CHAPTER 1

Hello Transformers

In 2017, researchers at Google published a paper that proposed a novel neural network architecture for sequence modeling.[1] 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 ULMFiT showed that training long short-term memory (LSTM) networks on a very large and diverse corpus could produce state-of-the-art text classifiers with little labeled data.[2]

These advances were the catalysts for two of today's most well-known transformers: the Generative Pretrained Transformer (GPT)[3] and Bidirectional Encoder Representations from Transformers (BERT).[4] By combining the Transformer architecture with unsupervised learning, 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 zoo of transformer models has emerged; a timeline of the most prominent entries is shown in Figure 1-1.

1 A. Vaswani et al., "Attention Is All You Need", (2017). This title was so catchy that no less than 50 follow-up papers have included "all you need" in their titles!

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

3 A. Radford et al., "Improving Language Understanding by Generative Pre-Training", (2018).

4 J. Devlin et al., "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding", (2018).

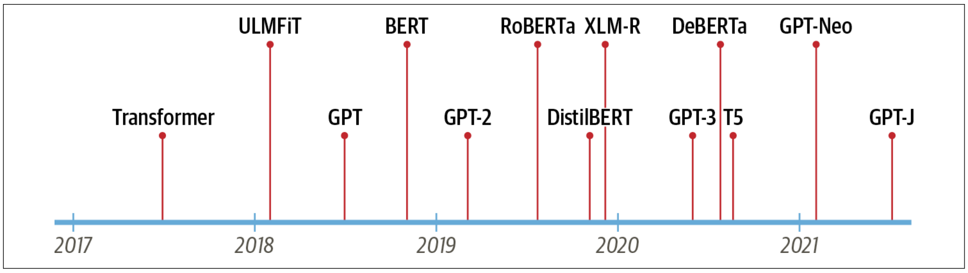


Figure 1-1. The transformers timeline

But we're getting ahead of ourselves. To understand what is novel about transformers, we first need to explain:

- The encoder-decoder framework
- Attention mechanisms
- Transfer learning

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 by exploring the encoder-decoder framework and the architectures that preceded the rise of transformers.

The Encoder-Decoder Framework

Prior to transformers, recurrent architectures such as LSTMs were the state of the art in NLP. These architectures contain a feedback loop in the network connections that allows information to propagate from one step to another, making them ideal for modeling sequential data like text. As illustrated on the left side of Figure 1-2, an RNN receives some input (which could be a word or character), feeds it through the network, and outputs a vector 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. This can be more clearly seen if we "unroll" the loop as shown on the right side of Figure 1-2: the RNN passes information about its state at each step 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.

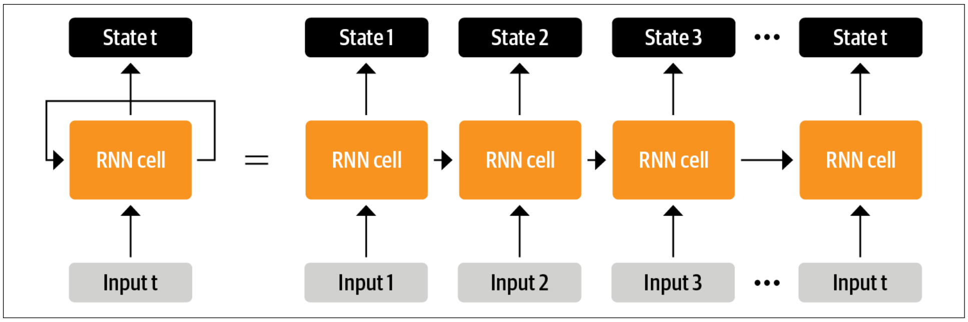


Figure 1-2. Unrolling an RNN in time

These architectures were (and continue to be) widely used for NLP tasks, speech processing, and time series. 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,[5] which is well suited for situations where the input and output are both sequences of arbitrary length. 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 can model sequences. This is illustrated for a pair of RNNs in Figure 1-3, where the English sentence "Transformers are great!" is encoded as a hidden state vector that is then decoded to produce the German translation "Transformer sind grossartig!" The input words are fed sequentially through the encoder and the output words are generated one at a time, from top to bottom.

5 I. Sutskever, O. Vinyals, and Q.V. Le, "Sequence to Sequence Learning with Neural Networks", (2014).

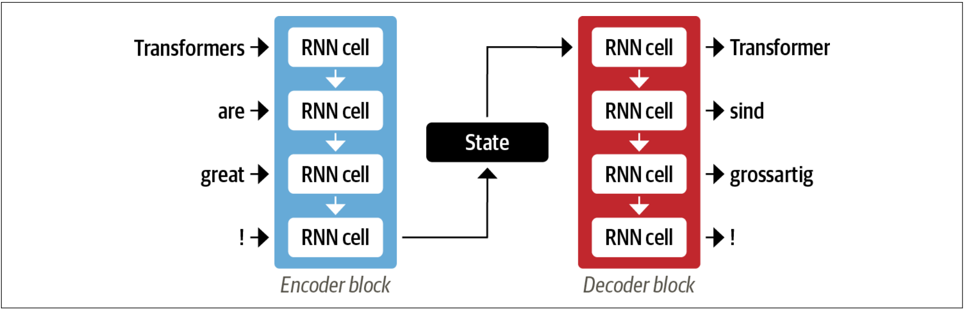


Figure 1-3. An encoder-decoder architecture with a pair of RNNs (in general, there are many more recurrent layers than those shown here)

Although elegant in its simplicity, one weakness of this architecture is that the final hidden state of the encoder creates an *information bottleneck*: it has to represent 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 compressing everything to 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 *attention*,[6] and it is a key component in many modern neural network architectures. Understanding how attention was developed for RNNs will put us in good shape to understand one of the main building blocks of the Transformer architecture. 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 that the decoder can access. However, using all the states at the same time would create 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 different amount of weight, or "attention," to each of the encoder states at every decoding timestep. This process is illustrated in Figure 1-4, where the role of attention is shown for predicting the third token in the output sequence.

6 D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate", (2014).

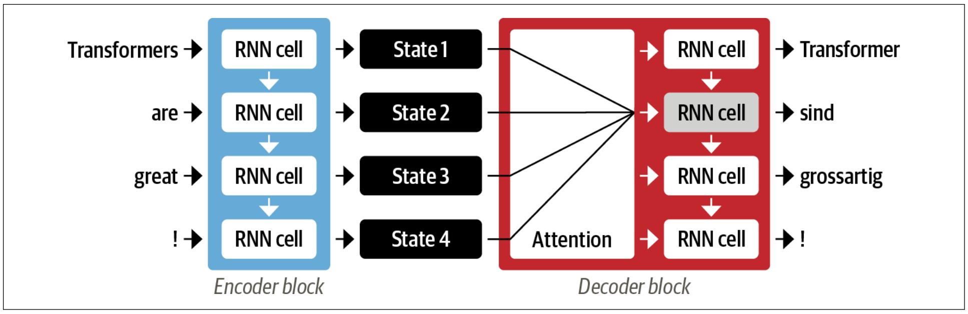


Figure 1-4. An encoder-decoder architecture with an attention mechanism for a pair of RNNs

By focusing on which input tokens are most relevant at each timestep, these attention-based models are able to learn nontrivial alignments between the words in a generated translation and those in a source sentence. For example, Figure 1-5 visualizes the attention weights for an English to French translation model, where each pixel denotes a weight. The figure shows how the decoder is able to correctly align the words "zone" and "Area", which are ordered differently in the two languages.

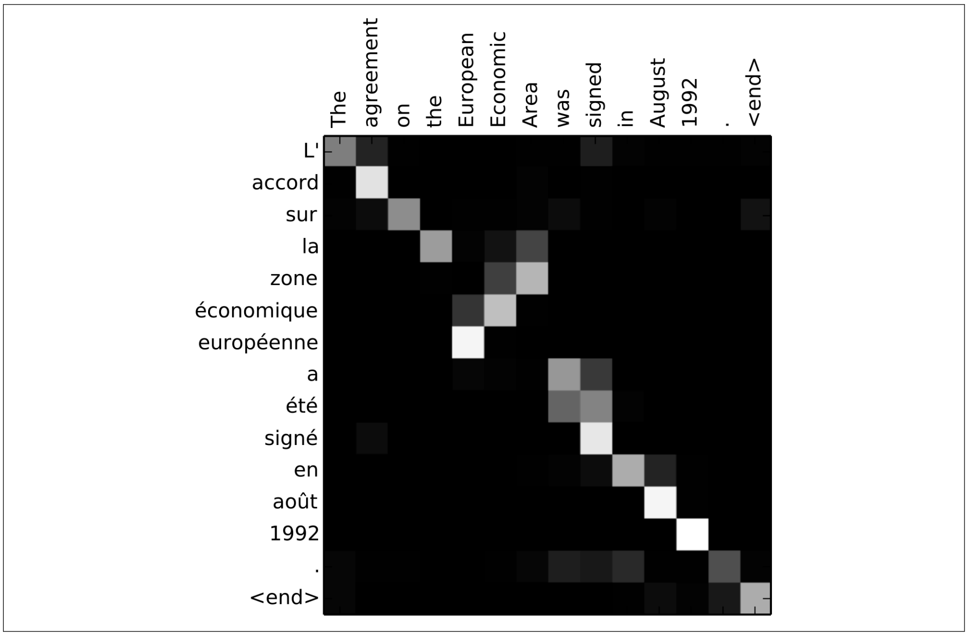


Figure 1-5. RNN encoder-decoder alignment of words in English and the generated translation in French (courtesy of Dzmitry Bahdanau)

Although attention enabled the production of much better translations, there was still a major shortcoming with using recurrent models for the encoder and decoder: the computations are inherently sequential and cannot be parallelized across 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 in Chapter 3, but the basic idea is to allow attention to operate on all the states in the *same layer* of the neural network. This is shown in Figure 1-6, where both the encoder and the decoder have their own self-attention mechanisms, whose outputs are fed to feed-forward neural networks (FF NNs). This architecture can be trained much faster than recurrent models and paved the way for many of the recent breakthroughs in NLP.

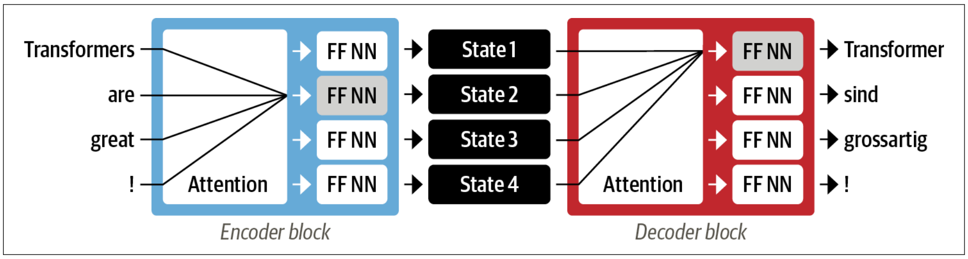


Figure 1-6. Encoder-decoder architecture of the original Transformer

In the original Transformer paper, the translation model was trained from scratch on a large corpus of sentence pairs in various languages. However, in many practical applications of NLP we do not have access to large amounts of labeled text data to train our models on. A final piece was missing to get the transformer revolution started: transfer learning.

Transfer Learning in NLP

It is nowadays common practice in computer vision to use transfer learning to train a convolutional neural network like ResNet on one task, and then adapt it to or *fine-tune* it on a new task. This allows the network to make use of the knowledge learned from the original task. Architecturally, this involves splitting the model into of a *body* and a *head*, where the head is a task-specific network. During training, the weights of the body learn broad features of the source domain, and these weights are used to initialize a new model for the new task.[7] Compared to traditional supervised learning, this approach typically produces high-quality models that can be trained much more

7 Weights are the learnable parameters of a neural network.

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.

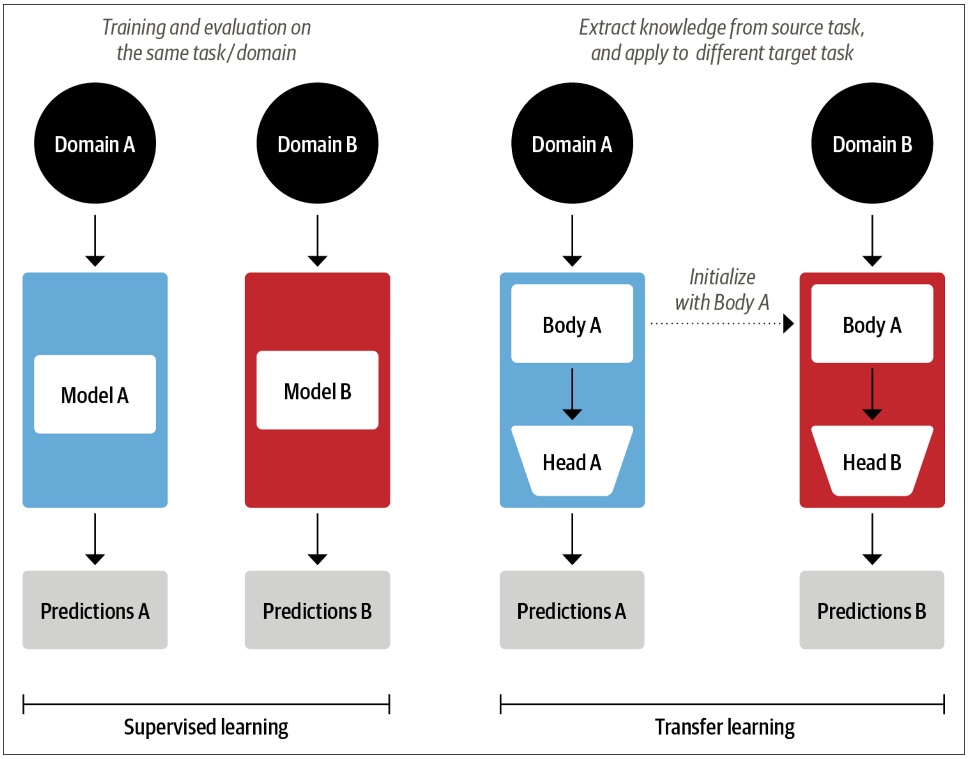


Figure 1-7. Comparison of traditional supervised learning (left) and transfer learning (right)

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 its main purpose is to teach the models the basic features of images, such as edges or colors. These pretrained models can then be fine-tuned on a downstream task such as classifying flower species with a relatively small number of labeled examples (usually a few hundred per class). Fine-tuned models typically achieve a higher accuracy than supervised models trained from scratch on the same amount of labeled data.

Although transfer learning became the standard approach in computer vision, for many years it was not clear what the analogous pretraining process was for NLP. 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 groups proposed new approaches that finally made transfer learning work for NLP. It started with an insight from researchers at OpenAI who obtained strong performance on a sentiment classification task by using features extracted from unsupervised pretraining.[8] This was followed by ULMFiT, which introduced a general framework to adapt pretrained LSTM models for various tasks.[9]

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. This task is referred to as *language modeling*. The elegance of this approach lies in the fact that no labeled data is required, and one can make use of abundantly available text from sources such as Wikipedia.[10]

Domain adaptation
: Once the language model is pretrained on a large-scale corpus, the next step is to adapt it to the in-domain corpus (e.g., from Wikipedia to the IMDb corpus of movie reviews, as in Figure 1-8). This stage still uses language modeling, but now the model has to predict the next word in the target corpus.

Fine-tuning
: In this step, the language model is fine-tuned with a classification layer for the target task (e.g., classifying the sentiment of movie reviews in Figure 1-8).

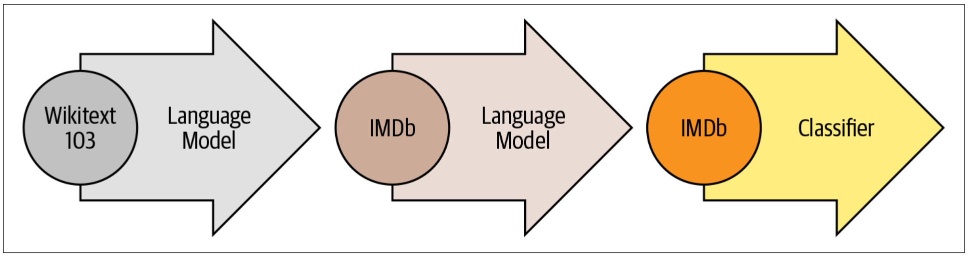


Figure 1-8. The ULMFiT process (courtesy of Jeremy Howard)

By introducing a viable framework for pretraining and transfer learning in NLP, ULMFiT provided the missing piece to make transformers take off. In 2018, two transformers were released that combined self-attention with transfer learning:

8 A. Radford, R. Jozefowicz, and I. Sutskever, "Learning to Generate Reviews and Discovering Sentiment", (2017).

9 A related work at this time was ELMo (Embeddings from Language Models), which showed how pretraining LSTMs could produce high-quality word embeddings for downstream tasks.

10 This is more true for English than for most of the world's languages, where obtaining a large corpus of digitized text can be difficult. Finding ways to bridge this gap is an active area of NLP research and activism.

GPT

Uses only the decoder part of the Transformer architecture, and the same language modeling approach as ULMFiT. GPT was pretrained on the BookCorpus,[11] which consists of 7,000 unpublished books from a variety of genres including Adventure, Fantasy, and Romance.

BERT

Uses the encoder part of the Transformer architecture, and a special form of language modeling called *masked language modeling*. The objective of masked language modeling is to predict randomly masked words in a text. For example, given a sentence like "I looked at my `[MASK]` and saw that `[MASK]` was late." the model needs to predict the most likely candidates for the masked words that are denoted by `[MASK]`. BERT was pretrained on the BookCorpus and English Wikipedia.

GPT and BERT set a new state of the art across a variety of NLP benchmarks and ushered in the age of transformers.

However, with different research labs releasing their models in incompatible frameworks (PyTorch or TensorFlow), it wasn't always easy for NLP practitioners to port these models to their own applications. With the release of 🤗 Transformers, a unified API across more than 50 architectures was progressively built. This library catalyzed the explosion of research into transformers and quickly trickled down to NLP practitioners, making it easy to integrate these models into 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 complex undertaking, and usually involves the following steps:

1. Implement the model architecture in code, typically based on PyTorch or TensorFlow.
2. Load the pretrained weights (if available) from a server.
3. Preprocess the inputs, pass them through the model, and apply some task-specific postprocessing.
4. Implement dataloaders and define loss functions and optimizers to train the model.

11 Y. Zhu et al., "Aligning Books and Movies: Towards Story-Like Visual Explanations by Watching Movies and Reading Books", (2015).

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

This is where 🤗 Transformers comes 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 currently supports three major deep learning frameworks (PyTorch, TensorFlow, and JAX) and allows you to easily switch between them. In addition, it provides task-specific heads so you can easily fine-tune transformers on downstream tasks such as text classification, named entity recognition, and question answering. This reduces the time it takes a practitioner to train and test a handful of models from a week to a single afternoon!

You'll see this for yourself in the next section, where we show that with just a few lines of code, 🤗 Transformers 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

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, the text you're working with 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 Chapter 2. For now, let's have a look at what it takes to extract the sentiment from our piece of text using 🤗 Transformers.

Text Classification

As we'll see in later chapters, 🤗 Transformers has a layered API that allows you to interact with the library at various levels of abstraction. In this chapter we'll start with *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 calling the `pipeline()` function and providing the name of the task we are interested in:

```
from transformers import pipeline

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

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've already downloaded the weights and will use the cached version instead. By default, the `text-classification` pipeline uses a model that's designed for sentiment analysis, but it also supports multiclass and multilabel classification.

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(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 a complaint from an angry customer! Note that for sentiment analysis tasks the pipeline only returns one of the `POSITIVE` or `NEGATIVE` labels, since the other can be inferred by computing `1-score`.

Let's now take a look at another common task, identifying named entities in text.

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 item or service. In NLP, real-world objects like products, places, and people are called *named entities*, and extracting them from text is called *named entity recognition* (NER). We can apply NER by loading the corresponding pipeline and feeding our customer review to it:

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

	entity_group	score	word	start	end
0	ORG	0.879010	Amazon	5	11
1	MISC	0.990859	Optimus Prime	36	49

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

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 each of them. Here we used the `aggregation_strategy` argument to group the words according to the model's predictions. For example, the entity "Optimus Prime" is composed of two words, but is assigned a single category: `MISC` (miscellaneous). The scores tell us how confident the model was about the entities it identified. 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.

See those weird hash symbols (#) in the `word` column in the previous table? These are produced by the model's *tokenizer*, which splits words into atomic units called *tokens*. You'll learn all about tokenization in Chapter 2.

Extracting all the named entities in a text 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. Let's see what we get when we ask a specific question about our customer feedback:

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

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

We can see that along with the answer, the pipeline also returned `start` and `end` integers that correspond to the character indices where the answer span was found (just like with NER tagging). There are several flavors of question answering that we will investigate in Chapter 7, 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 the relevant facts. This is a much more complicated task than the previous ones since it requires the model to *generate* coherent text. 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 summary isn't too bad! Although parts of the original text have been copied, the model was able to capture the essence of the problem and correctly identify that "Bumblebee" (which appeared at the end) was the author of the complaint. In this example you can also see that we passed some keyword arguments like `max_length` and `clean_up_tokenization_spaces` to the pipeline; these allow us to tweak the outputs at runtime.

But what happens when you get 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 a translation pipeline to translate an 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 produced a very good translation that correctly uses German's formal pronouns, like "Ihrem" and "Sie." 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 Hugging Face Hub. Before we take a step back and look at the whole Hugging Face ecosystem, let's examine one last application.

Text Generation

Let's say you would like to be able to provide faster replies to customer feedback by having access to an autocomplete function. With a text generation model you can do this as follows:

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

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

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

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

Now that you've seen a few cool applications of transformer models, you might be wondering where the training happens. All of the models that we've used in this chapter are publicly available and already fine-tuned for the task at hand. In general, however, you'll want to fine-tune 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, 🤗 Transformers is surrounded by a big ecosystem of useful tools that support much of the modern machine learning workflow. Let's take a look.

The Hugging Face Ecosystem

What started with 🤗 Transformers has quickly grown into a whole ecosystem consisting of many libraries and tools to accelerate your NLP and machine learning projects. 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 model weights, datasets, scripts for the evaluation metrics, and more. In this section we'll have a brief look at the various components. We'll skip 🤗 Transformers, as we've already discussed it and we will see a lot more of it throughout the course of the book.

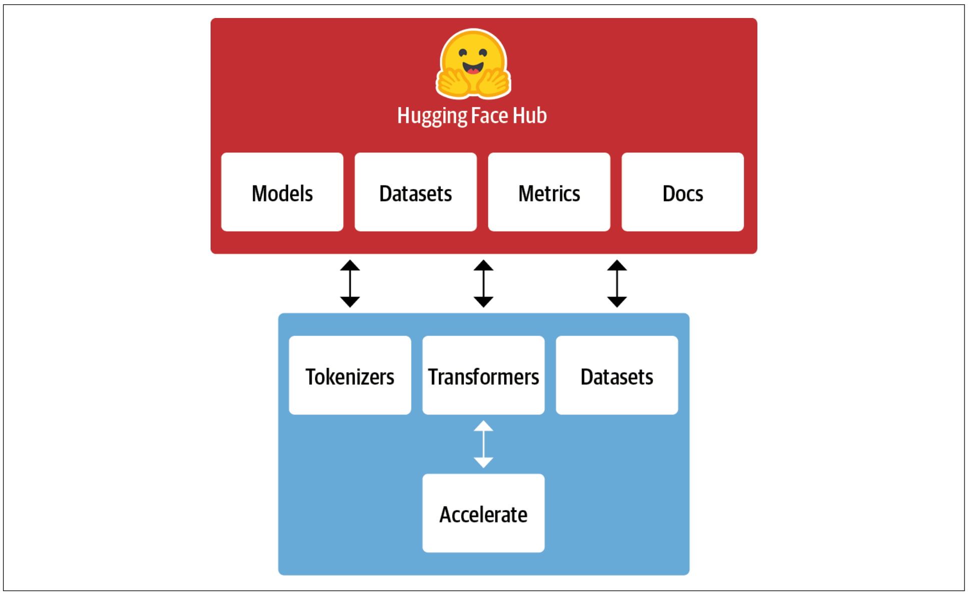
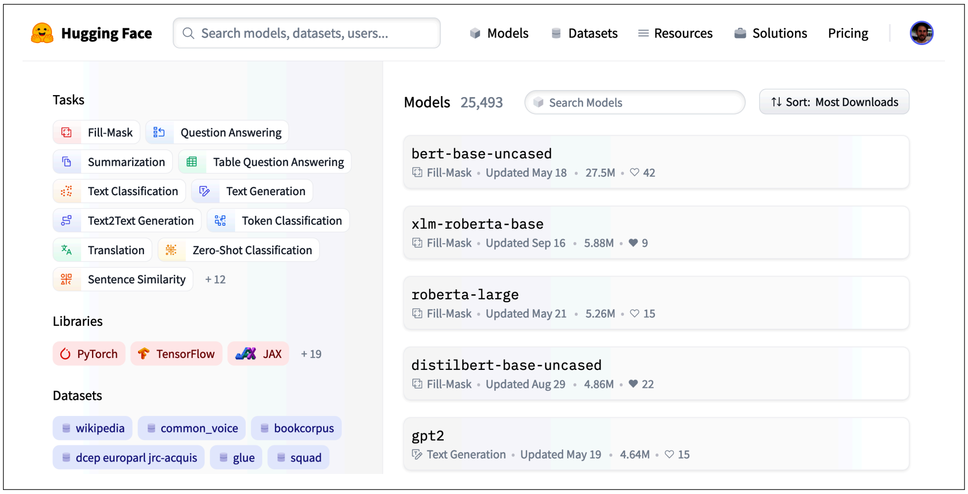


Figure 1-9. An overview of the Hugging Face ecosystem

The Hugging Face Hub

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

The Hugging Face Hub hosts over 20,000 freely available models. As shown in Figure 1-10, there are filters for tasks, frameworks, datasets, and more that are designed to help you navigate the Hub 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 simple, and allows you to focus on the domain-specific parts of your project.



bert-base-uncased
Fill-Mask • Updated May 18 • 27.5M • ♡ 42

xlm-roberta-base
Fill-Mask • Updated Sep 16 • 5.88M • ♥ 9

roberta-large
Fill-Mask • Updated May 21 • 5.26M • ♡ 15

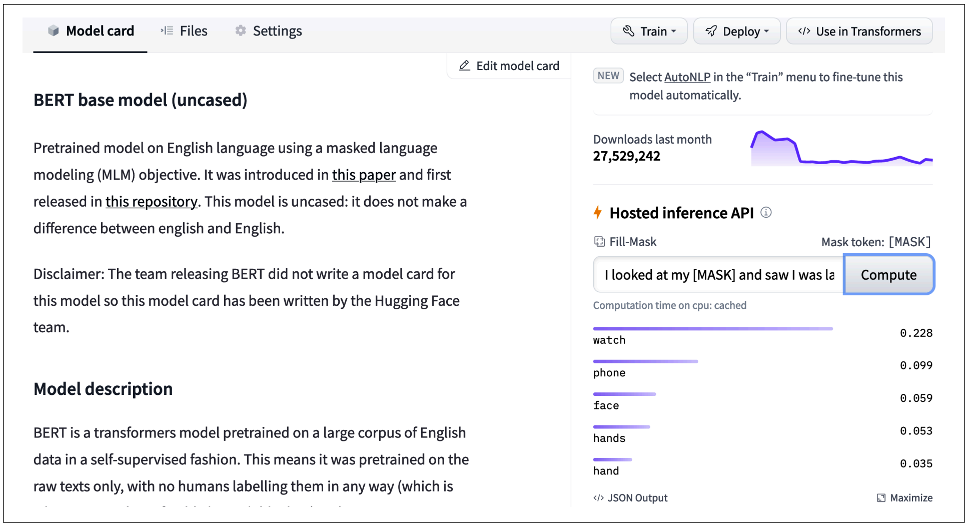
distilbert-base-uncased
Fill-Mask • Updated Aug 29 • 4.86M • ♥ 22

gpt2
Text Generation • Updated May 19 • 4.64M • ♡ 15

Figure 1-10. The Models 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 scripts for computing metrics, which let you reproduce published results or leverage additional data for your application.

The Hub also provides *model* and *dataset cards* to document the contents of models and datasets and help you make an informed decision about whether they're the right ones for you. One of the coolest features of the Hub is that you can try out any model directly through the various task-specific interactive widgets as shown in Figure 1-11.



BERT base model (uncased)

Pretrained model on English language using a masked language modeling (MLM) objective. It was introduced in this paper and first released in this repository. This model is uncased: it does not make a difference between english and English.

Disclaimer: The team releasing BERT did not write a model card for this model so this model card has been written by the Hugging Face team.

Model description

BERT is a transformers model pretrained on a large corpus of English data in a self-supervised fashion. This means it was pretrained on the raw texts only, with no humans labelling them in any way (which is

Select AutoNLP in the "Train" menu to fine-tune this model automatically.

Downloads last month
27,529,242

Mask token: [MASK]

I looked at my [MASK] and saw I was la

Computation time on cpu: cached

```
watch    0.228
phone    0.099
face     0.059
hands    0.053
hand     0.035
```

Figure 1-11. An example model card from the Hugging Face Hub: the inference widget, which allows you to interact with the model, is shown on the right

Let's continue our tour with 🤗 Tokenizers.

PyTorch and TensorFlow also offer hubs of their own and are worth checking out if a particular model or dataset is not available on the Hugging Face Hub.

Hugging Face Tokenizers

Behind each of the pipeline examples that we've seen in this chapter is 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 it's enough to understand that tokens may be words, parts of words, or just characters like punctuation. Transformer models are trained on numerical representations of these tokens, so getting this step right is pretty important for the whole NLP project!

🤗 Tokenizers provides many tokenization strategies and is extremely fast at tokenizing text thanks to its Rust backend.[12] It also takes care of all the pre- and postprocessing steps, 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.

12 Rust is a high-performance programming language.

We need a dataset and metrics to train and evaluate models, so let's take a look at 🤗 Datasets, 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 in your laptop's RAM. In addition, you usually need to implement various scripts to download the data and transform it into a standard format.

🤗 Datasets simplifies this process by providing a standard interface for thousands of datasets that can be 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 lead to deceptive results. By providing the scripts for many metrics, 🤗 Datasets helps make experiments more reproducible and the results more trustworthy.

With the 🤗 Transformers, 🤗 Tokenizers, and 🤗 Datasets libraries we have everything we need to train our very own transformer models! However, as we'll see in Chapter 10 there are situations where we need fine-grained control over the training loop. That's where the last library of the ecosystem comes into play: 🤗 Accelerate.

Hugging Face Accelerate

If you've ever had to write your own training script in PyTorch, chances are that you've had some headaches when trying to port the code that runs on your laptop to the code that runs on your organization's cluster. 🤗 Accelerate 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.

This sums up the core components of Hugging Face's open source ecosystem. But before wrapping up this chapter, let's take a look at a few of the common challenges that come with trying to deploy transformers in the real world.

Main Challenges with Transformers

In this chapter we've gotten a glimpse of the wide range of NLP tasks that can be tackled with transformer models. 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. Here are a few challenges associated with them that we will explore throughout the book:

Language
NLP 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 4, we'll explore multilingual transformers and their ability to perform zero-shot cross-lingual transfer.

Data availability
Although we can use transfer learning to dramatically reduce the amount of labeled training data our models need, it is still a lot compared to how much a human needs to perform the task. Tackling scenarios where you have little to no labeled data is the subject of Chapter 9.

Working with long documents
Self-attention works extremely well on paragraph-long texts, but it becomes very expensive when we move to longer texts like whole documents. Approaches to mitigate this are discussed in Chapter 11.

Opacity
As with other deep learning models, transformers are to a large extent opaque. It is hard or impossible to unravel "why" a model made a certain prediction. This is an especially hard challenge when these models are deployed to make critical decisions. We'll explore some ways to probe the errors of transformer models in Chapters 2 and 4.

Bias
Transformer models are predominantly pretrained on text data from the internet. This 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 Chapter 10.

Although daunting, many of these challenges can be overcome. As well as in the specific chapters mentioned, we will touch on these topics in almost every chapter ahead.

Conclusion

Hopefully, by now you are excited to learn how to start training and integrating these versatile models into your own applications! You've seen in this chapter that with just a few lines of code you can use state-of-the-art models for classification, named entity recognition, question answering, translation, and summarization, but this is really just the "tip of the iceberg."

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

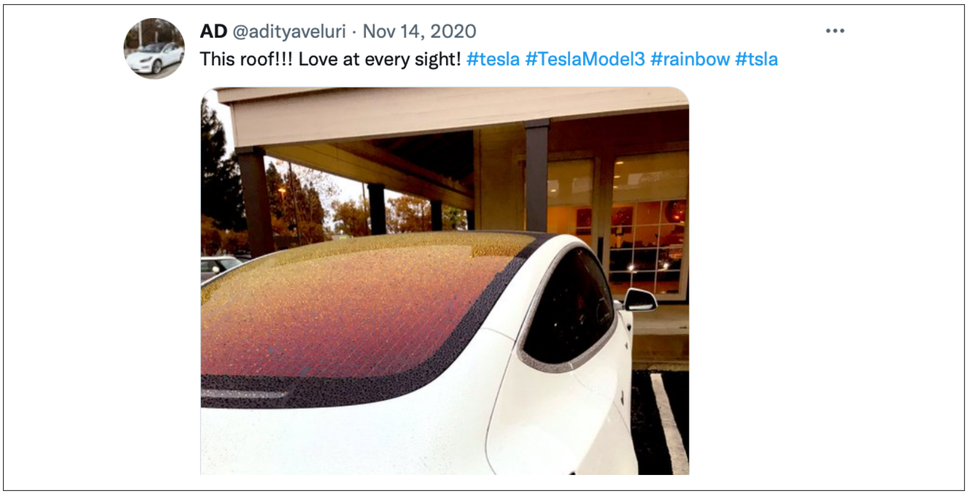
Now that we're armed with the basic concepts behind transformers, it's time to get our hands dirty with our first application: text classification. That's the topic of the next chapter!

CHAPTER 2

Text Classification

Text classification is one of the most common tasks in NLP; it can be used for a broad range of applications, such as tagging customer feedback into categories or routing support tickets according to their language. Chances are that your email program'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 (as we saw in Chapter 1) 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 whether people like its new car roofs or not.



AD @adityaveluri · Nov 14, 2020

This roof!!! Love at every sight! #tesla #TeslaModel3 #rainbow #tsla

Figure 2-1. Analyzing Twitter content can yield useful feedback from customers (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 about your company's product on Twitter. In this chapter, we'll tackle this task using a variant of BERT called DistilBERT.[1] 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 classifier in a few minutes, and if you want to train a larger BERT model you can simply change the checkpoint of the pretrained model. A *checkpoint* corresponds to the set of weights that are loaded into a given transformer architecture.

This will also be our first encounter with three of the 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 roll out!"[2]

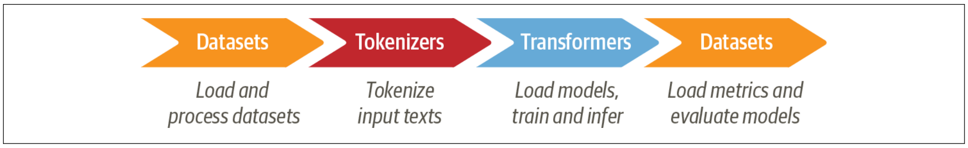

Datasets
Load and process datasets
Tokenizers
Tokenize input texts
Transformers
Load models, train and infer
Datasets
Load metrics and evaluate models

Figure 2-2. A typical pipeline for training transformer models with the 🤗 Datasets, 🤗 Tokenizers, and 🤗 Transformers libraries

The Dataset

To build our emotion detector we'll use a great dataset from an article that explored how emotions are represented in English Twitter messages.[3] 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.

1 V. Sanh et al., "DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter", (2019).

2 Optimus Prime is the leader of a race of robots in the popular Transformers franchise for children (and for those who are young at heart!).

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

A First Look at Hugging Face Datasets

We will use 🤗 Datasets to download the data from the Hugging Face Hub. We can use the `list_datasets()` function to see what datasets are available on the Hub:

```python
from datasets import list_datasets

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

```
There are 1753 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 load the `emotion` dataset with the `load_dataset()` function:

```python
from datasets import load_dataset

emotions = load_dataset("emotion")
```

If we look inside our `emotions` object:

```python
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 we can use the usual dictionary syntax to access an individual split:

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

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

which returns an instance of the `Dataset` class. The `Dataset` object is one of the core data structures in 🤗 Datasets, and we'll be exploring many of its features throughout the course of this book. For starters, it behaves like an ordinary Python array or list, so we can query its length:

```
len(train_ds)
```

```
16000
```

or access a single example by its index:

```
train_ds[0]
```

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

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 are the tweet and the 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:

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

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

In this case, the data type of the `text` column is `string`, while the `label` column is a special `ClassLabel` object that contains information about the class names and their mapping to integers. We can also access several rows with a slice:

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

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

Note that in this case, the dictionary values are now lists instead of individual elements. We can also get the full column by name:

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

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

Now that we've seen how to load and inspect data with 🤗 Datasets, let's do a few checks about the content of our tweets.

What If My Dataset Is Not on the Hub?

We'll be using the Hugging Face Hub to download datasets for most of the examples in this book. But in many cases, you'll find yourself working with data that is either stored on your laptop or on a remote server in your organization. 🤗 Datasets provides several loading scripts to handle local and remote datasets. Examples for the most common data formats are shown in Table 2-1.

Table 2-1. How to load datasets in various formats

Data format	Loading script	Example
CSV	`csv`	`load_dataset("csv", data_files="my_file.csv")`
Text	`text`	`load_dataset("text", data_files="my_file.txt")`
JSON	`json`	`load_dataset("json", data_files="my_file.jsonl")`

As you can see, for each data format, we just need to pass the relevant loading script to the `load_dataset()` function, along with a `data_files` argument that specifies the path or URL to one or more files. For example, the source files for the `emotion` dataset are actually hosted on Dropbox, so an alternative way to load the dataset is to first download one of the splits:

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

If you're wondering why there's a `!` character in the preceding shell command, that's because we're running the commands in a Jupyter notebook. Simply remove the prefix if you want to download and unzip the dataset within a terminal. Now, if we peek at the first row of the *train.txt* file:

```
!head -n 1 train.txt
```

```
i didnt feel humiliated;sadness
```

we can see that here are no column headers and each tweet and emotion are separated by a semicolon. Nevertheless, this is quite similar to a CSV file, so we can load the dataset locally by using the `csv` script and pointing the `data_files` argument to the *train.txt* file:

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

Here we've also specified the type of delimiter and the names of the columns. An even simpler approach is to just point the `data_files` argument to the URL itself:

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

which will automatically download and cache the dataset for you. As you can see, the `load_dataset()` function is very versatile. We recommend checking out the 🤗 Datasets documentation to get a complete overview.

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 `set_format()` method that allows us to change the *output format* of the `Dataset`. Note that this does not change the underlying *data format* (which is an Arrow table), and you can switch to another format later if needed:

```
import pandas as pd

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

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

As you 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 `int2str()` method of the `label` feature to create a new column in our `DataFrame` with the corresponding label names:

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

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

	text	label	label_name
0	i didnt feel humiliated	0	sadness
1	i can go from feeling so hopeless to so damned...	0	sadness

	text	label	label_name
2	im grabbing a minute to post i feel greedy wrong	3	anger
3	i am ever feeling nostalgic about the fireplac...	2	love
4	i am feeling grouchy	3	anger

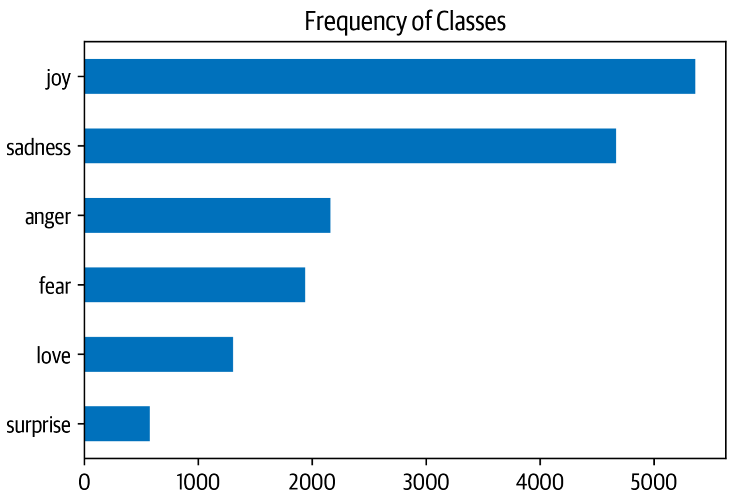
Before diving into building a classifier, let's take a closer look at the dataset. As Andrej Karpathy notes in his famous blog post "A Recipe for Training Neural Networks", becoming "one with the data" is an essential step for training great models!

Looking at the Class Distribution

Whenever you are working on text classification problems, it is a good idea to examine the distribution of examples across the classes. 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 Matplotlib, we can quickly visualize the class distribution as follows:

```
import matplotlib.pyplot as plt

df["label_name"].value_counts(ascending=True).plot.barh()
plt.title("Frequency of Classes")
plt.show()
```



In this case, we can see that the dataset is heavily imbalanced; the `joy` and `sadness` classes appear frequently, whereas `love` and `surprise` are about 5–10 times rarer. There are several ways to deal with imbalanced data, including:

- Randomly oversample the minority class.
- Randomly undersample the majority class.
- Gather more labeled data from the underrepresented classes.

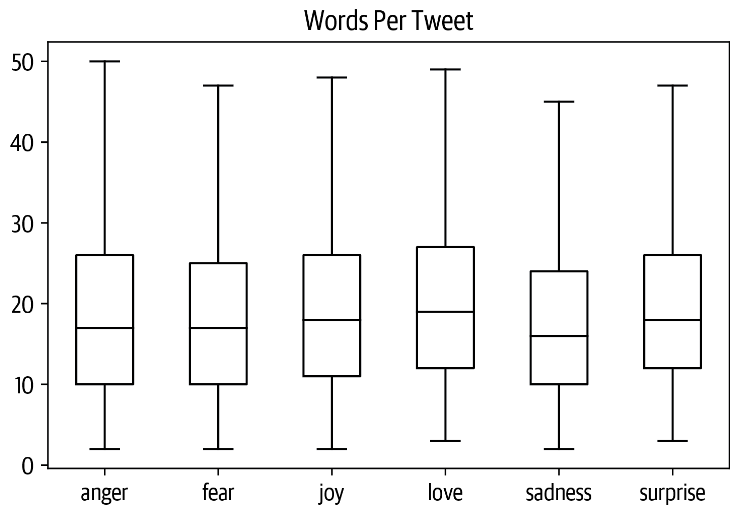
To keep things simple in this chapter, we'll work with the raw, unbalanced class frequencies. If you want to learn more about these sampling techniques, we recommend checking out the Imbalanced-learn library. Just make sure that you don't apply sampling methods *before* creating your train/test splits, or you'll get plenty of leakage between them!

Now that we've looked at the classes, let's take a look at the tweets themselves.

How Long Are Our Tweets?

Transformer models have a maximum input sequence length that is referred to as the *maximum context size*. For applications using DistilBERT, the maximum context size is 512 tokens, which amounts to a few paragraphs of text. As we'll see in the next section, a token is an atomic piece of text; for now, we'll treat a token as a single word. We can get a rough estimate of 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("")
plt.show()
```



From the plot we see that for each emotion, most tweets are around 15 words long and the longest tweets are well below DistilBERT's maximum context size. Texts that are longer than a model's context size need to be truncated, which can lead to a loss in performance if the truncated text contains crucial information; in this case, it looks like that won't be an issue.

Let's now figure out how we can convert these raw texts into a format suitable for 🤗 Transformers! While we're at it, let's also reset the output format of our dataset since we don't need the `DataFrame` format anymore:

```
emotions.reset_format()
```

From Text to Tokens

Transformer models like DistilBERT cannot receive raw strings as input; instead, they assume the text has been *tokenized* and *encoded* as 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 into subunits is usually learned from the corpus. Before looking at the tokenizer used for DistilBERT, let's consider two extreme cases: *character* and *word* tokenization.

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', 'i', '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're not done yet. Our model expects each character to be converted to an integer, a process sometimes 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 = {ch: idx for idx, ch in enumerate(sorted(set(tokenized_text)))}
print(token2idx)
```

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

This gives us a mapping from each character in our vocabulary to a unique integer. We can now use `token2idx` to transform the tokenized text to a list of integers:

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

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

Each token has now been mapped to a unique numerical identifier (hence the name `input_ids`). The last step is to convert `input_ids` to a 2D tensor of one-hot vectors. One-hot vectors are frequently used in machine learning to encode categorical data, which can be either ordinal or nominal. For example, suppose we wanted to encode the names of characters in the *Transformers* TV series. One way to do this would be to map each name to a unique ID, as follows:

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

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

The problem with this approach is that it creates a fictitious ordering between the names, and neural networks are *really* good at learning these kinds of relationships. So instead, we can create a new column for each category and assign a 1 where the category is true, and a 0 otherwise. In Pandas, this can be implemented with the `get_dummies()` function as follows:

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

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

The rows of this `DataFrame` are the one-hot vectors, which have a single "hot" entry with a 1 and 0s everywhere else. Now, looking at our `input_ids`, we have a similar problem: the elements create an ordinal scale. This means that adding or subtracting two IDs is a meaningless operation, since the result is a new ID that represents another random token.

On the other hand, the result of adding two one-hot encodings can easily be interpreted: the two entries that are "hot" indicate that the corresponding tokens co-occur. We can create the one-hot encodings in PyTorch by converting `input_ids` to a tensor and applying the `one_hot()` function as follows:

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

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

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

For each of the 38 input tokens we now have a one-hot vector with 20 dimensions, since our vocabulary consists of 20 unique characters.

It's important to always set `num_classes` in the `one_hot()` function because otherwise the one-hot vectors may end up being shorter than the length of the vocabulary (and need to be padded with zeros manually). In TensorFlow, the equivalent function is `tf.one_hot()`, where the `depth` argument plays the role of `num_classes`.

By examining the first vector, we can verify that a 1 appears in the location indicated by `input_ids[0]`:

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

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

From our simple example we can see that character-level tokenization ignores any structure in the text and treats the whole string as a stream 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* from the data. This requires significant compute, memory, and data. For this reason, character tokenization is rarely used in practice. Instead, some structure of the text is preserved during the tokenization step. *Word tokenization* is a straightforward approach to achieve this, so let's take 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. Using words from the outset enables the model to skip the step of learning words from characters, and thereby reduces the complexity of the training process.

One simple class of word tokenizers uses whitespace to tokenize the text. We can do this by applying Python's `split()` function directly on the raw text (just like we did to measure the tweet lengths):

```
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 to map each word to an ID. 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 declinations, conjugations, or misspellings, the size of the vocabulary can easily grow into the millions!

Some word tokenizers have extra rules for punctuation. One can also apply stemming or lemmatization, which normalizes words to their stem (e.g., "great", "greater", and "greatest" all become "great"), at the expense of losing some information in the text.

Having a large vocabulary is a problem because it requires neural networks to have 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 our neural network. This is a standard step in most NLP architectures, and the resulting weight matrix of this first layer would contain 1 million × 1 thousand = 1 billion weights. This is already comparable to the largest GPT-2 model,[4] which has around 1.5 billion parameters in total!

Naturally, we want to avoid being so wasteful with our model parameters since models 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 words associated with `UNK`.

Wouldn't it be nice if there was a compromise between character and word tokenization that preserved all the input information *and* some of the input structure? There is: *subword tokenization*.

[4] GPT-2 is the successor of GPT, and it captivated the public's attention with its impressive ability to generate realistic text. We'll explore GPT-2 in detail in Chapter 6.

Subword Tokenization

The basic idea behind subword tokenization is to combine the best aspects of character and word tokenization. On the one hand, we want to split rare words into smaller units to allow the model to deal with complex words and misspellings. On the other hand, we want to keep frequent words as unique entities so that we can keep the length of our inputs to a manageable size. The main distinguishing feature of subword tokenization (as well as word tokenization) is that it is *learned* from the pretraining corpus using a mix of statistical rules and algorithms.

There are several subword tokenization algorithms that are commonly used in NLP, but let's start with WordPiece,[5] which is used by the BERT and DistilBERT tokenizers. The easiest way to understand how WordPiece works is to see it in action. 🤗 Transformers provides a convenient `AutoTokenizer` class that allows you to quickly load the tokenizer associated with a pretrained model—we just call its `from_pretrained()` method, providing the ID of a model on the Hub or a local file path. Let's start by loading the tokenizer for DistilBERT:

```python
from transformers import AutoTokenizer

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

The `AutoTokenizer` class belongs to a larger set of "auto" classes whose job is to automatically retrieve the model's configuration, pretrained weights, or vocabulary from the name of the checkpoint. This allows you to quickly switch between models, but if you wish to load the specific class manually you can do so as well. For example, we could have loaded the DistilBERT tokenizer as follows:

```python
from transformers import DistilBertTokenizer

distilbert_tokenizer = DistilBertTokenizer.from_pretrained(model_ckpt)
```

When you run the `AutoTokenizer.from_pretrained()` method 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 the cache, usually at *~/.cache/huggingface*.

Let's examine how this tokenizer works by feeding it our simple "Tokenizing text is a core task of NLP." example text:

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

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

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

Just as with character tokenization, we can see that the words have been mapped to unique integers in the `input_ids` field. We'll discuss the role of the `attention_mask` field in the next section. Now that we have the `input_ids`, we can convert them back into tokens by using the tokenizer's `convert_ids_to_tokens()` method:

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

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

We can observe three things here. First, some special `[CLS]` and `[SEP]` tokens have been added to the start and end of the sequence. These tokens differ from model to model, but their main role is to indicate the start and end of a sequence. Second, the tokens have each been lowercased, which is a feature of this particular checkpoint. Finally, we can see that "tokenizing" and "NLP" have been split into two tokens, which makes sense since they are not common words. The `##` prefix in `##izing` and `##p` means that the preceding string is not whitespace; any token with this prefix should be merged with the previous token when you convert the tokens back to a string. The `AutoTokenizer` class has a `convert_tokens_to_string()` method for doing just that, so let's apply it to our tokens:

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

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

The `AutoTokenizer` class also has several attributes that provide information about the tokenizer. For example, we can inspect the vocabulary size:

```
tokenizer.vocab_size
```

```
30522
```

and the corresponding model's maximum context size:

```
tokenizer.model_max_length
```

```
512
```

Another interesting attribute to know about is the names of the fields that the model expects in its forward pass:

```
tokenizer.model_input_names
```

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

Now that we have a basic understanding of the tokenization process for a single string, let's see how we can tokenize the whole dataset!

When using pretrained models, it is *really* important to make sure that you use the same tokenizer that the model was trained with. From the model's perspective, switching the tokenizer is like shuffling the vocabulary. If everyone around you started swapping random words like "house" for "cat," you'd have a hard time understanding what was going on too!

Tokenizing the Whole Dataset

To tokenize the whole corpus, we'll use the `map()` method of our `DatasetDict` object. We'll encounter this method many times throughout this book, as it provides a convenient way to apply a processing function to each element in a dataset. As we'll soon see, the `map()` method can also be used to create new rows and columns.

To get started, the first thing we need is a processing function to tokenize our examples with:

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

This function applies the tokenizer to a batch of examples; `padding=True` will pad the examples with zeros to the size of the longest one in a batch, and `truncation=True` will truncate the examples to the model's maximum context size. To see `tokenize()` in action, let's pass a batch of two examples from the training set:

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

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

Here we can see the result of padding: the first element of `input_ids` is shorter than the second, so zeros have been added to that element to make them the same length. These zeros have a corresponding `[PAD]` token in the vocabulary, and the set of special tokens also includes the `[CLS]` and `[SEP]` tokens that we encountered earlier:

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

Also note that in addition to returning the encoded tweets as `input_ids`, the tokenizer returns a list of `attention_mask` arrays. This is because we do not want the model to get confused by the additional padding tokens: the attention mask allows the model to ignore the padded parts of the input. Figure 2-3 provides a visual explanation of how the input IDs and attention masks are padded.

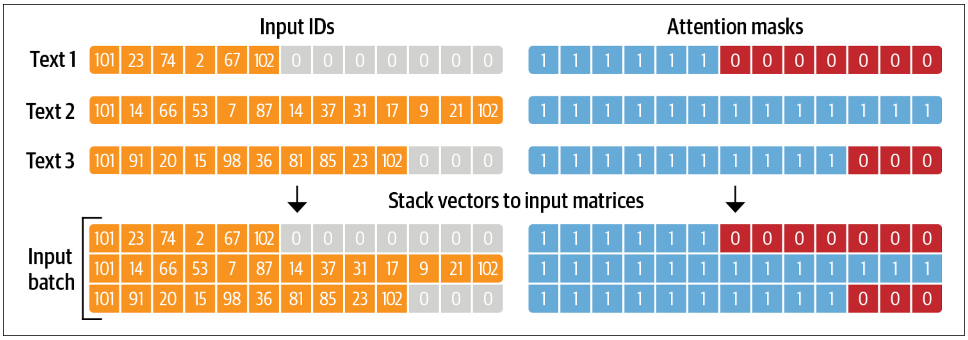


Figure 2-3. For each batch, the input sequences are padded to the maximum sequence length in the batch; the attention mask is used in the model to ignore the padded areas of the input tensors

Once we've defined a processing function, we can apply it across all the splits in the corpus in a single line of code:

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

By default, the `map()` method operates individually on every example in the corpus, so setting `batched=True` will encode the tweets in batches. Because we've set `batch_size=None`, our `tokenize()` function will be applied on the full dataset as a single batch. This ensures that the input tensors and attention masks have the same shape globally, and we can see that this operation has added new `input_ids` and `attention_mask` columns to the dataset:

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

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

In later chapters, we'll see how *data collators* can be used to dynamically pad the tensors in each batch. Padding globally will come in handy in the next section, where we extract a feature matrix from the whole corpus.

Training a Text Classifier

As discussed in Chapter 1, models like DistilBERT are pretrained to predict masked words in a sequence of text. However, we can't use these language models directly for text classification; we need to modify them slightly. To understand what modifications are necessary, let's take a look at the architecture of an encoder-based model like DistilBERT, which is depicted in Figure 2-4.

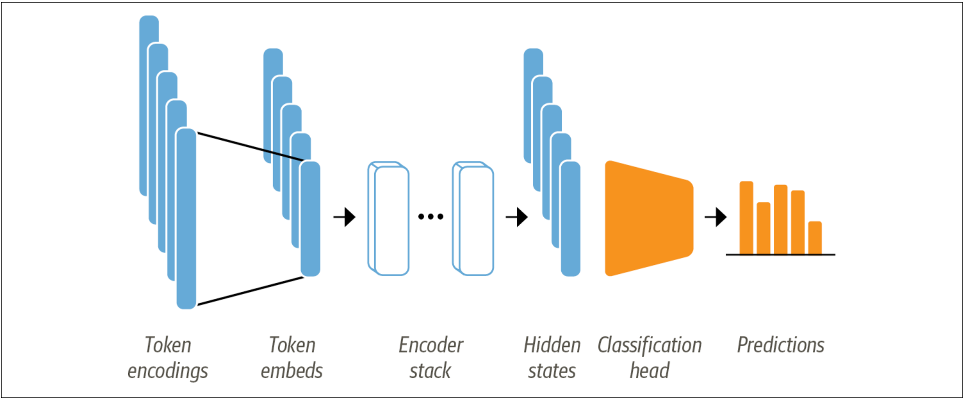


Figure 2-4. The architecture used for sequence classification with an encoder-based transformer; it consists of the model's pretrained body combined with a custom classification head

First, the text is tokenized and represented as one-hot vectors called *token encodings*. The size of the tokenizer vocabulary determines the dimension of the token encodings, and it usually consists of 20k–200k unique tokens. Next, these token encodings are converted to *token embeddings*, which are vectors living in a lower-dimensional space. The token embeddings are then passed through the encoder block layers to yield a *hidden state* for each input token. For the pretraining objective of language modeling,[6] each hidden state is fed to a layer that predicts the masked input tokens. For the classification task, we replace the language modeling layer with a classification layer.

In practice, PyTorch skips the step of creating one-hot vectors for token encodings because multiplying a matrix with a one-hot vector is the same as selecting a column from the matrix. This can be done directly by getting the column with the token ID from the matrix. We'll see this in Chapter 3 when we use the `nn.Embedding` class.

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, without modifying the pretrained model.

6 In the case of DistilBERT, it's guessing the masked tokens.

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

In the following sections we explore both options for DistilBERT and examine their trade-offs.

Transformers as Feature Extractors

Using a transformer as a feature extractor is fairly simple. As shown in Figure 2-5, 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 as a random forest. This method is especially convenient if GPUs are unavailable, since the hidden states only need to be precomputed once.

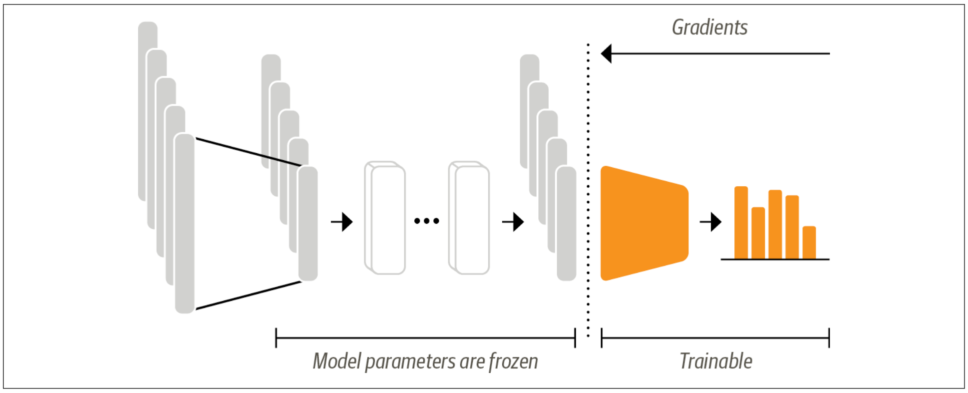


Figure 2-5. In the feature-based approach, the DistilBERT model is frozen and just provides features for a classifier

Using pretrained models

We will use another convenient auto class from 🤗 Transformers called `AutoModel`. Similar to the `AutoTokenizer` class, `AutoModel` has a `from_pretrained()` method to load the weights of a pretrained model. Let's use this method to load the DistilBERT checkpoint:

```
from transformers import AutoModel

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

Here we've used PyTorch to check whether a GPU is available or not, and then chained the PyTorch `nn.Module.to()` method to the model loader. This ensures that the model will run on the GPU if we have one. If not, the model will run on the CPU, which can be considerably slower.

The `AutoModel` class converts the token encodings to embeddings, and then feeds them through the encoder stack to return the hidden states. Let's take a look at how we can extract these states from our corpus.

Interoperability Between Frameworks

Although the code in this book is mostly written in PyTorch, 🤗 Transformers provides tight interoperability with TensorFlow and JAX. This means that you only need to change a few lines of code to load a pretrained model in your favorite deep learning framework! For example, we can load DistilBERT in TensorFlow by using the `TFAutoModel` class as follows:

```python
from transformers import TFAutoModel

tf_model = TFAutoModel.from_pretrained(model_ckpt)
```

This interoperability is especially useful when a model is only released in one framework, but you'd like to use it in another. For example, the XLM-RoBERTa model that we'll encounter in Chapter 4 only has PyTorch weights, so if you try to load it in TensorFlow as we did before:

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

you'll get an error. In these cases, you can specify a `from_pt=True` argument to the `TfAutoModel.from_pretrained()` function, and the library will automatically download and convert the PyTorch weights for you:

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

As you can see, it is very simple to switch between frameworks in 🤗 Transformers! In most cases, you can just add a "TF" prefix to the classes and you'll get the equivalent TensorFlow 2.0 classes. When we use the `"pt"` string (e.g., in the following section), which is short for PyTorch, just replace it with `"tf"`, which is short for TensorFlow.

Extracting the last hidden states

To warm up, let's retrieve the last hidden states for a single string. The first thing we need to do is encode the string and convert the tokens to PyTorch tensors. This can be done by providing the `return_tensors="pt"` argument to the tokenizer as follows:

```python
text = "this is a test"
inputs = tokenizer(text, return_tensors="pt")
print(f"Input tensor shape: {inputs['input_ids'].size()}")
```

```
Input tensor shape: torch.Size([1, 6])
```

As we can see, the resulting tensor has the shape [batch_size, n_tokens]. Now that we have the encodings as a tensor, the final step is to place them on the same device as the model and pass the inputs as follows:

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

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

Here we've used the torch.no_grad() context manager to disable the automatic calculation of the gradient. This is useful for inference since it reduces the memory footprint of the computations. Depending on the model configuration, the output can contain several objects, such as the hidden states, losses, or attentions, arranged in a class similar to a namedtuple in Python. In our example, the model output is an instance of BaseModelOutput, and we can simply access its attributes by name. The current model returns only one attribute, which is the last hidden state, so let's examine its shape:

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

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

Looking at the hidden state tensor, we see that it has the shape [batch_size, n_tokens, hidden_dim]. In other words, a 768-dimensional vector is returned for each of the 6 input tokens. For classification tasks, it is common practice to just use the hidden state associated with the [CLS] token as the input feature. Since this token appears at the start of each sequence, we can extract it by simply indexing into outputs.last_hidden_state as follows:

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

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

Now we know how to get the last hidden state for a single string; let's do the same for the whole dataset by creating a new hidden_state column that stores all these vectors. As we did with the tokenizer, we'll use the map() method of DatasetDict to extract all the hidden states in one go. The first thing we need to do is wrap the previous steps in a processing function:

```
def extract_hidden_states(batch):
    # Place model inputs on the GPU
    inputs = {k:v.to(device) for k,v in batch.items()
```

```
            if k in tokenizer.model_input_names}
    # Extract last hidden states
    with torch.no_grad():
        last_hidden_state = model(**inputs).last_hidden_state
    # Return vector for [CLS] token
    return {"hidden_state": last_hidden_state[:,0].cpu().numpy()}
```

The only difference between this function and our previous logic is the final step where we place the final hidden state back on the CPU as a NumPy array. The `map()` method requires the processing function to return Python or NumPy objects when we're using batched inputs.

Since our model expects tensors as inputs, the next thing to do is convert the `input_ids` and `attention_mask` columns to the `"torch"` format, as follows:

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

We can then go ahead and extract the hidden states across all splits in one go:

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

Notice that we did not set `batch_size=None` in this case, which means the default `batch_size=1000` is used instead. As expected, applying the `extract_ hidden_ states()` function has added a new `hidden_state` column to our dataset:

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

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

Now that we have the hidden states associated with each tweet, the next step is to train a classifier on them. To do that, we'll need a feature matrix—let's take a look.

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

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

Before we train a model on the hidden states, it's good practice to perform a quick check to ensure that they provide a useful representation of the emotions we want to classify. In the next section, we'll see how visualizing the features provides a fast way to achieve this.

Visualizing the training set

Since visualizing the hidden states in 768 dimensions is tricky to say the least, we'll use the powerful UMAP algorithm to project the vectors down to 2D.[7] 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 the UMAP implementation from the `umap-learn` library to reduce the hidden states:

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

# Scale features to [0,1] range
X_scaled = MinMaxScaler().fit_transform(X_train)
# Initialize and fit UMAP
mapper = UMAP(n_components=2, metric="cosine").fit(X_scaled)
# Create a DataFrame of 2D embeddings
df_emb = pd.DataFrame(mapper.embedding_, columns=["X", "Y"])
df_emb["label"] = y_train
df_emb.head()
```

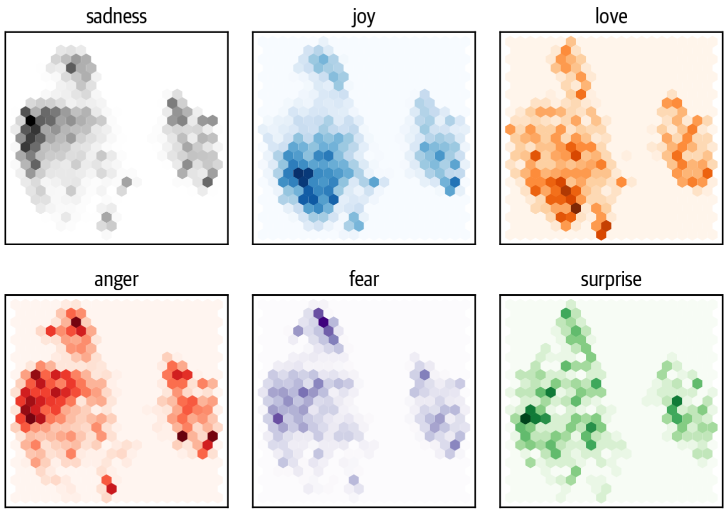
	X	Y	label
0	4.358075	6.140816	0
1	-3.134567	5.329446	0
2	5.152230	2.732643	3
3	-2.519018	3.067250	2
4	-3.364520	3.356613	3

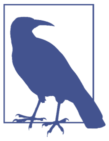
The result is an array with the same number of training samples, but with only 2 features instead of the 768 we started with! Let's investigate the compressed data a little bit further and plot the density of points for each category separately:

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

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

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

```
    axes[i].set_title(label)
    axes[i].set_xticks([]), axes[i].set_yticks([])

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





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.

From this plot we can see some clear patterns: the negative feelings such as `sadness`, `anger`, and `fear` all occupy 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. Although we may have hoped for some separation, this is in no way guaranteed since the model was not trained to know the difference between these emotions. It only learned them implicitly by guessing the masked words in texts.

Now that we've gained some insight into the features of our dataset, let's finally train a model on it!

Training a simple classifier

We've seen that the hidden states are somewhat different between the emotions, although for several of them there is no obvious boundary. Let's use these hidden states to train a logistic regression model with Scikit-learn. Training such a simple model is fast and does not require a GPU:

```
from sklearn.linear_model import LogisticRegression

# We increase `max_iter` to guarantee convergence
lr_clf = LogisticRegression(max_iter=3000)
lr_clf.fit(X_train, y_train)
lr_clf.score(X_valid, y_valid)
```

```
0.633
```

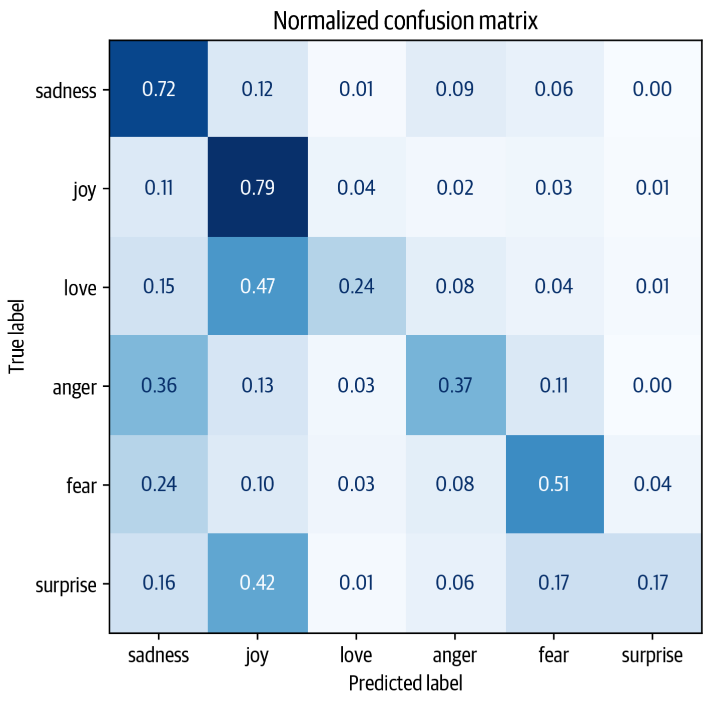
Looking at the accuracy, it might appear that our model is just a bit better than random—but since we are dealing with an unbalanced multiclass dataset, it's actually significantly better. We can examine whether our model is any good by comparing it against a simple baseline. In Scikit-learn there is a `DummyClassifier` that can be used to build a classifier with simple heuristics such as always choosing the majority class or always drawing a random class. In this case the best-performing heuristic is to always choose the most frequent class, which yields an accuracy of about 35%:

```
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
```

So, our simple classifier with DistilBERT 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:

```
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix

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

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`.

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 more computational resources, such as GPUs, that might not be available in your organization. In cases like these, 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.

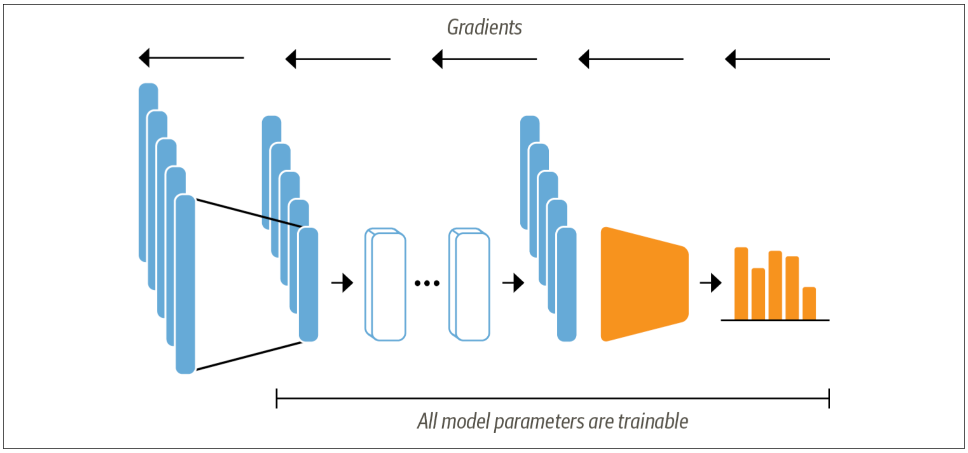


Figure 2-6. When using the fine-tuning approach the whole DistilBERT model is trained along with the classification head

Training the hidden states that serve as inputs to the classification model will help us 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.

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 pretrained 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_ckpt, num_labels=num_labels)
         .to(device))
```

You will see a warning that some parts of the model are randomly initialized. This is normal since the classification head has not yet been trained. The next step is to define the metrics that we'll use to evaluate our model's performance during fine-tuning.

Defining the performance metrics

To monitor metrics during training, we need to define a `compute_metrics()` function for the `Trainer`. This function receives an `EvalPrediction` object (which is a named tuple with `predictions` and `label_ids` attributes) and needs to return a dictionary that maps each metric's name to its value. For our application, we'll compute the F_1-score and the accuracy of the model as follows:

```
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}
```

With the dataset and metrics ready, we just have two final things to take care of before we define the `Trainer` class:

1. Log in to our account on the Hugging Face Hub. This will allow us to push our fine-tuned model to our account on the Hub and share it with the community.
2. Define all the hyperparameters for the training run.

We'll tackle these steps in the next section.

Training the model

If you're running this code in a Jupyter notebook, you can log in to the Hub with the following helper function:

```
from huggingface_hub import notebook_login

notebook_login()
```

This will display a widget in which you can enter your username and password, or an access token with write privileges. You can find details on how to create access tokens in the Hub documentation. If you're working in the terminal, you can log in by running the following command:

```
$ huggingface-cli login
```

To define the training parameters, we use the `TrainingArguments` class. This class stores a lot of information and gives you fine-grained control over the training and evaluation. The most important argument to specify is `output_dir`, which is where all the artifacts from training are stored. Here is an example of `TrainingArguments` in all its glory:

```
from transformers import Trainer, TrainingArguments

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

Here we also set the batch size, learning rate, and number of epochs, and 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"],
                  tokenizer=tokenizer)
trainer.train();
```

Epoch	Training Loss	Validation Loss	Accuracy	F1
1	0.840900	0.327445	0.896500	0.892285
2	0.255000	0.220472	0.922500	0.922550

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 take a more detailed look at the training metrics by calculating the confusion matrix. To visualize the confusion matrix, we first need to get the predictions on the validation set. The `predict()` method of the `Trainer` class returns several useful objects we can use for evaluation:

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

The output of the `predict()` method is a `PredictionOutput` object that 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_output.metrics
```

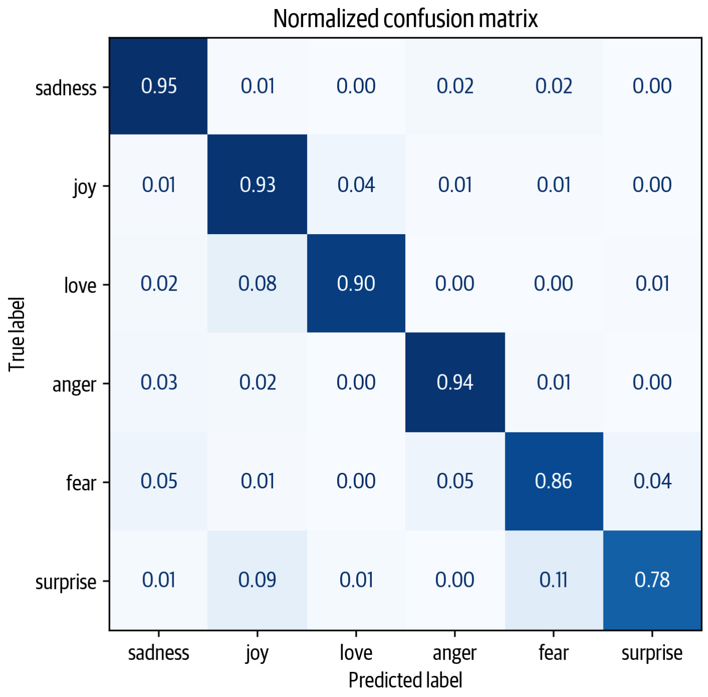
```
{'test_loss': 0.22047173976898193,
 'test_accuracy': 0.9225,
 'test_f1': 0.9225500751072866,
 'test_runtime': 1.6357,
 'test_samples_per_second': 1222.725,
 'test_steps_per_second': 19.564}
```

It also contains the raw predictions for each class. We can decode the predictions greedily using `np.argmax()`. This yields the predicted labels 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)
```



This is much closer to the ideal diagonal confusion matrix. The `love` category is still often confused with `joy`, which seems natural. `surprise` is also frequently mistaken for `joy`, or confused with `fear`. Overall the performance of the model seems quite good, but before we call it a day, let's dive a little deeper into the types of errors our model is likely to make.

Fine-Tuning with Keras

If you are using TensorFlow, it's also possible to fine-tune your models using the Keras API. The main difference from the PyTorch API is that there is no `Trainer` class, since Keras models already provide a built-in `fit()` method. To see how this works, let's first load DistilBERT as a TensorFlow model:

```
from transformers import TFAutoModelForSequenceClassification

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

Next, we'll convert our datasets into the `tf.data.Dataset` format. Because we have already padded our tokenized inputs, we can do this conversion easily by applying the `to_tf_dataset()` method to `emotions_encoded`:

```
# The column names to convert to TensorFlow tensors
tokenizer_columns = tokenizer.model_input_names

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

Here we've also shuffled the training set, and defined the batch size for it and the validation set. The last thing to do is compile and train the model:

```
import tensorflow as tf

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

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

Error analysis

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

```
from torch.nn.functional import cross_entropy

def forward_pass_with_label(batch):
    # Place all input tensors on the same device as the model
    inputs = {k:v.to(device) for k,v in batch.items()
```

```
              if k in tokenizer.model_input_names}

    with torch.no_grad():
        output = model(**inputs)
        pred_label = torch.argmax(output.logits, axis=-1)
        loss = cross_entropy(output.logits, batch["label"].to(device),
                             reduction="none")
    # Place outputs on CPU for compatibility with other dataset columns
    return {"loss": loss.cpu().numpy(),
            "predicted_label": pred_label.cpu().numpy()}
```

Using the `map()` method once more, we can apply this function to get the losses for all the samples:

```
# Convert our dataset back to PyTorch tensors
emotions_encoded.set_format("torch",
                            columns=["input_ids", "attention_mask", "label"])
# Compute loss values
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 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)
df_test["predicted_label"] = (df_test["predicted_label"]
                              .apply(label_int2str))
```

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

Wrong labels
: Every process that adds labels to data can be flawed. Annotators can make mistakes or disagree, while labels that are inferred from other features can be wrong. 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, special characters or strings in the inputs can have a big impact on the model's predictions. 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.

Let's first have a look at the data samples with the highest losses:

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

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

We can clearly see that the model predicted some of the labels incorrectly. On the other hand, it seems that there are quite a few examples with no clear class, which might be either mislabeled or require a new class altogether. In particular, `joy` seems to be mislabeled several times. With this information we can refine the dataset, which often can lead to as big a performance gain (or more) 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. For this reason, it is also worth investing time into looking at the examples that the model is most confident about, so that we can be confident that the model does not improperly exploit certain features of the text. So, let's also look at the predictions with the smallest loss:

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

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

We now know that the `joy` is sometimes mislabeled and that the model is most confident about predicting 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. 🤗 Transformers allows us to do this in a few steps, which we'll show you in the next section.

Saving and sharing the model

The NLP community benefits greatly from sharing pretrained and fine-tuned models, and everybody can share their models with others via the Hugging Face Hub. Any community-generated model can be downloaded from the Hub just like we downloaded the DistilBERT model. With the `Trainer` API, saving and sharing a model is simple:

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

We can also use the fine-tuned model to make predictions on new tweets. Since we've pushed our model to the Hub, we can now use it with the `pipeline()` function, just like we did in Chapter 1. First, let's load the pipeline:

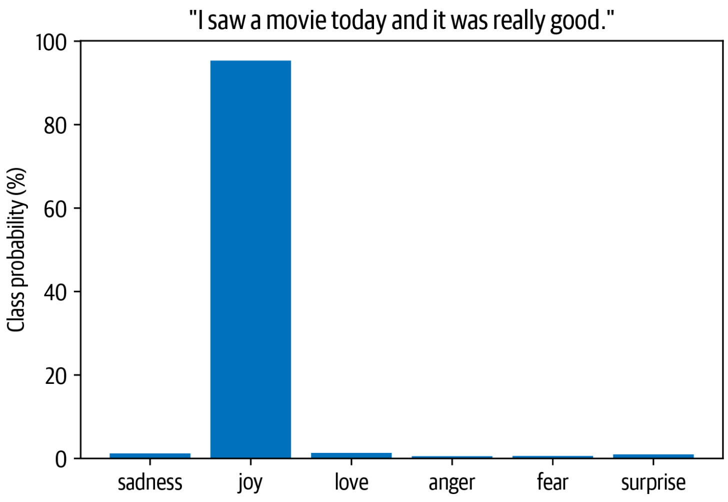
```
from transformers import pipeline

# Change `transformersbook` to your Hub username
model_id = "transformersbook/distilbert-base-uncased-finetuned-emotion"
classifier = pipeline("text-classification", model=model_id)
```

Then let's test the pipeline with a sample tweet:

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

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:

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



Conclusion

Congratulations, you now know how to train a transformer model to classify the emotions in tweets! We have seen two complementary approaches based on features and fine-tuning, and investigated their strengths and weaknesses.

However, this is just the first step in building a real-world application with transformer models, and we have a lot more ground to cover. Here's a list of challenges you're likely to experience in your NLP journey:

My boss wants my model in production yesterday!
In most applications, your model doesn't just sit somewhere gathering dust—you want to make sure it's serving predictions! When a model is pushed to the Hub, an inference endpoint is automatically created that can be called with HTTP requests. We recommend checking out the documentation of the Inference API if you want to learn more.

My users want faster predictions!
: We've already seen one approach to this problem: using DistilBERT. In Chapter 8 we'll dive into knowledge distillation (the process by which DistilBERT was created), 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. In 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 texts are in English!
: It turns out that transformers also come in a multilingual variety, and we'll use them in Chapter 4 to tackle several languages at once.

I don't have any labels!
: If there is very little labeled data available, fine-tuning may not be an option. In Chapter 9, we'll explore some techniques to deal with this situation.

Now that we've seen what's involved in training and sharing a transformer, in the next chapter we'll explore implementing our very own transformer model from scratch.

CHAPTER 3

Transformer Anatomy

In Chapter 2, we saw what it takes to fine-tune and evaluate a transformer. Now let's take a look at how they work under the hood. In this chapter we'll explore the main building blocks of transformer models and how to implement them using PyTorch. We'll also provide guidance on how to do the same in TensorFlow. We'll first focus on building the attention mechanism, and then add the bits and pieces necessary to make a transformer encoder work. We'll 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 🤗 Transformers and fine-tune models for your use case, it can be helpful for comprehending and navigating the limitations of transformers and using them in new domains.

This chapter also introduces a taxonomy of transformers to help you understand the zoo of models that have 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 Architecture

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

As illustrated in Figure 3-1, the encoder and decoder are themselves composed of several building blocks.

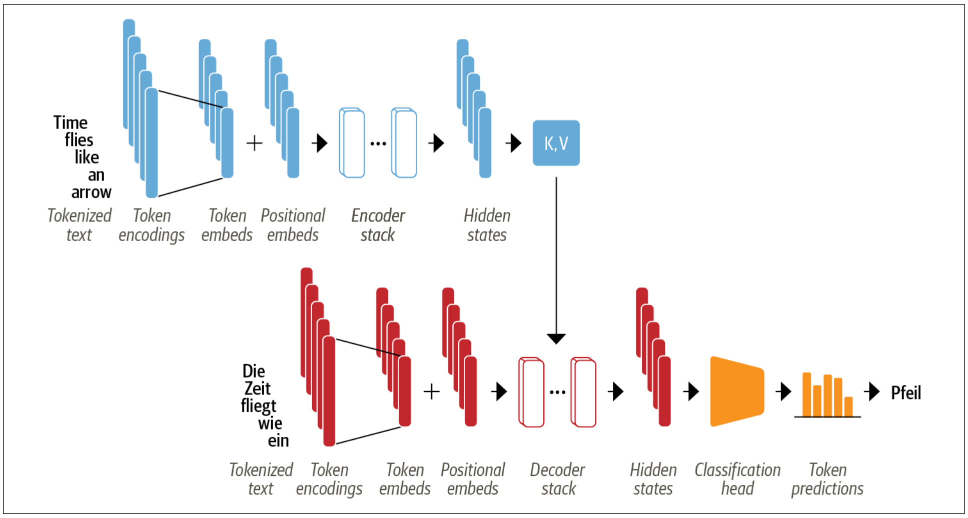


Figure 3-1. Encoder-decoder architecture of the transformer, with the encoder shown in the upper half of the figure and the decoder in the lower half

We'll look at each of the components in detail shortly, but we can already see a few things in Figure 3-1 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 into 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 is composed of a stack of *encoder layers* or "blocks," which is analogous to stacking convolutional layers in computer vision. The same is true of the decoder, which has its own stack of *decoder layers*.
- The encoder's output is fed to each decoder layer, and the decoder 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 (EOS) token is reached. In the example from Figure 3-1, imagine the decoder has already predicted "Die" and "Zeit". Now it

gets these two as an input as well as all the encoder's outputs to predict the next token, "fliegt". In the next step the decoder gets "fliegt" as an additional input. We repeat the process until the decoder predicts the EOS token or we reached a maximum length.

The Transformer architecture was originally designed for sequence-to-sequence tasks like machine translation, but both the encoder and decoder blocks were soon adapted as standalone 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. The representation computed for a given token in this architecture depends both on the left (before the token) and the right (after the token) contexts. This is often called *bidirectional attention*.

Decoder-only
: Given a prompt of text like "Thanks for lunch, I had a…" these models will autocomplete the sequence by iteratively predicting the most probable next word. The family of GPT models belong to this class. The representation computed for a given token in this architecture depends only on the left context. This is often called *causal* or *autoregressive attention*.

Encoder-decoder
: These are used for modeling complex mappings from one sequence of text to another; they're suitable for machine translation and summarization tasks. In addition to the Transformer architecture, which as we've seen combines an encoder and a decoder, the BART and T5 models belong to this class.

> 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 sequence-to-sequence tasks. Similarly, encoder-only models like BERT can be applied to summarization tasks that are usually associated with encoder-decoder or decoder-only models.[1]

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

1 Y. Liu and M. Lapata, "Text Summarization with Pretrained Encoder", (2019).

The Encoder

As we saw earlier, the transformer's encoder consists of many encoder layers stacked next to each other. As illustrated in Figure 3-2, each encoder layer receives a sequence of embeddings and feeds them through the following sublayers:

- A multi-head self-attention layer
- A fully connected feed-forward layer that is applied to each input embedding

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. For example, the word "apple" will be updated to be more "company-like" and less "fruit-like" if the words "keynote" or "phone" are close to it.

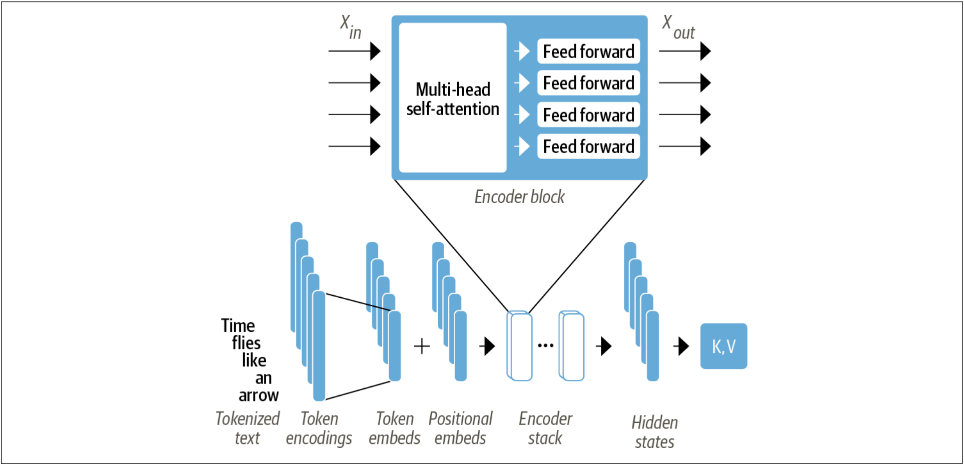


Figure 3-2. Zooming into the encoder layer

Each of these sublayers also uses skip connections 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 in Chapter 1, 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—for example, 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. Another way to formulate this is to say that given a sequence of token embeddings $x_1, ..., x_n$, self-attention produces a sequence of new embeddings $x'_1, ..., x'_n$ where each x'_i is a linear combination of all the x_j:

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

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 annoying insects, 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 in language models like ELMo.[2] A diagram of the process is shown in Figure 3-3, where we illustrate how, depending on the context, two different representations for "flies" can be generated via self-attention.

2 M.E. Peters et al., "Deep Contextualized Word Representations", (2017).

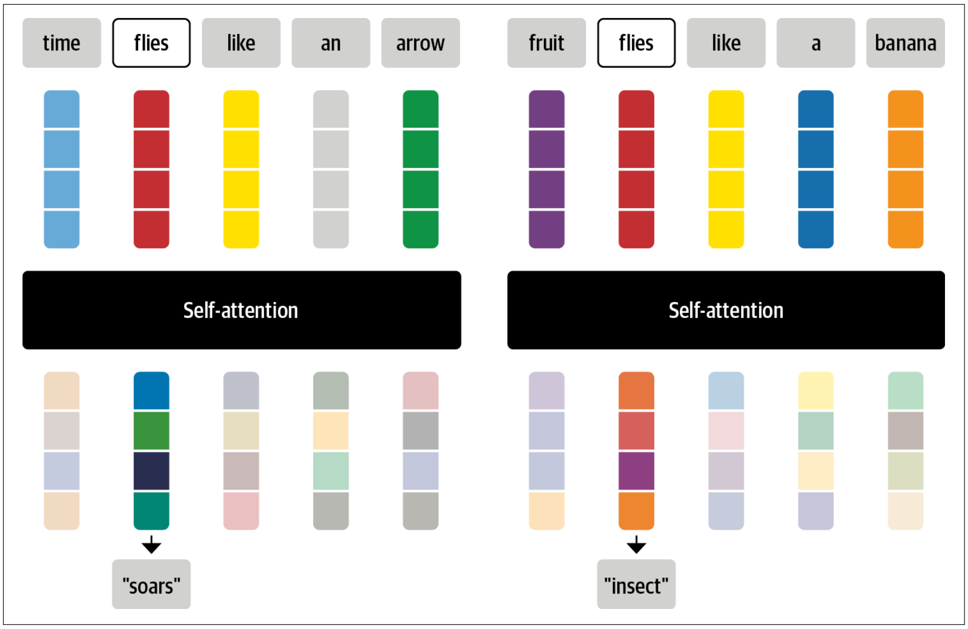


Figure 3-3. Diagram showing how self-attention updates raw token embeddings (upper) into contextualized embeddings (lower) to create representations that incorporate information from the whole sequence

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 paper introducing the Transformer architecture.[3] There are four main steps required to implement this mechanism:

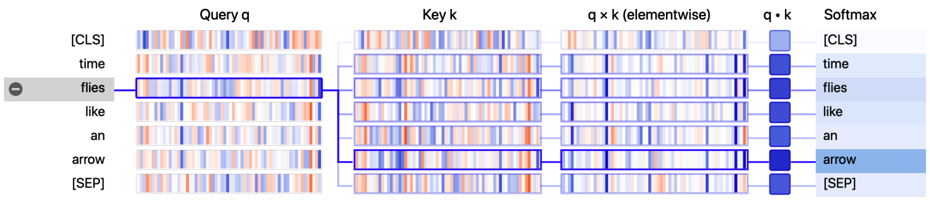
1. Project each token embedding into three vectors called *query*, *key*, and *value*.
2. Compute attention scores. We determine how much the query and key vectors relate to each other using a *similarity function*. As the name suggests, the similarity function for scaled dot-product attention is the dot product, computed efficiently using 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.

3 A. Vaswani et al., "Attention Is All You Need", (2017).

3. 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 to normalize their variance and then normalized with a softmax to ensure all the column values sum to 1. The resulting $n \times n$ matrix now contains all the attention weights, w_{ji}.
4. Update the token embeddings. Once the attention weights are computed, we multiply them by the value vector $v_1, \ldots, v_n$ to obtain an updated representation for embedding $x_i' = \sum_j w_{ji} v_j$.

We can visualize how the attention weights are calculated with a nifty library called *BertViz* for Jupyter. This library provides several functions that can be used to visualize different aspects of attention in transformer 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 the model class from BertViz and then use the `show()` function to generate the interactive visualization for a specific encoder layer and attention head. Note that you need to click the "+" on the left to activate the attention 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)
```



From the visualization, we can see the values of the query and key vectors are represented as vertical bands, where the intensity of each band corresponds to the magnitude. The connecting lines are weighted according to the attention between the tokens, and we can see that the query vector for "flies" has the strongest overlap with the key vector for "arrow".

Demystifying Queries, Keys, and Values

The notion of query, key, and value vectors may seem a bit cryptic the first time you encounter them. Their names were inspired by information retrieval systems, but we can motivate their meaning with a simple analogy. Imagine that you're at the supermarket buying all the ingredients you need for your dinner. You have the dish's recipe, and each of the required ingredients can be thought of as a query. As you scan the shelves, you look at the labels (keys) and check whether they match an ingredient on your list (similarity function). If you have a match, then you take the item (value) from the shelf.

In this analogy, you 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. So if your list includes a dozen eggs, then you might end up grabbing 10 eggs, an omelette, and a chicken wing.

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 3-4.

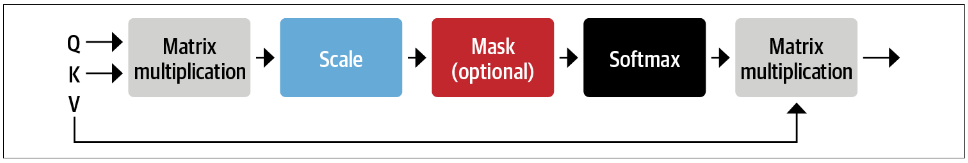


Figure 3-4. Operations in scaled dot-product attention

We will use PyTorch to implement the Transformer architecture in this chapter, but the steps in TensorFlow are analogous. We provide a mapping between the most important functions in the two frameworks in Table 3-1.

Table 3-1. PyTorch and TensorFlow (Keras) classes and methods used in this chapter

PyTorch	TensorFlow (Keras)	Creates/implements
`nn.Linear`	`keras.layers.Dense`	A dense neural network layer
`nn.Module`	`keras.layers.Layer`	The building blocks of models
`nn.Dropout`	`keras.layers.Dropout`	A dropout layer
`nn.LayerNorm`	`keras.layers.LayerNormalization`	Layer normalization
`nn.Embedding`	`keras.layers.Embedding`	An embedding layer
`nn.GELU`	`keras.activations.gelu`	The Gaussian Error Linear Unit activation function
`nn.bmm`	`tf.matmul`	Batched matrix multiplication
`model.forward`	`model.call`	The model's forward pass

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 by setting `add_special_tokens=False`. Next, we need to create some dense embeddings. *Dense* in this context means that each entry in the embeddings contains a nonzero value. In contrast, the one-hot encodings we saw in Chapter 2 are *sparse*, since all entries except one are zero. In PyTorch, we can do this by using a `torch.nn.Embedding` layer that acts as a lookup table for each input ID:

```
from torch import nn
from transformers import AutoConfig

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

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

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. The `AutoConfig` class also stores additional metadata, such as the label names, which are used to format the model's predictions.

Note that the token embeddings at this point are independent of their context. This means that homonyms (words that have the same spelling but different meaning), like "flies" in the previous example, have the same representation. The role of the subsequent attention layers will be to mix these token embeddings to disambiguate and inform the representation of each token with the content of its context.

Now that we have our lookup table, we can generate the embeddings by feeding in 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 shape `[batch_size, seq_len, hidden_dim]`, just like we saw in Chapter 2. We'll postpone the positional encodings, 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])
```

This has created a 5 × 5 matrix of attention scores per sample in the batch. 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 the softmax we will apply next to saturate.

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 the shape `[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 the 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 takes two batches of matrices and multiplies each matrix from the first batch with the corresponding matrix in the second batch.

Let's apply the softmax now:

```
import torch.nn.functional as F

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

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

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 you can think of "self-attention" as just a fancy form of averaging.

Let's wrap these steps into 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—for example, 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-head attention layer* is illustrated in Figure 3-5. But why do we need more than one attention head? The reason is that the softmax of one head tends to focus on mostly one aspect of similarity. Having several heads allows the model to focus on several aspects at once. For instance, one head can focus on subject-verb interaction, whereas another finds nearby adjectives. Obviously we don't handcraft these relations into the model, and they are fully learned from the data. If you are familiar with computer vision models you might see the resemblance to filters in convolutional neural networks, where one filter can be responsible for detecting faces and another one finds wheels of cars in images.

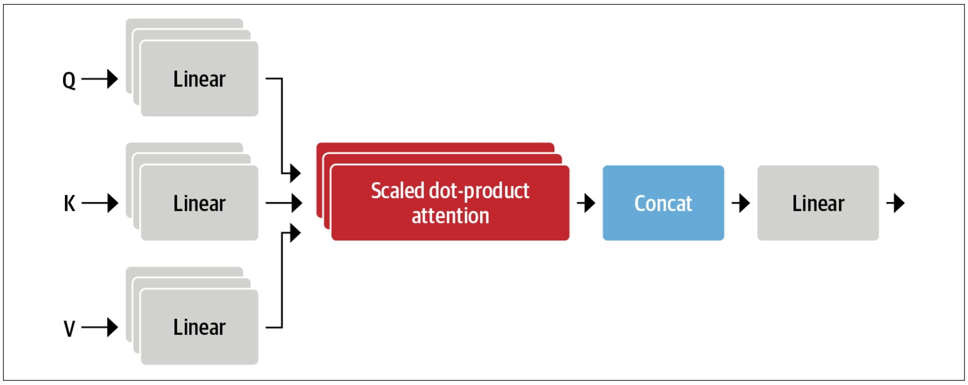


Figure 3-5. Multi-head attention

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 shape `[batch_size, seq_len, head_dim]`, where `head_dim` is the number of dimensions we are projecting into. Although `head_dim` does not have to be smaller than the number of embedding dimensions of the tokens (`embed_dim`), in practice it is chosen to be a multiple of `embed_dim` so that the computation across each head is constant. For example, 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-head 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
```

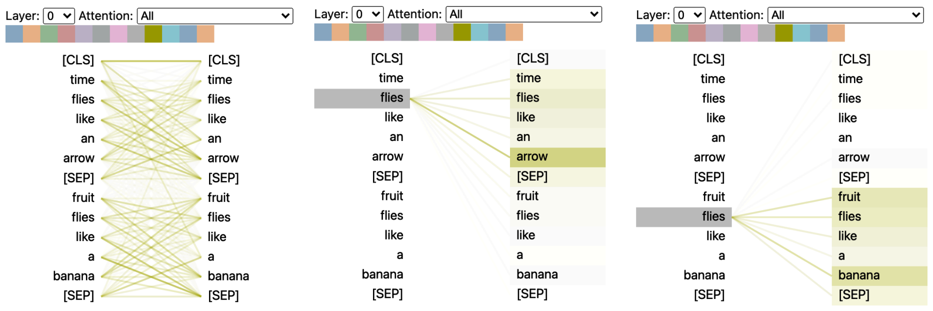
Notice that the concatenated output from the attention heads is also fed through a final linear layer to produce an output tensor of shape `[batch_size, seq_len, hidden_dim]` that is suitable for the feed-forward network downstream. To confirm, let's see if the multi-head attention layer produces the expected shape of our inputs. We pass the configuration we loaded earlier from the pretrained BERT model when initializing the `MultiHeadAttention` module. This ensures that we use the same settings as BERT:

```
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 visualize the attention for two different uses of the word "flies". Here we can use the `head_view()` function from BertViz by computing the attentions of a pretrained checkpoint 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 dark lines representing values close to 1, and faint lines representing values close to 0.

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 that 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" as 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 in which 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.

The Feed-Forward Layer

The feed-forward sublayer in the encoder and decoder is just a simple two-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*. You may also see it referred to as a one-dimensional convolution with a 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 for 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 it's the part that is most often scaled when scaling up the models. We can implement this as a simple `nn.Module` as follows:

```python
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
```

Note that a feed-forward layer such as `nn.Linear` is usually applied to a tensor of shape `(batch_size, input_dim)`, where it acts on each element of the batch dimension independently. This is actually true for any dimension except the last one, so when we pass a tensor of shape `(batch_size, seq_len, hidden_dim)` the layer is applied to all token embeddings of the batch and sequence independently, which is exactly what we want. 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 at how this affects the model architecture.

Adding Layer Normalization

As mentioned earlier, the Transformer architecture makes use of *layer normalization* and *skip connections*. The former normalizes each input in the batch to have zero mean and unity variance. Skip connections pass a tensor to the next layer of the model without processing and add it to the processed tensor. 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 used in the Transformer paper; it 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 value during training.

Pre layer normalization
: This is the most common arrangement found in the literature; it places layer normalization within the span of the skip connections. This tends to be much more stable during training, and it does not usually require any learning rate warm-up.

The difference between the two arrangements is illustrated in Figure 3-6.

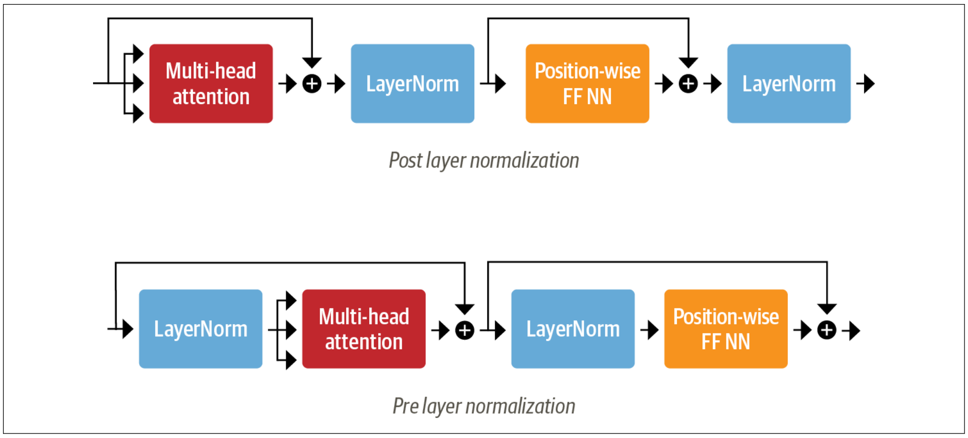


Figure 3-6. Different arrangements of layer normalization in a transformer encoder layer

We'll use the second 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]))
```

We've now implemented our very first transformer encoder layer from scratch! However, there is a caveat with the way we set up 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, the information on token position is lost.[4]

Luckily, there is an easy trick to incorporate positional information using positional embeddings. 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 into their transformations.

There are several ways to achieve this, and one of the most popular approaches is to use a learnable pattern, especially when the pretraining dataset is sufficiently large. 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 positions 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:

```python
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)
```

4 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.

```
        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 some alternatives:

Absolute positional representations
: Transformer models can use static patterns consisting of modulated sine and cosine signals to encode the positions of the tokens. This works especially well when there are not large volumes of data available.

Relative positional representations
: Although absolute positions are important, one can argue that when computing an embedding, the surrounding tokens are most important. Relative positional representations follow that intuition and encode the relative positions between tokens. This cannot be set up by just introducing a new relative embedding layer at the beginning, since the relative embedding changes for each token depending on where from the sequence we are attending to it. Instead, the attention mechanism itself is modified with additional terms that take the relative position between tokens into account. Models such as DeBERTa use such representations.[5]

Let's put all of this together now by building the full transformer encoder 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:

5 By combining the idea of absolute and relative positional representations, rotary position embeddings achieve excellent results on many tasks. GPT-Neo is an example of a model with rotary position embeddings.

```
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 the start and end position of an answer in question answering. In the following section we'll see how we can build a classifier like the one we used in Chapter 2.

Adding a Classification Head

Transformer models are usually divided into a task-independent body and a task-specific head. We'll encounter this pattern again in Chapter 4 when we look at the design pattern of 🤗 Transformers. What we have built so far is the body, so if we wish to build a text classifier, we will need to attach a classification head to that body. We have a hidden state for each token, but we only need to make one prediction. There are several options to approach this. Traditionally, the first token in such models is used for the prediction and we can attach a dropout and a 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, :] # select hidden state of [CLS] token
        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 unnormalized 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 and how we can combine it with a task-specific head. Let's now cast our attention (pun intended!) to the decoder.

The Decoder

As illustrated in Figure 3-7, the main difference between the decoder and encoder is that the decoder has *two* attention sublayers:

Masked multi-head self-attention layer
: 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; masking the inputs ensures the task is not trivial.

Encoder-decoder attention layer
: Performs multi-head attention over the output key and value vectors of the encoder stack, with the intermediate representations of the decoder acting as the queries.[6] This way the encoder-decoder attention layer learns how to relate tokens from two different sequences, such as two different languages. The decoder has access to the encoder keys and values in each block.

Let's take a look at the modifications we need to make to include masking in our self-attention layer, 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)).unsqueeze(0)
mask[0]
```

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

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 `Tensor.masked_fill()` to replace all the zeros with negative infinity:

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

6 Note that unlike the self-attention layer, the key and query vectors in encoder-decoder attention can have different lengths. This is because the encoder and decoder inputs will generally involve sequences of differing length. As a result, the matrix of attention scores in this layer is rectangular, not square.

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

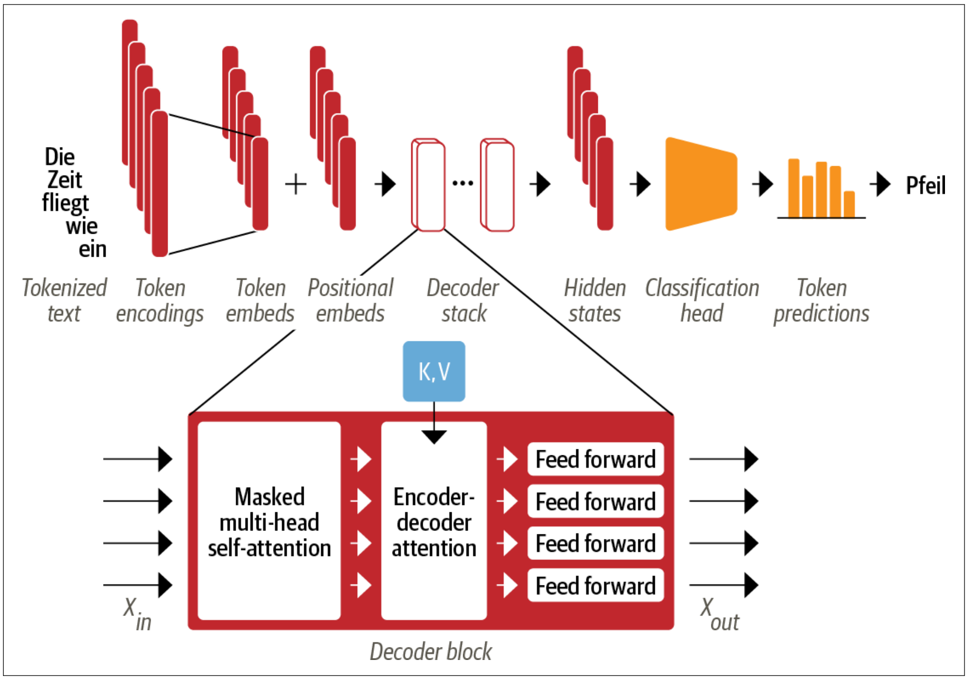


Figure 3-7. Zooming into the transformer decoder layer

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 $e^{-\infty} = 0$ (recall that softmax calculates the normalized exponential). 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; we point the reader to the excellent implementation of minGPT by Andrej Karpathy for details.

We've given you a lot of technical information here, but now you should have a good understanding of how every piece of the Transformer architecture works. Before we move on to building models for tasks more advanced than text classification, 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.

Demystifying Encoder-Decoder Attention

Let's see if we can shed some light on the mysteries of encoder-decoder attention. Imagine you (the decoder) are in class taking an exam. Your task is to predict the next word based on the previous words (decoder inputs), which sounds simple but is incredibly hard (try it yourself and predict the next words in a passage of this book). Fortunately, your neighbor (the encoder) has the full text. Unfortunately, they're a foreign exchange student and the text is in their mother tongue. Cunning students that you are, you figure out a way to cheat anyway. You draw a little cartoon illustrating the text you already have (the query) and give it to your neighbor. They try to figure out which passage matches that description (the key), draw a cartoon describing the word following that passage (the value), and pass that back to you. With this system in place, you ace the exam.

Meet the Transformers

As you've 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, they can still be divided into these three categories.

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

The Transformer Tree of Life

Over time, each of the three main architectures has undergone an evolution of its own. This is illustrated in Figure 3-8, which shows a few of the most prominent models and their descendants.

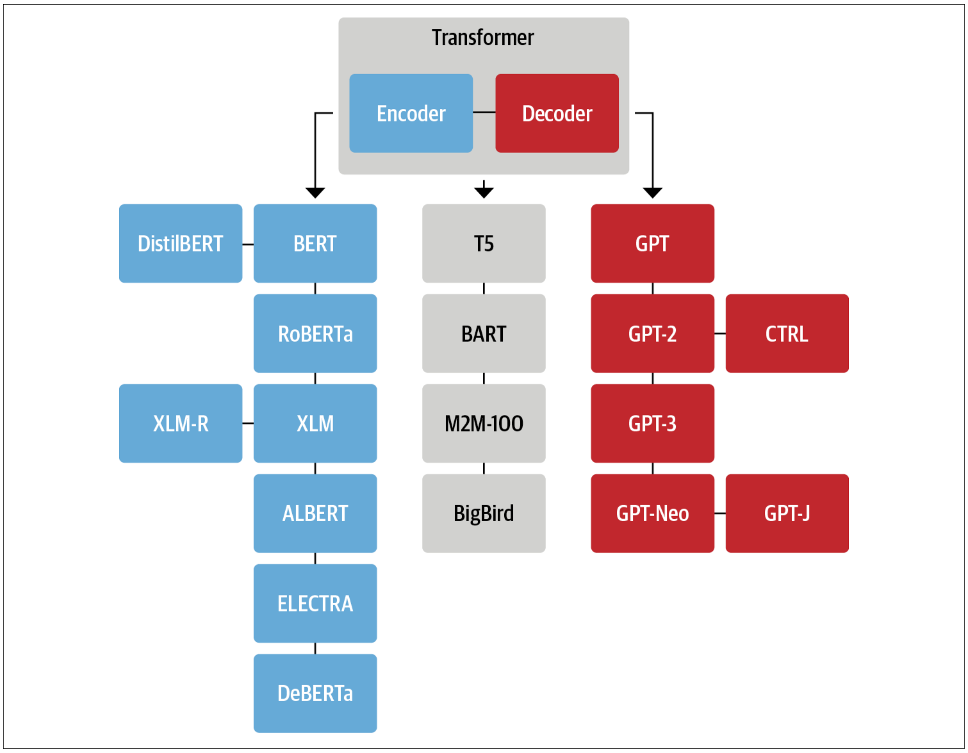


Figure 3-8. An overview of some of the most prominent transformer architectures

With over 50 different architectures included in 🤗 Transformers, this family tree by no means provides a complete overview of all the ones that exist: it simply highlights a few of the architectural milestones. We've covered the original Transformer architecture in depth in this chapter, so let's take a closer look at some 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 outperformed all the state-of-the-art models on the popular GLUE benchmark,[7] which measures natural language understanding (NLU) across several tasks of varying difficulty. Subsequently, the pretraining objective and the architecture of BERT have been adapted to further improve performance. Encoder-only models still dominate research and industry on NLU tasks such as text

7 A. Wang et al., "GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding", (2018).

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 one text passage is likely to follow another.[8] The former task is called *masked language modeling* (MLM) and the latter *next sentence prediction* (NSP).

DistilBERT
: Although BERT delivers great results, it's size can make it tricky to deploy in environments where low latencies are required. By using a technique known as knowledge distillation during pretraining, DistilBERT achieves 97% of BERT's performance while using 40% less memory and being 60% faster.[9] You can find more details on knowledge distillation in Chapter 8.

RoBERTa
: A study following the release of BERT revealed that its performance can be further improved by modifying the pretraining scheme. RoBERTa is trained longer, on larger batches with more training data, and it drops the NSP task.[10] Together, these changes significantly improve its performance compared to the original BERT model.

XLM
: Several pretraining objectives for building multilingual models were explored in the work on the cross-lingual language model (XLM),[11] including the autoregressive language modeling from GPT-like models and MLM from BERT. In addition, the authors of the paper on XLM pretraining 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 results 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 upscaling the training data.[12] Using the Common Crawl corpus, its developers created a dataset with 2.5 terabytes of text; they then trained an encoder with MLM on this

8 J. Devlin et al., "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding", (2018).

9 V. Sanh et al., "DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter", (2019).

10 Y. Liu et al., "RoBERTa: A Robustly Optimized BERT Pretraining Approach", (2019).

11 G. Lample, and A. Conneau, "Cross-Lingual Language Model Pretraining", (2019).

12 A. Conneau et al., "Unsupervised Cross-Lingual Representation Learning at Scale", (2019).

dataset. Since the dataset only contains data without parallel texts (i.e., translations), the TLM objective of XLM was 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.[13] First, it decouples the token embedding dimension from the hidden dimension, thus allowing the embedding dimension to be small and thereby saving parameters, especially when the vocabulary gets large. Second, all layers share the same parameters, which decreases the number of effective parameters even further. Finally, the NSP objective is replaced with a sentence-ordering prediction: the model needs to predict whether or not the order of two consecutive sentences was swapped rather than predicting if they belong together at all. These changes make it possible to train even larger models with fewer parameters and reach 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:[14] the first model (which is typically small) works like a standard masked language model and predicts masked tokens. The second model, called the *discriminator*, is then tasked to predict which of the tokens in the first model's output 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.[15] First, each token is represented as two vectors: one for the content, the other for relative position. By disentangling the tokens' content from their relative positions, the self-attention layers can better model the dependency of nearby token pairs. 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

13 Z. Lan et al., "ALBERT: A Lite BERT for Self-Supervised Learning of Language Representations", (2019).

14 K. Clark et al., "ELECTRA: Pre-Training Text Encoders as Discriminators Rather Than Generators", (2020).

15 P. He et al., "DeBERTa: Decoding-Enhanced BERT with Disentangled Attention", (2020).

beat the human baseline on the SuperGLUE benchmark,[16] a more difficult version of GLUE consisting of several subtasks used to measure NLU performance.

Now that we've highlighted some of the major encoder-only architectures, let's take a look at the decoder-only models.

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 5 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:[17] the novel and efficient transformer decoder architecture, and transfer learning. In that setup, the model was pretrained by predicting the next word based on the previous ones. 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 upscaled to produce GPT-2.[18] This model is able to produce long sequences of coherent text. Due to concerns about possible 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 an input sequence (also called a *prompt*). However, the user has little control over the style of the generated sequence. The Conditional Transformer Language (CTRL) model addresses this issue by adding "control tokens" at the beginning of the sequence.[19] These allow the style of the generated text to be controlled, which allows for diverse generation.

16 A. Wang et al., "SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems", (2019).

17 A. Radford et al., "Improving Language Understanding by Generative Pre-Training", OpenAI (2018).

18 A. Radford et al., "Language Models Are Unsupervised Multitask Learners", OpenAI (2019).

19 N.S. Keskar et al., "CTRL: A Conditional Transformer Language Model for Controllable Generation", (2019).

GPT-3

Following the success of scaling GPT up to GPT-2, a thorough analysis on the behavior of language models at different scales revealed that there are simple power laws that govern the relation between compute, dataset size, model size, and the performance of a language model.[20] Inspired by these insights, GPT-2 was upscaled by a factor of 100 to yield GPT-3,[21] 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 translating text to code, 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 were trained by EleutherAI, a collective of researchers who aim to re-create and release GPT-3 scale models.[22] The current models are smaller variants of the full 175-billion-parameter model, with 1.3, 2.7, and 6 billion parameters, and are competitive with the smaller GPT-3 models OpenAI offers.

The final branch in the transformers tree of life is the encoder-decoder models. Let's take a look.

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 architecture that have novel applications across both NLU and NLG domains:

T5

The T5 model unifies all NLU and NLG tasks by converting them into text-to-text tasks.[23] All tasks are framed as sequence-to-sequence tasks, where adopting an encoder-decoder architecture is natural. For text classification problems, for example, this means that the text is used as the encoder input and the decoder has to generate the label as normal text instead of a class. We will look at this in more detail in Chapter 6. The T5 architecture uses the original Transformer architecture. Using the large crawled C4 dataset, the model is pretrained with masked language modeling as well as the SuperGLUE tasks by translating all of

20 J. Kaplan et al., "Scaling Laws for Neural Language Models", (2020).

21 T. Brown et al., "Language Models Are Few-Shot Learners", (2020).

22 S. Black et al., "GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-TensorFlow", (2021); B. Wang and A. Komatsuzaki, "GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model", (2021).

23 C. Raffel et al., "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer", (2019).

them to text-to-text tasks. The largest model with 11 billion parameters yielded state-of-the-art results on several benchmarks.

BART

BART combines the pretraining procedures of BERT and GPT within the encoder-decoder architecture.[24] The input sequences undergo one of several possible transformations, from simple masking to sentence permutation, token deletion, and document rotation. These modified 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.[25] This allows for high-quality translations between rare and underrepresented languages. The model uses prefix tokens (similar to the special `[CLS]` token) to indicate the source and target language.

BigBird

One main limitation of transformer models 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.[26] This allows for the drastic scaling of contexts from 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.

Pretrained checkpoints of all models that we have seen in this section are available on the Hugging Face Hub and can be fine-tuned to your use case with 🤗 Transformers, as described in the previous chapter.

Conclusion

In this chapter we started at the heart of the Transformer architecture with a deep dive into self-attention, and we subsequently added all the necessary parts to build a

24 M. Lewis et al., "BART: Denoising Sequence-to-Sequence Pre-Training for Natural Language Generation, Translation, and Comprehension", (2019).

25 A. Fan et al., "Beyond English-Centric Multilingual Machine Translation", (2020).

26 M. Zaheer et al., "Big Bird: Transformers for Longer Sequences", (2020).