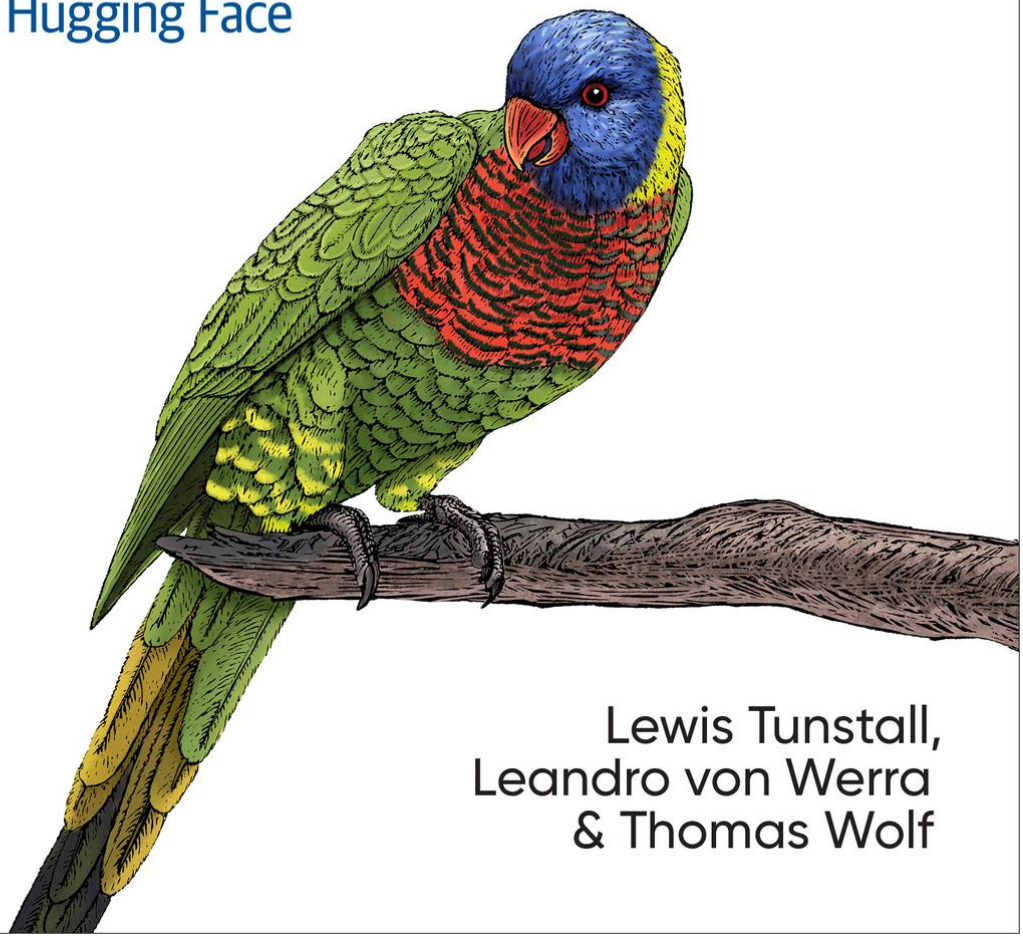
O'REILLY®

Natural Language Processing with Transformers

Building Language Applications with Hugging Face



Lewis Tunstall,
Leandro von Werra
& Thomas Wolf

Praise for *Natural Language Processing with Transformers*

Pretrained transformer language models have taken the NLP world by storm, while libraries such as 🤗 Transformers have made them much easier to use. Who better to teach you how to leverage the latest breakthroughs in NLP than the creators of said library? Natural Language Processing with Transformers is a tour de force, reflecting the deep subject matter expertise of its authors in both engineering and research. It is the rare book that offers both substantial breadth and depth of insight and deftly mixes research advances with real-world applications in an accessible way. The book gives informed coverage of the most important methods and applications in current NLP, from multilingual to efficient models and from question answering to text generation. Each chapter provides a nuanced overview grounded in rich code examples that highlights best practices as well as practical considerations and enables you to put research-focused models to impactful real-world use. Whether you're new to NLP or a veteran, this book will improve your understanding and fast-track your development and deployment of state-of-the-art models.

Sebastian Ruder, Google DeepMind

Transformers have changed how we do NLP, and Hugging Face has pioneered how we use transformers in product and research. Lewis Tunstall, Leandro von Werra, and Thomas Wolf from Hugging Face have written a timely volume providing a convenient and hands-on introduction to this critical topic. The book offers a solid conceptual grounding of transformer mechanics, a tour of the transformer menagerie, applications of transformers, and practical issues in training and bringing transformers to production. Having read chapters in this book, with the depth of its content and lucid presentation, I am confident that this will be the number one resource for anyone interested in learning transformers, particularly for natural language processing.

Delip Rao, Author of Natural Language Processing and Deep Learning with PyTorch

Complexity made simple. This is a rare and precious book about NLP, transformers, and the growing ecosystem around them, Hugging Face. Whether these are still buzzwords to you or you already have a solid grasp of it all, the authors will navigate you with humor, scientific rigor, and plenty of code examples into the deepest secrets of the coolest technology around. From "off-the-shelf pretrained" to "from-scratch custom" models, and from performance to missing labels issues, the authors address practically every real-life struggle of a ML engineer and provide state-of-the-art solutions, making this book destined to dictate the standards in the field for years to come.

Luca Perrozzi, PhD, Data Science and Machine Learning Associate Manager at Accenture

Natural Language Processing with Transformers

by Lewis Tunstall, Leandro von Werra, and Thomas Wolf

Printed in the United States of America.

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

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

- Acquisitions Editor: Rebecca Novack
- Development Editor: Melissa Potter
- Production Editor: Katherine Tozer
- Copyeditor: Rachel Head
- Proofreader: Kim Cofer
- Indexer: Potomac Indexing, LLC
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Christa Lanz

- February 2022: First Edition

Revision History for the First Edition

- 2022-01-26: First Release

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

978-1-098-10324-8 [LSI]

Foreword

A miracle is taking place as you read these lines: the squiggles on this page are transforming into words and concepts and emotions as they navigate their way through your cortex. My thoughts from November 2021 have now successfully invaded your brain. If they manage to catch your attention and survive long enough in this harsh and highly competitive environment, they may have a chance to reproduce again as you share these thoughts with others. Thanks to language, thoughts have become airborne and highly contagious brain germs—and no vaccine is coming.

Luckily, most brain germs are harmless,[1] and a few are wonderfully useful. In fact, humanity's brain germs constitute two of our most precious treasures: knowledge and culture. Much as we can't digest properly without healthy gut bacteria, we cannot think properly without healthy brain germs. Most of your thoughts are not actually yours: they arose and grew and evolved in many other brains before they infected you. So if we want to build intelligent machines, we will need to find a way to infect them too.

The good news is that another miracle has been unfolding over the last few years: several breakthroughs in deep learning have given birth to powerful language models. Since you are reading this book, you have probably seen some astonishing demos of these language models, such as GPT-3, which given a short prompt such as "a frog meets a crocodile" can write a whole story. Although it's not quite Shakespeare yet, it's sometimes hard to believe that these texts were written by an artificial neural network. In fact, GitHub's Copilot system is helping me write these lines: you'll never know how much I really wrote.

The revolution goes far beyond text generation. It encompasses the whole realm of natural language processing (NLP), from text classification to summarization, translation, question answering, chatbots, natural language understanding (NLU), and more. Wherever there's language, speech or text, there's an application for NLP. You can already ask your phone for tomorrow's weather, or chat with a virtual help desk assistant to troubleshoot a problem, or get meaningful results from search engines that seem to truly understand your query. But the technology is so new that the best is probably yet to come.

Like most advances in science, this recent revolution in NLP rests upon the hard work of hundreds of unsung heroes. But three key ingredients of its success do stand out:

- The *transformer* is a neural network architecture proposed in 2017 in a groundbreaking paper called "Attention Is All You Need", published by a team of Google researchers. In just a few years it swept across the field, crushing previous architectures that were typically based on recurrent neural networks (RNNs). The Transformer architecture is excellent at capturing patterns in long sequences of data and dealing with huge datasets—so much so that its use is now extending well beyond NLP, for example to image processing tasks.
- In most projects, you won't have access to a huge dataset to train a model from scratch. Luckily, it's often possible to download a model that was *pretrained* on a generic dataset: all you need to do then is fine-tune it on your own (much smaller) dataset. Pretraining has been mainstream in image processing since the early 2010s, but in NLP it was restricted to contextless word embeddings (i.e., dense vector representations of individual words). For example, the word "bear" had the same

pretrained embedding in “teddy bear” and in “to bear.” Then, in 2018, several papers proposed full-blown language models that could be pretrained and fine-tuned for a variety of NLP tasks; this completely changed the game.

- *Model hubs* like Hugging Face’s have also been a game-changer. In the early days, pretrained models were just posted anywhere, so it wasn’t easy to find what you needed. Murphy’s law guaranteed that PyTorch users would only find TensorFlow models, and vice versa. And when you did find a model, figuring out how to fine-tune it wasn’t always easy. This is where Hugging Face’s Transformers library comes in: it’s open source, it supports both TensorFlow and PyTorch, and it makes it easy to download a state-of-the-art pretrained model from the Hugging Face Hub, configure it for your task, fine-tune it on your dataset, and evaluate it. Use of the library is growing quickly: in Q4 2021 it was used by over five thousand organizations and was installed using `pip` over four million times per month. Moreover, the library and its ecosystem are expanding beyond NLP: image processing models are available too. You can also download numerous datasets from the Hub to train or evaluate your models.

So what more can you ask for? Well, this book! It was written by open source developers at Hugging Face—including the creator of the Transformers library!—and it shows: the breadth and depth of the information you will find in these pages is astounding. It covers everything from the Transformer architecture itself, to the Transformers library and the entire ecosystem around it. I particularly appreciated the hands-on approach: you can follow along in Jupyter notebooks, and all the code examples are straight to the point and simple to understand. The authors have extensive experience in training very large transformer models, and they provide a wealth of tips and tricks for getting everything to work efficiently. Last but not least, their writing style is direct and lively: it reads like a novel.

In short, I thoroughly enjoyed this book, and I’m certain you will too. Anyone interested in building products with state-of-the-art language-processing features needs to read it. It’s packed to the brim with all the right brain germs!

Aurélien Géron

November 2021, Auckland, NZ

1 For brain hygiene tips, see CGP Grey’s excellent video on memes.

Preface

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

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

But however useful it is, to gain traction in industry any fancy new method needs tools to make it accessible. The Transformers library and its surrounding ecosystem answered that call by making it easy for practitioners to use, train, and share models. This greatly accelerated the adoption of transformers, and the library is now used by over five thousand organizations. Throughout this book we'll guide you on how to train and optimize these models for practical applications.

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

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

Figure P-1. An example from GitHub Copilot where, given a brief description of the task, the application provides a suggestion for the entire class (everything following `class` is autogenerated)

Who Is This Book For?

This book is written for data scientists and machine learning engineers who may have heard about the recent breakthroughs involving transformers, but are lacking an in-depth guide to

help them adapt these models to their own use cases. The book is not meant to be an introduction to machine learning, and we assume you are comfortable programming in Python and has a basic understanding of deep learning frameworks like PyTorch and TensorFlow. We also assume you have some practical experience with training models on GPUs. Although the book focuses on the PyTorch API of Transformers, Chapter 2 shows you how to translate all the examples to TensorFlow.

The following resources provide a good foundation for the topics covered in this book. We assume your technical knowledge is roughly at their level:

- *Hands-On Machine Learning with Scikit-Learn and TensorFlow*, by Aurélien Géron (O'Reilly)
- *Deep Learning for Coders with fastai and PyTorch*, by Jeremy Howard and Sylvain Gugger (O'Reilly)

- *Natural Language Processing with PyTorch*, by Delip Rao and Brian McMahan (O'Reilly)
- *The Hugging Face Course*, by the open source team at Hugging Face

What You Will Learn

The goal of this book is to enable you to build your own language applications. To that end, it focuses on practical use cases, and delves into theory only where necessary. The style of the book is hands-on, and we highly recommend you experiment by running the code examples yourself.

The book covers all the major applications of transformers in NLP by having each chapter (with a few exceptions) dedicated to one task, combined with a realistic use case and dataset. Each chapter also introduces some additional concepts. Here's a high-level overview of the tasks and topics we'll cover:

- Chapter 1, *Hello Transformers*, introduces transformers and puts them into context. It also provides an introduction to the Hugging Face ecosystem.
- Chapter 2, *Text Classification*, focuses on the task of sentiment analysis (a common text classification problem) and introduces the `Trainer` API.
- Chapter 3, *Transformer Anatomy*, dives into the Transformer architecture in more depth, to prepare you for the chapters that follow.
- Chapter 4, *Multilingual Named Entity Recognition*, focuses on the task of identifying entities in texts in multiple languages (a token classification problem).
- Chapter 5, *Text Generation*, explores the ability of transformer models to generate text, and introduces decoding strategies and metrics.
- Chapter 6, *Summarization*, digs into the complex sequence-to-sequence task of text summarization and explores the metrics used for this task.
- Chapter 7, *Question Answering*, focuses on building a review-based question answering system and introduces retrieval with Haystack.
- Chapter 8, *Making Transformers Efficient in Production*, focuses on model performance. We'll look at the task of intent detection (a type of sequence classification problem) and explore techniques such a knowledge distillation, quantization, and pruning.

- Chapter 9, *Dealing with Few to No Labels*, looks at ways to improve model performance in the absence of large amounts of labeled data. We'll build a GitHub issues tagger and explore techniques such as zero-shot classification and data augmentation.
- Chapter 10, *Training Transformers from Scratch*, shows you how to build and train a model for autocompleting Python source code from scratch. We'll look at dataset streaming and large-scale training, and build our own tokenizer.
- Chapter 11, *Future Directions*, explores the challenges transformers face and some of the exciting new directions that research in this area is going into.

Transformers offers several layers of abstraction for using and training transformer models. We'll start with the easy-to-use pipelines that allow us to pass text examples through the models and investigate the predictions in just a few lines of code. Then we'll move on to tokenizers, model classes, and the `Trainer` API, which allow us to train models for our own use cases. Later, we'll show you how to replace the `Trainer` with the Accelerate library, which gives us full control over the training loop and allows us to train large-scale transformers entirely from scratch! Although each chapter is mostly self-contained, the difficulty of the tasks increases in the later chapters. For this reason, we recommend starting with Chapters 1 and 2, before branching off into the topic of most interest.

Besides Transformers and Accelerate, we will also make extensive use of Datasets, which seamlessly integrates with other libraries. Datasets offers similar functionality for data processing as Pandas but is designed from the ground up for tackling large datasets and machine learning.

With these tools, you have everything you need to tackle almost any NLP challenge!

Software and Hardware Requirements

Due to the hands-on approach of this book, we highly recommend that you run the code examples while you read each chapter. Since we're dealing with transformers, you'll need access to a computer with an NVIDIA GPU to train these models. Fortunately, there are several free online options that you can use, including:

- Google Colaboratory
- Kaggle Notebooks
- Paperspace Gradient Notebooks

To run the examples, you'll need to follow the installation guide that we provide in the book's GitHub repository. You can find this guide and the code examples at *https://github.com/nlp-with-transformers/notebooks*.

TIP

We developed most of the chapters using NVIDIA Tesla P100 GPUs, which have 16GB of memory. Some of the free platforms provide GPUs with less memory, so you may need to reduce the batch size when training the models.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/nlp-with-transformers/notebooks*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission.

Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Natural Language Processing with Transformers* by Lewis Tunstall, Leandro von Werra, and Thomas Wolf (O'Reilly). Copyright 2022 Lewis Tunstall, Leandro von Werra, and Thomas Wolf, 978-1-098-10324-8."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

NOTE

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

- O'Reilly Media, Inc.
- 1005 Gravenstein Highway North
- Sebastopol, CA 95472
- 800-998-9938 (in the United States or Canada)
- 707-829-0515 (international or local)
- 707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/nlp-with-transformers*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://youtube.com/oreillymedia*

Acknowledgments

Writing a book about one of the fastest-moving fields in machine learning would not have been possible without the help of many people. We thank the wonderful O'Reilly team, and especially Melissa Potter, Rebecca Novack, and Katherine Tozer for their support and advice. The book has also benefited from amazing reviewers who spent countless hours to provide us with invaluable feedback. We are especially grateful to Luca Perozzi, Hamel Husain, Shabie Iqbal, Umberto Lupo, Malte Pietsch, Timo Möller, and Aurélien Géron for their detailed reviews. We thank Branden Chan at deepset for his help with extending the Haystack library to support the use case in Chapter 7. The beautiful illustrations in this book are due to the amazing Christa Lanz—thank you for making this book extra special. We were also fortunate enough to have the support of the whole Hugging Face team. Many thanks to Quentin Lhoest for answering countless questions on Datasets, to Lysandre Debut for help on everything related to the Hugging Face Hub, Sylvain Gugger for his help with Accelerate, and Joe Davison for his inspiration for Chapter 9 with regard to zero-shot learning. We also thank Sidd Karamcheti and the whole Mistral team for adding stability tweaks for GPT-2 to make Chapter 10 possible. This book was written entirely in Jupyter Notebooks, and we thank Jeremy Howard and Sylvain Gugger for creating delightful tools like fastdoc that made this possible.

Lewis

To Sofia, thank you for being a constant source of support and encouragement—without both, this book would not exist. After a long stretch of writing, we can finally enjoy our weekends again!

Leandro

Thank you Janine, for your patience and encouraging support during this long year with many late nights and busy weekends.

Thomas

I would like to thank first and foremost Lewis and Leandro for coming up with the idea of this book and pushing strongly to produce it in such a beautiful and accessible format. I would also like to thank all the Hugging Face team for believing in the mission of AI as a community effort, and the whole NLP/AI community for building and using the libraries and research we describe in this book together with us.

More than what we build, the journey we take is what really matters, and we have the privilege to travel this path with thousands of community members and readers like you today. Thank you all from the bottom of our hearts.

1 NLP researchers tend to name their creations after characters in *Sesame Street*. We'll explain what all these acronyms mean in Chapter 1.

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.

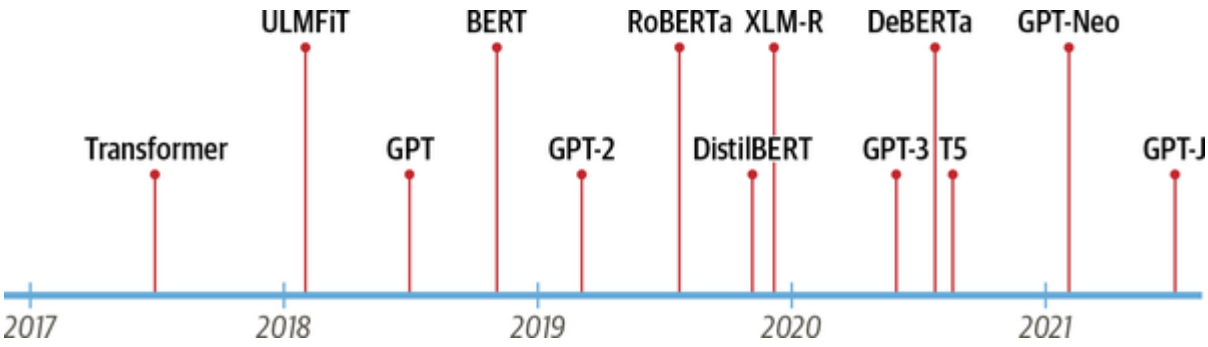


Figure 1-1. The transformers timeline

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.

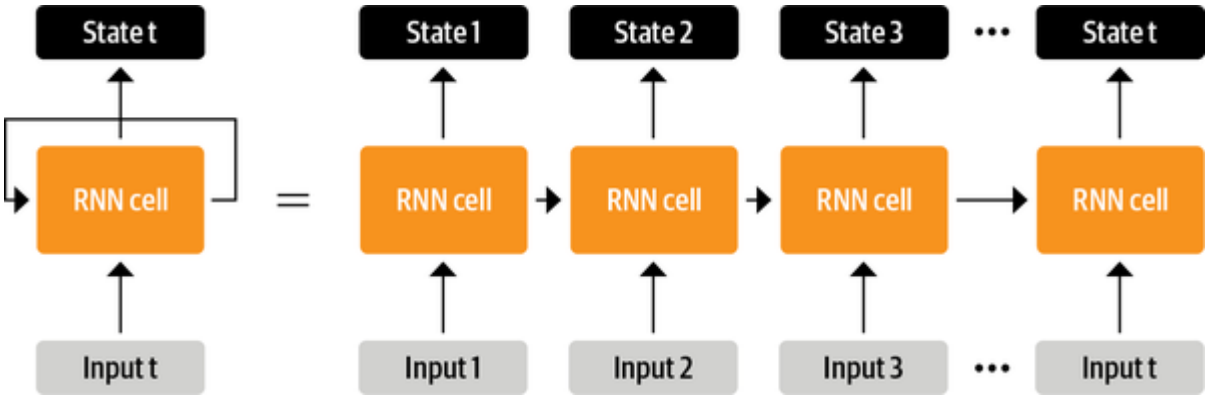


Figure 1-2. Unrolling an RNN in time

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.

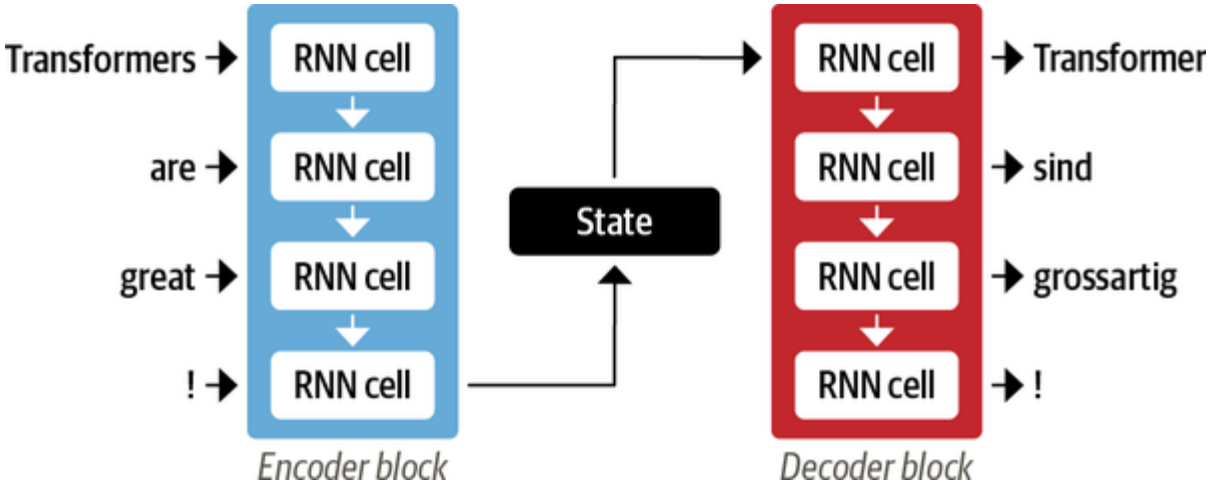


Figure 1-3. An encoder-decoder architecture with a pair of RNNs (in general, there are many more recurrent layers than those shown here)

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.

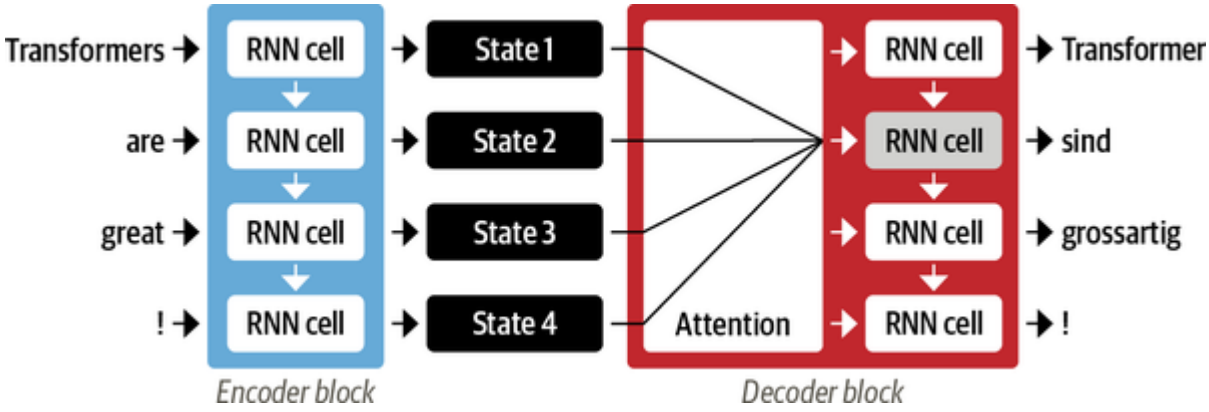


Figure 1-4. An encoder-decoder architecture with an attention mechanism for a pair of RNNs

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.

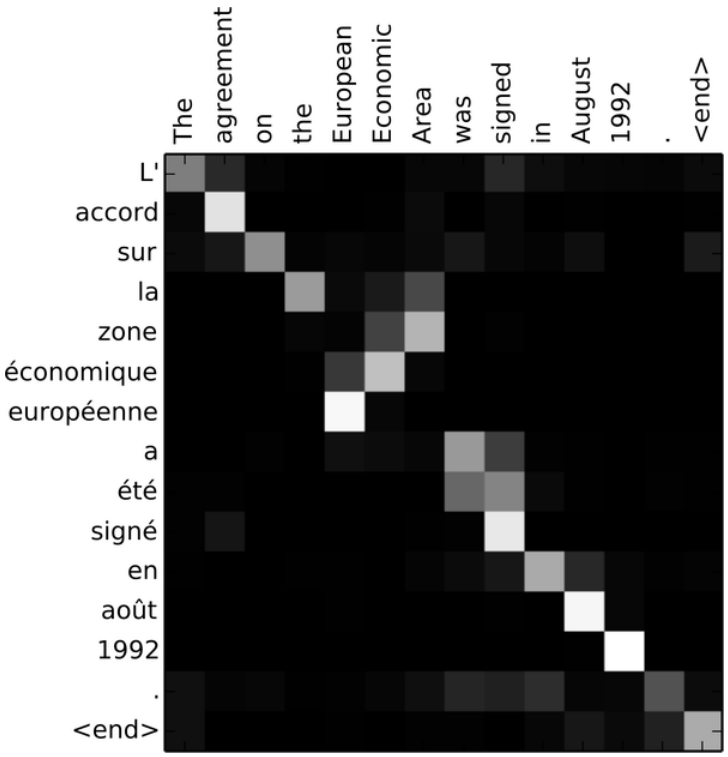


Figure 1-5. RNN encoder-decoder alignment of words in English and the generated translation in French (courtesy of Dzmitry Bahdanau)

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.

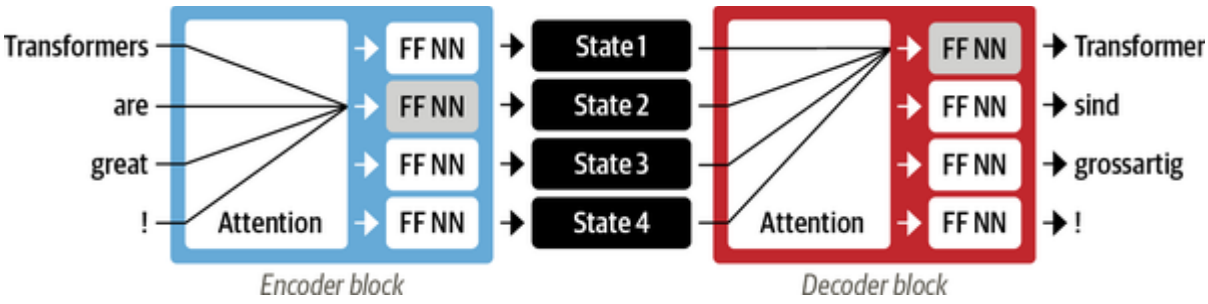


Figure 1-6. Encoder-decoder architecture of the original Transformer

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 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.

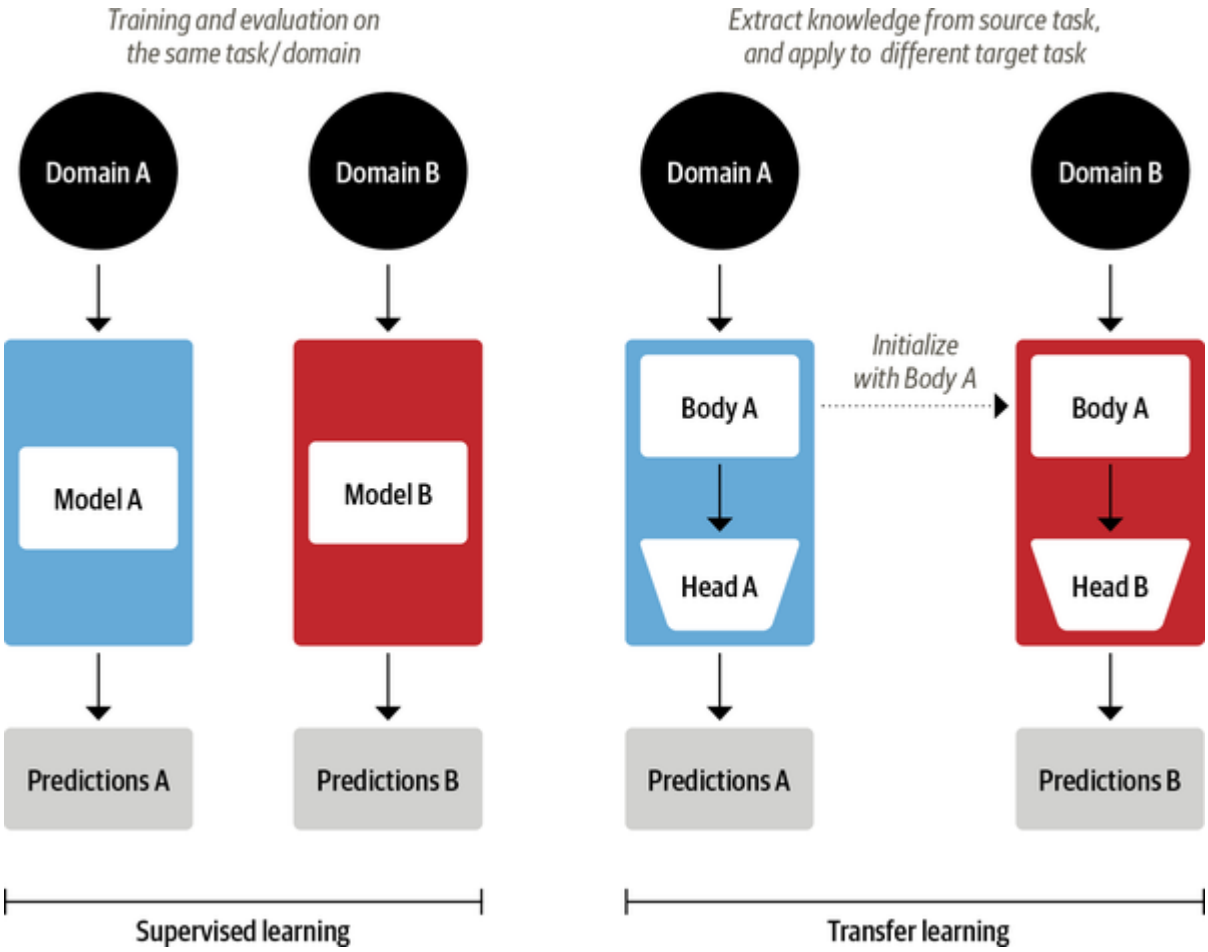


Figure 1-7. Comparison of traditional supervised learning (left) and transfer learning (right)

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).

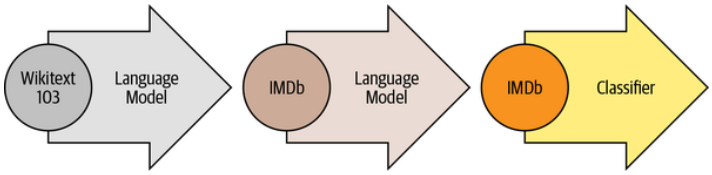


Figure 1-8. The ULMFiT process (courtesy of Jeremy Howard)

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:

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.

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
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.

NOTE

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.

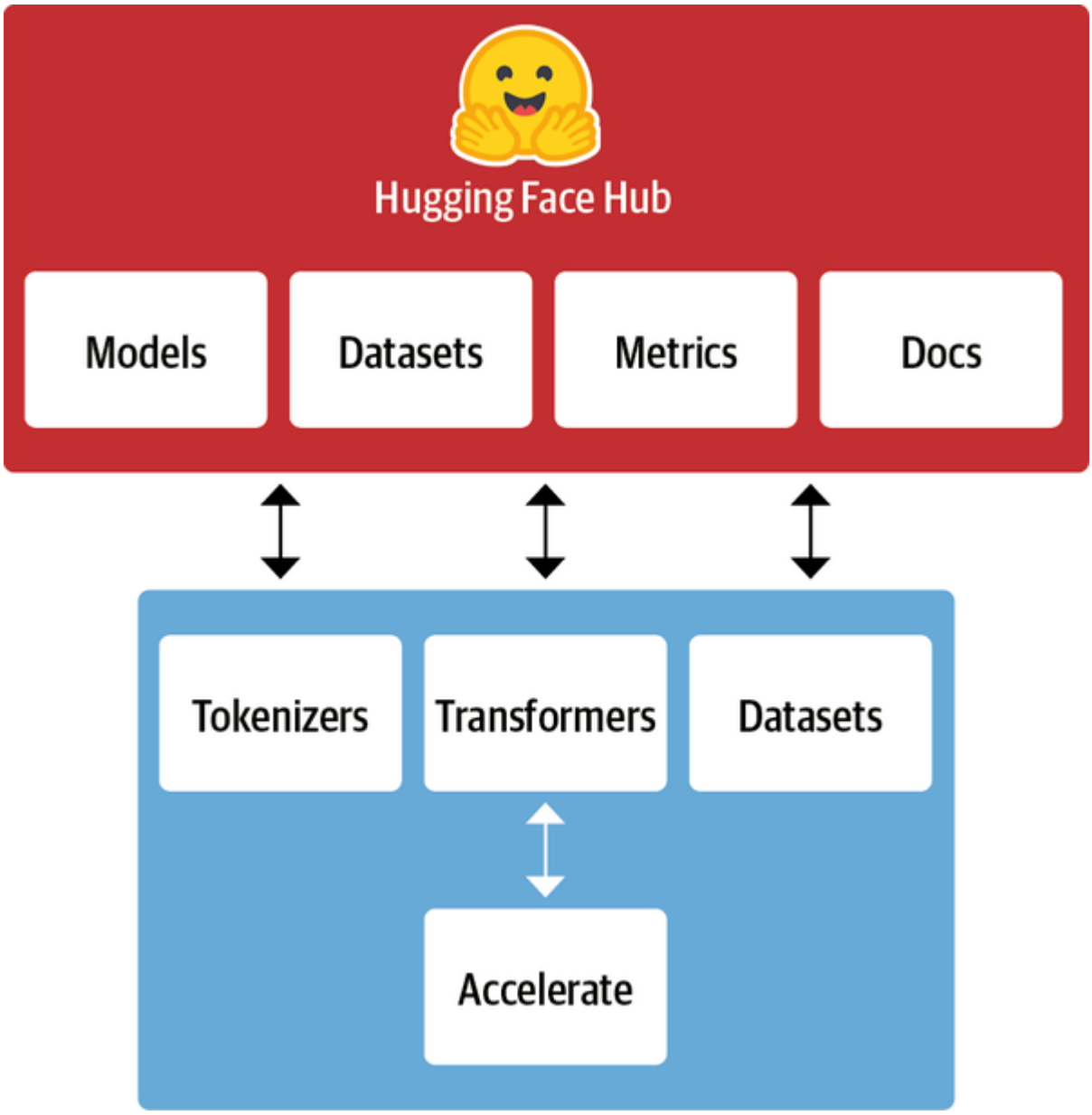


Figure 1-9. An overview of the Hugging Face ecosystem

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.

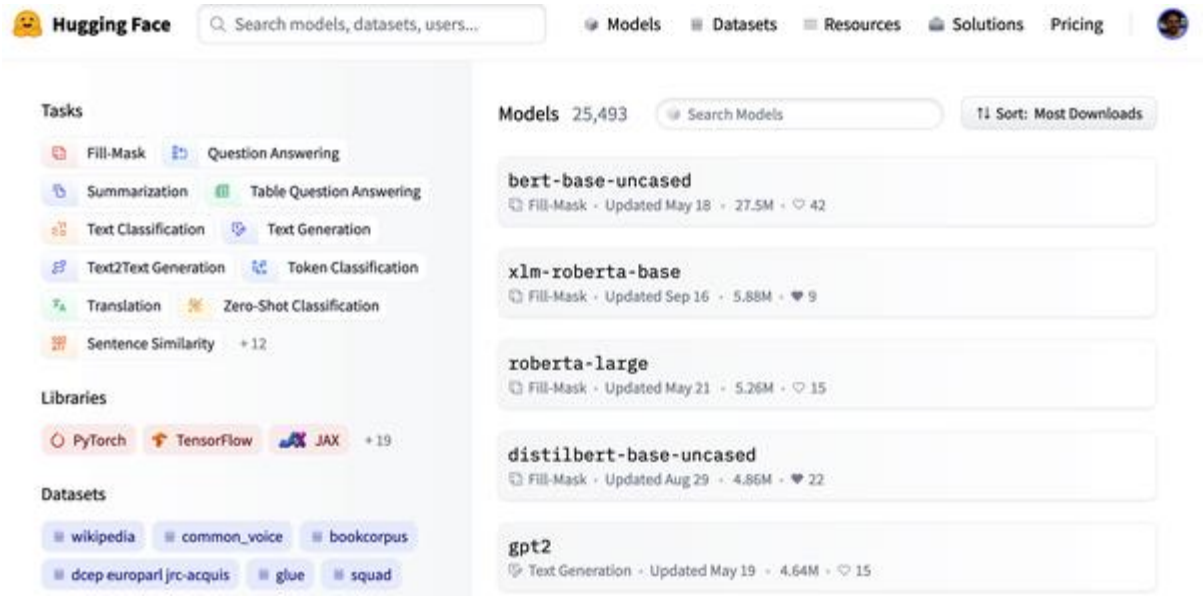


Figure 1-10. The Models page of the Hugging Face Hub, showing filters on the left and a list of models on the right

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.

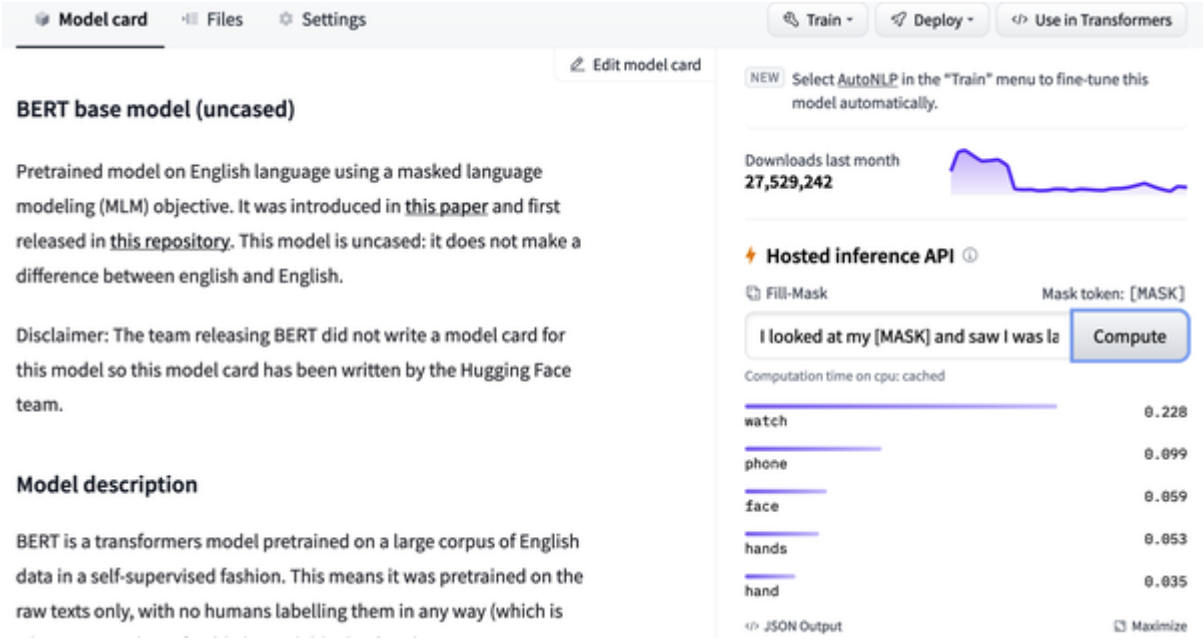


Figure 1-11. An example model card from the Hugging Face Hub: the inference widget, which allows you to interact with the model, is shown on the right

Let's continue our tour with Tokenizers.

NOTE

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.

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!

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).

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

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

7 Weights are the learnable parameters of a neural network.

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.

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

12 Rust is a high-performance programming language.