Anzahl von Parametern zu trainieren und eine bessere Leistung bei NLU-Aufgaben zu erzielen.

12 G. Lample, and A. Conneau, »Cross-Lingual Language Model Pretraining« (*https://arxiv.org/abs/1901.07291*), (2019).

13 A. Conneau et al., »Unsupervised Cross-Lingual Representation Learning at Scale« (*https://arxiv.org/abs/1911.02116*), (2019).

14 Z. Lan et al., »ALBERT: A Lite BERT for Self-Supervised Learning of Language Representations« (*https://arxiv.org/abs/1909.11942*), (2019).

ELECTRA

Eine Einschränkung des standardmäßigen MLM-Pretraining-Objective besteht darin, dass bei jedem Trainingsschritt nur die Darstellungen der maskierten Tokens aktualisiert werden, während die der restlichen Eingabe-Tokens nicht aktualisiert werden. Um dieses Problem zu lösen, verwendet ELECTRA[15] einen Ansatz, bei dem zwei Modelle zum Einsatz kommen: Das erste Modell (das in der Regel klein ist) arbeitet wie ein standardmäßiges MLM-Modell und sagt maskierte Tokens voraus. Das zweite Modell, der sogenannte *Diskriminator* (engl. Discriminator), hat dann die Aufgabe, vorherzusagen, welche der Tokens in der Ausgabe des ersten Modells ursprünglich maskiert waren. Dementsprechend muss der Diskriminator für jedes Token eine binäre Klassifizierung vornehmen, wodurch sich das Training 30 Mal effizienter gestaltet. Für nachgelagerte Aufgaben (engl. Downstream Tasks) wird der Diskriminator wie ein gewöhnliches BERT-Modell feingetunt.

DeBERTa

Das DeBERTa-Modell führt zwei architektonische Änderungen ein.[16] Zunächst wird jedes Token durch zwei Vektoren dargestellt: einen für den Informationsinhalt (engl. Content), der andere für die relative Position. Durch die Entflechtung des Inhalts der Tokens von ihrer relativen Position können die Self-Attention-Schichten die Abhängigkeit von nahe gelegenen Token-Paaren besser modellieren. Andererseits ist auch die absolute Position eines Worts wichtig, insbesondere für die Decodierung. Aus diesem Grund wird ein Embedding der absoluten Position (Absolute Position Embedding) direkt vor der Softmax-Schicht des Heads zur Token-Decodierung hinzugefügt. DeBERTa war das erste Modell (als Ensemble betrachtet), das die Leistung von Menschen bei der SuperGLUE-Benchmark[17] übertroffen hat, eine schwierigere Version von GLUE, die aus mehreren Teilaufgaben besteht, die dazu dienen, die Leistungsfähigkeit beim Verstehen von Sprache (NLU) zu beurteilen.

Nachdem wir nun einige der wichtigsten rein Encoder-basierten Architekturen vorgestellt haben, können wir uns den rein Decoder-basierten Modellen zuwenden.

Rein Decoder-basierte Transformer-Modelle

Der Fortschritt bei den rein Decoder-basierten Transformer-Modellen wurde weitgehend von OpenAI vorangetrieben. Diese Modelle sind außergewöhnlich gut darin, das nächste Wort in einer Sequenz vorherzusagen. Sie werden daher haupt-

15 K. Clark et al., »ELECTRA: Pre-Training Text Encoders as Discriminators Rather Than Generators« (*https://arxiv.org/abs/2003.10555*), (2020).

16 P. He et al., »DeBERTa: Decoding-Enhanced BERT with Disentangled Attention« (*https://arxiv.org/abs/2006.03654*), (2020).

17 A. Wang et al., »SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems« (*https://arxiv.org/abs/1905.00537*), 2019).

sächlich für Aufgaben im Rahmen der Textgenerierung verwendet (siehe Kapitel 5 für weitere Einzelheiten). Ihr Fortschritt beruht vor allem darauf, dass immer größere Datensätze verwendet werden und die Größe der Sprachmodelle immer weiter nach oben geschraubt wird. Werfen wir einen genaueren Blick auf die Entwicklung dieser faszinierenden Generierungsmodelle:

GPT

Mit der Einführung von GPT wurden zwei Schlüsselideen im NLP miteinander kombiniert: die neuartige und effiziente Decoder-basierte Transformer-Architektur und das Transfer Learning[18]. Dabei wurde das Modell dadurch vortrainiert, dass das nächste Wort auf der Grundlage der vorherigen Wörter vorhergesagt wurde. Das Modell wurde mit dem BookCorpus-Datensatz trainiert und erzielte bei nachgelagerten Aufgaben wie der Klassifizierung hervorragende Ergebnisse.

GPT-2

Inspiriert durch den Erfolg des einfachen und skalierbaren Pretraining-Ansatzes wurden das ursprüngliche Modell und der Trainingsdatensatz vergrößert, wodurch GPT-2 entstand.[19] Dieses Modell ist in der Lage, lange zusammenhängende Textsequenzen zu generieren. Aus Sorge vor möglichem Missbrauch wurde das Modell nur sukzessive freigegeben, wobei zunächst kleinere Modell-Versionen und erst später das vollständige Modell veröffentlicht wurden.

CTRL

Modelle wie GPT-2 können eine vorgegebene Eingabesequenz (die im Englischen als *Prompt* bezeichnet wird) fortsetzen. Allerdings hat der Benutzer nur begrenzte Möglichkeiten, den Stil der generierten Sequenz zu beeinflussen. Beim Conditional-Transformer-Language-(CTRL-)Modell wird dieses Problem behoben, indem sogenannte Control-Tokens am Anfang einer Sequenz hinzugefügt werden.[20] Diese ermöglichen es, den Stil des generierten Texts zu steuern, wodurch sich vielfältige Möglichkeiten der Generierung eröffnen.

GPT-3

Nachdem sich die Skalierung von GPT auf GPT-2 als erfolgreich erwiesen hatte, ergab eine gründliche Analyse des Verhaltens von Sprachmodellen verschiedener Größenordnungen, dass es einfache Potenzgesetze gibt, die das Verhältnis zwischen der Rechenleistung, der Größe des Datensatzes, der Modellgröße und der Leistung eines Sprachmodells bestimmen.[21] Inspiriert von diesen Erkenntnissen wurde GPT-2 um den Faktor 100 skaliert, woraufhin

18 A. Radford et al., »Improving Language Understanding by Generative Pre-Training« (*https://openai.com/blog/language-unsupervised*), OpenAI (2018).

19 A. Radford et al., »Language Models Are Unsupervised Multitask Learners« (*https://openai.com/blog/better-language-models*), OpenAI (2019).

20 N.S. Keskar et al., »CTRL: A Conditional Transformer Language Model for Controllable Generation« (*https://arxiv.org/abs/1909.05858*), (2019)

21 J. Kaplan et al., »Scaling Laws for Neural Language Models« (*https://arxiv.org/abs/2001.08361*), (2020).

das 175 Milliarden Parameter umfassende GPT-3-Modell entstand.[22] Das Modell ist nicht nur in der Lage, beeindruckend realistische Textpassagen zu generieren, sondern zeigt außerdem sogenannte »Few-Shot«-Lernfähigkeiten: Mit nur einigen wenigen Beispielen (»few Shots«) einer neuartigen Aufgabe, wie z. B. Text in Code zu übertragen, ist das Modell in der Lage, die entsprechende Aufgabe auf neue Beispiele anzuwenden. OpenAI hat dieses Modell nicht frei zugänglich gemacht, bietet aber eine Schnittstelle über die OpenAI-API (*https://oreil.ly/SEGRW*) an.

GPT-Neo/GPT-J-6B

GPT-Neo und GPT-J-6B sind GPT-artige Modelle, die von EleutherAI (*https://eleuther.ai*) trainiert wurden – einem Kollektiv von Forschern, das sich zum Ziel gesetzt hat, dem Maßstab von GPT-3 gleichkommende Modelle neu zu erstellen und zu veröffentlichen.[23] Bei den aktuellen Modellen handelt es sich um kleinere Varianten des vollständigen 175-Milliarden-Parameter-Modells mit je 1,3, 2,7 und 6 Milliarden Parametern, die mit den kleineren GPT-3-Modellen von OpenAI konkurrieren können.[24]

Der letzte »Zweig« im Stammbaum der Transformer sind die Encoder-Decoder-basierten Modelle. Werfen wir nun einen genaueren Blick auf sie.

Encoder-Decoder-basierte Transformer-Modelle

Obwohl es üblich geworden ist, Modelle mit einem einzelnen Encoder- oder Decoder-Stack zu erstellen, gibt es mehrere Encoder-Decoder-basierte Varianten der Transformer-Architektur, die sowohl im Bereich des NLU als auch der Natural Language Generation (NLG) neuartige Anwendungsmöglichkeiten bieten:

T5

Das T5-Modell vereinheitlicht alle NLU- und NLG-Aufgaben, indem es sie in Text-to-Text-Aufgaben überführt.[25] Alle Aufgaben sind als Sequence-to-Sequence-Aufgaben formuliert, weshalb die Verwendung einer Encoder-Decoder-Architektur naheliegend ist. Für Aufgaben im Rahmen der Textklassifizierung bedeutet dies zum Beispiel, dass der Text als Eingabe für den Encoder verwendet wird und der Decoder das Label in Form von normalem Text anstelle einer Kategorie generieren muss. In Kapitel 6 werden wir hierzu noch weiter ins Detail gehen. Die Architektur des T5-Modells entspricht der ursprünglichen

22 T. Brown et al., »Language Models Are Few-Shot Learners« (*https://arxiv.org/abs/2005.14165*), (2020).

23 S. Black et al., »GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-TensorFlow« (*https://doi.org/10.5281/zenodo.5297715*), (2021); B. Wang und A. Komatsuzaki, »GPT-J-6B: A 6 Billion Parameter Autoregressive Language-Modell« (*https://github.com/kingoflolz/mesh-transformer-jax*), (2021).

24 Anm. d. Übersetzers: Im Februar 2022 wurde zudem noch GPT-NeoX veröffentlicht, das 20 Milliarden Parameter umfasst.

25 C. Raffel et al., »Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer« (*https://arxiv.org/abs/1910.10683*), (2019).

Transformer-Architektur. Unter Verwendung des großen gecrawlten C4-Datensatzes wurde das Modell sowohl mittels Masked Language Modeling als auch auf Basis der SuperGLUE-Aufgaben vortrainiert, indem alle diese Aufgaben in Text-to-Text-Aufgaben überführt wurden. Das mit 11 Milliarden Parametern größte Modell lieferte bei mehreren Benchmarks Spitzenergebnisse.

BART

BART kombiniert die Pretraining-Verfahren von BERT und GPT auf Basis einer Encoder-Decoder-basierten Architektur.[26] Die Eingabesequenzen werden einer von mehreren möglichen Transformationen unterzogen, von einfacher Maskierung bis hin zur Permutation von Sätzen, dem Löschen von Tokens und dem Vertauschen von Dokumenten. Diese veränderten Eingaben werden durch den Encoder geleitet, und der Decoder muss die ursprünglichen Texte rekonstruieren. Dies verleiht dem Modell mehr Flexibilität, da es sowohl für NLU- als auch für NLG-Aufgaben verwendet werden kann und auf beiden Gebieten eine Spitzenleistung erzielt.

M2M-100

Normalerweise wird ein Übersetzungsmodell für ein Sprachpaar und eine Übersetzungsrichtung entwickelt. Das lässt sich allerdings nicht auf allzu viele Sprachen übertragen, und außerdem könnte es zwischen Sprachpaaren Gemeinsamkeiten geben, die für die Übersetzung zwischen seltener genutzten Sprachen verwendet werden könnten. M2M-100 ist das erste Übersetzungsmodell, das zwischen 100 verschiedenen Sprachen übersetzen kann.[27] Dies ermöglicht qualitativ hochwertige Übersetzungen zwischen seltenen bzw. unterrepräsentierten Sprachen. Das Modell verwendet Präfix-Tokens (ähnlich dem speziellen `[CLS]`-Token), mit denen jeweils gekennzeichnet wird, ob es sich um die Ausgangs- oder Zielsprache handelt.

BigBird

Eine wesentliche Einschränkung von Transformer-Modellen ist die maximale Kontextlänge (engl. Context Size), die sich dadurch ergibt, dass der Attention-Mechanismus einen sich hinsichtlich der Erhöhung der Kontextlänge quadrierenden Speicherbedarf zur Folge hat. Bei BigBird wird diesem Problem begegnet, indem eine »sparsamere« Form der Attention (d.h. eine dünnbesetzte Matrix) verwendet wird, die linear skaliert.[28] Dadurch konnte die Kontextlänge von 512 Tokens, die für die Mehrzahl von BERT-Modellen gilt, auf 4.096 Tokens im BigBird-Modell drastisch erhöht werden. Dies ist besonders nützlich in Fällen, in denen Abhängigkeiten über längere Passagen erhalten bleiben müssen, wie z.B. bei der Textzusammenfassung.

---

26 M. Lewis et al., »BART: Denoising Sequence-to-Sequence Pre-Training for Natural Language Generation, Translation, and Comprehension« (*https://arxiv.org/abs/1910.13461*), (2019).

27 A. Fan et al., »Beyond English-Centric Multilingual Machine Translation« (*https://arxiv.org/abs/2010.11125*), (2020).

28 M. Zaheer et al., »Big Bird: Transformers for Longer Sequences« (*https://arxiv.org/abs/2007.14062*), (2020).

Für alle Modelle, die wir in diesem Abschnitt vorgestellt haben, stehen vortrainierte Checkpoints auf dem Hugging Face Hub (*https://oreil.ly/EIOrN*) zur Verfügung. Diese können Sie – wie im vorherigen Kapitel beschrieben – mithilfe der 🤗 Transformers-Bibliothek für Ihren Anwendungsfall feintunen.

Zusammenfassung

In diesem Kapitel haben wir mit dem Herzstück der Transformer-Architektur begonnen und uns eingehend mit der Self-Attention beschäftigt. Anschließend haben wir alle notwendigen Bausteine hinzugefügt, um ein rein Encoder-basiertes Transformer-Modell zu erstellen. Wir haben Embedding-Schichten für Tokens und Positionsinformationen hinzugefügt, wir haben eine Feed-Forward-Schicht integriert, die die Attention-Heads ergänzt, und schließlich haben wir dem Body des Modells einen Head zur Klassifizierung hinzugefügt, der dafür zuständig ist, die Vorhersagen zu treffen. Außerdem haben wir den Decoder der Transformer-Architektur genauer betrachtet und das Kapitel mit einem Überblick über die wichtigsten Modellarchitekturen abgerundet.

Da Sie nun die zugrunde liegenden Prinzipien besser verstehen, können wir die einfache Klassifizierung hinter uns lassen und ein mehrsprachiges Modell zur Erkennung von Eigennamen bzw. benannten Entitäten erstellen.

KAPITEL 4

Multilinguale Named Entity Recognition

Bislang haben wir in diesem Buch Transformer-Modelle dazu verwendet, NLP-Aufgaben, die auf englischsprachigen Korpora basieren, zu lösen. Was aber, wenn Ihre Dokumente auf Griechisch, Swahili oder Klingonisch verfasst sind? Eine Möglichkeit besteht darin, im Hugging Face Hub nach einem geeigneten vortrainierten Sprachmodell zu suchen und es für die jeweilige Aufgabe feinzutunen. Diese vortrainierten Modelle gibt es jedoch in der Regel nur für Sprachen, für die viele Ressourcen zur Verfügung stehen, wie Deutsch, Russisch oder Mandarin, für die jede Menge an Texten für das Pretraining im Internet zur Verfügung steht. Eine weitere häufige Herausforderung ergibt sich, wenn Ihr Korpus mehr als eine Sprache umfasst: Mehrere einsprachige bzw. monolinguale Modelle in der Produktion zu warten, ist weder für Sie noch für Ihr Entwicklungsteam ein Vergnügen.

Glücklicherweise gibt es eine Gruppe von Transformer-Modellen, die mehrere Sprachen abdecken (engl. Multilingual Transformer) und hier Abhilfe schaffen. Wie BERT werden diese Modelle mithilfe des Masked Language Modeling vortrainiert, allerdings auf Texten in über hundert Sprachen gleichzeitig. Durch das Pretraining auf riesigen Korpora in einer Vielzahl von Sprachen ermöglichen diese multilingualen Transformer-Modelle einen sogenannten *Zero-Shot Cross-Lingual Transfer*. Das bedeutet, dass ein in einer Sprache feingetuntes Modell ohne weiteres Training auf andere Sprachen übertragen werden kann! Dadurch eignen sich die Modelle auch gut für das »Code-Switching«, bei dem ein Sprecher im Rahmen einer einzigen Konversation zwischen zwei oder mehr Sprachen oder Dialekten wechselt.

In diesem Kapitel werden wir untersuchen, wie ein einzelnes Transformer-Modell namens XLM-RoBERTa (das wir bereits in Kapitel 3 vorgestellt haben)[1] feingetunt werden kann, um eine Named Entity Recognition (NER) in mehreren Sprachen durchzuführen. Wie wir in Kapitel 1 gesehen haben, ist NER eine gängige NLP-Aufgabe, bei der Entitäten wie Personen, Organisationen oder Orte in Texten identifiziert werden. Diese Entitäten können für verschiedene Anwendungen, wie z.B. um Erkenntnisse aus Unternehmensdokumenten zu gewinnen, die Qualität von

1 A. Conneau et al., »Unsupervised Cross-Lingual Representation Learning at Scale« (*https://arxiv.org/abs/1911.02116*), (2019).

Suchmaschinen zu verbessern oder einfach eine strukturierte Datenbank aus einem Korpus aufzubauen, genutzt werden.

In diesem Kapitel gehen wir davon aus, dass wir eine Named Entity Recognition (NER) für einen Kunden durchführen möchten, der in der Schweiz ansässig ist – einem Land, in dem es vier Landessprachen gibt (wobei Englisch oft als »Brücke« zwischen ihnen dient). Als Erstes benötigen wir ein geeignetes mehrsprachiges Korpus, mit dem wir diese Aufgabe angehen können.

Der Begriff *Zero-Shot-Transfer* bzw. *Zero-Shot-Learning* bezieht sich in der Regel auf die Aufgabe, ein Modell mit einer Reihe von Labels zu trainieren und es dann auf einer Reihe anderer Labels zu evaluieren. Im Zusammenhang mit Transformern kann sich Zero-Shot-Learning auch darauf beziehen, dass ein Sprachmodell wie GPT-3 auf Basis einer nachgelagerten Aufgabe evaluiert wird, für die es nicht einmal feingetunt wurde.

Der Datensatz

In diesem Kapitel werden wir einen Teildatensatz des Cross-lingual TRansfer Evaluation of Multilingual Encoders (XTREME)-Benchmarks namens WikiANN bzw. PAN-X verwenden.[2] Dieser Datensatz besteht aus Wikipedia-Artikeln, die in einer Vielzahl verschiedener Sprachen verfasst wurden, darunter die vier in der Schweiz am häufigsten gesprochenen Sprachen: Deutsch (62,9 %), Französisch (22,9 %), Italienisch (8,4 %) und Englisch (5,9 %). Jeder Artikel ist mit den Tags `LOC` (engl. Location, Ort), `PER` (Person) und `ORG` (engl. Organization, Organisation) dem »Inside-Outside-Beginning« (IOB2)-Format (*https://oreil.ly/yXMUn*) entsprechend annotiert. Bei diesem Format kennzeichnet ein `B-`-Präfix den Anfang einer Entität. Aufeinanderfolgende Tokens, die zur selben Entität gehören, erhalten ein `I-`-Präfix. Ein `O` zeigt an, dass das Token keiner Entität angehört. Der folgende Satz zum Beispiel:

Jeff Dean is a computer scientist at Google in California

würde im IOB2-Format wie in Tabelle 4-1 gezeigt gelabelt sein.

Tabelle 4-1: Ein Beispiel für eine mit benannten Entitäten (Eigennamen) annotierte Sequenz

Tokens	Jeff	Dean	is	a	computer	scientist	at	Google	in	California
Tags	B-PER	I-PER	O	O	O	O	O	B-ORG	O	B-LOC

Um einen der in XTREME enthaltenen PAN-X-Teildatensätze zu laden, müssen wir wissen, welche *Datensatzkonfiguration* wir der Funktion `load_dataset()` übergeben müssen. Wenn Sie es mit einem Datensatz zu tun haben, der mehrere Domä-

2 J. Hu et al, »XTREME: A Massively Multilingual Multi-Task Benchmark for Evaluating Cross-Lingual Generalization« (*https://arxiv.org/abs/2003.11080*), (2020); X. Pan et al., »Cross-Lingual Name Tagging and Linking for 282 Languages,« *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics* 1 (Juli 2017): 1946–1958, *http://dx.doi.org/10.18653/v1/P17-1178*.

nen umfasst, können Sie die Funktion `get_dataset_config_names()` verwenden, um herauszufinden, welche Teildatensätze verfügbar sind:

```
from datasets import get_dataset_config_names

xtreme_subsets = get_dataset_config_names("xtreme")
print(f"XTREME hat {len(xtreme_subsets)} Konfigurationen")
```

```
XTREME hat 183 Konfigurationen
```

Hui, das sind eine Menge Konfigurationen! Schränken wir die Suche ein, indem wir nur nach den Konfigurationen suchen, die mit »PAN« beginnen:

```
panx_subsets = [s for s in xtreme_subsets if s.startswith("PAN")]
panx_subsets[:3]
```

```
['PAN-X.af', 'PAN-X.ar', 'PAN-X.bg']
```

Nun gut, offenbar liegt der Syntax der PAN-X-Teildatensätze ein gewisses Schema zugrunde: Jeder hat ein Suffix aus zwei Buchstaben, bei denen es sich offenbar um ISO-639-1-Sprachcodes (*https://oreil.ly/R8XNu*) handelt. Das bedeutet, dass wir zum Laden des deutschen Korpus lediglich `de` als Code für das Argument `name` der `load_dataset()`-Funktion angeben müssen:

```
from datasets import load_dataset

load_dataset("xtreme", name="PAN-X.de")
```

Um ein realistisches Schweizer Korpus zu erstellen, werden wir eine Stichprobe erstellen, in der wir jeweils das deutsche (`de`), französische (`fr`), italienische (`it`) und englische (`en`) PAN-X-Korpus entsprechend ihres Anteils an der gesprochenen Sprache berücksichtigen. Auf diese Weise schaffen wir eine Unausgewogenheit (engl. Imbalance) zwischen den Sprachen, wie sie sehr häufig auch in realen Datensätzen anzutreffen ist. Beispiele für eine wenig verbreitete Sprache zu beschaffen, kann kostspielig sein, da es einen großen Mangel an Experten gibt, die diese Sprache fließend beherrschen. Mit diesem unausgewogenen Datensatz simulieren wir eine Situation, die bei der Arbeit an mehrsprachigen Anwendungen häufig auftritt. Wir werden sehen, wie wir ein Modell aufbauen können, das in all diesen Sprachen funktioniert.

Damit wir den Überblick über die einzelnen Sprachen behalten, können wir zunächst ein Python-`defaultdict` erstellen, das den Sprachcode als Schlüssel und ein PAN-X-Korpus vom Typ `DatasetDict` als Wert speichert:

```
from collections import defaultdict
from datasets import DatasetDict

langs = ["de", "fr", "it", "en"]
fracs = [0.629, 0.229, 0.084, 0.059]
# Gibt ein DatasetDict zurück, wenn ein Schlüssel nicht existiert
panx_ch = defaultdict(DatasetDict)

for lang, frac in zip(langs, fracs):
    # Einsprachiges Korpus laden
```

```
ds = load_dataset("xtreme", name=f"PAN-X.{lang}")
# Durchmischen und "Downsampeln" jedes Teildatensatzes entsprechend
#  des Anteils an der gesprochenen Sprache
for split in ds:
    panx_ch[lang][split] = (
        ds[split]
        .shuffle(seed=0)
        .select(range(int(frac * ds[split].num_rows))))
```

Hier haben wir die Methode `shuffle()` verwendet, um sicherzustellen, dass wir die Aufteilung unserer Teildatensätze nicht versehentlich verfälschen. Mithilfe der Methode `select()` können wir die einzelnen Korpora entsprechend den (relativen) Werten in `fracs` »downsamplen«, d.h., dass wir die Korpora in unterschiedlichen Verhältnissen einbeziehen, indem wir jeweils eine den relativen Anteilen entsprechend große Stichprobe aus den Korpora ziehen. Prüfen wir, wie viele Beispiele wir pro Sprache in den Trainingsdatensätzen haben, indem wir auf das Attribut `Dataset.num_rows` zugreifen:

```
import pandas as pd

pd.DataFrame({lang: [panx_ch[lang]["train"].num_rows] for lang in langs},
             index=["Anzahl an Trainingsbeispielen"])
```

	de	fr	it	en
Anzahl an Trainingsbeispielen	12580	4580	1680	1180

Da wir mehr Beispiele auf Deutsch als in allen anderen Sprachen zusammen haben, nehmen wir diese Sprache für unsere sprachenübergreifende Übertragung (Zero-Shot Cross-Lingual Transfer) auf Französisch, Italienisch und Englisch als Ausgangspunkt. Sehen wir uns eines der Beispiele aus dem deutschen Korpus an.

```
element = panx_ch["de"]["train"][0]
for key, value in element.items():
    print(f"{key}: {value}")
```

```
langs: ['de', 'de', 'de', 'de', 'de', 'de', 'de', 'de', 'de', 'de', 'de', 'de']
ner_tags: [0, 0, 0, 0, 5, 6, 0, 0, 5, 5, 6, 0]
tokens: ['2.000', 'Einwohnern', 'an', 'der', 'Danziger', 'Bucht', 'in', 'der',
'polnischen', 'Woiwodschaft', 'Pommern', '.']
```

Wie schon bei den zuvor betrachteten `Dataset`-Objekten entsprechen die Schlüssel unseres Beispiels den Spaltennamen einer Arrow-Tabelle und die Werte den Einträgen jeder Spalte. In der Spalte `ner_tags` wird offensichtlich jeder Entität eine ID für die Kategorie zugeordnet. Da diese Form der Darstellung für das menschliche Auge ein wenig kryptisch ist, erstellen wir eine neue Spalte, in der stattdessen die uns bereits bekannten Tags `LOC`, `PER` und `ORG` angegeben werden. Hierfür können wir zunächst das Attribut `features` unseres `Dataset`-Objekts verwenden, um die jeder Spalte zugrunde liegenden Datentypen zu erhalten:

```
for key, value in panx_ch["de"]["train"].features.items():
    print(f"{key}: {value}")
```

```
tokens: Sequence(feature=Value(dtype='string', id=None), length=-1, id=None)
ner_tags: Sequence(feature=ClassLabel(num_classes=7, names=['O', 'B-PER',
'I-PER', 'B-ORG', 'I-ORG', 'B-LOC', 'I-LOC'], names_file=None, id=None),
length=-1, id=None)
langs: Sequence(feature=Value(dtype='string', id=None), length=-1, id=None)
```

Die `Sequence`-Klasse gibt an, dass das Feld eine Liste von Features enthält, was im Fall von `ner_tags` einer Liste von `ClassLabel`-Features entspricht. Wählen wir diese Features wie folgt aus dem Trainingsdatensatz aus:

```
tags = panx_ch["de"]["train"].features["ner_tags"].feature
print(tags)
```

```
ClassLabel(num_classes=7, names=['O', 'B-PER', 'I-PER', 'B-ORG', 'I-ORG',
'B-LOC', 'I-LOC'], names_file=None, id=None)
```

Mithilfe der in Kapitel 2 vorgestellten Methode `ClassLabel.int2str()` können wir eine neue Spalte in unserem Trainingsdatensatz erzeugen, die die Namen der Kategorien der einzelnen Tags enthält. Über die Methode `map()` erhalten wir ein Dictionary, dessen Schlüssel dem Namen der neuen Spalte entspricht und dessen Wert eine Liste ist, die die Namen der Kategorien enthält:

```
def create_tag_names(batch):
    return {"ner_tags_str": [tags.int2str(idx) for idx in batch["ner_tags"]]}

panx_de = panx_ch["de"].map(create_tag_names)
```

Nachdem wir nun unsere Tags in einem für Menschen lesbaren Format vorliegen haben, können wir anhand des ersten Beispiels im Trainingsdatensatz nachvollziehen, welche Tags für die verschiedenen Tokens angelegt wurden:

```
de_example = panx_de["train"][0]
pd.DataFrame([de_example["tokens"], de_example["ner_tags_str"]],
['Tokens', 'Tags'])
```

	0	1	2	3	4	5	6	7	8	9	10	11
Tokens	2.000	Einwohnern	an	der	Danziger	Bucht	in	der	polnischen	Woiwodschaft	Pommern	.
Tags	O	O	O	O	B-LOC	I-LOC	O	O	B-LOC	B-LOC	I-LOC	O

Die `LOC`-Tags, die für Ortsangaben genutzt werden, sind einleuchtend: Die Danziger Bucht ist eine Bucht an der Ostsee, wohingegen es sich bei »Woiwodschaft« um einen polnischen Verwaltungsbezirk handelt. Um sicherzustellen, dass die Tags nicht zu unausgewogen verteilt sind, können wir ermitteln, wie häufig die einzelnen Entitäten in jedem Teildatensatz vorkommen:

```
from collections import Counter

split2freqs = defaultdict(Counter)
for split, dataset in panx_de.items():
    for row in dataset["ner_tags_str"]:
        for tag in row:
```

```
            if tag.startswith("B"):
                tag_type = tag.split("-")[1]
                split2freqs[split][tag_type] += 1
pd.DataFrame.from_dict(split2freqs, orient="index")
```

	ORG	LOC	PER
Validierungsdatensatz	2683	3172	2893
Testdatensatz	2573	3180	3071
Trainingsdatensatz	5366	6186	5810

Das sieht gut aus – die Häufigkeitsverteilung von `PER`, `LOC` und `ORG` ist in jedem Teildatensatz in etwa gleich, sodass der Validierungs- und der Testdatensatz einen guten Maßstab dafür bieten sollten, wie gut unserer NER-Tagger generalisiert. Werfen wir als Nächstes einen Blick auf einige beliebte mehrsprachige Transformer-Modelle und wie diese auf unsere NER-Aufgabe übertragen und verwendet werden können.

Multilinguale Transformer-Modelle

Mehrsprachige bzw. multilinguale Transformer (engl. Multilingual Transformers) verwenden ähnliche Architekturen und Trainingsverfahren wie ihre einsprachigen Gegenstücke, mit dem Unterschied, dass das zum Pretraining verwendete Korpus aus Dokumenten in mehreren Sprachen besteht. Das Bemerkenswerte an diesem Ansatz ist, dass die resultierenden linguistischen Darstellungen – obwohl sie keine expliziten Informationen zur Unterscheidung zwischen den Sprachen erhalten – in der Lage sind, gut auf eine Vielzahl von nachgelagerten Aufgaben sprachenübergreifend zu generalisieren. In einigen Fällen kann diese Fähigkeit zum sprachenübergreifenden Transfer zu Ergebnissen führen, die mit denen einsprachiger Modelle konkurrieren können, wodurch sich vermeiden lässt, für jede Sprache ein eigenes Modell trainieren zu müssen.

Zur Bewertung des Fortschritts des sprachenübergreifenden Transfers bei NER-Aufgaben werden für die Sprachen Englisch, Niederländisch, Spanisch und Deutsch als Benchmark oftmals die CoNLL-2002- (*https://oreil.ly/nYd0o*) und CoNLL-2003-Datensätze (*https://oreil.ly/sVESv*) verwendet. Diese Benchmark besteht aus Nachrichtenartikeln, die mit denselben Kategorien wie PAN-X annotiert sind (`LOC`, `PER` und `ORG`), enthält jedoch ein zusätzliches `MISC`-Label, das für sonstige (engl. miscellaneous) Entitäten steht, die nicht zu den genannten drei Kategorien gehören. Mehrsprachige Transformer-Modelle werden in der Regel auf drei verschiedene Arten evaluiert:

en
: Das Feintuning wird mit den englischsprachigen Trainingsdaten durchgeführt und dann anhand der Testdatensätze der einzelnen Sprachen evaluiert.

`each`
: Das Feintuning und die Evaluierung erfolgen auf Basis der einsprachigen Trainings- bzw. Testdatensätze, um die Leistung für jede Sprache zu evaluieren.

`all`
: Das Feintuning wird mit den Trainingsdaten aller Sprachen und die Evaluierung mit allen Testdatensätzen der jeweiligen Sprachen durchgeführt.

Für unsere NER-Aufgabe werden wir im Rahmen der Evaluierung eine ähnliche Strategie verfolgen. Allerdings müssen wir zunächst klären, welches Modell wir evaluieren möchten. Eines der ersten mehrsprachigen Transformer-Modelle war mBERT. Es basiert auf der gleichen Architektur und der gleichen Zielsetzung beim Pretraining wie BERT, allerdings wurde das Pretraining-Korpus um Wikipedia-Artikel aus einer Vielzahl von Sprachen erweitert. Inzwischen wurde mBERT von XLM-RoBERTa (oder kurz XLM-R) abgelöst, sodass wir uns in diesem Kapitel mit ebendiesem Modell befassen werden.

Wie wir in Kapitel 3 gesehen haben, wird das XLM-R-Modell lediglich mittels MLM für 100 verschiedene Sprachen vortrainiert. Allerdings zeichnet sich XLM-R durch die enorme Größe seines Pretraining-Korpus im Vergleich zu seinen Vorgängern aus: Der Pretraining-Datensatz besteht aus den Wikipedia-Dumps, die für jede Sprache verfügbar sind, und 2,5 *Terabyte* an Common-Crawl-Daten aus dem Internet. Das Korpus ist dementsprechend um ein Vielfaches größer als die in den vorherigen Modellen verwendeten und bietet eine erhebliche Signalverstärkung für Sprachen wie Birmanisch und Suaheli, für die es nur wenig Ressourcen gibt bzw. nur eine geringe Anzahl von Wikipedia-Artikeln existiert.

Der Teil RoBERTa im Namen des Modells bezieht sich auf die Tatsache, dass der Pretraining-Ansatz derselbe ist wie bei den einsprachigen RoBERTa-Modellen. Die Entwickler von RoBERTa haben mehrere Aspekte von BERT verbessert, insbesondere indem sie die Aufgabe der Vorhersage des nächsten Satzes (NSP) ganz weggelassen haben.[3] Bei XLM-R wird auch auf die in XLM verwendeten sprachenspezifischen Embeddings verzichtet und SentencePiece verwendet, weshalb die Tokenisierung der Rohtexte direkt vorgenommen wird.[4] Abgesehen davon, dass es mehrere Sprachen beherrscht, unterscheiden sich XLM-R und RoBERTa vor allem in der Größe des jeweiligen Vokabulars – statt nur 55.000 umfasst es ganze 250.000 Tokens!

XLM-R eignet sich hervorragend für mehrsprachige NLU-Aufgaben. Im nächsten Abschnitt werden wir untersuchen, wie die Tokenisierung über viele Sprachen hinweg effizient durchgeführt werden kann.

3 Y. Liu et al., »RoBERTa: A Robustly Optimized BERT Pretraining Approach« (*https://arxiv.org/abs/1907.11692*), (2019).

4 T. Kudo und J. Richardson, »SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing« (*https://arxiv.org/abs/1808.06226*), (2018).

Ein genauerer Blick auf die Tokenisierung

Anstelle eines WordPiece-Tokenizers verwendet XLM-R einen Tokenizer namens SentencePiece, der auf der Grundlage des Rohtexts aller einhundert Sprachen trainiert wurde. Um ein Gefühl dafür zu bekommen, wie sich SentencePiece von WordPiece unterscheidet, können wir jeweils den Tokenizer von BERT und XLM-R wie gewohnt mithilfe der 🤗 Transformers-Bibliothek laden:

```
from transformers import AutoTokenizer

bert_model_name = "bert-base-cased"
xlmr_model_name = "xlm-roberta-base"
bert_tokenizer = AutoTokenizer.from_pretrained(bert_model_name)
xlmr_tokenizer = AutoTokenizer.from_pretrained(xlmr_model_name)
```

Codieren wir zunächst eine kleine Textsequenz, um die speziellen Tokens abzurufen, die von den einzelnen Modellen während des Pretrainings verwendet wurden:

```
text = "Jack Sparrow loves New York!"
bert_tokens = bert_tokenizer(text).tokens()
xlmr_tokens = xlmr_tokenizer(text).tokens()
```

BERT	[CLS]	Jack	Spa	##rrow	loves	New	York	!	[SEP]	None
XLM-R	<s>	_Jack	_Spar	row	_love	s	_New	_York	!	</s>

Wie Sie sehen, verwendet XLM-R anstelle der von BERT zur Klassifizierung von Sätzen genutzten Tokens `[CLS]` und `[SEP]` die Tokens `<s>` und `<\s>`, um den Anfang und das Ende einer Sequenz zu kennzeichnen. Wie Sie gleich sehen werden, werden diese Tokens im letzten Schritt der Tokenisierung hinzugefügt.

Die Tokenizer-Pipeline

Bislang haben wir die Tokenisierung als einen einzelnen Vorgang betrachtet, bei dem Strings bzw. Zeichenketten in Ganzzahlen konvertiert werden, die wir anschließend durch das Modell leiten können. Das ist jedoch nicht ganz richtig. Bei genauerer Betrachtung wird deutlich, dass es sich eigentlich um eine vollständige Verarbeitungspipeline (engl. Processing Pipeline) handelt, die normalerweise aus vier Schritten besteht (siehe Abbildung 4-1).

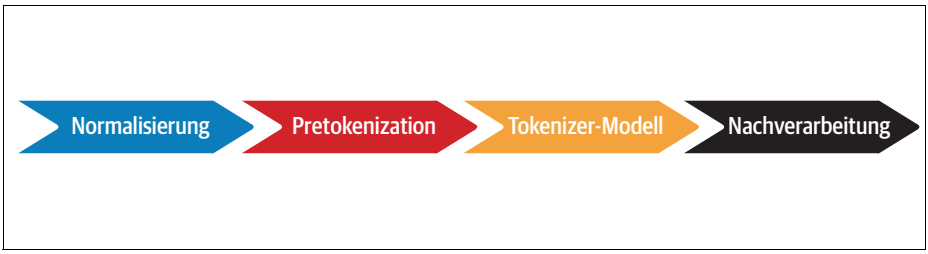


Abbildung 4-1: Die Schritte in der Tokenisierungspipeline

Nehmen wir nun die einzelnen Verarbeitungsschritte genauer unter die Lupe und veranschaulichen anhand des Beispielsatzes »Jack Sparrow loves New York!«, was sie bewirken:

Normalisierung (engl. Normalization)

Dieser Schritt entspricht einer Reihe von Operationen, die Sie auf die unveränderten Strings anwenden können, um sie zu bereinigen. Dazu gehört für gewöhnlich, dass sowohl Leer- als auch Akzentzeichen entfernt werden. Die Unicode-Normalisierung (*https://oreil.ly/2cp3w*) ist eine weitere gängige Operation zur Normalisierung, die von vielen Tokenizern angewandt wird, um der Tatsache Rechnung zu tragen, dass es oft unterschiedliche Möglichkeiten gibt, ein und dasselbe Zeichen zu schreiben bzw. darzustellen. Dies kann dazu führen, dass zwei Varianten des »gleichen« Strings (d. h. mit der gleichen Abfolge von abstrakten Zeichen) unterschiedlich erscheinen. Unicode-Normalisierungsschemata wie NFC, NFD, NFKC und NFKD ersetzen die verschiedenen Möglichkeiten, ein Zeichen auf unterschiedliche Weise darzustellen, durch einen einheitlichen Standard des Formats. Ein weiteres Beispiel für die Normalisierung ist die Kleinschreibung. Wenn ein Modell so konzipiert ist, dass es nur Kleinbuchstaben akzeptiert und verwendet, kann sie verwendet werden, um die Größe des erforderlichen Vokabulars zu verringern. Nach der Normalisierung würde unser Beispielstring »jack sparrow loves new york!« (»jack sparrow liebt new york!«) lauten.

Pretokenization

Bei diesem Schritt wird der Text in kleinere Objekte zerlegt, die einen Anhaltspunkt dafür bieten, wie hoch die Anzahl Ihrer Tokens nach Abschluss des Trainings sein wird. Sie können es sich in etwa so vorstellen, dass der Pretokenizer Ihren Text in »Wörter« aufteilt und Ihre endgültigen Tokens Teile dieser Wörter darstellen. In den Sprachen, in denen dies möglich ist (Englisch, Deutsch und viele indo-europäische Sprachen), können Strings in der Regel anhand von Leerzeichen und Interpunktion in Wörter aufgeteilt werden. In diesem Schritt könnte beispielsweise unser Satz zu `["jack", "sparrow", "loves", "new", "york", "!"]` umgewandelt werden. Diese Wörter können im nächsten Schritt der Pipeline einfacher in Teilwörter (engl. Subwords) mittels Byte-Pair Encoding (BPE) oder Unigram-Algorithmen aufgeteilt werden. Die Zerlegung in »Wörter« ist jedoch nicht immer ein trivialer und deterministischer Vorgang, und auch nicht immer sinnvoll. In Sprachen wie Chinesisch, Japanisch oder Koreanisch kann die Gruppierung von Symbolen in semantische Einheiten wie indogermanische Wörter eine nicht-deterministische Operation sein, bei der es mehrere gleichwertige Gruppen gibt. In diesem Fall ist es vielleicht sinnvoller, den Text nicht vorab zu tokenisieren (engl. pretokenize) und stattdessen eine sprachspezifische Bibliothek für die Pretokenization zu verwenden.

Tokenizer-Modell

Sobald die Eingabetexte normalisiert und vorab tokenisiert wurden, wendet der Tokenizer ein Modell an, mit dem die Wörter in Teilwörter aufgeteilt wer-

den. Dies ist der Teil der Pipeline, der auf Ihrem Korpus trainiert werden muss (oder der bereits trainiert wurde, wenn Sie einen vortrainierten Tokenizer verwenden). Die Aufgabe des Modells besteht darin, die Wörter in Teilwörter aufzuteilen, um die Größe des Vokabulars zu verkleinern und zu versuchen, die Anzahl der Tokens, die nicht zum Vokabular gehören, zu verringern. Es gibt mehrere Algorithmen zur Tokenisierung von Teilwörtern, darunter BPE, Unigram und WordPiece. Unser derzeitiges Beispiel könnte nach der Anwendung des Tokenizer-Modells etwa so aussehen: `[jack, spa, rrow, loves, new, york, !]`. Beachten Sie, dass wir zu diesem Zeitpunkt keine Liste von Strings bzw. Zeichenketten, sondern eine Liste von Ganzzahlen (Input-IDs) vorliegen haben. Um das Beispiel anschaulich zu halten, haben wir zur Verdeutlichung der Transformation zwar die Wörter beibehalten, aber die Anführungszeichen weggelassen.

Nachverarbeitung (engl. Postprocessing)
: Dies ist der letzte Schritt der Tokenisierungspipeline, in dem einige zusätzliche Transformationen auf die Liste von Tokens angewandt werden können – zum Beispiel, indem spezielle Tokens an den Anfang oder das Ende der aus Token-Indizes bestehenden Eingabesequenz hinzugefügt werden. Ein Tokenizer, der nach dem Prinzip arbeitet, das bei BERT zum Einsatz kommt, würde zum Beispiel Tokens hinzufügen, die jeweils angeben, dass es sich um eine Klassifizierung bzw. ein Trennzeichen handelt: `[CLS, jack, spa, rrow, loves, new, york, !, SEP]`. Diese Sequenz (denken Sie daran, dass es sich um eine Folge von Ganzzahlen handelt, nicht um die Tokens, die Sie hier sehen) kann dann in das Modell eingespeist werden.

Wenn wir zu unserem Vergleich von XLM-R und BERT zurückkehren, wissen wir jetzt, dass SentencePiece im Nachverarbeitungsschritt `<s>` und `<\s>` anstelle von `[CLS]` und `[SEP]` hinzufügt (in den Abbildungen werden wir aus Gründen der Einheitlichkeit weiterhin `[CLS]` und `[SEP]` verwenden). Kehren wir nun zum SentencePiece-Tokenizer zurück, um herauszufinden, was ihn so besonders macht.

Der SentencePiece-Tokenizer

Der SentencePiece-Tokenizer basiert auf einer Art von Teilwortsegmentierung (engl. Subword Segmentation) namens Unigram und codiert jeden Eingabetext als eine Folge von Unicode-Zeichen. Letzteres ist besonders nützlich bei mehrsprachigen Korpora, da es SentencePiece erlaubt, auf Akzentzeichen, Interpunktion und die Tatsache, dass viele Sprachen, wie z.B. Japanisch, keine Leerzeichen haben, keine Rücksicht nehmen zu müssen. Eine weitere Besonderheit von SentencePiece ist, dass Leerzeichen mit dem Unicode-Symbol U+2581 bzw. dem _-Zeichen, auch »Achtelblock unten« genannt, versehen sind. Dies ermöglicht es SentencePiece, eine Sequenz zu tokenisieren, ohne dass Mehrdeutigkeiten auftreten und ohne auf sprachspezifische Pretokenizer angewiesen zu sein. An unserem Beispiel aus dem vorigen Abschnitt können wir zum Beispiel sehen, dass WordPiece die Information verloren hat, dass sich zwischen »York« und »!« kein Leerzeichen befindet. Im Ge-

gensatz dazu behält SentencePiece die Leerzeichen im tokenisierten Text bei, sodass wir Texte wieder in den ursprünglichen Rohtext zurückkonvertieren können, ohne dass es dabei zu Mehrdeutigkeiten kommt:

```
"".join(xlmr_tokens).replace(u"\u2581", " ")
```

```
'<s> Jack Sparrow loves New York!</s>'
```

Nachdem wir nun nachvollziehen können, wie der SentencePiece-Tokenizer funktioniert, sollten wir uns überlegen, wie wir unser einfaches Beispiel in eine für die NER geeignete Form codieren können. Als Erstes müssen wir ein vortrainiertes Modell laden, das einen Head zur Klassifizierung von Tokens umfasst. Doch anstatt den Head direkt aus der 🤗 Transformers-Bibliothek zu laden, werden wir ihn selbst erstellen! Sobald wir uns etwas ausführlicher mit der 🤗 Transformers-API befasst haben, sind wir in der Lage, dies in wenigen Schritten umsetzen.

Transformer-Modelle für die Named Entity Recognition

In Kapitel 2 haben wir erfahren, dass BERT im Rahmen der Textklassifizierung das spezielle Token `[CLS]` verwendet, das kennzeichnet, dass es sich um eine eigen- bzw. vollständige Textsequenz handelt. Diese Darstellung wird dann durch eine vollständig verbundene (engl. fully connected) bzw. »dichte« Schicht (engl. Dense Layer) geleitet, die eine Wahrscheinlichkeitsverteilung aller diskreten Label-Werte ausgibt (siehe Abbildung 4-2).

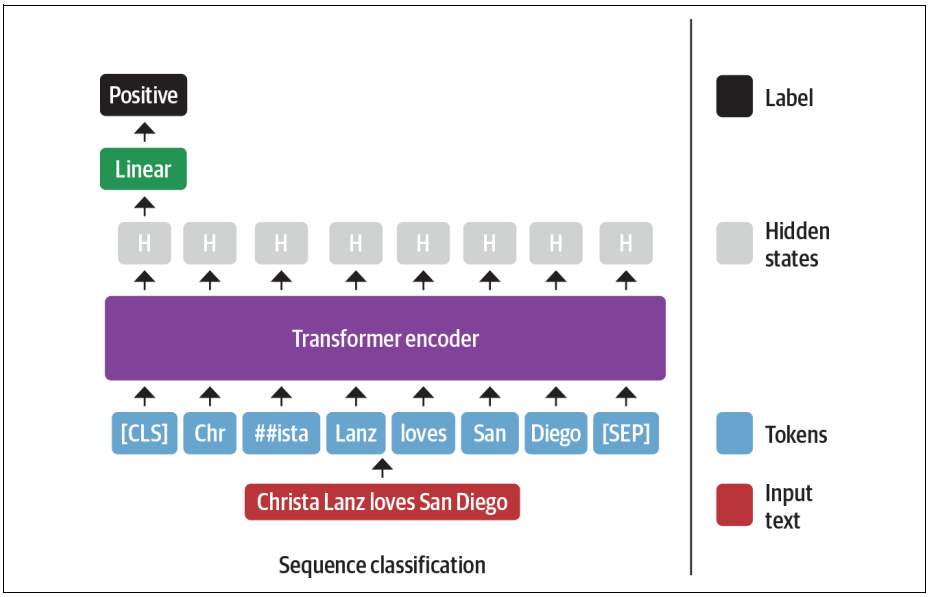


Abbildung 4-2: Feintuning eines auf einem Encoder basierenden Transformer-Modells zur Klassifizierung von Sequenzen

BERT und andere rein Encoder-basierte Transformer-Modelle verfolgen alle einen ähnlichen Ansatz bei der NER, mit der Ausnahme, dass die Darstellungen der einzelnen Eingabe- bzw. Input-Tokens in dieselbe vollständig verbundene Schicht eingespeist werden und anschließend die Entität des Tokens ausgegeben wird. Aus diesem Grund wird die Named Entity Recognition bzw. Eigennamenerkennung oft als eine Aufgabe bezeichnet, bei der es um die *Klassifizierung von Tokens* (engl. Token Classification) geht. Der Prozess läuft in etwa so ab wie in Abbildung 4-3 dargestellt.

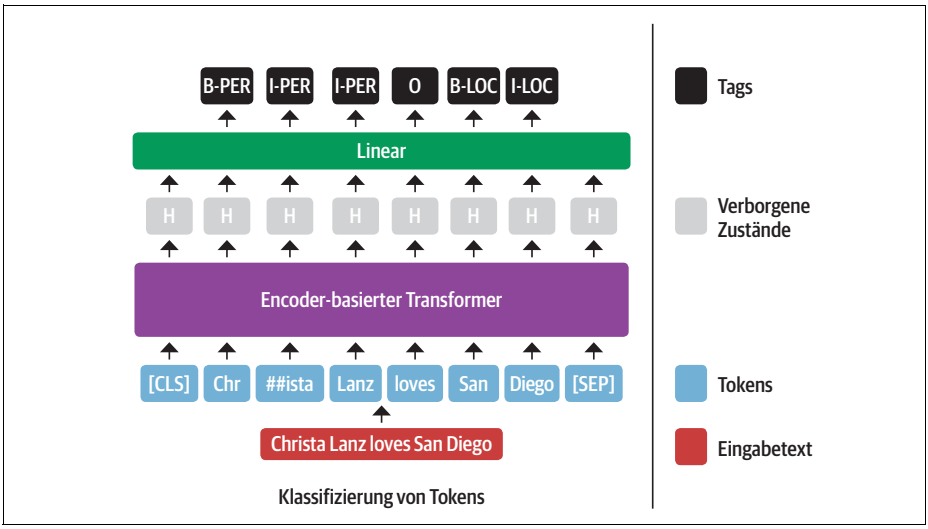


Abbildung 4-3: Feintuning eines auf einem Encoder basierenden Transformer-Modells zur Named Entity Recognition

So weit, so gut. Doch wie sollten wir die Teilwörter in einer Aufgabe zur Klassifizierung von Tokens handhaben? Zum Beispiel wird der Vorname »Christa« in Abbildung 4-3 in die Teilwörter »Chr« und »##ista« tokenisiert bzw. aufgeteilt. Welches der beiden sollte (oder sollten gar beide) dann mit dem Label `B-PER` versehen werden?

Die Autoren des BERT-Forschungspapiers haben sich dazu entschieden, dieses Label dem ersten Teilwort zuzuweisen (»Chr« in unserem Beispiel) und das folgende Teilwort zu ignorieren (»##ista«).[5] Dieser Konvention werden wir folgen und die ignorierten Teilwörter mit `IGN` kennzeichnen. Das vorhergesagte Label des ersten Teilworts können wir später im Nachverarbeitungsschritt leicht auf die nachfolgenden Teilwörter übertragen. Wir hätten auch die Darstellung des Teilworts »##ista« einbeziehen können, indem wir ihm ebenfalls das Label `B-PER` zugewiesen hätten (bzw. eine Kopie dessen). Allerdings würde dies gegen das IOB2-Format verstoßen.

5 J. Devlin et al., »BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding« (*https://arxiv.org/abs/1810.04805*), (2018).

Glücklicherweise lassen sich sämtliche Architekturaspekte, die wir bei BERT kennengelernt haben, auf XLM-R übertragen, da dessen Architektur auf RoBERTa basiert, die mit der von BERT identisch ist! Als Nächstes gehen wir der Frage auf den Grund, wie wir Transformer-Modelle mithilfe der 🤗 Transformers-Bibliothek so modifizieren können, dass wir sie auch für andere Aufgaben verwenden können.

Der Aufbau der Model-Klasse der Transformers-Bibliothek

Die 🤗 Transformers-Bibliothek ist so aufgebaut, dass es für jede Architektur und Aufgabe eine eigene Klasse gibt. Die Namen der den verschiedenen Aufgaben zugeordneten Modellklassen (`Model`) folgen jeweils dem Schema `<Modellname>For<Aufgabe>` bzw. `AutoModelFor<Aufgabe>`, wenn Sie die `AutoModel`-Klassen verwenden.

Dieser Ansatz hat jedoch seine Grenzen. Um Sie auf den Geschmack zu bringen, sich näher mit der API der 🤗 Transformers-Bibliothek zu befassen, betrachten Sie das folgende Szenario. Angenommen, Sie haben eine großartige Idee und wollen ein NLP-Problem, das Sie schon lange beschäftigt, mit einem Transformer-Modell lösen. Sie arrangieren also ein Treffen mit Ihrem Chef und preisen im Rahmen einer kunstvoll gestalteten PowerPoint-Präsentation an, dass Sie den Umsatz Ihrer Abteilung steigern könnten, wenn Sie das Problem endlich lösen würden. Beeindruckt von Ihrer farbenfrohen Präsentation und dem Gerede über Gewinne, erklärt sich Ihr Chef großzügig bereit, Ihnen eine Woche Zeit zu geben, um ein Proof-of-Concept zu entwickeln. Begeistert von dem Gesprächsergebnis machen Sie sich sofort an die Arbeit. Sie öffnen Ihr Notebook und aktivieren eine GPU. Sie führen die Zeile `from transformers import BertForTaskXY` aus (wobei `TaskXY` die imaginäre Aufgabe ist, die Sie lösen möchten), und Ihnen verschlägt es die Sprache, denn der gefürchtete rote Schriftzug füllt Ihren Bildschirm: `ImportError: cannot import name` *`BertForTaskXY`*. Oh nein, es gibt gar kein BERT-Modell für Ihren Anwendungsfall! Wie sollen Sie es schaffen, das Projekt in einer Woche abzuschließen, wenn Sie das komplette Modell erst selbst implementieren müssen?! Wo sollen Sie überhaupt anfangen?

Keine Sorge! Die 🤗 Transformers-Bibliothek wurde so konzipiert, dass Sie die bestehenden Modelle ganz einfach auf Ihren spezifischen Anwendungsfall ausweiten können. Sie können die Gewichtung vortrainierter Modelle laden und haben zudem Zugriff auf aufgabenspezifische Hilfsfunktionen. Dadurch können Sie mit sehr geringem Aufwand maßgeschneiderte Modelle für bestimmte Zielsetzungen (bzw. Aufgaben) erstellen. In diesem Abschnitt werden Sie lernen, wie Sie Ihr eigenes benutzerdefiniertes Modell implementieren können.

Bodies und Heads

Das grundlegende Konzept, das die 🤗 Transformers-Bibliothek so vielseitig macht, ist die Aufteilung der Architektur in einen *Body* (»Körper«) und einen *Head*

(»Kopf«) (wie wir bereits in Kapitel 1 gesehen haben). Wir haben bereits erfahren, dass wir, wenn wir von dem Pretraining-Objective (bzw. der Pretraining-Aufgabe) zur nachgelagerten Aufgabe (engl. Downstream Task) wechseln, die letzte Schicht des Modells durch eine für die Aufgabe geeignete ersetzen müssen. Diese Schicht wird als Head des Modells bezeichnet – es ist der Teil, der *aufgabenspezifisch* ist. Der Rest des Modells wird als Body bezeichnet – er umfasst die Token-Embeddings und Transformer-Schichten, die *aufgabenunabhängig* sind. Diese Struktur spiegelt sich auch in dem Code der 🤗 Transformers-Bibliothek wider: Der Body eines Modells ist in Klassen wie `BertModel` oder `GPT2Model` implementiert, bei denen die verborgenen Zustände der letzten Schicht zurück- bzw. ausgegeben werden. Bei aufgabenspezifischen Modellen wie `BertForMaskedLM` oder `BertForSequenceClassification` wird das entsprechende Grundmodell (engl. Base Model) verwendet und um den für die jeweilige Aufgabe erforderlichen (bzw. geeigneten) Head ergänzt, in den die verborgenen Zustände (des Bodys) eingespeist werden (siehe Abbildung 4-4).

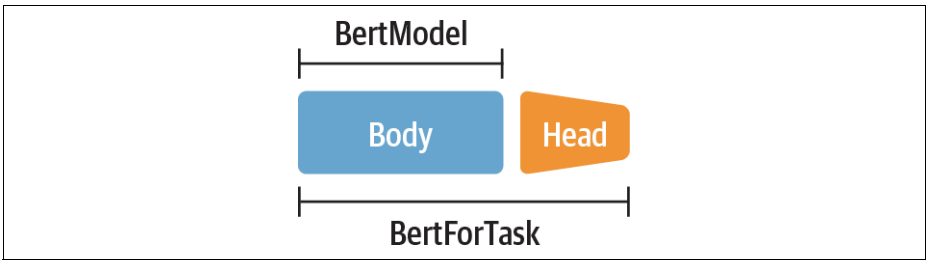


Abbildung 4-4: Die BertModel-Klasse enthält nur den Body des Modells, während die BertFor<Task>-Klassen den Body mit einem für eine bestimmte Aufgabe (<Task>) geeigneten Head kombinieren.

Wie wir gleich noch näher erläutern werden, erlaubt uns die Aufteilung in einen Body und einen Head, einen benutzerdefinierten Head für jede beliebige Aufgabe zu erstellen und ihn einfach auf ein vortrainiertes Modell aufzusetzen.

Ein selbst definiertes Modell zur Klassifizierung von Tokens erstellen

Beginnen wir nun damit, einen selbst definierten Head zur Klassifizierung von Tokens für das XLM-R-Modell zu erstellen. Da XLM-R auf der gleichen Modellarchitektur wie RoBERTa fußt, werden wir RoBERTa als Grundmodell verwenden, das wir jedoch um XLM-R-spezifische Anpassungen erweitern. Beachten Sie, dass dies eine Übung ist, die Ihnen zeigen soll, wie Sie ein selbst definiertes Modell für Ihre eigene Aufgabe erstellen können. Für die Klassifizierung von Tokens existiert bereits die Klasse `XLMRobertaForTokenClassification`, die Sie aus der 🤗 Transformers-Bibliothek importieren können. Wenn Sie diesen Abschnitt überspringen möchten, können Sie also einfach die bereits vorhandene Klasse verwenden.

Zu Beginn benötigen wir eine Datenstruktur, die unseren XLM-R-basierter NER-Tagger (»Eigennamenerkenner«) darstellt. Dann brauchen wir ein Konfigurationsobjekt, mit dem wir das Modell initialisieren, und eine Funktion, `forward()`, mit der wir die Ausgaben erzeugen können. Machen wir uns ans Werk und erstellen unsere XLM-R-Klasse, mit der wir Tokens klassifizieren können:

```
import torch.nn as nn
from transformers import XLMRobertaConfig
from transformers.modeling_outputs import TokenClassifierOutput
from transformers.models.roberta.modeling_roberta import RobertaModel
from transformers.models.roberta.modeling_roberta import RobertaPreTrainedModel

class XLMRobertaForTokenClassification(RobertaPreTrainedModel):
    config_class = XLMRobertaConfig

    def __init__(self, config):
        super().__init__(config)
        self.num_labels = config.num_labels
        # Laden des Modell-Bodys
        self.roberta = RobertaModel(config, add_pooling_layer=False)
        # Head zur Klassifizierung von Tokens einrichten
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.classifier = nn.Linear(config.hidden_size, config.num_labels)
        # Gewichtung laden und initialisieren
        self.init_weights()

    def forward(self, input_ids=None, attention_mask=None, token_type_ids=None,
                labels=None, **kwargs):
        # Modell-Body verwenden, um Darstellungen des Encoders zu erhalten
        outputs = self.roberta(input_ids, attention_mask=attention_mask,
                               token_type_ids=token_type_ids, **kwargs)
        # Klassifikator auf Darstellungen des Encoders anwenden
        sequence_output = self.dropout(outputs[0])
        logits = self.classifier(sequence_output)
        # Verlust berechnen
        loss = None
        if labels is not None:
            loss_fct = nn.CrossEntropyLoss()
            loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
        # Objekt für Modellausgabe zurückgeben
        return TokenClassifierOutput(loss=loss, logits=logits,
                                     hidden_states=outputs.hidden_states,
                                     attentions=outputs.attentions)
```

Die Verwendung von `config_class` stellt sicher, dass die Standardkonfiguration für XLM-R verwendet wird, wenn wir ein neues Modell initialisieren. Wenn Sie die Standardparameter ändern möchten, können Sie dies tun, indem Sie die Standardeinstellungen in der Konfiguration überschreiben. Mit der Methode `super()` rufen wir die Initialisierungsfunktion der `RobertaPreTrainedModel`-Klasse auf. Diese abstrakte Klasse übernimmt die Initialisierung bzw. das Laden der vortrainierten Gewichtung. Anschließend laden wir den Body unseres Modells, d.h. `RobertaModel`, und erweitern ihn um unseren eigenen Head zur Klassifizierung, der aus einer Dropout- und einer gewöhnlichen Feed-Forward-Schicht besteht. Beachten Sie, dass

wir `add_pooling_layer=False` setzen, um sicherzustellen, dass alle verborgenen Zustände zurückgegeben werden und nicht nur derjenige, der dem `[CLS]`-Token zugeordnet ist. Schließlich initialisieren wir alle Gewichte, indem wir die Methode `init_weights()` aufrufen, die wir von der `RobertaPreTrainedModel`-Klasse geerbt haben. Dadurch werden die vortrainierten Gewichte für den Body des Modells geladen und die Gewichte für den Head zur Klassifizierung der Tokens zufällig initialisiert.

Das Einzige, was noch aussteht, ist, zu definieren, wie das Modell im Rahmen eines Forward-Pass, der durch die `forward()`-Methode initiiert wird, verfahren soll. Während des Forward-Pass werden die Daten zunächst durch den Body des Modells geleitet. Es gäbe eine Reihe möglicher Eingabevariablen, aber die einzigen, die wir in diesem Zusammenhang benötigen, sind `input_ids` und `attention_mask`. Der verborgene Zustand, der Teil der Ausgabe des Modell-Bodys ist, wird dann durch die Dropout- und die Klassifizierungsschicht geleitet. Wenn wir im Rahmen des Forward-Pass zusätzlich noch die Labels bereitstellen, können wir direkt den Verlust berechnen. Wenn eine Attention-Mask vorhanden wäre, wären wir gezwungen, etwas mehr Aufwand zu betreiben, da wir sicherstellen müssten, dass wir den Verlust nur auf Basis der unmaskierten Tokens berechnen. Schließlich fassen wir alle Ausgaben in einem `TokenClassifierOutput`-Objekt zusammen, das uns den Zugriff auf die Elemente in Form eines benannten Tupels ermöglicht, so wie wir es bereits aus den vorherigen Kapiteln kennen.

Um ein selbst definiertes Transformer-Modell zu erstellen, genügt es also, zwei Funktionen einer einfachen Klasse zu implementieren. Da wir von einer `PreTrainedModel`-Klasse erben, haben wir sofort Zugriff auf alle nützlichen Funktionalitäten der 🤗 Transformers-Bibliothek, wie z.B. die `from_pretrained()`-Methode! Schauen wir uns an, wie wir vortrainierte Gewichte für unser eigenes Modell laden können.

Ein selbst definiertes Modell laden

Nun sind wir bereit, unser für die Klassifizierung von Tokens konzipiertes Modell zu laden. Neben dem Modellnamen müssen wir einige zusätzliche Informationen bereitstellen, darunter die Tags, die wir zur Kennzeichnung (d.h. als Labels) der einzelnen Entitäten verwenden werden, sowie die Zuordnung jedes Tags zu einer eindeutigen Ganzzahl (ID) und umgekehrt. Alle diese Informationen lassen sich aus unserer Variablen `tags` ableiten, die als `ClassLabel`-Objekt ein Attribut `names` besitzt, mit dem wir die Zuordnung (engl. Mapping) vornehmen können:

```
index2tag = {idx: tag for idx, tag in enumerate(tags.names)}
tag2index = {tag: idx for idx, tag in enumerate(tags.names)}
```

Die Zuordnungen als auch das Attribut `tags.num_classes` speichern wir nun in dem `AutoConfig`-Objekt, das wir bereits in Kapitel 3 kennengelernt haben. Dadurch, dass wir in der `from_pretrained()`-Methode die entsprechenden Schlüsselwortargumente angeben, werden die Standardwerte (engl. Default Values) überschrieben:

```
from transformers import AutoConfig

xlmr_config = AutoConfig.from_pretrained(xlmr_model_name,
                                         num_labels=tags.num_classes,
                                         id2label=index2tag, label2id=tag2index)
```

Die `AutoConfig`-Klasse enthält eine Blaupause für die Architektur eines Modells. Wenn wir mit der Methode `AutoModel.from_pretrained(`*`model_ckpt`*`)` ein Modell laden, wird die diesem Modell zugeordnete Konfigurationsdatei automatisch heruntergeladen. Wenn wir allerdings etwas ändern möchten, wie z.B. die Anzahl der Kategorien oder die Namen der Labels, dann können wir die Konfiguration einfach selbst laden und dabei die entsprechenden Parameter anpassen.

Nun können wir wie gewohnt die Modellgewichte mithilfe der Funktion `from_pretrained()` unter Angabe des `config`-Arguments laden. Beachten Sie, dass wir in unserer selbst definierten Modell-Klasse keine Funktion zum Laden vortrainierter Gewichte implementiert haben. Stattdessen erben wir diese Funktion einfach von der `RobertaPreTrainedModel`-Klasse:

```
import torch

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
xlmr_model = (XLMRobertaForTokenClassification
              .from_pretrained(xlmr_model_name, config=xlmr_config)
              .to(device))
```

Überprüfen wir kurz, ob wir den Tokenizer und das Modell richtig initialisiert haben, indem wir überprüfen, ob die Vorhersagen für die kurze Sequenz uns bekannter Entitäten zutreffen:

```
input_ids = xlmr_tokenizer.encode(text, return_tensors="pt")
pd.DataFrame([xlmr_tokens, input_ids[0].numpy()], index=["Tokens", "Input-IDs"])
```

	0	1	2	3	4	5	6	7	8	9
Tokens	<s>	_Jack	_Spar	row	_love	s	_New	_York	!	</s>
Input-IDs	0	21763	37456	15555	5161	7	2356	5753	38	2

Wie Sie sehen können, wird für das `<s>`- und `</s>`-Token, das jeweils den Anfang bzw. das Ende einer Sequenz kennzeichnet, eine 0 bzw. eine 2 als ID vergeben.

Nun können wir die Eingaben an das Modell übergeben und die Vorhersagen ermitteln, indem wir die `argmax`-Funktion anwenden. Auf diese Weise erhalten wir die für jedes Token wahrscheinlichste Kategorie:

```
outputs = xlmr_model(input_ids.to(device)).logits
predictions = torch.argmax(outputs, dim=-1)
print(f"Anzahl der Tokens in der Sequenz: {len(xlmr_tokens)}")
print(f"Shape der Modellausgabe: {outputs.shape}")
```

```
Anzahl der Tokens in der Sequenz: 10
Shape der Modellausgabe: torch.Size([1, 10, 7])
```

Hier sehen wir, dass die Logits mit einen Shape von `[batch_size, num_tokens, num_tags]` vorliegen, wobei für jedes Token jeweils ein Logit je NER-Kategorie ermittelt wurde. Wenn wir die Sequenz durchnummerieren, können wir rasch herausfinden, welche Vorhersagen das vortrainierte Modell getroffen hat:

```
preds = [tags.names[p] for p in predictions[0].cpu().numpy()]
pd.DataFrame([xlmr_tokens, preds], index=["Tokens", "Tags"])
```

	0	1	2	3	4	5	6	7	8	9
Tokens	<s>	_Jack	_Spar	row	_love	s	_New	_York	!	</s>
Tags	O	I-LOC	B-LOC	B-LOC	O	I-LOC	O	O	I-LOC	B-LOC

Wie zu erwarten war, ist unsere mit zufälligen Gewichten initialisierte Schicht zur Klassifizierung von Tokens noch nicht ganz ausgereift. Nehmen wir also ein Feintuning mit einigen gelabelten Daten vor, um sie zu verbessern. Zuvor sollten wir allerdings die vorangegangenen Schritte in einer Hilfsfunktion zusammenfassen, um sie später verwenden zu können:

```
def tag_text(text, tags, model, tokenizer):
    # Tokens inkl. spezieller Tokens erhalten
    tokens = tokenizer(text).tokens()
    # Codierung der Sequenz in Form von IDs
    input_ids = xlmr_tokenizer(text, return_tensors="pt").input_ids.to(device)
    # Vorhersagen als Wahrscheinlichkeitsverteilung über 7 mögliche Kategorien
    outputs = model(input_ids)[0]
    # Mit argmax die wahrscheinlichste Kategorie je Token erhalten
    predictions = torch.argmax(outputs, dim=2)
    # In DataFrame überführen
    preds = [tags.names[p] for p in predictions[0].cpu().numpy()]
    return pd.DataFrame([tokens, preds], index=["Tokens", "Tags"])
```

Doch bevor wir das Modell trainieren können, müssen wir noch die Eingaben tokenisieren und die Labels vorbereiten. Befassen wir uns damit also als Nächstes!

Tokenisierung von Texten für die Named Entity Recognition

Nachdem wir nun sichergestellt haben, dass der Tokenizer und das Modell ein einzelnes Beispiel codieren können, besteht unser nächster Schritt darin, den gesamten Datensatz zu tokenisieren, um ihn für das Feintuning in das XLM-R-Modell zu speisen. Wie wir in Kapitel 2 gesehen haben, bietet die 🤗 Datasets-Bibliothek eine schnelle Möglichkeit, ein `Dataset`-Objekt mithilfe der `map()`-Methode zu tokenisieren. Um dies bewerkstelligen zu können, müssen wir zunächst eine Funktion definieren, die die folgende Grundspezifikation aufweist:

```
function(examples: Dict[str, List]) -> Dict[str, List]
```

wobei `examples` einem Auszug (d.h. einem Slice) an Beispielen aus einem Datensatz (`Dataset`) entspricht, z.B. `panx_de['train'][:10]`. Da der XLM-R-Tokenizer die Input-

IDs als Eingaben für das Modell zurückgibt, müssen wir diese nur noch um die Attention-Mask und Label-IDs ergänzen, die codieren, welche NER-Tags den einzelnen Tokens zugeordnet sind.

Sehen wir uns einmal an, wie das mit unserem einzelnen deutschen Beispiel funktioniert, indem wir – dem Ansatz der 🤗 Transformers-Dokumentation (*https://oreil.ly/lGPgh*) folgend – zunächst die Wörter und Tags als gewöhnliche Listen sammeln:

```
words, labels = de_example["tokens"], de_example["ner_tags"]
```

Als Nächstes tokenisieren wir die Wörter und verwenden das Argument `is_split_into_words`, um dem Tokenizer mitzuteilen, dass unsere Eingabesequenz bereits in Wörter aufgeteilt wurde:

```
tokenized_input = xlmr_tokenizer(de_example["tokens"], is_split_into_words=True)
tokens = xlmr_tokenizer.convert_ids_to_tokens(tokenized_input["input_ids"])
pd.DataFrame([tokens], index=["Tokens"])
```

	0	1	2	3	4	5	6	...	18	19	20	21	22	23	24
Tokens	<s>	_2.000	_Einwohner	n	_an	_der	_Dan	...	schaft	_Po	mmer	n	_	.	</s>

In diesem Beispiel wird ersichtlich, dass der Tokenizer das Wort »Einwohnern« in zwei Teilwörter zerlegt hat, »_Einwohner« und »n«. Da wir uns an die Konvention halten, dass nur »_Einwohner« mit dem Label `B-LOC` assoziiert werden soll, benötigen wir eine Möglichkeit, die Teilwortdarstellungen nach dem ersten Teilwort zu maskieren. Glücklicherweise umfasst die `tokenized_input`-Klasse eine `word_ids()`-Funktion, mit der wir das umsetzen können:

```
word_ids = tokenized_input.word_ids()
pd.DataFrame([tokens, word_ids], index=["Tokens", "Word-IDs"])
```

	0	1	2	3	4	5	6	...	18	19	20	21	22	23	24
Tokens	<s>	_2.000	_Einwohner	n	_an	_der	_Dan	...	schaft	_Po	mmer	n	_	.	</s>
Word-IDs	None	0	1	1	2	3	4	...	9	10	10	10	11	11	None

In diesem Beispiel können wir sehen, dass in `word_ids` jedem Teilwort der entsprechende Index in der `words`-Sequenz zugeordnet wurde. Dem ersten Teilwort, »_2.000«, wird also der Index 0 zugewiesen, während »_Einwohner« und »n« den Index 1 erhalten (da »Einwohnern« das zweite Wort in `words` ist). Zudem können wir erkennen, dass spezielle Tokens wie `<s>` und `<s>` auf `None` gesetzt werden. Legen wir den Wert `-100` als Label für diese speziellen Tokens und die Teilwörter fest, die wir während des Trainings maskieren möchten:

```
previous_word_idx = None
label_ids = []

for word_idx in word_ids:
    if word_idx is None or word_idx == previous_word_idx:
```

```
            label_ids.append(-100)
        elif word_idx != previous_word_idx:
            label_ids.append(labels[word_idx])
        previous_word_idx = word_idx

    labels = [index2tag[l] if l != -100 else "IGN" for l in label_ids]
    index = ["Tokens", "Word-IDs", "Label-IDs", "Labels"]

    pd.DataFrame([tokens, word_ids, label_ids, labels], index=index)
```

	0	1	2	3	4	5	...	19	20	21	22	23	24
Tokens	<s>	_2.000	_Einwohner	n	_an	_der	...	_Po	mmer	n	_	.	</s>
Word-IDs	None	0	1	1	2	3	...	10	10	10	11	11	None
Label-IDs	-100	0	0	-100	0	0	...	6	-100	-100	0	-100	-100
Labels	IGN	O	O	IGN	O	O	...	I-LOC	IGN	IGN	O	IGN	IGN

Warum haben wir `-100` als ID für die Maskierung von Teilwortdarstellungen gewählt? Der Grund ist, dass die PyTorch-Klasse `torch.nn.CrossEntropyLoss` ein Attribut namens `ignore_index` hat, dessen Wert `-100` beträgt. Dadurch wird das entsprechende Token beim Training nicht berücksichtigt. Wir können ihn also verwenden, um die Tokens zu ignorieren, die mit aufeinanderfolgenden Teilwörtern assoziiert werden.

Fertig, das war's! Wir können deutlich erkennen, welche Label-IDs mit welchen Tokens übereinstimmen. Übertragen wir diesen Ansatz nun auf den gesamten Datensatz, indem wir eine einzelne Funktion definieren, die die gesamte Logik umfasst:

```
def tokenize_and_align_labels(examples):
    tokenized_inputs = xlmr_tokenizer(examples["tokens"], truncation=True,
                                      is_split_into_words=True)
    labels = []
    for idx, label in enumerate(examples["ner_tags"]):
        word_ids = tokenized_inputs.word_ids(batch_index=idx)
        previous_word_idx = None
        label_ids = []
        for word_idx in word_ids:
            if word_idx is None or word_idx == previous_word_idx:
                label_ids.append(-100)
            else:
                label_ids.append(label[word_idx])
            previous_word_idx = word_idx
        labels.append(label_ids)
    tokenized_inputs["labels"] = labels
    return tokenized_inputs
```

Jetzt haben wir alle Zutaten, die wir benötigen, um die aufgeteilten Teildatensätze (engl. Splits) codieren zu können. Formulieren wir zunächst eine Funktion, über die wir iterieren können:

```
def encode_panx_dataset(corpus):
    return corpus.map(tokenize_and_align_labels, batched=True,
                      remove_columns=['langs', 'ner_tags', 'tokens'])
```

Wenn wir diese Funktion auf ein `DatasetDict`-Objekt anwenden, erhalten wir für den jeweils verwendeten Datensatz ein codiertes `Dataset`-Objekt. Codieren wir nun auf diese Weise unser deutsches Korpus:

```
panx_de_encoded = encode_panx_dataset(panx_ch["de"])
```

Nachdem wir nun sowohl ein Modell als auch einen Datensatz zur Hand haben, benötigen wir noch ein Maß, mit dem wir die Leistung des Modells beurteilen können.

Qualitätsmaße

Ein NER-Modell wird ähnlich wie ein Textklassifizierungsmodell evaluiert, wobei es üblich ist, die Werte für die Precision (im Deutschen auch als Relevanz, positiver Vorhersagewert oder Wirksamkeit bezeichnet), den Recall (im Deutschen auch als Sensitivität, Richtig-positiv-Rate oder Trefferquote bekannt) und für das F_1-Maß (engl. F_1-Score) anzugeben. Der einzige Unterschied besteht darin, dass *alle* Wörter einer Entität korrekt vorhergesagt werden müssen, damit eine Vorhersage als korrekt gewertet wird. Glücklicherweise gibt es eine elegante Bibliothek namens *seqeval* (*https://oreil.ly/xbKOp*), die für diese Art von Aufgaben entwickelt wurde. Nehmen wir beispielhaft einige NER-Tags und Modellvorhersagen als Platzhalter, um die Maße mithilfe der `classification_report()`-Funktion der seqeval-Bibliothek zu berechnen:

```
from seqeval.metrics import classification_report

y_true = [["O", "O", "O", "B-MISC", "I-MISC", "I-MISC", "O"],
          ["B-PER", "I-PER", "O"]]
y_pred = [["O", "O", "B-MISC", "I-MISC", "I-MISC", "I-MISC", "O"],
          ["B-PER", "I-PER", "O"]]
print(classification_report(y_true, y_pred))
```

```
              precision    recall  f1-score   support

        MISC       0.00      0.00      0.00         1
         PER       1.00      1.00      1.00         1

   micro avg       0.50      0.50      0.50         2
   macro avg       0.50      0.50      0.50         2
weighted avg       0.50      0.50      0.50         2
```

Wie Sie sehen, erwartet die *seqeval*-Bibliothek die Vorhersagen und Labels in Form von Listen, die wiederum mehrere Listen enthalten, wobei jede Liste einem einzelnen Beispiel in unserem Validierungs- oder Testdatensatz entspricht. Um diese Maße während des Trainings ermitteln zu können, benötigen wir eine Funktion, die die Ausgaben des Modells noch in Listen konvertiert (wie sie von der *seqeval*-Bibliothek erwartet werden). Die folgende Funktion stellt sicher, dass wir die mit

den jeweils nachfolgenden Teilwörtern verbundenen Label-IDs nicht berücksichtigen:

```
import numpy as np

def align_predictions(predictions, label_ids):
    preds = np.argmax(predictions, axis=2)
    batch_size, seq_len = preds.shape
    labels_list, preds_list = [], []

    for batch_idx in range(batch_size):
        example_labels, example_preds = [], []
        for seq_idx in range(seq_len):
            # Ignoriere Label-IDs = -100
            if label_ids[batch_idx, seq_idx] != -100:
                example_labels.append(index2tag[label_ids[batch_idx][seq_idx]])
                example_preds.append(index2tag[preds[batch_idx][seq_idx]])

        labels_list.append(example_labels)
        preds_list.append(example_preds)

    return preds_list, labels_list
```

Nachdem wir nun über ein Qualitätsmaß verfügen, können wir mit dem eigentlichen Training des Modells fortfahren.

Feintuning eines XLM-RoBERTa-Modells

Jetzt haben wir alle erforderlichen Voraussetzungen, um unser Modell feintunen zu können. Unsere erste Strategie besteht darin, unser Grundmodell mit dem deutschen Teildatensatz von PAN-X feinzutunen und dann seine sprachenübergreifende Zero-Shot-Performance für die Sprachen Französisch, Italienisch und Englisch zu evaluieren. Wie gewohnt verwenden wir die `Trainer`-Klasse der 🤗 Transformers-Bibliothek, um unsere Trainingsschleife zu handhaben. Legen wir also zunächst die für das Training erforderlichen Attribute mithilfe der Klasse `TrainingArguments` fest:

```
from transformers import TrainingArguments

num_epochs = 3
batch_size = 24
logging_steps = len(panx_de_encoded["train"]) // batch_size
model_name = f"{xlmr_model_name}-finetuned-panx-de"
training_args = TrainingArguments(
    output_dir=model_name, log_level="error", num_train_epochs=num_epochs,
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size, evaluation_strategy="epoch",
    save_steps=1e6, weight_decay=0.01, disable_tqdm=False,
    logging_steps=logging_steps, push_to_hub=True)
```

Hierbei geben wir an, dass wir die Vorhersagen des Modells auf dem Validierungsdatensatz zum Ende einer jeden Epoche evaluieren und den Weight Decay – eine

Technik zur Regularisierung, die die Aktualisierung der Gewichte beeinflusst – anpassen. Für das Argument `save_steps` geben wir eine große Zahl an, damit keine Checkpoints erstellt werden und so das Training schneller vonstattengeht.

Dies ist auch ein guter Zeitpunkt, um sicherzustellen, dass wir beim Hugging Face Hub eingeloggt sind (wenn Sie in einem Terminal arbeiten, können Sie stattdessen den Befehl `huggingface-cli login` ausführen):

```
from huggingface_hub import notebook_login

notebook_login()
```

Wir müssen außerdem dem `Trainer` mitteilen, wie er die Maße für den Validierungsdatensatz berechnen soll. Hierzu können wir die zuvor definierte Funktion `align_predictions()` verwenden, um die Vorhersagen und Labels in einem Format zu extrahieren, das zur Berechnung des F_1-Maßes mithilfe der *seqeval*-Bibliothek erforderlich ist:

```
from seqeval.metrics import f1_score

def compute_metrics(eval_pred):
    y_pred, y_true = align_predictions(eval_pred.predictions,
                                       eval_pred.label_ids)
    return {"f1": f1_score(y_true, y_pred)}
```

In einem letzten Schritt müssen wir einen *Data-Collator* definieren, mit dem wir alle Eingabesequenzen eines Batches auf die darin jeweils längste vorkommende Eingabesequenz auffüllen (engl. pad) können. Die 🤗 Transformers-Bibliothek bietet einen speziellen Data-Collator für die Klassifizierung von Tokens, der neben den Eingabe- auch die Label-Sequenzen auffüllt:

```
from transformers import DataCollatorForTokenClassification

data_collator = DataCollatorForTokenClassification(xlmr_tokenizer)
```

Anders als bei der Klassifizierung von Texten müssen auch die Labels aufgefüllt werden, da sie ebenfalls Sequenzen darstellen. Ein wichtiges Detail dabei ist, dass die Label-Sequenzen mit dem Wert `-100` aufgefüllt werden, der, wie wir bereits erfahren haben, von PyTorch's Verlustfunktionen nicht berücksichtigt wird.

Da wir im Laufe dieses Kapitels mehrere Modelle trainieren werden, wollen wir vermeiden, für jeden `Trainer` ein neues Modell initialisieren zu müssen. Daher erstellen wir eine Methode `model_init()`, mit der ein noch untrainiertes Modell geladen wird und die zu Beginn des Aufrufs von `train()` aufgerufen wird:

```
def model_init():
    return (XLMRobertaForTokenClassification
            .from_pretrained(xlmr_model_name, config=xlmr_config)
            .to(device))
```

Wir können nun all diese Informationen zusammen mit den codierten Datensätzen an den `Trainer` übergeben:

```
from transformers import Trainer

trainer = Trainer(model_init=model_init, args=training_args,
                  data_collator=data_collator, compute_metrics=compute_metrics,
                  train_dataset=panx_de_encoded["train"],
                  eval_dataset=panx_de_encoded["validation"],
                  tokenizer=xlmr_tokenizer)
```

und anschließend die Trainingsschleife wie folgt ausführen und das endgültige Modell auf den Hub übertragen bzw. »pushen«:

```
trainer.train() trainer.push_to_hub(commit_message="Training completed!")
```

Epoche	Trainingsverlust	Validierungsverlust	F1-Maß
1	0.2652	0.160244	0.822974
2	0.1314	0.137195	0.852747
3	0.0806	0.138774	0.864591

Die Werte für das F1-Maß fallen für ein NER-Modell ziemlich gut aus. Überprüfen wir, ob unser Modell wie erwartet funktioniert, indem wir es mit der deutschen Übersetzung unseres einfachen Beispiels ausprobieren:

```
text_de = "Jeff Dean ist ein Informatiker bei Google in Kalifornien"
tag_text(text_de, tags, trainer.model, xlmr_tokenizer)
```

	0	1	2	3	4	5	...	8	9	10	11	12	13
Tokens	<s>	_Jeff	_De	an	_ist	_ein	...	_bei	_Google	_in	_Kaliforni	en	</s>
Tags	O	B-PER	I-PER	I-PER	O	O	...	O	B-ORG	O	B-LOC	I-LOC	O

Es funktioniert! Allerdings sollten wir nicht zu stark darauf vertrauen und die Leistung auf der Grundlage nur eines einzigen Beispiels beurteilen. Stattdessen sollten wir die Fehler des Modells angemessen und gründlich untersuchen. Im nächsten Abschnitt erfahren Sie, wie Sie dies für die NER-Aufgabe vornehmen können.

Fehleranalyse

Bevor wir uns eingehender mit den mehrsprachigen Aspekten des XLM-R-Modells befassen, sollten wir uns kurz die Zeit nehmen, die Fehler unseres Modells zu untersuchen. Wie wir in Kapitel 2 erfahren haben, ist einer der wichtigsten Aspekte, wenn Sie Transformer (und Machine-Learning-Modelle im Allgemeinen) trainieren und debuggen, dass Sie die Fehler Ihres Modells gründlich analysieren. Es gibt mehrere Arten von Fehlern, bei denen es den Anschein hat, als ob das Modell gut funktioniert, obwohl es in Wirklichkeit einige schwerwiegende Fehler aufweist. Beispiele, in denen das Training unter Umständen nicht zum gewünschten Erfolg führt, sind:

- Wir könnten versehentlich sowohl zu viele Tokens als auch einige unserer Labels maskieren, wodurch der Verlust auf vielversprechende Weise sinkt.
- Die `compute_metrics()`-Funktion könnte einen Fehler aufweisen, durch den die tatsächliche Leistung überschätzt wird.
- Wir könnten eine Null als Kategorie bzw. die Entität »O« als normale Kategorie in die NER einbeziehen, was die Treffergenauigkeit (engl. Accuracy) und das F_1-Maß stark verzerren würde, da ihr mit Abstand die meisten Tokens zuzuordnen wären.

Wenn das Modell wesentlich schlechter als erwartet abschneidet, kann ein Blick auf die Fehler nützliche Erkenntnisse liefern und wiederum Fehler aufdecken, die beim alleinigen Betrachten des Codes schwer zu erkennen wären. Doch selbst wenn das Modell gut abschneidet und keine Fehler im Code vorhanden sind, ist die Fehleranalyse ein nützliches Instrument, um seine Stärken und Schwächen zu eruieren. All diese Aspekte müssen wir immer im Auge behalten, wenn wir ein Modell in einer Produktionsumgebung deployen.

Für unsere Analyse werden wir wieder auf eines der mächtigsten Werkzeuge zurückgreifen, die uns zur Verfügung stehen, nämlich die Beispiele des Validierungsdatensatzes, die den höchsten Verlust aufweisen. Wir können einen Großteil der Funktion wiederverwenden, die wir zur Analyse des Sequenzklassifizierungsmodells in Kapitel 2 erstellt haben. Allerdings berechnen wir nun den Verlust pro Token in der Beispielsequenz.

Legen wir zunächst eine entsprechende Methode fest, die wir anschließend auf den Validierungsdatensatz anwenden können:

```
from torch.nn.functional import cross_entropy

def forward_pass_with_label(batch):
    # Konvertiert aus Listen bestehendes Dictionary in eine
    #  Liste aus Dictionarys, die für den Data-Collator geeignet sind
    features = [dict(zip(batch, t)) for t in zip(*batch.values())]
    # Eingaben und Labels auffüllen und alle Tensoren auf Device platzieren
    batch = data_collator(features)
    input_ids = batch["input_ids"].to(device)
    attention_mask = batch["attention_mask"].to(device)
    labels = batch["labels"].to(device)
    with torch.no_grad():
        # Daten durch das Modell leiten
        output = trainer.model(input_ids, attention_mask)
        # logit.size: [batch_size, sequence_length, classes]
        # Vorhersage der Kategorie, die den höchsten Logit aufweist
        #  (hinsichtlich Dimension für Kategorien)
        predicted_label = torch.argmax(output.logits, axis=-1).cpu().numpy()
    # Berechnung des Verlusts pro Token
    #  nach "Verflachung"/Flattening der Batch-Dimension mittels view()
    loss = cross_entropy(output.logits.view(-1, 7),
                         labels.view(-1), reduction="none")
    # Hinsichtl. Batch-Dimension "entflachen" und in Numpy-Array konvertieren
    loss = loss.view(len(input_ids), -1).cpu().numpy()

    return {"loss":loss, "predicted_label": predicted_label}
```

Die Funktion können wir nun mithilfe der `map()`-Methode auf den gesamten Validierungsdatensatz anwenden und alle Daten zur weiteren Analyse in einem `DataFrame` laden:

```
valid_set = panx_de_encoded["validation"]
valid_set = valid_set.map(forward_pass_with_label, batched=True, batch_size=32)
df = valid_set.to_pandas()
```

Die Tokens und die Labels sind immer noch mit ihren IDs codiert. Ordnen wir also die Tokens und Labels wieder Strings bzw. Zeichenketten zu, damit die Ergebnisse leichter zu lesen sind. Den Padding-Tokens, die mit dem Wert `-100` gekennzeichnet sind und zur Auffüllung (Padding) dienen, weisen wir ein spezielles Label (`IGN`) zu, damit wir sie später filtern können. Darüber hinaus entledigen wir uns aller Auffüllungen in den Feldern `loss` und `predicted_label`, indem wir sie auf die Länge der Eingaben stutzen (engl. truncate):

```
index2tag[-100] = "IGN"
df["input_tokens"] = df["input_ids"].apply(
    lambda x: xlmr_tokenizer.convert_ids_to_tokens(x))
df["predicted_label"] = df["predicted_label"].apply(
    lambda x: [index2tag[i] for i in x])
df["labels"] = df["labels"].apply(
    lambda x: [index2tag[i] for i in x])
df['loss'] = df.apply(
    lambda x: x['loss'][:len(x['input_ids'])], axis=1)
df['predicted_label'] = df.apply(
    lambda x: x['predicted_label'][:len(x['input_ids'])], axis=1)
df.head(1)
```

	attention_mask	input_ids	labels	loss	predicted_label	input_tokens
0	[1, 1, 1, 1, 1, 1, 1]	[0, 10699, 11, 15, 16104, 1388, 2]	[IGN, B-ORG, IGN, I-ORG, I-ORG, I-ORG, IGN]	[0.0, 0.014679872, 0.0, 0.009469474, 0.010393422, 0.01293836, 0.0]	[I-ORG, B-ORG, I-ORG, I-ORG, I-ORG, I-ORG, I-ORG]	[<s>, _Ham, a, _(, _Unternehmen, _), </s>]

Jede Zeile enthält jeweils ein Beispiel mit einer Liste von Tokens, (tatsächlichen) Labels, vorhergesagten Labels usw. Werfen wir nun einen Blick auf die einzelnen Tokens, indem wir diese Listen entpacken. Dank der Funktion `pandas.Series.explode()` können wir diesen Schritt in einer einzelnen Codezeile vornehmen, wobei jeweils für jedes Element in der ursprünglichen Liste eine Zeile erstellt wird. Da alle Listen, die in einer Zeile enthalten sind, die gleiche Länge haben, können wir die Funktion auf alle Spalten gleichzeitig anwenden. Wir lassen auch die Padding-Tokens weg, die wir mit `IGN` gekennzeichnet haben, da der mit ihnen verbundene Verlust ohnehin gleich null ist. Schließlich wandeln wir die Verlustwerte, die immer noch als `numpy.Array`-Objekte vorliegen, in normale Gleitkommazahlen um:

```
df_tokens = df.apply(pd.Series.explode)
df_tokens = df_tokens.query("labels != 'IGN'")
```

```
df_tokens["loss"] = df_tokens["loss"].astype(float).round(2)
df_tokens.head(7)
```

attention_mask	input_ids	labels	loss	predicted_label	input_tokens
1	10699	B-ORG	0.01	B-ORG	_Ham
1	15	I-ORG	0.01	I-ORG	_(
1	16104	I-ORG	0.01	I-ORG	_Unternehmen
1	1388	I-ORG	0.01	I-ORG	_)
1	56530	O	0.00	O	_WE
1	83982	B-ORG	0.34	B-ORG	_Luz
1	10	I-ORG	0.45	I-ORG	_a

Nachdem wir die Daten in diese Form gebracht haben, können wir sie hinsichtlich der Eingabe-Tokens gruppieren und zusammenfassende Statistiken für die Verluste jedes Tokens wie die Anzahl der Vorkommen (engl. Count), den Durchschnitt bzw. das arithmetische Mittel (engl. Mean) und die Summe (engl. Sum) ermitteln. Anschließend sortieren wir die aggregierten Daten entsprechend der Höhe der Summe des Verlusts und können so ermitteln, welche Tokens im Validierungsdatensatz insgesamt den höchsten Verlust aufweisen:

```
(
    df_tokens.groupby("input_tokens")[["loss"]]
    .agg(["count", "mean", "sum"])
    .droplevel(level=0, axis=1)  # unbenötigte Ebene verwefen
    .sort_values(by="sum", ascending=False)
    .reset_index()
    .round(2)
    .head(10)
    .T
)
```

	0	1	2	3	4	5	6	7	8	9
input_tokens	_	_der	_in	_von	_/	_und	_(	_)	_"	_A
count	6066	1388	989	808	163	1171	246	246	2898	125
mean	0.03	0.1	0.14	0.14	0.64	0.08	0.3	0.29	0.02	0.44
sum	200.71	138.05	137.33	114.92	104.28	99.15	74.49	72.35	59.31	54.48

Anhand dieser Auswahl von Tokens können wir gleich mehrere Muster erkennen:

- Das Leerzeichen hat den höchsten Gesamtverlust (engl. Total Loss), was nicht überraschend ist, da es auch das Token in der Auswahl ist, das am häufigsten vorkommt. Der durchschnittliche Verlust fällt jedoch wesentlich geringer aus als bei den anderen Tokens in der Auswahl. Das bedeutet, dass das Modell keine Schwierigkeiten hat, es zu klassifizieren.

- Wörter wie »in«, »von«, »der« und »und« kommen relativ häufig vor. Sie tauchen oft zusammen mit benannten Entitäten (Eigennamen) auf und sind manchmal auch ein Teil von ihnen, was erklärt, warum das Modell sie verwechseln könnte.
- Klammern, Schrägstriche und Großbuchstaben, die am Anfang von Wörtern stehen, sind seltener, weisen aber einen relativ hohen durchschnittlichen Verlust auf. Das werden wir noch genauer unter die Lupe nehmen.

Ebenso können wir die IDs für die Labels gruppieren und uns die Verluste für jede Kategorie ansehen:

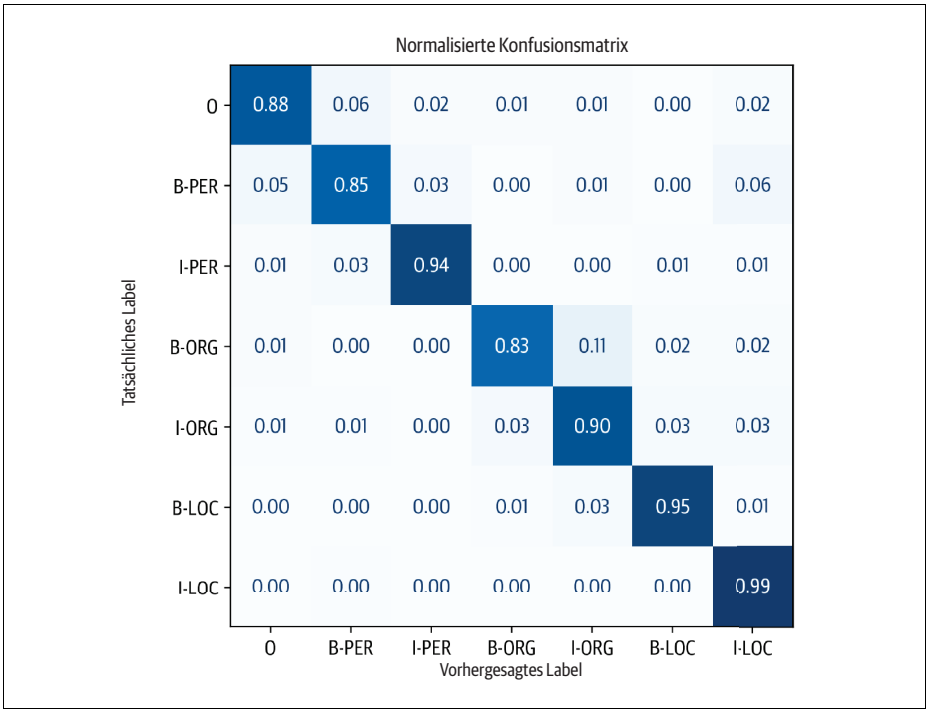
```
(
    df_tokens.groupby("labels")[["loss"]]
    .agg(["count", "mean", "sum"])
    .droplevel(level=0, axis=1)
    .sort_values(by="mean", ascending=False)
    .reset_index()
    .round(2)
    .T
)
```

	0	1	2	3	4	5	6
labels	B-ORG	I-LOC	I-ORG	B-LOC	B-PER	I-PER	O
count	2683	1462	3820	3172	2893	4139	43648
mean	0.66	0.64	0.48	0.35	0.26	0.18	0.03
sum	1769.47	930.94	1850.39	1111.03	760.56	750.91	1354.46

Wie wir sehen, verzeichnet `B-ORG` den höchsten durchschnittlichen Verlust, was bedeutet, dass die Bestimmung des Anfangs einer Organisation eine Herausforderung für unser Modell darstellt.

Um die Ergebnisse weiter aufzuschlüsseln, können wir eine Konfusionsmatrix erstellen, die uns Auskunft darüber gibt, wie gut die einzelnen Tokens klassifiziert wurden. Dabei stellen wir fest, dass der Anfang einer Organisation oft mit dem nachfolgenden Token `I-ORG` verwechselt wird:

```
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix

def plot_confusion_matrix(y_preds, y_true, labels):
    cm = confusion_matrix(y_true, y_preds, normalize="true")
    fig, ax = plt.subplots(figsize=(6, 6))
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=labels)
    disp.plot(cmap="Blues", values_format=".2f", ax=ax, colorbar=False)
    plt.title("Normalisierte Konfusionsmatrix")
    plt.show()

plot_confusion_matrix(df_tokens["labels"], df_tokens["predicted_label"],
                      tags.names)
```



In der Abbildung ist zu sehen, dass unser Modell am häufigsten die Entitäten `B-ORG` und `I-ORG` verwechselt. Abgesehen davon ist es jedoch recht gut darin, die übrigen Entitäten zu klassifizieren, was durch die deutliche Herausbildung einer Diagonalen in der Konfusionsmatrix deutlich wird.

Nachdem wir die Fehler auf Ebene der einzelnen Tokens untersucht haben, können wir nun die Sequenzen ermitteln, die hohe Verluste aufweisen. Dazu greifen wir auf unseren zuvor mithilfe der `pandas.Series.explode()`-Funktion erstellten `DataFrame` zurück und berechnen jeweils den Gesamtverlust für die einzelnen Sequenzen, indem wir die jeweiligen Verluste pro Token aufsummieren. Formulieren wir hierfür zunächst eine Funktion, die uns dabei hilft, uns die Labels und Verluste der einzelnen (aus Tokens bestehenden) Sequenzen anzeigen zu lassen:

```
def get_samples(df):
    for _, row in df.iterrows():
        labels, preds, tokens, losses = [], [], [], []
        for i, mask in enumerate(row["attention_mask"]):
            if i not in {0, len(row["attention_mask"])}:
                labels.append(row["labels"][i])
                preds.append(row["predicted_label"][i])
                tokens.append(row["input_tokens"][i])
                losses.append(f"{row['loss'][i]:.2f}")
        df_tmp = pd.DataFrame({"tokens": tokens, "labels": labels,
                               "preds": preds, "losses": losses}).T
        yield df_tmp
```

```
df["total_loss"] = df["loss"].apply(sum)
df_tmp = df.sort_values(by="total_loss", ascending=False).head(3)

for sample in get_samples(df_tmp):
    display(sample)
```

	0	1	2	3	4	...	13	14	15	16	17
tokens	_"	8	.	_Juli	_"	...	n	ischen	_Gar	de	</s>
labels	B-ORG	IGN	IGN	I-ORG	I-ORG	...	IGN	IGN	I-ORG	IGN	IGN
preds	0	0	0	0	0	...	I-ORG	I-ORG	I-ORG	I-ORG	0
losses	7.89	0.00	0.00	6.88	8.05	...	0.00	0.00	0.01	0.00	0.00

	0	1	2	3	4	...	14	15	16	17	18
tokens	_'	_"	_?	?	_"	...	k	_"	_'	ala	</s>
labels	0	0	0	IGN	0	...	IGN	I-LOC	I-LOC	IGN	IGN
preds	0	0	B-ORG	0	0	...	0	0	0	0	0
losses	0.00	0.00	3.59	0.00	0.00	...	0.00	7.66	7.78	0.00	0.00

	0	1	2	3	4	...	10	11	12	13	14
tokens	_United	_Nations	_Multi	dimensional	_Integra	...	_the	_Central	_African	_Republic	</s>
labels	B-PER	I-PER	I-PER	IGN	I-PER	...	I-PER	I-PER	I-PER	I-PER	IGN
preds	B-ORG	I-ORG	I-ORG	I-ORG	I-ORG	...	I-ORG	I-ORG	I-ORG	I-ORG	I-ORG
losses	6.46	5.59	5.51	0.00	5.11	...	4.77	5.32	5.10	4.87	0.00

Offensichtlich stimmt etwas mit den Labels dieser Beispiele nicht. Die Vereinten Nationen (engl. United Nations) und die Zentralafrikanische Republik (engl. Central African Republic) tragen zum Beispiel beide das Label, das für Personen steht. Gleichzeitig wird »8. Juli« im ersten Beispiel als eine Organisation ausgewiesen. Wie sich herausstellt, wurden die Annotationen für den PAN-X-Datensatz durch ein automatisiertes Verfahren erstellt. Solche Annotationen werden oft als »Silberstandard« (engl. Silver Standard) bezeichnet (im Gegensatz zum »Goldstandard«, nämlich den von Menschen erstellten Annotationen), und es überrascht nicht, dass es Fälle gibt, in denen der automatisierte Ansatz keine sinnvollen Labels geliefert hat. In der Tat sind solche Fehler nicht nur bei automatisierten Ansätzen zu beobachten. Fehler können selbst dann auftreten, wenn die Daten sorgfältig von Menschen annotiert bzw. gelabelt wurden, sei es, weil die Konzentration der annotierenden Person(en) nachgelassen oder sie den Satz einfach falsch verstanden hatten.

Darüber hinaus haben wir bereits festgestellt, dass Klammern und Schrägstriche einen relativ hohen Verlust aufweisen. Schauen wir uns ein paar Beispiele für Sequenzen an, die eine öffnende Klammer enthalten:

```
df_tmp = df.loc[df["input_tokens"].apply(lambda x: u"\u2581(" in x)].head(2)
for sample in get_samples(df_tmp):
    display(sample)
```

	0	1	2	3	4	5
tokens	_Ham	a	_(	_Unternehmen	_)	</s>
labels	B-ORG	IGN	I-ORG	I-ORG	I-ORG	IGN
preds	B-ORG	I-ORG	I-ORG	I-ORG	I-ORG	I-ORG
losses	0.01	0.00	0.01	0.01	0.01	0.00

	0	1	2	3	4	5	6	7
tokens	_Kesk	kül	a	_(	_Mart	na	_)	</s>
labels	B-LOC	IGN	IGN	I-LOC	I-LOC	IGN	I-LOC	IGN
preds	B-LOC	I-LOC	I-LOC	I-LOC	I-LOC	I-LOC	I-LOC	I-LOC
losses	0.02	0.00	0.00	0.01	0.01	0.00	0.01	0.00

Normalerweise würden wir Klammern und ihren Inhalt nicht als Teil der benannten Entität (bzw. des Eigennamens) einbeziehen. Allerdings scheinen die Dokumente im Rahmen der automatisierten Extraktion auf diese Art und Weise annotiert worden zu sein. In den anderen Beispielen enthalten die Klammern eine geografische Angabe. Auch wenn es sich hier um einen Ort handelt, möchten wir ihn in den Annotationen vielleicht nicht mit dem ursprünglichen Ort in Verbindung bringen. Dieser Datensatz besteht aus verschiedensprachigen Wikipedia-Artikeln, wobei die Artikelüberschriften oft eine Art Erklärung in Klammern enthalten. So weist im ersten Beispiel der Text in Klammern darauf hin, dass es sich bei Hama um ein Unternehmen handelt. Dies sind wichtige Details, die wir wissen müssen, wenn wir unser Modell in die Produktion bringen, da sie Auswirkungen auf die nachgelagerte Leistung der gesamten Pipeline haben können, zu der das Modell gehört.

Dank der relativ einfachen Analyse haben wir – sowohl in unserem Modell als auch in dem Datensatz – einige Schwachstellen ausgemacht. In einem echten Anwendungsfall würden wir diesen Schritt mehrfach wiederholen, den Datensatz bereinigen, das Modell neu trainieren und die neuen Fehler analysieren, bis wir mit der Leistung zufrieden sind.

Bisher haben wir die Fehler für eine einzelne Sprache analysiert. Wir sind jedoch auch an der sprachenübergreifenden Leistung interessiert. Im nächsten Abschnitt werden wir einige Experimente durchführen, um zu beurteilen, wie gut der sprachenübergreifende Transfer beim XLM-R-Modell funktioniert.

Sprachenübergreifender Transfer

Nachdem wir das XLM-R-Modell nun für die deutsche Sprache feingetunt haben, können wir mithilfe der `predict()`-Methode des `Trainer`-Objekts evaluieren, wie gut es sich auf andere Sprachen übertragen lässt (engl. Cross-Lingual Transfer). Da wir vorhaben, unser Modell im Hinblick auf mehrere Sprachen zu bewerten, erstellen wir hierfür eine einfache Funktion:

```
def get_f1_score(trainer, dataset):
    return trainer.predict(dataset).metrics["test_f1"]
```

Mit dieser Funktion können wir ermitteln, wie gut das Modell auf dem Testdatensatz abschneidet, und die Ergebnisse in einem Dictionary festhalten:

```
f1_scores = defaultdict(dict)
f1_scores["de"]["de"] = get_f1_score(trainer, panx_de_encoded["test"])
print(f"F1-Maß für [de]-Modell auf [de]-Datensatz: {f1_scores['de']['de']:.3f}")
```

```
F1-Maß für [de]-Modell auf [de]-Datensatz: 0.868
```

Das F_{-1}-Maß hat einen Wert von etwas mehr als 85 %, was ein recht guter Wert für eine NER-Aufgabe ist. Zudem können wir festhalten, dass das Modell bei den `ORG`-Entitäten am meisten zu kämpfen hat – wahrscheinlich, weil diese in den Trainingsdaten am seltensten vorkommen und zahlreiche Namen von Organisationen, die im Vokabular von XLM-R enthalten sind, nur selten verwendet werden. Aber wie gut schlägt sich das Modell bei den anderen Sprachen? Machen wir uns als Erstes ein Bild davon, wie unser auf Deutsch feingetuntes Modell auf Französisch abschneidet:

```
text_fr = "Jeff Dean est informaticien chez Google en Californie"
tag_text(text_fr, tags, trainer.model, xlmr_tokenizer)
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Tokens	<s>	_Jeff	_De	an	_est	_informatic	ien	_chez	_Google	_en	_Cali	for	nie	</s>
Tags	O	B-PER	I-PER	I-PER	O	O	O	O	B-ORG	O	B-LOC	I-LOC	I-LOC	O

Nicht übel! Obgleich der Name und die Organisation in beiden Sprachen gleich verwendet bzw. geschrieben werden, ist es dem Modell gelungen, die französische Übersetzung von »Kalifornien« korrekt zu labeln. Als Nächstes wollen wir herausfinden, wie gut unser deutsches Modell auf dem kompletten französischen Testdatensatz abschneidet, indem wir eine einfache Funktion erstellen, die einen Datensatz codiert und den resultierenden Klassifizierungsbericht (engl. Classification Report) erstellt:

```
def evaluate_lang_performance(lang, trainer):
    panx_ds = encode_panx_dataset(panx_ch[lang])
    return get_f1_score(trainer, panx_ds["test"])

f1_scores["de"]["fr"] = evaluate_lang_performance("fr", trainer)
print(f"F1-Maß für [de]-Modell auf [fr]-Datensatz: {f1_scores['de']['fr']:.3f}")
```

```
F1-Maß für [de]-Modell auf [fr]-Datensatz: 0.714
```

Auch wenn wir einen Rückgang von etwa 15 Prozentpunkten beim mikro-gemittelten (engl. micro-averaged) F_1-Maß verzeichnen, sollten wir uns vor Augen halten, dass unser Modell zuvor noch kein einziges gelabeltes französisches Beispiel gesehen hat! In der Regel hängt das Ausmaß des Leistungsabfalls davon ab, wie »weit« die Sprachen voneinander entfernt sind. Obwohl Deutsch und Französisch als indoeuropäische Sprachen betrachtet werden, gehören sie technisch gesehen zu verschiedenen Sprachfamilien, und zwar zu den germanischen bzw. romanischen Sprachen.

Sehen wir uns nun an, wie gut das Modell bei Italienisch abschneidet. Da Italienisch ebenfalls eine romanische Sprache ist, sollten wir ein ähnliches Ergebnis wie bei Französisch erzielen:

```
f1_scores["de"]["it"] = evaluate_lang_performance("it", trainer)
print(f"F1-Maß für [de]-Modell auf [it]-Datensatz: {f1_scores['de']['it']:.3f}")
```

```
F1-Maß für [de]-Modell auf [it]-Datensatz: 0.692
```

Der Wert des F_1-Maßes bestätigt in der Tat unsere Erwartung. Werfen wir abschließend noch einen Blick darauf, wie gut es sich bei Englisch schlägt, das zur germanischen Sprachfamilie gehört:

```
f1_scores["de"]["en"] = evaluate_lang_performance("en", trainer)
print(f"F1-Maß für [de]-Modell auf [en]-Datensatz: {f1_scores['de']['en']:.3f}")
```

```
F1-Maß für [de]-Modell auf [en]-Datensatz: 0.589
```

Überraschenderweise schneidet unser Modell im Englischen am *schlechtesten* ab, obwohl man rein intuitiv davon ausgehen würde, dass Deutsch dem Englischen ähnlicher ist als Französisch. Nachdem wir ein Modell für die deutsche Sprache feingetunt und dieses für einen »Zero-Shot«-Transfer auf Französisch, Italienisch und Englisch verwendet haben, wollen wir nun der Frage nachgehen, wann es sinnvoll ist, das Feintuning gleich direkt in der Zielsprache vorzunehmen.

Wann ist ein Zero-Shot-Transfer sinnvoll?

Bisher haben wir gesehen, dass das Feintuning des XLM-R-Modells auf dem deutschen Korpus zu einem Wert für das F_1-Maß von etwas mehr als 85 % führt. Unser Modell ist ohne *ein zusätzliches Training* in der Lage gewesen, eine recht ordentliche Leistung bei den anderen Sprachen in unserem Korpus zu erzielen. Hier stellt sich jedoch die Frage, wie gut diese Ergebnisse tatsächlich sind bzw. im Vergleich zu einem auf ein einsprachiges Korpus feingetunten XLM-R-Modell ausfallen.

In diesem Abschnitt werden wir auf das französische Korpus zurückgreifen, um dieser Frage nachzugehen, wobei wir das XLM-R-Modell mit immer größeren Trainingsdatensätzen feintunen. Auf diese Weise können wir feststellen, ab welchem Punkt der sprachenübergreifende »Zero-Shot«-Transfer überlegen ist. In der Praxis kann dies nützlich sein, um zu entscheiden, ob es sinnvoll ist, weitere gelabelte Daten zu sammeln.

Der Einfachheit halber übernehmen wir die Hyperparameter, die wir beim Feintuning des deutschen Korpus verwendet haben – mit der Ausnahme, dass wir das Argument `logging_steps` der `TrainingArguments`-Konfiguration anpassen, um der veränderten Größe der Trainingsdatensätze Rechnung zu tragen. Das Ganze können wir in einer einfachen Funktion zusammenfassen, die ein `DatasetDict`-Objekt entgegennimmt, das einem einsprachigen Korpus entspricht, eine Stichprobe der Größe `num_samples` zieht (Downsampling) und das XLM-R-Modell auf dieser Stichprobe feintunt und anschließend die Maße für die beste Epoche ausgibt:

```
def train_on_subset(dataset, num_samples):
    train_ds = dataset["train"].shuffle(seed=42).select(range(num_samples))
    valid_ds = dataset["validation"]
    test_ds = dataset["test"]
    training_args.logging_steps = len(train_ds) // batch_size

    trainer = Trainer(model_init=model_init, args=training_args,
        data_collator=data_collator, compute_metrics=compute_metrics,
        train_dataset=train_ds, eval_dataset=valid_ds, tokenizer=xlmr_tokenizer)
    trainer.train()
    if training_args.push_to_hub:
        trainer.push_to_hub(commit_message="Training completed!")

    f1_score = get_f1_score(trainer, test_ds)
    return pd.DataFrame.from_dict(
        {"num_samples": [len(train_ds)], "f1_score": [f1_score]})
```

Wie schon beim Feintuning des deutschen Korpus müssen wir auch das französische Korpus in Eingabe- bzw. Input-IDs, Attention-Masks und Label-IDs codieren:

```
panx_fr_encoded = encode_panx_dataset(panx_ch["fr"])
```

Als Nächstes überprüfen wir, ob unsere Funktion funktioniert, indem wir sie mit einem kleinen Trainingsdatensatz bestehend aus 250 Beispielen ausführen:

```
training_args.push_to_hub = False
metrics_df = train_on_subset(panx_fr_encoded, 250)
metrics_df
```

	num_samples	f1_score
0	250	0.137329

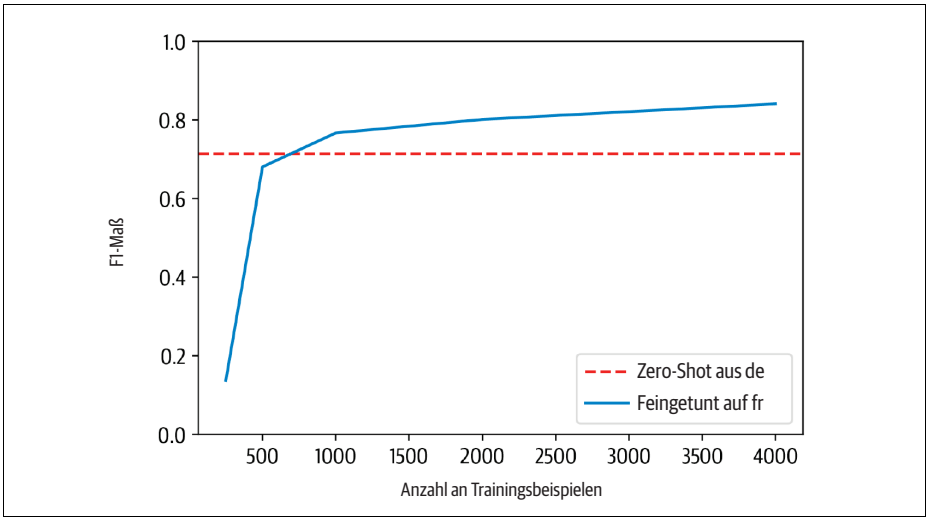
Mit nur 250 Beispielen schneiden wir mit dem Ansatz, das Feintuning auf Französisch durchzuführen, deutlich schlechter ab als beim »Zero-Shot«-Transfer aus dem Deutschen. Erhöhen wir nun die Größe unseres Trainingsdatensatzes auf 500, 1.000, 2.000 und 4.000 Beispiele, um einen Eindruck davon zu bekommen, wie sehr sich die Leistung verbessert:

```
for num_samples in [500, 1000, 2000, 4000]:
    metrics_df = metrics_df.append(
        train_on_subset(panx_fr_encoded, num_samples), ignore_index=True)
```

Um die Ergebnisse des Feintunings mit den französischen Stichproben mit denen des sprachenübergreifenden »Zero-Shot«-Transfers aus dem Deutschen einfacher

vergleichen zu können, können wir die auf dem Testdatensatz erzielten Werte für das F_1-Maß in Abhängigkeit von der Größe des Trainingsdatensatzes grafisch darstellen:

```
fig, ax = plt.subplots()
ax.axhline(f1_scores["de"]["fr"], ls="--", color="r")
metrics_df.set_index("num_samples").plot(ax=ax)
plt.legend(["Zero-Shot aus de", "Feingetunt auf fr"], loc="lower right")
plt.ylim((0, 1))
plt.xlabel("Anzahl an Trainingsbeispielen")
plt.ylabel("F1-Maß")
plt.show()
```



Aus der Grafik können wir schließen, dass der »Zero-Shot«-Transfer bis zu etwa 750 Trainingsbeispielen mithalten kann. Erst danach erreicht der Ansatz des Feintunings auf Französisch ein ähnliches Leistungsniveau wie das, das wir beim Feintuning auf Deutsch erreicht haben. Dennoch ist dieses Ergebnis bemerkenswert! Unserer Erfahrung nach kann es ziemlich kostspielig sein, selbst nur wenige Hunderte von Dokumenten von Fachleuten labeln zu lassen. Das gilt insbesondere für die NER, bei der der Annotationsprozess besonders detailliert und zeitaufwendig ist.

Schließlich gibt es noch eine letzte Technik, die wir ausprobieren können, um mehrsprachiges Lernen (engl. Multilingual Learning) zu evaluieren: Feintuning für mehrere Sprachen gleichzeitig! Mal sehen, wie gut das funktioniert.

Modelle für mehrere Sprachen gleichzeitig feintunen

Die bisherigen Analysen haben gezeigt, dass der sprachenübergreifende »Zero-Shot«-Transfer von Deutsch nach Französisch bzw. Italienisch einen Leistungsabfall von etwa 15 Prozentpunkten zur Folge hat. Eine Möglichkeit, diesen Effekt abzuschwächen, besteht darin, das Feintuning für mehrere Sprachen gleichzeitig durchzuführen. Um herauszufinden, wie groß die Verbesserungen sind, die wir er-

zielen können, verwenden wir zunächst die Funktion `concatenate_datasets()` aus der 🤗 Datasets-Bibliothek, um das deutsche und das französische Korpus zusammenzuführen:

```
from datasets import concatenate_datasets

def concatenate_splits(corpora):
    multi_corpus = DatasetDict()
    for split in corpora[0].keys():
        multi_corpus[split] = concatenate_datasets(
            [corpus[split] for corpus in corpora]).shuffle(seed=42)
    return multi_corpus

panx_de_fr_encoded = concatenate_splits([panx_de_encoded, panx_fr_encoded])
```

Für das Training verwenden wir wieder die gleichen Hyperparameter wie in den vorherigen Abschnitten. Dementsprechend müssen wir nur die Logging-Schritte, das Modell und die Datensätze aktualisieren, die wir der `Trainer`-Klasse übergeben:

```
training_args.logging_steps = len(panx_de_fr_encoded["train"]) // batch_size
training_args.push_to_hub = True
training_args.output_dir = "xlm-roberta-base-finetuned-panx-de-fr"

trainer = Trainer(model_init=model_init, args=training_args,
    data_collator=data_collator, compute_metrics=compute_metrics,
    tokenizer=xlmr_tokenizer, train_dataset=panx_de_fr_encoded["train"],
    eval_dataset=panx_de_fr_encoded["validation"])

trainer.train()
trainer.push_to_hub(commit_message="Training completed!")
```

Sehen wir uns nun an, wie das Modell auf den Testdatensätzen der einzelnen Sprachen abschneidet:

```
for lang in langs:
    f1 = evaluate_lang_performance(lang, trainer)
    print(f"F1-Maß für [de-fr]-Modell auf [{lang}]-Datensatz: {f1:.3f}")
```

```
F1-Maß für [de-fr]-Modell auf [de]-Datensatz: 0.866
F1-Maß für [de-fr]-Modell auf [fr]-Datensatz:  0.868
F1-Maß für [de-fr]-Modell auf [it]-Datensatz:  0.815
F1-Maß für [de-fr]-Modell auf [en]-Datensatz: 0.677
```

Auf dem französischen Testdatensatz schneidet das Modell bedeutend besser ab als zuvor und auf dem deutschen Testdatensatz erreicht es in etwa die gleiche Leistung. Interessanterweise erhöht sich die Qualität des Modells auch bei den italienischen und englischen Testdatensätzen um etwa 10 Prozentpunkte! Das bedeutet, dass Sie die Leistung des Modells auch bei unbekannten Sprachen verbessern könnten, wenn Sie zusätzliche Trainingsdaten aus einer anderen Sprache hinzufügen.

Zum Abschluss unserer Analyse untersuchen wir, wie sehr sich die Leistung unterscheidet, wenn wir das Feintuning für jede Sprache einzeln oder für mehrere Sprachen gleichzeitig (auf allen Korpora) vornehmen. Da wir das Feintuning mit dem deutschen Korpus bereits durchgeführt haben, können wir das Feintuning für die

übrigen Sprachen mithilfe unserer Funktion `train_on_subset()` durchführen, wobei `num_samples` der Anzahl der Beispiele im Trainingsdatensatz entspricht:

```
corpora = [panx_de_encoded]

# Deutsch von Iteration ausschließen
for lang in langs[1:]:
    training_args.output_dir = f"xlm-roberta-base-finetuned-panx-{lang}"
    # Feintuning mit einsprachigem Korpus
    ds_encoded = encode_panx_dataset(panx_ch[lang])
    metrics = train_on_subset(ds_encoded, ds_encoded["train"].num_rows)
    # Werte für F1-Maß in gemeinsamen Dictionary sammeln
    f1_scores[lang][lang] = metrics["f1_score"][0]
    # Einsprachiges Korpus zur Liste der zusammenzuführenden Korpora hinzufügen
    corpora.append(ds_encoded)
```

Nachdem wir nun das Feintuning für die einzelnen Sprachkorpora vorgenommen haben, müssen wir im nächsten Schritt alle Teildatensätze zusammenführen bzw. miteinander verketten (engl. concatenate), um ein mehrsprachiges Korpus für alle vier Sprachen zu erstellen. Wie bei der vorangegangenen Analyse für Deutsch und Französisch können wir die von uns erstellte Funktion `concatenate_splits()` verwenden, um diesen Schritt nun auch für unsere Liste von Korpora (`corpora`) durchzuführen:

```
corpora_encoded = concatenate_splits(corpora)
```

Nachdem wir nun unseren mehrsprachigen Korpus erstellt haben, führen wir die uns bereits bekannten Schritte im Zusammenhang mit dem `Trainer`-Objekt durch:

```
training_args.logging_steps = len(corpora_encoded["train"]) // batch_size
training_args.output_dir = "xlm-roberta-base-finetuned-panx-all"

trainer = Trainer(model_init=model_init, args=training_args,
    data_collator=data_collator, compute_metrics=compute_metrics,
    tokenizer=xlmr_tokenizer, train_dataset=corpora_encoded["train"],
    eval_dataset=corpora_encoded["validation"])

trainer.train()
trainer.push_to_hub(commit_message="Training completed!")
```

Der letzte Schritt besteht darin, die Vorhersagen des Trainers für die Testdatensätze der jeweiligen Sprachen zu ermitteln. Dadurch können wir uns ein Bild davon machen, wie gut das Multilingual Learning wirklich funktioniert. Wir sammeln die Werte für das F_1-Maß in unserem `f1_scores`-Dictionary und erstellen dann einen `DataFrame`, der die wichtigsten Ergebnisse unserer mehrsprachigen Experimente zusammenfasst:

```
for idx, lang in enumerate(langs):
    f1_scores["all"][lang] = get_f1_score(trainer, corpora[idx]["test"])

scores_data = {"de": f1_scores["de"],
               "each": {lang: f1_scores[lang][lang] for lang in langs},
               "all": f1_scores["all"]}
f1_scores_df = pd.DataFrame(scores_data).T.round(4)
f1_scores_df.rename_axis(index="Feingetunt mit", columns="Evaluiert auf",
                         inplace=True)
f1_scores_df
```

Evaluiert auf	de	fr	it	en
Feingetunt mit				
de	0.8677	0.7141	0.6923	0.5890
each	0.8677	0.8505	0.8192	0.7068
all	0.8682	0.8647	0.8575	0.7870

Aus diesen Ergebnissen können wir ein paar allgemeine Schlussfolgerungen ziehen:

- Multilingual Learning kann zu erheblichen Leistungssteigerungen führen. Dies passiert vor allem dann, wenn die Sprachen für den sprachenübergreifenden Transfer, für die wenig Ressourcen zur Verfügung stehen, zu ähnlichen Sprachfamilien gehören. In unseren Experimenten haben wir im Deutschen, Französischen und Italienischen ähnlich gute Ergebnisse in der Kategorie `all` erzielt, was darauf hindeutet, dass sich diese Sprachen untereinander stärker ähneln als jeweils dem Englischen.
- Im Allgemeinen empfiehlt es sich, das Augenmerk beim sprachenübergreifenden Transfer auf den Transfer *innerhalb* von Sprachfamilien zu richten, vor allem, wenn es sich um Sprachen mit anderen Schriftzeichen handelt, wie etwa beim Japanischen.

Interaktion mit den Modell-Widgets

In diesem Kapitel haben wir eine ganze Reihe feingetunter Modelle auf den Hub hochgeladen. Wenngleich wir die `pipeline()`-Funktion verwenden könnten, um mit ihnen auf unserem lokalen Rechner zu interagieren, bietet der Hub *Widgets*, die sich hervorragend für diese Art von Arbeitsabläufen eignen. In Abbildung 4-5 ist beispielsweise das Widget für unseren Checkpoint `transformersbook/xlm-roberta-base-finetuned-panx-all` dargestellt, der, wie Sie sehen können, alle Entitäten eines deutschen Texts treffsicher identifiziert hat.

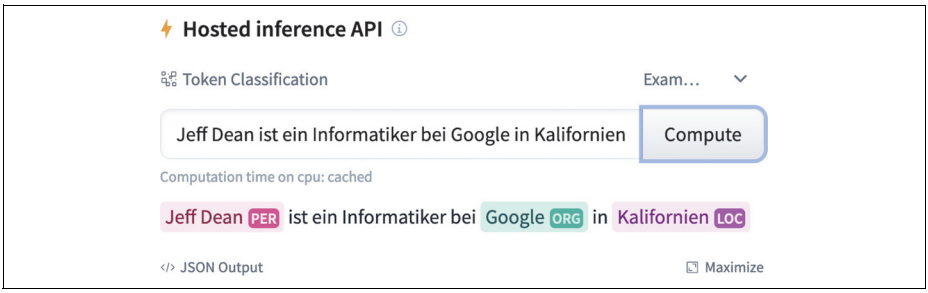


Abbildung 4-5: Beispiel für ein Widget auf dem Hugging Face Hub

Zusammenfassung

In diesem Kapitel haben wir untersucht, wie eine NLP-Aufgabe auf einem mehrsprachigen Korpus mithilfe eines einzelnen, auf 100 Sprachen vortrainierten Transformers gemeistert werden kann: mit dem XLM-R-Modell. Auch wenn wir zeigen konnten, dass der sprachenübergreifende Transfer vom Deutschen ins Französische durchaus mithalten kann, wenn nur eine geringe Anzahl von gelabelten Beispielen für das Feintuning zur Verfügung steht, wird in der Regel keine so gute Leistung erzielt, wenn sich die Zielsprache deutlich von der Sprache unterscheidet, mit der das Grundmodell feingetunt wurde, oder wenn sie nicht zu den 100 Sprachen gehört, auf die das Modell vortrainiert wurde. Neuere Lösungsansätze wie das MAD-X-Framework sind genau für diese Szenarien mit nur sehr wenig zur Verfügung stehenden Ressourcen konzipiert. Da das MAD-X-Framework auf der 🤗 Transformers-Bibliothek fußt, können Sie den Code in diesem Kapitel leicht anpassen und selbst damit arbeiten.[6]

Bis jetzt haben wir uns zwei Aufgaben angesehen: die Klassifizierung von Sequenzen und die Klassifizierung von Tokens. Beide fallen in den Bereich des Verstehens natürlicher Sprache (Natural Language Understanding), in dem Text zu Vorhersagen verarbeitet wird. Im nächsten Kapitel werfen wir einen ersten Blick auf die Textgenerierung, bei der nicht nur die Eingabe, sondern auch die Ausgabe des Modells einen Text darstellt.

6 J. Pfeiffer et al., »MAD-X: An Adapter-Based Framework for Multi-Task Cross-Lingual Transfer« (*https://arxiv.org/abs/2005.00052*), (2020).

KAPITEL 5
Textgenerierung

Eine der faszinierendsten Eigenschaften von Transformer-basierten Sprachmodellen ist ihre Fähigkeit, Texte generieren zu können, die von einem von Menschen verfassten Text fast nicht zu unterscheiden ist. Ein berühmtes Beispiel ist das GPT-2-Modell von OpenAI, das infolge der vorgegebenen Texteingabe bzw. Eingabeaufforderung (einem sogenannten Prompt):[1]

> In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

in der Lage war, einen überzeugenden Nachrichtenbeitrag über sprechende Einhörner zu verfassen:

> The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science. Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved. Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow. Pérez and the others then ventured further into the valley. »By the time we reached the top of one peak, the water looked blue, with some crystals on top,« said Pérez. Pérez and his friends were astonished to see the unicorn herd. These creatures could be seen from the air without having to move too much to see them—they were so close they could touch their horns. While examining these bizarre creatures the scientists discovered that the creatures also spoke some fairly regular English ...

Was dieses Beispiel so erstaunlich macht, ist der Umstand, dass es ohne explizite Überwachung (engl. Supervision) – also ohne, dass Labels erforderlich sind – erzeugt wurde! Indem sie einfach lernen, das nächste Wort in den Texten von Millionen von Webseiten vorherzusagen, sind GPT-2 und seine leistungsfähigeren Nachfolger wie GPT-3 in der Lage, sich eine breite Palette an Kompetenzen und

1 Dieses Beispiel stammt aus dem Blog-Post zum GPT-2-Modell (*https://openai.com/blog/better-language-models*) von OpenAI.

Mustererkennungsfähigkeiten anzueignen, die durch verschiedene Arten von Prompts aktiviert werden können. Abbildung 5-1 zeigt, wie Sprachmodelle während des Pretrainings manchmal mit einer Reihe von Aufgaben konfrontiert werden, bei denen sie die folgenden Tokens allein auf der Grundlage des Kontexts vorhersagen müssen, wie z. B. bei der Addition, der Entschlüsselung von Wörtern oder der Übersetzung. Auf diese Weise können sie dieses Wissen während des Feintunings bzw. (wenn das Modell groß genug ist) im Rahmen der Inferenz effektiv übertragen. Diese Aufgaben werden nicht im Voraus festgelegt, sondern kommen ganz natürlich in den riesigen Korpora vor, die zum Trainieren der Milliarden von Parametern umfassenden Sprachmodelle verwendet werden.

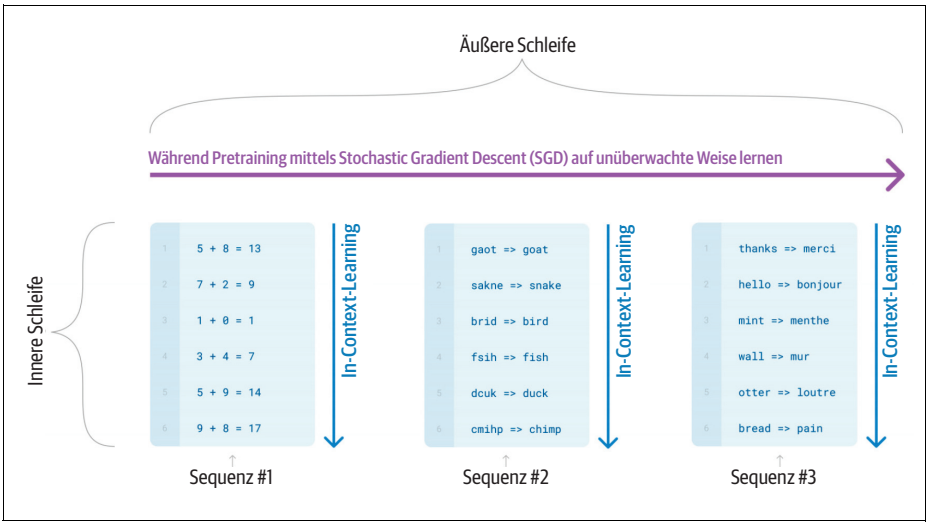


Abbildung 5-1: Während des Pretrainings werden die Sprachmodelle mit einer Reihe von Aufgaben konfrontiert, die dann im Rahmen der Inferenz adaptiert werden können (mit freundlicher Genehmigung von Tom B. Brown).

Die Fähigkeit von Transformer-Modellen, realistischen Text zu erzeugen, hat zu einer Vielzahl von Anwendungen geführt, wie InferKit (*https://oreil.ly/I4adh*), Write With Transformer (*https://oreil.ly/ipkap*), AI Dungeon (*https://oreil.ly/8ubC1*) und dialogfähige Systeme (sogenannte Conversational Agents) wie Googles Meena (*https://oreil.ly/gMegC*), die sogar abgedroschene Witze erzählen können (siehe Abbildung 5-2).[2]

In diesem Kapitel werden wir anhand des GPT-2-Modells veranschaulichen, was Sprachmodelle dazu befähigt, Text generieren zu können, und untersuchen, wie sich verschiedene Decodierungsstrategien auf die generierten Texte auswirken.

2 Wie Delip Rao angemerkt hat (*https://oreil.ly/mOM3V*), ist die Frage, ob Meena wirklich beabsichtigt, abgedroschene Witze zu erzählen, nicht ganz klar.

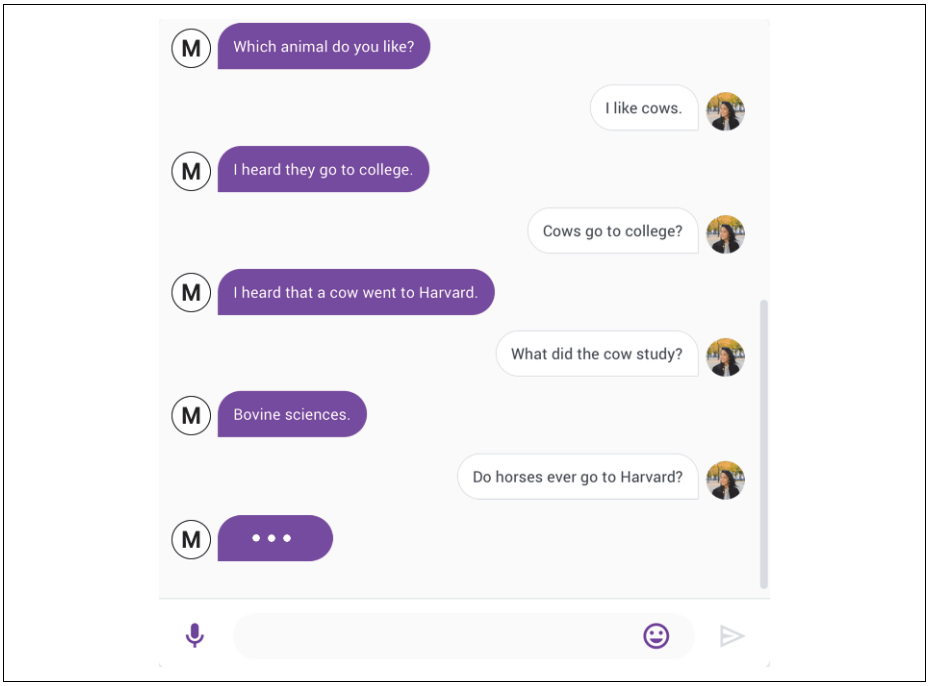


Abbildung 5-2: Meena (links) erzählt einem Menschen (rechts) einen abgedroschenen Witz (mit freundlicher Genehmigung von Daniel Adiwardana und Thang Luong).

Die Herausforderungen bei der Generierung von kohärenten Texten

Bisher haben wir uns in diesem Buch darauf konzentriert, NLP-Aufgaben zu bewältigen, indem wir ein vortrainiertes Modell heranziehen und dieses auf überwachte Weise (engl. Supervised) feintunen. Wie wir gesehen haben, ist es relativ einfach, Vorhersagen mithilfe aufgabenspezifischer Heads, die der Klassifizierung von Sequenzen oder Tokens dienen, zu erstellen. Das Modell liefert einige Logits, und wir nehmen entweder den Maximalwert, um die vorhergesagte Kategorie zu erhalten, oder wenden eine Softmax-Funktion an, um die vorhergesagten Wahrscheinlichkeiten für die einzelnen Kategorien zu erhalten. Im Gegensatz dazu erfordert die Umwandlung der probabilistischen Ausgabe des Modells in Text eine *Decodierungsmethode*, die einige Herausforderungen mit sich bringt, die ausschließlich im Rahmen der Textgenerierung auftreten:

- Die Decodierung erfolgt *iterativ* und ist daher wesentlich rechenintensiver als die einmalige Weiterleitung der Eingaben im Rahmen des Forward-Pass eines Modells.
- Die *Qualität* und die *Vielfältigkeit* des generierten Texts hängen von der Wahl der Decodierungsmethode und den zugehörigen Hyperparametern ab.

Um diesen Decodierungsprozess nachvollziehen zu können, sollten wir uns zunächst ansehen, wie GPT-2 vortrainiert und anschließend zur Generierung von Texten eingesetzt wird.

Wie andere *autoregressive* bzw. *kausale Sprachmodelle* wird auch GPT-2 vortrainiert, um – auf der Grundlage einer anfänglichen Texteingabe (Prompt) bzw. Kontextsequenz $\mathbf{x} = x_1, x_2, \ldots x_k$ – die Wahrscheinlichkeit $P(\mathbf{y}|\mathbf{x})$, dass eine Sequenz von Tokens $\mathbf{y} = y_1, y_2, \ldots y_t$ im Text vorkommt, zu schätzen. Da es praktisch nicht möglich ist, genügend Trainingsdaten zu erhalten, um $P(\mathbf{y}|\mathbf{x})$ direkt zu schätzen, ist es üblich, die Kettenregel für Wahrscheinlichkeiten anzuwenden, um sie als ein Produkt von *bedingten* Wahrscheinlichkeiten (engl. Conditional Probabilities) zu faktorisieren:

$$P(y_1, \ldots, y_t|\mathbf{x}) = \prod_{t=1}^{N} P(y_t|y_{<t}, \mathbf{x})$$

wobei $y_{<t}$ eine Kurzschreibweise für die Sequenz $y_1, \ldots, y_{t-1}$ ist. Anhand dieser bedingten Wahrscheinlichkeiten erkennen wir, dass es beim Autoregressive Language Modeling darum geht, jeweils einzelne Wörter auf Basis der vorangegangenen Wörter in einem Satz vorherzusagen (genau das sagt die Wahrscheinlichkeit auf der rechten Seite der vorangegangenen Gleichung aus). Beachten Sie, dass sich die Zielsetzung bzw. die Aufgabe, mit der das Modell vortainiert wird (Pretraining-Objective), deutlich von der bei BERT unterscheidet, bei dem sowohl *vergangene* als auch *zukünftige* Kontextinformationen verwendet werden, um ein *maskiertes* Token vorherzusagen.

Inzwischen haben Sie vielleicht schon erraten, wie sich diese Aufgabe zur Vorhersage des nächsten Tokens übertragen lässt, um Textsequenzen beliebiger Länge zu erzeugen. Wie in Abbildung 5-3 gezeigt, beginnen wir mit einem Prompt wie »Transformers are the«, auf dessen Grundlage das Modell anschließend das nächste Token vorhersagen soll.

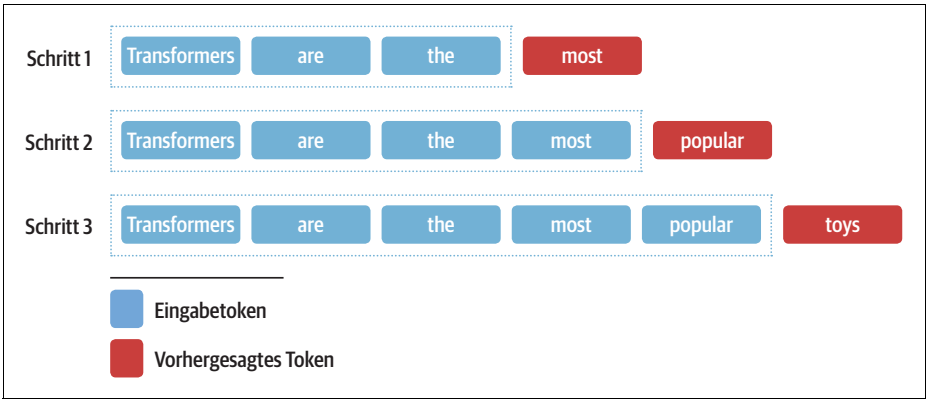


Abbildung 5-3: Text auf der Grundlage einer Eingabesequenz generieren, indem bei jedem Schritt ein neues Wort zur Eingabesequenz hinzugefügt wird

Sobald das nächste Token ermittelt wurde, wird es dem Prompt hinzugefügt und die resultierende neue Eingabesequenz verwendet, um ein weiteres Token zu erzeugen. Dieser Vorgang wird so lange wiederholt, bis ein spezielles Token, das das Ende der Sequenz kennzeichnet, vorhergesagt wird oder die vorgegebene Maximallänge erreicht wurde.

Da die Ausgabesequenz davon *abhängt*, welcher Prompt vorgegeben wurde, wird diese Art der Textgenerierung oft als *bedingte Textgenerierung* (engl. Conditional Text Generation) bezeichnet.

Der entscheidende Teil dieses Ansatzes ist eine Decodierungsmethode, die sich darauf auswirkt, welches Token in dem jeweiligen (Zeit-)Schritt ausgewählt wird. Da der Head des Sprachmodells bei jedem Schritt für jedes Token im Vokabular einen Logit $z_{t,i}$ liefert, können wir für das nächste mögliche Token eine Wahrscheinlichkeitsverteilung w_i ermitteln, indem wir die Softmax-Funktion anwenden:

$$P(y_t = w_i | y_{<t}, \mathbf{x}) = \text{softmax}(z_{t,i})$$

Bei den meisten Decodierungsmethoden wird das Ziel verfolgt, die wahrscheinlichste Gesamtsequenz zu ermitteln, indem $\hat{\mathbf{y}}$ so gewählt wird, dass:

$$\hat{y}_t = \underset{y_t}{\text{argmax}}\ P(y_t | y_{<t}, \mathbf{x})$$

Um $\hat{\mathbf{y}}$ direkt zu ermitteln, müssten alle möglichen Sequenzen mit dem Sprachmodell ausgewertet werden. Da es keinen Algorithmus gibt, der dies in angemessener Zeit bewerkstelligen kann, verlassen wir uns stattdessen auf Näherungswerte. In diesem Kapitel werden wir einige dieser Näherungen bzw. Approximationen erkunden und uns allmählich zu intelligenteren und komplexeren Algorithmen vorarbeiten, mit denen wir qualitativ hochwertige Texte erstellen können.

Greedy-Search-Decodierung

Der einfachste Ansatz, um im Rahmen der Decodierung die kontinuierliche Ausgabe eines Modells in diskrete Tokens zu überführen, besteht darin, bei jedem (Zeit-)Schritt einfach »gierig« (greedy) das wahrscheinlichste Token auszuwählen (d. h. dasjenige, für das die größte Wahrscheinlichkeit ermittelt wurde):

$$\hat{y}_t = \underset{y_t}{\text{argmax}}\ P(y_t | y_{<t}, \mathbf{x})$$

Laden wir zunächst die 1,5-Milliarden-Parameter-Version des GPT-2-Modells mit einem Sprachmodellierungs-Head (Language Modeling Head), um zu sehen, wie die »gierige« Suche (engl. Greedy Search) vonstattengeht:[3]

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM

device = "cuda" if torch.cuda.is_available() else "cpu"
model_name = "gpt2-xl"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name).to(device)
```

Machen wir uns nun ans Werk und generieren wir ein wenig Text! Obwohl die 🤗 Transformers-Bibliothek für autoregressive Modelle wie GPT-2 eigentlich eine `generate()`-Funktion bietet, werden wir die Decodierungsmethode selbst implementieren, um mehr darüber zu erfahren, was im Hintergrund vor sich geht. Zum Einstieg verwenden wir den gleichen iterativen Ansatz wie in Abbildung 5-3: Wir verwenden als Prompt »Transformers are the« und nehmen die Decodierung für acht Zeitschritte vor. Zu jedem Zeitschritt wählen wir die Logits des Modells aus, die für das letzte Token des Prompts vorliegen, und verarbeiten sie mithilfe einer Softmax-Funktion, um eine Wahrscheinlichkeitsverteilung zu erhalten. Anschließend wählen wir als nächstes Token das aus, das die höchste Wahrscheinlichkeit aufweist, fügen es der Eingabesequenz hinzu und führen den Vorgang erneut durch. Der folgende Code setzt genau dies um. Außerdem speichert er in jedem (Zeit-)Schritt die fünf wahrscheinlichsten Tokens, damit wir die verschiedenen Alternativen visualisieren können:

```
import pandas as pd

input_txt = "Transformers are the"
input_ids = tokenizer(input_txt, return_tensors="pt")["input_ids"].to(device)
iterations = []
n_steps = 8
choices_per_step = 5

with torch.no_grad():
    for _ in range(n_steps):
        iteration = dict()
        iteration["Input"] = tokenizer.decode(input_ids[0])
        output = model(input_ids=input_ids)
        # Logits des ersten Elements des Batches und des letzten
        #  Tokens auswählen und Softmax-Funktion anwenden
        next_token_logits = output.logits[0, -1, :]
        next_token_probs = torch.softmax(next_token_logits, dim=-1)
        sorted_ids = torch.argsort(next_token_probs, dim=-1, descending=True)
        # Tokens mit höchsten Wahrscheinlichkeiten speichern
        for choice_idx in range(choices_per_step):
            token_id = sorted_ids[choice_idx]
            token_prob = next_token_probs[token_id].cpu().numpy()
            token_choice = (
                f"{tokenizer.decode(token_id)} ({100 * token_prob:.2f}%)"
```

3 Wenn der auf Ihrem Rechner verfügbare Speicher nicht ausreicht, können Sie auch eine kleinere GPT-2-Version laden, indem Sie `model_name = "gpt2-xl"` durch `model_name = "gpt2"` ersetzen.

```
            )
            iteration[f"Choice {choice_idx+1}"] = token_choice
        # Nächstes vorhergesagtes Token an die Eingabe anfügen
        input_ids = torch.cat([input_ids, sorted_ids[None, 0, None]], dim=-1)
        iterations.append(iteration)

pd.DataFrame(iterations)
```

	Input	Choice 1	Choice 2	Choice 3	Choice 4	Choice 5
0	Transformers are the	most (8.53%)	only (4.96%)	best (4.65%)	Transformers (4.37%)	ultimate (2.16%)
1	Transformers are the most	popular (16.78%)	powerful (5.37%)	common (4.96%)	famous (3.72%)	successful (3.20%)
2	Transformers are the most popular	toy (10.63%)	toys (7.23%)	Transformers (6.60%)	of (5.46%)	and (3.76%)
3	Transformers are the most popular toy	line (34.38%)	in (18.20%)	of (11.71%)	brand (6.10%)	line (2.69%)
4	Transformers are the most popular toy line	in (46.28%)	of (15.09%)	, (4.94%)	on (4.40%)	ever (2.72%)
5	Transformers are the most popular toy line in	the (65.99%)	history (12.42%)	America (6.91%)	Japan (2.44%)	North (1.40%)
6	Transformers are the most popular toy line in the	world (69.26%)	United (4.55%)	history (4.29%)	US (4.23%)	U (2.30%)
7	Transformers are the most popular toy line in the world	, (39.73%)	. (30.64%)	and (9.87%)	with (2.32%)	today (1.74%)

Mit diesem einfachen Ansatz haben wir es geschafft, uns den Satz »Transformers are the most popular toy line in the world« generieren zu lassen. Interessanterweise deutet das Ergebnis darauf hin, dass GPT-2 ein gewisses Maß an Kenntnissen über das Medien-Franchise Transformers verinnerlicht hat, das von zwei Spielzeugfirmen (Hasbro und Takara Tomy) gegründet wurde. Darüber hinaus können wir nachvollziehen, welche anderen Möglichkeiten, den Text fortzusetzen, bei jedem Schritt zur Auswahl standen, was den iterativen Charakter der Textgenerierung verdeutlicht. Im Gegensatz zu anderen Aufgaben – wie der Klassifizierung von Sequenzen, bei denen ein einzelner Forward-Pass ausreicht, um die Vorhersagen zu erstellen – müssen wir bei der Textgenerierung die ausgegebenen Tokens Schritt für Schritt decodieren.

Obwohl es nicht allzu schwierig war, den Greedy-Search-Ansatz selbst zu implementieren, greifen wir im Folgenden nun doch auf die integrierte `generate()`-Funktion aus der 🤗 Transformers-Bibliothek zurück, um auch anspruchsvollere Decodierungsmethoden kennenlernen zu können. Damit Sie dieses einfache Beispiel reproduzieren können, müssen Sie sicherstellen, dass kein Sampling (d.h. keine Stichprobenziehung) erfolgt (bei der `generate()`-Funktion ist bereits als Voreinstellung kein Sampling vorgesehen, d.h. `do_sample=False`, allerdings kann die spezifische Konfiguration des Modells, dessen Checkpoint Sie laden, es anders vorsehen).

Zudem müssen Sie noch angeben, wie viele neue Tokens generiert werden sollen (max_new_tokens):

```
input_ids = tokenizer(input_txt, return_tensors="pt")["input_ids"].to(device)
output = model.generate(input_ids, max_new_tokens=n_steps, do_sample=False)
print(tokenizer.decode(output[0]))
```

```
Transformers are the most popular toy line in the world,
```

Probieren wir nun etwas noch Spannenderes aus: Gelingt es uns vielleicht, die Einhorngeschichte von OpenAI zu reproduzieren? Hierfür können wir denselben Prompt hernehmen, ihn wie gewohnt mit dem Tokenizer codieren und zusätzlich einen größeren Wert für max_length angeben, damit die generierte Textsequenz diesmal länger ausfällt:

```
max_length = 128
input_txt = """In a shocking finding, scientist discovered \
a herd of unicorns living in a remote, previously unexplored \
valley, in the Andes Mountains. Even more surprising to the \
researchers was the fact that the unicorns spoke perfect English.\n\n
"""
input_ids = tokenizer(input_txt, return_tensors="pt")["input_ids"].to(device)
output_greedy = model.generate(input_ids, max_length=max_length,
                               do_sample=False)
print(tokenizer.decode(output_greedy[0]))
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.

The researchers, from the University of California, Davis, and the University of
Colorado, Boulder, were conducting a study on the Andean cloud forest, which is
home to the rare species of cloud forest trees.

The researchers were surprised to find that the unicorns were able to
communicate with each other, and even with humans.

The researchers were surprised to find that the unicorns were able
```

Tja, die ersten paar Sätze unterscheiden sich ziemlich stark von dem Beispiel von OpenAI, und amüsanterweise werden hier andere Universitäten angeführt, denen die Entdeckung zugeschrieben wird. Wir sehen auch einen der Hauptnachteile der heuristischen Suche mittels Greedy-Algorithmus bei der Decodierung: Sie neigt dazu, sich wiederholende Ausgabesequenzen zu liefern, was in einem Nachrichtenartikel sicherlich nicht wünschenswert ist. Dies ist häufig ein Problem bei »gierigen« Suchalgorithmen, die unter Umständen nicht die optimale Lösung liefern: Im Zusammenhang mit der Decodierung können sie Wortfolgen verkennen bzw. außen vor lassen, deren Gesamtwahrscheinlichkeit an sich höher ist als die derjenigen, die letztlich gewählt wird, da den Wörtern mit hoher Wahrscheinlichkeit zufällig Wörter vorausgehen, die eine geringere Wahrscheinlichkeit aufweisen und dementsprechend vom Algorithmus, der jeweils auf die höchsten Einzelwahrscheinlichkeiten abzielt, nicht ausgewählt werden.

Glücklicherweise stehen uns noch bessere Ansätze zur Auswahl. Sehen wir uns ein beliebtes Verfahren an, das als *Beam Search Decoding* (Beam-Search-Decodierung) bekannt ist.

Obwohl die Greedy-Search-Decodierung nur selten für Textgenerierungsaufgaben verwendet wird, die eine gewisse Vielfalt erfordern, kann sie nützlich sein, um kurze Sequenzen wie arithmetische Folgen zu erzeugen, bei denen eine deterministische und sachlich korrekte Ausgabe erwünscht ist.[4] Für solche Aufgaben können Sie GPT-2 konditionieren, indem Sie einige zeilengetrennte Beispiele im Format `"5 + 8 => 13 \n 7 + 2 => 9 \n 1 + 0 =>"` als Prompt vorgeben.

Beam-Search-Decodierung

Anstatt bei jedem Schritt das Token zu decodieren, das die höchste Wahrscheinlichkeit aufweist, werden beim Beam-Search-Algorithmus die *b* wahrscheinlichsten nächsten Tokens festgehalten, wobei *b* als die Anzahl der *Beams* (auch Pfade oder Strahlen genannt) bzw. *Teilhypothesen* bezeichnet wird.[5] Der nächste Satz von Beams wird unter Berücksichtigung aller möglichen Weiterführungen des bestehenden Satzes ausgewählt und die *b* wahrscheinlichsten Weiterführungen ausgewählt. Der Vorgang wird wiederholt, bis die maximale Länge oder ein EOS-Token erreicht ist. Anschließend wird die wahrscheinlichste Sequenz ausgewählt, indem die *b* Beams entsprechend ihrer logarithmierten Wahrscheinlichkeiten (engl. Log Probabilities) geordnet werden. Ein Beispiel für den Beam-Search-Algorithmus finden Sie in Abbildung 5-4.

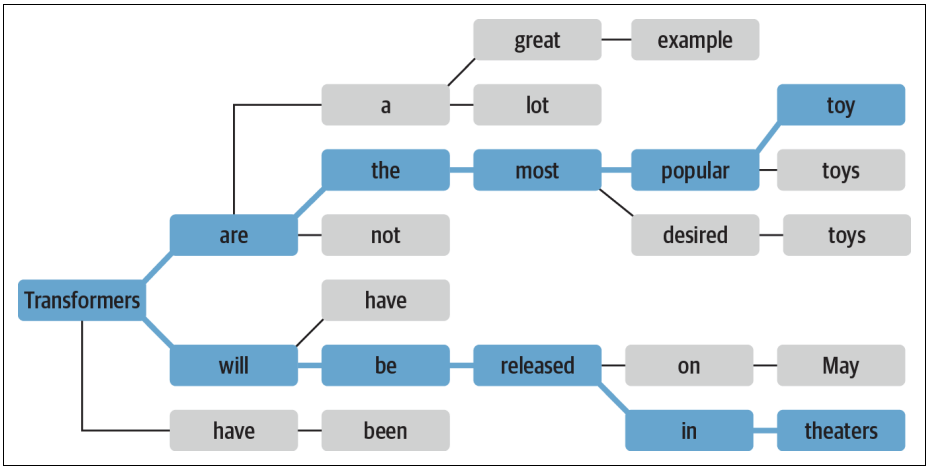


Abbildung 5-4: Beam-Search mit zwei Beams

4 N.S. Keskar et al., »CTRL: A Conditional Transformer Language Model for Controllable Generation« (*https://arxiv.org/abs/1909.05858*), (2019).

5 Anm. d. Übersetzers: Im Deutschen wird auch der Begriff Strahlensuche verwendet.

Warum werden die Sequenzen auf Basis der logarithmierten Wahrscheinlichkeiten ausgewertet und nicht auf Basis der Wahrscheinlichkeiten selbst? Ein Grund dafür ist, dass die Berechnung der Gesamtwahrscheinlichkeit einer Folge $P(y_1, y_2,..., y_t|\mathbf{x})$ die Berechnung eines *Produkts* bedingter Wahrscheinlichkeiten $P(y_1|y,..., y_{<t},\mathbf{x})$ beinhaltet. Da jede bedingte Wahrscheinlichkeit typischerweise eine niedrige Zahl im Intervall [0,1] darstellt, kann das Produkt zu einer Gesamtwahrscheinlichkeit führen, die möglicherweise äußerst klein ausfällt. Das bedeutet, dass der Computer das Ergebnis der Berechnung nicht mehr mit ausreichender numerischer Genauigkeit (engl. Precision) darstellen kann. Nehmen wir zum Beispiel an, wir haben eine Sequenz bzw. Folge von $t = 1024$ Tokens und treffen die einfache Annahme, dass die (Ziehungs-)Wahrscheinlichkeit für jedes Token 0,5 beträgt. Die Gesamtwahrscheinlichkeit, die sich für diese Sequenz ergibt, entspricht einer extrem kleinen Zahl:

```
0.5 ** 1024
```

```
5.562684646268003e-309
```

was zu numerischer Instabilität führt, da wir in einen arithmetischen Unterlauf geraten. Dies können wir vermeiden, indem wir einen verwandten Term berechnen, nämlich die logarithmierte Wahrscheinlichkeit. Wenn wir den Logarithmus auf die gemeinsamen und bedingten Wahrscheinlichkeiten anwenden, erhalten wir durch die Anwendung der Produktregel für Logarithmen:

$$\log P(y_1, ...y_t|\mathbf{x}) = \sum_{t=1}^{N} \log P(y_t|y_{<t}, \mathbf{x})$$

Mit anderen Worten, statt dem Produkt von Wahrscheinlichkeiten, das wir zuvor betrachtet haben, muss nun lediglich eine Summe aus logarithmierten Wahrscheinlichkeiten berechnet werden, bei der es wesentlich seltener zu numerischer Instabilität kommt. Wenn Sie zum Beispiel die logarithmierte Wahrscheinlichkeit für das eben gezeigte Beispiel berechnen, erhalten Sie:

```
import numpy as np
```

```
sum([np.log(0.5)] * 1024)
```

```
-709.7827128933695
```

Diese Zahl können wir problemlos handhaben. Zudem funktioniert der Ansatz auch bei deutlich kleineren Zahlen. Da wir nur daran interessiert sind, die Wahrscheinlichkeiten zu vergleichen, können wir den Vergleich direkt auf Basis der logarithmierten Wahrscheinlichkeiten anstellen.

Berechnen und vergleichen wir nun die logarithmierten Wahrscheinlichkeiten der generierten Texte beider Ansätze (Greedy-Search- und Beam-Search-Algorithmus), um zu ermitteln, ob wir mithilfe der Beam-Search Texte generieren können, die eine höhere Gesamtwahrscheinlichkeit aufweisen. Da die 🤗 Transformers-Bibliothek die nicht normalisierten Logits für das nächste Token (auf Basis einer gegebenen Sequenz von Eingabe-Tokens) zurückgibt, müssen wir zunächst noch die Log-

its normalisieren, um eine Wahrscheinlichkeitsverteilung für jedes Token in der Sequenz hinsichtlich des gesamten Vokabulars zu erstellen. Anschließend müssen wir nur die Wahrscheinlichkeiten der Tokens, die in der Sequenz vorkommen, auswählen. Die folgende Funktion implementiert diese Schritte:

```
import torch.nn.functional as F

def log_probs_from_logits(logits, labels):
    logp = F.log_softmax(logits, dim=-1)
    logp_label = torch.gather(logp, 2, labels.unsqueeze(2)).squeeze(-1)
    return logp_label
```

Auf diese Weise erhalten wir die logarithmierte Wahrscheinlichkeit für ein einzelnes Token. Um also die logarithmierte Gesamtwahrscheinlichkeit einer Sequenz zu erhalten, müssen wir nur die logarithmierten Wahrscheinlichkeiten der einzelnen Tokens addieren:

```
def sequence_logprob(model, labels, input_len=0):
    with torch.no_grad():
        output = model(labels)
        log_probs = log_probs_from_logits(
            output.logits[:, :-1, :], labels[:, 1:])
        seq_log_prob = torch.sum(log_probs[:, input_len:])
    return seq_log_prob.cpu().numpy()
```

Beachten Sie, dass wir die logarithmierten Wahrscheinlichkeiten, die der Eingabesequenz zuzuordnen sind, nicht auswählen, da diese Tokens nicht durch das Modell generiert worden sind. Zudem ist es wichtig, die Logits und die Labels noch anzupassen. Da das Modell das nächste Token vorhersagt, erhalten wir keinen Logit für das erste Label, und den letzten Logit benötigen wir nicht, da dafür kein Ground-Truth-Token vorliegt.

Mit diesen Funktionen können wir nun zunächst die logarithmierte Wahrscheinlichkeit der von dem Greedy-Algorithmus auf Basis des OpenAI-Prompts generierten Sequenz berechnen:

```
logp = sequence_logprob(model, output_greedy, input_len=len(input_ids[0]))
print(tokenizer.decode(output_greedy[0]))
print(f"\nlog-prob: {logp:.2f}")
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.

The researchers, from the University of California, Davis, and the University of
Colorado, Boulder, were conducting a study on the Andean cloud forest, which is
home to the rare species of cloud forest trees.

The researchers were surprised to find that the unicorns were able to
communicate with each other, and even with humans.

The researchers were surprised to find that the unicorns were able

log-prob: -87.43
```

Vergleichen wir dies nun mit einer Sequenz, die unter Verwendung der Beam-Search generiert wurde. Damit wir die Beam-Search mit der Funktion `generate()` nutzen können, müssen wir nur die Anzahl der Beams mit dem Parameter `num_beams` angeben. Umso mehr Beams wir wählen, desto besser könnte das Ergebnis potenziell ausfallen. Allerdings verlangsamt sich der Generierungsprozess erheblich, da mit steigender Anzahl an Beams auch mehr Sequenzen gleichzeitig generiert werden:

```
output_beam = model.generate(input_ids, max_length=max_length, num_beams=5,
                             do_sample=False)
logp = sequence_logprob(model, output_beam, input_len=len(input_ids[0]))
print(tokenizer.decode(output_beam[0]))
print(f"\nlog-prob: {logp:.2f}")
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.

The discovery of the unicorns was made by a team of scientists from the
University of California, Santa Cruz, and the National Geographic Society.

The scientists were conducting a study of the Andes Mountains when they
discovered a herd of unicorns living in a remote, previously unexplored valley,
in the Andes Mountains. Even more surprising to the researchers was the fact
that the unicorns spoke perfect English

log-prob: -55.23
```

Wie wir feststellen, erhalten wir mit der Beam-Search eine höhere logarithmierte Wahrscheinlichkeit (höher ist besser) als bei der einfachen Greedy-Decodierung. Allerdings besteht bei der Beam-Search-Decodierung ebenfalls das Problem, dass sich der Text wiederholt. Eine Möglichkeit, dies zu beheben, besteht darin, mittels des Parameters `no_repeat_ngram_size` eine *n*-Gramme-Penalisierung (engl. Penalty) zu verhängen, bei der nachvollzogen wird, welche *n*-Gramme bereits »gesehen« wurden. Die Wahrscheinlichkeit für das nächste Token wird auf null gesetzt, wenn ein bereits zuvor gesehenes *n*-Gramm generiert werden würde:

```
output_beam = model.generate(input_ids, max_length=max_length, num_beams=5,
                             do_sample=False, no_repeat_ngram_size=2)
logp = sequence_logprob(model, output_beam, input_len=len(input_ids[0]))
print(tokenizer.decode(output_beam[0]))
print(f"\nlog-prob: {logp:.2f}")
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.

The discovery was made by a team of scientists from the University of
California, Santa Cruz, and the National Geographic Society.

According to a press release, the scientists were conducting a survey of the
area when they came across the herd. They were surprised to find that they were
able to converse with the animals in English, even though they had never seen a
```

```
unicorn in person before. The researchers were

log-prob: -93.12
```

Gar nicht so schlecht! Wir haben es geschafft, dass es zu keinen Wiederholungen mehr kommt, und der inhaltliche Zusammenhang bzw. die Kohärenz des Texts bleibt erhalten, obwohl er eine niedrigere Wahrscheinlichkeit erzielt. Die Beam-Search mit *n*-Gramme-Penalisierung bietet einen guten Kompromiss zwischen der Fokussierung auf Tokens, die eine hohe Wahrscheinlichkeit aufweisen (mit Beam-Search), und der Reduzierung von Wiederholungen (mit *n*-Gramm-Penalisierung). Dieser Ansatz wird häufig bei Anwendungen wie der Textzusammenfassung oder der maschinellen Übersetzung verwendet, bei denen die sachliche Korrektheit wichtig ist. Wenn allerdings die sachliche Korrektheit weniger wichtig ist und besonders die Vielfalt der generierten Ausgabe im Vordergrund steht – z.B. beim Plaudern in Chats oder bei der Generierung von Geschichten –, gibt es einen weiteren Ansatz, mit dem sich Wiederholungen verringern lassen und gleichzeitig die Vielfalt verbessern lässt: die Verwendung von Stichprobenziehungsverfahren, oft auch als Sampling (aus dem Englischen) bezeichnet. Runden wir nun das Kapitel ab und widmen wir uns abschließend einigen der gängigsten Sampling-Verfahren zur Textgenerierung.

Sampling-Verfahren

Das einfachste Sampling-Verfahren beruht darauf, in jedem (Zeit-)Schritt zufällig ein Wort bzw. Token aus der Wahrscheinlichkeitsverteilung der Ausgaben des Modells über das gesamte Vokabular zu ziehen:

$$P(y_t = w_i | y_{<t}, \mathbf{x}) = \text{softmax}(z_{t,i}) = \frac{\exp(z_{t,i})}{\sum_{j=1}^{|V|} \exp(z_{t,j})}$$

wobei $|V|$ die Kardinalität des Vokabulars angibt. Wenn wir einen Temperature-Parameter T hinzufügen, mit dem die Logits vor Anwendung der Softmax-Funktion skaliert werden, können wir die Vielfältigkeit der Ausgabe auf einfache Weise steuern:

$$P(y_t = w_i | y_{<t}, \mathbf{x}) = \frac{\exp(z_{t,i}/T)}{\sum_{j=1}^{|V|} \exp(z_{t,j}/T)}$$

Je nachdem welchen Wert wir für T wählen, gestaltet sich die Form der Wahrscheinlichkeitsverteilung verschieden.[6] Wenn $T \ll 1$, dann verläuft die Verteilung steiler, wodurch die seltenen Tokens weniger wahrscheinlich gezogen werden (und

6 Wenn Sie sich ein wenig mit Physik auskennen, erkennen Sie vielleicht die verblüffende Ähnlichkeit zur Boltzmann-Verteilung (*https://oreil.ly/ZsMmx*).

die häufigen Tokens wahrscheinlicher). Wenn $T \gg 1$, dann verläuft die Verteilung hingegen flacher, d.h., die Ziehungswahrscheinlichkeiten der verschiedenen Tokens gleichen sich an. In Abbildung 5-5 können Sie sehen, wie unterschiedlich die Ziehungswahrscheinlichkeiten der Tokens bei verschiedenen Werten für die Temperature ausfallen.

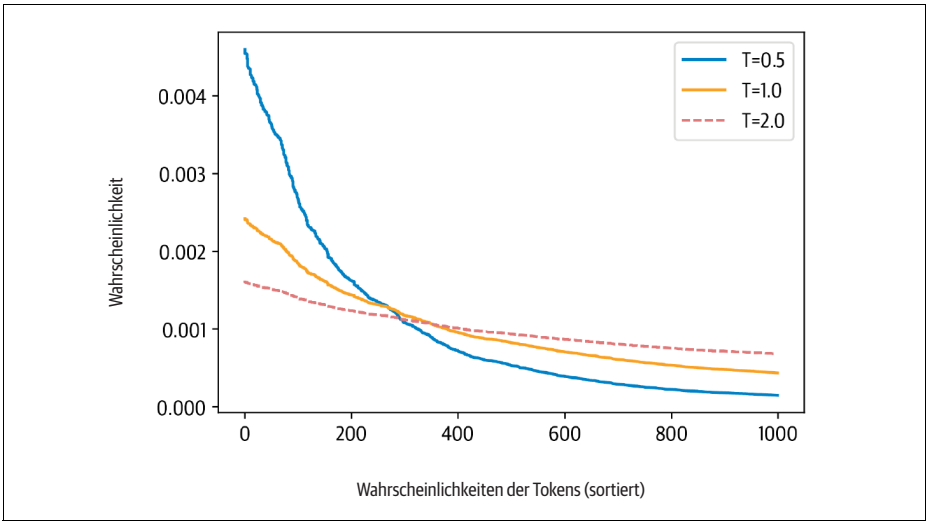


Abbildung 5-5: Die Wahrscheinlichkeitsverteilung zufällig gezogener Tokens für drei verschiedene Temperature-Werte

Untersuchen wir nun, wie der generierte Text durch die Wahl des Temperature-Parameters beeinflusst wird. Setzen wir für den Sampling-Prozess den `temperature`-Parameter in der `generate()`-Funktion zunächst auf $T = 2$ (die Bedeutung des Parameters `top_k` erläutern wir im nächsten Abschnitt):

```
output_temp = model.generate(input_ids, max_length=max_length, do_sample=True,
                             temperature=2.0, top_k=0)
print(tokenizer.decode(output_temp[0]))
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.


While the station aren protagonist receive Pengala nostalgiates tidbitRegarding
Jenny loclonju AgreementCON irrational ?rite Continent seaf A jer Turner
Dorbecue WILL Pumpkin mere Thatvernuildagain YoAniamond disse *
Runewitingkusstemprop});b zo coachinginventorymodules deflation press
Vaticanpres Wrestling chargesThingscutreddong Ty physician PET KimBi66 graz Oz
at aff da temporou MD6 radi iter
```

Wie wir unschwer feststellen können, bewirkt ein hoher Wert für den Temperature-Parameter, dass der Text größtenteils Kauderwelsch ist. Dadurch, dass wir die seltenen Tokens stärker berücksichtigen, haben wir das Modell dazu gebracht, eine

seltsame Grammatik zu verwenden und eine ganze Reihe erfundener Wörter zu generieren. Werfen wir einen Blick darauf, was passiert, wenn wir einen niedrigeren Wert für den Temperature-Parameter wählen:

```
output_temp = model.generate(input_ids, max_length=max_length, do_sample=True,
                             temperature=0.5, top_k=0)
print(tokenizer.decode(output_temp[0]))
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.


The scientists were searching for the source of the mysterious sound, which was
making the animals laugh and cry.


The unicorns were living in a remote valley in the Andes mountains

'When we first heard the noise of the animals, we thought it was a lion or a
tiger,' said Luis Guzman, a researcher from the University of Buenos Aires,
Argentina.


'But when
```

Dieser Text ist wesentlich stimmiger und enthält sogar ein Zitat eines Forschers einer weiteren Universität, der die Entdeckung zugeschrieben wird. Halten wir fest, dass wir mit dem Temperature-Parameter vor allem die Qualität der generierten Beispiele steuern können. Allerdings sind wir immer gezwungen, je nach Anwendungsfall zwischen Kohärenz (niedrige Temperature) und Vielfältigkeit (hohe Temperature) abzuwägen.

Eine andere Möglichkeit, den Zielkonflikt zwischen Kohärenz und Vielfältigkeit auszuloten, besteht darin, die Verteilung des Vokabulars zu stutzen (engl. truncate). Dadurch können wir die Vielfältigkeit mithilfe des Temperature-Parameters beliebig anpassen, allerdings nur in einem begrenzteren Rahmen, da Wörter ausgeschlossen werden, die in dem entsprechenden Kontext zu ungewöhnlich wären (d.h. Wörter mit geringer Wahrscheinlichkeit). Es gibt zwei Möglichkeiten, wie sich dies umsetzen lässt: Top-*k*- und Nucleus- bzw. Top-*p*-Sampling. Gehen wir es an!

Top-k- und Nucleus-Sampling

Top-*k*- und Nucleus-Sampling sind zwei beliebte Alternativen bzw. Erweiterungen zur Anwendung des Temperature-Parameters. In beiden Fällen besteht die Grundidee darin, die Anzahl der möglichen Tokens, aus denen wir zu jedem (Zeit-)Schritt ziehen können, zu begrenzen. Um die Funktionsweise dieses Ansatzes besser nachvollziehen zu können, sollten wir zunächst zwei Diagramme mit Wahrscheinlich-

keitsverteilungen der Ausgaben des Modells für den Fall $T = 1$ erstellen (siehe Abbildung 5-6).

Betrachten wir die beiden Diagramme im Einzelnen, da sie eine Menge Informationen enthalten. In dem oberen Diagramm sehen wir ein Histogramm der der Wahrscheinlichkeiten, mit der die Tokens vorhergesagt werden. Es hat einen Hochpunkt bei einer Wahrscheinlichkeit von 10^{-8} und einen zweiten, niedrigeren Hochpunkt bei 10^{4}. Anschließend fällt es relativ stark ab, d.h., dass es nur eine Handvoll von Tokens gibt, deren Wahrscheinlichkeit zwischen 10^{-2} und 10^{-1} liegt. Zudem sehen wir, dass die Wahrscheinlichkeit des Tokens mit der höchsten Wahrscheinlichkeit (der einzelne Balken bei 10^{-1}) 1 zu 10 bzw. 10 % beträgt.

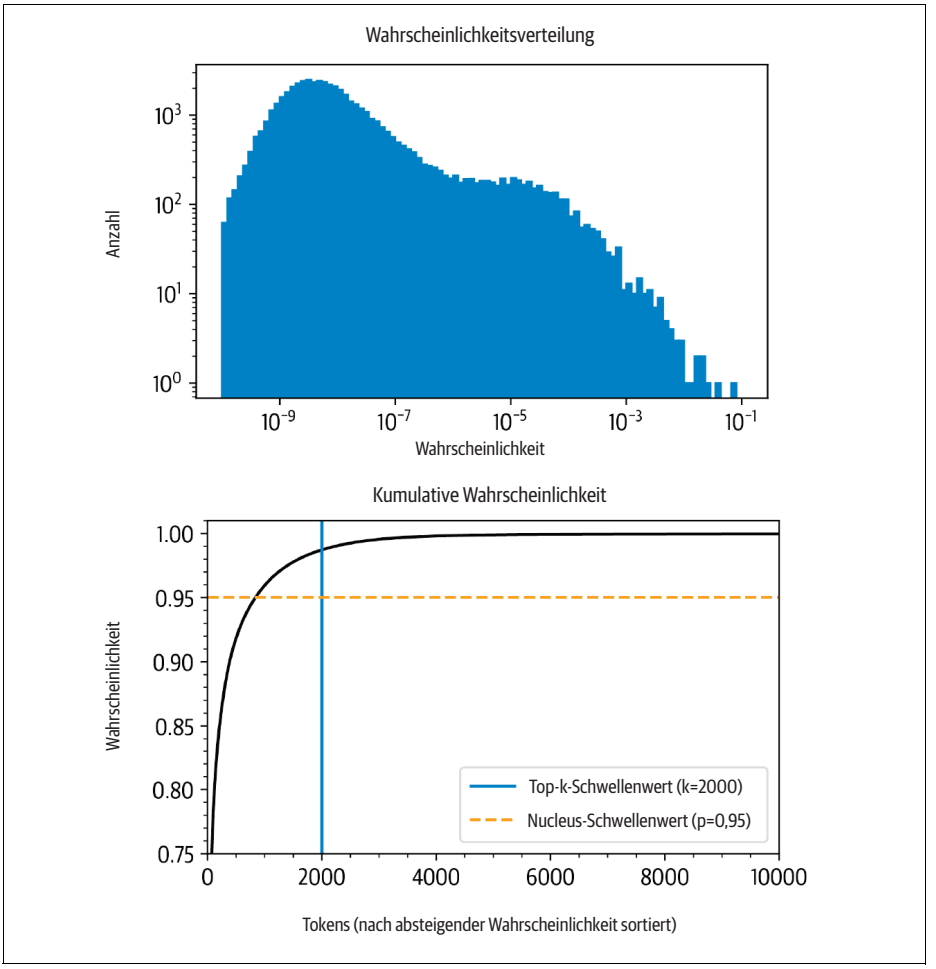


Abbildung 5-6: Histogramm der Vorhersagewahrscheinlichkeiten des nächsten Tokens (oben) und die kumulative Wahrscheinlichkeitsverteilung der Auswahlwahrscheinlichkeiten der Tokens (absteigend sortiert) (unten)

Im unteren Diagramm haben wir die Tokens in absteigender Wahrscheinlichkeit geordnet und die kumulative Wahrscheinlichkeit der ersten 10.000 Tokens berechnet (insgesamt gibt es 50.257 Tokens im Vokabular von GPT-2). Die Kurve stellt die Wahrscheinlichkeit dar, mit der eines der vorangegangenen Tokens ausgewählt wird. Die Wahrscheinlichkeit, eines der 1.000 Tokens mit der höchsten Wahrscheinlichkeit auszuwählen, liegt beispielsweise bei 96 %. Wie wir sehen, steigt die Wahrscheinlichkeit schnell über 90 %, erreicht aber erst nach mehreren Tausend Tokens die Sättigungsgrenze von nahezu 100 %. Die Grafik veranschaulicht, dass die Wahrscheinlichkeit, eines der Tokens zu wählen, die sich nicht unter den 2.000 wahrscheinlichsten befinden, bei 1 zu 100 bzw. 1 % liegt.

Auch wenn diese Zahlen auf den ersten Blick gering erscheinen mögen, sind sie doch von großer Bedeutung, da wir bei der Textgenerierung für jedes Token eine Stichprobenziehung bzw. ein Sampling vornehmen. Selbst wenn die Wahrscheinlichkeit nur 1 zu 100 oder 1.000 beträgt, besteht bei Hunderten von Ziehungen eine erhebliche Chance, dass wir irgendwann eines der unwahrscheinlichen Tokens ziehen. Wenn wir solche Tokens ziehen bzw. auswählen, kann dies die Qualität des generierten Texts in erheblichem Maße verringern. Aus diesem Grund möchten wir diese sehr unwahrscheinlichen Tokens in der Regel vermeiden. Genau an dieser Stelle kommen Top-*k*- und Top-*p*-Sampling ins Spiel.

Die Idee hinter dem Top-*k*-Sampling ist es, die Auswahlmöglichkeiten, die nur eine geringe Wahrscheinlichkeit aufweisen, zu vermeiden, indem wir nur aus den *k* Tokens mit der höchsten Wahrscheinlichkeit ziehen. Auf diese Weise wird der lange Rand der Verteilung eingegrenzt und sichergestellt, dass wir das Token jeweils nur aus wahrscheinlichen Wahlmöglichkeiten ziehen. Im Hinblick auf Abbildung 5-6 ist Top-*k*-Sampling gleichbedeutend damit, eine vertikale Linie zu ziehen und das Token jeweils nur aus den Tokens, die sich auf der linken Seite der Linie befinden, zu ziehen. Dieser Ansatz lässt sich ebenfalls auf einfache Weise mit der `generate()`-Funktion implementieren, indem Sie das Argument `top_k` angeben:

```
output_topk = model.generate(input_ids, max_length=max_length, do_sample=True,
                             top_k=50)
print(tokenizer.decode(output_topk[0]))
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.

The wild unicorns roam the Andes Mountains in the region of Cajamarca, on the
border with Argentina (Picture: Alamy/Ecole Nationale Supérieure d'Histoire
Naturelle)

The researchers came across about 50 of the animals in the valley. They had
lived in such a remote and isolated area at that location for nearly a thousand
years that
```

Unter den bisher von uns generierten Texten scheint dieser wohl von Menschen verfassten Texten am nächsten zu kommen. Doch welchen Wert wählen wir für *k*? Der Wert von *k* wird von Hand festgelegt und ist für jede Auswahl bzw. Ziehung

eines Tokens innerhalb der Sequenz derselbe, unabhängig von der jeweils aktuellen Verteilung der Ausgabe. Um einen guten Wert für *k* zu ermitteln, können wir einige Maße zur Beurteilung der Qualität des Texts heranziehen, die wir im nächsten Kapitel behandeln werden – allerdings könnte es sein, dass uns dieser fest vorgegebene »Cutoff«-Wert nicht sonderlich zufriedenstellt.

Alternativ können Sie auch einen *variablen* Cutoff-Wert verwenden. Beim Nucleus- bzw. Top-*p*-Sampling geben wir keinen festen Cutoff-Wert vor, sondern legen eine Bedingung für den Cutoff-Wert fest. Diese Bedingung ist erfüllt, wenn eine bestimmte kumulative Wahrscheinlichkeit hinsichtlich der ausgewählten Tokens erreicht ist. Nehmen wir an, wir setzen diesen Wert auf 95 %: Zunächst ordnen wir alle Tokens nach ihrer Wahrscheinlichkeit in absteigender Reihenfolge und fügen dann ein Token nach dem anderen vom Anfang der sortierten Liste hinzu, bis die Summe der Wahrscheinlichkeiten der ausgewählten Tokens 95 % beträgt. Bezogen auf Abbildung 5-6 definiert der Wert für *p* eine horizontale Linie im Diagramm (in dem die kumulative Wahrscheinlichkeit abgebildet ist), und wir ziehen nur aus den Tokens, die sich unterhalb dieser Linie befinden. Je nachdem, wie die Wahrscheinlichkeitsverteilung der Ausgabe ausfällt, kann dies ein einzelnes (sehr wahrscheinliches) Token sein oder hundert (eher gleich wahrscheinliche) Tokens. Vermutlich sind Sie jetzt nicht überrascht, dass die `generate()`-Funktion auch ein Argument bereitstellt, mit dem Sie das Top-*p*-Sampling vornehmen können. Probieren wir es gleich aus:

```
output_topp = model.generate(input_ids, max_length=max_length, do_sample=True,
                             top_p=0.90)
print(tokenizer.decode(output_topp[0]))
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.

The scientists studied the DNA of the animals and came to the conclusion that
the herd are descendants of a prehistoric herd that lived in Argentina about
50,000 years ago.

According to the scientific analysis, the first humans who migrated to South
America migrated into the Andes Mountains from South Africa and Australia, after
the last ice age had ended.

Since their migration, the animals have been adapting to
```

Offensichtlich können wir mit Top-*p*-Sampling ebenfalls eine in sich schlüssige Geschichte generieren, wobei es diesmal zu einer neuen Wendung kommt, was die Migration (Wanderung) von Australien nach Südamerika betrifft. Sie können die beiden Sampling-Ansätze sogar kombinieren und so das Beste aus beiden Ansätzen ziehen. Wenn Sie beispielsweise `top_k=50` und `top_p=0.9` setzen, entspricht dies der Regel, aus einem »Pool« (bzw. einer Grundgesamtheit) von Tokens, die die 90 % wahrscheinlichsten Tokens darstellen, höchstens 50 Tokens zu ziehen.

Darüber hinaus können wir auch die Beam-Search anwenden, wenn wir auf Sampling zurückgreifen. Anstatt die nächsten Token-Kandidaten auf »gierige« Weise auszuwählen, können wir sie auch stichprobenartig auswählen und die Beams auf gleiche Weise aufbauen.

Welcher Ansatz zur Decodierung ist der beste?

Leider gibt es keine allgemeingültig »beste« Decodierungsmethode. Welcher Ansatz der beste ist, hängt jeweils von der Art der Aufgabe ab, im Rahmen derer Sie Text generieren wollen. Wenn Sie möchten, dass Ihr Modell eine konkrete Aufgabe wie Rechnen oder die Beantwortung einer bestimmten Frage übernimmt, dann sollten Sie den Temperature-Parameter verringern oder deterministische Methoden wie die Greedy-Search in Kombination mit der Beam-Search verwenden, um die wahrscheinlichste Antwort zu erhalten. Wenn Sie möchten, dass das Modell längere Texte generiert und sogar ein wenig kreativ ist, dann sollten Sie auf Sampling-Verfahren zurückgreifen und den Temperature-Parameter erhöhen oder eine Mischung aus Top-*k*- und Nukleus-Sampling verwenden.

Zusammenfassung

In diesem Kapitel haben wir uns mit der Generierung von Text beschäftigt, einer Aufgabe, die sich stark von den NLU-Aufgaben unterscheidet, die wir zuvor kennengelernt haben. Die Textgenerierung erfordert mindestens einen Forward-Pass je generiertes Token, und sogar noch mehr, wenn wir die Beam-Search verwenden. Daher ist die automatische Textgenerierung sehr rechenintensiv, und man benötigt die richtige Infrastruktur, um ein Textgenerierungsmodell in großem Maßstab einsetzen zu können. Darüber hinaus kann eine gute Decodierungsstrategie, durch die die Ausgabewahrscheinlichkeiten des Modells in diskrete Tokens überführt werden, die Textqualität verbessern. Um herauszufinden, welche Decodierungsstrategie am besten geeignet ist, ist es erforderlich, einige Experimente durchzuführen und eine subjektive Bewertung der generierten Texte vorzunehmen.

In der Praxis sollten wir solche Entscheidungen jedoch nicht nur aus dem Bauch heraus treffen! Wie bei anderen NLP-Aufgaben auch, sollten wir ein Maß verwenden, mit dem wir die Qualität eines Modells beurteilen können, und das das Problem, das wir lösen möchten, auch gut abbildet. Wenig überraschend gibt es eine Vielzahl von Möglichkeiten. Die gängigsten werden Sie im nächsten Kapitel kennenlernen, in dem wir erkunden, wie sich ein Modell für die Textzusammenfassung trainieren und evaluieren lässt. Wenn Sie es nicht abwarten können, zu lernen, wie ein Modell vom Typ GPT von Grund auf trainiert werden kann, können Sie auch gleich zu Kapitel 10 vorspringen, in dem wir uns einen großen, aus Code bestehenden Datensatz beschaffen und dann ein autoregressives Sprachmodell damit trainieren.

KAPITEL 6

Automatische Textzusammenfassung (Summarization)

Wahrscheinlich mussten Sie schon einmal ein Dokument zusammenfassen, sei es ein Forschungsartikel, ein Finanzbericht oder eine längere E-Mail-Korrespondenz. Wenn man sich ein wenig länger damit befasst, wird einem bewusst, dass dies eine Reihe von Fähigkeiten erfordert, wie z. B. lange Passagen zu verstehen, sich Gedanken über den Inhalt zu machen oder einen flüssigen Text zu verfassen, der die wichtigsten Themen des Originaldokuments aufgreift. Darüber hinaus ist es etwas ganz anderes, einen Nachrichtenartikel oder einen juristischen Vertrag treffend zusammenzufassen, da dies ein hohes Maß an Fachwissen erfordert. Aus diesen Gründen ist die Zusammenfassung von Texten eine schwierige Aufgabe für auf neuronalen Netzen basierende Sprachmodelle, einschließlich Transformern. Trotz dieser Herausforderungen bietet die Textzusammenfassung Fachleuten die Möglichkeit, ihre Arbeitsabläufe erheblich zu beschleunigen. Die Vielfalt ihrer Anwendungsmöglichkeiten wird von Unternehmen genutzt, um beispielsweise internes Wissen zu bündeln, Verträge zusammenzufassen oder automatisch Inhalte für Veröffentlichungen in sozialen Medien zu generieren.

Damit Sie die damit verbundenen Herausforderungen besser einschätzen können, zeigen wir Ihnen in diesem Kapitel, wie Sie vortrainierte Transformer-Modelle für die Zusammenfassung von Dokumenten nutzen können. Die automatische Textzusammenfassung (engl. Summarization) ist eine typische Sequence-to-Sequence-(seq2seq-)Aufgabe mit einem Text als Eingabe und einem Text als Target bzw. Ausgabe, d. h., sowohl die Ein- als auch die Ausgabe stellen eine Sequenz dar. Wie wir bereits in Kapitel 1 in Erfahrung gebracht haben, haben hierbei vor allem Encoder-Decoder-basierte Transformer-Modelle ihre Stärken.

In diesem Kapitel werden wir unser eigenes Encoder-Decoder-Modell erstellen, um Dialoge zwischen mehreren Personen prägnant zusammenzufassen. Doch bevor wir dazu kommen, werfen wir zunächst einen Blick auf einen der anerkanntesten Datensätze für Zusammenfassungen: den CNN/DailyMail-Korpus.

Der CNN/DailyMail-Datensatz

Der CNN/DailyMail-Datensatz besteht aus etwa 300.000 Paaren von Nachrichtenartikeln und zugehörigen Zusammenfassungen, die stichpunktartige Zusammenfassungen darstellen, die CNN und DailyMail ihren Artikeln beifügen. Ein wichtiger Aspekt des Datensatzes ist, dass die Zusammenfassungen *abstraktiv* und nicht *extraktiv* sind, was bedeutet, dass sie aus neuen Sätzen und nicht aus einfachen Textauszügen bestehen. Der Datensatz ist auf dem Hugging Face Hub (*https://oreil.ly/jcRmb*) verfügbar, wobei wir die Version 3.0.0 verwenden – eine nicht anonymisierte Version, die für Zusammenfassungen vorgesehen ist. Die verschiedenen Versionen können wir auf ähnliche Weise auswählen wie die Teildatensätze (Splits), die wir in Kapitel 4 verwendet haben, und zwar mit dem Schlüsselwort `version`. Legen wir los und werfen wir einen Blick auf den Datensatz:

```
from datasets import load_dataset

dataset = load_dataset("cnn_dailymail", version="3.0.0")
print(f"Features: {dataset['train'].column_names}")
```

```
Features: ['article', 'highlights', 'id']
```

Der Datensatz hat drei Spalten: `article`, die Nachrichtenartikel enthält, `highlights`, in der die Zusammenfassungen enthalten sind, und `id`, die der eindeutigen Identifizierung jedes Artikels dient. Schauen wir uns einen Auszug eines Artikels an:

```
sample = dataset["train"][1]
print(f"""
Artikel (Auszug mit 500 Zeichen, Gesamtlänge: {len(sample["article"])}):
""")
print(sample["article"][:500])
print(f'\nZusammenfassung (Länge: {len(sample["highlights"])}):')
print(sample["highlights"])
```

```
Artikel (Auszug mit 500 Zeichen, Gesamtlänge: 3192):

(CNN) -- Usain Bolt rounded off the world championships Sunday by claiming his
third gold in Moscow as he anchored Jamaica to victory in the men's 4x100m
relay. The fastest man in the world charged clear of United States rival Justin
Gatlin as the Jamaican quartet of Nesta Carter, Kemar Bailey-Cole, Nickel
Ashmeade and Bolt won in 37.36 seconds. The U.S finished second in 37.56 seconds
with Canada taking the bronze after Britain were disqualified for a faulty
handover. The 26-year-old Bolt has n

Zusammenfassung (Länge: 180):
Usain Bolt wins third gold of world championship .
Anchors Jamaica to 4x100m relay victory .
Eighth gold at the championships for Bolt .
Jamaica double up in women's 4x100m relay .
```

Wie wir sehen, können die Artikel im Vergleich zur vorgegebenen (Ziel-)Zusammenfassung (engl. Target) sehr lang ausfallen. In diesem Fall ist der Artikel 17 Mal so lang. Für die meisten Transformer-Modelle stellen lange Artikel eine Herausfor-

derung dar, da die Kontextlänge (engl. Context Size) in der Regel auf ungefähr 1.000 Tokens begrenzt ist, was in etwa einigen Absätzen Text entspricht. Die gängige, wenn auch recht simple Methode, diesem Umstand bei der Zusammenfassung Rechnung zu tragen, besteht darin, den Text, der die Kontextlänge des Modells überschreitet, einfach wegzulassen bzw. zu stutzen. Natürlich könnten sich wichtige Informationen für die Zusammenfassung am Ende des Texts befinden, doch für den Moment müssen wir aufgrund der Architektur der Modelle mit dieser Einschränkung leben.

Pipelines für die automatische Textzusammenfassung

Verschaffen wir uns zunächst einen Überblick über die Leistung einiger der beliebtesten Transformer-Modelle zur Erstellung von Zusammenfassungen, indem wir die Qualität der Ergebnisse des vorangegangenen Beispiels genauer unter die Lupe nehmen. Da die Modellarchitekturen, die wir untersuchen werden, jeweils eine unterschiedliche maximale Länge für die Eingaben (engl. Input Size) vorsehen, beschränken wir den Eingabetext auf 2.000 Zeichen, um zu gewährleisten, dass alle Modelle die gleiche Eingabe erhalten und so die Ergebnisse vergleichbarer machen:

```
sample_text = dataset["train"][1]["article"][:2000]
# Wir sammeln die generierten Zusammenfassungen der Modelle in einem Dictionary
summaries = {}
```

Üblicherweise werden bei der automatischen Zusammenfassung die Sätze durch einen Zeilenumbruch getrennt. Wir könnten nach jedem Punkt einen Zeilenumbruch einfügen. Allerdings würde diese einfache Heuristik bei Strings wie »U.S.« oder »U.N.« nicht funktionieren. Das Natural-Language-Toolkit-(NLTK-)Paket enthält einen ausgefeilteren Algorithmus, der das Ende eines Satzes von Interpunktionszeichen, die in Abkürzungen vorkommen, unterscheiden kann:

```
import nltk
from nltk.tokenize import sent_tokenize

nltk.download("punkt")

string = "The U.S. are a country. The U.N. is an organization."
sent_tokenize(string)

['The U.S. are a country.', 'The U.N. is an organization.']
```

In den folgenden Abschnitten werden wir mehrere große Modelle laden. Sollten Sie nicht über ausreichend Speicher verfügen, können Sie die großen Modelle entweder durch kleinere Checkpoints ersetzen (z.B. `"gpt"`, `"t5-small"`) oder diesen Abschnitt überspringen und direkt zu »Evaluierung des PEGASUS-Modells auf dem CNN/DailyMail-Datensatz« auf Seite 190 übergehen. Zudem möchten wir Sie in diesem Zusammenhang nochmals auf die Möglichkeit der Nutzung der in der Einführung beschriebenen Plattformen hinweisen.

Ein einfacher Ansatz zur Textzusammenfassung

Ein einfacher erster Ansatz (Baseline-Ansatz) besteht darin, einfach die ersten drei Sätze eines Nachrichtenartikels als Zusammenfassung zu verwenden. Hierfür können wir erneut den Sentence-Tokenizer des NLTK-Pakets verwenden und unsere Baseline-Zusammenfassungen auf einfache Art und Weise erstellen:

```
def three_sentence_summary(text):
    return "\n".join(sent_tokenize(text)[:3])

summaries["baseline"] = three_sentence_summary(sample_text)
```

GPT-2

In Kapitel 5 haben wir bereits gesehen, dass sich das GPT-2-Modell dafür verwenden lässt, infolge eines vorgegebenen Eingabetexts bzw. Prompts einen Text zu generieren. Eine der überraschenden Eigenschaften des Modells ist, dass wir es auch dazu bringen können, Zusammenfassungen zu erzeugen, indem wir einfach »TL;DR« an das Ende des Prompts anhängen. Der Ausdruck »TL;DR« (too long; didn't read) wird häufig auf Plattformen wie Reddit verwendet, um eine Kurzfassung eines langen Beitrags mit anzugeben. Beginnen wir mit unserem Textzusammenfassungsexperiment, indem wir den Ansatz, der im Originalbeitrag verfolgt wurde, mithilfe der `pipeline()`-Funktion aus der 🤗 Transformers-Bibliothek nachbilden.[1] Erstellen wir nun eine Textgenerierungspipeline und laden wir das große GPT-2-Modell:

```
from transformers import pipeline, set_seed

set_seed(42)
pipe = pipeline("text-generation", model="gpt2-xl")
gpt2_query = sample_text + "\nTL;DR:\n"
pipe_out = pipe(gpt2_query, max_length=512, clean_up_tokenization_spaces=True)
summaries["gpt2"] = "\n".join(
    sent_tokenize(pipe_out[0]["generated_text"][len(gpt2_query) :]))
```

Wir speichern lediglich die Zusammenfassungen, die jeweils im generierten Text enthalten sind. Das bedeutet, dass wir den ebenfalls enthaltenen vorgegebenen Eingabetext (hier als Query bezeichnet) mithilfe von Slicing außen vor lassen, sodass wir nur den neu generierten Text (also die generierte Zusammenfassung) auswählen, um diesen in einem Python-Dictionary für einen späteren Vergleich zu speichern.

T5

Probieren wir als Nächstes das T5-Transformer-Modell aus. Wie wir in Kapitel 3 bereits erfahren haben, haben die Entwickler dieses Modells eine umfassende Studie zum Transfer Learning im Bereich des NLP durchgeführt und dabei festgestellt,

1 A. Radford et al., »Language Models Are Unsupervised Multitask Learners« (*https://openai.com/blog/better-language-models*), OpenAI (2019).

dass sie eine universell einsetzbare Transformer-Architektur entwickeln können, indem sie alle Aufgaben als Text-to-Text-Aufgaben formulieren. Die Checkpoints des T5-Modells wurden sowohl mit ungelabelten Daten auf unüberwachte (engl. Unsupervised) Weise (maskierte Wörter rekonstruieren) als auch auf Basis gelabelter Daten auf überwachte (engl. Supervised) Weise für verschiedene Aufgaben, einschließlich der Textzusammenfassung, trainiert. Die Checkpoints können daher direkt und ohne weiteres Feintuning zur Erstellung von Zusammenfassungen verwendet werden, indem dafür dieselben Prompts wie beim Pretraining vorgegeben werden. Im Rahmen dieses Frameworks muss der Prompt dem Schema `"summarize: <ARTICLE>"` folgen, um eine Zusammenfassung eines Dokuments zu erstellen, und beispielsweise `"translate English to German: <TEXT>"`, um eine Übersetzung vornehmen zu lassen. Wie in Abbildung 6-1 gezeigt, ist das T5-Modell daher extrem vielseitig und gestattet es Ihnen, viele Aufgaben mit nur einem einzelnen Modell bewältigen zu können.

Wir können den für die Textzusammenfassung vorgesehenen T5-Checkpoint direkt mit der `pipeline()`-Funktion laden, die sich auch um die Formatierung der Eingaben in das Text-to-Text-Format kümmert, sodass wir ihnen nicht das Wort `"summarize"` voranstellen müssen:

```
pipe = pipeline("summarization", model="t5-large")
pipe_out = pipe(sample_text)
summaries["t5"] = "\n".join(sent_tokenize(pipe_out[0]["summary_text"]))
```

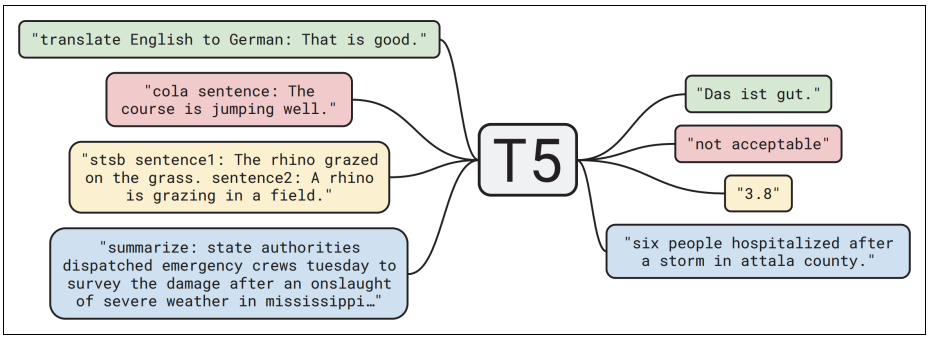


Abbildung 6-1: Darstellung des Text-to-Text-Frameworks von T5 (mit freundlicher Genehmigung von Colin Raffel). Zusätzlich zur Übersetzung und Zusammenfassung sind die Aufgaben CoLA (linguistische Akzeptabilität, engl. Linguistic Acceptability) und STSB (semantische Nähe, engl. Semantic Similarity) dargestellt.

BART

Das BART-Modell verwendet ebenfalls eine Encoder-Decoder-Architektur und ist darauf trainiert, verfälschte Eingaben zu rekonstruieren. Es basiert auf einer Kombination der Pretraining-Verfahren, die bei BERT und GPT-2 zum Einsatz kommen.[2] Wir verwenden den Checkpoint `facebook/bart-large-ccn`, der speziell mit dem CNN/DailyMail-Datensatz feingetunt wurde:

2 M. Lewis et al., »BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension« (*https://arxiv.org/abs/1910.13461*), (2019).

```
pipe = pipeline("Automatische Textzusammenfassung",
                model="facebook/bart-large-cnn")
pipe_out = pipe(sample_text)
summaries["bart"] = "\n".join(sent_tokenize(pipe_out[0]["summary_text"]))
```

PEGASUS

Wie BART ist auch PEGASUS ein Encoder-Decoder-basiertes Transformer-Modell.[3] Das Ziel beim Pretraining (Pretraining-Objective) besteht darin, maskierte Sätze in Texten, die mehrere Sätze umfassen, vorherzusagen (siehe Abbildung 6-2). Die Autoren sind der Auffassung, dass das Pretraining umso effektiver ist, je ähnlicher die beim Pretraining verfolgte Aufgabe (Objective) der nachgelagerten Aufgabe ist. Auf der Suche nach einem Pretraining-Objective, das der Zusammenfassung stärker gleicht als die allgemeine Sprachmodellierung, war es ihnen möglich, in einem sehr großen Korpus automatisch die Sätze zu identifizieren, die den größten Teil des Inhalts der sie umgebenden Absätze enthalten (unter Verwendung von Evaluierungsmaßen für die Zusammenfassung als Heuristik für inhaltliche Überschneidungen) und das PEGASUS-Modell darauf vorzutrainieren, diese Sätze zu rekonstruieren und so ein hochmodernes Modell für die Textzusammenfassung zu erhalten.

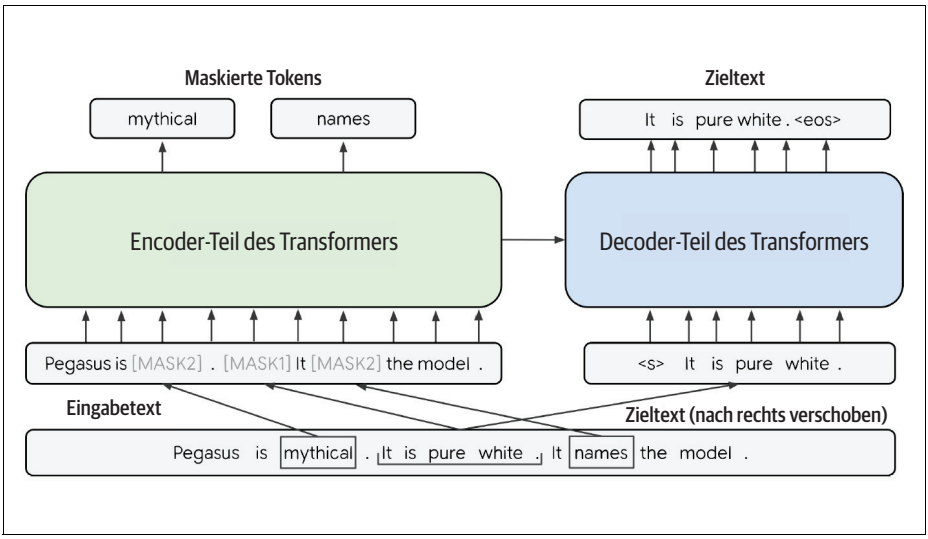


Abbildung 6-2: Schematische Darstellung der PEGASUS-Architektur (mit freundlicher Genehmigung von Jingqing Zhang et al.)

In diesem Modell gibt es ein spezielles Token für Zeilenumbrüche, weshalb wir in diesem Fall nicht auf die `sent_tokenize()`-Funktion zurückgreifen müssen:

3 J. Zhang et al., »PEGASUS: Pre-Training with Extracted Gap-Sentces for Abstractive Summarization« (*https://arxiv.org/abs/1912.08777*), (2019).

```
pipe = pipeline("Automatische Textzusammenfassung", model="google/pegasus-cnn_
                 dailymail")
pipe_out = pipe(sample_text)
summaries["pegasus"] = pipe_out[0]["summary_text"].replace(" .<n>", ".\n")
```

Verschiedene Zusammenfassungen vergleichen

Nachdem wir nun mit vier verschiedenen Modellen Zusammenfassungen erstellt haben, ist es an der Zeit, die Ergebnisse miteinander zu vergleichen. Beachten Sie, dass ein Modell überhaupt nicht auf dem Datensatz trainiert wurde (GPT-2), ein Modell nur unter anderem für diese Aufgabe feingetunt wurde (T5) und zwei Modelle ausschließlich für diese Aufgabe feingetunt wurden (BART und PEGASUS). Werfen wir einen Blick auf die Zusammenfassungen, die die Modelle generiert haben:

```
print("REFERENZZUSAMMENFASSUNG (GROUND TRUTH)")
print(dataset["train"][1]["highlights"])
print("")

for model_name in summaries:
    print(model_name.upper())
    print(summaries[model_name])
    print("")
```

```
REFERENZZUSAMMENFASSUNG (GROUND TRUTH)
Usain Bolt wins third gold of world championship .
Anchors Jamaica to 4x100m relay victory .
Eighth gold at the championships for Bolt .
Jamaica double up in women's 4x100m relay .

BASELINE
(CNN) -- Usain Bolt rounded off the world championships Sunday by claiming his
third gold in Moscow as he anchored Jamaica to victory in the men's 4x100m
relay.
The fastest man in the world charged clear of United States rival Justin Gatlin
as the Jamaican quartet of Nesta Carter, Kemar Bailey-Cole, Nickel Ashmeade and
Bolt won in 37.36 seconds.
The U.S finished second in 37.56 seconds with Canada taking the bronze after
Britain were disqualified for a faulty handover.

GPT2
Nesta, the fastest man in the world.
Gatlin, the most successful Olympian ever.
Kemar, a Jamaican legend.
Shelly-Ann, the fastest woman ever.
Bolt, the world's greatest athlete.
The team sport of pole vaulting

T5
usain bolt wins his third gold medal of the world championships in the men's
4x100m relay .
the 26-year-old anchored Jamaica to victory in the event in the Russian capital.
he has now collected eight gold medals at the championships, equaling the record.
```

```
BART
Usain Bolt wins his third gold of the world championships in Moscow.
Bolt anchors Jamaica to victory in the men's 4x100m relay.
The 26-year-old has now won eight gold medals at world championships.
Jamaica's women also win gold in the relay, beating France in the process.

PEGASUS
Usain Bolt wins third gold of world championships.
Anchors Jamaica to victory in men's 4x100m relay.
Eighth gold at the championships for Bolt.
Jamaica also win women's 4x100m relay .
```

Das Erste, was uns bei der Betrachtung der Modellausgaben auffällt, ist, dass die von GPT-2 generierte Zusammenfassung ganz anders als die anderen ausfällt. Anstatt den Text zusammenzufassen, fasst es die Charaktere zusammen. Oft »halluziniert« das GPT-2-Modell oder erfindet Fakten, da es nicht explizit darauf trainiert wurde, wahrheitsgemäße Zusammenfassungen zu erstellen. Zum Beispiel ist Nesta zum Zeitpunkt des Verfassens des Texts nicht der schnellste Mann der Welt, sondern liegt auf dem neunten Platz. Wenn wir die anderen drei Modellzusammenfassungen mit der sogenannten Ground Truth (d.h. den tatsächlichen Zusammenfassungen bzw. Referenzzusammenfassungen, die in dem Datensatz enthalten sind) vergleichen, sehen wir, dass es bemerkenswerte Überschneidungen gibt, wobei der vom PEGASUS-Modell ausgegebene Text die größte Ähnlichkeit aufweist.

Nachdem wir nun einige Modelle in Augenschein genommen haben, sollten wir herausfinden, welches wir auch in einer Produktionsumgebung verwenden würden. Alle vier Modelle scheinen qualitativ vernünftige Ergebnisse zu liefern, und wir könnten noch ein paar weitere Beispiele generieren, die uns bei der Entscheidung helfen. Allerdings würden wir damit keinen systematischen Ansatz verfolgen, um das beste Modell zu ermitteln! Idealerweise würden wir ein Maß festlegen, dieses für alle Modelle anhand eines Datensatzes, der als Benchmark fungiert, ermitteln und dann das Modell, das die beste Leistung aufweist, auswählen. Aber welches Maß könnte im Rahmen der Textgenerierung sinnvollerweise verwendet werden? Die uns bekannten Standardmaße wie Treffergenauigkeit (engl. Accuracy), Recall und Precision lassen sich nicht so einfach auf diese Aufgabe anwenden. Für jede »Goldstandard«-Zusammenfassung, die von einem Menschen formuliert wurde, könnten Dutzende anderer Zusammenfassungen mit Synonymen, Paraphrasen oder einer leicht abweichenden Formulierung der Fakten ebenso akzeptabel sein.

Im nächsten Abschnitt werden wir uns einige gängige Maße ansehen, die zur Evaluierung der Qualität generierter Texte entwickelt worden sind.

Evaluierung der Qualität von generierten Texten

Auf adäquate Maße zur Evaluierung zurückzugreifen, ist von zentraler Bedeutung, da sie nicht nur beim Training, sondern auch später in der Produktion herangezogen werden, um die Qualität der Modelle zu beurteilen. Wenn wir Maße zugrunde

legen, die nicht adäquat sind, könnte es uns unter Umständen entgehen, wenn sich ein Modell verschlechtert (engl. Model Degradation), und wenn sie nicht mit den Geschäftszielen übereinstimmen, könnte es passieren, dass wir keinen Beitrag zur Erreichung dieser leisten.

Die Leistung im Rahmen einer Textgenerierungsaufgabe zu beurteilen, gestaltet sich nicht so einfach wie bei standardmäßigen Klassifizierungsaufgaben wie der Sentimentanalyse oder der Named Entity Recognition. Wenn Sie beispielsweise einen Satz wie »I love dogs!« auf Englisch ins Spanische übersetzen, gibt es mehrere gültige Varianten, etwa »¡Me encantan los perros!« oder »¡Me gustan los perros!« Einfach nur nach einer exakten Übereinstimmung in einer Referenzübersetzung zu suchen, ist nicht optimal. Sogar Menschen würden bei einer solchen Art der Bewertung schlecht abschneiden, da jede bzw. jeder von uns einen Text etwas anders verfasst als andere (und sogar wir selbst texten nicht immer gleich, je nach Tages- oder Jahreszeit!). Doch zum Glück gibt es Alternativen.

Zwei der gebräuchlichsten Maße, die zur Evaluierung von generiertem Text herangezogen werden, sind das BLEU- und das ROUGE-Maß. Lassen Sie uns zunächst einen Blick darauf werfen, wie sie definiert sind.

BLEU

Die Idee hinter dem BLEU-Maß[4] ist einfach: Anstatt darauf zu achten, wie viele der Tokens des generierten Texts vollständig mit den Tokens des Referenztexts übereinstimmen, betrachten wir Wörter oder *n*-Gramme (engl. n-Gram). Das BLEU-Maß basiert auf der Precision. Das bedeutet, dass wir zum Vergleich der beiden Texte die Anzahl der Wörter in dem generierten Text ermitteln, die auch in dem Referenztext vorkommen, und diese Zahl durch die Länge des generierten Texts dividieren.

Allerdings gibt es ein Problem mit diesem originären »Vanilla«-Ansatz zur Berechnung der Precision. Nehmen wir an, im generierten Text wiederholt sich dasselbe Wort immer und immer wieder, und dieses Wort kommt auch in dem Referenztext vor. Wenn es so oft wiederholt wird, wie der Referenztext lang ist, dann erhalten wir für die Precision den Idealwert! Aus diesem Grund haben die Autoren des BLEU-Forschungsbeitrags eine kleine Anpassung (engl. Modification) vorgenommen: Ein Wort wird nur so oft gezählt, wie es im Referenztext vorkommt. Um diesen Punkt zu veranschaulichen, nehmen wir an, wir haben den Referenztext »the cat is on the mat« und den generierten Text »the the the the the the« vorliegen.

Ausgehend von diesem einfachen Beispiel können wir die Werte für die Precision wie folgt berechnen, wobei »vanilla« für den ursprünglichen Ansatz steht, und »mod« für die modifizierte Variante:

4 K. Papineni et al., »BLEU: A Method for Automatic Evaluation of Machine Translation,« *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics* (Juli 2002): 311–318, *http://dx.doi.org/10.3115/1073083.1073135*.

$$p_{vanilla} = \frac{6}{6}$$

$$p_{mod} = \frac{2}{6}$$

Durch die simple Anpassung erhalten wir einen wesentlich plausibleren Wert. Gehen wir noch einen Schritt weiter und betrachten wir nicht mehr nur einzelne Wörter, sondern auch *n*-Gramme. Nehmen wir an, wir haben einen generierten Satz, *snt*, den wir mit einem Referenzsatz, *snt'*, vergleichen möchten. Dann ermitteln wir zunächst alle *n*-Gramme vom Grad *n*, die gebildet werden können, und nehmen anschließend die Berechnung der Precision P_n vor:

$$p_n = \frac{\sum_{n\text{-}Gramm \in snt'} Anzahl_{clip}(n\text{-}Gramm)}{\sum_{n\text{-}Gramm \in snt} Anzahl(n\text{-}Gramm)}$$

Um zu vermeiden, dass sich wiederholende Generierungen belohnt werden, wird die Anzahl im Zähler bereinigt (»clip«). Das bedeutet, dass die Anzahl, wie häufig ein *n*-Gramm vorkommt, auf die Anzahl begrenzt wird, mit der es im Referenzsatz erscheint. Beachten Sie auch, dass die Definition eines Satzes in dieser Gleichung nicht sonderlich strikt ist. Wenn sich Ihr generierter Text über mehrere Sätze erstreckt, würden Sie ihn wie einen einzelnen Satz handhaben.

In der Regel enthält der Testdatensatz, mit dem wir die Evaluierung vornehmen möchten, mehr als nur ein Beispiel. Daher müssen wir die Gleichung geringfügig anpassen, indem wir die Summe über alle Beispiele im Korpus *C* bilden:

$$p_n = \frac{\sum_{snt' \in C} \sum_{n\text{-}Gramm \in snt'} Anzahl_{clip}(n\text{-}Gramm)}{\sum_{snt \in C} \sum_{n\text{-}Gramm \in snt} Anzahl(n\text{-}Gramm)}$$

Wir sind fast am Ziel. Da wir nicht den Recall als Maß einbeziehen, haben alle generierten Sequenzen, die zwar kurz, aber nicht präzise sind, einen Vorteil gegenüber Sätzen, die länger ausfallen. Dementsprechend begünstigt die Precision Generierungen, die kurz ausfallen. Als Ausgleich dafür haben die Autoren des BLEU-Forschungsbeitrags einen zusätzlichen Penalisierungsterm eingeführt, den *Brevity Penalty* (»Penalisierung aufgrund von Kurzheit«):

$$BR = \min\left(1, e^{1 - \ell_{ref}/\ell_{gen}}\right)$$

Dadurch, dass der Minimalwert beider Werte genommen wird, wird sichergestellt, dass der Brevity Penalty niemals den Wert 1 übersteigt. Der Exponentialterm wird äußerst klein, wenn die Länge des generierten Texts, l_{gen}, geringer ist als die des Referenztexts, l_{ref} (d. h., wenn der generierte Text kürzer ist als der Referenztext). An

dieser Stelle könnten Sie sich die Frage stellen, warum wir nicht einfach so etwas wie das F_1-Maß verwenden, um auch den Recall zu berücksichtigen. Die Antwort ist, dass es in vielen Datensätzen, die für Übersetzungsaufgaben herangezogen werden, nicht nur einen, sondern mehrere Referenzsätze gibt. Wenn wir also auch auf den Recall abzielen würden, gäbe es einen Anreiz, Übersetzungen zu generieren, in denen alle Wörter aus allen Referenztexten verwendet werden. Daher ist es besser, bei der Übersetzung auf eine hohe Precision abzuzielen und sicherzustellen, dass die Übersetzung und der Referenztext eine ähnliche Länge haben.

Wenn wir alles zusammenführen, erhalten wir die Gleichung für das BLEU-Maß:

$$\text{BLEU-}N = BR \times \left(\prod_{n=1}^{N} p_n \right)^{1/N}$$

Der letzte Term entspricht dem geometrischen Mittel der modifizierten Precision, wobei N der Anzahl entspricht, bis zu der die n-Gramme berücksichtigt werden. In der Praxis wird häufig der Wert für das BLEU-4-Maß angegeben. Vermutlich können Sie jedoch bereits erahnen, dass dieses Maß mit einigen Einschränkungen einhergeht. So werden beispielsweise Synonyme nicht berücksichtigt, und viele Schritte bei der Herleitung wirken wie Ad-hoc-Heuristiken, die relativ angreifbar sind. Eine wunderbare Darstellung der Schwächen des BLEU-Maßes finden Sie in Rachel Tatmans Blogbeitrag »Evaluating Text Output in NLP: BLEU at Your Own Risk« (*https://oreil.ly/nMXRh*).

Im Allgemeinen wird im Bereich der Textgenerierung immer noch nach besseren Maßen zur Evaluierung gesucht, wobei vor allem die Suche nach Möglichkeiten, wie die Schwächen von Maßen wie BLEU überwunden werden können, ein aktiver Bereich der Forschung ist. Eine weitere Schwäche des BLEU-Maßes ist, dass es voraussetzt, dass der Text bereits tokenisiert wurde. Wenn also nicht genau dieselbe Methode zur Tokenisierung des Texts verwendet wird, kann dies zu unterschiedlichen Ergebnissen führen. Das SacreBLEU-Maß löst dieses Problem, indem es auch den Tokenisierungsschritt einbezieht. Aus diesem Grund wird es beim Benchmarking bevorzugt.

So viel zur Theorie. Denn eigentlich liegt unser Hauptaugenmerk darauf, das Maß berechnen zu wollen. Müssen wir etwa die ganze dahinterstehende Logik selbst in Python implementieren? Keine Sorge, die 🤗 Datasets-Bibliothek stellt auch die entsprechenden Maße (engl. Metrics) bereit! Sie können die Maße auf die gleiche Weise laden wie einen Datensatz:

```
from datasets import load_metric

bleu_metric = load_metric("sacrebleu")
```

Das `bleu_metric`-Objekt ist eine Instanz der `Metric`-Klasse und funktioniert wie ein Aggregator: Sie können einzelne Instanzen mit `add()` oder ganze Batches mittels `add_batch()` hinzufügen. Sobald Sie alle Beispiele hinzugefügt haben, die Sie aus-

werten möchten, können Sie einfach die `compute()`-Methode aufrufen, woraufhin das Maß berechnet wird. Sie erhalten ein Dictionary, das mehrere Werte umfasst, wie z. B. die Precision für alle einbezogenen n-Gramme, den Brevity Penalty sowie den endgültigen Wert des BLEU-Maßes. Werfen wir noch einmal einen Blick auf das Beispiel von zuvor:

```
import pandas as pd
import numpy as np

bleu_metric.add(
    prediction="the the the the the the", reference=["the cat is on the mat"])
results = bleu_metric.compute(smooth_method="floor", smooth_value=0)
results["precisions"] = [np.round(p, 2) for p in results["precisions"]]
pd.DataFrame.from_dict(results, orient="index", columns=["Werte"])
```

	Werte
score	0.0
counts	[2, 0, 0, 0]
totals	[6, 5, 4, 3]
precisions	[33.33, 0.0, 0.0, 0.0]
bp	1.0
sys_len	6
ref_len	6

Das BLEU-Maß eignet sich auch dann, wenn mehrere Referenzübersetzungen vorliegen. Aus diesem Grund wird `reference` als Liste übergeben. Für den Fall, dass die Anzahl der *n*-Gramme null beträgt, ist es erforderlich, das Maß zu glätten bzw. anzupassen. Deshalb stehen für das BLEU-Maß Methoden bereit, mit denen die Berechnung der Precision modifiziert wird. Eine Methode besteht darin, im Zähler eine Konstante zu addieren. Auf diese Weise führt ein fehlendes *n*-Gramm nicht dazu, dass der Wert des Zählers automatisch null entspricht. Damit wir die Werte interpretieren können, sehen wir von dieser Anpassung ab, indem wir `smooth_value=0` setzen.

Wie Sie sehen, beträgt die Precision des 1-Gramms in der Tat 2/6, während die Precision der 2/3/4-Gramme alle 0 beträgt. (Weitere Informationen zu den einzelnen Maßen, wie z. B. counts und bp, finden Sie auf der Webseite des SacreBLEU-Repository (*https://oreil.ly/kiZPl*).) Das bedeutet, dass das geometrische Mittel null ergibt und somit auch der Wert des BLEU-Maßes. Nehmen wir ein weiteres Beispiel, bei dem die Vorhersage fast vollständig mit dem Referenztext übereinstimmt:

```
bleu_metric.add(
    prediction="the cat is on mat", reference=["the cat is on the mat"])
results = bleu_metric.compute(smooth_method="floor", smooth_value=0)
results["precisions"] = [np.round(p, 2) for p in results["precisions"]]
pd.DataFrame.from_dict(results, orient="index", columns=["Werte"])
```

	Werte
score	57.893007
counts	[5, 3, 2, 1]
totals	[5, 4, 3, 2]
precisions	[100.0, 75.0, 66.67, 50.0]
bp	0.818731
sys_len	5
ref_len	6

Das BLEU-Maß fällt in diesem Fall wesentlich besser aus. Die 1-Gramme stimmen in der Vorhersage alle überein, und nur an den Werten für die Precision sehen wir, dass etwas nicht stimmt. Für das 4-Gramm gibt es nur zwei mögliche Kandidaten, `["the", "cat", "is", "on"]` und `["cat", "is", "on", "mat"]`. Da das zweite 4-Gramm nicht übereinstimmt, beträgt die Precision 0,5.

Das BLEU-Maß wird häufig zur Bewertung von Texten verwendet, insbesondere im Rahmen der maschinellen Übersetzung, da präzise Übersetzungen in der Regel gegenüber Übersetzungen, die sämtliche Wörter enthalten, die möglich sind und passen, bevorzugt werden.

Es gibt andere Anwendungen, wie z. B. automatische Zusammenfassungen, bei denen die Sachlage allerdings eine andere ist. Hier wollen wir alle wichtigen Informationen im generierten Text berücksichtigt wissen und legen deshalb mehr Wert darauf, dass der Recall hoch ausfällt. Als Maß wird in der Regel das sogenannte ROUGE-Maß verwendet.

ROUGE

Das ROUGE-Maß wurde speziell für Anwendungen wie Zusammenfassungen entwickelt, bei denen der Recall von größerer Bedeutung ist als die Precision.[5] Der Ansatz ähnelt dem des BLEU-Maßes insofern, als dass wir verschiedene *n*-Gramme betrachten und vergleichen, wie oft sie im generierten Text und in den Referenztexten vorkommen. Der Unterschied ist, dass wir beim ROUGE-Maß prüfen, wie viele *n*-Gramme, die im Referenztext vorkommen, auch im generierten Text enthalten sind. Beim BLEU-Maß haben wir uns hingegen die Frage gestellt, wie viele *n*-Gramme, die im generierten Text vorhanden sind, auch im Referenztext vorkommen. Wir können also die Formel zur Berechnung der Precision wiederverwenden, allerdings mit der geringfügigen Anpassung, dass wir im Nenner die (unbereinigte – siehe »match« in Formel) Anzahl der *n*-Gramme des Referenztexts, die im generierten Text vorkommen, verwenden:

$$\text{ROUGE-}N = \frac{\sum_{\text{snt}' \in C} \sum_{n\text{-}Gramm \in snt'} Anzahl_{match}(n\text{-}Gramm)}{\sum_{\text{snt}' \in C} \sum_{n\text{-}Gramm \in snt'} Anzahl(n\text{-}Gramm)}$$

5 C-Y. Lin, »ROUGE: A Package for Automatic Evaluation of Summaries«, *Text Summarization Branches Out* (Juli 2004), *https://aclanthology.org/W04-1013.pdf*.

Genau so sah auch der ursprüngliche Vorschlag für das ROUGE-Maß aus. In der Folge haben Forscher herausgefunden, dass es erhebliche negative Auswirkungen haben kann, wenn die Precision überhaupt nicht einbezogen wird. Wenn wir zur ursprünglichen Formel für das BLEU-Maß zurückkommen (ohne die bereinigte Zählung), können wir auch die Precision ermitteln und dann die Precision- und Recall-Werte für das ROUGE-Maß als harmonisches Mittel kombinieren, um einen Wert für das F_1-Maß zu erhalten. In dieser Form wird das ROUGE-Maß heutzutage üblicherweise angegeben.

Für das ROUGE-Maß gibt es ein zusätzliches Maß, bei dem die längste gemeinsame Teilsequenz (engl. Longest Common Subsequence, LCS) ermittelt wird, genannt ROUGE-L. Der LCS-Wert kann für jedes beliebige Paar von Strings berechnet werden. Zum Beispiel wäre die längste gemeinsame Teilsequenz (LCS) für »abab« und »abc« »ab«, und ihre Länge betrüge 2. Wenn wir diesen Wert zwischen zwei Beispielen vergleichen möchten, müssen wir ihn in irgendeiner Weise normalisieren, da sonst ein längerer Text bevorteilt werden würde. Zu diesem Zweck hat der Erfinder des ROUGE-Maßes ein dem *F*-Maß ähnliches Schema entwickelt, bei dem der LCS-Wert auf Basis der Länge des Referenztexts und des generierten Texts normalisiert wird und die beiden normalisierten Werte dann miteinander kombiniert werden:

$$R_{LCS} = \frac{LCS(X,Y)}{m}$$

$$P_{LCS} = \frac{LCS(X,Y)}{n}$$

$$F_{LCS} = \frac{\left(1+\beta^2\right)R_{LCS}P_{LCS}}{R_{LCS}+\beta^2 P_{LCS}}, \text{wobei } \beta = P_{LCS}/R_{LCS}$$

Auf diese Weise wird das LCS-Maß (bzw. F_{LCS}-Maß) in geeigneter Weise normalisiert und lässt sich mit anderen Beispielen vergleichen. In der Implementierung, die in der 🤗 Datasets-Bibliothek bereitsteht, werden zwei Varianten des ROUGE-Maßes berechnet: Bei der einen wird das Maß pro Satz berechnet und ein Durchschnitt für die Zusammenfassungen gebildet (ROUGE-L), bei der anderen wird er direkt für die gesamte Zusammenfassung berechnet (ROUGE-Lsum).

Wir können das Maß wie folgt laden:

```
rouge_metric = load_metric("rouge")
```

Wir haben inzwischen eine Reihe von Zusammenfassungen mit GPT-2 und den anderen Modellen erstellt, und nun haben wir auch ein Maß zur Hand, mit dem wir die Zusammenfassungen systematisch vergleichen können. Ermitteln wir jetzt also verschiedene ROUGE-Maße für alle zuvor von den Modellen erstellten Zusammenfassungen:

```
reference = dataset["train"][1]["highlights"]
records = []
rouge_names = ["rouge1", "rouge2", "rougeL", "rougeLsum"]

for model_name in summaries:
    rouge_metric.add(prediction=summaries[model_name], reference=reference)
    score = rouge_metric.compute()
    rouge_dict = dict((rn, score[rn].mid.fmeasure) for rn in rouge_names)
    records.append(rouge_dict)
pd.DataFrame.from_records(records, index=summaries.keys())
```

	rouge1	rouge2	rougeL	rougeLsum
baseline	0.303571	0.090909	0.214286	0.232143
gpt2	0.187500	0.000000	0.125000	0.187500
t5	0.486486	0.222222	0.378378	0.486486
bart	0.582278	0.207792	0.455696	0.506329
pegasus	0.866667	0.655172	0.800000	0.833333

In der 🤗 Datasets-Bibliothek werden für das ROUGE-Maß auch die Konfidenzintervalle berechnet (standardmäßig das 5. und 95. Perzentil). Der Median wird unter dem Attribut `mid` gespeichert, und das Intervall kann über die Attribute `low` und `high` abgefragt werden.

Die Ergebnisse sind natürlich nicht sehr belastbar, da wir nur ein einziges Beispiel betrachtet haben. Aber dennoch können wir die Qualität der Zusammenfassungen im Hinblick auf das eine Beispiel vergleichen. Die Tabelle bestätigt unsere Beobachtung, dass das GPT-2-Modell von allen betrachteten Modellen am schlechtesten abschneidet. Das ist nicht überraschend, da es das einzige Modell aus der Gruppe ist, das nicht gezielt darauf trainiert wurde, Zusammenfassungen zu erstellen. Es fällt jedoch auf, dass das einfache Baseline-Modell, das als Zusammenfassung lediglich die ersten drei Sätze liefert, im Vergleich zu den Transformer-Modellen, die über eine Milliarde Parameter verfügen, gar nicht so schlecht abschneidet. Das PEGASUS-Modell schneidet insgesamt am besten ab (höhere Werte für das ROUGE-Maß sind besser), und das BART-Modell scheint – zumindest hinsichtlich des ROUGE-1-Maßes und der LCS-Maße – ein wenig besser als das T5-Modell zu sein. Allerdings sind diese Ergebnisse mit Vorsicht zu genießen, da wir die Modelle nur auf Basis eines einzigen Beispiels bewertet haben. Angesichts der Ergebnisse, die in dem Artikel zum PEGASUS-Modell vorgestellt wurden, würden wir auch im Hinblick auf den CNN/DailyMail-Datensatz erwarten, dass das PEGASUS- das T5-Modell übertrifft.

Mal sehen, ob wir dieses Ergebnis mit dem PEGASUS-Modell nachbilden können.

Evaluierung des PEGASUS-Modells auf dem CNN/DailyMail-Datensatz

Wir haben nun alle erforderlichen Komponenten zusammen, um das Modell auf geeignete Weise evaluieren zu können: Uns liegt der CNN/DailyMail-Datensatz vor, der auch einen Testdatensatz enthält, zudem haben wir mit ROUGE ein Maß, mit dem wir die Evaluierung vornehmen können, und wir haben ein Modell, mit dem wir Zusammenfassungen erstellen können. Jetzt müssen wir die einzelnen Komponenten nur noch zusammenführen. Bewerten wir zunächst die Leistung des Baseline-Modells, das jeweils die ersten drei Sätze als Zusammenfassung ausgibt:

```
def evaluate_summaries_baseline(dataset, metric,
                                column_text="article",
                                column_summary="highlights"):
    summaries = [three_sentence_summary(text) for text in dataset[column_text]]
    metric.add_batch(predictions=summaries,
                     references=dataset[column_summary])
    score = metric.compute()
    return score
```

Wir werden diese Funktion nun auf eine Teilmenge der Daten anwenden. Da der Testdatensatz des CNN/DailyMail-Datensatzes aus etwa 10.000 Beispielen besteht, würde es allerdings sehr viel Zeit in Anspruch nehmen, für alle Artikel die Zusammenfassungen zu erstellen. Rufen Sie sich aus Kapitel 5 in Erinnerung, dass jedes erzeugte Token einen Forward-Pass durch das Modell erfordert. Wenn also für jedes einzelne Beispiel lediglich 100 Tokens generiert werden müssten, würde dies bereits 1 Million solcher Forward-Passes erfordern, und wenn wir auf die Beam-Search zurückgreifen, müsste diese Zahl sogar noch mit der Anzahl der Beams multipliziert werden. Damit wir die Berechnungen relativ schnell durchführen können, verwenden wir stattdessen lediglich eine Teilmenge des Testdatensatzes und nehmen die Evaluierung auf Basis von nur 1.000 Beispielen vor. Dadurch sollten wir eine wesentlich robustere Schätzung für das Maß erhalten (im Vergleich zur Bewertung eines einzelnen Beispiels) und das PEGASUS-Modell in weniger als einer Stunde mit nur einer einzelnen GPU evaluieren können:

```
test_sampled = dataset["test"].shuffle(seed=42).select(range(1000))

score = evaluate_summaries_baseline(test_sampled, rouge_metric)
rouge_dict = dict((rn, score[rn].mid.fmeasure) for rn in rouge_names)
pd.DataFrame.from_dict(rouge_dict, orient="index", columns=["baseline"]).T
```

	rouge1	rouge2	rougeL	rougeLsum
baseline	0.396061	0.173995	0.245815	0.361158

Die Werte fallen größtenteils schlechter aus als im vorherigen Beispiel, aber immer noch besser als die vom GPT-2-Modell erzielten Werte! Erstellen wir nun eine Evaluierungsfunktion, mit der wir das PEGASUS-Modell evaluieren können:

```
from tqdm import tqdm
import torch

device = "cuda" if torch.cuda.is_available() else "cpu"

def chunks(list_of_elements, batch_size):
    """Liefert aufeinanderfolg. batch_size-große Chunks aus list_of_elements"""
    for i in range(0, len(list_of_elements), batch_size):
        yield list_of_elements[i : i + batch_size]

def evaluate_summaries_pegasus(dataset, metric, model, tokenizer,
                               batch_size=16, device=device,
                               column_text="article",
                               column_summary="highlights"):
    article_batches = list(chunks(dataset[column_text], batch_size))
    target_batches = list(chunks(dataset[column_summary], batch_size))

    for article_batch, target_batch in tqdm(
        zip(article_batches, target_batches), total=len(article_batches)):

        inputs = tokenizer(article_batch, max_length=1024,  truncation=True,
                        padding="max_length", return_tensors="pt")

        summaries = model.generate(input_ids=inputs["input_ids"].to(device),
                         attention_mask=inputs["attention_mask"].to(device),
                         length_penalty=0.8, num_beams=8, max_length=128)

        decoded_summaries = [tokenizer.decode(s, skip_special_tokens=True,
                              clean_up_tokenization_spaces=True)
               for s in summaries]
        decoded_summaries = [d.replace("<n>", " ") for d in decoded_summaries]
        metric.add_batch(predictions=decoded_summaries, references=target_batch)

    score = metric.compute()
    return score
```

Betrachten wir diesen Code zur Evaluierung einmal im Einzelnen. Zunächst teilen wir den Datensatz in kleinere Batches auf, die wir gleichzeitig verarbeiten können. Dann tokenisieren wir für jedes Batch die Eingabe- bzw. Input-Artikel und übergeben sie an die `generate()`-Funktion, mit der die Zusammenfassungen unter Verwendung der Beam-Search erstellt werden. Für die Generierung der Zusammenfassungen verwenden wir die gleichen Parameter, die in dem Forschungsbeitrag vorgeschlagen wurden. Der zusätzliche Parameter zur Penalisierung der Länge (`length_penalty`) stellt sicher, dass das Modell keine zu langen Sequenzen generiert. Anschließend decodieren wir die generierten Texte, ersetzen das Token `<n>` und fügen die decodierten Texte zusammen mit den Referenztexten dem `metric`-Objekt hinzu. Zum Schluss ermitteln wir die Werte für die ROUGE-Maße und geben sie zurück. Laden wir nun das Modell erneut mit der `AutoModelForSeq2SeqLM`-Klasse, die für seq2seq-Generierungsaufgaben geeignet ist, und evaluieren wir es mithilfe unserer Funktion:

```
from transformers import AutoModelForSeq2SeqLM, AutoTokenizer

model_ckpt = "google/pegasus-cnn_dailymail"
```

```
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
model = AutoModelForSeq2SeqLM.from_pretrained(model_ckpt).to(device)
score = evaluate_summaries_pegasus(test_sampled, rouge_metric,
                                   model, tokenizer, batch_size=8)
rouge_dict = dict((rn, score[rn].mid.fmeasure) for rn in rouge_names)
pd.DataFrame(rouge_dict, index=["pegasus"])
```

	rouge1	rouge2	rougeL	rougeLsum
pegasus	0.434381	0.210883	0.307195	0.373231

Diese Werte kommen den veröffentlichten Ergebnissen sehr nahe. Dabei ist zu beachten, dass der Verlust und die Treffergenauigkeit je Token (engl. per-Token Accuracy) zu einem gewissen Grad von den Werten der ROUGE-Maße entkoppelt sind. Der Verlust hängt nicht von der gewählten Decodierungsstrategie ab, wohingegen das ROUGE-Maß in erheblichem Maße davon abhängig ist.

Da das ROUGE- und das BLEU-Maß stärker mit dem menschlichen Urteilsvermögen korrelieren als der Verlust oder die Treffergenauigkeit, sollten wir uns auf diese beiden Maße konzentrieren und die von uns verfolgte Decodierungsstrategie bei der Erstellung von Textgenerierungsmodellen sorgfältig überdenken. Diese Maße sind jedoch bei Weitem nicht perfekt, daher sollte stets auch das menschliche Urteilsvermögen mit einbezogen werden.

Da wir nun über eine Evaluierungsfunktion verfügen, ist es an der Zeit, unser eigenes Modell zu trainieren, das Zusammenfassungen generieren kann.

Trainieren eines Modells zur Generierung von Zusammenfassungen

Wir haben nun ausführlich über die Textzusammenfassung und deren Evaluierung gesprochen. Darauf aufbauend können wir nun ein eigenes Modell für die Textzusammenfassung trainieren! Für unsere Anwendung verwenden wir den von Samsung erstellten SAMSum-Datensatz (*https://oreil.ly/n1ggq*), der aus einer Sammlung von Dialogen mit kurzen Zusammenfassungen besteht. In einem Unternehmen könnten diese Dialoge die Interaktionen zwischen einem Kunden und dem Kundensupport darstellen, sodass die Erstellung akkurater Zusammenfassungen dazu beitragen kann, den Kundenservice zu verbessern und gemeinsame Muster bei Kundenanfragen zu erkennen. Als Erstes können wir den Datensatz laden und uns ein Beispiel ausgeben lassen:

```
dataset_samsum = load_dataset("samsum")
split_lengths = [len(dataset_samsum[split])for split in dataset_samsum]

print(f"Länge der Teildatensätze: {split_lengths}")
print(f"Features: {dataset_samsum['train'].column_names}")
print("\nDialog:")
print(dataset_samsum["test"][0]["dialogue"])
print("\nZusammenfassung:")
print(dataset_samsum["test"][0]["summary"])
```

```
Länge der Teildatensätze: [14732, 819, 818]
Features: ['id', 'dialogue', 'summary']

Dialog:
Hannah: Hey, do you have Betty's number?
Amanda: Lemme check
Hannah: <file_gif>
Amanda: Sorry, can't find it.
Amanda: Ask Larry
Amanda: He called her last time we were at the park together
Hannah: I don't know him well
Hannah: <file_gif>
Amanda: Don't be shy, he's very nice
Hannah: If you say so..
Hannah: I'd rather you texted him
Amanda: Just text him 🤗
Hannah: Urgh.. Alright
Hannah: Bye
Amanda: Bye bye

Zusammenfassung:
Hannah needs Betty's number but Amanda doesn't have it. She needs to contact
Larry.
```

Die Dialoge sehen so aus, wie Sie es von einer Unterhaltung über SMS oder WhatsApp erwarten würden, insbesondere aufgrund der genutzten Emojis und der Platzhalter für GIFs. Das Feld `dialogue` enthält den vollständigen Text und das Feld `summary` den zusammengefassten Dialog. Kommt ein Modell, das auf der Grundlage des CNN/DailyMail-Datensatzes feingetunt wurde, damit zurecht? Lassen Sie es uns herausfinden!

Das PEGASUS-Modell auf dem SAMSum-Datensatz evaluieren

Beginnen wir damit, die gleiche Zusammenfassungspipeline wie zuvor mit dem PEGASUS-Modell auszuführen, um einmal zu testen, welche Art Ausgabe wir erwarten können. Hierzu können wir den Code wiederverwenden, den wir für die Generierung der Zusammenfassungen der CNN/DailyMail-Artikel verwendet haben:

```
pipe_out = pipe(dataset_samsum["test"][0]["dialogue"])
print("Zusammenfassung:")
print(pipe_out[0]["summary_text"].replace(" .<n>", ".\n"))
```

```
Zusammenfassung:
Amanda: Ask Larry Amanda: He called her last time we were at the park together.
Hannah: I'd rather you texted him.
Amanda: Just text him .
```

Wie wir feststellen, versucht das Modell zur Erstellung der Zusammenfassungen, vor allem die wichtigsten Sätze aus dem Dialog zu extrahieren. Das hat wahrscheinlich beim CNN/DailyMail-Datensatz relativ gut funktioniert, aber die Zusammenfassungen im SAMSum-Datensatz sind abstrakter. Nehmen wir am besten die vollständige Evaluierung auf Basis des Testdatensatzes vor, um zu sehen, ob sich unser Verdacht bestätigt. Dabei greifen wir wieder auf das ROUGE-Maß zurück:

```
score = evaluate_summaries_pegasus(dataset_samsum["test"], rouge_metric, model,
                                   tokenizer, column_text="dialogue",
                                   column_summary="summary", batch_size=8)

rouge_dict = dict((rn, score[rn].mid.fmeasure) for rn in rouge_names)
pd.DataFrame(rouge_dict, index=["pegasus"])
```

	rouge1	rouge2	rougeL	rougeLsum
pegasus	0.296168	0.087803	0.229604	0.229514

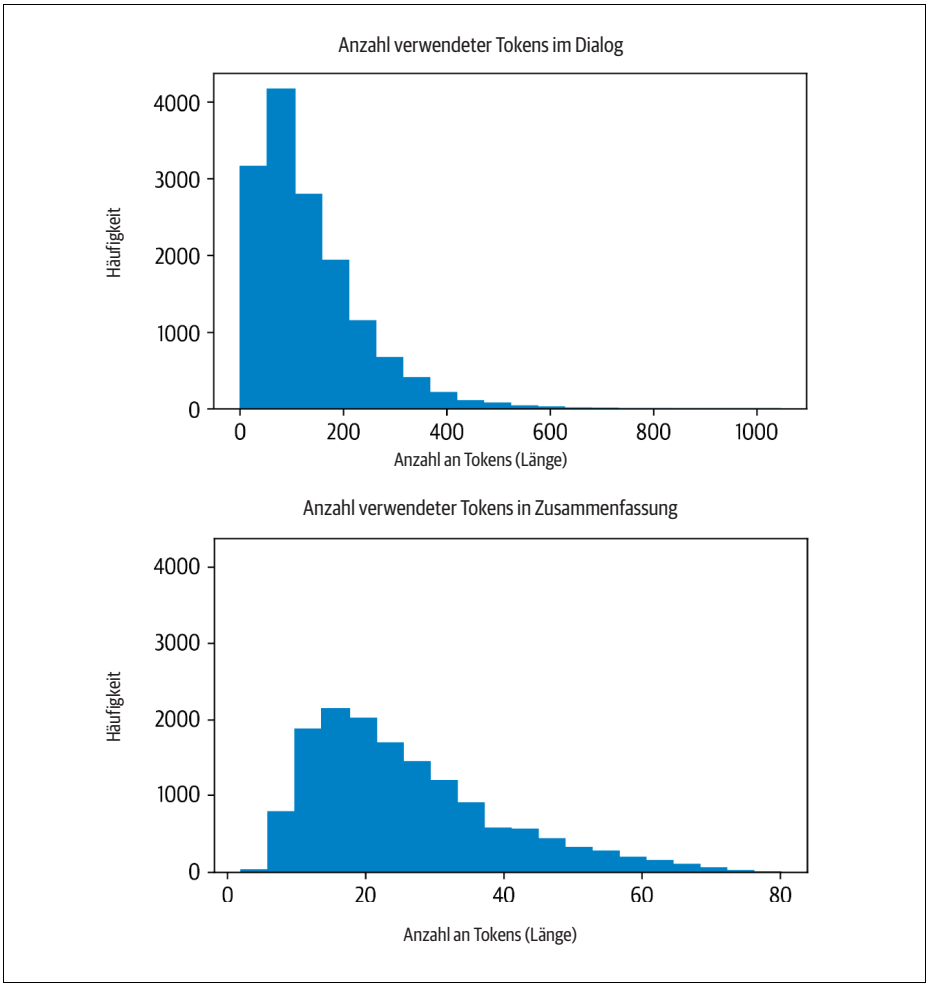
Nun, die Ergebnisse sind nicht großartig. Allerdings ist das auch nicht überraschend, da sich die Verteilung der SAMSum-Daten von der der CNN/DailyMail-Daten relativ stark unterscheidet. Nichtsdestotrotz hat es zwei Vorteile, die Evaluierungspipeline bereits vor dem Training einzurichten: Wir können den Erfolg des Trainings direkt mit dem Maß beurteilen und wir haben eine gute Baseline zur Hand. Als Nächstes werden wir unser Modell mit dem SAMSum-Datensatz feintunen, was zu einer sofortigen Verbesserung des ROUGE-Maßes führen sollte. Sollte dies nicht der Fall sein, wissen wir, dass etwas mit unserer Trainingsschleife nicht stimmt.

Das PEGASUS-Modell feintunen

Bevor wir die Daten für das Training verarbeiten, sollten wir einen kurzen Blick auf die Verteilung der Länge der Eingaben und Ausgaben werfen (d.h., wie viele Tokens jeweils verwendet werden):

```
d_len = [len(tokenizer.encode(s)) for s in dataset_samsum["train"]["dialogue"]]
s_len = [len(tokenizer.encode(s)) for s in dataset_samsum["train"]["summary"]]

fig, axes = plt.subplots(1, 2, figsize=(10, 3.5), sharey=True)
axes[0].hist(d_len, bins=20, color="C0", edgecolor="C0")
axes[0].set_title("Anzahl verwendeter Tokens im Dialog")
axes[0].set_xlabel("Anzahl an Tokens (Länge)")
axes[0].set_ylabel("Häufigkeit")
axes[1].hist(s_len, bins=20, color="C0", edgecolor="C0")
axes[1].set_title("Anzahl verwendeter Tokens in Zusammenfassung")
axes[1].set_xlabel("Anzahl an Tokens (Länge)")
plt.tight_layout()
plt.show()
```

Wie zu sehen ist, sind die meisten Dialoge mit rund 100 bis 200 Tokens pro Dialog wesentlich kürzer als die CNN/DailyMail-Artikel. Ebenso sind auch die Zusammenfassungen mit etwa 20 bis 40 Tokens bedeutend kürzer (was der durchschnittlichen Länge eines Tweets entspricht).



Behalten wir diese Erkenntnisse im Hinterkopf, bis wir den Data-Collator für den `Trainer` erstellen. Zunächst müssen wir den Datensatz noch tokenisieren. Für den Moment setzen wir jeweils die maximale Länge auf 1.024 Tokens für Dialoge und auf 128 Tokens für Zusammenfassungen:

```
def convert_examples_to_features(example_batch):
    input_encodings = tokenizer(example_batch["dialogue"], max_length=1024,
                                truncation=True)

    with tokenizer.as_target_tokenizer():
        target_encodings = tokenizer(example_batch["summary"], max_length=128,
                                     truncation=True)

    return {"input_ids": input_encodings["input_ids"],
            "attention_mask": input_encodings["attention_mask"],
            "labels": target_encodings["input_ids"]}
```

```
dataset_samsum_pt = dataset_samsum.map(convert_examples_to_features,
                                       batched=True)
columns = ["input_ids", "labels", "attention_mask"]
dataset_samsum_pt.set_format(type="torch", columns=columns)
```

Eine Neuerung im Rahmen des Tokenisierungsschritts ist die Verwendung eines Kontexts, `tokenizer.as_target_tokenizer()`. Einige Modelle erfordern spezielle Tokens in den Eingaben des Decoders, daher ist es wichtig, zwischen der Tokenisierung der Eingaben des Encoders und der des Decoders zu unterscheiden. Innerhalb der Anweisung `with` (als *Kontextmanager* bezeichnet) weiß der Tokenizer, dass er die Tokenisierung für den Decoder vornimmt, und kann die Sequenzen entsprechend verarbeiten.

Nun müssen wir den Data-Collator erstellen. Diese Funktion wird im `Trainer` aufgerufen, kurz bevor das Batch durch das Modell geleitet wird. In den meisten Fällen können wir den vorgegebenen Default-Collator verwenden, der alle Tensoren aus dem Batch sammelt und sie einfach aneinanderreiht bzw. zusammenführt. Für die Zusammenfassungsaufgabe müssen wir nicht nur die Eingaben aneinanderreihen, sondern auch die Targets auf der Decoder-Seite vorbereiten. Das PEGASUS-Modell ist ein Encoder-Decoder-basierter Transformer und hat somit die typische seq2seq-Architektur. In einem seq2seq-Aufbau ist es üblich, im Decoder ein »Teacher Forcing« anzuwenden. Bei dieser Strategie erhält der Decoder Eingabe- bzw. Input-Tokens (wie bei rein Decoder-basierten Modellen wie GPT-2), die zusätzlich zur Ausgabe des Encoders aus den um einen Schritt bzw. eine Stelle verschobenen Labels bestehen. Wenn er also die Vorhersage für das nächste Token trifft, erhält der Decoder das um einen Schritt verschobene Ground-Truth-Label als Eingabe, wie in der folgenden Tabelle dargestellt:

step	decoder_input	label
1	[PAD]	Transformers
2	[PAD, Transformers]	are
3	[PAD, Transformers, are]	awesome
4	[PAD, Transformers, are, awesome]	for
5	[PAD, Transformers, are, awesome, for]	text
6	[PAD, Transformers, are, awesome, for, text]	summarization

Wir verschieben sie um eine Stelle, sodass der Decoder nur die vorherigen Ground-Truth-Labels sieht und nicht die aktuellen oder zukünftigen. Die Verschiebung allein reicht aus, da der Decoder über eine Masked-Self-Attention verfügt, die alle Eingaben in der Gegenwart und in der Zukunft maskiert.

Daher richten wir im Rahmen der Vorbereitung unseres Batches die Eingaben des Decoders so ein, dass wir die Labels um eine Stelle nach rechts verschieben. Danach stellen wir sicher, dass die Padding-Tokens, die zum Auffüllen verwendet werden, von der Verlustfunktion nicht berücksichtigt werden, indem wir sie auf

den Wert -100 setzen. Die Schritte müssen wir jedoch glücklicherweise nicht manuell vornehmen, da wir auf die `DataCollatorForSeq2Seq`-Klasse zurückgreifen können, die das für uns übernimmt:

```
from transformers import DataCollatorForSeq2Seq

seq2seq_data_collator = DataCollatorForSeq2Seq(tokenizer, model=model)
```

Anschließend richten wir wie üblich das `TrainingArguments`-Objekt für das Training ein:

```
from transformers import TrainingArguments, Trainer

training_args = TrainingArguments(
    output_dir='pegasus-samsum', num_train_epochs=1, warmup_steps=500,
    per_device_train_batch_size=1, per_device_eval_batch_size=1,
    weight_decay=0.01, logging_steps=10, push_to_hub=True,
    evaluation_strategy='steps', eval_steps=500, save_steps=1e6,
    gradient_accumulation_steps=16)
```

Eine Sache, die sich von den vorherigen Vorgaben unterscheidet, ist das zusätzliche Argument `gradient_accumulation_steps`. Da das Modell recht groß ist, ist es notwendig, die Batchgröße auf 1 zu setzen. Eine zu kleine Batchgröße kann jedoch die Konvergenz im Rahmen der Optimierung beeinträchtigen. Um dieses Problem zu lösen, können wir eine raffinierte Technik namens *Gradient-Accumulation* verwenden. Wie der Name schon sagt, berechnen wir nicht die Gradienten des gesamten Batches auf einmal, sondern wir bilden kleinere Batches und aggregieren bzw. summieren die Gradienten. Wenn wir genügend Gradienten aggregiert haben, führen wir den Optimierungsschritt aus. Das geht natürlich etwas langsamer vonstatten, als wenn sie in einem Durchgang berechnet werden, aber es spart uns eine Menge an GPU-Speicher.

Stellen wir nun sicher, dass wir beim Hugging Face Hub angemeldet sind, damit wir das Modell nach dem Training auf den Hub übertragen können:

```
from huggingface_hub import notebook_login

notebook_login()
```

Jetzt haben wir alles zusammen, was wir benötigen, um das `Trainer`-Objekt mit dem Modell, dem Tokenizer, den Trainingsargumenten und dem Data-Collator sowie dem Trainings- und dem Validierungsdatensatz zu initialisieren:

```
trainer = Trainer(model=model, args=training_args,
                  tokenizer=tokenizer, data_collator=seq2seq_data_collator,
                  train_dataset=dataset_samsum_pt["train"],
                  eval_dataset=dataset_samsum_pt["validation"])
```

Wir sind nun so weit, dass wir das Modell trainieren können. Nach dem Training können wir die Evaluierungsfunktion direkt auf den Testdatensatz anwenden, um zu ermitteln, wie gut das Modell abschneidet:

```
trainer.train()
trainer.model.eval()
```

```
score = evaluate_summaries_pegasus(
    dataset_samsum["test"], rouge_metric, trainer.model, tokenizer,
    batch_size=2, column_text="dialogue", column_summary="summary")

rouge_dict = dict((rn, score[rn].mid.fmeasure) for rn in rouge_names)
pd.DataFrame(rouge_dict, index=[f"pegasus"])
```

	rouge1	rouge2	rougeL	rougeLsum
pegasus	0.427614	0.200571	0.340648	0.340738

Wir können feststellen, dass sich die Werte des ROUGE-Maßes gegenüber dem Modell, das nicht feingetunt wurde, erheblich verbessert haben. Obwohl das vorherige Modell auch darauf trainiert war, Zusammenfassungen zu erstellen, war es also nicht besonders gut für die neue Domäne geeignet. Übertragen bzw. pushen wir unser Modell nun auf den Hub:

```
trainer.push_to_hub("Training complete!")
```

Im nächsten Abschnitt werden wir das Modell dazu verwenden, ein paar Zusammenfassungen für uns zu erstellen.

Sie können die generierten Texte auch direkt im Rahmen der Trainingsschleife evaluieren: Verwenden Sie hierzu einfach eine Erweiterung der `TrainingArguments`-Klasse namens `Seq2SeqTrainingArguments` und geben Sie `predict_with_generate=True` an. Übergeben Sie es an den speziellen `Trainer` namens `Seq2SeqTrainer`, der dann die `generate()`-Funktion anstelle des Forward-Pass des Modells verwendet, um Vorhersagen zur Evaluierung zu erstellen. Probieren Sie es aus!

Zusammenfassungen von Dialogen erstellen

Ein Blick auf die Verluste und ROUGE-Maße zeigt, dass das Modell eine deutliche Verbesserung gegenüber dem Modell darstellt, das nur mit dem CNN/DailyMail-Datensatz trainiert wurde. Schauen wir uns an, wie eine Zusammenfassung aussieht, die auf der Grundlage eines Beispiels aus dem Testdatensatz erstellt wurde:

```
gen_kwargs = {"length_penalty": 0.8, "num_beams":8, "max_length": 128}
sample_text = dataset_samsum["test"][0]["dialogue"]
reference = dataset_samsum["test"][0]["summary"]
pipe = pipeline("summarization", model="transformersbook/pegasus-samsum")

print("Dialog:")
print(sample_text)
print("\nReferenzzusammenfassung:")
print(reference)
print("\nZusammenfassung des Modells:")
print(pipe(sample_text, **gen_kwargs)[0]["summary_text"])

Dialog:
Hannah: Hey, do you have Betty's number?
Amanda: Lemme check
Hannah: <file_gif>
```

```
Amanda: Sorry, can't find it.
Amanda: Ask Larry
Amanda: He called her last time we were at the park together
Hannah: I don't know him well
Hannah: <file_gif>
Amanda: Don't be shy, he's very nice
Hannah: If you say so..
Hannah: I'd rather you texted him
Amanda: Just text him 🤗
Hannah: Urgh.. Alright
Hannah: Bye
Amanda: Bye bye

Referenzzusammenfassung:
Hannah needs Betty's number but Amanda doesn't have it. She needs to contact
Larry.

Zusammenfassung des Modells:
Amanda can't find Betty's number. Larry called Betty last time they were at the
park together. Hannah wants Amanda to text Larry instead of calling Betty.
```

Die Zusammenfassung des Modells ähnelt nun der Referenzzusammenfassung bedeutend stärker. Es scheint, dass das Modell gelernt hat, den Dialog in eine Zusammenfassung zu überführen, ohne dabei einfach nur Passagen zu extrahieren. Kommen wir nun zum ultimativen Test: Wie gut funktioniert das Modell bei einer selbst verfassten Eingabe?

```
custom_dialogue = """\
Thom: Hi guys, have you heard of transformers?
Lewis: Yes, I used them recently!
Leandro: Indeed, there is a great library by Hugging Face.
Thom: I know, I helped build it ;)
Lewis: Cool, maybe we should write a book about it. What do you think?
Leandro: Great idea, how hard can it be?!
Thom: I am in!
Lewis: Awesome, let's do it together!
"""
print(pipe(custom_dialogue, **gen_kwargs)[0]["summary_text"])
```

```
Thom, Lewis and Leandro are going to write a book about transformers. Thom
helped build a library by Hugging Face. They are going to do it together.
```

Die generierte Zusammenfassung für den von uns selbst verfassten Dialog ist recht schlüssig. Sie fasst gut zusammen, dass alle Gesprächsteilnehmer das Buch gemeinsam schreiben wollen, und extrahiert nicht einfach einzelne Sätze. Sie kombiniert zum Beispiel die dritte und vierte Zeile zu einem logischen Zusammenhang.

Zusammenfassung

Die Textzusammenfassung birgt im Vergleich zu anderen Aufgaben, die als Klassifizierungsaufgaben betrachtet werden können, wie z.B. die Sentimentanalyse, die Named Entity Recognition oder dem Question Answering, einige besondere Herausforderungen. Konventionelle Maße wie die Treffergenauigkeit (engl. Accuracy) spiegeln nicht die Qualität des generierten Texts wider. Wie wir erfahren haben, eig-

nen sich das BLEU- und das ROUGE-Maß besser, um generierte Texte evaluieren zu können. Dennoch stellt das menschliche Urteilsvermögen immer noch das beste Maß dar!

Eine Frage, die sich bei der Arbeit mit Modellen für die automatische Textzusammenfassung häufig stellt, ist, wie wir Dokumente zusammenfassen können, deren Texte länger als die Kontextlänge des Modells sind. Leider gibt es keine einheitliche Strategie, mit der dieses Problem gelöst werden kann, und bis heute ist dieses Thema immer noch Gegenstand der laufenden Forschung. Die jüngste Studie von OpenAI hat zum Beispiel gezeigt, wie sich Zusammenfassungen skalieren lassen, indem sie bei langen Dokumenten rekursiv vorgenommen werden und menschliches Feedback in die Schleife einfließt.[6]

Im nächsten Kapitel befassen wir uns mit der automatischen Beantwortung von Fragen (Question Answering, QA), d.h. mit der Aufgabe, auf der Grundlage einer Textpassage eine Antwort auf eine Frage zu geben. Im Gegensatz zur Zusammenfassung gibt es bei dieser Aufgabe bereits gute Strategien, um mit langen oder vielen Dokumenten umzugehen, und wir werden Ihnen zeigen, wie Sie das Question Answering auf Tausende von Dokumenten skalieren können.

6 J. Wu et al., »Recursively Summarizing Books with Human Feedback« (*https://arxiv.org/abs/2109.10862*), (2021).

KAPITEL 7

Question Answering

Ganz gleich, ob Sie Forscher, Analyst oder Data Scientist sind – die Wahrscheinlichkeit ist groß, dass Sie sich schon einmal durch ein Meer von Dokumenten wühlen mussten, um die gesuchten Informationen zu finden. Erschwerend kommt hinzu, dass Sie von Google und Bing ständig daran erinnert werden, dass offensichtlich bessere Wege der Recherche existieren! Wenn wir zum Beispiel auf Google nach »When did Marie Curie win her first Nobel Prize?« (»Wann hat Marie Curie ihren ersten Nobelpreis erhalten?«) suchen, erhalten wir sofort die korrekte Antwort »1903« (siehe Abbildung 7-1).

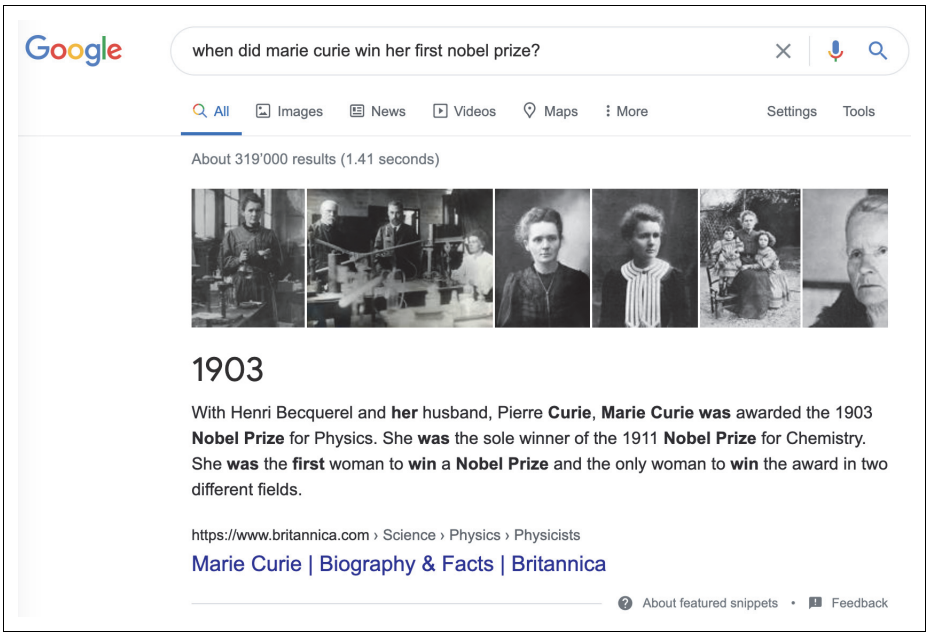


Abbildung 7-1: Eine Google-Suchanfrage und der entsprechende Antwortauszug

In diesem Beispiel hat Google zunächst rund 319.000 Dokumente abgerufen, die für die Anfrage relevant waren, und dann einen zusätzlichen Verarbeitungsschritt durchgeführt, um den Antwortauszug zu extrahieren, der die entsprechende Text-

passage und die Adresse der Webseite enthält. Es ist unschwer zu erkennen, warum diese Antwortauszüge nützlich sind. Wenn wir zum Beispiel nach einer kniffligen Frage suchen wie »Welche Gitarrenstimmung ist die beste?« liefert Google keine Antwort. Stattdessen müssen wir auf eine der von der Suchmaschine zurückgegebenen Webseiten klicken, um sie selbst zu finden.[1]

Der allgemeine Ansatz, der hinter dieser Technologie steht, wird *Fragenbeantwortung* bzw. *Question Answering* (QA) genannt. Es gibt viele Varianten des QA, aber die gebräuchlichste ist das *extraktive QA*, bei dem es um Fragen geht, deren Antwort als eine Textpassage in einem Dokument identifiziert werden kann, wobei es sich bei dem Dokument um eine Webseite, einen juristischen Vertrag oder einen Nachrichtenartikel handeln kann. Der zweistufige Prozess, bei dem zunächst relevante Dokumente abgerufen und dann Antworten daraus extrahiert werden, ist auch die Grundlage für viele moderne QA-Systeme, darunter semantische Suchmaschinen, intelligente Assistenten und automatische Informationsextraktionssysteme. In diesem Kapitel werden wir diesen Prozess dazu verwenden, ein weit verbreitetes Problem zu lösen, mit dem E-Commerce-Websites konfrontiert sind: Unterstützung der Verbraucher bei der Beantwortung bestimmter Fragen, damit sie ein Produkt besser einschätzen können. Dabei werden wir feststellen, dass Kundenrezensionen eine reichhaltige und anspruchsvolle Informationsquelle für die Qualitätssicherung darstellen. Zudem lernen wir, wie Transformer als leistungsstarke Modelle für das *Leseverstehen* (engl. Reading Comprehension) fungieren, die die Bedeutung eines Texts extrahieren können. Machen wir uns zunächst ein genaueres Bild von dem Anwendungsfall.

Dieses Kapitel behandelt vor allem das extraktive QA, auch wenn andere Formen des QA für Ihren Anwendungsfall besser geeignet sein können. Bei dem *Community-QA* werden beispielsweise Frage-Antwort-Paare gesammelt, die von Benutzern in Foren wie Stack Overflow (*https://stackoverflow.com*) erzeugt werden, und dann mithilfe der semantischen Ähnlichkeitssuche (engl. Semantic Similarity Search) die am besten passende Antwort auf eine neue Frage gefunden. Es gibt auch das *Long-Form-QA*, das darauf abzielt, komplexe Antworten, die die Länge eines ganzen Absatzes haben, auf offene Fragen wie »Warum ist der Himmel blau?« zu generieren. Bemerkenswerterweise ist es auch möglich, QA anhand von Tabellen durchzuführen. Transformer-Modelle wie TAPAS (*https://oreil.ly/vVPWO*) können sogar Aggregationen vornehmen, um die endgültige Antwort zu erzeugen!

Aufbau eines rezensionsbasierten QA-Systems

Wenn Sie schon einmal ein Produkt online gekauft haben, dann haben Sie sich wahrscheinlich auf Kundenrezensionen verlassen, die Ihnen bei Ihrer Entscheidung geholfen haben. Diese Rezensionen können oft bei der Beantwortung spezifi-

1 Obwohl sich in diesem speziellen Fall alle einig sind, dass die Dropped-C-Stimmung die beste Gitarrenstimmung darstellt.

scher Fragen helfen, die anhand der Produktbeschreibung allein schwer zu beantworten sind, wie z.B. »Wird diese Gitarre mit einem Gurt geliefert?« oder »Kann ich diese Kamera auch nachts verwenden?«. Für beliebte Produkte können jedoch Hunderte bis Tausende von Rezensionen vorliegen, sodass es sehr mühsam sein kann, eine relevante Rezension zu finden. Eine Alternative ist, Ihre Frage auf den Community-QA-Plattformen zu stellen, die von Webseiten wie Amazon angeboten werden – allerdings dauert es in der Regel Tage, bis Sie eine Antwort erhalten (wenn Sie überhaupt eine erhalten). Wäre es nicht schön, wenn wir eine sofortige Antwort erhalten könnten, wie im Google-Beispiel in Abbildung 7-1? Mal sehen, ob wir das Ganze mithilfe von Transformern hinbekommen.

Der Datensatz

Um unser QA-System aufzubauen, verwenden wir den SubjQA-Datensatz.[2] Dieser besteht aus mehr als 10.000 englischsprachigen Kundenrezensionen zu Produkten und Dienstleistungen aus sechs verschiedenen Bereichen: TripAdvisor, Restaurants, Filme (engl. Movies), Bücher (engl. Books), elektronische Geräte (engl. Electronics) und Lebensmittel (engl. Grocery). Wie in Abbildung 7-2 dargestellt, ist jede Rezension mit einer Frage verbunden, die mit einem oder mehreren Sätzen aus der Rezension beantwortet werden kann.[3]

Product: Nokia Lumia 521 RM-917 8GB

Query: Why is the camera of poor quality?

Review: Item like the picture, fast deliver 3 days well packed, good quality for the price. The camera is decent (as phone cameras go), There is no flash though ...

Abbildung 7-2: Eine Frage zu einem Produkt und die entsprechende Rezension (der entsprechende Abschnitt, in dem die Antwort gegeben wird (engl. Answer Span), ist unterstrichen)

2 J. Bjerva et al., »SubjQA: A Dataset for Subjectivity and Review Comprehension« (*https://arxiv.org/abs/2004.14283*), (2020).

3 Wie wir gleich sehen werden, gibt es auch *unbeantwortbare* Fragen, die dazu dienen, robustere Modelle zu erstellen.

Der interessante Aspekt dieses Datensatzes ist, dass die meisten Fragen und Antworten *subjektiv*, also von den persönlichen Erfahrungen der Benutzer abhängig sind. Das Beispiel in Abbildung 7-2 zeigt, warum diese Charakteristik die Aufgabe potenziell schwieriger macht als die Beantwortung von sachlichen Fragen wie »Wie heißt die Währung des Vereinigten Königreichs?«. Erstens geht es bei der Nachfrage um »schlechte Qualität« (»Why is the camera of poor quality?«), was subjektiv ist und entsprechend von der individuellen Vorstellung des Benutzers abhängt, was unter Qualität zu verstehen ist. Zweitens erscheinen wichtige Teile der Anfrage überhaupt nicht in der Rezension, was bedeutet, dass sie nicht mit einfachen Mitteln wie der Suche nach Schlüsselwörtern oder der Paraphrasierung der eingegebenen Frage beantwortet werden kann. Diese Eigenschaften machen SubjQA zu einem realistischen Datensatz, den wir als Benchmark bzw. Vergleichsgrundlage für unsere rezensionsbasierten QA-Modelle nutzen können, da nutzergenerierte Inhalte wie die in Abbildung 7-2 gezeigten dem ähneln, was wir auch in der realen Welt antreffen würden.

QA-Systeme werden in der Regel entsprechend der *Domäne* der Daten kategorisiert, auf die sie bei der Beantwortung einer Anfrage Zugriff haben. Bei dem *Closed-Domain*-QA geht es um Fragen zu einem eingegrenzten Thema (z.B. eine einzelne Produktkategorie), während das *Open-Domain*-QA Fragen zu fast allem (z.B. den gesamten Produktkatalog von Amazon) umfasst. Im Allgemeinen müssen beim Closed-Domain-QA weniger Dokumente durchsucht werden als im Fall des Open-Domain-QA.

Legen wir los und laden wir den Datensatz vom Hugging Face Hub (*https://oreil.ly/iO0s5*) herunter. Wie in Kapitel 4 können wir die Funktion `get_dataset_config_names()` verwenden, um herauszufinden, welche Teildatensätze (engl. Subsets) verfügbar sind:

```
from datasets import get_dataset_config_names

domains = get_dataset_config_names("subjqa")
domains
```

```
['books', 'electronics', 'grocery', 'movies', 'restaurants', 'tripadvisor']
```

In unserem Anwendungsfall konzentrieren wir uns darauf, ein QA-System für Elektronikprodukte zu erstellen. Um den entsprechenden Teildatensatz herunterzuladen, müssen wir lediglich `electronics` für das Argument `name` der `load_dataset()`-Funktion angeben:

```
from datasets import load_dataset

subjqa = load_dataset("subjqa", name="electronics")
```

Wie auch bei anderen QA-Datensätzen auf dem Hub, werden beim SubjQA-Datensatz die Antworten auf die einzelnen Fragen in Form eines verschachtelten Dictionarys gespeichert. Wenn wir zum Beispiel eine der Zeilen in der Spalte `answers` inspizieren:

```
print(subjqa["train"]["answers"][1])
```

```
{'text': ['Bass is weak as expected', 'Bass is weak as expected, even with EQ
adjusted up'], 'answer_start': [1302, 1302], 'answer_subj_level': [1, 1],
'ans_subj_score': [0.5083333253860474, 0.5083333253860474], 'is_ans_subjective':
[True, True]}
```

sehen wir, dass die Antworten in dem Feld `text` gespeichert sind, während die Indizes der Anfangszeichen in dem Feld `answer_start` enthalten sind. Um den Datensatz einfacher erkunden zu können, verflachen wir diese verschachtelten Spalten mithilfe der `flatten()`-Methode und überführen den Trainings-, Validierungs- und Testdatensatz (engl. Split) jeweils in ein Pandas-`DataFrame`:

```
import pandas as pd

dfs = {split: dset.to_pandas() for split, dset in subjqa.flatten().items()}

for split, df in dfs.items():
    print(f"Anzahl an Fragen in {split}: {df['id'].nunique()}")
```

```
Anzahl an Fragen in train: 1295
Anzahl an Fragen in test: 358
Anzahl an Fragen in validation: 255
```

Beachten Sie, dass der Datensatz mit insgesamt nur 1.908 Beispielen relativ klein ist. Dies spiegelt ein reales Szenario wider, da es arbeitsintensiv und teuer ist, Experten zum Labeln extraktiver QA-Datensätze zu gewinnen. Beispielsweise wird der Wert des CUAD-Datensatzes, der für das extraktive QA auf Basis handelsrechtlicher Verträge konzipiert ist, auf 2 Millionen Dollar geschätzt, um dem juristischen Fachwissen Rechnung zu tragen, das für die Annotation der 13.000 Beispiele erforderlich war.[4]

Es gibt eine ganze Reihe von Spalten im SubjQA-Datensatz, wobei die für den Aufbau unseres QA-Systems relevanten in Tabelle 7-1 aufgeführt sind.

Tabelle 7-1: Spaltennamen und ihre Beschreibungen im SubjQA-Datensatz

Spaltenname	Beschreibung
`title`	Die Amazon Standard Identification Number (ASIN), die jedes Produkt kennzeichnet
`question`	Die Frage
`answers.text`	Der vom Annotator gelabelte Textabschnitt in der Rezension
`answers.answer_start`	Der Index des Anfangszeichens des Bereichs, in dem die Antwort enthalten ist (Answer Span)
`context`	Die Kundenrezension

Konzentrieren wir uns auf diese Spalten und werfen einen Blick auf einige Trainingsbeispiele. Mithilfe der `sample()`-Methode können wir eine Zufallsstichprobe (engl. Random Sample) mit zwei Beispielen auswählen:

4 D. Hendrycks et al., »CUAD: An Expert-Annotated NLP Dataset for Legal Contract Review« (*https://arxiv.org/abs/2103.06268*), (2021).

```
qa_cols = ["title", "question", "answers.text",
           "answers.answer_start", "context"]
sample_df = dfs["train"][qa_cols].sample(2, random_state=7)
sample_df
```

title	question	answers. text	answers. answer_start	context
B005DKZTMG	Does the keyboard lightweight?	[this keyboard is compact]	[215]	I really like this keyboard. I give it 4 stars because it doesn't have a CAPS LOCK key so I never know if my caps are on. But for the price, it really suffices as a wireless keyboard. I have very large hands and this keyboard is compact, but I have no complaints.
B00AAIPT76	How is the battery?	[]	[]	I bought this after the first spare gopro battery I bought wouldn't hold a charge. I have very realistic expectations of this sort of product, I am skeptical of amazing stories of charge time and battery life but I do expect the batteries to hold a charge for a couple of weeks at least and for the charger to work like a charger. In this I was not disappointed. I am a river rafter and found that the gopro burns through power in a hurry so this purchase solved that issue. the batteries held a charge, on shorter trips the extra two batteries were enough and on longer trips I could use my friends JOOS Orange to recharge them.I just bought a newtrent xtreme powerpak and expect to be able to charge these with that so I will not run out of power again.

Die Beispiele geben uns einige Anhaltspunkte. Erstens sind die Fragen grammatikalisch nicht korrekt, was in den FAQ-Abschnitten von E-Commerce-Webseiten durchaus üblich ist. Zweitens kennzeichnet ein leerer `answers.text`-Eintrag »unbeantwortbare« Fragen, deren Antwort nicht in der Rezension zu finden ist. Wir können schließlich den Startindex und die Länge des für die Antwort relevanten Textabschnitts (Answer Span) verwenden, um die Textpassage, die der Antwort entspricht, mittels Slicing aus der Rezension herauszufiltern:

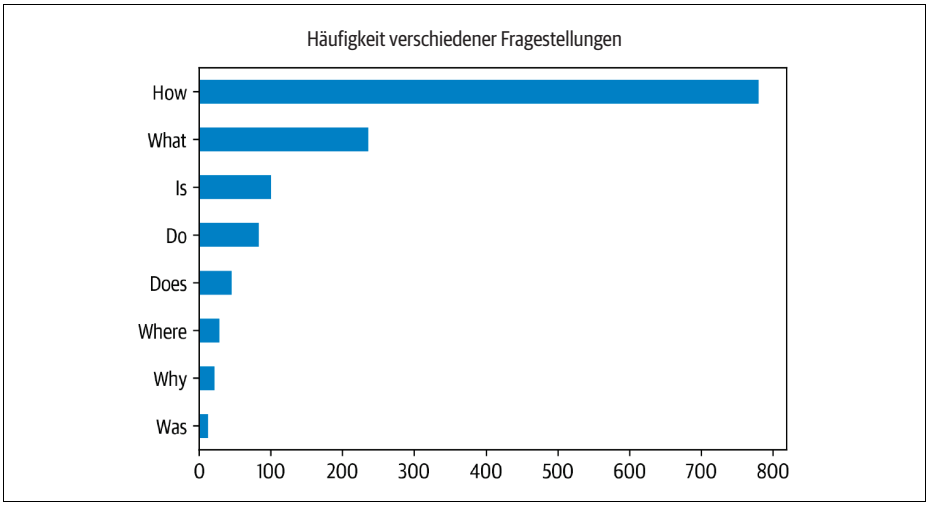
```
start_idx = sample_df["answers.answer_start"].iloc[0][0]
end_idx = start_idx + len(sample_df["answers.text"].iloc[0][0])
sample_df["context"].iloc[0][start_idx:end_idx]
```

```
'this keyboard is compact'
```

Als Nächstes sollten wir ein Gefühl dafür bekommen, welche Arten von Fragestellungen im Trainingsdatensatz enthalten sind, indem wir die Fragen zählen, die jeweils mit bestimmten gängigen Einstiegswörtern beginnen:

```
counts = {}
question_types = ["What", "How", "Is", "Does", "Do", "Was", "Where", "Why"]
```

```
for q in question_types:
    counts[q] = dfs["train"]["question"].str.startswith(q).value_counts()[True]

pd.Series(counts).sort_values().plot.barh()
plt.title("Häufigkeit verschiedener Fragestellungen")
plt.show()
```



Wie wir feststellen können, kommen Fragen, die mit »What«, »How« und »Is« beginnen, am häufigsten vor. Sehen wir uns ein paar Beispiele für diese Fragen an:

```
for question_type in ["How", "What", "Is"]:
    for question in (
        dfs["train"][dfs["train"].question.str.startswith(question_type)]
        .sample(n=3, random_state=42)['question']):
        print(question)

How is the camera?
How do you like the control?
How fast is the charger?
What is direction?
What is the quality of the construction of the bag?
What is your impression of the product?
Is this how zoom works?
Is sound clear?
Is it a wireless keyboard?
```

Das Stanford Question Answering Dataset (SQuAD)

Das dem SubjQA-Datensatz zugrunde gelegte Format *(Frage, Rezension, [Antwortsätze])* wird häufig in extraktiven QA-Datensätzen verwendet und wurde im Rahmen des Stanford Question Answering Dataset (SQuAD) entwickelt.[5] Dies ist ein

5 P. Rajpurkar et al., »SQuAD: 100,000+ Questions for Machine Comprehension of Text« (*https://arxiv.org/abs/1606.05250*), (2016).

berühmter Datensatz, der häufig verwendet wird, um die Fähigkeit von Maschinen zu testen, eine Textpassage zu lesen und Fragen dazu zu beantworten. Der Datensatz wurde erstellt, indem Hunderte von englischen Artikeln aus Wikipedia entnommen wurden, jeder Artikel in Absätze unterteilt und dann Crowdworker gebeten wurden, eine Reihe von Fragen und Antworten für jeden Absatz zu erstellen. In der ersten Version von SQuAD wurde gewährleistet, dass jede Antwort auf eine Frage in der entsprechenden Passage vorkommt. Allerdings dauerte es nicht lange, bis die Sequenzmodelle bei der Extraktion des richtigen Textabschnitts, der die Antwort enthält, besser abschnitten als Menschen. Um die Aufgabe zu erschweren, wurde SQuAD 2.0 entwickelt, indem SQuAD 1.1 um eine Reihe von Gegenfragen erweitert wurde, die für eine bestimmte Passage relevant sind, aber nicht aus dem Text allein beantwortet werden können.[6] Der Anfang 2022 aktuelle »State of the Art« (SOTA) ist in Abbildung 7-3 dargestellt, wobei die meisten Modelle seit dem Jahr 2019 die menschliche Leistung übertreffen.

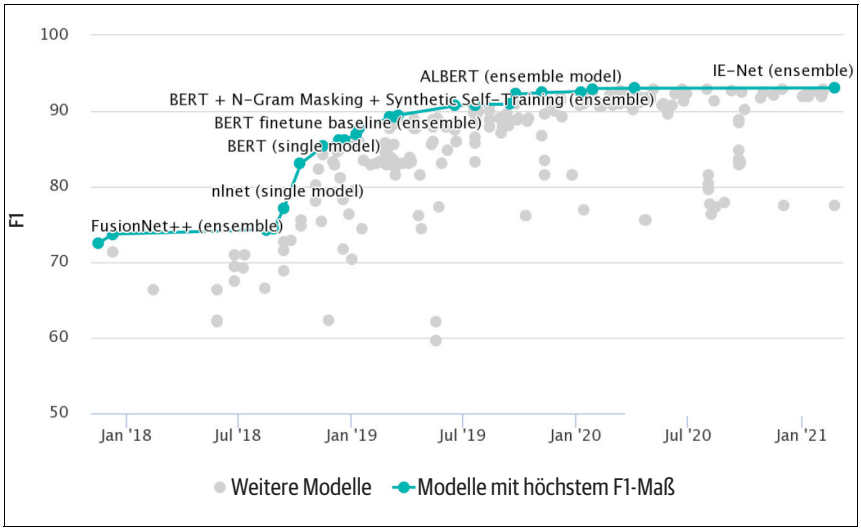


Abbildung 7-3: Fortschritte beim SQuAD-2.0-Benchmark (Diagramm von Papers with Code)

Diese übermenschliche Leistung scheint jedoch kein echtes Leseverstehen widerzuspiegeln, da die Antworten auf die »unbeantwortbaren« Fragen in der Regel anhand von Mustern in den Texten, wie z.B. Antonymen, identifiziert werden können. Zur Lösung dieser Probleme hat Google den Datensatz Natural Questions (NQ) veröffentlicht[7], bei dem es um Fragen nach Fakten geht, die von Nutzern der Google-Suche gestellt wurden. Die Antworten im NQ-Datensatz sind viel länger als die im SQuAD-Datensatz und stellen eine anspruchsvollere Benchmark dar.

6 P. Rajpurkar, R. Jia, and P. Liang, »Know What You Don't Know: Unanswerable Questions for SQuAD« (*https://arxiv.org/abs/1806.03822*), (2018).

7 T. Kwiatkowski et al., »Natural Questions: A Benchmark for Question Answering Research«, *Transactions of the Association for Computational Linguistics* 7 (März 2019): 452–466, *http://dx.doi.org/10.1162/tacl_a_00276*.

Nachdem wir nun unseren Datensatz ein wenig unter die Lupe genommen haben, können wir der Frage nachgehen, wie es Transformer-Modellen gelingt, Antworten aus einem Text zu extrahieren.

Antworten aus einem Text extrahieren

Als Erstes müssen wir für unser QA-System einen Weg finden, wie wir den für die mögliche Antwort relevanten Textabschnitt (Answer Span) in einer Kundenrezension identifizieren können. Wenn wir z. B. eine Frage wie »Ist sie wasserdicht?« haben und der Text der Rezension lautet »Diese Uhr ist bis zu einer Tiefe von 30 m wasserdicht«, dann sollte das Modell »bis zu einer Tiefe von 30 m wasserdicht« ausgeben. Zu diesem Zweck sollten wir wissen, wie wir:

- die Problemstellung beim überwachten Lernen (engl. Supervised Learning) formulieren,
- die Texte für QA-Aufgaben tokenisieren sowie codieren und
- mit langen Passagen umgehen, die die maximale Kontextlänge eines Modells überschreiten.

Beginnen wir mit einem Blick darauf, wie das Problem umrissen werden kann.

Span-Classification

Der gebräuchlichste Ansatz, Antworten aus Text zu extrahieren, besteht darin, das Problem als *Span-Classification*-Aufgabe zu formulieren, bei der die Anfangs- und Endtokens eines zur Beantwortung der Frage relevanten Textabschnitts (Answer Span) als Labels dienen, die ein Modell vorhersagen muss. Dieser Vorgang wird in Abbildung 7-4 veranschaulicht.

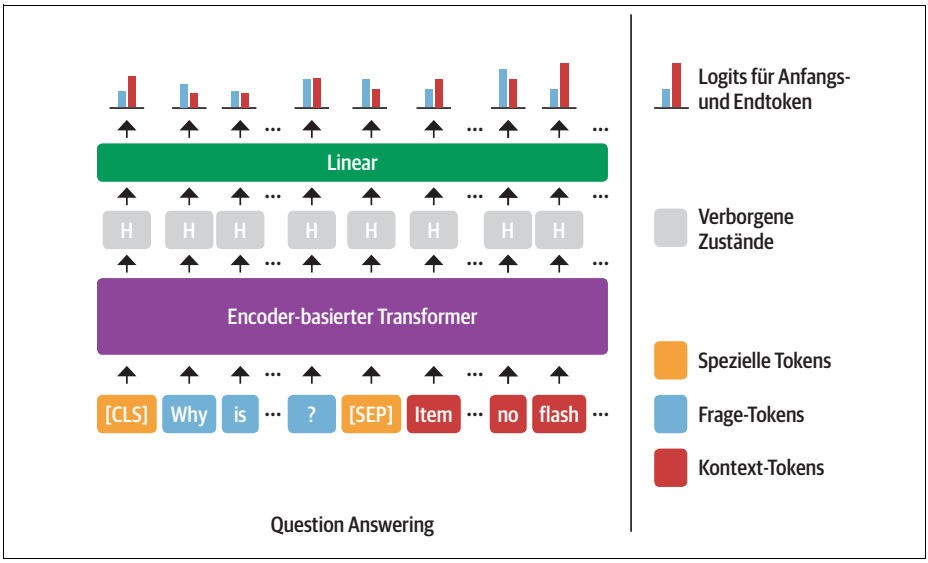


Abbildung 7-4: Der Head für die Span-Classification für QA-Aufgaben

Da unser Trainingssatz mit nur 1.295 Beispielen relativ klein ist, ist es eine gute Strategie, mit einem Sprachmodell zu beginnen, das bereits auf einem großen QA-Datensatz wie SQuAD feingetunt wurde. Diese Modelle verfügen in der Regel über eine ausgeprägte Lesekompetenz und bieten eine gute Grundlage, auf der ein akkurateres System aufgebaut werden kann. Dies ist ein etwas anderer Ansatz als in den vorangegangenen Kapiteln, in denen wir in der Regel mit einem vortrainierten Modell begonnen und den aufgabenspezifischen Head selbst feingetunt haben. In Kapitel 2 mussten wir zum Beispiel den Head für die Klassifizierung feintunen, um die Anzahl der Kategorien an den vorliegenden Datensatz anzupassen. Bei dem extraktiven QA können wir gleich mit einem feingetunten Modell beginnen, da die Struktur der Labels in allen Datensätzen gleich ist.

Auf dem Hugging Face Hub (*https://oreil.ly/dzCsC*) können Sie sich zahlreiche extraktive QA-Modelle auflisten lassen, indem Sie auf die Schaltfläche »Models« klicken und dann nach »squad« suchen (siehe Abbildung 7-5).

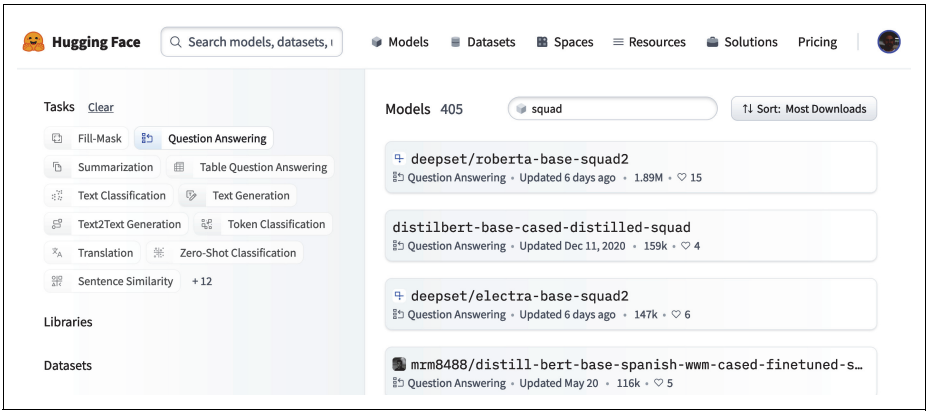


Abbildung 7-5: Eine Auswahl an extraktiven QA-Modellen auf dem Hugging Face Hub

Wie Sie sehen, stehen zum Zeitpunkt des Verfassens dieses Abschnitts mehr als 350 QA-Modelle zur Auswahl – für welches sollten Sie sich also entscheiden? Im Allgemeinen hängt die Antwort von verschiedenen Faktoren ab, z. B. davon, ob Ihr Korpus ein- oder mehrsprachig ist, und von den Einschränkungen, die sich ergeben, wenn Sie das Modell in einer Produktionsumgebung einsetzen. Tabelle 7-2 listet einige Modelle auf, die eine gute Grundlage bieten, auf der Sie aufbauen können.

Tabelle 7-2: Transformer-Modelle, die auf SQuAD 2.0 feingetunt wurden und als Baseline dienen

Transformer	Beschreibung	Anzahl an Parametern	F_1-Maß auf SQuAD 2.0
MiniLM	Eine schlankere (»distilled«) Version vom BERT-Base-Modell, die 99 % der Leistung beibehält und doppelt so schnell ist	66 Mio.	79,5

Tabelle 7-2: Transformer-Modelle, die auf SQuAD 2.0 feingetunt wurden und als Baseline dienen (Fortsetzung)

Transformer	Beschreibung	Anzahl an Parametern	F_1-Maß auf SQuAD 2.0
RoBERTa-base	RoBERTa-Modelle sind leistungsfähiger als die entsprechenden BERT-Modelle und können auf den meisten QA-Datensätzen mit einer einzigen GPU feingetunt werden.	125 Mio.	83,0
ALBERT-XXL	Leistung auf SQuAD 2.0 ist State of the Art, allerdings rechenintensiv und schwierig zu implementieren	235 Mio.	88,1
XLM-RoBERTa-large	Mehrsprachiges Modell für 100 Sprachen mit ausgezeichneter Zero-Shot-Leistung	570 Mio.	83,8

Im Rahmen dieses Kapitels werden wir ein feingetuntes MiniLM-Modell verwenden, da es schnell zu trainieren ist und es uns ermöglicht, die Verfahren, die wir erkunden werden, zügig zu variieren.[8] Wie üblich benötigen wir zunächst einen Tokenizer, mit dem wir unsere Texte codieren können. Sehen wir uns also einmal an, wie das bei QA-Aufgaben funktioniert.

Einen Text für QA-Aufgaben tokenisieren

Zur Codierung unserer Texte laden wir wie üblich zunächst den Checkpoint des MiniLM-Modells vom Hugging Face Hub (*https://oreil.ly/df5Cu*):

```
from transformers import AutoTokenizer

model_ckpt = "deepset/minilm-uncased-squad2"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
```

Sehen wir uns das Modell in Aktion an und versuchen wir zunächst, eine Antwort aus einer kurzen Textpassage zu extrahieren. Bei extraktiven QA-Aufgaben werden die Eingaben als (Frage-, Kontext-)Paare bereitgestellt, weshalb wir sie beide wie folgt an den Tokenizer übergeben:

```
question = "How much music can this hold?"
context = """An MP3 is about 1 MB/minute, so about 6000 hours depending on \
file size."""
inputs = tokenizer(question, context, return_tensors="pt")
```

Hier haben wir angegeben, dass PyTorch-`Tensor`-Objekte zurückgegeben werden sollen, da wir sie benötigen, um sie im Rahmen des Forward-Pass durch das Modell laufen zu lassen. Wenn wir die tokenisierten Eingaben in Form einer Tabelle betrachten:

input_ids	101	2129	2172	2189	2064	2023	...	5834	2006	5371	2946	1012	102
token_type_ids	0	0	0	0	0	0	...	1	1	1	1	1	1
attention_mask	1	1	1	1	1	1	...	1	1	1	1	1	1

8 W. Wang et al., »MINILM: Deep Self-Attention Distillation for Task-Agnostic Compression of Pre-Trained Transformers« (*https://arxiv.org/abs/2002.10957*), (2020).

können wir die uns bereits bekannten Tensoren `input_ids` und `attention_mask` erkennen, während der Tensor `token_type_ids` angibt, welcher Teil der Eingaben der Frage und welcher dem Kontext entspricht (eine 0 steht für ein Token, das einer Frage entstammt, eine 1 für ein Token, das einem Kontext zuzuordnen ist).[9]

Vollziehen wir nun nach, wie der Tokenizer die Eingaben bei QA-Aufgaben formatiert. Hierzu können wir den `input_ids`-Tensor decodieren:

```
print(tokenizer.decode(inputs["input_ids"][0]))
```

```
[CLS] how much music can this hold? [SEP] an mp3 is about 1 mb / minute, so
about 6000 hours depending on file size. [SEP]
```

Wie wir feststellen, haben die Eingaben für jedes QA-Beispiel das folgende Format:

```
[CLS] Frage-Tokens [SEP] Kontext-Tokens [SEP]
```

wobei die Position des ersten `[SEP]`-Tokens durch die `token_type_ids` bestimmt wird. Da unser Text nun tokenisiert ist, müssen wir nur noch das Modell mit einem für das QA spezifischen Head instanziieren und die Eingaben im Rahmen des Forward-Pass durch das Modell laufen lassen:

```
import torch
from transformers import AutoModelForQuestionAnswering

model = AutoModelForQuestionAnswering.from_pretrained(model_ckpt)

with torch.no_grad():
    outputs = model(**inputs)
print(outputs)
```

```
QuestionAnsweringModelOutput(loss=None, start_logits=tensor([[-0.9862, -4.7750,
         -5.4025, -5.2378, -5.2863, -5.5117, -4.9819, -6.1880,
         -0.9862,  0.2596, -0.2144, -1.7136,  3.7806,  4.8561, -1.0546, -3.9097,
         -1.7374, -4.5944, -1.4278,  3.9949,  5.0390, -0.2018, -3.0193, -4.8549,
         -2.3107, -3.5110, -3.5713, -0.9862]]), end_logits=tensor([[-0.9623,
         -5.4733, -5.0326, -5.1639, -5.4278, -5.5151, -5.1749, -4.6233,
         -0.9623, -3.7855, -0.8715, -3.7745, -3.0161, -1.1780,  0.1758, -2.7365,
          4.8934,  0.3046, -3.1761, -3.2762,  0.8937,  5.6606, -0.3623, -4.9554,
         -3.2531, -0.0914,  1.6211, -0.9623]]), hidden_states=None,
attentions=None)
```

Hier sehen wir, dass wir ein `QuestionAnsweringModelOutput`-Objekt als Ausgabe des QA-spezifischen Heads erhalten. Wie in Abbildung 7-4 dargestellt, entspricht der QA-spezifische Head einer linearen Schicht, die die verborgenen Zustände des Encoders entgegennimmt und die Logits für die Anfangs- und Endposition der relevanten Textabschnitte (Answer Span) berechnet.[10] Das bedeutet, dass wir das QA als eine Form der Klassifizierung von Tokens behandeln, ähnlich wie wir es bei der Named Entity Recognition in Kapitel 4 umgesetzt haben. Um die Ausgaben in einen

9 Beachten Sie, dass der `token_type_ids`-Tensor nicht in allen Transformer-Modellen vorkommt. Bei BERT-ähnlichen Modellen wie MiniLM werden die `token_type_ids` auch während des Pretrainings verwendet, um sie für die Aufgabe der Vorhersage des nächsten Satzes einzubeziehen.

10 In Kapitel 2 erfahren Sie Näheres darüber, wie diese verborgenen Zustände extrahiert werden können.

für die Antwort relevanten Textabschnitt (Answer Span) zu überführen, müssen wir zunächst die Logits für das Anfangs- und das Endtoken ermitteln:

```
start_logits = outputs.start_logits
end_logits = outputs.end_logits
```

Wenn wir den Shape dieser Logits mit dem der Input-IDs vergleichen:

```
print(f"Shape der Input-IDs: {inputs.input_ids.size()}")
print(f"Shape der Logits für Anfangstoken: {start_logits.size()}")
print(f"Shape der Logits für Endtoken: {end_logits.size()}")
```

```
Shape der Input-IDs: torch.Size([1, 28])
Shape der Logits für Anfangstoken: torch.Size([1, 28])
Shape der Logits für Endtoken: torch.Size([1, 28])
```

stellen wir fest, dass mit jedem eingegebenen Token zwei Logits (einer für den Anfang und einer für das Ende) verknüpft sind. Wie in Abbildung 7-6 dargestellt, entsprechen höhere, positive Logits wahrscheinlicheren Kandidaten für die Anfangs- und Endtokens. In diesem Beispiel sehen wir, dass das Modell die höchsten Logits für das Anfangstoken den Zahlen »1« und »6000« zuordnet, was schlüssig ist, da sich unsere Frage auf bestimmte Mengenangaben bezieht. Gleichermaßen können wir erkennen, dass für das Endtoken die höchsten Logits für »minute« und »hours« ermittelt wurden.

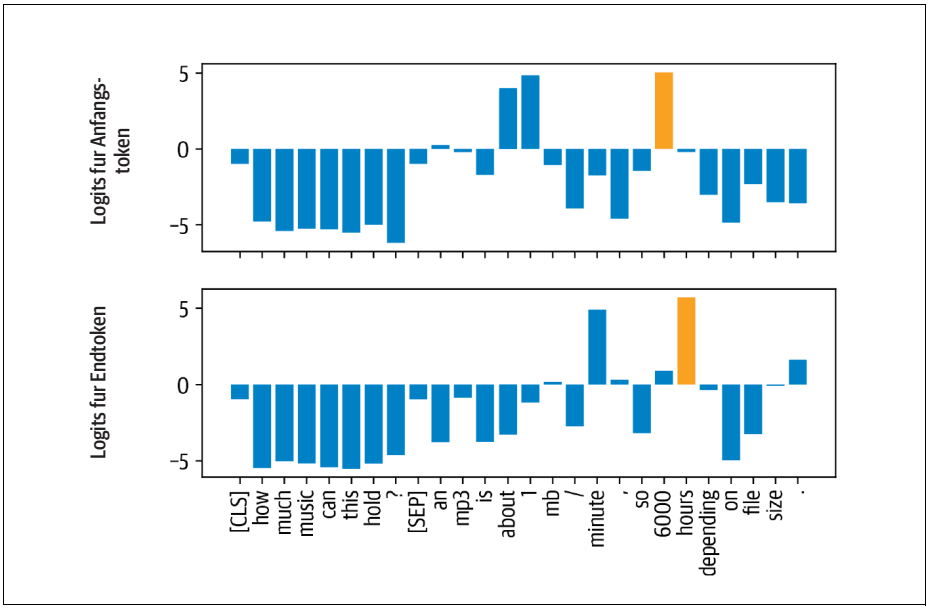


Abbildung 7-6: Vorhergesagte Logits für das Anfangs- und das Endtoken. Das Token mit dem höchsten Wert ist in Hellgrau (bzw. im E-Book in Orange) zu sehen.

Um die endgültige Antwort zu erhalten, können wir die `argmax`-Funktion auf die Logits für das Anfangs- und das Endtoken anwenden und dann die Textpassage, in der die Antwort enthalten ist (Answer Span), mittels Slicing aus den Eingaben ex-

trahieren. Dies können wir mit dem nachfolgenden Code erledigen, in dem das Ergebnis anschließend noch decodiert wird, sodass wir uns den resultierenden Text anzeigen lassen können:

```
import torch

start_idx = torch.argmax(start_logits)
end_idx = torch.argmax(end_logits) + 1
answer_span = inputs["input_ids"][0][start_idx:end_idx]
answer = tokenizer.decode(answer_span)
print(f"Frage: {question}")
print(f"Antwort: {answer}")
```

```
Frage: How much music can this hold?
Antwort: 6000 hours
```

Großartig, es hat funktioniert! Praktischerweise sind all diese Vor- und Nachverarbeitungsschritte in der entsprechenden Pipeline der 🤗 Transformers-Bibliothek bereits integriert. Greifen wir nun auf die Pipeline zurück und instanziieren wir sie, wobei wir unseren Tokenizer und das feingetunte Modell wie folgt übergeben:

```
from transformers import pipeline

pipe = pipeline("question-answering", model=model, tokenizer=tokenizer)
pipe(question=question, context=context, topk=3)
```

```
[{'score': 0.26516005396842957,
  'start': 38,
  'end': 48,
  'answer': '6000 hours'},
 {'score': 0.2208300083875656,
  'start': 16,
  'end': 48,
  'answer': '1 MB/minute, so about 6000 hours'},
 {'score': 0.10253632068634033,
  'start': 16,
  'end': 27,
  'answer': '1 MB/minute'}]
```

Zusätzlich zur Antwort gibt die Pipeline auch die vom Modell geschätzten Wahrscheinlichkeiten im Feld `score` zurück (die wir durch Anwendung der Softmax-Funktion auf die Logits erhalten). Das erweist sich als praktisch, wenn wir mehrere Antworten innerhalb eines einzelnen Kontexts vergleichen möchten. Wir haben auch gezeigt, dass wir das Modell dazu veranlassen können, mehrere Antworten vorherzusagen, indem wir den Parameter `topk` angeben. Mitunter gibt es Fragen, auf die es keine Antwort gibt, wie dies beispielsweise auch bei manchen Beispielen im SubjQA-Datensatz der Fall ist, bei denen das `answers.answer_start`-Feld leer ist. In diesen Fällen weist das Modell dem `[CLS]`-Token einen hohen Score (bzw. eine hohe Wahrscheinlichkeit) zu, dass es das Start- und Endtoken darstellt, und die Pipeline ordnet diese Ausgabe wiederum einem leeren String zu:

```
pipe(question="Why is there no data?", context=context,
     handle_impossible_answer=True)
```

```
{'score': 0.9068416357040405, 'start': 0, 'end': 0, 'answer': ''}
```

In unserem einfachen Beispiel haben wir die Start- und Endindizes erhalten, indem uns die `argmax`-Funktion jeweils den Index der Maximalwerte der entsprechenden Logits geliefert hat. Diese Heuristik kann jedoch zu Antworten führen, die nicht in den Kontext fallen, da Tokens ausgewählt werden könnten, die zur Frage und nicht zum Kontext gehören. Daher ermittelt die Pipeline genau genommen die beste Kombination aus den Start- und Endindizes, die verschiedenen Einschränkungen unterliegen, z.B. dass sie in den Gültigkeitsbereich fallen, dass die Startindizes vor den Endindizes liegen müssen usw.

Mit langen Passagen umgehen

Eine Schwierigkeit bei Modellen, die für das Leseverstehen (engl. Reading Comprehension) verwendet werden, besteht darin, dass der Kontext oft mehr Tokens enthält, als es die maximale Sequenzlänge des Modells (die in der Regel höchstens einige Hundert Tokens beträgt) zulässt. Wie aus Abbildung 7-7 hervorgeht, enthält ein beträchtlicher Teil des SubjQA-Trainingsdatensatzes Frage-Kontext-Paare, die die maximale Kontextlänge des MiniLM-Modells von 512 Tokens überschreiten würden.

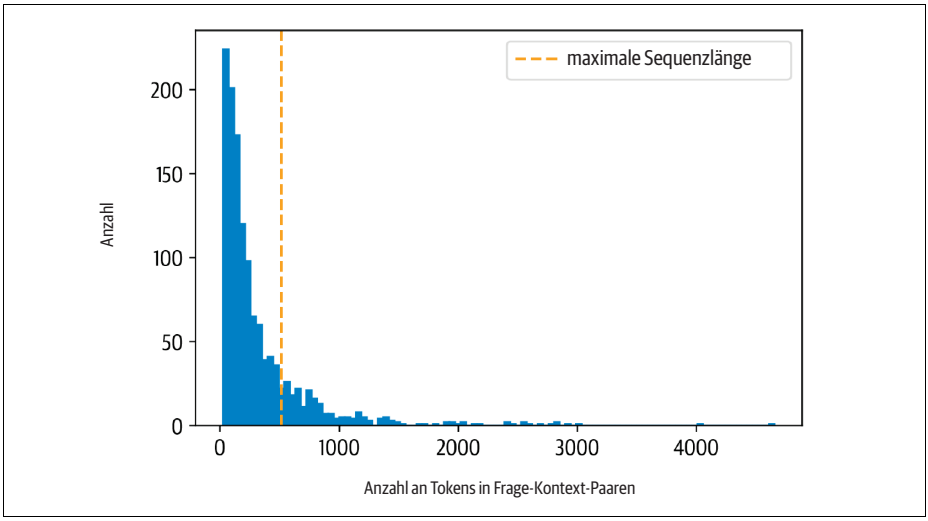


Abbildung 7-7: Verteilung der Anzahl der Tokens für alle Frage-Kontext-Paare im SubjQA-Trainingsdatensatz

Bei den vorherigen Aufgaben, wie z.B. der Textklassifizierung, haben wir lange Texte einfach abgeschnitten bzw. gestutzt, in der Annahme, dass in dem Embedding des `[CLS]`-Tokens genügend Informationen enthalten waren, um akkurate Vorhersagen treffen zu können. Für das QA ist diese Strategie jedoch problematisch, da die Antwort auf eine Frage nahe am Ende des Kontexts stehen könnte und dementsprechend nicht einbezogen bzw. entfernt werden würde. Wie in Abbildung 7-8 veranschaulicht, besteht die gängige Vorgehensweise darin, ein *Sliding Window* (gleitendes Fenster) auf die Eingaben anzuwenden, wobei jedes Window eine Passage von Tokens enthält, die in den Kontext des Modells fallen.

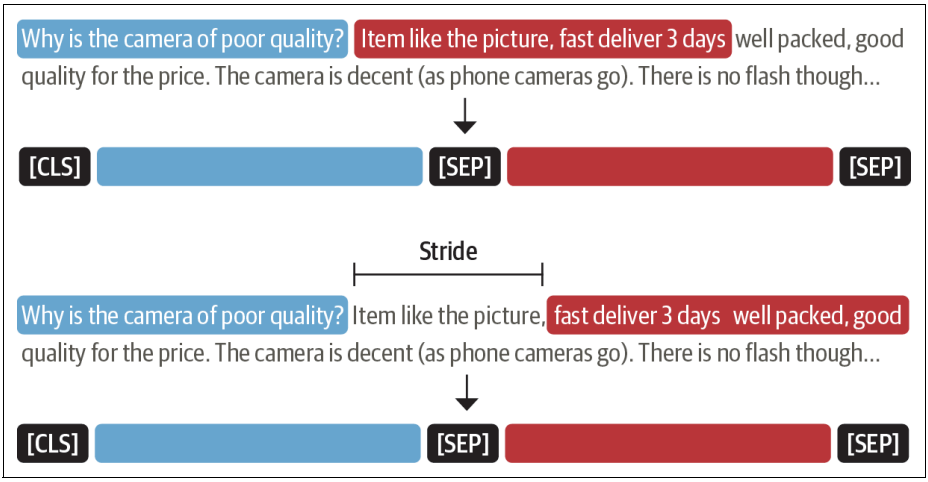


Abbildung 7-8: Wie mehrere Frage-Kontext-Paare für lange Dokumente durch ein Sliding Window erzeugt werden – die erste Hervorhebung entspricht jeweils der Frage, während die zweite den in jedem Window erfassten Kontext darstellt.

In der 🤗 Transformers-Bibliothek können wir `return_overflowing_tokens=True` für den Tokenizer setzen, um die Verwendung eines Sliding Windows zu aktivieren. Die Länge bzw. Größe des Sliding Window wird durch das Argument `max_seq_length` festgelegt, und die Länge des Stride (d.h. die Schrittweite) durch `doc_stride` vorgegeben. Nehmen wir das erste Beispiel aus unserem Trainingsdatensatz und legen wir eine relativ geringe Länge für das Window fest, um uns einmal vor Augen zu führen, wie das Ganze funktioniert:

```
example = dfs["train"].iloc[0][["question", "context"]]
tokenized_example = tokenizer(example["question"], example["context"],
                              return_overflowing_tokens=True, max_length=100,
                              stride=25)
```

Im vorliegenden Fall erhalten wir eine Liste von `input_ids`, und zwar jeweils eine je Window. Bestimmen wir nun die Anzahl der Tokens, die wir in den Windows erhalten haben:

```
for idx, window in enumerate(tokenized_example["input_ids"]):
    print(f"Window #{idx} umfasst {len(window)} Tokens")
```

```
Window #0 umfasst 100 Tokens
Window #1 umfasst 88 Tokens
```

Wenn wir die Eingaben decodieren, können wir zu guter Letzt noch herausfinden, inwieweit sich die beiden Windows überschneiden:

```
for window in tokenized_example["input_ids"]:
    print(f"{tokenizer.decode(window)} \n")
```

```
[CLS] how is the bass? [SEP] i have had koss headphones in the past, pro 4aa and
qz - 99. the koss portapro is portable and has great bass response. the work
great with my android phone and can be " rolled up " to be carried in my
motorcycle jacket or computer bag without getting crunched. they are very light
```

```
and don't feel heavy or bear down on your ears even after listening to music
with them on all day. the sound is [SEP]

[CLS] how is the bass? [SEP] and don't feel heavy or bear down on your ears even
after listening to music with them on all day. the sound is night and day better
than any ear - bud could be and are almost as good as the pro 4aa. they are "
open air " headphones so you cannot match the bass to the sealed types, but it
comes close. for $ 32, you cannot go wrong. [SEP]
```

Da wir nun in Erfahrung gebracht haben, wie QA-Modelle Antworten aus einem Text extrahieren können, sind wir nun so weit, dass wie einen Blick auf die anderen Komponenten werfen können, die wir zum Aufbau einer vollständigen (»End-to-End«-)QA-Pipeline benötigen.

Die Haystack-Bibliothek zum Aufbau einer QA-Pipeline verwenden

In unserem einfachen Beispiel haben wir dem Modell sowohl die Frage als auch den Kontext, aus dem die Antwort extrahiert werden sollte, zur Verfügung gestellt. In der Realität werden die Benutzer unseres Systems jedoch nur eine Frage zu einem Produkt stellen, sodass wir eine Möglichkeit benötigen, die relevanten Passagen aus allen Rezensionen in unserem Korpus herauszufiltern. Eine Möglichkeit wäre, alle Rezensionen zu einem bestimmten Produkt miteinander zu verknüpfen und sie als einen einzigen langen Kontext in das Modell einzuspeisen. Dieser Ansatz ist zwar einfach, hat aber den Nachteil, dass der Kontext extrem lang werden kann und dadurch die Abfrage unserer Benutzer mit einer inakzeptablen Latenzzeit verbunden ist. Nehmen wir zum Beispiel an, dass jedes Produkt im Durchschnitt 30 Rezensionen hat und die Verarbeitung jeder Rezension 100 Millisekunden in Anspruch nimmt. Wenn wir alle Rezensionen verarbeiten müssten, um eine Antwort zu erhalten, würde dies zu einer durchschnittlichen Latenzzeit von 3 Sekunden pro Benutzeranfrage führen – viel zu lang für eine E-Commerce-Webseite!

Aus diesem Grund basieren moderne QA-Systeme in der Regel auf der *Retriever-Reader*-Architektur, die aus zwei Hauptkomponenten besteht:

Retriever
: Ist dafür zuständig, relevante Dokumente für eine bestimmte Abfrage abzurufen. Retriever werden normalerweise als *Sparse* oder *Dense* kategorisiert. Sparse-Retriever verwenden Worthäufigkeiten, um jedes Dokument und jede Abfrage als dünnbesetzten (engl. sparse) Vektor darzustellen.[11] Die Relevanz einer Abfrage und eines Dokuments wird dann durch Berechnung des Skalarprodukts der Vektoren bestimmt. Dense-Retriever hingegen verwenden Encoder (wie Transformer-Modelle), um die Abfrage und das Dokument als kontextualisierte Einbettungen bzw. Contextualized Embeddings (die dichtbesetzten Vektoren entsprechen) darzustellen. Diese Embeddings codieren die semantische Bedeutung und ermöglichen es den Dense-Retrievern, die Suchgenauigkeit zu verbessern, indem sie den Inhalt der Abfrage verstehen.

11 Ein Vektor wird als dünnbesetzt bezeichnet, wenn die meisten seiner Elemente null sind.

Reader

Ist dafür zuständig, eine Antwort aus den vom Retriever bereitgestellten Dokumenten zu extrahieren. Der Reader ist in der Regel ein Modell für das Leseverstehen (engl. Reading Comprehension Model), obwohl wir am Ende des Kapitels auch Beispiele für Modelle sehen werden, die Freiformantworten generieren können.

Wie in Abbildung 7-9 dargestellt, kann es auch andere Komponenten geben, die die vom Retriever abgerufenen Dokumente oder die vom Reader extrahierten Antworten nachverarbeiten. So kann es beispielsweise erforderlich sein, den abgerufenen Dokumenten eine neue Rangordnung zu geben, um schwer nachvollziehbare oder irrelevante Dokumente zu entfernen, die den Reader verwirren könnten. Ebenso ist es häufig erforderlich, die Antworten des Readers nachzuverarbeiten, wenn die richtige Antwort aus verschiedenen Passagen eines langen Dokuments stammt.

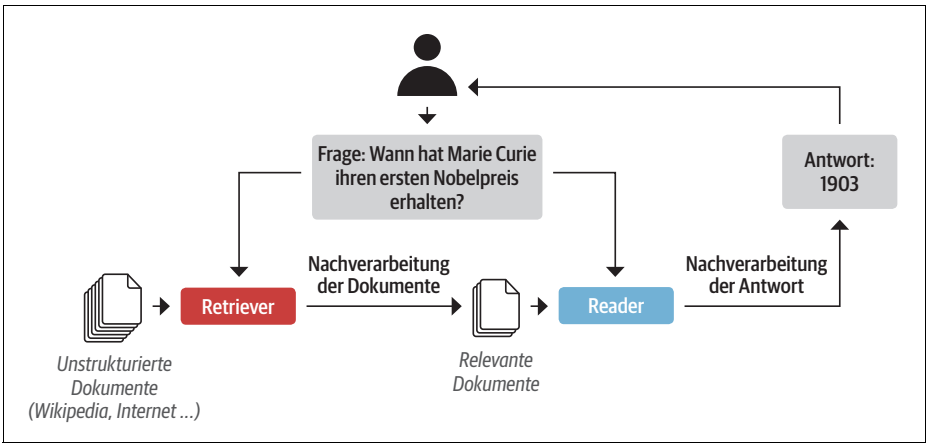


Abbildung 7-9: Die Retriever-Reader-Architektur moderner QA-Systeme

Zum Aufbau unseres QA-Systems verwenden wir die *Haystack*-Bibliothek (*https://haystack.deepset.ai*), die von deepset (*https://deepset.ai*), einem deutschen Unternehmen, das sich auf den Aufbau von NLP-basierten Anwendungen spezialisiert hat, entwickelt wurde. Die Haystack-Bibliothek basiert auf der Retriever-Reader-Architektur, abstrahiert einen Großteil der Komplexität, die mit dem Aufbau dieser Systeme verbunden ist, und ist eng mit der 🤗 Transformers-Bibliothek verzahnt.

Neben dem Retriever und dem Reader gibt es zwei weitere Komponenten, die beim Aufbau einer QA-Pipeline mit der Haystack-Bibliothek eine Rolle spielen:

Document Store

Eine dokumentenorientierte Datenbank, die Dokumente und Metadaten speichert, die dem Retriever zum Zeitpunkt der Abfrage zur Verfügung gestellt werden

Pipeline

Kombiniert alle Komponenten eines QA-Systems, um benutzerdefinierte Abfrageabläufe (engl. Query Flows), das Zusammenführen von Dokumenten aus mehreren Retrievern und mehr zu ermöglichen.

Im weiteren Verlauf des Abschnitts werden wir erörtern, wie wir diese Komponenten verwenden können, um in möglichst kurzer Zeit einen Prototyp einer QA-Pipeline zu erstellen. Anschließend werden wir untersuchen, wie wir die Leistungsfähigkeit des Systems verbessern können.

Dieses Kapitel wurde auf Basis von Version 0.9.0 der Haystack-Bibliothek verfasst. In Version 0.10.0 (*https://oreil.ly/qbqgv*) wurden die APIs zur Nutzung von Pipelines und zur Evaluierung umgestaltet, damit leichter überprüft werden kann, ob die Leistung des Systems durch den Retriever oder den Reader beeinträchtigt wird. Wenn Sie wissen wollen, wie der Code dieses Kapitels mit den neuen APIs aussieht, besuchen Sie bitte das zugehörige GitHub-Repository (*https://github.com/nlp-with-transformers/notebooks*).

Einen Document Store initialisieren

In der Haystack-Bibliothek stehen verschiedene Document Stores (Dokumentenspeicher) zur Auswahl, wobei jeder nur mit einem speziellen Satz von Retrievern gepaart werden kann. In Tabelle 7-3 können Sie nachvollziehen, inwieweit die Sparse- (TF-IDF, BM25) und Dense-Retriever (Embedding, DPR) mit den verfügbaren Document Stores kompatibel sind. Was all diese Akronyme bedeuten, werden wir im weiteren Verlauf dieses Kapitels erläutern.

Tabelle 7-3: Kompatibilität von Retrievern und Document Stores in der Haystack-Bibliothek

	In-Memory	Elasticsearch	FAISS	Milvus
TF-IDF	Ja	Ja	Nein	Nein
BM25	Nein	Ja	Nein	Nein
Embedding	Ja	Ja	Ja	Ja
DPR	Ja	Ja	Ja	Ja

Da wir uns in diesem Kapitel sowohl mit Sparse- als auch mit Dense-Retrievern befassen werden, verwenden wir den `ElasticsearchDocumentStore`, der mit beiden Retriever-Typen kompatibel ist. Elasticsearch ist eine Suchmaschine, die in der Lage ist, eine Vielzahl von Datentypen zu verarbeiten, darunter Textdaten, numerische Daten, raumbezogene Daten (»Geodaten«) als auch strukturierte und unstrukturierte Daten. Dank ihrer Fähigkeit, riesige Datenmengen zu speichern und sie mithilfe von Volltextsuchfunktionen schnell zu filtern, eignet sie sich besonders gut für die Entwicklung von QA-Systemen. Außerdem hat sie den Vorteil, dass sie den Industriestandard für Infrastrukturanalysen darstellt. Es ist also gut möglich, dass Ihr Unternehmen bereits über ein Cluster verfügt, mit dem Sie arbeiten können.

Damit wir den Document Store initialisieren können, müssen wir zunächst Elasticsearch herunterladen und installieren. Entsprechend dem Leitfaden (*https://oreil.ly/bgmKq*)[12] können wir die zum Zeitpunkt des Verfassens dieses Buchs neueste Version für Linux mit `wget` herunterladen und sie mit dem Kommandozeilenbefehl `tar` entpacken:

```
url = """https://artifacts.elastic.co/downloads/elasticsearch/\
elasticsearch-7.9.2-linux-x86_64.tar.gz"""
!wget -nc -q {url}
!tar -xzf elasticsearch-7.9.2-linux-x86_64.tar.gz
```

Als Nächstes müssen wir den Elasticsearch-Server starten. Da wir den gesamten Code in diesem Buch in Jupyter Notebooks ausführen, müssen wir die Python-Funktion `Popen()` verwenden, um einen neuen Prozess zu starten. Da wir schon dabei sind, können wir ihn auch gleich als Unterprozess (engl. Subprocess) im Hintergrund laufen lassen, indem wir den Kommandozeilenbefehl `chown` verwenden:

```
import os
from subprocess import Popen, PIPE, STDOUT

# Elasticsearch als Hintergrundprozess ausführen
!chown -R daemon:daemon elasticsearch-7.9.2
es_server = Popen(args=['elasticsearch-7.9.2/bin/elasticsearch'],
                  stdout=PIPE, stderr=STDOUT, preexec_fn=lambda: os.setuid(1))
# Warten, bis Elasticsearch läuft
!sleep 30
```

In der Funktion `Popen()` geben wir über `args` das Programm an, das wir ausführen möchten. Mit `stdout=PIPE` wird eine neue Pipe für die Standardausgabe erstellt, und mit `stderr=STDOUT` werden die Fehler in der gleichen Pipe gesammelt. Das Argument `preexec_fn` gibt die ID des Unterprozesses an, den wir verwenden möchten. Da Elasticsearch standardmäßig lokal auf Port 9200 läuft, können wir die Verbindung testen, indem wir eine HTTP-Anfrage (engl. Request) an `localhost` senden:

```
!curl -X GET "localhost:9200/?pretty"
```

```
{
  "name" : "96938eee37cd",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "ABGDdvbbRWmMb9Umz79HbA",
  "version" : {
    "number" : "7.9.2",
    "build_flavor" : "default",
    "build_type" : "docker",
    "build_hash" : "d34da0ea4a966c4e49417f2da2f244e3e97b4e6e",
    "build_date" : "2020-09-23T00:45:33.626720Z",
    "build_snapshot" : false,
    "lucene_version" : "8.6.2",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
```

12 Der Leitfaden enthält auch Installationsanweisungen für macOS und Windows.

```
  "tagline" : "You Know, for Search"
}
```

Unser Elasticsearch-Server ist nun einsatzbereit. Als Nächstes müssen wir den Document Store instanziieren:

```
from haystack.document_store.elasticsearch import ElasticsearchDocumentStore

# Liefert Document-Embedding zur späteren Verwendung mit Dense-Retriever
document_store = ElasticsearchDocumentStore(return_embedding=True)
```

Die `ElasticsearchDocumentStore`-Klasse erstellt standardmäßig zwei Indizes in Elasticsearch: einen mit dem Namen `document` für (Sie haben es erraten) die Speicherung von Dokumenten und einen weiteren mit dem Namen `label` für die Speicherung der annotierten Textbereiche, in denen die Antworten enthalten sind (Answer Span). Für den Moment werden wir nur den `document`-Index mit den SubjQA-Rezensionen befüllen. Die Document Stores, die in der Haystack-Bibliothek zur Verfügung stehen, erwarten eine Liste von Dictionarys, die jeweils die Schlüssel `text` und `meta` umfassen:

```
{
 "text": »<the-context>",
 "meta": {
 "field_01": »<additional-metadata>",
 "field_02": »<additional-metadata>",
 ...
 }
}
```

Die für `meta` hinterlegten Felder können dazu verwendet werden, im Rahmen des Abrufs (engl. Retrieval) Filter anzuwenden. Für unsere Zwecke werden wir die Spalten `title` (als `item_id`) und `id` (als `question_id`) des SubjQA-Datensatzes einbeziehen, sodass wir nach den IDs der Produkte und Fragen filtern können. Zudem hinterlegen wir noch, aus welchem Teildatensatz (engl. Split) das jeweilige Beispiel stammt, um auch diese Information im Rahmen der Filterung berücksichtigen zu können. Anschließend können wir über die Beispiele in den einzelnen `DataFrames` unter Verwendung einer Schleife iterieren und sie mithilfe der Methode `write_documents()` wie folgt zum Index hinzufügen:

```
for split, df in dfs.items():
    # Duplikate von Rezensionen ausschließen
    docs = [{"text": row["context"],
             "meta":{"item_id": row["title"], "question_id": row["id"],
                     "split": split}}
        for _,row in df.drop_duplicates(subset="context").iterrows()]
    document_store.write_documents(docs, index="document")

print(f"{document_store.get_document_count()} Dokumente geladen")
```

```
1615 Dokumente geladen
```

Großartig, wir haben alle unsere Rezensionen in einen Index geladen! Um den Index durchsuchen bzw. abfragen zu können, benötigen wir einen Retriever. Sehen wir uns also an, wie wir einen Retriever für Elasticsearch initialisieren können.

Einen Retriever initialisieren

Der Document Store von Elasticsearch kann mit jedem der Retriever, die in der Haystack-Bibliothek verfügbar sind, gepaart werden. Beginnen wir daher gleich mit einem Sparse-Retriever, der auf BM25 (kurz für »Best Match 25«) basiert. BM25 ist eine verbesserte Version des klassischen TF-IDF-(Term-Frequency-Inverse-Document-Frequency-)Algorithmus und bildet die Frage und den Kontext als dünnbesetzte Vektoren ab, die effizient in Elasticsearch durchsucht werden können. Der BM25-Score gibt an, wie stark der Text mit einer Suchanfrage übereinstimmt, und verbessert den TF-IDF-Algorithmus insofern, als die TF-Werte schnell zur Sättigung gebracht werden und die Dokumentenlänge normalisiert wird, sodass kurze Dokumente gegenüber langen bevorzugt werden.[13]

In der Haystack-Bibliothek wird für die `ElasticsearchRetriever`-Klasse standardmäßig der BM25-Retriever verwendet. Initialisieren wir nun die Klasse und geben wir den Document Store an, den wir durchsuchen möchten:

```
from haystack.retriever.sparse import ElasticsearchRetriever

es_retriever = ElasticsearchRetriever(document_store=document_store)
```

Betrachten wir als Nächstes eine einfache Abfrage für ein einzelnes Elektronikprodukt, das im Trainingsdatensatz enthalten ist. Bei rezensionsbasierten QA-Systemen wie unserem ist es wichtig, die Abfragen auf ein einziges Produkt zu beschränken, da der Retriever sonst Rezensionen zu Produkten finden würde, die nichts mit der Anfrage des Benutzers zu tun haben. Wenn Sie beispielsweise die Frage »Ist die Qualität der Kamera gut?« ohne einen Produktfilter stellen, könnten Sie Rezensionen zu Telefonen erhalten, obwohl der Benutzer eigentlich nach einer bestimmten Laptop-Kamera fragt. Die ASIN-Werte in unserem Datensatz sind an sich etwas kryptisch. Allerdings können wir sie mithilfe von Online-Tools wie *amazon ASIN* (*https://amazon-asin.com*) entschlüsseln oder indem wir einfach den Wert der `item_id` an die URL *www.amazon.com/dp/* anhängen. Die Artikel-ID (`item_id`), die wir im folgenden Codebeispiel zur Filterung verwenden, entspricht einem Fire-Tablet von Amazon. Finden wir heraus, ob sich das Gerät zum Lesen eignet, indem wir die `retrieve()`-Methode des Retrievers verwenden:

```
item_id = "B0074BW614"
query = "Is it good for reading?"
retrieved_docs = es_retriever.retrieve(
    query=query, top_k=3, filters={"item_id":[item_id], "split":["train"]})
```

Hier haben wir mit dem Argument `top_k` angegeben, wie viele Dokumente zurückgegeben werden sollen. Zudem haben wir einen Filter auf die Schlüssel `item_id` und `split` angewandt, die im Feld `meta` unserer Dokumente enthalten sind. Jedes Element von `retrieved_docs` ist ein Haystack-`Document`-Objekt, das zur Darstellung von Dokumenten verwendet wird und den Abfrage- bzw. Document-Score des Re-

13 Eine ausführliche Erklärung des Document Scoring mittels TF-IDF und BM25 finden Sie in Kapitel 23 von *Speech and Language Processing* von D. Jurafsky und J. H. Martin (Prentice Hall), 3. Auflage.

trievers zusammen mit anderen Metadaten enthält. Werfen wir nun einen Blick auf eines der abgerufenen Dokumente:

```
print(retrieved_docs[0])
```

```
{'text': 'This is a gift to myself.  I have been a kindle user for 4 years and
this is my third one.  I never thought I would want a fire for I mainly use it
for book reading.  I decided to try the fire for when I travel I take my laptop,
my phone and my iPod classic.  I love my iPod but watching movies on the plane
with it can be challenging because it is so small. Laptops battery life is not
as good as the Kindle.  So the Fire combines for me what I needed all three to
do. So far so good.', 'score': 6.243799, 'probability': 0.6857824513476455,
'question': None, 'meta': {'item_id': 'B0074BW614', 'question_id':
'868e311275e26dbafe5af70774a300f3', 'split': 'train'}, 'embedding': None, 'id':
'252e83e25d52df7311d597dc89eef9f6'}
```

Zusätzlich zum Text des Dokuments sehen wir den `score`, den Elasticsearch in Bezug auf die Relevanz des Dokuments für die Abfrage berechnet hat (höhere Scores bedeuten eine größere Übereinstimmung). Technisch gesehen stützt sich Elasticsearch bei der Indexierung und Suche auf Lucene (*https://lucene.apache.org*) und verwendet daher standardmäßig die *Practical Scoring Function* von Lucene. Die genauen Details der Scoring-Funktion finden Sie in der Elasticsearch-Dokumentation (*https://oreil.ly/b1Seu*). Kurz gesagt, filtert sie zunächst die infrage kommenden Dokumente, indem sie eine boolesche Überprüfung vornimmt (stimmt das Dokument mit der Abfrage überein?), und wendet dann ein Ähnlichkeitsmaß an, das darauf fußt, dass sowohl das Dokument als auch die Abfrage als Vektoren dargestellt werden.

Da es uns nun möglich ist, relevante Dokumente abzurufen, benötigen wir als Nächstes noch eine Möglichkeit, Antworten aus ihnen zu extrahieren. Hier kommt der Reader ins Spiel. Schauen wir uns also an, wie wir unser MiniLM-Modell mit der Haystack-Bibliothek laden können.

Einen Reader intitialisieren

In der Haystack-Bibliothek gibt es zwei Arten von Readern, die Sie verwenden können, um Antworten aus einem gegebenen Kontext zu extrahieren:

`FARMReader`
: Basiert auf dem *FARM*-Framework (*https://farm.deepset.ai*) von deepset, mit dem Transformer-Modelle feingetunt und deployt werden können. Kompatibel mit Modellen, die mit der 🤗 Transformers-Bibliothek trainiert wurden, und kann Modelle direkt aus dem Hugging Face Hub laden.

`TransformersReader`
: Basierend auf der QA-Pipeline aus der 🤗 Transformers-Bibliothek. Nur im Rahmen der Inferenz geeignet.

Obwohl beide Reader die Gewichte eines Modells auf die gleiche Weise verarbeiten, gibt es einige Unterschiede in der Art und Weise, wie die Vorhersagen in Antworten überführt werden:

- In der 🤗 Transformers-Bibliothek normalisiert die QA-Pipeline die Logits für das vorhergesagte Anfangs- und das Endtoken durch Anwendung der Soft-

max-Funktion in Bezug auf die einzelnen Passagen. Das bedeutet, dass es nur sinnvoll ist, die Scores von Antworten zu vergleichen, die aus derselben Passage extrahiert wurden, da sich deren Wahrscheinlichkeiten zu 1 summieren. Ein Antwort-Score von 0,9 aus einer Passage ist beispielsweise nicht unbedingt besser als ein Score von 0,8 in einer anderen. In der FARM-Bibliothek werden die Logits nicht normalisiert, sodass die Antworten zwischen den Passagen leichter verglichen werden können.

- Der `TransformersReader` sagt manchmal dieselbe Antwort zweimal voraus, aber mit unterschiedlichem Score. Dies kann in längeren Kontexten vorkommen, wenn die Antwort in zwei sich überschneidenden Fenstern bzw. Windows liegt. In der FARM-Bibliothek werden diese Duplikate entfernt.

Da wir im weiteren Verlauf des Kapitels noch ein Feintuning des Readers vornehmen werden, greifen wir auf den `FARMReader` zurück. Wie bei der 🤗 Transformers-Bibliothek müssen wir zum Laden des Modells lediglich den sich auf dem Hugging Face Hub befindlichen Checkpoint für das MiniLM-Modell zusammen mit einigen QA-spezifischen Argumenten angeben:

```
from haystack.reader.farm import FARMReader

model_ckpt = "deepset/minilm-uncased-squad2"
max_seq_length, doc_stride = 384, 128
reader = FARMReader(model_name_or_path=model_ckpt, progress_bar=False,
                    max_seq_len=max_seq_length, doc_stride=doc_stride,
                    return_no_answer=True)
```

Es ist auch möglich, ein Modell für das Leseverstehen (engl. Reading Comprehension) direkt mit der 🤗 Transformers-Bibliothek feinzutunen und es dann über die `TransformersReader`-Klasse zu laden, um es für die Inferenz zu nutzen. Wie Sie beim Feintuning konkret vorgehen können, erfahren Sie in dem Tutorial zum Question Answering in der Dokumentation der Bibliothek (*https://oreil.ly/VkhIQ*).

Wie wir es bereits für den Tokenizer kennengelernt haben, verwendet der `FARMReader` ebenfalls die Argumente `max_seq_length` und `doc_stride`, mit denen sich das Verhalten des Sliding Windows steuern lässt. Hier haben wir die gleichen Parameterwerte wie im MiniLM-Forschungsbeitrag verwendet. Nun sind wir so weit, dass wir den Reader anhand unseres einfachen Beispiels von zuvor testen können:

```
print(reader.predict_on_texts(question=question, texts=[context], top_k=1))
```

```
{'query': 'How much music can this hold?', 'no_ans_gap': 12.648084878921509,
'answers': [{'answer': '6000 hours', 'score': 10.69961929321289, 'probability':
0.3988136053085327, 'context': 'An MP3 is about 1 MB/minute, so about 6000 hours
depending on file size.', 'offset_start': 38, 'offset_end': 48,
'offset_start_in_doc': 38, 'offset_end_in_doc': 48, 'document_id':
'e344757014e804eff50faa3ecf1c9c75'}]}
```

Prima, der Reader scheint wie erwartet zu funktionieren. Lassen Sie uns als Nächstes alle unsere Komponenten mithilfe einer der Haystack-Pipelines zusammenführen.

Alle Komponenten in einem

Die Haystack-Bibliothek bietet eine `Pipeline`-Abstraktion, die es uns ermöglicht, Retriever, Reader und andere Komponenten in einem Graphen zu kombinieren, der sich leicht für den jeweiligen Anwendungsfall anpassen lässt. Es gibt auch vordefinierte Pipelines, die denen in der 🤗 Transformers-Bibliothek entsprechen, jedoch speziell auf QA-Systeme zugeschnitten sind. In unserem Fall sind wir daran interessiert, Antworten zu extrahieren – daher verwenden wir die `ExtractiveQA Pipeline`, die ein einzelnes Retriever-Reader-Paar als Argumente entgegennimmt:

```
from haystack.pipeline import ExtractiveQAPipeline

pipe = ExtractiveQAPipeline(reader, es_retriever)
```

Jede `Pipeline` hat eine `run()`-Methode, mit der angegeben werden kann, wie der Abfrageablauf (engl. Query Flow) ausgeführt werden soll. Für die `ExtractiveQA Pipeline` müssen wir nur die `query`, `top_k_retriever` für die Anzahl der abzurufenden Dokumente und `top_k_reader` für die Anzahl der aus diesen Dokumenten zu extrahierenden Antworten übergeben. In unserem Fall müssen wir außerdem noch einen Filter für die Artikel-ID spezifizieren, was wir mit dem Argument `filter` vornehmen können, wie wir es zuvor beim Retriever getan haben. Probieren wir die vollständige Pipeline mit unserer Frage zum Amazon-Fire-Tablet aus, wobei wir uns dieses Mal allerdings die extrahierten Antworten ausgeben lassen:

```
n_answers = 3
preds = pipe.run(query=query, top_k_retriever=3, top_k_reader=n_answers,
                 filters={"item_id": [item_id], "split":["train"]})

print(f"Frage: {preds['query']} \n")
for idx in range(n_answers):
    print(f"Antwort {idx+1}: {preds['answers'][idx]['answer']}")
    print(f"Auszug aus Rezension: ...{preds['answers'][idx]['context']}...")
    print("\n\n")
```

```
Frage: Is it good for reading?

Antwort 1: I mainly use it for book reading
Auszug aus Rezension: ... is my third one.  I never thought I would want a fire
for I mainly use it for book reading.  I decided to try the fire for when I
travel I take my la...


Antwort 2: the larger screen compared to the Kindle makes for easier reading
Auszug aus Rezension: ...ght enough that I can hold it to read, but the larger
screen compared to the Kindle makes for easier reading. I love the color,
something I never thou...


Antwort 3: it is great for reading books when no light is available
Auszug aus Rezension: ...ecoming addicted to hers! Our son LOVES it and it is
great for reading books when no light is available. Amazing sound but I suggest
good headphones t...
```

Großartig, wir haben jetzt wir ein vollständiges (»End-to-End«) QA-System für Produktrezensionen von Amazon! Nicht schlecht für den Anfang! Allerdings kommen nur die zweite und dritte Antwort der eigentlichen Fragestellung recht nahe. Es gibt also durchaus noch Spielraum, unser System weiter zu verbessern. Dafür benötigen wir jedoch Maße, mit denen wir die Leistung des Retrievers und des Readers quantifizieren können. Nehmen wir also die Verbesserung der Pipeline in Angriff!

Verbesserung unserer QA-Pipeline

Obwohl sich ein Großteil der jüngsten Forschung im Bereich des Question Answering auf die Verbesserung von Modellen für das Leseverstehen konzentriert hat, spielt es praktisch keine Rolle, wie gut Ihr Reader ist, wenn der Retriever die relevanten Dokumente gar nicht erst finden kann! Insbesondere der Retriever bestimmt maßgeblich die Leistungsfähigkeit des gesamten QA-Systems – daher ist es wichtig, dass er gute Arbeit leistet. Führen wir daher zunächst einige gängige Maße ein, mit denen wir den Retriever evaluieren können. In diesem Zusammenhang beurteilen wir auch, wie unterschiedlich die Leistung von dünn- und dichtbesetzten Darstellungen ausfällt.

Den Retriever evaluieren

Ein gängiges Maß zur Evaluierung von Retrievern ist der *Recall*, mit dem gemessen wird, welcher Anteil aller abgerufen Dokumente relevant ist. In diesem Zusammenhang bedeutet »relevant« einfach, ob die Antwort in einer Textpassage vorkommt oder nicht. Wenn wir also eine Reihe von Fragen haben und den Recall berechnen möchten, dann müssen wir zählen, wie oft eine Antwort in den ersten k Dokumenten, die der Retriever zurückgibt, enthalten ist.

In der Haystack-Bibliothek gibt es zwei Möglichkeiten, wie Sie einen Retriever bewerten können:

- Verwenden Sie die integrierte `eval()`-Methode des Retrievers. Diese Methode kann sowohl für Open- als auch für Closed-Domain-QA verwendet werden, aber nicht für Datensätze wie SubjQA, bei denen jedes Dokument an ein einzelnes Produkt gekoppelt ist und wir die Dokumente bei sämtlichen Abfragen auf Basis der Produkt-ID filtern müssen.
- Erstellen Sie eine eigene `Pipeline`, die einen Retriever mit der `EvalDocuments`-Klasse kombiniert. Dies ermöglicht Ihnen, benutzerdefinierte Maße und Abfrageabläufe (engl. Query Flows) zu implementieren.

Zusätzlich zum Recall gibt es ein Maß namens *mean Average Precision* (mAP), sozusagen der Mittelwert der durchschnittlichen Precision, bei dem die Retriever belohnt werden, die die richtigen Antworten weiter oben in der Rangliste der Dokumente platzieren können.

Da wir den Recall erst für jedes einzelne Produkt ermitteln und diese Werte dann über alle Produkte verdichten müssen, entscheiden wir uns für den zweiten Ansatz. Jeder Knoten (engl. Node) im `Pipeline`-Graphen stellt eine Klasse dar, die einige Eingaben entgegennimmt und einige Ausgaben über eine `run()`-Methode erzeugt:

```
class PipelineNode:
    def __init__(self):
        self.outgoing_edges = 1

    def run(self, **kwargs):
        ...
        return (outputs, "outgoing_edge_name")
```

Hier entspricht `kwargs` den Ausgaben des vorherigen Knotens im Graphen, die im Rahmen der `run()`-Methode so verarbeitet werden, dass ein Tupel zurückgegeben wird, der die Ausgaben für den nächsten Knoten zusammen mit dem Namen für die ausgehende Kante (engl. Edge) umfasst. Die einzige weitere Anforderung ist die Berücksichtigung eines Attributs (`outgoing_edges`), das die Anzahl der Ausgaben des Knotens angibt (in den meisten Fällen ist `outgoing_edges=1`, es sei denn, Sie haben Verzweigungen in der Pipeline, an denen die Eingaben nach einem bestimmten Kriterium weitergeleitet werden).

In unserem Fall benötigen wir einen Knoten, mit dem wir den Retriever bewerten können. Daher verwenden wir die `EvalDocuments`-Klasse, deren `run()`-Methode festhält, welche Dokumente Antworten enthalten, die mit der Ground Truth übereinstimmen. Mit dieser Klasse können wir dann einen `Pipeline`-Graphen aufbauen, indem wir den Evaluierungsknoten nach dem Knoten hinzufügen, der den Retriever selbst darstellt:

```
from haystack.pipeline import Pipeline
from haystack.eval import EvalDocuments

class EvalRetrieverPipeline:
    def __init__(self, retriever):
        self.retriever = retriever
        self.eval_retriever = EvalDocuments()
        pipe = Pipeline()
        pipe.add_node(component=self.retriever, name="ESRetriever",
                      inputs=["Query"])
        pipe.add_node(component=self.eval_retriever, name="EvalRetriever",
                      inputs=["ESRetriever"])
        self.pipeline = pipe


pipe = EvalRetrieverPipeline(es_retriever)
```

Beachten Sie, dass jeder Knoten einen `name` und eine Liste von `inputs` erhält. In den meisten Fällen hat jeder Knoten eine einzige ausgehende Kante, sodass wir nur den Namen des vorherigen Knotens in `inputs` angeben müssen.

Nachdem wir nun unsere Evaluierungspipeline eingerichtet haben, können wir jetzt einige Abfragen und die entsprechenden Antworten übergeben. Zu diesem Zweck fügen wir die Antworten einem speziellen `label`-Index in unserem Document Store

hinzu. Die Haystack-Bibliothek bietet ein `Label`-Objekt, mit dem die für die Antwort relevanten Textbereiche (Answer Span) und die jeweiligen Metadaten auf standardisierte Weise dargestellt werden. Damit wir den `label`-Index befüllen können, erzeugen wir zunächst eine Liste, die aus den entsprechenden `Label`-Objekten besteht, indem wir mithilfe einer Schleife jede Frage im Testdatensatz durchgehen und die passenden Antworten sowie zusätzliche Metadaten extrahieren:

```
from haystack import Label

labels = []
for i, row in dfs["test"].iterrows():
    # Metadaten, die für die Filterung im Retriever verwendet werden
    meta = {"item_id": row["title"], "question_id": row["id"]}
    # labels-Liste mit Fragen befüllen, die Antwort aufweisen
    if len(row["answers.text"]):
        for answer in row["answers.text"]:
            label = Label(
                question=row["question"], answer=answer, id=i, origin=row["id"],
                meta=meta, is_correct_answer=True, is_correct_document=True,
                no_answer=False)
            labels.append(label)
    # labels-Liste mit Fragen befüllen, die keine Antwort aufweisen
    else:
        label = Label(
            question=row["question"], answer="", id=i, origin=row["id"],
            meta=meta, is_correct_answer=True, is_correct_document=True,
            no_answer=True)
        labels.append(label)
```

Wenn wir einen Blick auf eines dieser `Label`-Objekte werfen:

```
print(labels[0])
```

```
{'id': 'e28f5e62-85e8-41b2-8a34-fbff63b7a466', 'created_at': None, 'updated_at':
None, 'question': 'What is the tonal balance of these headphones?', 'answer': 'I
have been a headphone fanatic for thirty years', 'is_correct_answer': True,
'is_correct_document': True, 'origin': 'd0781d13200014aa25860e44da9d5ea7',
'document_id': None, 'offset_start_in_doc': None, 'no_answer': False,
'model_id': None, 'meta': {'item_id': 'B00001WRSJ', 'question_id':
'd0781d13200014aa25860e44da9d5ea7'}}
```

können wir sehen, dass außer dem entsprechenden Frage-Antwort-Paar noch ein Feld `origin` enthalten ist, in dem jeweils die eindeutige ID der Frage hinterlegt ist, wodurch wir in der Lage sind, den Document Store nach den einzelnen Fragen zu filtern. Zudem haben wir auch die ID des jeweiligen Produkts zum `meta`-Feld hinzugefügt, um die Label-Objekte nach Produkten filtern zu können. Nachdem wir unsere Liste mit `Label`-Objekten erstellt haben, können wir sie wie folgt in den `label`-Index in Elasticsearch schreiben:

```
document_store.write_labels(labels, index="label")
print(f"""{document_store.get_label_count(index="label")} \
Frage-Antwort-Paare geladen""")
```

```
358 Frage-Antwort-Paare geladen
```

Als Nächstes müssen wir eine Zuordnung (engl. Mapping) zwischen den IDs unserer Fragen und den entsprechenden Antworten erstellen, die wir anschließend an die Pipeline übergeben können. Um alle Label-Objekte zu erhalten, können wir die `get_all_labels_aggregated()`-Methode des Document Store verwenden, die alle Frage-Antwort-Paare zusammenführt, die einer eindeutigen ID zugeordnet sind. Diese Methode gibt eine Liste von `MultiLabel`-Objekten zurück, wobei wir in unserem Fall nur ein Element erhalten, da wir nach der ID der Frage filtern. Die Liste mit den zusammengeführten Labels können wir wie folgt erstellen:

```
labels_agg = document_store.get_all_labels_aggregated(
    index="label",
    open_domain=True,
    aggregate_by_meta=["item_id"]
)
print(len(labels_agg))
```

```
330
```

Wenn Sie sich eines dieser Label-Objekte ausgeben lassen, sehen Sie, dass nun alle Antworten, die mit einer bestimmten Frage verbunden sind, in einem Feld namens `multiple_answers` zusammengeführt wurden:

```
print(labels_agg[109])
```

```
{'question': 'How does the fan work?', 'multiple_answers': ['the fan is really
really good', "the fan itself isn't super loud. There is an adjustable dial to
change fan speed"], 'is_correct_answer': True, 'is_correct_document': True,
'origin': '5a9b7616541f700f103d21f8ad41bc4b', 'multiple_document_ids': [None,
None], 'multiple_offset_start_in_docs': [None, None], 'no_answer': False,
'model_id': None, 'meta': {'item_id': 'B002MU1ZRS'}}
```

Jetzt verfügen wir über alle erforderlichen Zutaten, um den Retriever evaluieren zu können. Definieren wir also eine Funktion, die jedes Frage-Antwort-Paar, das mit einem bestimmten Produkt verknüpft ist, in die Evaluierungspipeline einspeist und die Abrufe, bei denen die Antwort zur Frage passend ist, in unserem `pipe`-Objekt erfasst:

```
def run_pipeline(pipeline, top_k_retriever=10, top_k_reader=4):
    for l in labels_agg:
        _ = pipeline.pipeline.run(
            query=l.question,
            top_k_retriever=top_k_retriever,
            top_k_reader=top_k_reader,
            top_k_eval_documents=top_k_retriever,
            labels=l,
            filters={"item_id": [l.meta["item_id"]], "split": ["test"]})

run_pipeline(pipe, top_k_retriever=3)
print(f"Recall@3: {pipe.eval_retriever.recall:.2f}")
```

```
Recall@3: 0.95
```

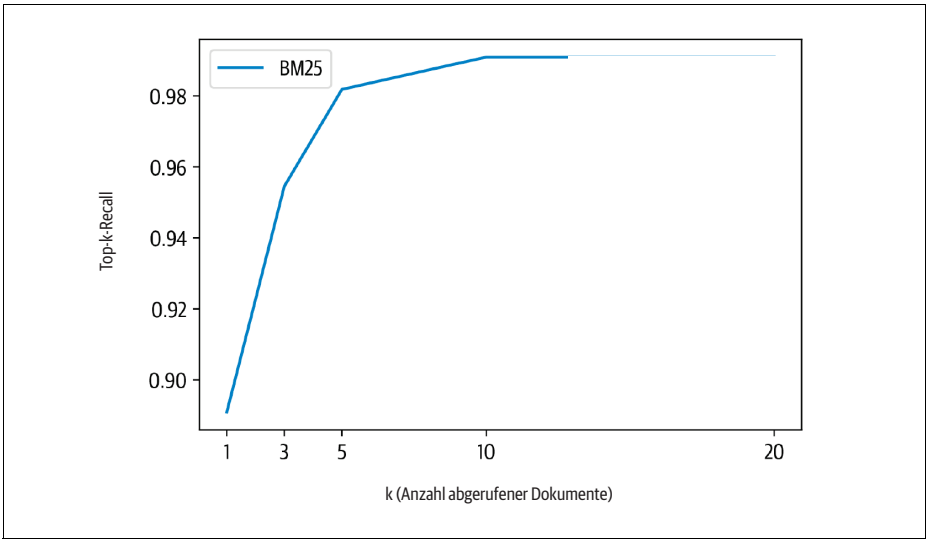
Sehr gut, es klappt! Sie werden feststellen, dass wir einen bestimmten Wert für `top_k_retriever` gewählt haben, mit dem wir die Anzahl der abzurufenden Dokumente festlegen. Im Allgemeinen verbessert eine Erhöhung dieses Parameters den Recall – dies allerdings zu dem Preis, dass dem Reader mehr Dokumente zur Verfügung ge-

stellt werden und sich die End-to-End-Pipeline verlangsamt. Um eine Entscheidungshilfe zu haben, welchen Wert wir wählen sollen, erstellen wir eine Funktion, die im Rahmen einer Schleife mehrere Werte für *k* durchläuft und für jeden Wert von *k* den Recall auf dem gesamten Testdatensatz ermittelt:

```
def evaluate_retriever(retriever, topk_values = [1,3,5,10,20]):
    topk_results = {}

    for topk in topk_values:
        # Pipeline erstellen
        p = EvalRetrieverPipeline(retriever)
        # Schleife über jedes Frage-Antwort-Paar im Testdatensatz
        run_pipeline(p, top_k_retriever=topk)
        # Maße ermitteln
        topk_results[topk] = {"recall": p.eval_retriever.recall}

    return pd.DataFrame.from_dict(topk_results, orient="index")


es_topk_df = evaluate_retriever(es_retriever)
```

Halten wir die Ergebnisse am besten in einem Diagramm fest. Wir können in der Tat feststellen, dass sich der Recall verbessert, wenn wir die Anzahl der abzurufenden Dokumente, *k*, erhöhen:

```
def plot_retriever_eval(dfs, retriever_names):
    fig, ax = plt.subplots()
    for df, retriever_name in zip(dfs, retriever_names):
        df.plot(y="recall", ax=ax, label=retriever_name)
    plt.xticks(df.index)
    plt.ylabel("Top-k-Recall")
    plt.xlabel("k (Anzahl abgerufener Dokumente)")
    plt.show()

plot_retriever_eval([es_topk_df], ["BM25"])
```



In dem Diagramm können wir erkennen, dass es einen Wendepunkt bei $k = 5$ gibt und dass wir ab $k = 10$ einen nahezu idealen Wert für den Recall erhalten. Sehen wir uns nun an, wie wir Dokumente mit Methoden abrufen können, die auf dichtbesetzten Vektoren basieren.

Dense Passage Retrieval

Wie wir herausgefunden haben, würden wir einen nahezu idealen Wert für den Recall erzielen, wenn unser Sparse-Retriever $k = 10$ Dokumente abruft bzw. zurückgibt. Doch könnte es nicht auch einen Anreiz geben, kleinere Werte für k zu wählen? In der Tat wäre es von Vorteil, wenn weniger Dokumente an den Reader weitergeben werden könnten und sich dadurch die Gesamtlatenz unserer QA-Pipeline verringern würde. Zudem sind Sparse-Retriever wie der BM25 bekannt dafür, dass sie die relevanten Dokumente nicht erfassen können, wenn die Benutzeranfrage Begriffe enthält, die nicht exakt mit denen der Rezension übereinstimmen. Eine vielversprechende Alternative ist die Verwendung von dichtbesetzten Einbettungen, sogenannten Dense Embeddings, mit denen die Frage und das Dokument abgebildet werden. Derzeit erzielt eine Architektur State-of-the-Art-Ergebnisse, die als *Dense Passage Retrieval* (DPR) bekannt ist.[14] Die Hauptidee hinter DPR besteht darin, zwei BERT-Modelle zu verwenden – eins als Encoder für die Frage und eins als Encoder für die Textpassage. Wie in Abbildung 7-10 dargestellt, bilden diese Encoder den Eingabetext auf eine d-dimensionale Vektordarstellung des `[CLS]`-Tokens ab.

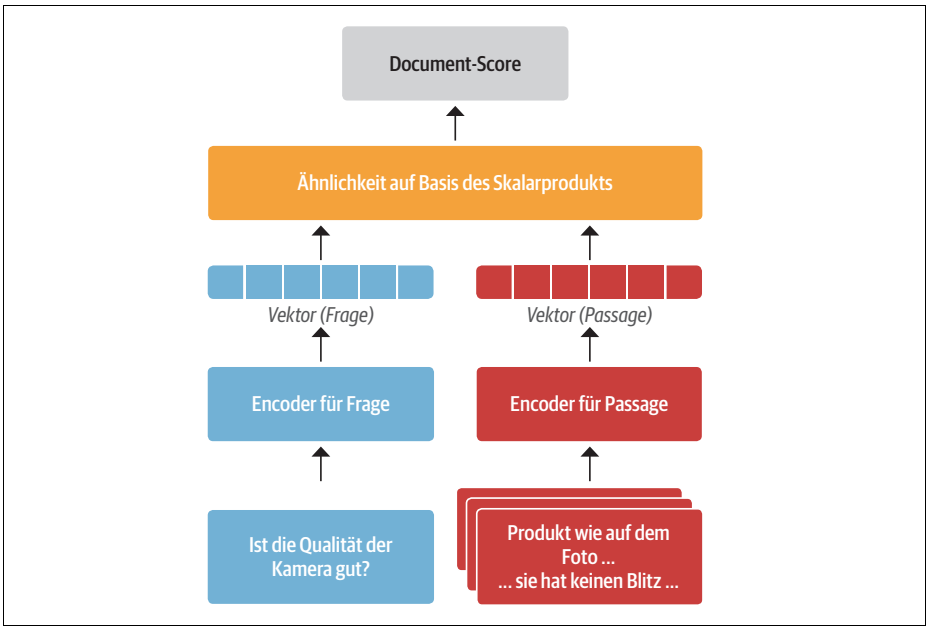


Abbildung 7-10: Die Bi-Encoder-Architektur von DPR zur Berechnung der Relevanz eines Dokuments auf Basis einer Abfrage

14 V. Karpukhin et al., »Dense Passage Retrieval for Open-Domain Question Answering« (*https://arxiv.org/abs/2004.04906*), (2020).

In der Haystack-Bibliothek können wir einen Retriever für das DPR auf ähnliche Weise initialisieren, wie wir es beim BM25 getan haben. Zusätzlich zur Angabe des Document Store müssen wir auch die BERT-basierten Encoder für die Frage und die Textpassage festlegen. Diese Encoder werden trainiert. Hierfür werden ihnen Fragen mit relevanten (positiven) Passagen und irrelevanten (negativen) Passagen vorgegeben, wobei das Ziel darin besteht, dass sie lernen, dass relevante Frage-Passage-Paare eine höhere Ähnlichkeit (engl. Similarity) aufweisen. Für unseren Anwendungsfall werden wir Encoder verwenden, die auf die gleiche Weise auf dem NQ-Korpus feingetunt wurden:

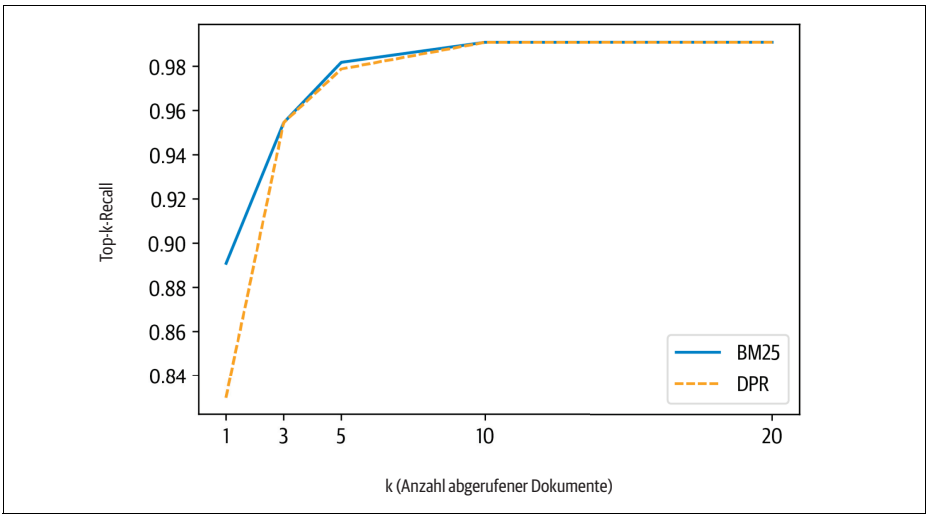
```
from haystack.retriever.dense import DensePassageRetriever

dpr_retriever = DensePassageRetriever(document_store=document_store,
    query_embedding_model="facebook/dpr-question_encoder-single-nq-base",
    passage_embedding_model="facebook/dpr-ctx_encoder-single-nq-base",
    embed_title=False)
```

Hier haben wir auch `embed_title=False` gesetzt, da die Berücksichtigung des Dokumententitels (d.h. `item_id`) keine zusätzlichen Informationen liefert, da wir nach einzelnen Produkten filtern. Sobald wir den Dense-Retriever initialisiert haben, besteht der nächste Schritt darin, alle indexierten Dokumente in unserem Elasticsearch-Index zu durchlaufen und die Encoder zu verwenden, um die (Darstellung als) Embeddings zu aktualisieren. Dies kann folgendermaßen vorgenommen werden:

```
document_store.update_embeddings(retriever=dpr_retriever)
```

Nun sind wir so weit, dass wir die Evaluierung vornehmen können! Den Dense-Retriever können wir auf die gleiche Weise wie den BM25 evaluieren und den Recall für unterschiedliche Werte von *k* vergleichen:

```
dpr_topk_df = evaluate_retriever(dpr_retriever)
plot_retriever_eval([es_topk_df, dpr_topk_df], ["BM25", "DPR"])
```



Wie wir feststellen können, bietet der DPR hinsichtlich des Recalls keinen Vorteil gegenüber dem BM25, und der Wendepunkt liegt bei $k = 3$.

Die Ähnlichkeitssuche auf Basis von Embeddings kann durch die Verwendung von Facebooks FAISS-Bibliothek (*https://oreil.ly/1E8Z0*) als Document Store beschleunigt werden. Ebenso lässt sich die Leistung von Dense-Passage-Retrievern (DPR) verbessern, indem sie auf der Zieldomäne feingetunt werden. Wenn Sie erfahren möchten, wie sich DPR feintunen lassen, dann sehen Sie sich am besten das Tutorial (*https://oreil.ly/eXyro*) von Haystack an.

Nachdem wir uns nun mit der Evaluierung des Retrievers beschäftigt haben, können wir uns jetzt der Evaluierung des Readers zuwenden.

Den Reader evaluieren

Beim extraktiven QA gibt es zwei wesentliche Maße, die zur Evaluierung von Readern verwendet werden:

Exact Match (EM)
: Ein binäres Maß, wobei EM = 1 ist, wenn die Zeichen in der vorhergesagten und der tatsächlichen Antwort exakt übereinstimmen, und ansonsten EM = 0. Sollte keine Antwort erwartet werden, erhält das Modell EM = 0, sofern es überhaupt einen Text vorhersagt.

F_1-Maß
: Entspricht dem harmonischen Mittel aus Precision und Recall.

Um uns mit der Funktionsweise dieser Maße vertraut zu machen, können wir einige Hilfsfunktionen aus der FARM-Bibliothek importieren und sie auf ein einfaches Beispiel anwenden:

```
from farm.evaluation.squad_evaluation import compute_f1, compute_exact

pred = "about 6000 hours"
label = "6000 hours"
print(f"EM: {compute_exact(label, pred)}")
print(f"F1: {compute_f1(label, pred)}")
```

```
EM: 0
F1: 0.8
```

Zunächst normalisieren diese Funktionen die Vorhersage und das Label, indem Satzzeichen entfernt, Leerzeichen korrigiert und die Zeichen in Kleinbuchstaben konvertiert werden. Die normalisierten Strings werden dann in Form von Bag-of-Words tokenisiert, und anschließend das Maß auf der Ebene von Tokens ermittelt. Anhand dieses einfachen Beispiels können wir sehen, dass das EM-Maß wesentlich strenger verfährt als das F_1-Maß: Wenn Sie der Vorhersage nur ein einziges Token hinzufügen, nimmt das EM-Maß einen Wert von null an. Andererseits kann es vorkommen, dass der Wert des F_1-Maßes nicht darauf schließen lässt, das eine Antwort in Wirklichkeit völlig falsch ist. Wenn unser vorhergesagter für die Antwort

relevanter Textabschnitt (Answer Span) zum Beispiel »about 6000 dollars« lautet, dann erhalten wir als Werte:

```
pred = "about 6000 dollars"
print(f"EM: {compute_exact(label, pred)}")
print(f"F1: {compute_f1(label, pred)}")
```

```
EM: 0
F1: 0.4
```

Offensichtlich können Sie durchaus zu falschen Schlüssen gelangen, wenn Sie nur das F_1-Maß zur Evaluierung heranziehen. Daher ist es sinnvoll, beide Maße zu berücksichtigen und so zu verhindern, dass Sie die Modellleistung tendenziell entweder eher unter- (EM) oder überschätzen (F_1-Maß).

Da es im Allgemeinen mehrere gültige Antworten pro Frage gibt, werden die Maße für jedes Frage-Antwort-Paar des zur Evaluierung herangezogenen Datensatzes (bzw. Validierungsdatensatzes) berechnet und jeweils der beste Wert unter allen möglichen Antworten ausgewählt. Die Gesamtwerte für das EM- und das F_1-Maß für das Modell ergeben sich dann aus dem Durchschnitt der einzelnen Werte jedes Frage-Antwort-Paars.

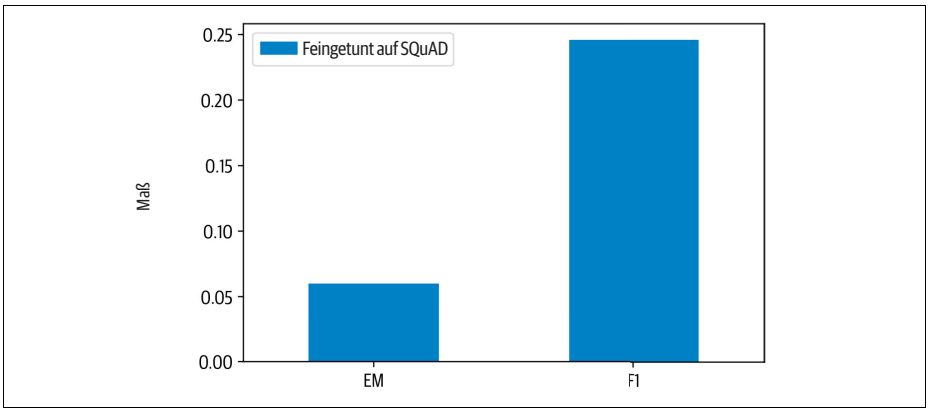
Nehmen wir nun die Evaluierung des Readers vor und erstellen wir eine neue Pipeline mit zwei Knoten (engl. Nodes), wobei ein Knoten dem Reader und der andere Knoten der Evaluierung des Readers dient. Wir greifen hierfür auf die `EvalAnswers`-Klasse zurück, die die Vorhersagen des Readers entgegennimmt und die entsprechenden Werte für das EM- und das F_1-Maß berechnet. Um unsere Ergebnisse mit denen, die wir bei der Evaluierung auf Basis des SQuAD-Datensatzes erzielt haben, vergleichen zu können, ermitteln wir die besten Antworten für jede Abfrage auf Basis des `top_1_em`- und des `top_1_f1`-Maßes, die in dem `EvalAnswers`-Objekt gespeichert sind:

```
from haystack.eval import EvalAnswers

def evaluate_reader(reader):
    score_keys = ['top_1_em', 'top_1_f1']
    eval_reader = EvalAnswers(skip_incorrect_retrieval=False)
    pipe = Pipeline()
    pipe.add_node(component=reader, name="QAReader", inputs=["Query"])
    pipe.add_node(component=eval_reader, name="EvalReader", inputs=["QAReader"])

    for l in labels_agg:
        doc = document_store.query(l.question,
                                   filters={"question_id":[l.origin]})
        _ = pipe.run(query=l.question, documents=doc, labels=l)

    return {k:v for k,v in eval_reader.__dict__.items() if k in score_keys}

reader_eval = {}
reader_eval["Feingetunt auf SQuAD"] = evaluate_reader(reader)
```

Beachten Sie, dass wir `skip_incorrect_retrieval=False` gesetzt haben. Damit stellen wir sicher, dass der Retriever den Kontext immer an den Reader weitergibt (wie bei der Evaluierung auf Basis des SQuAD-Datensatzes). Nachdem wir alle Fragen

durch den Reader laufen lassen haben, können wir unsere Ergebnisse in einem Diagramm darstellen:

```
def plot_reader_eval(reader_eval):
    fig, ax = plt.subplots()
    df = pd.DataFrame.from_dict(reader_eval)
    df.plot(kind="bar", ylabel="Maß", rot=0, ax=ax)
    ax.set_xticklabels(["EM", "F1"])
    plt.legend(loc='upper left')
    plt.show()

plot_reader_eval(reader_eval)
```



Nun ja, es scheint, dass das feingetunte Modell in Bezug auf den SubjQA-Datensatz deutlich schlechter abschneidet als in Bezug auf den SQuAD-2.0-Datensatz, auf dem das MiniLM-Modell Werte für das EM- und das F_1-Maß von 76,1 bzw. 79,5 erreicht. Einer der Gründe, warum die Leistung bzw. Qualität geringer ausfällt, ist, dass sich Kundenrezensionen deutlich von Wikipedia-Artikeln unterscheiden, auf denen der SQuAD-2.0-Datensatz basiert, und dass die verwendete Sprache oft nicht so förmlich ist. Darüber hinaus dürfte ein weiterer Faktor die unserem Datensatz zugrunde liegende inhärente Subjektivität sein, da sich sowohl die Fragen als auch die Antworten von den Inhalten auf Wikipedia unterscheiden, die von Fakten geprägt sind. Werfen wir nun einen Blick darauf, wie ein Modell auf einem Datensatz feingetunt werden kann, um so mithilfe der sogenannten Domain Adaptation bessere Ergebnisse zu erzielen.

Domain Adaptation

Obwohl Modelle, die auf den SQuAD-Datensatz feingetunt wurden, oft gut auf andere Domänen verallgemeinern bzw. übertragen werden können, haben wir gesehen, dass unser Modell sowohl hinsichtlich des EM- als auch des F_1-Maßes auf dem SubjQA-Datensatz wesentlich schlechter abschneidet als auf dem SQuAD-Datensatz. Diese mangelnde Generalisierungsfähigkeit wurde auch bei anderen extraktiven QA-Datensätzen beobachtet und wird als Beweis dafür verstanden, dass Transformer-Modelle hinsichtlich des SQuAD-Datensatzes übermäßig stark an die

Daten angepasst sind (Overfitting).[15] Die einfachste Möglichkeit, den Reader zu verbessern, besteht darin, unser MiniLM-Modell zusätzlich mit dem SubjQA-Trainingsdatensatz feinzutunen. Der `FARMReader` verfügt über eine für diesen Zweck entwickelte `train()`-Methode und erwartet die Daten in dem für den SQuAD-Datensatz vorgesehenen JSON-Format, bei dem alle Frage-Antwort-Paare im Hinblick auf die einzelnen Produkte gruppiert sind (siehe Abbildung 7-11).

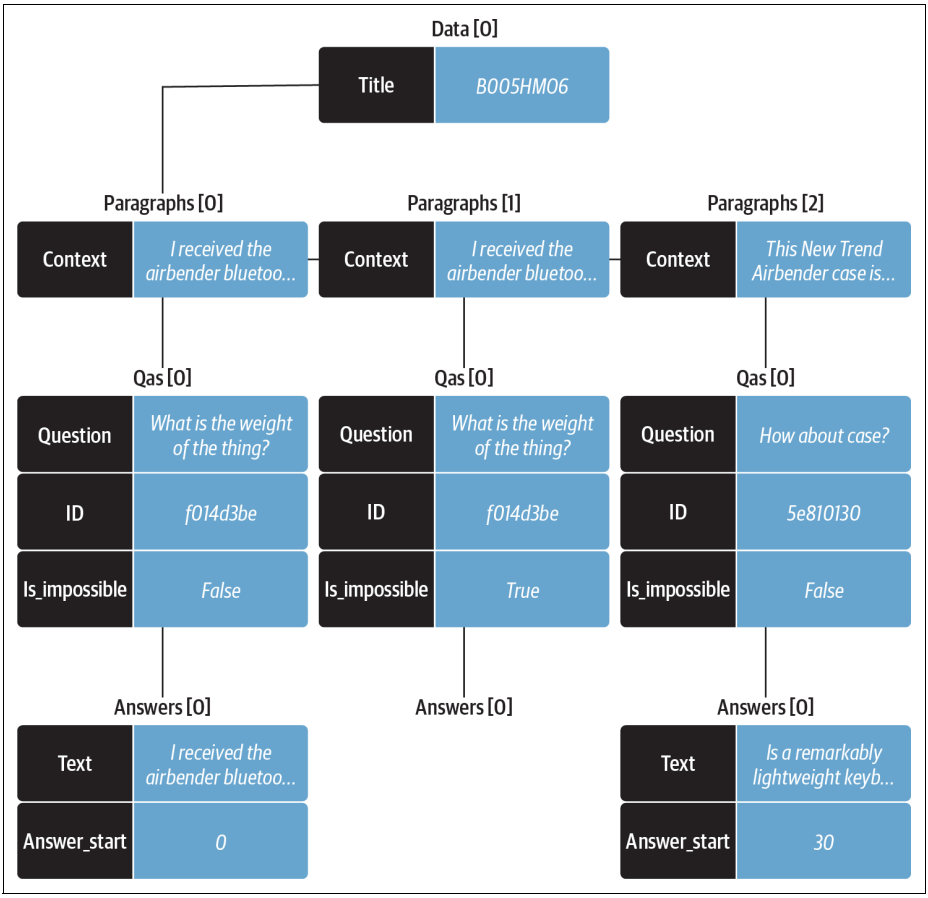


Abbildung 7-11: Das dem SQuAD-Datensatz zugrunde liegende JSON-Format veranschaulicht

Es handelt sich hierbei um ein recht komplexes Datenformat. Daher benötigen wir einige Funktionen und etwas Pandas-Zauberei, die uns dabei helfen, die Daten in dieses Format zu bringen. Als Erstes müssen wir eine Funktion implementieren, die für jede Produkt-ID ein Array, `paragraphs`, erstellen kann. Jeder Eintrag in diesem Array umfasst einen einzelnen Kontext (d.h. eine Rezension) und ein `qas`-Array, das die entsprechenden Frage-Antwort-Paare enthält. Das `paragraphs`-Array können Sie mit der folgenden Funktion erstellen:

15 D. Yogatama et al., »Learning and Evaluating General Linguistic Intelligence« (*https://arXiv.org/abs/1901.11373*), (2019).

```
def create_paragraphs(df):
    paragraphs = []
    id2context = dict(zip(df["review_id"], df["context"]))
    for review_id, review in id2context.items():
        qas = []
        # Herausfiltern aller Frage-Antwort-Paare für einen bestimmten Kontext
        review_df = df.query(f"review_id == '{review_id}'")
        id2question = dict(zip(review_df["id"], review_df["question"]))
        # qas-Array erstellen
        for qid, question in id2question.items():
            # Entsprechend der ID einer Frage filtern
            question_df = df.query(f"id == '{qid}'").to_dict(orient="list")
            ans_start_idxs = question_df["answers.answer_start"][0].tolist()
            ans_text = question_df["answers.text"][0].tolist()
            # Beantwortbare Fragen in Array sammeln
            if len(ans_start_idxs):
                answers = [
                    {"text": text, "answer_start": answer_start}
                    for text, answer_start in zip(ans_text, ans_start_idxs)]
                is_impossible = False
            else:
                answers = []
                is_impossible = True
            # Frage-Antwort-Paare zum qas-Array hinzufügen
            qas.append({"question": question, "id": qid,
                        "is_impossible": is_impossible, "answers": answers})
        # Kontext und Frage-Antwort-Paare zum paragraphs-Array hinzufügen
        paragraphs.append({"qas": qas, "context": review})
    return paragraphs
```

Wenn wir diese Funktion nun auf die Zeilen eines `DataFrame`, die jeweils einer einzelnen Produkt-ID entsprechen, anwenden, erhalten wir die Daten im SQuAD-Format:

```
product = dfs["train"].query("title == 'B00001P4ZH'")
create_paragraphs(product)
```

```
[{'qas': [{'question': 'How is the bass?',
    'id': '2543d296da9766d8d17d040ecc781699',
    'is_impossible': True,
    'answers': []}],
  'context': 'I have had Koss headphones ...',
    'id': 'd476830bf9282e2b9033e2bb44bbb995',
    'is_impossible': False,
    'answers': [{'text': 'Bass is weak as expected', 'answer_start': 1302},
     {'text': 'Bass is weak as expected, even with EQ adjusted up',
      'answer_start': 1302}]}],
  'context': 'To anyone who hasn\'t tried all ...'},
 {'qas': [{'question': 'How is the bass?',
    'id': '455575557886d6dfeea5aa19577e5de4',
    'is_impossible': False,
    'answers': [{'text': 'The only fault in the sound is the bass',
      'answer_start': 650}]}],
  'context': "I have had many sub-$100 headphones ..."}]
```

Der letzte Schritt besteht darin, die Funktion auf alle Produkt-IDs, die in dem als `DataFrame` vorliegenden Trainings- und Validierungs- und Testdatensatz (engl. Splits)

enthalten sind, anzuwenden. Die folgende Funktion, `convert_to_squad()`, erledigt dies für uns und speichert das Ergebnis in einer Datei namens *electronics-{split}.json*:

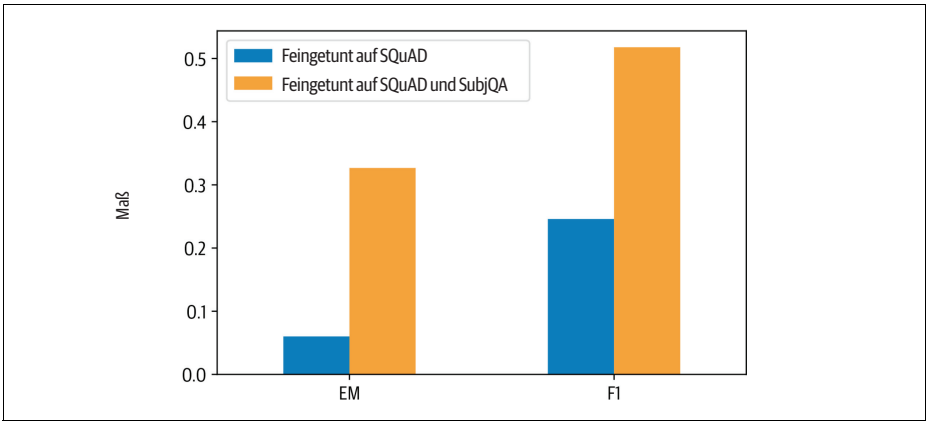
```
import json

def convert_to_squad(dfs):
    for split, df in dfs.items():
        subjqa_data = {}
        # Jeweils paragraphs-Array für alle Produkt-IDs erstellen
        groups = (df.groupby("title").apply(create_paragraphs)
            .to_frame(name="paragraphs").reset_index())
        subjqa_data["data"] = groups.to_dict(orient="records")
        # Ergebnis auf Festplatte speichern
        with open(f"electronics-{split}.json", "w+", encoding="utf-8") as f:
            json.dump(subjqa_data, f)

convert_to_squad(dfs)
```

Nachdem wir nun die Teildatensätze (bzw. Splits) im richtigen Format haben, können wir unseren Reader feintunen. Hierfür müssen wir angeben, wo sich der Trainings- und der Validierungsdatensatz befinden, sowie den Pfad mitteilen, wo das feingetunte Modell gespeichert werden soll:

```
train_filename = "electronics-train.json"
dev_filename = "electronics-validation.json"

reader.train(data_dir=".", use_gpu=True, n_epochs=1, batch_size=16,
             train_filename=train_filename, dev_filename=dev_filename)
```

Sobald das Feintuning für den Reader abgeschlossen ist, können wir ermitteln, wie gut er im Vergleich zu unserem Baseline-Modell auf dem Testdatensatz abschneidet:

```
reader_eval["Feingetunt auf SQuAD und SubjQA"] = evaluate_reader(reader)
plot_reader_eval(reader_eval)
```



Wow, die Domain Adaptation hat dazu geführt, dass sich das EM-Maß um das Sechsfache erhöht und sich das F_1-Maß mehr als verdoppelt hat! An dieser Stelle fragen Sie sich vielleicht, warum wir das vortrainierte Modell nicht gleich

direkt mit dem SubjQA-Trainingssatz feingetunt haben. Ein Grund dafür ist, dass wir im SubjQA-Datensatz nur 1.295 Trainingsbeispiele vorliegen haben, während der SQuAD-Datensatz über 100.000 umfasst. Dementsprechend könnten wir Schwierigkeiten mit einer Überanpassung (engl. Overfitting) bekommen. Gehen wir dennoch der Frage nach, was uns dieser naive Feintuning-Ansatz bringen würde. Um den Vergleich fair zu gestalten, verwenden wir dasselbe Sprachmodell, das auch zum Feintuning unseres Baseline-Modells auf der Grundlage des SQuAD-Datensatzes verwendet wurde. Das Modell können wir wie zuvor mit dem `FARMReader` laden:

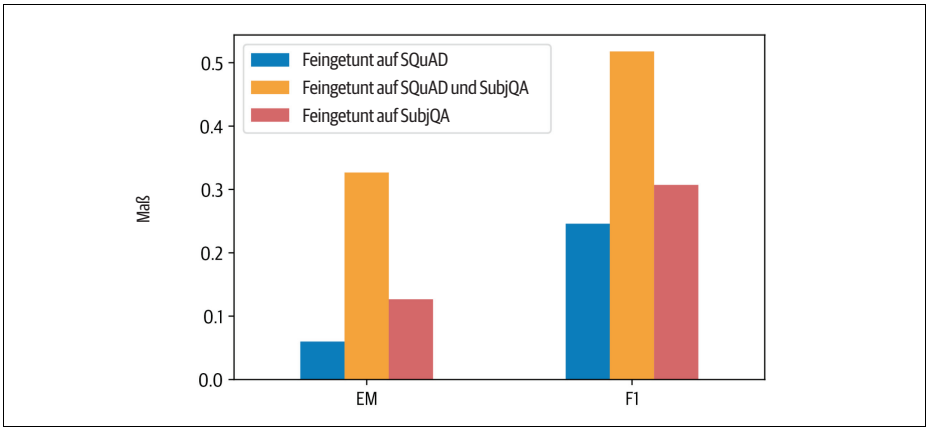
```
minilm_ckpt = "microsoft/MiniLM-L12-H384-uncased"
minilm_reader = FARMReader(model_name_or_path=minilm_ckpt, progress_bar=False,
                           max_seq_len=max_seq_length, doc_stride=doc_stride,
                           return_no_answer=True)
```

Als Nächstes nehmen wir das Feintuning für eine Epoche vor:

```
minilm_reader.train(data_dir=".", use_gpu=True, n_epochs=1, batch_size=16,
             train_filename=train_filename, dev_filename=dev_filename)
```

und evaluieren das Modell auf Basis des Validierungsdatensatzes:

```
reader_eval["Feingetunt mit SubjQA"] = evaluate_reader(minilm_reader)
plot_reader_eval(reader_eval)
```



Wie wir feststellen, schneidet das Sprachmodell bedeutend schlechter ab, wenn wir es gleich direkt mit dem SubjQA-Datensatz feintunen – und nicht erst den SQuAD- und dann den SubjQA-Datensatz für das Feintuning verwenden.

Wenn Sie mit kleinen Datensätzen arbeiten, empfiehlt es sich, im Rahmen der Evaluierung von Transformer-Modellen eine Kreuzvalidierung (engl. Cross-Validation) vorzunehmen, da es zu einer Überanpassung bzw. einem Overfitting kommen könnte. Im FARM-Repository (*https://oreil.ly/K3nK8*) finden Sie ein Beispiel dafür, wie Sie eine Kreuzvalidierung mit SQuAD-formatierten Datensätzen durchführen können.

Die gesamte QA-Pipeline evaluieren

Nachdem wir nun eruiert haben, wie der Reader und der Retriever jeweils einzeln evaluiert werden können, werden wir sie nun miteinander verknüpfen, um die Gesamtleistung unserer Pipeline zu ermitteln. Dazu müssen wir unsere Retriever-Pipeline jeweils um einen zusätzlichen Knoten für den Reader und für die Evaluierung des Readers erweitern. Wie wir festgestellt haben, erhalten wir für $k = 10$ einen nahezu optimalen Wert für den Recall. Dementsprechend liegt es nahe, dass wir zunächst diesen Wert für k wählen und evaluieren, wie sich unsere Wahl von k auf die Leistung des Readers auswirkt (da er nun im Vergleich zur Evaluierung auf Basis des SQuAD-Datensatzes mehrere Kontexte pro Abfrage erhält):

```
# Retriever-Pipeline initialisieren
pipe = EvalRetrieverPipeline(es_retriever)
# Die Knoten für Reader hinzufügen
eval_reader = EvalAnswers()
pipe.pipeline.add_node(component=reader, name="QAReader",
              inputs=["EvalRetriever"])
pipe.pipeline.add_node(component=eval_reader, name="EvalReader",
              inputs=["QAReader"])
# Evaluieren!
run_pipeline(pipe)
# Maße aus Reader extrahieren
reader_eval["QA-Pipeline (Top-1)"] = {
    k:v for k,v in eval_reader.__dict__.items()
    if k in ["top_1_em", "top_1_f1"]}
```

Anschließend können wir die Top-1-Werte für das EM- und das F_1-Maß für das Modell hinsichtlich der Vorhersage einer Antwort, die in den vom Retriever zurückgegebenen Dokumenten enthalten ist, vergleichen (siehe Abbildung 7-12).

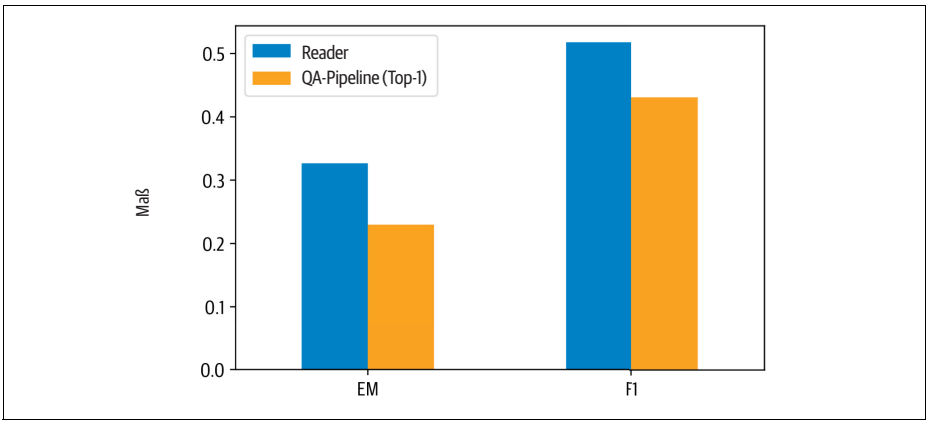


Abbildung 7-12: Werte des EM- und des F_1-Maßes für den Reader im Vergleich zur gesamten QA-Pipeline

Wie aus dem Diagramm hervorgeht, beeinflusst der Retriever die Gesamtleistung. Verglichen mit dem Abgleich der Frage-Kontext-Paare, wie er im Rahmen der Evaluierung auf Basis des SQuAD-Datensatzes durchgeführt wird, ist insgesamt eine

Verschlechterung der Leistung festzustellen. Diesem Umstand können Sie begegnen, indem Sie die Anzahl der möglichen Antworten, die der Reader vorhersagen darf, erhöhen.

Bisher haben wir nur einzelne für die Antwort relevante Textabschnitte (Answer Span) aus dem jeweiligen Kontext extrahiert. Es ist jedoch auch möglich, dass die einzelnen Teile einer Antwort über das gesamte Dokument verteilt sind. In diesem Fall möchten wir, dass unser Modell diese einzelnen Fragmente zu einer einzigen kohärenten Antwort zusammenführt. Sehen wir uns nun an, wie uns das generative QA dabei helfen kann, diese Aufgabe zu bewältigen.

Jenseits des extraktiven QA

Eine interessante Alternative zur Extraktion von Antworten aus einem Dokument in Form von Textabschnitten besteht darin, sie mit einem vortrainierten Modell zu generieren. Dieser Ansatz wird oft als *abstraktives* (engl. abstractive) oder *generatives Question Answering* bezeichnet und eröffnet die Möglichkeit, die Antworten besser zu formulieren, indem sie die entsprechenden Fakten über mehrere Passagen hinweg zusammenfassen. Auch wenn dieser Ansatz noch nicht so ausgereift ist wie das extraktive QA, schreitet die Forschung zügig voran, sodass die Wahrscheinlichkeit groß ist, dass diese Ansätze bereits in der Industrie zum Einsatz kommen, wenn Sie dies lesen! In diesem Abschnitt werden wir kurz auf den aktuellen Stand der Forschung eingehen und uns der *Retrieval-Augmented Generation* (RAG) widmen.[16]

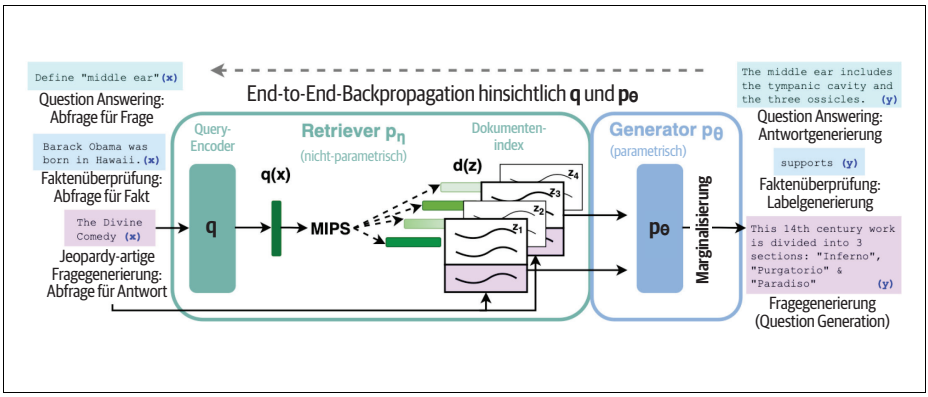


Abbildung 7-13: Die RAG-Architektur, mit der der Retriever und der Generator vollständig (»End-to-End«) feingetunt werden können (mit freundlicher Genehmigung von Ethan Perez)

Bei der RAG wird die klassische Retriever-Reader-Architektur, die wir in diesem Kapitel kennengelernt haben, erweitert, indem der Reader durch einen *Generator* ersetzt und ein DPR als Retriever verwendet wird. Der Generator entspricht einem

16 P. Lewis et al., »Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks« (*https://arxiv.org/abs/2005.11401*), (2020).

vortrainierten Sequence-to-Sequence-Transformer wie T5 oder BART, der latente Vektoren, die die Dokumente abbilden, von dem DPR erhält und dann auf iterative Weise eine Antwort auf der Grundlage der Anfrage und dieser Dokumente generiert. Da der DPR und der Generator differenzierbar sind, kann die gesamte Architektur durchgängig (»End-to-End«) feingetunt werden (siehe Abbildung 7-13).

Sehen wir uns die RAG in Aktion an. Hierzu verwenden wir den bereits zuvor genutzten `dpr_retriever`, weshalb wir nur einen Generator instanziieren müssen. Es gibt zwei Arten von RAG-Modellen, zwischen denen Sie wählen können:

RAG-Sequence-Modelle
: Verwenden stets dasselbe abgerufene Dokument, um die vollständige Antwort zu generieren. Dabei werden die *k* relevantesten Dokumente, die der Retriever ermittelt hat, in den Generator eingespeist. Dieser generiert für jedes Dokument eine Ausgabesequenz, wobei das Ergebnis anschließend in Wahrscheinlichkeiten überführt wird (engl. marginalized), auf Basis derer die beste Antwort ausgewählt wird.

RAG-Token-Modelle
: Können zur Generierung der einzelnen Tokens der Antwort auch ein anderes Dokument verwenden. Auf diese Weise kann der Generator die Fakten aus mehreren Dokumenten zusammenführen.

Da RAG-Token-Modelle tendenziell besser abschneiden als RAG-Sequence-Modelle, verwenden wir ein mit dem NQ-Datensatz feingetuntes Token-Modell als Generator. In der Haystack-Bibliothek können Sie einen Generator ähnlich wie einen Reader instanziieren. Anstatt jedoch die Parameter `max_seq_length` und `doc_stride` festzulegen, die ein Sliding Window über die Kontexte definieren, geben wir die Hyperparameter an, mit denen die Generierung des Texts gesteuert werden kann:

```
from haystack.generator.transformers import RAGenerator

generator = RAGenerator(model_name_or_path="facebook/rag-token-nq",
                        embed_title=False, num_beams=5)
```

Hier gibt `num_beams` die Anzahl der Beams an, die bei der Beam-Search verwendet werden sollen (die Textgenerierung wird ausführlich in Kapitel 5 behandelt). Wie bereits beim DPR-Retriever beziehen wir die Dokumententitel nicht ein, wenn wir die Embeddings erstellen, da wir unser Korpus immer nach der ID des Produkts filtern.

Als Nächstes müssen wir den Retriever und den Generator mithilfe der `GenerativeQAPipeline` von Haystack miteinander verknüpfen:

```
from haystack.pipeline import GenerativeQAPipeline

pipe = GenerativeQAPipeline(generator=generator, retriever=dpr_retriever)
```

Bei der RAG werden sowohl der Encoder für die Abfragen (Query-Encoder) als auch der Generator vollständig (»End-to-End«) trainiert, während der Encoder für die Kontexte (Context-Encoder) nicht trainiert bzw. seine Gewichtung eingefroren (engl. freeze) wird. In der Haystack-Bibliothek wird für die `GenerativeQAPipeline` die `RAGenerator`-Klasse für den Query-Encoder und die `DensePassageRetriever`-Klasse für den Context-Encoder verwendet.

Probieren wir die RAG nun aus, indem wir unsere Abfrage zum Amazon-Fire-Tablet von zuvor einspeisen. Um den Abfragevorgang zu vereinfachen, definieren wir eine einfache Funktion, die die Abfrage entgegennimmt und die besten bzw. relevantesten Antworten ausgibt:

```
def generate_answers(query, top_k_generator=3):
    preds = pipe.run(query=query, top_k_generator=top_k_generator,
                     top_k_retriever=5, filters={"item_id":["B0074BW614"]})
    print(f"Frage: {preds['query']} \n")
    for idx in range(top_k_generator):
        print(f"Antwort {idx+1}: {preds['answers'][idx]['answer']}")
```

Gut, wenden wir die Funktion gleich an:

```
generate_answers(query)
```

```
Frage: Is it good for reading?

Antwort 1:  the screen is absolutely beautiful
Antwort 2:  the Screen is absolutely beautiful
Antwort 3:  Kindle fire
```

Die Antworten sind gar nicht so schlecht. Allerdings deuten sie darauf hin, dass der Generator durch den subjektiven Charakter der Frage in Schwierigkeiten gerät. Versuchen wir es erneut mit einer etwas sachlicheren Frage:

```
generate_answers("What is the main drawback?")
```

```
Frage: What is the main drawback?

Antwort 1:  the price
Antwort 2:  no flash support
Antwort 3:  the cost
```

Diese Antworten ergeben deutlich mehr Sinn! Um noch bessere Ergebnisse zu erzielen, könnten wir die RAG vollständig (»End-to-End«) mit dem SubjQA-Datensatz feintunen. Wir belassen dies als Übung, aber wenn Sie daran interessiert sind, finden Sie im 🤗 Repository der Transformers-Bibliothek (*https://oreil.ly/oZz4S*) einige Skripte, die Ihnen den Einstieg erleichtern.

Zusammenfassung

Das war eine rasante Tour durch das Gebiet des Question Answering, und Sie haben wahrscheinlich noch etliche Fragen, die Sie gerne beantwortet haben möchten (Wortspiel beabsichtigt!). In diesem Kapitel haben wir zwei QA-Ansätze (extraktiv

und generativ) erörtert und zwei verschiedene Algorithmen für den Retrieval (BM25 und DPR) untersucht. Dabei haben wir gesehen, dass die Domain Adaptation ein einfacher Ansatz sein kann, um die Leistung unseres QA-Systems erheblich zu steigern. Zudem haben wir uns einige der gebräuchlichsten Maße angesehen, die zur Evaluierung solcher Systeme verwendet werden. Obwohl wir uns auf das QA innerhalb einer geschlossenen Domäne (d.h. eine einzelne Domäne im Bereich elektronischer Produkte) konzentriert haben, können die Techniken in diesem Kapitel leicht auf den Fall einer offenen Domäne (Open-Domain-QA) verallgemeinert werden. Wenn Sie mehr darüber erfahren möchten, empfehlen wir Ihnen Clouderas exzellente Fast Forward QA Series (*https://oreil.ly/Fd6lc*).

Das Deployment von QA-Systemen kann sich in der Praxis als kompliziert erweisen. Unserer Erfahrung nach liegt ein wesentlicher Teil des Nutzens darin, den Endnutzern zunächst nützliche Suchfunktionen zur Verfügung zu stellen, gefolgt von einer extraktiven Komponente. In diesem Sinne kann der Reader auf neuartige Weise genutzt werden, die je nach Bedarf über die Beantwortung von Benutzeranfragen hinausgeht. Beispielsweise konnten Forscher von Grid Dynamics (*https://oreil.ly/CGLh1*) ihren Reader dazu verwenden, automatisch eine Reihe von Vor- und Nachteilen für jedes der im Katalog eines Kunden enthaltenen Produkte zu extrahieren. Sie haben auch gezeigt, dass ein Reader dazu verwendet werden kann, benannte Entitäten (Eigennamen) im Stile einer ZeroShot-Klassifizierung zu extrahieren, indem er Abfragen wie »Welche Art von Kamera?« erzeugt. In Anbetracht der Tatsache, dass das generative QA noch in den Kinderschuhen steckt und mit leichten Fehlern behaftet ist, empfehlen wir, es erst dann auszuprobieren, wenn Sie die beiden anderen vorgestellten Ansätze bereits ausgeschöpft haben. Diese »Hierarchie der Bedürfnisse« bzw. Bedürfnispyramide für die Bewältigung von QA-Problemstellungen wird in Abbildung 7-14 dargestellt.

Ein spannender Forschungsbereich ist das *multimodale QA*, bei dem es um QA über mehrere Modalitäten wie Text, Tabellen und Bilder geht. Wie in der MultiModalQA-Benchmark beschrieben[17], könnten solche Systeme Benutzern die Beantwortung komplexer Fragen ermöglichen, die Informationen über verschiedene Modalitäten hinweg integrieren, wie z.B. »Wann wurde das berühmte Gemälde mit den zwei sich berührenden Fingern fertiggestellt?« Ein weiterer Bereich mit praktischen Geschäftsanwendungen ist das QA auf Basis eines *Knowledge Graph* (»Wissensgraph« bzw. auch semantisches Netz genannt), wobei die Knoten des Graphen realen Entitäten entsprechen und ihre Beziehungen durch die Kanten definiert sind. Indem Fakten als Tripel (*Subjekt*, *Prädikat*, *Objekt*) codiert werden, kann der Graph dazu verwendet werden, Fragen über ein fehlendes Element zu beantworten. Ein Beispiel dafür, wie sich Transformer-Modelle und Knowledge Graphs kombinieren lassen, finden Sie in den Tutorials von Haystack (*https://oreil.ly/n7lZb*). Eine weitere vielversprechende Forschungsrichtung ist die *automatische Fragengenerierung* (engl. Automatic Question Generation), bei der eine

17 A. Talmor et al., »MultiModalQA: Complex Question Answering over Text, Tables and Images« (*https://arxiv.org/abs/2104.06039*), (2021).

Form von unüberwachtem oder schwach überwachtem (engl. unsupervised/weakly supervised) Training unter Verwendung von ungelabelten oder augmentierten Daten (engl. Data Augmentation) durchgeführt wird. Zwei neuere Beispiele sind die Forschungsbeiträge zur Probably-Answered-Questions-(PAQ-)Benchmark und zur synthetischen Datenaugmentierung für sprachenübergreifende Settings.[18]

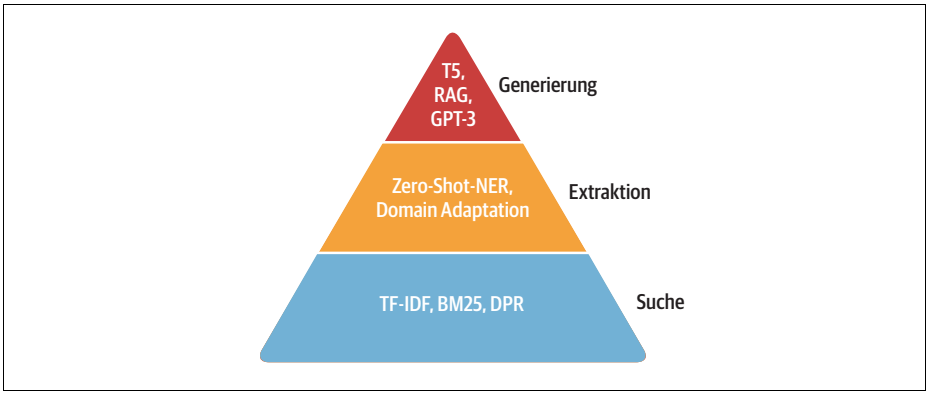


Abbildung 7-14: Die Bedürfnispyramide beim Question Answering

In diesem Kapitel haben wir gelernt, dass wir einige Tricks anwenden müssen, um QA-Modelle erfolgreich in der Praxis einsetzen zu können, z. B. eine schnelle Retrieval-Pipeline zu implementieren, um Vorhersagen nahezu in Echtzeit treffen zu können. Dennoch kann die Anwendung eines QA-Modells auf eine Handvoll vorausgewählter Dokumente auf Produktionssystemen ein paar Sekunden dauern. Das mag zunächst nicht sonderlich lang erscheinen. Doch stellen Sie sich einmal vor, wie Sie sich fühlen würden, wenn Sie ein paar Sekunden auf die Ergebnisse einer Google-Suche warten müssten – zumal ein paar Sekunden Wartezeit über das Schicksal Ihrer Transformer-gestützten Anwendung entscheiden könnten. Im nächsten Kapitel werden wir uns einigen Methoden widmen, mit denen wir die Vorhersage von Modellen zusätzlich beschleunigen können.

18 P. Lewis et al., »PAQ: 65 Million Probably-Asked Questions and What You Can Do with Them« (*https://arxiv.org/abs/2102.07033*), (2021); A. Riabi et al., »Synthetic Data Augmentation for Zero-Shot Cross-Lingual Question Answering« (*https://arxiv.org/abs/2010.12643*), (2020).

KAPITEL 8

Effizientere Transformer-Modelle für die Produktion

In den vorangegangenen Kapiteln haben Sie erfahren, wie Transformer-Modelle feingetunt werden können, sodass sie bei einer Vielzahl von Aufgaben hervorragende Ergebnisse liefern. In vielen Situationen reicht es allerdings nicht aus, nur auf die Treffergenauigkeit (bzw. das Maß, das Sie optimieren wollen) abzuzielen. Auch wenn Ihr Modell State of the Art ist, nützt es Ihnen nicht viel, wenn es zu langsam oder zu groß ist, um die Anforderungen Ihrer Anwendung zu erfüllen. Eine naheliegende Alternative ist, ein schnelleres und kompakteres Modell zu trainieren. Allerdings geht eine geringere Größe eines Modells oft mit einer schlechteren Leistung einher. Was können Sie also tun, wenn Sie ein schnelles, kompaktes und dennoch äußerst genaues Modell benötigen?

In diesem Kapitel werden wir vier sich ergänzende Verfahren untersuchen, mit denen Sie Vorhersagen schneller treffen und den Speicherbedarf Ihrer Transformer-Modelle verringern können: die *Knowledge Distillation* (»Wissensdestillation«), die *Quantisierung* (engl. Quantization), das *Pruning* und die *Optimierung über Graphen* (engl. Graph Optimization), wobei wir das Format Open Neural Network Exchange (ONNX) und die ONNX Runtime (ORT) verwenden. Wir werden auch sehen, wie sich einige dieser Verfahren kombinieren lassen, sodass erhebliche Leistungssteigerungen erzielt werden können. Beispielsweise wurde dieser Ansatz vom Roblox-Entwicklungsteam in dem Artikel »How We Scaled Bert to Serve 1+ Billion Daily Requests on CPUs« (*https://oreil.ly/QdNIk*) verfolgt. Wie in Abbildung 8-1 gezeigt, konnten sie die Latenzzeit und den Durchsatz ihres BERT-Klassifikators um mehr als den Faktor 30 verbessern, indem sie die Knowledge Distillation und die Quantisierung miteinander kombiniert haben!

Um die mit den einzelnen Verfahren einhergehenden Vorteile und Kompromisse zu veranschaulichen, werden wir die Intentions- bzw. Absichtserkennung (engl. Intent Detection) als Fallstudie heranziehen. Sie ist eine wichtige Komponente textbasierter Assistenten, bei denen niedrige Latenzzeiten entscheidend sind, um ein Gespräch in Echtzeit aufrechterhalten zu können. Nebenbei lernen Sie, wie Sie selbst definierte `Trainer`-Klassen erstellen und die Suche nach Hyperparametern effizient gestalten können. Zudem gewinnen Sie einen ersten Einblick, was es bedarf, um modernste Forschungsmethoden mithilfe der 🤗 Transformers-Bibliothek zu implementieren. Legen wir los!

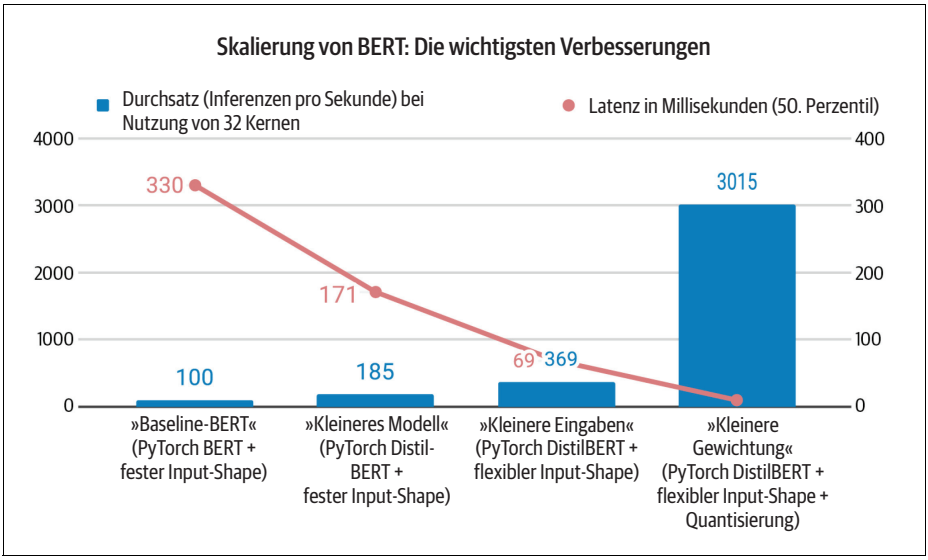


Abbildung 8-1: Wie das Entwicklungsteam von Roblox ein BERT-Modell unter Verwendung von Knowledge Distillation, Dynamic Padding und Weight Quantization (Quantisierung der Gewichtung) entwickelt hat (mit freundlicher Genehmigung der Roblox-Mitarbeiter Quoc N. Le und Kip Kaehler)

Die Intentionserkennung als Fallstudie

Nehmen wir an, wir versuchen, einen textbasierten Assistenten für das Callcenter unseres Unternehmens zu entwickeln, damit Kunden ihren Kontostand abfragen oder Buchungen vornehmen können, ohne mit einem Mitarbeiter sprechen zu müssen. Um das Anliegen eines Kunden zu verstehen, muss unser Assistent in der Lage sein, eine Vielzahl von Texten in natürlicher Sprache in eine Reihe von vordefinierten Aktionen bzw. *Intentionen* zu klassifizieren. Ein Kunde könnte zum Beispiel eine Nachricht wie die folgende über eine bevorstehende Reise zusenden:

> Hallo, ich möchte vom 1. bis 15. November ein Fahrzeug in Paris mieten und benötige einen Kleinbus für 15 Personen.

Und unser Intentionsklassifikator könnte dies automatisch als Intention, die *Autovermietung* in Anspruch zu nehmen, kategorisieren, woraufhin eine Aktion und eine Antwort ausgelöst werden. Um sich in einer Produktionsumgebung als zuverlässig zu erweisen, muss unser Klassifikator auch in der Lage sein, Anfragen zu behandeln, die nicht zu den vordefinierten Intentionen gehören (sogenannte Out-of-Scope-Anfragen). Das System sollte dann eine Ausweichantwort geben. Im zweiten Beispiel, das in Abbildung 8-2 gezeigt wird, stellt ein Kunde zum Beispiel eine Frage zum Thema Sport (das außerhalb der vordefinierten Kategorien liegt). Der Textassistent stuft diese Frage fälschlicherweise als eine der bekannten Kategorien ein und gibt als Ergebnis einen Text zurück, der sich auf die Kategorie Zahltag bezieht. Im dritten Beispiel wurde der Textassistent darauf trainiert, Anfragen zu erkennen, die außerhalb des vordefinierten Bereichs liegen (normalerweise als eigene

Kategorie gekennzeichnet), und informiert den Kunden darüber, zu welchem Thema er Fragen beantworten kann.

Abbildung 8-2: Drei Dialoge zwischen einem Menschen (rechts) und einem textbasierten Assistenten (links) aus dem Bereich der persönlichen Finanzberatung (mit freundlicher Genehmigung von Stefan Larson et al.)

Als Referenz bzw. Baseline verwenden wir ein feingetuntes Modell, das auf Basis des BERT-Modells entwickelt wurde und auf dem CLINC150-Datensatz eine Treffergenauigkeit von etwa 94 % erreicht.[1] Dieser Datensatz enthält 22.500 In-Scope-Anfragen aus 150 verschiedenen Intentionen und 10 Bereichen wie Bankwesen und Reisen sowie 1.200 Out-of-Scope-Anfragen, die einer zusätzlichen Intentionskategorie namens `oos` zugewiesen wurden. In der Praxis würden wir zudem unseren eigenen Datensatz erstellen, aber die Verwendung öffentlicher Daten bietet eine hervorragende Möglichkeit, schnell zu iterieren und erste Ergebnisse zu erzielen.

Laden wir zunächst ein feingetuntes Modell vom Hugging Face Hub herunter und binden wir es in eine Pipeline zur Textklassifizierung ein:

```
from transformers import pipeline

bert_ckpt = "transformersbook/bert-base-uncased-finetuned-clinc"
pipe = pipeline("text-classification", model=bert_ckpt)
```

Nachdem wir nun eine Pipeline instanziiert haben, können wir der Pipeline eine Anfrage übergeben, um von dem Modell die vorhergesagte Intention und den Konfidenzwert (engl. Confidence Score) zu erhalten:

```
query = """Hey, I'd like to rent a vehicle from Nov 1st to Nov 15th in
Paris and I need a 15 passenger van"""
pipe(query)
```

```
[{'label': 'car_rental', 'score': 0.549003541469574}]
```

1 S. Larson et al., »An Evaluation Dataset for Intent Classification and Out-of-Scope Prediction« (*https://arxiv.org/abs/1909.02027*), (2019).

Sehr gut, die Intention »Autovermietung« (`car_rental`) ergibt Sinn. Erstellen wir nun eine Benchmark, mit der wir die Leistung unseres Baseline-Modells evaluieren können.

Eine Benchmark-Klasse zur Beurteilung der Performance erstellen

Wie bei anderen Machine-Learning-Modellen auch, müssen Sie zwischen verschiedenen Kriterien abwägen, wenn Sie Transformer-Modelle in Produktionsumgebungen deployen. Die häufigsten[2] sind:

Modellleistung bzw. Qualität des Modells (engl. Model Performance)
: Wie gut schneidet unser Modell auf einem sorgfältig erstellten Testdatensatz ab, der die Produktionsdaten gut abbildet? Dies ist besonders wichtig, wenn die Kosten von Fehlern hoch ausfallen (und am besten von Menschen in der Schleife (»Human-in-the-Loop«) abgemildert werden) oder wenn wir im Rahmen der Inferenz Vorhersagen für Millionen von Beispielen treffen müssen und kleine Verbesserungen der Modellmaße zu großen Gewinnen in der Gesamtheit führen können.

Latenz (engl. Latency)
: Wie schnell kann unser Modell Vorhersagen liefern? In Echtzeitumgebungen, die von einem hohen Datenverkehr geprägt sind, ist die Latenzzeit von großer Bedeutung. Zum Beispiel benötigte Stack Overflow einen Klassifikator, um schneller unerwünschte Kommentare auf der Webseite erkennen zu können (*https://oreil.ly/cf7QX*).

Speicher (engl. Memory)
: Wie können wir Modelle mit Milliarden von Parametern wie GPT-2 oder T5 deployen, die Gigabytes an Festplatten- und Arbeitsspeicher (RAM) erfordern? Der vorhandene Speicher spielt eine besonders wichtige Rolle bei mobilen oder Edge-Geräten, wo ein Modell Vorhersagen treffen muss, ohne Zugang zu einem leistungsstarken Cloud-Server zu haben.

Wenn Sie diese Kriterien nicht beachten, kann sich dies negativ auf die Benutzerfreundlichkeit Ihrer Anwendung auswirken. Noch häufiger führt es zu steigenden Kosten, da teure Cloud-Server betrieben werden, die vielleicht nur wenige Anfragen bearbeiten müssen. Damit wir besser nachvollziehen können, wie jedes dieser Kriterien mit verschiedenen Komprimierungsverfahren optimiert werden kann, erstellen wir zunächst eine einfache Benchmark-Klasse, die jedes der genannten Kriterien für eine vorgegebene Pipeline und einen vorgegebenen Testdatensatz erfasst. Als Grundgerüst verwenden wir die folgende Klasse:

2 Wie von Emmanuel Ameisen in *Building Machine Learning Powered Applications* (O'Reilly) beschrieben, sind Geschäfts- oder Produktkennzahlen die *wichtigsten*, die Sie berücksichtigen müssen. Schließlich ist es egal, wie genau Ihr Modell ist, wenn es keines der Probleme löst, die für Ihr Unternehmen von Bedeutung sind. In diesem Kapitel gehen wir davon aus, dass Sie die für Ihre Anwendung relevanten Kennzahlen bereits festgelegt haben, und konzentrieren uns auf die Optimierung der Modellmaße.

```
class PerformanceBenchmark:
    def __init__(self, pipeline, dataset, optim_type="BERT baseline"):
        self.pipeline = pipeline
        self.dataset = dataset
        self.optim_type = optim_type

    def compute_accuracy(self):
        # Werden wir später definieren
        pass

    def compute_size(self):
        # Werden wir später definieren
        pass

    def time_pipeline(self):
        # Werden wir später definieren
        pass

    def run_benchmark(self):
        metrics = {}
        metrics[self.optim_type] = self.compute_size()
        metrics[self.optim_type].update(self.time_pipeline())
        metrics[self.optim_type].update(self.compute_accuracy())
        return metrics
```

Wir haben einen Parameter `optim_type` definiert, mit dem wir die Ergebnisse der verschiedenen Optimierungsverfahren festhalten, die wir in diesem Kapitel behandeln werden. Die Methode `run_benchmark()` verwenden wir, um alle Maße in einem Dictionary zu sammeln, wobei die Schlüssel mittels `optim_type` angegeben werden.

Verwenden wir nun die Klasse, um die Treffergenauigkeit (engl. Accuracy) des Modells auf dem Testdatensatz zu ermitteln. Zunächst benötigen wir einige Daten zum Testen. Laden wir also den CLINC150-Datensatz herunter, der für das Feintuning unseres (zuvor geladenen) Baseline-Modells verwendet wurde. Den Datensatz können wir mithilfe der 🤗 Datasets-Bibliothek vom Hub abrufen:

```
from datasets import load_dataset

clinc = load_dataset("clinc_oos", "plus")
```

Hier bezieht sich die `plus`-Konfiguration auf den Teil der Daten, in dem die Trainingsbeispiele enthalten sind, die sich nicht zu den vordefinierten Kategorien zuordnen lassen (Out-of-Scope-Daten). Jedes Beispiel im CLINC150-Datensatz besteht aus einer Anfrage, die in der Spalte `text` enthalten ist, und der entsprechenden Intention. Um unsere Modelle zu vergleichen, werden wir auf den Testdatensatz zurückgreifen. Sehen wir uns also eines der Beispiele aus dem Testdatensatz an:

```
sample = clinc["test"][42]
sample
```

```
{'intent': 133, 'text': 'transfer $100 from my checking to saving account'}
```

Die Intentionen werden in Form von IDs bereitgestellt. Wir können jedoch ganz einfach die jeweils zugeordneten Strings erhalten (und umgekehrt), indem wir auf das `features`-Attribut des Datensatz-Objekts zugreifen:

```
intents = clinc["test"].features["intent"]
intents.int2str(sample["intent"])
```

```
'transfer'
```

Nachdem wir nun einen Überblick über den CLINC150-Datensatz gewonnen haben, können wir die Methode `compute_accuracy()` der `PerformanceBenchmark`-Klasse implementieren. Da der Datensatz hinsichtlich der verschiedenen Kategorien von Intentionen ausgewogen verteilt ist (engl. balanced), verwenden wir als Maß die Treffergenauigkeit bzw. Korrektklassifikationsrate. In der 🤗 Datasets-Bibliothek können wir das Maß wie folgt laden:

```
from datasets import load_metric

accuracy_score = load_metric("accuracy")
```

Zur Ermittlung der Treffergenauigkeit wird vorausgesetzt, dass sowohl die Vorhersagen als auch die entsprechenden Referenzen (d.h. die Ground-Truth-Labels) als Ganzzahlen vorliegen. Mithilfe der Pipeline können wir die Vorhersagen aus dem Feld `text` extrahieren und anschließend die Methode `str2int()` unseres `intents`-Objekts verwenden, um jeder Vorhersage die entsprechende ID zuzuordnen. Mit dem folgenden Code werden alle Vorhersagen und Labels in Listen gesammelt und anschließend die auf dem Testdatensatz erzielte Treffergenauigkeit zurückgegeben. Integrieren wir nun diese Funktion ebenfalls in unsere `PerformanceBenchmark`-Klasse:

```
def compute_accuracy(self):
    """Überschreibt die PerformanceBenchmark.compute_accuracy()-Methode"""
    preds, labels = [], []
    for example in self.dataset:
        pred = self.pipeline(example["text"])[0]["label"]
        label = example["intent"]
        preds.append(intents.str2int(pred))
        labels.append(label)
    accuracy = accuracy_score.compute(predictions=preds, references=labels)
    print(f"Treffergenauigkeit auf Testdatensatz - {accuracy['accuracy']:.3f}")
    return accuracy

PerformanceBenchmark.compute_accuracy = compute_accuracy
```

Als Nächstes ermitteln wir die Größe (engl. Size) unseres Modells, indem wir die `torch.save()`-Funktion aus PyTorch verwenden, mit der das Modell auf der Festplatte serialisiert wird. Die `torch.save()`-Funktion verwendet das Python-Modul `pickle` und kann zum Speichern aller Arten von Modellen, Tensoren und gewöhnlichen Python-Objekten verwendet werden. In PyTorch wird empfohlen, Modelle unter Verwendung ihres `state_dict` zu speichern. Dabei handelt es sich um ein Python-Dictionary, das jede Schicht in einem Modell auf seine lernbaren Parameter (d.h. Gewichte und Bias-Terme) abbildet. Werfen wir einen Blick darauf, was in dem `state_dict` unseres Baseline-Modells gespeichert ist:

```
list(pipe.model.state_dict().items())[42]
```

```
('bert.encoder.layer.2.attention.self.value.weight',
 tensor([[-1.0526e-02, -3.2215e-02,  2.2097e-02,  ..., -6.0953e-03,
           4.6521e-03,  2.9844e-02],
         [-1.4964e-02, -1.0915e-02,  5.2396e-04,  ...,  3.2047e-05,
          -2.6890e-02, -2.1943e-02],
         [-2.9640e-02, -3.7842e-03, -1.2582e-02,  ..., -1.0917e-02,
           3.1152e-02, -9.7786e-03],
         ...,
         [-1.5116e-02, -3.3226e-02,  4.2063e-02,  ..., -5.2652e-03,
           1.1093e-02,  2.9703e-03],
         [-3.6809e-02,  5.6848e-02, -2.6544e-02,  ..., -4.0114e-02,
           6.7487e-03,  1.0511e-03],
         [-2.4961e-02,  1.4747e-03, -5.4271e-02,  ...,  2.0004e-02,
           2.3981e-02, -4.2880e-02]]))
```

Offensichtlich entspricht jedes Schlüssel/Wert-Paar einer bestimmten Schicht bzw. einem bestimmten Tensor im BERT-Modell. Wenn wir also unser Modell speichern mit:

```
torch.save(pipe.model.state_dict(), "model.pt")
```

können wir anschließend die Funktion `Path.stat()` aus Pythons `pathlib`-Modul verwenden, um Informationen über die zugrunde liegenden Dateien zu erhalten. Im vorliegenden Fall erhalten wir über `Path("model.pt").stat().st_size` die Modellgröße in Bytes. Bringen wir das nun alles in einer Funktion, `compute_size()`, unter, die wir in der `PerformanceBenchmark`-Klasse aufnehmen (bzw. überschreiben wir die bereits angelegte Funktion):

```
import torch
from pathlib import Path

def compute_size(self):
    """Überschreibt die PerformanceBenchmark.compute_size()-Methode"""
    state_dict = self.pipeline.model.state_dict()
    tmp_path = Path("model.pt")
    torch.save(state_dict, tmp_path)
    # Größe in Megabytes ermitteln
    size_mb = Path(tmp_path).stat().st_size / (1024 * 1024)
    # Temporäre Datei löschen
    tmp_path.unlink()
    print(f"Modellgröße (MB) - {size_mb:.2f}")
    return {"size_mb": size_mb}

PerformanceBenchmark.compute_size = compute_size
```

Zu guter Letzt implementieren wir die Funktion `time_pipeline()`, mit der wir die durchschnittliche Latenzzeit pro Anfrage ermitteln. Bei dieser Anwendung bezieht sich die Latenzzeit auf die Zeit, die benötigt wird, um eine Anfrage an die Pipeline zu übermitteln und die vom Modell vorhergesagte Intention zurückzuliefern. Im Hintergrund führt die Pipeline auch die Tokenisierung des Texts durch, die jedoch rund tausendmal schneller als die Erstellung der Vorhersagen vonstattengeht und daher nur unwesentlich zur Gesamtlatenz beiträgt. Sie kön-

nen die Ausführungszeit eines Codeabschnitts ziemlich einfach messen, indem Sie die Funktion `perf_counter()` aus dem Python-Modul `time` verwenden. Bei dieser Funktion wird die Zeit genauer erfasst als bei der Funktion `time.time()`, weshalb sie besser geeignet ist, wenn wir möglichst präzise Ergebnisse erhalten möchten.

Die `perf_counter()`-Funktion können wir dazu verwenden, einen Timer für unsere Pipeline zu setzen. Dazu übergeben wir unsere Testanfrage und ermitteln die Zeit, die zwischen Beginn und Ende der Ausführung der Pipeline vergeht, in Millisekunden:

```
from time import perf_counter

for _ in range(3):
    start_time = perf_counter()
    _ = pipe(query)
    latency = perf_counter() - start_time
    print(f"Latenz (ms) - {1000 * latency:.3f}")
```

```
Latenz (ms) - 85.367
Latenz (ms) - 85.241
Latenz (ms) - 87.275
```

Die ermittelten Latenzen variieren ziemlich stark, was darauf hindeutet, dass die gemessene Zeit von Durchlauf zu Durchlauf völlig unterschiedlich ausfallen kann. Aus diesem Grund erheben wir die Latenzzeiten über viele Durchläufe hinweg und berechnen anschließend den Mittelwert und die Standardabweichung für die resultierende Verteilung, auf deren Basis wir dann beurteilen können, wie groß die Streuung der Werte ausfällt. Der folgende Code setzt diesen Ansatz für die Zeitmessung um. Bevor die eigentliche Zeitmessung durchgeführt wird, ist eine Phase zum Aufwärmen der CPU (»Warm up«) vorgesehen:

```
import numpy as np

def time_pipeline(self, query="What is the pin number for my account?"):
    """Überschreibt die PerformanceBenchmark.time_pipeline()-Methode"""
    latencies = []
    # Warm-up-Phase
    for _ in range(10):
        _ = self.pipeline(query)
    # Timer für jeden Durchlauf
    for _ in range(100):
        start_time = perf_counter()
        _ = self.pipeline(query)
        latency = perf_counter() - start_time
        latencies.append(latency)
    # Statistiken für Durchläufe ermitteln
    time_avg_ms = 1000 * np.mean(latencies)
    time_std_ms = 1000 * np.std(latencies)
    print(f"Durchschnittliche Latenz (ms) - {time_avg_ms:.2f} +\-
                                          {time_std_ms:.2f}")
    return {"time_avg_ms": time_avg_ms, "time_std_ms": time_std_ms}

PerformanceBenchmark.time_pipeline = time_pipeline
```

Der Einfachheit halber verwenden wir immer dieselbe Anfrage (`query`), damit alle unsere Modelle vergleichbar sind. Im Allgemeinen hängt die Latenzzeit von der Länge der Anfrage ab. Es empfiehlt sich daher, Ihre Modelle mit Anfragen zu testen, die Sie wahrscheinlich auch in einer Produktionsumgebung antreffen würden.

Unsere `PerformanceBenchmark`-Klasse ist nun vollständig – legen wir also los! Zunächst führen wir das Benchmarking unseres BERT-Baseline-Modells durch. Für das Baseline-Modell müssen wir nur die Pipeline und den Datensatz übergeben, anhand dessen wir die Performance-Benchmark erstellen möchten. Um die Leistung aller Modelle festzuhalten, sammeln wir die Ergebnisse im `perf_metrics`-Dictionary:

```
pb = PerformanceBenchmark(pipe, clinc["test"])
perf_metrics = pb.run_benchmark()
```

```
Modellgröße (MB) - 418.16
Durchschnittliche Latenz (ms) - 54.20 +\- 1.91
Treffergenauigkeit auf Testdatensatz - 0.867
```

Damit haben wir nun eine Vergleichsgrundlage. Nun können wir uns unserem ersten Kompressionsverfahren zuwenden: der Knowledge Distillation.

Die durchschnittlichen Latenzwerte hängen davon ab, auf welcher Art von Hardware Sie den Code ausführen. Zum Beispiel können Sie in der Regel eine bessere Performance erzielen, wenn Sie die Inferenz auf einer GPU durchführen, da diese eine Verarbeitung in Form von Batches (engl. Batch Processing) unterstützt. Im Rahmen dieses Kapitels ist für unsere Zwecke vor allem der relative Unterschied in den Latenzen zwischen den Modellen relevant. Sobald wir das performanteste Modell ermittelt haben, können wir anschließend – insofern erforderlich – verschiedene Backends vergleichen, um auch die absolute Latenz zu verringern.

Verkleinerung von Modellen mithilfe der Knowledge Distillation

Die Knowledge Distillation ist ein allgemein anwendbares Verfahren zum Trainieren eines kleineren *Schüler*-Modells (engl. Student), um das Verhalten eines langsameren, größeren, aber leistungsfähigeren *Lehrer*-Modells (Teacher) zu imitieren. Ursprünglich wurde es im Jahr 2006[3] im Zusammenhang mit Ensemble-Modellen eingeführt, gewann jedoch erst später durch einen bekannten Artikel aus dem Jahr 2015 an Popularität, in dem es auf tiefe neuronale Netze verallgemeinert und auf die Klassifizierung von Bildern und die automatische Spracherkennung übertragen wurde.[4]

3 C. Buciluǎ et al., »Model Compression,« *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (August 2006): 535–541, *https://doi.org/10.1145/1150402.1150464*.

4 G. Hinton, O. Vinyals, and J. Dean, »Distilling the Knowledge in a Neural Network« (*https://arxiv.org/abs/1503.02531*), (2015).

Angesichts des Trends, Sprachmodelle vorzutrainieren, die immer mehr Parameter umfassen (das größte Modell hat zum Zeitpunkt des Verfassens dieses Buchs über eine Billion Parameter)[5], ist die Knowledge Distillation ebenfalls zu einer beliebten Strategie geworden, um diese riesigen Modelle zu komprimieren, sodass sie sich besser dazu eignen, praktische Anwendungen zu entwickeln.

Knowledge Distillation im Rahmen des Feintunings

Wie wird das Wissen also tatsächlich »destilliert« bzw. während des Trainings vom Lehrer auf den Schüler übertragen? Bei überwachten Aufgaben wie dem Feintuning besteht die Hauptidee darin, die Ground-Truth-Labels mit einer vom Lehrer erhaltenen Verteilung »geglätteter Wahrscheinlichkeiten« (engl. Soft Probabilities) zu ergänzen, die dem Schüler zusätzliche Informationen liefern, aus denen er lernen kann. Wenn unser BERT-basierter Klassifikator beispielsweise mehreren Intentionen hohe Wahrscheinlichkeiten zuweist, dann könnte dies ein Zeichen dafür sein, dass diese Intentionen im Merkmalsraum bzw. Feature-Space nahe beieinanderliegen. Indem wir den Schüler darauf trainieren, diese Wahrscheinlichkeiten nachzuahmen, wollen wir einen Teil dieses »Dark Knowledge«[6] herausdestillieren, das der Lehrer gelernt hat – d. h. Wissen, das sich nicht allein auf Basis der Labels ermitteln lässt.

Mathematisch gesehen funktioniert das Ganze folgendermaßen. Nehmen wir an, wir geben dem Lehrer eine Eingabesequenz x, um einen Vektor aus Logits $\mathbf{z}(x) = [z_1(x),..., z_N(x)]$ zu erzeugen. Diese Logits können wir durch Anwendung einer Softmax-Funktion in Wahrscheinlichkeiten überführen:

$$\frac{\exp\left(z_i(x)\right)}{\sum_j \exp\left(z_i(x)\right)}$$

Das entspricht jedoch nicht ganz dem, worauf wir abzielen, denn in vielen Fällen wird der Lehrer einer Kategorie eine hohe Wahrscheinlichkeit zuweisen, während die Wahrscheinlichkeiten aller anderen Kategorien gegen null gehen. Wenn dies der Fall sein sollte, liefert der Lehrer neben den Ground-Truth-Labels nicht sonderlich viele zusätzliche Informationen. Deshalb »glätten« wir zunächst die Wahrscheinlichkeiten, indem wir die Logits mithilfe eines Hyperparameters, der Temperature T, skalieren, bevor wir die Softmax-Funktion anwenden[7]:

5 W. Fedus, B. Zoph und N. Shazeer, »Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity« (*https://arxiv.org/abs/2101.03961*), (2021).

6 Geoff Hinton hat diesen Begriff in einem Vortrag (*https://oreil.ly/OkHGp*) geprägt, um sich auf die Beobachtung zu beziehen, dass geglättete Wahrscheinlichkeiten das verborgene Wissen des Lehrers offenbaren.

7 Auf die Temperature sind wir bereits im Zusammenhang mit der Textgenerierung in Kapitel 5 eingegangen.

$$p_i(x) = \frac{\exp\left(z_i(x)/T\right)}{\sum_j \exp\left(z_i(x)/T\right)}$$

Wie in Abbildung 8-3 gezeigt, bedingen höhere Werte von T eine glatter (bzw. weniger steil) verlaufende Wahrscheinlichkeitsverteilung über die verschiedenen Kategorien hinweg, wodurch wir wesentlich mehr Aufschluss über die vom Lehrer erlernte Entscheidungsgrenze im Hinblick auf die einzelnen Trainingsbeispiele erhalten. Wenn $T = 1$ gesetzt wird, erhalten wir wieder die ursprüngliche Softmax-Verteilung.

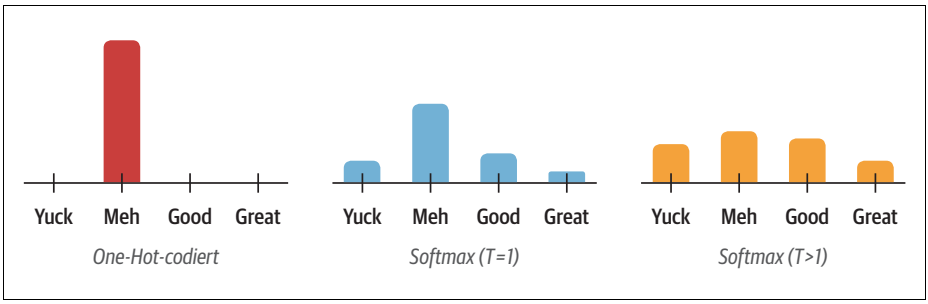


Abbildung 8-3: Unterschied zwischen einem »festen« Label, das 1-aus-n- bzw. One-Hot-codiert wurde (links), Softmax-Wahrscheinlichkeiten (Mitte) und geglätteter Wahrscheinlichkeiten für die Kategorien (rechts)

Da auch der Schüler geglättete Wahrscheinlichkeiten $q_i(x)$ ermittelt bzw. ausgibt, können wir die Kullback-Leibler-(KL-)Divergenz (*https://oreil.ly/8nKQG*) verwenden, um den Unterschied zwischen den beiden Wahrscheinlichkeitsverteilungen zu quantifizieren:

$$D_{KL}(p, q) = \sum_i p_i(x) \log \frac{p_i(x)}{q_i(x)}$$

Mit der KL-Divergenz (auch relative Entropie genannt) können wir berechnen, wie viel an Informationen verloren geht, wenn wir die Wahrscheinlichkeitsverteilung des Lehrers (Teachers) durch die des Schülers (Student) approximieren. Dementsprechend können wir mithilfe der KL-Divergenz einen Verlust definieren, der auf die Knowledge Distillation zurückzuführen ist:

$$L_{KD} = T^2 D_{KL}$$

wobei T^2 ein Normalisierungsfaktor ist, der der Tatsache Rechnung trägt, dass die Größe der Gradienten, die durch »geglättete Labels« erzeugt werden, mit dem Faktor $1/T^2$ skaliert wurden. Bei Klassifizierungsaufgaben entspricht der Verlust des Schülers (Student) dann dem gewichteten Durchschnitt des Destillationsverlusts

und des gewöhnlichen Kreuzentropieverlusts I_{CE} (engl. Cross-Entropy Loss), der auf die Abweichung von den Ground-Truth-Labels zurückzuführen ist:

$$L_{\text{Student}} = \alpha L_{CE} + (1 - \alpha) L_{KD}$$

wobei α ein Hyperparameter ist, der die relative Gewichtung der beiden Verluste bestimmt. In Abbildung 8-4 können Sie den gesamten Vorgang nachvollziehen. Zum Zeitpunkt der Inferenz wird die Temperature auf 1 gesetzt, um die standardmäßigen Softmax-Wahrscheinlichkeiten zu erhalten.

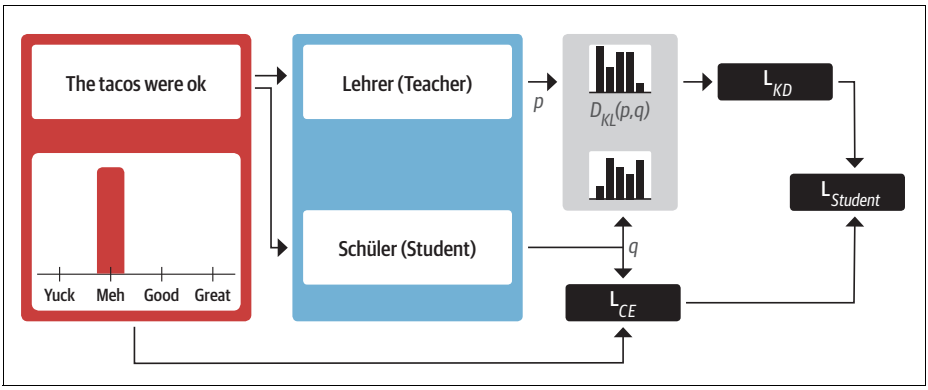


Abbildung 8-4: Das Vorgehen bei der Knowledge Distillation

Knowledge Distillation im Rahmen des Pretrainings

Die Knowledge Distillation kann auch während des Pretrainings angewandt werden, um einen universell einsetzbaren Schüler zu erstellen, der anschließend auf nachgelagerte Aufgaben feingetunt werden kann. In diesem Fall ist der Lehrer ein vortrainiertes Sprachmodell wie BERT, das sein Wissen über die maskierte Sprachmodellierung an den Studenten weitergibt. Im DistilBERT-Forschungspapier wird zum Beispiel der Verlust,[8] der durch die maskierte Sprachmodellierung (MLM) bedingt ist, L_{mlm}, um einen auf die Knowledge Distillation zurückzuführenden Term und einen auf der Cosinusfunktion basierenden Einbettungsverlust bzw. Embedding-Loss, $L_{cos} = 1 - \cos(h_s, h_t)$, ergänzt, um die Ausrichtung der verborgenen Zustandsvektoren zwischen dem Lehrer und dem Schüler anzugleichen:

$$L_{\text{DistilBERT}} = \alpha L_{mlm} + \beta L_{KD} + \gamma L_{cos}$$

Da wir bereits ein feingetuntes Modell auf Basis des BERT-Modells zur Hand haben, können wir nun versuchen, mithilfe der Knowledge Distillation ein Modell feinzutunen, das kleiner und schneller ist. Dazu benötigen wir allerdings die Mög-

8 V. Sanh et al., »DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter« (*https://arxiv.org/abs/1910.01108*), (2019).

lichkeit, den Kreuzentropieverlust um den Term L_{KD} zu erweitern. Glücklicherweise können wir das mithilfe eines selbst definierten Trainers umsetzen!

Eine Trainer-Klasse für die Knowledge Distillation erstellen

Um die Knowledge Distillation implementieren zu können, müssen wir der Basisklasse `Trainer` einige Dinge hinzufügen:

- die neuen Hyperparameter α und T, die die relative Gewichtung des Destillationsverlusts und den Grad der Glättung der Wahrscheinlichkeitsverteilung der Labels steuern
- das feingetunte Lehrer-Modell, das in unserem Fall BERT-basiert ist
- eine neue Verlustfunktion, die den Kreuzentropieverlust und den Verlust, der aus der Knowledge Distillation resultiert, kombiniert

Die neuen Hyperparameter hinzuzufügen, ist ganz einfach: Wir müssen lediglich eine Klasse anlegen, die von der `TrainingArguments`-Klasse erbt, und die Hyperparameter als neue Attribute aufnehmen:

```
from transformers import TrainingArguments

class DistillationTrainingArguments(TrainingArguments):
    def __init__(self, *args, alpha=0.5, temperature=2.0, **kwargs):
        super().__init__(*args, **kwargs)
        self.alpha = alpha
        self.temperature = temperature
```

Für die `Trainer`-Klasse benötigen wir eine neue Verlustfunktion. Dazu müssen wir eine Klasse anlegen, die von der `Trainer`-Klasse erbt, und die Methode `compute_loss()` überschreiben, um den Verlustterm der Knowledge Distillation L_{KD} einbeziehen zu können:

```
import torch.nn as nn
import torch.nn.functional as F
from transformers import Trainer

class DistillationTrainer(Trainer):
    def __init__(self, *args, teacher_model=None, **kwargs):
        super().__init__(*args, **kwargs)
        self.teacher_model = teacher_model

    def compute_loss(self, model, inputs, return_outputs=False):
        outputs_stu = model(**inputs)
        # Kreuzentropieverlust und Logits vom Schüler (Student) extrahieren
        loss_ce = outputs_stu.loss
        logits_stu = outputs_stu.logits
        # Logits vom Lehrer (Teacher) extrahieren
        with torch.no_grad():
            outputs_tea = self.teacher_model(**inputs)
            logits_tea = outputs_tea.logits
        # Glätten der Wahrscheinl. und Berechnung des Destillationsverlusts
        loss_fct = nn.KLDivLoss(reduction="batchmean")
        loss_kd = self.args.temperature ** 2 * loss_fct(
            F.log_softmax(logits_stu / self.args.temperature, dim=-1),
```

```
        F.softmax(logits_tea / self.args.temperature, dim=-1))
    # Gewichteten Verlust für Schüler/Student zurückgeben
    loss = self.args.alpha * loss_ce + (1. - self.args.alpha) * loss_kd
    return (loss, outputs_stu) if return_outputs else loss
```

Nehmen wir diesen Code ein wenig genauer unter die Lupe. Wenn wir die `DistillationTrainer`-Klasse instanziieren, übergeben wir als Argument ein `teacher_model` – ein Lehrer-Modell, das bereits auf unsere Aufgabe feingetunt wurde. Als Nächstes extrahieren wir mit der `compute_loss()`-Methode die Logits des Schüler- und des Lehrer-Modells, skalieren sie mithilfe der Temperature und normalisieren sie dann mittels der Softmax-Funktion. Anschließend übergeben wir sie an die PyTorch-Funktion `nn.KLDivLoss()`, um die KL-Divergenz zu berechnen. Eine Besonderheit der Funktion `nn.KLDivLoss()` ist, dass sie die Eingaben in Form von logarithmierten Wahrscheinlichkeiten (engl. Log Probabilities) und die Labels in Form gewöhnlicher Wahrscheinlichkeiten erwartet. Aus diesem Grund haben wir die Funktion `F.log_softmax()` verwendet, die die Logits des Schüler-Modells normalisiert, wohingegen die Logits des Lehrer-Modells unter Verwendung einer standardmäßigen Softmax-Funktion in Wahrscheinlichkeiten überführt werden. Das Argument `reduction="batchmean"` in `nn.KLDivLoss()` legt fest, dass wir die Durchschnittsbildung für den Verlust auf der Basis von Batches bzw. hinsichtlich der Batch-Dimension vornehmen.

Sie können die Knowledge Distillation auch über die Keras-API der 🤗 Transformers-Bibliothek vornehmen. Dazu müssen Sie eine eigene `Distiller`-Klasse implementieren, die die Methoden `train_step()`, `test_step()` und `compile()` der `tf.keras.Model()`-Klasse überschreibt. In der Keras-Dokumentation (*https://oreil.ly/6qp0F*) finden Sie ein Beispiel, das Ihnen zeigt, wie Sie vorgehen können, und einen guten Einstieg bietet.

Ein geeignetes Modell als Ausgangspunkt für das Schüler-Modell wählen

Nachdem wir nun unsere selbst definierte `Trainer`-Klasse erstellt haben, stellt sich als Erstes die Frage: Welches vortrainierte Modell sollten wir als Schüler (Student) wählen? Im Allgemeinen sollten wir uns für ein kleineres Modell als Schüler entscheiden, um die Latenzzeit und den Speicherbedarf zu verringern. Eine gute Faustregel aus der Literatur besagt, dass die Knowledge Distillation am besten gelingt, wenn der Lehrer und der Schüler vom gleichen *Modelltyp* sind.[9] Ein möglicher Grund dafür ist, dass verschiedene Modelltypen, z. B. BERT und RoBERTa, unterschiedliche Einbettungsräume für die Ausgaben bzw. Output-Embedding-Spaces haben können, was die Fähigkeit des Schülers, den Lehrer zu imitieren, beeinträchtigt. In unserem Fall handelt es sich bei dem Lehrer um ein BERT-Modell. Dementsprechend ist ein DistilBERT-basiertes Modell ein naheliegender Kandidat zur Initialisierung des Schülers, da es

9 Y. Kim und H. Awadalla, »FastFormers: Highly Efficient Transformer Models for Natural Language Understanding« (*https://arxiv.org/abs/2010.13382*), (2020).

40% weniger Parameter umfasst und erwiesenermaßen gute Ergebnisse bei nachgelagerten Aufgaben erzielt.

Zuerst müssen wir unsere Anfragen tokenisieren und codieren. Hierzu können wir den Tokenizer des DistilBERT-Modells instanziieren und eine simple Funktion, `tokenize_text()`, die der Vorverarbeitung (engl. Preprocessing) dient, erstellen:

```
from transformers import AutoTokenizer

student_ckpt = "distilbert-base-uncased"
student_tokenizer = AutoTokenizer.from_pretrained(student_ckpt)

def tokenize_text(batch):
    return student_tokenizer(batch["text"], truncation=True)

clinc_enc = clinc.map(tokenize_text, batched=True, remove_columns=["text"])
clinc_enc = clinc_enc.rename_column("intent", "labels")
```

Die `text`-Spalte haben wir hier außen vor gelassen, da wir sie nicht mehr benötigen. Außerdem haben wir die Spalte `intent` in `labels` umbenannt, damit sie vom Trainer automatisch erkannt werden kann.[10]

Nachdem wir unsere Texte verarbeitet haben, müssen wir als Nächstes die Hyperparameter und die Funktion `compute_metrics()` für unseren `DistillationTrainer` definieren. Außerdem werden wir alle unsere Modelle auf den Hugging Face Hub übertragen. Beginnen wir also erst einmal damit, uns bei unserem Konto anzumelden:

```
from huggingface_hub import notebook_login

notebook_login()
```

Anschließend legen wir die Maße fest, die während des Trainings festgehalten werden sollen. Wie schon bei der Performance-Benchmark verwenden wir als wichtigstes Maß die Treffergenauigkeit (engl. Accuracy). Folglich können wir unsere `accuracy_score()`-Funktion innerhalb der `compute_metrics()`-Funktion wiederverwenden, die wir in die `DistillationTrainer`-Klasse integrieren werden:

```
def compute_metrics(pred):
    predictions, labels = pred
    predictions = np.argmax(predictions, axis=1)
    return accuracy_score.compute(predictions=predictions, references=labels)
```

Bei dieser Funktion liegen die Vorhersagen des Head, der für die Sequenzmodellierung zuständig ist, in Form von Logits vor. Daher verwenden wir die Funktion `np.argmax()`, um die Kategorie als Vorhersage zu bestimmen, die den höchsten Logit aufweist, und vergleichen diese mit dem Ground-Truth-Label.

Nun legen wir die Argumente für das Training fest. Zum Einstieg setzen wir $\alpha = 1$, um zu sehen, wie gut sich das DistilBERT-Modell ohne ein Signal bzw. ohne Einbezug des Lehrer-Modells schlägt.[11] Anschließend übertragen wir unser feingetun-

10 Der `Trainer` sucht beim Feintuning im Rahmen von Klassifizierungsaufgaben standardmäßig nach einer Spalte namens `labels`. Sie können dieses Verhalten auch ändern bzw. überschreiben, indem Sie das Argument `label_names` von `TrainingArguments` spezifizieren.

tes Modell in ein neues Repository namens `distilbert-base-uncased-finetuned-clinc`. Dementsprechend geben wir diesen Namen für das `output_dir`-Argument der `DistillationTrainingArguments`-Klasse an:

```
batch_size = 48

finetuned_ckpt = "distilbert-base-uncased-finetuned-clinc"
student_training_args = DistillationTrainingArguments(
    output_dir=finetuned_ckpt, evaluation_strategy = "epoch",
    num_train_epochs=5, learning_rate=2e-5,
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size, alpha=1, weight_decay=0.01,
    push_to_hub=True)
```

Wir haben auch einige der standardmäßig vorgegebenen (engl. default) Hyperparameterwerte angepasst, wie die Anzahl der Epochen, den Weight Decay und die Lernrate. Als Nächstes gilt es, das Schüler-Modell zu initialisieren. Da wir mehrere Läufe mit dem Trainer vornehmen werden, erstellen wir eine Funktion, `student_init()`, die das Modell bei jedem neuen Lauf initialisiert. Indem wir diese Funktion an die `DistillationTrainer`-Klasse übergeben, stellen wir sicher, dass wir jedes Mal ein neues Modell initialisieren, wenn wir die `train()`-Methode aufrufen.

Zudem müssen wir noch das Schüler-Modell mit dem Mapping bzw. Zuordnungen zwischen jeder Intention und den IDs für die Labels versehen. Diese Zuordnungen können wir von unserem BERT-Base-Modell, das wir in unsere Pipeline geladen haben, übernehmen:

```
id2label = pipe.model.config.id2label
label2id = pipe.model.config.label2id
```

Mit diesen Mappings können wir nun unter Verwendung der `AutoConfig`-Klasse, die wir in den Kapiteln Kapitel 3 und Kapitel 4 kennengelernt haben, eine selbst definierte Modellkonfiguration erstellen. Verwenden wir nun die Klasse, um eine Konfiguration für unser Schüler-Modell zu erstellen, in der die Zuordnung zwischen den Labels und IDs berücksichtigt wird:

```
from transformers import AutoConfig

num_labels = intents.num_classes
student_config = (AutoConfig
                  .from_pretrained(student_ckpt, num_labels=num_labels,
                                   id2label=id2label, label2id=label2id))
```

Darüber hinaus haben wir auch die Anzahl der Kategorien angegeben, die unser Modell zu erwarten hat. Die Konfiguration können wir dann der Funktion `from_pretrained()` der `AutoModelForSequenceClassification`-Klasse übergeben:

```
import torch
from transformers import AutoModelForSequenceClassification
```

11 Dieser Ansatz des Feintunings eines allgemeinen, destillierten Sprachmodells wird manchmal auch als »Task-agnostic« Distillation (also »aufgabenunabhängige Destillation«) bezeichnet.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

def student_init():
    return (AutoModelForSequenceClassification
            .from_pretrained(student_ckpt, config=student_config).to(device))
```

Wir haben nun alle erforderlichen Komponenten für unseren Distillation-Trainer vorbereitet. Laden wir also das Lehrer-Modell und nehmen wir anschließend das Feintuning des Modells vor:

```
teacher_ckpt = "transformersbook/bert-base-uncased-finetuned-clinc"
teacher_model = (AutoModelForSequenceClassification
                 .from_pretrained(teacher_ckpt, num_labels=num_labels)
                 .to(device))

distilbert_trainer = DistillationTrainer(model_init=student_init,
    teacher_model=teacher_model, args=student_training_args,
    train_dataset=clinc_enc['train'], eval_dataset=clinc_enc['validation'],
    compute_metrics=compute_metrics, tokenizer=student_tokenizer)

distilbert_trainer.train()
```

Epoche	Trainingsverlust	Validierungsverlust	Treffergenauigkeit
1	4.2923	3.289337	0.742258
2	2.6307	1.883680	0.828065
3	1.5483	1.158315	0.896774
4	1.0153	0.861815	0.909355
5	0.7958	0.777289	0.917419

Die auf dem Validierungsdatensatz erzielte Treffergenauigkeit von 92 % lässt sich im Vergleich zu den 94 %, die das BERT-basierte Lehrer-Modell erreicht, durchaus als gut bezeichnen. Nachdem wir nun das DistilBERT-Modell feingetunt haben, können wir es auf den Hub übertragen, um es später wiederverwenden zu können:

```
distilbert_trainer.push_to_hub("Training completed!")
```

Da unser Modell jetzt sicher auf dem Hub gespeichert ist, können wir es direkt in der Pipeline für unsere Performance-Benchmark verwenden:

```
finetuned_ckpt = "transformersbook/distilbert-base-uncased-finetuned-clinc"
pipe = pipeline("text-classification", model=finetuned_ckpt)
```

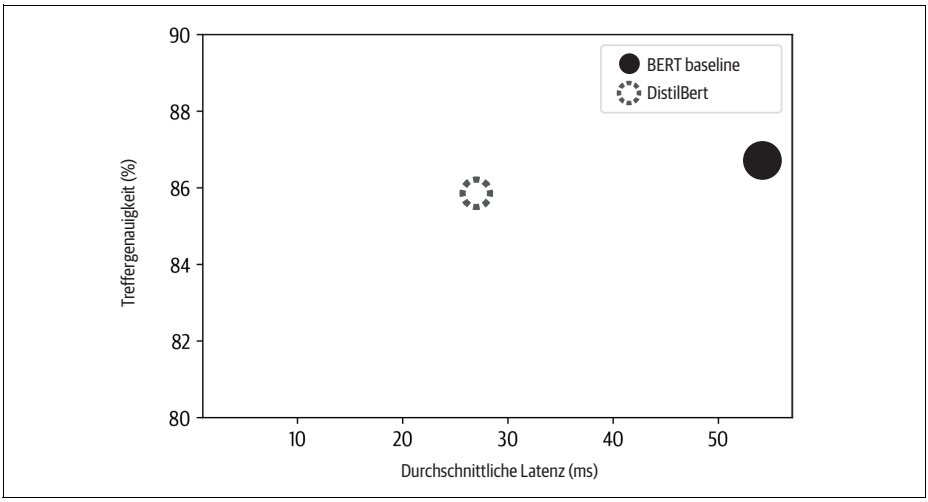
Anschließend können wir die Pipeline an unsere `PerformanceBenchmark`-Klasse übergeben, um die mit diesem Modell erzielten Maße bzw. Kennzahlen zu berechnen:

```
optim_type = "DistilBERT"
pb = PerformanceBenchmark(pipe, clinc["test"], optim_type=optim_type)
perf_metrics.update(pb.run_benchmark())
```

```
Modellgröße (MB) - 255.89
Durchschnittliche Latenz (ms) - 27.53 +\- 0.60
Treffergenauigkeit auf Testdatensatz - 0.858
```

Vergleichen wir diese Ergebnisse mit unserem Baseline-Modell, indem wir ein Streudiagramm erstellen, in dem die Treffergenauigkeit der Latenz gegenübergestellt wird, wobei die Größe der einzelnen Punkte die Größe des Modells auf der Festplatte widerspiegelt. Genau dazu dient die folgende Funktion, die zudem den jeweils aktuellen Optimierungsansatz mit einem gestrichelten Kreis hervorhebt, um den (späteren) Vergleich mit den vorherigen Ergebnissen zu erleichtern:

```
import pandas as pd

def plot_metrics(perf_metrics, current_optim_type):
    df = pd.DataFrame.from_dict(perf_metrics, orient='index')

    for idx in df.index:
        df_opt = df.loc[idx]
        # Gestrichelten Kreis um den aktuellen Optimierungsansatz erzeugen
        if idx == current_optim_type:
            plt.scatter(df_opt["time_avg_ms"], df_opt["accuracy"] * 100,
                        alpha=0.5, s=df_opt["size_mb"], label=idx,
                        marker='$\u25CC$')
        else:
            plt.scatter(df_opt["time_avg_ms"], df_opt["accuracy"] * 100,
                        s=df_opt["size_mb"], label=idx, alpha=0.5)

    legend = plt.legend(bbox_to_anchor=(1,1))
    for handle in legend.legendHandles:
        handle.set_sizes([20])

    plt.ylim(80,90)
    # Langsamstes Modell hernehmen, um Wertebereich der x-Achse zu definieren
    xlim = int(perf_metrics["BERT-baseline"]["time_avg_ms"] + 3)
    plt.xlim(1, xlim)
    plt.ylabel("Treffergenauigkeit (%)")
    plt.xlabel("Durchschnittliche Latenz (ms)")
    plt.show()

plot_metrics(perf_metrics, optim_type)
```



Aus dem Diagramm können wir entnehmen, dass wir die durchschnittliche Latenzzeit durch die Verwendung eines kleineren Modells erheblich verringern konnten. Der Preis dafür war eine Verringerung der Treffergenauigkeit um lediglich etwas mehr als ein Prozent! Mal sehen, ob wir diese Diskrepanz noch schließen können, indem wir zusätzlich den Destillationsverlust des Lehrer-Modells berücksichtigen und außerdem bessere Werte für α und T finden.

Geeignete Hyperparameter mit Optuna finden

Um nach geeigneten Werten für α und T zu suchen, könnten Sie eine Rastersuche (engl. Grid Search) über den zweidimensionalen Parameterraum durchführen. Eine wesentlich bessere Alternative ist jedoch die Verwendung von *Optuna*[12], ein Optimierungsframework, das genau für diese Art von Aufgaben entwickelt wurde. Optuna formuliert das Suchproblem in Form einer Zielfunktion, die im Rahmen mehrerer *Versuche* (engl. Trials) optimiert wird. Nehmen wir zum Beispiel an, wir wollten Rosenbrocks »Bananenfunktion« (*https://oreil.ly/hPk8h*) minimieren:

$$f(x, y) = (1 - x)^2 + 100\left(y - x^2\right)^2$$

die einen berühmten Testfall für Optimierungs-Frameworks darstellt. Wie in Abbildung 8-5 gezeigt, verdankt die Funktion ihren Namen den gekrümmten Konturen bzw. Höhenlinien und weist ein globales Minimum bei $(x, y)=(1, 1)$ auf. Das Tal bzw. den Tiefpunkt der Funktion zu finden, stellt ein einfaches Optimierungsproblem dar, nicht aber, sich an das globale Minimum anzunähern.

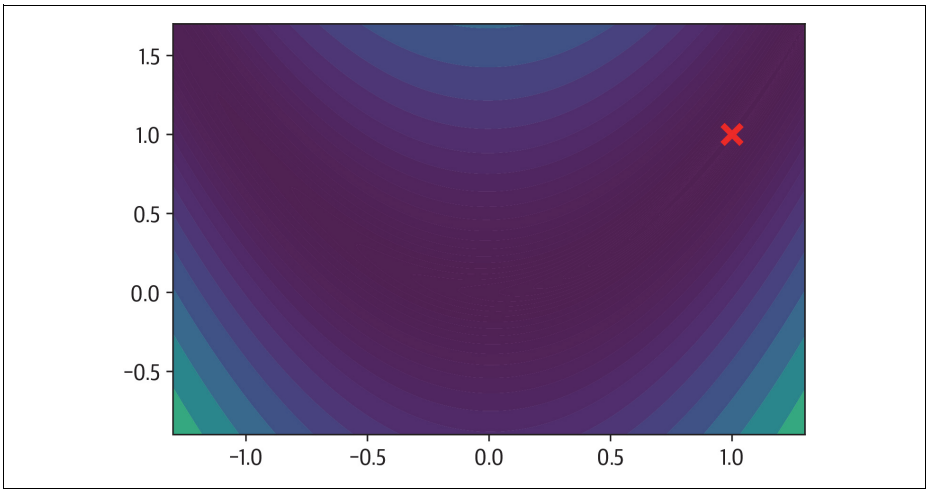


Abbildung 8-5: Visualisierung der Rosenbrock-Funktion für den Fall mit zwei Variablen

12 T. Akiba et al., »Optuna: A Next-Generation Hyperparameter Optimization Framework« (*https://arxiv.org/abs/1907.10902*), (2019).

In Optuna können wir das Minimum einer Funktion $f(x, y)$ bestimmen, indem wir eine Ziel- bzw. Optimierungsfunktion, `objective()`, definieren, die den entsprechenden Wert für $f(x, y)$ liefert:

```
def objective(trial):
    x = trial.suggest_float("x", -2, 2)
    y = trial.suggest_float("y", -2, 2)
    return (1 - x) ** 2 + 100 * (y - x ** 2) ** 2
```

In den `trial.suggest_float`-Objekten werden die Parameterbereiche spezifiziert, aus denen jeweils ein Wert aus einer Gleichverteilung gezogen werden soll. Optuna bietet auch eine `suggest_int`- und eine `suggest_categorical`-Methode für ganzzahlige bzw. kategoriale Parameter. In Optuna werden mehrere Versuche (Trials) als *Study* bzw. Studie zusammengefasst. Zur Erstellung einer Study müssen Sie lediglich die `objective()`-Funktion an die `study.optimize()`-Methode übergeben:

```
import optuna

study = optuna.create_study()
study.optimize(objective, n_trials=1000)
```

Sobald die Study abgeschlossen ist, können wir die besten Parameter ermitteln:

```
study.best_params
```

```
{'x': 1.003024865971437, 'y': 1.00315167589307}
```

Wie Sie sehen, ist es Optuna gelungen, innerhalb von eintausend Versuchen Werte für x und y zu finden, die recht nah am globalen Minimum liegen. Wenn wir Optuna in der 🤗 Transformers-Bibliothek verwenden möchten, müssen wir ähnlich vorgehen und zunächst den Hyperparameterraum definieren, der zur Optimierung einbezogen werden soll. Zusätzlich zu α und T geben wir noch die Anzahl der Trainingsepochen an:

```
def hp_space(trial):
    return {"num_train_epochs": trial.suggest_int("num_train_epochs", 5, 10),
        "alpha": trial.suggest_float("alpha", 0, 1),
        "temperature": trial.suggest_int("temperature", 2, 20)}
```

Die Hyperparametersuche mit dem `Trainer` durchzuführen, ist dann ganz einfach. Wir müssen lediglich die Anzahl der durchzuführenden Versuche und das Ziel der Optimierung (`direction`) angeben. Da wir die bestmögliche Treffergenauigkeit erzielen möchten, geben wir in der Methode `hyperparameter_search()` des Trainers `direction="maximize"` an und übergeben den Hyperparameterraum, der in die Suche einbezogen werden soll:

```
best_run = distilbert_trainer.hyperparameter_search(
    n_trials=20, direction="maximize", hp_space=hp_space)
```

Die Methode `hyperparameter_search()` gibt ein `BestRun`-Objekt zurück, das den Wert der Zielfunktion, die maximiert wurde (als Voreinstellung die Summe aller Maße) und die Hyperparameter, die für diesen Lauf verwendet wurden, enthält:

```
print(best_run)
```

```
BestRun(run_id='1', objective=0.927741935483871,
hyperparameters={'num_train_epochs': 10, 'alpha': 0.12468168730193585,
'temperature': 7})
```

Dieser Wert für α lässt darauf schließen, dass der größte Teil des Trainingssignals dem Term, der auf der Knowledge Distillation beruht, zuzurechnen ist. Diese Werte können wir nun zur Aktualisierung unserer Trainingsargumente verwenden und den letzten Trainingslauf durchführen:

```
for k,v in best_run.hyperparameters.items():
    setattr(student_training_args, k, v)

# Neues Repository einrichten, um destilliertes Modell zu speichern
distilled_ckpt = "distilbert-base-uncased-distilled-clinc"
student_training_args.output_dir = distilled_ckpt

# Neuen Trainer mit optimalen Parametern erstellen
distil_trainer = DistillationTrainer(model_init=student_init,
    teacher_model=teacher_model, args=student_training_args,
    train_dataset=clinc_enc['train'], eval_dataset=clinc_enc['validation'],
    compute_metrics=compute_metrics, tokenizer=student_tokenizer)

distil_trainer.train();
```

Epoche	Trainingsverlust	Validierungsverlust	Treffergenauigkeit
1	0.9031	0.574540	0.736452
2	0.4481	0.285621	0.874839
3	0.2528	0.179766	0.918710
4	0.1760	0.139828	0.929355
5	0.1416	0.121053	0.934839
6	0.1243	0.111640	0.934839
7	0.1133	0.106174	0.937742
8	0.1075	0.103526	0.938710
9	0.1039	0.101432	0.938065
10	0.1018	0.100493	0.939355

Obwohl es nur rund halb so viele Parameter umfasst, ist es uns bemerkenswerterweise gelungen, das Schüler-Modell so zu trainieren, dass es nun sogar die Treffergenauigkeit des Lehrer-Modells erreicht. Übertragen wir das Modell noch auf den Hub, um es später wiederverwenden zu können:

```
distil_trainer.push_to_hub("Training complete")
```

Unser destilliertes Modell im Vergleich

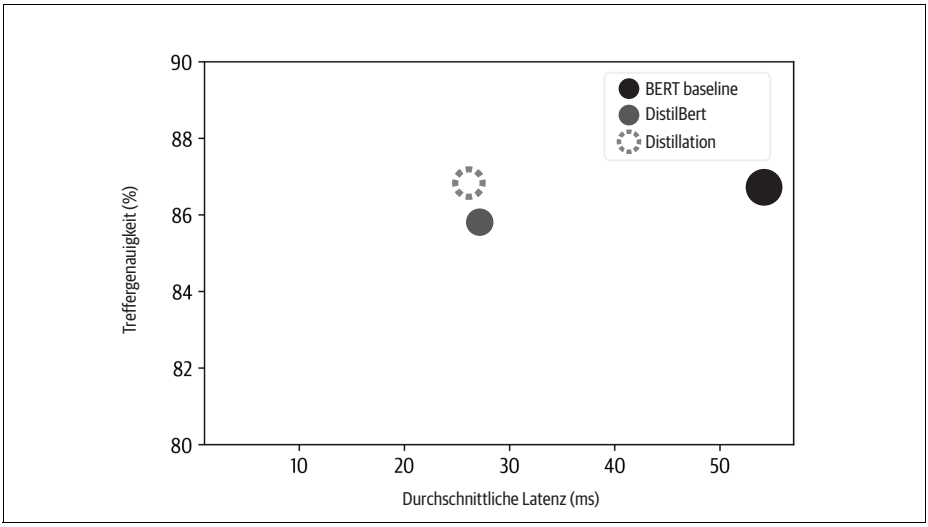
Mittlerweile haben wir ein relativ akkurates Schüler-Modell entwickelt. Wir können nun erneut eine Pipeline erstellen und einen Vergleich anhand unserer Performance-Benchmark anstellen (wobei wir wie zuvor den Testdatensatz heranziehen):

```
distilled_ckpt = "transformersbook/distilbert-base-uncased-distilled-clinc"
pipe = pipeline("text-classification", model=distilled_ckpt)
optim_type = "Distillation"
pb = PerformanceBenchmark(pipe, clinc["test"], optim_type=optim_type)
perf_metrics.update(pb.run_benchmark())
```

```
Modellgröße (MB) - 255.89
Durchsschnittliche Latenz (ms) - 25.96 +\- 1.63
Treffergenauigkeit auf Testdatensatz - 0.868
```

Lassen Sie uns jetzt die Ergebnisse mithilfe unserer `plot_metrics()`-Funktion visualisieren, um sie besser mit den vorherigen Ergebnissen vergleichen zu können:

```
plot_metrics(perf_metrics, optim_type)
```



Wie zu erwarten war, bleiben sowohl die Modellgröße als auch die Latenzzeit im Vergleich zur DistilBERT-Benchmark im Wesentlichen unverändert – allerdings hat sich die Treffergenauigkeit verbessert und übertrifft nun sogar die des Lehrer-Modells. Eine Möglichkeit, dieses doch recht überraschende Ergebnis zu interpretieren, ist, dass das Lehrer-Modell wahrscheinlich nicht auf solch systematische Weise feingetunt wurde wie das Schüler-Modell. Damit haben wir bereits ein hervorragendes Ergebnis erzielt! Doch wir können unser destilliertes Modell noch weiter komprimieren, indem wir ein Verfahren anwenden, das als Quantisierung (engl. Quantization) bekannt, und Gegenstand des nächsten Abschnitts ist.

Beschleunigung von Modellen mithilfe der Quantisierung

Wir haben inzwischen festgestellt, dass wir mithilfe der Knowledge Distillation die Rechen- und Speicherkosten des Inferenzvorgangs verringern können, indem wir die Informationen von einem Lehrer- auf ein kleineres Schüler-Modell übertragen.

Bei der Quantisierung wird ein anderer Ansatz verfolgt. Anstatt die Anzahl der Berechnungen zu reduzieren, zielt sie darauf ab, diese deutlich effizienter zu gestalten, indem die Gewichte und Aktivierungen mit Datentypen dargestellt werden, die eine geringe numerische Genauigkeit (engl. Precision) aufweisen, wie z. B. 8-Bit-Ganzzahlen (bzw. 8-Bit-Integers) (INT8) anstelle der üblichen 32-Bit-Gleitkommazahlen (FP32, wobei FP für Floating Point steht). Eine geringere Anzahl von Bits zu verwenden, führt dazu, dass das resultierende Modell weniger Speicherplatz benötigt und Operationen wie die Matrixmultiplikation wesentlich schneller ausgeführt werden können, da sie auf Basis der Ganzzahlarithmetik durchgeführt werden. Bemerkenswerterweise können diese Performance-Steigerungen erzielt werden, ohne dass die Treffergenauigkeit darunter leidet!

Eine Einführung in Gleitkomma- und Festkommazahlen

Die meisten Transformer werden heute in Gleitkommadarstellung bzw. mit Gleitkommazahlen (engl. Floating Point Numbers) (in der Regel FP32 oder einer Mischung aus FP16 und FP32) vortrainiert und feingetunt, da sie die nötige numerische Genauigkeit bieten, um die sehr unterschiedlichen Wertebereiche von Gewichten, Aktivierungen und Gradienten zu verarbeiten. Eine Gleitkommazahl wie FP32 stellt eine Folge von 32 Bits dar, die in Form eines *Vorzeichens* (engl. Sign), eines *Exponenten* (engl. Exponent) und einer *Mantisse* (engl. Significand) angeordnet sind. Das Vorzeichen bestimmt, ob die Zahl positiv oder negativ ist, während die Mantisse der Anzahl der signifikanten Ziffern entspricht, die mit dem Exponenten mit einer gegebenen Basis skaliert werden (normalerweise 2 für Binär- oder 10 für Dezimalzahlen).

Die Zahl 137,035 kann zum Beispiel durch die folgende Formel als dezimale Gleitkommazahl ausgedrückt werden:

$$137{,}035 = (-1)^0 \times 1{,}37035 \times 10^2$$

wobei 1,37035 die Mantisse und 2 den Exponent zur Basis 10 darstellt. Der Exponent ermöglicht die Darstellung eines breiten Spektrums reeller Zahlen, und das Dezimaltrennzeichen kann an einer beliebigen Stelle im Verhältnis zu den signifikanten Ziffern platziert werden (daher der Name »Gleit- bzw. Fließkomma«).

Wenn ein Modell jedoch erst einmal trainiert ist, ist nur der Forward-Pass erforderlich, um die Inferenz durchzuführen. Dementsprechend können wir die numerische Genauigkeit der Datentypen verringern, ohne die Treffergenauigkeit des Modells allzu stark zu beeinträchtigen. Bei neuronalen Netzen ist es üblich, für die Datentypen, die eine geringe numerische Genauigkeit besitzen, ein *Festkommaformat* (engl. Fixed Point Format) zu verwenden, bei dem reelle Zahlen als *B*-Bit-Ganzzahlen, die für alle Variablen desselben Typs mit einem einheitlichen Faktor skaliert werden, dargestellt werden. Zum Beispiel kann 137,035 als die Ganzzahl 137.035 dargestellt werden, die mit 1/1.000 skaliert wurde. Über den Skalierungsfaktor können wir den Wertebereich und die numerische Genauigkeit einer Festkommazahl steuern.

Die Grundidee hinter der Quantisierung ist, dass wir die Gleitkommawerte f in jedem Tensor »diskretisieren« können, indem wir ihren Wertebereich $[f_{max}, f_{min}]$ in einen kleineren Wertebereich $[q_{max}, q_{min}]$ von Festkommazahlen q überführen und die in dem Intervall liegenden Werte gleichmäßig verteilen. Mathematisch wird diese Abbildung durch die folgende Gleichung beschrieben:

$$f = \left(\frac{f_{\max} - f_{\min}}{q_{\max} - q_{\min}}\right)(q - Z) = S(q - Z)$$

wobei der Skalierungsfaktor S einer positiven Gleitkommazahl entspricht. Die Konstante Z hat den gleichen Typ wie q und wird als *Nullstelle* (engl. Zero Point) bezeichnet, da sie dem quantisierten Wert der Gleitkommazahl $f = 0$ entspricht. Beachten Sie, dass die Abbildung *affin* sein muss, damit wir wieder Gleitkommazahlen erhalten, wenn wir die Festkommazahlen entquantisieren.[13] Abbildung 8-6 veranschaulicht die Umrechnung.

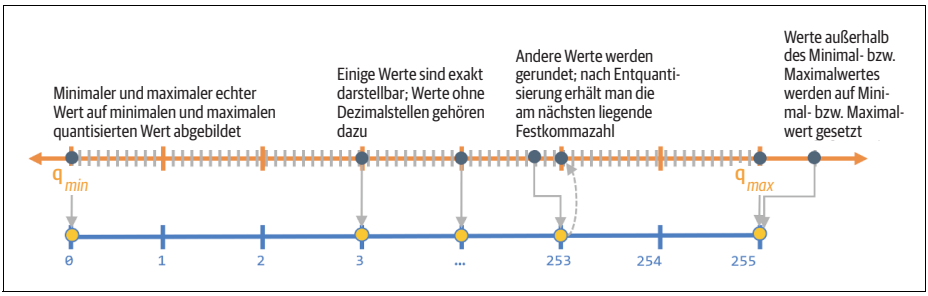
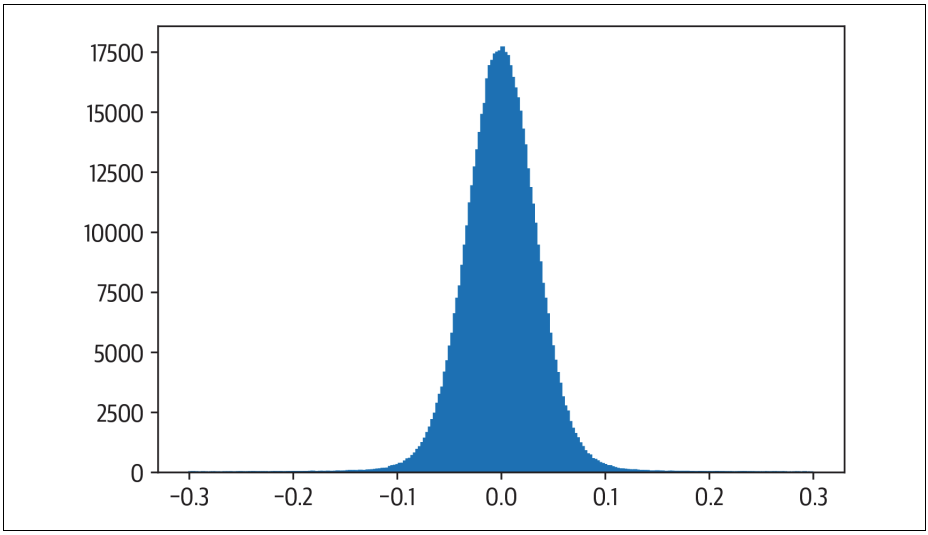


Abbildung 8-6: Quantisierung von Gleitkommazahlen als vorzeichenlose 8-Bit-Ganzzahlen (mit freundlicher Genehmigung von Manas Sahni)

Einer der Hauptgründe, warum Transformer-Modelle (und tiefe neuronale Netze im Allgemeinen) sich hervorragend für die Quantisierung eignen, ist, dass die Gewichte und Aktivierungen in der Regel Werte annehmen, die einen relativ kleinen Wertebereich abdecken. Das bedeutet, dass wir nicht den vollständigen Wertebereich möglicher FP32-Zahlen in die durch INT8 dargestellten $2^8 = 256$ Zahlen »quetschen« müssen. Um das zu verdeutlichen, können wir eine der Matrizen mit den Attention-Gewichten aus unserem destillierten Modell heranziehen und die Häufigkeitsverteilung der Werte visualisieren:

```
import matplotlib.pyplot as plt

state_dict = pipe.model.state_dict()
weights = state_dict["distilbert.transformer.layer.0.attention.out_lin.weight"]
plt.hist(weights.flatten().numpy(), bins=250, range=(-0.3,0.3), edgecolor="C0")
plt.show()
```

13 Eine affine Abbildung ist nur ein ausgefallener Name für die Abbildung $y = \Delta x + b$, die Sie von den linearen Schichten eines neuronalen Netzes kennen.



Wie wir sehen können, sind die Werte der Gewichte innerhalb des kleinen Intervalls [–0,1, 0,1] um null herum verteilt. Nehmen wir nun an, wir möchten diesen Tensor als vorzeichenbehaftete 8-Bit-Ganzzahl quantisieren. In diesem Fall erstreckt sich der mögliche Wertebereich für unsere Ganzzahlen auf das Intervall $[q_{max}, q_{min}] = [-128,127]$. Die Nullstelle fällt mit der Nullstelle des FP32-Datentyps zusammen und der Skalierungsfaktor wird gemäß der vorherigen Gleichung berechnet:

```
zero_point = 0
scale = (weights.max() - weights.min()) / (127 - (-128))
```

Zur Ermittlung des quantisierten Tensors müssen wir lediglich die Abbildung $q = f / S + Z$ invertieren, die Werte eingrenzen, sie auf die nächste Ganzzahl runden und das Ergebnis mithilfe der Funktion `Tensor.char()` als `torch.int8`-Datentyp darstellen:

```
(weights / scale + zero_point).clamp(-128, 127).round().char()
```

```
tensor([[ -5,  -8,   0,  ...,  -6,  -4,   8],
        [  8,   3,   1,  ...,  -4,   7,   0],
        [ -9,  -6,   5,  ...,   1,   5,  -3],
        ...,
        [  6,   0,  12,  ...,   0,   6,  -1],
        [  0,  -2, -12,  ...,  12,  -7, -13],
        [-13,  -1, -10,  ...,   8,   2,  -2]], dtype=torch.int8)
```

Sehr gut, wir haben soeben unseren ersten Tensor quantisiert! In PyTorch können wir die Umrechnung vereinfachen, indem wir die Funktion `quantize_per_tensor()` zusammen mit einem quantisierten Datentyp, `torch.qint`, verwenden, der für arithmetische Operationen mit Ganzzahlen optimiert ist:

```
from torch import quantize_per_tensor

dtype = torch.qint8
quantized_weights = quantize_per_tensor(weights, scale, zero_point, dtype)
quantized_weights.int_repr()
```

```
tensor([[ -5,  -8,   0,  ...,  -6,  -4,   8],
        [  8,   3,   1,  ...,  -4,   7,   0],
        [ -9,  -6,   5,  ...,   1,   5,  -3],
        ...,
        [  6,   0,  12,  ...,   0,   6,  -1],
        [  0,  -2, -12,  ...,  12,  -7, -13],
        [-13,  -1, -10,  ...,   8,   2,  -2]], dtype=torch.int8)
```

In dem Diagramm in Abbildung 8-7 können Sie deutlich die Diskretisierung der Werte erkennen, die dadurch entsteht, dass nur ein Teil der Werte der Gewichte exakt abgebildet wird und die restlichen gerundet werden.

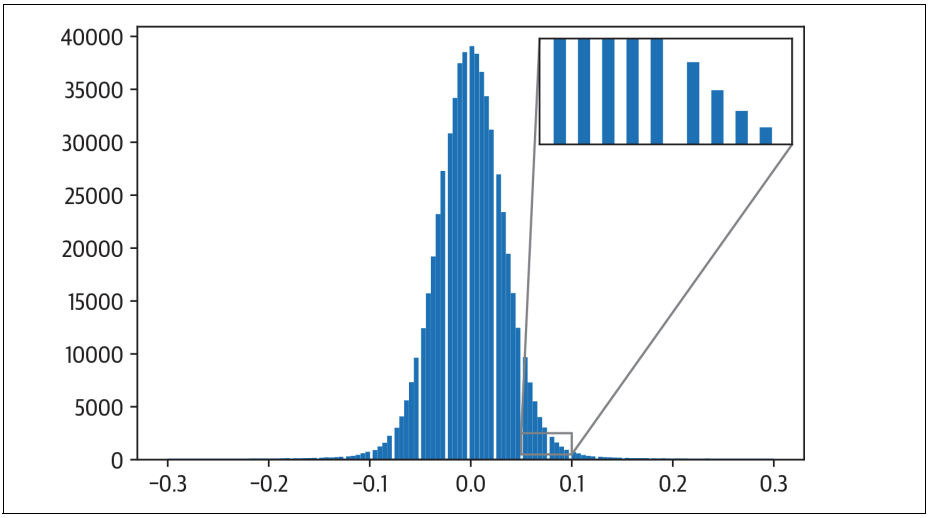


Abbildung 8-7: Auswirkung der Quantisierung auf die Gewichte eines Transformer-Modells

Um nun unsere kleine Analyse abzurunden, können wir vergleichen, wie lange es dauert, die Multiplikation zweier Gewichtungs-Tensoren entweder mit FP32- oder mit INT8-Werten zu berechnen. Die im FP32-Datentyp gehaltenen Tensoren können wir in PyTorch mit dem eleganten `@`-Operator multiplizieren:

```
%%timeit
weights @ weights
```

```
393 µs ± 3.84 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Für die quantisierten Tensoren benötigen wir die `QFunctional`-Klasse, da wir die Operationen mit dem speziellen Datentyp `torch.qint8` durchführen wollen:

```
from torch.nn.quantized import QFunctional

q_fn = QFunctional()
```

Die Klasse unterstützt verschiedene elementare Operationen wie etwa die Addition. Im vorliegenden Fall können wir die (durchschnittliche) Ausführungszeit für die Multiplikation unserer quantisierten Tensoren wie folgt ermitteln:

```
%%timeit
q_fn.mul(quantized_weights, quantized_weights)
```

```
23.3 µs ± 298 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Im Vergleich zu unseren mit dem FP32-Datentyp durchgeführten Berechnungen werden die Berechnungen mit den im INT8-Format dargestellten Tensoren fast 100-mal so schnell durchgeführt! Sie können sogar noch mehr herausholen, wenn Sie spezielle Backends verwenden, die quantisierte Operatoren auf effiziente Weise ausführen. Zum Zeitpunkt der Erstellung dieses Buchs unterstützt PyTorch:

- x86-CPUs mit AVX2-Unterstützung oder höher
- ARM-CPUs (typischerweise in mobilen bzw. Embedded-Geräten zu finden)

Da Zahlen im INT8-Format viermal weniger Bits haben als im FP32-Format, verringert die Quantisierung auch den Speicherbedarf um bis zu einem Faktor von Vier. Im Rahmen unseres einfachen Beispiels lässt sich das leicht nachprüfen, indem wir die zugrunde liegende Speichergröße unseres Gewichtungs-Tensors und seines quantisierten Pendants mithilfe der `Tensor.storage()`- und der `getsizeof()`-Funktion aus dem Python-Modul `sys` vergleichen:

```
import sys

sys.getsizeof(weights.storage()) / sys.getsizeof(quantized_weights.storage())
```

```
3.999633833760527
```

Bei einem vollwertigen Transformer-Modell würde die tatsächliche Kompressionsrate davon abhängen, welche Schichten quantisiert werden (wie Sie im nächsten Abschnitt erfahren werden, werden normalerweise nur die linearen Schichten quantisiert).

Wo liegt also der Haken bei der Quantisierung? Die Änderung der numerischen Genauigkeit aller Berechnungen in unserem Modell führt zu kleinen Abweichungen an jedem Punkt im Berechnungsgraphen des Modells, die sich kumulieren und die Leistung des Modells beeinträchtigen können. Es gibt mehrere Möglichkeiten, wie Sie ein Modell quantisieren können, wobei sie alle ihre Vor- und Nachteile haben. Im Hinblick auf tiefe neuronale Netze gibt es in der Regel drei Hauptansätze für die Quantisierung:

Dynamische Quantisierung

Bei der dynamischen Quantisierung bleibt im Rahmen des Trainings alles unverändert, und die Anpassungen werden erst im Rahmen der Inferenz vorgenommen. Wie bei allen Quantisierungsverfahren, die wir behandeln werden, werden die Gewichte des Modells vor der Inferenz in das INT8-Format umgewandelt. Zusätzlich zu den Gewichten werden auch die Aktivierungen des Modells quantisiert. Dieser Ansatz ist dynamisch, denn die Quantisierung erfolgt on the fly. Das bedeutet, dass alle Matrixmultiplikationen mit hochoptimierten INT8-Funktionen berechnet werden können. Von allen hier besprochenen Quantisierungsverfahren ist die dynamische Quantisierung die einfachste. Al-

lerdings werden bei der dynamischen Quantisierung die Aktivierungen im Gleitkommaformat in den Speicher geschrieben und aus ihm gelesen. Diese erforderliche Umrechnung zwischen Ganz- und Fließkommazahlen kann zu einem Perfomance-Engpass werden.

Statische Quantisierung
: Anstatt die Quantisierung der Aktivierungen unmittelbar bzw. on the fly vorzunehmen, können wir die Umwandlung in Gleitkommazahlen vermeiden, indem wir das Quantisierungsschema vorberechnen. Hierzu werden bei der statischen Quantisierung die Aktivierungsmuster anhand einer repräsentativen Stichprobe der Daten vor der Durchführung der Inferenz festgehalten. Darauf basierend wird dann das ideale Quantisierungsschema ermittelt und gespeichert. Dadurch können wir die Umwandlung zwischen INT8- und FP32-formatierten Werten umgehen und die Berechnungen beschleunigen. Allerdings erfordert dies, auf eine Stichprobe der Daten zurückgreifen zu können, die repräsentativ ist. Zudem führt es einen zusätzlichen Schritt in der Pipeline ein, da wir nun das Quantisierungsschema trainieren und bestimmen müssen, bevor wir die Inferenz durchführen können. Darüber hinaus gibt es einen Aspekt, den die statische Quantisierung nicht berücksichtigt: die Diskrepanz zwischen der numerischen Genauigkeit beim Training und der bei der Inferenz, die zu einem Rückgang der Güte- bzw. Qualitätsmaße des Modells (z. B. der Treffergenauigkeit) führt.

Quantization-aware Training
: Der Effekt der Quantisierung kann während des Trainings durch eine »unechte« Quantisierung der im FP32-Format dargestellten Werte auf effektive Weise simuliert werden. Anstatt INT8-Werte während des Trainings zu verwenden, werden die FP32-Werte gerundet, um so den Effekt der Quantisierung zu imitieren. Dies betrifft sowohl den Forward- als auch den Backward-Pass und führt dazu, dass sich die Güte bzw. Qualität des Modells – verglichen mit der statischen und dynamischen Quantisierung – verbessert.

Der wesentliche Knackpunkt, der bei der Inferenz mit Transformer-Modellen auftritt, ist die erforderliche Rechen- und Speicherbandbreite, die mit der enormen Anzahl an Gewichten in diesen Modellen verbunden ist. Aus diesem Grund ist die dynamische Quantisierung derzeit der beste Ansatz für Transformer-Modelle im Bereich des NLP. Bei kleineren Computer-Vision-Modellen ist die Speicherbandbreite der Aktivierungen der restringierende Faktor, weshalb in der Regel statische Quantisierung verwendet wird (oder Quantization-aware Training in Fällen, in denen die Güte des Modells zu stark beeinträchtigt wird).

In PyTorch ist es relativ einfach, die dynamische Quantisierung zu implementieren, da sie mit einer einzigen Codezeile vorgenommen werden kann:

```
from torch.quantization import quantize_dynamic

model_ckpt = "transformersbook/distilbert-base-uncased-distilled-clinc"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
```

```
model = (AutoModelForSequenceClassification
         .from_pretrained(model_ckpt).to("cpu"))

model_quantized = quantize_dynamic(model, {nn.Linear}, dtype=torch.qint8)
```

Dabei übergeben wir der `quantize_dynamic()`-Methode das Modell mit voller numerischer Genauigkeit und geben die PyTorch-Klassen der Schichten des Modells an, die wir quantisieren möchten. Mit dem Argument `dtype` wird die numerische Genauigkeit der Zielvariablen angegeben, wobei als Wert entweder `fp16` oder `qint8` gewählt werden kann. Es empfiehlt sich, die niedrigste numerische Genauigkeit zu wählen, die Sie im Hinblick auf Ihre zur Evaluierung verwendeten Maße tolerieren können. In diesem Kapitel verwenden wir das INT8-Format, was, wie wir gleich sehen werden, nur geringe Auswirkungen auf die Treffergenauigkeit unseres Modells hat.

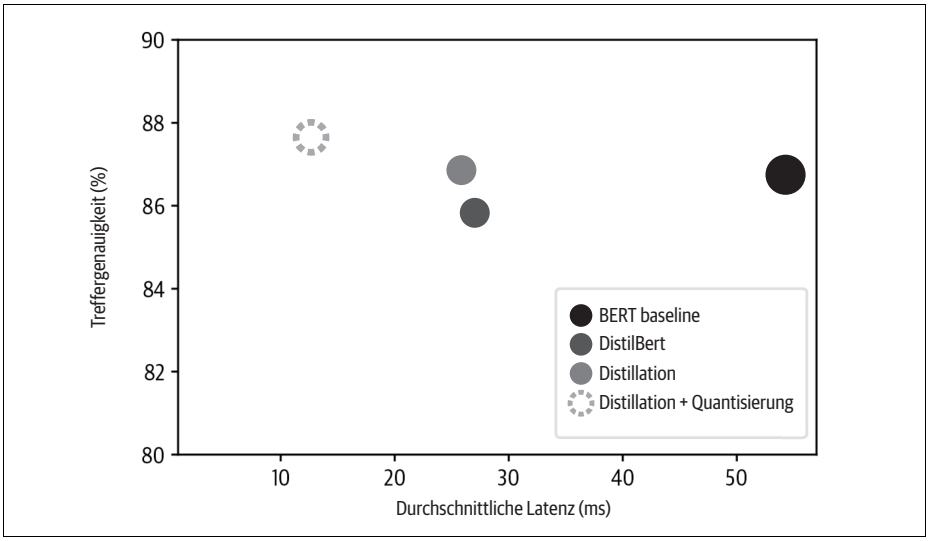
Das quantisierte Modell im Vergleich

Da unser Modell nun quantisiert ist, ist es an der Zeit, es mit den anderen Modellen zu vergleichen und die Ergebnisse zu visualisieren:

```
pipe = pipeline("text-classification", model=model_quantized,
                tokenizer=tokenizer)
optim_type = "Distillation + Quantisierung"
pb = PerformanceBenchmark(pipe, clinc["test"], optim_type=optim_type)
perf_metrics.update(pb.run_benchmark())
```

```
Modelgröße (MB) - 132.40
Durchschnittliche Latenz (ms) - 12.54 +\- 0.73
Treffergenauigkeit auf Testdatensatz - 0.876
```

```
plot_metrics(perf_metrics, optim_type)
```



Prima, das quantisierte Modell ist nur etwa halb so groß wie unser destilliertes Modell – und schneidet hinsichtlich der Treffergenauigkeit sogar noch besser ab! Mal sehen, ob wir unsere Optimierung mit einem leistungsstarken Framework namens ONNX Runtime noch weiter ausreizen können.

Optimierung der Inferenz mit ONNX und der ONNX Runtime

ONNX (*https://onnx.ai*) ist ein offener Standard, der einen gemeinsamen Satz von Operatoren und ein gemeinsames Dateiformat zur Darstellung von Deep-Learning-Modellen in einer Vielzahl von Frameworks, einschließlich PyTorch und TensorFlow, definiert.[14] Wenn ein Modell in das ONNX-Format exportiert wird, werden diese Operatoren verwendet, um einen Berechnungsgraphen (oft als *Zwischendarstellung*, engl. Intermediate Representation, bezeichnet) zu erstellen, der den Datenfluss durch das neuronale Netz abbildet. Ein Beispiel für einen solchen Graphen für das BERT-base-Modell finden Sie in Abbildung 8-8, in dem jeder Knoten eine Eingabe erhält, eine Operation wie `Add` oder `Squeeze` ausführt und dann die Ausgabe an die nächsten Knoten weiterleitet.

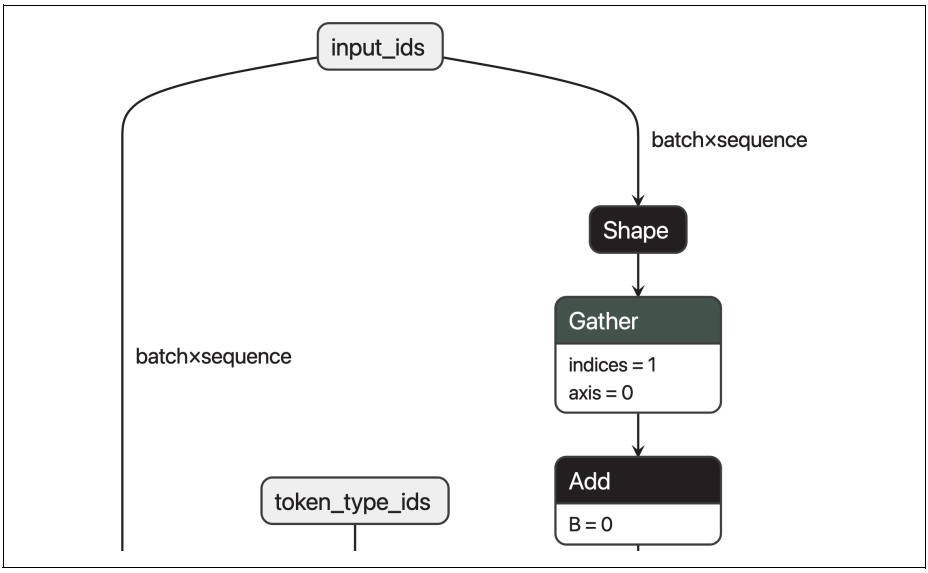


Abbildung 8-8: Ein Ausschnitt aus dem ONNX-Graphen für das BERT-base-Modell, visualisiert in Netron

Dadurch, dass ONNX einen Graphen mit standardisierten Operatoren und Datentypen bereitstellt, ist es einfach, zwischen den verschiedenen Frameworks zu wech-

14 Es gibt einen separaten Standard namens ONNX-ML, der für traditionelle Machine- Learning-Modelle wie Random Forests und Frameworks wie Scikit-learn entwickelt wurde.

seln. Zum Beispiel kann ein in PyTorch trainiertes Modell in das ONNX-Format exportiert und dann in TensorFlow importiert werden (und umgekehrt).

ONNX brilliert vor allem dann, wenn es mit einem speziellen Beschleuniger (bzw. Akzelerator, engl. Accelerator) wie der ONNX Runtime (*https://onnxruntime.ai*), kurz ORT, gekoppelt wird.[15] ORT bietet Tools zur Optimierung des ONNX-Graphen, die auf Methoden wie der Operator-Fusion und der Konstantenfaltung (engl. Constant Folding) basieren[16], und definiert eine Schnittstelle zu sogenannten *Execution Providern*, die es Ihnen ermöglichen, das Modell auf verschiedenen Arten von Hardware auszuführen. Kurzum, es bietet Ihnen eine sehr umfangreiche Möglichkeit zur Abstrahierung. Abbildung 8-9 gibt Ihnen einen Überblick über die grundlegende Architektur des ONNX- und ORT-Ökosystems.

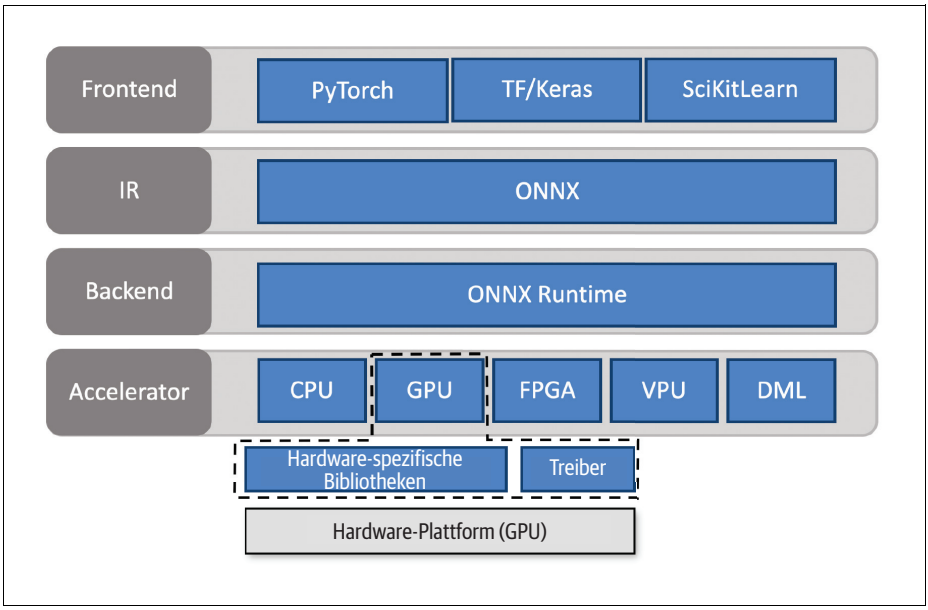


Abbildung 8-9: Die Architektur des ONNX- und ONNX-Runtime-Ökosystems (mit freundlicher Genehmigung des Teams von ONNX Runtime)

Um ORT nutzen zu können, müssen wir zunächst unser destilliertes Modell in das ONNX-Format konvertieren. Die 🤗 Transformers-Bibliothek verfügt über eine in-

15 Andere beliebte Beschleuniger sind NVIDIA's TensorRT (*https://oreil.ly/HnNZx*) und Apache TVM (*https://oreil.ly/7KUyt*).

16 Bei einer »fusionierten« Operation wird ein Operator (in der Regel eine Aktivierungsfunktion) mit einem anderen »verschmolzen«, sodass sie gemeinsam ausgeführt werden können. Nehmen wir zum Beispiel an, wir möchten eine Aktivierungsfunktion f auf ein Matrixprodukt $A \times B$ anwenden. Normalerweise muss das Ergebnis des Produkts in den GPU-Speicher zurückgeschrieben werden, bevor die Aktivierungsfunktion berechnet werden kann. Mit der Operator-Fusion ist es möglich, $f(A \times B)$ in einem einzigen Schritt zu berechnen. Die Konstantenfaltung bezieht sich darauf, dass der Prozess zur Auswertung konstanter Ausdrücke zum Zeitpunkt der Kompilierung stattfindet, und nicht zur Laufzeit.

tegrierte Funktion namens `convert_graph_to_onnx.convert()`, die den Vorgang vereinfacht, indem sie die folgenden Schritte vornimmt:

1. Das Modell als `Pipeline` initialisieren.
2. Eingaben in Form von Platzhaltern durch die Pipeline speisen, damit ONNX den Berechnungsgraphen erstellen und festhalten kann.
3. Dynamische Achsen definieren, damit variierende Sequenzlängen verarbeitet werden können.
4. Den Graphen samt Parametern des neuronalen Netzes speichern.

Damit wir die Funktion verwenden können, müssen wir zunächst einige für die Nutzung von OpenMP (*https://openmp.org*) erforderliche Umgebungsvariablen für ONNX setzen:

```
import os
from psutil import cpu_count

os.environ["OMP_NUM_THREADS"] = f"{cpu_count()}"
os.environ["OMP_WAIT_POLICY"] = "ACTIVE"
```

Bei OpenMP handelt es sich um eine API, die für die Entwicklung hochgradig paralleler Anwendungen entwickelt wurde. Mit der Umgebungsvariablen `OMP_NUM_THREADS` wird die Anzahl der Threads festgelegt, die für parallele Berechnungen in der ONNX Runtime verwendet werden sollen, während `OMP_WAIT_POLICY=ACTIVE` angibt, dass wartende Threads aktiv bleiben sollen (d.h. Prozessorzyklen der CPU in Anspruch nehmen).

Als Nächstes konvertieren wir unser destilliertes Modell in das ONNX-Format. Hier müssen wir für das Argument `pipeline_name="text-classification"` angeben, da die `convert()`-Funktion das Modell im Rahmen der Konvertierung in die `pipeline()`-Funktion der 🤗 Transformers-Bibliothek einbindet. Zur Initialisierung der Pipeline übergeben wir zusätzlich zum Checkpoint des Modells (`model_ckpt`) noch den Tokenizer:

```
from transformers.convert_graph_to_onnx import convert

model_ckpt = "transformersbook/distilbert-base-uncased-distilled-clinc"
onnx_model_path = Path("onnx/model.onnx")
convert(framework="pt", model=model_ckpt, tokenizer=tokenizer,
        output=onnx_model_path, opset=12, pipeline_name="text-classification")
```

ONNX verwendet sogenannte *Operator Sets*, mit denen unveränderliche Operator-Spezifikationen zusammengefasst werden. Dementsprechend wird mit `opset=12` eine bestimmte Version der ONNX-Bibliothek angegeben.

Da unser Modell nun gespeichert ist, benötigen wir nur noch eine Instanz der `InferenceSession`-Klasse, mit der die Eingaben in das Modell vorgenommen werden:

```
from onnxruntime import (GraphOptimizationLevel, InferenceSession,
                         SessionOptions)
```

```
def create_model_for_provider(model_path, provider="CPUExecutionProvider"):
    options = SessionOptions()
    options.intra_op_num_threads = 1
    options.graph_optimization_level = GraphOptimizationLevel.ORT_ENABLE_ALL
    session = InferenceSession(str(model_path), options, providers=[provider])
    session.disable_fallback()
    return session

onnx_model = create_model_for_provider(onnx_model_path)
```

Wenn wir nun die `onnx_model.run()`-Methode aufrufen, können wir die vom ONNX-Modell ermittelten Logits für alle Kategorien abrufen. Probieren wir es anhand eines Beispiels aus dem Testdatensatz aus. Da die Ausgabe der `convert()`-Funktion darauf schließen lässt, dass ONNX nur die `input_ids` und `attention_mask` als Eingaben erwartet, müssen wir die Spalte `label` aus unserem Beispiel löschen:

```
inputs = clinc_enc["test"][:1]
del inputs["labels"]
logits_onnx = onnx_model.run(None, inputs)[0]
logits_onnx.shape
```

```
(1, 151)
```

Sobald uns die Logits vorliegen, müssen wir lediglich die `argmax()`-Funktion (aus der Numpy-Bibliothek) anwenden, um das vorhergesagte Label zu erhalten:

```
np.argmax(logits_onnx)
```

```
61
```

das tatsächlich mit dem Ground-Truth-Label (d.h. der wahren zugrunde liegenden Kategorie) übereinstimmt:

```
clinc_enc["test"][0]["labels"]
```

```
61
```

Da das ONNX-Modell nicht mit der `text-classification`-Pipeline kompatibel ist, erstellen wir unsere eigene Klasse, die die grundlegende Funktionsweise nachbildet:

```
from scipy.special import softmax

class OnnxPipeline:
    def __init__(self, model, tokenizer):
        self.model = model
        self.tokenizer = tokenizer

    def __call__(self, query):
        model_inputs = self.tokenizer(query, return_tensors="pt")
        inputs_onnx = {k: v.cpu().detach().numpy()
                       for k, v in model_inputs.items()}
        logits = self.model.run(None, inputs_onnx)[0][0, :]
        probs = softmax(logits)
        pred_idx = np.argmax(probs).item()
        return [{"label": intents.int2str(pred_idx), "score": probs[pred_idx]}]
```

Anschließend können wir die Pipeline mit unserer einfachen Anfrage testen und uns vergewissern, ob wir als Intention erneut `car_rental` erhalten:

```
pipe = OnnxPipeline(onnx_model, tokenizer)
pipe(query)
```

```
[{'label': 'car_rental', 'score': 0.7848334}]
```

Prima, unsere Pipeline funktioniert wie erwartet! Der nächste Schritt besteht darin, die Performance der ONNX-Modelle untereinander und mit der der anderen zu vergleichen. Da wir bereits die `PerformanceBenchmark`-Klasse erstellt haben, können wir uns Arbeit ersparen und diese einfach übernehmen. Allerdings überschreiben wir die `compute_size()`-Methode, die Methoden `compute_accuracy()` und `time_pipeline()` behalten wir hingegen bei. Der Grund, warum wir die Methode `compute_size()` überschreiben müssen, ist, dass wir uns nicht auf das `state_dict`-Dictionary und die `torch.save()`-Methode stützen können, um die Größe eines Modells zu ermitteln, da `onnx_model` technisch gesehen ein ONNX-`InferenceSession`-Objekt ist, das nicht auf die Attribute von PyTorch's `nn.Module`-Klasse zugreifen kann. Der alternative Ansatz, den wir verfolgen, ist dennoch einfach und kann wie folgt implementiert werden:

```
class OnnxPerformanceBenchmark(PerformanceBenchmark):
    def __init__(self, *args, model_path, **kwargs):
        super().__init__(*args, **kwargs)
        self.model_path = model_path

    def compute_size(self):
        size_mb = Path(self.model_path).stat().st_size / (1024 * 1024)
        print(f"Modellgröße (MB) - {size_mb:.2f}")
        return {"size_mb": size_mb}
```

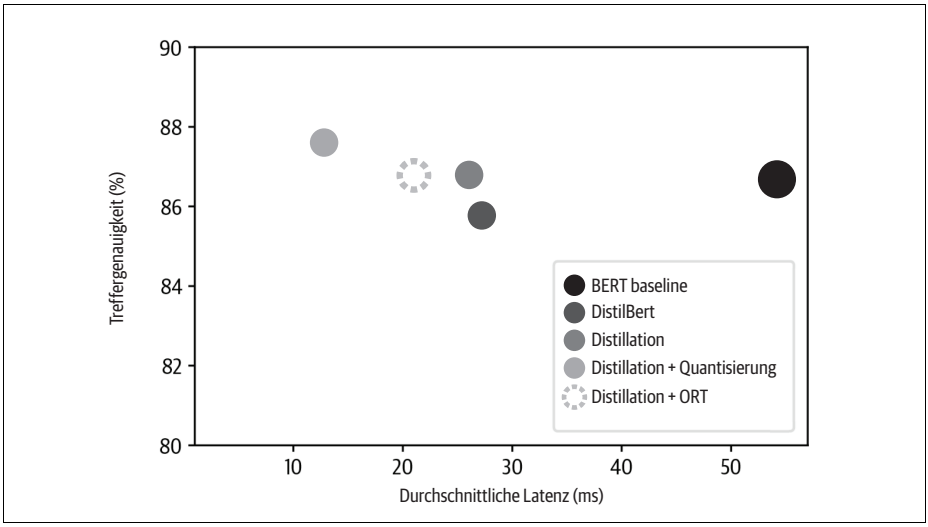
Verwenden wir nun unsere neu erstellte Benchmark-Klasse, um in Erfahrung zu bringen, wie unser destilliertes Modell abschneidet, nachdem es in das ONNX-Format konvertiert wurde:

```
optim_type = "Distillation + ORT"
pb = OnnxPerformanceBenchmark(pipe, clinc["test"], optim_type,
                              model_path="onnx/model.onnx")
perf_metrics.update(pb.run_benchmark())
```

```
Modellgröße (MB) - 255.88
Durchschnittliche Latenz (ms) - 21.02 +\- 0.55
Treffergenauigkeit auf Testdatensatz - 0.868
```

```
plot_metrics(perf_metrics, optim_type)
```

Bemerkenswerterweise haben die Konvertierung in das ONNX-Format und die Verwendung der ONNX Runtime bewirkt, dass sich die Latenzzeit unseres (ursprünglichen) destillierten Modells (als Distillation im Diagramm gekennzeichnet) deutlich verbessert hat. Mal sehen, ob wir noch etwas mehr an Performance herausholen können, wenn wir das Modell zusätzlich noch quantisieren.



Ähnlich wie bei PyTorch gibt es bei der ORT drei Möglichkeiten, ein Modell zu quantisieren: dynamische und statische Quantisierung und auch Quantization-aware Training. Wie bereits zuvor in PyTorch unterziehen wir unser destilliertes Modell einer dynamischen Quantisierung. In der ORT erfolgt die Quantisierung über die Funktion `quantize_dynamic()`, die einen Pfad zu dem zu quantisierenden ONNX-Modell, einen Zielpfad, in dem das quantisierte Modell gespeichert werden soll, und den Datentyp, mit dem die Gewichte komprimiert werden sollen, als Argumente benötigt:

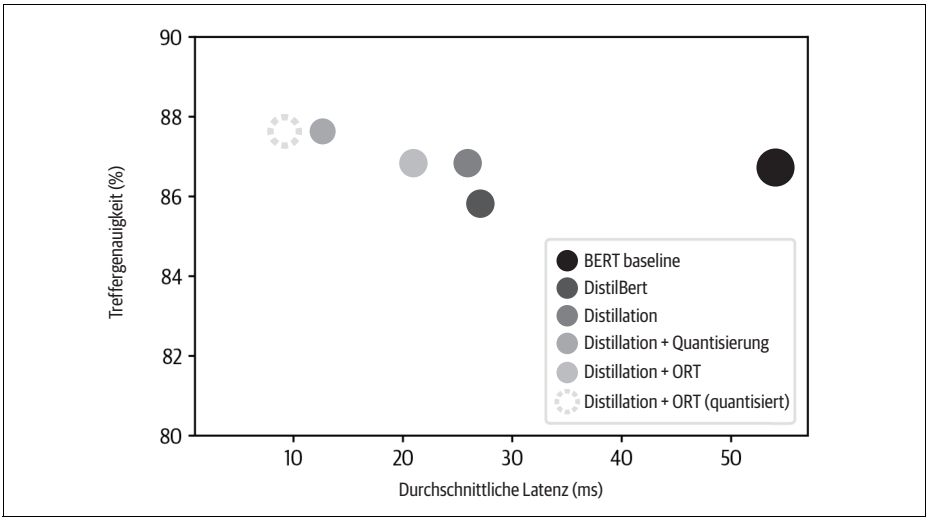
```
from onnxruntime.quantization import quantize_dynamic, QuantType

model_input = "onnx/model.onnx"
model_output = "onnx/model.quant.onnx"
quantize_dynamic(model_input, model_output, weight_type=QuantType.QInt8)
```

Nachdem das Modell nun quantisiert ist, können wir es mithilfe unserer Benchmark-Klasse vergleichen:

```
onnx_quantized_model = create_model_for_provider(model_output)
pipe = OnnxPipeline(onnx_quantized_model, tokenizer)
optim_type = "Distillation + ORT (quantisiert)"
pb = OnnxPerformanceBenchmark(pipe, clinc["test"], optim_type,
                              model_path=model_output)
perf_metrics.update(pb.run_benchmark())
```

```
Modelgröße (MB) - 64.20
Durchschnittliche Latenz (ms) - 9.24 +\- 0.29
Treffergenauigkeit auf Testdatensatz - 0.877
```

```
plot_metrics(perf_metrics, optim_type)
```



Die Quantisierung auf Basis der ORT hat die Größe des Modells und die Latenzzeit – verglichen mit dem Modell, das wir durch die PyTorch-basierte Quantisierung erhalten haben (der Kreis »Distillation + Quantisierung«) – um ca. 30 % reduziert. Ein Grund dafür ist, dass PyTorch nur die `nn.Linear`-Module optimiert, wohingegen ONNX auch die Embedding-Schicht quantisiert. Anhand des Diagramms können wir auch erkennen, dass die ORT-basierte Quantisierung unseres destillierten Modells die Latenzzeit gegenüber dem DistilBERT-Modell fast um den Faktor 3, und gegenüber dem BERT-Baseline-Modell mehr als um den Faktor 5, verringert hat.

Wir sind nun am Ende unserer Analyse von Verfahren zur Optimierung der Performance von Transformern im Rahmen der Inferenz angekommen. Wir haben gesehen, dass Verfahren wie die Quantisierung die Größe des Modells reduzieren, indem sie die numerische Genauigkeit der Darstellung (der Zahlen) verringern. Eine andere Strategie zur Verringerung der Größe besteht darin, einige Gewichte ganz zu entfernen. Dieser Ansatz wird als *Weight Pruning* bezeichnet und ist das Thema des nächsten Abschnitts.

Erhöhung der Sparsität von Modellen mithilfe von Weight Pruning

Bisher haben wir herausgefunden, dass die Knowledge Distillation und die Quantisierung von Gewichten recht effektiv sind, um schnellere Modelle für die Inferenz zu erstellen. In manchen Fällen unterliegen Sie jedoch auch starken Einschränkungen hinsichtlich des Speicherbedarfs Ihres Modells. Wenn zum Beispiel unser Produktmanager plötzlich entscheidet, dass unser Textassistent auf einem mobilen Gerät deployt werden soll, dann muss unser Intentionsklassifikator so wenig Speicherplatz wie möglich beanspruchen. Zum Abschluss unserer Analyse der verschiedenen Komprimierungsverfahren gehen wir nun der Frage nach, wie wir die

Anzahl der Parameter in unserem Modell verringern können, indem wir die unwichtigsten Gewichte im neuronalen Netz identifizieren und entfernen.

Sparsität tiefer neuronaler Netze

Wie in Abbildung 8-10 gezeigt, besteht die Hauptidee hinter Pruning darin, während des Trainings schrittweise einzelne Gewichte (und gegebenenfalls Neuronen) zu entfernen, sodass das Modell immer dünnbesetzter wird (engl. sparser). Das resultierende »beschnittene« Modell hat eine geringere Anzahl an Parametern, die ungleich null sind, weshalb es in einem kompakten (dünnbesetzten) Matrixformat gespeichert werden kann. Sie können Pruning auch mit der Quantisierung kombinieren, wenn Sie eine noch weitgehendere Komprimierung erzielen möchten.

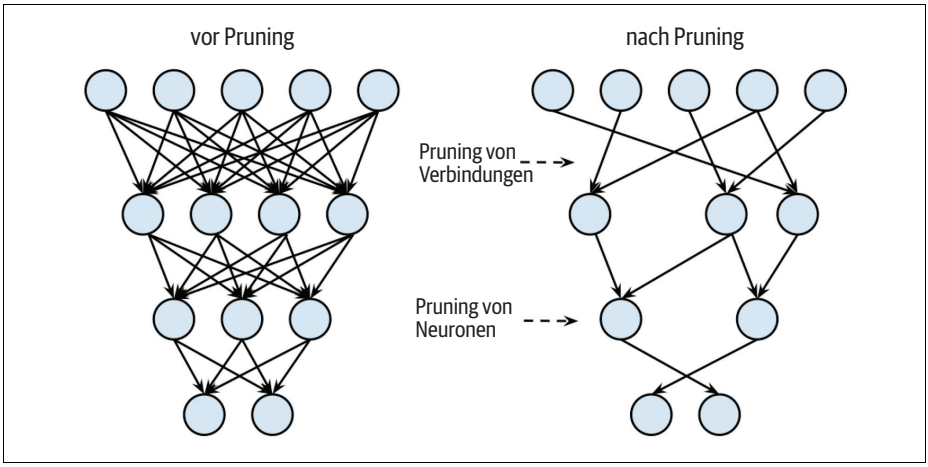


Abbildung 8-10: Die Gewichte und Neuronen vor und nach dem Pruning (mit freundlicher Genehmigung von Song Han)

Weight-Pruning-Methoden

Mathematisch gesehen besteht die Funktionsweise der meisten Weight-Pruning-Verfahren darin, eine Matrix S von *Importance-Scores* zu berechnen und dann die nach Höhe des Importance-Scores geordneten ersten k Prozent der Gewichte auszuwählen:

$$\mathrm{Top}_k(\mathbf{S})_{ij} = \begin{cases} 1 & \text{falls } S_{ij} \text{ in obersten } k\,\% \\ 0 & \text{andernfalls} \end{cases}$$

Im Grunde genommen fungiert k als neuer Hyperparameter, der den Grad der Sparsität (engl. Sparsity) des Modells steuert, d. h. den Anteil der Gewichte, die einen Wert von null aufweisen. Niedrigere Werte von k gehen mit dünnbesetzteren Matrizen einher. Basierend auf den Importance-Scores können wir dann eine *Maskierungsmatrix* $\mathbf{M}$ definieren, die die Gewichte W_{ij} während des Forward-Pass

einer Eingabe x_i maskiert und praktisch ein dünnbesetztes Netz von Aktivierungen a_i erzeugt:

$$a_i = \sum_k W_{ik} M_{ik} x_k$$

Wie in dem Diskussionspapier »Optimal Brain Surgeon«[17] humorvoll erörtert, steht im Mittelpunkt jedes Pruning-Verfahrens eine Reihe von Fragen, die es zu berücksichtigen gilt:

- Welche Gewichte sollten entfernt werden?
- Wie sollten die verbleibenden Gewichte angepasst werden, um die bestmögliche Leistung zu erzielen?
- Wie kann ein solches Pruning des Netzwerks möglichst recheneffizient durchgeführt werden?

Die Antworten auf diese Fragen sind maßgeblich dafür, wie die Score-Matrix **S** berechnet wird. Beginnen wir mit einem der frühesten und beliebtesten Pruning-Verfahren: dem Magnitude Pruning.

Magnitude Pruning

Wie der Begriff bereits vermuten lässt, berechnet das Magnitude Pruning die Scores nach der Größe der Gewichte $\mathbf{S} = (\,|W_{ij}|\,)_{1 \le j, j \le n}$ und leitet dann die Maskierungen aus $\mathbf{M} = \mathbf{Top}_k(\mathbf{S})$ ab. In der Literatur ist es üblich, das Magnitude Pruning schrittweise anzuwenden, indem zunächst das Modell trainiert wird, um zu lernen, welche Verbindungen wichtig sind, und dann die Gewichte mit dem geringsten Einfluss (Importance) entfernt werden.[18] Das dünnbesetzte Modell wird dann erneut trainiert und der Prozess wiederholt, bis der gewünschte Grad an Sparsität erreicht ist.

Ein Nachteil dieses Ansatzes ist, dass er rechenintensiv ist: Im Rahmen des Pruning müssen wir das Modell in jedem Schritt so lange trainieren, bis es konvergiert. Aus diesem Grund ist es im Allgemeinen besser, die anfängliche Sparsität s_i (die in der Regel null ist) nach einer gewissen Anzahl von Schritten s_f auf einen finalen Wert N zu erhöhen[19]:

$$s_t = s_f + \left(s_i - s_f\right)\left(1 - \frac{t - t_0}{N\Delta t}\right)^3 \quad \text{für } t \in \{t_0, t_0 + \Delta t, \ldots, t_0 + N\Delta t\}$$

17 B. Hassibi und D. Stork, »Second Order Derivatives for Network Pruning: Optimal Brain Surgeon«, *Proceedings of the 5th International Conference on Neural Information Processing Systems* (November 1992): 164–171, *https://papers.nips.cc/paper/1992/hash/303ed4c69846ab36c2904d3ba8573050-Abstract.html*.

18 S. Han et al., »Learning Both Weights and Connections for Efficient Neural Networks« (*https://arxiv.org/abs/1506.02626*), (2015).

19 M. Zhu und S. Gupta, »To Prune, or Not to Prune: Exploring the Efficacy of Pruning for Model Compression« (*https://arxiv.org/abs/1710.01878*), (2017).

Die Idee besteht darin, die binären Maskierungen **M** in jedem Δt Schritt zu aktualisieren, damit zuvor maskierte Gewichte während des Trainings wieder aktiviert werden und sich das Modell von einem möglichen Genauigkeitsverlust infolge des Pruning erholen kann. Wie in Abbildung 8-11 gezeigt, impliziert der kubische Faktor, dass die Anzahl der Gewichte, die entfernt werden (Weight Pruning), in den ersten Schritten am höchsten ist (wenn die Anzahl der redundanten Gewichte groß ist) und dann allmählich abnimmt.

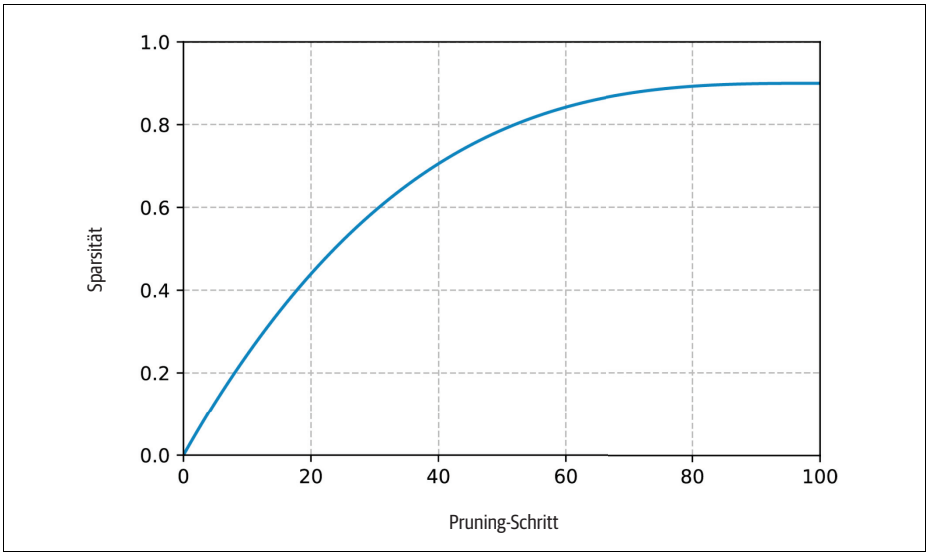


Abbildung 8-11: Der kubische Sparsity Scheduler, der für das Pruning verwendet wird

Ein Problem beim Magnitude Pruning ist, dass es eigentlich für das reine überwachte Lernen (engl. Supervised Learning) konzipiert ist, bei dem die Bedeutsamkeit (Importance) der einzelnen Gewichte direkt mit der jeweiligen Aufgabe zusammenhängt. Im Gegensatz dazu wird beim Transfer Learning der Einfluss der einzelnen Gewichte in erster Linie durch die Pretraining-Phase bestimmt, sodass beim Magnitude Pruning Verbindungen entfernt werden könnten, die für die im Rahmen des Feintunings verfolgte Aufgabe wichtig sind. In jüngster Zeit wurde von den Forschern von Hugging Face ein adaptiver Ansatz, das sogenannte Movement Pruning, vorgeschlagen – sehen wir uns das einmal an.[20]

Movement Pruning

Das Grundprinzip des Movement Pruning besteht darin, während des Feintunings *nach und nach* Gewichte zu entfernen, sodass das Modell zunehmend *dünnbesetzter* (engl. sparser) wird. Die wichtigste Neuerung ist, dass sowohl die Gewichte als auch die Scores während des Feintunings gelernt werden. Anstatt also direkt von den Gewichten abgeleitet zu werden (wie beim Magnitude Pruning), sind die Scores beim

20 V. Sanh, T. Wolf, and A.M. Rush, »Movement Pruning: Adaptive Sparsity by Fine-Tuning« (*https://arxiv.org/abs/2005.07683*), (2020).

Movement Pruning nicht fest vorgegeben und werden wie jeder andere Parameter des neuronalen Netzes mittels des Gradientenabstiegsverfahrens (engl. Gradient Descent) gelernt. Das bedeutet, dass im Rahmen des Backward-Pass auch der Gradient des Verlusts L in Bezug auf die Scores S_{ij} berücksichtigt wird.

Sobald die Werte erlernt wurden, ist es einfach, die binäre Maskierung, $\mathbf{M} = \mathbf{Top}_k(\mathbf{S})$, zu erzeugen.[21]

Die Grundidee hinter dem Movement Pruning ist, dass die Gewichte, die sich am stärksten von null wegbewegen (»moving«), die wichtigsten sind, und man sie beibehalten sollte. Mit anderen Worten, die positiven Gewichte werden während des Feintunings immer größer (und umgekehrt für die negativen Gewichte), was gleichbedeutend mit der Aussage ist, dass die Scores zunehmen, je weiter sich die Gewichte von null wegbewegen. Wie in Abbildung 8-12 gezeigt, unterscheidet sich dieses Vorgehen vom Magnitude Pruning, bei dem die Gewichte, die am *weitesten* von null entfernt sind, als die relevantesten bzw. einflussreichsten ausgewählt werden.

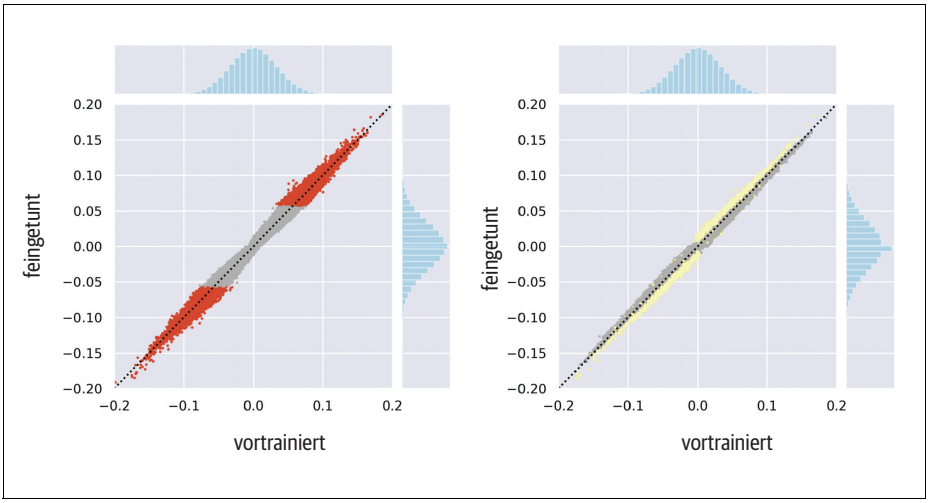


Abbildung 8-12: Vergleich der Gewichte, die im Rahmen des Magnitude Pruning (links) bzw. des Movement Pruning (rechts) entfernt wurden

Die Unterschiede zwischen den beiden Pruning-Verfahren machen sich auch in der Verteilung der verbleibenden Gewichte bemerkbar. Wie in Abbildung 8-13 zu sehen ist, führt das Magnitude Pruning dazu, dass sich zwei Cluster von Gewichten bilden, während das Movement Pruning eine gleichmäßigere Verteilung zur Folge hat.

Zum Zeitpunkt des Verfassens dieses Buchs unterstützt die 🤗 Transformers-Bibliothek noch keine Pruning-Verfahren. Glücklicherweise gibt es eine elegante Bibliothek namens *Neural Networks Block Movement Pruning* (*https://oreil.ly/*

21 Es gibt auch eine »sanfte« Version des Movement Pruning, bei der anstelle der relevantesten k% der Gewichte ein globaler Schwellenwert τ verwendet wird, um eine binäre Maskierung zu definieren, für die $\mathbf{M} = (\mathbf{S} > \tau)$ gilt.

aHEvD), mit der sich viele dieser Ansätze umsetzen lassen. Wir empfehlen Ihnen, sich insbesondere dann mit der Bibliothek zu befassen, wenn Sie Bedenken haben, dass Ihr Speicher nicht ausreicht.

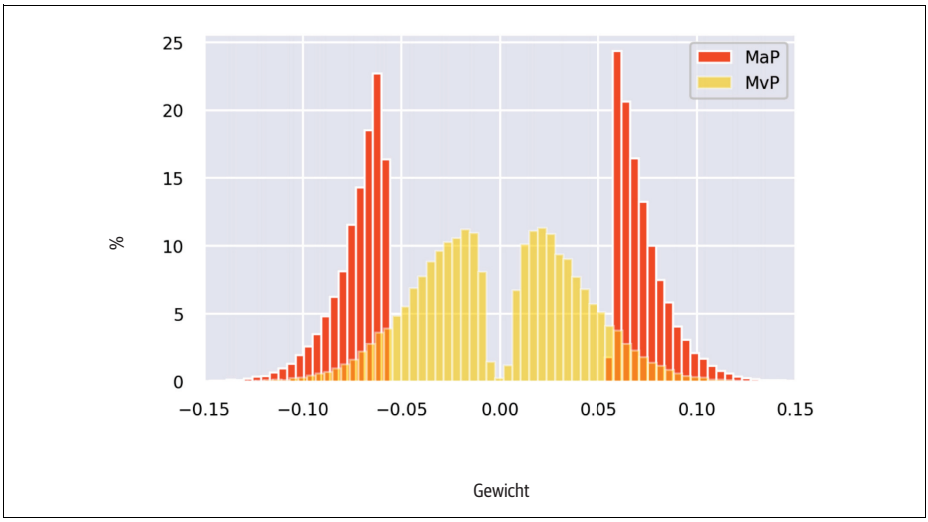


Abbildung 8-13: Verteilung der verbleibenden Gewichte infolge des Magnitude Pruning (MaP) und des Movement Pruning (MvP)

Zusammenfassung

Wie sich gezeigt hat, erfordert die Optimierung von Transformern für das Deployment in Produktionsumgebungen eine Komprimierung in zweierlei Hinsicht: in Bezug auf die Latenzzeit und auf den Speicherbedarf. Ausgehend von einem feingetunten Modell haben wir die Knowledge Distillation, die Quantisierung und überdies Optimierungen mittels der ORT angewandt, um sowohl die Latenz als auch den Speicherbedarf deutlich zu reduzieren. Dabei haben wir festgestellt, dass die Quantisierung in Verbindung mit der Konvertierung in das ORT-Format die größten Vorteile bietet – und das bei einem relativ geringen Aufwand.

Obwohl das Pruning eine effektive Strategie darstellt, um die Speichergröße von Transformer-Modellen zu verringern, ist die derzeitige Hardware nicht für Operationen auf Basis dünnbesetzter Matrizen optimiert, weshalb der Nutzen dieses Ansatzes begrenzt ist. Allerdings wird auf diesem Gebiet intensiv geforscht, sodass wahrscheinlich viele dieser Hindernisse bereits überwunden sein könnten, wenn dieses Buch in den Regalen steht.

Was bleibt noch zu sagen? Alle in diesem Kapitel vorgestellten Verfahren können auch auf andere Aufgaben übertragen werden, z. B. auf das Question Answering, die Named Entity Recognition oder das Language Modeling. Sollten Sie damit zu kämpfen haben, die erforderliche Latenz zu erreichen, oder sollte Ihr Modell Ihr gesamtes Rechenbudget verschlingen, empfehlen wir Ihnen, eines dieser Verfahren auszuprobieren.

So weit zur Optimierung der Performance. Im nächsten Kapitel befassen wir uns mit dem schlimmsten Albtraum eines jeden Data Scientists: der Situation, dass nur wenige oder gar keine gelabelte Daten vorliegen.

KAPITEL 9

Ansätze bei wenigen bis keinen Labels

Eine Frage ist so tief in den Köpfen der Data Scientists verankert, dass sie in der Regel die erste ist, die zu Beginn eines neuen Projekts gestellt wird: Gibt es denn überhaupt gelabelte Daten? Meistens lautet die Antwort »nein« oder »ein paar«, einhergehend mit der Erwartung des Kunden, dass die ausgefallenen Machine-Learning-Modelle Ihres Teams dennoch gut funktionieren sollten. Da Modelle, die nur mit sehr kleinen Datensätzen trainiert wurden, in der Regel keine guten Ergebnisse erzielen, besteht eine naheliegende Lösung darin, schlicht mehr Daten zu annotieren bzw. zu labeln. Dies ist jedoch zeitaufwendig und kann sehr kostspielig sein, vor allem dann, wenn die Annotationen Fachwissen erfordern, um sie validieren zu können.

Glücklicherweise gibt es mehrere Methoden, die sich gut für den Umgang mit wenigen bis gar keinen Labels eignen! Einige davon sind Ihnen vielleicht schon bekannt, wie z.B. *Zero-Shot-* oder *Few-Shot-Learning*. Wie GPT-3 auf beeindruckende Weise beweist, können mit diesen Methoden eine ganze Reihe von Aufgaben auf Basis von nur ein paar Dutzend Beispielen bewältigt werden.

Welche Methode am besten funktioniert, hängt im Allgemeinen von der Aufgabe, der Menge an verfügbaren Daten und dem Anteil dieser Daten ab, der gelabelt ist. Der in Abbildung 9-1 gezeigte Entscheidungsbaum hilft Ihnen bei der Auswahl der am besten geeigneten Methode.

Gehen wir diesen Entscheidungsbaum Schritt für Schritt durch:

1. *Verfügen Sie über gelabelte Daten?*
   Selbst eine Handvoll gelabelter Beispiele kann einen Unterschied machen, wenn es darum geht, welche Methode am besten funktioniert. Wenn Sie überhaupt keine gelabelten Daten vorliegen haben, können Sie mit dem Zero-Shot-Learning-Ansatz beginnen, der oft eine gute Ausgangsbasis bietet.
2. *Wenn ja, wie viele gelabelte Daten haben Sie vorliegen?*
   Wenn gelabelte Daten zur Verfügung stehen, ist die entscheidende Frage, wie viele vorhanden sind. Wenn Sie viele gelabelte Trainingsdaten zur Verfügung haben, können Sie den standardmäßigen Feintuning-Ansatz verwenden, der in Kapitel 2 vorgestellt wurde.

3. *Verfügen Sie über ungelabelte Daten?*

Wenn Sie nur eine Handvoll gelabelter Beispiele vorliegen haben, kann es ungemein hilfreich sein, wenn Sie Zugang zu großen Mengen an nicht gelabelten Daten haben. Sollte dies der Fall sein, können Sie die ungelabelten Daten entweder dazu verwenden, das Sprachmodell erst auf der Domäne feinzutunen, bevor Sie einen Klassifikator trainieren, oder Sie können auf anspruchsvollere Methoden wie die Unsupervised Data Augmentation (UDA) oder das Uncertainty-Aware Self-Training (UST) zurückgreifen.[1] Wenn Ihnen keine ungelabelten Daten zur Verfügung stehen, haben Sie auch nicht die Möglichkeit, weitere Daten zu labeln bzw. zu annotieren. In diesem Fall können Sie den Ansatz des Few-Shot-Learning verfolgen oder die Embeddings eines vortrainierten Sprachmodells als Nachschlagetabelle verwenden und eine Suche auf Basis des Nächste-Nachbarn-Algorithmus (engl. Nearest-Neighbor) durchführen.

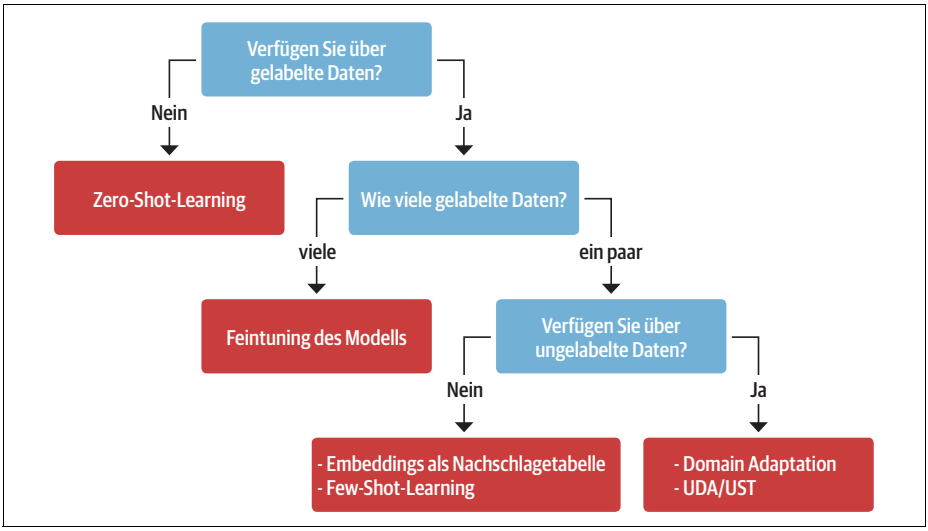


Abbildung 9-1: Verschiedene Methoden, die zur Verbesserung der Leistung des Modells verwendet werden können, wenn nur wenige oder keine gelabelten Daten zur Verfügung stehen

Wir werden uns in diesem Kapitel durch die in dem Entscheidungsbaum dargestellten Methoden durcharbeiten. Hierfür gehen wir ein Problem an, mit dem viele Support-Teams häufig konfrontiert sind, die Issue-Tracker wie Jira (*https://oreil.ly/TVqZQ*) oder GitHub (*https://oreil.ly/e0Bd1*) verwenden, um ihre Benutzer zu unterstützen: die Kennzeichnung (»Taggen«) von Issues mit Metadaten auf der Grundlage der Beschreibung (»Description«) des Issue. Diese Tags können den Typ des Issue, das Produkt, das das Problem bzw. den Issue verursacht, oder das Team, das mit der Bearbeitung des gemeldeten Issue betraut ist, festlegen. Wenn Sie diesen Prozess automatisieren,

1 Q. Xie et al., »Unsupervised Data Augmentation for Consistency Training« (*https://arxiv.org/abs/1904.12848*), (2019); S. Mukherjee und A.H. Awadallah, »Uncertainty-Aware Self-Training for Few-Shot Text Classification« (*https://arxiv.org/abs/2006.15315*), (2020).

kann die Produktivität erheblich gesteigert werden und ermöglicht es somit dem Support-Team, sich auf die Unterstützung der Benutzer zu konzentrieren. Als Beispiel nehmen wir die GitHub-Issues eines beliebten Open-Source-Projekts: der 🤗 Transformers-Bibliothek. Werfen wir zunächst einen Blick darauf, welche Informationen in diesen Issues enthalten sind, wie die Aufgabe formuliert werden kann und wie die Daten beschafft werden können.

Die in diesem Kapitel vorgestellten Methoden eignen sich gut für die Klassifizierung von Texten. Für komplexere Aufgaben wie der Named Entity Recognition, dem Question Answering oder der automatischen Zusammenfassung können jedoch andere Methoden wie die Augmentierung von Daten erforderlich sein.

Erstellung eines GitHub-Issues-Tagger

Wenn Sie zur Registerkarte Issues (*https://oreil.ly/StdH3*) des Repository der 🤗 Transformers-Bibliothek navigieren, finden Sie Issues wie den in Abbildung 9-2 vor, die jeweils einen Titel, eine Beschreibung und eine Reihe von Tags bzw. Labels, die den Issue charakterisieren, enthalten.

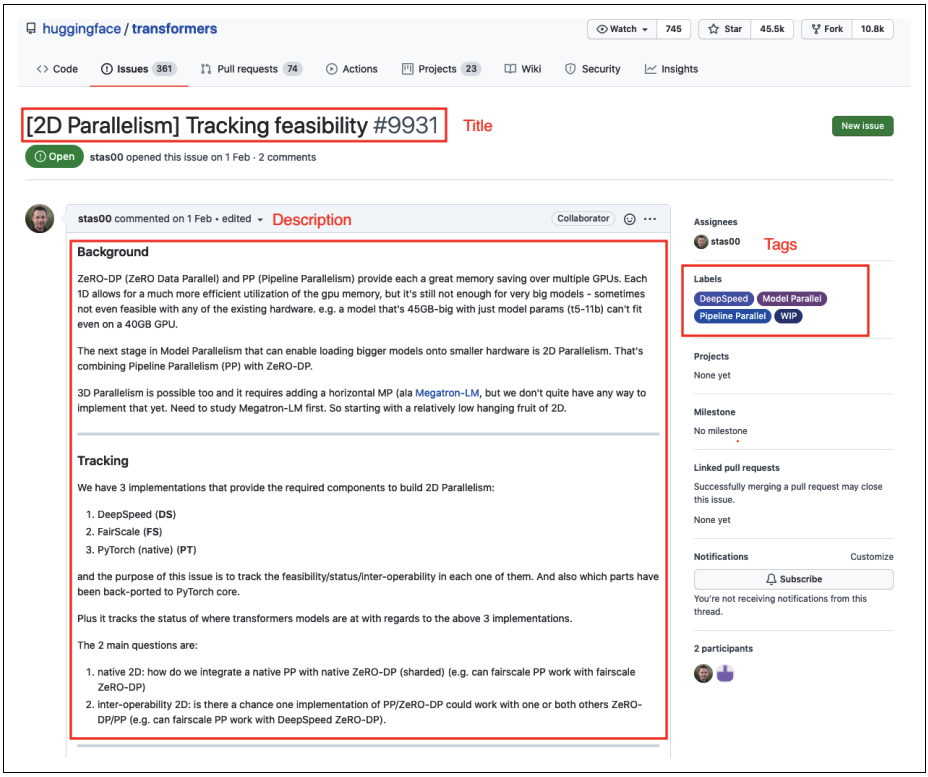


Abbildung 9-2: Ein typischer Issue auf dem GitHub-Repository der 🤗 Transformers-Bibliothek

Die Aufgabe können wir dementsprechend als überwachte Lernaufgabe (engl. Supervised Learning Task) formulieren: Wir möchten ein oder mehrere Labels auf Basis des Titels und der Beschreibung eines Issue vorhersagen. Da jedem Issue eine variable Anzahl von Labels zugewiesen werden kann, bedeutet dies, dass wir es hier mit einem sogenannten *Multilabel-Textklassifizierungsproblem* zu tun haben. Dies stellt in der Regel eine größere Herausforderung als ein Multiklassen- bzw. multikategoriales Problem dar, wie wir es in Kapitel 2 behandelt haben, bei dem jeder Tweet nur einer einzigen Emotion zugeordnet wird.

Nachdem wir nun gesehen haben, wie GitHub-Issues aufgebaut sind, sollten wir uns ein Bild davon machen, wie wir sie herunterladen können, damit wir unseren Datensatz erstellen können.

Die Daten beschaffen

Für unseren Datensatz möchten wir alle Issues des Repository abrufen. Hierfür verwenden wir die REST API von GitHub (*https://oreil.ly/q605k*), über die wir den `Issues`-Endpunkt (*https://oreil.ly/qXdWV*) abfragen. Dieser Endpunkt (engl. Endpoint) gibt eine Liste von JSON-Objekten zurück, von denen jedes eine große Anzahl von Feldern bezüglich des jeweiligen Issue enthält – darunter der Status (offen oder geschlossen), die Information, wer den Issue angelegt hat, sowie der Titel, der Textkörper mit der Beschreibung und die Labels (wie in Abbildung 9-2 gezeigt).

Da es eine gewisse Zeit dauert, ehe alle Issues abgerufen worden sind, haben wir Ihnen zusätzlich zur im Notebook des Kapitels befindlichen Funktion `fetch_issues()`, mit der Sie die Issues selbst herunterladen können, eine Datei *github-issues-transformers.jsonl* im GitHub Repository (*https://oreil.ly/if2dm*) dieses Buchs bereitgestellt, die den Datensatz enthält.

Da die REST API von GitHub Pull Requests als Issues behandelt, sind in unserem Datensatz beide Arten vertreten. Der Einfachheit halber entwickeln wir unseren Klassifikator für beide Arten von Issues. In der Praxis sollten Sie jedoch in Erwägung ziehen, zwei separate Klassifikatoren zu erstellen, um ein leistungsfähigeres Modell zu erhalten.

Nachdem wir nun wissen, wie wir die Daten abrufen können, sollten wir uns darum kümmern, sie zu bereinigen.

Die Daten vorbereiten

Sobald alle Issues heruntergeladen wurden, können wir sie mit Pandas laden:

```
import pandas as pd

dataset_url = "https://git.io/nlp-with-transformers"
```

```
df_issues = pd.read_json(dataset_url, lines=True)
print(f"Shape des DataFrame: {df_issues.shape}")
```

```
Shape des DataFrame: (9930, 26)
```

Unser Datensatz umfasst rund 10.000 Issues. Wenn wir eine einzelne Zeile betrachten, können wir feststellen, dass die von der GitHub-API abgerufenen Informationen viele Felder wie die URL, die ID, das Datum, den Benutzer, den Titel, den Textkörper sowie die Labels enthalten:

```
cols = ["url", "id", "title", "user", "labels", "state", "created_at", "body"]
df_issues.loc[2, cols].to_frame()
```

	2
url	https://api.github.com/repos/huggingface/trans...
id	849529761
title	[DeepSpeed] ZeRO stage 3 integration: getting ...
user	{'login': 'stas00', 'id': 10676103, 'node_id':...
labels	[{'id': 2659267025, 'node_id': 'MDU6TGFiZWwyNj...
state	open
created_at	2021-04-02 23:40:42
body	**[This is not yet alive, preparing for the re...

Unser Interesse gilt der `labels`-Spalte. Die einzelnen Zeilen dieser Spalte enthalten jeweils eine Liste, bestehend aus JSON-Objekten, die Metadaten zu jedem Label beinhalten:

```
[
   {
      "id":2659267025,
      "node_id":"MDU6TGFiZWwyNjU5MjY3MDI1",
      "url":"https://api.github.com/repos/huggingface...",
      "name":"DeepSpeed",
      "color":"4D34F7",
      "default":false,
      "description":""
   }
]
```

In unserem Fall benötigen wir nur das `name`-Feld jedes Label-Objekts, weshalb wir wir die `labels`-Spalte einfach mit den Namen bzw. Bezeichnungen der Labels überschreiben:

```
df_issues["labels"] = (df_issues["labels"]
                       .apply(lambda x: [meta["name"] for meta in x]))
df_issues[["labels"]].head()
```

	labels
0	[]
1	[]

	labels
2	[DeepSpeed]
3	[]
4	[]

Jede Zeile in der Spalte `labels` entspricht nun einer Liste, in der die GitHub-Labels enthalten sind. Um die Anzahl der Labels je Issue zu bestimmen, müssen wir lediglich die Länge der Zeilen ermitteln:

```
df_issues["labels"].apply(lambda x : len(x)).value_counts().to_frame().T
```

	0	1	2	3	4	5
labels	6440	3057	305	100	25	3

Wie Sie sehen, wurde den meisten Issues kein oder nur ein einziges Label zugeordnet, und nur vergleichsweise wenige sind mit mehr als einem Label versehen. Als Nächstes sehen wir uns die acht am häufigsten vergebenen Labels in dem Datensatz an. Hierzu können wir die `explode()`-Funktion der Pandas-Bibliothek auf die `labels`-Spalte anwenden, sodass jedes Label in den einzelnen Listen in eine eigene Zeile überführt wird. Nun können wir auf einfache Weise ermitteln, wie häufig jedes Label vorkommt, indem wir einfach zählen, wie oft die jeweiligen Labels in der Spalte vorkommen:

```
df_counts = df_issues["labels"].explode().value_counts()
print(f"Anzahl verschiedener Labels: {len(df_counts)}")
# Häufigkeiten der acht am häufigsten vorkommenden Labels ausgeben lassen
df_counts.to_frame().head(8).T
```

```
Anzahl verschiedener Labels: 65
```

	wontfix	model card	Core: Tokenization	New model	Core: Modeling	Help wanted	Good First Issue	Usage
labels	2284	649	106	98	64	52	50	46

Wie wir ermittelt haben, gibt es in dem Datensatz 65 verschiedene Labels. Die Kategorien sind offenbar sehr unausgewogen verteilt, wobei `wontfix` und `model card` die am häufigsten verwendeten Labels sind. Damit wir unser Klassifizierungsproblem besser handhaben können, konzentrieren wir uns darauf, den Tagger lediglich für eine Untergruppe von Labels zu erstellen. Einige Labels, wie z.B. `Good First Issue` oder `Help Wanted`, sind möglicherweise sehr schwer auf Basis der Beschreibung eines Issue vorherzusagen, während andere anhand einer einfachen Regel klassifiziert werden könnten, wie beispielsweise `model card`, wenn eine Model Card auf dem Hugging Face Hub hinzugefügt wird.

Mit dem folgenden Code filtern wir den Datensatz, sodass lediglich die Auswahl an Labels übrig bleibt, die wir einbeziehen möchten. Darüber hinaus vereinheitlichen wir die Namen, um sie leichter lesbar zu machen:

```
label_map = {"Core: Tokenization": "tokenization",
             "New model": "new model",
             "Core: Modeling": "model training",
             "Usage": "usage",
             "Core: Pipeline": "pipeline",
             "TensorFlow": "tensorflow or tf",
             "PyTorch": "pytorch",
             "Examples": "examples",
             "Documentation": "documentation"}

def filter_labels(x):
    return [label_map[label] for label in x if label in label_map]

df_issues["labels"] = df_issues["labels"].apply(filter_labels)
all_labels = list(label_map.values())
```

Werfen wir nun einen Blick auf die Häufigkeitsverteilung der verbleibenden Labels:

```
df_counts = df_issues["labels"].explode().value_counts()
df_counts.to_frame().T
```

	tokeni-zation	new model	model trai-ning	usage	pipe-line	tensor-flow or tf	pytorch	docu-menta-tion	ex-amples
labels	106	98	64	46	42	41	37	28	24

Im weiteren Verlauf dieses Kapitels wird es nützlich sein, die ungelabelten Issues als separaten Trainingsdatensatz zu behandeln. Vor diesem Hintergrund bietet es sich an, eine neue Spalte zu erstellen, die angibt, ob der jeweilige Issue mit einem Label versehen ist oder nicht:

```
df_issues["split"] = "unlabeled"
mask = df_issues["labels"].apply(lambda x: len(x)) > 0
df_issues.loc[mask, "split"] = "labeled"
df_issues["split"].value_counts().to_frame()
```

	split
unlabeled	9489
labeled	441

Schauen wir uns nun eines der Beispiele genauer an:

```
for column in ["title", "body", "labels"]:
    print(f"{column}: {df_issues[column].iloc[26][:500]}\n")


title: Add new CANINE model

body: # 🤗 New model addition

## Model description
```

```
Google recently proposed a new **C**haracter **A**rchitecture with **N**o
 tokenization **I**n **N**eural **E**ncoders architecture (CANINE). Not only
 the title is exciting:

Pipelined NLP systems have largely been superseded by end-to-end neural
 modeling, yet nearly all commonly-used models still require an explicit
 tokenization step. While recent tokenization approaches based on data-derived
 subword lexicons are less brittle than manually en

labels: ['new model']
```

Das vorliegende Beispiel ist mit dem Tag `new model` versehen, was sinnvoll ist, da eine neue Modellarchitektur vorgeschlagen wird. Wir können auch erkennen, dass der Titel (`title`) Informationen enthält, die für unseren Klassifikator von Nutzen sein werden, weshalb wir ihn mit der Beschreibung des Issue, die im `body`-Feld enthalten ist, zusammenführen bzw. verketten:

```
df_issues["text"] = (df_issues
                     .apply(lambda x: x["title"] + "\n\n" + x["body"], axis=1))
```

Bevor wir uns die restlichen Daten vornehmen, sollten wir noch nach Duplikaten in den Daten suchen und diese mithilfe der Methode `drop_duplicates()` löschen:

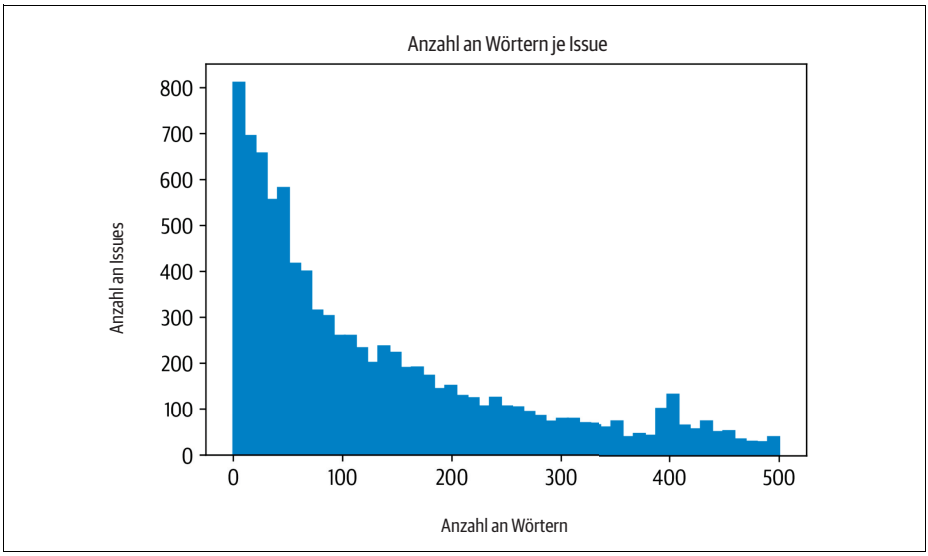
```
len_before = len(df_issues)
df_issues = df_issues.drop_duplicates(subset="text")
print(f"Anteil von {(len_before-len(df_issues))/len_before:.2%} " \
       "an Duplikaten gelöscht.")

Anteil von 1.88% an Duplikaten gelöscht.
```

Wie wir feststellen, gab es in unserem Datensatz einige doppelte Issues, die aber nur einen geringen Anteil ausmachen. Wie bereits in vorherigen Kapiteln, ist es auch hier sinnvoll, einen Blick auf die Anzahl der Wörter in unseren Texten zu werfen, um in Erfahrung zu bringen, ob wir große Informationsverluste erleiden würden, wenn wir die Kontextlänge (engl. Context Size) der einzelnen Modelle verringern:

```
import numpy as np
import matplotlib.pyplot as plt

(df_issues["text"].str.split().apply(len)
 .hist(bins=np.linspace(0, 500, 50), grid=False, edgecolor="C0"))
plt.title("Anzahl an Wörtern je Issue")
plt.xlabel("Anzahl an Wörtern")
plt.ylabel("Anzahl an Issues")
plt.show()
```

Die Häufigkeitsverteilung weist den häufig für Textdatensätze charakteristischen langen Schwanz bzw. Rand auf. Die Mehrzahl der Texte ist relativ kurz, aber es gibt auch Issues, die mehr als 500 Wörter umfassen. Es ist nicht ungewöhnlich, dass einige Issues sehr lang ausfallen, insbesondere wenn Fehlermeldungen und Codeausschnitte direkt in der Beschreibung mit aufgeführt werden. Da die meisten Transformer-Modelle eine Kontextlänge von 512 Tokens oder mehr haben, ist es eher

unwahrscheinlich, dass die Gesamtleistung beeinträchtigt wird, da nur eine Handvoll sehr langer Issues gekürzt werden müsste. Nachdem wir nun unseren Datensatz erkundet und bereinigt haben, müssen wir als Letztes unseren Trainings- und Validierungsdatensatz bestimmen, mit denen wir unsere Klassifikatoren vergleichen können. Werfen wir einen Blick darauf, wie wir das anstellen können.



Trainingsdatensätze erstellen

Die Erstellung von Trainings- und Validierungsdatensätzen gestaltet sich bei Multilabel-Problemstellungen etwas schwieriger, da nicht garantiert werden kann, dass alle Labels ausgewogen (engl. balanced) verteilt sind. Allerdings ist es möglich, sich diesem Zustand anzunähern. Hierfür können wir die Scikit-multilearn-Bibliothek (*http://scikit.ml*) verwenden, die speziell für diesen Zweck vorgesehen ist. Als Erstes müssen wir unsere Labels (wie `pytorch` und `tokenization`) in ein Format überführen, das das Modell verarbeiten kann. Dazu können wir die Klasse `MultiLabelBinarizer` der Scikit-learn-Bibliothek verwenden, die jeweils für eine gegebene Liste von Labelnamen einen Vektor erzeugt, der jeweils eine Null für fehlende und eine Eins für vorhandene Labels aufweist. Wenden wir am besten gleich den `MultiLabelBinarizer` auf unsere Liste mit den Labels (`all_labels`) an, um besser nachvollziehen zu können, wie die Labelnamen in dem Vektor zugeordnet werden:

```
from sklearn.preprocessing import MultiLabelBinarizer

mlb = MultiLabelBinarizer()
mlb.fit([all_labels])
mlb.transform([["tokenization", "new model"], ["pytorch"]])

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

In diesem einfachen Beispiel wird ersichtlich, dass es in der ersten Liste zwei Treffer gibt, die den Labels `tokenization` und `new model` entsprechen, während bei der zweiten Liste nur ein Treffer (`pytorch`) zu verzeichnen ist.

Um den Datensatz in Teildatensätze (engl. Splits) aufzuteilen, können wir die Funktion `iterative_train_test_split()` aus der Scikit-multilearn-Bibliothek verwenden, die den Trainings- und den Testdatensatz schrittweise erstellt, um eine ausgewogene Verteilung in Bezug auf die Labels zu gewährleisten. Allerdings integrieren wir sie in eine zusätzliche Funktion, damit sie direkt auf `DataFrame`-Objekte angewandt werden kann. Da die `iterative_train_test_split()`-Funktion eine zweidimensionale Feature-Matrix erwartet, müssen wir eine Dimension zum Index, der von der Länge des Dataframe abgeleitet wird, hinzufügen, bevor die Aufteilung vorgenommen werden kann:

```
from skmultilearn.model_selection import iterative_train_test_split

def balanced_split(df, test_size=0.5):
    ind = np.expand_dims(np.arange(len(df)), axis=1)
    labels = mlb.transform(df["labels"])
    ind_train, _, ind_test, _ = iterative_train_test_split(ind, labels,
                                                           test_size)
    return df.iloc[ind_train[:, 0]], df.iloc[ind_test[:,0]]
```

Mit der Funktion `balanced_split()` können wir die Daten in einen gelabelten und einen ungelabelten Datensatz aufteilen und dann auf Basis des gelabelten Datensatzes ausgewogene Trainings-, Validierungs- und Testdatensätze erstellen:

```
from sklearn.model_selection import train_test_split

df_clean = df_issues[["text", "labels", "split"]].reset_index(drop=True).copy()
df_unsup = df_clean.loc[df_clean["split"] == "unlabeled", ["text", "labels"]]
df_sup = df_clean.loc[df_clean["split"] == "labeled", ["text", "labels"]]

np.random.seed(0)
df_train, df_tmp = balanced_split(df_sup, test_size=0.5)
df_valid, df_test = balanced_split(df_tmp, test_size=0.5)
```

Anschließend erstellen wir ein `DatasetDict`-Objekt, in dem alle (gelabelten) Teildatensätze und auch der ungelabelte Datensatz enthalten sind. Dadurch gestaltet es sich später für uns einfacher, die Datensätze zu tokenisieren und in dem `Trainer`-Objekt zu integrieren. Um die einzelnen Datensätze direkt aus den entsprechenden Pandas-`DataFrame`-Objekten zu laden, greifen wir auf die praktische `from_pandas()`-Methode zurück:

```
from datasets import Dataset, DatasetDict

ds = DatasetDict({
    "train": Dataset.from_pandas(df_train.reset_index(drop=True)),
    "valid": Dataset.from_pandas(df_valid.reset_index(drop=True)),
    "test": Dataset.from_pandas(df_test.reset_index(drop=True)),
    "unsup": Dataset.from_pandas(df_unsup.reset_index(drop=True))})
```

So weit, so gut. Als Letztes können wir nun den Trainingsdatensatz in kleinere Trainingsdatensätze (»Slices«) unterteilen, die jeweils unterschiedlich groß sind, damit wir die Leistung bzw. Genauigkeit der einzelnen Klassifikatoren in Abhängigkeit von der Größe des jeweils genutzten Trainingsdatensatzes (bzw. Slice) evaluieren können.

Unterschiedlich große Trainingsdatensätze erstellen

Der Datensatz weist beide Merkmale auf, die im Rahmen dieses Kapitels im Vordergrund stehen: Zum einen haben wir nur wenige gelabelte Daten vorliegen, und zum anderen handelt es sich dabei um ein Multilabel-Klassifizierungsproblem. Der Trainingsdatensatz besteht aus lediglich 220 Beispielen, was selbst im Rahmen des Transfer Learning eine Herausforderung darstellt. Um herauszufinden, wie die einzelnen Methoden in diesem Kapitel – vor dem Hintergrund, dass nur wenig gelabelte Daten zur Verfügung stehen – abschneiden, werden wir auch Slices mit noch weniger Beispielen aus dem Trainingsdatensatz (bzw. mehrere kleinere Trainingsdatensätze) erstellen. Wir können dann mithilfe eines Diagramms analysieren, wie sich die Anzahl der verfügbaren Trainingsbeispiele auf die Leistung der Modelle auswirkt, und die verschiedenen Ansätze miteinander vergleichen. Wir beginnen mit nur acht Beispielen je Label und erhöhen unter Verwendung der Funktion `iterative_train_test_split()` nach und nach die Anzahl der Trainingsbeispiele, bis das Slice schließlich den kompletten Trainingsdatensatz umfasst:

```
np.random.seed(0)
all_indices = np.expand_dims(list(range(len(ds["train"]))), axis=1)
indices_pool = all_indices
labels = mlb.transform(ds["train"]["labels"])
train_samples = [8, 16, 32, 64, 128]
train_slices, last_k = [], 0

for i, k in enumerate(train_samples):
    # Jeweils so viele Beispiele ziehen, bis entsprech. Slice-Größe erreicht ist
    indices_pool, labels, new_slice, _ = iterative_train_test_split(
        indices_pool, labels, (k-last_k)/len(labels))
    last_k = k
    if i==0: train_slices.append(new_slice)
    else: train_slices.append(np.concatenate((train_slices[-1], new_slice)))

# Vollständigen Trainingsdatensatz als letztes Slice hinzufügen
train_slices.append(all_indices), train_samples.append(len(ds["train"]))
train_slices = [np.squeeze(train_slice) for train_slice in train_slices]
```

Beachten Sie, dass die Stichproben bzw. Slices bei diesem iterativen Ansatz nur näherungsweise in den von uns vorgegebenen Größen erstellt werden, da es nicht immer möglich ist, eine hinsichtlich der Labels ausgewogene Verteilung für eine gegebene Slice-Größe zu finden:

```
print("Sollgröße der Slices:")
print(train_samples)
```

```
print("Tatsächliche Größe der Slices:")
print([len(x) for x in train_slices])
```

```
Sollgröße der Slices:
[8, 16, 32, 64, 128, 223]
Tatsächliche Größe der Slices:
[10, 19, 36, 68, 134, 223]
```

Die Sollgrößen der Slices werden wir in den folgenden Diagrammen als Achsenbeschriftung verwenden. Prima, wir haben nun mehrere (Teil-)Trainingsdatensätze auf Basis unseres Datensatzes erstellt. Werfen wir als Nächstes einen Blick darauf, wie wir ein solides Baseline-Modell trainieren können!

Implementierung eines naiven Bayes-Klassifikators als Baseline

Wenn Sie ein neues NLP-Projekt in Angriff nehmen, ist es stets eine gute Idee, eine Reihe von Baseline-Modellen zu implementieren, die eine solide Grundlage bieten. Hierfür gibt es zwei Hauptgründe:

1. Eine Baseline, die auf regulären Ausdrücken, selbst erstellten Regeln oder einem sehr einfachen Modell basiert, kann das Problem bereits sehr gut lösen. In diesen Fällen gibt es keinen Grund, schwere Geschütze wie Transformer-Modelle aufzufahren, deren Deployment und Wartung in Produktionsumgebungen in der Regel komplexer sind.
2. Baseline-Modelle bieten Ihnen eine Referenz, anhand der Sie komplexere Modelle auf die Schnelle vergleichen können. Nehmen wir zum Beispiel an, Sie trainieren ein BERT-large-basiertes Modell und erhalten eine Treffergenauigkeit von 80 % auf Ihrem Validierungsdatensatz. Dadurch könnten Sie sich dazu veranlasst sehen, den Datensatz als schwierig abzutun, und es dabei belassen. Doch was wäre, wenn Sie wüssten, dass ein einfacher Klassifikator, der auf der logistischen Regression basiert, eine Treffergenauigkeit von 95 % erreicht? Dann würden Sie wahrscheinlich stutzig werden und Ihr Modell überarbeiten.

Beginnen wir unsere Analyse also damit, ein Baseline-Modell zu trainieren. Im Rahmen der Textklassifizierung stellt der *naive Bayes-Klassifikator* eine gute Ausgangsbasis dar, da er sehr einfach und schnell zu trainieren und ziemlich robust gegenüber Beeinträchtigungen in den Eingabedaten ist. Die Scikit-learn-Implementierung für den naiven Bayes-Klassifikator unterstützt standardmäßig allerdings keine Multilabel-Klassifizierung. Glücklicherweise können wir auch in diesem Fall auf die Scikit-multilearn-Bibliothek zurückgreifen und die Problemstellung als eine One-versus-Rest-Klassifizierungsaufgabe interpretieren, bei der wir L binäre Klassifikatoren für L verschiedene Labels trainieren. Hierzu verwenden wir einen Multilabel-Binarizer, wobei wir das Ergebnis in einer neuen Spalte (`label_ids`) in unse-

ren Trainingsdatensätzen festhalten. Mithilfe der `map()`-Methode können wir wie gewohnt die gesamte Verarbeitung bequem in einem Schritt erledigen:

```
def prepare_labels(batch):
    batch["label_ids"] = mlb.transform(batch["labels"])
    return batch

ds = ds.map(prepare_labels, batched=True)
```

Zur Bewertung unseres Klassifikators ziehen wir sowohl das mikro- als auch das makro-gemittelte F_1-Maß heran, wobei Ersteres die Leistung in Bezug auf die häufig vorkommenden Labels erfasst und Letzteres in Bezug auf alle Labels, ohne dabei die Häufigkeit dieser zu berücksichtigen. Da wir jedes Modell über unterschiedlich große Trainingsdatensätze hinweg evaluieren möchten, erstellen wir ein auf einer Liste basierendes `defaultdict`, in dem wir die Werte für die Maße für den jeweiligen Teildatensatz speichern:

```
from collections import defaultdict

macro_scores, micro_scores = defaultdict(list), defaultdict(list)
```

Wir sind nun endlich so weit, unser Baseline-Modell trainieren zu können! Hier ist der Code, mit dem wir das Modell trainieren und unseren Klassifikator anhand immer größerer Trainingssätze evaluieren können:

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import classification_report
from skmultilearn.problem_transform import BinaryRelevance
from sklearn.feature_extraction.text import CountVectorizer

for train_slice in train_slices:
    # Trainingsdaten-Slices und Testdaten erhalten
    ds_train_sample = ds["train"].select(train_slice)
    y_train = np.array(ds_train_sample["label_ids"])
    y_test = np.array(ds["test"]["label_ids"])
    # Einen einfachen Count-Vectorizer verwenden, um Texte
    #  in Form von Häufigkeiten der Tokens zu codieren
    count_vect = CountVectorizer()
    X_train_counts = count_vect.fit_transform(ds_train_sample["text"])
    X_test_counts = count_vect.transform(ds["test"]["text"])
    # Modell erstellen und trainieren
    classifier = BinaryRelevance(classifier=MultinomialNB())
    classifier.fit(X_train_counts, y_train)
    # Vorhersagen erstellen und evaluieren
    y_pred_test = classifier.predict(X_test_counts)
    clf_report = classification_report(
        y_test, y_pred_test, target_names=mlb.classes_, zero_division=0,
        output_dict=True)
    # Maße speichern
    macro_scores["Naive Bayes"].append(clf_report["macro avg"]["f1-score"])
    micro_scores["Naive Bayes"].append(clf_report["micro avg"]["f1-score"])
```

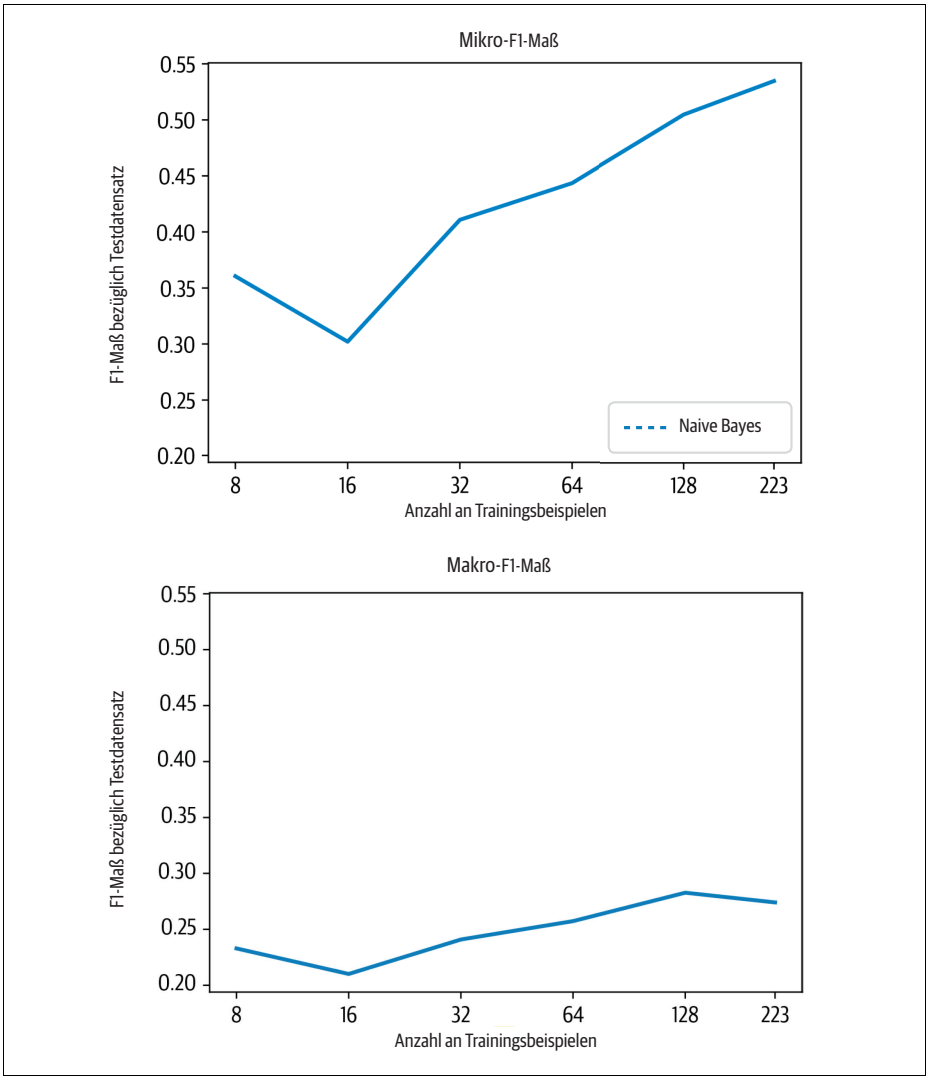
In diesem Codeabschnitt geschieht recht viel, daher sollten wir ihn etwas genauer betrachten. Zunächst wählen wir das Trainingsdaten-Slice (d.h. jeweils einen der unterschiedlich großen Trainingsdatensätze) aus und codieren die Labels. Im

nächsten Schritt codieren wir die Texte mithilfe eines Count-Vectorizer, der einen Vektor erstellt, dessen Länge der Größe des Vokabulars entspricht, wobei jeder Eintrag der Häufigkeit entspricht, mit der ein Token im Text vorkommt. Dieser Ansatz wird als *Bag-of-Words* bezeichnet, da jegliche Information über die Reihenfolge der Wörter verloren geht. Anschließend trainieren wir den Klassifikator und verwenden die Vorhersagen, die das Modell für den Testdatensatz trifft, um die mikro- und makro-gemittelten F_1-Werte über den Klassifizierungsbericht (`clf_report`) zu erhalten.

Mit der folgenden Hilfsfunktion können wir die Ergebnisse des Experiments visualisieren:

```
import matplotlib.pyplot as plt

def plot_metrics(micro_scores, macro_scores, sample_sizes, current_model):
    fig, (ax0, ax1) = plt.subplots(1, 2, figsize=(10, 4), sharey=True)

    for run in micro_scores.keys():
        if run == current_model:
            ax0.plot(sample_sizes, micro_scores[run], label=run, linewidth=2)
            ax1.plot(sample_sizes, macro_scores[run], label=run, linewidth=2)
        else:
            ax0.plot(sample_sizes, micro_scores[run], label=run,
                     linestyle="dashed")
            ax1.plot(sample_sizes, macro_scores[run], label=run,
                     linestyle="dashed")

    ax0.set_title("Mikro-F1-Maß")
    ax1.set_title("Makro-F1-Maß")
    ax0.set_ylabel("F1-Maß bezüglich Testdatensatz")
    ax0.legend(loc="lower right")
    for ax in [ax0, ax1]:
        ax.set_xlabel("Anzahl an Trainingsbeispielen")
        ax.set_xscale("log")
        ax.set_xticks(sample_sizes)
        ax.set_xticklabels(sample_sizes)
        ax.minorticks_off()
    plt.tight_layout()
    plt.show()

plot_metrics(micro_scores, macro_scores, train_samples, "Naive Bayes")
```

Beachten Sie, dass wir die Anzahl der Beispiele (d.h. die Stichprobengröße) auf einer logarithmischen Skala darstellen. Aus der Abbildung geht hervor, dass sich sowohl die mikro- als auch die makro-gemittelten Werte des F_1-Maßes verbessern, wenn wir die Anzahl der Trainingsbeispiele erhöhen. Bei derart kleinen Stichproben können die Ergebnisse auch leicht verzerrt sein, da die Kategorien in jedem Slice unterschiedlich verteilt sein können – dies ändert jedoch nichts daran, dass ein allgemeiner Zusammenhang zu erkennen ist, und darauf kommt es hier an. Sehen wir uns nun an, wie Transformer-basierte Ansätze im Vergleich zu unserem Baseline-Modell abschneiden.



Ansätze, wenn keine gelabelten Daten vorliegen

Der erste Ansatz, den wir in Betracht ziehen, ist die *Zero-Shot-Klassifizierung*, die für Situationen geeignet ist, in denen Sie überhaupt keine Daten vorliegen haben, die gelabelt sind. Dies ist in der Industrie erstaunlich häufig der Fall und kann vorkommen, weil es keine historischen Daten gibt, die mit Labels versehen sind, oder weil es schwierig ist, Labels für die Daten zu gewinnen. Wir werden in diesem Abschnitt ein wenig schummeln, da wir die Testdaten immer noch dazu verwenden, die Leistung zu evaluieren, aber keinerlei Daten zum Trainieren des Modells verwenden (sonst wäre es schwierig, einen Vergleich mit den folgenden Ansätzen anzustellen).

Das Ziel bei der Zero-Shot-Klassifizierung besteht darin, ein vortrainiertes Modell zu verwenden, ohne ein zusätzliches Feintuning auf Ihrem aufgabenspezifischen Korpus vornehmen zu müssen. Um eine bessere Vorstellung davon zu erhalten, wie dies funktionieren könnte, halten Sie sich noch einmal vor Augen, dass Sprachmodelle wie BERT darauf vortrainiert sind, maskierte Tokens in Texten aus Tausenden von Büchern und großen Wikipedia-Dumps vorherzusagen. Um ein fehlendes Token erfolgreich vorhersagen zu können, muss dem Modell die im Kontext vorliegende Thematik (engl. Topic) allerdings bekannt sein. Folglich können wir versuchen, das Modell dazu zu bringen, ein Dokument für uns zu klassifizieren (also die Thematik vorherzusagen), indem wir einen Satz wie diesen eingeben:

"This section was about the topic [MASK]."

Das Modell sollte dann einen passenden Vorschlag liefern können, um welche Thematik es in dem Dokument geht, da dies ein Text ist, der in der Form auch ganz normal im Datensatz, auf den das Modell vortrainiert wurde, vorkommt.[2]

Betrachten wir zur Veranschaulichung das folgende exemplarische Dilemma: Nehmen wir an, Sie haben zwei Kinder, von denen eines Autofilme mag, während das andere Tierfilme bevorzugt. Leider haben beide schon alle Filme, die Sie kennen, gesehen. Sie überlegen sich, eine Funktion zu entwickeln, die Ihnen Auskunft darüber gibt, um welche Thematik es in einem neuen Film geht. Naheliegenderweise greifen Sie für diese Aufgabe natürlich auf ein Transformer-Modell zurück. Als Erstes laden Sie das BERT-base-Modell in die `fill-mask`-Pipeline, die auf Basis eines maskierten Sprachmodells in der Lage ist, vorherzusagen, welcher Inhalt anstelle des Mask-Tokens stehen müsste:

```
from transformers import pipeline

pipe = pipeline("fill-mask", model="bert-base-uncased")
```

Als Nächstes verfassen wir eine kleine Filmbeschreibung, die wir als Prompt verwenden, wobei wir ein Wort maskieren. Mithilfe des Prompts wollen wir das Modell dazu bringen, dass es eine Klassifizierung vornimmt und die `fill-mask`-Pipeline letztlich die wahrscheinlichsten Tokens zurückgibt, mit denen die maskierte Stelle aufgefüllt werden könnte:

```
movie_desc = "The main characters of the movie madacascar \
are a lion, a zebra, a giraffe, and a hippo. "
prompt = "The movie is about [MASK]."

output = pipe(movie_desc + prompt)
for element in output:
    print(f"Token {element['token_str']}:\t{element['score']:.3f}%")
```

```
Token animals:  0.103%
Token lions:    0.066%
Token birds:    0.025%
Token love:     0.015%
Token hunting:  0.013%
```

2 Wir danken Joe Davison (*https://joeddav.github.io*) dafür, dass er uns diesen Ansatz vorgeschlagen hat.

Offensichtlich sagt das Modell nur Tokens voraus, die sich auf Tiere beziehen. Wir können die Aufgabe auch umformulieren und die Pipeline dazu veranlassen, uns nicht die wahrscheinlichsten Begriffe auszugeben, sondern die Wahrscheinlichkeit für einige wenige vorgegebene Begriffe. Um die Wahrscheinlichkeit für die Kategorien `cars` und `animals` zu erhalten, müssen wir sie als Targets an die Pipeline übergeben:

```
output = pipe(movie_desc + prompt, targets=["animals", "cars"])
for element in output:
    print(f"Token {element['token_str']}:\t{element['score']:.3f}%")
```

```
Token animals:  0.103%
Token cars:     0.001%
```

Wenig überraschend ist die vorhergesagte Wahrscheinlichkeit für das Token `cars` wesentlich geringer als für `animals`. Mal sehen, ob unser Ansatz auch mit einer Beschreibung funktioniert, in der es um Autos geht:

```
movie_desc = "In the movie transformers aliens \
can morph into a wide range of vehicles."

output = pipe(movie_desc + prompt, targets=["animals", "cars"])
for element in output:
    print(f"Token {element['token_str']}:\t{element['score']:.3f}%")
```

```
Token cars:     0.139%
Token animals:  0.006%
```

Es klappt! Es handelt sich hier lediglich um ein einfaches Beispiel, und wenn wir sichergehen wollen, dass es ohne Probleme funktioniert, sollten wir es gründlich testen. Aber es veranschaulicht den Kerngedanken vieler in diesem Kapitel besprochener Ansätze: einen Weg zu finden, ein vortrainiertes Modell für eine andere Aufgabe einzusetzen, ohne es dafür trainieren zu müssen. In diesem Fall haben wir einen Prompt, bei dem ein Wort maskiert ist, so ausgestaltet, dass wir direkt ein maskiertes Sprachmodell verwenden können, um eine Klassifizierung durchzuführen. Probieren wir nun aus, ob wir durch die Adaption eines feingetunten Modells für eine Aufgabe, die näher an der Textklassifizierung liegt, mehr erreichen können: die *Natural Language Inference* (NLI).

Ein maskiertes Sprachmodell für die Klassifizierung zu verwenden, ist zwar ein nützlicher Trick, allerdings lässt sich mit einem anderen Ansatz noch mehr herausholen: indem wir ein Modell verwenden, das auf der Grundlage einer Aufgabe trainiert wurde, die näher an der Klassifizierung liegt. Dafür eignet sich eine praktische Proxy-Aufgabe namens *Text Entailment* (im Deutschen auch Textuelle Inferenz bzw. Textuelles Schließen genannt). Beim Text Entailment muss das Modell feststellen, ob es wahrscheinlich ist, dass zwei Textpassagen zusammenhängen oder sich widersprechen. Die Modelle werden in der Regel mit Datensätzen wie dem Multi-Genre NLI Corpus (MNLI) oder dem Cross-Lingual NLI Corpus (XNLI) trainiert, um erkennen zu können, ob zwei Textpassagen in einem Zusammenhang (engl. Entailment) oder in einem Widerspruch zueinander stehen (engl. Contradiction).[3]

3 A. Williams, N. Nangia und S. R. Bowman, »A Broad-Coverage Challenge Corpus for Sentence Understanding Through Inference« (*https://arxiv.org/abs/1704.05426*), (2018); A. Conneau et al., »XNLI: Evaluating Cross-Lingual Sentence Representations« (*https://arxiv.org/abs/1809.05053*), (2018).

Jedes Beispiel in diesen Datensätzen besteht aus drei Teilen: einem vorgegebenen Text bzw. einer Prämisse (engl. Premise), einer Hypothese und einem Label, bei dem es sich um eines der folgenden handeln kann: `entailment`, `neutral` oder `contradiction`. Das Label `entailment` wird vergeben, wenn der Text der Hypothese in Bezug auf die Prämisse eindeutig wahr ist. Das Label `contradiction` wird verwendet, wenn die Hypothese unter der Prämisse mit Sicherheit falsch bzw. unzutreffend ist. Wenn keiner dieser Fälle zutrifft, wird das Label `neutral` zugewiesen. In Tabelle 9-1 finden Sie Beispiele für jeden dieser Fälle.

Tabelle 9-1: Die drei Kategorien im MLNI-Datensatz (aus dem Englischen übersetzt)

Prämisse	Hypothese	Label
Seine Lieblingsfarbe ist Blau.	Er steht auf Heavy-Metal-Musik.	`neutral`
Sie findet den Witz sehr lustig.	Sie findet den Witz überhaupt nicht lustig.	`contradiction`
Das Haus wurde erst kürzlich erbaut.	Das Haus ist neu.	`entailment`

Der Clou ist, dass wir ein auf dem MNLI-Datensatz trainiertes Modell nutzen können, um einen Klassifikator zu erstellen, für den wir überhaupt keine Labels benötigen! Der Grundgedanke dabei ist, dass wir den Text, den wir klassifizieren möchten, als Prämisse behandeln und die Hypothese wie folgt formulieren:

"This example is about {label}."

wobei wir den Namen der Kategorie anstelle von `label` einfügen. Der Entailment-Score sagt uns dann, wie wahrscheinlich es ist, dass sich die Prämisse auf die entsprechende Thematik bezieht. Wir können für eine beliebige Anzahl von Kategorien nacheinander so vorgehen. Der Nachteil dieses Ansatzes ist, dass für jede Kategorie ein Forward-Pass vorgenommen werden muss, wodurch er weniger effizient ist als ein standardmäßiger Klassifikator. Ein weiterer, etwas heikler Aspekt ist, dass die Treffergenauigkeit stark davon beeinflusst wird, welche Namen den Labels gegeben werden. Im Allgemeinen ist es am besten, Namen für die Labels zu wählen, die eine semantische Bedeutung haben. Wenn das Label zum Beispiel einfach `Kategorie 1` heißt, hat das Modell keinen Hinweis darauf, was dies bedeuten könnte, und kann dementsprechend auch nicht erkennen, ob es sich um einen Widerspruch (Contradiction) oder einen Zusammenhang (Entailment) handelt.

In der 🤗 Transformers-Bibliothek ist ein MNLI-Modell integriert, das die Zero-Shot-Klassifizierung unterstützt. Wir können es wie folgt über eine Pipeline initialisieren:

```
from transformers import pipeline

pipe = pipeline("zero-shot-classification", device=0)
```

Indem wir `device=0` setzen, stellen wir sicher, dass das Modell auf der GPU und nicht auf der gewöhnlichen CPU läuft, um den Inferenzvorgang zu beschleunigen. Damit wir einen Text klassifizieren können, müssen wir ihn einfach zusammen mit den Namen der Labels an die Pipeline übergeben. Zusätzlich können wir `multi_`

`label=True` setzen, um sicherzustellen, dass alle Ergebnisse zurückgegeben werden und nicht nur der wahrscheinlichste Wert, wie es bei der Klassifizierung, bei der nur ein Label vorhergesagt wird, der Fall ist:

```
sample = ds["train"][0]
print(f"Labels: {sample['labels']}")
output = pipe(sample["text"], all_labels, multi_label=True)
print(output["sequence"][:400])
print("\nVorhersagen:")

for label, score in zip(output["labels"], output["scores"]):
    print(f"{label}, {score:.2f}")
```

```
Labels: ['new model']
Add new CANINE model

# 🤗 New model addition

## Model description

Google recently proposed a new **C**haracter **A**rchitecture with **N**o
tokenization **I**n **N**eural **E**ncoders architecture (CANINE). Not only the
title is exciting:

> Pipelined NLP systems have largely been superseded by end-to-end neural
modeling, yet nearly all commonly-used models still require an explicit tokeni

Vorhersagen:
new model, 0.98
tensorflow or tf, 0.37
examples, 0.34
usage, 0.30
pytorch, 0.25
documentation, 0.25
model training, 0.24
tokenization, 0.17
pipeline, 0.16
```

Da wir einen Subword-Tokenizer verwenden, können wir sogar Code an das Modell übergeben. Die Tokenisierung mag dann zwar nicht sehr effizient sein, da nur ein geringer Teil des Pretraining-Datensatzes für die Zero-Shot-Pipeline aus Code-Snippets besteht, da aber Code teils auch aus natürlichen Wörtern besteht, stellt dies kein allzu großes Problem dar. Außerdem könnte der Codeblock wichtige Informationen enthalten, wie zum Beispiel das verwendete Framework (PyTorch oder TensorFlow).

Die Ausgabe verrät uns, dass das Modell mit hoher Wahrscheinlichkeit davon ausgeht, dass in dem vorliegenden Text ein neues Modell thematisiert wird, wobei es aber auch für die anderen Labels (bzw. Kategorien) relativ hohe Werte ermittelt hat. Ein Aspekt, der bei der Zero-Shot-Klassifizierung wichtig ist, ist die Domäne, in der wir uns bewegen. Die Texte, mit denen wir es hier zu tun haben, sind sehr

technisch und handeln überwiegend von Programmierung. Damit unterscheiden sie sich deutlich von der ursprünglichen Charakteristik der Texte, die im MNLI-Datensatz enthalten sind. Daher ist es wenig verwunderlich, dass die vorliegende Aufgabe für das Modell eine Herausforderung darstellt. In anderen Domänen schneidet das Modell vermutlich deutlich besser ab, je nachdem, wie gut die jeweilige Domäne durch die Trainingsdaten abgebildet wird.

Definieren wir nun eine Funktion, mit der ein einzelnes Beispiel durch die Zero-Shot-Pipeline geschickt wird, und übertragen sie anschließend auf den gesamten Validierungsdatensatz, indem wir die `map()`-Methode verwenden:

```
def zero_shot_pipeline(example):
    output = pipe(example["text"], all_labels, multi_label=True)
    example["predicted_labels"] = output["labels"]
    example["scores"] = output["scores"]
    return example

ds_zero_shot = ds["valid"].map(zero_shot_pipeline)
```

Da wir nun die Scores ermittelt haben, besteht der nächste Schritt darin, zu bestimmen, welche Labels den einzelnen Beispielen zugewiesen werden sollen. Sie können zwei verschiedene Ansätze verfolgen:

- Definieren Sie einen Schwellenwert und wählen Sie alle Labels aus, deren Score (bzw. Wahrscheinlichkeit) über diesem Schwellenwert liegt.
- Wählen Sie *k* wahrscheinlichsten Labels, d.h. die, die die höchsten Scores aufweisen.

Um herauszufinden, welcher der beiden Ansätze sich besser eignet, definieren wir eine Funktion `get_preds()`, für die wir angeben können, welcher der beiden Ansätze gewählt werden soll, um die Vorhersagen zu ermitteln:

```
def get_preds(example, threshold=None, topk=None):
    preds = []
    if threshold:
        for label, score in zip(example["predicted_labels"], example["scores"]):
            if score >= threshold:
                preds.append(label)
    elif topk:
        for i in range(topk):
            preds.append(example["predicted_labels"][i])
    else:
        raise ValueError("Set either `threshold` or `topk`.")
    return {"pred_label_ids": list(np.squeeze(mlb.transform([preds])))}
```

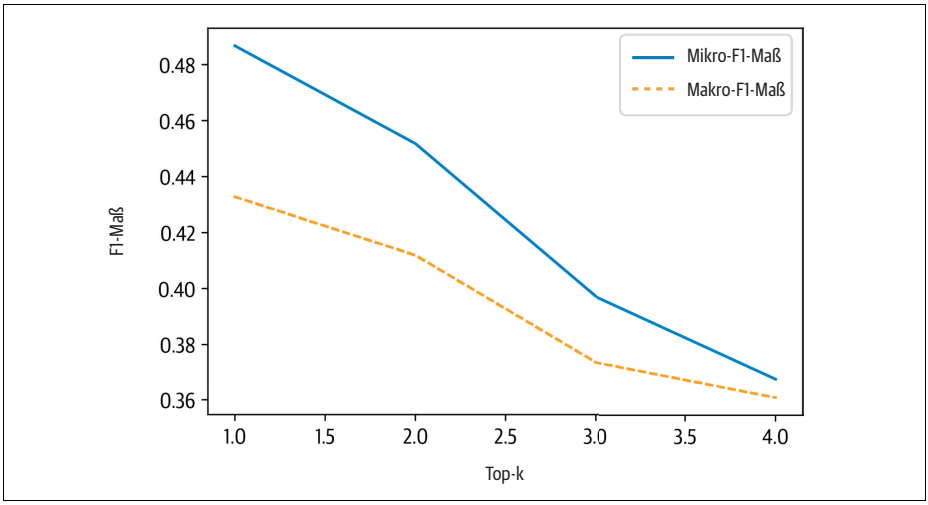
Als Nächstes erstellen wir eine zweite Funktion, `get_clf_report()`, die uns den in der Scikit-learn-Bibliothek verfügbaren Klassifizierungsbericht (Classification Report) auf der Grundlage eines vorgegebenen Datensatzes für die vorhergesagten Labels zurückgibt:

```
def get_clf_report(ds):
    y_true = np.array(ds["label_ids"])
    y_pred = np.array(ds["pred_label_ids"])
    return classification_report(
```

```
        y_true, y_pred, target_names=mlb.classes_, zero_division=0,
        output_dict=True)
```

Beginnen wir mit dem Ansatz, die wahrscheinlichsten *k* Labels auszuwählen (Top-*k*-Methode). Mithilfe der beiden Funktionen können wir *k* schrittweise erhöhen und anschließend ein Diagramm erstellen, in dem wir die Werte für das mikro- bzw. makro-gemittelte F_1-Maß in Bezug auf den Validierungsdatensatz abtragen:

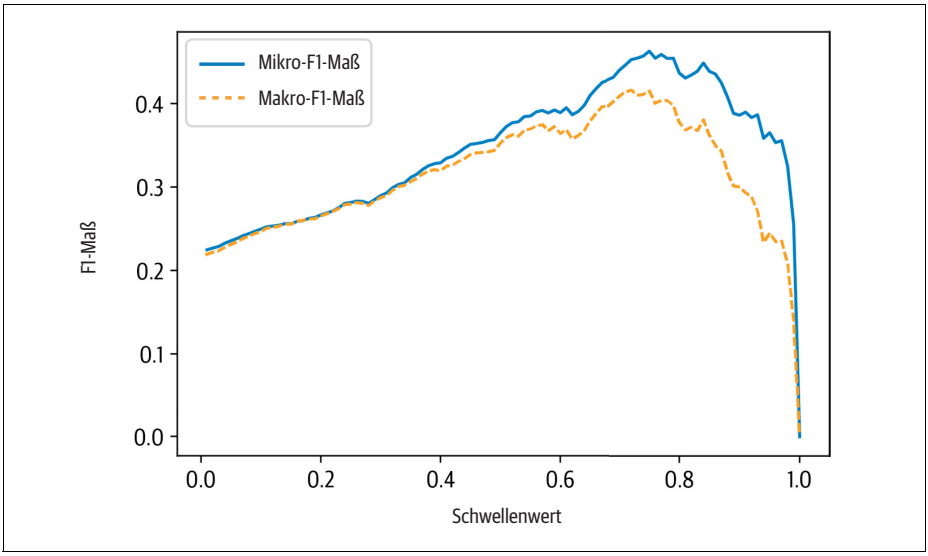
```
macros, micros = [], []
topks = [1, 2, 3, 4]
for topk in topks:
    ds_zero_shot = ds_zero_shot.map(get_preds, batched=False,
                                    fn_kwargs={'topk': topk})
    clf_report = get_clf_report(ds_zero_shot)
    micros.append(clf_report['micro avg']['f1-score'])
    macros.append(clf_report['macro avg']['f1-score'])

plt.plot(topks, micros, label='Mikro-F1-Maß')
plt.plot(topks, macros, label='Makro-F1-Maß')
plt.xlabel("Top-k")
plt.ylabel("F1-Maß")
plt.legend(loc='best')
plt.show()
```



Aus dem Diagramm können wir schließen, dass die Ergebnisse am besten ausfallen, wenn wir lediglich das Label auswählen, das den höchsten Score für ein Beispiel aufweist (Top-1). Angesichts der Tatsache, dass die meisten Beispiele in unseren Datensätzen nur ein Label aufweisen, ist das nicht verwunderlich. Vergleichen wir nun die Ergebnisse, die wir auf Basis des anderen Ansatzes, einen Schwellenwert festzulegen, erzielen. Vielleicht schaffen wir es ja mit diesem Ansatz, mehr als nur ein Label pro Beispiel vorherzusagen:

```
macros, micros = [], []
thresholds = np.linspace(0.01, 1, 100)
for threshold in thresholds:
```

```
    ds_zero_shot = ds_zero_shot.map(get_preds,
                                    fn_kwargs={"threshold": threshold})
    clf_report = get_clf_report(ds_zero_shot)
    micros.append(clf_report["micro avg"]["f1-score"])
    macros.append(clf_report["macro avg"]["f1-score"])

plt.plot(thresholds, micros, label="Mikro-F1-Maß")
plt.plot(thresholds, macros, label="Makro-F1-Maß")
plt.xlabel("Schwellenwert")
plt.ylabel("F1-Maß")
plt.legend(loc="best")
plt.show()
```




```
best_t, best_micro = thresholds[np.argmax(micros)], np.max(micros)
print(f'Bester Schwellenwert (Mikro): {best_t} mit F1-Maß {best_micro:.2f}.')
best_t, best_macro = thresholds[np.argmax(macros)], np.max(macros)
print(f'Bester Schwellenwert (Makro): {best_t} mit F1-Maß {best_macro:.2f}.')
```

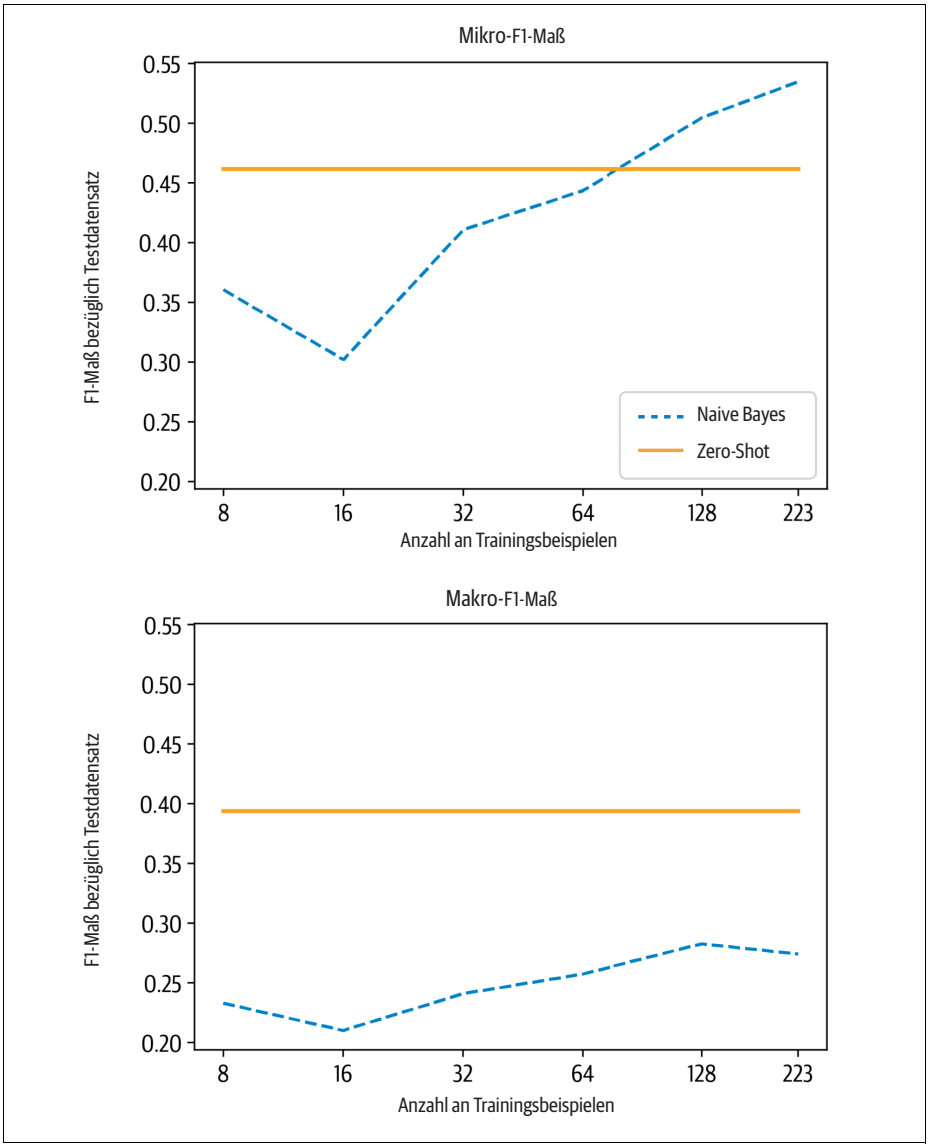
```
Bester Schwellenwert (Mikro): 0.75 mit F1-Maß 0.46.
Bester Schwellenwert (Makro): 0.72 mit F1-Maß 0.42.
```

Mit diesem Ansatz erzielen wir etwas schlechtere Ergebnisse im Vergleich zum Ansatz, nur das wahrscheinlichste Label auszuwählen (Top-1-Ansatz). In dem vorliegenden Diagramm können wir deutlich den Zielkonflikt (bzw. Trade-off) zwischen Precision und Recall erkennen. Wenn wir den Schwellenwert zu niedrig ansetzen, werden zu viele Vorhersagen getroffen, wodurch die Precision gering ausfällt. Wenn wir den Schwellenwert zu hoch ansetzen, erhalten wir kaum Vorhersagen und dementsprechend einen niedrigen Recall. Anhand des Diagramms können wir abschätzen, dass ein Schwellenwert von etwa 0,8 den goldenen Mittelweg zwischen diesen beiden Extremen darstellt.

Da der Top-1-Ansatz bessere Resultate verspricht, greifen wir auf diesen für die Zero-Shot-Klassifizierung zurück. Anschließend vergleichen wir die Ergebnisse mit

denen, die wir im Rahmen der naiven Bayes-Klassifizierung (unser Baseline-Modell) auf dem Testdatensatz erzielen konnten:

```
ds_zero_shot = ds['test'].map(zero_shot_pipeline)
ds_zero_shot = ds_zero_shot.map(get_preds, fn_kwargs={'topk': 1})
clf_report = get_clf_report(ds_zero_shot)
for train_slice in train_slices:
    macro_scores['Zero Shot'].append(clf_report['macro avg']['f1-score'])
    micro_scores['Zero Shot'].append(clf_report['micro avg']['f1-score'])

plot_metrics(micro_scores, macro_scores, train_samples, "Zero-Shot")
```



Beim Vergleich der Zero-Shot-Pipeline mit dem Baseline-Modell stellen wir zwei Dinge fest:

1. Wenn wir weniger als 50 gelabelte Beispiele einbezogen haben, schneidet die Zero-Shot-Pipeline deutlich besser ab als das Baseline-Modell.
2. Selbst bei mehr als 50 Beispielen erzielt die Zero-Shot-Pipeline eine bessere Leistung – zumindest, wenn das mikro- und makro-gemittelte F_1-Maß gemeinsam betrachtet werden. Die Ergebnisse für das mikro-gemittelte F_1-Maß zeigen uns, dass das Baseline-Modell bei Kategorien, die häufig vorkommen, zwar recht gut abschneidet, die Zero-Shot-Pipeline in den seltenen Kategorien, die beim makro-gemittelten F_1-Maß vergleichsweise stärker gewichtet werden, allerdings überragend ist, da sie keine Beispiele zum Lernen benötigt.

Vielleicht fällt Ihnen in diesem Abschnitt ein kleines Paradoxon auf: Obwohl wir davon sprechen, dass wir auf keine Labels zurückgreifen, benutzen wir dennoch einen Validierungs- und einen Testdatensatz. Sie dienen dazu, verschiedene Verfahren vorzustellen und die Ergebnisse miteinander vergleichbar zu machen. Selbst in einem realen Anwendungsfall ist es sinnvoll, eine Handvoll gelabelter Beispiele zu sammeln, um rasch einige Vergleiche bzw. Evaluierungen vornehmen zu können. Wichtig ist, dass wir nicht die Parameter des Modells an die Daten angepasst haben, sondern nur einige Hyperparameter.

Wenn Sie Schwierigkeiten haben, gute Ergebnisse mit Ihrem eigenen Datensatz zu erzielen, können Sie einige Maßnahmen ergreifen, mit denen die Zero-Shot-Pipeline verbessert werden kann:

- Aufgrund der Art und Weise, wie die Pipeline funktioniert, reagiert sie sehr empfindlich darauf, welche Namen den Labels gegeben werden. Wenn die Namen wenig Aussagekraft haben bzw. nicht leicht mit den Textinhalten in Verbindung gebracht werden können, dürfte die Leistungsfähigkeit der Pipeline eingeschränkt sein. Versuchen Sie entweder, andere Namen zu verwenden, oder verwenden Sie verschiedene Namen gleichzeitig und fassen Sie sie in einem zusätzlichen Schritt zusammen.
- Außerdem können Sie die Formulierung der Hypothese optimieren. Normalerweise wird die Hypothese als `hypothesis="This is example is about {}"` formuliert, Sie können der Pipeline aber auch jeden anderen denkbaren Text übergeben. Je nach Anwendungsfall könnte dies zu einer besseren Leistung beitragen.

Kommen wir nun zu dem Fall, dass uns einige wenige gelabelte Beispiele zur Verfügung stehen, die wir zum Trainieren eines Modells verwenden können.

Ansätze, wenn nur wenige gelabelte Daten zur Verfügung stehen

In den meisten Fällen können Sie im Rahmen eines NLP-Projekts zumindest auf einige wenige gelabelte Beispiele zurückgreifen. Die Labels können direkt von einem Kunden oder einem unternehmensübergreifenden Team bereitgestellt werden, oder Sie setzen sich einfach hin und labeln einige Beispiele selbst. Sogar für den vorherigen Ansatz haben wir einige mit Labels versehene Beispiele benötigt, um beurteilen zu können, wie gut der Zero-Shot-Ansatz funktioniert. In diesem Abschnitt sehen wir uns an, wie wir die wenigen, kostbaren gelabelten Beispiele, die wir zur Verfügung haben, am besten ausschöpfen können. Beginnen wir mit einer Technik, die als Datenaugmentierung (engl. Data Augmentation) bekannt ist und uns ermöglicht, die wenigen gelabelten Daten, die wir zur Verfügung haben, zu vervielfachen.

Datenaugmentierung

Eine einfache, aber effektive Möglichkeit, die Leistung von Textklassifikatoren zu steigern, wenn der vorliegende Datensatz klein ausfällt, ist die Anwendung von Techniken zur *Datenaugmentierung*, bei denen neue Trainingsbeispiele aus den vorhandenen generiert werden. Dies ist eine gängige Strategie in der Computer Vision, bei der Bilder zufällig verändert werden, ohne die eigentliche Bedeutung der Daten zu verändern (z. B. ist eine leicht gedrehte Katze immer noch eine Katze). Bei Texten gestaltet sich eine solche Datenaugmentierung etwas schwieriger, da sich die Bedeutung eines Texts komplett verändern kann, wenn Wörter oder Zeichen vertauscht werden. Beispielsweise unterscheiden sich die beiden Fragen »Sind Elefanten schwerer als Mäuse?« und »Sind Mäuse schwerer als Elefanten?« lediglich dadurch, dass zwei Wörter miteinander vertauscht wurden – doch sie liefern gegenteilige Antworten. Wenn der Text allerdings aus mehr als ein paar Sätzen besteht (wie bei unseren GitHub-Issues), dann hat das Rauschen, das durch diese Arten von Transformationen entsteht, in der Regel keine Auswirkungen auf die Vorhersagbarkeit des Labels. In der Praxis werden üblicherweise zwei Arten von Techniken zur Datenaugmentierung verwendet, auf die Sie sich stützen können:

Rückübersetzung (engl. Back Translation)
: Übersetzen Sie einen Text mithilfe maschineller Übersetzung aus der Ausgangssprache in eine oder mehrere Zielsprachen und übersetzen Sie ihn dann zurück in die Ausgangssprache. Eine Rückübersetzung funktioniert in der Regel am besten bei Sprachen, für die viele Ressourcen zur Verfügung stehen, oder bei Korpora, die nicht zu viele domänenspezifische Wörter enthalten.

Token-Perturbations
: Wählen Sie zufällig einen Text aus dem Trainingsdatensatz aus und führen Sie einfache Transformationen durch, wie z. B. zufällig Synonyme zu ersetzen (engl. Synonym Replacement), Wörter einzufügen (engl. Insertion), zu vertauschen (engl. Swap) oder zu löschen (engl. Deletion).[4]

Einige Beispiele für diese Transformationen finden Sie in Tabelle 9-2. Für eine detaillierte Erläuterung anderer im NLP verwendeter Techniken zur Datenaugmentierung empfehlen wir den Blogbeitrag von Amit Chaudhary »A Visual Survey of Data Augmentation in NLP« (*https://oreil.ly/j6euX*).

Tabelle 9-2: Verschiedene Arten von Datenaugmentierungstechniken für Textdaten

Augmentierung	Satz
Ohne	Megatron, selbst wenn du mich besiegen solltest, werden sich andere erheben, um deine Tyrannei zu beenden.
Durch Synonym ersetzen	Megatron, selbst wenn du mich töten solltest, werden sich andere erheben, um deine Tyrannei zu beenden.
Zufällig einfügen	Megatron, selbst wenn du mich besiegen solltest, werden sich andere Menschen erheben, um deine Tyrannei zu beenden.
Zufällig austauschen	Megatron, selbst wenn du mich besiegen solltest, werden andere sich erheben, um Tyrannei deine zu beenden.
Zufällig löschen	Megatron, selbst wenn du mich solltest, sich andere, um Tyrannei zu beenden.
Rückübersetzen (aus dem Englischen)	Selbst wenn Sie mich besiegen sollten, werden sich andere erheben, um Ihre Tyrannei zu beenden.

Für die Rückübersetzung können Sie auf maschinelle Übersetzungsmodelle wie M2M100 (*https://oreil.ly/gfJCq*) zurückgreifen, während Bibliotheken wie *NlpAug* (*https://oreil.ly/UVRci*) und *TextAttack* (*https://oreil.ly/NMtYi*) Ihnen verschiedene Möglichkeiten bieten, Token-Perturbations vorzunehmen. In diesem Abschnitt werden wir uns darauf beschränken, Wörter durch Synonyme zu ersetzen, da dies einfach zu implementieren ist und Ihnen den Grundgedanken der Datenaugmentierung verdeutlicht.

Hierzu greifen wir auf den `ContextualWordEmbsAug`-Augmenter der NlpAug-Bibliothek zurück, um die kontextuellen Worteinbettungen bzw. Contextual Word Embeddings des DistilBERT-Modells nutzen zu können, mit denen wir Wörter auf zufällige Weise durch Synonyme ersetzen. Beginnen wir zunächst mit einem einfachen Beispiel:

```
from transformers import set_seed
import nlpaug.augmenter.word as naw

set_seed(3)
aug = naw.ContextualWordEmbsAug(model_path="distilbert-base-uncased",
                                device="cpu", action="substitute")
```

4 J. Wei und K. Zou, »EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks« (*https://arxiv.org/abs/1901.11196*), (2019).

```
text = "Transformers are the most popular toys"
print(f"Ursprünglicher Text: {text}")
print(f"Augmentierter Text: {aug.augment(text)}")
```

```
Ursprünglicher Text: Transformers are the most popular toys
Augmentierter Text: transformers'the most popular toys
```

Im vorliegenden Beispiel wurde das Wort »are« durch einen Apostroph ersetzt, um ein neues synthetisches Trainingsbeispiel zu erzeugen. Die Augmentierung können wir wie folgt in eine einfache Funktion einbinden:

```
def augment_text(batch, transformations_per_example=1):
    text_aug, label_ids = [], []
    for text, labels in zip(batch["text"], batch["label_ids"]):
        text_aug += [text]
        label_ids += [labels]
        for _ in range(transformations_per_example):
            text_aug += [aug.augment(text)]
            label_ids += [labels]
    return {"text": text_aug, "label_ids": label_ids}
```

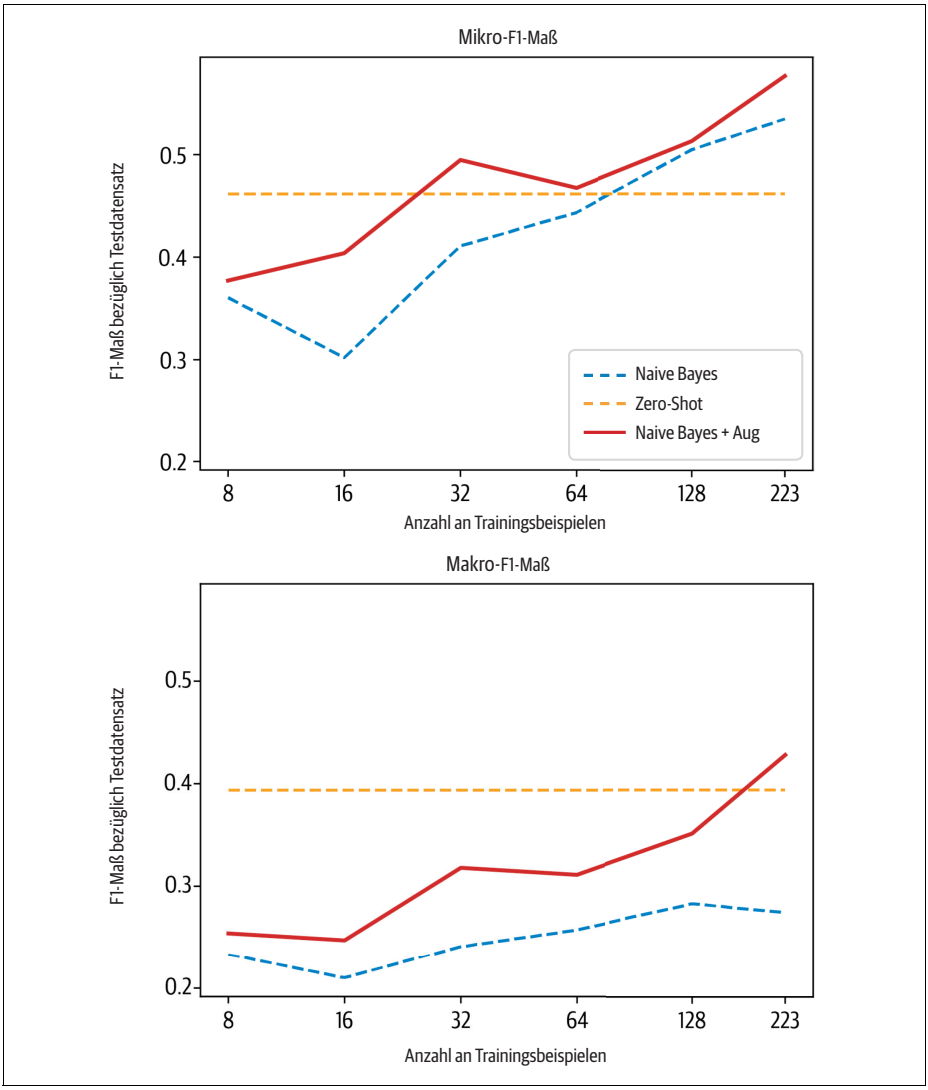
Wenn wir die Funktion nun an die `map()`-Methode übergeben, können wir über das Argument `transformations_per_example` eine beliebige Anzahl von neuen Beispielen generieren. Die Funktion können wir nun in unserem Code einfügen, mit dem wir den naiven Bayes-Klassifikator trainiert haben. Hierfür fügen wir einfach die nachfolgende Zeile nach der Stelle hinzu, an der wir jeweils einen der Trainingsdatensätze (Slice) ausgewählt haben:

```
ds_train_sample = ds_train_sample.map(augment_text, batched=True,
    remove_columns=ds_train_sample.column_names).shuffle(seed=42)
```

Nachdem wir diese Zeile hineinkopiert haben und die Analyse erneut ausführen, erhalten wir das folgende Diagramm:

```
plot_metrics(micro_scores, macro_scores, train_samples, "Naive Bayes + Aug")
```

Aus der Abbildung können wir entnehmen, dass die geringfügige Datenaugmentierung das F_1-Maß des naiven Bayes-Klassifikators um etwa 5 Prozentpunkte verbessert und dass er die Zero-Shot-Pipeline in Bezug auf das makro-gemittelte Maß übertrifft, sobald wir etwa 170 Trainingsbeispiele einbeziehen. Werfen wir nun einen Blick auf eine Methode, die sich auf die Verwendung der Embeddings großer Sprachmodelle stützt.



Embeddings als Nachschlagetabelle verwenden

Große Sprachmodelle wie GPT-3 haben auf eindrucksvolle Weise bewiesen, dass sie dazu imstande sind, Aufgaben auch dann lösen zu können, wenn nur wenige Daten vorliegen. Der Grund dafür ist, dass diese Modelle lernen, den Text in einer aussagekräftigen Form abzubilden, die Informationen über viele Dimensionen hinweg codieren, wie z.B. die Stimmungslage, die Thematik, die Textstruktur und mehr. Aus diesem Grund können die Einbettungen bzw. Embeddings von großen Sprachmodellen dazu verwendet werden, eine semantische Suchmaschine zu entwickeln, ähnliche Dokumente oder Kommentare zu finden oder sogar Texte zu klassifizieren.

In diesem Abschnitt werden wir einen Textklassifikator erstellen, der dem Endpunkt zur Klassifizierung der API von OpenAI (*https://oreil.ly/aMgIr*) nachempfunden ist. Wir gehen dabei in drei Schritten vor:

1. Wir verwenden das Sprachmodell, um alle gelabelten Texte einzubetten bzw. in Embeddings zu überführen.
2. Anschließend führen wir eine Suche auf Basis des Nächste-Nachbarn-Algorithmus (engl. Nearest Neighbour) über die gespeicherten Embeddings durch.
3. Im letzten Schritt treffen wir auf Basis der Labels der nächsten Nachbarn eine Vorhersage.

Diese Vorgehensweise wird in Abbildung 9-3 veranschaulicht: Zunächst werden die gelabelten Daten mithilfe eines Modells eingebettet und samt Labels gespeichert. Wenn ein neuer Text klassifiziert werden soll, wird er ebenfalls eingebettet und das Label auf der Grundlage der Labels der nächsten Nachbarn vergeben. Dabei ist es ist wichtig, die Anzahl der zu suchenden Nachbarn sorgfältig zu spezifizieren: Eine zu geringe Anzahl kann zu verfälschten Ergebnissen führen, und eine zu große Anzahl könnte wiederum bewirken, dass benachbarte Gruppen bzw. Kategorien eine zu starke Berücksichtigung erfahren.

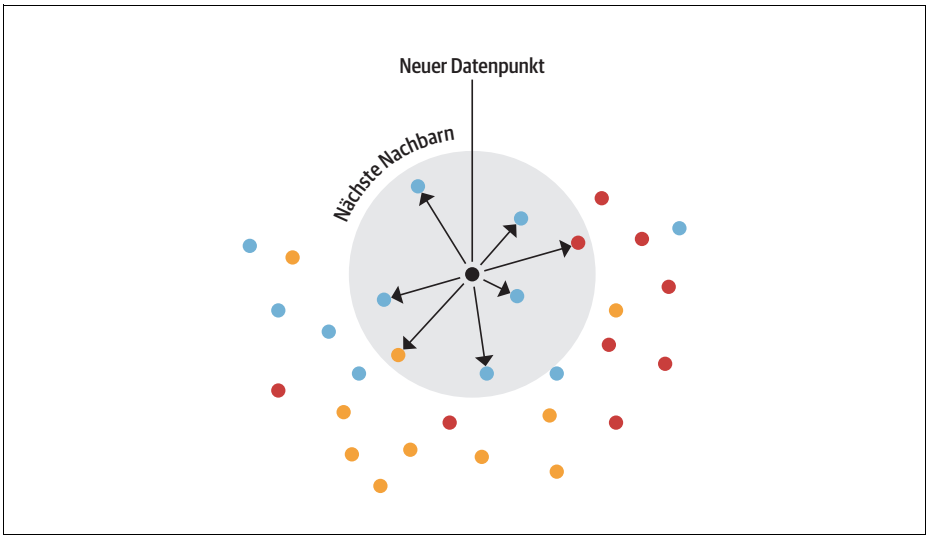


Abbildung 9-3: Wie Embeddings unter Verwendung des Nächste-Nachbarn-Suchalgorithmus als Nachschlagetabellen (engl. Lookup Tables) genutzt werden können

Das Schöne an diesem Ansatz ist, dass das Modell nicht erst feingetunt werden muss, um die wenigen verfügbaren gelabelten Datenpunkte nutzen zu können. Allerdings müssen Sie ein Modell auswählen, das geeignet ist, damit dieser Ansatz auch funktioniert, d.h., dass es idealerweise auf einer ähnlichen Domäne vortrainiert wurde, aus der Ihr Datensatz stammt.

Da das GPT-3-Modell nur über die API von OpenAI verfügbar ist, werden wir auf das GPT-2-Modell zurückgreifen, um den Ansatz auszuprobieren. Genauer gesagt

verwenden wir eine Variante von GPT-2, die auf Python-Code trainiert wurde, wodurch hoffentlich ein Teil des in unseren GitHub-Issues enthaltenen Kontexts erfasst werden kann.

Erstellen wir nun eine Hilfsfunktion, die eine Liste von Texten annimmt und jeden eingegebenen Text auf Basis des Modells in eine Darstellung in Form eines einzelnen Vektors überführt. Allerdings geben Transformer-Modelle wie GPT-2 in Wirklichkeit jeweils einen Embedding-Vektor je Token zurück. Bei dem Satz »Ich bin mit meinem Hund spazieren gegangen« würden wir zum Beispiel mehrere Embedding-Vektoren (bzw. Einbettungsvektoren) erhalten, einen für jedes Token. Was wir jedoch tatsächlich wollen, ist ein einziger Embedding-Vektor für den gesamten Satz (bzw. GitHub-Issue im Rahmen unserer Anwendung). Hierzu können wir eine Technik namens *Pooling* verwenden. Eine der einfachsten Pooling-Methoden besteht darin, eine Mittelwertbildung über die Token-Embeddings durchzuführen, was als *Mean Pooling* bezeichnet wird. Beim Mean Pooling müssen wir nur darauf achten, dass wir im Rahmen der Berechnung des arithmetischen Mittels keine Padding-Tokens einbeziehen, was sich mithilfe einer Attention-Mask bewerkstelligen lässt.

Wie das funktioniert, sehen wir, indem wir einen GPT-2-Tokenizer und ein Modell laden, die Mean-Pooling-Operation definieren und den gesamten Prozess in einer einfachen Funktion, `embed_text()`, einbinden:

```
import torch
from transformers import AutoTokenizer, AutoModel

model_ckpt = "miguelvictor/python-gpt2-large"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
model = AutoModel.from_pretrained(model_ckpt)

def mean_pooling(model_output, attention_mask):
    # Token-Embeddings extrahieren
    token_embeddings = model_output[0]
    # Attention-Mask berechnen
    input_mask_expanded = (attention_mask
                           .unsqueeze(-1)
                           .expand(token_embeddings.size())
                           .float())
    # Summe über Embeddings bilden (ohne maskierte Tokens zu berücksichtigen)
    sum_embeddings = torch.sum(token_embeddings * input_mask_expanded, 1)
    sum_mask = torch.clamp(input_mask_expanded.sum(1), min=1e-9)
    # Mittelwert als einzelnen Vektor ausgeben
    return sum_embeddings / sum_mask

def embed_text(examples):
    inputs = tokenizer(examples["text"], padding=True, truncation=True,
                       max_length=128, return_tensors="pt")
    with torch.no_grad():
        model_output = model(**inputs)
    pooled_embeds = mean_pooling(model_output, inputs["attention_mask"])
    return {"embedding": pooled_embeds.cpu().numpy()}
```

Nun sind wir so weit, dass wir die Embeddings für die jeweiligen Teildatensätze (Splits) ermitteln können. GPT-basierte Modelle besitzen allerdings kein Padding-Token. Aus diesem Grund sind wir dazu gezwungen, eines hinzuzufügen, ehe wir bereit sind, die Embeddings – wie in der Beschreibung des vorherigen Codes erläutert – über alle Batches hinweg zu ermitteln. Als Lösung verwenden wir als Padding-Token einfach das End-of-Sequence-(EOS-)Token, das ansonsten das Ende einer Sequenz markiert:

```
tokenizer.pad_token = tokenizer.eos_token
embs_train = ds["train"].map(embed_text, batched=True, batch_size=16)
embs_valid = ds["valid"].map(embed_text, batched=True, batch_size=16)
embs_test = ds["test"].map(embed_text, batched=True, batch_size=16)
```

Nachdem wir nun alle Embeddings ermittelt haben, müssen wir ein System einrichten, mit dem wir sie durchsuchen bzw. abfragen können. Hierzu könnten wir eine Funktion erstellen, die z. B. die Cosinusähnlichkeit zwischen dem Embedding eines neuen Texts, den wir abfragen, und den auf Basis des Trainingsdatensatzes ermittelten Embeddings berechnet. Alternativ können wir eine in der 🤗 Datasets-Bibliothek integrierte Struktur verwenden, die *FAISS-Index* genannt wird.[5] Wir sind der FAISS-Bibliothek bereits in Kapitel 7 begegnet. Sie können sich den Algorithmus als eine Suchmaschine für Embeddings vorstellen, wobei wir uns gleich noch genauer ansehen werden, wie er funktioniert. Um einen FAISS-Index zu erstellen, können wir ein bereits bestehendes Feld des Dataset-Objekts verwenden und die Funktion `add_faiss_index()` heranziehen. Zudem können wir neue Embeddings mithilfe der `add_faiss_index_from_external_arrays()`-Funktion in das Dataset-Objekt laden. Verwenden wir die erstgenannte Funktion, um dem Dataset-Objekt unsere für den Trainingsdatensatz ermittelten Embeddings hinzuzufügen:

```
embs_train.add_faiss_index("embedding")
```

Auf diese Weise wurde ein neuer FAISS-Index namens `embedding` erstellt. Wir können nun eine Suche mithilfe des Nächsten-Nachbarn-Algorithmus durchführen, indem wir die Funktion `get_nearest_examples()` aufrufen. Sie liefert uns die am nächsten liegenden Nachbarn sowie den entsprechenden Matching-Score für jeden Nachbarn. Dabei müssen wir neben dem Embedding der Abfrage auch die Anzahl der nächsten Nachbarn angeben, die abgerufen werden sollen. Probieren wir es aus und werfen wir einen Blick auf die Dokumente (bzw. Issues), die einem Beispiel am nächsten sind:

```
i, k = 0, 3 # Die erste Abfrage und die 3 nächsten Nachbarn auswählen
# Dient dem Entfernen von Zeilenumbrüchen im Text für kompaktere Darstellung:
rn, nl = "\r\n\r\n", "\n"

query =  np.array(embs_valid[i]["embedding"], dtype=np.float32)
scores, samples = embs_train.get_nearest_examples("embedding", query, k=k)

print(f"LABELS DER ABFRAGE: {embs_valid[i]['labels']}")
print(f"ABFRAGETEXT:\n{embs_valid[i]['text'][:200].replace(rn, nl)} [...]\n")
```

5 J. Johnson, M. Douze und H. Jégou, »Billion-Scale Similarity Search with GPUs« (*https://arxiv.org/abs/1702.08734*), (2017).

```
print("="*50)
print(f"Abgerufene Dokumente:")
for score, label, text in zip(scores, samples["labels"], samples["text"]):
    print("="*50)
    print(f"TEXT:\n{text[:200].replace(rn, nl)} [...]")
    print(f"SCORE: {score:.2f}")
    print(f"LABELS: {label}")
```

```
LABELS DER ABFRAGE: ['new model']
ABFRAGETEXT:
Implementing efficient self attention in T5

# 🤗 New model addition
My teammates and I (including @ice-americano) would like to use efficient self
attention methods such as Linformer, Performer and [...]

==================================================
Abgerufene Dokumente:
==================================================
TEXT:
Add Linformer model

# 🤗 New model addition
## Model description
### Linformer: Self-Attention with Linear Complexity
Paper published June 9th on ArXiv: https://arxiv.org/abs/2006.04768
La [...]
SCORE: 54.92
LABELS: ['new model']
==================================================
TEXT:
Add FAVOR+ / Performer attention

# 🤗 FAVOR+ / Performer attention addition
Are there any plans to add this new attention approximation block to
Transformers library?
## Model description
The n [...]
SCORE: 57.90
LABELS: ['new model']
==================================================
TEXT:
Implement DeLighT: Very Deep and Light-weight Transformers

# 🤗 New model addition
## Model description
DeLight, that delivers similar or better performance than transformer-based
models with sign [...]
SCORE: 60.12
LABELS: ['new model']
```

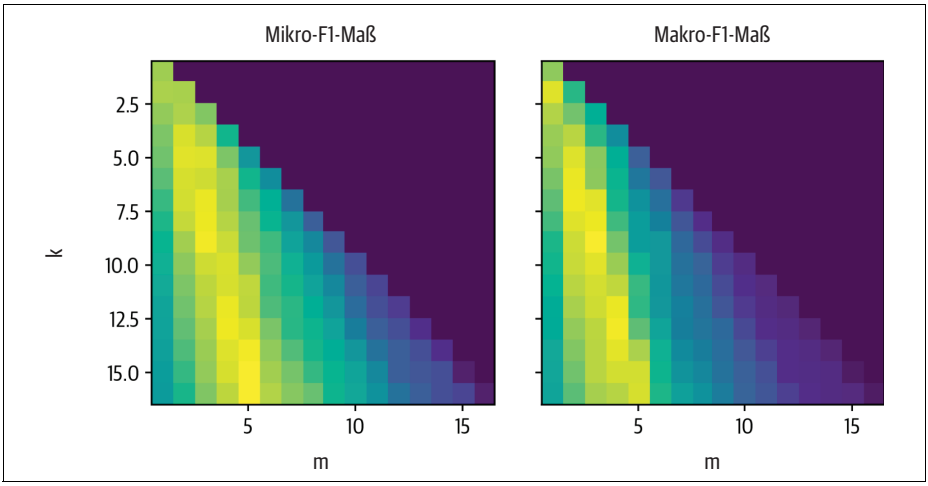
Sehr schön! Das entspricht genau dem, was wir uns erhofft haben: Die drei abgerufenen Dokumente, die wir über die Suche mittels der Embeddings erhalten haben, weisen alle die gleichen Labels auf, und wir können bereits an ihren Titeln erkennen, dass sie alle sehr ähnlich sind. Sowohl die Abfrage als auch die abgerufenen Dokumente beziehen sich auf das Hinzufügen neuer und effizienter Transformer-

Modelle. Es bleibt jedoch die Frage, welcher Wert für k optimalerweise gewählt werden sollte. Und auch die Frage, wie wir dann die Labels der abgerufenen Dokumente verdichten sollten? Sollten wir z. B. drei Dokumente abrufen und anschließend alle Labels zuordnen, die mindestens zweimal vorkommen? Oder sollten wir uns für 20 Dokumente entscheiden und alle Labels verwenden, die mindestens fünfmal vorhanden sind? Gehen wir dieser Frage systematisch auf den Grund: Wir probieren verschiedene Werte für k aus und variieren dann den Schwellenwert $m < k$, mit dem wir die Labels mithilfe einer Hilfsfunktion zuordnen können. Um später beurteilen zu können, welche Kombination von Werten am besten funktioniert hat, halten wir die Werte für die Leistung in Bezug auf das makro- und das mikro-gemittelte F_1-Maß für alle verschiedenen Kombinationen fest. Anstatt die Schleife über jedes einzelne Beispiel im Validierungsdatensatz laufen zu lassen, können wir die Funktion `get_nearest_examples_batch()` verwenden, die ein Batch von Abfragen erwartet:

```
def get_sample_preds(sample, m):
    return (np.sum(sample["label_ids"], axis=0) >= m).astype(int)

def find_best_k_m(ds_train, valid_queries, valid_labels, max_k=17):
    max_k = min(len(ds_train), max_k)
    perf_micro = np.zeros((max_k, max_k))
    perf_macro = np.zeros((max_k, max_k))
    for k in range(1, max_k):
        for m in range(1, k + 1):
            _, samples = ds_train.get_nearest_examples_batch("embedding",
                                                             valid_queries, k=k)
            y_pred = np.array([get_sample_preds(s, m) for s in samples])
            clf_report = classification_report(valid_labels, y_pred,
                target_names=mlb.classes_, zero_division=0, output_dict=True)
            perf_micro[k, m] = clf_report["micro avg"]["f1-score"]
            perf_macro[k, m] = clf_report["macro avg"]["f1-score"]
    return perf_micro, perf_macro
```

Um zu erkennen, welche Kombination von Werten in Bezug auf alle Trainingsbeispiele zu dem besten Ergebnis führt, erstellen wir zwei Diagramme, in denen wir die Ergebnisse für alle vorgegebenen Kombinationen von k und m darstellen:

```
valid_labels = np.array(embs_valid["label_ids"])
valid_queries = np.array(embs_valid["embedding"], dtype=np.float32)
perf_micro, perf_macro = find_best_k_m(embs_train, valid_queries, valid_labels)

fig, (ax0, ax1) = plt.subplots(1, 2, figsize=(10, 3.5), sharey=True)
ax0.imshow(perf_micro)
ax1.imshow(perf_macro)

ax0.set_title("Mikro-F1-Maß")
ax0.set_ylabel("k")
ax1.set_title("Makro-F1-Maß")
for ax in [ax0, ax1]:
    ax.set_xlim([0.5, 17 - 0.5])
    ax.set_ylim([17 - 0.5, 0.5])
    ax.set_xlabel("m")
plt.show()
```



Die Diagramme offenbaren, dass es ein Muster gibt: Wird m für einen gegebenen Wert von k zu groß oder zu klein gewählt, führt dies zu suboptimalen Ergebnissen. Das beste Ergebnis wird erzielt, wenn die beiden Werte so gewählt werden, dass ihr Quotient ungefähr $m/k = 1/3$ beträgt. Finden wir nun heraus, welche Werte für k und m insgesamt am besten abschneiden:

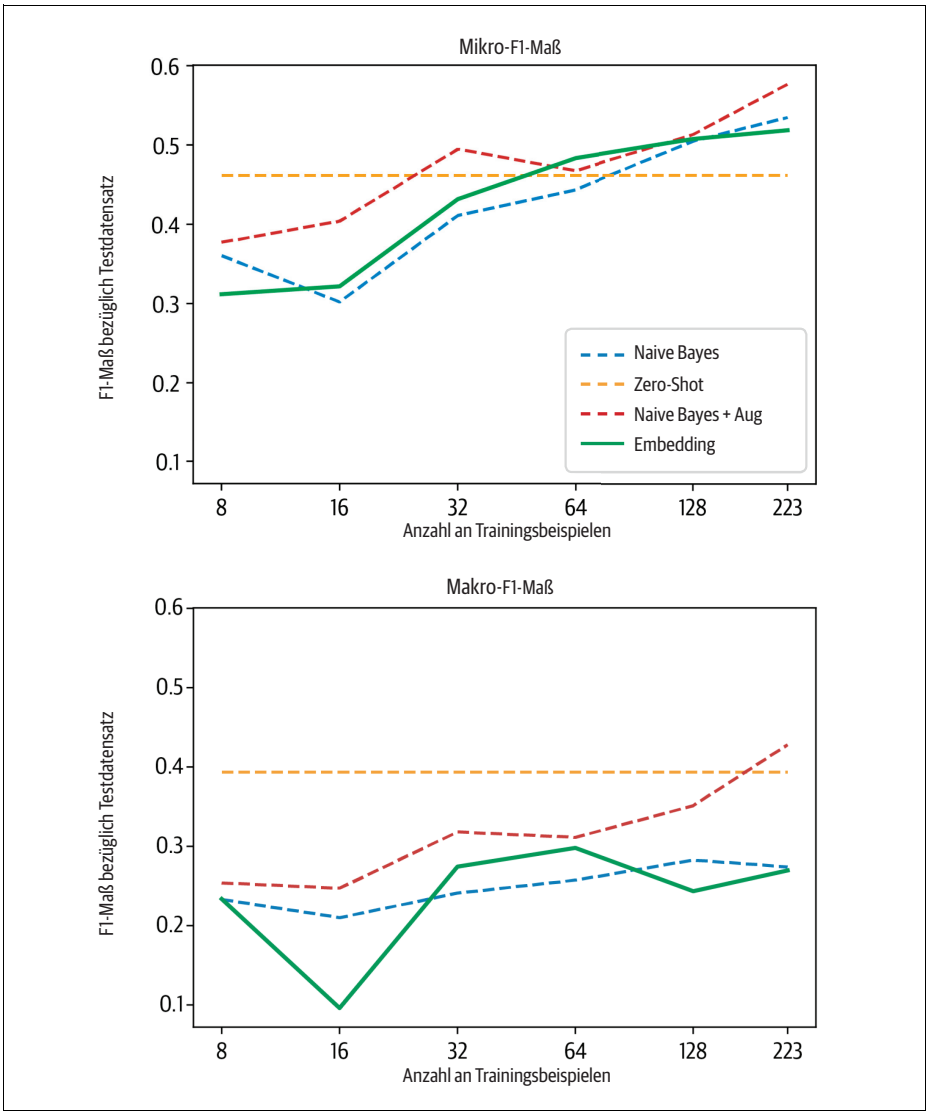
```
k, m = np.unravel_index(perf_micro.argmax(), perf_micro.shape)
print(f"Bester Wert für k: {k}, bester Wert für m: {m}")
```

```
Bester Wert für k: 15, bester Wert für m: 5
```

Das Ergebnis im Rahmen dieses Ansatzes fällt am besten aus, wenn wir $k = 15$ und $m = 5$ wählen, bzw. anders ausgedrückt, wenn wir die 15 nächsten Nachbarn abrufen und dann die Labels zuweisen, die mindestens fünfmal vorkommen. Da sich unsere Methode, wie wir die besten Werte ermittelt haben, um die Embeddings abzufragen, bewährt hat, können wir nun nach dem gleichen Schema wie beim naiven Bayes-Klassifikator vorgehen, indem wir die Slices des Trainingsdatensatzes durchlaufen und die Leistung beurteilen. Bevor wir den Datensatz in Slices aufteilen können, müssen wir den Index löschen (und neu erstellen), da wir den FAISS-Index nicht wie den Datensatz aufteilen können. Den Rest lassen wir unverändert, außer dass wir den Validierungsdatensatz verwenden, um die besten Werte für k und m zu ermitteln:

```
embs_train.drop_index("embedding")
test_labels = np.array(embs_test["label_ids"])
test_queries = np.array(embs_test["embedding"], dtype=np.float32)

for train_slice in train_slices:
    # FAISS-Index auf Basis von Trainingsslice erstellen
    embs_train_tmp = embs_train.select(train_slice)
    embs_train_tmp.add_faiss_index("embedding")
    # Besten Werte für k und m anhand des Validierungsdatensatzes ermitteln
    perf_micro, _ = find_best_k_m(embs_train_tmp, valid_queries, valid_labels)
```

```
    k, m = np.unravel_index(perf_micro.argmax(), perf_micro.shape)
    # Vorhersagen für Testdatensatz erhalten
    _, samples = embs_train_tmp.get_nearest_examples_batch("embedding",
                                                            test_queries,
                                                            k=int(k))
    y_pred = np.array([get_sample_preds(s, m) for s in samples])
    # Vorhersagen evaluieren
    clf_report = classification_report(test_labels, y_pred,
        target_names=mlb.classes_, zero_division=0, output_dict=True,)
    macro_scores["Embedding"].append(clf_report["macro avg"]["f1-score"])
    micro_scores["Embedding"].append(clf_report["micro avg"]["f1-score"])

plot_metrics(micro_scores, macro_scores, train_samples, "Embedding")
```



Der Ansatz, die Embeddings als Nachschlagetabelle zu nutzen, ist hinsichtlich des mikro-gemittelten F_1-Maßes zu den vorherigen Ansätzen durchaus konkurrenzfähig, obwohl es nur zwei »lernbare« Parameter, k und m, gibt. Hinsichtlich des makro-gemittelten F_1-Maßes schneidet er jedoch etwas schlechter ab.

Diese Ergebnisse sind allerdings mit Vorsicht zu genießen. Welche Methode am besten funktioniert, hängt stark von der jeweiligen Domäne ab. Die Daten, mit denen die Zero-Shot-Pipeline trainiert wurde, unterscheiden sich deutlich von dem GitHub-Issue-Datensatz, mit dem wir sie verwenden. Dieser enthält jede Menge Programmcode, mit dem das Modell wahrscheinlich noch nicht oft in Berührung gekommen ist. Für eine gängigere Aufgabe wie die Sentimentanalyse von Rezensionen könnte die Pipeline bedeutend besser geeignet sein. Auch die Qualität der Embeddings hängt von dem Modell und den Daten ab, mit denen es trainiert wurde. Wir haben rund ein halbes Dutzend verschiedener Modelle ausprobiert, z.B. `sentence-transformers/stsb-roberta-large`, das auf qualitativ hochwertige Satzeinbettungen trainiert wurde, sowie `microsoft/codebert-base` und `dbernsohn/roberta-python`, die mittels Programmcode und Dokumentationen trainiert wurden. Für unseren speziellen Anwendungsfall hat sich das auf Python-Code trainierte GPT-2-Modell als am besten erwiesen.

Da Sie nichts an dem Code ändern müssen, außer den Namen des Modell-Checkpoints auszutauschen, um ein anderes Modell zu testen, können Sie innerhalb kurzer Zeit einige Modelle ausprobieren, sobald Sie die Evaluierungspipeline eingerichtet haben.

Vergleichen wir nun diesen einfachen Trick, auf Embeddings zurückzugreifen, mit dem Ansatz, ein Transformer-Modell auf der Grundlage der nur begrenzt verfügbaren gelabelten Daten feinzutunen.

Effiziente Ähnlichkeitssuche mit FAISS

Wir sind der FAISS-Bibliothek bereits in Kapitel 7 begegnet, wo wir sie zum Abrufen von Dokumenten über die DPR-Embeddings verwendet haben. Im Folgenden erklären wir kurz, wie die FAISS-Bibliothek funktioniert und warum sie ein leistungsstarkes Tool innerhalb der ML-Toolbox ist.

Wir sind es gewohnt, schnelle Textabfragen auf riesigen Datenbeständen wie Wikipedia oder im Internet mithilfe von Suchmaschinen wie Google durchzuführen. Wenn wir einen Text in Embeddings umwandeln, wollen wir eine ähnliche Performance erreichen. Die Methoden, die eingesetzt werden, um Textabfragen zu beschleunigen, lassen sich jedoch nicht auf Embeddings übertragen.

Um die Textsuche zu beschleunigen, erstellen wir in der Regel einen invertierten Index, der die Begriffe den Dokumenten zuordnet. Ein invertierter Index funktioniert wie ein Index am Ende eines Buchs: Jedes Wort wird den Seiten (bzw. in unserem Fall den Dokumenten) zugeordnet, in denen es vorkommt. Wenn wir später eine Abfrage vornehmen, können wir auf diese Weise schnell herausfinden, in welchen Dokumenten die Suchbegriffe vorkommen. Dieser Ansatz funktioniert gut bei dis-

kreten Objekten wie Wörtern, aber nicht bei kontinuierlichen Objekten wie Vektoren. Jedes Dokument hat wahrscheinlich einen eindeutigen bzw. anderen Vektor, sodass der Index nie mit einem neuen Vektor übereinstimmen wird. Anstatt nach exakten Übereinstimmungen zu suchen, müssen wir nach naheliegenden bzw. ähnlichen Übereinstimmungen suchen.

Wenn wir die Vektoren in einer Datenbank finden wollen, die einem Abfragevektor am ähnlichsten sind, müssen wir theoretisch den Abfragevektor mit jedem der n Vektoren in der Datenbank vergleichen. Im Falle einer kleinen Datenbank, wie wir sie in diesem Kapitel nutzen, ist das kein Problem, aber wenn wir dies auf Tausende oder gar Millionen von Einträgen skalieren würden, müssten wir eine Weile warten, bis jede Abfrage verarbeitet wurde.

Bei FAISS wird dieser Problematik mit verschiedenen Tricks begegnet. Die Hauptidee besteht darin, den Datensatz zu partitionieren. Wenn wir den Abfragevektor nur mit einer Teilmenge der Datenbank vergleichen müssen, können wir den Prozess erheblich beschleunigen. Doch wenn wir den Datensatz nur zufällig partitionieren, wie können wir dann entscheiden, welche Partition durchsucht werden soll? Und welche Garantien gibt es, dass wir die ähnlichsten Einträge finden? Offensichtlich muss es eine bessere Lösung geben: Die Anwendung von k-Means-Clustering auf den Datensatz! Dadurch werden die Embeddings nach Ähnlichkeit in Gruppen geclustert. Außerdem erhalten wir für jede Gruppe einen geometrischen Schwerpunkt- bzw. Mittelpunktvektor (engl. Centroid Vector), der dem Durchschnitt aller Mitglieder der Gruppe entspricht (siehe Abbildung 9-4).

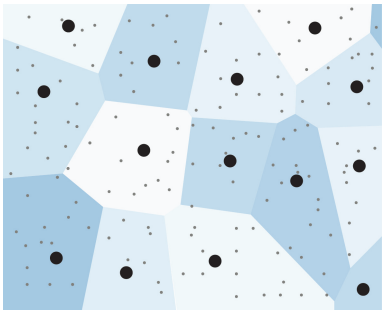
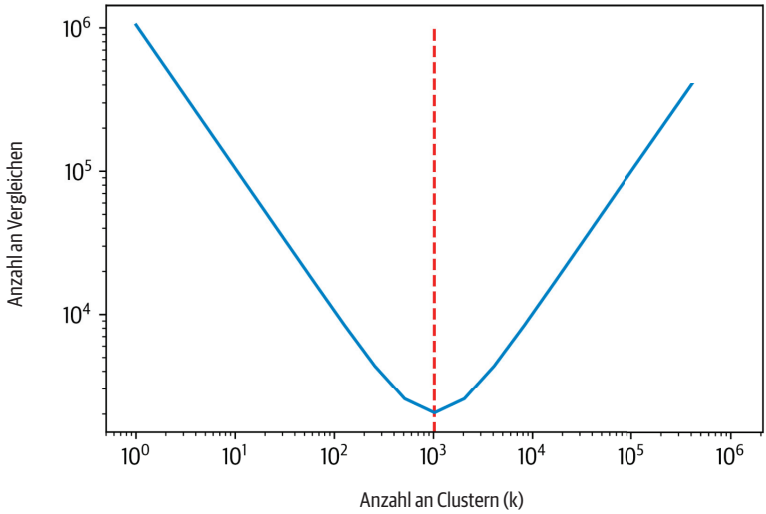


Abbildung 9-4: Die Struktur eines FAISS-Index: Die grauen Punkte stellen Datenpunkte dar, die dem Index hinzugefügt wurden, die fett gedruckten schwarzen Punkte sind die Clusterzentren, die im Rahmen des k-Means-Clustering ermittelt wurden, und die farbigen Bereiche stellen die Bereiche dar, die jeweils zu einem Clusterzentrum gehören.

Sobald eine solche Gruppierung vorgenommen wurde, gestaltet sich die Suche unter den n Vektoren bedeutend einfacher: Wir suchen zunächst unter den k Clusterzentren (engl. Centroids) nach demjenigen, der unserer Abfrage am ähnlichsten ist (k Vergleiche), und führen dann eine Suche innerhalb der Gruppe durch (um $\frac{k}{n}$ Elemente zu vergleichen). Dadurch verringert sich die Anzahl der Vergleiche, die vorgenommen werden müssen, von n auf $k + \frac{n}{k}$. Doch welcher Wert für k ist am besten? Wird er zu klein gewählt, enthält jede Gruppe relativ viele Beispiele, für die wir in dem zweiten Schritt noch einen Vergleich anstellen müssen, und wenn k zu hoch angesetzt wird, gibt es viele Clusterzentren, die wir durchsuchen bzw. deren

Beispiele wir vergleichen müssen. Wenn wir das Minimum der Funktion $f(k) = k + \frac{n}{k}$ in Abhängigkeit von k ermitteln, erhalten wir als optimalen Wert $k = \sqrt{n}$. Dies wird auch aus dem folgenden Diagramm deutlich, wobei wir $n = 2^{20}$ wählen.



In dem Diagramm wird die Anzahl der Vergleiche in Abhängigkeit von der Anzahl der Cluster dargestellt. Wir suchen nach dem Minimum dieser Funktion, d. h. dem Punkt, an dem wir die wenigsten Vergleiche durchführen müssen. Wie wir sehen können, liegt das Minimum genau dort, wo wir es erwartet haben, nämlich bei $\sqrt{2^{20}} = 2^{10} = 1{,}024$.

Zusätzlich dazu, dass die Abfragen mittels Partitionierung beschleunigt werden, können Sie mit der FAISS-Bibliothek auch GPUs nutzen, um eine weitere Beschleunigung zu erzielen. Sollte der Speicher zu einem Engpass werden, gibt es auch mehrere Möglichkeiten, die Vektoren mithilfe fortschrittlicher Quantisierungsverfahren zu komprimieren. Wenn Sie die FAISS-Bibliothek für Ihr Projekt verwenden möchten, finden Sie im Repository einen einfachen Leitfaden (*https://oreil.ly/QmvzR*), der Ihnen hilft, die richtigen Methoden für Ihren Anwendungsfall auszuwählen.

Eines der größten Projekte, bei dem die FAISS-Bibliothek verwendet wurde, war die Erstellung des CCMatrix-Korpus durch Facebook (*https://oreil.ly/ennlr*). Die Autoren verwendeten mehrsprachige Embeddings, um vergleichbare Sätze in verschiedenen Sprachen zu finden. Mit diesem riesigen Korpus wurde anschließend M2M100 (*https://oreil.ly/XzSH9*) trainiert, ein großes maschinelles Übersetzungsmodell, das in der Lage ist, unmittelbar zwischen 100 verschiedenen Sprachen zu übersetzen.

Ein standardmäßiges Transformer-Modell feintunen

Wenn wir auf gelabelte Daten zugreifen können, dann können wir auch einfach das Naheliegendste tun: ein vortrainiertes Transformer-Modell feintunen. In diesem Abschnitt verwenden wir als Ausgangspunkt den standardmäßigen (engl. Vanilla) BERT-Checkpoint. Im Anschluss untersuchen wir, wie sich das Feintuning des Sprachmodells auf seine Leistung auswirkt.

Bei vielen Anwendungen ist es eine gute Idee, mit einem vortrainierten BERT-basierten Modell zu beginnen. Wenn sich die Domäne Ihres Korpus jedoch erheblich vom Pretraining-Korpus unterscheidet (in der Regel ist das Wikipedia), sollten Sie einen Blick auf die unzähligen Modelle werfen, die auf dem Hugging Face Hub verfügbar sind. Die Chancen stehen gut, dass jemand bereits ein Modell auf Ihre Domäne vortrainiert hat!

Beginnen wir damit, den vortrainierten Tokenizer zu laden, unseren Datensatz zu tokenisieren und die Spalten zu entfernen, die wir nicht benötigen, um das Modell zu trainieren und zu evaluieren:

```
import torch
from transformers import (AutoTokenizer, AutoConfig,
                          AutoModelForSequenceClassification)

model_ckpt = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)

def tokenize(batch):
    return tokenizer(batch["text"], truncation=True, max_length=128)
ds_enc = ds.map(tokenize, batched=True)
ds_enc = ds_enc.remove_columns(['labels', 'text'])
```

Die Verlustfunktion erwartet in diesem Fall, dass mehrere Labels klassifiziert werden (Multilabel-Klassifizierung) und dass die Labels vom Typ Float sind, da sie auch die Wahrscheinlichkeit der Kategorien anstelle diskreter Labels akzeptiert. Daher müssen wir den Typ der `label_ids`-Spalte verändern. Da sich eine elementweise Änderung des Formats einer Spalte nicht gut mit dem typisierten Format von Arrow verträgt, müssen wir unser Vorgehen ein wenig anpassen: Zunächst erstellen wir eine neue Spalte, die die Labels enthält, wobei das Format der Spalte aus dem ersten Element abgeleitet wird. Dann löschen wir die ursprüngliche Spalte und benennen die neue Spalte so um, dass sie die ursprüngliche Spalte ersetzt:

```
ds_enc.set_format("torch")
ds_enc = ds_enc.map(lambda x: {"label_ids_f": x["label_ids"].to(torch.float)},
                    remove_columns=["label_ids"])
ds_enc = ds_enc.rename_column("label_ids_f", "label_ids")
```

Allerdings ist die Gefahr groß, dass das Modell aufgrund des begrenzten Umfangs der Trainingsdaten schnell eine Überanpassung (engl. Overfitting) erfährt. Deshalb setzen wir `load_best_model_at_end=True`. Das beste Modell wählen wir auf der Grundlage des mikro-gemittelten F_1-Maßes:

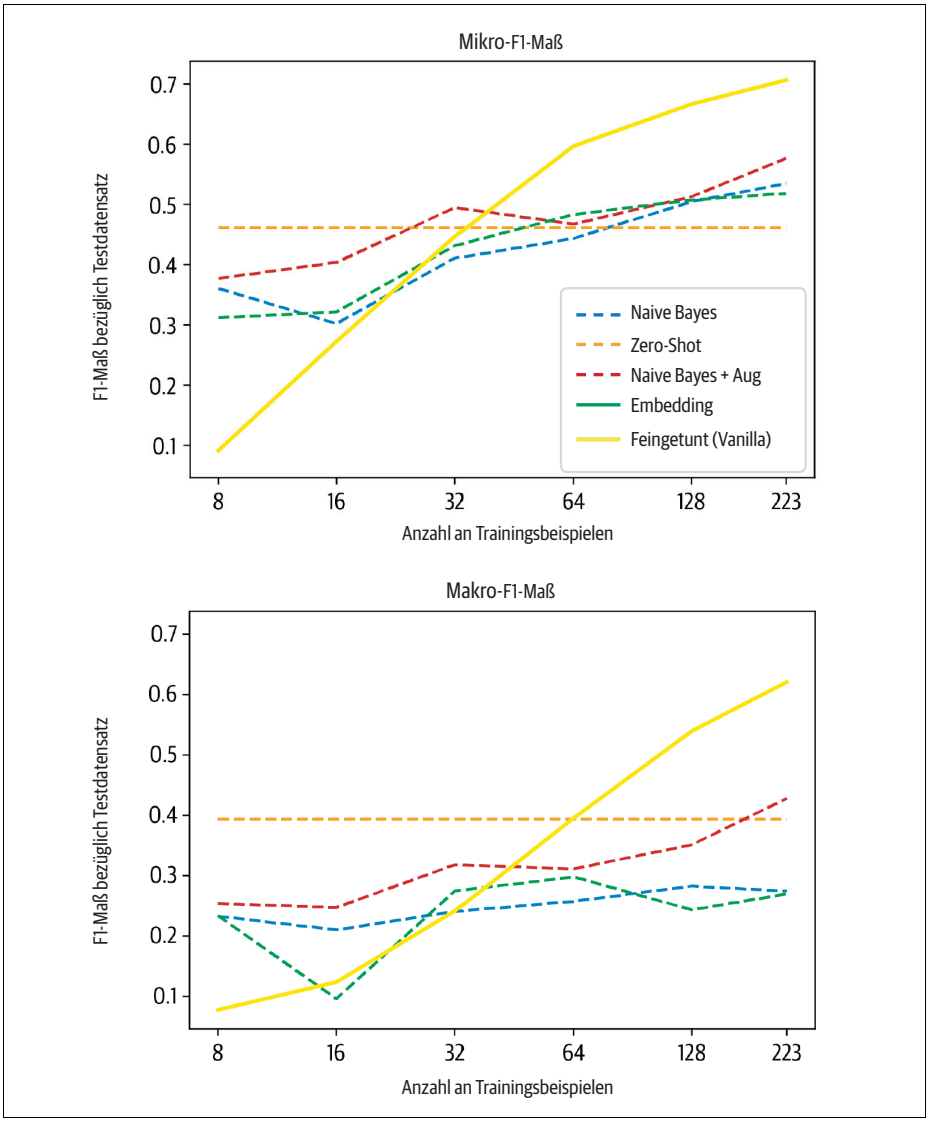
```
from transformers import Trainer, TrainingArguments

training_args_fine_tune = TrainingArguments(
    output_dir="./results", num_train_epochs=20, learning_rate=3e-5,
    lr_scheduler_type='constant', per_device_train_batch_size=4,
    per_device_eval_batch_size=32, weight_decay=0.0,
    evaluation_strategy="epoch", save_strategy="epoch",logging_strategy="epoch",
    load_best_model_at_end=True, metric_for_best_model='micro f1',
    save_total_limit=1, log_level='error')
```

Da wir zur Auswahl des besten Modells auf das F_1-Maß zurückgreifen, müssen wir sicherstellen, dass es während der Evaluierung ermittelt wird. Da das Modell Logits zurückgibt, müssen wir die Vorhersagen zunächst mithilfe der Sigmoidfunktion normalisieren, ehe wir sie unter Verwendung eines einfachen Schwellenwerts binarisieren können. Im Anschluss erhalten wir die für uns relevanten Werte aus dem Klassifizierungsbericht (»Classification Report«):

```
from scipy.special import expit as sigmoid

def compute_metrics(pred):
    y_true = pred.label_ids
    y_pred = sigmoid(pred.predictions)
    y_pred = (y_pred>0.5).astype(float)

    clf_dict = classification_report(y_true, y_pred, target_names=all_labels,
                                     zero_division=0, output_dict=True)
    return {"micro f1": clf_dict["micro avg"]["f1-score"],
            "macro f1": clf_dict["macro avg"]["f1-score"]}
```

Wir sind nun bereit, loszulegen! Mit jedem unserer Trainings-Slices trainieren wir nun einen Klassifikator von Grund auf, laden am Ende der Trainingsschleife das beste Modell und speichern die auf dem Testdatensatz erzielten Ergebnisse:

```
config = AutoConfig.from_pretrained(model_ckpt)
config.num_labels = len(all_labels)
config.problem_type = "multi_label_classification"

for train_slice in train_slices:
    model = AutoModelForSequenceClassification.from_pretrained(model_ckpt,
                                                               config=config)
    trainer = Trainer(
        model=model, tokenizer=tokenizer,
        args=training_args_fine_tune,
        compute_metrics=compute_metrics,
        train_dataset=ds_enc["train"].select(train_slice),
        eval_dataset=ds_enc["valid"],)

    trainer.train()
    pred = trainer.predict(ds_enc["test"])
    metrics = compute_metrics(pred)
    macro_scores["Feingetunt (Vanilla)"].append(metrics["macro f1"])
    micro_scores["Feingetunt (Vanilla)"].append(metrics["micro f1"])

plot_metrics(micro_scores, macro_scores, train_samples, "Feingetunt (Vanilla)")
```



Zuerst fällt auf, dass ein simples Feintuning eines standardmäßigen (»Vanilla«-) BERT-Modells auf dem Datensatz zu mehr als konkurrenzfähigen Ergebnissen führt, sobald wir auf etwa 64 (gelabelte) Beispiele zurückgreifen können. Darüber hinaus sehen wir, dass die Ergebnisse vorher etwas sprunghaft sind, was wiederum darauf zurückzuführen ist, dass das Modell auf einer kleinen Stichprobe trainiert wird, bei der einige Labels so unausgewogen sein könnten, dass dies zu Verzerrungen führt. Bevor wir den ungelabelten Teil unseres Datensatzes verwenden, werfen wir einen kurzen Blick auf einen anderen vielversprechenden Ansatz, um Sprachmodelle in einer bestimmten Domäne verwenden zu können, wenn nur wenige Beispiele zur Verfügung stehen.

In-Context- und Few-Shot-Learning auf Basis von Prompts

In diesem Kapitel haben Sie bereits gelernt, dass Sie ein Sprachmodell wie BERT oder GPT-2 verwenden und es auf eine überwachte Lernaufgabe übertragen können, indem Sie Prompts verwenden und die Vorhersagen des Modells für die Tokens parsen. Dies unterscheidet sich von dem herkömmlichen Ansatz, einen aufgabenspezifischen Head hinzuzufügen und die Modellparameter für die Aufgabe im Rahmen eines Feintunings zu optimieren. Der Vorteil dieses Ansatzes auf der Basis von Prompts ist, dass keine Trainingsdaten benötigt werden. Der Nachteil ist wiederum, dass wir keinen Nutzen aus gelabelten Daten ziehen könnten, selbst wenn wir diese zur Verfügung hätten. Es gibt einen Mittelweg, den wir unter gewissen Umständen nutzen können, das sogenannte *In-Context-* bzw. *Few-Shot-Learning*.

Sehen wir uns das dahinterstehende Konzept einmal genauer an und nehmen wir uns exemplarisch eine Übersetzungsaufgabe vom Englischen ins Französische vor. Im Rahmen des Zero-Shot-Ansatzes würden wir einen Prompt formulieren, der in etwa wie folgt lauten könnte:

```
prompt = """\
Translate English to French:
thanks =>
"""
```

Dadurch wird das Modell hoffentlich dazu veranlasst, die Tokens des Worts »merci« vorherzusagen. In Kapitel 6 haben wir uns bereits mithilfe des GPT-2-Modells Zusammenfassungen generieren lassen. Hierfür mussten wir lediglich die zusätzliche Angabe »TL;DR« an den zusammenzufassenden Text anhängen, wodurch wir das Modell dazu veranlasst haben, eine Zusammenfassung zu generieren – und das, obwohl es nicht explizit für diese Aufgabe trainiert wurde. Ein interessantes Ergebnis des GPT-3-Forschungsbeitrags war, dass große Sprachmodelle die Fähigkeit besitzen, von Beispielen zu lernen, die über den Prompt bereitgestellt werden. Folglich könnte das vorherige Übersetzungsbeispiel auch um mehrere Textbeispiele, die eine Übersetzung vom Englischen ins Deutsche darstellen, erweitert werden, wodurch sich die Leistung des Modells im Hinblick auf diese Aufgabe deutlich verbessern würde.[6]

Darüber hinaus fanden die Autoren heraus, dass die Modelle die kontextbezogenen Beispiele umso besser nutzen, je stärker sie skaliert werden, wodurch sich die Leistung deutlich steigern lässt. Obgleich Modelle in der Größe von GPT-3 eine Herausforderung für den Einsatz in der Produktion darstellen, ist dies ein aufregendes, neues Forschungsgebiet, in dem bereits tolle Anwendungen entwickelt wurden, wie z. B. eine Kommandozeile, deren Befehle in natürlicher Sprache eingegeben werden können und mithilfe des GPT-3-Modells in Kommandozeilenbefehle umgewandelt werden.

Dementsprechend besteht eine Alternative zur Verwendung gelabelter Daten darin, Beispiele für die Prompts und die gewünschten Vorhersagen zu erstellen und

6 T. Brown et al., »Language Models Are Few-Shot Learners« (*https://arxiv.org/abs/2005.14165*), (2020).

das Sprachmodell auf diesen Beispielen weiter zu trainieren. Eine neu entwickelte Methode namens ADAPET[7] verwendet einen solchen Ansatz und übertrifft GPT-3 bei einer Vielzahl von Aufgaben, wobei das Modell mithilfe generierter Prompts optimiert wird. Neuere Untersuchungen der Forscher von Hugging Face legen nahe, dass ein solcher Ansatz dateneffizienter sein kann als einen benutzerdefinierten Head feinzutunen.[8]

In diesem Abschnitt haben wir einige Möglichkeiten eruiert, wie wir die wenigen gelabelten Beispiele, die uns zur Verfügung stehen, sinnvoll nutzen können. In den meisten Fällen haben wir zusätzlich zu den gelabelten Beispielen auch Zugang zu vielen nicht gelabelten Daten. Daher werden wir im nächsten Abschnitt erörtern, wie diese eine sinnvolle Verwendung finden.

Ungelabelte Daten nutzbar machen

Obwohl es unbestritten am besten ist, große Mengen an akkurat gelabelten Daten zur Verfügung zu haben, wenn Sie einen Klassifikator trainieren möchten, bedeutet das nicht, dass ungelabelte Daten grundsätzlich wertlos für Sie sind. Führen Sie sich noch einmal die (meisten) vortrainierten Modelle vor Augen, die wir bereits verwendet haben: Auch wenn sie auf größtenteils zusammenhangslosen Daten aus dem Internet trainiert wurden, können wir die vortrainierten Gewichte für andere Aufgaben und für eine Vielzahl von Texten nutzen. Genau darum geht es beim Transfer Learning im NLP. Natürlich funktioniert der Transfer besser, wenn in der nachgelagerten Aufgabe Texte verwendet werden, die eine ähnliche Struktur wie die im Pretraining genutzten aufweisen. Wenn wir also die Aufgabe, mit der das Modell vortrainiert wird, stärker an die nachgelagerte Aufgabe angleichen, können wir den Transfer möglicherweise verbessern.

Überlegen wir uns das einmal anhand unseres konkreten Anwendungsfalls: Das BERT-Modell wurde auf dem BookCorpus-Datensatz und der englischsprachigen Wikipedia vortrainiert, wobei Texte, die Code und GitHub-Issues enthalten, definitiv nur einen kleinen Teil dieser Datensätze ausmachen. Wenn wir ein BERT-Modell von Grund auf vortrainieren würden, könnten wir zum Beispiel alle Issues auf GitHub crawlen und diese für das Pretraining heranziehen. Das wäre jedoch recht kostspielig, und viele sprachliche Aspekte, die BERT bereits gelernt hat, gelten auch für GitHub-Issues. Gibt es also einen Mittelweg zwischen einem erneuten Training von Grund auf und der Verwendung des unveränderten Modells für die Klassifizierung? Ja, und zwar die sogenannte Domain Adaptation (die wir bereits im Rahmen des Question Answering in Kapitel 7 kennengelernt haben). Anstatt das Sprachmodell von Grund auf neu zu trainieren, können wir es auf Basis von Daten, die aus unserer Domäne stammen, weiter trainieren. In diesem Schritt ver-

7 D. Tam et al., »Improving and Simplifying Pattern Exploiting Training« (*https://arxiv.org/abs/2103.11955*), (2021).

8 T. Le Scao und A.M. Rush, »How Many Data Points Is a Prompt Worth?« (*https://arxiv.org/abs/2103.08493*), (2021).

wenden wir die klassische Zielsetzung (Objective) der Sprachmodellierung, maskierte Wörter vorherzusagen, d.h., wir benötigen keine gelabelten Daten. Im Anschluss daran können wir das adaptierte Modell als Klassifikator laden und auf der Grundlage der ungelabelten Daten feintunen.

Der Reiz der Domain Adaptation liegt vor allem darin, dass ungelabelte Daten im Gegensatz zu gelabelten Daten häufig in großem Umfang verfügbar sind. Außerdem kann das adaptierte Modell für zahlreiche andere Anwendungsfälle wiederverwendet werden. Stellen Sie sich vor, Sie möchten einen E-Mail-Klassifikator erstellen und wenden dafür eine Domain Adaptation auf alle Ihre historischen E-Mails an. Zu einem späteren Zeitpunkt können Sie dasselbe Modell für die Named Entity Recognition oder eine andere Klassifizierungsaufgabe wie die Sentimentanalyse verwenden, da der Ansatz unabhängig von der nachgelagerten Aufgabe ist.

Sehen wir uns nun die erforderlichen Schritte an, um ein vortrainiertes Sprachmodell feinzutunen.

Ein Sprachmodell feintunen

In diesem Abschnitt nehmen wir ein Feintuning des vortrainierten BERT-Modells mittels maskierter Sprachmodellierung auf Basis des ungelabelten Teils unseres Datensatzes vor. Dazu benötigen wir nur zwei neue Elemente: einen zusätzlichen Schritt im Rahmen der Tokenisierung der Daten und einen speziellen Data-Collator, wobei wir uns als Erstes der Tokenisierung widmen.

Zusätzlich zu den gewöhnlichen Tokens im Text fügt der Tokenizer jeder Sequenz auch spezielle Tokens hinzu, z.B. die Tokens `[CLS]` und `[SEP]`, die für die Klassifizierung und die Vorhersage des nächsten Satzes verwendet werden. Im Rahmen der maskierten Sprachmodellierung möchten wir sicherstellen, dass wir das Modell nicht darauf trainieren, auch die speziellen Tokens vorherzusagen. Aus diesem Grund maskieren wir sie, bevor der Verlust ermittelt wird. Damit eine Maskierung im Rahmen der Tokenisierung vorgenommen wird, müssen wir für den Tokenizer `return_special_tokens_mask=True` setzen. Nehmen wir eine erneute Tokenisierung des Texts mit diesem Setting vor:

```
def tokenize(batch):
    return tokenizer(batch["text"], truncation=True,
                     max_length=128, return_special_tokens_mask=True)

ds_mlm = ds.map(tokenize, batched=True)
ds_mlm = ds_mlm.remove_columns(["labels", "text", "label_ids"])
```

Was für die Durchführung der maskierten Sprachmodellierung noch fehlt, ist ein Mechanismus, der es ermöglicht, einzelne Tokens der Eingabesequenz zu maskieren und die vorherzusagenden Tokens (engl. Target Tokens) mit auszugeben. Eine Möglichkeit, dies zu erreichen, besteht darin, eine Funktion einzurichten, die zufällig ausgewählte Tokens maskiert und Labels für die Sequenzen erstellt. Dies würde jedoch die Größe des Datensatzes verdoppeln, da auch die vorherzusagende

Sequenz (engl. Target Sequence) im Datensatz gespeichert werden müsste. Zudem würden wir die Sequenzen innerhalb einer Epoche immer gleich maskieren.

Eine weitaus elegantere Lösung ist die Verwendung eines Data-Collators. Wie wir bereits erläutert haben, handelt es sich bei einem Data-Collator um eine Funktion, die die Brücke zwischen dem Datensatz und den Aufrufen des Modells schlägt. Nachdem ein Batch aus dem Datensatz gezogen wurde, bereitet der Data-Collator die Elemente des Batches so vor, dass sie in das Modell eingespeist werden können. Im einfachsten Fall verkettet er einfach die Tensoren der einzelnen Elemente zu einem einzigen Tensor. In unserem Fall können wir ihn verwenden, um bereits während dieses Prozesses (*on the fly*) die Maskierung durchzuführen und die Labels zu erzeugen. Auf diese Weise müssen die Labels nicht gespeichert werden, und für jedes gezogene Batch werden neue Maskierungen vorgenommen. Der Data-Collator, mit dem wir diese Aufgabe bewältigen können, heißt `DataCollatorForLanguageModeling`. Wir initialisieren ihn mit dem Tokenizer des Modells, wobei wir über das Argument `mlm_probability` den Anteil der Tokens angeben, die wir maskieren wollen. In unserem Fall maskieren wir mithilfe des Data-Collators 15 % der Tokens, was dem Wert im BERT-Forschungspapier entspricht:

```
from transformers import DataCollatorForLanguageModeling, set_seed

data_collator = DataCollatorForLanguageModeling(tokenizer=tokenizer,
                                                mlm_probability=0.15)
```

Sehen wir uns nun genauer an, wie der Data-Collator funktioniert. Damit wir uns auf die Schnelle die Ergebnisse als `DataFrame` ausgeben lassen können, sorgen wir dafür, dass die Ausgaben des Tokenizers und des Data-Collators im NumPy-Format sind:

```
set_seed(3)
data_collator.return_tensors = "np"
inputs = tokenizer("Transformers are awesome!", return_tensors="np")
outputs = data_collator([{"input_ids": inputs["input_ids"][0]}])

pd.DataFrame({
    "Ursprüngliche Tokens": \
        tokenizer.convert_ids_to_tokens(inputs["input_ids"][0]),
    "Maskierte Tokens": \
        tokenizer.convert_ids_to_tokens(outputs["input_ids"][0]),
    "Ursprüngliche input_ids": original_input_ids,
    "Maskierte input_ids": masked_input_ids,
    "Labels": outputs["labels"][0]}).T
```

	0	1	2	3	4	5
Ursprüngliche Tokens	[CLS]	transformers	are	awesome	!	[SEP]
Maskierte Tokens	[CLS]	transformers	are	awesome	[MASK]	[SEP]
Ursprüngliche input_ids	101	19081	2024	12476	999	102
Maskierte input_ids	101	19081	2024	12476	103	102
Labels	-100	-100	-100	-100	999	-100

Wie ersichtlich ist, wurde das Token, das dem Ausrufezeichen entspricht, durch ein Maskierungstoken bzw. Mask-Token ersetzt. Außerdem hat der Data-Collator ein Array für die Labels zurückgegeben, wobei ein Wert von -100 kennzeichnet, dass es sich um das ursprüngliche Token handelt und für die maskierten Tokens jeweils die ID des Tokens, das maskiert wurde, angegeben wird. Wie wir bereits zuvor erfahren haben, werden die Einträge, die den Wert -100 aufweisen, bei der Berechnung des Verlusts nicht berücksichtigt. Ändern wir nun das Ausgabeformat des Data-Collators zurück auf das PyTorch-Format:

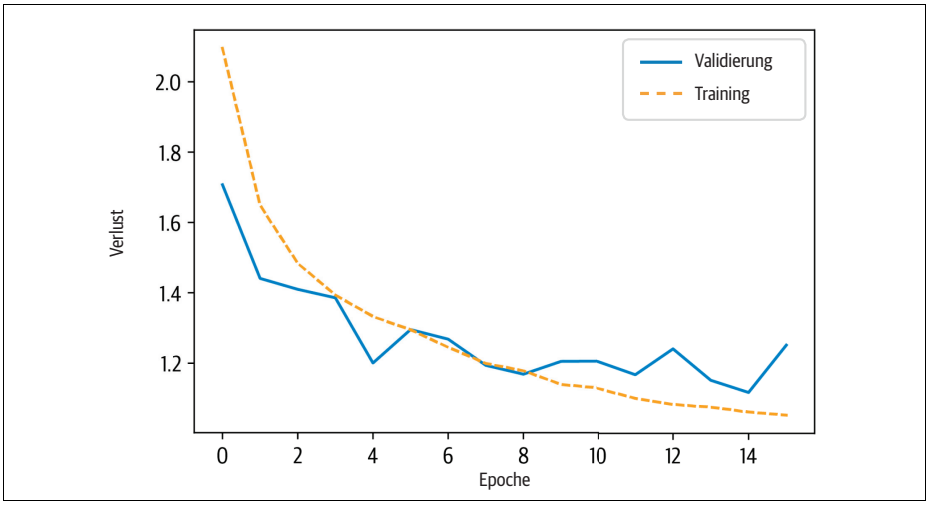
```
data_collator.return_tensors = "pt"
```

Nachdem wir den Tokenizer und den Data-Collator eingerichtet haben, können wir nun das Feintuning des maskierten Sprachmodells vornehmen. Dazu richten wir wie üblich zunächst das `TrainingArguments`- und das `Trainer`-Objekt ein:

```
from transformers import AutoModelForMaskedLM

training_args = TrainingArguments(
    output_dir = f"{model_ckpt}-issues-128", per_device_train_batch_size=32,
    logging_strategy="epoch", evaluation_strategy="epoch", save_strategy="no",
    num_train_epochs=16, push_to_hub=True, log_level="error", report_to="none")

trainer = Trainer(
        model=AutoModelForMaskedLM.from_pretrained("bert-base-uncased"),
        tokenizer=tokenizer, args=training_args, data_collator=data_collator,
        train_dataset=ds_mlm["unsup"], eval_dataset=ds_mlm["train"])

trainer.train()

trainer.push_to_hub("Training complete!")
```

Wir können auf die Logging-Historie des Trainers zugreifen, um uns die Trainings- und Validierungsverluste des Modells ausgeben zu lassen. Alle Logs werden in `trainer.state.log_history` als eine aus Dictionarys bestehende Liste gespeichert, die wir auf einfache Weise in ein Pandas-`DataFrame` laden können. Da der Trainings- und der Validierungsverlust in unterschiedlichen Schritten festgehalten werden, weist der DataFrame fehlende Werte auf. Deshalb müssen wir die fehlenden Werte erst entfernen, bevor wir die Maße visualisieren können:

```
df_log = pd.DataFrame(trainer.state.log_history)

(df_log.dropna(subset=["eval_loss"]).reset_index()["eval_loss"]
 .plot(label="Validierung"))
df_log.dropna(subset=["loss"]).reset_index()["loss"].plot(label="Training")

plt.xlabel("Epoche")
plt.ylabel("Verlust")
plt.legend(loc="upper right")
plt.show()
```



Es scheint, dass sowohl der Trainings- als auch der Validierungsverlust bedeutend niedriger ausfallen. Dementsprechend sollten wir überprüfen, ob wir auch eine bessere Leistung erzielen können, wenn wir einen Klassifikator auf der Grundlage dieses Modells feingetunt haben.

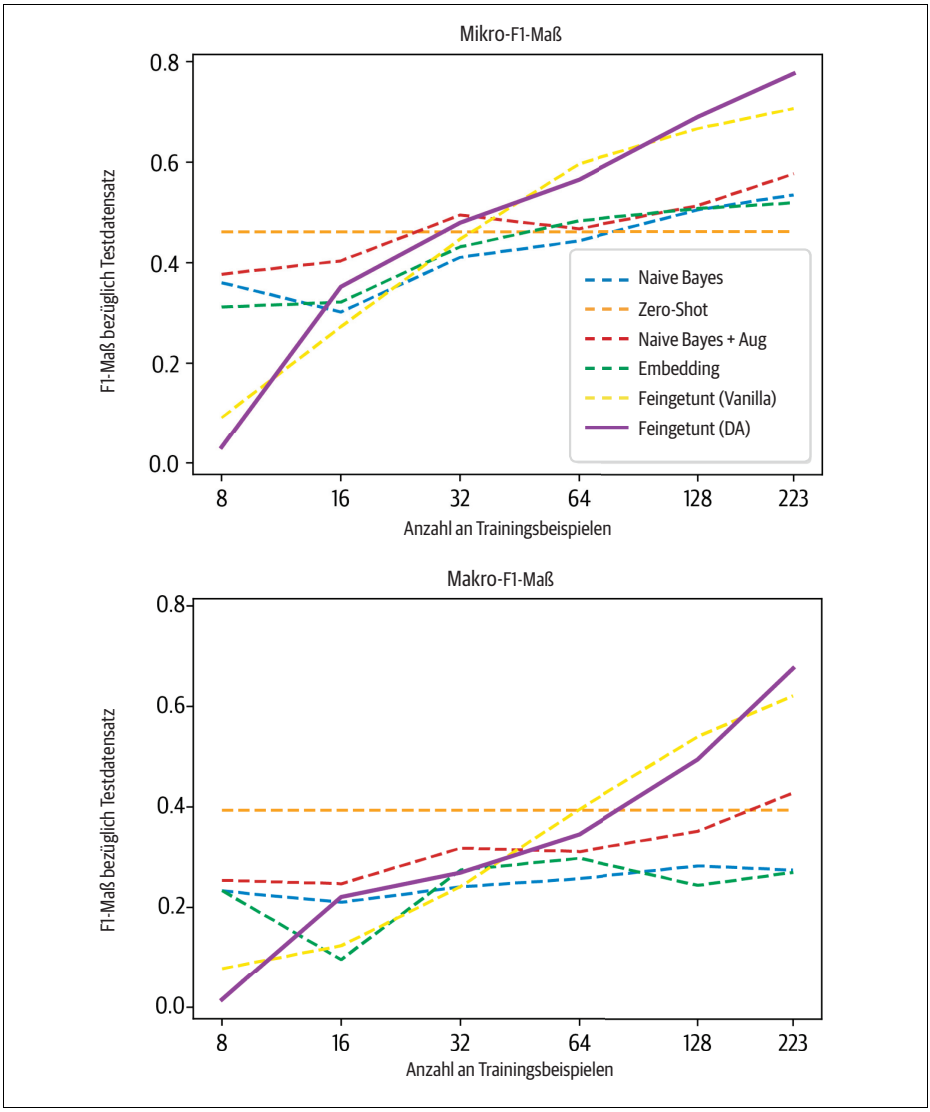
Einen Klassifikator feintunen

Jetzt wiederholen wir das Feintuning, allerdings mit dem geringfügigen Unterschied, dass wir unseren selbst erstellten Checkpoint laden:

```
model_ckpt = f'{model_ckpt}-issues-128'
config = AutoConfig.from_pretrained(model_ckpt)
config.num_labels = len(all_labels)
config.problem_type = "multi_label_classification"

for train_slice in train_slices:
    model = AutoModelForSequenceClassification.from_pretrained(model_ckpt,
                                                               config=config)
    trainer = Trainer(
        model=model,
        tokenizer=tokenizer,
        args=training_args_fine_tune,
        compute_metrics=compute_metrics,
        train_dataset=ds_enc["train"].select(train_slice),
        eval_dataset=ds_enc["valid"],
    )

    trainer.train()
    pred = trainer.predict(ds_enc['test'])
    metrics = compute_metrics(pred)
    # DA steht für Domain Adaptation
    macro_scores['Feingetunt (DA)'].append(metrics['macro f1'])
    micro_scores['Feingetunt (DA)'].append(metrics['micro f1'])
```

Wenn wir die Ergebnisse mit dem Feintuning auf der Basis des unveränderten standardmäßigen (»Vanilla«-)BERT-Modells vergleichen, stellen wir fest, dass wir vor allem dann besser damit fahren, wenn nur wenige Daten vorliegen. Auch in Fällen, in denen eine größere Anzahl gelabelter Daten zur Verfügung steht, können wir einige Prozentpunkte gutmachen:

```
plot_metrics(micro_scores, macro_scores, train_samples, "Feingetunt (DA)")
```



Hier zeigt sich deutlich, dass die Domain Adaptation dazu beitragen kann, die Leistung des Modells mithilfe von ungelabelten Daten zu verbessern – und das mit relativ geringem Aufwand. Je mehr ungelabelte Daten und je weniger gelabelte Daten Sie zur Hand haben, desto größer ist natürlich der positive Effekt, den Sie mit

dieser Methode erzielen. Bevor wir zum Schluss dieses Kapitels kommen, zeigen wir Ihnen noch ein paar weitere Tricks, wie Sie aus ungelabelten Daten Nutzen ziehen können.

Fortgeschrittene Methoden

Ein Sprachmodell feinzutunen, bevor der Head zur Klassifizierung (engl. Classification Head) optimiert wird, ist eine einfache, aber zuverlässige Methode, die Leistung eines Modells zu erhöhen. Es gibt jedoch noch ausgefeiltere Methoden, mit denen Sie den Nutzen ungelabelter Daten deutlich besser ausschöpfen können. Wir fassen hier einige dieser Methoden zusammen, die Ihnen als Ausgangspunkt dienen können, wenn Sie nach Wegen suchen, um eine bessere Leistung zu erzielen.

Unsupervised Data Augmentation

Die Hauptidee hinter der unüberwachten Datenaugmentierung bzw. der Unsupervised Data Augmentation (UDA) ist, dass die Vorhersage, die ein Modell für ein ungelabeltes Beispiel trifft, mit der Vorhersage übereinstimmen sollte, die das Modell für das gleiche Beispiel trifft, nachdem es leicht verfälscht wurde. Derartige Verfälschungen können mit den üblichen Strategien zur Datenaugmentierung wie z.B. Tokens zu ersetzen oder eine Rückübersetzung vorzunehmen, herbeigeführt werden. Um eine Übereinstimmung bzw. Konsistenz der Vorhersagen zu erzwingen, kann die KL-Divergenz zwischen den Vorhersagen des ursprünglichen und des verfälschten Beispiels minimiert werden. Dieser Vorgang wird in Abbildung 9-5 veranschaulicht, wobei die Konsistenzbedingung berücksichtigt wird, indem der Kreuzentropieverlust um einen zusätzlichen Term, der sich auf die ungelabelten Beispiele bezieht, erweitert wird. Das bedeutet, dass das Modell auf den gelabelten Daten mittels des üblichen überwachten Ansatzes trainiert wird, wobei das Modell jedoch dazu gezwungen wird, konsistente Vorhersagen für die ungelabelten Daten zu treffen.

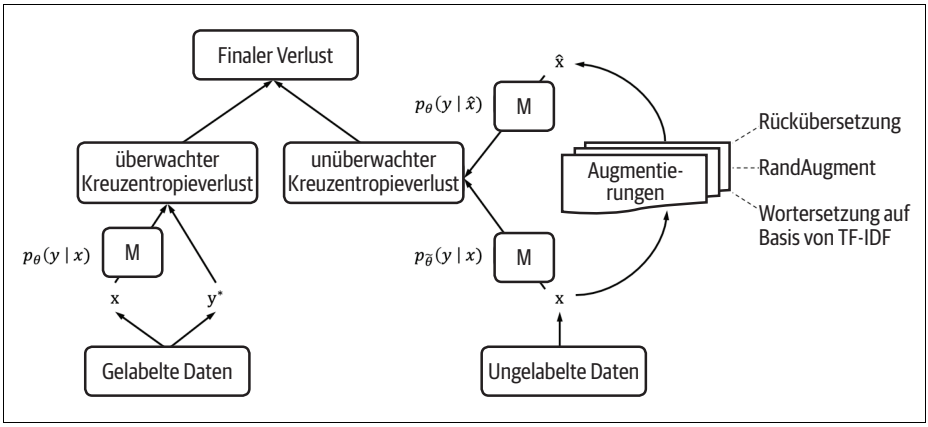


Abbildung 9-5: Training eines Modells, M, unter Verwendung der Unsupervised Data Augmentation (mit freundlicher Genehmigung von Qizhe Xie)

Die Ergebnisse dieses Ansatzes sind ziemlich beeindruckend: Mit nur einer Handvoll gelabelter Beispiele erzielen BERT-Modelle, die mittels UDA trainiert wurden, eine ähnliche Leistung wie Modelle, die auf der Grundlage von Tausenden von Beispielen trainiert wurden. Der Nachteil ist, dass Sie erst eine Pipeline zur Datenaugmentierung einrichten müssen und dass das Training wesentlich länger dauert, da mehrere Forward-Passes erforderlich sind, um die vorhergesagten Verteilungen für die ungelabelten und augmentierten Beispiele zu generieren.

Uncertainty-Aware Self-Training

Eine weitere vielversprechende Methode, ungelabelte Daten nutzbringend zu verwenden, ist das Uncertainty-Aware Self-Training (UST). Die Idee dabei ist, ein Lehrermodell (engl. Teacher Model) mit den gelabelten Daten zu trainieren und es dann dafür zu verwenden, Pseudo-Labels für die ungelabelten Daten zu erstellen. Anschließend wird ein Schülermodell (engl. Student) auf die pseudo-gelabelten Daten trainiert, das nach erfolgreichem Training als Lehrermodell in der nächsten Iteration fungiert.

Ein interessanter Aspekt dieser Methode ist die Art und Weise, wie die Pseudo-Labels generiert werden: Um ein Maß zu erhalten, das die Unsicherheit des Modells bei den Vorhersagen quantifiziert, wird dieselbe Eingabe mehrmals durch das Modell geleitet, wobei jeweils ein Dropout (d. h. zufällig ein paar Gewichte auf null gesetzt werden) vorgenommen wird. Die Varianz in den Vorhersagen liefert dann einen Anhaltspunkt dafür, wie verlässlich das Modell in Bezug auf ein bestimmtes Beispiel ist. Auf der Grundlage dieses Unsicherheitsmaßes werden dann die Pseudo-Labels mithilfe einer Methode namens Bayesian Active Learning by Disagreement (BALD) ermittelt. Die vollständige Trainings-Pipeline ist in Abbildung 9-6 dargestellt.

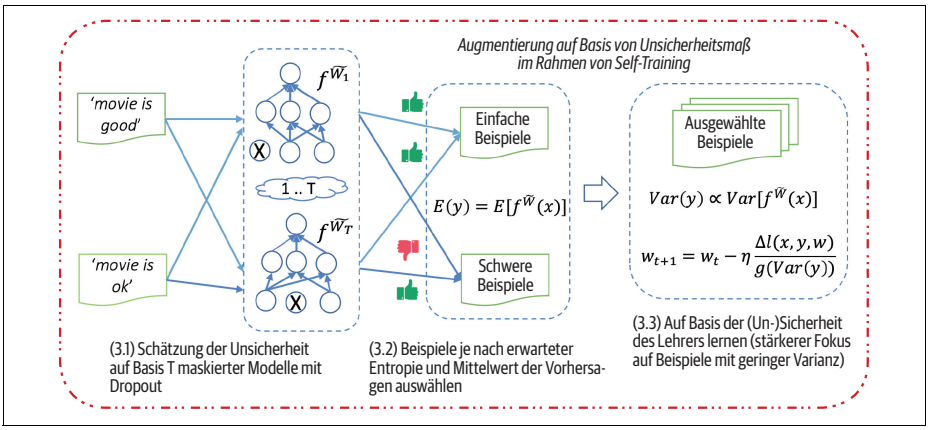


Abbildung 9-6: Die UST-Methode besteht aus einem Lehrermodell, das Pseudo-Labels generiert, und einem Schülermodell, das anschließend auf Basis dieser Labels trainiert wird. Nachdem das Schülermodell trainiert wurde, übernimmt es die Rolle des Lehrermodells, und der Vorgang wird erneut durchgeführt (mit freundlicher Genehmigung von Subhabrata Mukherjee).[9]

9 S. Mukherjee und A.H. Awadallah, »Uncertainty-Aware Self-Training for Few-Shot Text Classification« (https://arxiv.org/abs/2006.15315), (2020).

Durch diesen iterativen Ansatz wird das Lehrermodell immer besser darin, Pseudo-Labels zu generieren, wodurch sich auch die Leistung des Modells verbessert. Letztendlich erreicht dieser Ansatz fast das Niveau von Modellen, die auf Basis der gesamten Trainingsdaten mit Tausenden von Beispielen trainiert wurden, und übertrifft bei einigen Datensätzen sogar die UDA.

Nachdem wir nun einige fortgeschrittene Methoden kennengelernt haben, sollten wir uns noch einmal vor Augen führen, was wir in diesem Kapitel gelernt haben.

Zusammenfassung

In diesem Kapitel haben wir festgestellt, dass selbst dann, wenn wir nur wenige oder gar keine Labels zur Hand haben, längst nicht alle Hoffnung verloren ist. Wir können Modelle heranziehen, die für andere Aufgaben vortrainiert wurden, wie z.B. Sprachmodelle wie BERT oder ein GPT-2-Modell, das auf Basis von Python-Code trainiert wurde, um Vorhersagen im Rahmen einer neuen Aufgabe – in unserem Fall GitHub-Issues zu klassifizieren – zu treffen. Außerdem können wir eine Domain Adaption vornehmen, um beim Training des Modells mit einem gewöhnlichen Head zur Klassifizierung einen zusätzlichen Leistungsschub zu erhalten.

Welcher der vorgestellten Ansätze im Ihrem konkreten Anwendungsfall am besten funktioniert, hängt von einer Vielzahl von Aspekten ab: wie viele gelabelte Daten Sie zur Verfügung haben, wie verrauscht bzw. fehlerhaft sie sind, wie sehr die Daten dem Pretraining-Korpus ähneln usw. Um herauszufinden, welcher Ansatz am besten funktioniert, ist es eine gute Idee, eine Evaluierungspipeline einzurichten und zügig zu iterieren. Die flexible API der 🤗 Transformers-Bibliothek ermöglicht es Ihnen, innerhalb kürzester Zeit eine Reihe von Modellen zu laden und sie zu vergleichen, ohne dass Sie dafür eine Anpassung des Codes vornehmen müssen. Derzeit sind über 10.000 Modelle auf dem Hugging Face Hub verfügbar. Die Chancen stehen also nicht schlecht, dass sich jemand in der Vergangenheit bereits mit einem ähnlichen Problem beschäftigt hat und Sie darauf aufbauen können.

Ein Aspekt, der über den Rahmen dieses Buchs hinausgeht, betrifft die Abwägung, die Sie zwischen einem komplexeren Ansatz wie der UDA oder dem UST und dem Erhalt von mehr Daten vornehmen müssen. Um Ihren Ansatz evaluieren zu können, ist es sinnvoll, möglichst frühzeitig einen Validierungs- und einen Testdatensatz zu erstellen. Dabei können Sie Schritt für Schritt auch mehr gelabelte Daten sammeln. In der Regel dauert die Annotation von ein paar Hundert Beispielen lediglich ein paar Stunden bis hin zu einigen wenigen Tagen, und es gibt viele Tools, die Sie dabei unterstützen können. Je nachdem, welches Ziel Sie verfolgen, kann es sinnvoll sein, etwas Zeit darauf zu verwenden, einen kleinen, qualitativ hochwertigen Datensatz zu erstellen, anstatt eine sehr komplexe Methode zu erarbeiten, um den Mangel an Daten zu kompensieren. Mit den in diesem Kapitel vorgestellten Methoden können Sie sicherstellen, dass Sie den größtmöglichen Nutzen aus Ihren wertvollen gelabelten Daten ziehen.

Wir haben uns in eine Welt vorgewagt, in der nur geringe Datenmengen zur Verfügung stehen, und erfahren, dass Transformer-Modelle sogar dann noch leistungsfähig sein können, wenn wir lediglich auf etwa hundert Beispiele zurückgreifen können. Im nächsten Kapitel werden wir uns genau dem umgekehrten Fall widmen: Wir werden in Erfahrung bringen, was wir unternehmen können, wenn uns Hunderte von Gigabytes an Daten und reichlich Rechenressourcen bereitstehen. Wir werden ein großes Transformer-Modell von Grund auf so trainieren, dass es automatisch Code für uns vervollständigen kann.

KAPITEL 10

Transformer-Modelle von Grund auf trainieren

In diesem Buch haben wir eingangs eine ausgeklügelte Anwendung namens GitHub Copilot erwähnt, die mithilfe GPT-basierter Transformer-Modelle in der Lage ist, eine automatische Vervollständigung von Code vorzunehmen. Dieses Feature ist besonders nützlich, wenn Sie in einer neuen Sprache oder einem neuen Framework programmieren oder erst lernen, zu programmieren, oder um automatisch Codegerüste (sogenannten Boilerplate-Code) zu erstellen. Darüber hinaus gibt es noch die Produkte TabNine (*https://tabnine.com*) und Kite (*https://kite.com*), bei denen ebenfalls KI-Modelle für diesen Zweck zum Einsatz kommen. In Kapitel 5 haben wir uns näher angesehen, wie wir GPT-Modelle verwenden können, um hochwertige Texte zu generieren. In diesem Kapitel schließen wir den Kreis und erstellen unser eigenes GPT-basiertes Modell, um Python-Quellcode zu generieren. Das resultierende Modell nennen wir *CodeParrot*.

Bisher haben wir uns vor allem mit Lösungen beschäftigt, bei denen die verfügbare Anzahl an gelabelten Trainingsdaten begrenzt ist. In diesen Fällen hat uns das Transfer Learning geholfen, leistungsfähige Modelle zu erstellen. In Kapitel 9 haben wir die Möglichkeiten des Transfer Learning bis zum Äußersten ausgereizt und nahezu keine Trainingsdaten verwendet.

In diesem Kapitel gehen wir zum anderen Extrem über und befassen uns mit dem Fall, dass wir auf eine beinahe unbegrenzte Menge an Daten zugreifen können. Dabei werden wir den Schritt des Pretrainings genauer beleuchten und lernen, wie wir einen Transformer von Grund auf trainieren können. Dabei werden wir uns mit einigen Aspekten des Trainings befassen, die wir bisher noch nicht berücksichtigt haben, wie zum Beispiel den folgenden:

- einen sehr großen Datensatz zusammenzustellen und zu verarbeiten
- einen eigens konzipierten Tokenizer für unseren Datensatz zu erstellen
- ein Modell in großem Maßstab auf mehreren GPUs zu trainieren

Um große Modelle mit Milliarden von Parametern effizient trainieren zu können, benötigen wir spezielle Tools, die ein verteiltes Training (engl. Distributed Training) ermöglichen. Obwohl die `Trainer`-Klasse der 🤗 Transformers-Bibliothek verteiltes Training unterstützt, werden wir die Gelegenheit nutzen, um eine leistungs-

starke PyTorch-Bibliothek namens 🤗 Accelerate vorzustellen. Zum Schluss werden wir uns mit einigen der größten NLP-Modelle befassen, die derzeit verwendet werden – doch zunächst müssen wir erst einmal einen ausreichend großen Datensatz beschaffen.

Anders als der Code in den anderen Kapiteln dieses Buchs (der mit einem Jupyter Notebook auf einer einzelnen GPU ausgeführt werden kann), ist der Trainingscode in diesem Kapitel so konzipiert, dass er als Skript auf mehreren GPUs ausgeführt wird. Wenn Sie Ihre eigene Version von CodeParrot trainieren möchten, empfehlen wir Ihnen, das dafür vorgesehene Skript auszuführen, das Sie in dem 🤗 Repository der Transformers-Bibliothek (*https://oreil.ly/ZyPPR*) finden.

Große Datensätze und wie sie beschafft werden können

Es gibt viele Bereiche, in denen Sie eine sehr umfangreiche Menge an Daten zur Verfügung haben, von juristischen Dokumenten über biomedizinische Datensätze bis hin zu Codebases. In den meisten Fällen sind diese Datensätze nicht gelabelt und können aufgrund ihrer Größe in der Regel nur mithilfe von Heuristiken oder mitgelieferten Metadaten, die während des Erfassungsprozesses gespeichert wurden, gelabelt werden.

Ein sehr großes Korpus kann jedoch auch dann von Nutzen sein, wenn es nicht gelabelt ist oder nur auf Basis von Heuristiken gelabelt wurde. Ein Beispiel dafür haben wir in Kapitel 9 kennengelernt, bei dem wir den ungelabelten Teil eines Datensatzes dazu verwendet haben, ein Sprachmodell feinzutunen, um es auf eine andere Domäne zu übertragen (Domain Adaptation). Dieser Ansatz führt in der Regel zu einer besseren Leistung, wenn nur wenige Daten verfügbar sind. Die Entscheidung, ein Modell von Grund auf zu trainieren, anstatt ein bestehendes feinzutunen, hängt hauptsächlich von der Größe des Korpus ab, das Ihnen für das Feintuning zur Verfügung steht, sowie davon, wie stark sich die zugrunde liegenden Domänen zwischen den verfügbaren vortrainierten Modellen und dem Korpus unterscheiden.

Wenn Sie ein vortrainiertes Modell verwenden, sind Sie praktisch gezwungen, den entsprechenden Tokenizer des Modells zu verwenden. Sollten Sie einen Tokenizer verwenden, der mit einem Korpus aus einer anderen Domäne trainiert wurde, ist das in der Regel suboptimal. Wenn Sie zum Beispiel den vortrainierten Tokenizer des GPT-Modells auf juristische Dokumente, andere Sprachen oder sogar völlig andere Sequenzen wie etwa Musiknoten oder DNA-Sequenzen anwenden, resultiert dies in einer unzureichenden Tokenisierung (wie wir gleich sehen werden).

Je eher die Größe des Datensatzes, der Ihnen für ein Training von Grund auf zur Verfügung steht, an die Größe der Pretraining-Datensätze herankommt, mit denen Modelle üblicherweise vortrainiert werden, desto eher lohnt es sich, das Modell und den Tokenizer selbst von Grund auf zu trainieren, vorausgesetzt, Sie verfügen über die erforderlichen Rechenressourcen. Bevor wir die verschiedenen Pretrai-

ning-Objectives weiter erörtern, müssen wir zunächst ein großes Korpus aufbauen, das für das Pretraining eines Modells geeignet ist. Ein solches Korpus selbst aufzubauen, bringt eine Reihe von Herausforderungen mit sich, die wir im nächsten Abschnitt untersuchen werden.

Herausforderungen beim Aufbau eines großen Korpus

Die Qualität eines Modells nach dem Pretraining spiegelt weitgehend die Qualität des für das Pretraining verwendeten Korpus wider. Beispielsweise übernimmt das Modell alle Fehler, die im Pretraining-Korpus enthalten sind. Bevor wir also versuchen, ein eigenes Modell zu erstellen, ist es ratsam, sich darüber im Klaren zu sein, welche Probleme und Herausforderungen im Zusammenhang mit der Erstellung großer Korpora für das Pretraining üblicherweise auftreten.

Je größer ein Datensatz ist, desto geringer sind die Chancen, dass Sie ihn vollständig beherrschen bzw. zumindest eine genaue Vorstellung davon haben, was in ihm enthalten ist. Ein sehr großer Datensatz wurde höchstwahrscheinlich nicht von engagierten Entwicklern zusammengestellt, die ein Beispiel nach dem anderen erstellen und dabei die gesamte Pipeline und die Aufgabe, für die das Machine-Learning-Modell vorgesehen ist, kennen. Stattdessen ist es bedeutend wahrscheinlicher, dass ein sehr großer Datensatz auf automatische oder halbautomatische Weise erstellt wurde, indem Daten gesammelt wurden, die als Nebenprodukt anderer Aktivitäten entstanden sind. Er kann zum Beispiel aus allen Dokumenten (z. B. Verträgen, Bestellungen usw.) bestehen, die ein Unternehmen aufbewahrt, aus Loggingdaten von Benutzeraktivitäten oder aus Daten, die im Internet zusammengetragen wurden.

Der Umstand, dass große Datensätze meist weitgehend automatisiert erstellt werden, hat mehrere wichtige Konsequenzen zur Folge. Ein wichtiger Aspekt ist, dass sowohl ihr Inhalt als auch die Art und Weise, wie sie erstellt werden, nur begrenzt kontrolliert werden können. Dadurch steigt das Risiko, dass ein Modell anhand von Daten trainiert wird, die mit Vorurteilen behaftet (engl. biased) oder von geringerer Qualität sind. Jüngste Untersuchungen[1] berühmter großer Datensätze wie dem BookCorpus- und dem C4-Datensatz, die für das Training von BERT bzw. T5 verwendet wurden, haben (unter anderem) zutage gebracht, dass:

- ein beträchtlicher Teil des C4-Korpus nicht von Menschen, sondern maschinell übersetzt wurde.
- infolge der Filterung von Stoppwörtern im C4-Datensatz übermäßig viele Begriffe gelöscht wurden, die dem afroamerikanischen Englisch zuzuordnen sind, weshalb derartige Inhalte unterrepräsentiert sind.
- es in der Regel schwierig ist, einen guten Mittelweg für ein großes Textkorpus zu finden und einerseits nicht zu viel von sexuellen oder anderen vulgären In-

1 Y. Zhu et al., »Aligning Books and Movies: Towards Story-Like Visual Explanations by Watching Movies and Reading Books« (*https://arxiv.org/abs/1506.06724*), (2015); J. Dodge et al., »Documenting the English Colossal Clean Crawled Corpus« (*https://arxiv.org/abs/2104.08758*), (2021).

halten einzubeziehen und andererseits auch nicht vollständig auf Inhalte, die das Geschlecht oder die Sexualität betreffen, zu verzichten. Überraschenderweise ist z. B. ein recht gebräuchliches englisches Wort wie »sex« (das sowohl eine neutrale als auch eine sexuelle Bedeutung haben kann) einem Tokenizer, der auf dem C4-Datensatz trainiert wurde, völlig unbekannt, da dieses Wort in dem Korpus überhaupt nicht vorkommt.

- es an vielen Stellen Urheberrechtsverletzungen im BookCorpus-Datensatz und wahrscheinlich auch in anderen großen Datensätzen gibt.[2]
- im BookCorpus-Datensatz das Genre »Liebesroman« übermäßig häufig vertreten ist.

Das heißt jedoch nicht, dass diese Beobachtungen mit der späteren Verwendung der auf diesen Korpora trainierten Modelle unvereinbar sind. Zum Beispiel ist die übermäßige Berücksichtigung von Liebesromanen im BookCorpus-Datensatz wahrscheinlich vertretbar, wenn das Modell als Hilfsmittel zum Schreiben von Liebesromanen oder für die Entwicklung eines Spiels vorgesehen ist.

Führen wir uns nun vor Augen, was es bedeutet, wenn ein Modell durch die Trainingsdaten verzerrt ist, indem wir die generierten Texte des GPT- und GPT-2-Modells miteinander vergleichen. Das GPT-Modell wurde hauptsächlich auf dem BookCorpus-Datensatz trainiert, während das GPT-2-Modell auf Basis von Webseiten, Blogs und von auf Reddit verlinkten Nachrichtenartikeln trainiert wurde. Für unseren Vergleich verwenden wir von beiden Modelltypen ähnlich große Versionen und geben jeweils denselben Prompt vor. Der Unterschied besteht somit hauptsächlich darin, dass beiden Modellen ein anderer Datensatz für das Pretraining zugrunde lag. Richten wir zunächst die entsprechenden `text-generation`-Pipelines ein, um die Modellausgaben untersuchen zu können:

```
from transformers import pipeline, set_seed

generation_gpt = pipeline("text-generation", model="openai-gpt")
generation_gpt2 = pipeline("text-generation", model="gpt2")
```

Als Nächstes erstellen wir eine einfache Funktion, mit der wir die Anzahl der Parameter der Modelle ermitteln können:

```
def model_size(model):
    return sum(t.numel() for t in model.parameters())

print(f"Größe von GPT: \
       {model_size(generation_gpt.model)/1000**2:.1f} Mio. Parameter")
print(f"Größe von GPT2: \
       {model_size(generation_gpt2.model)/1000**2:.1f} Mio. Parameter")
```

```
Größe von GPT: 116.5 Mio. Parameter
Größe von GPT2: 124.4 Mio. Parameter
```

2 J. Bandy und N. Vincent, »Addressing *Documentation Debt* in Machine Learning Research: A Retrospective Datasheet for BookCorpus« (*https://arxiv.org/abs/2105.05241*), (2021).

Das ursprüngliche GPT-Modell ist ungefähr genauso groß wie das kleinste GPT-2-Modell. Nun können wir mit jedem Modell drei verschiedene Vervollständigungen generieren lassen, wobei wir jeweils denselben Prompt als Eingabe vorgeben:

```
def enum_pipeline_ouputs(pipe, prompt, num_return_sequences):
    out = pipe(prompt, num_return_sequences=num_return_sequences,
               clean_up_tokenization_spaces=True)
    return "\n".join(f"{i+1}." + s["generated_text"] for i, s in enumerate(out))

prompt = "\nWhen they came back"
print("Vervollständigungen von GPT:\n" + \
       enum_pipeline_ouputs(generation_gpt, prompt, 3))
print("")
print("Vervollständigungen von GPT-2:\n" + \
       enum_pipeline_ouputs(generation_gpt2, prompt, 3))
```

```
Vervollständigungen von GPT:
1.
When they came back.
 " we need all we can get, " jason said once they had settled into the back of
the truck without anyone stopping them. " after getting out here, it 'll be up
to us what to find. for now
2.
When they came back.
 his gaze swept over her body. he 'd dressed her, too, in the borrowed clothes
that she 'd worn for the journey.
 " i thought it would be easier to just leave you there. " a woman like
3.
When they came back to the house and she was sitting there with the little boy.
 " don't be afraid, " he told her. she nodded slowly, her eyes wide. she was so
lost in whatever she discovered that tom knew her mistake

Vervollständigungen von GPT-2:
1.
When they came back we had a big dinner and the other guys went to see what
their opinion was on her. I did an hour and they were happy with it.
2.
When they came back to this island there had been another massacre, but he could
not help but feel pity for the helpless victim who had been left to die, and
that they had failed that day. And so was very, very grateful indeed.
3.
When they came back to our house after the morning, I asked if she was sure. She
said, "Nope." The two kids were gone that morning. I thought they were back to
being a good friend.
```

Bereits ein paar wenige beispielhafte Generierungen lassen deutlich erkennen, dass das GPT-Modell eine ausgeprägte »romantische« Neigung besitzt, die sich darin zeigt, dass die Dialoge in der Regel mit einer romantischen Interaktion zwischen einer Frau und einem Mann verbunden sind. Das GPT-2-Modell wurde hingegen auf Onlinetexten trainiert, die auf Reddit-Artikel verlinken als auch in Reddit-Artikeln verlinkt wurden, und verwendet in seinen Generationen, die »blogartige« oder abenteuerliche Elemente enthalten, meist das neutrale Personalpronomen »they«.

Im Allgemeinen wird jedes Modell die sprachlichen Verzerrungen bzw. Eigenheiten und die Über- oder Unterrepräsentierung von Bevölkerungsgruppen und Ereig-

nissen, die in seinen Trainingsdaten inhärent sind, widerspiegeln. Diese Verzerrungen (engl. Bias) im Verhalten des Modells müssen im Hinblick auf die Zielgruppe, die mit dem Modell interagiert, unbedingt berücksichtigt werden. Für einige nützliche Richtlinien verweisen wir Sie auf einen Artikel von Google, der einen Rahmen für die Entwicklung von Datensätzen bietet.[3]

Diese kurze Einführung sollte Ihnen ein Bild von den schwierigen Herausforderungen vermittelt haben, denen Sie sich bei der Erstellung großer Textkorpora gegenübersehen. Werfen wir nun einen Blick darauf, wie wir vorgehen können, um unseren eigenen Datensatz zu erstellen!

Einen eigenen Codedatensatz erstellen

Um die Aufgabe etwas zu vereinfachen, konzentrieren wir uns auf die Erstellung eines Modells zur Codegenerierung, das ausschließlich die Programmiersprache Python unterstützt.[4] Als Erstes benötigen wir ein großes Pretraining-Korpus, das aus Python-Quellcode besteht. Glücklicherweise gibt es eine naheliegende Ressource, die jeder Softwareentwickler kennt: GitHub! Die berühmte Code-Sharing-Webseite beherbergt Terabytes an Code-Repositories, die frei zugänglich sind und gemäß ihrer jeweiligen Lizenzen heruntergeladen und verwendet werden dürfen. Zum Zeitpunkt des Verfassens dieses Buchs befinden sich auf GitHub mehr als 20 Millionen Repositories, die Code enthalten. Viele davon sind kleine oder Test-Repositories, die von Benutzern angelegt wurden, um Programmieren zu lernen, künftige Nebenprojekte zu entwickeln oder zu testen.

Auf die GitHub-Repositories kann auf zwei Arten zugegriffen werden:

- über die REST API von GitHub (*https://oreil.ly/brhxw*), die wir bereits in Kapitel 9 kennengelernt haben, als wir alle GitHub-Issues des Repository der 🤗 Transformers-Bibliothek heruntergeladen haben
- über öffentlich zugängliche Datenbestände wie Google BigQuery (*https://oreil.ly/dYsVT*)

Da die Anzahl der Aufrufe, die wir über die REST API vornehmen können, begrenzt ist und wir eine große Menge an Daten für unser Pretraining-Korpus benötigen, werden wir auf Google BigQuery zurückgreifen, um alle Python-Repositories zu extrahieren. Die Tabelle `bigquery-public-data.github_repos.contents` enthält Kopien aller ASCII-Dateien, die weniger als 10 MB groß sind. Wie in der Lizenz zur Nutzung der GitHub-API (*https://oreil.ly/N9zHb*) vorgegeben, werden nur Projekte berücksichtigt, die mit einer Open-Source-Lizenz verknüpft sind.

3 B. Hutchinson et al., »Towards Accountability for Machine Learning Datasets: Practices from Software Engineering and Infrastructure« (*https://arxiv.org/abs/2010.13561*), (2020).

4 Zum Vergleich: GitHub Copilot unterstützt mehr als ein Dutzend Programmiersprachen.

Der Google-BigQuery-Datensatz enthält keine Informationen darüber, wie viele Stars ein Repository erhalten hat oder ob es in anderen Projekten verwendet wird. Für diese Attribute können wir die REST API von GitHub oder einen Dienst wie Libraries.io (*https://libraries.io*) verwenden, der Open-Source-Pakete trackt. Beispielsweise hat ein Team von GitHub vor Kurzem einen Datensatz namens CodeSearchNet (*https://oreil.ly/daE43*) veröffentlicht, bei dem Repositories anhand von Informationen, die auf Libraries.io zur Verfügung stehen, herausgefiltert wurden, die in mindestens einem anderen Projekt verwendet werden.

Sehen wir uns nun an, wie wir unseren Codedatensatz mithilfe von Google BigQuery erstellen können.

Einen Datensatz mit Google BigQuery erstellen

Beginnen wir damit, alle Python-Dateien aus den GitHub-Repositories, die in dem Snapshot auf Google BigQuery enthalten sind, zu extrahieren. Aus Gründen der Reproduzierbarkeit und für den Fall, dass sich die Richtlinien für die kostenlose Nutzung von BigQuery in Zukunft ändern, stellen wir Ihnen den Datensatz auch auf dem Hugging Face Hub zur Verfügung. Die Schritte, die erforderlich sind, um diese Dateien extrahieren zu können, sind der TransCoder-Implementation (*https://oreil.ly/vih2m*) entnommen[5] und lauten wie folgt:

1. Erstellen Sie ein Google-Cloud-Konto (eine kostenlose Testversion sollte ausreichend sein).
2. Legen Sie über Ihr Konto ein Google-BigQuery-Projekt an, indem Sie »New Project« auswählen.
3. In diesem Projekt erstellen Sie nun einen Datensatz über »View actions > Create dataset«.
4. Erstellen Sie in diesem Datensatz eine Tabelle (»Create table«), in der die Ergebnisse der SQL-Abfrage gespeichert werden sollen.
5. Bereiten Sie die folgende SQL-Abfrage vor und führen Sie sie auf `github_repos` aus (um die Abfrageergebnisse zu speichern, wählen Sie »More > Query Options«, markieren das Feld »Set a destination table for query results« und geben den Tabellennamen an):

```
SELECT
  f.repo_name, f.path, c.copies, c.size, c.content, l.license
FROM
  `bigquery-public-data.github_repos.files` AS f
JOIN
  `bigquery-public-data.github_repos.contents` AS c
ON
  f.id = c.id
```

5 M.-A. Lachaux et al., »Unsupervised Translation of Programming Languages« (*https://arxiv.org/abs/2006.03511*), (2020).

```
JOIN
  `bigquery-public-data.github_repos.licenses` AS l
ON
  f.repo_name = l.repo_name
WHERE
  NOT c.binary
  AND ((f.path LIKE '%.py')
    AND (c.size BETWEEN 1024
      AND 1048575))
```

Mit diesem Befehl werden etwa 2,6 TB an Daten verarbeitet und 26,8 Millionen Dateien extrahiert. Der resultierende Datensatz besteht aus etwa 50 GB komprimierter JSON-Dateien, die jeweils den Quellcode von Python-Dateien enthalten. Wir haben leere Dateien und kleine Dateien wie *__init__.py* herausgefiltert, die kaum nützliche Informationen enthalten. Zudem haben wir Dateien außen vor gelassen, die größer als 1 MB sind. Außerdem haben wir die Lizenzen für alle Dateien heruntergeladen, damit wir die Trainingsdaten später bei Bedarf anhand der Lizenzen filtern können.

Als Nächstes werden wir das Ergebnis der Abfrage auf unseren lokalen Rechner herunterladen. Falls Sie dies in Ihrer lokalen Umgebung umsetzen, stellen Sie sicher, dass Sie über eine angemessene Bandbreite und mindestens 50 GB freien Speicherplatz verfügen. Der einfachste Weg, die resultierende Tabelle auf Ihren lokalen Rechner zu bekommen, ist der folgende zweistufige Ansatz:

1. Exportieren Sie Ihre Ergebnisse in die Google Cloud:
   a. Erstellen Sie ein Bucket und einen Ordner in Google Cloud Storage (GCS).
   b. Exportieren Sie Ihre Tabelle in diesen Bucket, indem Sie »Export > Export to GCS« wählen, und geben Sie als Exportformat JSON sowie als Komprimierungstyp gzip an.
2. Um das Bucket auf Ihren Rechner herunterzuladen, können Sie die `gsutil`-Bibliothek (*https://oreil.ly/JzgRk*) verwenden:
   a. Installieren Sie `gsutil` mit dem Kommmandozeilenbefehl `pip install gsutil`.
   b. Konfigurieren Sie `gsutil` mit Ihrem Google-Konto mithilfe des Befehls `gsutil config`.
   c. Kopieren Sie das Bucket auf Ihren Rechner:

```
$ gsutil -m -o
"GSUtil:parallel_process_count=1" cp -r gs://<name_of_bucket>
```

Alternativ können Sie den Datensatz auch direkt vom Hugging Face Hub über den folgenden Befehl herunterladen:

```
$ git clone https://huggingface.co/datasets/transformersbook/codeparrot
```

Das Rauschen filtern oder nicht?

Dadurch, dass jeder beliebiger Mensch ein GitHub-Repository anlegen kann, variiert die Qualität der Projekte. Wir müssen einige grundlegende Entscheidungen darüber treffen, wie wir das System in einer realen Umgebung einsetzen möchten. Ein gewisses Maß an (Grund-)Rauschen (engl. Noise) im Trainingsdatensatz macht unser System robuster gegenüber verrauschten Eingaben, die im Rahmen von Vorhersagen (Inferenz) im Produktionsbetrieb getroffen werden sollen, bewirkt aber auch, dass die Vorhersagen stärker durch Zufall bestimmt werden. Je nachdem, welcher Zweck mit dem Modell verfolgt wird, und je nachdem, wie sehr das System integriert werden soll, können Sie mehr oder weniger verrauschte Daten und auch zusätzliche Operationen zur Vor- und Nachfilterung berücksichtigen.

Für die didaktischen Zwecke dieses Kapitels und um den Code für die Datenaufbereitung übersichtlich zu halten, werden wir nicht danach filtern, wie viele Stars für ein Repository vergeben wurden bzw. von wie vielen es genutzt wird, und einfach alle Python-Dateien des GitHub-Datensatzes von BigQuery nehmen. In welchem Maße die Daten aufbereitet werden, ist jedoch von entscheidender Bedeutung, und Sie sollten sicherstellen, dass Sie Ihren Datensatz so gut wie möglich bereinigen. In unserem Fall müssen wir entscheiden, ob wir die Verteilung der Programmiersprachen im Datensatz ausgewogener gestalten, Daten von geringer Qualität herausfiltern (z. B. mithilfe von GitHub-Stars oder Referenzierungen aus anderen Repositories), Duplikate von Codebeispielen entfernen, Urheberrechtsinformationen berücksichtigen, die in der Dokumentation, in Kommentaren oder Docstrings verwendete Sprache hinzuziehen, oder auch, ob wir personenbezogene Daten wie Passwörter oder Schlüssel löschen.

Mit einem 50 GB großen Datensatz zu arbeiten, kann durchaus eine Herausforderung darstellen. Zum einen benötigen Sie ausreichend Festplattenspeicher und zum anderen müssen Sie sicherstellen, dass Ihr Arbeitsspeicher groß genug ist. Im folgenden Abschnitt lernen Sie, wie Ihnen die 🤗 Datasets-Bibliothek dabei helfen kann, mit diesen Einschränkungen bei der Arbeit mit großen Datensätzen auf einem kleinen Rechner zurechtzukommen.

Mit großen Datensätzen arbeiten

Einen sehr umfangreichen Datensatz zu laden, gestaltet sich oft recht schwierig, insbesondere dann, wenn die Daten nicht in den Arbeitsspeicher (RAM) Ihres Rechners passen, was häufig bei großen Pretrainings-Datensätzen der Fall ist. In unserem Fall liegen 50 GB komprimierte und etwa 200 GB unkomprimierte Daten vor. Angesichts dieser Größe könnte es schwierig sein, sie in den Arbeitsspeicher eines Laptops oder Desktop-PCs normaler Größe zu laden.

Zu unserem Glück wurde die Datasets-Bibliothek von Grund auf so konzipiert, dass diesem Problem dank zweier spezieller Features begegnet werden kann und Sie nicht an die Beschränkungen Ihres Festplatten- und Arbeitsspeichers gebunden sind: Memory Mapping und Streaming.

Memory Mapping

Um die Beschränkungen des Arbeitsspeichers zu überwinden, verwendet die 🤗 Datasets-Bibliothek einen Mechanismus für sogenanntes Zero-Copy- und Zero-Overhead-Memory Mapping (»Speichereinblendung«), das standardmäßig aktiviert ist. Im Grunde genommen wird jeder Datensatz auf dem Laufwerk in einer Datei zwischengespeichert, die ein direktes Abbild des Inhalts des Arbeitsspeichers darstellt. Anstatt den Datensatz in den Arbeitsspeicher zu laden, richtet die 🤗 Datasets-Bibliothek einen Read-Only-Pointer auf diese Datei und verwendet diesen als Ersatz für den Arbeitsspeicher, wobei die Festplatte im Prinzip als eine direkte Erweiterung des Arbeitsspeichers verwendet wird.

Bis jetzt haben wir die 🤗 Datasets-Bibliothek meist dafür verwendet, auf Remote-Datensätze im Hugging Face Hub zuzugreifen. Hier werden wir direkt die 50 GB an komprimierten JSON-Dateien laden, die wir lokal im `codeparrot`-Repository gespeichert haben. Da die JSON-Dateien komprimiert sind, müssen wir sie zunächst mithilfe der 🤗 Datasets-Bibliothek entpacken. Achten Sie jedoch darauf, dass Sie ausreichend Speicherplatz zur Verfügung haben – es erfordert etwa 180 GB freien Speicher! Allerdings benötigen Sie dabei fast keinen Arbeitsspeicher. Indem wir `delete_extracted=True` in der Download-Konfiguration (`DownloadConfig`) des Dataset-Objekts setzen, können wir sicherstellen, dass wir alle Dateien, die wir nicht mehr benötigen, so schnell wie möglich löschen:

```
from datasets import load_dataset, DownloadConfig

download_config = DownloadConfig(delete_extracted=True)
dataset = load_dataset("./codeparrot", split="train",
                       download_config=download_config)
```

Im Hintergrund hat die 🤗 Datasets-Bibliothek dafür gesorgt, dass alle komprimierten JSON-Dateien in eine einzelne optimierte Cache-Datei geladen wurden. Ermitteln wir nun, wie groß dieser Datensatz ist, nachdem er geladen wurde:

```
import psutil, os

print(f"Anzahl der Python-Dateien im Datensatz: {len(dataset)}")
ds_size = sum(os.stat(f["filename"]).st_size for f in dataset.cache_files)
# os.stat.st_size ist in Bytes angegeben, also konvertieren wir den Wert in GB
print(f"Größe des Datensatzes (Cache-Datei): {ds_size / 2**30:.2f} GB")
# Process.memory_info ist in Bytes angegeben, also konvertieren wir Wert in MB
print(f"Verwendeter RAM: \
        {psutil.Process(os.getpid()).memory_info().rss >> 20} MB")
```

```
Anzahl der Python-Dateien im Datensatz: 18695559
Größe des Datensatzes (Cache-Datei): 183.68 GB
Verwendeter RAM: 4924 MB
```

Wie wir feststellen, ist der Datensatz bedeutend größer als ein gewöhnlicher Arbeitsspeicher, wobei es dennoch möglich ist, ihn zu laden und darauf zuzugreifen. Hinsichtlich des Arbeitsspeichers können wir festhalten, dass dieser tatsächlich nur sehr begrenzt genutzt wird.

Sie fragen sich vielleicht, ob unser Training dadurch an die Grenzen des I/O (Input/Output bzw. E/A, Eingabe/Ausgabe) gebunden ist. In der Praxis können NLP-Daten in der Regel relativ leicht geladen werden, insbesondere im Vergleich zu den erforderlichen Berechnungen im Rahmen der Verarbeitung eines Modells, sodass dies nur selten ein Problem darstellt. Darüber hinaus verwendet das Zero-Copy- bzw. Zero-Overhead-Format im Hintergrund Apache Arrow, wodurch der Zugriff auf jedes Element sehr effizient erfolgt. Abhängig von der Geschwindigkeit Ihrer Festplatte und der Batchgröße kann über den Datensatz in der Regel mit einer Geschwindigkeit von einigen Zehnteln GB/s bis zu mehreren GB/s iteriert werden. Das mag zwar großartig sein, aber was ist, wenn Sie nicht genügend Speicherplatz zur Verfügung haben, um den gesamten Datensatz lokal zu speichern? Sicherlich kennen Sie auch dieses Gefühl der Hilflosigkeit, wenn Ihnen eine Warnung eingeblendet wird, dass die Festplatte bereits vollständig belegt ist und Sie erst mühsam versuchen müssen, ein paar GB freizubekommen, indem Sie nach versteckten Dateien suchen, die Sie löschen können. Doch glücklicherweise sind Sie nicht darauf angewiesen, den gesamten Datensatz lokal speichern zu müssen, wenn Sie auf das Streaming-Feature der 🤗 Datasets-Bibliothek zurückgreifen.

Streaming

Große Datensätze, die 1 TB groß oder gar noch größer sind, lassen sich selbst auf einer Standardfestplatte nur schwer unterbringen. In diesem Fall besteht eine Alternative zum Hochskalieren des von Ihnen verwendeten Servers darin, den Datensatz zu *streamen*. Dies ist auch mit der 🤗 Datasets-Bibliothek für eine Reihe von komprimierten oder unkomprimierten Dateiformaten möglich, die zeilenweise gelesen werden können, wie z. B. bei JSON-, CSV- oder Textdateien (entweder in Form von Rohdaten oder komprimiert im zip-, gzip- oder zstandard-Format). Laden wir unseren Datensatz nun direkt über die komprimierten JSON-Dateien, anstatt eine Cache-Datei daraus zu erstellen:

```
streamed_dataset = load_dataset('./codeparrot', split="train", streaming=True)
```

Wie Sie feststellen werden, wird der Datensatz bei diesem Ansatz unmittelbar geladen. Im Streaming-Modus werden die komprimierten JSON-Dateien geöffnet und in Echtzeit (on the fly) ausgelesen. Unser Datensatz entspricht nun einem `IterableDataset`-Objekt. Das bedeutet, dass wir nicht beliebig auf Elemente wie `streamed_dataset[1264]` zugreifen können, sondern dass wir sie der Reihe nach einspeisen müssen, zum Beispiel mit `next(iter(streamed_dataset))`. Methoden wie `shuffle()` können zwar immer noch verwendet werden, allerdings wird dabei ein Puffer von Beispielen abgerufen, und die Beispiele innerhalb dieses Puffers werden gemischt (die Größe des Puffers kann angepasst werden). Wenn mehrere Dateien als Rohdateien bereitgestellt werden (wie in unserem Fall mit 184 Dateien), sorgt `shuffle()` auch dafür, dass die Reihenfolge der Dateien beim Iterieren zufällig bestimmt wird.

Wie ersichtlich ist, sind die Beispiele des gestreamten Datensatzes identisch mit den Beispielen des nicht gestreamten Datensatzes:

```
iterator = iter(streamed_dataset)

print(dataset[0] == next(iterator))
print(dataset[1] == next(iterator))
```

```
True
True
```

Der Hauptvorteil eines gestreamten Datensatzes besteht darin, dass beim Laden dieses Datensatzes keine Cache-Datei auf dem Laufwerk erstellt und nur sehr wenig Arbeitsspeicher benötigt wird. Die ursprünglichen Rohdateien werden extrahiert und unmittelbar gelesen, wenn ein neues Batch von Beispielen abgerufen wird, wobei nur dieses Batch in den Speicher geladen wird. Dadurch verringert sich der Speicherbedarf unseres Datensatzes von 180 GB auf 50 GB. Doch wir können noch einen Schritt weiter gehen: Anstatt auf den lokalen Datensatz können wir auf den Datensatz, der sich auf dem Hub befindet, verweisen und die Beispiele direkt herunterladen, ohne dass dabei die Rohdateien lokal heruntergeladen werden müssen:

```
remote_dataset = load_dataset('transformersbook/codeparrot', split="train",
                              streaming=True)
```

Dieses Dataset-Objekt verhält sich genau so wie das vorherige, allerdings werden die Beispiele im Hintergrund »on the fly« heruntergeladen. Mit diesem Ansatz können wir beliebig große Datensätze auf einem (fast) beliebig kleinen Server verwenden. Übertragen bzw. pushen wir nun unseren Datensatz auf den Hugging Face Hub, wobei wir eine Aufteilung in einen Trainings- und einen Validierungsdatensatz vornehmen und ihn mittels Streaming abrufen.

Datensätze zum Hugging Face Hub hinzufügen

Wenn wir unseren Datensatz auf den Hugging Face Hub übertragen, können wir:

- ganz einfach von unserem Trainingsserver aus darauf zugreifen.
- beobachten, wie Streaming-Datensätze völlig nahtlos mit Datensätzen aus dem Hub zusammenarbeiten.
- ihn mit der Community teilen, auch mit Ihnen, liebe Leserin und lieber Leser!

Um den Datensatz hochladen zu können, müssen wir uns zunächst bei unserem Hugging-Face-Konto anmelden, indem wir den folgenden Befehl im Terminal ausführen und die entsprechenden Zugangsdaten (Credentials) eingeben:

```
$ huggingface-cli login
```

Dieser Befehl entspricht der Hilfsfunktion `notebook_login()`, die wir in den vorherigen Kapiteln verwendet haben. Anschließend können wir direkt einen neuen Datensatz auf dem Hub erstellen und die komprimierten JSON-Dateien hochladen. Der Einfachheit halber werden wir zwei Repositories erstellen: eines für den Trainingsdatensatz und eines für den Validierungsdatensatz. Hierzu können wir im Terminal den Befehl `repo create` wie folgt ausführen:

```
$ huggingface-cli repo create --type dataset --organization transformersbook \
codeparrot-train
$ huggingface-cli repo create --type dataset --organization transformersbook \
codeparrot-valid
```

Hier haben wir angegeben, dass es sich bei dem Repository um einen Datensatz handeln soll (im Gegensatz zu den Modell-Repositories, die zum Speichern von Gewichtungen verwendet werden), sowie die Organisation, unter der wir die Repositories speichern möchten. Wenn Sie diesen Code über Ihr persönliches Konto ausführen, können Sie das `--organization`-Flag weglassen. Als Nächstes müssen wir diese leeren Repositories auf unseren lokalen Rechner klonen, die JSON-Dateien hineinkopieren und die Änderungen auf den Hub übertragen bzw. pushen. Die letzte komprimierte der 184 JSON-Dateien, die wir vorliegen haben, nehmen wir als Validierungsdatei (d.h. etwa 0,5 Prozent unseres Datensatzes). Führen Sie die folgenden Befehle aus, um die Repositories vom Hub auf Ihren lokalen Rechner zu klonen:

```
$ git clone https://huggingface.co/datasets/transformersbook/codeparrot-train
$ git clone https://huggingface.co/datasets/transformersbook/codeparrot-valid
```

Anschließend können Sie alle Dateien außer der letzten GitHub-Datei als Trainingsdatensatz kopieren:

```
$ cd codeparrot-train
$ cp ../codeparrot/*.json.gz .
$ rm ./file-000000000183.json.gz
```

Committen Sie dann die Dateien und pushen Sie sie auf den Hub:

```
$ git add .
$ git commit -m "Adding dataset files"
$ git push
```

Nehmen Sie nun die gleichen Schritte für den Validierungsdatensatz vor:

```
$ cd ../codeparrot-valid
$ cp ../codeparrot/file-000000000183.json.gz .
$ mv ./file-000000000183.json.gz ./file-000000000183_validation.json.gz
$ git add .
$ git commit -m "Adding dataset files"
$ git push
```

Nachdem Sie den Befehl `git add .` eingegeben haben, kann es ein paar Minuten dauern, da infolge dessen ein Hashing aller Dateien vorgenommen wird. Das Hochladen aller Dateien nimmt ebenfalls eine Weile in Anspruch. Da wir später in diesem Kapitel das Streaming-Feature nutzen werden, verlieren wir hierdurch keine Zeit, weil wir auf diese Weise den Rest unserer Experimente deutlich schneller durchführen können. Beachten Sie, dass wir dem Namen der Validierungsdatei das Suffix `__validation_` hinzugefügt haben. Auf diese Weise können wir sie später relativ bequem als Validierungsdatensatz laden.

So, wir sind nun so weit! Unsere beiden Teildatensätze sowie der vollständige Datensatz sind jetzt auf dem Hugging Face Hub unter den folgenden URLs verfügbar:

- *https://huggingface.co/datasets/transformersbook/codeparrot*
- *https://huggingface.co/datasets/transformersbook/codeparrot-train*
- *https://huggingface.co/datasets/transformersbook/codeparrot-valid*

Es empfiehlt sich, sogenannte README-Cards bereitzustellen, in denen erklärt wird, wie die Datensätze erstellt wurden, und so viele nützliche Informationen wie möglich über sie zur Verfügung zu stellen. Ein gut dokumentierter Datensatz ist mit größerer Wahrscheinlichkeit auch für andere Personen und auch für Sie selbst nützlich. Lesen Sie am besten den README-Leitfaden (*https://oreil.ly/Tv9bq*) der 🤗 Datasets-Bibliothek, in dem ausführlich beschrieben wird, wie Sie eine gute Dokumentation für einen Datensatz anfertigen können. Sie können auch den Webeditor verwenden, wenn Sie Ihre README-Cards zu einem späteren Zeitpunkt direkt auf dem Hub ändern möchten.

Erstellung eines Tokenizers

Nachdem wir nun unseren großen Datensatz zusammengestellt und geladen haben, widmen wir uns anschließend der Frage, wie wir die Daten effizient verarbeiten können, um sie in unser Modell einzuspeisen. In den vorangegangenen Kapiteln haben wir Tokenizer verwendet, die jeweils an die von uns verwendeten Modelle gekoppelt waren. Das war durchaus sinnvoll, denn diese Modelle wurden auf Basis von Daten vortrainiert, die eine vom Tokenizer abhängige Vorverarbeitungspipeline durchlaufen hatten. Wenn Sie ein vortrainiertes Modell verwenden, ist es wichtig, dass Sie dieselben Entscheidungen für die Vorverarbeitung treffen, die Sie auch für das Pretraining gewählt haben. Andernfalls könnte das Modell mit Mustern, die nicht in der Verteilung des Modells vorkommen, oder unbekannten Tokens konfrontiert werden.

Wenn wir jedoch ein neues Modell trainieren, kann es sich als suboptimal erweisen, einen Tokenizer zu verwenden, der für einen anderen Datensatz konzipiert wurde. Die folgenden zwei Beispiele verdeutlichen die Probleme, die bei der Verwendung eines vorhandenen Tokenizers auftreten können:

- Der T5-Tokenizer wurde mit dem C4 (*https://oreil.ly/wsYIC*)-Korpus trainiert, das wir bereits zuvor kennengelernt haben. Allerdings wurde im Rahmen der Erstellung ein umfangreicher Schritt zur Filterung von Stoppwörtern durchgeführt. Infolgedessen hat der T5-Tokenizer einige gebräuchliche englische Wörter wie »sex« noch nie gesehen.
- Der CamemBERT-Tokenizer wurde ebenfalls mit einem sehr großen Textkorpus trainiert, das jedoch nur französische Texte enthält (die französische Teilmenge des OSCAR-Korpus (*https://oreil.ly/hgO5J*)). Deshalb kennt er keine gängigen englischen Wörter wie »being«.

Dass dies bei den beiden Tokenizern tatsächlich der Fall ist, können wir leicht selbst nachprüfen:

```
from transformers import AutoTokenizer

def tok_list(tokenizer, string):
    input_ids = tokenizer(string, add_special_tokens=False)["input_ids"]
    return [tokenizer.decode(tok) for tok in input_ids]

tokenizer_T5 = AutoTokenizer.from_pretrained("t5-base")
tokenizer_camembert = AutoTokenizer.from_pretrained("camembert-base")

print(f'Tokens von T5 für "sex": {tok_list(tokenizer_T5,"sex")}')
print(f'Tokens von CamemBERT für "being": \
        {tok_list(tokenizer_camembert,"being")}')

Tokens von T5 für "sex": ['', 's', 'ex']
Tokens von CamemBERT für "being": ['be', 'ing']
```

In vielen Fällen ist es ineffizient, so kurze und gebräuchliche Wörter in einzelne Teile aufzuteilen, da dies die Länge der Eingabesequenz des Modells (das nur eine begrenzte Kontextlänge hat) erhöht. Daher ist es wichtig, sich bewusst zu machen, mit welcher Vorverarbeitung der Datensatz, der zum Trainieren des Tokenizers verwendet wurde, erstellt wurde und aus welcher Domäne er stammt. Der Tokenizer und das Modell können die in dem Datensatz enthaltenen Vorurteile codieren, was sich auf das spätere Verhalten des Modells auswirkt. Damit wir einen Tokenizer erstellen können, der für unseren Datensatz optimal ist, müssen wir folglich selbst einen trainieren. Werfen wir ein Blick darauf, wie sich das umsetzen lässt.

Beim Training eines Modells gehen Sie von einem gegebenen Satz von Gewichten aus und wenden eine Backpropagation (Fehlerrückführung durch das Netz) an, um den Verlust des Modells zu minimieren und dabei eine optimale Gewichtung für das Modell zu finden, sodass die für das Training vorgegebene Aufgabe (Objective) bestmöglich erfüllt wird. Beim Trainieren eines Tokenizers wird hingegen *keine* Backpropagation durchgeführt und es werden auch *keine* Gewichte verwendet. Beim Training des Tokenizers besteht das Ziel darin, einen String auf eine Liste von Ganzzahlen auf optimale Weise abzubilden, sodass diese anschließend vom Modell verwendet werden kann. Bei den Tokenizern, die derzeit verwendet werden, wird für die optimale Konvertierung von Strings in Ganzzahlen ein Vokabular genutzt, das aus einer Liste von einzelnen Strings besteht, und eine zugehörige Methode, mit der die Strings in eine Liste von Indexen auf Basis dieses Vokabulars konvertiert, normalisiert, aufgeteilt oder darauf abgebildet werden. Diese Liste von Indexen wird dann als Eingabe für unser neuronales Netz verwendet.

Das Tokenizer-Modell

Wie Sie bereits in Kapitel 4 erfahren haben, handelt es sich bei einem Tokenizer um eine Verarbeitungspipeline, die aus vier Schritten besteht: der Normalisierung, der Pretokenization, dem Tokenizer-Modell und der Nachverarbeitung. Der Teil der Tokenizer-Pipeline, der auf Basis von Daten trainiert werden kann, ist das Tokenizer-Modell. Wie in Kapitel 2 erläutert, gibt es mehrere Algorithmen, die zur Toke-

nisierung auf der Ebene von Teilwörtern (engl. Subwords) verwendet werden können, z. B. Byte-Pair Encoding (BPE), WordPiece und Unigram.

Beim BPE wird von einer Liste von Basiseinheiten (einzelnen Zeichen) ausgegangen und ein Vokabular erstellt, indem nach und nach neue Tokens erstellt und dem Vokabular hinzugefügt werden, die durch Zusammenführen der am häufigsten vorkommenden Basiseinheiten gebildet werden. Dieser Vorgang wird so lange wiederholt, bis eine vorgegebene Größe des Vokabulars erreicht ist.

Der Unigram-Tokenizer geht umgekehrt vor, indem er sein Basisvokabular mit allen Wörtern im Korpus und potenziellen Teilwörtern initialisiert. Dann werden nach und nach die weniger nützlichen Tokens entfernt oder aufgeteilt, um ein immer kleineres Vokabular zu erhalten, bis die angestrebte Größe des Vokabulars erreicht ist. Der WordPiece-Tokenizer ist ein Vorgänger des Unigram-Tokenizers, dessen offizielle Implementierung von Google nie veröffentlicht wurde.

Die Auswirkungen dieser verschiedenen Algorithmen auf die Leistung nachgelagerter Aufgaben variieren je nach Aufgabe, und insgesamt ist es recht schwierig, zu erkennen, ob ein Algorithmus einem anderen eindeutig überlegen ist. Sowohl der BPE- als auch Unigram-Algorithmus weisen in den meisten Fällen eine angemessene Leistung auf. Werfen wir dennoch einen Blick auf einige Aspekte, die bei der Evaluierung zu berücksichtigen sind.

Die Leistung eines Tokenizers beurteilen

Ob ein Tokenizer wirklich optimal und leistungsfähig ist, lässt sich nur schwer beurteilen. Einige mögliche Maße, die herangezogen werden können, sind:

- die *Subword Fertility*, bei der die durchschnittliche Anzahl von Teilwörtern je tokenisiertem Wort ermittelt wird
- der *Proportion of continued Words*, wobei es sich um den Anteil der tokenisierten Wörter in einem Korpus handelt, die in mindestens zwei Subtokens aufgeteilt wurden
- *Abdeckungsmaße* (engl. Coverage Metrics) wie der Anteil unbekannter Wörter oder selten verwendeter Tokens in einem tokenisierten Korpus

Darüber hinaus wird häufig über eine Schätzung ermittelt, wie robust das Modell gegenüber Rechtschreibfehlern oder Rauschen in den Daten ist und wie gut es bei Beispielen, die außerhalb der Domäne liegen, funktioniert, da dies in hohem Maße vom Tokenisierungsvorgang abhängt.

Diese Maße vermitteln eine Reihe verschiedener Aspekte der Leistung des Tokenizers, aber sie berücksichtigen in der Regel nicht, wie der Tokenizer mit dem Modell interagiert. Zum Beispiel kann die Subword Fertility verringert werden, indem alle möglichen Wörter in das Vokabular aufgenommen werden – allerdings auf Kosten eines sehr großen Vokabulars für das Modell.

Letztendlich lässt sich die Leistung der verschiedenen Tokenisierungsansätze also am besten abschätzen, indem die nachgelagerte Leistung des Modells als entscheidendes Kriterium herangezogen wird. Die gute Leistung der frühen Ansätze beim BPE-Tokenizer wurde beispielsweise dadurch nachgewiesen, dass Modelle, die mit diesen Tokenizern und Vokabularen (anstelle der Tokenisierung auf der Ebene von Zeichen oder Wörtern) trainiert wurden, eine bessere Leistung bei maschinellen Übersetzungsaufgaben erzielten.

Sehen wir uns an, wie wir unseren eigenen Tokenizer erstellen können, der für Python-Code optimiert ist.

Ein Tokenizer für die Programmiersprache Python

Im Rahmen unseres Anwendungsfalls müssen wir Python-Code tokenisieren – aus diesem Grund benötigen wir einen selbst erstellten Tokenizer. Bei Programmiersprachen ist die Frage, wie die Pretokenization durchgeführt wird, durchaus diskussionswürdig. Wenn wir einen Text anhand von Leerzeichen aufteilen und diese entfernen, verlieren wir alle Informationen darüber, wie der Text eingerückt ist. In Python sind Einrückungen allerdings sehr wichtig für die Semantik des Programms (denken Sie nur an `while`-Schleifen oder `if-else`-Anweisungen). Im Gegensatz dazu, sind Zeilenumbrüche nicht von Bedeutung und können eingefügt oder entfernt werden, ohne dass sich dies auf die Semantik auswirkt. Auch die Aufteilung anhand von Interpunktionszeichen, wie z. B. an einem Unterstrich, ist gegebenenfalls nicht so sinnvoll wie bei natürlichen Sprachen, da er in Python oftmals dazu verwendet wird, um einzelne Variablennamen aus mehreren Teilen zusammenzusetzen. Daher scheint es möglicherweise suboptimal, einen natürlichsprachlichen Pretokenizer für die Tokenisierung von Code zu verwenden.

Mal sehen, ob es in der auf dem Hub bereitgestellten Sammlung Tokenizer gibt, die für uns von Nutzen sein könnten. Da wir einen Tokenizer verwenden möchten, der Leerzeichen beibehält, könnte ein guter Kandidat ein Tokenizer sein, der auf der Ebene von Bytes operiert – wie der von GPT-2. Laden wir diesen Tokenizer, um einmal genauer nachzuvollziehen, wie er Texte tokenisiert:

```
from transformers import AutoTokenizer

python_code = r"""def say_hello():
    print("Hello, World!")

# Print it
say_hello()
"""
tokenizer = AutoTokenizer.from_pretrained("gpt2")
print(tokenizer(python_code).tokens())
```

```
['def', 'Ġsay', '_', 'hello', '():', 'Ċ', 'Ġ', 'Ġ', 'Ġ', 'Ġprint', '("',
'Hello', ',', 'ĠWorld', '!"', ')', 'Ġ#', 'ĠPrint', 'Ġit', 'Ċ', 'Ċ', 'say', '_',
'hello', '()', 'Ċ']
```

Python verfügt über ein integriertes `tokenize`-Modul, das Python-Code-Strings in sinnvolle Einheiten aufteilt (Codeoperation, Kommentare, Einrückung, Ausrückung usw.). Eine Schwachstelle bei diesem Ansatz ist, dass der Pretokenizer in Python programmiert wurde und als solcher in der Regel recht langsam und zudem durch den Global Interpreter Lock (GIL) von Python eingeschränkt ist. Die meisten Tokenizer, die in der 🤗 Transformers-Bibliothek verfügbar sind, stammen aus der 🤗 Tokenizers-Bibliothek. Der Vorteil bei diesen Tokenizern ist, dass sie in Rust programmiert wurden. Daher können sie um ein Vielfaches schneller trainiert und verwendet werden, was in Anbetracht der Größe unseres Korpus von Vorteil ist.

Die Ausgabe ist recht seltsam. Versuchen wir einmal zu verstehen, was hier vor sich geht, indem wir die verschiedenen Untermodule der Tokenizer-Pipeline ausführen. Sehen wir uns zunächst an, welche Normalisierung in diesem Tokenizer angewandt wird:

```
print(tokenizer.backend_tokenizer.normalizer)
```

```
None
```

Scheinbar wird beim GPT-2-Tokenizer keine Normalisierung vorgenommen. Er arbeitet direkt mit den Rohdaten im Unicode-Format als Eingaben – ohne, dass es einen Schritt zur Normalisierung gibt. Werfen wir nun einen Blick auf die Pretokenization:

```
print(tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str(python_code))
```

```
[('def', (0, 3)), ('Ġsay', (3, 7)), ('_', (7, 8)), ('hello', (8, 13)), ('():',
(13, 16)), ('ĊĠĠĠ', (16, 20)), ('Ġprint', (20, 26)), ('("', (26, 28)), ('Hello'
,
(28, 33)), (',', (33, 34)), ('ĠWorld', (34, 40)), ('!")', (40, 43)), ('Ġ#', (43,
45)), ('ĠPrint', (45, 51)), ('Ġit', (51, 54)), ('Ċ', (54, 55)), ('Ċ', (55, 56)),
('say', (56, 59)), ('_', (59, 60)), ('hello', (60, 65)), ('()', (65, 67)), ('Ċ',
(67, 68))]
```

Was bedeuten all diese Symbole (wie Ġ), und was bedeuten die Zahlen, die den Tokens zugeordnet sind? Werfen wir einen genaueren Blick auf die Funktionsweise dieses Tokenizers.

Beginnen wir mit den Zahlen. Die 🤗 Tokenizers-Bibliothek verfügt über ein sehr nützliches Feature, mit dem Sie zwischen den Darstellungen als Strings und Tokens wechseln können: *Offset-Tracking*. Alle Operationen, die an einem Input-String vorgenommen werden, werden festgehalten, sodass es möglich ist, genau zu wissen, welchem Teil des Input-String ein Token infolge der Tokenisierung entspricht. Die Zahlen geben schlicht und einfach an, an welcher Stelle des ursprünglichen String sich das jeweilige Token befindet. So entspricht beispielsweise das Wort `'hello'` in der ersten Zeile dem 8. bis 13. Zeichen in dem ursprünglichen String. Selbst wenn in einem Normalisierungsschritt einige Zeichen entfernt würden, können wir jedes Token dem entsprechenden Teil des ursprünglichen String zuordnen.

Das andere eigenartige Merkmal des tokenisierten Texts sind die seltsam aussehenden Zeichen, wie Ċ und Ġ. *Byte-Level* (bzw. auf der Ebene von Bytes) bedeutet, dass dieser Tokenizer auf Basis von Bytes anstelle von Unicode-Zeichen arbeitet. Je nach Zeichen besteht jedes Unicode-Zeichen aus 1 bis 4 Bytes. Das Schöne an Bytes ist, dass es zwar 143.859 Unicode-Zeichen im Unicode-Alphabet gibt, aber nur 256 Elemente im Byte-Alphabet, wobei jedes Unicode-Zeichen als eine Folge von Bytes dargestellt werden kann. Wenn wir mit Bytes arbeiten, können wir also alle im UTF-8-Format zusammengesetzten Strings als längere Strings in diesem Alphabet mit 256 Werten ausdrücken. Das bedeutet, dass wir ein Modell verwenden können, das lediglich ein Alphabet verwendet, das aus 256 Wörtern besteht, und in der Lage ist, jeden Unicode-String zu verarbeiten. Schauen wir uns einmal an, wie einige Zeichen in Form von Bytes dargestellt werden:

```
a, e = u"a", u"€"
byte = ord(a.encode("utf-8"))
print(f'`{a}` ist codiert als `{a.encode("utf-8")}` \
        mit einem einzelnen Byte: {byte}')
byte = [ord(chr(i)) for i in e.encode("utf-8")]
print(f'`{e}` ist codiert als `{e.encode("utf-8")}` mit drei Bytes: {byte}')
```

```
`a` ist codiert als `b'a'` mit einem einzelnen Byte: 97
`€` ist codiert als `b'\xe2\x82\xac'` mit drei Bytes: [226, 130, 172]
```

An dieser Stelle fragen Sie sich vielleicht: Warum sollte ein Tokenizer auf der Ebene von Bytes arbeiten? Denken Sie an unsere Diskussion in Kapitel 2 zurück, in der es darum ging, zwischen der Tokenisierung auf der Ebene von Zeichen und Wörtern abzuwägen. Wir könnten uns dafür entscheiden, unser Vokabular aus den 143.859 Unicode-Zeichen aufzubauen, allerdings wollen wir auch Wörter – d.h. Kombinationen von Unicode-Zeichen – in unser Vokabular aufnehmen, sodass diese (bereits sehr große) Größe nur eine untere Grenze für die Gesamtgröße des Vokabulars darstellt. Dadurch wird die Embedding-Schicht unseres Modells sehr groß, da sie für jedes Token im Vokabular einen Vektor umfasst.

Wenn wir hingegen nur die 256 Byte-Werte als Vokabular verwenden, werden die eingegebenen Sequenzen in viele kleine Teile zerlegt (wobei die einzelnen Bytes Unicode-Zeichen repräsentieren). Dadurch sind die Eingaben, die unser Modell erhält, relativ lang – dementsprechend erfordert es sehr viel Rechenleistung, um die Unicode-Zeichen aus den einzelnen Bytes rekonstruieren und dann Wörter aus diesen Zeichen bilden zu können. Eine ausführliche Studie zu diesem Thema finden Sie im Begleitpapier zur Veröffentlichung des ByT5-Modells.[6]

Ein Mittelweg besteht darin, ein mittelgroßes Vokabular zu konstruieren, indem das 256 Bytes umfassende Vokabular zusätzlich um die häufigsten Kombinationen von Bytes erweitert wird. Dieser Ansatz wird beim BPE-Algorithmus verfolgt. Die Idee besteht darin, schrittweise ein Vokabular einer vorgegebenen Größe aufzu-

6 L. Xue et al., »ByT5: Towards a Token-Free Future with Pre-Trained Byte-to-Byte Models« (*https://arxiv.org/abs/2105.13626*), (2021).

bauen, wobei neue Tokens im Vokabular erstellt werden, indem jeweils das am häufigsten vorkommende Token-Paar im Vokabular iterativ zusammengeführt wird. Wenn zum Beispiel, wie im Englischen, `t` und `h` sehr häufig zusammen vorkommen, fügen wir dem Vokabular ein Token `th` hinzu, um dieses Token-Paar zu modellieren, anstatt sie voneinander getrennt zu lassen. Die Tokens `t` und `h` werden im Vokabular beibehalten, damit Instanzen bzw. Beispiele tokenisiert werden können, in denen die Tokens nicht zusammen vorkommen. Ausgehend von einem Grundvokabular elementarer Einheiten kann dadurch jeder beliebige String effizient modelliert werden.

Vorsicht, verwechseln Sie das »Byte« in »Byte-Pair Encoding« nicht mit dem »Byte« in »Byte-Level«. Der Name Byte-Pair Encoding stammt von einem im Jahr 1994 von Philip Gage vorgeschlagenen Datenkomprimierungsverfahren, das ursprünglich auf Basis von Bytes arbeitete.[7] Anders als der Name vermuten lässt, arbeiten die im NLP standardmäßig verwendeten BPE-Algorithmen in der Regel auf der Basis von Unicode-Strings und nicht auf der Basis von Bytes (obwohl es eine neue Art von Byte-Pair Encoding gibt, das auf Bytes basiert, das sogenannte *Byte-Level BPE*). Deshalb können wir einen einfachen BPE-Algorithmus zur Aufteilung in Teilwörter verwenden, wenn wir unsere Unicode-Strings in Form von Bytes darstellen.

Allerdings gibt es ein Problem bei der Verwendung eines typischen BPE-Algorithmus im NLP. Diese Algorithmen sind darauf ausgelegt, als Eingaben reine Unicode-Strings und nicht Bytes zu verarbeiten, und erwarten normale ASCII-Zeichen – ohne Leer- oder Steuerzeichen. Aber in den Unicode-Zeichen, die den 256 ersten Bytes entsprechen, befinden sich viele Steuerzeichen (Zeilenumbruch, Tabulator, Escape, Zeilenvorschub und andere nicht druckbare Zeichen). Um diesem Problem zu begegnen, ordnet der GPT-2-Tokenizer zunächst alle 256 Input-Bytes Unicode-Strings zu, die von den standardmäßigen BPE-Algorithmen verarbeitet werden können, d.h., wir ordnen unseren 256 elementaren Werten Unicode-Strings zu, die standardmäßigen, druck- bzw. anzeigbaren Unicode-Zeichen entsprechen.

Dabei ist es nicht so wichtig, ob diese Unicode-Zeichen jeweils mit einem oder mehreren Bytes codiert sind. Wichtig ist, dass wir am Ende 256 einzelne Werte haben, die unser Basisvokabular darstellen, und dass diese 256 Werte von unserem BPE-Algorithmus korrekt verarbeitet werden. Sehen wir uns einige Beispiele an, um besser verstehen zu können, wie diese Zuordnung bzw. dieses Mapping beim GPT-2-Tokenizer funktioniert. Wir können auf das gesamte Mapping wie folgt zugreifen:

```
from transformers.models.gpt2.tokenization_gpt2 import bytes_to_unicode

byte_to_unicode_map = bytes_to_unicode()
unicode_to_byte_map = dict((v, k) for k, v in byte_to_unicode_map.items())
base_vocab = list(unicode_to_byte_map.keys())
print(f'Größe unseres Basisvokabulars: {len(base_vocab)}')
print(f'Erstes Element: `{base_vocab[0]}`, letztes Element: `{base_vocab[-1]}`')
```

7 P. Gage, »A New Algorithm for Data Compression«, *The C Users Journal* 12, No. 2 (1994): 23–38, *https://dx.doi.org/10.14569/IJACSA.2012.030803*.

```
Größe unseres Basisvokabulars: 256
Erstes Element: `!`, letztes Element: `Ń`
```

In Tabelle 10-1 finden Sie eine Gegenüberstellung einiger gängiger Byte-Werte und der ihnen zugeordneten Unicode-Zeichen.

Tabelle 10-1: Beispiele für die Zuordnung von Zeichen beim BPE

Beschreibung	Zeichen	Bytes	Zugeordnete Bytes
Gewöhnliche Zeichen	'a' und '?'	97 und 63	'a' und '?'
Nicht druckbares Steuerzeichen (Wagenrücklauf)	'U+000D'	13	'č'
Leerzeichen	''	32	'Ġ'
Geschütztes Leerzeichen	'\xa0'	160	'Ł'
Zeilenumbruchzeichen	'\n'	10	'Ċ'

Wir hätten eine explizitere Form der Konvertierung verwenden können, wie z. B. Zeilenumbrüche auf einen `NEWLINE`-String abzubilden, doch BPE-Algorithmen sind in der Regel auf Zeichen ausgelegt. Aus diesem Grund ist es besser, die Byte-Zeichen jeweils nur mit einem Unicode-Zeichen darzustellen (bzw. zuzuordnen). Nachdem wir nun in die dunkle Magie der Unicode-Codierungen eingetaucht sind, können wir das Ergebnis unserer Tokenisierung ein wenig besser interpretieren:

```
print(tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str(python_code))
```

```
[('def', (0, 3)), ('Ġsay', (3, 7)), ('_', (7, 8)), ('hello', (8, 13)), ('():',
(13, 16)), ('ĊĠĠĠ', (16, 20)), ('Ġprint', (20, 26)), ('("', (26, 28)), ('Hello',
(28, 33)), (',', (33, 34)), ('ĠWorld', (34, 40)), ('!")', (40, 43)), ('Ġ#', (43,
45)), ('ĠPrint', (45, 51)), ('Ġit', (51, 54)), ('Ċ', (54, 55)), ('Ċ', (55, 56)),
('say', (56, 59)), ('_', (59, 60)), ('hello', (60, 65)), ('()', (65, 67)), ('Ċ',
(67, 68))]
```

Wir können die Zeilenumbrüche erkennen, die, wie wir jetzt wissen, dem Zeichen Ċ zugeordnet sind, und die Leerzeichen, die dem Zeichen Ġ zugeordnet sind. Wir sehen auch, dass:

- Leerzeichen, und insbesondere aufeinanderfolgende Leerzeichen, erhalten bleiben (zum Beispiel die drei Leerzeichen in `'ĊĠĠĠ'`).
- aufeinanderfolgende Leerzeichen als ein einzelnes Wort betrachtet werden.
- jedes Leerzeichen, das einem Wort vorausgeht, an das nachfolgende Wort angefügt und als Teil desselben betrachtet wird (z. B. in `'Ġsay'`).

Lassen Sie uns nun ein wenig mit dem BPE-Modell experimentieren. Wie bereits erwähnt, ist es dafür zuständig, die Wörter in Untereinheiten aufzuteilen, bis die vorgegebene Größe des Vokabulars erreicht ist.

Das Vokabular unseres GPT-2-Tokenizers umfasst 50.257 Wörter:

- das Basisvokabular mit den 256 Byte-Werten
- 50.000 zusätzliche Tokens, die durch wiederholtes Zusammenführen der am häufigsten vorkommenden Tokens erstellt wurden

- ein Sonderzeichen, das dem Vokabular hinzugefügt wurde, um die Dokumente voneinander abgrenzen zu können

Das können wir leicht nachprüfen, indem wir uns die Länge des Tokenizer-Objekts ausgeben lassen:

```
print(f"Größe des Vokabulars: {len(tokenizer)}")
```

```
Größe des Vokabulars: 50257
```

Wenn wir unser Python-Codebeispiel in die vollständige Pipeline einspeisen, erhalten wir die folgende Ausgabe:

```
print(tokenizer(python_code).tokens())
```

```
['def', 'Ġsay', '_', 'hello', '():', 'Ċ', 'Ġ', 'Ġ', 'Ġ', 'Ġprint', '("',
'Hello', ',', 'ĠWorld', '!"', ')', 'Ġ#', 'ĠPrint', 'Ġit', 'Ċ', 'Ċ', 'say', '_',
'hello', '()', 'Ċ']
```

Wie wir feststellen, behält der BPE-Tokenizer die meisten Wörter bei, teilt aber die aufeinanderfolgenden Leerzeichen für die Einrückung in mehrere aufeinanderfolgende Leerzeichen auf. Dies liegt daran, dass dieser Tokenizer nicht speziell auf Code trainiert wurde, sondern hauptsächlich auf Texte, in denen aufeinanderfolgende Leerzeichen selten vorkommen. Das BPE-Modell nimmt also kein spezielles Token für eine Einrückung in das Vokabular auf. In diesem Fall ist das Tokenizer-Modell nicht sonderlich gut für die Domäne des Datensatzes geeignet. Wie bereits erläutert, besteht die Lösung darin, den Tokenizer mit dem Zielkorpus neu zu trainieren. Machen wir uns also gleich ans Werk!

Einen Tokenizer trainieren

Wir wollen unseren BPE-Tokenizer, der auf der Ebene von Bytes operiert, auf einen Teil unseres Korpus neu trainieren (engl. retrain), um ein Vokabular zu erhalten, das besser auf Python-Code abgestimmt ist. Das Retraining eines von der 🤗 Transformers-Bibliothek bereitgestellten Tokenizers ist einfach. Wir müssen lediglich:

- die Größe unseres angestrebten Zielvokabulars (engl. Target Vocabulary) angeben.
- einen Iterator vorbereiten, der Listen von Eingabe-Strings liefert, die im Rahmen des Trainings des Tokenizer-Modells verarbeitet werden sollen.
- die Methode `train_new_from_iterator()` aufrufen.

Im Gegensatz zu Deep-Learning-Modellen, von denen oft erwartet wird, dass sie sich eine Menge spezifischer Details aus dem Trainingskorpus merken, werden Tokenizer wirklich nur darauf trainiert, die wichtigsten Statistiken zu extrahieren. Kurz gesagt, der Tokenizer wird nur darauf trainiert, zu ermitteln, welche Buchstabenkombinationen in unserem Korpus am häufigsten vorkommen.

Daher müssen Sie Ihren Tokenizer nicht unbedingt mit einem sehr großen Korpus trainieren. Das Korpus muss lediglich repräsentativ für Ihre Domäne und groß genug sein, damit der Tokenizer statistisch signifikante Ergebnisse liefern kann.

Doch je nach Größe des Vokabulars und der konkreten Texte im Korpus kann der Tokenizer am Ende auch Wörter speichern, die wir so nicht erwartet haben. Dies zeigt sich zum Beispiel, wenn wir uns die längsten Wörter im Vokabular des GPT-2-Tokenizers ausgeben lassen:

```
tokens = sorted(tokenizer.vocab.items(), key=lambda x: len(x[0]), reverse=True)
print([f'{tokenizer.convert_tokens_to_string(t)}' for t, _ in tokens[:8]]);
```

```
['ÃÃÃ
```

```
from tqdm.auto import tqdm

length = 100000
dataset_name = 'transformersbook/codeparrot-train'
dataset = load_dataset(dataset_name, split="train", streaming=True)
iter_dataset = iter(dataset)

def batch_iterator(batch_size=10):
    for _ in tqdm(range(0, length, batch_size)):
        yield [next(iter_dataset)['content'] for _ in range(batch_size)]

new_tokenizer = tokenizer.train_new_from_iterator(batch_iterator(),
                                                  vocab_size=12500,
                                                  initial_alphabet=base_vocab)
```

Lassen wir uns zunächst die ersten und die letzten Wörter ausgeben, die von unserem BPE-Algorithmus erstellt wurden, damit wir beurteilen können, wie gut sich unser Vokabular eignet. Wir überspringen die 256 Byte-Tokens und sehen uns die ersten Tokens an, die nach den Byte-Tokens hinzugefügt wurden:

```
tokens = sorted(new_tokenizer.vocab.items(), key=lambda x: x[1], reverse=False)
print([f'{tokenizer.convert_tokens_to_string(t)}' for t, _ in tokens[257:280]]);
```

```
['  ', '    ', '   ', '        ', 'se', 'in', '       ', 're', 'on', 'te', '\n
', '\n       ', 'or', 'st', 'de', '\n   ', 'th', 'le', ' =', 'lf', 'self',
'me', 'al']
```

Hier sind verschiedene standardmäßige Einrückungen und Leerzeichen zu sehen, sowie kurze gängige Python-Schlüsselwörter wie `self`, `or` und `in`. Dies deutet darauf hin, dass unser BPE-Algorithmus wie gewünscht funktioniert. Überprüfen wir nun die letzten Wörter:

```
print([f'{new_tokenizer.convert_tokens_to_string(t)}' for t,_ in tokens[-12:]]);
```

```
[' capt', ' embedded', ' regarding', 'Bundle', '355', ' recv', ' dmp', ' vault',
' Mongo', ' possibly', 'implementation', 'Matches']
```

Auch hier finden sich noch einige relativ gebräuchliche Wörter, wie `recv` (*https://oreil.ly/tliPP*), sowie einige verrauschte Wörter, die wahrscheinlich aus den Kommentaren stammen.

Wir können auch unser Python-Codebeispiel tokenisieren, um zu sehen, wie sich unser Tokenizer bei einem einfachen Beispiel schlägt:

```
print(new_tokenizer(python_code).tokens())
```

```
['def', 'Ġs', 'ay', '_', 'hello', '():', 'ĊĠĠĠ', 'Ġprint', '("', 'Hello', ',',
'ĠWor', 'ld', '!")', 'Ġ#', 'ĠPrint', 'Ġit', 'Ċ', 'Ċ', 's', 'ay', '_', 'hello',
'()', 'Ċ']
```

Auch wenn es sich dabei nicht um Schlüsselwörter handelt, ist es doch ein wenig ärgerlich, dass gebräuchliche englische Wörter wie `World` oder `say` von unserem Tokenizer zerlegt werden, da wir davon ausgehen können, dass sie im Korpus recht häufig vorkommen. Überprüfen wir, ob alle in Python reservierten Schlüsselwörter (engl. Keywords) im Vokabular enthalten sind:

```
import keyword

print(f'Es gibt insgesamt {len(keyword.kwlist)} Schlüsselwörter in Python.')
for keyw in keyword.kwlist:
    if keyw not in new_tokenizer.vocab:
        print(f'Nein, das Schlüsselwort `{keyw}` \
                ist nicht im Vokabular enthalten')
```

```
Es gibt insgesamt 35 Schlüsselwörter in Python.
Nein, das Schlüsselwort `await` ist nicht im Vokabular enthalten
Nein, das Schlüsselwort `finally` ist nicht im Vokabular enthalten
Nein, das Schlüsselwort `nonlocal` ist nicht im Vokabular enthalten
```

Offenbar sind auch einige recht häufige Schlüsselwörter, wie z. B. `finally`, nicht im Vokabular enthalten. Versuchen wir also, ein größeres Vokabular zu erstellen, indem wir eine größere Stichprobe aus unserem Datensatz verwenden. Wir können beispielsweise ein Vokabular mit 32.768 Wörtern erstellen (Vielfache von 8 sind besser, damit gewisse GPU- bzw. TPU-Berechnungen effizient vorgenommen werden können) und den Tokenizer auf einer doppelt so großen Stichprobe unseres Korpus trainieren:

```
length = 200000
new_tokenizer_larger = tokenizer.train_new_from_iterator(batch_iterator(),
    vocab_size=32768, initial_alphabet=base_vocab)
```

Vermutlich haben sich die Tokens, die am häufigsten vorkommen, nicht wesentlich geändert, nachdem die zusätzlichen Dokumente einbezogen wurden. Es lohnt sich allerdings, einen Blick auf die zuletzt hinzugefügten Tokens zu werfen:

```
tokens = sorted(new_tokenizer_larger.vocab.items(), key=lambda x: x[1],
                reverse=False)
print([f'{tokenizer.convert_tokens_to_string(t)}' for t, _ in tokens[-12:]]);
```

```
['lineEdit', 'spik', ' BC', 'pective', 'OTA', 'theus', 'FLUSH', ' excutils',
'00000002', ' DIVISION', 'CursorPosition', ' InfoBar']
```

Wie sich zeigt, sind hier keine Python-Schlüsselwörter enthalten. Das ist schon mal ein gutes Zeichen! Lassen wir unser Python-Codebeispiel mit dem neuen umfassenderen Tokenizer tokenisieren:

```
print(new_tokenizer_larger(python_code).tokens())
```

```
['def', 'Ġsay', '_', 'hello', '():', 'ĊĠĠĠ', 'Ġprint', '("', 'Hello', ',',
'ĠWorld', '!")', 'Ġ#', 'ĠPrint', 'Ġit', 'Ċ', 'Ċ', 'say', '_', 'hello', '()',
'Ċ']
```

Hier werden die Einrückungen ebenfalls im Vokabular beibehalten, und wir sehen, dass gebräuchliche englische Wörter wie `Hello`, `World` und `say` auch als einzelne Tokens enthalten sind. Dies scheint eher unseren Erwartungen an die Daten zu entsprechen, die dem Modell in der nachgelagerten Aufgabe begegnen könnten. Mal sehen, ob nun alle Python-Schlüsselwörter im Vokabular enthalten sind:

```
for keyw in keyword.kwlist:
    if keyw not in new_tokenizer_larger.vocab:
        print(f'No, keyword `{keyw}` is not in the vocabulary')
```

```
Nein, das Schlüsselwort `nonlocal` ist nicht im Vokabular enthalten
```

Das Schlüsselwort `nonlocal` (*https://oreil.ly/IHAMu*) ist immer noch nicht enthalten, allerdings wird es in der Praxis auch selten verwendet, da es die Syntax verkompliziert. Es nicht in das Vokabular einzubeziehen, erscheint daher vernünftig. Unserer händischen Inspektion nach, scheint sich die größer angelegte Tokenisierung gut für unsere Aufgabe zu eignen. Doch wie wir bereits erwähnt, ist es schwierig, objektiv zu bewerten, wie gut eine Tokenisierung vonstattengeht, ohne dabei die Leistung des Modells zu kennen. Aus diesem Grund werden wir als Nächstes ein Modell trainieren, um zu sehen, wie gut der Tokenizer in der Praxis funktioniert.

Sie können leicht überprüfen, dass der neue Tokenizer etwa doppelt so effizient wie der standardmäßige GPT-2-Tokenizer ist, indem Sie die Sequenzlängen der tokenisierten Codebeispiele vergleichen. Unser Tokenizer verwendet zur Codierung von Texten etwa halb so viele Tokens wie der bereits zur Verfügung stehende, wodurch wir effektiv eine doppelt so lange Kontextlänge (engl. Context Size) quasi »umsonst« zur Verfügung haben. Wenn wir ein neues Modell mit einer Kontextlänge von 1.024 mit dem neuen Tokenizer trainieren, entspricht dies dem Training des gleichen Modells mit einer Kontextlänge von 2.048 mit dem alten Tokenizer, mit dem Vorteil, dass es viel schneller und hinsichtlich des Speichers effizienter vonstattengeht.

Einen selbst erstellten Tokenizer auf dem Hub speichern

Da unser Tokenizer nun trainiert ist, sollten wir ihn noch speichern. Der einfachste Weg, ihn zu speichern und später von überall darauf zugreifen zu können, ist, ihn auf den Hugging Face Hub zu übertragen. Dies wird insbesondere später nützlich sein, wenn wir einen separaten Trainingsserver verwenden.

Um ein privates Modell-Repository zu erstellen und unseren Tokenizer darin als erste Datei zu speichern, können wir direkt die `push_to_hub()`-Methode des Tokenizers verwenden. Da wir unser Konto bereits mit dem Befehl `huggingface-cli login` authentifiziert haben, können wir den Tokenizer einfach wie folgt auf den Hub pushen:

```
model_ckpt = "codeparrot"
org = "transformersbook"
new_tokenizer_larger.push_to_hub(model_ckpt, organization=org)
```

Wenn Sie nicht möchten, dass Ihr Tokenizer einer Organisation zugeordnet wird, können Sie das Argument `organization` einfach weglassen. Nachdem Sie den Code ausgeführt haben, wird in Ihrem Namensraum ein Repository mit dem Namen `codeparrot` angelegt, das dann von jedem mithilfe des folgenden Befehls geladen werden kann:

```
reloaded_tokenizer = AutoTokenizer.from_pretrained(org + "/" + model_ckpt)
print(reloaded_tokenizer(python_code).tokens())
```

```
['def', 'Ġsay', '_', 'hello', '():', 'ĊĠĠĠ', 'Ġprint', '("', 'Hello', ',',
'ĠWorld', '!")', 'Ġ#', 'ĠPrint', 'Ġit', 'Ċ', 'Ċ', 'say', '_', 'hello', '()',
'Ċ']
```

Unser vom Hub geladener Tokenizer verhält sich genauso wie bevor wir ihn auf den Hub übertragen haben. Darüber hinaus ist es möglich, die zugehörigen Da-

teien und das gespeicherte Vokabular auf dem Hub (*https://oreil.ly/vcLeo*) zu inspizieren. Zu guter Letzt sollten wir auch noch unseren kleineren Tokenizer speichern, um eine Reproduzierbarkeit zu gewährleisten:

```
new_tokenizer.push_to_hub(model_ckpt+ "-small-vocabulary", organization=org)
```

Wir haben uns nun eingehend damit beschäftigt, wie sich ein Tokenizer für einen bestimmten Anwendungsfall erstellen lässt. Dementsprechend sind wir nun so weit, ein neues Modell erstellen und es von Grund auf trainieren zu können.

Ein Modell von Grund auf trainieren

Jetzt kommt der Teil, auf den Sie wahrscheinlich schon gewartet haben: das Training des Modells. In diesem Abschnitt entscheiden wir, welche Architektur für die Aufgabe am besten geeignet ist, und initialisieren ein neues Modell, ohne dabei auf vortrainierte Gewichte zurückzugreifen. Zudem richten wir eine selbst definierte Klasse ein, mit der die Daten geladen werden können, und erstellen eine skalierbare Trainingsschleife. Zum krönenden Abschluss werden wir sowohl ein kleines als auch ein großes GPT-2-Modell mit 111 Millionen bzw. 1,5 Milliarden Parametern trainieren! Aber greifen wir nicht zu weit vor. Zunächst müssen wir entscheiden, welche Architektur sich am besten dazu eignet, Code automatisch zu vervollständigen.

In diesem Abschnitt werden wir ein Skript implementieren, das etwas umfangreicher als üblich ist, um ein Modell auf einer verteilten Infrastruktur zu trainieren. Führen Sie daher nicht jeden Codeabschnitt einzeln aus, sondern laden Sie das Skript herunter, das im 🤗 Repository der Transformers-Bibliothek (*https://oreil.ly/ZyPPR*) bereitgestellt wird. In den zugehörigen Anweisungen erfahren Sie, wie Sie das Skript mithilfe der 🤗 Accelerate-Bibliothek auf Ihrer Hardware ausführen können.

Verschiedene Pretraining-Objectives im Überblick

Da uns nun ein umfangreiches Pretraining-Korpus und ein effizienter Tokenizer zur Verfügung stehen, sollten wir die Überlegung anstellen, wie wir ein Transformer-Modell vortrainieren können. Mit einer solch großen Codebasis, die aus Code-Snippets wie dem in Abbildung 10-1 gezeigten besteht, könnten wir mehrere Aufgaben in Angriff nehmen. Je nachdem, für welche wir uns entscheiden, hat dies Einfluss auf die Wahl der Pretraining-Objectives. Sehen wir uns drei gängige Aufgaben an.

Beispiel aus Korpus

```
def add_numbers(a,b):
    "add two numbers"
    return a+b
```

Abbildung 10-1: Ein Beispiel für eine Python-Funktion, wie sie in unserem Datensatz vorkommen könnte

Causal Language Modeling

Eine typische Aufgabe im Rahmen der Verarbeitung von Textdaten besteht darin, einem Modell den Anfang eines Codebeispiels zu übergeben und es zu beauftragen, mögliche Vervollständigungen zu generieren. Dies ist ein selbstüberwachtes (engl. Self-Supervised) Pretraining-Objective, bei dem wir einen Datensatz verwenden können, ohne zuvor eine Annotation vornehmen zu müssen. Dieser Ansatz sollte Ihnen bereits bekannt vorkommen: Es handelt sich um die Aufgabe des *Causal Language Modeling* (»kausale Sprachmodellierung«), auf die wir in Kapitel 5 eingegangen sind. Eine direkt verwandte nachgelagerte Aufgabe ist die automatische Vervollständigung von Code, weshalb wir dieses Modell definitiv in die engere Wahl nehmen sollten. Eine rein Decoder-basierte Architektur wie die der GPT-Modellfamilie ist in der Regel am besten für diese Aufgabe geeignet (siehe Abbildung 10-2).

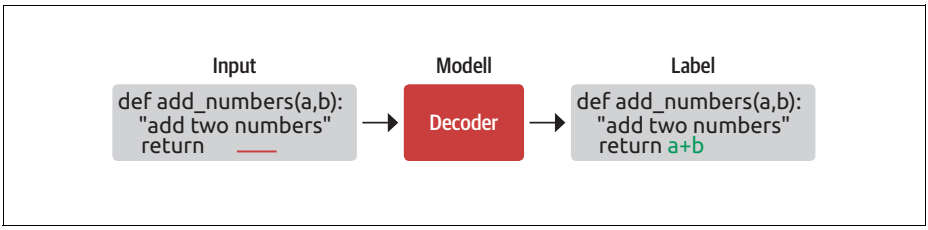


Abbildung 10-2: Im Rahmen des Causal Language Modeling werden die zukünftigen Tokens maskiert, und das Modell hat dann die Aufgabe, diese vorherzusagen. Normalerweise wird für eine solche Aufgabe ein rein Decoder-basiertes Modell wie GPT verwendet.

Masked Language Modeling

Eine verwandte, aber etwas andere Aufgabe besteht darin, einem Modell ein verrauschtes bzw. verfälschtes Codebeispiel vorzulegen, bei dem beispielsweise in einer Codeanweisung ein Wort zufällig ersetzt oder ein Wort maskiert wurde, und es dazu zu bringen, das ursprüngliche Beispiel zu rekonstruieren (siehe Abbildung 10-3). Es handelt sich dabei ebenfalls um ein selbstüberwachtes Pretraining-Objective und wird gemeinhin als *Masked Language Modeling* (»Maskierte Sprachmodellierung«) bzw. als *Denoising Objective* bezeichnet. Allerdings fällt es schwerer, sich eine nachgelagerte Aufgabe vor Augen zu führen, die direkt mit dem Denoising in Zusammenhang gebracht werden kann. Dennoch ist das Denoising im Allgemeinen ein gutes Pretraining-Objective, da es das Modell dazu befähigt, allgemeinere Darstellungen zu lernen, die für spätere nachgelagerte Aufgaben verwendet werden können. Viele der Modelle, auf die wir in den vorangegangenen Kapiteln zurückgegriffen haben (wie BERT und XLM-RoBERTa), wurden auf diese Weise vortrainiert. Dementsprechend ist es möglich, das Training eines maskierten Sprachmodells auf ein großes Korpus mit einem Feintuning des Modells für eine nachgelagerte Aufgabe auf Basis einer begrenzten Anzahl von gelabelten Beispielen zu kombinieren.

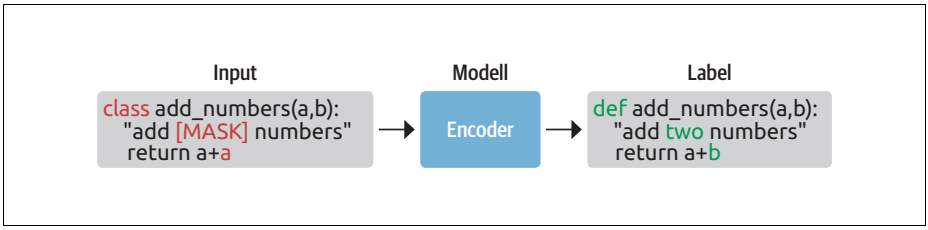


Abbildung 10-3: Im Rahmen des Masked Language Modeling werden einige der eingegebenen Tokens entweder maskiert oder ersetzt, und die Aufgabe des Modells ist es, die ursprünglichen Tokens vorherzusagen. Diese Architektur liegt den rein Encoder-basierten Transformer-Modellen zugrunde.

Sequence-to-Sequence-Training

Eine alternative Aufgabe besteht darin, mithilfe von Heuristiken wie regulären Ausdrücken Kommentare oder Docstrings vom Code zu trennen und einen großen Datensatz, der aus Code/Kommentar-Paaren besteht, zu erstellen, der als gelabelter Datensatz verwendet werden kann. Die Trainingsaufgabe entspricht dann einem überwachten Pretraining-Objective, bei dem eine der beiden Komponenten (Code oder Kommentar) als Eingabe für das Modell dient und die andere Komponente (Kommentar oder Code) als Label verwendet wird. Dies ist ein Fall von *überwachtem Lernen* mit Input/Label-Paaren (siehe Abbildung 10-4). Mit einem großen, bereinigten und vielfältigen Datensatz sowie einem Modell, das groß genug ist, können wir versuchen, ein Modell zu trainieren, das lernt, Kommentare im Code zu transkribieren oder umgekehrt. Eine nachgelagerte Aufgabe, die direkt mit dieser überwachten Trainingsaufgabe zusammenhängt, ist dann die Generierung von Dokumentation aus Code oder die Generierung von Code auf Basis einer Dokumentation, je nachdem, was wir als Ein- und Ausgabe festlegen. Im vorliegenden Fall wird eine Sequenz in eine andere Sequenz übersetzt. Hierbei kommen Encoder-Decoder-basierte Architekturen wie T5, BART und PEGASUS zum Einsatz.

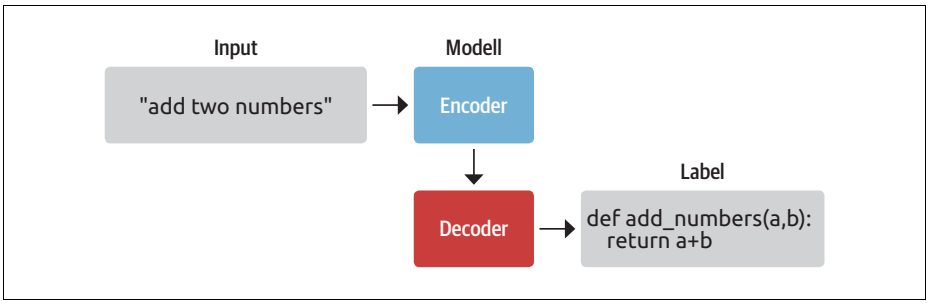


Abbildung 10-4: Eine Encoder-Decoder-basierte Architektur für eine Sequence-to-Sequence-Aufgabe, bei der die Eingaben mithilfe von Heuristiken in Kommentar/Code-Paare aufgeteilt werden: Das Modell erhält ein Element als Eingabe und soll das andere generieren.

Da wir ein Modell zur automatischen Vervollständigung von Code erstellen möchten, wählen wir das erste der genannten Pretraining-Objectives und entscheiden uns, eine GPT-Architektur für diese Aufgabe zu verwenden. Beginnen wir also damit, ein neues GPT-2-Modell zu initialisieren!

Das Modell initialisieren

Dies ist das erste Mal in diesem Buch, dass wir nicht die Methode `from_pretrained()` zum Laden eines Modells verwenden, sondern ein neues Modell initialisieren. Allerdings werden wir die Konfiguration des `gpt2-xl`-Modells laden, sodass wir die gleichen Hyperparameter verwenden und nur die Größe des Vokabulars für den neuen Tokenizer anpassen. Anschließend initialisieren wir mit der Methode `from_config()` ein neues mit dieser Konfiguration versehenes Modell:

```
from transformers import AutoConfig, AutoModelForCausalLM, AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
config = AutoConfig.from_pretrained("gpt2-xl", vocab_size=len(tokenizer))
model = AutoModelForCausalLM.from_config(config)
```

Finden wir zunächst heraus, wie groß das Modell ist:

```
print(f'Größe von GPT-2 (xl): {model_size(model)/1000**2:.1f} Mio. Parameter')
```

```
Größe von GPT-2 (xl): 1529.6 Mio. Parameter
```

Das Modell umfasst 1,5 Milliarden Parameter! Das ist relativ groß – aber schließlich haben wir auch einen großen Datensatz. Im Allgemeinen können große Sprachmodelle effizienter trainiert werden, wenn der Datensatz eine hinreichende Größe hat. Speichern wir nun das neu initialisierte Modell in einem Ordner namens *models/* und übertragen wir es auf den Hub:

```
model.save_pretrained("models/" + model_ckpt, push_to_hub=True,
                      organization=org)
```

Die Übertragung des Modells auf den Hub kann angesichts der Größe des Checkpoints (> 5 GB) einige Minuten in Anspruch nehmen. Da das Modell recht groß ist, erstellen wir auch eine kleinere Version, die wir trainieren können, um sicherzugehen, dass alles funktioniert, bevor wir es skalieren. Dazu nehmen wir die Standardgröße von GPT-2 als Ausgangspunkt:

```
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
config_small = AutoConfig.from_pretrained("gpt2", vocab_size=len(tokenizer))
model_small = AutoModelForCausalLM.from_config(config_small)
```

```
print(f'Größe von GPT-2: {model_size(model_small)/1000**2:.1f} Mio. Parameter')
```

```
Größe von GPT-2: 111.0 Mio. Parameter
```

Wir können das Modell zudem noch auf dem Hub speichern, um es auf einfache Weise mit anderen teilen und wiederverwenden zu können:

```
model_small.save_pretrained("models/" + model_ckpt + "-small", push_to_hub=True,
                            organization=org)
```

Nachdem wir nun zwei Modelle zum Trainieren zur Verfügung haben, müssen wir sicherstellen, dass wir die Eingabedaten im Rahmen des Trainings in effizienter Weise einspeisen können.

Den Dataloader implementieren

Um das Training möglichst effizient zu gestalten, möchten wir unser Modell mit Sequenzen versorgen, die der maximalen Kontextlänge entsprechen. Wenn die Kontextlänge unseres Modells beispielsweise 1.024 Tokens beträgt, dann möchten wir während des Trainings immer Sequenzen bereitstellen, die 1.024-Tokens umfassen. Einige unserer Codebeispiele könnten jedoch kürzer oder länger als 1.024 Zeichen sein. Um unserem Modell Batches mit vollständigen Sequenzen der Länge `sequence_length` zuzuführen, sollten wir also die letzte unvollständige Sequenz weglassen bzw. sie mittels Padding auffüllen. Dadurch würde unser Training jedoch etwas weniger effizient vonstattengehen, und wir wären gezwungen, uns um das Auffüllen und Maskieren der aufgefüllten Token-Labels zu kümmern. Da wir jedoch eher hinsichtlich der verfügbaren Rechenkapazität als der Menge an vorliegenden Daten restringiert sind, wählen wir hier den einfachen und effizienten Weg. Wir können zudem einen kleinen Trick anwenden, um sicherzustellen, dass wir nicht zu viele nachfolgende Abschnitte verlieren: Wir können mehrere Beispiele tokenisieren und sie dann miteinander verketten, um eine sehr lange Sequenz zu erhalten, wobei die einzelnen Beispiele jeweils durch das spezielle End-of-Sequence-Token voneinander getrennt werden. Zum Schluss teilen wir diese Sequenz wiederum in gleich lange Sequenzen auf (siehe Abbildung 10-5), sodass uns bei diesem Ansatz am Ende höchstens ein kleiner Teil der Daten verloren geht.

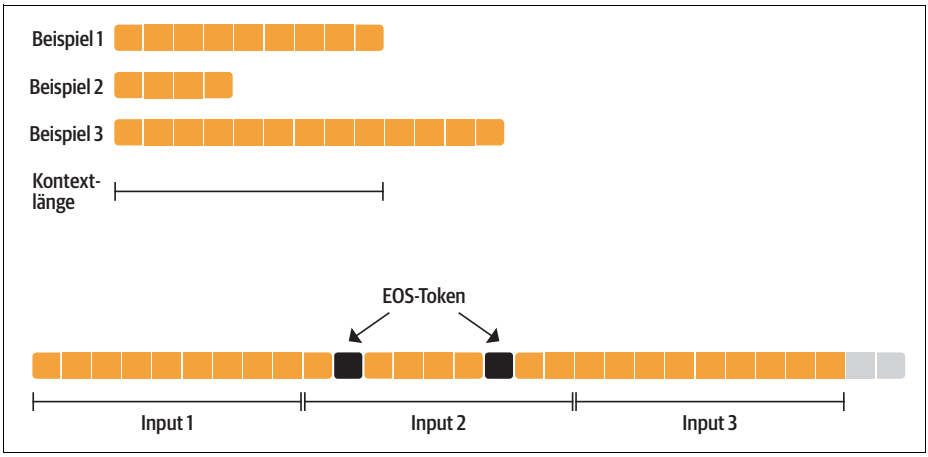


Abbildung 10-5: Vorbereitung von Sequenzen unterschiedlicher Länge für das Causal Language Modeling, indem mehrere tokenisierte Beispiele unter Verwendung eines EOS-Tokens verkettet werden, bevor sie wiederum in einzelne Chunks aufgeteilt werden

Wir können zum Beispiel sicherstellen, dass wir ungefähr hundert vollständige Sequenzen in unseren tokenisierten Beispielen haben, indem wir die Anzahl der Zeichen für die Eingabe-Strings (bzw. deren Länge) (engl. Input String Character Length) wie folgt vorgeben:

```
input_characters = number_of_sequences * sequence_length * characters_per_token
```

wobei:

- `input_characters` steht für die Anzahl der Zeichen des String, der in unseren Tokenizer eingespeist wird.
- `number_of_sequences` entspricht der Anzahl der (gestutzten) Sequenzen, die wir von unserem Tokenizer erhalten wollen (z. B. 100).
- `sequence_length` gibt die Anzahl der Tokens pro Sequenz an, die der Tokenizer zurückgibt (z. B. 1.024).
- `characters_per_token` steht für die durchschnittliche Anzahl von Zeichen je ausgegebenem Token, die wir zunächst schätzen müssen.

Wenn wir einen String mit `input_characters` Zeichen eingeben, erhalten wir also im Durchschnitt `number_of_sequences` Ausgabesequenzen. Dementsprechend können wir leicht berechnen, wie viele Eingabedaten wir verlieren, wenn wir die letzte Sequenz weglassen. Wenn `number_of_sequences=100` gewählt wird, bedeutet das, dass wir ungefähr 100 Sequenzen aneinanderreihen und höchstens die letzte Sequenz verlieren, die zu kurz oder zu lang sein könnte. Insgesamt verlieren wir also höchstens 1 % unseres Datensatzes. Gleichzeitig stellt dieser Ansatz sicher, dass wir keine statistischen Verzerrungen (engl. Bias) verursachen, was der Fall wäre, wenn wir den Großteil der Dateien stutzen bzw. am Ende abschneiden würden.

Nehmen wir nun noch eine Schätzung der durchschnittlichen Anzahl an Zeichen je Token in unserem Datensatz vor:

```
examples, total_characters, total_tokens = 500, 0, 0
dataset = load_dataset('transformersbook/codeparrot-train', split='train',
                       streaming=True)

for _, example in tqdm(zip(range(examples), iter(dataset)), total=examples):
    total_characters += len(example['content'])
    total_tokens += len(tokenizer(example['content']).tokens())

characters_per_token = total_characters / total_tokens

print(characters_per_token)

3.6233025034779565
```

Wir haben nun alle Voraussetzungen, um unsere eigene `IterableDataset`-Klasse (eine von PyTorch bereitgestellte Hilfsklasse) zu erstellen, mit der wir Eingaben konstanter Länge für das Modell aufbereiten können. Wir müssen nur dafür sorgen, dass unsere Klasse von der `IterableDataset`-Klasse erbt und die Methode `__iter__()` einrichten, die – entsprechend der beschriebenen Vorgehensweise – das nächste Element liefert:

```
import torch
from torch.utils.data import IterableDataset

class ConstantLengthDataset(IterableDataset):

    def __init__(self, tokenizer, dataset, seq_length=1024,
                 num_of_sequences=1024, chars_per_token=3.6):
        self.tokenizer = tokenizer
        self.concat_token_id = tokenizer.eos_token_id
        self.dataset = dataset
        self.seq_length = seq_length
        self.input_characters = seq_length * chars_per_token * num_of_sequences

    def __iter__(self):
        iterator = iter(self.dataset)
        more_examples = True
        while more_examples:
            buffer, buffer_len = [], 0
            while True:
                if buffer_len >= self.input_characters:
                    m=f"Buffer full: {buffer_len}>={self.input_characters:.0f}"
                    print(m)
                    break
                try:
                    m=f"Fill buffer: {buffer_len}<{self.input_characters:.0f}"
                    print(m)
                    buffer.append(next(iterator)["content"])
                    buffer_len += len(buffer[-1])
                except StopIteration:
                    iterator = iter(self.dataset)

            all_token_ids = []
            tokenized_inputs = self.tokenizer(buffer, truncation=False)
            for tokenized_input in tokenized_inputs["input_ids'"]:
            for tokenized_input in tokenized_inputs:
                all_token_ids.extend(tokenized_input + [self.concat_token_id])

            for i in range(0, len(all_token_ids), self.seq_length):
                input_ids = all_token_ids[i : i + self.seq_length]
                if len(input_ids) == self.seq_length:
                    yield torch.tensor(input_ids)
```

Die Funktion `__iter__()` baut einen Puffer mit Strings auf, bis er die vorgegebene Anzahl an Zeichen enthält. Alle Elemente im Puffer werden tokenisiert und unter Verwendung des EOS-Tokens verkettet. Anschließend wird die lange (verkettete) Sequenz `all_token_ids` in kleinere Sequenzen (bzw. Chunks) der Länge `seq_length` unterteilt. Normalerweise müssten wir Attention-Masks verwenden, um aufgefüllte Sequenzen unterschiedlicher Länge aneinanderreihen zu können, und auch sicherstellen, dass die entsprechenden Padding-Tokens ignoriert werden, damit das Padding keinen Einfluss auf das Training hat. Hier haben wir allerdings dafür gesorgt, dass alle Sequenzen, die bereitgestellt werden, die gleiche (maximale) Länge aufweisen. Deshalb benötigen wir in diesem Zusammenhang keine Maskierungen und geben lediglich die `input_ids` zurück. Probieren wir nun unser iterierbares Dataset-Objekt aus:

```
shuffled_dataset = dataset.shuffle(buffer_size=100)
constant_length_dataset = ConstantLengthDataset(tokenizer, shuffled_dataset,
                                                num_of_sequences=10)
dataset_iterator = iter(constant_length_dataset)

lengths = [len(b) for _, b in zip(range(5), dataset_iterator)]
print(f"Länge der Sequenzen: {lengths}")
```

```
Fill buffer: 0<36864
Fill buffer: 3311<36864
Fill buffer: 9590<36864
Fill buffer: 22177<36864
Fill buffer: 25530<36864
Fill buffer: 31098<36864
Fill buffer: 32232<36864
Fill buffer: 33867<36864
Buffer full: 41172>=36864
Länge der Sequenzen: [1024, 1024, 1024, 1024, 1024]
```

Sehr gut! Es funktioniert wie vorgesehen. Wir erhalten Eingaben für das Modell, die alle gleich lang sind. Nachdem wir nun dafür Sorge getragen haben, dass die Daten zuverlässig in das Modell eingespeist werden, ist es an der Zeit, die eigentliche Trainingsschleife zu erstellen.

Beachten Sie, dass wir den Rohdatensatz gemischt haben, bevor wir ein `ConstantLengthDataset` erstellt haben. Da es sich um ein iterierbares Dataset-Objekt handelt, können wir nicht einfach den gesamten Datensatz direkt am Anfang mischen. Stattdessen richten wir einen Puffer der Größe `buffer_size` ein und mischen die Elemente in diesem Puffer, bevor wir Elemente aus dem Dataset-Objekt abrufen.

Die Trainingsschleife einrichten

Wir haben nun alle Elemente zusammen, die erforderlich sind, um unsere Trainingsschleife erstellen zu können. Wenn wir ein eigenes Sprachmodell trainieren, stellt vor allem der begrenzte Speicher der von uns verwendeten GPUs eine Einschränkung dar. Ein Modell der Größe von GPT-2 kann selbst auf einer modernen GPU nicht in angemessener Zeit trainiert werden. In diesem Leitfaden werden wir *Datenparallelität* (engl. Data Parallelism) implementieren, die es uns ermöglichen wird, im Rahmen des Trainings mehrere GPUs einzubeziehen. Glücklicherweise können wir die 🤗 Accelerate-Bibliothek verwenden, um unseren Code skalierbar zu machen. Die 🤗 Accelerate-Bibliothek wurde entwickelt, um verteiltes Training – und eine Veränderung der zugrunde liegenden Hardware für das Training – möglichst einfach zu gestalten. Wir könnten auch die `Trainer`-Klasse für verteiltes Training verwenden, aber die 🤗 Accelerate-Bibliothek ermöglicht es uns, die volle Kontrolle über die Trainingsschleife zu gewinnen, was in diesem Zusammenhang von Vorteil ist.

Die 🤗 Accelerate-Bibliothek bietet eine einfache API, mit der Sie Trainingsskripte auf Basis einer gemischten numerischen Genauigkeit (engl. Mixed Precision) und in jeder Art von verteilter Umgebung (einzelne GPU, mehrere GPUs und TPUs)

ausführen können. Derselbe Code kann dann nahtlos auf Ihrem lokalen Rechner zu Debuggingzwecken oder auf Ihrem leistungsstarken Trainingscluster für das eigentliche Training ausgeführt werden. Sie müssen nur eine Handvoll Änderungen an der ursprünglichen PyTorch-Trainingsschleife vornehmen:

```
  import torch
  import torch.nn.functional as F
  from datasets import load_dataset
+ from accelerate import Accelerator

- device = 'cpu'
+ accelerator = Accelerator()

- model = torch.nn.Transformer().to(device)
+ model = torch.nn.Transformer()
  optimizer = torch.optim.Adam(model.parameters())
  dataset = load_dataset('my_dataset')
  data = torch.utils.data.DataLoader(dataset, shuffle=True)
+ model, optimizer, data = accelerator.prepare(model, optimizer, data)

  model.train()
  for epoch in range(10):
      for source, targets in data:
-         source = source.to(device)
-         targets = targets.to(device)
          optimizer.zero_grad()
          output = model(source)
          loss = F.cross_entropy(output, targets)
-         loss.backward()
+         accelerator.backward(loss)
          optimizer.step()
```

Die wichtigste Änderung, die wir vorgenommen haben, ist der zusätzliche Aufruf der `prepare()`-Funktion, die sicherstellt, dass das Modell, die Optimizer und die Dataloader alle bereit und auf der Infrastruktur verteilt sind. Durch diese kleinen Änderungen an der PyTorch-Trainingsschleife können Sie das Training problemlos auf verschiedene Infrastrukturen skalieren. Beginnen wir nun also damit, unser Trainingsskript zu erstellen und ein paar Hilfsfunktionen zu definieren. Zunächst richten wir die Hyperparameter für das Training ein und binden sie für einen einfachen Zugriff in einen `Namespace` ein:

```
from argparse import Namespace

# Die auskommentierten Parameter entsprechen dem kleinen Modell
config = {"train_batch_size": 2, # 12
          "valid_batch_size": 2, # 12
          "weight_decay": 0.1,
          "shuffle_buffer": 1000,
          "learning_rate": 2e-4, # 5e-4
          "lr_scheduler_type": "cosine",
          "num_warmup_steps": 750, # 2000
          "gradient_accumulation_steps": 16, # 1
          "max_train_steps": 50000, # 150000
          "max_eval_steps": -1,
          "seq_length": 1024,
```

```
            "seed": 1,
            "save_checkpoint_steps": 50000} # 15000

    args = Namespace(**config)
```

Als Nächstes richten wir das Logging für das Training ein. Da wir ein Modell von Grund auf trainieren, wird das Training eine Weile dauern und eine Infrastruktur erfordern, die recht kostspielig ist. Daher möchten wir sicherstellen, dass alle relevanten Informationen gespeichert werden und leicht zugänglich sind. Die Methode `setup_logging()` richtet drei Ebenen des Logging ein: einen standardmäßigen Python-`Logger` (*https://oreil.ly/P9Xrm*), TensorBoard (*https://oreil.ly/kY5ri*) und Weights & Biases (*https://oreil.ly/BCC3k*). Je nachdem, welchen Logging-Ansatz Ssie präferieren und wie es Ihr Anwendungsfall erfordert, können Sie Ihr bevorzugtes Logging-Framework im folgenden Code einfach hinzufügen bzw. die vorgesehenen entfernen:

```
from torch.utils.tensorboard import SummaryWriter
import logging
import wandb

def setup_logging(project_name):
    logger = logging.getLogger(__name__)
    logging.basicConfig(
        format="%(asctime)s - %(levelname)s - %(name)s - %(message)s",
        datefmt="%m/%d/%Y %H:%M:%S", level=logging.INFO, handlers=[
        logging.FileHandler(f"log/debug_{accelerator.process_index}.log"),
        logging.StreamHandler()])
    if accelerator.is_main_process: # Wir möchten Logger nur einmal einrichten
        wandb.init(project=project_name, config=args)
        run_name = wandb.run.name
        tb_writer = SummaryWriter()
        tb_writer.add_hparams(vars(args), {'0': 0})
        logger.setLevel(logging.INFO)
        datasets.utils.logging.set_verbosity_debug()
        transformers.utils.logging.set_verbosity_info()
    else:
        tb_writer = None
        run_name = ''
        logger.setLevel(logging.ERROR)
        datasets.utils.logging.set_verbosity_error()
        transformers.utils.logging.set_verbosity_error()
    return logger, tb_writer, run_name
```

Jeder Worker erhält einen eindeutigen `accelerator.process_index`, der vom `File Handler` verwendet wird, um die Logs der einzelnen Worker in eine eigene Datei zu schreiben. Wir verwenden auch das Attribut `accelerator.is_main_process`, das nur für den Main-Worker `true` ist. Zudem stellen wir sicher, dass wir die Logger TensorBoard und Weights & Biases nicht mehrmals initialisieren. Zudem verringern wir das Logging-Level, auf dem die anderen Worker operieren. Die Funktion gibt zusätzlich den automatisch generierten, eindeutigen Namen des Trainingslaufs (`wandb.run.name`) zurück, den wir später verwenden, um den Branch für unsere Experimente auf dem Hub zu benennen.

Wir erstellen auch eine Funktion, mit der wir die Maße, die von TensorBoard und Weights & Biases erfasst werden, loggen können. Auch hier verwenden wir die Funktion `accelerator.is_main_process`, um sicherzustellen, dass wir die Maße nur einmal und nicht für jeden Worker loggen:

```
def log_metrics(step, metrics):
    logger.info(f"Step {step}: {metrics}")
    if accelerator.is_main_process:
        wandb.log(metrics)
        [tb_writer.add_scalar(k, v, step) for k, v in metrics.items()]
```

Als Nächstes benötigen wir eine Funktion, die die Dataloader für den Trainings- und den Validierungsdatensatz mittels unserer neu eingerichteten `ConstantLength Dataset`-Klasse erstellt:

```
from torch.utils.data.dataloader import DataLoader

def create_dataloaders(dataset_name):
    train_data = load_dataset(dataset_name+'-train', split="train",
                              streaming=True)
    train_data = train_data.shuffle(buffer_size=args.shuffle_buffer,
                                    seed=args.seed)
    valid_data = load_dataset(dataset_name+'-valid', split="validation",
                              streaming=True)

    train_dataset = ConstantLengthDataset(tokenizer, train_data,
                                          seq_length=args.seq_length)
    valid_dataset = ConstantLengthDataset(tokenizer, valid_data,
                                          seq_length=args.seq_length)

    train_dataloader=DataLoader(train_dataset, batch_size=args.train_batch_size)
    eval_dataloader=DataLoader(valid_dataset, batch_size=args.valid_batch_size)
    return train_dataloader, eval_dataloader
```

Schließlich übergeben wir den Datensatz an einen `DataLoader`, der auch die Aufteilung in Batches übernimmt. Die 🤗 Accelerate-Bibliothek sorgt dann dafür, dass die Batches auf die einzelnen Worker verteilt werden.

Darüber hinaus müssen wir noch die Optimierung implementieren. Den Optimizer und das Schema für die Anpassung der Lernrate (Learning Rate Schedule) werden wir in der Hauptschleife einrichten. Allerdings definieren wir hier eine Hilfsfunktion, um zu zu differenzieren, welche Parameter dem Weight Decay unterliegen sollen. In der Regel wird für die Bias-Terme und die Gewichte der `LayerNorm`-Schicht kein Weight Decay vorgenommen:

```
def get_grouped_params(model, no_decay=["bias", "LayerNorm.weight"]):
    params_with_wd, params_without_wd = [], []
    for n, p in model.named_parameters():
        if any(nd in n for nd in no_decay):
            params_without_wd.append(p)
        else:
            params_with_wd.append(p)
    return [{'params': params_with_wd, 'weight_decay': args.weight_decay},
            {'params': params_without_wd, 'weight_decay': 0.0}]
```

Darüber hinaus möchten wir natürlich das Modell hin und wieder anhand des Validierungsdatensatzes evaluieren. Fügen wir also eine Evaluierungsfunktion hinzu, die wir aufrufen können, um den Verlust und die Perplexität (engl. Perplexity) auf Basis des Evaluierungsdatensatzes zu berechnen:

```
def evaluate():
    model.eval()
    losses = []
    for step, batch in enumerate(eval_dataloader):
        with torch.no_grad():
            outputs = model(batch, labels=batch)
        loss = outputs.loss.repeat(args.valid_batch_size)
        losses.append(accelerator.gather(loss))
        if args.max_eval_steps > 0 and step >= args.max_eval_steps: break
    loss = torch.mean(torch.cat(losses))
    try:
    perplexity = torch.exp(loss)
    except OverflowError:
    perplexity = torch.tensor(float("inf"))
    return loss.item(), perplexity.item()
```

Die Perplexität ist ein Maß, das angibt, wie gut die Wahrscheinlichkeitsverteilungen der Ausgaben des Modells die vorherzusagenden Tokens tatsächlich vorhersagen. Je geringer die Perplexität, desto besser ist also die Leistung. Beachten Sie, dass wir die Perplexität berechnen können, indem wir den Kreuzentropieverlust, den wir auf Basis der Ausgabe des Modells ermitteln, potenzieren. Insbesondere zu Beginn des Trainings, wenn der Verlust noch hoch ist, kann es bei der Berechnung der Perplexität zu einem numerischen Überlauf kommen. Diesen Fehler fangen wir ab und setzen die Perplexität in solchen Fällen auf unendlich.

Bevor wir alles im Trainingsskript zusammenführen, richten wir noch eine zusätzliche Funktion ein. Wie Sie inzwischen wissen, verwendet der Hugging Face Hub im Hintergrund Git, um Modelle und Datensätze zu speichern und zu versionieren. Mit der `Repository`-Klasse aus der *huggingface_hub*-Bibliothek können Sie programmgesteuert auf das Repository zugreifen und Befehle wie `pull`, `branch`, `commit` oder `push` ausführen. In unserem Skript verwenden wir diese Funktion, um während des Trainings fortlaufend Modell-Checkpoints auf den Hub zu übertragen.

Nachdem wir nun alle Hilfsfunktionen eingerichtet haben, sind wir bereit, das Herzstück des Trainingsskripts zu erstellen:

```
set_seed(args.seed)

# Accelerator
accelerator = Accelerator()
samples_per_step = accelerator.state.num_processes * args.train_batch_size

# Logging
logger, tb_writer, run_name = setup_logging(project_name.split("/")[1])
logger.info(accelerator.state)

# Modell und Tokenizer laden
if accelerator.is_main_process:
    hf_repo = Repository("./", clone_from=project_name, revision=run_name)
```

```
model = AutoModelForCausalLM.from_pretrained("./", gradient_checkpointing=True)
tokenizer = AutoTokenizer.from_pretrained("./")

# Datensatz und Dataloader laden
train_dataloader, eval_dataloader = create_dataloaders(dataset_name)

# Optimizer und Learning-Rate-Scheduler vorbereiten
optimizer = AdamW(get_grouped_params(model), lr=args.learning_rate)
lr_scheduler = get_scheduler(name=args.lr_scheduler_type, optimizer=optimizer,
                             num_warmup_steps=args.num_warmup_steps,
                             num_training_steps=args.max_train_steps,)
def get_lr():
    return optimizer.param_groups[0]['lr']

# Alles mit unserem `accelerator` vorbereiten
#  (die Reihenfolge der Argumente ist nicht wichtig)
model, optimizer, train_dataloader, eval_dataloader = accelerator.prepare(
    model, optimizer, train_dataloader, eval_dataloader)

# Modell trainieren
model.train()
completed_steps = 0
for step, batch in enumerate(train_dataloader, start=1):
    loss = model(batch, labels=batch).loss
    log_metrics(step, {'lr': get_lr(), 'samples': step*samples_per_step,
                       'steps': completed_steps, 'loss/train': loss.item()})
    loss = loss / args.gradient_accumulation_steps
    accelerator.backward(loss)
    if step % args.gradient_accumulation_steps == 0:
        optimizer.step()
        lr_scheduler.step()
        optimizer.zero_grad()
        completed_steps += 1
    if step % args.save_checkpoint_steps == 0:
        logger.info('Evaluating and saving model checkpoint')
        eval_loss, perplexity = evaluate()
        log_metrics(step, {'loss/eval': eval_loss, 'perplexity': perplexity})
        accelerator.wait_for_everyone()
        unwrapped_model = accelerator.unwrap_model(model)
        if accelerator.is_main_process:
            unwrapped_model.save_pretrained("./")
            hf_repo.push_to_hub(commit_message=f'step {step}')
        model.train()
    if completed_steps >= args.max_train_steps:
        break

# Evaluieren und letzten Checkpoint speichern
logger.info('Evaluating and saving model after training')
eval_loss, perplexity = evaluate()
log_metrics(step, {'loss/eval': eval_loss, 'perplexity': perplexity})
accelerator.wait_for_everyone()
unwrapped_model = accelerator.unwrap_model(model)
if accelerator.is_main_process:
    unwrapped_model.save_pretrained("./")
    hf_repo.push_to_hub(commit_message=f'final model')
```

Dieser Codeabschnitt mag zwar recht umfangreich sein, doch führen Sie sich vor Augen, dass dies der komplette Code ist, den Sie benötigen, um ein großes ausgefallenes Sprachmodell auf einer verteilten Infrastruktur zu trainieren. Gehen wir das Skript durch und heben wir die wichtigsten Teile hervor:

Speichern des Modells

Wir führen das Skript im Modell-Repository aus und checken zu Beginn einen neuen Branch aus, der nach dem Namen des Trainingslaufs (`run_name`) benannt ist, den wir von Weights & Biases erhalten. Nachdem anschließend jeweils ein Checkpoint erstellt wurde, committen wir das Modell und pushen es auf den Hub. Dementsprechend befindet sich jedes Experiment in einem neuen Branch, und jeder Commit stellt einen Modell-Checkpoint dar. Beachten Sie, dass wir die Funktionen `wait_for_everyone()` und `unwrap_model()` aufrufen müssen, um sicherzustellen, dass das Modell richtig synchronisiert wird, wenn wir es speichern.

Optimierung

Zur Optimierung des Modells verwenden wir den `AdamW`-Algorithmus mit einem Cosine-Learning-Rate-Schedule (d.h. eine auf die Cosinusfunktion beruhende Anpassung der Lernrate) infolge einer Warmup-Phase, in der die Lernrate linear erhöht wird. Bei den Hyperparametern lehnen wir uns eng an die Parameter an, die im GPT-3-Papier für Modelle ähnlicher Größe beschrieben werden.[8]

Evaluierung

Das Modell wird jedes Mal, wenn wir es speichern, d.h. in allen `save_checkpoint_steps` und im Anschluss an das Training, auf Basis des Validierungsdatensatzes evaluiert. Zusätzlich zum Validierungsverlust erfassen bzw. loggen wir auch die Validierungsperplexität.

Gradient-Accumulation und -Checkpointing

Die Batches passen aufgrund ihrer erforderlichen Größe nicht in den Speicher einer GPU, selbst dann nicht, wenn wir auf neueste GPUs zurückgreifen können. Daher implementieren wir eine Gradient-Accumulation, bei der die Gradienten über mehrere Backward-Passes erfasst und erst optimiert werden, sobald genügend Gradienten akkumuliert wurden. In Kapitel 6 haben wir bereits beschrieben, wie Sie dies mit der `Trainer`-Klasse umsetzen können. Bei der großen Version des Modells passt selbst ein einzelnes Batch nicht ganz auf eine GPU. Mithilfe einer Methode namens *Gradient-Checkpointing* können wir den Speicherbedarf verringern, was allerdings dazu führt, dass wir eine Verlangsamung des Trainings um ca. 20 % in Kauf nehmen müssen.[9] Auf diese Weise können wir selbst ein großes Modell auf einer einzelnen GPU unterbringen.

8 T. Brown et al., »Language Models Are Few-Shot Learners« (*https://arxiv.org/abs/2005.14165*), (2020).

9 Mehr über das Gradient-Checkpointing erfahren Sie in OpenAIs Release-Post (*https://oreil.ly/94oj1*).

Ein vielleicht noch etwas unklarer Aspekt ist, was es bedeutet, ein Modell auf mehreren GPUs zu trainieren. Je nach Größe Ihres Modells und wie umfangreich die Daten sind, gibt es verschiedene Ansätze, wie sich Modelle auf verteilte Weise trainieren lassen. Der Ansatz, der von der 🤗 Accelerate-Bibliothek verfolgt wird, heißt `DataDistributedParallelism` (DDP) (*https://oreil.ly/m4iNm*) (siehe Abbildung 10-6). Der Hauptvorteil dieses Ansatzes besteht darin, dass Sie Modelle mit größeren Batchgrößen, die nicht in eine einzelne GPU passen würden, schneller trainieren können.

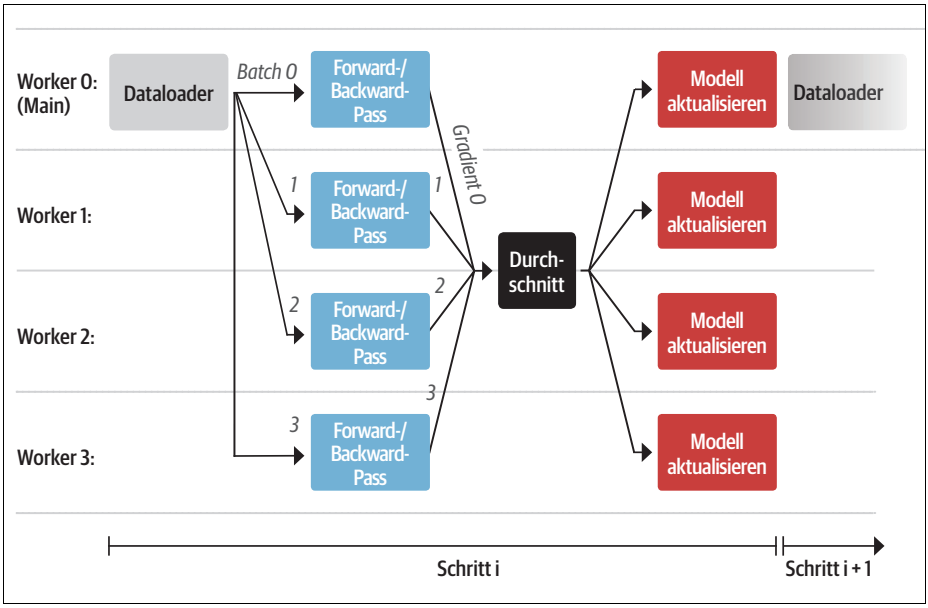


Abbildung 10-6: Schematische Darstellung der Verarbeitungsschritte beim DDP-Ansatz bei Verwendung von vier GPUs

Gehen wir die Pipeline Schritt für Schritt durch:

1. Jeder Worker besteht aus einer GPU. In der 🤗 Accelerate-Bibliothek läuft ein Dataloader über den Hauptprozess, der die einzelnen Batches von Daten vorbereitet und sie an alle Worker übermittelt.
2. Jede GPU empfängt ein Batch von Daten und berechnet den Verlust und die entsprechenden akkumulierten Gradienten aus den Forward- und Backward-Passes, die mit einer lokalen Kopie des Modells durchgeführt werden.
3. Für die Gradienten jedes Knotens (engl. Node) wird mittels eines *reduce*-Musters ein Durchschnitt gebildet, und die durchschnittlichen Gradienten werden an jeden Worker zurückgeschickt.

4. Die Gradienten werden mithilfe des Optimizer bzw. Optimierungsalgorithmus auf jeden einzelnen Knoten angewandt. Auch wenn dies vielleicht überflüssig erscheinen mag, wird dadurch vermieden, dass zwischen den Knoten Kopien der großen Modelle übertragen werden müssen. Wir müssen das Modell mindestens einmal aktualisieren, und ohne diesen Ansatz müssten die anderen Knoten jeweils warten, bis sie die aktualisierte Version erhalten haben.
5. Sobald alle Modelle aktualisiert wurden, fangen wir wieder von vorne an, indem der Main-Worker neue Batches vorbereitet.

Dieses einfache Schema ermöglicht es uns, große Modelle extrem schnell zu trainieren, da es ohne nennenswerte Anpassungen auf die Anzahl an verfügbaren GPUs hochskaliert. In manchen Fällen reicht dies jedoch nicht aus. Wenn das Modell z. B. nicht auf eine einzelne GPU passt, benötigen Sie möglicherweise ausgefeiltere Strategien zur Parralelisierung (*https://oreil.ly/3uhfq*). Nachdem wir nun alle Komponenten eingerichtet haben, die wir zum Trainieren benötigen, ist es an der Zeit, einen Job zu starten. Wie Sie im nächsten Abschnitt sehen werden, ist das ziemlich einfach zu bewerkstelligen.

Der Trainingslauf

Wir speichern das Trainingsskript in einer Datei namens *codeparrot_training.py*, damit wir es auf unserem Trainingsserver ausführen können. Um uns die Arbeit noch weiter zu erleichtern, fügen wir das Skript zusammen mit einer Datei *requirements.txt*, in der alle erforderlichen Python-Abhängigkeiten aufgeführt sind, dem Modell-Repository auf dem Hub (*https://oreil.ly/ndqSB*) hinzu. Denken Sie daran, dass die Modelle auf dem Hub im Grunde genommen Git-Repositories sind. Wir können also einfach das Repository klonen, alle gewünschten Dateien hinzufügen und sie dann zurück zum Hub pushen. Anschließend können wir das Training auf dem Trainingsserver mit den folgenden Befehlen starten:

```
$ git clone https://huggingface.co/transformersbook/codeparrot
$ cd codeparrot
$ pip install -r requirements.txt
$ wandb login
$ accelerate config
$ accelerate launch codeparrot_training.py
```

Das war's bereits – unser Modell wird jetzt trainiert! Beachten Sie, dass Sie infolge des Befehls `wandb login` aufgefordert werden, sich bei Weights & Biases zu authentifizieren, damit das Logging vorgenommen wird. Nach Eingabe des Befehls `accelerate config` werden Sie durch die Einrichtung der Infrastruktur geführt. Das für dieses Experiment verwendete Setting können Sie in Tabelle 10-2 nachvollziehen. Wir verwenden für alle Experimente eine `a2-megagpu-16g`-Instanz (*https://oreil.ly/ZJIG3*), d. h. eine Workstation mit 16 A100-GPUs, die jeweils über 40 GB Speicher verfügen.

Tabelle 10-2: Das zum Trainieren der CodeParrot-Modelle verwendete Setting

Setting	verwendet
Rechenumgebung?	mehrere GPUs
Wie viele Maschinen/Rechner?	1
DeepSpeed?	Nein
Wie viele Prozesse?	16
Verwendung von FP16?	Ja

Wenn Sie das Trainingsskript mit diesem Setting auf einer solchen Infrastruktur ausführen, dauert es etwa 24 Stunden bei der kleinen und 7 Tage bei der großen Version des Modells, bis das Training abgeschlossen ist. Wenn Sie Ihr eigenes benutzerdefiniertes Modell trainieren, sollten Sie zunächst sicherstellen, dass Ihr Code auf einer kleineren Infrastruktur reibungslos läuft, um zu gewährleisten, dass auch der kostspielige lange Trainingsvorgang reibungslos abläuft. Nachdem der vollständige Trainingslauf (engl. Training Run) erfolgreich abgeschlossen wurde, können Sie den für das Experiment genutzten Branch auf dem Hub mit den folgenden Befehlen wieder mit dem Main Branch zusammenführen bzw. mergen:

```
$ git checkout main
$ git merge <RUN_NAME>
$ git push
```

Anstelle von *RUN_NAME* sollten Sie natürlich den Namen des für das Experiment genutzten Branch auf dem Hub angeben, den Sie mergen möchten. Da unser Modell nun fertig trainiert ist, sollten wir uns noch ein Bild davon machen, wie wir die Leistung bzw. Qualität des Modells beurteilen können.

Ergebnisse und Analyse

Nachdem Sie die Log-Daten eine Woche lang unentwegt verfolgt haben, werden Sie wahrscheinlich Verlust- und Perplexitätskurven vorfinden, die wie die in Abbildung 10-7 gezeigten aussehen. Der Trainingsverlust und die Validierungsperplexität verringern sich kontinuierlich, und die Verlustkurve hat auf Basis einer doppelt-logarithmischen Skalierung (engl. Log-Log Scale), d.h., sowohl die Werte auf der Abszisse als auch auf der Ordinate sind logarithmisch skaliert, einen fast linearen Verlauf. Wir können auch erkennen, dass das große Modell in Bezug auf die Anzahl der verarbeiteten Tokens schneller konvergiert, obwohl das Training insgesamt mehr Zeit erfordert.

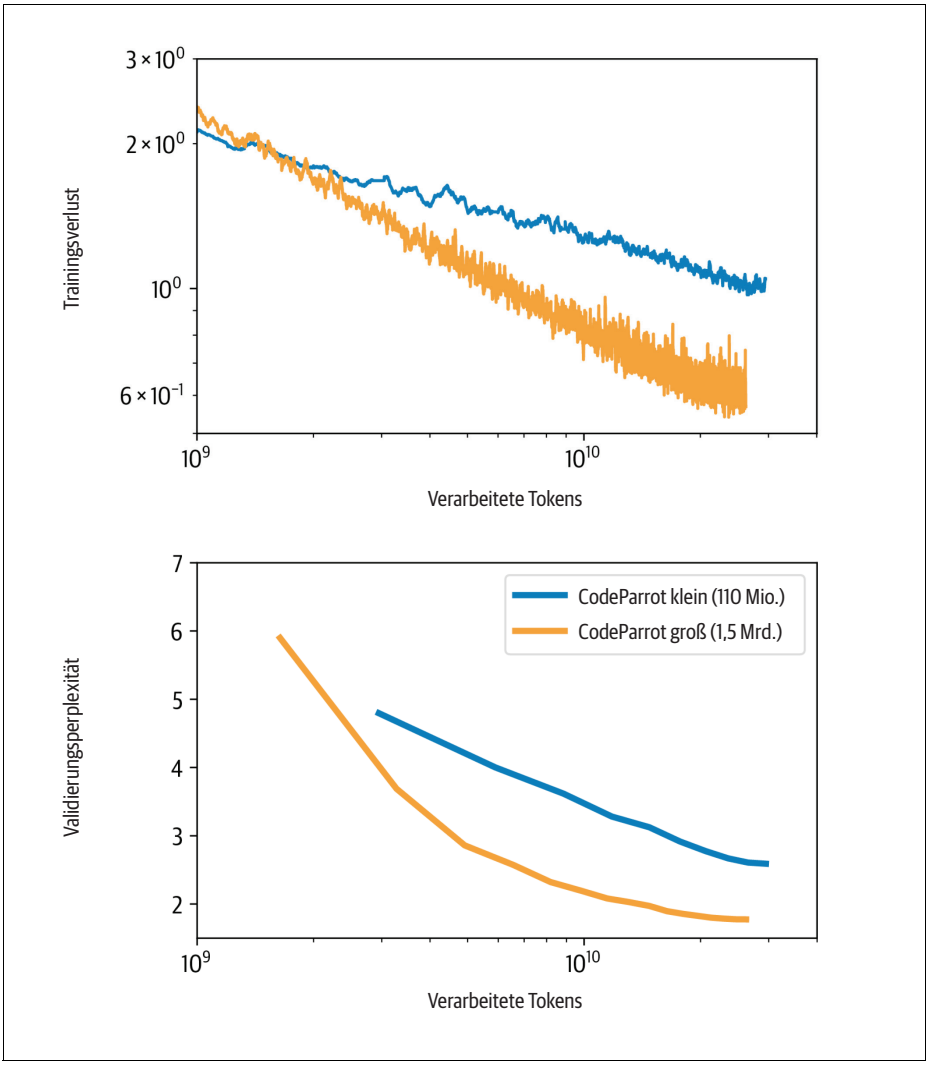


Abbildung 10-7: Trainingsverlust und Validierungsperplexität für das kleine und das große CodeParrot-Modell in Abhängigkeit der Anzahl verarbeiteter Tokens

Was können wir nun mit unserem brandneuen Sprachmodell anstellen? Nun, wir können es dafür einsetzen, ein wenig Code für uns zu erstellen. Die Analyse des Modells können wir sowohl in qualitativer als auch in quantitativer Hinsicht durchführen. Im ersten Fall untersuchen wir konkrete Beispiele und versuchen, besser zu verstehen, in welchen Fällen sich das Modell bewährt und in welchen es versagt. Im zweiten Fall bewerten wir die Leistung des Modells statistisch anhand einer großen Anzahl von Testbeispielen. Zunächst werfen wir einen Blick auf einige Beispiele und erörtern dann kurz, wie wir das Modell systematisch und auch belastbarer evaluieren können. Packen wir als Erstes das kleine Modell in eine Pipe-

line zur Generierung und lassen wir es einige von uns vorgegebene Codeeingaben fortsetzen:

```
from transformers import pipeline, set_seed

model_ckpt = 'transformersbook/codeparrot-small'
generation = pipeline('text-generation', model=model_ckpt, device=0)
```

Jetzt können wir die Pipeline verwenden, und für eine gegebene Eingabe (Prompt) mehrere Vervollständigungen (bzw. Vervollständigungskandidaten) generieren. Die Pipeline generiert standardmäßig so viel Code, bis die vordefinierte Maximallänge erreicht ist, und die Ausgabe könnte mehrere Funktionen oder Klassen enthalten. Um die Ausgaben übersichtlich zu halten, implementieren wir eine Funktion, `first_block()`, die reguläre Ausdrücke verwendet, um das erste Vorkommen einer Funktion oder Klasse zu extrahieren. Die nachfolgende Funktion, `complete_code()`, greift auf diese Funktion zurück und gibt die von CodeParrot generierten Vervollständigungen aus:

```
import re
from transformers import set_seed

def first_block(string):
    return re.split('\nclass|\ndef|\n#|\n@|\nprint|\nif', string)[0].rstrip()

def complete_code(pipe, prompt, max_length=64, num_completions=4, seed=1):
    set_seed(seed)
    gen_kwargs = {"temperature":0.4, "top_p":0.95, "top_k":0, "num_beams":1,
                  "do_sample":True,}
    code_gens = generation(prompt, num_return_sequences=num_completions,
                           max_length=max_length, **gen_kwargs)
    code_strings = []
    for code_gen in code_gens:
        generated_code = first_block(code_gen['generated_text'][len(prompt):])
        code_strings.append(generated_code)
    print(('\n'+'='*80 + '\n').join(code_strings))
```

Beginnen wir mit einem einfachen Beispiel und lassen wir das Modell eine Funktion für uns erstellen, mit der wir die Fläche eines Rechtecks berechnen können:

```
prompt = '''def area_of_rectangle(a: float, b: float):
    """Return the area of the rectangle."""'''
complete_code(generation, prompt)

    return math.sqrt(a * b)
================================================================================

    return a * b / 2.0
================================================================================

    return a * b
================================================================================

    return a * b / a
```

Die Vorschläge machen einen guten Eindruck. Zwar sind nicht alle Generierungen korrekt, doch die richtige Lösung ist bereits enthalten. Kann das Modell auch eine

komplexere Aufgabe lösen, nämlich URLs aus einem HTML-String zu extrahieren? Probieren wir es aus:

```
prompt = '''def get_urls_from_html(html):
    """Get all embedded URLs in a HTML string."""'''
complete_code(generation, prompt)
```

```
    if not html:
        return []
    return [url for url in re.findall(r'<a href="(/[^/]+/[^"]+?)">', html)]
================================================================================

    return [url for url in re.findall(r'<a href="(.*?)"', html)
            if url]
================================================================================

    return [url for url in re.findall(r'<a href="(/.*)",', html)]
================================================================================

    return re.findall(r'<a href="(.*?)" class="url"[^>]*>', html)
```

Auch hier hat es nicht bei allen Vorschlägen geklappt, aber die zweite Lösung scheint richtig zu sein. Die Funktion können wir gleich mal mit der Homepage von Hugging Face testen:

```
import requests

def get_urls_from_html(html):
    return [url for url in re.findall(r'<a href="(.*?)"', html) if url]

print(" | ".join(get_urls_from_html(requests.get('https://hf.co/').text)))
```

```
https://github.com/huggingface/transformers | /allenai | /facebook |
/asteroid-team | /google | /amazon | /speechbrain | /microsoft | /grammarly |
/models | /inference-api | /distilbert-base-uncased |
/dbmdz/bert-large-cased-finetuned-conll03-english |
https://huggingface.co/transformers | https://arxiv.org/abs/1811.06031 |
https://arxiv.org/abs/1803.10631 | https://transformer.huggingface.co/ | /coref
| https://medium.com/huggingface/distilbert-8cf3380435b5
```

Wie wir sehen, entsprechen alle URLs, die mit `https` beginnen, externen Seiten, während die anderen lediglich Unterseiten der Hauptwebseite sind. – Genau das wollten wir erreichen. Lassen Sie uns zum Schluss das große Modell laden und damit testen, ob wir eine Funktion, die in reinem Python geschrieben wurde, in NumPy überführen können:

```
model_ckpt = 'transformersbook/codeparrot'
generation = pipeline('text-generation', model=model_ckpt, device=0)

prompt = '''# a function in native python:
def mean(a):
    return sum(a)/len(a)

# the same function using numpy:
import numpy as np
def mean(a):'''
complete_code(generation, prompt, max_length=64)
```

```
Setting `pad_token_id` to `eos_token_id`:0 for open-end generation.


    return np.mean(a)
================================================================================

    return np.mean(a)
================================================================================

    return np.mean(a)
================================================================================

    return np.mean(a)
```

Es hat geklappt! Mal sehen, ob wir das CodeParrot-Modell auch für die Erstellung eines Scikit-learn-Modells verwenden können:

```
prompt = '''X = np.random.randn(100, 100)
y = np.random.randint(0, 1, 100)

# fit random forest classifier with 20 estimators'''
complete_code(generation, prompt, max_length=96)
```

```
Setting `pad_token_id` to `eos_token_id`:0 for open-end generation.


reg = DummyRegressor()

forest = RandomForestClassifier(n_estimators=20)

forest.fit(X, y)
================================================================================

clf = ExtraTreesClassifier(n_estimators=100, max_features='sqrt')
clf.fit(X, y)
================================================================================

clf = RandomForestClassifier(n_estimators=20, n_jobs=n_jobs, random_state=1)
clf.fit(X, y)
================================================================================

clf = RandomForestClassifier(n_estimators=20)
clf.fit(X, y)
```

Obwohl im zweiten Ansatz versucht wurde, einen Extra-Trees-Classifier (*https://oreil.ly/40Uy7*) zu trainieren, hat er in den anderen Fällen das erzeugt, was wir uns erhofft hatten.

In Kapitel 5 haben wir einige Maße erkundet, anhand derer wir die Qualität generierter Texte beurteilen können. Dazu gehörte unter anderem das BLEU-Maß, das häufig für diesen Zweck verwendet wird. Obgleich dieses Maß bereits im Allgemeinen seine Grenzen hat, ist es für unseren Anwendungsfall besonders schlecht geeignet. Das BLEU-Maß gibt an, wie viele *n*-Gramme sich zwischen den Referenztexten und den generierten Texten decken. Wenn wir Code schreiben, haben wir in Bezug auf Variablen und Klassen sehr viele Freiheiten, und der Erfolg eines Programms hängt nicht davon ab, welche Namen vergeben werden, solange sie konsistent verwendet

werden. Das BLEU-Maß würde jedoch eine Generierung bestrafen, die von der in den der Referenztexten verwendeten Namensgebung abweicht, was in der Tat fast unmöglich vorherzusagen ist (selbst für einen menschlichen Programmierer).

Im Bereich der Softwareentwicklung gibt es viel bessere und zuverlässigere Methoden, mit der sich die Qualität von Code beurteilen lässt, wie z.B. Unit-Tests. Genau auf diese Weise wurden alle Codex-Modelle von OpenAI evaluiert: indem mehrere Codegenerierungen für Programmieraufgaben durch eine Reihe von Unit-Tests laufen und der Anteil der Generierungen ermittelt wird, die die Tests bestehen.[10] Um die Leistung unserer Modelle wirklich in angemessener Weise bewerten zu können, sollten wir den gleichen Ansatz zur Evaluierung auf unsere Modelle anwenden, was allerdings den Rahmen dieses Kapitels sprengen würde. Sie finden jedoch im begleitenden Blogpost zum CodeParrot-Modell (*https://oreil.ly/hKOP8*) ausführliche Informationen darüber, wie es bei der HumanEval-Benchmark abgeschnitten hat.

Zusammenfassung

Nehmen Sie sich einen Moment Zeit und denken Sie darüber nach, was wir in diesem Kapitel erreicht haben. Wir haben uns vorgenommen, eine Funktion zur automatischen Vervollständigung von Code für Python zu entwickeln. Zunächst haben wir einen eigenen, großen Datensatz erstellt, der sich für das Pretraining eines großen Sprachmodells eignet. Dann haben wir einen selbst definierten Tokenizer erstellt, der in der Lage ist, Python-Code auf Basis dieses Datensatzes effizient zu codieren. Schließlich haben wir mithilfe der 🤗 Accelerate-Bibliothek alles zusammengeführt und ein Trainingsskript erstellt, mit dem wir die kleine und die große Version eines GPT-2-Modells von Grund auf auf einer Infrastruktur mit mehreren GPUs trainieren konnten – und das mit weniger als zweihundert Codezeilen. Die Analyse der Ausgaben des Modells ergab, dass es in der Lage ist, brauchbare Codefortsetzungen zu generieren, und wir haben erörtert, wie das Modell systematisch evaluiert werden kann.

Sie wissen jetzt nicht nur, wie Sie eines der vielen vortrainierten Modelle, die Ihnen auf dem Hub zur Verfügung stehen, feintunen können, sondern auch, wie Sie ein selbst definiertes Modell von Grund auf vortrainieren können, vorausgesetzt, Ihnen stehen genügend Daten und Rechenressourcen zur Verfügung. Sie sind nun darauf vorbereitet, fast jeden NLP-Anwendungsfall mithilfe von Transformer-Modellen zu bewältigen. Daher bleibt natürlich noch die Frage: Wohin als Nächstes? Im nächsten und letzten Kapitel werfen wir einen Blick darauf, wohin sich das Forschungsgebiet derzeit bewegt und welche neuen spannenden Anwendungen und Bereiche jenseits von Transformer-Modellen für NLP in Angriff genommen werden können.

10 M. Chen et al., »Evaluating Large Language Models Trained on Code« (*https://arxiv.org/abs/2107.03374*), (2021).

KAPITEL 11

Künftige Herausforderungen

Im Verlauf dieses Buchs haben wir die eindrucksvollen Fähigkeiten von Transformern im Rahmen einer Vielzahl von NLP-Aufgaben erkundet. In diesem abschließenden Kapitel wechseln wir nun die Perspektive und befassen uns mit einigen der aktuellen Herausforderungen im Zusammenhang mit diesen Modellen und den Entwicklungen in der Forschung, die darauf abzielen, diese zu überwinden. Im ersten Teil beschäftigen wir uns mit der Skalierung von Transformer-Modellen, sowohl in Bezug auf das Modell als auch auf die Größe des verwendeten Korpus. Anschließend richten wir unsere Aufmerksamkeit auf verschiedene Techniken, die dazu beitragen sollen, den Self-Attention-Mechanismus effizienter zu gestalten. Schließlich sehen wir uns das aufstrebende und spannende Feld der *multimodalen Transformer* (engl. Multimodal Transformers) an, die Eingaben aus verschiedenen Quellformaten wie Text-, Bild- und Tondaten (gleichzeitig) verarbeiten können.

Skalierung von Transformer-Modellen

Im Jahr 2019 verfasste der Forscher Richard Sutton (*https://oreil.ly/119br*) einen provokanten Aufsatz mit dem Titel »The Bitter Lesson« (*https://oreil.ly/YtD3V*), in dem er die folgende Ansicht vertrat (aus dem Englischen übersetzt):

> Die wichtigste Lehre, die man aus 70 Jahren KI-Forschung ziehen kann, ist, dass allgemeine Methoden, die Rechenressourcen ausschöpfen, letztendlich am effektivsten sind, und zwar mit großem Vorsprung ... Auf der Suche nach einer Verbesserung, die kurzfristig einen Unterschied macht, versuchen die Forscher, ihr Domänenwissen zu nutzen, aber das Einzige, was auf lange Sicht zählt, ist die Ausschöpfung von Rechenressourcen. Diese beiden Ziele müssen nicht im Widerspruch zueinander stehen, tun es aber in der Praxis häufig ... Und der Ansatz, das menschliche Domänenwissen einzubeziehen, neigt dazu, die Methoden so zu verkomplizieren, dass sie weniger geeignet sind, um die Vorteile allgemeiner Methoden zur Ausschöpfung von Rechenressourcen voll auszunutzen.

Im Aufsatz werden mehrere historische Beispiele angeführt, wie z.B. Schach oder das Go-Spiel, bei denen der Ansatz, menschliches Wissen in KI-Systemen zu codieren, letztendlich von der zunehmenden Rechenleistung überholt wurde. Sutton nennt dies die »bittere Lektion« für die KI-Forschung (aus dem Englischen übersetzt):

> Wir müssen die bittere Lektion hinnehmen, dass es auf lange Sicht nicht funktioniert, darauf zu bauen, wie wir denken, dass wir denken ... Eine Sache, die wir aus ebendieser bitteren Lektion lernen sollten, ist die große Macht von Allzweckmethoden, also Methoden, die mit zunehmender Rechenleistung immer weiter skalierbar sind, selbst wenn die verfügbare Rechenleistung sehr groß wird. Die beiden Methoden, die auf diese Weise beliebig skalierbar zu sein scheinen, sind die *Suche* und das *Lernen*.

Es gibt inzwischen Anzeichen dafür, dass bei den Transformern eine ähnliche Lektion zum Tragen kommt. Während sich viele der frühen Weiterentwicklungen von BERT und GPT auf die Optimierung der Architektur oder auf die Anpassung von Training-Objectives konzentrierten, sind die leistungsstärksten Modelle bis Mitte des Jahrs 2021, darunter auch GPT-3, im Wesentlichen einfache, skalierte Versionen der ursprünglichen Modelle, ohne dass dabei große Änderungen an der Architektur vorgenommen wurden. In Abbildung 11-1 ist die chronologische Entwicklung der größten Modelle seit der Veröffentlichung der ursprünglichen Transformer-Architektur im Jahr 2017 dargestellt, die aufzeigt, dass die Modellgröße in nur wenigen Jahren um mehr als den Faktor Vier zugenommen hat.

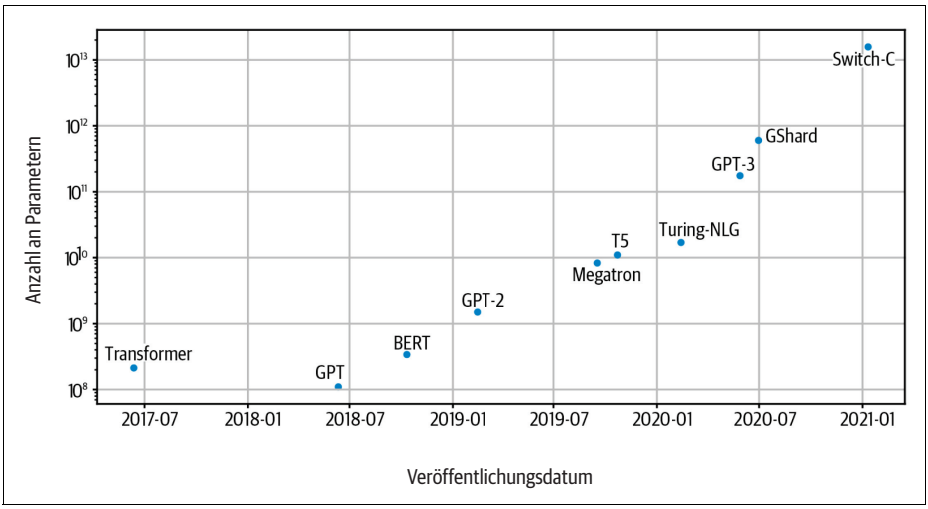


Abbildung 11-1: Bekannte Transformer-Architekturen und ihre Anzahl an Parametern im Laufe der Zeit

Diese dramatische Skalierung ist auf die empirische Evidenz zurückzuführen, wonach große Sprachmodelle bei nachgelagerten Aufgaben besser abschneiden und im Bereich von 10 bis 100 Milliarden Parametern interessante Fähigkeiten wie das Zero-Shot- und Little-Shot-Learning erlangen. Die Anzahl der Parameter ist jedoch nicht der einzige Faktor, der sich auf die Leistung eines Modells auswirkt. Gleichermaßen müssen auch die Rechenkapazitäten und die Menge an Trainingsdaten erhöht werden, um diese Ungetüme trainieren zu können. Angesichts der Tatsache, dass das Training großer Sprachmodelle hohe Kosten verursacht – wie im Fall des GPT-3-Modells, dessen Trainingskosten auf 4,6 Millionen $ (*https://oreil.ly/DUVcq*) geschätzt werden –, ist es natürlich wünschenswert, die Leistung eines

Modells im Voraus einschätzen zu können. Überraschenderweise scheint die Leistung von Sprachmodellen einer *Potenzbeziehung* hinsichtlich der Skalierung der Modellgröße und anderen Faktoren zu folgen und somit gewissen Gesetzmäßigkeiten im Hinblick auf die Skalierung zu unterliegen.[1] Sehen wir uns diesen spannenden Bereich der Forschung nun einmal genauer an.

Skalierungsgesetze

Skalierungs- bzw. Skalengesetze ermöglichen es, das Paradigma »größer ist besser« empirisch für Sprachmodelle zu quantifizieren, indem untersucht wird, wie sie sich verhalten, wenn das Rechenbudget C, die Größe des Datensatzes D und die Größe des Modells N variiert werden.[2] Die Grundidee besteht darin, den Kreuzentropieverlust L in Abhängigkeit von diesen drei Faktoren darzustellen und zu ermitteln, ob eine Beziehung besteht. In Abbildung 11-2 werden die resultierenden Verlustkurven für autoregressive Modelle wie die der GPT-Familie dargestellt, wobei jede Kurve dem Trainingslauf eines einzelnen Modells entspricht.

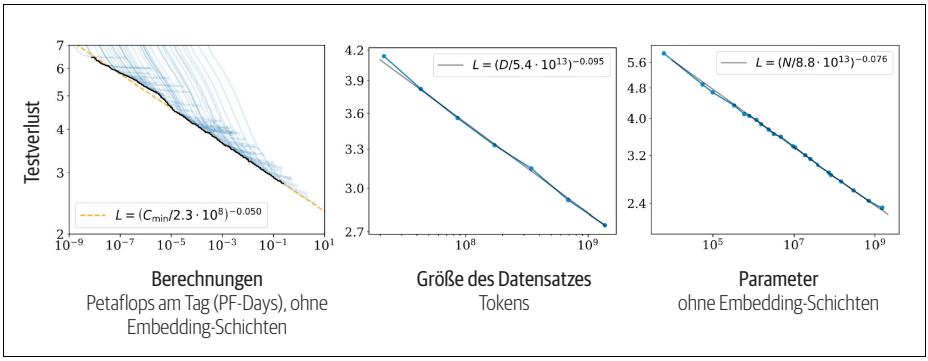


Abbildung 11-2: Skalierung, die dem Potenzgesetz folgt: Der Testverlust in Abhängigkeit vom Rechenbudget (links), der Größe des Datensatzes (Mitte) und der Größe des Modells (rechts) (mit freundlicher Genehmigung von Jared Kaplan)

Aus diesen Verlustkurven können wir mehrere Rückschlüsse ziehen:

Die Beziehung zwischen Leistung und Skalierung

Auch wenn sich viele NLP-Forscher auf architektonische Verbesserungen oder die Optimierung von Hyperparametern (wie z.B. die Justierung der Anzahl von Schichten oder Attention-Heads) konzentrieren, um die Leistung hinsichtlich vorgegebener Datensätze zu verbessern, ergibt sich aus den Skalierungsgesetzen, dass ein effektiverer Ansatz zu besseren Modellen darin besteht, sich darauf zu konzentrieren, N, C und D gleichzeitig zu erhöhen.

1 J. Kaplan et al., »Scaling Laws for Neural Language Models« (*https://arxiv.org/abs/2001.08361*), (2020).

2 Die Größe des Datensatzes wird anhand der Anzahl der Tokens bemessen, und für die Modellgröße werden die Parameter der Embedding-Schichten nicht berücksichtigt.

Potenzgesetze

Der Testverlust L steht sowohl mit N und C als auch mit D in einer Potenzbeziehung mehrerer Größenordnungen (Potenzbeziehungen entsprechen auf einer doppeltlogarithmischen Skala (engl. Log-Log Scale) einer linearen Beziehung). Für $X = N, C, D$ können wir diese Potenzbeziehungen als $L(X) \sim 1/X^{\alpha}$ ausdrücken, wobei α der Skalierungsexponent ist, der durch eine Anpassung an die in Abbildung 11-2 gezeigten Verlustkurven bestimmt wird.[3] Typischerweise liegen die Werte für α im Bereich von 0,05 bis 0,095. Ein Vorteil, der sich durch diese Potenzgesetze ergibt, ist, dass der anfängliche Teil einer Verlustkurve extrapoliert werden kann und es dementsprechend möglich ist, vorherzusagen, wie hoch der Verlust ungefähr wäre, wenn das Training noch deutlich länger durchgeführt würde.

Durch die Größe des Datensatzes bedingte Effizienz

Große Modelle sind in der Lage, mit einer geringeren Anzahl von Trainingsschritten die gleiche Leistung wie kleinere Modelle zu erbringen. Dies wird deutlich, wenn man die Bereiche vergleicht, in denen die Verlustkurve nach einer bestimmten Anzahl von Trainingsschritten einen Wendepunkt erreicht, was darauf hindeutet, dass die Leistungszuwächse kleiner ausfallen, als wenn das Modell einfach weiter skaliert werden würde.

Überraschenderweise konnten Skalierungsgesetze auch für andere Modalitäten, wie Bilder, Videos und mathematische Problemlösungen, beobachtet werden (siehe Abbildung 11-3).

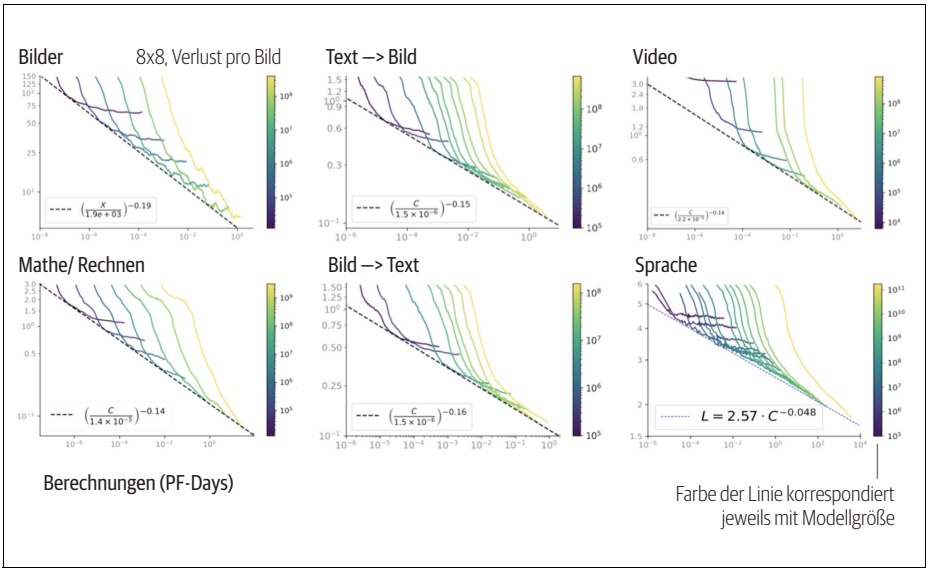


Abbildung 11-3: Skalierung, die dem Potenzgesetz folgt: Der Testverlust in Abhängigkeit vom Rechenbudget für verschiedene Modalitäten (mit freundlicher Genehmigung von Tom Henighan).

3 T. Henighan et al., »Scaling Laws for Autoregressive Generative Modeling« (*https://arxiv.org/abs/2010.14701*), (2020).

Ob Transformer-basierte Sprachmodelle grundsätzlich gemäß dem Potenzgesetz skalieren, ist bis jetzt nicht bekannt. Derzeit können wir die Skalierungsgesetze als Werkzeug verwenden, um den Verlust großer kostspieliger Modelle zu extrapolieren, ohne sie überhaupt trainieren zu müssen. Transformer-Modelle zu skalieren, ist allerdings nicht ganz so einfach, wie es klingt. Sehen wir uns nun einige Herausforderungen an, die sich bei der Erschließung dieser Grenzbereiche ergeben.

Herausforderungen bei der Skalierung

Während die Skalierung in der Theorie recht einfach erscheint (»fügen Sie doch einfach mehr Schichten hinzu!«), ergeben sich in der Praxis zahlreiche Schwierigkeiten. Es folgen einige der größten Herausforderungen, auf die Sie bei der Skalierung von Sprachmodellen wahrscheinlich einstellen müssen:

Infrastruktur

Die Bereitstellung und das Management einer Infrastruktur, die potenziell Hunderte oder Tausende von Knoten mit ebenso vielen GPUs umfasst, sind nichts für schwache Nerven. Ist überhaupt die erforderliche Anzahl an Knoten verfügbar? Stellt die Kommunikation zwischen den Knoten womöglich einen Engpass dar? Um Probleme dieser Art bewältigen zu können, sind Fähigkeiten erforderlich, die von den meisten Data-Science-Teams nicht abgedeckt werden, und setzt in der Regel spezialisierte Engineers voraus, die bereits damit vertraut sind, Experimente in großem Maßstab in einer verteilten Infrastruktur durchzuführen.

Kosten

Die meisten ML-Praktiker kennen wohl das Gefühl, mitten in der Nacht schweißgebadet aufzuwachen und sich daran zu erinnern, dass sie vergessen haben, die hochmoderne GPU in der Cloud abzuschalten. Dieses Gefühl verstärkt sich noch, wenn die Experimente, die durchgeführt werden, groß angelegt sind. Die meisten Unternehmen können sich die Teams und Ressourcen nicht leisten, die für das Training von Modellen in sehr großem Maßstab erforderlich sind, denn das Training eines einzigen Modells der Größe von GPT-3 kann bereits mehrere Millionen Dollar verschlingen, und für gewöhnlich haben die meisten Unternehmen dafür nicht das nötige Kleingeld.[4]

Erstellung der Datensätze

Ein Modell ist nur so gut wie die Daten, auf denen es trainiert wird. Das Trainieren großer Modelle erfordert Datensätze, die umfangreich und hochwertig sind. Wenn Terabytes an Textdaten in einen Datensatz einbezogen werden, wird es schwieriger, sicherzustellen, dass er aus qualitativ hochwertigem Text besteht. Selbst die Aufbereitung des Datensatzes wird hierbei zur Herausforderung. Darüber hinaus muss sichergestellt werden, dass es einen Weg gibt,

4 Kürzlich wurde jedoch ein verteiltes Deep-Learning-Framework vorgeschlagen, das es kleineren Teams ermöglicht, ihre Rechenressourcen zusammenzulegen und Modelle gemeinsam zu trainieren. Siehe M. Diskin et al., »Distributed Deep Learning in Open Collaborations« (*https://arxiv.org/abs/2106.10207*), (2021).

möglicherweise enthaltene sexistische oder rassistische Vorurteile zu kontrollieren, die diese Sprachmodelle beim Training auf großen, auf Onlinetexten basierende Korpora übernehmen können. Zudem muss auch die Lizenz bzw. das Urheberrecht der Trainingsdaten berücksichtigt werden, und es muss darauf Acht gegeben werden, dass auch persönliche Informationen in den großen Textdatensätzen enthalten sein könnten.

Evaluierung von Modellen
: Auch nach dem Training des Modells bleiben Herausforderungen nicht aus. Die Evaluierung eines Modells im Hinblick auf nachgelagerte Aufgaben erfordert erneut Zeit und Ressourcen. Darüber hinaus sollten Sie das Modell auf vorurteilsbehaftete bzw. abträgliche Generierungen untersuchen, selbst wenn Sie sicher sind, dass der Datensatz, den Sie erstellt haben, ausreichend bereinigt wurde. Diese Schritte erfordern Zeit und müssen sorgfältig durchgeführt werden, denn nur so kann das Risiko späterer negativer Auswirkungen minimiert werden.

Deployment
: Schließlich stellt auch die Bereitstellung großer Sprachmodelle eine große Herausforderung dar. In Kapitel 8 haben Sie einige Ansätze wie Knowledge Distillation, Pruning und Quantisierung kennengelernt, die bei Problemen dieser Art weiterhelfen. Wenn Sie ein Modell haben, das Hunderte von Gigabytes groß ist, reicht dies jedoch möglicherweise nicht aus. Gehostete Dienste wie die API von OpenAI (*https://beta.openai.com*) oder die Accelerated Inference API (*https://oreil.ly/E4q3b*) von Hugging Face wurden entwickelt, um Unternehmen, die sich nicht mit diesen Problemen beim Deployment auseinandersetzen können oder wollen, eine Unterstützung zu bieten.

Diese Auflistung ist keineswegs vollständig, sollte Ihnen aber eine Vorstellung vermitteln, welche Herausforderungen mit der zunehmenden Skalierung von Sprachmodellen verbunden sind und welche Überlegungen angestellt werden müssen, um diesen zu begegnen. Während die meisten dieser Bemühungen auf einige wenige Institutionen konzentriert sind, die über die Ressourcen und das Know-how verfügen, um diese Einschränkungen zu überwinden, gibt es derzeit zwei von der Community getragene Projekte, die darauf abzielen, große Sprachmodelle öffentlich nachvollziehbar zu erstellen und zu erproben:

BigScience
: Dies ist ein einjähriger Forschungsworkshop, der im Zeitraum von 2021 bis 2022 durchgeführt wurde und sich auf große Sprachmodelle konzentriert hat. Der Workshop zielt darauf ab, Diskussionen und Überlegungen zu den Forschungsfragen rund um diese Modelle (Fähigkeiten, Grenzen, Verbesserungsmöglichkeiten, Verzerrungen bzw. Voreingenommenheiten, Ethik, Auswirkungen auf die Umwelt, Rolle in der allgemeinen KI bzw. in der Cognitive Science) sowie zu den Herausforderungen bei der Erstellung und gemeinsamen Nutzung solcher Modelle und Datensätze für Forschungszwecke und innerhalb der Forschungsgemeinschaft zu fördern. Die gemeinschaftlichen Aufgaben umfassen die Erstellung, die gemeinsame Nutzung und die Evaluierung

eines großen mehrsprachigen Datensatzes und großen Sprachmodells. Für diese Aufgaben wurde ein außergewöhnlich hohes Rechenbudget bereitgestellt (mehrere Millionen GPU-Stunden auf mehreren Tausend GPUs). Sollte dieser Workshop von Erfolg gekrönt sein, wird er in Zukunft erneut stattfinden, wobei der Schwerpunkt darauf liegen wird, die kollaborativen Aufgaben zu aktualisieren oder weitere einzubeziehen. Wenn Sie sich an den Bemühungen beteiligen möchten, finden Sie hierzu weitere Informationen auf der Webseite des Projekts (*https://oreil.ly/13xfb*).

EleutherAI
: Es handelt sich hierbei um ein dezentral organisiertes Kollektiv von freiwilligen Forschern, Engineers und Entwicklern, die sich auf die Ausrichtung (engl. Alignment) von künstlicher Intelligenz (KI), die Skalierung und die Open-Source-KI-Forschung konzentrieren. Eines ihrer Ziele ist es, ein Modell in der Größe von GPT-3 zu trainieren und es als Open-Source-Version zur Verfügung zu stellen. Die Gruppe hat bereits einige beeindruckende Modelle veröffentlicht, wie GPT-Neo (*https://oreil.ly/ZVGaz*) oder GPT-J (*https://oreil.ly/Kup60*), ein Modell mit 6 Milliarden Parametern, das das derzeit leistungsstärkste, öffentlich zugängliche Transformer-Modell im Hinblick auf die Zero-Shot-Performance ist. Weitere Informationen finden Sie auf der Website (*https://eleuther.ai*) von EleutherAI.

Nachdem wir nun herausgefunden haben, dass Transformer hinsichtlich der Rechenleistung, der Modellgröße und der Größe des Datensatzes skaliert werden können, widmen wir uns einem weiteren aktuellen Forschungsbereich: der Effizienz des Self-Attention-Mechanismus.

Attention Please! – Den Attention-Mechanismus effizienter gestalten

Wir haben in diesem Buch bereits erfahren, dass der Self-Attention-Mechanismus eine zentrale Rolle in der Architektur von Transformern spielt – schließlich heißt der ursprüngliche Transformer-Forschungsbeitrag ja auch »Attention Is All You Need«. Es gibt jedoch eine entscheidende Herausforderung im Zusammenhang mit der Self-Attention: Da die Gewichte aus paarweisen Vergleichen aller Tokens in einer Sequenz generiert werden, wird diese Schicht zu einem Rechenengpass, wenn Sie versuchen, lange Dokumente zu verarbeiten oder Transformer auf Bereiche wie die Sprachverarbeitung oder Computer Vision anzuwenden. In Bezug auf die Zeit- und Speicherkomplexität skaliert die Self-Attention-Schicht der Transformer-Architektur an sich mit einer Komplexität von $O(n^2)$, wobei n die Länge der Sequenz darstellt.[5]

5 Obwohl die Standardimplementierungen von Self-Attention eine Zeit- und Speicherkomplexität von $O(n^2)$ aufweisen, zeigt ein kürzlich veröffentlichtes Forschungspapier von Google-Forschern (*https://arxiv.org/abs/2112.05682*), dass die Speicherkomplexität durch eine einfache Anpassung der Anordnung der Operationen auf $O(\log n)$ verringert werden kann.

Daher konzentriert sich ein wesentlicher Teil der jüngsten Forschung zu Transformern darauf, den Self-Attention-Mechanismus effizienter zu gestalten. Die wichtigsten Forschungsstränge sind weitestgehend in Abbildung 11-4 zusammengefasst.

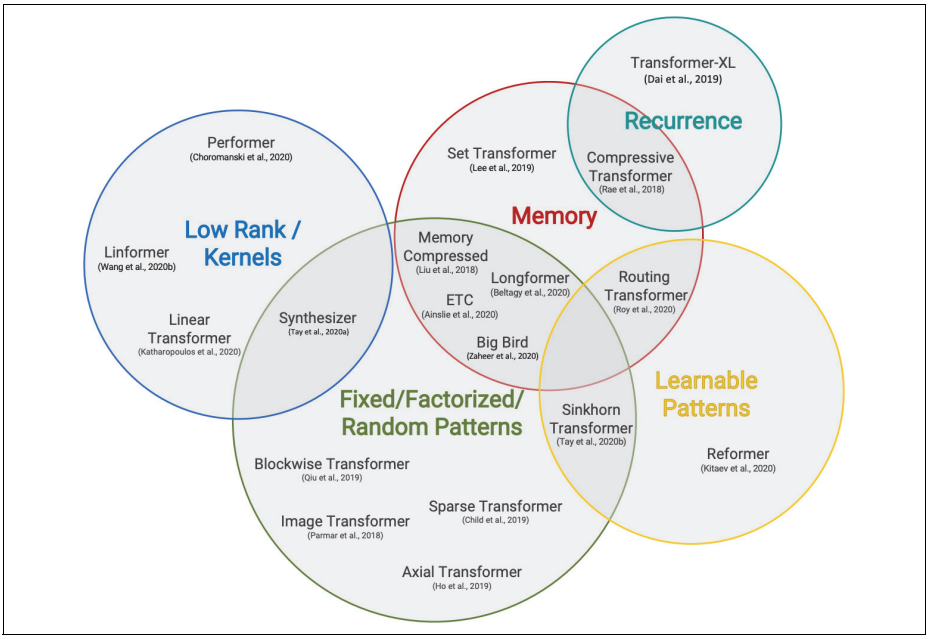


Abbildung 11-4: Eine Übersicht über die Forschungsstränge, die darauf abzielen, den Attention-Mechanismus effizienter zu gestalten (mit freundlicher Genehmigung von Yi Tay et al.)[6]

Üblicherweise wird die Effizienz der Attention entweder durch die Erhöhung der Sparsität (engl. Sparsity) des Attention-Mechanismus oder durch die Anwendung von Kerneln bzw. Filtern auf die Attention-Matrix erhöht. Werfen wir einen kurzen Blick auf einige der beliebtesten Ansätze, mit denen versucht wird, den Self-Attention-Mechanismus zu verbessern. Beginnen wir mit den Ansätzen, die darauf abzielen, die Sparsität zu erhöhen.

Sparse-Attention

Eine Möglichkeit, die Anzahl der in der Self-Attention-Schicht durchgeführten Berechnungen zu reduzieren, besteht darin, einfach die Anzahl der Abfrage/Schlüssel-Paare, die nach einem vordefinierten Muster erzeugt werden, zu begrenzen. In der Literatur wurden bereits zahlreiche Ansätze bzw. Muster (engl. Patterns) zur Erhöhung der Sparsität erforscht, die meisten von ihnen lassen sich jedoch in eine Handvoll »grundlegender« Patterns zerlegen, wie in Abbildung 11-5 dargestellt.

6 Yi Tay et al., »Efficient Transformers: A Survey« (https://arxiv.org/abs/2009.06732), (2020).

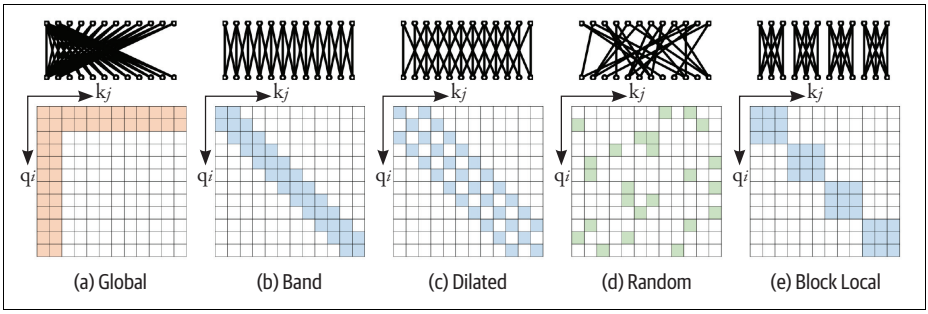


Abbildung 11-5: Geläufige dünnbesetzte Attention-Patterns für die Self-Attention: Ein eingefärbtes Quadrat bedeutet, dass der Attention-Score berechnet wird, während ein leeres Quadrat bedeutet, dass der Score nicht ermittelt wird (mit freundlicher Genehmigung von Tianyang Lin).

Wir können diese Patterns[7] wie folgt beschreiben:

Global-Attention
: Gibt einige spezielle Tokens in der Sequenz vor, die die Attention auf alle anderen Tokens richten können.

Band-Attention
: Berechnet die Attention mithilfe einer Bandmatrix.

Dilated-Attention
: Überspringt einige Abfrage/Schlüssel-Paare, indem es ein Dilated Window (bzw. Fenster mit Dilatation) verwendet, d.h. ein Window, das Lücken aufweist.

Random-Attention
: Ermittelt zufällig einige Schlüssel für jede Abfrage, um die Attention-Scores zu berechnen.

Block-Local-Attention
: Unterteilt die Sequenz in Blöcke und schränkt die Attention innerhalb dieser Blöcke ein.

In der Praxis verwenden die meisten Transformer-Modelle mit Sparse-Attention eine Mischung aus den in Abbildung 11-5 gezeigten atomaren Patterns, um die finale Attention-Matrix zu erzeugen. Wie in Abbildung 11-6 veranschaulicht, verwenden Modelle wie Longformer (*https://oreil.ly/F7xCY*) eine Mischung aus Global- und Band-Attention, während bei BigBird (*https://oreil.ly/yFPyj*) diese Kombination um Random-Attention ergänzt wird. Die Erhöhung der Sparsität der Attention-Matrix (d.h., sie dünnbesetzter zu machen) ermöglicht es diesen Modellen, viel längere Sequenzen zu verarbeiten. Im Fall von Longformer und BigBird beträgt die maximale Sequenzlänge 4.096 Tokens, also eine 8-mal so lange Sequenzlänge wie bei BERT!

7 T. Lin et al., »A Survey of Transformers« (*https://arxiv.org/abs/2106.04554*), (2021).

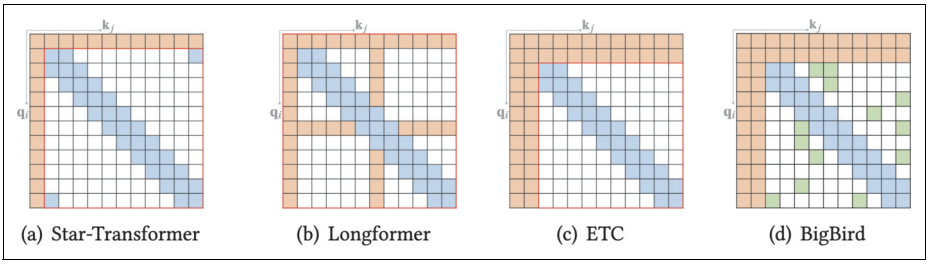


Abbildung 11-6: Sparse-Attention-Patterns, die bei aktuellen Transformer-Modellen genutzt werden (mit freundlicher Genehmigung von Tianyang Lin)

Außerdem ist es möglich, das Sparsität-Pattern auf Basis von Daten zu *lernen*. Die Grundidee hinter solchen Ansätzen besteht darin, die Tokens in Chunks zu clustern. So verwendet Reformer (*https://oreil.ly/yIVvX*) beispielsweise eine Hash-Funktion, um sich ähnelnde Tokens zusammenzufassen.

Nachdem wir nun wissen, wie Sparsität dazu beitragen kann, die Komplexität der Self-Attention zu verringern, werfen wir einen Blick auf einen weiteren beliebten Ansatz, der auf der direkten Anpassung der Operationen beruht.

Linearisierte Attention

Alternativ können Sie die Self-Attention effizienter gestalten, indem Sie die Reihenfolge der Operationen ändern, die zur Berechnung der Attention-Scores erforderlich sind. Denken Sie bitte daran, dass wir zur Berechnung der Self-Attention Scores der Abfragen und Schlüssel eine Ähnlichkeitsfunktion benötigen, die für den Transformer nur ein einfaches Skalarprodukt ist. Für eine allgemeine Ähnlichkeitsfunktion $\text{sim}(q_i, k_j)$ können wir jedoch die Ausgaben für die Attention durch folgende Gleichung ausdrücken:

$$y_i = \sum_j \frac{\text{sim}(Q_i, K_j)}{\sum_k \text{sim}(Q_i, K_k)} V_j$$

Der Trick hinter linearisierten Attention-Mechanismen (engl. Linearized Attention Mechanisms) besteht darin, die Ähnlichkeitsfunktion als *Kernel-Funktion* auszudrücken, die die Operation in zwei Teile zerlegt:

$$\text{sim}(Q_j, K_j) = \varphi(Q_i)^T \varphi(K_j)$$

wobei ϕ normalerweise eine hochdimensionale Feature-Map ist. Da $\phi(Q_i)$ unabhängig von j und k ist, können wir es mit in die Summen einbeziehen und die Ausgaben für die Attention wie folgt umformulieren:

$$y_i = \frac{\varphi(Q_i)^T \sum_j \varphi(K_j) V_j^T}{\varphi(Q_i)^T \sum_k \varphi(K_k)}$$

Indem wir zuerst $\sum_j \phi(K_j) V_j^T$ und $\sum_k \phi(K_k)$ berechnen, ist es uns möglich, die räumliche und zeitliche Komplexität der Self-Attention in effektiver Weise zu linearisieren! In Abbildung 11-7 werden die beiden Ansätze noch einmal gegenübergestellt. Beliebte Modelle, die eine linearisierte Form der Self-Attention implementieren, sind der Linear Transformer und der Performer.[8]

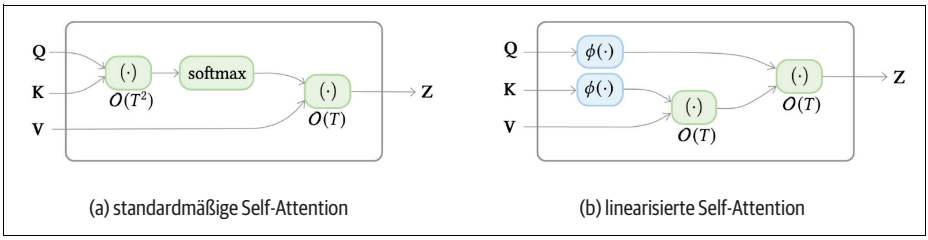


Abbildung 11-7: Komplexitätsunterschied zwischen der standardmäßigen und der linearisierten Form der Self-Attention (mit freundlicher Genehmigung von Tianyang Lin)

In diesem Abschnitt haben wir uns damit befasst, wie Transformer-Architekturen im Allgemeinen und Attention im Besonderen skaliert werden können, um eine noch bessere Performance bei einer Vielzahl von Aufgaben zu erzielen. Im nächsten Abschnitt sehen wir uns an, wie Transformer über NLP hinaus in andere Bereiche wie der Verarbeitung von Audio- bzw. Tondaten und der Computer Vision vorgedrungen sind.

Jenseits von Textdaten

Der Ansatz, Texte dafür zu verwenden, große Sprachmodelle zu trainieren, und diese mit Transfer Learning in Anwendung zu bringen, war der Schlüssel zum Erfolg Transformer-basierter Sprachmodelle. Zum einen ermöglichen Textdaten, die schon heutzutage im Überfluss vorhanden sind, große Modelle auf selbstüberwachte (engl. Self-Supervised) Weise zu trainieren. Zum anderen sind Aufgaben, bei denen Textdaten zum Einsatz kommen, wie z. B. die Klassifizierung und das Question Answering, bereits weit verbreitet. Dabei hat die Entwicklung effektiver Strategien, mit denen diese Aufgaben gelöst werden können, dazu beigetragen, dass sich diese auch auf eine breite Palette anderer realer Probleme übertragen lassen.

Allerdings geht dieser Ansatz auch mit diversen Einschränkungen einher, wie den folgenden:

8 A. Katharopoulos et al., »Transformers Are RNNs: Fast Autoregressive Transformers with Linear Attention« (*https://arxiv.org/abs/2006.16236*), (2020); K. Choromanski et al., »Rethinking Attention with Performers« (*https://arxiv.org/abs/2009.14794*), (2020).

Human Reporting Bias (Verzerrungen in der Berichterstattung durch Menschen)
Die Häufigkeit, mit der Ereignisse im Text Erwähnung finden, entspricht nicht zwingend der Häufigkeit, mit der sie tatsächlich auftreten.[9] Ein Modell, das ausschließlich auf Texten aus dem Internet trainiert wurde, könnte dementsprechend ein sehr verfälschtes Bild von der Welt haben.

Gesunder Menschenverstand (engl. Common Sense)
Der gesunde Menschenverstand ist eine grundlegende Eigenschaft menschlichen Denkens, wird aber nur selten schriftlich festgehalten. Daher kennen Sprachmodelle, die anhand von Texten trainiert wurden, zwar viele Fakten über die Welt, sind aber oftmals nicht dazu fähig, vernünftige Rückschlüsse zu ziehen.

Fakten
Ein wahrscheinlichkeitsbasiertes Sprachmodell kann Fakten nicht zuverlässig speichern und kann daher Texte generieren, die sachlich falsch sind. Ebenso können solche Modelle zwar benannte Entitäten bzw. Eigennamen erkennen, haben jedoch keine unmittelbare Möglichkeit, auf weitere Informationen über sie zuzugreifen.

Modalität (engl. Modality)
Sprachmodelle haben keine Möglichkeit, andere Modalitäten, wie z. B. Audio- oder visuelle Signale oder tabellarische Daten, einzubeziehen, die die vorherigen Einschränkungen adressieren könnten.

Wenn wir also die Grenzen der Modalität überwinden könnten, würden wir möglicherweise auch einige der anderen Einschränkungen angehen können. In letzter Zeit gab es viele Entwicklungen, um Transformer-Modelle auf weitere Modalitäten auszuweiten und sogar multimodale Modelle zu erstellen. In diesem Abschnitt werden wir einige dieser Fortschritte beleuchten.

Computer Vision

Mit Beginn der Deep-Learning-Revolution traten Convolutional Neural Networks (CNNs) in der Computer Vision in den Vordergrund. Inzwischen kommen in diesem Bereich auch Transformer-Modelle zum Einsatz, die ähnlich effizient oder gar effizienter als CNNs sind. Sehen wir uns ein paar Beispiele an.

iGPT

Inspiriert durch den Erfolg von GPT-basierten Modellen bei Texten, wendet iGPT (kurz für »image GPT«) die gleichen Methoden auf Bilder an.[10] Das iGPT-Modell betrachtet Bilder als Sequenzen von Pixeln, wobei es auf die GPT-Architektur zu-

9 J. Gordon und B. Van Durme, »Reporting Bias and Knowledge Extraction« (*https://openreview.net/pdf?id=AzxEzvpdE3Wcy*), (2013).

10 M. Chen et al., »Generative Pretraining from Pixels,« *Proceedings of the 37th International Conference on Machine Learning* 119 (2020):1691–1703, *https://proceedings.mlr.press/v119/chen20s.html*.

rückgreift und das autoregressive Pretraining-Objective nutzt, um die nächsten Pixelwerte vorherzusagen. Indem iGPT auf großen Bilddatensätzen vortrainiert wird, ist es in der Lage, Teile von Bildern »automatisch zu vervollständigen«, wie in Abbildung 11-8 dargestellt. Es erzielt auch hervorragende Ergebnisse bei Klassifizierungsaufgaben, wenn dem Modell ein Head zur Klassifizierung hinzugefügt wird.

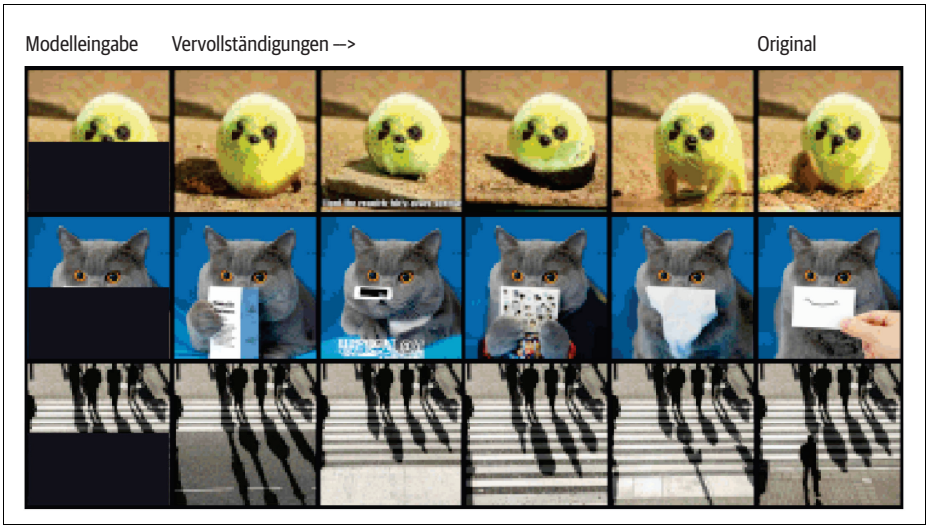


Abbildung 11-8: Beispiele für Bildvervollständigungen von iGPT (mit freundlicher Genehmigung von Mark Chen)

ViT

Wir haben gesehen, dass sich iGPT eng an die Architektur von GPT-basierten Modellen anlehnt und der gleiche Ansatz beim Pretraining verfolgt wird. Der Vision Transformer (ViT) ist eine an BERT angelehnte Version von Transformern zur Verarbeitung von Bildern, wie in Abbildung 11-9 dargestellt. Zunächst wird das Bild in kleinere Bereiche aufgeteilt, die jeweils in Form einer linearen Projektion eingebettet werden. Diese ähneln stark den Token-Embeddings im BERT-Modell, und auch die anschließenden Schritte sind nahezu identisch. Die Patch-Embeddings werden mit Position-Embeddings kombiniert und dann durch einen gewöhnlichen Encoder-basierten Transformer geleitet. Während des Pretrainings werden einige der Patches maskiert oder verfälscht, und das Ziel ist es, die durchschnittliche Farbe des maskierten Patches vorherzusagen.

Obwohl dieser Ansatz keine besseren Ergebnisse lieferte, wenn er auf dem standardmäßigen ImageNet-Datensatz vortrainiert wurde, skalierte er deutlich besser als CNNs auf größeren Datensätzen.

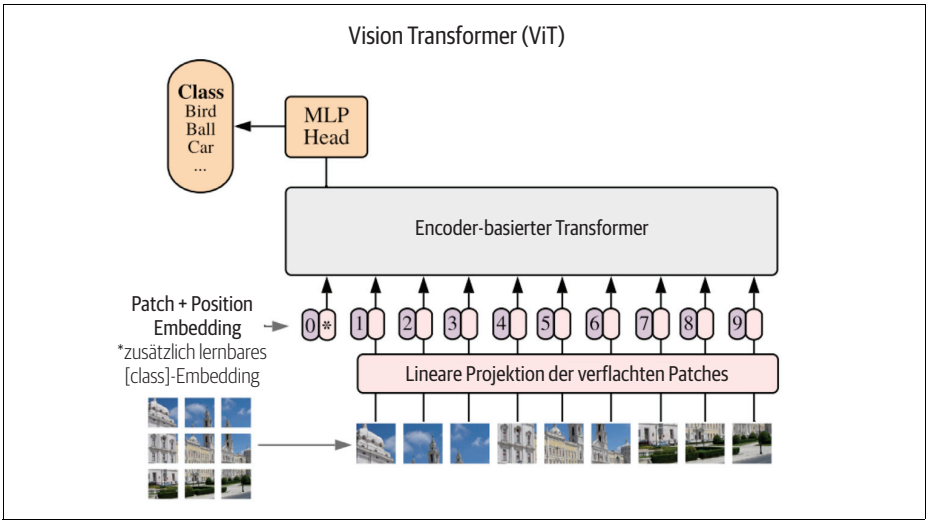


Abbildung 11-9: Die ViT-Architektur (mit freundlicher Genehmigung von Alexey Dosovitskiy et al.)

Der ViT ist in die 🤗 Transformers-Bibliothek integriert und er kann ähnlich wie die NLP-Pipelines genutzt werden, die wir in diesem Buch verwendet haben. Probieren wir ihn aus und laden wir zunächst das Bild eines ziemlich berühmten Hunds:

```
from PIL import Image
import matplotlib.pyplot as plt

image = Image.open("images/doge.jpg")
plt.imshow(image)
plt.axis("off")
plt.show()
```

Um ein ViT-Modell zu laden, müssen wir lediglich die Pipeline `image-classification` angeben. Anschließend speisen wir das Bild ein und erhalten so die vorhergesagten Kategorien:

```
import pandas as pd
from transformers import pipeline

image_classifier = pipeline("image-classification")
preds = image_classifier(image)
preds_df = pd.DataFrame(preds)
preds_df
```

	score	label
0	0.643599	Eskimo dog, husky
1	0.207407	Siberian husky
2	0.060160	dingo, warrigal, warragal, Canis dingo
3	0.035359	Norwegian elkhound, elkhound
4	0.012927	malamute, malemute, Alaskan malamute

Großartig, die vorhergesagte Kategorie scheint dem Bild zu entsprechen!

Eine logische Erweiterung von Bildmodellen sind Videomodelle. Zusätzlich zu den räumlichen gibt es bei Videos auch eine zeitliche Dimension. Das macht die Aufgabe noch schwieriger, da die Menge an Daten wesentlich größer wird und auch eine zusätzliche Dimension berücksichtigt werden muss. Um beiden Aspekten Rechnung zu tragen, führen Modelle wie der TimeSformer zusätzlich zum räumlichen auch einen zeitlichen Attention-Mechanismus ein.[11] In Zukunft können solche Modelle dabei helfen, Tools für eine breite Palette von Aufgaben wie die Klassifizierung oder Kommentierung von Videosequenzen zu entwickeln.

Tabellen

Viele Daten, wie z. B. Kundendaten in einem Unternehmen, werden in strukturierten Datenbanken und nicht in Form von Rohtext gespeichert. Wir haben in Kapitel 7 gesehen, dass wir mit Question-Answering-Modellen Texte mit einer in natürlicher Sprache gehaltener Frage abfragen können. Wäre es nicht praktisch, wenn wir dasselbe für Tabellen durchführen könnten, wie in Abbildung 11-10 gezeigt?

TAPAS (kurz für Table Parser)[12] ist die Lösung! Dieses Modell wendet die Transformer-Architektur auf Tabellen an, indem es die tabellarischen Informationen mit der Abfrage kombiniert, wie in Abbildung 11-11 dargestellt.

11 G. Bertasius, H. Wang, and L. Torresani, »Is Space-Time Attention All You Need for Video Understanding?« (*https://arxiv.org/abs/2102.05095*), (2021).

12 J. Herzig et al., »TAPAS: Weakly Supervised Table Parsing via Pre-Training« (*https://arxiv.org/abs/2004.02349*), (2020).

Tabelle

Rank	Name	No. of reigns	Combined days
1	Lou Thesz	3	3,749
2	Ric Flair	8	3,103
3	Harley Race	7	1,799
4	Dory Funk Jr.	1	1,563
5	Dan Severn	2	1,559
6	Gene Kiniski	1	1,131

Beispielfragen

#	Question	Answer	Example Type
1	*Which wrestler had the most number of reigns?*	Ric Flair	Cell selection
2	*Average time as champion for top 2 wrestlers?*	AVG(3749,3103)=3426	Scalar answer
3	*How many world champions are there with only one reign?*	COUNT(Dory Funk Jr., Gene Kiniski)=2	Ambiguous answer
4	*What is the number of reigns for* Harley Race?	7	Ambiguous answer
5	*Which of the following wrestlers were ranked in the bottom 3?*	{Dory Funk Jr., Dan Severn, Gene Kiniski}	Cell selection
	Out of these, who had more than one reign?	Dan Severn	Cell selection

Abbildung 11-10: Question Answering auf Basis einer Tabelle (mit freundlicher Genehmigung von Jonathan Herzig)

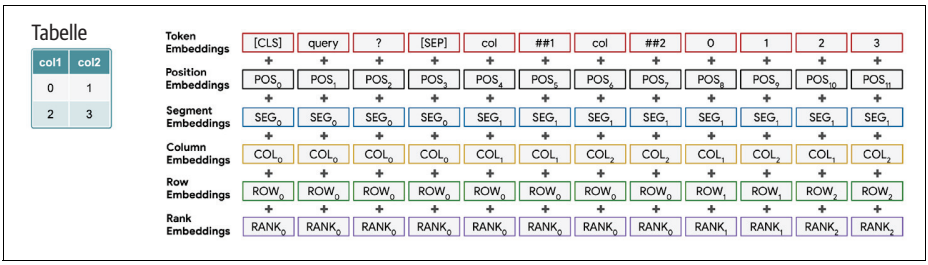


Abbildung 11-11: Die Architektur von TAPAS (mit freundlicher Genehmigung von Jonathan Herzig)

Sehen wir uns anhand eines praktischen Beispiels an, wie TAPAS funktioniert. Wir haben eine fiktive Version des Inhaltsverzeichnisses der englischen Ausgabe dieses Buchs erstellt. Es enthält die Kapitelnummer, den Namen des Kapitels sowie die Anfangs- und Endseiten der Kapitel:

```
book_data = [
    {"chapter": 0, "name": "Introduction", "start_page": 1, "end_page": 11},
    {"chapter": 1, "name": "Text classification", "start_page": 12,
     "end_page": 48},
    {"chapter": 2, "name": "Named Entity Recognition", "start_page": 49,
     "end_page": 73},
    {"chapter": 3, "name": "Question Answering", "start_page": 74,
     "end_page": 120},
    {"chapter": 4, "name": "Summarization", "start_page": 121,
     "end_page": 140},
    {"chapter": 5, "name": "Conclusion", "start_page": 141,
     "end_page": 144}
]
```

Auf Basis der vorhandenen Felder können wir zudem die Anzahl der Seiten pro Kapitel hinzufügen. Damit das TAPAS-Modell die Daten verarbeiten kann, müssen wir sicherstellen, dass alle Spalten vom Typ `str` sind:

```
table = pd.DataFrame(book_data)
table['number_of_pages'] = table['end_page']-table['start_page']
table = table.astype(str)
table
```

	chapter	name	start_page	end_page	number_of_pages
0	0	Introduction	1	11	10
1	1	Text classification	12	48	36
2	2	Named Entity Recognition	49	73	24
3	3	Question Answering	74	120	46
4	4	Summarization	121	140	19
5	5	Conclusion	141	144	3

Mittlerweile dürfte Ihnen der Ablauf vertraut sein. Zuerst laden wir die Pipeline `table-question-answering`:

```
table_qa = pipeline("table-question-answering")
```

und übergeben der Pipeline anschließend einige Abfragen, um die entsprechenden Antworten zu erhalten:

```
table_qa = pipeline("table-question-answering")
queries = ["What's the topic in chapter 4?",
           "What is the total number of pages?",
           "On which page does the chapter about question-answering start?",
           "How many chapters have more than 20 pages?"]
preds = table_qa(table, queries)
```

In dem Vorhersage-Objekt werden sowohl die Art der Tabellenoperation, die in einem `aggregator`-Feld hinterlegt ist, als auch die Antwort gespeichert. Schauen wir uns nun an, wie gut TAPAS bei unseren Fragen abgeschnitten hat:

```
for query, pred in zip(queries, preds):
    print(query)
    if pred["aggregator"] == "NONE":
        print("Vorhergesagte Antwort: " + pred["answer"])
    else:
        print("Vorhergesagte Antwort: " + pred["answer"])
    print('='*50)
```

```
What's the topic in chapter 4?
Vorhergesagte Antwort: Summarization
==================================================
What is the total number of pages?
Vorhergesagte Antwort: SUM > 10, 36, 24, 46, 19, 3
==================================================
On which page does the chapter about question-answering start?
Vorhergesagte Antwort: AVERAGE > 74
==================================================
How many chapters have more than 20 pages?
Vorhergesagte Antwort: COUNT > 1, 2, 3
==================================================
```

Für die erste Frage hat das Modell genau eine Zelle vorhergesagt, wobei keine Aggregation vorgenommen werden muss. Ein Blick auf die Tabelle zeigt, dass die Antwort in der Tat richtig ist. Im nächsten Beispiel hat das Modell alle Zellen vorhergesagt, die die Anzahl der Seiten enthalten, und zudem die Aggregatfunktion SUM, was wiederum die korrekte Methode zur Berechnung der Gesamtzahl der Seiten

ist. Die Antwort auf Frage drei ist ebenfalls korrekt. Die Aggregatfunktion AVERAGE ist in diesem Fall nicht notwendig, macht aber auch keinen Unterschied. Schließlich haben wir noch eine Frage, die etwas komplexer ist. Um festzustellen, wie viele Kapitel mehr als 20 Seiten haben, müssen wir zunächst herausfinden, welche Kapitel dieses Kriterium erfüllen, und sie dann zählen. Es scheint, als hätte TAPAS wieder alles richtig gemacht. Es hat korrekt festgestellt, dass die Kapitel 1, 2 und 3 mehr als 20 Seiten haben, und dass auf die entsprechenden Zellen die Aggregatfunktion COUNT angewandt werden müsste.

Die Arten von Fragen, die wir gestellt haben, können natürlich auch mit ein paar einfachen Pandas-Befehlen beantwortet werden. Die Möglichkeit, Fragen in natürlicher Sprache anstelle von Python-Code zu stellen, ermöglicht es jedoch einem wesentlich breiteren Publikum, die Daten abzufragen, um auf spezifische Fragen Antworten zu erhalten. Denken Sie vor diesem Hintergrund beispielsweise an Unternehmensanalysten oder Manager, die mithilfe solcher Tools ihre eigenen Hypothesen über die Daten völlig selbstständig überprüfen könnten.

Multimodale Transformer

Bisher haben wir uns mit der Erweiterung von Transformern auf eine einzige neue Modalität beschäftigt. TAPAS ist zwar multimodal, da es Text und Tabellen kombiniert, allerdings wird die Tabelle ebenfalls als Text behandelt. In diesem Abschnitt untersuchen wir Transformer, die zwei Modalitäten gleichzeitig kombinieren: Audio- und Textdaten sowie Bild- und Textdaten.

Speech-to-Text

Obwohl die Möglichkeit, Text als Schnittstelle zu einem Computer zu verwenden, ein großer Fortschritt ist, ist die gesprochene Sprache für uns eine noch natürlichere Art der Kommunikation. Dieser Trend lässt sich in der Industrie beobachten, wo Anwendungen wie Siri und Alexa auf dem Vormarsch sind und immer nützlicher werden. Außerdem sind Schreiben und Lesen für einen großen Teil der Bevölkerung eine größere Herausforderung als das Sprechen. Die Fähigkeit, Audiosignale zu verarbeiten und zu verstehen, ist also nicht nur praktisch, sondern kann vielen Menschen auch den Zugang zu mehr Informationen erleichtern. Eine häufige Aufgabe in diesem Bereich ist die *automatische Spracherkennung* (engl. Automatic Speech Recognition, ASR), bei der gesprochene Wörter in Text umgewandelt und Sprachtechnologien wie Siri dazu befähigt werden, Fragen wie »Wie ist heute das Wetter?« beantworten zu können.

Die Modelle der wav2vec-2.0 (*https://oreil.ly/tPpC7*)-Familie sind eine der jüngsten Entwicklungen im Bereich ASR: Sie verwenden eine Transformer-Schicht in Kombination mit einem CNN, wie in Abbildung 11-12 dargestellt.[13] Durch die Nut-

13 A. Baevski et al., »wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations« (*https://arxiv.org/abs/2006.11477*), (2020).

zung von ungelabelten Daten während des Pretrainings erzielen diese Modelle mit gelabelten Daten in nur wenigen Minuten konkurrenzfähige Ergebnisse.

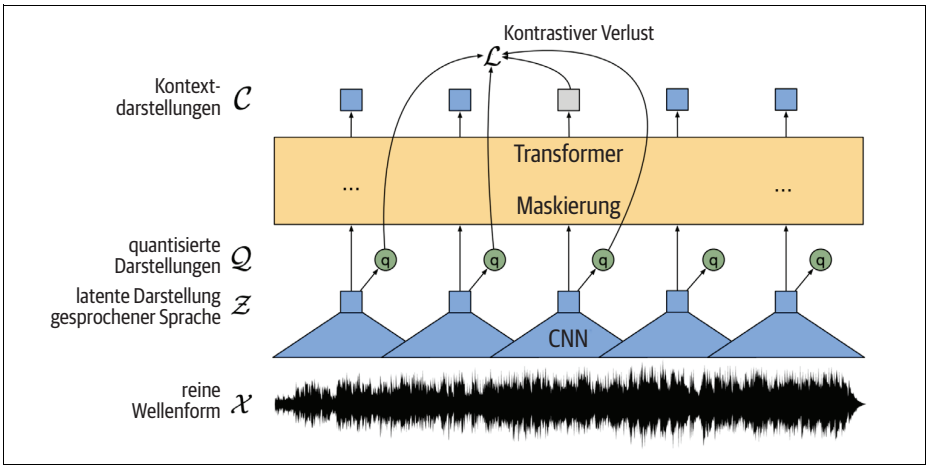


Abbildung 11-12: Architektur von wav2vec 2.0 (mit freundlicher Genehmigung von Alexei Baevski)

Es wird Sie nicht überraschen, dass das Laden und Verwenden der wav2vec-2.0-Modelle in der 🤗 Transformers-Bibliothek den altbekannten Schritten folgt, die wir in diesem Buch bereits kennengelernt haben. Wir können nun ein vortrainiertes Modell laden, das auf Basis von 960 Stunden langen Sprachaufnahmen trainiert wurde:

```
asr = pipeline("automatic-speech-recognition")
```

Um dieses Modell auf einige Audiodateien anzuwenden, verwenden wir die für die automatische Spracherkennung vorgesehene Teilmenge (ASR Subset) des SUPERB-Datensatzes (*https://oreil.ly/iBAK8*), d.h. denselben Datensatz, mit dem das Modell vortrainiert wurde. Da der Datensatz recht groß ist, laden wir zu Demonstrationszwecken lediglich ein Beispiel:

```
from datasets import load_dataset

ds = load_dataset("superb", "asr", split="validation[:1]")
print(ds[0])
```

```
{'chapter_id': 128104, 'speaker_id': 1272, 'file': '~/.cache/huggingf
ace/datasets/downloads/extracted/e4e70a454363bec1c1a8ce336139866a39442114d86a433
6014acd4b1ed55e55/LibriSpeech/dev-clean/1272/128104/1272-128104-0000.flac',
'id': '1272-128104-0000', 'text': 'MISTER QUILTER IS THE APOSTLE OF THE MIDDLE
CLASSES AND WE ARE GLAD TO WELCOME HIS GOSPEL'}
```

Wie wir sehen, sind die Audiodaten in der Spalte `file` im FLAC-Codierungsformat gespeichert, während die entsprechende Transkription in der Spalte `text` enthalten ist. Um die Audiodaten in ein Array von Gleitkommazahlen zu konvertieren, kön-

nen wir die *SoundFile*-Bibliothek (*https://oreil.ly/eo106*) verwenden und jede Datei in unserem Datensatz mithilfe der `map()`-Methode einlesen:

```
import soundfile as sf

def map_to_array(batch):
    speech, _ = sf.read(batch["file"])
    batch["speech"] = speech
    return batch

ds = ds.map(map_to_array)
```

Sollten Sie in einem Jupyter Notebook arbeiten, können Sie sich die Dateien auf einfache Weise mit dem folgenden `IPython`-Widget anhören:

```
from IPython.display import Audio

display(Audio(ds[0]['speech'], rate=16000))
```

Abschließend können wir die Eingabedaten an die Pipeline übergeben und die Vorhersage inspizieren:

```
ds.set_format("numpy")
pred = asr(ds[0]["speech"])
print(pred)
```

```
{'text': 'MISTER QUILTER IS THE APOSTLE OF THE MIDDLE CLASSES AND WE ARE GLAD TO
WELCOME HIS GOSPEL'}
```

Die Transkription scheint korrekt zu sein. Wir stellen zwar fest, dass einige Satzzeichen fehlen. Allerdings ist dies allein auf Basis des Audiomitschnitts allgemein schwer zu erkennen, und sie könnten in einem Nachverarbeitungsschritt hinzugefügt werden. Mithilfe von nur einer Handvoll Codezeilen können wir eine hochmoderne Speech-to-Text-Anwendung aufbauen!

Ein Modell für eine neue Sprache zu erstellen, erfordert immer noch ein Mindestmaß an gelabelten Daten, deren Beschaffung, insbesondere bei Sprachen, für die nur wenig Ressourcen verfügbar sind, eine Herausforderung darstellen kann. Kurz nach der Veröffentlichung von wav2vec 2.0 wurde ein Forschungsbeitrag veröffentlicht, der eine Methode namens wav2vec-U beschreibt.[14] In diesem Beitrag wird eine geschickte Kombination aus einem Clustering-Verfahren und einem Training, das an GANs angelehnt ist, verwendet, um ein Speech-to-Text-Modell zu erstellen, das nur voneinander unabhängige, nicht gelabelte Sprach- und Textdaten erfordert. Dieses Vorgehen wird in Abbildung 11-13 im Detail dargestellt. Dabei müssen die Sprach- und Textdaten nicht harmonisiert werden, wodurch das Training von leistungsfähigen Speech-to-Text-Modellen für ein wesentlich größeres Spektrum an Sprachen ermöglicht wird.

14 A. Baevski et al., »Unsupervised Speech Recognition« (*https://arxiv.org/abs/2105.11084*), (2021).

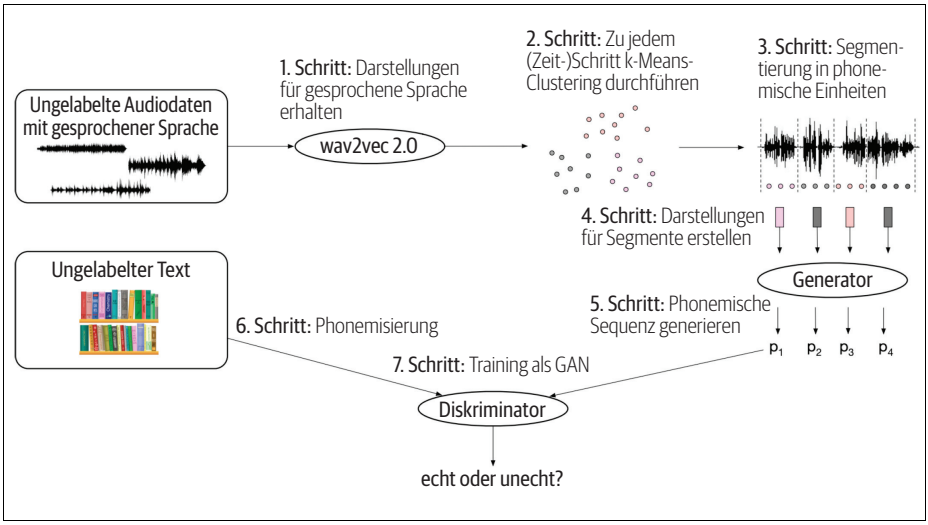


Abbildung 11-13: Das Vorgehen beim Training für wav2vec-U (mit freundlicher Genehmigung von Alexsei Baevski)

Großartig, Transformer können bereits Text »lesen« und Audiosignale »hören« – aber können sie auch »sehen«? Die Antwort lautet ja, und es handelt sich dabei um eine der aktuellsten Forschungsfragen auf diesem Gebiet.

Computer Vision und Text

Bilder und Texte sind zwei Modalitäten, bei denen es ebenfalls naheliegend ist, sie zusammenzuführen, da wir häufig Sprache verwenden, um über den Inhalt von Bildern und Videos zu kommunizieren und darüber zu diskutieren. Neben den Vision-Transformern gibt es mehrere Entwicklungen, die auf die Kombination von visuellen und textuellen Informationen abzielen. Im Folgenden sehen wir uns vier Beispiele für Modelle an, die Bild- und Textdaten miteinander kombinieren: VisualQA, LayoutLM, DALL-E und CLIP.

VQA

In Kapitel 7 haben wir herausgefunden, wie wir Transformer-Modelle verwenden können, um Antworten auf textbasierte Fragen zu erhalten. Dies kann ad hoc geschehen, um Informationen aus Texten zu erhalten, oder »offline«, wenn das Modell zur Beantwortung von Fragen verwendet wird, um strukturierte Informationen aus einer Reihe von Dokumenten zu extrahieren. Es gab zahlreiche Bestrebungen, diesen Ansatz mit Datensätzen wie VQA auf den Bereich der Computer Vision auszuweiten (siehe Abbildung 11-14).[15]

15 Y. Goyal et al., »Making the V in VQA Matter: Elevating the Role of Image Understanding in Visual Question Answering« (*https://arxiv.org/abs/1612.00837*), (2016).

Abbildung 11-14: Ein Beispiel für eine Visual-Question-Answering-Aufgabe aus dem VQA-Datensatz (mit freundlicher Genehmigung von Yash Goyal)

Modelle wie LXMERT und VisualBERT verwenden Bildverarbeitungsmodelle wie ResNets, um Features bzw. Merkmale aus den Bildern zu erhalten. Anschließend setzen sie einen Encoder-basierten Transformer ein, um sie mit natürlichsprachigen Fragen zu kombinieren und eine Antwort vorherzusagen.[16]

LayoutLM

Die Analyse von gescannten Geschäftsdokumenten wie Quittungen, Rechnungen oder Berichten ist ein weiterer Bereich, in dem die Extraktion von Bild- und Layout-Informationen ein nützlicher Weg sein kann, um relevante Textfelder zu erkennen. Hier gelten die Modelle der LayoutLM (*https://oreil.ly/uQc5t*)-Familie als State of the Art. Sie verwenden eine erweiterte Transformer-Architektur, die drei Modalitäten als Eingabe entgegennimmt: Texte, Bilder und Layout-Informationen. Dementsprechend gibt es, wie in Abbildung 11-15 gezeigt, mit den jeweiligen Modalitäten verbundene Embedding-Schichten, einen räumlichen bzw. Spatial-aware Self-Attention-Mechanismus und eine Mischung aus bild- sowie text- und bildbasiertem Pretraining, um die verschiedenen Modalitäten aufeinander abzustimmen. Durch das Pretraining auf Millionen von gescannten Dokumenten können LayoutLM-Modelle ähnlich wie BERT beim NLP auf verschiedene nachgelagerte Aufgaben übertragen werden.

16 H. Tan und M. Bansal, »LXMERT: Learning Cross-Modality Encoder Representations from Transformers« (*https://arxiv.org/abs/1908.07490*), (2019); L.H. Li et al., »VisualBERT: A Simple and Performant Baseline for Vision and Language« (*https://arxiv.org/abs/1908.03557*), (2019).

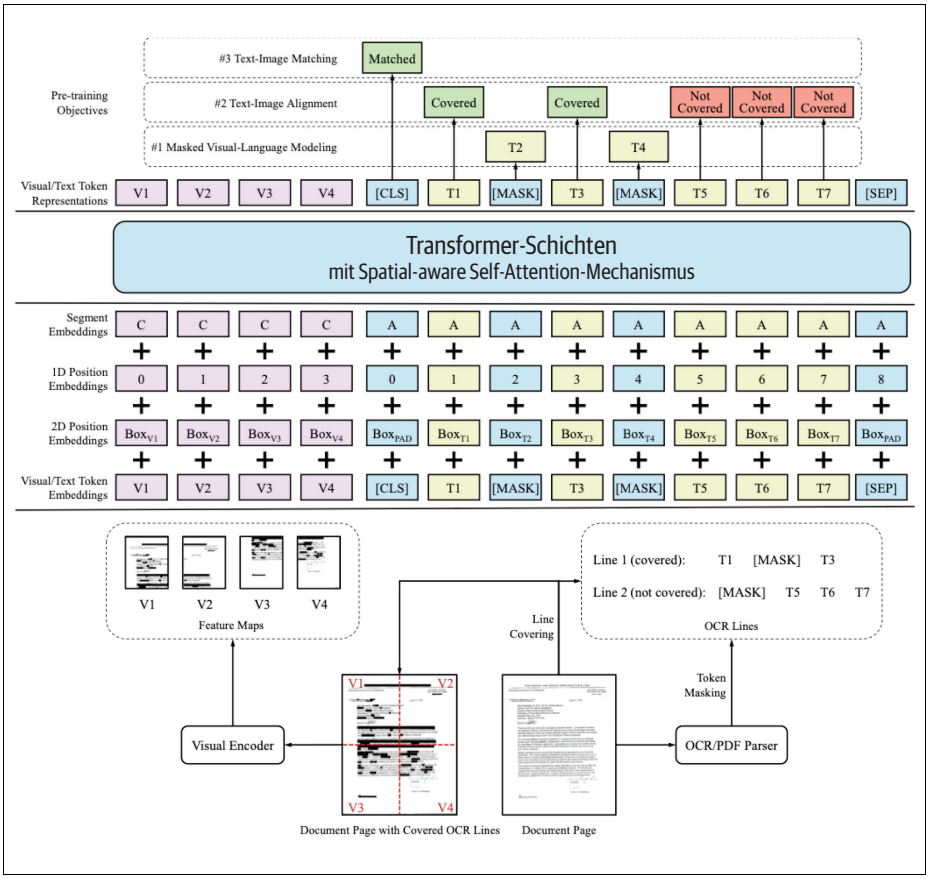


Abbildung 11-15: Die Modellarchitektur von LayoutLMv2 und Strategien für das Pretraining (mit freundlicher Genehmigung von Yang Xu)

DALL-E

Das DALL-E ist ein Modell, das Bild- und Textverarbeitung für *generative* Aufgaben kombiniert. Es verwendet die GPT-Architektur und Autoregressive Modeling, um Bilder auf Basis von Texten zu generieren. Inspiriert von iGPT betrachtet es die Wörter und Pixel als eine Sequenz von Tokens und ist daher in der Lage, ein Bild auf der Grundlage eines vorgegebenen Texts (Prompt) zu erzeugen (siehe Abbildung 11-16).[17]

17 A. Ramesh et al., »Zero-Shot Text-to-Image Generation« (*https://arxiv.org/abs/2102.12092*), (2021).

Vorgegebener Text (Prompt)

an illustration of a baby daikon radish in a tutu walking a dog

KI-generierte Bilder

Abbildung 11-16: Von DALL-E generierte Beispiele (mit freundlicher Genehmigung von Aditya Ramesh)

CLIP

Werfen wir abschließend noch einen Blick auf CLIP[18], das ebenfalls Text- und Bilddaten kombiniert, jedoch für überwachte Aufgaben konzipiert ist. Die Entwickler des Modells haben einen Datensatz mit 400 Millionen Paaren Bildern und den entsprechenden Bildunterschriften erstellt und das Modell mithilfe von Contrastive Learning vortrainiert. Die CLIP-Architektur besteht aus einem Text- und einem Bild-Encoder (beide Transformer), die Embeddings für die Bildunterschriften und Bilder erstellen. Dabei wird zunächst ein Batch bestehend aus Paaren von Bildern und Bildunterschriften gezogen. Das Pretraining-Objective, das auf Contrastive Learning basiert, besteht dann darin, die Ähnlichkeit der Embeddings (gemessen durch das Skalarprodukt) des entsprechenden Paars zu maximieren, während die Ähnlichkeit zu den restlichen Bildern bzw. Bildunterschriften minimiert wird (siehe Abbildung 11-17).

Um das vortrainierte Modell für die Klassifizierung zu verwenden, werden die möglichen Kategorien mithilfe des Text-Encoders eingebettet, ähnlich wie wir es bei der Zero-Shot-Pipeline gehandhabt haben. Dann werden die Embeddings aller Kategorien mit dem Embedding des Bilds verglichen, das wir klassifizieren möchten. Am Ende wird die Kategorie ausgewählt, die die größte Ähnlichkeit aufweist.

18 A. Radford et al., »Learning Transferable Visual Models from Natural Language Supervision« (*https://arxiv.org/abs/2103.00020*), (2021).

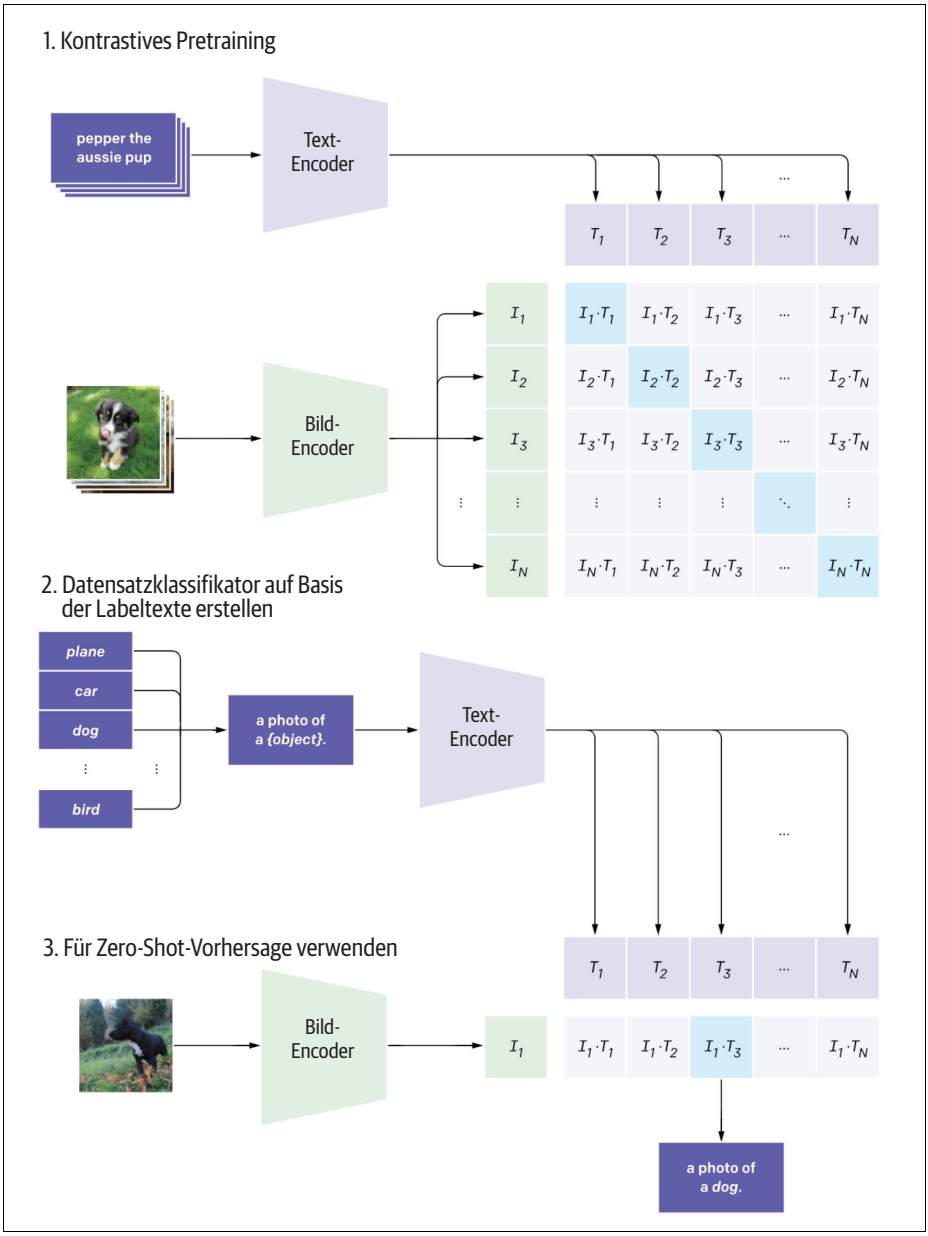


Abbildung 11-17: Die Architektur von CLIP (mit freundlicher Genehmigung von Alec Radford)

Die Leistung von CLIP bei der Zero-Shot-Klassifizierung von Bildern ist bemerkenswert und zudem konkurrenzfähig mit vollständig überwacht trainierten Bildverarbeitungs- bzw. Vision-Modellen. Gleichzeitig ist es flexibler in Bezug auf neue Kategorien. CLIP ist zudem vollständig in die 🤗 Transformers-Bibliothek integriert, sodass wir es gleich ausprobieren können. Für Image-to-Text-Aufgaben instanziieren wir einen *Prozessor*, der aus einem *Feature-Extraktor* (bzw. Merkmalsextraktor) und einem

Tokenizer besteht. Die Aufgabe des Feature-Extractor ist es, das Bild in eine für das Modell geeignete Form zu überführen, während der Tokenizer für die Decodierung der Vorhersagen des Modells in Text zuständig ist:

```
from transformers import CLIPProcessor, CLIPModel

clip_ckpt = "openai/clip-vit-base-patch32"
model = CLIPModel.from_pretrained(clip_ckpt)
processor = CLIPProcessor.from_pretrained(clip_ckpt)
```

Wir benötigen nun ein passendes Bild, um das Ganze auszuprobieren. Was wäre da besser geeignet als ein Bild von Optimus Prime?

```
image = Image.open("images/optimusprime.jpg")
plt.imshow(image)
plt.axis("off")
plt.show()
```

Als Nächstes legen wir die Texte fest, mit denen das Bild verglichen werden soll, und lassen sie durch das Modell laufen:

```
import torch

texts = ["a photo of a transformer", "a photo of a robot", "a photo of agi"]
inputs = processor(text=texts, images=image, return_tensors="pt", padding=True)
with torch.no_grad():
    outputs = model(**inputs)
logits_per_image = outputs.logits_per_image
probs = logits_per_image.softmax(dim=1)
probs
```

```
tensor([[0.9557, 0.0413, 0.0031]])
```

Die Antwort war sogar nahezu richtig (die richtige wäre natürlich das Foto einer übermenschlichen künstlichen Intelligenz, kurz AGI, gewesen). Doch Spaß beiseite, CLIP macht die Bildklassifizierung sehr flexibel, da wir die Kategorien durch Text vorgeben können, anstatt sie in der Modellarchitektur fest zu codieren. Damit ist unser Rundgang durch die Welt multimodaler Transformer-Modelle beendet, doch wir hoffen, wir haben Ihnen Appetit auf mehr gemacht.

Wie geht es weiter?

Nun, das war's. Vielen Dank, dass Sie uns auf der Reise durch die Welt der Transformer begleitet haben! In diesem Buch haben wir uns damit beschäftigt, wie Transformer eine Vielzahl von Aufgaben bewältigen und dabei Ergebnisse erzielen können, die State of the Art sind. In diesem Kapitel haben wir gesehen, wie die aktuelle Generation von Modellen bei der Skalierung an ihre Grenzen stößt und wie Transformer auch in neue Bereiche und Modalitäten vordringen.

Wenn Sie die Konzepte und Fähigkeiten, die Sie in diesem Buch gelernt haben, vertiefen möchten, finden Sie hier einige Ideen, wie Sie weitermachen können:

Nehmen Sie an einer Veranstaltung der Hugging Face Community teil
: Hugging Face veranstaltet kurze Sprints, die auf die Verbesserung der Bibliotheken des Ökosystems abzielen. Diese Veranstaltungen sind eine großartige Möglichkeit, die Community kennenzulernen und einen Eindruck von der Entwicklung von Open-Source-Software zu bekommen. Bisher gab es Sprints, in denen mehr als 600 Datensätze zur 🤗 Datasets-Bibliothek hinzugefügt, mehr als 300 ASR-Modelle in verschiedenen Sprachen feingetunt und Hunderte von Projekten in JAX bzw. Flax implementiert wurden.

Stellen Sie Ihr eigenes Projekt auf die Beine
: Eine sehr effektive Möglichkeit, Ihre Kenntnisse im Bereich des Machine Learning unter Beweis zu stellen, ist, ein Projekt zu erstellen, mit dem Sie ein Problem lösen, das Ihnen nicht aus dem Kopf geht. Sie könnten ein Transformer-Forschungspapier selbst implementieren oder Transformer gar auf eine neue Domäne übertragen.

Erstellen Sie ein Modell für die 🤗 Transformers-Bibliothek
: Wenn Sie etwas Anspruchsvolleres suchen, dann können Sie eine neu veröffentlichte Architektur in die 🤗 Transformers-Bibliothek integrieren. Das ist ein guter Weg, um auch in das Innenleben der Bibliothek einzutauchen. In der 🤗 Dokumentation der Transformers-Bibliothek (*https://oreil.ly/3f4wZ*) finden Sie eine ausführliche Anleitung, die Ihnen den Einstieg erleichtern sollte.

Bloggen Sie darüber, was sie gelernt haben
: Anderen beizubringen, was Sie gelernt haben, ist ein mächtiger Prüfstein für Ihr eigenes Wissen. In gewisser Weise war dieser Punkt einer der treibenden Beweggründe, warum wir dieses Buch verfasst haben. Es gibt großartige Tools, die Ihnen den Einstieg in das technische Bloggen erleichtern. Wir empfehlen Ihnen *fastpages* (*https://oreil.ly/f0L9u*), da Sie für einfach alles, was Sie zum Bloggen benötigen, Jupyter Notebooks verwenden können.

Index

Symbole

1-aus-n-codierte Vektoren *siehe* One-Hot-codierte Vektoren

A

Abdeckungsmaße 356
Abfragevektor 89
absolute Positional-Embeddings 101
abstraktive Zusammenfassungen 176
abstraktives QA 241
Accelerate-Bibliothek
  allgemein 42
  als Teil des Hugging-Face-Ökosystems 38
  Änderungen an der Trainingsschleife 374
  im Vergleich zur Trainer-Klasse 374
  Infrastruktur einrichten 382
  Training-Jobs starten 382
Accelerator()
  is_main_process 377
  prepare() 375
  process_index 377
Accuracy *siehe* Treffergenauigkeit (Maß)
Achitektur von neuronalen Netzen 26
ADAPET-Methode 330
Ähnlichkeitsfunktion 89
AI Dungeon 156
ALBERT-Modell 109, 210
Amazon ASIN 222
Ameisen, Emmanuel 250
Analyse des Pretrainings 383–388
Anwendungsmöglichkeiten von Transformern
  allgemein 33
  automatische Textzusammenfassung 36
  maschinelle Übersetzung 37
  Named Entity Recognition 34
  Question Answering 35
  Textgenerierung 37
  Textklassifizierung 33
Apache Arrow 48, 351
Architektur neuronaler Netze 12
argmax()-Funktion 131, 159, 213, 215, 279
ASR (Automatic Speech Recognition) 406
Attention
  auf Basis des skalierten Skalarprodukts 89–94
  Block-Local- 397
  Dilated- 397
  Encoder-Decoder- 104
  Global- 397
  kausale 85
  linearisierte 398
  Masked-Multi-Head-Self- 104
  Multi-Headed 94
  Self- 28, 395
  skaliertes Skalarprodukt 89
  Sparse 396
»Attention Is All You Need« 12
Attention-Gewichte 87
Attention-Head 95
Attention-Mechanismus 26
Attention-Scores 89
AutoConfig
  allgemein 92
  Default-Werte überschreiben 131, 262, 370
  from_pretrained() 262
Auto-Klassen 64
Automatische Spracherkennung *siehe* ASR (Automatic Speech Recognition)
automatische Textzusammenfassung 175–199
  abstraktive Zusammenfassungen 176

allgemein 175
als Anwendungsmöglichkeit von Transformern 36
Baseline 178
CNN/DailyMail-Datensatz 176
extraktive Zusammenfassungen 176
Modelle trainieren für 192–199
PEGASUS auf dem CNN/DailyMail-Datensatz evaluieren 190
Pipelines für die Textzusammenfassung 177–180
Qualität von generiertem Text evaluieren 182–189
Zusammenfassungen vergleichen 181
Zusammenfassungen von Dialogen erstellen 198
AutoModel
allgemein 64
from_pretrained() 64
output_attentions 96
TensorFlow-Klasse 65
AutoModelForCausalLM
from_config() 370
from_pretrained() 159, 370
gradient_checkpointing 378
AutoModelForMaskedLM 334
AutoModelForQuestionAnswering 212
AutoModelForSeq2SeqLM 191
AutoModelForSequenceClassification
allgemein 73
from_pretrained() 73
TensorFlow-Klasse 76
autoregressive Attention 85
autoregressive Sprachmodelle 158
AutoTokenizer
add_special_tokens 354
as_target_tokenizer() 194
aus dem Cache laden 59
backend_tokenizer.normalizer 358
backend_tokenizer.pre_tokenizer 358
convert_ids_to_tokens() 333
convert_tokens_to_string() 59
decode() 135, 160, 212
from_pretrained() 58
Padding 60, 196
push_to_hub() 366
return_special_tokens_mask 332
return_tensors 190
Trunkierung 61
vocab_size 60

B

balanced_split()-Funktion 298
BALD (Bayesian Active Learning by Disagreement) 338
Band-Attention 397
BART-Modell 113, 179
Baseline-Zusammenfassung 178
Beams 163
Beam-Search-Decodierung 163–167
bedingte Textgenerierung 159
benannte Entitäten *siehe* Eigennamen
BERT-Modell 23, 31, 108, 249, 256, 258, 263, 276, 300, 304
BertViz-Bibliothek 89
Beschleunigung 326
Bias 43, 343
bidirektionale Attention 85
BigBird-Modell 113, 397
BigQuery 349
BigScience 394
BLEU-Maß 183–187
Body (des neuronalen Netzes) 127
Boltzmann-Verteilung 167
BookCorpus-Datensatz 31, 108, 343
BPE (Byte-Pair Encoding) 123, 356, 360
Byte-Level 359

C

C4-Datensatz 112, 343, 354
CamemBERT-Tokenizer 354
Causal Language Modeling 368
CCMatrix-Korpus 326
Character Tokenization *siehe* Tokenisierung
Chaudhary, Amit 314
Class Distribution *siehe* Verteilung der Kategorien
ClassLabel
allgemein 48
int2str() 51, 119
Namen 130
str2int() 252
CLINC150-Datensatz 251
CLIP-Modell 412
Closed-Domain-QA 204
[CLS]-Token
allgemein 59
ID des speziellen Token 61
Rolle bei Textklassifizierung 63
Rolle beim Question Answering 215
vom Tokenizer ausschließen 92

CNN (Convolutional Neural Network) 400
CNN/DailyMail-Datensatz 176, 190
CodeParrot-Modell 341, 382, 387
CodeSearchNet-Datensatz 347
Colab-Notebook 18, 44
Common Crawl Corpus 109, 121
Community-QA 202
compile()-Methode 260
compute()-Funktion 185
compute_accuracy()-Methode 252, 280
compute_loss()-Methode 259
compute_metrics()-Funktion 73, 138, 261
compute_size()-Funktion 253, 280
concatenate_datasets()-Funktion 149
CoNLL-Datensatz 120
Context Size *siehe* Kontextlänge
convert_graph_to_onnx.convert()-Funktion 277
convert_ids_to_tokens()-Methode 59
convert_tokens_to_string()-Methode 59
CSV-Datensatz 49
CTRL-Modell 111
CUAD-Datensatz 205
Cutoff-Wert 172

D

DALL-E-Modell 411
Data-Collator 62, 137, 196, 332
DataFrame, Dataset-Objekt aus 50, 140, 298
Dataloader, implementieren 371–374
Dataset (Objekt)
  Ausgabeformat ändern 50, 54, 67
  DataFrame konvertiert in 50, 140, 298
  Daten mit der map()-Methode verarbeiten 60, 78, 132–135
  einen FAISS-Index erstellen 319
  Features 47
  flatten() 204
  select() 118
  shuffle() 118
Dataset-Cards 40, 354
Datasets-Bibliothek
  alle Datensatzkonfigurationen inspizieren 116
  allgemein 41
  als Teil des Hugging-Face-Ökosystems 39
  Datensätze auf dem Hub auflisten 47
  Datensätze vom Hub herunterladen 47
  lokale Datensätze laden 49
  Maße vom Hub laden 185, 188, 252
  Metadaten inspizieren 47
  Remote-Datensätze laden 49
Datasets-Bibliothek von Hugging Face 47
Daten
  Augmentierung von 313
  Domäne der 204
  Format ändern 50
  Verfügbarkeit von, als Herausforderung bei Transformern 43
Datenparallelität 374
Datensätze
  add_faiss_index()-Funktion 319
  add_faiss_index_from_external_arrays()-Funktion 319
  BookCorpus 31, 108, 343
  C4 112, 343, 354
  CLINC150 251
  CNN/DailyMail 176, 190
  CodeParrot 341, 382, 387
  CodeSearchNet 347
  CommonCrawl 109
  CoNLL 120
  CSV-Format 49
  CUAD 205
  eigene 49
  eigenen Codedatensatz erstellen 346–349
  Emotionen 47
  Erstellung der, als Herausforderung bei der Skalierung 393
  für Aufbau rezensionsbasierter QA-Systeme 203–208
  für die multilinguale Named Entity Recognition 116–120
  GitHub 292–296
  GLUE 47
  große 342–354
  ImageNet 29
  in verschiedenen Formaten laden 49
  JSON-Format 49
  mit Google BigQuery erstellen 347
  MNLI 305
  NQ 208
  OSCAR 354
  PAN-X 116
  SAMSum 192
  SQUAD 47, 207
  SubjQA 203–208
  SUPERB 407
  SuperGLUE 110, 112
  Text- 49

Tokenisierung der gesamten 60
Unvollkommenheiten 78
VQA 409
WikiANN 116
Wikipedia 108
XTREME 116
zum Hugging Face Hub hinzufügen 352
Davison, Joe 304
DDP (DataDistributedParallelism) 381
DeBERTa-Modell 110
Decoder 84, 104
Decoder-basierte Modelle 85
Decoder-basierte Transformer-Modelle 110
Decoder-Schichten 84
Decodierung
Beam-Search-Decodierung 163–167
Greedy-Search 159–163
Decodierungsmethoden 157, 173
deepset 20, 218, 223
Deployment, als Herausforderung bei der Skalierung 394
Dialog (Konversation) 175, 192
Dilated-Attention 397
Diskriminator 110
DistilBERT-Modell 46, 52, 58, 62, 108
Document Store
Definition 218
Dokumente laden mit 222
Kompatibilität mit Retrievern von Haystack 218
Labels laden mit 227
mit Elasticsearch initialisieren 219
Dokumentlänge, als Herausforderung bei Transformern 43
Domain Adaptation 30, 235–239, 331
Domäne, Domain Adaptation 30, 235–239, 331
Downsampling 118, 148
DPR (Dense Passage Retrieval) 231
dynamische Quantisierung 273

E

Effizienz 247–286
allgemein 247
Benchmark-Klasse zur Beurteilung der Performance erstellen 250–255
durch Datensatzgröße 392
Intentionserkennung 248
Knowledge Distillation 255–268
Optimierung der Inferenz mit ONNX/ONNX Runtime 276–282
quantisierte Modelle vergleichen 275
Quantisierung 268–275
Weight Pruning 282–286
eigene Datensätze 49
Eigennamen 34
Eigennamenerkennung *siehe* Multilinguale Named Entity Recognition
Elasticsearch 219, 222
ElasticsearchRetriever.eval()-Methode 226
ELECTRA-Modell 110
EleutherAI 112, 395
ELMO-Modell 30, 87
EM(Exact-Match)-Maß 233
Embeddings
als Nachschlagetabelle verwenden 316–324
Dense- (dichtbesetzte) 92
distilBERT 63
kontextualisierte 87
Positional- 100
Token- 84
Word- 12, 30
Emotionsdatensatz 47
Encoder
allgemein 84, 86
einen Klassifizierungs-Head hinzufügen 103
Feed-Forward-Schicht 98
Layer Normalization integrieren 99
Positional-Embeddings 100
Self-Attention 87–97
Encoder-basierte Modelle 85
Encoder-basierte Transformer-Modelle 108–110
Encoder-Decoder-Attention 104
Encoder-Decoder-basierte Transformer-Modelle 112
Encoder-Decoder-Modell 24, 85
Encoder-Schichten 84
End-to-End 63, 72, 217, 226, 229, 241
englischsprachiger Wikipedia-Datensatz 108
EOS (End-of-Sequence)-Token 84
Exponent 269
extrahieren
Antworten aus einem Text 209–217
letzte verborgene Zustände 66
extraktive Zusammenfassungen 176
extraktives QA 36, 202

F

F.log_softmax()-Funktion 260
F1-Maß(e) 75, 135, 151, 184, 233, 301, 327
FAISS
  Bibliothek 233, 324
  Document Store 219, 233
  effiziente Ähnlichkeitssuche mit 324
  Index, zu einem Dataset-Objekt hinzufügen 319
Fakten, Einschränkungen im Zusammenhang mit Texten 400
FARM-Bibliothek
  Modelle trainieren mit 235–239
  Reader für Question Answering 223
FARMReader
  allgemein 223
  ein Modell laden mit 223
  im Vergleich zur pipeline()-Funktion 223
  predict_on_texts() 224
  train()-Methode 235, 238
Fast Forward QA Series 243
fastdoc-Bibliothek 20
fastpages 415
Feature-Extraktoren 64–72, 413
Feature-Matrix erstellen 68
Feed-Forward Neural Networks 28
Feed-Forward-Schicht 98
Fehleranalyse 77–80, 138–145
Feintuning
  als Schritt beim Vorgehen bei ULMFiT 31
  für mehrere Sprachen gleichzeitig 149–152
  Klassifikatoren 74, 335
  Knowledge Distillation im Rahmen des 256
  mit Keras 76
  PEGASUS 194–198
  Sprachmodelle 332–335
  standardmäßige Transformer (Vanilla) 327
  Transformer 72–81
  XLM-RoBERTa 136–145
Festkommazahlen 269
Few-Shot-Learning 330
Filtern von Rauschen 349
fit()-Methode 76
flatten()-Methode 204
forward()-Funktion 129, 130
Frage-Antwort-Paar 202, 228, 234, 235
Frage-Kontext-Paar 215
Frameworks, Interoperabilität zwischen 65
from_config()-Methode 370
from_pandas()-Methode 298
from_pretrained()-Methode 58, 64, 131, 262, 370

G

generate()-Funktion 160, 166, 168, 172, 191
generative Aufgaben 411
generatives QA 241
Géron, Aurélien 16
gestreamte Datensätze 351
gesunder Menschenverstand, Einschränkungen im Zusammenhang mit Texten 400
get_all_labels_aggregated()-Methode 229
get_dataset_config_names()-Funktion 116, 204
get_dummies()-Funktion 55
get_nearest_examples()-Funktion 319
get_nearest_examples_batch()-Funktion 320
get_preds()-Funktion 308
getsizeof()-Funktion 273
gewichteter Durchschnitt 87
GitHub
  einen Issues-Tagger erstellen 291–300
  Lizenz zur Nutzung der API 346
  Repository 292, 342
  Webseite 290
GitHub Copilot 341, 346
GitHub REST API 292, 346
Gleitkommazahlen 269
Global-Attention 397
GLUE-Datensatz 47, 108
Google Colaboratory (Colab) 18, 44
Googles Meena 156
Google-Suchanfrage 201
GPT-2-Modell 111, 155, 161, 178, 181, 317, 344, 357, 366, 374
GPT-3-Modell 111, 317, 390
GPT-J-Modell 112, 395
GPT-Modell 23, 31, 111, 344
GPT-Neo-Modell 112, 395
Gradient-Accumulation 197, 380
Gradient-Checkpointing 380
Greedy-Search-Decodierung 159–163
Grid Dynamics 244
Ground Truth 165, 182, 196, 227, 233, 252, 256, 261, 279
Gugger, Sylvain 16

H

Hardwareanforderungen 18
Hash-Symbole (#) 35
Haystack-Bibliothek
  Document Store initialisieren 219
  gesamte Pipeline evaluieren 240
  Pipeline initialisieren 225
  Reader evaluieren 233
  Reader intitialisieren 223
  Retriever evaluieren 226–233
  Retriever initialisieren 222
  Retriever-Reader-Architektur 217
  Tutorial 233, 244
  Webseite 218
  zum Aufbau von QA-Pipelines verwenden 217–226
Head (des neuronalen Netzes) 127
head_view()-Funktion 96
Hidden State *siehe* verborgener Zustand
Hinton, Geoff 256
»How We Scaled Bert to Serve 1+ Billion Daily Requests on CPUs« 247
Howard, Jeremy 16
Hub *siehe* Hugging Face Hub
Hugging Face
  Accelerate-Bibliothek 42
  Datasets-Bibliothek 41
  Ökosystem 38
  Tokenizers-Bibliothek 41
  Veranstaltungen der Community 415
Hugging Face Hub
  alle Datensätze auflisten 47
  allgemein 39
  Datensätze hinzufügen zum 352
  einloggen 73
  Modelle speichern auf dem 80
  QA-Modelle auswählen auf 204
  selbst erstellte Tokenizer speichern auf 366
  Widgets 152
Human Reporting Bias, Einschränkungen im Zusammenhang mit Texten 400
Hyperparameter, mit Optuna finden 265
hyperparameter_search()-Methode 266

I

iGPT-Modell 400
ImageNet-Datensatz 29
Imbalanced-learn-Bibliothek 52
IMDb 30
In-Context-Learning 330
Inference API 82, 394
Inference-Widget 40
InferKit 156
Informationsengpass 26
Infrastruktur, als Herausforderung bei der Skalierung 393
initialisieren
  Document Store 219
  Modelle 370
  Reader 223
  Retriever 222
init_weights()-Methode 129
int2str()-Methode 51
Intentionserkennung 248
Interoperabilität, zwischen Frameworks 65
Intransparenz, als Herausforderung bei Transformern 43
ISO-639-1-Sprachcodes 117
Issues-Registerkarte 291
Issues-Tagger erstellen 291–300
iter()-Methode 372
iterative_train_test_split()-Funktion 298, 299

J

JAX-Bibliothek 32
Jira 290
JSON-Datensatz 49
Jupyter Notebook 74, 342, 408

K

Kaggle Notebooks 18
Karpathy, Andrej 25, 51, 105
kausale Attention 85
kausale Sprachmodelle 158
Keras-Bibliothek 76, 260
Kernel-Function 398
Key *siehe* Schlüsselvektor
Kite 341
KL-(Kullback-Leibler)-Divergenz 257
Klassifikatoren feintunen 74, 335
Klassifizierung von Tokens, selbst definierte Modelle erstellen zur 128–132
Klassifizierungs-Head 103
kleinere Trainingsdatensätze (Slices), erstellen 299

Knowledge Distillation
allgemein 255
Ausgangspunkt für das Schüler-Modell wählen 260–265
eine selbst definierte Trainer-Klasse erstellen 259
Hyperparameter mit Optuna finden 265
im Rahmen des Feintunings 256
im Rahmen des Pretrainings 258
Modell im Vergleich 267
Konstantenfaltung 277
Kontext 35
Kontextlänge 52, 60, 113, 366
Kontextmanager 196
kontextualisierte Embeddings 87
Korpus 31, 108, 121, 326, 343–346, 354
Kosten, als Herausforderung bei der Skalierung 393
Kreuzentropieverlust 134, 257, 337, 391

L

Labels 289–339
allgemein 289
Ansätze, wenn keine gelabelten Daten vorliegen 303–312
Ansätze, wenn nur wenige gelabelte Daten zu Verfügung stehen 313–331
einen GitHub-Issues-Tagger erstellen 291–300
Nutzen aus ungelabelten Daten ziehen 331–339
unzutreffende 78
Laden
eigene Datensätze 49
selbst definierte Modelle 130
Tokenizer 58
vortrainierte Modelle 73
Latenz, als Benchmark für die Performance 250
Layer Normalization 99
LayoutLM-Modell 410
LCS (Longest Common Subsequence) 188
Learning Rate Warm-up 99
Leistung
Beziehung zwischen Skalierung und 391
Maße festlegen 73
letzter verborgener Zustand 25, 66
Libraries.io 347
linearisierte Attention 398
list_datasets()-Funktion 47
load_dataset()-Funktion
Download-Konfiguration 350
eine einzelne Kategorie laden 204
einzelne Konfiguration laden 116
spezifische Version laden 176
Streaming 351
logarithmierte Wahrscheinlichkeiten 164
Logits 103, 132, 157, 159, 164, 167, 212–215, 223, 256, 260, 279, 328
Longformer-Modell 397
Long-Form-QA 202
LSTM (Long-Short Term Memory)-Netzwerke 23
Lucene 222
LXMERT-Modell 410

M

M2M100-Modell 113, 314, 326
MAD-X-Framework 153
Magnitude Pruning 284
Mantisse 269
mAP (mean Average Precision) 226
map()-Methode 60, 67, 78, 132, 300, 308, 315
maschinelle Übersetzung, als Anwendungsmöglichkeit von Transformern 37
Masked-Multi-Head-Self-Attention 104
Maskierungsmatrix 104, 283
Maße
add()-Funktion 185
add_batch()-Funktion 185
BLEU 183–187
compute() 185
Exact Match 233
F1-Maß 75, 135, 151, 184, 233, 301, 327
logarithmierte Wahrscheinlichkeiten 163
mean Average Precision 226
Perplexität 378
Precision 135, 183
Recall 135, 184, 187, 226
ROUGE 187
SacreBLEU 185
Treffergenauigkeit 73, 199, 252
Matrizen 93, 270, 283
maximale Kontextlänge 52
Mean Pooling 318
Meena (Google) 156
Memory Mapping 41, 350
Metriken *siehe* Maße
minGPT-Modell 105

MiniLM-Modell 210
MLM (Masked Language Modeling) 31, 108, 368
MNLI-Datensatz 305
Modalität, Einschränkungen im Zusammenhang mit Texten 400
Model Hub 13
Model-Cards 40
model_init()-Methode 137
Modelle
  ALBERT 109, 210
  Arten von 260
  BART 113, 179
  BERT 23, 31, 108, 249, 256, 258, 263, 276, 300, 304
  BigBird 113, 397
  CamemBERT 354
  CLIP 412
  CodeParrot 341, 382, 387
  CTRL 111
  DALL-E 411
  DeBERTa 110
  DistilBERT 46, 52, 58, 62, 108
  DPR 231
  ELECTRA 110
  ELMO 30, 87
  Evaluierung von, als Herausforderung bei der Skalierung 394
  GPT 23, 31, 111, 344
  GPT-2 111, 317, 344, 357, 366, 374
  GPT-3 111, 317, 390
  GPT-J 112, 395
  GPT-Neo 112, 395
  iGPT 400
  intialisieren 370
  LayoutLM 410
  Longformer 397
  LSTM 23
  LXMERT 410
  M2M100 113, 314, 326
  Meena 156
  minGPT 105
  miniLM 210
  Modellleistung, als Benchmark für die Performance 250
  naiver Bayes-Klassifikator 300–302
  PEGASUS 180, 190, 193, 194–198
  RAG 241
  Reformer 398
  ResNet 28, 410
  RNN 24
  RoBERTa 108, 210
  speichern 80
  T5 112, 178, 354
  TAPAS 202, 403
  teilen 80
  Training 74
  ULMFiT 23, 30
  VisualBERT 410
  Wav2Vec2 406
  XLM 109
  XLM-RoBERTa 65, 109, 121, 136–145, 210
Modelle für das Leseverstehen 202
Modellgewichte 40
Modell-Widgets, interagieren mit 152
Movement Pruning 285
Multi-Headed Attention 95
Multilabel-Textklassifizierungsproblem 291
Multilinguale Named Entity Recognition 115–152
  allgemein 115
  Aufbau der Model-Klasse der Transformers-Bibliothek 127
  Body 127
  Datensatz 116–120
  Fehleranalyse 138–145
  Feintuning für mehrere Sprachen gleichzeitig 149–152
  Feintuning von XLM-RoBERTa 136–145
  Heads 127
  mit Modell-Widgets interagieren 152
  Multilinguale Transformer 120
  Qualitätsmaße 135
  selbst definierte Modelle laden 130–132
  selbst definierte Modelle zur Klassifizierung von Tokens erstellen 128–132
  SentencePiece-Tokenizer 124
  sprachenübergreifender Transfer 146–152
  Texte für NER tokenisieren 132–135
  Tokenisierung 122–125
  Tokenizer-Pipeline 122
  Transformer für 125
  XLM-RoBERTa 121
  Zero-Shot-Transfer 147
Multilinguale Transformer 120
Multimodale Transformer 406–409

N

Nachschlagetabelle, Embeddings verwenden als 316–324
Nachverarbeitung 124
naiven Bayes-Klassifikator als Baseline implementieren 300–302
naiver Bayes-Klassifikator 300
NER (Named Entity Recognition)
  Anpassung der Vorhersagen 135
  Aufgabe 121, 138, 146
  Texte tokenisieren für 132–135
  Transformer für 125
Neural Networks Block Movement Pruning (Bibliothek) 286
neuronale Feed-Forward-Netze *siehe* Feed-Forward Neural Networks
Neuronale Konvolutionsnetze *siehe* CNN (Convolutional Neural Network)
neuronale Netzwerkarchitektur 23
n-Gramme 187
n-Gramme-Penalisierung 166
NLI (Natural Language Inference) 305–312
NLP (Natural Language Processing), Transfer Learning im 28–33
NlpAug-Bibliothek 314
NLU (Natural Language Understanding) 108
nonlocal-Schlüsselwort 366
Normalisierung 99, 123
notebook_login()-Funktion 352
NQ-Datensatz 208
NSP (Next Sentence Prediction) 108
Nucleus-Sampling 169–172
Nullstelle 270
Numericalization 54

O

objective()-Funktion 266
Offset-Tracking 358
one_hot()-Funktion 56
One-Hot-codierte Vektoren 55, 63
One-Hot-Codierung 56, 92
Onlinekurs, von Hugging Face 16
Onlinetexte 115, 345, 393
ONNX/ONNX Runtime, Optimierung der Inferenz mit 276–282
ONNX-ML 276
Open Source 290, 346, 356, 395, 415
OpenAI 30, 111, 155, 162, 317, 394
Open-Domain-QA 204
OpenMP 278
Operator Sets 278
Operator-Fusion 277
»Optimal-Brain-Surgeon«-Diskussionspapier 284
Optuna, Hyperparameter finden mit 265
ORT (ONNX Runtime) 281
OSCAR-Korpus 354
Out-of-Scope-Anfragen 248

P

pandas.Series.explode()-Funktion 140
PAN-X-Datensatz 116, 144
Paperspace Gradient Notebooks 18
Path.stat()-Funktion 253
PEGASUS-Modell
  allgemein 180
  auf CNN/DailyMail-Datensatz evaluieren 190
  auf SAMSum-Datensatz evaluieren 193
  Feintuning 194–198
perf_counter()-Funktion 253
Performance, Benchmark-Klasse erstellen 250–255
permutationsäquivariant 100
pip 13
Pipeline
  aufbauen mithilfe von Haystack 217–226
  für die Textzusammenfassung 177–180
  Tokenizer 122, 355
  Transformers-Bibliothek 33
pipeline()-Funktion
  aggregation_strategy 34
  allgemein 33
  automatische Textzusammenfassung 36
  ein Modell vom Hub verwenden 36
  maschinelle Übersetzung 37
  Named Entity Recognition 34
  Question Answering 35
  Textgenerierung 37
  Textklassifizierung 33
plot_metrics()-Funktion 268
Pooling 318
Popen()-Funktion 220
Positional-Embeddings 84, 100
positionsbezogene vorwärtsgerichtete Schicht *siehe* Position-Wise Feed-Forward-Schicht
Position-Wise Feed-Forward-Schicht 98
Post Layer Normalization 99
Potenzgesetze 392

Pre Layer Normalization 99
predict()-Methode 75, 146
prepare()-Funktion 375
Pretokenization 123
Pretraining
  allgemein 29
  als Schritt beim Vorgehen bei ULMFiT 30
  Knowledge Distillation im Rahmen des 258
  Objectives beim 367
Prompts 330
Proportion of Continued Words 356
Pseudo-Labels 339
push_to_hub()-Methode 80, 366
Python, Tokenizer für 357–362
PyTorch-Bibliothek
  allgemein 32
  Hub 41
  Interoperabilität mit 65
  Klassen und Methoden 91
  tril()-Funktion 104
  Webseite 16

Q

QA (Question Answering) 201–243
  abstraktives 241
  allgemein 201
  als Anwendungsmöglichkeit von Transformern 35
  Antworten aus einem Text extrahieren 209–217
  Closed-Domain-QA 204
  Community 202
  Datensatz 203–208
  Domain Adaptation 235–239
  extraktives 36, 202
  generatives 241
  gesamte Pipeline evaluieren 240
  lange Passagen beim 215
  Long-Form- 202
  Open-Domain-QA 204
  Pipeline verbessern 226–235
  Pipelines mithilfe von Haystack aufbauen 217–226
  RAG (Retrieval-Augmented Generation) 241
  Reader evaluieren 233–235
  Retriever evaluieren 226–233
  rezensionsbasierte QA-Systeme aufbauen 202–226
  Span-Classification-Aufgabe 209
  SQuAD-Datensatz 207
  Tabellen (QA) 403
  Text tokenisieren für 211–215
Qualität, von generiertem Text evaluieren 182–189
Qualitätsmaße 73, 135, 151, 183, 184, 187, 193, 226
Quantisierung
  allgemein 268–273
  dynamische 273
  Modelle vergleichen 275
  Quantization-aware Training 274
  statische 274
  Strategien für 273
quantize_dynamic()-Funktion 275, 281
quantize_per_tensor()-Funktion 271
Query *siehe* Abfragevektor

R

Radixpunkt 269
RAG (Retrieval-Augmented Generation) 241
RAG-Sequence-Modelle 242
RAG-Token-Modelle 242
Random-Attention 397
Rauschen filtern 349
Reader
  als Komponente der Retriever-Reader-Architektur 218
  evaluieren 233–235
  initialisieren 223
README-Cards 354
Recall 226
recv-Schlüsselwort 364
Reformer-Modell 398
rekurrente neuronale Netze *siehe* RNNs (Recurrent Neural Networks)
Relative Positionsdarstellungen 102
ResNet-Modell 28, 410
retrieve()-Methode 222
Retriever
  als Komponente der Retriever-Reader-Architektur 217
  evaluieren 226–233
  initialisieren 222
Retriever-Reader-Architektur 217
rezensionsbasierte QA-Systeme aufbauen 202–226
RNNs (Recurrent Neural Networks) 24
RoBERTa-Modell 108, 210

ROUGE-Maß 187
Rückübersetzung 313
run()-Methode 225, 227
run_benchmark()-Methode 251
Rust (Programmiersprache) 41, 358

S

SacreBLEU-Maß 185
sample()-Methode 205
Sampling-Verfahren 167–172
SAMSum-Datensatz 192
Samsung 192
Schichtnormalisierung *siehe* Layer Normalization
Schlüssel/Wert-Paar 253
Schlüsselvektor 89
Scikit-learn-Format 68
Scikit-multilearn-Bibliothek 297
selbst definierte Modelle 128–132
select()-Methode 118
Self-Attention 28, 395
Self-Attention-Schicht
  allgemein 87
  Attention auf Basis des skalierten Skalarprodukts 89–94
  Multi-Headed Attention 94–97
SentencePiece-Tokenizer 122, 124
Sentimentanalyse 33
sent_tokenize()-Funktion 180
[SEP]-Token 59, 61, 92, 97, 122, 124, 212, 216, 332
seq2seq (Sequence-to-Sequence) 25, 369
seqeval-Bibliothek 135
Sequence-Klasse 119
set_format()-Methode 50
setup_logging()-Methode 376
shuffle()-Methode 118, 351
Signifikant 269
Silberstandard 144
Skalarprodukt 89–94, 105, 398, 412
Skalieren 111
Skalierung von Transformern
  allgemein 389
  Herausforderungen bei 393
  linearisierte Attention 398
  Self-Attention-Mechanismus 395
  Skalierungsgesetze 391
  Sparse-Attention 396
Skalierungsgesetze 391
Skip-Verbindungen 99
Softmax-Funktion 89, 93, 105, 110, 157, 160, 167, 214, 223, 256, 260
Softwareanforderungen 18
SoundFile-Bibliothek 407
Span-Classification-Aufgabe 209
Sparse-Attention, Skalierung und 396
Speech-to-Text 406–409
Speicher, als Benchmark für die Performance 250
speichern
  Modelle 80
  selbst erstellte Tokenizer auf dem Hugging Face Hub 366
split()-Funktion 57
Sprache, als Herausforderung bei Transformern 43
sprachenübergreifender Transfer
  allgemein 146
  Feintuning für mehrere Sprachen gleichzeitig 149–152
  Zero-Shot-Transfer 147
Sprache-zu-Text *siehe* Speech-to-Text
Sprachmodelle feintunen 332–335
SQuAD (Stanford Question Answering Dataset) 47, 207, 234, 238
Stack Overflow 202, 250
Stammbaum von Transformern *siehe* Transformer im Überblick
standardmäßige (Vanilla-)Transformer feintunen 327
State of the Art 247
statische Quantisierung 274
str2int()-Methode 252
Subjektivität 204
SubjQA-Datensatz 204–208
Sublayer *siehe* Teilschichten
Subword Fertility 356
Subword Tokenization *siehe* Tokenisierung auf der Ebene von Teilwörtern (Subword Tokenization)
Summarization *siehe* automatische Textzusammenfassung
SUPERB-Datensatz 407
SuperGLUE-Datensatz 110, 112
Sutton, Richard 389

T

T5-Modell 112, 178, 354
Tabellen (QA) 403
TabNine 341
TAPAS-Modell 202, 403

Task-agnostic Distillation 261
Teilen von Modellen 80
Teilhypothesen 163
Teilschichten 86, 98, 104
Tensor.masked_fill()-Funktion 104
Tensor.storage()-Funktion 273
TensorBoard 376
Tensoren
  Batch-Matrix-Matrixprodukt 93
  Darstellung in Form von Ganzzahlen 271
  Elemente maskieren 104
  in TensorFlow konvertieren 76
  in Tokenizer zurückgeben 66
  One-Hot-Codierung erstellen 55
  Quantisierung 270
  Speichergröße 273
TensorFlow
  allgemein 32
  Feintuning von Modellen mit der Keras-API 76
  Hub 41
  Klassen und Methoden 91
  Webseite 16
test_step()-Methode 260
Text
  Antworten extrahieren aus einem 209–217
  Computer Vision und 409–415
  jenseits von 399–415
  tokenisieren für QA 211–215
Text Entailment 305
TextAttack-Bibliothek 314
Textdatensatz 49
Textgenerierung 155–173
  allgemein 155
  als Anwendungsmöglichkeit von Transformern 37
  Beam-Search-Decodierung 163–167
  Decodierungsmethode auswählen 173
  Greedy-Search-Decodierung 159
  Herausforderungen bei 157
  Sampling-Verfahren 167–172
Textklassifizierung 45–82
  allgemein 45
  als Anwendungsmöglichkeit von Transformern 33
  DataFrames 50
  Datasets-Bibliothek 46–50
  Datensätze 46–50
  Feintuning von Transformern 72–81
  Länge von Tweets 52
  Textklassifikatoren trainieren 62–72
  Tokenisierung auf der Ebene von Teilwörtern (Subword Tokenization) 58
  Tokenisierung auf der Ebene von Wörtern (Word Tokenization) 57
  Tokenisierung auf der Ebene von Zeichen (Character Tokenization) 54
  Tokenisierung des gesamten Datensatzes 60
  Transformer-Modelle als Feature-Extraktoren 64–72
  Verteilung der Kategorien 51
TF-IDF(Term Frequency-Inverse Document Frequency)-Algorithmus 222
tiefe neuronale Netze 283
time_pipeline()-Funktion 253
TimeSformer-Modell 403
TLM (Translation Language Modeling) 109
Token Perturbations 313
Token-Embeddings 84, 87
Tokenisierung
  allgemein 54, 122
  auf der Ebene von Teilwörtern (Subword Tokenization) 58
  auf der Ebene von Wörtern (Word Tokenization) 57
  auf der Ebene von Zeichen (Character Tokenization) 54
  des gesamten Datensatzes 60
  Texte für NER 132–135
  von Text für QA 211–215
Tokenizer
  allgemein 35
  auf dem Hugging Face Hub speichern 366
  erstellen 354–366
  für Python 357–362
  Leistung beurteilen 356
  Training 362–366
Tokenizer-Modell 123, 355
Tokenizer-Pipeline 122
Tokenizers-Bibliothek
  allgemein 41
  als Teil des Hugging-Face-Ökosystems 38
  Auto-Klasse 64
  Text tokenisieren 64
  Tokenizer vom Hub laden 64
Top-k-Sampling 169–172

Top-p-Sampling 169–172
torch.bmm()-Funktion 93
torch.save()-Funktion 252, 280
torch.tril()-Funktion 104
to_tf_dataset()-Methode 76
train()-Methode 235, 262
Trainer
  allgemein 18
  eine selbst definierte Trainer-Klasse erstellen 259
  einen Data-Collator verwenden 137, 196, 333
  Feintuning von Modellen 75
  hyperparameter_search() 266
  Knowledge Distillation 259
  Logging-Historie 334
  Maße festlegen 73, 137, 261, 328
  model_init() 137
  push_to_hub() 80
  selbst definierten Verlust berechnen 259
  Vorhersagen erstellen 75
Training
  Modelle 74
  Modelle für Zusammenfassungen 192–199
  Textklassifikatoren 62
  Tokenizer 362–366
TrainingArguments
  allgemein 74
  eine selbst definierte TrainingArguments-Klasse erstellen 259
  Gradient-Accumulation 197
  label_names 261
  save_steps 136
Trainingsdatensätze 68, 297
  visualisieren 68
Trainingslauf 382
Trainingsschleife einrichten 374–382
train_new_from_iterator()-Methode 362
train_on_subset()-Funktion 150
train_step()-Methode 260
TransCoder-Modell 347
Transfer Learning
  im NLP 28–33
  im Vergleich zu Supervised Learning 28
  in der Computer Vision 28
  Weight Pruning und 282
Transformer
  allgemein 12
  als Feature-Extraktoren 64–72
  BERT 31
  Effizienz von 247–286
  feingetunt auf SQuAD 210
  Feintuning 72–81
  für Named Entity Recognition 125
  GPT 31
  größte Herausforderungen mit 42
  im Überblick 106
  multilinguale 120
  skalieren *siehe* Skalierung von Transformern
  Training *siehe* Transformer-Modelle von Grund auf trainieren
Transformer-Architektur 83–114
  allgemein 23, 83–86
  Decoder 104
  Decoder-basierte Transformer-Modelle 110
  einen Klassifizierungs-Head hinzufügen 103
  Encoder 86–103
  Encoder-basierte Transformer-Modelle 108–110
  Encoder-Decoder-Attention 104
  Encoder-Decoder-basierte Transformer-Modelle 112
  Feed-Forward-Schicht 98
  Layer Normalization integrieren 99
  Positional-Embeddings 100
  Self-Attention 87–97
  Stammbaum 106
Transformer-Modelle von Grund auf trainieren 341–388
  allgemein 341
  Dataloader implementieren 371–374
  Datensätze zum Hugging Face Hub hinzufügen 352
  eigenen Codedatensatz erstellen 346–349
  Ergebnisse und Analyse 383–388
  Erstellung eines Tokenizers 354–366
  große Datensätze 342–354
  Herausforderungen beim Aufbau eines großen Korpus 343–346
  Leistung eines Tokenizers beurteilen 356
  Modelle intialisieren 370
  Pretraining-Objectives 367
  selbst erstellte Tokenizer auf dem Hugging Face Hub speichern 366
  Tokenizer für Python 357–362
  Tokenizer trainieren 362–366
  Tokenizer-Modell 355

Trainingslauf 382
Trainingsschleife einrichten 374–382
Transformers-Bibliothek
allgemein 31
als Teil des Hugging-Face-Ökosystems 38
Auto-Klassen 59
Feintuning von Modellen mit 72–76
Modelle auf dem Hub speichern 80
Modelle vom Hub laden 64
Pipelines 33–38
Tokenizer vom Hub laden 58
TransformersReader 223
Translation *siehe* maschinelle Übersetzung
Treffergenauigkeit (Maß) 199, 252

U

UDA (Unsupervised Data Augmentation) 290, 337
ULMFiT (Universal Language Model Fine-Tuning) 23, 30
UMAP-Algorithmus 68
ungelabelte Daten, Nutzen ziehen aus 331–339
Unicode-Normalisierung 123, 359
Unigram 356
UST (Uncertainty-Aware Self-Training) 290, 338

V

Value *siehe* Wertvektor
verborgener Zustand 24, 63
Verteilung der Kategorien 51
Vision 400–403, 409–415
VisualBERT-Modell 410
Voreingenommenheit *siehe* Bias
vortrainierte Modelle 64, 73
Vorzeichen 269
VQA-Datensatz 409

W

Wahrscheinlichkeit für das nächste Token 166
Wav2Vec2-Modelle 406
Weight Pruning
allgemein 282
Methoden 283–286
Sparsität tiefer neuronaler Netze 283
Weights & Biases 376
Wertvektor 89
WikiANN-Datensatz 116
Word Tokenization *siehe* Tokenisierung auf der Ebene von Wörtern (Word Tokenization)
word_ids()-Funktion 133
WordPiece 58, 122, 356
Worteinbettungen *siehe* Embeddings
Write With Transformer 156
write_documents()-Methode 221

X

XLM-Modell 109
XLM-RoBERTa-Modell 65, 109, 121, 136–145, 210
XTREME-Benchmark 116

Z

Zeitschritt 26, 87, 104, 159, 163, 167
Zero-Shot Cross-Lingual Transfer 115
Zero-Shot-Klassifizierung 304–312
Zero-Shot-Learning 116
Zero-Shot-Transfer 116, 147
Zusammenfassungen von Dialogen, erstellen 198
Zwischendarstellung 276

Über die Autoren

Lewis Tunstall ist Machine Learning Engineer bei Hugging Face. Er hat Machine-Learning-Anwendungen für Start-ups und Unternehmen in den Bereichen NLP, topologische Datenanalyse und Zeitreihen entwickelt. Lewis hat in theoretischer Physik promoviert und war in Australien, den USA und der Schweiz in der Forschung tätig. Der Schwerpunkt seiner Arbeit liegt derzeit auf der Entwicklung von Tools für die NLP-Community und darauf, Menschen zu schulen, diese effektiv zu nutzen.

Leandro von Werra ist Machine Learning Engineer im Open-Source-Team von Hugging Face. Er verfügt über viele Jahre Erfahrung darin, NLP-Projekte in die Produktion zu bringen, indem er über den gesamten Machine-Learning-Stack hinweg arbeitet. Er ist der Entwickler der beliebten Python-Bibliothek TRL, die Transformer mit Reinforcement Learning kombiniert.

Thomas Wolf ist Chief Science Officer und Mitbegründer von Hugging Face. Sein Team hat sich der Aufgabe verschrieben, die NLP-Forschung voranzutreiben und sie weiter zu demokratisieren. Vor der Gründung von Hugging Face erwarb Thomas Wolf einen Doktortitel in Physik und später einen Abschluss in Jura. Er hat als Physiker in der Forschung gearbeitet und war europäischer Patentanwalt.

Kolophon

Der Vogel auf dem Cover von *Natural Language Processing mit Transformern* ist ein Allfarblori (*Trichoglossus haematodus*), ein Verwandter von Sittichen und Papageien. Er ist in Indonesien, Neukaledonien, Papua-Neuguinea, auf den Salomon-Inseln und Vanuatu beheimatet.

Das Gefieder des Allfarbloris fügt sich in seine farbenfrohe tropische und subtropische Umgebung ein. Der grüne Nacken geht in einen gelben Kragen über, der sich unter dem tief dunkelblauen Kopf befindet. Der Schnabel ist orangerot, die Iris des Männchens ist leuchtend rot, während sie beim Weibchen orangerot ist. Die Brustfedern sind rot mit einer blauschwarzen Bänderung, der Bauch ist grün mit gelber Bänderung. Allfarbloris haben einen der längsten, spitz zulaufenden Schwänze unter den sieben Loris-Arten, er ist oben grün und unten grün-gelb gestreift. Diese Vögel werden 25 bis 30 Zentimeter lang und wiegen 109 bis 137 Gramm.

Allfarbloris leben monogam mit einem Partner und legen jeweils zwei mattweiße Eier. Als Höhlenbrüter bauen sie ihre Nester in über 30 Metern Höhe in hohlen Ästen oder Aushöhlungen von Eukalyptusbäumen. In freier Wildbahn leben sie 15 bis 20 Jahre.

Diese Art leidet unter dem Verlust ihres Lebensraums und dem Fang für den Heimtierhandel, die Population weist eine sinkende Tendenz im Bestand auf. Viele der Tiere auf den O'Reilly-Covern sind vom Aussterben bedroht, doch jedes einzelne von ihnen ist für den Erhalt unserer Erde wichtig.

Die Illustration auf dem Umschlag dieses Buchs stammt von Karen Montgomery, die hierfür einen Stich aus der *English Cyclopedia* verwendet hat. Der Umschlag der deutschen Ausgabe wurde von Karen Montgomery und Michael Oréal entworfen. Auf dem Cover verwenden wir die Schriften Gilroy Semibold und Guardian Sans, als Textschrift die Linotype Birka, die Überschriftenschrift ist die Adobe Myriad Condensed und die Nichtproportionalschrift für Codes ist LucasFonts TheSans Mono Condensed.