EXPERT INSIGHT

Transformers for Natural Language Processing

Build, train, and fine-tune deep neural network architectures for NLP with Python, PyTorch, TensorFlow, BERT, and GPT-3

Foreword by:
Antonio Gulli
Engineering Director
Office of the CTO, Google

Second Edition

Denis Rothman

Transformers for Natural Language Processing

Second Edition

Build, train, and fine-tune deep neural network architectures for NLP with Python, PyTorch, TensorFlow, BERT, and GPT-3

Denis Rothman

Packt

BIRMINGHAM—MUMBAI

Transformers for Natural Language Processing

Second Edition

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Producer: Tushar Gupta

Acquisition Editor – Peer Reviews: Saby Dsilva

Project Editor: Janice Gonsalves

Content Development Editor: Bhavesh Amin

Copy Editor: Safis Editing

Technical Editor: Karan Sonawane

Proofreader: Safis Editing

Indexer: Pratik Shirodkar

Presentation Designer: Pranit Padwal

First published: January 2021

Second edition: March 2022

Production reference: 3170322

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-80324-733-5

`www.packt.com`

Foreword

In less than four years, Transformers took the NLP community by storm, breaking any record achieved in the previous 30 years. Models such as BERT, T5, and GPT, now constitute the fundamental building bricks for new applications in everything from computer vision to speech recognition to translation to protein sequencing to writing code. For this reason, Stanford has recently introduced the term *foundation models* to define a set of large language models based on giant pre-trained transformers. All of this progress is thanks to a few simple ideas.

This book is a reference for everyone interested in understanding how transformers work both from a theoretical and from a practical perspective. The author does a tremendous job of explaining how to use transformers step-by-step with a hands-on approach. After reading this book, you will be ready to use this state-of-the-art set of techniques for empowering your deep learning applications. In Particular, this book gives a solid background on the architecture of transformers before covering, in detail, popular models, such as BERT, RoBERTa, T5, and GPT-3. It also explains many use cases (text summarization, image labeling, question-answering, sentiment analysis, and fake news analysis) that transformers can cover.

If these topics interest you, then this is definitely a worthwhile book. The first edition always has a place on my desk, and the same is going to happen with the second edition.

Antonio Gulli
Engineering Director for the Office of the CTO, Google

Contributors

About the author

Denis Rothman graduated from Sorbonne University and Paris Diderot University, designing one of the first patented encoding and embedding systems. He authored one of the first patented AI cognitive robots and bots. He began his career delivering **Natural Language Processing** (**NLP**) chatbots for Moët et Chandon and an **AI tactical defense optimizer** for Airbus (formerly Aerospatiale). Denis then authored an **AI resource optimizer** for IBM and luxury brands, leading to an **Advanced Planning and Scheduling** (**APS**) solution used worldwide

I want to thank the corporations that trusted me from the start to deliver artificial intelligence solutions and shared the risks of continuous innovation. I also want to thank my family, who always believed I would make it.

About the reviewer

George Mihaila is a Ph.D. candidate at the University of North Texas in the Department of Computer Science, where he also got his master's degree in computer science. He received his bachelor's degree in electrical engineering in his home country, Romania.

He worked for 10 months at TCF Bank, where he helped put together the machine learning operation framework for automatic model deployment and monitoring. He did three internships for State Farm as a data scientist and machine learning engineer. He worked as a data scientist and machine learning engineer for the University of North Texas' High-Performance Computing Center for 2 years. He has been working in the research field of natural language processing for 5 years, with the last 3 years spent working with transformer models. His research interests are in dialogue generation with persona.

He was a technical reviewer for the first edition of **Transformers for Natural Language Processing** by *Denis Rothman*.

He is currently working toward his doctoral thesis in casual dialog generation with persona.

In his free time, George likes to share his knowledge of state-of-the-art language models with tutorials and articles and help other researchers in the field of NLP.

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the authors:

```
https://www.packt.link/Transformers
```

Table of Contents

Chapter 3: Fine-Tuning BERT Models 61

Chapter 6: Machine Translation with the Transformer 145

Chapter 14: Interpreting Black Box Transformer Models 367

Appendix V — Answers to the Questions 495

Other Books You May Enjoy 515

Index 519

Preface

Transformers are a game-changer for **Natural Language Understanding** (**NLU**), a subset of **Natural Language Processing** (**NLP**), which has become one of the pillars of artificial intelligence in a global digital economy.

Transformer models mark the beginning of a new era in artificial intelligence. Language understanding has become the pillar of language modeling, chatbots, personal assistants, question answering, text summarizing, speech-to-text, sentiment analysis, machine translation, and more. We are witnessing the expansion of social networks versus physical encounters, e-commerce versus physical shopping, digital newspapers, streaming versus physical theaters, remote doctor consultations versus physical visits, remote work instead of on-site tasks, and similar trends in hundreds of more domains. It would be incredibly difficult for society to use web browsers, streaming services, and any digital activity involving language without AI language understanding. The paradigm shift of our societies from physical to massive digital information forced artificial intelligence into a new era. Artificial intelligence has evolved to billion-parameter models to face the challenge of trillion-word datasets.

The Transformer architecture is both revolutionary and disruptive. It breaks with the past, leaving the dominance of RNNs and CNNs behind. BERT and GPT models abandoned recurrent network layers and replaced them with self-attention. Transformer models outperform RNNs and CNNs. The 2020s are experiencing a major change in AI.

Transformer encoders and decoders contain attention heads that train separately, parallelizing cutting-edge hardware. Attention heads can run on separate GPUs opening the door to billion-parameter models and soon-to-come trillion-parameter models. OpenAI trained a 175 billion parameter GPT-3 Transformer model on a supercomputer with 10,000 GPUs and 285,000 CPU cores.

The increasing amount of data requires training AI models at scale. As such, transformers pave the way to a new era of parameter-driven AI. Learning to understand how hundreds of millions of words fit together in sentences requires a tremendous amount of parameters.

Transformer models such as Google BERT and OpenAI GPT-3 have taken emergence to another level. Transformers can perform hundreds of NLP tasks they were not trained for.

Transformers can also learn image classification and reconstruction by embedding images as sequences of words. This book will introduce you to cutting-edge computer vision transformers such as **Vision Transformers (ViT)**, CLIP, and DALL-E.

Foundation models are fully trained transformer models that can carry out hundreds of tasks without fine-tuning. Foundation models at this scale offer the tools we need in this massive information era.

Think of how many humans it would take to control the content of the billions of messages posted on social networks per day to decide if they are legal and ethical before extracting the information they contain.

Think of how many humans would be required to translate the millions of pages published each day on the web. Or imagine how many people it would take to manually control the millions of messages made per minute!

Finally, think of how many humans it would take to write the transcripts of all of the vast amount of hours of streaming published per day on the web. Finally, think about the human resources required to replace AI image captioning for the billions of images that continuously appear online.

This book will take you from developing code to prompt design, a new "programming" skill that controls the behavior of a transformer model. Each chapter will take you through the key aspects of language understanding from scratch in Python, PyTorch, and TensorFlow.

You will learn the architecture of the original Transformer, Google BERT, OpenAI GPT-3, T5, and several other models. You will fine-tune transformers, train models from scratch, and learn to use powerful APIs. Facebook, Google, Microsoft, and other big tech corporations share large datasets for us to explore.

You will keep close to the market and its demand for language understanding in many fields such as media, social media, and research papers, for example. Among hundreds of AI tasks, we need to summarize the vast amounts of data for research, translate documents for every area of our economy, and scan all social media posts for ethical and legal reasons.

Throughout the book, you will work hands-on with Python, PyTorch, and TensorFlow. You will be introduced to the key AI language understanding neural network models. You will then learn how to explore and implement transformers.

You will learn the new skills required to become an Industry 4.0 AI Specialist in this disruptive AI era. The book aims to give readers the knowledge and tools for Python deep learning needed for effectively developing the key aspects of language understanding.

Who this book is for

This book is not an introduction to Python programming or machine learning concepts. Instead, it focuses on deep learning for machine translations, speech-to-text, text-to-speech, language modeling, question answering, and many more NLP domains.

Readers who can benefit the most from this book are:

- Deep learning and NLP practitioners who are familiar with Python programming.
- Data analysts and data scientists who want an introduction to AI language understanding to process the increasing amounts of language-driven functions.

What this book covers

Part I: Introduction to Transformer Architectures

Chapter 1, *What are Transformers?*, explains, at a high level, what transformers are. We'll look at the transformer ecosystem and the properties of foundation models. The chapter highlights many of the platforms available and the evolution of Industry 4.0 AI specialists.

Chapter 2, *Getting Started with the Architecture of the Transformer Model*, goes through the background of NLP to understand how RNN, LSTM, and CNN deep learning architectures evolved into the Transformer architecture that opened a new era. We will go through the Transformer's architecture through the unique *Attention Is All You Need* approach invented by the Google Research and Google Brain authors. We will describe the theory of transformers. We will get our hands dirty in Python to see how the multi-attention head sub-layers work. By the end of this chapter, you will have understood the original architecture of the Transformer. You will be ready to explore the multiple variants and usages of the Transformer in the following chapters.

Chapter 3, *Fine-Tuning BERT Models*, builds on the architecture of the original Transformer. **Bidirectional Encoder Representations from Transformers** (**BERT**) shows you a new way of perceiving the world of NLP. Instead of analyzing a past sequence to predict a future sequence, BERT attends to the whole sequence! We will first go through the key innovations of BERT's architecture and then fine-tune a BERT model by going through each step in a Google Colaboratory notebook. Like humans, BERT can learn tasks and perform other new ones without having to learn the topic from scratch.

Chapter 4, *Pretraining a RoBERTa Model from Scratch*, builds a RoBERTa transformer model from scratch using the Hugging Face PyTorch modules. The transformer will be both BERT-like and DistilBERT-like. First, we will train a tokenizer from scratch on a customized dataset. The trained transformer will then run on a downstream masked language modeling task.

Part II: Applying Transformers for Natural Language Understanding and Generation

Chapter 5, *Downstream NLP Tasks with Transformers*, reveals the magic of transformer models with downstream NLP tasks. A pretrained transformer model can be fine-tuned to solve a range of NLP tasks such as BoolQ, CB, MultiRC, RTE, WiC, and more, dominating the GLUE and SuperGLUE leaderboards. We will go through the evaluation process of transformers, the tasks, datasets, and metrics. We will then run some of the downstream tasks with Hugging Face's pipeline of transformers.

Chapter 6, *Machine Translation with the Transformer*, defines machine translation to understand how to go from human baselines to machine transduction methods. We will then preprocess a WMT French-English dataset from the European Parliament. Machine translation requires precise evaluation methods, and in this chapter, we explore the BLEU scoring method. Finally, we will implement a Transformer machine translation model with Trax.

Chapter 7, *The Rise of Suprahuman Transformers with GPT-3 Engines*, explores many aspects of OpenAI's GPT-2 and GPT-3 transformers. We will first examine the architecture of OpenAI's GPT models before explaining the different GPT-3 engines. Then we will run a GPT-2 345M parameter model and interact with it to generate text. Next, we'll see the GPT-3 playground in action before coding a GPT-3 model for NLP tasks and comparing the results to GPT-2.

Chapter 8, *Applying Transformers to Legal and Financial Documents for AI Text Summarization*, goes through the concepts and architecture of the T5 transformer model. We will initialize a T5 model from Hugging Face to summarize documents. We will task the T5 model to summarize various documents, including a sample from the *Bill of Rights*, exploring the successes and limitations of transfer learning approaches applied to transformers. Finally, we will use GPT-3 to summarize some corporation law text to a second-grader.

Chapter 9, *Matching Tokenizers and Datasets*, analyzes the limits of tokenizers and looks at some of the methods applied to improve the data encoding process's quality. We will first build a Python program to investigate why some words are omitted or misinterpreted by word2vector tokenizers. Following this, we find the limits of pretrained tokenizers with a tokenizer-agonistic method.

We will improve a T5 summary by applying some of the ideas that show that there is still much room left to improve the methodology of the tokenization process. Finally, we will test the limits of GPT-3's language understanding.

Chapter 10, *Semantic Role Labeling with BERT-Based Transformers*, explores how transformers learn to understand a text's content. **Semantic Role Labeling** (**SRL**) is a challenging exercise for a human. Transformers can produce surprising results. We will implement a BERT-based transformer model designed by the Allen Institute for AI in a Google Colab notebook. We will also use their online resources to visualize SRL outputs. Finally, we will question the scope of SRL and understand the reasons behind its limitations.

Part III: Advanced Language Understanding Techniques

Chapter 11, *Let Your Data Do the Talking: Story, Questions, and Answers*, shows how a transformer can learn how to reason. A transformer must be able to understand a text, a story, and also display reasoning skills. We will see how question answering can be enhanced by adding NER and SRL to the process. We will build the blueprint for a question generator that can be used to train transformers or as a stand-alone solution.

Chapter 12, *Detecting Customer Emotions to Make Predictions*, shows how transformers have improved sentiment analysis. We will analyze complex sentences using the Stanford Sentiment Treebank, challenging several transformer models to understand not only the structure of a sequence but also its logical form. We will see how to use transformers to make predictions that trigger different actions depending on the sentiment analysis output. The chapter finishes with some edge cases using GPT-3.

Chapter 13, *Analyzing Fake News with Transformers*, delves into the hot topic of fake news and how transformers can help us understand the different perspectives of the online content we see each day. Every day, billions of messages, posts, and articles are published on the web through social media, websites, and every form of real-time communication available. Using several techniques from the previous chapters, we will analyze debates on climate change and gun control and the Tweets from a former president. We will go through the moral and ethical problem of determining what can be considered fake news beyond reasonable doubt and what news remains subjective.

Chapter 14, *Interpreting Black Box Transformer Models*, lifts the lid on the black box that is transformer models by visualizing their activity. We will use BertViz to visualize attention heads and **Language Interpretability Tool** (**LIT**) to carry out a **principal component analysis** (**PCA**). Finally, we will use LIME to visualize transformers via dictionary learning.

Chapter 15, *From NLP to Task-Agnostic Transformer Models*, delves into the advanced models, Reformer and DeBERTa, running examples using Hugging Face. Transformers can process images as sequences of words. We will also look at different vision transformers such as ViT, CLIP, and DALL-E. We will test them on computer vision tasks, including generating computer images.

Chapter 16, *The Emergence of Transformer-Driven Copilots*, explores the maturity of Industry 4.0. The chapter begins with prompt engineering examples using informal/casual English. Next, we will use GitHub Copilot and OpenAI Codex to create code from a few lines of instructions. We will see that vision transformers can help NLP transformers visualize the world around them. We will create a transformer-based recommendation system, which can be used by digital humans in whatever metaverse you may end up in!

Appendix I, *Terminology of Transformer Models*, examines the high-level structure of a transformer, from stacks and sublayers to attention heads.

Appendix II, *Hardware Constraints for Transformer Models*, looks at CPU and GPU performance running transformers. We will see why transformers and GPUs and transformers are a perfect match, concluding with a test using Google Colab CPU, Google Colab Free GPU, and Google Colab Pro GPU.

Appendix III, *Generic Text Completion with GPT-2*, provides a detailed explanation of generic text completion using GPT-2 from *Chapter 7*, *The Rise of Suprahuman Transformers with GPT-3 Engines*.

Appendix IV, *Custom Text Completion with GPT-2*, supplements *Chapter 7*, *The Rise of Suprahuman Transformers with GPT-3 Engines* by building and training a GPT-2 model and making it interact with custom text.

Appendix V, *Answers to the Questions*, provides answers to the questions at the end of each chapter.

To get the most out of this book

Most of the programs in the book are Colaboratory notebooks. All you will need is a free Google Gmail account, and you will be able to run the notebooks on Google Colaboratory's free VM.

You will need Python installed on your machine for some of the educational programs.

Take the necessary time to read *Chapter 2*, *Getting Started with the Architecture of the Transformer Model* and *Appendix I*, *Terminology of Transformer Models*. *Chapter 2* contains the description of the original Transformer, which is built from building blocks explained in *Appendix I*, *Terminology of Transformer Models*, that will be implemented throughout the book. If you find it difficult, then pick up the general intuitive ideas out of the chapter. You can then go back to these chapters when you feel more comfortable with transformers after a few chapters.

After reading each chapter, consider how you could implement transformers for your customers or use them to move up in your career with novel ideas.

> Please note that we use OpenAI Codex later on in the book, which currently has a waiting list. Sign up now to avoid a long wait time at `https://openai.com/blog/openai-codex/`.

Download the example code files

The code bundle for the book is hosted on GitHub at `https://github.com/Denis2054/Transformers-for-NLP-2nd-Edition`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that contains color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781803247335_ColorImages.pdf`.

Conventions used

There are several text conventions used throughout this book.

`CodeInText`: Indicates sentences and words run through the models in the book, code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example, "However, if you wish to explore the code, you will find it in the Google Colaboratory `positional_encoding.ipynb` notebook and the `text.txt` file in this chapter's GitHub repository."

A block of code is set as follows:

```
import numpy as np
from scipy.special import softmax
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
The black cat sat on the couch and the brown dog slept on the rug.
```

Any command-line input or output is written as follows:

```
vector similarity
[[0.9627094]] final positional encoding similarity
```

Bold: Indicates a new term, an important word, or words that you see on the screen, for example, in menus or dialog boxes, also appear in the text like this. For example: "In our case, we are looking for **t5-large**, a t5-large model we can smoothly run in Google Colaboratory."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit, `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packtpub.com`.

Share your thoughts

Once you've read *Transformers for Natural Language Processing - Second Edition*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

1

What are Transformers?

Transformers are industrialized, homogenized post-deep learning models designed for parallel computing on supercomputers. Through homogenization, one transformer model can carry out a wide range of tasks with no fine-tuning. Transformers can perform self-supervised learning on billions of records of raw unlabeled data with billions of parameters.

These particular architectures of post-deep learning are called **foundation models**. Foundation model transformers represent the epitome of the Fourth Industrial Revolution that began in 2015 with machine-to-machine automation that will connect everything to everything. Artificial intelligence in general and specifically **Natural Language Processing** (**NLP**) for **Industry 4.0** (**I4.0**) has gone far beyond the software practices of the past.

In less than five years, AI has become an effective cloud service with seamless APIs. The former paradigm of downloading libraries and developing is becoming an educational exercise in many cases.

An Industry 4.0 project manager can go to OpenAI's cloud platform, sign up, obtain an API key, and get to work in a few minutes. A user can then enter a text, specify the NLP task, and obtain a response sent by a GPT-3 transformer engine. Finally, a user can go to GPT-3 Codex and create applications with no knowledge of programming. Prompt engineering is a new skill that emerged from these models.

However, sometimes a GPT-3 model might not fit a specific task. For example, a project manager, consultant, or developer might want to use another system provided by Google AI, **Amazon Web Services** (**AWS**), the Allen Institute for AI, or Hugging Face.

Should a project manager choose to work locally? Or should the implementation be done directly on Google Cloud, Microsoft Azure, or AWS? Should a development team select Hugging Face, Google Trax, OpenAI, or AllenNLP? Should an artificial intelligence specialist or a data scientist use an API with practically no AI development?

The answer is *all* the above. You do not know what a future employer, customer, or user may want or specify. Therefore, you must be ready to adapt to any need that comes up. This book does not describe all the offers that exist on the market. However, this book provides the reader with enough solutions to adapt to Industry 4.0 AI-driven NLP challenges.

This chapter first explains what transformers are at a high level. Then the chapter explains the importance of acquiring a flexible understanding of all types of methods to implement transformers. The definition of platforms, frameworks, libraries, and languages is blurred by the number of APIs and automation available on the market.

Finally, this chapter introduces the role of an Industry 4.0 AI specialist with advances in embedded transformers.

We need to address these critical notions before starting our journey to explore the variety of transformer model implementations described in this book.

This chapter covers the following topics:

- The emergence of the Fourth Industrial Revolution, Industry 4.0
- The paradigm change of foundation models
- Introducing prompt engineering, a new skill
- The background of transformers
- The challenges of implementing transformers
- The game-changing transformer model APIs
- The difficulty of choosing a transformer library
- The difficulty of choosing a transformer model
- The new role of an Industry 4.0 artificial intelligence specialist
- Embedded transformers

Our first step will be to explore the ecosystem of transformers.

The ecosystem of transformers

Transformer models represent such a paradigm change that they require a new name to describe them: **foundation models**. Accordingly, Stanford University created the **Center for Research on Foundation Models** (**CRFM**). In August 2021, the CRFM published a two-hundred-page paper (see the *References* section) written by over one hundred scientists and professionals: *On the Opportunities and Risks of Foundation Models*.

Foundation models were not created by academia but by the big tech industry. For example, Google invented the transformer model, which led to Google BERT. Microsoft entered a partnership with OpenAI to produce GPT-3.

Big tech had to find a better model to face the exponential increase of petabytes of data flowing into their data centers. Transformers were thus born out of necessity.

Let's first take Industry 4.0 into consideration to understand the need to have industrialized artificial intelligence models.

Industry 4.0

The Agricultural Revolution led to the First Industrial Revolution, which introduced machinery. The Second Industrial Revolution gave birth to electricity, the telephone, and airplanes. The Third Industrial Revolution was digital.

The Fourth Industrial Revolution, or Industry 4.0, has given birth to an unlimited number of machine to machine connections: bots, robots, connected devices, autonomous cars, smartphones, bots that collect data from social media storage, and more.

In turn, these millions of machines and bots generate billions of data records every day: images, sound, words, and events, as shown in *Figure 1.1*:

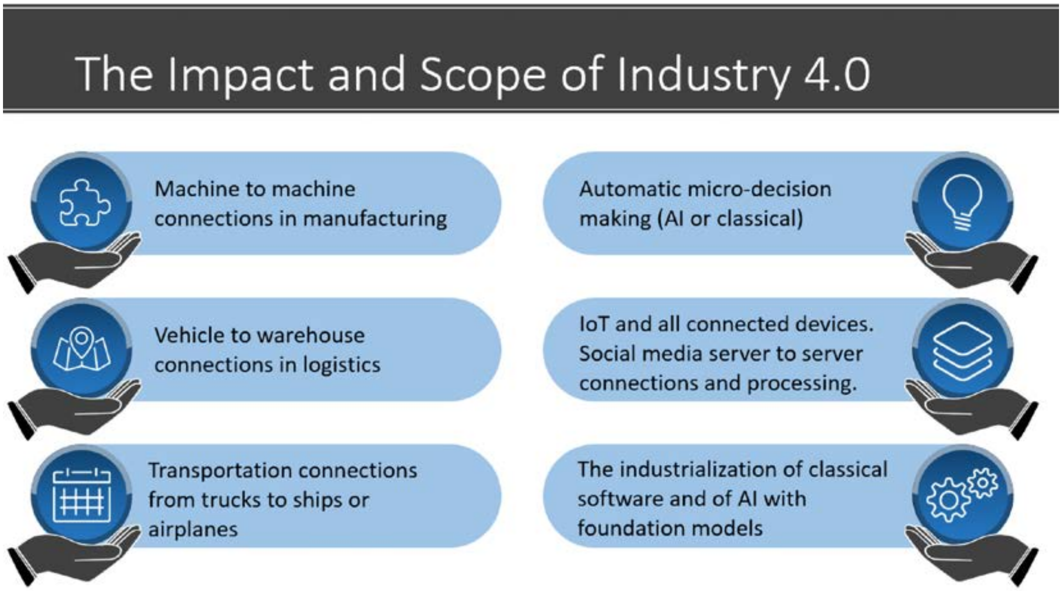

The Impact and Scope of Industry 4.0
Machine to machine connections in manufacturing
Automatic micro-decision making (AI or classical)
Vehicle to warehouse connections in logistics
IoT and all connected devices. Social media server to server connections and processing.
Transportation connections from trucks to ships or airplanes
The industrialization of classical software and of AI with foundation models

Figure 1.1: The scope of Industry 4.0

Industry 4.0 requires intelligent algorithms that process data and make decisions without human intervention on a large scale to face this unseen amount of data in the history of humanity.

Big tech needed to find a single AI model that could perform a variety of tasks that required several separate algorithms in the past.

Foundation models

Transformers have two distinct features: a high level of homogenization and mind-blowing emergence properties. *Homogenization* makes it possible to use one model to perform a wide variety of tasks. These abilities *emerge* through training billion-parameter models on supercomputers.

The paradigm change makes foundation models a post-deep learning ecosystem, as shown in *Figure 1.2*:

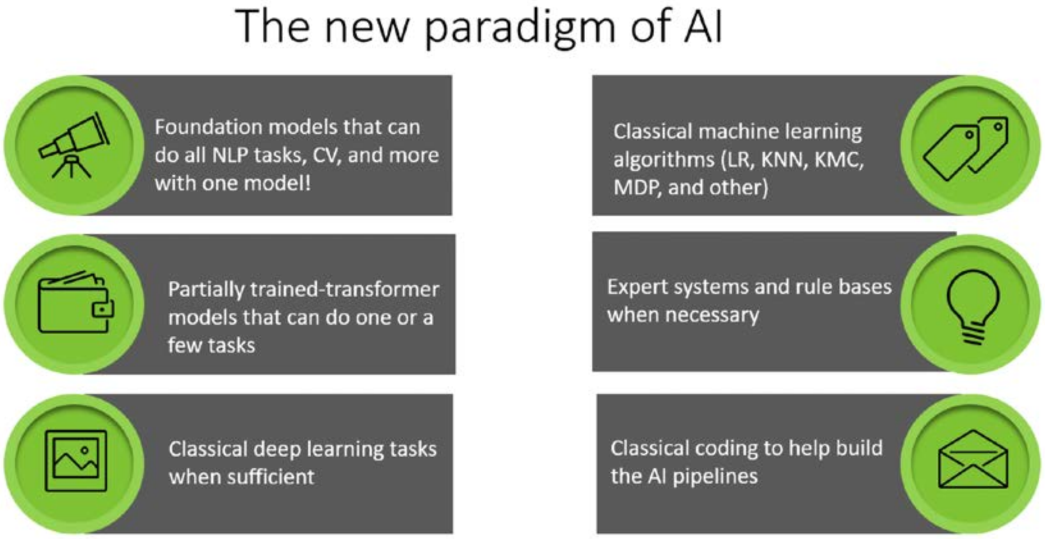

The new paradigm of AI
Foundation models that can do all NLP tasks, CV, and more with one model!
Classical machine learning algorithms (LR, KNN, KMC, MDP, and other)
Partially trained-transformer models that can do one or a few tasks
Expert systems and rule bases when necessary
Classical deep learning tasks when sufficient
Classical coding to help build the AI pipelines

Figure 1.2: The scope of an I4.0 AI specialist

Foundation models, although designed with an innovative architecture, are built on top of the history of AI. As a result, an artificial intelligence specialist's range of skills is stretching!

The present ecosystem of transformer models is unlike any other evolution in artificial intelligence and can be summed up with four properties:

- **Model architecture**

  The model is industrial. The layers of the model are identical, and they are specifically designed for parallel processing. We will go through the architecture of transformers in *Chapter 2*, *Getting Started with the Architecture of the Transformer Model*.

- **Data**

  Big tech possesses the hugest data source in the history of humanity, first generated by the Third Industrial Revolution (digital) and boosted to unfathomable sizes by Industry 4.0.

- **Computing power**

  Big tech possesses computer power never seen before at that scale. For example, GPT-3 was trained at about 50 PetaFLOPS/second, and Google now has domain-specific supercomputers that exceed 80 PetaFLOPS/second.

- **Prompt engineering**

  Highly trained transformers can be triggered to do a task with a prompt. The prompt is entered in natural language. However, the words used require some structure, making prompts a metalanguage.

A foundation model is thus a transformer model that has been trained on supercomputers on billions of records of data and billions of parameters. The model can then perform a wide range of tasks with no further fine-tuning. Thus, the scale of foundation models is unique. These fully trained models are often called engines. Only GPT-3, Google BERT, and a handful of transformer engines can thus qualify as foundation models.

> I will only refer to foundation models in this book when mentioning OpenAI's GPT-3 or Google's BERT model. This is because GPT-3 and Google BERT were fully trained on supercomputers. Though interesting and effective for limited use, other models do not reach the homogenization level of foundation models due to the lack of resources.

Let's now explore an example of how foundation models work and have changed the way we develop programs.

Is programming becoming a sub-domain of NLP?

Chen et al. (2021) published a bombshell paper in August 2021 on Codex, a GPT-3 model that can convert natural language into source code. Codex was trained on 54 million public GitHub software repositories. Codex can produce interesting natural language to source code, as we will see in *Chapter 16*, *The Emergence of Transformer-Driven Copilots*.

Is programming now a translation task from natural language to source code languages?

Is programming becoming an NLP task for GPT-3 engines?

Let's look into an example before answering that question.

Bear in mind that Codex is a stochastic algorithm, so the metalanguage is tricky. You might not generate what you expect if you are not careful to engineer the prompt correctly.

I created some prompts as I was experimenting with Codex. This example is just to give an idea of how Codex works and is purely for educational purposes.

My prompts were:

- “generate a random distribution of 200 integers between 1 and 100” in Python
- “plot the data using matplotlib”
- “create a k-means clustering model with 3 centroids and fit the model”
- “print the cluster labels”
- “plot the clusters”
- “plot the clusters with centroids”

Codex translated my natural metalanguage prompts into Python automatically!

Codex is a stochastic model, so it might not reproduce exactly the same code if you try again. You will have to learn the metalanguage through experimentation until you can drive it like a race car!

The Python program is generated automatically and can be copied and tested:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets.samples_generator import make_blobs

# generate random data
np.random.seed(0)
X, y = make_blobs(n_samples=200, centers=3, n_features=2, cluster_std=2,
random_state=0)

# plot the data
plt.scatter(X[:, 0], X[:, 1], s=50)
plt.show()

# create the k-means model
kmeans = KMeans(n_clusters=3, random_state=0)

# fit the model to the data
kmeans.fit(X)
```

```
# print the cluster labels
print(kmeans.labels_)

#plot the clusters
plt.scatter(X[:, 0], X[:, 1], c=kmeans.labels_, cmap='rainbow')
plt.show()

# plot the clusters with centroids
plt.scatter(X[:, 0], X[:, 1], c=kmeans.labels_, cmap='rainbow')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
c='black', s=100, alpha=0.5)
plt.show()
```

You can copy and paste this program. It works. You can also try JavaScript, among other experiments.

GitHub Copilot is now available with some Microsoft developing tools, as we will see in *Chapter 16*, *The Emergence of Transformer-Driven Copilots*. If you learn the prompt engineering metalanguage, you will reduce your development time in years to come.

End users can create prototypes and or small tasks if they master the metalanguage. In the future, coding copilots will expand.

We will see where Codex fits in the future of artificial intelligence in *Chapter 16*.

At this point, let's take a glimpse into the bright future of artificial intelligence specialists.

The future of artificial intelligence specialists

The societal impact of foundation models should not be underestimated. Prompt engineering has become a skill required for artificial intelligence specialists. However, the future of AI specialists cannot be limited to transformers. AI and data science overlap in I4.0.

An AI specialist will be involved in machine to machine algorithms using classical AI, IoT, edge computing, and more. An AI specialist will also design and develop fascinating connections between bots, robots, servers, and all types of connected devices using classical algorithms.

This book is thus not limited to prompt engineering but to a wide range of design skills required to be an "Industry 4.0 artificial intelligence specialist" or "I4.0 AI specialist."

Prompt engineering is a subset of the design skills an AI specialist will have to develop. In this book, I will thus refer to the future AI specialist as an "Industry 4.0 artificial intelligence specialist."

Let's now get a general view of how transformers optimize NLP models.

Optimizing NLP models with transformers

Recurrent Neural Networks (**RNNs**), including LSTMs, have applied neural networks to NLP sequence models for decades. However, using recurrent functionality reaches its limit when faced with long sequences and large numbers of parameters. Thus, state-of-the-art transformer models now prevail.

This section goes through a brief background of NLP that led to transformers, which we'll describe in more detail in *Chapter 2*, *Getting Started with the Architecture of the Transformer Model*. First, however, let's have an intuitive look at the attention head of a transformer that has replaced the RNN layers of an NLP neural network.

The core concept of a transformer can be summed up loosely as "mixing tokens." NLP models first convert word sequences into tokens. RNNs analyze tokens in recurrent functions. Transformers do not analyze tokens in sequences but relate every token to the other tokens in a sequence, as shown in *Figure 1.3*:

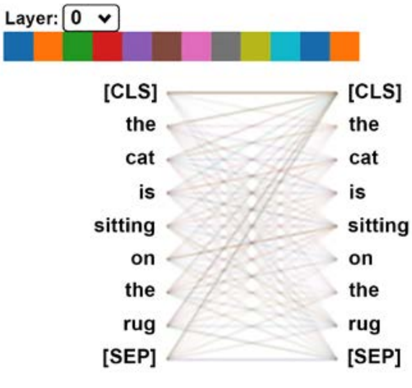


Figure 1.3: An attention head of a layer of a transformer

We will go through the details of an attention head in *Chapter 2*. For the moment, the takeaway of *Figure 1.3* is that *each word* (*token*) *of a sequence is related to all the other words of a sequence*. This model opens the door to Industry 4.0 NLP.

Let's briefly go through the background of transformers.

The background of transformers

Over the past 100+ years, many great minds have worked on sequence patterns and language modeling. As a result, machines progressively learned how to predict probable sequences of words. It would take a whole book to cite all the giants that made this happen.

In this section, I will share some of my favorite researchers with you to lay the grounds for the arrival of the Transformer.

In the early 20th century, Andrey Markov introduced the concept of random values and created a theory of stochastic processes. We know them in AI as the **Markov Decision Process** (**MDP**), **Markov Chains**, and **Markov Processes**. In the early 20th century, Markov showed that we could predict the next element of a chain, a sequence, using only the last past elements of that chain. He applied his method to a dataset containing thousands of letters using past sequences to predict the following letters of a sentence. Bear in mind that he had no computer but proved a theory still in use today in artificial intelligence.

In 1948, Claude Shannon's *The Mathematical Theory of Communication* was published. Claude Shannon laid the grounds for a communication model based on a source encoder, transmitter, and a receiver or semantic decoder. He created information theory as we know it today.

In 1950, Alan Turing published his seminal article: *Computing Machinery and Intelligence*. Alan Turing based this article on machine intelligence on the successful Turing machine, which decrypted German messages during World War II. The messages consisted of sequences of words and numbers.

In 1954, the Georgetown-IBM experiment used computers to translate Russian sentences into English using a rule system. A rule system is a program that runs a list of rules that will analyze language structures. Rule systems still exist and are everywhere. However, in some cases, machine intelligence can replace rule lists for the billions of language combinations by automatically learning the patterns.

The expression "Artificial Intelligence" was first used by John McCarthy in 1956 when it was established that machines could learn.

In 1982, John Hopfield introduced an **RNN**, known as Hopfield networks or "associative" neural networks. John Hopfield was inspired by W.A. Little, who wrote *The existence of persistent states in the brain* in 1974, which laid the theoretical grounds of learning processes for decades. RNNs evolved, and LSTMs emerged as we know them today.

An RNN memorizes the persistent states of a sequence efficiently, as shown in *Figure 1.4*:

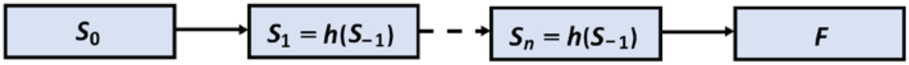


Figure 1.4: The RNN process

Each state S_n captures the information of S_{n-1}. When the network's end is reached, a function F will perform an action: transduction, modeling, or any other type of sequence-based task.

In the 1980s, Yann LeCun designed the multipurpose **Convolutional Neural Network** (**CNN**). He applied CNNs to text sequences, and they also apply to sequence transduction and modeling. They are also based on W.A. Little's persistent states that process information layer by layer. In the 1990s, summing up several years of work, Yann LeCun produced LeNet-5, which led to the many CNN models we know today. However, a CNN's otherwise efficient architecture faces problems when dealing with long-term dependencies in lengthy and complex sequences.

We could mention many other great names, papers, and models that would humble any AI specialist. It seemed that everybody in AI was on the right track for all these years. Markov Fields, RNNs, and CNNs evolved into multiple other models. The notion of attention appeared: peeking at other tokens in a sequence, not just the last one. It was added to the RNN and CNN models.

After that, if AI models needed to analyze longer sequences requiring increasing computer power, AI developers used more powerful machines and found ways to optimize gradients.

Some research was done on sequence-to-sequence models, but they did not meet expectations.

It seemed that nothing else could be done to make more progress. Thirty years passed this way. And then, starting late 2017, the industrialized state-of-the-art Transformer came with its attention head sublayers and more. RNNs did not appear as a pre-requisite for sequence modeling anymore.

Before diving into the original Transformer's architecture, which we will do in *Chapter 2*, *Getting Started with the Architecture of the Transformer Model*, let's start at a high level by examining the paradigm change in software resources we should use to learn and implement transformer models.

What resources should we use?

Industry 4.0 AI has blurred the lines between cloud platforms, frameworks, libraries, languages, and models. Transformers are new, and the range and number of ecosystems are mind-blowing. Google Cloud provides ready-to-use transformer models.

OpenAI has deployed a "Transformer" API that requires practically no programming. Hugging Face provides a cloud library service, and the list is endless.

This chapter will go through a high-level analysis of some of the transformer ecosystems we will be implementing throughout this book.

Your choice of resources to implement transformers for NLP is critical. It is a question of survival in a project. Imagine a real-life interview or presentation. Imagine you are talking to your future employer, your employer, your team, or a customer.

You begin your presentation with an excellent PowerPoint with Hugging Face, for example. You might get an adverse reaction from a manager who may say, "*I'm sorry, but we use Google Trax here for this type of project, not Hugging Face. Can you implement Google Trax, please?*" If you don't, it's game over for you.

The same problem could have arisen by specializing in Google Trax. But, instead, you might get the reaction of a manager who wants to use OpenAI's GPT-3 engines with an API and no development. If you specialize in OpenAI's GPT-3 engines with APIs and no development, you might face a project manager or customer who prefers Hugging Face's AutoML APIs. The worst thing that could happen to you is that a manager accepts your solution, but in the end, it does not work at all for the NLP tasks of that project.

> The key concept to keep in mind is that if you only focus on the solution that you like, you will most likely sink with the ship at some point.
>
> Focus on the system you need, not the one you like.
>
> This book is not designed to explain every transformer solution that exists on the market. Instead, this book aims to explain enough transformer ecosystems for you to be flexible and adapt to any situation you face in an NLP project.

In this section, we will go through some of the challenges that you'll face. But first, let's begin with APIs.

The rise of Transformer 4.0 seamless APIs

We are now well into the industrialization era of artificial intelligence. Microsoft, Google, **Amazon Web Services** (**AWS**), and IBM, among others, offer AI services that no developer or team of developers could hope to outperform. Tech giants have million-dollar supercomputers with massive datasets to train transformer models and AI models in general.

Big tech giants have a wide range of corporate customers that already use their cloud services. As a result, adding a transformer API to an existing cloud architecture requires less effort than any other solution.

A small company or even an individual can access the most powerful transformer models through an API with practically no investment in development. An intern can implement the API in a few days. There is no need to be an engineer or have a Ph.D. for such a simple implementation.

For example, the OpenAI platform now has a **SaaS** (**Software as a Service**) API for some of the most effective transformer models on the market.

OpenAI transformer models are so effective and humanlike that the present policy requires a potential user to fill out a request form. Once the request has been accepted, the user can access a universe of natural language processing!

The simplicity of OpenAI's API takes the user by surprise:

1. Obtain an API key in one click
2. Import OpenAI in a notebook in one lin
3. Enter any NLP task you wish in a *prompt*
4. You will receive the response as a *completion* in a certain number of *tokens* (length)

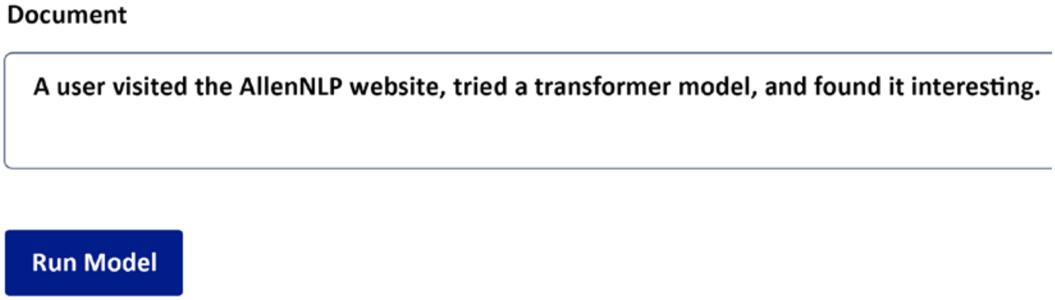
And that's it! Welcome to the Fourth Industrial Revolution and AI 4.0!

Industry 3.0 developers that focus on code-only solutions will evolve into Industry 4.0 developers with cross-disciplinary mindsets.

> The 4.0 developer will learn how to design ways to *show* a transformer model what is expected and not intuitively *tell* it what to do, like a 3.0 developer would do. We will explore this new approach through GPT-2 and GPT-3 models in *Chapter 7*, *The Rise of Suprahuman Transformers with GPT-3 Engines*.

AllenNLP offers the free use of an online educational interface for transformers. AllenNLP also provides a library that can be installed in a notebook. For example, suppose we are asked to implement coreference resolution. We can start by running an example online.

Coreference resolution tasks involve finding the entity to which a word refers, as in the sentence shown in *Figure 1.5*:

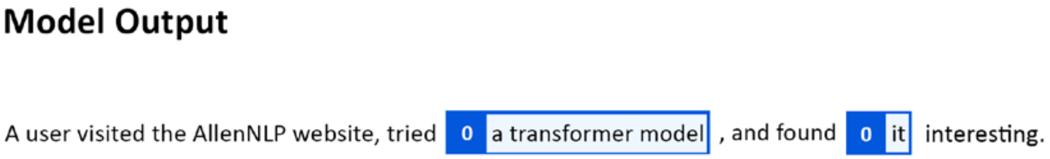


Document

A user visited the AllenNLP website, tried a transformer model, and found it interesting.

Figure 1.5: Running an NLP task online

The word "it" could refer to the website or the transformer model. In this case, the BERT-like model decided to link "it" to the transformer model. AllenNLP provides a formatted output, as shown in *Figure 1.6*:



Model Output

A user visited the AllenNLP website, tried 0 a transformer model , and found 0 it interesting.

Figure 1.6: The output of an AllenNLP transformer model

This example can be run at `https://demo.allennlp.org/coreference-resolution`. Transformer models are continuously updated, so you might obtain a different result.

Though APIs may satisfy many needs, they also have limits. A multipurpose API might be reasonably good in all tasks but not good enough for a specific NLP task. Translating with transformers is no easy task. In that case, a 4.0 developer, consultant, or project manager will have to prove that an API alone cannot solve the specific NLP task required. We need to search for a solid library.

Choosing ready-to-use API-driven libraries

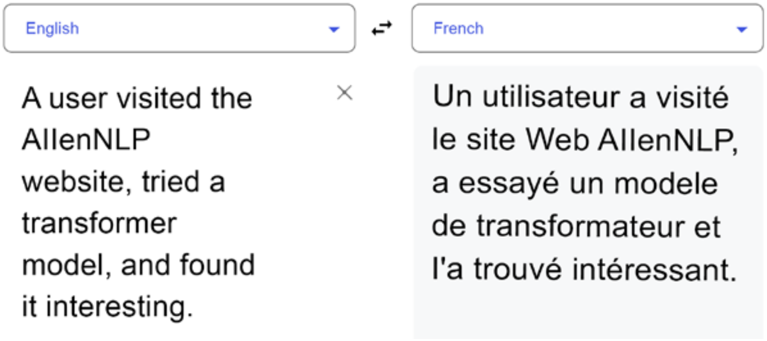
In this book, we will explore several libraries. For example, Google has some of the most advanced AI labs in the world. Google Trax can be installed in a few lines in Google Colab. You can choose free or paid services. We can get our hands on source code, tweak the models, and even train them on our servers or Google Cloud. For example, it's a step down from ready-to-use APIs to customize a transformer model for translation tasks.

However, it can prove to be both educational and effective in some cases. We will explore the recent evolution of Google in translations and implement Google Trax in *Chapter 6*, *Machine Translation with the Transformer*.

We have seen that APIs such as OpenAI require limited developer skills, and libraries such as Google Trax dig a bit deeper into code. Both approaches show that AI 4.0 APIs will require more development on the editor side of the API but much less effort when implementing transformers.

One of the most famous online applications that use transformers, among other algorithms, is Google Translate. Google Translate can be used online or through an API.

Let's try to translate a sentence requiring coreference resolution in an English to French translation using Google Translate:



A user visited the AllenNLP website, tried a transformer model, and found it interesting.

Un utilisateur a visité le site Web AllenNLP, a essayé un modele de transformateur et l'a trouvé intéressant.

Figure 1.7: Coreference resolution in a translation using Google Translate

Google Translate appears to have solved the coreference resolution, but the word *transformateur* in French means an electric device. The word *transformer* is a neologism (new word) in French. An artificial intelligence specialist might be required to have language and linguistic skills for a specific project. Significant development is not required in this case. However, the project might require clarifying the input before requesting a translation.

This example shows that you might have to team up with a linguist or acquire linguistic skills to work on an input context. In addition, it might take a lot of development to enhance the input with an interface for contexts.

So, we still might have to get our hands dirty to add scripts to use Google Translate. Or we might have to find a transformer model for a specific translation need, such as BERT, T5, or other models we will explore in this book.

Choosing a model is no easy task with the increasing range of solutions.

Choosing a Transformer Model

Big tech corporations dominate the NLP market. Google, Facebook, and Microsoft alone run billions of NLP routines per day, increasing their AI models' unequaled power. The big giants now offer a wide range of transformer models and have top-ranking foundation models.

However, smaller companies, spotting the vast NLP market, have entered the game. Hugging Face now has a free or paid service approach too. It will be challenging for Hugging Face to reach the level of efficiency acquired through the billions of dollars poured into Google's research labs and Microsoft's funding of OpenAI. The entry point of foundation models is fully trained transformers on supercomputers such as GPT-3 or Google BERT.

Hugging Face has a different approach and offers a wide range and number of transformer models for a task, which is an interesting philosophy. Hugging Face offers flexible models. In addition, Hugging Face offers high-level APIs and developer-controlled APIs. We will explore Hugging Face in several chapters of this book as an educational tool and a possible solution for specific tasks.

Yet, OpenAI has focused on a handful of the most potent transformer engines globally and can perform many NLP tasks at human levels. We will show the power of OpenAI's GPT-3 engines in *Chapter 7*, *The Rise of Suprahuman Transformers with GPT-3 Engines*.

These opposing and often conflicting strategies leave us with a wide range of possible implementations. We must thus define the role of Industry 4.0 artificial intelligence specialists.

The role of Industry 4.0 artificial intelligence specialists

Industry 4.0 is connecting everything to everything, everywhere. Machines communicate directly with other machines. AI-driven IoT signals trigger automated decisions without human intervention. NLP algorithms send automated reports, summaries, emails, advertisements, and more.

Artificial intelligence specialists will have to adapt to this new era of increasingly automated tasks, including transformer model implementations. Artificial intelligence specialists will have new functions. If we list transformer NLP tasks that an AI specialist will have to do, from top to bottom, it appears that some high-level tasks require little to no development on the part of an artificial intelligence specialist. An AI specialist can be an AI guru, providing design ideas, explanations, and implementations.

The pragmatic definition of what a transformer represents for an artificial intelligence specialist will vary with the ecosystem.

Let's go through a few examples:

- **API**: The OpenAI API does not require an AI developer. A web designer can create a form, and a linguist or Subject Matter Expert (SME) can prepare the prompt input texts. The primary role of an AI specialist will require linguistic skills to show, not just tell, the GPT-3 engines how to accomplish a task. Showing, for example, involves working on the context of the input. This new task is named *prompt engineering*. A *prompt engineer* has quite a future in AI!
- **Library**: The Google Trax library requires a limited amount of development to start with ready-to-use models. An AI specialist mastering linguistics and NLP tasks can work on the datasets and the outputs.
- **Training and fine-tuning**: Some of the Hugging Face functionality requires a limited amount of development, providing both APIs and libraries. However, in some cases, we still have to get our hands dirty. In that case, training, fine-tuning the models, and finding the correct hyperparameters will require the expertise of an artificial intelligence specialist.
- **Development-level skills**: In some projects, the tokenizers and the datasets do not match, as explained in *Chapter 9*, *Matching Tokenizers and Datasets*. In this case, an artificial intelligence developer working with a linguist, for example, can play a crucial role. Therefore, computational linguistics training can come in very handy at this level.

The recent evolution of NLP AI can be termed as "embedded transformers," which is disrupting the AI development ecosystem:

- GPT-3 transformers are currently embedded in several Microsoft Azure applications with GitHub Copilot, for example. As introduced in the *Foundation models* section of this chapter, Codex is another example we will look into in *Chapter 16*, *The Emergence of Transformer-Driven Copilots*.
- The embedded transformers are not accessible directly but provide automatic development support such as automatic code generation.
- The usage of embedded transformers is seamless for the end user with assisted text completion.

To access GPT-3 engines directly, you must first create an OpenAI account. Then you can use the API or directly run examples in the OpenAI user interface.

We will explore this fascinating new world of embedded transformers in *Chapter 16*. But to get the best out of that chapter, you should first master the previous chapters' concepts, examples, and programs.

The skillset of an Industry 4.0 AI specialist requires flexibility, cross-disciplinary knowledge, and above all, *flexibility*. This book will provide the artificial intelligence specialist with a variety of transformer ecosystems to adapt to the new paradigms of the market.

It's time to summarize the ideas of this chapter before diving into the fascinating architecture of the original Transformer in *Chapter 2*.

Summary

The Fourth Industrial Revolution, or Industry 4.0, has forced artificial intelligence to make profound evolutions. The Third Industrial Revolution was digital. Industry 4.0 is built on top of the digital revolution connecting everything to everything, everywhere. Automated processes are replacing human decisions in critical areas, including NLP.

RNNs had limitations that slowed the progression of automated NLP tasks required in a fast-moving world. Transformers filled the gap. A corporation needs summarization, translation, and a wide range of NLP tools to meet the challenges of Industry 4.0.

Industry 4.0 (I4.0) has thus spurred an age of artificial intelligence industrialization. The evolution of the concepts of platforms, frameworks, language, and models represents a challenge for an industry 4.0 developer. Foundation models bridge the gap between the Third Industrial Revolution and I4.0 by providing homogenous models that can carry out a wide range of tasks without further training or fine-tuning.

Websites such as AllenNLP, for example, provide educational NLP tasks with no installation, but it also provides resources to implement a transformer model in customized programs. OpenAI provides an API requiring only a few code lines to run one of the powerful GPT-3 engines. Google Trax provides an end-to-end library, and Hugging Face offers various transformer models and implementations. We will be exploring these ecosystems throughout this book.

Industry 4.0 is a radical deviation from former AI with a broader skillset. For example, a project manager can decide to implement transformers by asking a web designer to create an interface for OpenAI's API through prompt engineering. Or, when required, a project manager can ask an artificial intelligence specialist to download Google Trax or Hugging Face to develop a full-blown project with a customized transformer model.

Industry 4.0 is a game-changer for developers whose role will expand and require more designing than programming. In addition, embedded transformers will provide assisted code development and usage. These new skillsets are a challenge but open new exciting horizons.

In *Chapter 2*, *Getting Started with the Architecture of the Transformer Model*, we will get started with the architecture of the original Transformer.

Questions

1. We are still in the Third Industrial Revolution. (True/False)
2. The Fourth Industrial Revolution is connecting everything to everything. (True/False)
3. Industry 4.0 developers will sometimes have no AI development to do. (True/False)
4. Industry 4.0 developers might have to implement transformers from scratch. (True/False)
5. It's not necessary to learn more than one transformer ecosystem, such as Hugging Face, for example. (True/False)
6. A ready-to-use transformer API can satisfy all needs. (True/False)
7. A company will accept the transformer ecosystem a developer knows best. (True/False)
8. Cloud transformers have become mainstream. (True/False)
9. A transformer project can be run on a laptop. (True/False)
10. Industry 4.0 artificial intelligence specialists will have to be more flexible (True/False)

References

- *Bommansani* et al. 2021, *On the Opportunities and Risks of Foundation Models*, `https://arxiv.org/abs/2108.07258`
- *Chen* et al.,2021, *Evaluating Large Language Models Trained on Code*, `https://arxiv.org/abs/2107.03374`
- Microsoft AI: `https://innovation.microsoft.com/en-us/ai-at-scale`
- OpenAI: `https://openai.com/`
- Google AI: `https://ai.google/`

- Google Trax: `https://github.com/google/trax`
- AllenNLP: `https://allennlp.org/`
- Hugging Face: `https://huggingface.co/`

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the authors:

`https://www.packt.link/Transformers`

2

Getting Started with the Architecture of the Transformer Model

Language is the essence of human communication. Civilizations would never have been born without the word sequences that form language. We now mostly live in a world of digital representations of language. Our daily lives rely on NLP digitalized language functions: web search engines, emails, social networks, posts, tweets, smartphone texting, translations, web pages, speech-to-text on streaming sites for transcripts, text-to-speech on hotline services, and many more everyday functions.

Chapter 1, *What are Transformers?*, explained the limits of RNNs and the birth of cloud AI transformers taking over a fair share of design and development. The role of the Industry 4.0 developer is to understand the architecture of the original Transformer and the multiple transformer ecosystems that followed.

In December 2017, Google Brain and Google Research published the seminal *Vaswani* et al., *Attention is All You Need* paper. The Transformer was born. The Transformer outperformed the existing state-of-the-art NLP models. The Transformer trained faster than previous architectures and obtained higher evaluation results. As a result, transformers have become a key component of NLP.

The idea of the attention head of the Transformer is to do away with recurrent neural network features. In this chapter, we will open the hood of the Transformer model described by *Vaswani* et al. (2017) and examine the main components of its architecture. We will explore the fascinating world of attention and illustrate the key components of the Transformer.

This chapter covers the following topics:

- The architecture of the Transformer
- The Transformer's self-attention model
- The encoding and decoding stacks
- Input and output embedding
- Positional embedding
- Self-attention
- Multi-head attention
- Masked multi-attention
- Residual connections
- Normalization
- Feedforward network
- Output probabilities

Let's dive directly into the structure of the original Transformer's architecture.

The rise of the Transformer: Attention is All You Need

In December 2017, *Vaswani* et al. (2017) published their seminal paper, *Attention is All You Need.* They performed their work at Google Research and Google Brain. I will refer to the model described in *Attention is All You Need* as the "original Transformer model" throughout this chapter and book.

> *Appendix I*, *Terminology of Transformer Models*, can help the transition from the classical usage of deep learning words to transformer vocabulary. *Appendix I* summarizes some of the changes to the classical AI definition of neural network models.

In this section, we will look at the structure of the Transformer model they built. In the following sections, we will explore what is inside each component of the model.

The original Transformer model is a stack of 6 layers. The output of layer *l* is the input of layer *l*+1 until the final prediction is reached. There is a 6-layer encoder stack on the left and a 6-layer decoder stack on the right:

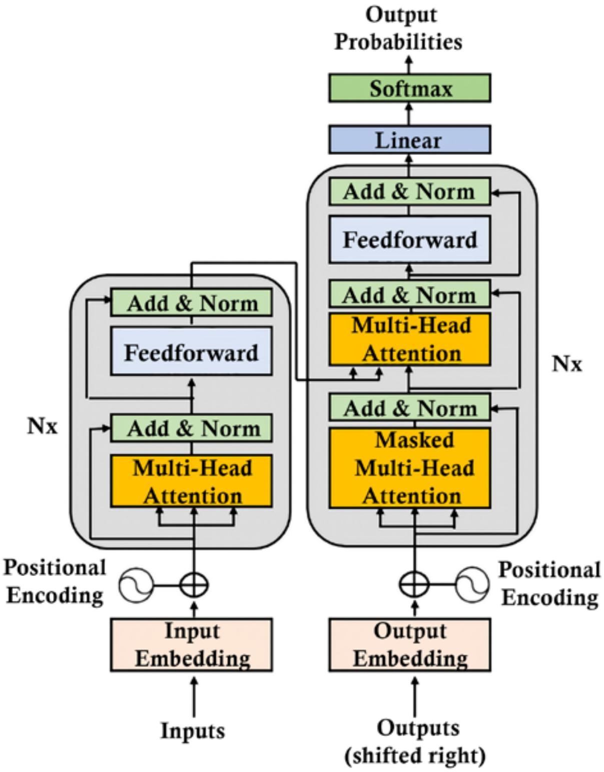


Figure 2.1: The architecture of the Transformer

On the left, the inputs enter the encoder side of the Transformer through an attention sublayer and a feedforward sublayer. On the right, the target outputs go into the decoder side of the Transformer through two attention sublayers and a feedforward network sublayer. We immediately notice that there is no RNN, LSTM, or CNN. Recurrence has been abandoned in this architecture.

Attention has replaced recurrence functions requiring increasing parameters as the distance between two words increases. The attention mechanism is a "word to word" operation. It is actually a token-to-token operation, but we will keep it to the word level to keep the explanation simple. The attention mechanism will find how each word is related to all other words in a sequence, including the word being analyzed itself. Let's examine the following sequence:

```
The cat sat on the mat.
```

Attention will run dot products between word vectors and determine the strongest relationships of a word with all the other words, including itself ("cat" and "cat"):

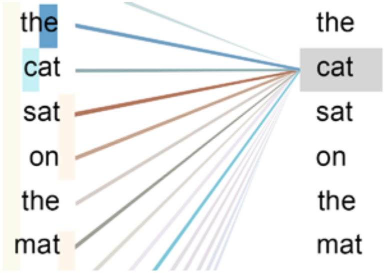


Figure 2.2: Attending to all the words

The attention mechanism will provide a deeper relationship between words and produce better results.

For each attention sublayer, the original Transformer model runs not one but eight attention mechanisms in parallel to speed up the calculations. We will explore this architecture in the following section, *The encoder stack*. This process is named "multi-head attention," providing:

- A broader in-depth analysis of sequences
- The preclusion of recurrence reducing calculation operations
- Implementation of parallelization, which reduces training time
- Each attention mechanism learns different perspectives of the same input sequence

Attention replaced recurrence. However, there are several other creative aspects of the Transformer, which are as critical as the attention mechanism, as you will see when we look inside the architecture.

We just looked at the Transformer structure from the outside. Let's now go into each component of the Transformer. We will start with the encoder.

The encoder stack

The layers of the encoder and decoder of the original Transformer model are *stacks of layers*. Each layer of the encoder stack has the following structure:

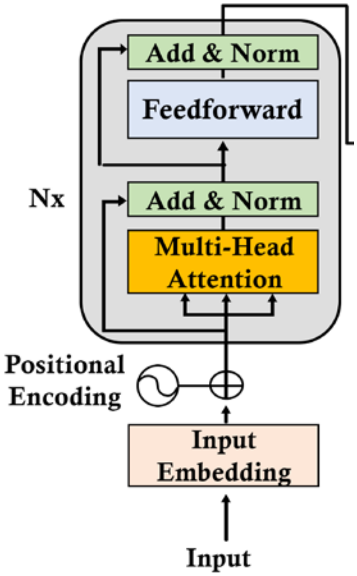


Figure 2.3: A layer of the encoder stack of the Transformer

The original encoder layer structure remains the same for all *N*=6 layers of the Transformer model. Each layer contains two main sublayers: a multi-headed attention mechanism and a fully connected position-wise feedforward network.

Notice that a residual connection surrounds each main sublayer, *sublayer*(*x*), in the Transformer model. These connections transport the unprocessed input *x* of a sublayer to a layer normalization function. This way, we are certain that key information such as positional encoding is not lost on the way. The normalized output of each layer is thus:

$$LayerNormalization\ (x + Sublayer(x))$$

Though the structure of each of the *N*=6 layers of the encoder is identical, the content of each layer is not strictly identical to the previous layer.

For example, the embedding sublayer is only present at the bottom level of the stack. The other five layers do not contain an embedding layer, and this guarantees that the encoded input is stable through all the layers.

Also, the multi-head attention mechanisms perform the same functions from layer 1 to 6. However, they do not perform the same tasks. Each layer learns from the previous layer and explores different ways of associating the tokens in the sequence. It looks for various associations of words, just like we look for different associations of letters and words when we solve a crossword puzzle.

The designers of the Transformer introduced a very efficient constraint. The output of every sublayer of the model has a constant dimension, including the embedding layer and the residual connections. This dimension is d_{model} and can be set to another value depending on your goals. In the original Transformer architecture, $d_{model} = 512$.

d_{model} has a powerful consequence. Practically all the key operations are dot products. As a result, the dimensions remain stable, which reduces the number of operations to calculate, reduces machine consumption, and makes it easier to trace the information as it flows through the model.

This global view of the encoder shows the highly optimized architecture of the Transformer. In the following sections, we will zoom into each of the sublayers and mechanisms.

We will begin with the embedding sublayer.

Input embedding

The input embedding sublayer converts the input tokens to vectors of dimension $d_{model} = 512$ using learned embeddings in the original Transformer model. The structure of the input embedding is classical:

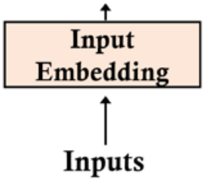


Figure 2.4: The input embedding sublayer of the Transformer

The embedding sublayer works like other standard transduction models. A tokenizer will transform a sentence into tokens. Each tokenizer has its methods, such as BPE, word piece, and sentence piece methods. The Transformer initially used BPE, but other models use other methods.

The goals are similar, and the choice depends on the strategy chosen. For example, a tokenizer applied to the sequence `the Transformer is an innovative NLP model!` will produce the following tokens in one type of model:

```
['the', 'transform', 'er', 'is', 'an', 'innovative', 'n', 'l', 'p',
'model', '!']
```

You will notice that this tokenizer normalized the string to lowercase and truncated it into subparts. A tokenizer will generally provide an integer representation that will be used for the embedding process. For example:

```
text = "The cat slept on the couch.It was too tired to get up."
tokenized text= [1996, 4937, 7771, 2006, 1996, 6411, 1012, 2009, 2001,
2205, 5458, 2000, 2131, 2039, 1012]
```

There is not enough information in the tokenized text at this point to go further. The tokenized text must be embedded.

The Transformer contains a learned embedding sublayer. Many embedding methods can be applied to the tokenized input.

I chose the skip-gram architecture of the `word2vec` embedding approach Google made available in 2013 to illustrate the embedding sublayer of the Transformer. A skip-gram will focus on a center word in a window of words and predicts *context* words. For example, if word(i) is the center word in a two-step window, a skip-gram model will analyze word(i-2), word(i-1), word(i+1), and word(i+2). Then the window will *slide* and repeat the process. A skip-gram model generally contains an input layer, weights, a hidden layer, and an output containing the word embeddings of the tokenized input words.

Suppose we need to perform embedding for the following sentence:

```
The black cat sat on the couch and the brown dog slept on the rug.
```

We will focus on two words, `black` and `brown`. The word embedding vectors of these two words should be similar.

Since we must produce a vector of size d_{model} = 512 for each word, we will obtain a size 512 vector embedding for each word:

```
black=[[-0.01206071  0.11632373  0.06206119  0.01403395  0.09541149
0.10695464 0.02560172  0.00185677 -0.04284821  0.06146432  0.09466285
0.04642421 0.08680347  0.05684567 -0.00717266 -0.03163519  0.03292002
-0.11397766 0.01304929  0.01964396  0.01902409  0.02831945  0.05870414
0.03390711 -0.06204525  0.06173197 -0.08613958 -0.04654748  0.02728105
-0.07830904
    …
0.04340003 -0.13192849 -0.00945092 -0.00835463 -0.06487109  0.05862355
-0.03407936 -0.00059001 -0.01640179  0.04123065
-0.04756588  0.08812257 0.00200338 -0.0931043  -0.03507337  0.02153351
-0.02621627 -0.02492662 -0.05771535 -0.01164199
-0.03879078 -0.05506947  0.01693138 -0.04124579 -0.03779858
-0.01950983 -0.05398201  0.07582296  0.00038318 -0.04639162
-0.06819214  0.01366171  0.01411388  0.00853774  0.02183574
-0.03016279 -0.03184025 -0.04273562]]
```

The word `black` is now represented by 512 dimensions. Other embedding methods could be used and d_{model} could have a higher number of dimensions.

The word embedding of `brown` is also represented by 512 dimensions:

```
brown=[[ 1.35794589e-02 -2.18823571e-02  1.34526128e-02  6.74355254e-02
   1.04376070e-01  1.09921647e-02 -5.46298288e-02 -1.18385479e-02
   4.41223830e-02 -1.84863899e-02 -6.84073642e-02  3.21860164e-02
   4.09143828e-02 -2.74433400e-02 -2.47369967e-02  7.74542615e-02
   9.80964210e-03  2.94299088e-02  2.93895267e-02 -3.29437815e-02
…
  7.20389187e-02  1.57317147e-02 -3.10291946e-02 -5.51304631e-02
  -7.03861639e-02  7.40829483e-02  1.04319192e-02 -2.01565702e-03
   2.43322570e-02  1.92969330e-02  2.57341694e-02 -1.13280728e-01
   8.45847875e-02  4.90090018e-03  5.33546880e-02 -2.31553353e-02
   3.87288055e-05  3.31782512e-02 -4.00604047e-02 -1.02028981e-01
   3.49597558e-02 -1.71501152e-02  3.55573371e-02 -1.77437533e-02
  -5.94457164e-02  2.21221056e-02  9.73121971e-02 -4.90022525e-02]]
```

To verify the word embedding produced for these two words, we can use cosine similarity to see if the word embeddings of the words `black` and `brown` are similar.

Cosine similarity uses Euclidean (L2) norm to create vectors in a unit sphere. The dot product of the vectors we are comparing is the cosine between the points of those two vectors. For more on the theory of cosine similarity, you can consult scikit-learn's documentation, among many other sources: `https://scikit-learn.org/stable/modules/metrics.html#cosine-similarity`.

The cosine similarity between the black vector of size d_{model}= 512 and the brown vector of size d_{model} = 512 in the embedding of the example is:

```
cosine_similarity(black, brown)= [[0.9998901]]
```

The skip-gram produced two vectors that are close to each other. It detected that black and brown form a color subset of the dictionary of words.

The Transformer's subsequent layers do not start empty-handed. They have learned word embeddings that already provide information on how the words can be associated.

However, a big chunk of information is missing because no additional vector or information indicates a word's position in a sequence.

The designers of the Transformer came up with yet another innovative feature: positional encoding.

Let's see how positional encoding works.

Positional encoding

We enter this positional encoding function of the Transformer with no idea of the position of a word in a sequence:

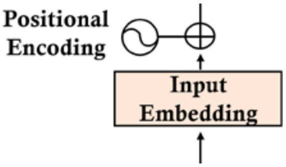


Figure 2.5: Positional encoding

We cannot create independent positional vectors that would have a high cost on the training speed of the Transformer and make attention sublayers overly complex to work with. The idea is to add a positional encoding value to the input embedding instead of having additional vectors to describe the position of a token in a sequence.

> Industry 4.0 is pragmatic and model-agnostic. The original Transformer model has only one vector that contains word embedding and position encoding. We will explore disentangled attention with a separate matrix for positional encoding in *Chapter 15, From NLP to Task-Agnostic Transformer Models*.

The Transformer expects a fixed size d_{model}= 512 (or other constant value for the model) for each vector of the output of the positional encoding function.

If we go back to the sentence we used in the word embedding sublayer, we can see that black and brown may be semantically similar, but they are far apart in the sentence:

```
The black cat sat on the couch and the brown dog slept on the rug.
```

The word `black` is in position 2, `pos=2`, and the word `brown` is in position 10, `pos=10`.

Our problem is to find a way to add a value to the word embedding of each word so that it has that information. However, we need to add a value to the d_{model}= 512 dimensions! For each word embedding vector, we need to find a way to provide information to `i` in the `range(0,512)` dimensions of the word embedding vector of `black` and `brown`.

There are many ways to achieve positional encoding. This section will focus on the designers' clever way to use a unit sphere to represent positional encoding with sine and cosine values that will thus remain small but useful.

Vaswani et al. (2017) provide sine and cosine functions so that we can generate different frequencies for the positional encoding (***PE***) for each position and each dimension *i* of the d_{model}= 512 of the word embedding vector:

$$PE_{(pos\ 2i)} = sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$PE_{(pos\ 2i+1)} = cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

If we start at the beginning of the word embedding vector, we will begin with a constant (512), `i=0`, and end with `i=511`. This means that the sine function will be applied to the even numbers and the cosine function to the odd numbers. Some implementations do it differently. In that case, the domain of the sine function can be $i \in [0,255]$ and the domain of the cosine function can be $i \in [256,512]$. This will produce similar results.

In this section, we will use the functions the way they were described by *Vaswani* et al. (2017). A literal translation into Python pseudo code produces the following code for a positional vector `pe[0][i]` for a position `pos`:

```
def positional_encoding(pos,pe):
for i in range(0, 512,2):
         pe[0][i] = math.sin(pos / (10000 ** ((2 * i)/d_model)))
         pe[0][i+1] = math.cos(pos / (10000 ** ((2 * i)/d_model)))
return pe
```

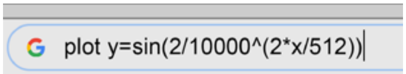
Google Brain Trax and Hugging Face, among others, provide ready-to-use libraries for the word embedding section and the present positional encoding section. Thus, you don't need to run the code I share in this section. However, if you wish to explore the code, you will find it in the Google Colaboratory `positional_encoding.ipynb` notebook and the `text.txt` file in this chapter's GitHub repository.

Before going further, you might want to see the plot of the sine function, for example, for `pos=2`.

You can Google the following plot, for example:

```
plot y=sin(2/10000^(2*x/512))
```

Just enter the plot request:



plot y=sin(2/10000^(2*x/512))

Figure 2.6: Plotting with Google

You will obtain the following graph:

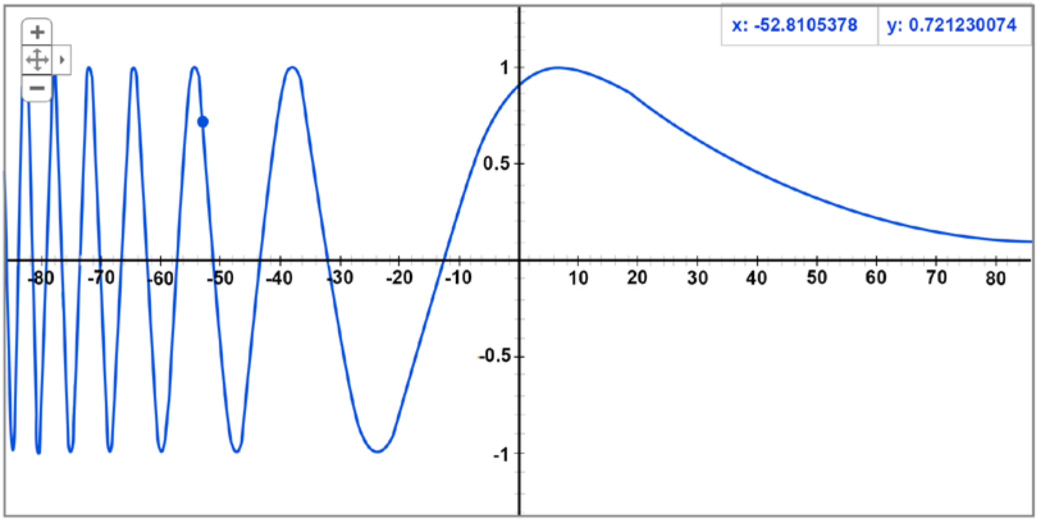


Figure 2.7: The graph

If we go back to the sentence we are parsing in this section, we can see that `black` is in position `pos=2` and `brown` is in position `pos=10`:

```
The black cat sat on the couch and the brown dog slept on the rug.
```

If we apply the sine and cosine functions literally for `pos=2`, we obtain a `size=512` positional encoding vector:

```
PE(2)=
[[ 9.09297407e-01 -4.16146845e-01  9.58144367e-01 -2.86285430e-01
   9.87046242e-01 -1.60435960e-01  9.99164224e-01 -4.08766568e-02
   9.97479975e-01  7.09482506e-02  9.84703004e-01  1.74241230e-01
   9.63226616e-01  2.68690288e-01  9.35118318e-01  3.54335666e-01
   9.02130723e-01  4.31462824e-01  8.65725577e-01  5.00518918e-01
   8.27103794e-01  5.62049210e-01  7.87237823e-01  6.16649508e-01
   7.46903539e-01  6.64932430e-01  7.06710517e-01  7.07502782e-01
…
   5.47683925e-08  1.00000000e+00  5.09659337e-08  1.00000000e+00
   4.74274735e-08  1.00000000e+00  4.41346799e-08  1.00000000e+00
   4.10704999e-08  1.00000000e+00  3.82190599e-08  1.00000000e+00
   3.55655878e-08  1.00000000e+00  3.30963417e-08  1.00000000e+00
   3.07985317e-08  1.00000000e+00  2.86602511e-08  1.00000000e+00
   2.66704294e-08  1.00000000e+00  2.48187551e-08  1.00000000e+00
   2.30956392e-08  1.00000000e+00  2.14921574e-08  1.00000000e+00]]
```

We also obtain a size=512 positional encoding vector for position 10, *pos=10*:

```
PE(10)=
[[-5.44021130e-01 -8.39071512e-01  1.18776485e-01 -9.92920995e-01
   6.92634165e-01 -7.21289039e-01  9.79174793e-01 -2.03019097e-01
   9.37632740e-01  3.47627431e-01  6.40478015e-01  7.67976522e-01
   2.09077001e-01  9.77899194e-01 -2.37917677e-01  9.71285343e-01
  -6.12936735e-01  7.90131986e-01 -8.67519796e-01  4.97402608e-01
  -9.87655997e-01  1.56638563e-01 -9.83699203e-01 -1.79821849e-01
…
  2.73841977e-07  1.00000000e+00  2.54829672e-07  1.00000000e+00
   2.37137371e-07  1.00000000e+00  2.20673414e-07  1.00000000e+00
   2.05352507e-07  1.00000000e+00  1.91095296e-07  1.00000000e+00
   1.77827943e-07  1.00000000e+00  1.65481708e-07  1.00000000e+00
   1.53992659e-07  1.00000000e+00  1.43301250e-07  1.00000000e+00
   1.33352145e-07  1.00000000e+00  1.24093773e-07  1.00000000e+00
   1.15478201e-07  1.00000000e+00  1.07460785e-07  1.00000000e+00]]
```

When we look at the results we obtained with an intuitive literal translation of the *Vaswani* et al. (2017) functions into Python, we would like to check whether the results are meaningful.

The cosine similarity function used for word embedding comes in handy for having a better visualization of the proximity of the positions:

```
cosine_similarity(pos(2), pos(10))= [[0.8600013]]
```

The similarity between the position of the words `black` and `brown` and the lexical field (groups of words that go together) similarity is different:

```
cosine_similarity(black, brown)= [[0.9998901]]
```

The encoding of the position shows a lower similarity value than the word embedding similarity.

The positional encoding has taken these words apart. Bear in mind that word embeddings will vary with the corpus used to train them. The problem is now how to add the positional encoding to the word embedding vectors.

Adding positional encoding to the embedding vector

The authors of the Transformer found a simple way by merely adding the positional encoding vector to the word embedding vector:

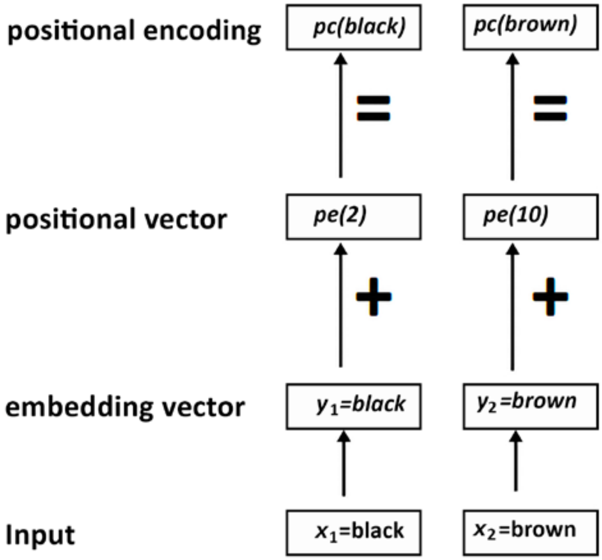


Figure 2.8: Positional encoding

If we go back and take the word embedding of `black`, for example, and name it y_1 = *black*, we are ready to add it to the positional vector *pe*(2) we obtained with positional encoding functions. We will obtain the positional encoding *pc*(*black*) of the input word `black`:

$$pc(black) = y_1 + pe(2)$$

The solution is straightforward. However, if we apply it as shown, we might lose the information of the word embedding, which will be minimized by the positional encoding vector.

There are many possibilities to increase the value of y_1 to make sure that the information of the word embedding layer can be used efficiently in the subsequent layers.

One of the many possibilities is to add an arbitrary value to y_1, the word embedding of `black`:

$$y_1 * math.sqrt(d\_model)$$

We can now add the positional vector to the embedding vector of the word `black`, which are both of the same size (512):

```
for i in range(0, 512,2):
          pe[0][i] = math.sin(pos / (10000 ** ((2 * i)/d_model)))
          pc[0][i] = (y[0][i]*math.sqrt(d_model))+ pe[0][i]

          pe[0][i+1] = math.cos(pos / (10000 ** ((2 * i)/d_model)))
          pc[0][i+1] = (y[0][i+1]*math.sqrt(d_model))+ pe[0][i+1]
```

The result obtained is the final positional encoding vector of dimension d_{model} = 512:

```
pc(black)=
[[ 9.09297407e-01 -4.16146845e-01  9.58144367e-01 -2.86285430e-01
   9.87046242e-01 -1.60435960e-01  9.99164224e-01 -4.08766568e-02
   …
  4.74274735e-08  1.00000000e+00  4.41346799e-08  1.00000000e+00
   4.10704999e-08  1.00000000e+00  3.82190599e-08  1.00000000e+00
   2.66704294e-08  1.00000000e+00  2.48187551e-08  1.00000000e+00
   2.30956392e-08  1.00000000e+00  2.14921574e-08  1.00000000e+00]]
```

The same operation is applied to the word `brown` and all of the other words in a sequence.

We can apply the cosine similarity function to the positional encoding vectors of `black` and `brown`:

```
cosine_similarity(pc(black), pc(brown))= [[0.9627094]]
```

We now have a clear view of the positional encoding process through the three cosine similarity functions we applied to the three states representing the words `black` and `brown`:

```
[[0.99987495]] word similarity
[[0.8600013]] positional encoding vector similarity
[[0.9627094]] final positional encoding similarity
```

We saw that the initial word similarity of their embeddings was high, with a value of `0.99`. Then we saw the positional encoding vector of positions 2 and 10 drew these two words apart with a lower similarity value of `0.86`.

Finally, we added the word embedding vector of each word to its respective positional encoding vector. We saw that this brought the cosine similarity of the two words to `0.96`.

The positional encoding of each word now contains the initial word embedding information and the positional encoding values.

The output of positional encoding leads to the multi-head attention sublayer.

Sublayer 1: Multi-head attention

The multi-head attention sublayer contains eight heads and is followed by post-layer normalization, which will add residual connections to the output of the sublayer and normalize it:

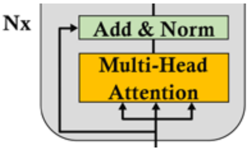


Figure 2.9: Multi-head attention sublayer

This section begins with the architecture of an attention layer. Then, an example of multi-attention is implemented in a small module in Python. Finally, post-layer normalization is described.

Let's start with the architecture of multi-head attention.

The architecture of multi-head attention

The input of the multi-attention sublayer of the first layer of the encoder stack is a vector that contains the embedding and the positional encoding of each word. The next layers of the stack do not start these operations over.

The dimension of the vector of each word x_n of an input sequence is d_{model} = 512:

$$pe(x_n)=[d_1=9.09297407e^{-}01,\ d_2=-4.16146845e^{-}01,\ ..,\ d_{512}=1.00000000e+00]$$

The representation of each word x_n has become a vector of d_{model} = 512 dimensions.

Each word is mapped to all the other words to determine how it fits in a sequence.

In the following sentence, we can see that it could be related to `cat` and `rug` in the sequence:

```
Sequence =The cat sat on the rug and it was dry-cleaned.
```

The model will train to find out if `it` is related to `cat` or `rug`. We could run a huge calculation by training the model using the d_{model} = 512 dimensions as they are now.

However, we would only get one point of view at a time by analyzing the sequence with one d_{model} block. Furthermore, it would take quite some calculation time to find other perspectives.

A better way is to divide the d_{model}= 512 dimensions of each word x_n of x (all the words of a sequence) into 8 d_k=64 dimensions.

We then can run the 8 “heads” in parallel to speed up the training and obtain 8 different representation subspaces of how each word relates to another:

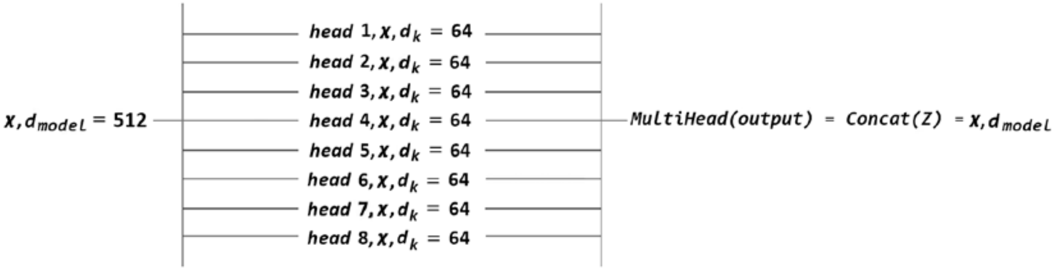


Figure 2.10: Multi-head representations

You can see that there are now 8 heads running in parallel. One head might decide that `it` fits well with `cat` and another that `it` fits well with `rug` and another that `rug` fits well with `dry-cleaned`.

The output of each head is a matrix Z_i with a shape of $x * d_k$. The output of a multi-attention head is Z defined as:

$$Z = (Z_0, Z_1, Z_2, Z_3, Z_4, Z_5, Z_6, Z_7)$$

However, Z must be concatenated so that the output of the multi-head sublayer is not a sequence of dimensions but one line of an $xm * d_{model}$ matrix.

Before exiting the multi-head attention sublayer, the elements of Z are concatenated:

$$MultiHead(output) = Concat(Z_0, Z_1, Z_2, Z_3, Z_4, Z_5, Z_6, Z_7) = x, d_{model}$$

Notice that each head is concatenated into z that has a dimension of d_{model}= 512. The output of the multi-headed layer respects the constraint of the original Transformer model.

Inside each head h_n of the attention mechanism, the “word” matrices have three representations:

- A query matrix (Q) that has a dimension of d_q=64, which seeks all the key-value pairs of the other “word” matrices.
- A key matrix (K) that has a dimension of d_k=64, which will be trained to provide an attention value.

- A value matrix (V) that has a dimension of d_v=64, which will be trained to provide another attention value.

Attention is defined as "Scaled Dot-Product Attention," which is represented in the following equation in which we plug Q, K, and V:

$$Attention(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}) = softmax\left(\frac{\boldsymbol{QK^T}}{\sqrt{\boldsymbol{d_k}}}\right)V$$

The matrices all have the same dimension, making it relatively simple to use a scaled dot product to obtain the attention values for each head and then concatenate the output Z of the 8 heads.

To obtain Q, K, and V, we must train the model with their weight matrices Q_w, K_w, and V_w, which have d_k=64 columns and d_{model}= 512 rows. For example, Q is obtained by a dot-product between x and Q_w. Q will have a dimension of d_k=64.

You can modify all the parameters, such as the number of layers, heads, d_{model}, d_k, and other variables of the Transformer to fit your model. This chapter describes the original Transformer parameters by *Vaswani* et al. (2017). It is essential to understand the original architecture before modifying it or exploring variants of the original model designed by others.

Google Brain Trax, OpenAI, and Hugging Face, among others, provide ready-to-use libraries that we will be using throughout this book.

However, let's open the hood of the Transformer model and get our hands dirty in Python to illustrate the architecture we just explored to visualize the model in code and show it with intermediate images.

We will use basic Python code with only `numpy` and a `softmax` function in 10 steps to run the key aspects of the attention mechanism.

Bear in mind that an Industry 4.0 developer will face the challenge of multiple architectures for the same algorithm.

Let's now start building *Step 1* of our model to represent the input.

Step 1: Represent the input

Save `Multi_Head_Attention_Sub_Layer.ipynb` to your Google Drive (make sure you have a Gmail account) and then open it in Google Colaboratory. The notebook is in the GitHub repository for this chapter.

We will start by only using minimal Python functions to understand the Transformer at a low level with the inner workings of an attention head. We will explore the inner workings of the multi-head attention sublayer using basic code:

```python
import numpy as np
from scipy.special import softmax
```

The input of the attention mechanism we are building is scaled down to d_{model}==4 instead of d_{model} = 512. This brings the dimensions of the vector of an input x down to d_{model}=4, which is easier to visualize.

x contains 3 inputs with 4 dimensions each instead of 512:

```python
print("Step 1: Input : 3 inputs, d_model=4")
x =np.array([[1.0, 0.0, 1.0, 0.0],   # Input 1
             [0.0, 2.0, 0.0, 2.0],   # Input 2
             [1.0, 1.0, 1.0, 1.0]])  # Input 3
print(x)
```

The output shows that we have 3 vectors of d_{model}=4:

```
Step 1: Input : 3 inputs, d_model=4
[[1. 0. 1. 0.]
 [0. 2. 0. 2.]
 [1. 1. 1. 1.]]
```

The first step of our model is ready:

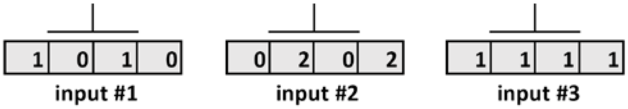


Figure 2.11: Input of a multi-head attention sublayer

We will now add the weight matrices to our model.

Step 2: Initializing the weight matrices

Each input has 3 weight matrices:

- Q_w to train the queries
- K_w to train the keys
- V_w to train the values

These 3 weight matrices will be applied to all the inputs in this model.

The weight matrices described by *Vaswani* et al. (2017) are d_K==64 dimensions. However, let's scale the matrices down to d_K==3. The dimensions are scaled down to 3*4 weight matrices to be able to visualize the intermediate results more easily and perform dot products with the input x.

> The size and shape of the matrices in this educational notebook are arbitrary. The goal is to go through the overall process of an attention mechanism.

The three weight matrices are initialized starting with the query weight matrix:

```
print("Step 2: weights 3 dimensions x d_model=4")
print("w_query")
w_query =np.array([[1, 0, 1],
                   [1, 0, 0],
                   [0, 0, 1],
                   [0, 1, 1]])
print(w_query)
```

The output is the `w_query` weight matrix:

```
w_query
[[1 0 1]
 [1 0 0]
 [0 0 1]
 [0 1 1]]
```

We will now initialize the key weight matrix:

```
print("w_key")
w_key =np.array([[0, 0, 1],
                 [1, 1, 0],
                 [0, 1, 0],
                  [1, 1, 0]])
print(w_key)
```

The output is the key weight matrix:

```
w_key
[[0 0 1]
 [1 1 0]
 [0 1 0]
 [1 1 0]]
```

Finally, we initialize the value weight matrix:

```
print("w_value")
w_value = np.array([[0, 2, 0],
                    [0, 3, 0],
                    [1, 0, 3],
                    [1, 1, 0]])
print(w_value)
```

The output is the value weight matrix:

```
w_value
[[0 2 0]
 [0 3 0]
 [1 0 3]
 [1 1 0]]
```

The second step of our model is ready:

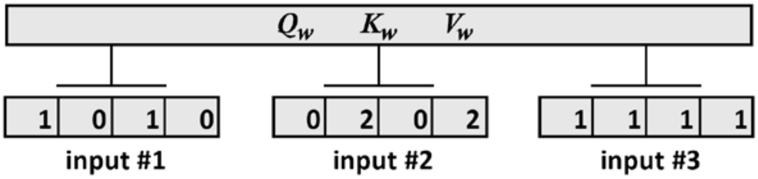


Figure 2.12: Weight matrices added to the model

We will now multiply the weights by the input vectors to obtain *Q*, *K*, and *V*.

Step 3: Matrix multiplication to obtain Q, K, and V

We will now multiply the input vectors by the weight matrices to obtain a query, key, and value vector for each input.

In this model, we will assume that there is one `w_query`, `w_key`, and `w_value` weight matrix for all inputs. Other approaches are possible.

Let's first multiply the input vectors by the `w_query` weight matrix:

```
print("Step 3: Matrix multiplication to obtain Q,K,V")
print("Query: x * w_query")
Q=np.matmul(x,w_query)
print(Q)
```

The output is a vector for Q_1==64= [1, 0, 2], Q_2= [2,2, 2], and Q_3= [2,1, 3]:

```
Step 3: Matrix multiplication to obtain Q,K,V
Query: x * w_query
[[1. 0. 2.]
 [2. 2. 2.]
 [2. 1. 3.]]
```

We now multiply the input vectors by the `w_key` weight matrix:

```
print("Key: x * w_key")
K=np.matmul(x,w_key)
print(K)
```

We obtain a vector for K_1= [0, 1, 1], K_2= [4, 4, 0], and K_3= [2 ,3, 1]:

```
Key: x * w_key
[[0. 1. 1.]
 [4. 4. 0.]
 [2. 3. 1.]]
```

Finally, we multiply the input vectors by the `w_value` weight matrix:

```
print("Value: x * w_value")
V=np.matmul(x,w_value)
print(V)
```

We obtain a vector for V_1= [1, 2, 3], V_2= [2, 8, 0], and V_3= [2 ,6, 3]:

```
Value: x * w_value
[[1. 2. 3.]
 [2. 8. 0.]
 [2. 6. 3.]]
```

The third step of our model is ready:

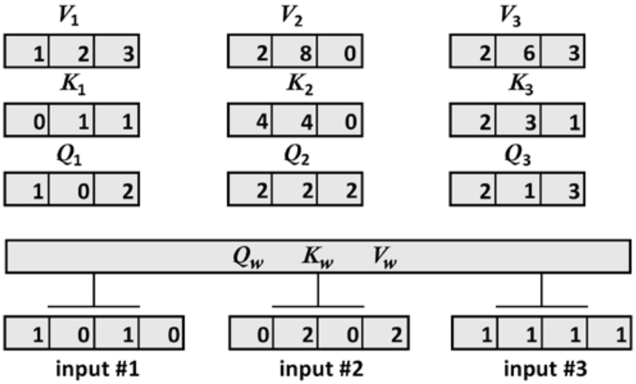


Figure 2.13: Q, K, and V are generated

We have the *Q*, *K*, and *V* values we need to calculate the attention scores.

Step 4: Scaled attention scores

The attention head now implements the original Transformer equation:

$$Attention(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}) = softmax\left(\frac{\boldsymbol{QK^T}}{\sqrt{\boldsymbol{d_k}}}\right)V$$

Step 4 focuses on *Q* and *K*:

$$\left(\frac{\boldsymbol{QK^T}}{\sqrt{\boldsymbol{d_k}}}\right)$$

For this model, we will round $\sqrt{d_k} = \sqrt{3} = 1.75$ to 1 and plug the values into the *Q* and *K* part of the equation:

```
print("Step 4: Scaled Attention Scores")
k_d=1   #square root of k_d=3 rounded down to 1 for this example
attention_scores = (Q @ K.transpose())/k_d
print(attention_scores)
```

The intermediate result is displayed:

```
Step 4: Scaled Attention Scores
[[ 2.  4.  4.]
 [ 4. 16. 12.]
 [ 4. 12. 10.]]
```

Step 4 is now complete. For example, the score for x_1 is [2,4,4] across the *K* vectors across the head as displayed:

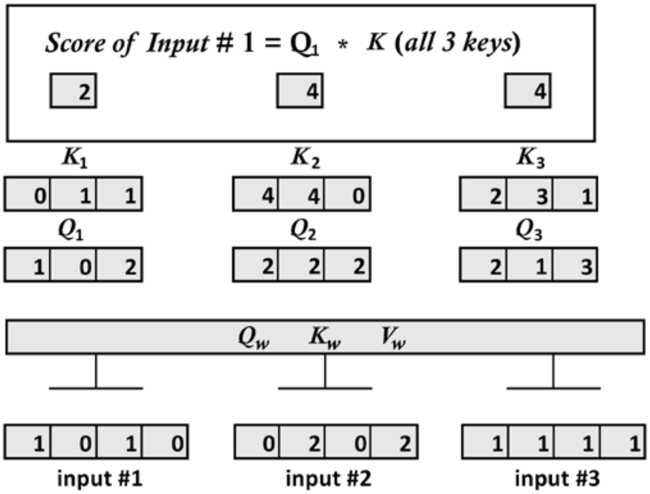


Figure 2.14: Scaled attention scores for input #1

The attention equation will now apply softmax to the intermediate scores for each vector.

Step 5: Scaled softmax attention scores for each vector

We now apply a softmax function to each intermediate attention score. Instead of doing a matrix multiplication, let's zoom down to each individual vector:

```
print("Step 5: Scaled softmax attention_scores for each vector")
attention_scores[0]=softmax(attention_scores[0])
attention_scores[1]=softmax(attention_scores[1])
attention_scores[2]=softmax(attention_scores[2])
print(attention_scores[0])
print(attention_scores[1])
print(attention_scores[2])
```

We obtain scaled softmax attention scores for each vector:

```
Step 5: Scaled softmax attention_scores for each vector
[0.06337894 0.46831053 0.46831053]
[6.03366485e-06 9.82007865e-01 1.79861014e-02]
[2.95387223e-04 8.80536902e-01 1.19167711e-01]
```

Step 5 is now complete. For example, the softmax of the score of x_1 for all the keys is:

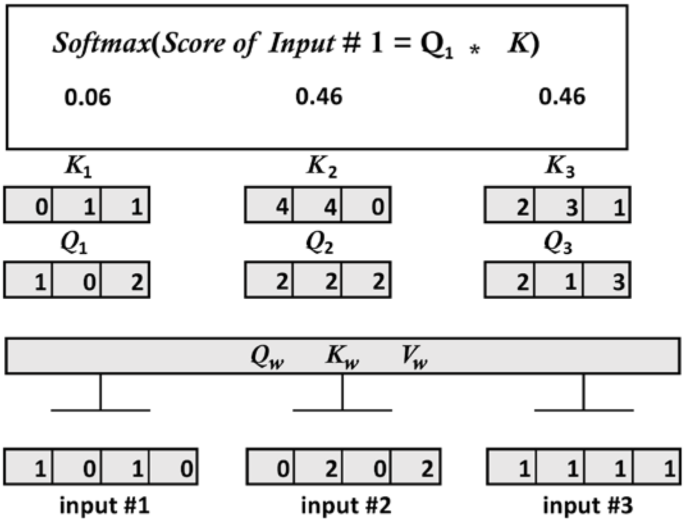


Figure 2.15: The softmax score of input #1 for all of the keys

We can now calculate the final attention values with the complete equation.

Step 6: The final attention representations

We now can finalize the attention equation by plugging *V* in:

$$Attention(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = softmax\left(\frac{\boldsymbol{QK^T}}{\sqrt{\boldsymbol{d_k}}}\right)V$$

We will first calculate the attention score of input x_1 for *Steps 6* and *7*. We calculate one attention value for one word vector. When we reach *Step 8*, we will generalize the attention calculation to the other two input vectors.

To obtain *Attention*(*Q*,*K*,*V*) for x_1 we multiply the intermediate attention score by the 3 value vectors one by one to zoom down into the inner workings of the equation:

```
print("Step 6: attention value obtained by score1/k_d * V")
print(V[0])
print(V[1])
print(V[2])
print("Attention 1")
attention1=attention_scores[0].reshape(-1,1)
```

```
attention1=attention_scores[0][0]*V[0]
print(attention1)

print("Attention 2")
attention2=attention_scores[0][1]*V[1]
print(attention2)

print("Attention 3")
attention3=attention_scores[0][2]*V[2]
print(attention3)

Step 6: attention value obtained by score1/k_d * V
[1. 2. 3.]
[2. 8. 0.]
[2. 6. 3.]

Attention 1
[0.06337894 0.12675788 0.19013681]
Attention 2
[0.93662106 3.74648425 0.        ]
Attention 3
[0.93662106 2.80986319 1.40493159]
```

Step 6 is complete. For example, the 3 attention values for x_1 for each input have been calculated:

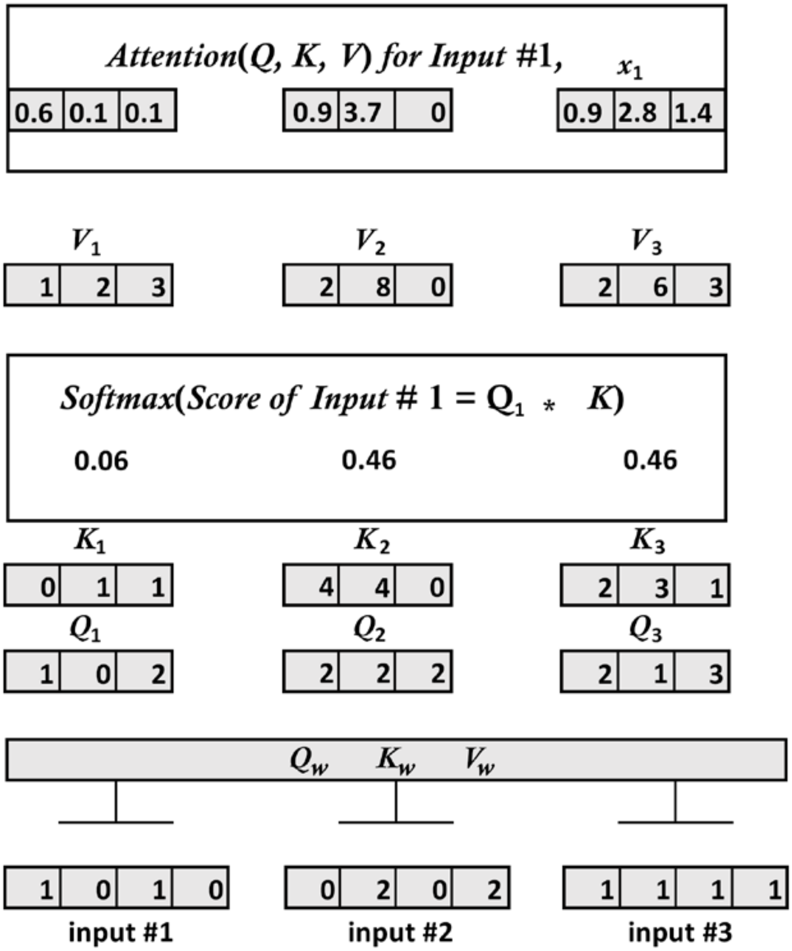


Figure 2.16: Attention representations

The attention values now need to be summed up.

Step 7: Summing up the results

The 3 attention values of input #1 obtained will now be summed to obtain the first line of the output matrix:

```
print("Step 7: summed the results to create the first line of the output
matrix")
attention_input1=attention1+attention2+attention3
print(attention_input1)
```

The output is the first line of the output matrix for input #1:

```
Step 7: summed the results to create the first line of the output matrix
[1.93662106 6.68310531 1.59506841]]
```

The second line will be for the output of the next input, input #2, for example.

We can see the summed attention value for x_1 in *Figure 2.17*:

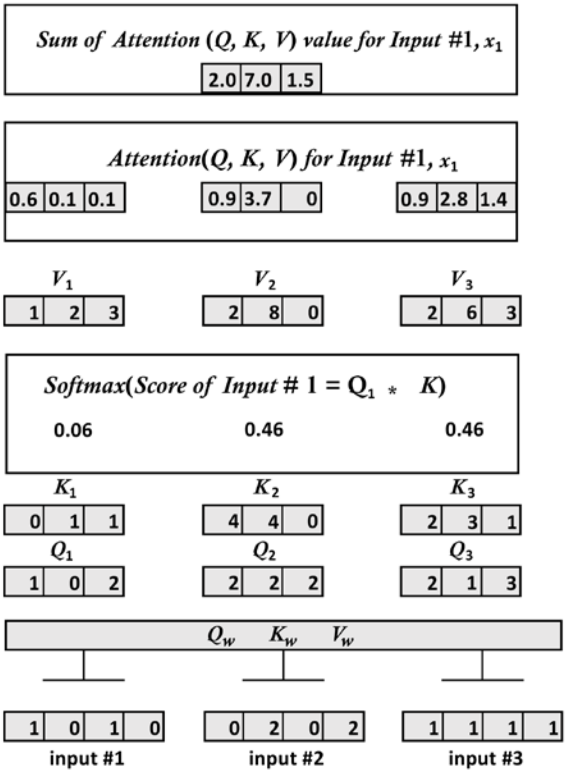


Figure 2.17: Summed results for one input

We have completed the steps for input #1. We now need to add the results of all the inputs to the model.

Step 8: Steps 1 to 7 for all the inputs

The Transformer can now produce the attention values of input #2 and input #3 using the same method described from *Step 1* to *Step 7* for one attention head.

From this step onwards, we will assume we have 3 attention values with learned weights with $d_{model} = 64$. We now want to see what the original dimensions look like when they reach the sublayer's output.

We have seen the attention representation process in detail with a small model. Let's go directly to the result and assume we have generated the 3 attention representations with a dimension of $d_{model} = 64$:

```
print("Step 8: Step 1 to 7 for inputs 1 to 3")
#We assume we have 3 results with learned weights (they were not trained 
in this example)
#We assume we are implementing the original Transformer paper.We will have 
3 results of 64 dimensions each
attention_head1=np.random.random((3, 64))
print(attention_head1)
```

The following output displays the simulation of z_0, which represents the 3 output vectors of d_{model} = 64 dimensions for head 1:

```
Step 8: Step 1 to 7 for inputs 1 to 3
[[0.31982626 0.99175996…(61 squeezed values)…0.16233212]
 [0.99584327 0.55528662…(61 squeezed values)…0.70160307]
 [0.14811583 0.50875291…(61 squeezed values)…0.83141355]]
```

The results will vary when you run the notebook because of the stochastic nature of the generation of the vectors.

The Transformer now has the output vectors for the inputs of one head. The next step is to generate the output of the 8 heads to create the final output of the attention sublayer.

Step 9: The output of the heads of the attention sublayer

We assume that we have trained the 8 heads of the attention sublayer. The Transformer now has 3 output vectors (of the 3 input vectors that are words or word pieces) of $d_{model} = 64$ dimensions each:

```
print("Step 9: We assume we have trained the 8 heads of the attention
sublayer")
z0h1=np.random.random((3, 64))
z1h2=np.random.random((3, 64))
z2h3=np.random.random((3, 64))
z3h4=np.random.random((3, 64))
z4h5=np.random.random((3, 64))
z5h6=np.random.random((3, 64))
z6h7=np.random.random((3, 64))
z7h8=np.random.random((3, 64))
print("shape of one head",z0h1.shape,"dimension of 8 heads",64*8)
```

The output shows the shape of one of the heads:

```
Step 9: We assume we have trained the 8 heads of the attention sublayer
shape of one head (3, 64) dimension of 8 heads 512
```

The 8 heads have now produced Z:

$$Z = (Z_0, Z_1, Z_2, Z_3, Z_4, Z_5, Z_6, Z_7)$$

The Transformer will now concatenate the 8 elements of Z for the final output of the multi-head attention sublayer.

Step 10: Concatenation of the output of the heads

The Transformer concatenates the 8 elements of Z:

$$MultiHead(Output) = Concat\ (Z_0, Z_1, Z_2, Z_3, Z_4, Z_5, Z_6, Z_7)\ W^0 = x, d_{model}$$

Note that Z is multiplied by W^0, which is a weight matrix that is trained as well. In this model, we will assume W^0 is trained and integrated into the concatenation function.

Z_0 to Z_7 are concantenated:

```
print("Step 10: Concantenation of heads 1 to 8 to obtain the original
8x64=512 ouput dimension of the model")
output_attention=np.hstack((z0h1,z1h2,z2h3,z3h4,z4h5,z5h6,z6h7,z7h8))
print(output_attention)
```

The output is the concatenation of *Z*:

```
Step 10: Concatenation of heads 1 to 8 to obtain the original 8x64=512
output dimension of the model
[[0.65218495 0.11961095 0.9555153  ... 0.48399266 0.80186221 0.16486792]
 [0.95510952 0.29918492 0.7010377  ... 0.20682832 0.4123836  0.90879359]
 [0.20211378 0.86541746 0.01557758 ... 0.69449636 0.02458972 0.889699  ]]
```

The concatenation can be visualized as stacking the elements of *Z* side by side:

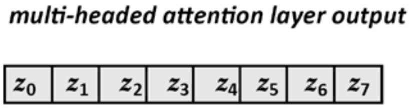


Figure 2.18: Attention sublayer output

The concatenation produced a standard d_{model} = 512 dimensional output:

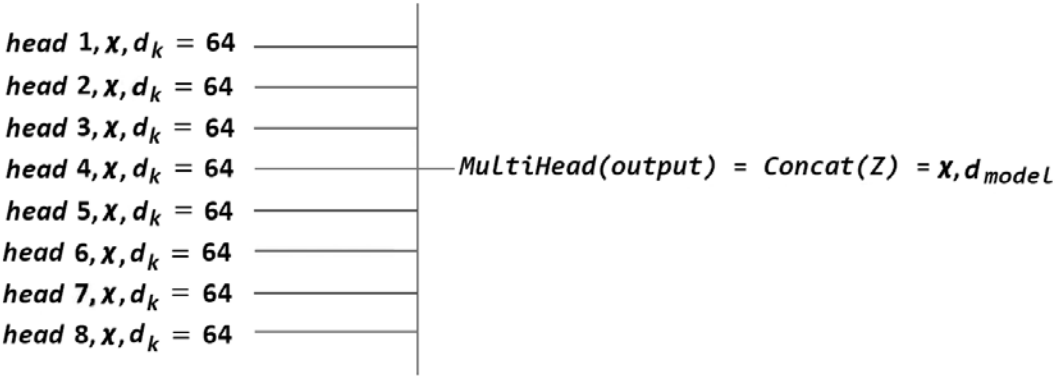


Figure 2.19: Concatenation of the output of the 8 heads

Layer normalization will now process the attention sublayer.

Post-layer normalization

Each attention sublayer and each Feedforward sublayer of the Transformer is followed by **post-layer normalization (Post-LN)**:

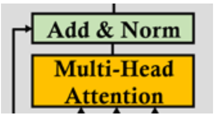


Figure 2.20: Post-layer normalization

The Post-LN contains an add function and a layer normalization process. The add function processes the residual connections that come from the input of the sublayer. The goal of the residual connections is to make sure critical information is not lost. The Post-LN or layer normalization can thus be described as follows:

$$LayerNormalization\ (x + Sublayer(x))$$

$Sublayer(x)$ is the sublayer itself. x is the information available at the input step of $Sublayer(x)$.

The input of the *LayerNormalization* is a vector v resulting from $x + Sublayer(x)$. $d_{model} = 512$ for every input and output of the Transformer, which standardizes all the processes.

Many layer normalization methods exist, and variations exist from one model to another. The basic concept for $v = x + Sublayer(x)$ can be defined by $LayerNormalization\ (v)$:

$$LayerNormalization(\boldsymbol{v}) = \gamma \frac{\boldsymbol{v} - \boldsymbol{\mu}}{\boldsymbol{\sigma}} + \beta$$

The variables are:

- μ is the mean of v of dimension d. As such:

$$\mu = \frac{\boldsymbol{1}}{\boldsymbol{d}} \sum_{\boldsymbol{k=1}}^{\boldsymbol{d}} \boldsymbol{v_k}$$

- σ is the standard deviation v of dimension d. As such:

$$\boldsymbol{\sigma^2} = \frac{\boldsymbol{1}}{\boldsymbol{d}} \sum_{\boldsymbol{k=1}}^{\boldsymbol{d}} (\boldsymbol{v_{k-\mu}})$$

- γ is a scaling parameter.
- β is a bias vector.

This version of $LayerNormalization\ (v)$ shows the general idea of the many possible post-LN methods. The next sublayer can now process the output of the post-LN or $LayerNormalization\ (v)$. In this case, the sublayer is a feedforward network.

Sublayer 2: Feedforward network

The input of the **feedforward network** (**FFN**) is the d_{model} = 512 output of the post-LN of the previous sublayer:

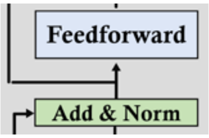


Figure 2.21: Feedforward sublayer

The FFN sublayer can be described as follows:

- The FFNs in the encoder and decoder are fully connected.
- The FFN is a position-wise network. Each position is processed separately and in an identical way.
- The FFN contains two layers and applies a ReLU activation function.
- The input and output of the FFN layers is d_{model} = 512, but the inner layer is larger with d_{ff} =2048.
- The FFN can be viewed as performing two convolutions with size 1 kernels.

Taking this description into account, we can describe the optimized and standardized FFN as follows:

$$FFN(x) = \max\left(0, xW_1 + b_1\right) W_2+b_2$$

The output of the FFN goes to post-LN, as described in the previous section. Then the output is sent to the next layer of the encoder stack and the multi-head attention layer of the decoder stack.

Let’s now explore the decoder stack.

The decoder stack

The layers of the decoder of the Transformer model are *stacks of layers* like the encoder layers. Each layer of the decoder stack has the following structure:

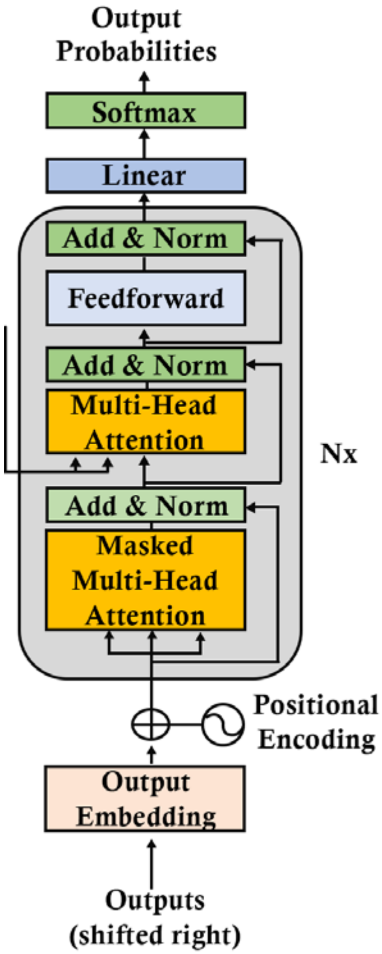


Figure 2.22: A layer of the decoder stack of the Transformer

The structure of the decoder layer remains the same as the encoder for all the *N*=6 layers of the Transformer model. Each layer contains three sublayers: a multi-headed masked attention mechanism, a multi-headed attention mechanism, and a fully connected position-wise feedforward network.

The decoder has a third main sublayer, which is the masked multi-head attention mechanism. In this sublayer output, at a given position, the following words are masked so that the Transformer bases its assumptions on its inferences without seeing the rest of the sequence. That way, in this model, it cannot see future parts of the sequence.

A residual connection, $Sublayer(x)$, surrounds each of the three main sublayers in the Transformer model like in the encoder stack:

$$LayerNormalization\ (x + Sublayer(x))$$

The embedding layer sublayer is only present at the bottom level of the stack, like for the encoder stack. The output of every sublayer of the decoder stack has a constant dimension, d_{model} like in the encoder stack, including the embedding layer and the output of the residual connections.

We can see that the designers worked hard to create symmetrical encoder and decoder stacks.

The structure of each sublayer and function of the decoder is similar to the encoder. In this section, we can refer to the encoder for the same functionality when we need to. We will only focus on the differences between the decoder and the encoder.

Output embedding and position encoding

The structure of the sublayers of the decoder is mostly the same as the sublayers of the encoder. The output embedding layer and position encoding function are the same as in the encoder stack.

In the Transformer usage we are exploring through the model presented by *Vaswani* et al. (2017), the output is a translation we need to learn. I chose to use a French translation:

```
Output=Le chat noir était assis sur le canapé et le chien marron dormait
sur le tapis
```

This output is the French translation of the English input sentence:

```
Input=The black cat sat on the couch and the brown dog slept on the rug.
```

The output words go through the word embedding layer and then the positional encoding function like in the first layer of the encoder stack.

Let's see the specific properties of the multi-head attention layers of the decoder stack.

The attention layers

The Transformer is an auto-regressive model. It uses the previous output sequences as an additional input. The multi-head attention layers of the decoder use the same process as the encoder.

However, the masked multi-head attention sublayer 1 only lets attention apply to the positions up to and including the current position. The future words are hidden from the Transformer, and this forces it to learn how to predict.

A post-layer normalization process follows the masked multi-head attention sublayer 1 as in the encoder.

The multi-head attention sublayer 2 also only attends to the positions up to the current position the Transformer is predicting to avoid seeing the sequence it must predict.

The multi-head attention sublayer 2 draws information from the encoder by taking encoder (K, V) into account during the dot-product attention operations. This sublayer also draws information from the masked multi-head attention sublayer 1 (masked attention) by also taking sublayer 1(Q) into account during the dot-product attention operations. The decoder thus uses the trained information of the encoder. We can define the input of the self-attention multi-head sublayer of a decoder as:

$$Input\_Attention = (Output\_decoder\_sub\_layer - 1(Q), Output\_encoder\_layer(K, V))$$

A post-layer normalization process follows the masked multi-head attention sublayer 1 as in the encoder.

The Transformer then goes to the FFN sublayer, followed by a post-LN and the linear layer.

The FFN sublayer, the post-LN, and the linear layer

The FFN sublayer has the same structure as the FFN of the encoder stack. The post-layer normalization of the FFN works as the layer normalization of the encoder stack.

The Transformer produces an output sequence of only one element at a time:

$$Output\ sequence = (y_1, y_2, \dots y_n)$$

The linear layer produces an output sequence with a linear function that varies per model but relies on the standard method:

$$y = w * x + b$$

w and b are learned parameters.

The linear layer will thus produce the next probable elements of a sequence that a softmax function will convert into a probable element.

The decoder layer, like the encoder layer, will then go from layer *l* to layer *l+1*, up to the top layer of the *N*=6-layer transformer stack.

Let's now see how the Transformer was trained and the performance it obtained.

Training and performance

The original Transformer was trained on a 4.5 million sentence pair English-German dataset and a 36 million sentence pair English-French dataset.

The datasets come from **Workshops on Machine Translation** (**WMT**), which can be found at the following link if you wish to explore the WMT datasets: `http://www.statmt.org/wmt14/`

The training of the original Transformer base models took 12 hours to train for 100,000 steps on a machine with 8 NVIDIA P100 GPUs. The big models took 3.5 days for 300,000 steps.

The original Transformer outperformed all the previous machine translation models with a BLEU score of 41.8. The result was obtained on the WMT English-to-French dataset.

BLEU stands for Bilingual Evaluation Understudy. It is an algorithm that evaluates the quality of the results of machine translations.

The Google Research and Google Brain team applied optimization strategies to improve the performance of the Transformer. For example, the Adam optimizer was used, but the learning rate varied by first going through warmup states with a linear rate and decreasing the rate afterward.

Different types of regularization techniques, such as residual dropout and dropouts, were applied to the sums of embeddings. Also, the Transformer applies label smoothing that avoids overfitting with overconfident one-hot outputs. It introduces less accurate evaluations and forces the model to train more and better.

Several other Transformer model variations have led to other models and usages that we will explore in the subsequent chapters.

Before end the chapter, let's get a feel of the simplicity of ready-to-use transformer models in Hugging Face, for example.

Tranformer models in Hugging Face

Everything you saw in this chapter can be condensed in to a ready-to-use Hugging Face transformer model.

With Hugging Face, you can implement machine translation in three lines of code!

Open `Multi_Head_Attention_Sub_Layer.ipynb` in Google Colaboratory. Save the notebook in your Google Drive (make sure you have a Gmail account). Go to the two last cells.

We first ensure that Hugging Face transformers are installed:

```
!pip -q install transformers
```

The first cell imports the Hugging Face pipeline that contains several transformer usages:

```
#@title Retrieve pipeline of modules and choose English to French
translation
from transformers import pipeline
```

We then implement the Hugging Face pipeline, which contains ready-to-use functions. In our case, to illustrate the Transformer model of this chapter, we activate the translator model and enter a sentence to translate from English to French:

```
translator = pipeline("translation_en_to_fr")
#One line of code!
print(translator("It is easy to translate languages with transformers",
max_length=40))
```

And *voilà*! The translation is displayed:

```
[{'translation_text': 'Il est facile de traduire des langues à l'aide de
transformateurs.'}]
```

Hugging Face shows how transformer architectures can be used in ready-to-use models.

Summary

In this chapter, we first got started by examining the mind-blowing long-distance dependencies transformer architectures can uncover. Transformers can perform transductions from written and oral sequences to meaningful representations as never before in the history of **Natural Language Understanding** (**NLU**).

These two dimensions, the expansion of transduction and the simplification of implementation, are taking artificial intelligence to a level never seen before.

We explored the bold approach of removing RNNs, LSTMs, and CNNs from transduction problems and sequence modeling to build the Transformer architecture. The symmetrical design of the standardized dimensions of the encoder and decoder makes the flow from one sublayer to another nearly seamless.

We saw that beyond removing recurrent network models, transformers introduce parallelized layers that reduce training time. We discovered other innovations, such as positional encoding and masked multi-headed attention.

The flexible, original Transformer architecture provides the basis for many other innovative variations that open the way for yet more powerful transduction problems and language modeling.

We will zoom more in-depth into some aspects of the Transformer's architecture in the following chapters when describing the many variants of the original model.

The arrival of the Transformer marks the beginning of a new generation of ready-to-use artificial intelligence models. For example, Hugging Face and Google Brain make artificial intelligence easy to implement with a few lines of code.

In the next chapter, *Fine-Tuning BERT Models*, we will explore the powerful evolutions of the original Transformer model.

Questions

1. NLP transduction can encode and decode text representations. (True/False)
2. **Natural Language Understanding** (**NLU**) is a subset of **Natural Language Processing** (**NLP**). (True/False)
3. Language modeling algorithms generate probable sequences of words based on input sequences. (True/False)
4. A transformer is a customized LSTM with a CNN layer. (True/False)
5. A transformer does not contain LSTM or CNN layers. (True/False)
6. Attention examines all the tokens in a sequence, not just the last one. (True/False)
7. A transformer uses a positional vector, not positional encoding. (True/False)
8. A transformer contains a feedforward network. (True/False)
9. The masked multi-headed attention component of the decoder of a transformer prevents the algorithm parsing a given position from seeing the rest of a sequence that is being processed. (True/False)
10. Transformers can analyze long-distance dependencies better than LSTMs. (True/False)

References

- *Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin, 2017, Attention Is All You Need*, `https://arxiv.org/abs/1706.03762`
- Hugging Face Transformer Usage: `https://huggingface.co/transformers/usage.html`
- Tensor2Tensor (T2T) Introduction: `https://colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb?hl=en`
- Manuel Romero Notebook with link to explanations by *Raimi Karim*: `https://colab.research.google.com/drive/1rPk3ohrmVclqhH7uQ7qys4oznDdAhpzF`
- Google language research: `https://research.google/teams/language/`
- Hugging Face research: `https://huggingface.co/transformers/index.html`
- *The Annotated Transformer*: `http://nlp.seas.harvard.edu/2018/04/03/attention.html`
- *Jay Alammar, The Illustrated Transformer*: `http://jalammar.github.io/illustrated-transformer/`

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the authors:

`https://www.packt.link/Transformers`

3

Fine-Tuning BERT Models

In *Chapter 2*, *Getting Started with the Architecture of the Transformer Model*, we defined the building blocks of the architecture of the original Transformer. Think of the original Transformer as a model built with LEGO® bricks. The construction set contains bricks such as encoders, decoders, embedding layers, positional encoding methods, multi-head attention layers, masked multi-head attention layers, post-layer normalization, feed-forward sub-layers, and linear output layers.

The bricks come in various sizes and forms. You can spend hours building all sorts of models using the same building kit! Some constructions will only require some of the bricks. Other constructions will add a new piece, just like when we obtain additional bricks for a model built using LEGO® components.

BERT added a new piece to the Transformer building kit: a bidirectional multi-head attention sub-layer. When we humans have problems understanding a sentence, we do not just look at the past words. BERT, like us, looks at all the words in the same sentence at the same time.

This chapter will first explore the architecture of **Bidirectional Encoder Representations from Transformers** (**BERT**). BERT only uses the blocks of the encoders of the Transformer in a novel way and does not use the decoder stack.

Then we will fine-tune a pretrained BERT model. The BERT model we will fine-tune was trained by a third party and uploaded to Hugging Face. Transformers can be pretrained. Then, a pretrained BERT, for example, can be fine-tuned on several NLP tasks. We will go through this fascinating experience of downstream Transformer usage using Hugging Face modules.

This chapter covers the following topics:

- Bidirectional Encoder Representations from Transformers (BERT)
- The architecture of BERT
- The two-step BERT framework
- Preparing the pretraining environment
- Defining pretraining encoder layers
- Defining fine-tuning
- Downstream multitasking
- Building a fine-tuned BERT model
- Loading an acceptability judgment dataset
- Creating attention masks
- BERT model configuration
- Measuring the performance of the fine-tuned model

Our first step will be to explore the background of the BERT model.

The architecture of BERT

BERT introduces bidirectional attention to transformer models. Bidirectional attention requires many other changes to the original Transformer model.

We will not go through the building blocks of transformers described in *Chapter 2*, *Getting Started with the Architecture of the Transformer Model*. You can consult *Chapter 2* at any time to review an aspect of the building blocks of transformers. In this section, we will focus on the specific aspects of BERT models.

We will focus on the evolutions designed by *Devlin* et al. (2018), which describe the encoder stack. We will first go through the encoder stack, then the preparation of the pretraining input environment. Then we will describe the two-step framework of BERT: pretraining and fine-tuning.

Let's first explore the encoder stack.

The encoder stack

The first building block we will take from the original Transformer model is an encoder layer. The encoder layer, as described in *Chapter 2*, *Getting Started with the Architecture of the Transformer Model*, is shown in *Figure 3.1*:

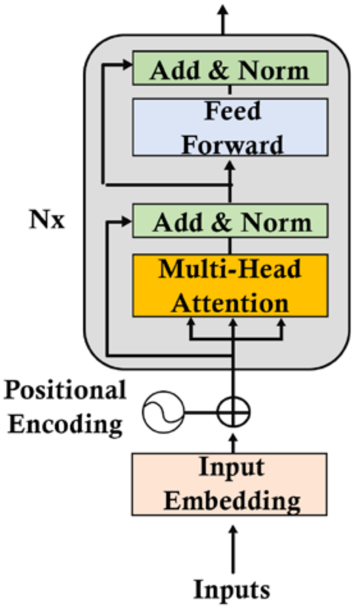


Figure 3.1: The encoder layer

The BERT model does not use decoder layers. A BERT model has an encoder stack but no decoder stacks. The masked tokens (hiding the tokens to predict) are in the attention layers of the encoder, as we will see when we zoom into a BERT encoder layer in the following sections.

The original Transformer contains a stack of *N*=6 layers. The number of dimensions of the original Transformer is $d_{\text{model}} = 512$. The number of attention heads of the original Transformer is *A*=8. The dimensions of the head of the original Transformer are:

$$d_k = \frac{d_{model}}{A} = \frac{512}{8} = 64$$

BERT encoder layers are larger than the original Transformer model.

Two BERT models can be built with the encoder layers:

- BERT$_{\text{BASE}}$, which contains a stack of *N*=12 encoder layers. $d_{\text{model}} = 768$ and can also be expressed as *H*=768, as in the BERT paper. A multi-head attention sub-layer contains *A*=12 heads. The dimension of each head z_A remains 64 as in the original Transformer model:

$$d_k = \frac{d_{model}}{A} = \frac{768}{12} = 64$$

- The output of each multi-head attention sub-layer before concatenation will be the output of the 12 heads:

$$output\_multi\text{-}head\_attention=\{z_0, z_1, z_2,..., z_{11}\}$$

- BERT$_{\text{LARGE}}$, which contains a stack of N=24 encoder layers. d_{model} = 1024. A multi-head attention sub-layer contains A=16 heads. The dimension of each head z_A also remains 64 as in the original Transformer model:

$$d_k = \frac{d_{model}}{A} = \frac{1024}{16} = 64$$

- The output of each multi-head attention sub-layer before concatenation will be the output of the 16 heads:

$$output\_multi\text{-}head\_attention=\{z_0, z_1, z_2,..., z_{15}\}$$

The sizes of the models can be summed up as follows:

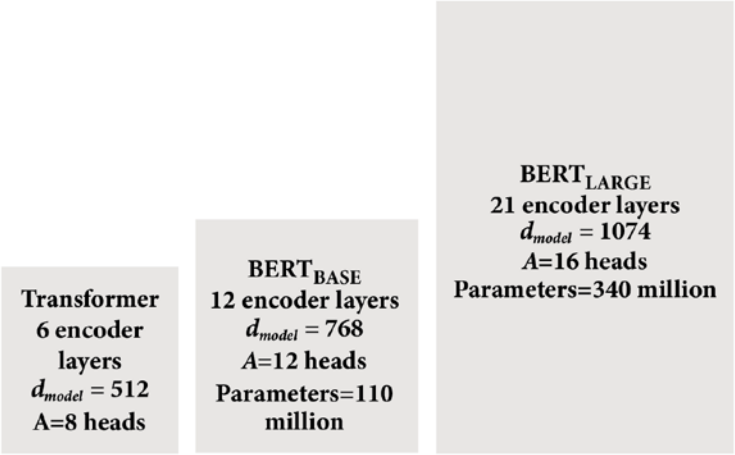


Figure 3.2: Transformer models

BERT models are not limited to these three configurations. These three configurations illustrate the main aspects of BERT models. Numerous variations are possible.

Size and dimensions play an essential role in BERT-style pretraining. BERT models are like humans. BERT models produce better results with more working memory (dimensions) and more knowledge (data). Large transformer models that learn large amounts of data will pretrain better for downstream NLP tasks.

Let's go to the first sub-layer and see the fundamental aspects of input embedding and positional encoding in a BERT model.

Preparing the pretraining input environment

The BERT model has no decoder stack of layers. As such, it does not have a masked multi-head attention sub-layer. BERT designers state that a masked multi-head attention layer that masks the rest of the sequence impedes the attention process.

A masked multi-head attention layer masks all of the tokens that are beyond the present position. For example, take the following sentence:

```
The cat sat on it because it was a nice rug.
```

If we have just reached the word `it`, the input of the encoder could be:

```
The cat sat on it<masked sequence>
```

The motivation of this approach is to prevent the model from seeing the output it is supposed to predict. This left-to-right approach produces relatively good results.

However, the model cannot learn much this way. To know what `it` refers to, we need to see the whole sentence to reach the word `rug` and figure out that `it` was the rug.

The authors of BERT came up with an idea. Why not pretrain the model to make predictions using a different approach?

The authors of BERT came up with bidirectional attention, letting an attention head attend to *all* of the words both from left to right and right to left. In other words, the self-attention mask of an encoder could do the job without being hindered by the masked multi-head attention sub-layer of the decoder.

The model was trained with two tasks. The first method is **Masked Language Modeling** (**MLM**). The second method is **Next Sentence Prediction** (**NSP**).

Let's start with masked language modeling.

Masked language modeling

Masked language modeling does not require training a model with a sequence of visible words followed by a masked sequence to predict.

BERT introduces the *bidirectional* analysis of a sentence with a random mask on a word of the sentence.

It is important to note that BERT applies `WordPiece`, a subword segmentation tokenization method, to the inputs. It also uses learned positional encoding, not the sine-cosine approach.

A potential input sequence could be:

```
The cat sat on it because it was a nice rug.
```

The decoder would mask the attention sequence after the model reached the word `it`:

```
The cat sat on it <masked sequence>.
```

But the BERT encoder masks a random token to make a prediction:

```
The cat sat on it [MASK] it was a nice rug.
```

The multi-attention sub-layer can now see the whole sequence, run the self-attention process, and predict the masked token.

The input tokens were masked in a tricky way *to force the model to train longer but produce better results* with three methods:

- Surprise the model by not masking a single token on 10% of the dataset; for example:

  ```
  The cat sat on it [because] it was a nice rug.
  ```

- Surprise the model by replacing the token with a random token on 10% of the dataset; for example:

  ```
  The cat sat on it [often] it was a nice rug.
  ```

- Replace a token by a `[MASK]` token on 80% of the dataset; for example:

  ```
  The cat sat on it [MASK] it was a nice rug.
  ```

The authors' bold approach avoids overfitting and forces the model to train efficiently.

BERT was also trained to perform next sentence prediction.

Next sentence prediction

The second method found to train BERT is **Next Sentence Prediction** (**NSP**). The input contains two sentences. In 50% of the cases, the second sentence is the actual second sentence of a document. In 50% of the cases, the second sentence was selected randomly and has no relation to the first one.

Two new tokens were added:

- `[CLS]` is a binary classification token added to the beginning of the first sequence to predict if the second sequence follows the first sequence. A positive sample is usually a pair of consecutive sentences taken from a dataset. A negative sample is created using sequences from different documents.
- `[SEP]` is a separation token that signals the end of a sequence.

For example, the input sentences taken out of a book could be:

```
The cat slept on the rug. It likes sleeping all day.
```

These two sentences will become one complete input sequence:

```
[CLS] the cat slept on the rug [SEP] it likes sleep ##ing all day[SEP]
```

This approach requires additional encoding information to distinguish sequence A from sequence B.

If we put the whole embedding process together, we obtain:

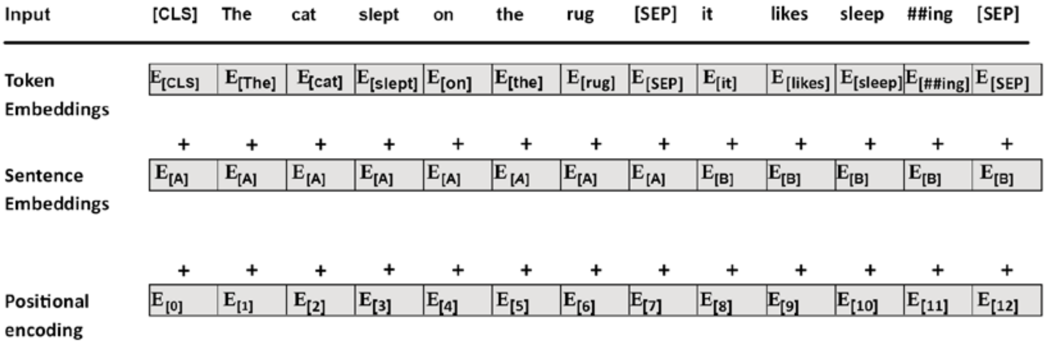

Input
[CLS] The cat slept on the rug [SEP] it likes sleep ##ing [SEP]
Token Embeddings
E[CLS] E[The] E[cat] E[slept] E[on] E[the] E[rug] E[SEP] E[it] E[likes] E[sleep] E[##ing] E[SEP]
+ + + + + + + + + + + + +
Sentence Embeddings
E[A] E[A] E[A] E[A] E[A] E[A] E[A] E[A] E[B] E[B] E[B] E[B] E[B]
+ + + + + + + + + + + + +
Positional encoding
E[0] E[1] E[2] E[3] E[4] E[5] E[6] E[7] E[8] E[9] E[10] E[11] E[12]

Figure 3.3: Input embeddings

The input embeddings are obtained by summing the token embeddings, the segment (sentence, phrase, word) embeddings, and the positional encoding embeddings.

The input embedding and positional encoding sub-layer of a BERT model can be summed up as follows:

- A sequence of words is broken down into `WordPiece` tokens.
- A `[MASK]` token will randomly replace the initial word tokens for masked language modeling training.
- A `[CLS]` classification token is inserted at the beginning of a sequence for classification purposes.
- A `[SEP]` token separates two sentences (segments, phrases) for NSP training.
- Sentence embedding is added to token embedding, so that sentence A has a different sentence embedding value than sentence B.
- Positional encoding is learned. The sine-cosine positional encoding method of the original Transformer is not applied.

Some additional key features are:

- BERT uses bidirectional attention in its multi-head attention sub-layers, opening vast horizons of learning and understanding relationships between tokens.
- BERT introduces scenarios of unsupervised embedding, pretraining models with unlabeled text. Unsupervised scenarios force the model to think harder during the multi-head attention learning process. This makes BERT learn how languages are built and apply this knowledge to downstream tasks without having to pretrain each time.
- BERT also uses supervised learning, covering all bases in the pretraining process.

BERT has improved the training environment of transformers. Let’s now see the motivation of pretraining and how it helps the fine-tuning process.

Pretraining and fine-tuning a BERT model

BERT is a two-step framework. The first step is pretraining, and the second is fine-tuning, as shown in *Figure 3.4*:

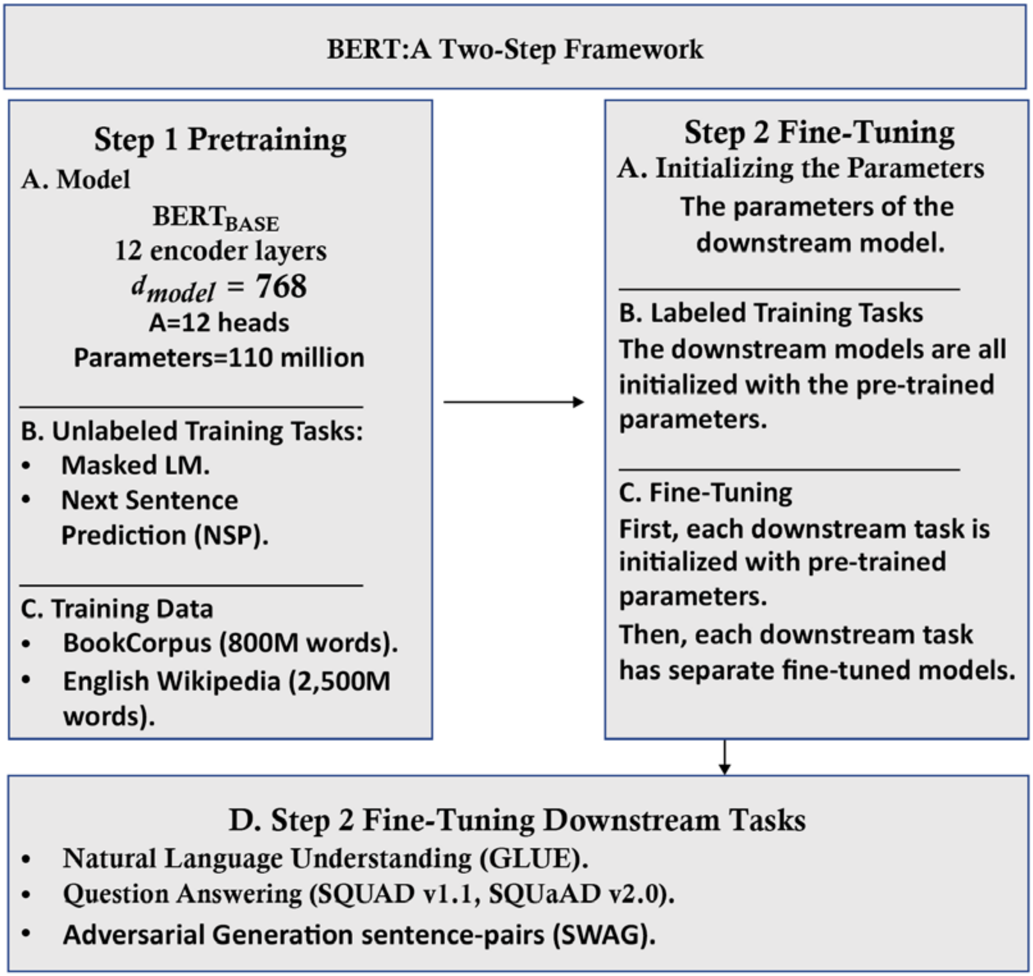

BERT:A Two-Step Framework
Step 1 Pretraining
A. Model
$\text{BERT}_{\text{BASE}}$
12 encoder layers
$d_{model} = 768$
A=12 heads
Parameters=110 million
B. Unlabeled Training Tasks:
• Masked LM.
• Next Sentence Prediction (NSP).
C. Training Data
• BookCorpus (800M words).
• English Wikipedia (2,500M words).
Step 2 Fine-Tuning
A. Initializing the Parameters
The parameters of the downstream model.
B. Labeled Training Tasks
The downstream models are all initialized with the pre-trained parameters.
C. Fine-Tuning
First, each downstream task is initialized with pre-trained parameters.
Then, each downstream task has separate fine-tuned models.
D. Step 2 Fine-Tuning Downstream Tasks
• Natural Language Understanding (GLUE).
• Question Answering (SQUAD v1.1, SQUaAD v2.0).
• Adversarial Generation sentence-pairs (SWAG).

Figure 3.4: The BERT framework

Training a transformer model can take hours, if not days. It takes quite some time to engineer the architecture and parameters and select the proper datasets to train a transformer model.

Pretraining is the first step of the BERT framework, which can be broken down into two sub-steps:

- Defining the model's architecture: number of layers, number of heads, dimensions, and the other building blocks of the model
- Training the model on **MLM** and **NSP** tasks

The second step of the BERT framework is fine-tuning, which can also be broken down into two sub-steps:

- Initializing the downstream model chosen with the trained parameters of the pretrained BERT model
- Fine-tuning the parameters for specific downstream tasks such as **Recognizing Textual Entailment** (**RTE**), question answering (`SQuAD v1.1`, `SQuAD v2.0`), and **Situations With Adversarial Generations** (**SWAG**)

In this section, we covered the information we need to fine-tune a BERT model. In the following chapters, we will explore the topics we brought up in this section in more depth:

- In *Chapter 4*, *Pretraining a RoBERTa Model from Scratch*, we will pretrain a BERT-like model from scratch in 15 steps. We will even compile our data, train a tokenizer, and then train the model. This chapter aims to first go through the specific building blocks of BERT and then fine-tune an existing model.
- In *Chapter 5*, *Downstream NLP Tasks with Transformers*, we will go through many downstream tasks, exploring `GLUE`, `SQuAD v1.1`, `SQuAD`, `SWAG`, and several other NLP evaluation datasets. We will run several downstream transformer models to illustrate key tasks. The goal of this chapter is to fine-tune a downstream model.
- In *Chapter 7*, *The Rise of Suprahuman Transformers with GPT-3 Engines*, will explore the architecture and unleashed usage of OpenAI `GPT-2` and `GPT-3` transformers. $\text{BERT}_{\text{BASE}}$ was configured to be close to OpenAI `GPT` to show that it produced better performance. However, OpenAI transformers keep evolving too! We will see how they have reached suprahuman NLP levels.

In this chapter, the BERT model we will fine-tune will be trained on **The Corpus of Linguistic Acceptability** (**CoLA**). The downstream task is based on *Neural Network Acceptability Judgments* by *Alex Warstadt*, *Amanpreet Singh*, and *Samuel R. Bowman*.

We will fine-tune a BERT model that will determine the grammatical acceptability of a sentence. The fine-tuned model will have acquired a certain level of linguistic competence.

We have gone through BERT architecture and its pretraining and fine-tuning framework. Let's now fine-tune a BERT model.

Fine-tuning BERT

This section will fine-tune a BERT model to predict the downstream task of *Acceptability Judgments* and measure the predictions with the **Matthews Correlation Coefficient** (**MCC**), which will be explained in the *Evaluating using Matthews Correlation Coefficient* section of this chapter.

Open `BERT_Fine_Tuning_Sentence_Classification_GPU.ipynb` in Google Colab (make sure you have an email account). The notebook is in `Chapter03` in the GitHub repository of this book.

The title of each cell in the notebook is also the same as or very close to the title of each subsection of this chapter.

We will first examine why transformer models must take hardware constraints into account.

Hardware constraints

Transformer models require multiprocessing hardware. Go to the **Runtime** menu in Google Colab, select **Change runtime type**, and select **GPU** in the **Hardware Accelerator** drop-down list.

Transformer models are hardware-driven. I recommend reading *Appendix II*, *Hardware Constraints for Transformer Models*, before continuing this chapter.

The program will be using `Hugging Face` modules, which we'll install next.

Installing the Hugging Face PyTorch interface for BERT

Hugging Face provides a pretrained BERT model. Hugging Face developed a base class named `PreTrainedModel`. By installing this class, we can load a model from a pretrained model configuration.

Hugging Face provides modules in TensorFlow and PyTorch. I recommend that a developer be comfortable with both environments. Excellent AI research teams use either or both environments.

In this chapter, we will install the modules required as follows:

```
#@title Installing the Hugging Face PyTorch Interface for Bert
!pip install -q transformers
```

The installation will run, or requirement satisfied messages will be displayed.

We can now import the modules needed for the program.

Importing the modules

We will import the pretrained modules required, such as the pretrained `BERT tokenizer` and the configuration of the BERT model. The `BERTAdam` optimizer is imported along with the sequence classification module:

```
#@title Importing the modules
import torch
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader, RandomSampler,
SequentialSampler
from keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
from transformers import BertTokenizer, BertConfig
from transformers import AdamW, BertForSequenceClassification, get_linear_
schedule_with_warmup
```

A nice progress bar module is imported from `tqdm`:

```
from tqdm import tqdm, trange
```

We can now import the widely used standard Python modules:

```
import pandas as pd
import io
import numpy as np
import matplotlib.pyplot as plt
% matplotlib inline
```

If all goes well, no message will be displayed, bearing in mind that Google Colab has pre-installed the modules on the VM we are using.

Specifying CUDA as the device for torch

We will now specify that torch uses the **Compute Unified Device Architecture** (**CUDA**) to put the parallel computing power of the NVIDIA card to work for our multi-head attention model:

```
#@title Harware verification and device attribution
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
!nvidia-smi
```

The output may vary with Google Colab configurations. See *Appendix II*: *Hardware Constraints for Transformer Models* for explanations and screenshots.

We will now load the dataset.

Loading the dataset

We will now load the *CoLA* based on the *Warstadt* et al. (2018) paper.

General Language Understanding Evaluation (**GLUE**) considers *Linguistic Acceptability* as a top-priority NLP task. In *Chapter 5*, *Downstream NLP Tasks with Transformers*, we will explore the key tasks transformers must perform to prove their efficiency.

The following cells in the notebook automatically download the necessary files:

```
import os
!curl -L https://raw.githubusercontent.com/Denis2054/Transformers-for-
NLP-2nd-Edition/master/Chapter03/in_domain_train.tsv --output "in_domain_
train.tsv"

!curl -L https://raw.githubusercontent.com/Denis2054/Transformers-for-
NLP-2nd-Edition/master/Chapter03/out_of_domain_dev.tsv --output "out_of_
domain_dev.tsv"
```

You should see them appear in the file manager:

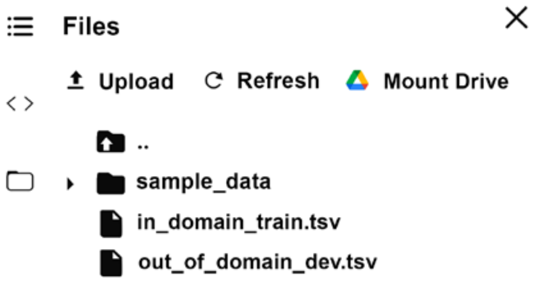


Figure 3.5: Uploading the datasets

Now the program will load the datasets:

```
#@title Loading the Dataset
#source of dataset : https://nyu-mll.github.io/CoLA/
df = pd.read_csv("in_domain_train.tsv", delimiter='\t', header=None,
names=['sentence_source', 'label', 'label_notes', 'sentence'])
df.shape
```

The output displays the shape of the dataset we have imported:

```
(8551, 4)
```

A 10-line sample is displayed to visualize the *Acceptability Judgment* task and see if a sequence makes sense or not:

```
df.sample(10)
```

The output shows `10` lines of the labeled dataset, which may change after each run:

	sentence_source	**label**	**label_notes**	**sentence**
1742	r-67	1	NaN	they said that tom would n't pay up , but pay…
937	bc01	1	NaN	although he likes cabbage too , fred likes egg…
5655	c_13	1	NaN	wendy 's mother country is iceland .
500	bc01	0	*	john is wanted to win .
4596	ks08	1	NaN	i did n't find any bugs in my bed .
7412	sks13	1	NaN	the girl he met at the departmental party will...
8456	ad03	0	*	peter is the old pigs .
744	bc01	0	*	frank promised the men all to leave .
5420	b_73	0	*	i 've seen as much of a coward as frank .
5749	c_13	1	NaN	we drove all the way to buenos aires .

Each sample in the `.tsv` files contains four tab-separated columns:

- Column 1: the source of the sentence (code)
- Column 2: the label (`0`=unacceptable, `1`=acceptable)
- Column 3: the label annotated by the author
- Column 4: the sentence to be classified

You can open the `.tsv` files locally to read a few samples of the dataset. The program will now process the data for the BERT model.

Creating sentences, label lists, and adding BERT tokens

The program will now create the sentences as described in the *Preparing the pretraining input environment* section of this chapter:

```
#@ Creating sentence, label lists and adding Bert tokens
sentences = df.sentence.values
# Adding CLS and SEP tokens at the beginning and end of each sentence for
BERT
sentences = ["[CLS] " + sentence + " [SEP]" for sentence in sentences]
labels = df.label.values
```

The `[CLS]` and `[SEP]` have now been added.

The program now activates the tokenizer.

Activating the BERT tokenizer

In this section, we will initialize a pretrained BERT tokenizer. This will save the time it would take to train it from scratch.

The program selects an uncased tokenizer, activates it, and displays the first tokenized sentence:

```
#@title Activating the BERT Tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_lower_
case=True)
tokenized_texts = [tokenizer.tokenize(sent) for sent in sentences]
print ("Tokenize the first sentence:")
print (tokenized_texts[0])
```

The output contains the classification token and the sequence segmentation token:

```
Tokenize the first sentence:
['[CLS]', 'our', 'friends', 'wo', 'n', "'", 't', 'buy', 'this',
'analysis', ',', 'let', 'alone', 'the', 'next', 'one', 'we', 'propose',
'.', '[SEP]']
```

The program will now process the data.

Processing the data

We need to determine a fixed maximum length and process the data for the model. The sentences in the datasets are short. But, to make sure of this, the program sets the maximum length of a sequence to 128, and the sequences are padded:

```
#@title Processing the data
# Set the maximum sequence length. The longest sequence in our training
set is 47, but we'll leave room on the end anyway.
# In the original paper, the authors used a length of 512.
MAX_LEN = 128
# Use the BERT tokenizer to convert the tokens to their index numbers in
the BERT vocabulary
input_ids = [tokenizer.convert_tokens_to_ids(x) for x in tokenized_texts]
# Pad our input tokens
input_ids = pad_sequences(input_ids, maxlen=MAX_LEN, dtype="long",
truncating="post", padding="post")
```

The sequences have been processed and now the program creates the attention masks.

Creating attention masks

Now comes a tricky part of the process. We padded the sequences in the previous cell. But we want to prevent the model from performing attention to those padded tokens!

The idea is to apply a mask with a value of 1 for each token, which 0s will follow for padding:

```
#@title Create attention masks
attention_masks = []
# Create a mask of 1s for each token followed by 0s for padding
for seq in input_ids:
  seq_mask = [float(i>0) for i in seq]
  attention_masks.append(seq_mask)
```

The program will now split the data.

Splitting the data into training and validation sets

The program now performs the standard process of splitting the data into training and validation sets:

```
#@title Splitting data into train and validation sets
# Use train_test_split to split our data into train and validation sets
```

```
for training
train_inputs, validation_inputs, train_labels, validation_labels = train_
test_split(input_ids, labels, random_state=2018, test_size=0.1)
train_masks, validation_masks, _, _ = train_test_split(attention_masks,
input_ids,random_state=2018, test_size=0.1)
```

The data is ready to be trained, but it still needs to be adapted to torch.

Converting all the data into torch tensors

The fine-tuning model uses torch tensors. The program must convert the data into torch tensors:

```
#@title Converting all the data into torch tensors
# Torch tensors are the required datatype for our model
train_inputs = torch.tensor(train_inputs)
validation_inputs = torch.tensor(validation_inputs)
train_labels = torch.tensor(train_labels)
validation_labels = torch.tensor(validation_labels)
train_masks = torch.tensor(train_masks)
validation_masks = torch.tensor(validation_masks)
```

The conversion is over. Now we need to create an iterator.

Selecting a batch size and creating an iterator

In this cell, the program selects a batch size and creates an iterator. The iterator is a clever way of avoiding a loop that would load all the data in memory. The iterator, coupled with the torch `DataLoader`, can batch train massive datasets without crashing the machine's memory.

In this model, the batch size is 32:

```
#@title Selecting a Batch Size and Creating and Iterator
# Select a batch size for training. For fine-tuning BERT on a specific
task, the authors recommend a batch size of 16 or 32
batch_size = 32
# Create an iterator of our data with torch DataLoader. This helps save on
memory during training because, unlike a for loop,
# with an iterator the entire dataset does not need to be loaded into
memory
train_data = TensorDataset(train_inputs, train_masks, train_labels)
train_sampler = RandomSampler(train_data)
train_dataloader = DataLoader(train_data, sampler=train_sampler, batch_
size=batch_size)
validation_data = TensorDataset(validation_inputs, validation_masks,
validation_labels)
validation_sampler = SequentialSampler(validation_data)
validation_dataloader = DataLoader(validation_data, sampler=validation_
sampler, batch_size=batch_size)
```

The data has been processed and is all set. The program can now load and configure the BERT model.

BERT model configuration

The program now initializes a BERT uncased configuration:

```
#@title BERT Model Configuration
# Initializing a BERT bert-base-uncased style configuration
#@title Transformer Installation
try:
  import transformers
except:
  print("Installing transformers")
  !pip -qq install transformers

from transformers import BertModel, BertConfig
configuration = BertConfig()
# Initializing a model from the bert-base-uncased style configuration
model = BertModel(configuration)
# Accessing the model configuration
configuration = model.config
print(configuration)
```

The output displays the main Hugging Face parameters similar to the following (the library is often updated):

```
BertConfig {
  "attention_probs_dropout_prob": 0.1,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "bert",
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "pad_token_id": 0,
  "type_vocab_size": 2,
  "vocab_size": 30522
}
```

Let's go through these main parameters:

- `attention_probs_dropout_prob`: `0.1` applies a `0.1` dropout ratio to the attention probabilities.
- `hidden_act`: `"gelu"` is a non-linear activation function in the encoder. It is a *Gaussian Error Linear Units* activation function. The input is weighted by its magnitude, which makes it non-linear.
- `hidden_dropout_prob`: `0.1` is the dropout probability applied to the fully connected layers. Full connections can be found in the embeddings, encoder, and pooler layers. The output is not always a good reflection of the content of a sequence. Pooling the sequence of hidden states improves the output sequence.
- `hidden_size`: `768` is the dimension of the encoded layers and also the pooler layer.
- `initializer_range`: `0.02` is the standard deviation value when initializing the weight matrices.
- `intermediate_size`: `3072` is the dimension of the feed-forward layer of the encoder.
- `layer_norm_eps`: `1e-12` is the epsilon value for layer normalization layers.
- `max_position_embeddings`: `512` is the maximum length the model uses.
- `model_type`: `"bert"` is the name of the model.
- `num_attention_heads`: `12` is the number of heads.
- `num_hidden_layers`: `12` is the number of layers.
- `pad_token_id`: `0` is the ID of the padding token to avoid training padding tokens.
- `type_vocab_size`: `2` is the size of the `token_type_ids`, which identify the sequences. For example, "the dog`[SEP]` The `cat.[SEP]`" can be represented with token IDs `[0,0,0, 1,1,1]`.

- `vocab_size`: `30522` is the number of different tokens used by the model to represent the `input_ids`.

With these parameters in mind, we can load the pretrained model.

Loading the Hugging Face BERT uncased base model

The program now loads the pretrained BERT model:

```
#@title Loading the Hugging Face Bert uncased base model
model = BertForSequenceClassification.from_pretrained("bert-base-uncased",
num_labels=2)
model = nn.DataParallel(model)
model.to(device)
```

We have defined the model, defined parallel processing, and sent the model to the device. For more explanations, see *Appendix II*, *Hardware Constraints for Transformer Models*.

This pretrained model can be trained further if necessary. It is interesting to explore the architecture in detail to visualize the parameters of each sublayer, as shown in the following excerpt:

```
BertForSequenceClassification(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(30522, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): BertLayerNorm()
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768,
bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768,
bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768,
bias=True)
              (LayerNorm): BertLayerNorm()
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): BertLayerNorm()
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (1): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768,
bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768,
bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768,
bias=True)
              (LayerNorm): BertLayerNorm()
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): BertLayerNorm()
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
```

Let's now go through the main parameters of the optimizer.

Optimizer grouped parameters

The program will now initialize the optimizer for the model's parameters. Fine-tuning a model begins with initializing the pretrained model parameter values (not their names).

The parameters of the optimizer include a weight decay rate to avoid overfitting, and some parameters are filtered.

The goal is to prepare the model's parameters for the training loop:

```
##@title Optimizer Grouped Parameters
#This code is taken from:
# https://github.com/huggingface/transformers/
blob/5bfcd0485ece086ebcbed2d008813037968a9e58/examples/run_glue.py#L102
# Don't apply weight decay to any parameters whose names include these
tokens.
# (Here, the BERT doesn't have 'gamma' or 'beta' parameters, only 'bias'
terms)
param_optimizer = list(model.named_parameters())
no_decay = ['bias', 'LayerNorm.weight']
# Separate the 'weight' parameters from the 'bias' parameters.
# - For the 'weight' parameters, this specifies a 'weight_decay_rate' of
0.01.
# - For the 'bias' parameters, the 'weight_decay_rate' is 0.0.
optimizer_grouped_parameters = [
    # Filter for all parameters which *don't* include 'bias', 'gamma',
'beta'.
    {'params': [p for n, p in param_optimizer if not any(nd in n for nd in
no_decay)],
     'weight_decay_rate': 0.1},

    # Filter for parameters which *do* include those.
    {'params': [p for n, p in param_optimizer if any(nd in n for nd in
no_decay)],
     'weight_decay_rate': 0.0}
]
# Note - 'optimizer_grouped_parameters' only includes the parameter
values, not the names.
```

The parameters have been prepared and cleaned up. They are ready for the training loop.

The hyperparameters for the training loop

The hyperparameters for the training loop are critical, though they seem innocuous. Adam will activate weight decay and also go through a warm-up phase, for example.

The learning rate (`lr`) and warm-up rate (`warmup`) should be set to a very small value early in the optimization phase and gradually increase after a certain number of iterations. This avoids large gradients and overshooting the optimization goals.

Some researchers argue that the gradients at the output level of the sub-layers before layer normalization do not require a warm-up rate. Solving this problem requires many experimental runs.

The optimizer is a BERT version of Adam called `BertAdam`:

```
#@title The Hyperparameters for the Training Loop
optimizer = BertAdam(optimizer_grouped_parameters,
                     lr=2e-5,
                     warmup=.1)
```

The program adds an accuracy measurement function to compare the predictions to the labels:

```
#Creating the Accuracy Measurement Function
# Function to calculate the accuracy of our predictions vs labels
def flat_accuracy(preds, labels):
    pred_flat = np.argmax(preds, axis=1).flatten()
    labels_flat = labels.flatten()
    return np.sum(pred_flat == labels_flat) / len(labels_flat)
```

The data is ready. The parameters are ready. It's time to activate the training loop!

The training loop

The training loop follows standard learning processes. The number of epochs is set to 4, and measurement for loss and accuracy will be plotted. The training loop uses the `dataloader` to load and train batches. The training process is measured and evaluated.

The code starts by initializing the `train_loss_set`, which will store the loss and accuracy, which will be plotted. It starts training its epochs and runs a standard training loop, as shown in the following excerpt:

```
#@title The Training Loop
t = []
# Store our loss and accuracy for plotting
train_loss_set = []
# Number of training epochs (authors recommend between 2 and 4)
epochs = 4
# trange is a tqdm wrapper around the normal python range
for _ in trange(epochs, desc="Epoch"):
…./…
    tmp_eval_accuracy = flat_accuracy(logits, label_ids)

    eval_accuracy += tmp_eval_accuracy
    nb_eval_steps += 1
  print("Validation Accuracy: {}".format(eval_accuracy/nb_eval_steps))
```

The output displays the information for each `epoch` with the `trange` wrapper, `for _ in trange(epochs, desc="Epoch"):`

```
***output***
Epoch:   0%|          | 0/4 [00:00<?, ?it/s]
Train loss: 0.5381132976395461
Epoch:  25%|██▌       | 1/4 [07:54<23:43, 474.47s/it]
Validation Accuracy: 0.788966049382716
Train loss: 0.315329696132929
Epoch:  50%|█████     | 2/4 [15:49<15:49, 474.55s/it]
Validation Accuracy: 0.836033950617284
Train loss: 0.1474070605354314
Epoch:  75%|███████▌  | 3/4 [23:43<07:54, 474.53s/it]
Validation Accuracy: 0.814429012345679
Train loss: 0.07655430570461196
Epoch: 100%|██████████| 4/4 [31:38<00:00, 474.58s/it]
Validation Accuracy: 0.810570987654321
```

Transformer models are evolving very quickly, and deprecation messages and even errors might occur. Hugging Face is no exception to this, and we must update our code accordingly when this happens.

The model is trained. We can now display the training evaluation.

Training evaluation

The loss and accuracy values were stored in `train_loss_set` as defined at the beginning of the training loop.

The program now plots the measurements:

```
#@title Training Evaluation
plt.figure(figsize=(15,8))
plt.title("Training loss")
plt.xlabel("Batch")
plt.ylabel("Loss")
plt.plot(train_loss_set)
plt.show()
```

The output is a graph that shows that the training process went well and was efficient:

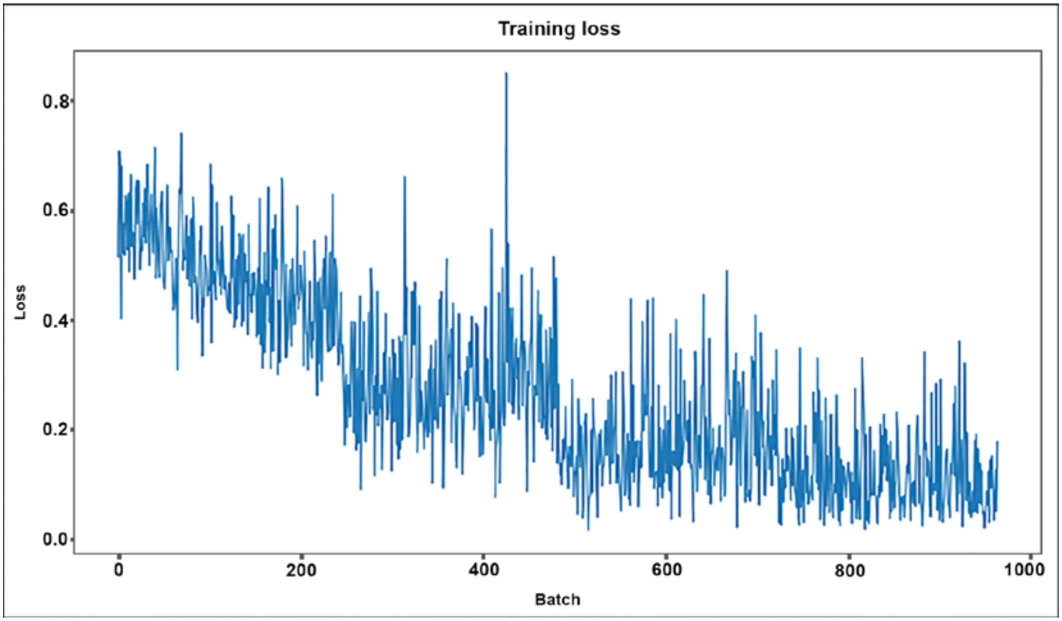


Figure 3.6: Training loss per batch

The model has been fine-tuned. We can now run predictions.

Predicting and evaluating using the holdout dataset

The BERT downstream model was trained with the `in_domain_train.tsv` dataset. The program will now make predictions using the holdout (testing) dataset in the `out_of_domain_dev.tsv` file. The goal is to predict whether the sentence is grammatically correct.

The following excerpt of the code shows that the data preparation process applied to the training data is repeated in the part of the code for the holdout dataset:

```
#@title Predicting and Evaluating Using the Holdout Dataset
df = pd.read_csv("out_of_domain_dev.tsv", delimiter='\t', header=None,
names=['sentence_source', 'label', 'label_notes', 'sentence'])
# Create sentence and label lists
sentences = df.sentence.values
# We need to add special tokens at the beginning and end of each sentence
for BERT to work properly
sentences = ["[CLS] " + sentence + " [SEP]" for sentence in sentences]
labels = df.label.values
tokenized_texts = [tokenizer.tokenize(sent) for sent in sentences]
.../...
```

The program then runs batch predictions using the `dataloader`:

```
# Predict
for batch in prediction_dataloader:
  # Add batch to GPU
  batch = tuple(t.to(device) for t in batch)
  # Unpack the inputs from our dataloader
  b_input_ids, b_input_mask, b_labels = batch
  # Telling the model not to compute or store gradients, saving memory and
speeding up prediction
  with torch.no_grad():
    # Forward pass, calculate logit predictions
    logits = model(b_input_ids, token_type_ids=None, attention_mask=b_
input_mask)
```

The logits and labels of the predictions are moved to the CPU:

```
  # Move logits and labels to CPU
  logits =  logits['logits'].detach().cpu().numpy()
  label_ids = b_labels.to('cpu').numpy()
```

The predictions and their true labels are stored:

```
  # Store predictions and true labels
  predictions.append(logits)
  true_labels.append(label_ids)
```

The program can now evaluate the predictions.

Evaluating using the Matthews Correlation Coefficient

The **Matthews Correlation Coefficient** (**MCC**) was initially designed to measure the quality of binary classifications and can be modified to be a multi-class correlation coefficient. A two-class classification can be made with four probabilities at each prediction:

- TP = True Positive
- TN = True Negative
- FP = False Positive
- FN = False Negative

Brian W. Matthews, a biochemist, designed it in 1975, inspired by his predecessors' *phi* function. Since then, it has evolved into various formats such as the following one:

$$MCC = \frac{TP\ x\ TN - FP\ x\ FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

The value produced by MCC is between `-1` and `+1`. `+1` is the maximum positive value of a prediction. `-1` is an inverse prediction. `0` is an average random prediction.

GLUE evaluates *Linguistic Acceptability* with MCC.

MCC is imported from `sklearn.metrics`:

```
#@title Evaluating Using Matthew's Correlation Coefficient
# Import and evaluate each test batch using Matthew's correlation
coefficient
from sklearn.metrics import matthews_corrcoef
```

A set of predictions is created:

```
matthews_set = []
```

The MCC value is calculated and stored in matthews_set:

```
for i in range(len(true_labels)):
  matthews = matthews_corrcoef(true_labels[i],
                 np.argmax(predictions[i], axis=1).flatten())
  matthews_set.append(matthews)
```

You may see messages due to library and module version changes. The final score will be based on the entire test set, but let's look at the scores on the individual batches to get a sense of the variability in the metric between batches.

The scores of individual batches

Let's view the scores of the individual batches:

```
#@title Score of Individual Batches
matthews_set
```

The output produces MCC values between -1 and +1 as expected:

```
[0.049286405809014416,
 -0.2548235957188128,
 0.4732058754737091,
 0.30508307783296046,
 0.3567530340063379,
 0.8050112948805689,
 0.23329882422520506,
 0.47519096331149147,
 0.4364357804719848,
 0.4700159919404217,
 0.7679476477883045,
 0.8320502943378436,
 0.5807564950208268,
 0.5897435897435898,
 0.38461538461538464,
 0.5716350506349809,
 0.0]
```

Almost all the MCC values are positive, which is good news. Let's see what the evaluation is for the whole dataset.

Matthews evaluation for the whole dataset

The MCC is a practical way to evaluate a classification model.

The program will now aggregate the true values for the whole dataset:

```
#@title Matthew's Evaluation on the Whole Dataset
# Flatten the predictions and true values for aggregate Matthew's
evaluation on the whole dataset
flat_predictions = [item for sublist in predictions for item in sublist]
flat_predictions = np.argmax(flat_predictions, axis=1).flatten()
flat_true_labels = [item for sublist in true_labels for item in sublist]
matthews_corrcoef(flat_true_labels, flat_predictions)
```

MCC produces a correlation value between `-1` and `+1`. `0` is an average prediction, `-1` is an inverse one, and `1` is perfect. In this case, the output confirms that the MCC is positive, which shows that there is a correlation between this model and dataset:

```
0.45439842471680725
```

On this final positive evaluation of the fine-tuning of the BERT model, we have an overall view of the BERT training framework.

Summary

BERT brings bidirectional attention to transformers. Predicting sequences from left to right and masking the future tokens to train a model has serious limitations. If the masked sequence contains the meaning we are looking for, the model will produce errors. BERT attends to all of the tokens of a sequence at the same time.

We explored the architecture of BERT, which only uses the encoder stack of transformers. BERT was designed as a two-step framework. The first step of the framework is to pretrain a model. The second step is to fine-tune the model. We built a fine-tuning BERT model for an *Acceptability Judgment* downstream task. The fine-tuning process went through all phases of the process. First, we loaded the dataset and loaded the necessary pretrained modules of the model. Then the model was trained, and its performance was measured.

Fine-tuning a pretrained model takes fewer machine resources than training downstream tasks from scratch. Fine-tuned models can perform a variety of tasks. BERT proves that we can pretrain a model on two tasks only, which is remarkable in itself. But producing a multitask fine-tuned model based on the trained parameters of the BERT pretrained model is extraordinary.

Chapter 7, *The Rise of Suprahuman Transformers with GPT-3 Engines*, shows that OpenAI has reached a zero-shot level with little to no fine-tuning.

In this chapter, we fine-tuned a BERT model. In the next chapter, *Chapter 4*, *Pretraining a RoBERTa Model from Scratch*, we will dig deeper into the BERT framework and build a pretrained BERT-like model from scratch.

Questions

1. BERT stands for Bidirectional Encoder Representations from Transformers. (True/False)
2. BERT is a two-step framework. *Step 1* is pretraining. *Step 2* is fine-tuning. (True/False)
3. Fine-tuning a BERT model implies training parameters from scratch. (True/False)
4. BERT only pretrains using all downstream tasks. (True/False)
5. BERT pretrains with **Masked Language Modeling** (**MLM**). (True/False)
6. BERT pretrains with **Next Sentence Predictions** (**NSP**). (True/False)
7. BERT pretrains mathematical functions. (True/False)
8. A question-answer task is a downstream task. (True/False)
9. A BERT pretraining model does not require tokenization. (True/False)
10. Fine-tuning a BERT model takes less time than pretraining. (True/False)

References

- *Ashish Vaswani*, *Noam Shazeer*, *Niki Parmar*, *Jakob Uszkoreit*, *Llion Jones*, *Aidan N. Gomez*, *Lukasz Kaiser*, *Illia Polosukhin*, 2017, *Attention Is All You Need*: `https://arxiv.org/abs/1706.03762`
- *Jacob Devlin*, *Ming-Wei Chang*, *Kenton Lee*, and *Kristina Toutanova*, *2018*, *BERT: Pretraining of Deep Bidirectional Transformers for Language Understanding*: `https://arxiv.org/abs/1810.04805`
- *Alex Warstadt*, *Amanpreet Singh*, and *Samuel R. Bowman*, 2018, *Neural Network Acceptability Judgments*: `https://arxiv.org/abs/1805.12471`
- The **Corpus of Linguistic Acceptability** (**CoLA**): `https://nyu-mll.github.io/CoLA/`

- Documentation on Hugging Face models:
  - `https://huggingface.co/transformers/pretrained_models.html`
  - `https://huggingface.co/transformers/model_doc/bert.html`
  - `https://huggingface.co/transformers/model_doc/roberta.html`
  - `https://huggingface.co/transformers/model_doc/distilbert.html`

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the authors:

`https://www.packt.link/Transformers`

4

Pretraining a RoBERTa Model from Scratch

In this chapter, we will build a RoBERTa model from scratch. The model will use the bricks of the transformer construction kit we need for BERT models. Also, no pretrained tokenizers or models will be used. The RoBERTa model will be built following the fifteen-step process described in this chapter.

We will use the knowledge of transformers acquired in the previous chapters to build a model that can perform language modeling on masked tokens step by step. In *Chapter 2*, *Getting Started with the Architecture of the Transformer Model*, we went through the building blocks of the original Transformer. In *Chapter 3*, *Fine-Tuning BERT Models*, we fine-tuned a pretrained BERT model.

This chapter will focus on building a pretrained transformer model from scratch using a Jupyter notebook based on Hugging Face's seamless modules. The model is named KantaiBERT.

KantaiBERT first loads a compilation of Immanuel Kant's books created for this chapter. You will see how the data was obtained. You will also see how to create your own datasets for this notebook.

KantaiBERT trains its own tokenizer from scratch. It will build its merge and vocabulary files, which will be used during the pretraining process.

KantaiBERT then processes the dataset, initializes a trainer, and trains the model.

Finally, KantaiBERT uses the trained model to perform an experimental downstream language modeling task and fills a mask using Immanuel Kant's logic.

By the end of the chapter, you will know how to build a transformer model from scratch. You will have enough knowledge of transformers to face the Industry 4.0 challenge of using powerful pretrained transformers such as GPT-3 engines that require more than development skills to implement them. This chapter prepares you for *Chapter 7*, *The Rise of Suprahuman Transformers with GPT-3 Engines*.

This chapter covers the following topics:

- RoBERTa- and DistilBERT-like models
- How to train a tokenizer from scratch
- Byte-level byte-pair encoding
- Saving the trained tokenizer to files
- Recreating the tokenizer for the pretraining process
- Initializing a RoBERTa model from scratch
- Exploring the configuration of the model
- Exploring the 80 million parameters of the model
- Building the dataset for the trainer
- Initializing the trainer
- Pretraining the model
- Saving the model
- Applying the model to the downstream tasks of **Masked Language Modeling** (**MLM**)

Our first step will be to describe the transformer model that we are going to build.

Training a tokenizer and pretraining a transformer

In this chapter, we will train a transformer model named KantaiBERT using the building blocks provided by Hugging Face for BERT-like models. We covered the theory of the building blocks of the model we will be using in *Chapter 3*, *Fine-Tuning BERT Models*.

We will describe KantaiBERT, building on the knowledge we acquired in previous chapters.

KantaiBERT is a **Robustly Optimized BERT Pretraining Approach** (**RoBERTa**)-like model based on the architecture of BERT.

The initial BERT models brought innovative features to the initial transformer models, as we saw in *Chapter 3*. RoBERTa increases the performance of transformers for downstream tasks by improving the mechanics of the pretraining process.

For example, it does not use `WordPiece` tokenization but goes down to byte-level **Byte-Pair Encoding** (**BPE**). This method paved the way for a wide variety of BERT and BERT-like models.

In this chapter, KantaiBERT, like BERT, will be trained using **Masked Language Modeling** (**MLM**). MLM is a language modeling technique that masks a word in a sequence. The transformer model must train to predict the masked word.

KantaiBERT will be trained as a small model with 6 layers, 12 heads, and 84,095,008 parameters. It might seem that 84 million parameters is a lot. However, the parameters are spread over 12 heads, which makes it a relatively small model. A small model will make the pretraining experience smooth so that each step can be viewed in real time without waiting for hours to see a result.

KantaiBERT is a DistilBERT-like model because it has the same architecture of 6 layers and 12 heads. DistilBERT is a distilled version of BERT. DistilBERT, as the name suggests, contains fewer parameters than a RoBERTa model. As such, it runs much faster, but the results are slightly less accurate than with a RoBERTa model.

We know that large models achieve excellent performance. But what if you want to run a model on a smartphone? Miniaturization has been the key to technological evolution. Transformers will sometimes have to follow the same path during implementation. The Hugging Face approach using a distilled version of BERT is thus a good step forward. Distillation using fewer parameters or other such methods in the future is a clever way of taking the best of pretraining and making it efficient for the needs of many downstream tasks.

It is important to show all the possible architectures, including running a small model on a smartphone. However, the future of transformers will also be ready-to-use APIs, as we will see in *Chapter 7, The Rise of Suprahuman Transformers with GPT-3 Engines*.

KantaiBERT will implement a byte-level byte-pair encoding tokenizer like the one used by GPT-2. The special tokens will be the ones used by RoBERTa. BERT models most often use a WordPiece tokenizer.

There are no token type IDs to indicate which part of a segment a token is a part of. The segments will be separated with the separation token `</s>`.

KantaiBERT will use a custom dataset, train a tokenizer, train the transformer model, save it, and run it with an MLM example.

Let’s get going and build a transformer from scratch.

Building KantaiBERT from scratch

We will build KantaiBERT in 15 steps from scratch and run it on an MLM example.

Open Google Colaboratory (you need a Gmail account). Then upload `KantaiBERT.ipynb`, which is on GitHub in this chapter's directory.

The titles of the 15 steps of this section are similar to the titles of the notebook cells, which makes them easy to follow.

Let's start by loading the dataset.

Step 1: Loading the dataset

Ready-to-use datasets provide an objective way to train and compare transformers. In *Chapter 5, Downstream NLP Tasks with Transformers*, we will explore several datasets. However, this chapter aims to understand the training process of a transformer with notebook cells that can be run in real time without waiting for hours to obtain a result.

I chose to use the works of Immanuel Kant (1724-1804), the German philosopher who was the epitome of the *Age of Enlightenment*. The idea is to introduce human-like logic and pretrained reasoning for downstream reasoning tasks.

Project Gutenberg, `https://www.gutenberg.org`, offers a wide range of free eBooks that can be downloaded in text format. You can use other books if you want to create customized datasets of your own based on books.

I compiled the following three books by Immanuel Kant into a text file named `kant.txt`:

- *The Critique of Pure Reason*
- *The Critique of Practical Reason*
- *Fundamental Principles of the Metaphysic of Morals*

`kant.txt` provides a small training dataset to train the transformer model of this chapter. The result obtained remains experimental. For a real-life project, I would add the complete works of Immanuel Kant, Rene Descartes, Pascal, and Leibnitz, for example.

The text file contains the raw text of the books:

```
…For it is in reality vain to profess _indifference_ in regard to such
inquiries, the object of which cannot be indifferent to humanity.
```

The dataset is downloaded automatically from GitHub in the first cell of the `KantaiBERT.ipynb` notebook.

You can also load `kant.txt`, which is in the directory of this chapter on GitHub, using Colab's file manager. In this case, `curl` is used to retrieve it from GitHub:

```
#@title Step 1: Loading the Dataset
#1.Load kant.txt using the Colab file manager
#2.Downloading the file from GitHub
!curl -L https://raw.githubusercontent.com/Denis2054/Transformers-for-NLP-
2nd-Edition/master/Chapter04/kant.txt --output "kant.txt"
```

You can see it appear in the Colab file manager pane once you have loaded or downloaded it:

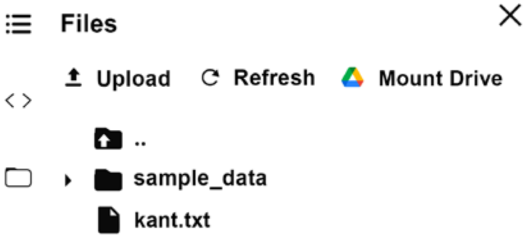


Figure 4.1: Colab file manager

Note that Google Colab deletes the files when you restart the VM.

The dataset is defined and loaded.

Do not run the subsequent cells without `kant.txt`. Training data is a prerequisite.

Now, the program will install the Hugging Face transformers.

Step 2: Installing Hugging Face transformers

We will need to install Hugging Face transformers and tokenizers, but we will not need TensorFlow in this instance of the Google Colab VM:

```
#@title Step 2:Installing Hugging Face Transformers
# We won't need TensorFlow here
```

```
!pip uninstall -y tensorflow
# Install 'transformers' from master
!pip install git+https://github.com/huggingface/transformers
!pip list | grep -E 'transformers|tokenizers'
# transformers version at notebook update --- 2.9.1
# tokenizers version at notebook update --- 0.7.0
```

The output displays the versions installed:

```
Successfully built transformers
tokenizers                0.7.0
transformers              2.10.0
```

Transformer versions are evolving at quite a speed. The version you run may differ and be displayed differently.

The program will now begin by training a tokenizer.

Step 3: Training a tokenizer

In this section, the program does not use a pretrained tokenizer. For example, a pretrained GPT-2 tokenizer could be used. However, the training process in this chapter includes training a tokenizer from scratch.

Hugging Face's `ByteLevelBPETokenizer()` will be trained using `kant.txt`. A BPE tokenizer will break a string or word down into substrings or subwords. There are two main advantages to this, among many others:

- The tokenizer can break words into minimal components. Then it will merge these small components into statistically interesting ones. For example, “`smaller" and smallest"` can become “`small`,” “`er`,” and “`est`.” The tokenizer can go further. We could get “`sm"` and “`all`,” for example. In any case, the words are broken down into subword tokens and smaller units of subword parts such as “`sm"` and “`all"` instead of simply “`small`.”
- The chunks of strings classified as unknown, `unk_token`, using `WordPiece` level encoding, will practically disappear.

In this model, we will be training the tokenizer with the following parameters:

- `files=paths` is the path to the dataset
- `vocab_size=52_000` is the size of our tokenizer's model length

- `min_frequency=2` is the minimum frequency threshold
- `special_tokens=[]` is a list of special tokens

In this case, the list of special tokens is:

- `<s>`: a start token
- `<pad>`: a padding token
- `</s>`: an end token
- `<unk>`: an unknown token
- `<mask>`: the mask token for language modeling

The tokenizer will be trained to generate merged substring tokens and analyze their frequency.

Let's take these two words in the middle of a sentence:

```
...the tokenizer...
```

The first step will be to tokenize the string:

```
'Ġthe', 'Ġtoken',   'izer',
```

The string is now tokenized into tokens with `Ġ` (whitespace) information.

The next step is to replace them with their indices:

'Ġthe'	'Ġtoken'	'izer'
150	5430	4712

Table 4.1: Indices for the three tokens

The program runs the tokenizer as expected:

```
#@title Step 3: Training a Tokenizer
%%time
from pathlib import Path
from tokenizers import ByteLevelBPETokenizer
paths = [str(x) for x in Path(".").glob("**/*.txt")]
# Initialize a tokenizer
tokenizer = ByteLevelBPETokenizer()
# Customize training
```

```
tokenizer.train(files=paths, vocab_size=52_000, min_frequency=2, special_
tokens=[
    "<s>",
    "<pad>",
    "</s>",
    "<unk>",
    "<mask>",
])
```

The tokenizer outputs the time taken to train:

```
CPU times: user 14.8 s, sys: 14.2 s, total: 29 s
Wall time: 7.72 s
```

The tokenizer is trained and ready to be saved.

Step 4: Saving the files to disk

The tokenizer will generate two files when trained:

- `merges.txt`, which contains the merged tokenized substrings
- `vocab.json`, which contains the indices of the tokenized substrings

The program first creates the `KantaiBERT` directory and then saves the two files:

```
#@title Step 4: Saving the files to disk
import os
token_dir = '/content/KantaiBERT'
if not os.path.exists(token_dir):
  os.makedirs(token_dir)
tokenizer.save_model('KantaiBERT')
```

The program output shows that the two files have been saved:

```
['KantaiBERT/vocab.json', 'KantaiBERT/merges.txt']
```

The two files should appear in the file manager pane:

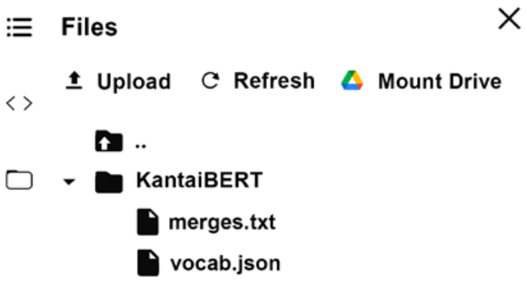


Figure 4.2: Colab file manager

The files in this example are small. You can double-click on them to view their contents. `merges.txt` contains the tokenized substrings as planned:

```
#version: 0.2 - Trained by 'huggingface/tokenizers'
Ġ t
h e
Ġ a
o n
i n
Ġ o
Ġt he
r e
i t
Ġo f
```

`vocab.json` contains the indices:

```
[…,"Ġthink":955,"preme":956,"ĠE":957,"Ġout":958,"Ġdut":959,
"aly":960,"Ġexp":961,…]
```

The trained tokenized dataset files are ready to be processed.

Step 5: Loading the trained tokenizer files

We could have loaded pretrained tokenizer files. However, we trained our own tokenizer and now are ready to load the files:

```
#@title Step 5 Loading the Trained Tokenizer Files
from tokenizers.implementations import ByteLevelBPETokenizer
from tokenizers.processors import BertProcessing
tokenizer = ByteLevelBPETokenizer(
    "./KantaiBERT/vocab.json",
    "./KantaiBERT/merges.txt",
)
```

The tokenizer can encode a sequence:

```
tokenizer.encode("The Critique of Pure Reason.").tokens
```

`"The Critique of Pure Reason"` will become:

```
['The', 'ĠCritique', 'Ġof', 'ĠPure', 'ĠReason', '.']
```

We can also ask to see the number of tokens in this sequence:

```
tokenizer.encode("The Critique of Pure Reason.")
```

The output will show that there are 6 tokens in the sequence:

```
Encoding(num_tokens=6, attributes=[ids, type_ids, tokens, offsets,
attention_mask, special_tokens_mask, overflowing])
```

The tokenizer now processes the tokens to fit the BERT model variant used in this notebook. The post-processor will add a start and end token; for example:

```
tokenizer._tokenizer.post_processor = BertProcessing(
    ("</s>", tokenizer.token_to_id("</s>")),
    ("<s>", tokenizer.token_to_id("<s>")),
)
tokenizer.enable_truncation(max_length=512)
```

Let's encode a post-processed sequence:

```
tokenizer.encode("The Critique of Pure Reason.")
```

The output shows that we now have 8 tokens:

```
Encoding(num_tokens=8, attributes=[ids, type_ids, tokens, offsets,
attention_mask, special_tokens_mask, overflowing])
```

If we want to see what was added, we can ask the tokenizer to encode the post-processed sequence by running the following cell:

```
tokenizer.encode("The Critique of Pure Reason.").tokens
```

The output shows that the start and end tokens have been added, which brings the number of tokens to 8, including start and end tokens:

```
['<s>', 'The', 'ĠCritique', 'Ġof', 'ĠPure', 'ĠReason', '.', '</s>']
```

The data for the training model is now ready to be trained. We will now check the system information of the machine we are running the notebook on.

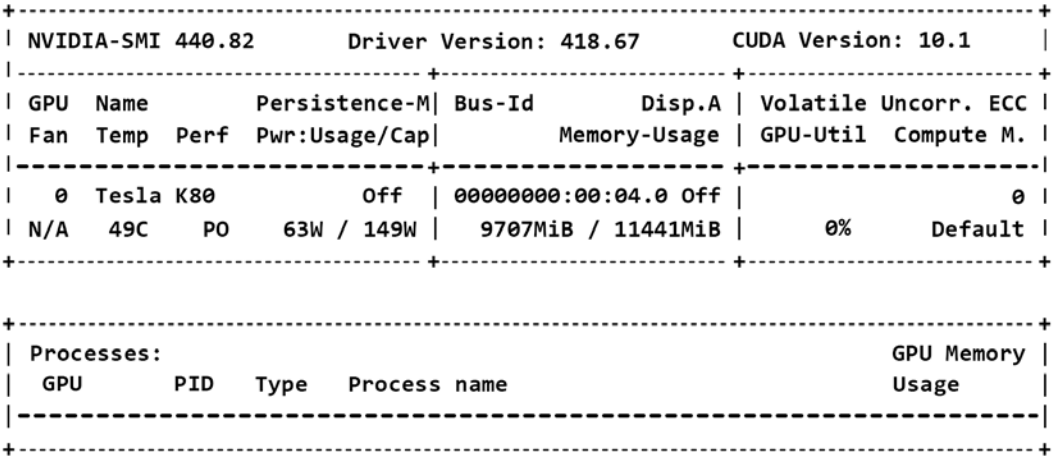
Step 6: Checking resource constraints: GPU and CUDA

KantaiBERT runs at optimal speed with a **Graphics Processing Unit** (**GPU**).

We will first run a command to see if an NVIDIA GPU card is present:

```
#@title Step 6: Checking Resource Constraints: GPU and NVIDIA
!nvidia-smi
```

The output displays the information and version on the card:



```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.82       Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|-------------------------------+----------------------+----------------------|
|   0  Tesla K80           Off  | 00000000:00:04.0 Off |                    0 |
| N/A   49C    PO    63W / 149W |   9707MiB / 11441MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|-----------------------------------------------------------------------------|
+-----------------------------------------------------------------------------+
```

Figure 4.3: Information on the NVIDIA card

The output may vary with each Google Colab VM configuration.

We will now check to make sure `PyTorch` sees CUDA:

```
#@title Checking that PyTorch Sees CUDA
import torch
torch.cuda.is_available()
```

The result should be `True`:

```
True
```

Compute Unified Device Architecture (**CUDA**) was developed by NVIDIA to use the parallel computing power of its GPUs.

For more on NVIDIA GPUs and CUDA, see *Appendix II*, *Hardware Constraints for Transformer Models*.

We are now ready to define the configuration of the model.

Step 7: Defining the configuration of the model

We will be pretraining a RoBERTa-type transformer model using the same number of layers and heads as a DistilBERT transformer. The model will have a vocabulary size set to 52,000, 12 attention heads, and 6 layers:

```
#@title Step 7: Defining the configuration of the Model
from transformers import RobertaConfig
config = RobertaConfig(
    vocab_size=52_000,
    max_position_embeddings=514,
    num_attention_heads=12,
    num_hidden_layers=6,
    type_vocab_size=1,
)
```

We will explore the configuration in more detail in *Step 9: Initializing a model from scratch*.

Let's first recreate the tokenizer in our model.

Step 8: Reloading the tokenizer in transformers

We are now ready to load our trained tokenizer, which is our pretrained tokenizer in `RobertaTokenizer.from_pretained()`:

```
#@title Step 8: Re-creating the Tokenizer in Transformers
from transformers import RobertaTokenizer
tokenizer = RobertaTokenizer.from_pretrained("./KantaiBERT", max_
length=512)
```

Now that we have loaded our trained tokenizer, let's initialize a RoBERTa model from scratch.

Step 9: Initializing a model from scratch

In this section, we will initialize a model from scratch and examine the size of the model.

The program first imports a RoBERTa masked model for language modeling:

```
#@title Step 9: Initializing a Model From Scratch
from transformers import RobertaForMaskedLM
```

The model is initialized with the configuration defined in *Step 7*:

```
model = RobertaForMaskedLM(config=config)
```

If we print the model, we can see that it is a BERT model with 6 layers and 12 heads:

```
print(model)
```

The building blocks of the encoder of the original Transformer model are present with different dimensions, as shown in this excerpt of the output:

```
RobertaForMaskedLM(
  (roberta): RobertaModel(
    (embeddings): RobertaEmbeddings(
      (word_embeddings): Embedding(52000, 768, padding_idx=1)
      (position_embeddings): Embedding(514, 768, padding_idx=1)
      (token_type_embeddings): Embedding(1, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, 
bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, 
bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, 
bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_
affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_
affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
…/…
```

Take some time to go through the details of the output of the configuration before continuing. You will get to know the model from the inside.

The LEGO®-type building blocks of transformers make it fun to analyze. For example, you will note that dropout regularization is present throughout the sublayers.

Now, let's explore the parameters.

Exploring the parameters

The model is small and contains 84,095,008 parameters.

We can check its size:

```
print(model.num_parameters())
```

The output shows the approximate number of parameters, which might vary from one transformer version to another:

```
84095008
```

Let's now look into the parameters. We first store the parameters in `LP` and calculate the length of the list of parameters:

```
#@title Exploring the Parameters
LP=list(model.parameters())
lp=len(LP)
print(lp)
```

The output shows that there are approximately `108` matrices and vectors, which might vary from one transformer model to another:

```
108
```

Now, let's display the `108` matrices and vectors in the tensors that contain them:

```
for p in range(0,lp):
  print(LP[p])
```

The output displays all the parameters as shown in the following excerpt of the output:

```
Parameter containing:
tensor([[-0.0175, -0.0210, -0.0334,  ...,  0.0054, -0.0113,  0.0183],
        [ 0.0020, -0.0354, -0.0221,  ...,  0.0220, -0.0060, -0.0032],
        [ 0.0001, -0.0002,  0.0036,  ..., -0.0265, -0.0057, -0.0352],
        ...,
        [-0.0125, -0.0418,  0.0190,  ..., -0.0069,  0.0175, -0.0308],
        [ 0.0072, -0.0131,  0.0069,  ...,  0.0002, -0.0234,  0.0042],
        [ 0.0008,  0.0281,  0.0168,  ..., -0.0113, -0.0075,  0.0014]],
       requires_grad=True)
```

Take a few minutes to peek inside the parameters to understand how transformers are built.

The number of parameters is calculated by taking all parameters in the model and adding them up; for example:

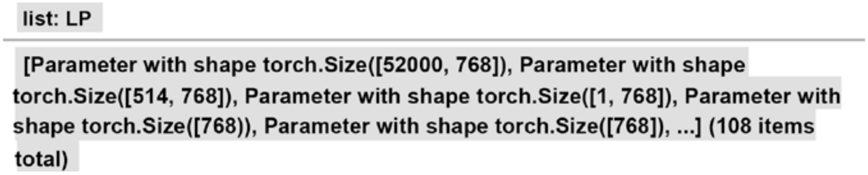
- The vocabulary (52,000) x dimensions (768)
- The size of the vectors is `1 x 768`
- The many other dimensions found

You will note that $d_{model} = 768$. There are 12 heads in the model. The dimension of d_k for each head will thus be $d_k = \frac{d_{model}}{12} = 64$.

This shows, once again, the optimized LEGO® concept of the building blocks of a transformer.

We will now see how the number of parameters of a model is calculated and how the figure 84,095,008 is reached.

If we hover over **LP** in the notebook, we will see some of the shapes of the torch tensors:



list: LP

[Parameter with shape torch.Size([52000, 768]), Parameter with shape torch.Size([514, 768]), Parameter with shape torch.Size([1, 768]), Parameter with shape torch.Size([768)), Parameter with shape torch.Size([768]), ...] (108 items total)

Figure 4.4: LP

Note that the numbers might vary depending on the version of the transformers module you use.

We will take this further and count the number of parameters of each tensor. First, the program initializes a parameter counter named `np` (number of parameters) and goes through the `lp` (`108`) number of elements in the list of parameters:

```
#@title Counting the parameters
np=0
for p in range(0,lp):#number of tensors
```

The parameters are matrices and vectors of different sizes; for example:

- 768 x 768
- 768 x 1
- 768

We can see that some parameters are two-dimensional, and some are one-dimensional.

An easy way to see if a parameter p in the list LP[p] has two dimensions or not is by doing the following:

```
  PL2=True
  try:
    L2=len(LP[p][0]) #check if 2D
  except:
    L2=1             #not 2D but 1D
    PL2=False
```

If the parameter has two dimensions, its second dimension will be L2>0 and PL2=True (2 dimensions=True). If the parameter has only one dimension, its second dimension will be L2=1 and PL2=False (2 dimensions=False).

L1 is the size of the first dimension of the parameter. L3 is the size of the parameters defined by:

```
L1=len(LP[p])
L3=L1*L2
```

We can now add the parameters up at each step of the loop:

```
np+=L3             # number of parameters per tensor
```

We will obtain the sum of the parameters, but we also want to see exactly how the number of parameters of a transformer model is calculated:

```
  if PL2==True:
    print(p,L1,L2,L3)  # displaying the sizes of the parameters
  if PL2==False:
    print(p,L1,L3)  # displaying the sizes of the parameters
print(np)              # total number of parameters
```

Note that if a parameter only has one dimension, PL2=False, then we only display the first dimension.

The output is the list of how the number of parameters was calculated for all the tensors in the model, as shown in the following excerpt:

```
0 52000 768 39936000
1 514 768 394752
2 1 768 768
```

```
3 768 768
4 768 768
5 768 768 589824
6 768 768
7 768 768 589824
8 768 768
9 768 768 589824
10 768 768
```

The total number of parameters of the RoBERTa model is displayed at the end of the list:

```
84,095,008
```

The number of parameters might vary with the version of the libraries used.

We now know precisely what the number of parameters represents in a transformer model. Take a few minutes to go back and look at the output of the configuration, the content of the parameters, and the size of the parameters. At this point, you will have a precise mental representation of the building blocks of the model.

The program now builds the dataset.

Step 10: Building the dataset

The program will now load the dataset line by line to generate samples for batch training with `block_size=128` limiting the length of an example:

```
#@title Step 10: Building the Dataset
%%time
from transformers import LineByLineTextDataset
dataset = LineByLineTextDataset(
    tokenizer=tokenizer,
    file_path="./kant.txt",
    block_size=128,
)
```

The output shows that Hugging Face has invested a considerable amount of resources in optimizing the time it takes to process data:

```
CPU times: user 8.48 s, sys: 234 ms, total: 8.71 s
Wall time: 3.88 s
```

The wall time, the actual time the processors were active, is optimized.

The program will now define a data collator to create an object for backpropagation.

Step 11: Defining a data collator

We need to run a data collator before initializing the trainer. A data collator will take samples from the dataset and collate them into batches. The results are dictionary-like objects.

We are preparing a batched sample process for MLM by setting `mlm=True`.

We also set the number of masked tokens to train `mlm_probability=0.15`. This will determine the percentage of tokens masked during the pretraining process.

We now initialize `data_collator` with our tokenizer, MLM activated, and the proportion of masked tokens set to `0.15`:

```
#@title Step 11: Defining a Data Collator
from transformers import DataCollatorForLanguageModeling
data_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer, mlm=True, mlm_probability=0.15
)
```

We are now ready to initialize the trainer.

Step 12: Initializing the trainer

The previous steps have prepared the information required to initialize the trainer. The dataset has been tokenized and loaded. Our model is built. The data collator has been created.

The program can now initialize the trainer. For educational purposes, the program trains the model quickly. The number of epochs is limited to one. The GPU comes in handy since we can share the batches and multi-process the training tasks:

```
#@title Step 12: Initializing the Trainer
from transformers import Trainer, TrainingArguments
training_args = TrainingArguments(
    output_dir="./KantaiBERT",
    overwrite_output_dir=True,
    num_train_epochs=1,
    per_device_train_batch_size=64,
    save_steps=10_000,
    save_total_limit=2,
)
trainer = Trainer(
    model=model,
    args=training_args,
    data_collator=data_collator,
    train_dataset=dataset,
)
```

The model is now ready for training.

Step 13: Pretraining the model

Everything is ready. The trainer is launched with one line of code:

```
#@title Step 13: Pre-training the Model
%%time
trainer.train()
```

The output displays the training process in real time, showing `loss`, `learning rate`, `epoch`, and the steps:

```
Epoch: 100%
1/1 [17:59<00:00, 1079.91s/it]
Iteration: 100%
2672/2672 [17:59<00:00, 2.47it/s]
{"loss": 5.6455852394104005, "learning_rate": 4.06437125748503e-05,
"epoch": 0.18712574850299402, "step": 500}
{"loss": 4.940259679794312, "learning_rate": 3.12874251497006e-05,
"epoch": 0.37425149700598803, "step": 1000}
{"loss": 4.639936000347137, "learning_rate": 2.1931137724550898e-05,
"epoch": 0.561377245508982, "step": 1500}
{"loss": 4.361462069988251, "learning_rate": 1.2574850299401197e-05,
"epoch": 0.7485029940119761, "step": 2000}
{"loss": 4.228510192394257, "learning_rate": 3.218562874251497e-06,
"epoch": 0.9356287425149701, "step": 2500}
CPU times: user 11min 36s, sys: 6min 25s, total: 18min 2s
Wall time: 17min 59s
TrainOutput(global_step=2672, training_loss=4.7226536670130885)
```

The model has been trained. It's time to save our work.

Step 14: Saving the final model (+tokenizer + config) to disk

We will now save the model and configuration:

```
#@title Step 14: Saving the Final Model(+tokenizer + config) to disk
trainer.save_model("./KantaiBERT")
```

Click on **Refresh** in the file manager, and the files should appear:

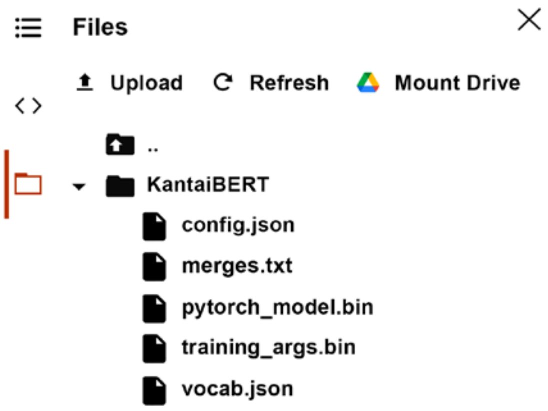


Figure 4.5: Colab file manager

`config.json`, `pytorh_model.bin`, and `training_args.bin` should now appear in the file manager.

`merges.txt` and `vocab.json` contain the pretrained tokenization of the dataset.

We have built a model from scratch. Let's import the pipeline to perform a language modeling task with our pretrained model and tokenizer.

Step 15: Language modeling with FillMaskPipeline

We will now import a language modeling `fill-mask` task. We will use our trained model and trained tokenizer to perform MLM:

```
#@title Step 15: Language Modeling with the FillMaskPipeline
from transformers import pipeline
fill_mask = pipeline(
    "fill-mask",
    model="./KantaiBERT",
    tokenizer="./KantaiBERT"
)
```

We can now ask our model to think like Immanuel Kant:

```
fill_mask("Human thinking involves human <mask>.")
```

The output will likely change after each run because we are pretraining the model from scratch with a limited amount of data. However, the output obtained in this run is interesting because it introduces conceptual language modeling:

```
[{'score': 0.022831793874502182,
  'sequence': '<s> Human thinking involves human reason.</s>',
  'token': 393},
 {'score': 0.011635891161859035,
  'sequence': '<s> Human thinking involves human object.</s>',
  'token': 394},
 {'score': 0.010641072876751423,
  'sequence': '<s> Human thinking involves human priori.</s>',
  'token': 575},
 {'score': 0.009517930448055267,
  'sequence': '<s> Human thinking involves human conception.</s>',
  'token': 418},
 {'score': 0.00923212617635727,
  'sequence': '<s> Human thinking involves human experience.</s>',
  'token': 531}]
```

The predictions might vary each run and each time Hugging Face updates its models.

However, the following output comes out often:

```
Human thinking involves human reason
```

The goal here was to see how to train a transformer model. We can see that interesting human-like predictions can be made.

These results are experimental and subject to variations during the training process. They will change each time we train the model again.

The model would require much more data from other *Age of Enlightenment* thinkers.

However, the goal of this model is to show that we can create datasets to train a transformer for a specific type of complex language modeling task.

Thanks to transformers, we are only at the beginning of a new era of AI!

Next steps

You have trained a transformer from scratch. Take some time to imagine what you could do in your personal or corporate environment. You could create a dataset for a specific task and train it from scratch. Use your areas of interest or company projects to experiment with the fascinating world of transformer construction kits!

Once you have made a model you like, you can share it with the Hugging Face community. Your model will appear on the Hugging Face models page: `https://huggingface.co/models`

You can upload your model in a few steps using the instructions described on this page: `https://huggingface.co/transformers/model_sharing.html`

You can also download models the Hugging Face community has shared to get new ideas for your personal and professional projects.

Summary

In this chapter, we built `KantaiBERT`, a RoBERTa-like model transformer, from scratch using the building blocks provided by Hugging Face.

We first started by loading a customized dataset on a specific topic related to the works of Immanuel Kant. You can load an existing dataset or create your own, depending on your goals. We saw that using a customized dataset provides insights into the way a transformer model thinks. However, this experimental approach has its limits. It would take a much larger dataset to train a model beyond educational purposes.

The KantaiBERT project was used to train a tokenizer on the `kant.txt` dataset. The trained `merges.txt` and `vocab.json` files were saved. A tokenizer was recreated with our pretrained files. KantaiBERT built the customized dataset and defined a data collator to process the training batches for backpropagation. The trainer was initialized, and we explored the parameters of the RoBERTa model in detail. The model was trained and saved.

Finally, the saved model was loaded for a downstream language modeling task. The goal was to fill the mask using Immanuel Kant’s logic.

The door is now wide open for you to experiment on existing or customized datasets to see what results you get. You can share your model with the Hugging Face community. Transformers are data-driven. You can use this to your advantage to discover new ways of using transformers.

You are now ready to learn how to run ready-to-use transformer engines with APIs that require no pretraining or fine-tuning. *Chapter 7*, *The Rise of Suprahuman Transformers with GPT-3 Engines*, will take you into the future of AI. And with the knowledge of this chapter and the past chapters, you will be ready!

In the next chapter, *Downstream NLP Tasks with Transformers*, we will continue our preparation to implement transformers.

Questions

1. RoBERTa uses a byte-level byte-pair encoding tokenizer. (True/False)
2. A trained Hugging Face tokenizer produces `merges.txt` and `vocab.json`. (True/False)
3. RoBERTa does not use token-type IDs. (True/False)
4. DistilBERT has 6 layers and 12 heads. (True/False)
5. A transformer model with 80 million parameters is enormous. (True/False)
6. We cannot train a tokenizer. (True/False)
7. A BERT-like model has 6 decoder layers. (True/False)
8. **Masked Language Modeling** (**MLM**) predicts a word contained in a mask token in a sentence. (True/False)
9. A BERT-like model has no self-attention sublayers. (True/False)
10. Data collators are helpful for backpropagation. (True/False)

References

- *Yinhan Liu*, *Myle Ott*, *Naman Goyal*, *Jingfei Du*, *Mandar Joshi*, *Danqi Chen*, *Omer Levy*, *Mike Lewis*, *Luke Zettlemoyer*, and *Veselin Stoyano*, 2019, *RoBERTa: A Robustly Optimized BERT Pretraining Approach*: `https://arxiv.org/abs/1907.11692`
- Hugging Face Tokenizer documentation: `https://huggingface.co/transformers/main_classes/tokenizer.html?highlight=tokenizer`
- The Hugging Face reference notebook: `https://colab.research.google.com/github/huggingface/blog/blob/master/notebooks/01_how_to_train.ipynb`

- The Hugging Face reference blog: `https://colab.research.google.com/github/huggingface/blog/blob/master/notebooks/01_how_to_train.ipynb`
- More on BERT: `https://huggingface.co/transformers/model_doc/bert.html`
- More DistilBERT: `https://arxiv.org/pdf/1910.01108.pdf`
- More on RoBERTa: `https://huggingface.co/transformers/model_doc/roberta.html`
- Even more on DistilBERT: `https://huggingface.co/transformers/model_doc/distilbert.html`

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the authors:

`https://www.packt.link/Transformers`

5

Downstream NLP Tasks with Transformers

Transformers reveal their full potential when we unleash pretrained models and watch them perform downstream **Natural Language Understanding** (**NLU**) tasks. It takes a lot of time and effort to pretrain and fine-tune a transformer model, but the effort is worthwhile when we see a multi-million parameter transformer model in action on a range of NLU tasks.

We will begin this chapter with the quest of outperforming the human baseline. The human baseline represents the performance of humans on an NLU task. Humans learn transduction at an early age and quickly develop inductive thinking. We humans perceive the world directly with our senses. Machine intelligence relies entirely on our perceptions transcribed into words to make sense of our language.

We will then see how to measure the performance of transformers. Measuring **Natural Language Processing** (**NLP**) tasks remains a straightforward approach involving accuracy scores in various forms based on true and false results. These results are obtained through benchmark tasks and datasets. SuperGLUE, for example, is a wonderful example of how Google DeepMind, Facebook AI, the University of New York, the University of Washington, and others worked together to set high standards to measure NLP performances.

Finally, we will explore several downstream tasks, such as the **Standard Sentiment TreeBank** (**SST-2**), linguistic acceptability, and Winograd schemas.

Transformers are rapidly taking NLP to the next level by outperforming other models on well-designed benchmark tasks. Alternative transformer architectures will continue to emerge and evolve.

This chapter covers the following topics:

- Machine versus human intelligence for transduction and induction
- The NLP transduction and induction process
- Measuring transformer performance versus Human Baselines
- Measurement methods (Accuracy, F1-score, and MCC)
- Benchmark tasks and datasets
- SuperGLUE downstream tasks
- Linguistic acceptability with CoLA
- Sentiment analysis with SST-2
- Winograd schemas

Let's start by understanding how humans and machines represent language.

Transduction and the inductive inheritance of transformers

The emergence of **Automated Machine Learning** (**AutoML**), meaning APIs in automated cloud AI platforms, has deeply changed the job description of every AI specialist. Google Vertex, for example, boasts a reduction of 80% of the development required to implement ML. This suggests that anybody can implement ML with ready-to-use systems. Does that mean an 80% reduction of the workforce of developers? I don't think so. I see an Industry 4.0 AI specialist assemble AI with added value to a project.

> Industry 4.0. NLP AI specialists invest less in source code and more in knowledge to become the AI guru of a team.

Transformers possess the unique ability to apply their knowledge to tasks they did not learn. A BERT transformer, for example, acquires language through sequence-to-sequence and masked language modeling. The BERT transformer can then be fine-tuned to perform downstream tasks that it did not learn from scratch.

In this section, we will do a mind experiment. We will use the graph of a transformer to represent how humans and machines make sense of information using language. Machines make sense of information in a different way from humans but reach very efficient results.

Figure 5.1, a mind experiment designed with transformer architecture layers and sublayers, shows the deceptive similarity between humans and machines. Let's study the learning process of transformer models to understand downstream tasks:

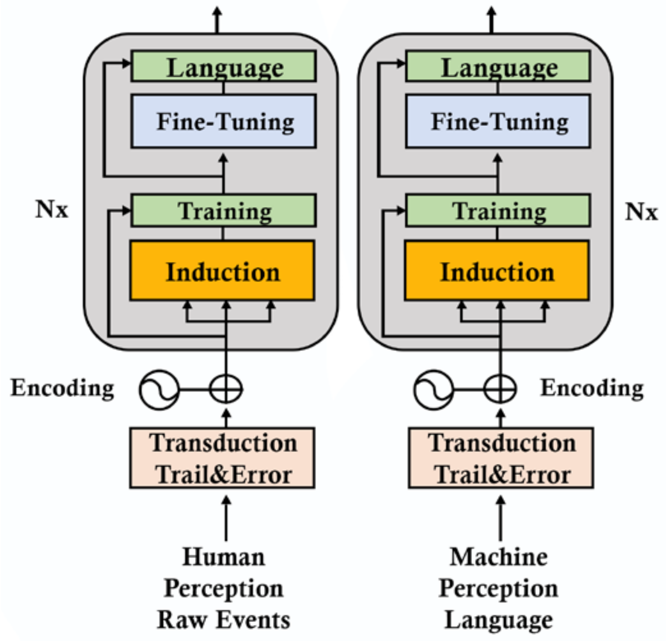


Figure 5.1: Human and ML methods

For our example, `N=2`. This conceptual representation has two layers. The two layers show that humans build on accumulated knowledge from generation to generation. Machines only process what we give them. Machines use our outputs as inputs.

The human intelligence stack

On the left side of *Figure 5.1*, we can see that the input for humans is the perception of raw events for layer 0, and the output is language. We first perceive events with our senses as children. Gradually, the output becomes burbling language and then structured language.

For humans, *transduction* goes through a trial-and-error process. Transduction means that we take structures we perceive and represent them with patterns, for example. We make representations of the world that we apply to our inductive thinking. Our inductive thinking relies on the quality of our transductions.

For example, as children, we were often forced to take a nap early in the afternoon. Famous child psychologist Piaget found that this could lead to some children saying, for example, "I haven't taken a nap, so it's not the afternoon." The child sees two events, creates a link between them with transduction, and then makes an inference to generalize and make an induction.

At first, humans notice these patterns through transduction and generalize them through *inductions*. We are trained by trial and error to understand that many events are related:

Trained_related events = {sunrise – light, sunset – dark, dark clouds – rain, blue sky – running, food – good, fire – warm, snow – cold}

Over time, we are trained to understand millions of related events. New generations of humans did not have to start from scratch. They were only *fine-tuned for many tasks by previous generations*. They were taught that "fire burns you," for example. From then on, a child knew that this knowledge could be fine-tuned to any form of "fire": candles, wildfires, volcanoes, and every instance of "fire."

Finally, humans transcribed everything they knew, imagined, or predicted into written *language*. The output of layer 0 was born.

For humans, the input of the next layer, layer 1, is the vast amount of trained and fine-tuned knowledge. On top of that, humans perceive massive amounts of events that then go through the transduction, induction, training, and fine-tuning of sublayers, along with previous transcribed knowledge.

Many of these events stem from odors, emotions, situations, experiences, and everything that makes a human unique. Machines do not have access to this individual identity. Humans have a personal perception of a word in a homogeneous way specific for each individual.

A machine takes what we give it through masses of heterogeneous unfiltered impersonal data. The goal of a machine is to perform an impersonal, efficient task. The goal of a human being is personal wellbeing.

Our infinite approach loop goes from layer 0 to layer 1 and back to layer 0 with more raw and processed information.

The result is fascinating! We do not need to learn (train on) our native language from scratch to acquire summarization abilities. We use our pretrained knowledge to adjust (fine-tune) to summarization tasks.

Transformers go through the same process but in a different way.

The machine intelligence stack

Machines learn basic tasks like the ones described in this chapter, and then perform hundreds of tasks using the sequences they learned how to predict.

On the right side of *Figure 5.1*, we can see that the input for machines is second-hand information in the form of language. *Our output is the only input machines have to analyze language*.

At this point in human and machine history, computer vision identifies images but does not contain the grammatical structure of language. Speech recognition converts sound into words, which brings us back to written language. Music pattern recognition cannot lead to objective concepts expressed in words.

Machines start with a handicap. We impose an artificial disadvantage on them. Machines must rely on our random quality language outputs to:

- Perform transductions connecting all the tokens (subwords) that occur together in language sequences
- Build inductions from these transductions
- Train those inductions based on tokens to produce patterns of tokens

Let's stop at this point and peek into the process of the attention sublayer, which works hard to produce valid inductions:

- The transformer model excluded the former recurrence-based learning operations and used self-attention to heighten the vision of the model
- Attention sublayers have an advantage over humans at this point: they can process millions of examples for their inductive thinking operations
- Like us, they find patterns in sequences through transduction and induction
- They memorize these patterns using parameters that are stored with their model

They have acquired language understanding by using their abilities: substantial data volumes, excellent NLP transformer algorithms, and computer power. *Thanks to their deep understanding of language, they are ready to run hundreds of tasks they were not trained for*.

Transformers, like humans, acquire language understanding through a limited number of tasks. Like us, they detect connections through transduction and then generalize them through inductive operations.

When the transformer model reaches the fine-tuning sub-layer of machine intelligence, it reacts like us. It does not start training from scratch to perform a new task. Like us, it considers it as a downstream task that only requires fine-tuning. If it needs to learn how to answer a question, it does not start learning a language from scratch. A transformer model just fine-tunes its parameters like us.

In this section, we saw that transformer models struggle to learn in the way we do. They start with a handicap from the moment they rely on our perceptions transcribed into language. However, they have access to infinitely more data than we do with massive computing power.

Let's now see how to measure transformer performances versus Human Baselines.

Transformer performances versus Human Baselines

Transformers, like humans, can be fine-tuned to perform downstream tasks by inheriting the properties of a pretrained model. The pretrained model provides its architecture and language representations through its parameters.

A pretrained model trains on key tasks to acquire a general knowledge of the language. A fine-tuned model trains on downstream tasks. Not every transformer model uses the same tasks for pretraining. Potentially, all tasks can be pretrained or fine-tuned.

Every NLP model needs to be evaluated with a standard method.

This section will first go through some of the key measurement methods. Then, we will go through some of the main benchmark tasks and datasets.

Let's start by going through some of the key metric methods.

Evaluating models with metrics

It is impossible to compare one transformer model to another transformer model (or any other NLP model) without a universal measurement system that uses metrics.

In this section, we will analyze three measurement scoring methods that are used by GLUE and SuperGLUE.

Accuracy score

The accuracy score, in whatever variant you use, is a practical evaluation. The score function calculates a straightforward true or false value for each result. Either the model's outputs, $\hat{y}$, match the correct predictions, y, for a given subset, *samples*$_i$, of a set of samples or not. The basic function is:

$$Accuracy(y, \hat{y}) = \frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} 1(\hat{y}_i = y_i)$$

We will obtain `1` if the result for the subset is correct and `0` if it is false.

Let's now examine the more flexible F1-score.

F1-score

The F1-score introduces a more flexible approach that can help when faced with datasets containing uneven class distributions.

The F1-score uses the weighted values of precision and recall. It is a weighted average of precision and recall values:

F1score= 2 (precision * recall)/(precision + recall)*

In this equation, true (T) positives (p), false (F) positives (p), and false (F) negatives (n) are plugged into the precision (P) and recall (R) equations:

$$P = \frac{T_p}{T_p + F_p}$$

$$R = \frac{T_p}{T_p + F_n}$$

The F1-score can thus be viewed as the harmonic mean (reciprocal of the arithmetic mean) of precision (P) and recall (R):

$$F1\ score = 2 \times \frac{P \times R}{P + R}$$

Let's now review the MCC approach.

Matthews Correlation Coefficient (MCC)

MCC was described and implemented in the *Evaluating using Matthews Correlation Coefficient* section in *Chapter 3*, *Fine-Tuning BERT Models*. MCC computes a measurement with true positives (T_P), true negatives (T_N), false positives (F_P), and false negatives (F_N).

The MCC can be summarized by the following equation:

$$\frac{TP\ x\ TN - FP\ x\ FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

MCC provides an excellent metric for binary classification models, even if the sizes of the classes are different.

We now have a good idea of how to measure a given transformer model's results and compare them to other transformer models or NLP models.

With measurement scoring methods in mind, let's now look into benchmark tasks and datasets.

Benchmark tasks and datasets

Three prerequisites are required to prove that transformers have reached state-of-the-art performance levels:

- A model
- A dataset-driven task
- A metric as described in the *Evaluating models with metrics* section of this chapter

We will begin by exploring the SuperGLUE benchmark to illustrate the evaluation process of a transformer model.

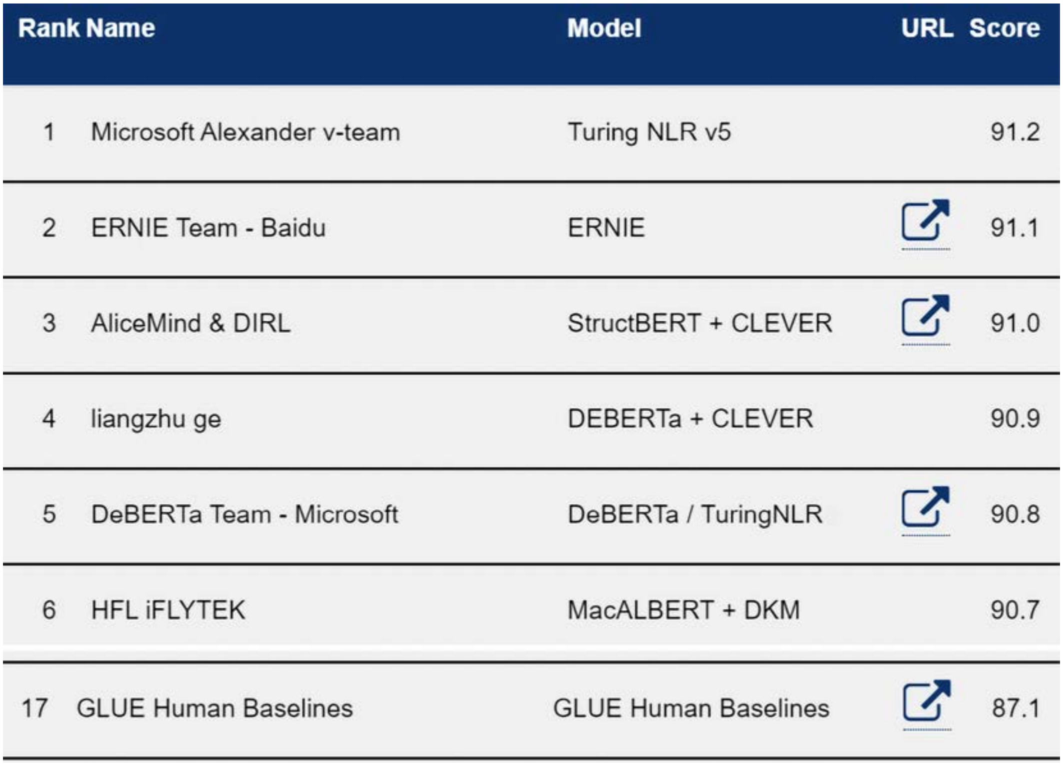
From GLUE to SuperGLUE

The SuperGLUE benchmark was designed and made public by *Wang* et al. (2019). *Wang* et al. (2019) first designed the **General Language Understanding Evaluation** (**GLUE**) benchmark.

The motivation of the GLUE benchmark was to show that to be useful, NLU has to be applicable to a wide range of tasks. Relatively small GLUE datasets were designed to encourage an NLU model to solve a set of tasks.

However, the performance of NLU models, boosted by the arrival of transformers, began to exceed the level of the average human, as we can see in the GLUE leaderboard (December 2021). The GLUE leaderboard, available at `https://gluebenchmark.com/leaderboard`, shows a remarkable display of NLU talent, retaining some of the former RNN/CNN ideas while mainly focusing on the ground-breaking transformer models.

The following excerpt of the leaderboard shows the top leaders and the position of GLUE's Human Baselines:



Rank	Name	Model	URL	Score
1	Microsoft Alexander v-team	Turing NLR v5		91.2
2	ERNIE Team - Baidu	ERNIE	↗	91.1
3	AliceMind & DIRL	StructBERT + CLEVER	↗	91.0
4	liangzhu ge	DEBERTa + CLEVER		90.9
5	DeBERTa Team - Microsoft	DeBERTa / TuringNLR	↗	90.8
6	HFL iFLYTEK	MacALBERT + DKM		90.7
17	GLUE Human Baselines	GLUE Human Baselines	↗	87.1

Figure 5.2: GLUE Leaderboard – December 2021

New models and the Human Baselines ranking will constantly change. These rankings just give an idea of how far classical NLP and transformers have taken us!

We first notice the GLUE Human Baselines are not in a top position, which shows that NLU models have surpassed non-expert humans on GLUE tasks. Human Baselines represent what we humans can achieve. AI can now outperform humans. In December 2021, the Human Baselines are only in position 17. This is a problem. Without a standard to beat, it is challenging to fish around for benchmark datasets to improve our models blindly.

We also notice that transformer models have taken the lead.

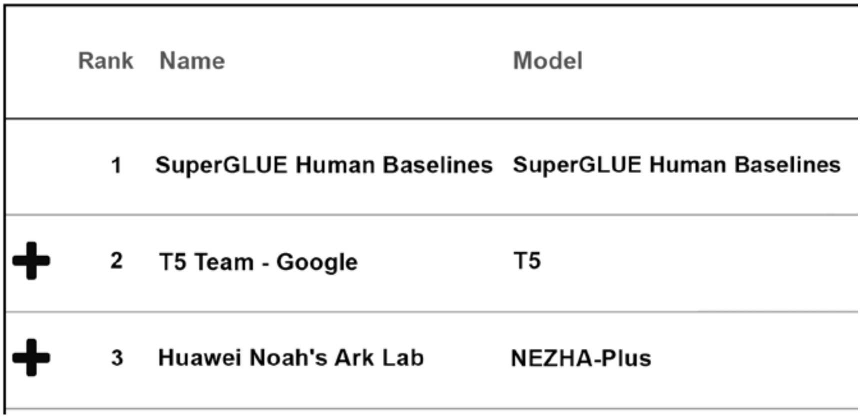
I like to think of GLUE and SuperGLUE as the point when words go from chaos to order with language understanding. For me, understanding is the glue that makes words fit together and become a language.

The GLUE leaderboard will continuously evolve as NLU progresses. However, *Wang* et al. (2019) introduced SuperGLUE to set a higher standard for Human Baselines.

Introducing higher Human Baselines standards

Wang et al. (2019) recognized the limits of GLUE. They designed SuperGLUE for more difficult NLU tasks.

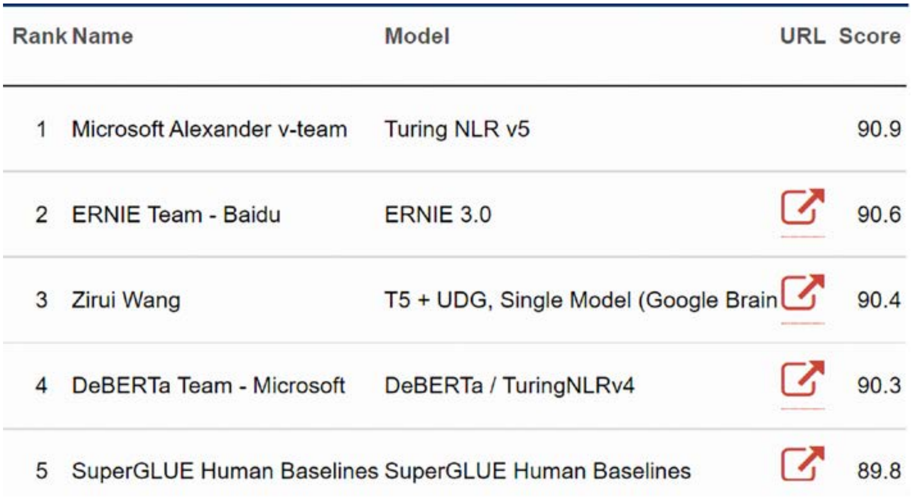
SuperGLUE immediately re-established Human Baselines at rank #1 (December 2020), as shown in the following excerpt of the leaderboard, `https://super.gluebenchmark.com/leaderboard`:



	Rank	Name	Model
	1	SuperGLUE Human Baselines	SuperGLUE Human Baselines
+	2	T5 Team - Google	T5
+	3	Huawei Noah's Ark Lab	NEZHA-Plus

Figure 5.3: SuperGLUE Leaderboard 2.0 – December 2020

However, the SuperGLUE leaderboard evolved as we produced better NLU models. In 2021, transformers had already surpassed Human Baselines. In December 2021, Human Baselines have gone down to rank #5, as shown in *Figure 5.4*:



Rank	Name	Model	URL	Score
1	Microsoft Alexander v-team	Turing NLR v5		90.9
2	ERNIE Team - Baidu	ERNIE 3.0	↗	90.6
3	Zirui Wang	T5 + UDG, Single Model (Google Brain	↗	90.4
4	DeBERTa Team - Microsoft	DeBERTa / TuringNLRv4	↗	90.3
5	SuperGLUE Human Baselines	SuperGLUE Human Baselines	↗	89.8

Figure 5.4: SuperGLUE Leaderboard 2.0 – December 2021

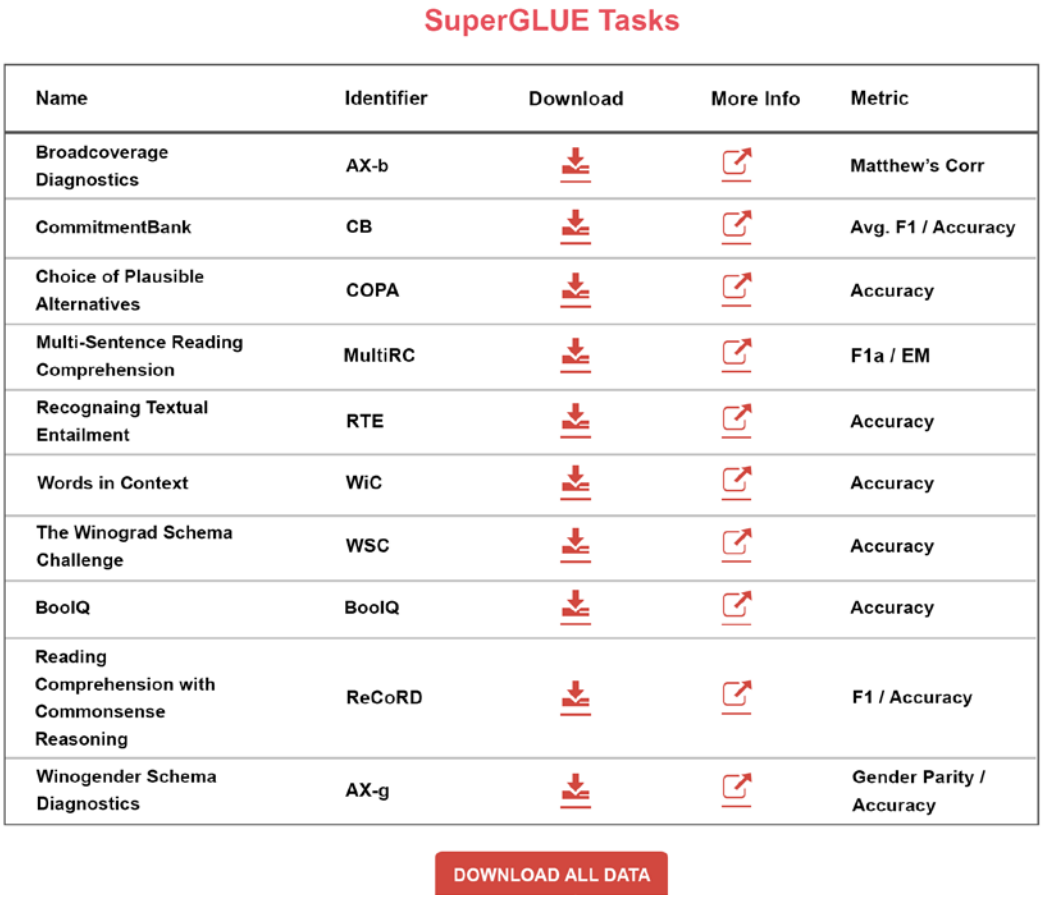
AI algorithm rankings will constantly change as new innovative models arrive. These rankings just give an idea of how hard the battle for NLP supremacy is being fought!

Let's now see how the evaluation process works.

The SuperGLUE evaluation process

Wang et al. (2019) selected eight tasks for their SuperGLUE benchmark. The selection criteria for these tasks were stricter than for GLUE. For example, the tasks had to not only understand texts but also to reason. The level of reasoning is not that of a top human expert. However, the level of performance is sufficient to replace many human tasks.

The eight SuperGLUE tasks are presented in a ready-to-use list:



SuperGLUE Tasks

Name	Identifier	Download	More Info	Metric
Broadcoverage Diagnostics	AX-b	⤓	↗	Matthew's Corr
CommitmentBank	CB	⤓	↗	Avg. F1 / Accuracy
Choice of Plausible Alternatives	COPA	⤓	↗	Accuracy
Multi-Sentence Reading Comprehension	MultiRC	⤓	↗	F1a / EM
Recognaing Textual Entailment	RTE	⤓	↗	Accuracy
Words in Context	WiC	⤓	↗	Accuracy
The Winograd Schema Challenge	WSC	⤓	↗	Accuracy
BoolQ	BoolQ	⤓	↗	Accuracy
Reading Comprehension with Commonsense Reasoning	ReCoRD	⤓	↗	F1 / Accuracy
Winogender Schema Diagnostics	AX-g	⤓	↗	Gender Parity / Accuracy

Figure 5.5: SuperGLUE tasks

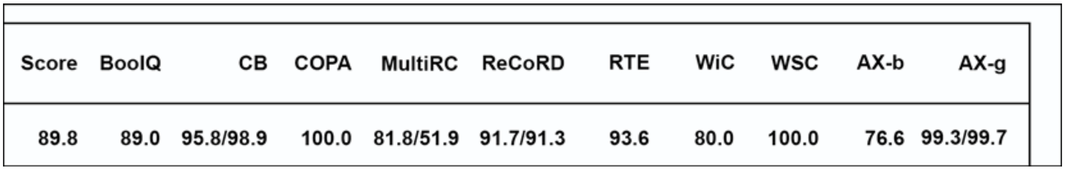
The task list is interactive: `https://super.gluebenchmark.com/tasks`.

Each task contains links to the required information to perform that task:

- **Name** is the name of the downstream task of a fine-tuned, pretrained model
- **Identifier** is the abbreviation or short version of the name
- **Download** is the download link to the datasets

- **More Info** offers greater detail through a link to the paper or website of the team that designed the dataset-driven task(s)
- **Metric** is the measurement score used to evaluate the model

SuperGLUE provides the task instructions, the software, the datasets, and papers or websites describing the problem to be solved. Once a team runs the benchmark tasks and reaches the leaderboard, the results are displayed:



Score	BoolQ	CB	COPA	MultiRC	ReCoRD	RTE	WiC	WSC	AX-b	AX-g
89.8	89.0	95.8/98.9	100.0	81.8/51.9	91.7/91.3	93.6	80.0	100.0	76.6	99.3/99.7

Figure 5.6: SuperGLUE task scores

SuperGLUE displays the overall score and the score for each task.

For example, let's take the instructions *Wang* et al. (2019) provided for the **Choice of Plausible Answers (COPA)** task in *Table 6* of their paper.

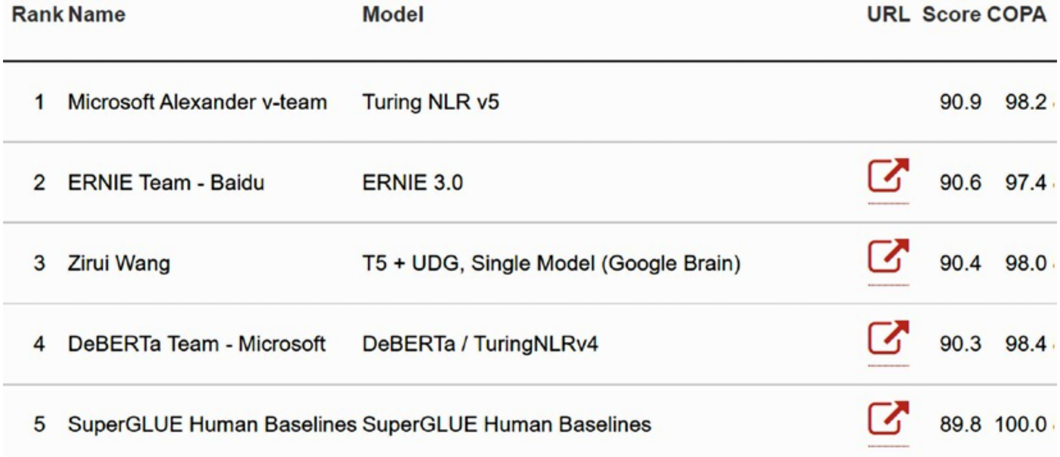
The first step is to read the remarkable paper written by *Roemmele* et al. (2011). In a nutshell, the goal is for the NLU model to demonstrate its machine thinking (not human thinking, of course) potential. In our case, the transformer must choose the most plausible answer to a question. The dataset provides a premise, and the transformer model must find the most plausible answer.

For example:

```
Premise: I knocked on my neighbor's door.

What happened as a result?

Alternative 1: My neighbor invited me in.

Alternative 2: My neighbor left his house.
```

This question requires a second or two for a human to answer, which shows that it requires some commonsense machine thinking. `COPA.zip`, a ready-to-use dataset, can be downloaded directly from the SuperGLUE task page. The metric provided makes the process equal and reliable for all participants in the benchmark race.

The examples might seem difficult. However, transformers are nearing COPA Human Baselines (*Figure 5.7*), which is only at rank #5 if we take all of the tasks into account:



Rank	Name	Model	URL	Score	COPA
1	Microsoft Alexander v-team	Turing NLR v5		90.9	98.2
2	ERNIE Team - Baidu	ERNIE 3.0	↗	90.6	97.4
3	Zirui Wang	T5 + UDG, Single Model (Google Brain)	↗	90.4	98.0
4	DeBERTa Team - Microsoft	DeBERTa / TuringNLRv4	↗	90.3	98.4
5	SuperGLUE Human Baselines	SuperGLUE Human Baselines	↗	89.8	100.0

Figure 5.7: SuperGLUE results for COPA

As incredible as it seems, transformers climbed the leaderboard ladder in a very short time! And this is just the beginning. New ideas are emerging nearly every month!

We have introduced COPA. Let's define the seven other SuperGLUE benchmark tasks.

Defining the SuperGLUE benchmark tasks

A task can be a pretraining task to generate a trained model. That same task can be a downstream task for another model that will fine-tune it. However, the goal of SuperGLUE is to show that a given NLU model can perform multiple downstream tasks with fine-tuning. Multi-task models are the ones that prove the thinking power of transformers.

The power of any transformer resides in its ability to perform multiple tasks using a pretrained model and then applying it to fine-tuned downstream tasks. The original Transformer model and its variants now lead in all the GLUE and SuperGLUE tasks. We will continue to focus on SuperGLUE downstream tasks for which Human Baselines is tough to beat.

In the previous section, we went through COPA. In this section, we will go through the seven other tasks defined by *Wang* et al. (2019) in *Table 2* of their paper.

Let's continue with a Boolean question task.

BoolQ

BoolQ is a Boolean yes-or-no answer task. The dataset, as defined on SuperGLUE, contains 15,942 naturally occurring examples. A raw sample of line #3 of the `train.jsonl` dataset contains a passage, a question, and the answer (`true`):

```
{"question": "is windows movie maker part of windows essentials"
"passage": "Windows Movie Maker -- Windows Movie Maker (formerly known as
Windows Live Movie Maker in Windows 7) is a discontinued video editing
software by Microsoft. It is a part of Windows Essentials software suite
and offers the ability to create and edit videos as well as to publish
them on OneDrive, Facebook, Vimeo, YouTube, and Flickr.", "idx": 2,
"label": true}
```

The datasets provided may change in time, but the concepts remain the same.

Now, let's examine CB, a task that requires both humans and machines to focus.

Commitment Bank (CB)

Commitment Bank (**CB**) is a difficult *entailment* task. We are asking the transformer model to read a *premise*, then examine a *hypothesis* built on the premise. For example, the hypothesis will confirm the premise or contradict it. Then the transformer model must *label* the hypothesis as *neutral*, an *entailment*, or a *contradiction* of the premise, for example.

The dataset contains discourses, which are natural discourses.

The following sample #77, taken from the `train.jsonl` training dataset, shows how difficult the CB task is:

```
{"premise": "The Susweca. It means ''dragonfly'' in Sioux, you know. Did I
ever tell you that's where Paul and I met?"
"hypothesis": "Susweca is where she and Paul met,"
"label": "entailment", "idx": 77}
```

We will now have a look at the multi-sentence problem.

Multi-Sentence Reading Comprehension (MultiRC)

Multi-Sentence Reading Comprehension (**MultiRC**) asks the model to read a text and choose from several possible choices. The task is difficult for both humans and machines. The model is presented with a *text*, several *questions*, and possible *answers* to each question with a `0` (false) or `1` (true) *label*.

Let's take the second sample in `train.jsonl`:

```
"Text": "text": "The rally took place on October 17, the shooting on
February 29. Again, standard filmmaking techniques are interpreted as
smooth distortion: \"Moore works by depriving you of context and guiding
your mind to fill the vacuum -- with completely false ideas. It is
brilliantly, if unethically, done.\" As noted above, the \"from my cold
dead hands\" part is simply Moore's way to introduce Heston. Did anyone
but Moore's critics view it as anything else? He certainly does not
\"attribute it to a speech where it was not uttered\" and, as noted above,
doing so twice would make no sense whatsoever if Moore was the mastermind
deceiver that his critics claim he is. Concerning the Georgetown Hoya
interview where Heston was asked about Rolland, you write: \"There is
no indication that [Heston] recognized Kayla Rolland's case.\" This is
naive to the extreme -- Heston would not be president of the NRA if he
was not kept up to date on the most prominent cases of gun violence.
Even if he did not respond to that part of the interview, he certainly
knew about the case at that point. Regarding the NRA website excerpt
about the case and the highlighting of the phrase \"48 hours after Kayla
Rolland is pronounced dead\": This is one valid criticism, but far from
the deliberate distortion you make it out to be; rather, it is an example
for how the facts can sometimes be easy to miss with Moore's fast pace
editing. The reason the sentence is highlighted is not to deceive the
viewer into believing that Heston hurried to Flint to immediately hold a
rally there (as will become quite obvious), but simply to highlight the
first mention of the name \"Kayla Rolland\" in the text, which is in this
paragraph. "
```

The sample contains four questions. To illustrate the task, we will just investigate two of them. The model must predict the correct labels. Notice how the information that the model is asked to obtain is distributed throughout the text:

```
"question": "When was Kayla Rolland shot?"
"answers":
[{"text": "February 17", "idx": 168, "label": 0},
{"text": "February 29", "idx": 169, "label": 1},
{"text": "October 29", "idx": 170, "label": 0},
{"text": "October 17", "idx": 171, "label": 0},
{"text": "February 17", "idx": 172, "label": 0}], "idx": 26},

{"question": "Who was president of the NRA on February 29?",
"answers": [{"text": "Charleton Heston", "idx": 173, "label": 1},
{"text": "Moore", "idx": 174, "label": 0},
{"text": "George Hoya", "idx": 175, "label": 0},
{"text": "Rolland", "idx": 176, "label": 0},
{"text": "Hoya", "idx": 177, "label": 0}, {"text": "Kayla", "idx": 178,
"label": 0}], "idx": 27},
```

At this point, one can only admire the performance of a single fine-tuned, pretrained model on these difficult downstream tasks.

Now, let's see the reading comprehension task.

Reading Comprehension with Commonsense Reasoning Dataset (ReCoRD)

Reading Comprehension with Commonsense Reasoning Dataset (**ReCoRD**) represents another challenging task. The dataset contains over 120,000 queries from more than 70,000 news articles. The transformer must use common-sense reasoning to solve this problem.

Let's examine a sample from `train.jsonl`:

```
"source": "Daily mail"
A passage contains the text and indications as to where the entities are
located.
A passage begins with the text:
"passage": {
    "text": "A Peruvian tribe once revered by the Inca's for their
fierce hunting skills and formidable warriors are clinging on to their
traditional existence in the coca growing valleys of South America,
sharing their land with drug traffickers, rebels and illegal loggers.
Ashaninka Indians are the largest group of indigenous people in the
mountainous nation's Amazon region, but their settlements are so sparse
that they now make up less than one per cent of Peru's 30 million
population. Ever since they battled rival tribes for territory and food
during native rule in the rainforests of South America, the Ashaninka
have rarely known peace.\n@highlight\nThe Ashaninka tribe once shared
the Amazon with the like of the Incas hundreds of years ago\n@highlight\
nThey have been forced to share their land after years of conflict forced
rebels and drug dealers into the forest\n@highlight\n. Despite settling
in valleys rich with valuable coca, they live a poor pre-industrial
existence",
```

The *entities* are indicated, as shown in the following excerpt:

```
  "entities": [{"start": 2,"end": 9}, …,"start": 711,"end": 715}]
```

Finally, the model must *answer* a *query* by finding the proper value for the *placeholder*:

```
{"query": "Innocence of youth: Many of the @placeholder's younger
generations have turned their backs on tribal life and moved to the
cities where living conditions are better",
"answers":[{"start":263,"end":271,"text":"Ashaninka"},{"start":601,
"end":609,"text":"Ashaninka"},{"start":651,"end":659,"text":"Ashaninka"}],
"idx":9}],"idx":3}
```

Once the transformer model has gone through this problem, it must now face an entailment task.

Recognizing Textual Entailment (RTE)

For **Recognizing Textual Entailment** (**RTE**), the transformer model must read the *premise*, examine a *hypothesis*, and predict the *label* of the *entailment hypothesis status*.

Let's examine sample #19 of the `train.jsonl` dataset:

```
{"premise": "U.S. crude settled $1.32 lower at $42.83 a barrel.",
"hypothesis": "Crude the light American lowered to the closing 1.32
dollars, to 42.83 dollars the barrel.", "label": "not_entailment", >"idx":
19}
```

RTE requires understanding and logic. Let's now see the Words in Context task.

Words in Context (WiC)

Words in Context (**WiC**) and the following Winograd task test a model's ability to process an ambiguous word. In WiC, the multi-task transformer will have to analyze two sentences and determine whether the target word has the same meaning in both sentences.

Let's examine the first sample of the `train.jsonl` dataset.

First, the target word is specified:

```
  "word": "place"
```

The model has to read two sentences containing the target word:

```
  "sentence1": "Do you want to come over to my place later?",
  "sentence2": "A political system with no place for the less prominent
groups."
```

`train.jsonl` specifies the sample index, the value of the label, and the position of the target word in `sentence1(start1, end1)` and `sentence2(start2, end2)`:

```
    "idx": 0,
    "label": false,
    "start1": 31,
    "start2": 27,
    "end1": 36,
    "end2": 32,
```

After this daunting task, the transformer model has to face the Winograd task.

The Winograd schema challenge (WSC)

The Winograd schema task is named after Terry Winograd. If a transformer is well trained, it should be able to solve disambiguation problems.

The dataset contains sentences that target slight differences in the gender of a pronoun.

This constitutes a coreference resolution problem, which is one of the most challenging tasks to perform. However, the transformer architecture, which allows self-attention, is ideal for this task.

Each sentence contains an *occupation*, a *participant*, and a *pronoun*. The problem to solve is to find whether the pronoun is *coreferent* with the occupation or the participant.

Let's examine a sample taken from `train.jsonl`.

First, the sample asks the model to read a *text*:

```
{"text": >"I poured water from the bottle into the cup until it was
full.",
The WSC ask the model to find the target pronoun token number 10 starting
at 0:
"target": {"span2_index": 10,
Then it asks the model to determine if "it" refers to "the cup" or not:
"span1_index": 7,
"span1_text": "the cup",
"span2_text": "it"},
For sample index #4, the label is true:
"idx": 4, "label": true}
```

We have gone through some of the main SuperGLUE tasks. There are many other tasks.

However, once you understand the architecture of transformers and the mechanism of the benchmark tasks, you will rapidly adapt to any model and benchmark.

Let's now run some downstream tasks.

Running downstream tasks

In this section, we will just jump into some transformer cars and drive them around a bit to see what they do. There are many models and tasks. We will run a few of them in this section. Once you understand the process of running a few tasks, you will quickly understand all of them. *After all, the human baseline for all these tasks is us!*

A downstream task is a fine-tuned transformer task that inherits the model and parameters from a pretrained transformer model.

A downstream task is thus the perspective of a pretrained model running fine-tuned tasks. That means, depending on the model, a task is downstream if it was not used to fully pretrain the model. In this section, we will consider all the tasks as downstream since we did not pretrain them.

Models will evolve, as will databases, benchmark methods, accuracy measurement methods, and leaderboard criteria. But the structure of human thought reflected through the downstream tasks in this chapter will remain.

Let's start with CoLA.

The Corpus of Linguistic Acceptability (CoLA)

The **Corpus of Linguistic Acceptability** (**CoLA**), a GLUE task, `https://gluebenchmark.com/tasks`, contains thousands of samples of English sentences annotated for grammatical acceptability.

The goal of *Alex Warstadt* et al. (2019) was to evaluate the linguistic competence of an NLP model to judge the linguistic acceptability of a sentence. The NLP model is expected to classify the sentences accordingly.

The sentences are labeled as grammatical or ungrammatical. The sentence is labeled `0` if the sentence is not grammatically acceptable. The sentence is labeled `1` if the sentence is grammatically acceptable. For example:

Classification = `1` for 'we yelled ourselves hoarse.'

Classification = `0` for 'we yelled ourselves.'

You can go through `BERT_Fine_Tuning_Sentence_Classification_GPU.ipynb` in *Chapter 3*, *Fine-Tuning BERT Models*, to view the BERT model that we fine-tuned on CoLA datasets. We used CoLA data:

```
#@title Loading the Dataset
#source of dataset : https://nyu-mll.github.io/CoLA/
df = pd.read_csv("in_domain_train.tsv", delimiter='\t', header=None,
names=['sentence_source', 'label', 'label_notes', 'sentence'])
df.shape
```

We also load a pretrained BERT model:

```
#@title Loading the Hugging Face Bert Uncased Base Model
model = BertForSequenceClassification.from_pretrained("bert-base-uncased",
num_labels=2)
```

Finally, the measurement method, or metric, we used is MCC, which was described in the *Evaluating using Matthews Correlation Coefficient* section of *Chapter 3*, *Fine-Tuning BERT Models*, and earlier in this chapter.

You can refer to that section for the mathematical description of MCC and take the time to rerun the source code if necessary.

A sentence can be grammatically unacceptable but still convey a sentiment. Sentiment analysis can add some form of empathy to a machine.

Stanford Sentiment TreeBank (SST-2)

Stanford Sentiment TreeBank (**SST-2**) contains movie reviews. In this section, we will describe the SST-2 (binary classification) task. However, the datasets go beyond that, and it is possible to classify sentiments in a range of *0* (negative) to *n* (positive).

Socher et al. (2013) took sentiment analysis beyond the binary positive-negative NLP classification. We will explore the SST-2 multi-label sentiment classification with a transformer model in *Chapter 12*, *Detecting Customer Emotions to Make Predictions*.

In this section, we will run a sample taken from SST on a Hugging Face transformer pipeline model to illustrate binary classification.

Open `Transformer_tasks.ipynb` and run the following cell, which contains positive and negative movie reviews taken from SST:

```
#@title SST-2 Binary Classification
from transformers import pipeline
nlp = pipeline("sentiment-analysis")
print(nlp("If you sometimes like to go to the movies to have fun , Wasabi
is a good place to start."),"If you sometimes like to go to the movies to
have fun , Wasabi is a good place to start.")
print(nlp("Effective but too-tepid biopic."),"Effective but too-tepid
biopic.")
```

The output is accurate:

```
[{'label': 'POSITIVE', 'score': 0.999825656414032}] If you sometimes like
to go to the movies to have fun , Wasabi is a good place to start .
[{'label': 'NEGATIVE', 'score': 0.9974064230918884}] Effective but too-
tepid biopic.
```

The SST-2 task is evaluated using the accuracy metric.

We classify sentiments of a sequence. Let's now see whether two sentences in a sequence are paraphrases or not.

Microsoft Research Paraphrase Corpus (MRPC)

The **Microsoft Research Paraphrase Corpus** (**MRPC**), a GLUE task, contains pairs of sentences extracted from new sources on the web. Each pair has been annotated by a human to indicate whether the sentences are equivalent based on two closely related properties:

- Paraphrase equivalent
- Semantic equivalent

Let's run a sample using the Hugging Face BERT model. Open `Transformer_tasks.ipynb` and go to the following cell, and then run the sample taken from MRPC:

```
#@title Sequence Classification : paraphrase classification
from transformers import AutoTokenizer,
TFAutoModelForSequenceClassification
import tensorflow as tf
tokenizer = AutoTokenizer.from_pretrained("bert-base-cased-finetuned-
mrpc")
```

```
model = TFAutoModelForSequenceClassification.from_pretrained("bert-base-cased-finetuned-mrpc")
classes = ["not paraphrase", "is paraphrase"]
sequence_A = "The DVD-CCA then appealed to the state Supreme Court."
sequence_B = "The DVD CCA appealed that decision to the U.S. Supreme Court."
paraphrase = tokenizer.encode_plus(sequence_A, sequence_B, return_tensors="tf")
paraphrase_classification_logits = model(paraphrase)[0]
paraphrase_results = tf.nn.softmax(paraphrase_classification_logits, axis=1).numpy()[0]
print(sequence_B, "should be a paraphrase")
for i in range(len(classes)):
    print(f"{classes[i]}: {round(paraphrase_results[i] * 100)}%")
```

The output is accurate, though you may get messages warning you that the model needs more downstream training:

```
The DVD CCA appealed that decision to the U.S. Supreme Court. should be a paraphrase
not paraphrase: 8.0%
is paraphrase: 92.0%
```

The MRPC task is measured with the F1/Accuracy score method.

Let's now run a Winograd schema.

Winograd schemas

We described the Winograd schemas in this chapter's *The Winograd schema challenge (WSC)* section. The training set was in English.

But what happens if we ask a transformer model to solve a pronoun gender problem in an English-French translation? French has different spellings for nouns that have grammatical genders (feminine, masculine).

The following sentence contains the pronoun *it*, which can refer to the words *car* or *garage*. Can a transformer disambiguate this pronoun?

Open `Transformer_tasks.ipynb`, go to the `#Winograd` cell, and run our example:

```
#@title Winograd
from transformers import pipeline
translator = pipeline("translation_en_to_fr")
print(translator("The car could not go in the garage because it was too
big.", max_length=40))
```

The translation is perfect:

```
[{'translation_text': "La voiture ne pouvait pas aller dans le garage
parce qu'elle était trop grosse."}]
```

The transformer detected that the word *it* refers to the word *car*, which is a feminine form. The feminine form applies to *it* and the adjective *big*:

elle means *she* in French, which is the translation of *it*. The masculine form would have been *il*, which means *he*.

grosse is the feminine form of the translation of the word *big*. Otherwise, the masculine form would have been *gros*.

We gave the transformer a difficult Winograd schema to solve, and it produced the right answer.

There are many more dataset-driven NLU tasks available. We will explore some of them throughout this book to add more building blocks to our toolbox of transformers.

Summary

This chapter analyzed the difference between the human language representation process and the way machine intelligence performs transduction. We saw that transformers must rely on the outputs of our incredibly complex thought processes expressed in written language. Language remains the most precise way to express a massive amount of information. The machine has no senses and must convert speech to text to extract meaning from raw datasets.

We then explored how to measure the performance of multi-task transformers. Transformers' ability to obtain top-ranking results for downstream tasks is unique in NLP history. We went through the tough SuperGLUE tasks that brought transformers up to the top ranks of the GLUE and SuperGLUE leaderboards.

BoolQ, CB, WiC, and the many other tasks we covered are by no means easy to process, even for humans. We went through an example of several downstream tasks that show the difficulty transformer models face in proving their efficiency.

Transformers have proven their value by outperforming the former NLU architectures. To illustrate how simple it is to implement downstream fine-tuned tasks, we then ran several tasks in a Google Colaboratory notebook using Hugging Face's pipeline for transformers.

In *Winograd schemas*, we gave the transformer the difficult task of solving a Winograd disambiguation problem for an English-French translation.

In the next chapter, *Chapter 6*, *Machine Translation with the Transformer*, we will take translation tasks a step further and build a translation model with Trax.

Questions

1. Machine intelligence uses the same data as humans to make predictions. (True/False)
2. SuperGLUE is more difficult than GLUE for NLP models. (True/False)
3. BoolQ expects a binary answer. (True/False)
4. WiC stands for Words in Context. (True/False)
5. **Recognizing Textual Entailment** (**RTE**) detects whether one sequence entails another sequence. (True/False)
6. A Winograd schema predicts whether a verb is spelled correctly. (True/False)
7. Transformer models now occupy the top ranks of GLUE and SuperGLUE. (True/False)
8. Human Baselines standards are not defined once and for all. They were made tougher to attain by SuperGLUE. (True/False)
9. Transformer models will never beat SuperGLUE Human Baselines standards. (True/False)
10. Variants of transformer models have outperformed RNN and CNN models. (True/False)

References

- *Alex Wang*, *Yada Pruksachatkun*, *Nikita Nangia*, *Amanpreet Singh*, *Julian Michael*, *Felix Hill*, *Omer Levy*, *Samuel R. Bowman*, 2019, *SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems*: `https://w4ngatang.github.io/static/papers/superglue.pdf`

- *Alex Wang*, *Yada Pruksachatkun*, *Nikita Nangia*, *Amanpreet Singh*, *Julian Michael*, *Felix Hill*, *Omer Levy*, *Samuel R. Bowman*, 2019, *GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding*
- *Yu Sun*, *Shuohuan Wang*, *Yukun Li*, *Shikun Feng*, *Hao Tian*, *Hua Wu*, *Haifeng Wang*, 2019, *ERNIE 2.0: A Continual Pretraining Framework for Language Understanding*: `https://arxiv.org/pdf/1907.12412.pdf`
- *Melissa Roemmele*, *Cosmin Adrian Bejan*, and *Andrew S. Gordon*, 2011, *Choice of Plausible Alternatives: An Evaluation of Commonsense Causal Reasoning*: `https://people.ict.usc.edu/~gordon/publications/AAAI-SPRING11A.PDF`
- *Richard Socher*, *Alex Perelygin*, *Jean Y. Wu*, *Jason Chuang*, *Christopher D. Manning*, *Andrew Y. Ng*, and *Christopher Potts*, 2013, *Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank*: `https://nlp.stanford.edu/~socherr/EMNLP2013_RNTN.pdf`
- *Thomas Wolf*, *Lysandre Debut*, *Victor Sanh*, *Julien Chaumond*, *Clement Delangue*, *Anthony Moi*, *Pierric Cistac*, *Tim Rault*, *Rémi Louf*, *Morgan Funtowicz*, *Jamie Brew*, 2019, *HuggingFace's Transformers: State-of-the-art Natural Language Processing*: `https://arxiv.org/abs/1910.03771`
- Hugging Face Transformer Usage: `https://huggingface.co/transformers/usage.html`

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the authors:

`https://www.packt.link/Transformers`

6

Machine Translation with the Transformer

Humans master sequence transduction, transferring a representation to another object. We can easily imagine a mental representation of a sequence. If somebody says *The flowers in my garden are beautiful*, we can easily visualize a garden with flowers in it. We see images of the garden, although we might never have seen that garden. We might even imagine chirping birds and the scent of flowers.

A machine must learn transduction from scratch with numerical representations. Recurrent or convolutional approaches have produced interesting results but have not reached significant BLEU translation evaluation scores. Translating requires the representation of language *A* transposed into language *B*.

The transformer model's self-attention innovation increases the analytic ability of machine intelligence. A sequence in language *A* is adequately represented before attempting to translate it into language *B*. Self-attention brings the level of intelligence required by a machine to obtain better BLEU scores.

The seminal *Attention Is All You Need* Transformer obtained the best results for English-German and English-French translations in 2017. Since then, the scores have been improved by other transformers.

At this point in the book, we have covered the essential aspects of transformers: the *architecture* of the Transformer, *training* a RoBERTa model from scratch, *fine-tuning* a BERT, *evaluating* a fine-tuned BERT, and exploring *downstream tasks* with some transformer examples.

In this chapter, we will go through machine translation in three additional topics. We will first define what machine translation is. We will then preprocess a **Workshop on Machine Translation (WMT)** dataset. Finally, we will see how to implement machine translations.

This chapter covers the following topics:

- Defining machine translation
- Human transductions and translations
- Machine transductions and translations
- Preprocessing a WMT dataset
- Evaluating machine translation with BLEU
- Geometric evaluations
- Chencherry smoothing
- Introducing Google Translate's API
- Initializing the English-German problem with Trax

Our first step will be to define machine translation.

Defining machine translation

Vaswani et al. (2017) tackled one of the most difficult NLP problems when designing the Transformer. The human baseline for machine translation seems out of reach for us human-machine intelligence designers. This did not stop *Vaswani* et al. (2017) from publishing the Transformer's architecture and achieving state-of-the-art BLEU results.

In this section, we will define machine translation. Machine translation is the process of reproducing human translation by machine transductions and outputs:

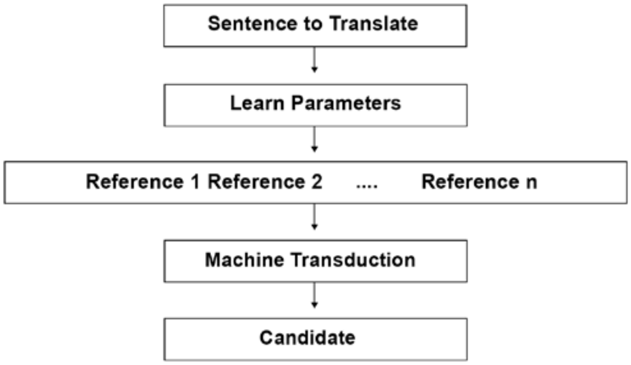


Figure 6.1: Machine translation process

The general idea in *Figure 6.1* is for the machine to do the following in a few steps:

- Choose a sentence to translate
- Learn how words relate to each other with hundreds of millions of parameters
- Learn the many ways in which words refer to each other
- Use machine transduction to transfer the learned parameters to new sequences
- Choose a candidate translation for a word or sequence

The process always starts with a sentence to translate from a source language, *A*. The process ends with an output containing a translated sentence in language *B*. The intermediate calculations involve transductions.

Human transductions and translations

A human interpreter at the European Parliament, for instance, will not translate a sentence word by word. Word-by-word translations often make no sense because they lack the proper *grammatical structure* and cannot produce the right translation because the *context* of each word is ignored.

Human transduction takes a sentence in language *A* and builds a cognitive *representation* of the sentence's meaning. An interpreter (oral translations) or a translator (written translations) at the European Parliament will only then transform that transduction into an interpretation of that sentence in language *B*.

We will name the translation done by the interpreter or translator in language *B* a *reference* sentence.

You will notice several references in the *Machine translation process* described in *Figure 6.1*.

A human translator will not translate sentence *A* into sentence *B* several times but only once in real life. However, more than one translator could translate sentence *A* in real life. For example, you can find several French to English translations of *Les Essais* by Montaigne. If you take one sentence, *A*, out of the original French version, you will thus find several versions of sentence *B* noted as references `1` to `n`.

If you go to the European Parliament one day, you might notice that the interpreters only translate for a limited time of two hours, for example. Then another interpreter takes over. No two interpreters have the same style, just like writers have different styles. Sentence *A* in the source language might be repeated by the same person several times in a day but be translated into several reference sentence *B* versions:

$$reference = \{reference\ 1, reference\ 2, \ldots reference\ n\}$$

Machines have to find a way to think the same way as human translators.

Machine transductions and translations

The transduction process of the original Transformer architecture uses the encoder stack, the decoder stack, and all the model's parameters to represent a *reference sequence*. We will refer to that output sequence as the *reference*.

Why not just say "output prediction"? The problem is that there is no single output prediction. The Transformer, like humans, will produce a result we can refer to, but that can change if we train it differently or use different transformer models!

We immediately realize that the human baseline of human transduction, representations of a language sequence, is quite a challenge. However, much progress has been made.

An evaluation of machine translation proves that NLP has progressed. To determine that one solution is better than another, each NLP challenger, lab, or organization must refer to the same datasets for the comparison to be valid.

Let's now explore a WMT dataset.

Preprocessing a WMT dataset

Vaswani et al. (2017) present the Transformer's achievements on the WMT 2014 English-to-German translation task and the WMT 2014 English-to-French translation task. The Transformer achieves a state-of-the-art BLEU score. BLEU will be described in the *Evaluating machine translation with BLEU* section of this chapter.

The 2014 WMT contained several European language datasets. One of the datasets contained data taken from version 7 of the Europarl corpus. We will be using the French-English dataset from the *European Parliament Proceedings Parallel Corpus*, 1996-2011 (`https://www.statmt.org/europarl/v7/fr-en.tgz`).

Once you have downloaded the files and have extracted them, we will preprocess the two parallel files:

- `europarl-v7.fr-en.en`
- `europarl-v7.fr-en.fr`

We will load, clear, and reduce the size of the corpus.

Let's start the preprocessing.

Preprocessing the raw data

In this section, we will preprocess `europarl-v7.fr-en.en` and `europarl-v7.fr-en.fr`.

Open `read.py`, which is in this chapter's GitHub directory. Ensure that the two europarl files are in the same directory as `read.py`.

The program begins using standard Python functions and `pickle` to dump the serialized output files:

```
import pickle
from pickle import dump
```

Then we define the function to load the file into memory:

```
# load doc into memory
def load_doc(filename):
        # open the file as read only
        file = open(filename, mode='rt', encoding='utf-8')
        # read all text
        text = file.read()
        # close the file
        file.close()
        return text
```

The loaded document is then split into sentences:

```
# split a loaded document into sentences
def to_sentences(doc):
        return doc.strip().split('\n')
```

The shortest and the longest lengths are retrieved:

```
# shortest and longest sentence lengths
def sentence_lengths(sentences):
        lengths = [len(s.split()) for s in sentences]
        return min(lengths), max(lengths)
```

The imported sentence lines must be cleaned to avoid training useless and noisy tokens. The lines are normalized, tokenized on white spaces, and converted to lowercase. The punctuation is removed from each token, non-printable characters are removed, and tokens containing numbers are excluded. The cleaned line is stored as a string.

The program runs the cleaning function and returns clean appended strings:

```
# clean lines
import re
import string
import unicodedata
def clean_lines(lines):
        cleaned = list()
        # prepare regex for char filtering
        re_print = re.compile('[^%s]' % re.escape(string.printable))
        # prepare translation table for removing punctuation
        table = str.maketrans('', '', string.punctuation)
        for line in lines:
                # normalize unicode characters
                line = unicodedata.normalize('NFD', line).encode('ascii',
'ignore')
                line = line.decode('UTF-8')
                # tokenize on white space
                line = line.split()
                # convert to lower case
                line = [word.lower() for word in line]
                # remove punctuation from each token
                line = [word.translate(table) for word in line]
                # remove non-printable chars form each token
                line = [re_print.sub('', w) for w in line]
                # remove tokens with numbers in them
                line = [word for word in line if word.isalpha()]
                # store as string
                cleaned.append(' '.join(line))
        return cleaned
```

We have defined the key functions we will call to prepare the datasets. The English data is loaded and cleaned first:

```
# load English data
filename = 'europarl-v7.fr-en.en'
doc = load_doc(filename)
sentences = to_sentences(doc)
```

```
minlen, maxlen = sentence_lengths(sentences)
print('English data: sentences=%d, min=%d, max=%d' % (len(sentences),
minlen, maxlen))
cleanf=clean_lines(sentences)
```

The dataset is now clean, and `pickle` dumps it into a serialized file named `English.pkl`:

```
filename = 'English.pkl'
outfile = open(filename,'wb')
pickle.dump(cleanf,outfile)
outfile.close()
print(filename," saved")
```

The output shows the key statistics and confirms that `English.pkl` is saved:

```
English data: sentences=2007723, min=0, max=668
English.pkl  saved
```

We now repeat the same process with the French data and dump it into a serialized file named `French.pkl`:

```
# Load French data
filename = 'europarl-v7.fr-en.fr'
doc = load_doc(filename)
sentences = to_sentences(doc)
minlen, maxlen = sentence_lengths(sentences)
print('French data: sentences=%d, min=%d, max=%d' % (len(sentences),
minlen, maxlen))
cleanf=clean_lines(sentences)
filename = 'French.pkl'
outfile = open(filename,'wb')
pickle.dump(cleanf,outfile)
outfile.close()
print(filename," saved")
```

The output shows the key statistics for the French dataset and confirms that `French.pkl` is saved.

```
French data: sentences=2007723, min=0, max=693
French.pkl saved
```

The main preprocessing is done. But we still need to make sure the datasets do not contain noisy and confusing tokens.

Finalizing the preprocessing of the datasets

Now, open read_clean.py in the same directory as read.py. Our process now defines the function that will load the datasets that were cleaned up in the previous section and then save them once the preprocessing is finalized:

```
from pickle import load
from pickle import dump
from collections import Counter

# load a clean dataset
def load_clean_sentences(filename):
        return load(open(filename, 'rb'))

# save a list of clean sentences to file
def save_clean_sentences(sentences, filename):
        dump(sentences, open(filename, 'wb'))
        print('Saved: %s' % filename)
```

We now define a function that will create a vocabulary counter. It is important to know how many times a word is used in the sequences we will parse. For example, if a word is only used once in a dataset containing two million lines, we will waste our energy using precious GPU resources to learn it. Let's define the counter:

```
# create a frequency table for all words
def to_vocab(lines):
        vocab = Counter()
        for line in lines:
                tokens = line.split()
                vocab.update(tokens)
        return vocab
```

The vocabulary counter will detect words with a frequency that is below min_occurrence:

```
# remove all words with a frequency below a threshold
def trim_vocab(vocab, min_occurrence):
        tokens = [k for k,c in vocab.items() if c >= min_occurrence]
        return set(tokens)
```

In this case, `min_occurrence=5` and the words below or equal to this threshold have been removed to avoid wasting the training model's time analyzing them.

We now have to deal with **Out-Of-Vocabulary** (**OOV**) words. OOV words can be misspelled words, abbreviations, or any word that does not fit standard vocabulary representations. We could use automatic spelling, but it would not solve all of the problems. For this example, we will simply replace OOV words with the `unk` (unknown) token:

```
# mark all OOV with "unk" for all lines
def update_dataset(lines, vocab):
        new_lines = list()
        for line in lines:
                new_tokens = list()
                for token in line.split():
                        if token in vocab:
                                new_tokens.append(token)
                        else:
                                new_tokens.append('unk')
                new_line = ' '.join(new_tokens)
                new_lines.append(new_line)
        return new_lines
```

We will now run the functions for the English dataset, save the output, and then display `20` lines:

```
# load English dataset
filename = 'English.pkl'
lines = load_clean_sentences(filename)
# calculate vocabulary
vocab = to_vocab(lines)
print('English Vocabulary: %d' % len(vocab))
# reduce vocabulary
vocab = trim_vocab(vocab, 5)
print('New English Vocabulary: %d' % len(vocab))
# mark out of vocabulary words
lines = update_dataset(lines, vocab)
# save updated dataset
filename = 'english_vocab.pkl'
```

```
save_clean_sentences(lines, filename)
# spot check
for i in range(20):
        print("line",i,":",lines[i])
```

The output functions first show the vocabulary compression obtained:

```
English Vocabulary: 105357
New English Vocabulary: 41746
Saved: english_vocab.pkl
```

The preprocessed dataset is saved. The output function then displays `20` lines, as shown in the following excerpt:

```
line 0 : resumption of the session
line 1 : i declare resumed the session of the european parliament
adjourned on friday december and i would like once again to wish you a
happy new year in the hope that you enjoyed a pleasant festive period
line 2 : although, as you will have seen, the dreaded millennium bug
failed to materialise still the people in a number of countries suffered a
series of natural disasters that truly were dreadful
line 3 : you have requested a debate on this subject in the course of the
next few days during this partsession
```

Let's now run the functions for the French dataset, save the output, and then display `20` lines:

```
# load French dataset
filename = 'French.pkl'
lines = load_clean_sentences(filename)
# calculate vocabulary
vocab = to_vocab(lines)
print('French Vocabulary: %d' % len(vocab))
# reduce vocabulary
vocab = trim_vocab(vocab, 5)
print('New French Vocabulary: %d' % len(vocab))
# mark out of vocabulary words
lines = update_dataset(lines, vocab)
# save updated dataset
filename = 'french_vocab.pkl'
```

```
save_clean_sentences(lines, filename)
# spot check
for i in range(20):
        print("line",i,":",lines[i])
```

The output functions first show the vocabulary compression obtained:

```
French Vocabulary: 141642
New French Vocabulary: 58800
Saved: french_vocab.pkl
```

The preprocessed dataset is saved. The output function then displays `20` lines, as shown in the following excerpt:

```
line 0 : reprise de la session
line 1 : je declare reprise la session du parlement europeen qui avait ete
interrompue le vendredi decembre dernier et je vous renouvelle tous mes
vux en esperant que vous avez passe de bonnes vacances
line 2 : comme vous avez pu le constater le grand bogue de lan ne sest pas
produit en revanche les citoyens dun certain nombre de nos pays ont ete
victimes de catastrophes naturelles qui ont vraiment ete terribles
line 3 : vous avez souhaite un debat a ce sujet dans les prochains jours
au cours de cette periode de session
```

This section shows how raw data must be processed before training. The datasets are now ready to be plugged into a transformer to be trained.

Each line of the French dataset is the *sentence* to translate. Each line of the English dataset is the *reference* for a machine translation model. The machine translation model must produce an *English candidate translation* that matches the *reference*.

BLEU provides a method to evaluate `candidate` translations produced by machine translation models.

Evaluating machine translation with BLEU

Papineni et al. (2002) came up with an efficient way to evaluate a human translation. The human baseline was difficult to define. However, they realized that we could obtain efficient results if we compared human translation with machine translation, word for word.

Papineni et al. (2002) named their method the **Bilingual Evaluation Understudy Score** (BLEU).

In this section, we will use the **Natural Language Toolkit** (**NLTK**) to implement BLEU:

`http://www.nltk.org/api/nltk.translate.html#nltk.translate.bleu_score.sentence_bleu`

We will begin with geometric evaluations.

Geometric evaluations

The BLEU method compares the parts of a candidate sentence to a reference sentence or several reference sentences.

Open `BLEU.py`, which is in the chapter directory of the GitHub repository of this book.

The program imports the `nltk` module:

```
from nltk.translate.bleu_score import sentence_bleu
from nltk.translate.bleu_score import SmoothingFunction
```

It then simulates a comparison between a candidate translation produced by the machine translation model and the actual translation(s) references in the dataset. Remember that a sentence could have been repeated several times and translated by different translators in different ways, making it challenging to find efficient evaluation strategies.

The program can evaluate one or more references:

```
#Example 1
reference = [['the', 'cat', 'likes', 'milk'], ['cat', 'likes' 'milk']]
candidate = ['the', 'cat', 'likes', 'milk']
score = sentence_bleu(reference, candidate)
print('Example 1', score)
#Example 2
reference = [['the', 'cat', 'likes', 'milk']]
candidate = ['the', 'cat', 'likes', 'milk']
score = sentence_bleu(reference, candidate)
print('Example 2', score)
```

The score for both examples is 1:

```
Example 1 1.0
Example 2 1.0
```

A straightforward evaluation *P* of the candidate *C*, the reference *R*, and the number of correct tokens found in *C (N)* can be represented as a geometric function:

$$P(N, C, R) = \prod_{n=1}^{N} p_n$$

This geometric approach is rigid if you are looking for a 3-gram overlap, for example:

```
#Example 3
reference = [['the', 'cat', 'likes', 'milk']]
candidate = ['the', 'cat', 'enjoys','milk']
score = sentence_bleu(reference, candidate)
print('Example 3', score)
```

The output is severe if you are looking for 3-gram overlaps:

```
Warning (from warnings module):
  File
"C:\Users\Denis\AppData\Local\Programs\Python\Python37\lib\site-packages\
nltk\translate\bleu_score.py", line 490
    warnings.warn(_msg)
UserWarning:
Corpus/Sentence contains 0 counts of 3-gram overlaps.
BLEU scores might be undesirable; use SmoothingFunction().
Example 3 0.7071067811865475
```

A human can see that the score should be `1` and not `0.7`. The hyperparameters can be changed, but the approach remains rigid.

The warning in the code above is a good one that announces the next section.

The messages may vary with each version of the program and each run since this is a stochastic process.

Papineni et al. (2002) came up with a modified unigram approach. The idea was to count the word occurrences in the reference sentence and ensure the word was not over-evaluated in the candidate sentence.

Consider the following example explained by *Papineni* et al. (2002):

`Reference 1: The cat is on the mat.`

`Reference 2: There is a cat on the mat.`

Now consider the following candidate sequence:

`Candidate: the the the the the the the`

We now look for the number of words in the candidate sentence (the `7` occurrences of the same word "the") present in the `Reference 1` sentence (`2` occurrences of the word "the").

A standard unigram precision would be `7/7`.

The modified unigram precision is `2/7`.

Note that the BLEU function output warning agrees and suggests using smoothing.

Let's add smoothing techniques to the BLEU toolkit.

Applying a smoothing technique

Chen and *Cherry* (2014) introduced a smoothing technique that improves standard BLEU techniques' geometric evaluation approach.

Label smoothing is a very efficient method that improves the performance of a transformer model during the training phase. It has a negative impact on perplexity. However, it forces the model to be more uncertain. In turn, this has a positive effect on accuracy.

For example, suppose we have to predict what the masked word is in the following sequence:

`The cat [mask] milk.`

Imagine the output comes out as a softmax vector:

```
candidate_words=[drinks, likes, enjoys, appreciates]
candidate_softmax=[0.7, 0.1, 0.1,0.1]
candidate_one_hot=[1,0,0,0]
```

This would be a brutal approach. Label smoothing can make the system more open-minded by introducing epsilon = ε.

The number of elements of `candidate_softmax` is *k=4*.

For label smoothing, we can set ε to `0.25`, for example.

One of the several approaches to label smoothing can be a straightforward function.

First, reduce the value of `candidate_one_hot` by $1\text{-}\varepsilon$.

Increase the `0` values by $0 + \frac{\varepsilon}{k\text{-}1}$.

We obtain the following result if we apply this approach:

`candidate_smoothed=[0.75,0.083,0.083,0.083]`, making the output open to future transformations and changes.

The transformer uses variants of label smoothing.

A variant of BLEU is chencherry smoothing.

Chencherry smoothing

Chen and *Cherry* (2014) introduced an interesting way of smoothing candidate evaluations by adding ε to otherwise `0` values. There are several chencherry (Boxing Chen + Colin Cherry) methods: `https://www.nltk.org/api/nltk.translate.html`.

Let's first evaluate a French-English example with smoothing:

```
#Example 4
reference = [['je','vous','invite', 'a', 'vous', 'lever','pour', 'cette',
'minute', 'de', 'silence']]
candidate = ['levez','vous','svp','pour', 'cette', 'minute', 'de',
'silence']
score = sentence_bleu(reference, candidate)
print("without soothing score", score)
```

Although a human could accept the candidate, the output score is weak:

```
without smoothing score 0.37188004246466494
```

Now, let's add some openminded smoothing to the evaluation:

```
chencherry = SmoothingFunction()
r1=list('je vous invite a vous lever pour cette minute de silence')
candidate=list('levez vous svp pour cette minute de silence')

#sentence_bleu([reference1, reference2, reference3],
hypothesis2,smoothing_function=chencherry.method1)
print("with smoothing score",sentence_bleu([r1], candidate,smoothing_
function=chencherry.method1))
```

The score does not reach human acceptability:

```
with smoothing score 0.6194291765462159
```

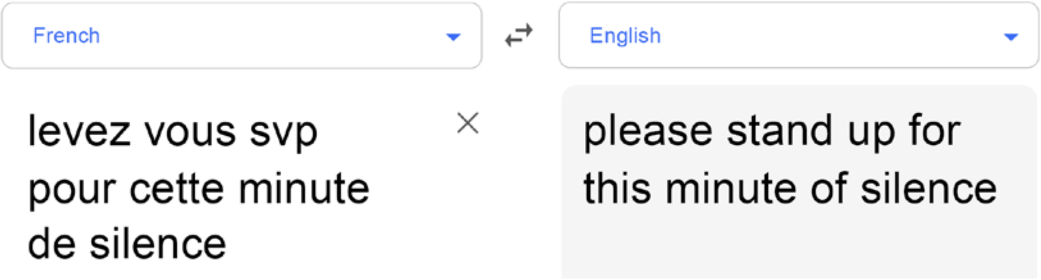
We have now seen how a dataset is preprocessed and how BLEU evaluates machine translations.

Translation with Google Translate

Google Translate, `https://translate.google.com/`, provides a ready-to-use interface for translations. Google is progressively introducing a transformer encoder into its translation algorithms. In the following section, we will implement a transformer model for a translation task with Google Trax.

However, an AI specialist may not be required at all.

If we enter the sentence analyzed in the previous section in Google Translate, `Levez-vous svp pour cette minute de silence`, we obtain an English translation in real time:



levez vous svp pour cette minute de silence

please stand up for this minute of silence

Figure 6.2: Google Translate

The translation is correct.

Does Industry 4.0 still require AI specialists for translation tasks or simply a web interface developer?

Google provides every service required for translations on their Google Translate platform: `https://cloud.google.com/translate`:

- A translation API: A web developer can create an interface for a customer
- A media translation API that can translate your streaming content
- An AutoML translation service that will train a custom model for a specific domain

A Google translate project requires a web developer for the interfaces, a **Subject Matter Expert (SME)**, and perhaps a linguist. However, an AI specialist is not a prerequisite.

Industry 4.0 is going toward AI as a service. So why bother studying AI development with transformers? There are two important reasons to become an Industry 4.0 AI specialist:

- In real life, AI projects often run into unexpected problems. For example, Google Translate might not fit a specific need no matter how much goodwill was put into the project. In that case, Google Trax will come in handy!
- To use Google Trax for AI, you need to be an AI developer!

You never know! The Fourth Industrial Revolution is connecting everything to everything. Some AI projects might run smoothly, and some will require AI expertise to solve complex problems. For example, in *Chapter 14*, *Interpreting Black Box Transformer Models*, we will show how AI development is sometimes required to implement Google Translate.

We are now ready to implement translations with Trax.

Translations with Trax

Google Brain developed **Tensor2Tensor** (**T2T**) to make deep learning development easier. T2T is an extension of TensorFlow and contains a library of deep learning models that contains many transformer examploes.

Although T2T was a good start, Google Brain then produced Trax, an end-to-end deep learning library. Trax contains a transformer model that can be applied to translations. The Google Brain team presently maintains Trax.

This section will focus on the minimum functions to initialize the English-German problem described by *Vaswani* et al. (2017) to illustrate the Transformer's performance.

We will be using preprocessed English and German datasets to show that the Transformer architecture is language-agnostic.

Open `Trax_Translation.ipynb`.

We will begin by installing the modules we need.

Installing Trax

Google Brain has made Trax easy to install and run. We will import the basics along with Trax, which can be installed in one line:

```
#@title Installing Trax
import os
import numpy as np
!pip install -q -U trax
import trax
```

Yes, it's that simple!

Now, let's create our transformer model.

Creating the original Transformer model

We will create the original Transformer model as described in *Chapter 2, Getting Started with the Architecture of the Transformer Model.*

Our Trax function will retrieve a pretrained model configuration in a few lines of code:

```
#@title Creating a Transformer model.
# Pre-trained model config in gs://trax-ml/models/translation/ende_wmt32k.
gin
model = trax.models.Transformer(
    input_vocab_size=33300,
    d_model=512, d_ff=2048,
    n_heads=8, n_encoder_layers=6, n_decoder_layers=6,
    max_len=2048, mode='predict')
```

The model is the Transformer with an encoder and decoder stack. Each stack contains 6 layers and 8 heads. `d_model=512`, as in the architecture of the original Transformer.

The Transformer requires the pretrained weights to run.

Initializing the model using pretrained weights

The pretrained weights contain the intelligence of the Transformer. The weights constitute the Transformer's representation of language. The weights can be expressed as a number of parameters that will produce some form of *machine intelligence IQ*.

Let's give life to the model by initializing the weights:

```
#@title Initializing the model using pre-trained weights
model.init_from_file('gs://trax-ml/models/translation/ende_wmt32k.pkl.gz',
                     weights_only=True)
```

The machine configuration and its *intelligence* are now ready to run. Let's tokenize a sentence.

Tokenizing a sentence

Our machine translator is ready to tokenize a sentence. The notebook uses the vocabulary preprocessed by Trax. The preprocessing method is similar to the one described in this chapter's *Preprocessing a WMT dataset* section.

The sentence will now be tokenized:

```
#@title Tokenizing a sentence.
sentence = 'I am only a machine but I have machine intelligence.'
tokenized = list(trax.data.tokenize(iter([sentence]), # Operates on
streams.
                                   vocab_dir='gs://trax-ml/vocabs/',
                                   vocab_file='ende_32k.subword'))[0]
```

The program will now decode the sentence and produce a translation.

Decoding from the Transformer

The Transformer encodes the sentence in English and will decode it in German. The model and its weights constitute its set of abilities.

Trax has made the decoding function intuitive to use:

```
#@title Decoding from the Transformer
tokenized = tokenized[None, :]  # Add batch dimension.
tokenized_translation = trax.supervised.decoding.autoregressive_sample(
    model, tokenized, temperature=0.0)  # Higher temperature: more diverse
results.
```

Note that higher temperatures will produce diverse results, just as with human translators, as explained in this chapter's *Defining machine translation* section.

Finally, the program will de-tokenize and display the translation.

De-tokenizing and displaying the translation

Google Brain has produced a mainstream, disruptive, and intuitive implementation of the Transformer with Trax.

The program now de-tokenizes and displays the translation in a few lines:

```
#@title De-tokenizing and Displaying the Translation
tokenized_translation = tokenized_translation[0][:-1]  # Remove batch and
EOS.
translation = trax.data.detokenize(tokenized_translation,
                                   vocab_dir='gs://trax-ml/vocabs/',
                                   vocab_file='ende_32k.subword')
print("The sentence:",sentence)
print("The translation:",translation)
```

The output is quite impressive:

```
The sentence: I am only a machine but I have machine intelligence.
The translation: Ich bin nur eine Maschine, aber ich habe
Maschinenübersicht.
```

The Transformer translated `machine intelligence` into `Maschinenübersicht`.

If we deconstruct `Maschinenübersicht` into `Maschin (machine)` + `übersicht (intelligence)`, we can see that:

- `über` literally means “over”
- `sicht` means “sight” or “view”

The transformer tells us that although it is a machine, it has vision. Machine intelligence is growing through Transformers, but it is not human intelligence. Machines learn languages with an intelligence of their own.

That concludes our experiment with Google Trax.

Summary

In this chapter, we went through three additional essential aspects of the original Transformer.

We started by defining machine translation. Human translation sets an extremely high baseline for machines to reach. We saw that English-French and English-German translations imply numerous problems to solve. The transformer tackled these problems and set state-of-the-art BLEU records to beat.

We then preprocessed a WMT French-English dataset from the European Parliament that required cleaning. We had to transform the datasets into lines and clean the data up. Once that was done, we reduced the dataset's size by suppressing words that occurred below a frequency threshold.

Machine translation NLP models require identical evaluation methods. Training a model on a WMT dataset requires BLEU evaluations. We saw that geometric assessments are a good basis for scoring translations, but even modified BLEU has its limits. We thus added a smoothing technique to enhance BLEU.

We saw that Google Translate provides a standard translation API, a media streaming API, and custom AutoML model training services. Implementing Google Translate APIs may require no AI development if the project rolls out smoothly. If not, we will have to get our hands dirty, like in the old days!

We implemented an English-to-German translation transformer with Trax, Google Brain's end-to-end deep learning library.

We have now covered the main building blocks to construct transformers: architecture, pretraining, training, preprocessing datasets, and evaluation methods.

In the next chapter, *The Rise of Suprahuman Transformers with GPT-3 Engines*, we will discover mind-blowing ways of implementing transformers with the building blocks we explored in the previous chapters.

Questions

1. Machine translation has now exceeded human baselines. (True/False)
2. Machine translation requires large datasets. (True/False)
3. There is no need to compare transformer models using the same datasets. (True/False)
4. BLEU is the French word for *blue* and is the acronym of an NLP metric (True/False)
5. Smoothing techniques enhance BERT. (True/False)
6. German-English is the same as English-German for machine translation. (True/False)
7. The original Transformer multi-head attention sub-layer has 2 heads. (True/False)
8. The original Transformer encoder has 6 layers. (True/False)
9. The original Transformer encoder has 6 layers but only 2 decoder layers. (True/False)
10. You can train transformers without decoders. (True/False)

References

- English-German BLEU scores with reference papers and code: `https://paperswithcode.com/sota/machine-translation-on-wmt2014-english-german`
- The 2014 **Workshop on Machine Translation** (**WMT**): `https://www.statmt.org/wmt14/translation-task.html`
- *European Parliament Proceedings Parallel Corpus 1996-2011*, parallel corpus French-English: `https://www.statmt.org/europarl/v7/fr-en.tgz`
- *Jason Brownlee, Ph.D.*, *How to Prepare a French-to-English Dataset for Machine Translation*: `https://machinelearningmastery.com/prepare-french-english-dataset-machine-translation/`
- *Kishore Papineni*, *Salim Roukos*, *Todd Ward*, and *Wei-Jing Zhu*, *2002*, *BLEU: a Method for Automatic Evaluation of Machine Translation*: `https://aclanthology.org/P02-1040.pdf`
- *Jason Brownlee, Ph.D.*, *A Gentle Introduction to Calculating the BLEU Score for Text in Python*: `https://machinelearningmastery.com/calculate-bleu-score-for-text-python/`
- *Boxing Chen* and *Colin Cherry*, 2014, *A Systematic Comparison of Smoothing Techniques for Sentence-Level BLEU*: `http://acl2014.org/acl2014/W14-33/pdf/W14-3346.pdf`
- *Ashish Vaswani*, *Noam Shazeer*, *Niki Parmar*, *Jakob Uszkoreit*, *Llion Jones*, *Aidan N. Gomez*, *Lukasz Kaiser*, and *Illia Polosukhin*, 2017, *Attention Is All You Need*: `https://arxiv.org/abs/1706.03762`
- Trax repository: `https://github.com/google/trax`
- Trax tutorial: `https://trax-ml.readthedocs.io/en/latest/`

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the authors:

`https://www.packt.link/Transformers`

7

The Rise of Suprahuman Transformers with GPT-3 Engines

In 2020, *Brown* et al. (2020) described the training of an OpenAI GPT-3 model containing 175 billion parameters that learned using huge datasets such as the 400 billion byte-pair-encoded tokens extracted from Common Crawl data. OpenAI ran the training on a Microsoft Azure supercomputer with 285,00 CPUs and 10,000 GPUs.

The machine intelligence of OpenAI's GPT-3 engines and their supercomputer led *Brown* et al. (2020) to zero-shot experiments. The idea was to use a trained model for downstream tasks without further training the parameters. The goal would be for a trained model to go directly into multi-task production with an API that could even perform tasks it wasn't trained for.

The era of suprahuman cloud AI engines was born. OpenAI's API requires no high-level software skills or AI knowledge. You might wonder why I used the term "suprahuman." You will discover that a GPT-3 engine can perform many tasks as well as a human in many cases. For the moment, *it is essential to understand how GPT models are built and run to appreciate the magic.*

This chapter will first examine the architecture and the evolution of the size of the transformer model. We will investigate the zero-shot challenge of using trained transformer models with little to no fine-tuning of the model's parameters for downstream tasks. We will explore the innovative architecture of GPT transformer models. OpenAI provides specially trained versions of their models named engines.

We will use a 345M parameter GPT-2 transformer in TensorFlow from OpenAI's repository. We must get our hands dirty to understand GPT models. We will interact with the model to produce text completion with general conditioning sentences.

We will continue by using a 117M parameter customized GPT-2 model. We will tokenize the high-level conceptual Kant dataset we used to train the RoBERTa model in *Chapter 4*, *Pretraining a RoBERTa Model from Scratch*.

The chapter will then explore using a GPT-3 engine that does not require a data scientist, an artificial specialist, or even an experienced developer to *get started*. However, that does not mean that a data scientist or an AI specialist will not be required down the line.

We will see that GPT-3 engines do sometimes require fine-tuning. We will run a Google Colab notebook to fine-tune a GPT-3 Ada engine.

The chapter will end with the new mindset and skillset of an Industry 4.0 AI specialist.

By the end of the chapter, you will know how a GPT model is built and how to use a seamless GPT-3 API. You will understand the gratifying tasks an Industry 4.0 AI specialist can accomplish in the 2020s!

This chapter covers the following topics:

- Getting started with a GPT-3 model
- The architecture of OpenAI GPT models
- Defining zero-shot transformer models
- The path from few-shots to one-shot
- Building a near-human GPT-2 text completion model
- Implementing a 345M parameter model and running it
- Interacting with GPT-2 with a standard model
- Training a language modeling GPT-2 117M parameter model
- Importing a customized and specific dataset
- Encoding a customized dataset
- Conditioning the model
- Conditioning a GPT-2 model for specific text completion tasks
- Fine-tuning a GPT-3 model
- The role of an Industry 4.0 AI specialist

Let's begin our journey by exploring GPT-3 transformer models.

Suprahuman NLP with GPT-3 transformer models

GPT-3 is built on the GPT-2 architecture. However, a fully trained GPT-3 transformer is a foundation model. A foundation model can do many tasks it wasn't trained for. GPT-3 completion applies all NLP tasks and even programming tasks.

> GPT-3 is one of the few fully trained transformer models that qualify as a foundation models. GPT-3 will no doubt lead to more powerful OpenAI models. Google will produce foundation models beyond the Google BERT version they trained on their supercomputers. Foundation models represent a new way of thinking about AI.

It will not take long for companies to realize they do not need a data scientist or an AI specialist to start an NLP project with an API like the one that OpenAI provides.

Why bother with any other tool? An OpenAI API is available with access to one of the most efficient transformer models trained on one of the most powerful supercomputers in the world.

Why develop tools, download libraries, or use any other tool if an API exists that only deep pockets and the best research teams in the world can design, such as Google or OpenAI?

The answer to these questions is quite simple. It's easy to start a GPT-3 engine, just as it is to start a Formula 1 or Indy 500 race car. No problem. But then, trying to drive such a car is nearly impossible without months of training! GPT-3 engines are powerful AI race cars. You can get them to run in a few clicks. However, mastering their incredible horsepower requires the knowledge you have acquired from the beginning of this book up to now and what you will discover in the following chapters!

We first need to understand the architecture of GPT models to see where developers, AI specialists, and data scientists fit in the era of suprahuman NLP models.

The architecture of OpenAI GPT transformer models

Transformers went from training, to fine-tuning, and finally to zero-shot models in less than three years between the end of 2017 and the first part of 2020. A zero-shot GPT-3 transformer model requires no fine-tuning. The trained model parameters are not updated for downstream multi-tasks, which opens a new era for NLP/NLU tasks.

In this section, we will first learn about the motivation of the OpenAI team that designed GPT models. We will begin by going through the fine-tuning of zero-shot models. Then we will see how to condition a transformer model to generate mind-blowing text completion. Finally, we will explore the architecture of GPT models.

We will first go through the creation process of the OpenAI team.

The rise of billion-parameter transformer models

The speed at which transformers went from small models trained for NLP tasks to models that require little to no fine-tuning is staggering.

Vaswani et al. (2017) introduced the Transformer, which surpassed CNNs and RNNs on BLEU tasks. *Radford* et al. (2018) introduced the **Generative Pre-Training** (**GPT**) model, which could perform downstream tasks with fine-tuning. *Devlin* et al. (2019) perfected fine-tuning with the BERT model. *Radford* et al. (2019) went further with GPT-2 models.

Brown et al. (2020) defined a GPT-3 zero-shot approach to transformers that does not require fine-tuning!

At the same time, *Wang* et al. (2019) created GLUE to benchmark NLP models. But transformer models evolved so quickly that they surpassed human baselines!

Wang et al. (2019, 2020) rapidly created SuperGLUE, set the human baselines much higher, and made the NLU/NLP tasks more challenging. Transformers are rapidly progressing, and some have already surpassed Human Baselines on the SuperGLUE leaderboards at the time of writing.

How did this happen so quickly?

We will look at one aspect, the models' sizes, to understand how this evolution happened.

The increasing size of transformer models

From 2017 to 2020 alone, the number of parameters increased from 65M parameters in the original Transformer model to 175B parameters in the GPT-3 model, as shown in *Table 7.1*:

Transformer Model	**Paper**	**Parameters**
Transformer Base	*Vaswani* et al. (2017)	65M
Transformer Big	*Vaswani* et al. (2017)	213M
BERT-Base	*Devlin* et al. (2019)	110M
BERT-Large	*Devlin* et al. (2019)	340M
GPT-2	*Radford* et al. (2019)	117M

GPT-2	*Radford* et al. (2019)	345M
GPT-2	*Radford* et al. (2019)	1.5B
GPT-3	*Brown* et al. (2020)	175B

Table 7.1: The evolution of the number of transformer parameters

Table 7.1 only contains the main models designed during that short time. The dates of the publications come after the date the models were actually designed. Also, the authors updated the papers. For example, once the original Transformer set the market in motion, transformers emerged from Google Brain and Research, OpenAI, and Facebook AI, which all produced new models in parallel.

Furthermore, some GPT-2 models are larger than the smaller GPT-3 models. For example, the GPT-3 Small model contains 125M parameters, which is smaller than the 345M parameter GPT-2 model.

The size of the architecture evolved at the same time:

- The number of layers of a model went from 6 layers in the original Transformer to 96 layers in the GPT-3 model
- The number of heads of a layer went from 8 in the original Transformer model to 96 in the GPT-3 model
- The context size went from 512 tokens in the original Transformer model to 12,288 in the GPT-3 model

The architecture's size explains why GPT-3 175B, with its 96 layers, produces more impressive results than GPT-2 1,542M, with only 40 layers. The parameters of both models are comparable, but the number of layers has doubled.

Let's focus on the context size to understand another aspect of the rapid evolution of transformers.

Context size and maximum path length

The cornerstone of transformer models resides in the attention sub-layers. In turn, the key property of attention sub-layers is the method used to process context size.

The context size is one of the main ways humans and machines can learn languages. The larger the context size, the more we can understand a sequence presented to us.

However, the drawback of context size is the distance it takes to understand what a word refers to. The path taken to analyze long-term dependencies requires changing from recurrent to attention layers.

The following sentence requires a long path to find what the pronoun "it" refers to:

"Our *house* was too small to fit a big couch, a large table, and other furniture we would have liked in such a tiny space. We thought about staying for some time, but finally, we decided to sell *it*."

The meaning of "it" can only be explained if we take a long path back to the word "house" at the beginning of the sentence. That's quite a path for a machine!

The order of function that defines the maximum path length can be summed up as shown in *Table 7.2* in Big *O* notation:

Layer Type	**Maximum Path Length**	**Context Size**
Self-Attention	0(1)	1
Recurrent	0(n)	100

Table 7.2: Maximum path length

Vaswani et al. (2017) optimized the design of context analysis in the original Transformer model. Attention brings the operations down to a one-to-one token operation. The fact that all of the layers are identical makes it much easier to scale up the size of transformer models. A GPT-3 model with a size 100 context window has the same maximum length path as a size 10 context window.

For example, a recurrent layer in an RNN has to store the total length of the context step by step. The maximum path length is the context size. The maximum length size for an RNN that would process the context size of a GPT-3 model would be 0(n) times longer. Furthermore, an RNN cannot split the context into 96 heads running on a parallelized machine architecture, distributing the operations over 96 GPUs, for example.

The flexible and optimized architecture of transformers has led to an impact on several other factors:

- *Vaswani* et al. (2017) trained a state-of-the-art transformer model with 36M sentences. *Brown* et al. (2020) trained a GPT-3 model with 400 billion byte-pair-encoded tokens extracted from Common Crawl data.
- Training large transformer models requires machine power that is only available to a small number of teams in the world. It took a total of $2.14*10^{23}$ FLOPS for *Brown* et al. (2020) to train GPT-3 175B.
- Designing the architecture of transformers requires highly qualified teams that can only be funded by a small number of organizations in the world.

The size and architecture will continue to evolve and probably increase to trillion-parameter models in the near future. Supercomputers will continue to provide the necessary resources to train transformers.

We will now see how zero-shot models were achieved.

From fine-tuning to zero-shot models

From the start, OpenAI's research teams, led by *Radford* et al. (2018), wanted to take transformers from trained models to GPT models. The goal was to train transformers on unlabeled data. Letting attention layers learn a language from unsupervised data was a smart move. Instead of teaching transformers to do specific NLP tasks, OpenAI decided to train transformers to learn a language.

OpenAI wanted to create a task-agnostic model. So they began to train transformer models on raw data instead of relying on labeled data by specialists. Labeling data is time-consuming and considerably slows down the transformer's training process.

The first step was to start with unsupervised training in a transformer model. Then, they would only fine-tune the model's supervised learning.

OpenAI opted for a decoder-only transformer described in the Stacking decoder layers section. The metrics of the results were convincing and quickly reached the level of the best NLP models of fellow NLP research labs.

The promising results of the first version of GPT transformer models soon led *Radford* et al. (2019) to come up with zero-shot transfer models. The core of their philosophy was to continue training GPT models to learn from raw text. They then took their research a step further, focusing on language modeling through examples of unsupervised distributions:

$$\text{Examples}=(x_1, x_2, x_3, ,x_n)$$

The examples are composed of sequences of symbols:

$$\text{Sequences}=(s_1, s_2, s_3, ,s_n)$$

This led to a metamodel that can be expressed as a probability distribution for any type of input:

$$p\ (output/input)$$

The goal was to generalize this concept to any type of downstream task once the trained GPT model understands a language through intensive training.

The GPT models rapidly evolved from 117M parameters to 345M parameters, to other sizes, and then to 1,542M parameters. 1,000,000,000+ parameter transformers were born. The amount of fine-tuning was sharply reduced. The results reached state-of-the-art metrics again.

This encouraged OpenAI to go further, much further. *Brown* et al. (2020) went on the assumption that conditional probability transformer models could be trained in-depth and were able to produce excellent results with little to no fine-tuning for downstream tasks:

$$p\ (output/multi\text{-}tasks)$$

OpenAI was reaching its goal of training a model and then running downstream tasks directly without further fine-tuning. This phenomenal progress can be described in four phases:

- **Fine-Tuning** (**FT**) is meant to be performed in the sense we have been exploring in previous chapters. A transformer model is trained and then fine-tuned on downstream tasks. *Radford* et al. (2018) designed many fine-tuning tasks. The OpenAI team then reduced the number of tasks progressively to `0` in the following steps.
- **Few-Shot** (**FS**) represents a huge step forward. The GPT is trained. When the model needs to make inferences, it is presented with demonstrations of the task to perform as conditioning. Conditioning replaces weight updating, which the GPT team excluded from the process. We will be applying conditioning to our model through the context we provide to obtain text completion in the notebooks we will go through in this chapter.
- **One-Shot** (**1S**) takes the process further. The trained GPT model is presented with only one demonstration of the downstream task to perform. No weight updating is permitted either.
- **Zero-Shot** (**ZS**) is the ultimate goal. The trained GPT model is presented with no demonstration of the downstream task to perform.

Each of these approaches has various levels of efficiency. The OpenAI GPT team has worked hard to produce these state-of-the-art transformer models.

We can now explain the motivations that led to the architecture of the GPT models:

- Teaching transformer models how to learn a language through extensive training.
- Focusing on language modeling through context conditioning.
- The transformer takes the context and generates text completion in a novel way. Instead of consuming resources on learning downstream tasks, it works on understanding the input and making inferences no matter what the task is.

- Finding efficient ways to train models by masking portions of the input sequences forces the transformer to think with machine intelligence. Thus, machine intelligence, though not human, is efficient.

We understand the motivations that led to the architecture of GPT models. Let's now have a look at the decoder-layer-only GPT model.

Stacking decoder layers

We now understand that the OpenAI team focused on language modeling. Therefore, it makes sense to keep the masked attention sublayer. Hence, the choice to retain the decoder stacks and exclude the encoder stacks. *Brown* et al. (2020) dramatically increased the size of the decoder-only transformer models to get excellent results.

GPT models have the same structure as the decoder stacks of the original Transformer designed by *Vaswani* et al. (2017). We described the decoder stacks in *Chapter 2, Getting Started with the Architecture of the Transformer Model*. If necessary, take a few minutes to go back through the architecture of the original Transformer.

The GPT model has a decoder-only architecture, as shown in *Figure 7.1*:

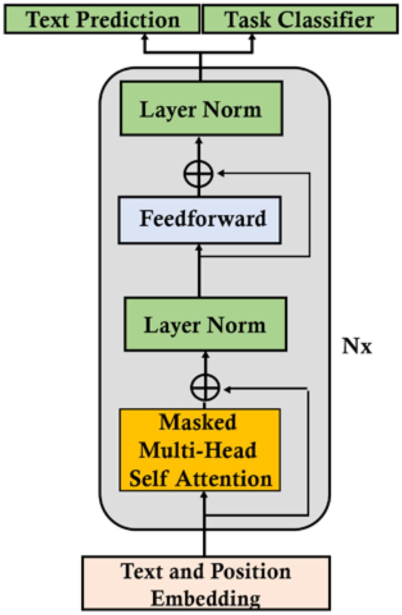


Figure 7.1: GPT decoder-only architecture

We can recognize the text and position embedding sub-layer, the masked multi-head self-attention layer, the normalization sub-layers, the feedforward sub-layer, and the outputs. In addition, there is a version of GPT-2 with both text prediction and task classification.

The OpenAI team customized and tweaked the decoder model by model. *Radford* et al. (2019) presented no fewer than four GPT models, and *Brown* et al. (2020) described no fewer than eight models.

The GPT-3 175B model has reached a unique size that requires computer resources that few teams in the world can access:

$$n_{params} = 175.0\text{B}, n_{layers} = 96, d_{model} = 12288, n_{heads} = 96$$

Let's look into the growing number of GPT-3 engines.

GPT-3 engines

A GPT-3 model can be trained to accomplish specific tasks of different sizes. The list of engines available at this time is documented by OpenAI: `https://beta.openai.com/docs/engines`

The base series of engines have different functions – for example:

- The Davinci engine can analyze complex intent
- The Curie engine is fast and has good summarization
- The Babbage engine is good at semantic search
- The Ada engine is good at parsing text

OpenAI is producing more engines to put on the market:

- The Instruct series provides instructions based on a description. An example is available in the *More GPT-3 examples* section of this chapter.
- The Codex series can translate language to code. We will explore this series in *Chapter 16*, *The Emergence of Transformer-Driven Copilots*.
- The Content filter series filters unsafe or sensitive text. We will explore this series in *Chapter 16*, *The Emergence of Transformer-Driven Copilots*.

We have explored the process that led us from fine-tuning to zero-shot GPT-3 models. We have seen that GPT-3 can produce a wide range of engines.

It is now time to see how the source code of GPT models is built. Although the GPT-3 transformer model source code is not publicly available at this time, GPT-2 models are sufficiently powerful to understand the inner workings of GPT models.

We are ready to interact with a GPT-2 model and train it.

We will first use a trained GPT-2 345M model for text completion with 24 decoder layers with self-attention sublayers of 16 heads.

We will then train a GPT-2 117M model for customized text completion with 12 decoder layers with self-attention layers of 12 heads.

Let's start by interacting with a pretrained 345M parameter GPT-2 model.

Generic text completion with GPT-2

We will explore an example with a GPT-2 generic model from top to bottom. *The goal of the example we will run is to determine the level of abstract reasoning a GPT model can attain.*

This section describes the interaction with a GPT-2 model for text completion. We will focus on *Step 9* of the `OpenAI_GPT_2.ipynb` notebook described in detail in *Appendix III*, *Generic Text Completion with GPT-2*.

> You can first read this section to see how the generic pretrained GPT-2 model will react to a specific example. Then read *Appendix III*, *Generic Text Completion with GPT-2*, to go into the details of how a generic GPT-2 model is implemented in a Google Colab notebook.
>
> You can also read *Appendix III* directly, which contains the interaction of *Step 9* described below.

First, let's understand the specific example of the pretrained GPT-2 being applied.

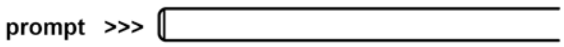
Step 9: Interacting with GPT-2

In this section, we will interact with the GPT-2 345M model.

To interact with the model, run the `interact_model` cell:

```
#@title Step 9: Interacting with GPT-2
interact_model('345M',None,1,1,300,1,0,'/content/gpt-2/models')
```

You will be prompted to enter some context:



prompt >>>

Figure 7.2: Context input for text completion

You can try any type of context you wish since this is a standard GPT-2 model.

We can try a sentence written by Immanuel Kant:

```
Human reason, in one sphere of its cognition, is called upon to
consider questions, which it cannot decline, as they are presented by
its own nature, but which it cannot answer, as they transcend every
faculty of the mind.
```

Press *Enter* to generate text. The output will be relatively random since the GPT-2 model was not trained on our dataset, and we are running a stochastic model anyway.

Let's have a look at the first few lines the GPT model generated at the time I ran it:

```
"We may grant to this conception the peculiarity that it is the only
causal logic.
In the second law of logic as in the third, experience is measured at its
end: apprehension is afterwards closed in consciousness.
The solution of scholastic perplexities, whether moral or religious, is
not only impossible, but your own existence is blasphemous."
```

To stop the cell, double-click on the run button of the cell.

You can also press *Ctrl* + *M* to stop generating text, but it may transform the code into text, and you will have to copy it back into a program cell.

The output is rich. We can observe several facts:

- The context we entered *conditioned* the output generated by the model.
- The context was a demonstration of the model. It learned what to say from the context without modifying its parameters.
- Text completion is conditioned by context. This opens the door to transformer models that do not require fine-tuning.
- From a semantic perspective, the output could be more interesting.
- From a grammatical perspective, the output is convincing.

Can we do better? The following section presents the interaction of custom text completion.

Training a custom GPT-2 language model

We will continue our top-to-bottom approach in this section by exploring an example with a GPT-2 custom model that we will train on a specific dataset. *The goal remains to determine the level of abstract reasoning a GPT model can attain*.

This section describes the interaction with a GPT-2 model for text completion trained on a specific dataset. We will focus on *Step 12* of the `Training_OpenAI_GPT_2.ipynb` notebook described in detail in *Appendix IV*, *Custom Text Completion with GPT-2*.

> You can read this section first to see how an example with a custom GPT-2 model will improve responses. Then read *Appendix IV*, *Custom Text Completion with GPT-2*, to understand how to train a GPT-2 to obtain specific responses.
>
> You can also decide to read *Appendix IV* directly, which also contains the interaction of *Step 12* described below.

First, let's understand how the interaction with GPT-2 improved by training it.

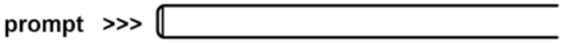
Step 12: Interactive context and completion examples

We will now run a conditional sample. The context we enter will condition the model to think as we want it to, to complete the text by generating tailor-made paragraphs.

Run the cell and explore the magic:

```
#@title Step 12: Interactive Context and Completion Examples
import os # import after runtime is restarted
os.chdir("/content/gpt-2/src")
!python interactive_conditional_samples.py --temperature 0.8 --top_k 40
--model_name '117M'
```

If necessary, take a few minutes to go back to *Step 9*, *Interacting with GPT-2* of *Appendix III*, *Generic Text Completion with GPT-2*, to see the differences in the responses. The program prompts us to enter the context:



prompt >>>

Figure 7.3: Context input for text completion

Let’s enter the same paragraph written by Immanuel Kant as we did in *Step 9* of the *Generic text completion with GPT-2* section of this chapter:

```
Human reason, in one sphere of its cognition, is called upon to
consider questions, which it cannot decline, as they are presented by
its own nature, but which it cannot answer, as they transcend every
faculty of the mind.
```

Press *Enter* to generate text as we did previously. Though structured and logical, the outputs might change from one run to another, making transformers attractive. This time, the result is not random and is impressive.

Let’s look at the first few lines the GPT-2 model produced:

```
"Now, what is the nature of the reason, except that it is of a kind and
power of mind that is so called on to think, and to understand?
The reason is the thing which apprehends, and seeks to understand,
the objects of its minds, which apprehend and apprehend the things of
itself. No one can speak of this reason without the understanding and the
understanding of the human reason. But that knowledge and understanding
which is the most natural of all the faculties, which, because it is
understood by the human reason, is the most natural of all the faculties,
is not without reason, but only without understanding. The human reason,
who sees and knows the things which the natural sense, which, though in
the natural sense, does not understand, is the source of every other
knowledge which we understand. The reason is the thing which does not
understand, but understands the things which we comprehend, and therefore,
is not to be understood. I mean that the reason is the thing which does
not understand, but knows only the things which we do understand…."
```

To stop the cell, double-click on the run button of the cell or press *Ctrl* + *M*.

Wow! I doubt anybody could see the difference between the text completion produced by our trained GPT-2 model and a human. It might also generate different outputs at each run.

In fact, I think our model could outperform many humans in this abstract exercise in philosophy, reason, and logic!

We can draw some conclusions from our experiment:

- A well-trained transformer model can produce text completion at a human level
- A GPT-2 model can almost reach human level in text generation on complex and abstract reasoning

- Text context is an efficient way of conditioning a model by demonstrating what is expected
- Text completion is text generation based on text conditioning if context sentences are provided

You can enter conditioning text context examples to experiment with text completion. You can also train the model on your own data. Just replace the content of the `dset.txt` file with your own data and see what happens!

Remember that our trained GPT-2 model will react like a human. If you enter a short, incomplete, uninteresting, or tricky context, you will obtain puzzled or bad results. This is because GPT-2 expects the best out of us, as in real life!

Let's go to the GPT-3 playground to see how a trained GPT-3 reacts to the example tested with GPT-2.

Running OpenAI GPT-3 tasks

In this section, we will run GPT-3 in two different ways:

- We will first run the GPT-3 tasks online with no code
- We will then implement GPT-3 in Google Colab notebook

> We will be using GPT-3 engines in this book. When you sign up for the GPT-3 API, OpenAI gives you a free budget to get started. This free budget should cover most of the cost, if not all of the cost, of running the examples in this book once or twice.

Let's begin by running NLP tasks online.

Running NLP tasks online

We will now go through some Industry 4.0 examples without an API, directly asking GPT-3 to do something for us.

Let us define a standard structure of a prompt and response as:

- *N* = name of the NLP task (INPUT).
- *E* = explanation for the GPT-3 engine. *E* precedes *T* (INPUT).
- *T* = the text or content we wish GPT-3 to look into (INPUT).
- *S* = showing GPT-3 what is expected. *S* follows *T* and is added when necessary (INPUT).
- *R* = GPT-3's response (OUTPUT).

The structure of the prompt described above is a guideline. However, GPT-3 is very flexible, and many variations are possible.

We are now ready to run some educational examples online with no API:

- Questions and answers (**Q&A**) on existing knowledge:

  E = `Q`

  T = `Who was the president of the United States in 1965?`

  S = None

  R = `A`

  Prompts and responses:

  `Q: Who was the president of the United States in 1965?`

  `A: Lyndon B. Johnson was president of the United States in 1965.`

  `Q: Who was the first human on the moon?`

  `A: Neil Armstrong was the first human on the moon.`

- **Movie to Emoji**:

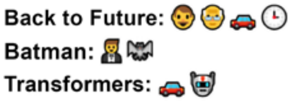
  E = Some examples of movie titles

  T = None

  S = Implicit through examples

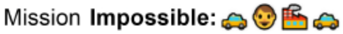
  R = Some examples of emojis

  Prompts and responses:



Back to Future: 👨👴🚗🕒
Batman: 🤵🦇
Transformers: 🚗🤖

  A new prompt and response:



Mission **Impossible:** 🚕👨🏭🚕

- Summarizing for a second grader (**Summarize for a 2nd grader**):

  E = `My second grader asked me what this passage means:`

  T = `"""The initial conclusions…."""`

S = I rephrased it for him, in plain language a second grader can understand: """

R = The summary

Prompt and response:

My second grader asked me what this passage means:

""" The initial conclusions can be divided into two categories: facts and fiction. The facts are that OpenAI has one of the most powerful NLP services in the world. The main facts are: OpenAI engines are powerful zero-shot that require no hunting for all kinds of transformer models, no pre-training, and no fine-tuning. The supercomputers used to train the models are unique. If the prompt is well-designed, we obtain surprisingly accurate responses. Implementing the NLP tasks in this section required a copy and paste action that any software beginner can perform. Fiction begins with dystopian and hype assertions AI will replace data scientists and AI specialists. Is that true? Before answering that question, first ask yourself the following questions about the example we just ran: How do we know the sentence was incorrect in the first place? How do we know the answer is correct without us humans reading and confirming this? How did the engine know it was a grammar correction task? If the response is incorrect, how can we understand what happened to help improve the prompt or revert to manual mode in a well-designed human interface? The truth is that humans will need to intervene to answers these questions manually, with rule-bases, quality control automated pipelines, and many other tools. The facts are convincing. It is true that running an NLP task requires little development. The fiction is not convincing. Humans are still required. OpenAI engines are not there to replace humans but to help them perform more high-level gratifying tasks. You can now fly a jet without having to build it! """

I rephrased it for him, in plain language a second grader can understand: """

The first conclusion is that OpenAI has one of the most powerful NLP services in the world. The most important facts are: OpenAI engines are very powerful zero-shot that require no hunting for all kinds of transformer models, no pre-training, and no fine-tuning. The supercomputers used to train the models are unique. If the prompt is well-designed, we get surprisingly accurate responses.

The results are impressive and a huge step forward in the history of NLP.

The number of NLP tasks we can implement with the zero-shot GPT-3 engines provided by OpenAI is endless since we control how we show what we expect of the transformer models.

However, a true Industry 4.0 AI guru must get their hands dirty before implementing ready-to-use APIs. We will now explore the architecture of OpenAI GPT models and then build GPT-2 models to see how these engines work.

The more we know about GPT models, the better an Industry 4.0 NLP expert can implement them in real-life projects.

Let's continue our top-to-bottom approach and drill down into the architecture of OpenAI GPT transformer models.

Getting started with GPT-3 engines

OpenAI has some of the most powerful transformer engines in the world. One GPT-3 model can perform hundreds of tasks. GPT-3 can do many tasks it wasn't trained for.

This section will use the API in `Getting_Started_GPT_3.ipynb`.

To use a GPT-3, you must first go to OpenAI's website, `https://openai.com/`, and sign up.

OpenAI has a playground for everybody to try, just like Google Translate or any user-friendly online service. So, let's try some tasks.

Running our first NLP task with GPT-3

Let's start using GPT-3 in a few steps.

Go to Google Colab and open `Getting_Started_GPT_3.ipynb`, which is the chapter directory of the book on GitHub.

You do not need to change the settings of the notebook. We are using an API, so we will not need much local computing power for the tasks in this section.

The steps of this section are the same ones as in the notebook.

Running an NLP is done in three simple steps:

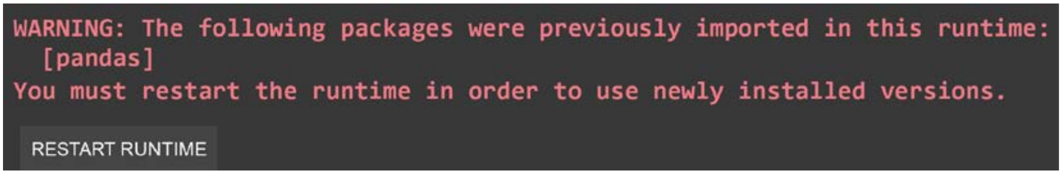
Step 1: Installing OpenAI

Install `openai` using the following command:

```
try:
  import openai
```

```
except:
  !pip install openai
  import openai
```

If openai is not installed, you must restart the runtime. A message will indicate when to do this, as shown in the following output:



```
WARNING: The following packages were previously imported in this runtime:
  [pandas]
You must restart the runtime in order to use newly installed versions.
```

Restart the runtime and then run this cell again to make sure openai is imported.

Step 2: Entering the API key

An API key is given that can be used with Python, C#, Java, and many other options. We will be using Python in this section:

```
openai.api_key=[YOUR API KEY]
```

You can now update the next cell with your API key:

```
import os
import openai
os.environ['OPENAI_API_KEY'] ='[YOUR_KEY or KEY variable]'
print(os.getenv('OPENAI_API_KEY'))
openai.api_key = os.getenv("OPENAI_API_KEY")
```

Let's now run an NLP task.

Step 3: Running an NLP task with the default parameters

We copy and paste an OpenAI example for a **grammar correction** task:

```
response = openai.Completion.create(
 engine="davinci",
 prompt="Original: She no went to the market.\nStandard American
English:",
  temperature=0,
  max_tokens=60,
  top_p=1.0,
  frequency_penalty=0.0,
  presence_penalty=0.0,
  stop=["\n"]
)
```

The task is to correct this grammar mistake: `She no went to the market`.

We can process the response as we wish by parsing it. OpenAI's response is a dictionary object. The OpenAI object contains detailed information on the task. We can ask the object to be displayed:

```
#displaying the response object
print(response)
```

We can explore the object:

```
{
  "choices": [
    {
      "finish_reason": "stop",
      "index": 0,
      "logprobs": null,
      "text": " She didn't go to the market."
    }
  ],
  "created": 1639424815,
  "id": "cmpl-4ElZfXLl9jGRNQoojWRRGof8AKr4y",
  "model": "davinci:2020-05-03",
  "object": "text_completion"}
```

The "created" number and "id", and "model" name can vary with each run.

We can then ask the object dictionary to display `"text"` and to print the processed output:

```
#displaying the response object
r = (response["choices"][0])
print(r["text"])
```

The output of "`text"` in the dictionary is the grammatically correct sentence:

```
She didn't go to the market.
```

NLP tasks and examples

Now we will cover an industrial approach to GPT-3 engine usage. For example, OpenAI provides an interactive educational interface that does not require an API. So a school teacher, a consultant, a linguist, a philosopher, or anybody that wishes to use a GPT-3 engine for educational purposes can do so with no experience at all in AI.

We will first begin by using an API in a notebook.

Grammar correction

If we go back to `Getting_Started_GPT_3.ipynb`, which we began to explore in the *Getting started with GPT-3 engines* section of this chapter, we can experiment with grammar correction with different prompts.

Open the notebook and go to *Step 4: Example 1: Grammar correction*:

```
#Step 6: Running an NLP task with custom parameters
response = openai.Completion.create(
  #defult engine: davinci
  engine="davinci",
  #default prompt for task:"Original"
  prompt="Original: She no went to the market.\n Standard American
English:",
  temperature=0,
  max_tokens=60,
  top_p=1.0,
  frequency_penalty=0.0,
  presence_penalty=0.0,
  stop=["\n"]
)
```

The request body is not limited to the prompt. The body contains several key parameters:

- `engine="davinci"`. The choice of the OpenAI GPT-3 engine to use and possibly other models in the future.
- `temperature=0`. A higher value such as `0.9` will force the model to take more risks. Do not modify the temperature and `top_p` at the same time.
- `max_tokens=60`. The maximum number of tokens of the response.

- `top_p=1.0`. Another way to control sampling like `temperature`. In this case, the `top_p` percentage of tokens of the probability mass will be considered. `0.2` would make the system only take 20% of the top probability mass.
- `frequency_penalty=0.0`. A value between `0` and `1` limits the frequency of tokens in a given response.
- `presence_penalty=0.0`. A value between `0` and `1` forces the system to use new tokens and produce new ideas.
- `stop=["\n"]`. A signal to the model to stop producing new tokens.

Some of these parameters are described at the source code level in the *Steps 7b-8: Importing and defining the model* section of *Appendix III*, *Generic Text Completion with GPT-2*.

You can play around with these parameters in the GPT-3 model if you gain access or in the GPT-2 model in *Appendix III*, *Generic Text Completion with GPT-2*. The concepts are the same in both cases.

This section will focus on the prompt:

```
prompt="Original: She no went to the market.\n Standard American English:"
```

The prompt can be divided into three parts:

- `Original`: This signals to the model that what follows is the original text, which the model will do something with
- `She no went to the market.\n`: This part of the prompt shows the model that this is the original text
- `Standard American English`: This shows the model what task is expected

Let's see how far we can get by changing the task:

- Standard American English produces:

  ```
  prompt="Original: She no went to the market.\n Standard American English:"
  ```

  The text in response is:

  ```
  "text": " She didn't go to the market."
  ```

  That is fine, but what if we do not want a contraction in the sentence?

- English with no contractions produces:

  ```
  prompt="Original: She no went to the market.\n English with no contractions:"
  ```

  The text in response is:

```
"text": " She did not go to the market."
```

Wow! This is impressive. Let's try another language.

- French with no contractions produces:

```
"text": " Elle n'est pas all\u00e9e au march\u00e9."
```

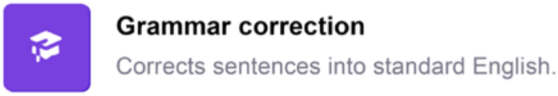
This is impressive. `\u00e9` simply needs to be post-processed into é.

Many more options are possible. Your Industry 4.0 cross-disciplinary imagination is the limit!

More GPT-3 examples

OpenAI contains many examples. OpenAI provides an online playground to explore tasks. OpenAI also provides source code for each example: `https://beta.openai.com/examples`

Just click on an example such as the grammar example we explored in the *Grammar correction* section:

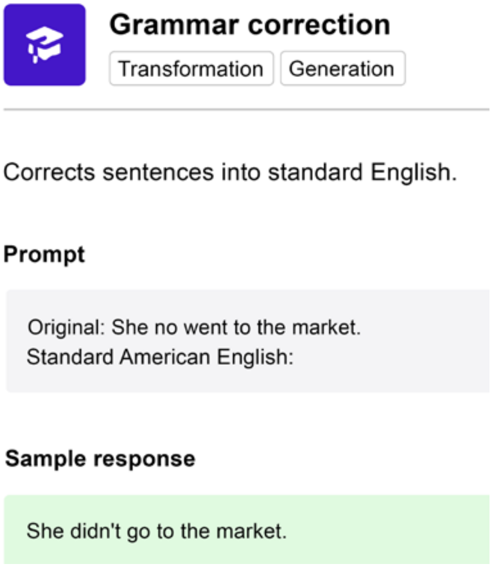


Grammar correction

Corrects sentences into standard English.

Figure 7.4: The Grammar correction section of OpenAI

OpenAI will describe the prompt and the sample response for each task.



Grammar correction

Corrects sentences into standard English.

Prompt

Original: She no went to the market.
Standard American English:

Sample response

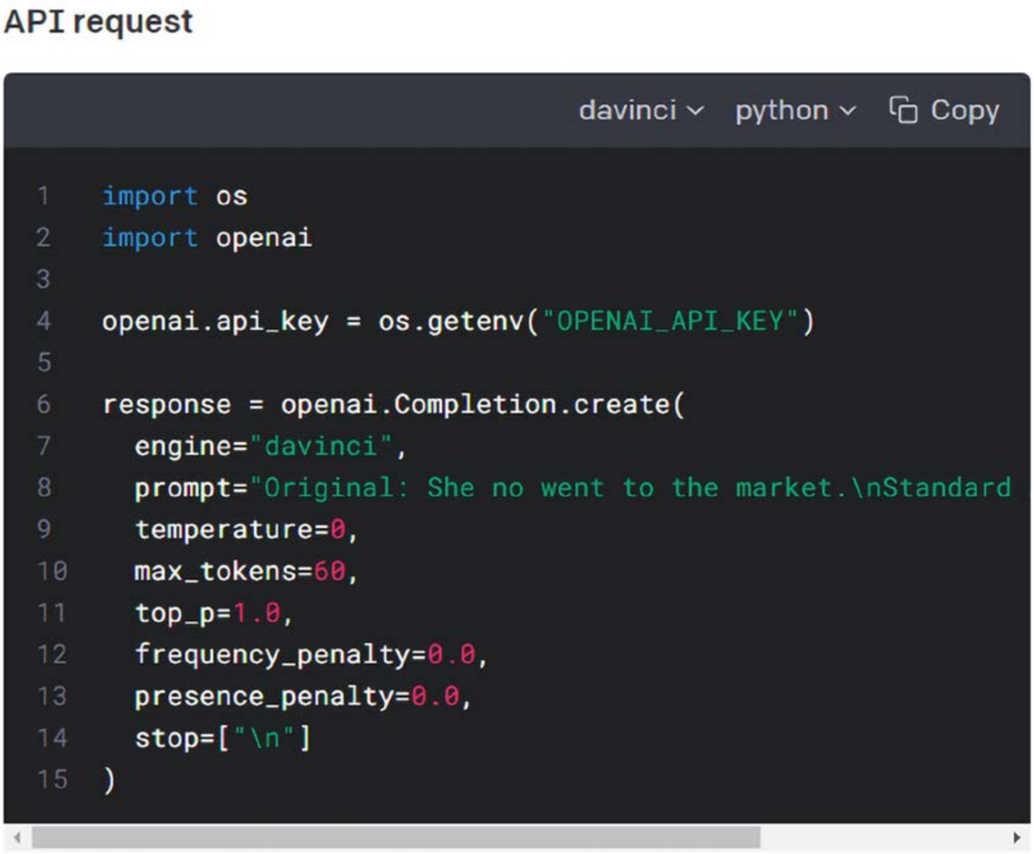
She didn't go to the market.

Figure 7.5: The sample response corrects the prompt

You can choose to go to the playground and run it online as we did in this chapter's *Running NLP tasks online* section. To do so, click on the **Open in Playground** button:

Figure 7.6: The Open in Playground button

You can choose to copy and paste the code to run the API as we are doing in the Google Colab notebook of this chapter:



```python
import os
import openai

openai.api_key = os.getenv("OPENAI_API_KEY")

response = openai.Completion.create(
  engine="davinci",
  prompt="Original: She no went to the market.\nStandard
  temperature=0,
  max_tokens=60,
  top_p=1.0,
  frequency_penalty=0.0,
  presence_penalty=0.0,
  stop=["\n"]
)
```

Figure 7.7: Running code using the Davinci engine

`Getting_Started_GPT_3.ipynb` contains ten examples that you can run to practice implementing the OpenAI GPT-3.

For each example:

- You can first read the link to the explanation provided by OpenAI. A link to the documentation is provided above each cell.
- You can then run the cell to observe GPT-3's behavior.

Run these ten examples in the notebook:

- Example 1: Grammar correction
- Example 2: English-to-French translation
- Example 3: Instruct series that provides instructions
- Example 4: Movie to emoji
- Example 5: Programming language to another language. For example, Python to JavaScript. Warning: you may need to obtain special permission from OpenAI to run this example, which uses the Davinci Codex engine, the code generator. If this example does not run in your notebook, please contact OpenAI to request access to Codex.
- Example 6: Advanced tweet classifier
- Example 7: Q&A
- Example 8 Summarize a text
- Example 9: Parse unstructured data
- Example 10: Calculate time complexity

You can run many other tasks on the Examples page: `https://beta.openai.com/examples`

Let's now compare the output of GPT-2 and GPT-3.

Comparing the output of GPT-2 and GPT-3

Our curiosity must be satisfied before we move on. What can the powerful GPT-3 model produce with the example we submitted to a pretrained GPT-2 model and then our custom-trained GPT-2 model?

Our example used for the GPT-2 model:

```
Human reason, in one sphere of its cognition, is called upon to consider
questions, which it cannot decline, as they are presented by its own nature,
but which it cannot answer, as they transcend every faculty of the mind.
GPT-3's response:
```

```
What is meant by this is that there are some things in our experience that
are beyond reason. It is not that reason is a bad guide or is insufficient,
but that there are some things that are beyond its reach. An example of this
would be the existence of God.
```

The result is mind-blowing! It explains what the text means, including some deep philosophical reflections!

We have proven our point in this chapter. Transformer models can attain to abstract reasoning, which can help make micro-decisions in our fast-moving world.

OpenAI GPT-3 is a fully trained model. However, GPT-3 can be fine-tuned. Let's see how.

Fine-tuning GPT-3

This section shows how to fine-tune GPT-3 to learn logic. Transformers need to learn logic, inferences, and entailment to understand language at a human level.

> Fine-tuning is the key to making GPT-3 your own application, to customizing it to make it fit the needs of your project. It's a ticket to AI freedom to rid your application of bias, teach it things you want it to know, and leave your footprint on AI.

In this section, GPT-3 will be trained on the works of Immanuel Kant using `kantgpt.csv`. We used a similar file to train the BERT-type model in *Chapter 4*, *Pretraining a RoBERTa Model from Scratch*.

Once you master fine-tuning GPT-3, you can use other types of data to teach it specific domains, knowledge graphs, and texts.

OpenAI provides an efficient, well-documented service to fine-tune GPT-3 engines. It has trained GPT-3 models to become different types of engines, as seen in the *The rise of billion-parameter transformer models* section of this chapter.

The Davinci engine is powerful but can be more expensive to use. The Ada engine is less expensive and produces sufficient results to explore GPT-3 in our experiment.

Fine-tuning GPT-3 involves two phases:

- Preparing the data
- Fine-tuning a GPT-3 model

Preparing the data

Open `Fine_Tuning_GPT_3.ipynb` in Google Colab in the GitHub chapter directory.

OpenAI has documented the data preparation process in detail:

`https://beta.openai.com/docs/guides/fine-tuning/prepare-training-data`

Step 1: Installing OpenAI

Step 1 is to install and import `openai`:

```
try:
  import openai
except:
  !pip install openai
  import openai
```

Restart the runtime once the installation is complete and run the cell again to make sure import openai has been executed

```
import openai
```

You can also install wand to visualize the logs:

```
try:
  import wandb
except:
  !pip install wandb
  import wandb
```

We will now enter the API key

Step 2: Entering the API key

Step 2 is to enter your key:

```
openai.api_key="[YOUR_KEY]"
```

Step 3: Activating OpenAI's data preparation module

First, load your file. In this section, load `kantgpt.csv`. Now, `kantgpt.csv`. is a raw unstructured file. OpenAI has an inbuilt data cleaner that will ask questions at each step.

OpenAI detects that the file is a CSV file and will convert it to a `JSONL` file. `JSONL` contains lines in plain structured text.

OpenAI tracks all the changes we approve:

```
Based on the analysis we will perform the following actions:
- [Necessary] Your format 'CSV' will be converted to 'JSONL'
- [Necessary] Remove 27750 rows with empty completions
- [Recommended] Remove 903 duplicate rows [Y/n]: y
- [Recommended] Add a suffix separator ' ->' to all prompts [Y/n]: y
- [Recommended] Remove prefix 'completion:' from all completions [Y/n]: y
- [Recommended] Add a suffix ending '\n' to all completions [Y/n]: y
- [Recommended] Add a whitespace character to the beginning of the
completion [Y/n]: y
```

OpenAI saves the converted file to `kantgpt_prepared.jsonl`.

We are ready to fine-tune GPT-3.

Fine-tuning GPT-3

You can split the notebook into two separate notebooks: one for data preparation and one for fine-tuning.

Step 4: Creating an OS environment

Step 4 in the fine-tuning process creates an os environment for the API key:

```
import openai
import os
os.environ['OPENAI_API_KEY'] =[YOUR_KEY]
print(os.getenv('OPENAI_API_KEY'))
```

Step 5: Fine-tuning OpenAI's Ada engine

Step 5 triggers fine-tuning the OpenAI Ada engine with the JSONL file that was saved after data preparation:

```
!openai api fine_tunes.create -t "kantgpt_prepared.jsonl" -m "ada"
```

OpenAI has many requests.

If your steam is interrupted, OpenAI will indicate the instruction to continue fine-tuning. Execute `fine_tunes.follow` instruction:

```
!openai api fine_tunes.follow -i [YOUR_FINE_TUNE]
```

Step 6: Interacting with the fine-tuned model

Step 6 is interacting with the fine-tuned model. The prompt is a sequence that is close to what *Immanuel Kant* might say:

```
!openai api completions.create -m ada:[YOUR_MODEL INFO] "Several concepts
are a priori such as"
```

The instruction to run a completion task with `[YOUR_MODEL INFO]` is often displayed by OpenAI at the end of your fine-tune task. You can copy and paste it in a cell(add `"!"` to run the command line) or insert your `[YOUR_MODEL INFO]` in the following cell.

The completion is quite convincing:

```
Several concepts are a priori such as the term  freedom and the concept of
_free will_.substance
```

We have fine-tuned GPT-3, which shows the importance of understanding transformers and designing AI pipelines with APIs. Let's see how this changes the role of AI specialists.

The role of an Industry 4.0 AI specialist

In a nutshell, the role of an Industry 4.0 developer is to become a cross-disciplinary AI guru. Developers, data scientists, and AI specialists will progressively learn more about linguistics, business goals, subject matter expertise, and more. An Industry 4.0 AI specialist will guide teams with practical cross-disciplinary knowledge and experience.

Human experts are mandatory in three domains when implementing transformers:

- **Morals and ethics**

  An Industry 4.0 AI guru ensures moral and ethical practices are enforced when implementing humanlike transformers. European regulations, for example, are strict and require that automated decisions be explained to the users when necessary. The US has anti-discrimination laws to protect citizens from automated bias.

- **Prompts and responses**

  Users and UI developers will need Industry 4.0 AI gurus to explain how to create the right prompts for NLP tasks, show a transformer model how to do a task, and verify the response.

- **Quality control and understanding the model**

What happens when the model does not behave as expected even after tweaking its hyperparameters? We will go deeper into such issues in *Chapter 14*, *Interpreting Black Box Transformer Models*.

Initial conclusions

The initial conclusions can be divided into two categories: facts and fiction.

One fact is that OpenAI has one of the most powerful NLP services in the world. Other facts include:

- OpenAI engines are powerful zero-shot engines that require no hunting for all kinds of transformer models, no pre-training, and no fine-tuning
- The supercomputers used to train the models are unique
- If a prompt is well designed, we can get surprisingly accurate responses
- Implementing the NLP tasks in this chapter only required a copy and paste action that any software beginner can perform

Many people believe AI will replace data scientists and AI specialists. Is that true? Before answering that question, first, ask yourself the following questions about the examples we ran in this chapter:

- How do we know if a sentence is incorrect?
- How do we know an answer is correct without us humans reading and confirming this?
- How did the engine know it was a grammar correction task?
- If a response is incorrect, how can we understand what happened to help improve the prompt or revert to manual mode in a well-designed human interface?

The truth is that humans will need to intervene to answers these questions manually, with rule bases, quality controlled automated pipelines, and many other tools.

The facts are convincing. Running an NLP task with a transformer requires little development in many cases.

Humans are still required. OpenAI engines are not there to replace humans but to help them perform more high-level gratifying tasks. You can now fly a jet without having to build it!

We need to answer the exciting questions we brought up in this section. So let's now explore your new fascinating Industry 4.0 role on a wonderful path into the future of AI!

Let's sum up the chapter and move on to the next exploration!

Summary

In this chapter, we discovered the new era of transformer models training billions of parameters on supercomputers. OpenAI's GPT models are taking NLU beyond the reach of most NLP development teams.

We saw how a GPT-3 zero-shot model performs many NLP tasks through an API and even directly online without an API. The online version of Google Translate has already paved the way for mainstream online usage of AI.

We explored the design of GPT models, which are all built on the original transformer's decoder stack. The masked attention sub-layer continues the philosophy of left-to-right training. However, the sheer power of the calculations and the subsequent self-attention sub-layer makes it highly efficient.

We then implemented a 345M parameter GPT-2 model with TensorFlow. The goal was to interact with a trained model to see how far we could get with it. We saw that the context provided conditioned the outputs. However, it did not reach the results expected when entering a specific input from the `Kant` dataset.

We trained a 117M parameter GPT-2 model on a customized dataset. The interactions with this relatively small trained model produced fascinating results.

We ran NLP tasks online with OpenAI's API and fine-tuned a GPT-3 model. This chapter showed that the fully pretrained transformers and their engines can automatically accomplish many tasks with little help from engineers.

Does this mean that users will not need AI NLP developers, data scientists, and AI specialists anymore in the future? Instead, will users simply upload the task definition and input text to cloud transformer models and download the results?

No, it doesn't mean that at all. Industry 4.0 data scientists and AI specialists will evolve into pilots of powerful AI systems. They will be increasingly necessary to ensure the inputs are ethical and secure. These modern-age AI pilots will also understand how transformers are built and adjust the hyperparameters of an AI ecosystem.

In the next chapter, *Applying Transformers to Legal and Financial Documents for AI Text Summarization*, we will take transformer models to their limits as multi-task models and explore new frontiers.

Questions

1. A zero-shot method trains the parameters once. (True/False)
2. Gradient updates are performed when running zero-shot models. (True/False)
3. GPT models only have a decoder stack. (True/False)
4. It is impossible to train a 117M GPT model on a local machine. (True/False)
5. It is impossible to train the GPT-2 model with a specific dataset. (True/False)
6. A GPT-2 model cannot be conditioned to generate text. (True/False)
7. A GPT-2 model can analyze the context of an input and produce completion content. (True/False)
8. We cannot interact with a 345M-parameter GPT model on a machine with less than 8 GPUs. (True/False)
9. Supercomputers with 285,000 CPUs do not exist. (True/False)
10. Supercomputers with thousands of GPUs are game-changers in AI. (True/False)

References

- OpenAI and GPT-3 engines: `https://beta.openai.com/docs/engines/engines`
- `BertViz` GitHub Repository by *Jesse Vig*: `https://github.com/jessevig/bertviz`
- OpenAI's supercomputer: `https://blogs.microsoft.com/ai/openai-azure-supercomputer/`
- *Ashish Vaswani*, *Noam Shazeer*, *Niki Parmar*, *Jakob Uszkoreit*, *Llion Jones*, *Aidan N. Gomez*, *Lukasz Kaiser*, *Illia Polosukhin*, 2017, *Attention is All You Need*: `https://arxiv.org/abs/1706.03762`
- *Alec Radford*, *Karthik Narasimhan*, *Tim Salimans*, *Ilya Sutskever*, 2018, *Improving Language Understanding by Generative Pre-Training*: `https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf`
- *Jacob Devlin*, *Ming-Wei Chang*, *Kenton Lee*, and *Kristina Toutanova*, 2019, *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*: `https://arxiv.org/abs/1810.04805`

- *Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever*, 2019, *Language Models are Unsupervised Multitask Learners*: `https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf`
- *Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplany, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, Dario Amodei*, 2020, *Language Models are Few-Shot Learners*: `https://arxiv.org/abs/2005.14165`
- *Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, Samuel R. Bowman*, 2019, *SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems*: `https://w4ngatang.github.io/static/papers/superglue.pdf`
- *Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, Samuel R. Bowman*, 2019, *GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding*: `https://arxiv.org/pdf/1804.07461.pdf`
- OpenAI GPT-2 GitHub Repository: `https://github.com/openai/gpt-2`
- N. Shepperd's GitHub Repository: `https://github.com/nshepperd/gpt-2`
- Common Crawl data: `https://commoncrawl.org/big-picture/`

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the authors:

`https://www.packt.link/Transformers`

8

Applying Transformers to Legal and Financial Documents for AI Text Summarization

We explored the architecture training, fine-tuning, and usage of several transformer ecosystems during the first seven chapters. In *Chapter 7, The Rise of Suprahuman Transformers with GPT-3 Engines*, we discovered that OpenAI has begun to experiment with zero-shot models that require no fine-tuning, no development, and can be implemented in a few lines.

The underlying concept of such an evolution relies on how transformers strive to teach a machine how to understand a language and express itself in a human-like manner. Thus, we have gone from training a model to teaching languages to machines.

Raffel et al. (2019) designed a transformer meta-model based on a simple assertion: every NLP problem can be represented as a text-to-text function. Every type of NLP task requires some kind of text context that generates some form of text response.

A text-to-text representation of any NLP task provides a unique framework to analyze a transformer's methodology and practice. The idea is for a transformer to learn a language through transfer learning during the training and fine-tuning phases with a text-to-text approach.

Raffel et al. (2019) named this approach a **T**ext-**T**o-**T**ext **T**ransfer **T**ransformer. The 5 Ts became **T5**, and a new model was born.

We will begin this chapter by going through the concepts and architecture of the T5 transformer model. We will then apply T5 to summarizing documents with Hugging Face models.

Finally, we will transpose the text-to-text approach to the show-and-context process of GPT-3 engine usage. The mind-blowing, though not perfect, zero-shot responses exceed anything a human could imagine.

This chapter covers the following topics:

- Text-to-text transformer models
- The architecture of T5 models
- T5 methodology
- The evolution of transformer models from training to learning
- Hugging Face transformer models
- Implementing a T5 model
- Summarizing a legal text
- Summarizing a financial text
- The limits of transformer models
- GPT-3 usage

Our first step will be to explore the text-to-text methodology defined by *Raffel* et al. (2019).

Designing a universal text-to-text model

Google's NLP technical revolution started with *Vaswani* et al. (2017), the original Transformer, in 2017. *Attention is All You Need* toppled 30+ years of artificial intelligence belief in RNNs and CNNs applied to NLP tasks. It took us from the stone age of NLP/NLU to the 21st century in a long-overdue evolution.

Chapter 7, The Rise of Suprahuman Transformers with GPT-3 Engines, summed up a second revolution that boiled up and erupted between Google's *Vaswani* et al. (2017) original Transformer and OpenAI's *Brown* et al. (2020) GPT-3 transformers. The original Transformer was focused on performance to prove that attention was all we needed for NLP/NLU tasks.

OpenAI's second revolution, through GPT-3, focused on taking transformer models from fine-tuned pretrained models to few-shot trained models that required no fine-tuning. The second revolution was to show that a machine can learn a language and apply it to downstream tasks as we humans do.

It is essential to perceive those two revolutions to understand what T5 models represent. The first revolution was an attention technique. The second revolution was to teach a machine to understand a language (NLU) and then let it solve NLP problems as we do.

In 2019, Google was thinking along the same lines as OpenAI about how transformers could be perceived beyond technical considerations and take them to an abstract level of natural language understanding.

These revolutions became disruptive. It was time to settle down, forget about source code and machine resources, and analyze transformers at a higher level.

Raffel et al. (2019) designed a conceptual text-to-text model and then implemented it.

Let's go through this representation of the second transformer revolution: abstract models.

The rise of text-to-text transformer models

Raffel et al. (2019) set out on a journey as pioneers with one goal: *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. The Google team working on this approach emphasized that it would not modify the original Transformer's fundamental architecture from the start.

At that point, *Raffel* et al. (2019) wanted to focus on concepts, not techniques. Therefore, they showed no interest in producing the latest transformer model as we often see a so-called silver bullet transformer model with *n* parameters and layers. This time, the T5 team wanted to find out how good transformers could be at understanding a language.

Humans learn a language and then apply that knowledge to a wide range of NLP tasks through transfer learning. The core concept of a T5 model is to find an abstract model that can do things like us.

When we communicate, we always start with a sequence (A) followed by another sequence (B). B, in turn, becomes the start sequence leading to another sequence, as shown in *Figure 8.1*:

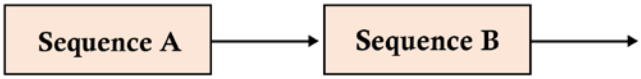


Figure 8.1: A sequence-to-sequence representation of communication

We also communicate through music with organized sounds. We communicate through dancing with organized body movements. We express ourselves through painting with coordinated shapes and colors.

We communicate through language with a word or a group of words we call "text." When we try to understand a text, we pay attention to all of the words in the sentence in *all* directions. We try to measure the importance of each term. When we do not understand a sentence, we focus on a word and *query* the rest of the *keywords* in the sentence to determine their *values* and the *attention* we must pay to them. This defines the attention layers of transformers.

Take a few seconds and let this sink in. It seems deceptively simple, right? Yet, it took 35+ years to topple the old beliefs surrounding RNNs, CNNs, and the thought process accompanying them!

It is quite fascinating to watch T5 learn, progress, and even help us think better sometimes!

The technical revolution of attention layers that simultaneously attend to all of the tokens in a sequence led to the T5 conceptual revolution.

The T5 model can be summed up as a **T**ext-**T**o-**T**ext **T**ransfer **T**ransformer. Thus, every NLP task is expressed as a text-to-text problem to solve.

A prefix instead of task-specific formats

Raffel et al. (2019) still had one problem to solve: unifying task-specific formats. The idea was to find a way to have one input format for every task submitted to the transformer. That way, the model parameters would be trained for all types of tasks with one text-to-text format.

The Google T5 team came up with a simple solution: adding a prefix to an input sequence. We would need thousands of additional vocabularies in many languages without the invention of the *prefix* by some long-forgotten genius. For example, we would need to find words to describe prepayment, prehistoric, Precambrian, and thousands of other words if we did not use "pre" as a prefix.

Raffel et al. (2019) proposed to add a *prefix* to an input sequence. A T5 prefix is not just a tag or indicator like `[CLS]` for classification in some transformer models. Instead, a T5 prefix contains the essence of a task a transformer needs to solve. A prefix conveys meaning as in the following examples, among others:

- `translate English to German: + [sequence]` for translations, as we did in *Chapter 6, Machine Translation with the Transformer*
- `cola sentence: + [sequence]` for *The Corpus of Linguistic Acceptability (CoLA)*, as we used in *Chapter 3, Fine-Tuning BERT Models*, when we fine-tuned a BERT transformer model

- `stsb sentence 1:+[sequence]` for semantic textual similarity benchmarks. Natural language inferences and entailment are similar problems, as described in *Chapter 5*, *Downstream NLP Tasks with Transformers*
- `summarize + [sequence]` for text summarization problems, as we will solve in the *Text summarization with T5* section of this chapter

We've now obtained a unified format for a wide range of NLP tasks, expressed in *Figure 8.2*:

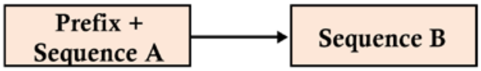


Figure 8.2: Unifying the input format of a transformer model

The unified input format leads to a transformer model that produces a result sequence no matter which problem it has to solve in the **T5**. The input and output of many NLP tasks have been unified, as shown in *Figure 8.3*:

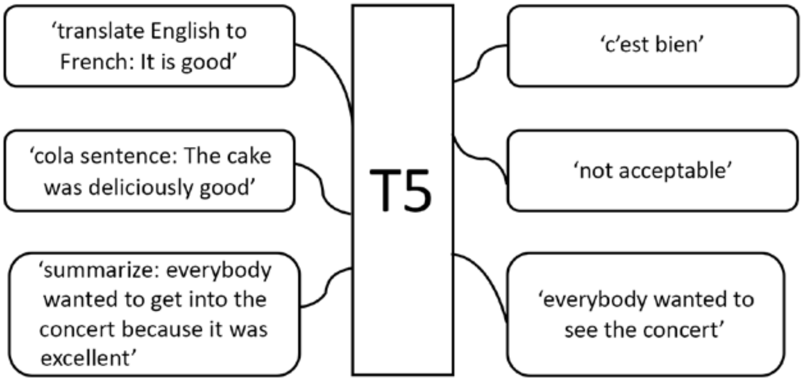

'translate English to French: It is good'
'cola sentence: The cake was deliciously good'
'summarize: everybody wanted to get into the concert because it was excellent'
T5
'c'est bien'
'not acceptable'
'everybody wanted to see the concert'

Figure 8.3: The T5 text-to-text framework

The unification process makes it possible to use the same model, hyperparameters, and optimizer for a wide range of tasks.

We have gone through the standard text-to-text input-output format. Let's now look at the architecture of the T5 transformer model.

The T5 model

Raffel et al. (2019) focused on designing a standard input format to obtain text output. The Google T5 team did not want to try new architectures derived from the original Transformer, such as BERT-like encoder-only layers or GPT-like decoder-only layers. Instead, the team focused on defining NLP tasks in a standard format.

They chose to use the original Transformer model we defined in *Chapter 2*, *Getting Started with the Architecture of the Transformer Model*, as we can see in *Figure 8.4*:

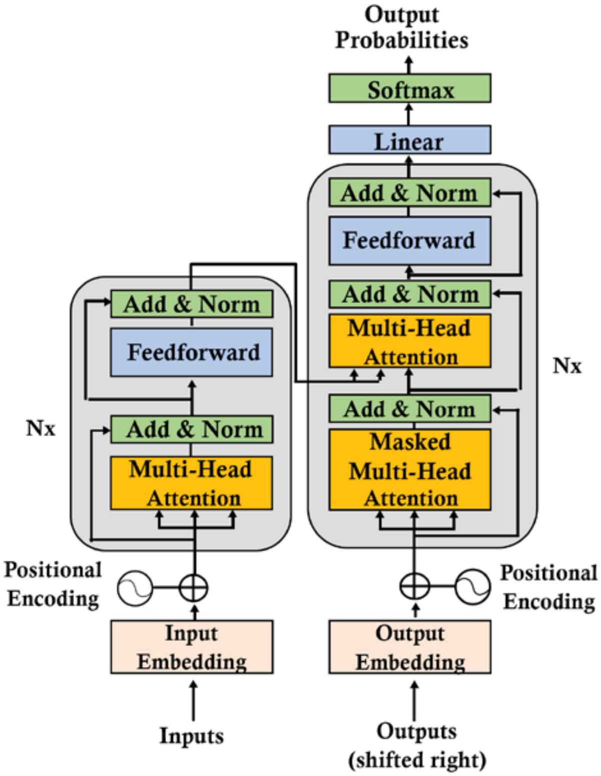


Figure 8.4: The original Transformer model used by T5

Raffel et al. (2019) kept most of the original Transformer architecture and terms. However, they emphasized some key aspects. Also, they made some slight vocabulary and functional changes. The following list contains some of the main aspects of the T5 model:

- The encoder and decoder remain in the model. The encoder and decoder layers become "blocks," and the sublayers become "subcomponents" containing a self-attention layer and a feedforward network. The use of the word "blocks" and "subcomponents" in a LEGO®-like language allows you to assemble "blocks," pieces, and components to build your model. Transformer components are standard building blocks you can assemble in many ways. You can understand any transformer model once you understand the basic building blocks we went through in *Chapter 2*, *Getting Started with the Architecture of the Transformer Model*.
- Self-attention is "order-independent," meaning it performs operations on sets, as we saw in *Chapter 2*. Self-attention uses dot products of matrices, not recurrence. It explores the relationship between each word and the others in a sequence. Positional encoding is added to the word's embedding before making the dot products.
- The original Transformer applied sinusoidal and cosine signals to the Transformer. Or it used learned position embeddings. T5 uses relative position embeddings instead of adding arbitrary positions to the input. In T5, positional encoding relies on an extension of self-attention to make comparisons between pairwise relationships. For more, see *Shaw* et al. (2018) in the *References* section of this chapter.
- Positional embeddings are shared and re-evaluated through all the layers of the model.

We have defined the standardization of the input of the T5 transformer model through the text-to-text approach.

Let's now use T5 to summarize documents.

Text summarization with T5

NLP summarizing tasks extract succinct parts of a text. This section will start by presenting the Hugging Face resources we will use in this chapter. Then we will initialize a T5-large transformer model. Finally, we will see how to use T5 to summarize any document, including legal and corporate documents.

Let's begin by introducing Hugging Face's framework.

Hugging Face

Hugging Face designed a framework to implement Transformers at a higher level. We used Hugging Face to fine-tune a BERT model in *Chapter 3*, *Fine-Tuning BERT Models*, and train a RoBERTa model in *Chapter 4*, *Pretraining a RoBERTa Model from Scratch*.

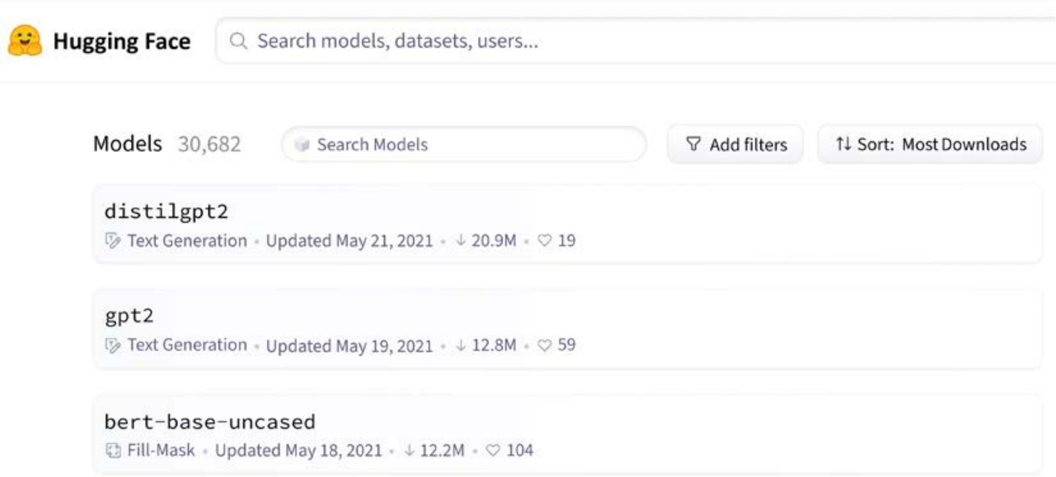
To expand our knowledge, we needed to explore other approaches, such as Trax, in *Chapter 6*, *Machine Translation with the Transformer*, and OpenAI's models, in *Chapter 7*, *The Rise of Suprahuman Transformers with GPT-3 Engines*. This chapter will use Hugging Face's framework again and explain more about the online resources. We end the chapter using the unique potential of a GPT-3 engine.

Hugging Face provides three primary resources within its framework: models, datasets, and metrics.

Hugging Face transformer resources

In this subsection, we will choose the T5 model that we will be implementing in this chapter.

A wide range of models can be found on the Hugging Face models page, as we can see in *Figure 8.5*:



Models 30,682

distilgpt2
Text Generation · Updated May 21, 2021 · ↓ 20.9M · ♡ 19

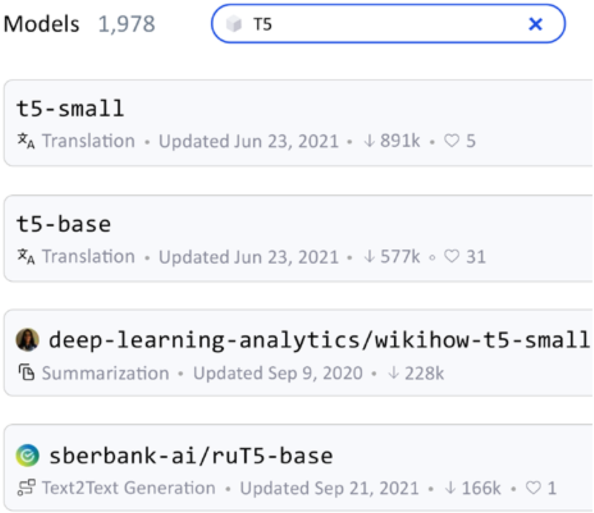
gpt2
Text Generation · Updated May 19, 2021 · ↓ 12.8M · ♡ 59

bert-base-uncased
Fill-Mask · Updated May 18, 2021 · ↓ 12.2M · ♡ 104

Figure 8.5: Hugging Face models

On this page, `https://huggingface.co/models`, we can search for a model. In our case, we are looking for **t5-large**, a model we can smoothly run in Google Colaboratory.

We first type T5 to search for a T5 model and obtain a list of T5 models we can choose from:



Models 1,978 T5

t5-small
Translation • Updated Jun 23, 2021 • ↓ 891k • ♡ 5

t5-base
Translation • Updated Jun 23, 2021 • ↓ 577k • ♡ 31

deep-learning-analytics/wikihow-t5-small
Summarization • Updated Sep 9, 2020 • ↓ 228k

sberbank-ai/ruT5-base
Text2Text Generation • Updated Sep 21, 2021 • ↓ 166k • ♡ 1

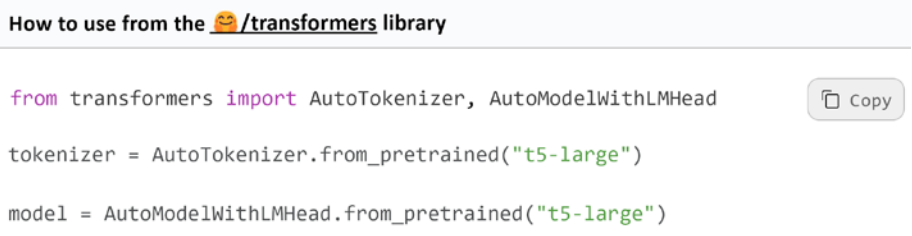
Figure 8.6: Searching for a T5 model

We can see that several of the original T5 transformers are available, among which are:

- **base**, which is the baseline model. It was designed to be similar to the $\text{BERT}_{\text{BASE}}$ with 12 layers and around 220 million parameters
- **small**, which is a smaller model with 6 layers and 60 million parameters
- **large** is designed to be similar to $\text{BERT}_{\text{LARGE}}$ with 12 layers and 770 million parameters
- **3B** and **11B** use 24-layer encoders and decoders with around 2.8 billion and 11 billion parameters

For more on the description of $\text{BERT}_{\text{BASE}}$ and $\text{BERT}_{\text{LARGE}}$, you can take a few minutes now or later to review these models in *Chapter 3*, *Fine-Tuning BERT Models*.

In our case, we select **t5-large**:



How to use from the 🤗/transformers library

```
from transformers import AutoTokenizer, AutoModelWithLMHead

tokenizer = AutoTokenizer.from_pretrained("t5-large")

model = AutoModelWithLMHead.from_pretrained("t5-large")
```

Figure 8.7: How to use a Hugging Face model

Figure 8.7 shows how to use the model in the code we will write. We can also look into the list of files in the model and the basic configuration file. We will look into the configuration file when we initialize the model in the *Initializing the T5-large transformer model* section of this chapter.

Hugging Face also provides datasets and metrics:

- The datasets can be used to train and test your models: `https://huggingface.co/datasets`
- The metrics resources can be used to measure the performance of your models: `https://huggingface.co/metrics`

Datasets and metrics are a classical aspect of NLP. In this chapter, we will not implement these datasets or metrics. Instead, we will focus on how to implement any text to summarize.

Let's start by initializing the T5 transformer model.

Initializing the T5-large transformer model

In this subsection, we will initialize a T5-large model. Open the following notebook, `Summarizing_Text_with_T5.ipynb`, which you will find in the directory of this chapter on GitHub.

Let's get started with T5!

Getting started with T5

In this subsection, we will install Hugging Face's framework and then initialize a T5 model.

We will first install Hugging Face's transformers:

```
!pip install transformers
```

Note: Hugging Face transformers continually evolve, updating libraries and modules to adapt to the market. If the default version doesn't work, you might have to pin one with `!pip install transformers==[version that runs with the other functions in the notebook]`.

We pinned version `0.1.94` of `sentencepiece` to keep the notebook using Hugging Face as stable as possible:

```
!pip install sentencepiece==0.1.94
```

Hugging Face has a GitHub repository that can be cloned. However, Hugging Face's framework provides a range of high-level transformer functions we can implement.

We can choose to display the architecture of the model or not when we initialize the model:

```
display_architecture=False
```

If we set `display_architecture` to `True`, the structure of the encoder layers, decoder layers, and feedforward sublayers will be displayed.

The program now imports `torch` and `json`:

```
import torch
import json
```

Working on transformers means being open to the many transformer architectures and frameworks that research labs share with us. Also, I recommend using PyTorch and TensorFlow as much as possible to get used to both environments. What matters is the level of abstraction of the transformer model (specific-task models or zero-shot models) and its overall performance.

Let's import the tokenizer, generation, and configuration classes:

```
from transformers import T5Tokenizer, T5ForConditionalGeneration, T5Config
```

We will use the `T5-large` model here, but you can select other T5 models in the Hugging Face list we went through in this chapter's *Hugging Face* section.

We will now import the `T5-large` conditional generation model to generate text and the T5-large tokenizer:

```
model = T5ForConditionalGeneration.from_pretrained('t5-large')
tokenizer = T5Tokenizer.from_pretrained('t5-large')
```

Initializing a pretrained tokenizer only takes one line. However, nothing proves that the tokenized dictionary contains all the vocabulary we need. We will investigate the relation between tokenizers and datasets in *Chapter 9*, *Matching Tokenizers and Datasets*.

The program now initializes `torch.device` with `'cpu.'` A CPU is enough for this notebook. The `torch.device` object is the device on which torch tensors will be allocated:

```
device = torch.device('cpu')
```

We are ready to explore the architecture of the T5 model.

Exploring the architecture of the T5 model

In this subsection, we will explore the architecture and configuration of a T5-large model.

If `display_architecture==true`, we can see the configuration of the model:

```
if display_architecture==True:
  print(model.config)
```

For example, we can see the basic parameters of the model:

```
.../...
"num_heads": 16,
"num_layers": 24,
.../...
```

The model is a T5 transformer with 16 heads and 24 layers.

We can also see the text-to-text implementation of T5, which adds a *prefix* to an input sentence to trigger the task to perform. The *prefix* makes it possible to represent a wide range of tasks in a text-to-text format without modifying the model's parameters. In our case, the prefix is `summarization`:

```
"task_specific_params": {
    "summarization": {
      "early_stopping": true,
      "length_penalty": 2.0,
      "max_length": 200,
      "min_length": 30,
      "no_repeat_ngram_size": 3,
      "num_beams": 4,
      "prefix": "summarize: "
    },
```

We can see that T5:

- Implements the *beam search* algorithm, which will expand the four most significant text completion predictions
- Applies early stopping when `num_beam` sentences are completed per batch
- Makes sure not to repeat ngrams equal to `no_repeat_ngram_size`
- Controls the length of the samples with `min_length` and `max_length`
- Applies a length penalty

Another interesting parameter is the vocabulary size:

```
"vocab_size": 32128
```

Vocabulary size is a topic in itself. Too much vocabulary will lead to sparse representations. On the other hand, too little vocabulary will distort the NLP tasks. We will explore this further in *Chapter 9*, *Matching Tokenizers and Datasets*.

We can also see the details of the transformer stacks by simply printing the `model`:

```
if(display_architecture==True):
  print(model)
```

For example, we can peek inside a block (`layer`) of the encoder stack (numbered from `0` to 23):

```
(12): T5Block(
        (layer): ModuleList(
          (0): T5LayerSelfAttention(
            (SelfAttention): T5Attention(
              (q): Linear(in_features=1024, out_features=1024, bias=False)
              (k): Linear(in_features=1024, out_features=1024, bias=False)
              (v): Linear(in_features=1024, out_features=1024, bias=False)
              (o): Linear(in_features=1024, out_features=1024, bias=False)
            )
```

```
      (layer_norm): T5LayerNorm()
      (dropout): Dropout(p=0.1, inplace=False)
     )
     (1): T5LayerFF(
      (DenseReluDense): T5DenseReluDense(
       (wi): Linear(in_features=1024, out_features=4096,
bias=False)
       (wo): Linear(in_features=4096, out_features=1024,
bias=False)
       (dropout): Dropout(p=0.1, inplace=False)
      )
      (layer_norm): T5LayerNorm()
      (dropout): Dropout(p=0.1, inplace=False)
     )
    )
   )
```

We can see that the model runs operations on `1,024` features for the attention sublayers and `4,096` for the inner calculations of the feedforward network sublayer that will produce outputs of `1,024` features. The symmetrical structure of transformers is maintained through all of the layers.

You can take a few minutes to go through the encoder stacks, the decoder stacks, the attention sublayers, and the feedforward sublayers.

You can also choose to select a specific aspect of the model by only running the cells you wish:

```
if display_architecture==True:
 print(model.encoder)
if display_architecture==True:
 print(model.decoder)
if display_architecture==True:
 print(model.forward)
```

We have initialized the T5 transformer. Let's now summarize documents.

Summarizing documents with T5-large

This section will create a summarizing function that you can call with any text you wish to summarize. We will summarize legal and financial examples. Finally, we will define the limits of the approach.

We will first start by creating a summarization function.

Creating a summarization function

First, let's create a summarizing function named `summarize`. That way, we will just send the texts we want to summarize to our function. The function takes two parameters. The first parameter is `preprocess_text`, the text to summarize. The second parameter is `ml`, the maximum length of the summarized text. Both parameters are variables you send to the function each time you call it:

```
def summarize(text,ml):
```

Hugging Face, among others, provides ready-to-use summarizing functions. However, I recommend learning how to build your own functions to customize this critical task when necessary.

The context text or ground truth is then stripped of the `\n` characters:

```
preprocess_text = text.strip().replace("\n","")
```

We then apply the innovative `T5` task *prefix* `summarize` to the input text:

```
t5_prepared_Text = "summarize: "+preprocess_text
```

The T5 model has a unified structure, whatever the task is through the *prefix + input sequence* approach. It may seem simple, but it takes NLP transformer models closer to universal training and zero-shot downstream tasks.

We can display the processed (stripped) and prepared text (task prefix):

```
print ("Preprocessed and prepared text: \n", t5_prepared_Text)
```

Simple right? Well, it took 35+ years to go from RNNs and CNNs to transformers. Then it took some of the brightest research teams in the world to go from transformers designed for specific tasks to multi-task models requiring little to no fine-tuning. Finally, the Google research team created a standard format for a transformer's input text that contained a prefix that indicates the NLP problem to solve. That is quite a feat!

The output displayed contains the preprocessed and prepared text:

```
Preprocessed and prepared text:
summarize: The United States Declaration of Independence
```

We can see the `summarize` prefix that indicates the task to solve.

The text is now encoded to token IDs and returns them as torch tensors:

```
tokenized_text = tokenizer.encode(t5_prepared_Text, return_tensors="pt").
to(device)
```

The encoded text is ready to be sent to the model to generate a summary with the parameters we described in the *Getting started with T5* section:

```
# Summarize
  summary_ids = model.generate(tokenized_text,
                                      num_beams=4,
                                      no_repeat_ngram_size=2,
                                      min_length=30,
                                      max_length=ml,
                                      early_stopping=True)
```

The number of beams remains the same as in the model we imported. However, `no_repeat_ngram_size` has been brought down to 2 instead of 3.

The generated output is now decoded with the `tokenizer`:

```
output = tokenizer.decode(summary_ids[0], skip_special_tokens=True)
return output
```

We imported, initialized, and defined the summarization function. Let's now experiment with the T5 model with a general topic.

A general topic sample

In this subsection, we will run a text written by *Project Gutenberg* through the T5 model. We will use the sample to run a test on our summarizing function. You can copy and paste any other text you wish or load a text by adding code. You can also load a dataset of your choice and call the summaries in a loop.

The program's goal in this chapter is to run a few samples to see how T5 works. The input text is the beginning of the *Project Gutenberg* e-book containing the *Declaration of Independence of the United States of America*:

```
text ="""
The United States Declaration of Independence was the first Etext
released by Project Gutenberg, early in 1971.  The title was stored
in an emailed instruction set which required a tape or diskpack be
```

```
hand mounted for retrieval.  The diskpack was the size of a large
cake in a cake carrier, cost $1500, and contained 5 megabytes, of
which this file took 1-2%.  Two tape backups were kept plus one on
paper tape.  The 10,000 files we hope to have online by the end of
2001 should take about 1-2% of a comparably priced drive in 2001.
"""
```

We then call our `summarize` function and send the text we want to summarize and the maximum length of the summary:

```
print("Number of characters:",len(text))
summary=summarize(text,50)
print ("\n\nSummarized text: \n",summary)
```

The output shows we sent 534 characters, the original text (ground truth) that was preprocessed, and the summary (prediction):

```
Number of characters: 534
Preprocessed and prepared text:
 summarize: The United States Declaration of Independence...

Summarized text:
 the united states declaration of independence was the first etext
published by project gutenberg, early in 1971. the 10,000 files we hope
to have online by the end of2001 should take about 1-2% of a comparably
priced drive in 2001. the united states declaration of independence was
the first Etext released by project gutenberg, early in 1971
```

Let's now use T5 for a more difficult summary.

The Bill of Rights sample

The following sample, taken from the *Bill of Rights*, is more difficult because it expresses the special rights of a person:

```
#Bill of Rights,V
text ="""
No person shall be held to answer for a capital, or otherwise infamous
crime,
unless on a presentment or indictment of a Grand Jury, except in cases
arising
 in the land or naval forces, or in the Militia, when in actual service
in time of War or public danger; nor shall any person be subject for
the same offense to be twice put in jeopardy of life or limb;
nor shall be compelled in any criminal case to be a witness against
himself,
nor be deprived of life, liberty, or property, without due process of law;
nor shall private property be taken for public use without just
compensation.
"""
print("Number of characters:",len(text))
summary=summarize(text,50)
print ("\n\nSummarized text: \n",summary)
```

Remember that transformers are stochastic algorithms, so the output might vary each time you run one. That being said, we can see that T5 did not really summarize the input text but simply shortened it:

```
Number of characters: 591
Preprocessed and prepared text:
 summarize: No person shall be held to answer..

Summarized text:
 no person shall be held to answer for a capital, or otherwise infamous
crime. except in cases arisingin the land or naval forces or in the
militia, when in actual service in time of war or public danger
```

This sample is significant because it shows the limits that any transformer model or other NLP model faces when faced with a text such as this one. We cannot just present samples that always work and make users believe that transformers have solved all of the NLP challenges we face, no matter how innovative they are.

Maybe we should have provided a longer text to summarize, used other parameters, used a larger model, or changed the structure of the T5 model. However, no matter how hard you try to summarize a complex text with an NLP model, you will always find documents that the model fails to summarize.

When a model fails on a task, we must be humble and admit it. The SuperGLUE human baseline is a difficult one to beat. We need to be patient, work harder, and improve transformer models until they can perform better than they do today. There is still room for a lot of progress.

Raffel et al. (2018) chose an appropriate title to describe their approach to T5: *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*.

Take the necessary time to experiment with examples of your own that you find in your legal documents. Explore the limits of transfer learning as a modern-day NLP pioneer! Sometimes you will discover exciting results, and sometimes you will find areas that need improvement.

Now, let's try a corporate law sample.

A corporate law sample

Corporate law contains many legal subtleties, making summarizing tasks quite tricky.

The input of this sample is an excerpt of the corporate law in the state of Montana, USA:

```
#Montana Corporate Law
#https://corporations.uslegal.com/state-corporation-law/montana-
corporation-law/#:~:text=Montana%20Corporation%20Law,carrying%20out%20
its%20business%20activities.
text ="""The law regarding corporations prescribes that a corporation can
be incorporated in the state of Montana to serve any lawful purpose.  In
the state of Montana, a corporation has all the powers of a natural person
for carrying out its business activities.  The corporation can sue and be
sued in its corporate name.  It has perpetual succession.  The corporation
can buy, sell or otherwise acquire an interest in a real or personal
property.  It can conduct business, carry on operations, and have offices
and exercise the powers in a state, territory or district in possession of
the U.S., or in a foreign country.  It can appoint officers and agents of
the corporation for various duties and fix their compensation.
The name of a corporation must contain the word "corporation" or its
abbreviation "corp."  The name of a corporation should not be deceptively
similar to the name of another corporation incorporated in the same state.
It should not be deceptively identical to the fictitious name adopted by a
foreign corporation having business transactions in the state.
The corporation is formed by one or more natural persons by executing and
filing articles of incorporation to the secretary of state of filing.  The
qualifications for directors are fixed either by articles of incorporation
or bylaws.  The names and addresses of the initial directors and purpose
of incorporation should be set forth in the articles of incorporation.
The articles of incorporation should contain the corporate name, the
number of shares authorized to issue, a brief statement of the character
of business carried out by the corporation, the names and addresses of
the directors until successors are elected, and name and addresses of
incorporators.  The shareholders have the power to change the size of
board of directors.
"""
```

```
print("Number of characters:",len(text))
summary=summarize(text,50)
print ("\n\nSummarized text: \n",summary)
```

The result is satisfying:

```
Number of characters: 1816
Preprocessed and prepared text:
 summarize: The law regarding the corporation prescribes that a
corporation...

Summarized text:
 a corporations can be incorporated in the state of Montana to serve
any lawful purpose. a corporation can sue and be sued in its corporate
name, and it has perpetual succession. it can conduct business, carry on
operations and have offices
```

This time, T5 found some of the essential aspects of the text to summarize. Take some time to incorporate samples of your own to see what happens. Play with the parameters to see if it affects the outcome.

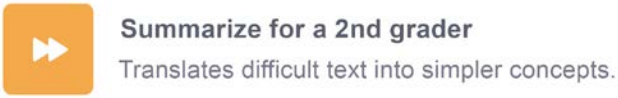
We have implemented T5 to summarize texts. It is time to experiment with summarization with an OpenAI GPT-3 engine.

Summarization with GPT-3

It was essential to understand the architecture of a T5 transformer. We will also see how GPT-3 engines behave on one of the texts. The goal is *not* to benchmark companies and models. The goal is for an Industry 4.0 AI Guru to have a broad knowledge of NLP.

First, go to `https://openai.com/` and sign up and sign in.

Then go to the examples page and select **Summarize for a 2nd grader**:



Summarize for a 2nd grader

Translates difficult text into simpler concepts.

Figure 8.8: GPT-3 examples page

A window will open, and we can enter our prompt.

We submit the text *T* of the corporate sample of the previous section to the GPT-3 model.

The prompt is $P = E + T + S$:

- E tells the model to make the explanation simple:

  ```
  My second grader asked me what this passage means:
  ```

- The text T is the same as in the previous section and is in quotes:

  ```
  """The law regarding corporations prescribes that a corporation can be
  incorporated in the state of Montana to serve any lawful purpose.  In the
  state of Montana, a corporation has all the powers of a natural person for
  carrying out its business activities.

  .../...

  The shareholders have the power to change the size of board of directors."""
  ```

- S shows the type of task that is expected:

  ```
  I rephrased it for him, in plain language a second grader can understand: """
  ```

The response produced by the GPT-3 Davinci engine is quite convincing:

```
A corporation is a business.

A corporation is a person.

Corporations can do everything people can do.

Corporations can be sued.

Corporations can buy, sell, or trade things.

A corporation has its own name that is different from the names of the people who
own it.

Corporations are formed by people who want to start a business, and they file
papers with the government to make it official.
```

We can keep T and modify E and S:

- E can be changed to:

  ```
  My MBA college student asked me what this passage means:
  ```

- S can be changed to:

  ```
  I rephrased it for him, in MBA college language :
  ```

The GPT-3 engine generated the response of the previous request but added useful information for a college student:

```
Corporations can have directors who are in charge of making decisions for the
business. The directors are elected by the shareholders. Shareholders can change
the size of the board of directors.
```

GPT-3 models are quite convincing and represent the rising power of Cloud AI. We will go deeper into summarizing prompts in *Chapter 16*, *The Emergence of Transformer-Driven Copilots*. However, there is more, much more, to explore before we do that.

Summary

In this chapter, we saw how the T5 transformer models standardized the input of the encoder and decoder stacks of the original Transformer. The original Transformer architecture has an identical structure for each block (or layer) of the encoder and decoder stacks. However, the original Transformer did not have a standardized input format for NLP tasks.

Raffel et al. (2018) designed a standard input for a wide range of NLP tasks by defining a text-to-text model. They added a prefix to an input sequence, indicating the NLP problem type to solve. This led to a standard text-to-text format. The **Text-To-Text Transfer Transformer (T5)** was born. We saw that this deceivingly simple evolution made it possible to use the same model and hyperparameters for a wide range of NLP tasks. The invention of T5 takes the standardization process of transformer models a step further.

We then implemented a T5 model that could summarize any text. We tested the model on texts that were not part of ready-to-use training datasets. We tested the model on constitutional and corporate samples. The results were interesting, but we also discovered some of the limits of transformer models, as predicted by *Raffel* et al. (2018).

Finally, we explored the tremendous power of a GPT-3 engine's methodology and calculation efficiency. Showing a transformer is a brilliant approach. Having one of the most powerful transformer engines in the world helps attain effective, though not always perfect, results.

The goal is not to benchmark companies and models but for an Industry 4.0 AI Guru to have a deep understanding of transformers.

In the next chapter, *Chapter 9*, *Matching Tokenizers and Datasets*, we will explore the limits of tokenizers and define methods to possibly improve NLP tasks.

Questions

1. T5 models only have encoder stacks like BERT models. (True/False)
2. T5 models have both encoder and decoder stacks. (True/False)
3. T5 models use relative positional encoding, not absolute positional encoding. (True/False)
4. Text-to-text models are only designed for summarization. (True/False)
5. Text-to-text models apply a prefix to the input sequence that determines the NLP task. (True/False)
6. T5 models require specific hyperparameters for each task. (True/False)
7. One of the advantages of text-to-text models is that they use the same hyperparameters for all NLP tasks. (True/False)
8. T5 transformers do not contain a feedforward network. (True/False)
9. Hugging Face is a framework that makes transformers easier to implement. (True/False)
10. OpenAI's transformer engines are game changers. (True/False)

References

- *Colin Raffel*, *Noam Shazeer*, *Adam Roberts*, *Katherine Lee*, *Sharan Narang*, *Michael Matena*, *Yanqi Zhou*, *Wei Li*, *Peter J. Liu*, 2019, *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*: `https://arxiv.org/pdf/1910.10683.pdf`
- *Ashish Vaswani*, *Noam Shazeer*, *Niki Parmar*, *Jakob Uszkoreit*, *Llion Jones*, *Aidan N. Gomez*, *Lukasz Kaiser*, *Illia Polosukhin*, 2017, *Attention is All You Need*: `https://arxiv.org/abs/1706.03762`
- *Peter Shaw*, *Jakob Uszkoreit*, and *Ashish Vaswani*, 2018, *Self-Attention with Relative Position Representations*: `https://arxiv.org/abs/1803.02155`
- Hugging Face Framework and Resources: `https://huggingface.co/`
- U.S. Legal, *Montana Corporate Laws*: `https://corporations.uslegal.com/state-corporation-law/montana-corporation-law/#:~:text=Montana%20Corporation%20Law,carrying%20out%20its%20business%20activities`
- *The Declaration of Independence of the United States of America* by Thomas Jefferson: `https://www.gutenberg.org/ebooks/1`
- *The United States Bill of Rights* by the United States: `https://www.gutenberg.org/ebooks/2`

Join our book’s Discord space

Join the book’s Discord workspace for a monthly *Ask me Anything* session with the authors:

```
https://www.packt.link/Transformers
```

9

Matching Tokenizers and Datasets

When studying transformer models, we tend to focus on the models' architecture and the datasets provided to train them. We have explored the original Transformer, fine-tuned a BERT-like model, trained a RoBERTa model, explored a GPT-3 model, trained a GPT-2 model, implemented a T5 model, and more. We have also gone through the main benchmark tasks and datasets.

We trained a RoBERTa tokenizer and used tokenizers to encode data. However, we did not explore the limits of tokenizers to evaluate how they fit the models we build. AI is data-driven. *Raffel* et al. (2019), like all the authors cited in this book, spent time preparing datasets for transformer models.

In this chapter, we will go through some of the limits of tokenizers that hinder the quality of downstream transformer tasks. Do not take pretrained tokenizers at face value. You might have a specific dictionary of words you use (advanced medical language, for example) with words not processed by a generic pretrained tokenizer.

We will start by introducing some tokenizer-agnostic best practices to measure the quality of a tokenizer. We will describe basic guidelines for datasets and tokenizers from a tokenization perspective.

Then, we will see the limits of tokenizers with a Word2Vec tokenizer to describe the problems we face with any tokenizing method. The limits will be illustrated with a Python program.

We will continue our investigation by running a GPT-2 model on a dataset containing specific vocabulary with unconditional and conditional samples.

We will go further and see the limits of byte-level BPE methods. We will build a Python program that displays the results produced by a GPT-2 tokenizer and go through the problems that occur during the data encoding process. This will show that the superiority of GPT-3 is not always necessary for common NLP analysis.

However, at the end of the chapter, we will probe a GPT-3 engine with a **Part-of-Speech** (**POS**) task to see how much the model understands and if a ready-to-use tokenized dictionary fits our needs.

This chapter covers the following topics:

- Basic guidelines to control the output of tokenizers
- Raw data strategies and preprocessing data strategies
- Word2Vec tokenization problems and limits
- Creating a Python program to evaluate Word2Vec tokenizers
- Building a Python program to evaluate the output of byte-level BPE algorithms
- Customizing NLP tasks with specific vocabulary
- Running unconditional and conditional samples with GPT-2
- Evaluating GPT-2 tokenizers

Our first step will be to explore the text-to-text methodology defined by *Raffel* et al. (2019).

Matching datasets and tokenizers

Downloading benchmark datasets to train transformers has many advantages. The data has been prepared, and every research lab uses the same references. Also, the performance of a transformer model can be compared to another model with the same data.

However, more needs to be done to improve the performance of transformers. Furthermore, implementing a transformer model in production requires careful planning and defining best practices.

In this section, we will define some best practices to avoid critical stumbling blocks.

Then we will go through a few examples in Python using cosine similarity to measure the limits of tokenization and encoding datasets.

Let’s start with best practices.

Best practices

Raffel et al. (2019) defined a standard text-to-text T5 transformer model. They also went further. They began destroying the myth of using raw data without preprocessing it first.

Preprocessing data reduces training time. Common Crawl, for example, contains unlabeled text obtained through web extraction. Non-text and markup have been removed from the dataset.

However, the Google T5 team found that much of the text obtained through Common Crawl did not reach the level of natural language or English. So they decided that datasets need to be cleaned before using them.

We will take the recommendations *Raffel* et al. (2019) made and apply corporate quality control best practices to the preprocessing and quality control phases. Among many other rules to apply, the examples described show the tremendous work required to obtain acceptable real-life project datasets.

Figure 9.1 lists some of the key quality control processes to apply to datasets:

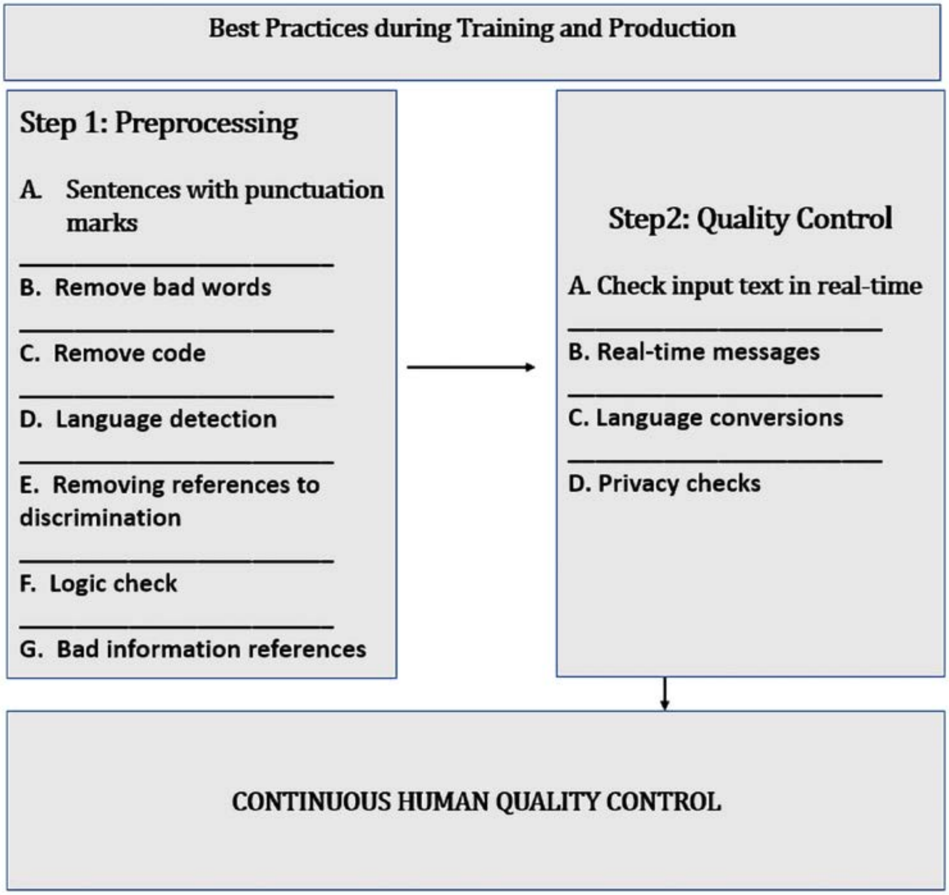

Best Practices during Training and Production
Step 1: Preprocessing
A. Sentences with punctuation marks
B. Remove bad words
C. Remove code
D. Language detection
E. Removing references to discrimination
F. Logic check
G. Bad information references
Step2: Quality Control
A. Check input text in real-time
B. Real-time messages
C. Language conversions
D. Privacy checks
CONTINUOUS HUMAN QUALITY CONTROL

Figure 9.1: Best practices for transformer datasets

As shown in *Figure 9.1*, quality control is divided into the preprocessing phase (*Step 1*) when training a transformer and quality control when the transformer is in production (*Step 2*).

Let's go through some of the main aspects of the preprocessing phase.

Step 1: Preprocessing

Raffel et al. (2019) recommended preprocessing datasets before training models on them, and I added some extra ideas.

Transformers have become language learners, and we have become their teachers. But to teach a machine-student a language, we must explain what proper English is, for example.

We need to apply some standard heuristics to datasets before using them:

- **Sentences with punctuation marks**

  The recommendation is to select sentences that end with punctuation marks such as a period or a question mark.

- **Remove bad words**

  Bad words should be removed. Lists can be found at the following site, for example: `https://github.com/LDNOOBW/List-of-Dirty-Naughty-Obscene-and-Otherwise-Bad-Words`.

- **Remove code**

  This is tricky because sometimes code is the content we are looking for. However, it is generally best to remove code from content for NLP tasks.

- **Language detection**

  Sometimes, websites contain pages with the default "lorem ipsum" text. It is necessary to make sure all of a dataset's content is in the language we wish. An excellent way to start is with `langdetect`, which can detect 50+ languages: `https://pypi.org/project/langdetect/`.

- **Removing references to discrimination**

  This is a must. My recommendation is to build a knowledge base with everything you can scrape on the web or from specific datasets you can get your hands on. *Suppress any form of discrimination*. You certainly want your machine to be ethical!

- **Logic check**

  It could be a good idea to run a trained transformer model on a dataset that performs **Natural Language Inferences** (**NLI**) to filter sentences that make no sense.

- **Bad information references**

  Eliminate text that refers to links that do not work, unethical websites, or persons. This is a tough job, but certainly worthwhile.

This list contains some of the primary best practices. However, more is required, such as filtering privacy law violations and other actions for specific projects.

Once a transformer is trained to learn proper English, we need to help it detect problems in the input texts in the production phase.

Step 2: Quality control

A trained model will behave like a person who learned a language. It will understand what it can and learn from input data. Input data should go through the same process as *Step 1: Preprocessing* and add new information to the training dataset. The training dataset, in turn, can become the knowledge base in a corporate project. Users will be able to run NLP tasks on the dataset and obtain reliable answers to questions, useful summaries of specific documents, and more.

We should apply the best practices described in *Step 1: Preprocessing* to real-time input data. For example, a transformer can be running on input from a user or an NLP task, such as summarizing a list of documents.

Transformers are the most powerful NLP models ever. This means that our ethical responsibility is heightened as well.

Let's go through some of the best practices:

- **Check input text in real time**

  Do not accept bad information. Instead, parse the input in real time and filter the unacceptable data (see *Step 1*).

- **Real-time messages**

  Store the rejected data along with the reason it was filtered so that users can consult the logs. Display real-time messages if a transformer is asked to answer an unfitting question.

- **Language conversions**

  You can convert rare vocabulary into standard vocabulary when it is possible. See *Case 4* of the *Word2Vec tokenization* section in this chapter. This is not always possible. When it is, it could represent a step forward.

- **Privacy checks**

  Whether you are streaming data into a transformer model or analyzing user input, private data must be excluded from the dataset and tasks unless authorized by the user or country the transformer is running in. It's a tricky topic. Consult a legal adviser when necessary.

We just went through some of the best practices. Let's now see why human quality control is mandatory.

Continuous human quality control

Transformers will progressively take over most of the complex NLP tasks. However, human intervention remains mandatory. We think social media giants have automized everything. Then we discover there are content managers that decide what is good or bad for their platform.

The right approach is to train a transformer, implement it, control the output, and feed the significant results back into the training set. Thus, the training set will continuously improve, and the transformer will continue to learn.

Figure 9.2 shows how continuous quality control will help the transformer's training dataset grow and increase its performance in production:

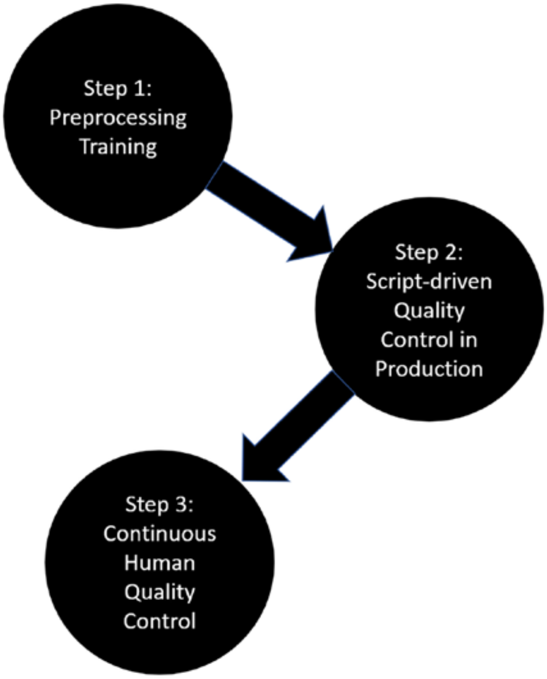


Figure 9.2: Continuous human quality control

We have gone through several best practices described by *Raffel* et al. (2019), and I have added some guidance based on of my experience in corporate AI project management.

Let’s go through a Python program with some examples of some of the limits encountered with tokenizers.

Word2Vec tokenization

As long as things go well, nobody thinks about pretrained tokenizers. It's like in real life. We can drive a car for years without thinking about the engine. Then, one day, our car breaks down, and we try to find the reasons to explain the situation.

The same happens with pretrained tokenizers. Sometimes the results are not what we expect. For example, some word pairs just don't fit together, as we can see in *Figure 9.3*:

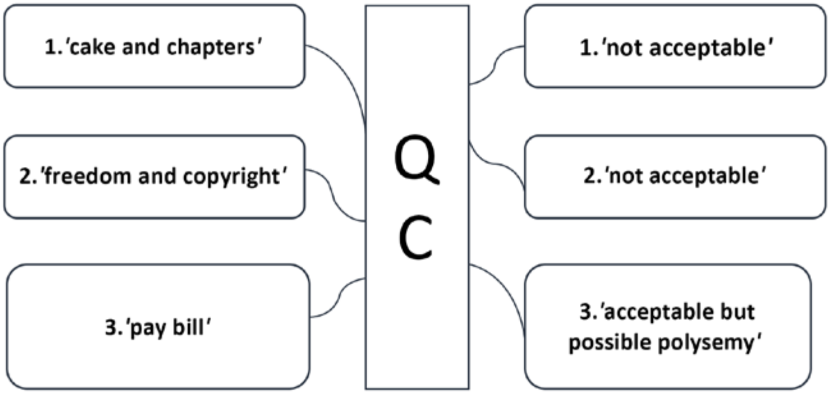


Figure 9.3: Word pairs that tokenizers miscalculated

The examples shown in *Figure 9.3* are drawn from the *American Declaration of Independence*, the *Bill of Rights*, and the *English Magna Carta*:

- `cake` and `chapters` do not fit together, although a tokenizer computed them as having a high value of cosine similarity.
- `freedom` refers to the freedom of speech, for example. `copyright` refers to the note written by the editor of the free ebook.
- `pay` and `bill` fit together in everyday English. `polysemy` is when a word can have several meanings. For example, `Bill` means an amount to pay but also refers to the `Bill of Rights`. The result is acceptable, but it may be pure luck.

Before continuing, let's take a moment to clarify some points. **QC** refers to **quality control**. In any strategic corporate project, QC is mandatory. The quality of the output will determine the survival of a critical project. If the project is not strategic, errors will sometimes be acceptable. In a strategic project, even a few errors imply a risk management audit's intervention to see if the project should be continued or abandoned.

From the perspectives of quality control and risk management, tokenizing datasets that are irrelevant (too many useless words or critical words missing) will confuse the embedding algorithms and produce "poor results." That is why in this chapter, I use the word "tokenizing" loosely, including some embedding because of the impact of one upon the other.

In a strategic AI project, "poor results" can be a single error with a dramatic consequence (especially in the medical sphere, airplane or rocket assembly, or other critical domains).

Open `Tokenizer.ipynb`, based on `positional_encoding.ipynb`, which we created in *Chapter 2, Getting Started with the Architecture of the Transformer Model*.

Results might vary from one run to another due to the stochastic nature of Word2Vec algorithms.

The prerequisites are installed and imported first:

```
#@title Pre-Requisistes
!pip install gensim==3.8.3
import nltk
nltk.download('punkt')
import math
import numpy as np
from nltk.tokenize import sent_tokenize, word_tokenize
import gensim
from gensim.models import Word2Vec
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings(action = 'ignore')
```

`text.txt`, our dataset, contains the *American Declaration of Independence*, the *Bill of Rights*, the *Magna Carta*, the works of Immanuel Kant, and other texts.

We will now tokenize `text.txt` and train a word2vec model:

```
#@title Word2Vec Tokenization
#'text.txt' file
sample = open("text.txt", "r")
s = sample.read()
# processing escape characters
```

```
f = s.replace("\n", " ")
data = []
# sentence parsing
for i in sent_tokenize(f):
  temp = []
  # tokenize the sentence into words
  for j in word_tokenize(i):
    temp.append(j.lower())
  data.append(temp)
# Creating Skip Gram model
model2 = gensim.models.Word2Vec(data, min_count = 1, size = 512,window =
5, sg = 1)
print(model2)
```

`window = 5` is an interesting parameter. It limits the *distance* between the current word and the predicted word in an input sentence. `sg = 1` means a skip-gram training algorithm is used.

The output shows that the size of the vocabulary is `10816`, the dimensionality of the embeddings is `512`, and the learning rate was set to `alpha=0.025`:

```
Word2Vec(vocab=10816, size=512, alpha=0.025)
```

We have a word representation model with embedding and can create a cosine similarity function named `similarity(word1,word2)`. We will send `word1` and `word2` to the function, which will return a cosine similarity value between them. The higher the value, the higher the similarity.

The function will first detect unknown words, `[unk]`, and display a message:

```
#@title Cosine Similarity
def similarity(word1,word2):
        cosine=False #default value
        try:
                a=model2[word1]
                cosine=True
        except KeyError:     #The KeyError exception is raised
                print(word1, ":[unk] key not found in dictionary")#False
implied
        try:
                b=model2[word2]#a=True implied
        except KeyError:       #The KeyError exception is raised
                cosine=False   #both a and b must be true
                print(word2, ":[unk] key not found in dictionary")
```

Cosine similarity will only be calculated if `cosine==True`, which means that both `word1` and `word2` are known:

```
        if(cosine==True):
                b=model2[word2]
                # compute cosine similarity
                dot = np.dot(a, b)
                norma = np.linalg.norm(a)
                normb = np.linalg.norm(b)
                cos = dot / (norma * normb)
                aa = a.reshape(1,512)
                ba = b.reshape(1,512)
                #print("Word1",aa)
                #print("Word2",ba)
                cos_lib = cosine_similarity(aa, ba)
                #print(cos_lib,"word similarity")

        if(cosine==False):cos_lib=0;
        return cos_lib
```

The function will return `cos_lib`, the computed value of cosine similarity.

We will now go through six cases. We will name `text.txt` the "dataset."

Let's begin with *Case 0*.

Case 0: Words in the dataset and the dictionary

The words `freedom` and `liberty` are in the dataset, and their cosine similarity can be computed:

```
#@title Case 0: Words in text and dictionary
word1="freedom";word2="liberty"
print("Similarity",similarity(word1,word2),word1,word2)
```

The similarity is limited to `0.79` because a lot of content was inserted from various texts to explore the limits of the function:

```
Similarity [[0.79085565]] freedom liberty
```

The similarity algorithm is not an iterative deterministic calculation. This section's results might change with the dataset's content, the dataset's size after another run, or the module's versions. If you run the cell 10 times, you may or may not obtain different values, such as in the following 10 runs.

In the following case, I obtained the same result 10 times with a Google Colab VM and a CPU:

```
Run 1: Similarity [[0.62018466]] freedom liberty
Run 2: Similarity [[0.62018466]] freedom liberty
...
Run 10: Similarity [[0.62018466]] freedom liberty
```

However, I did a "factory reset runtime" of the runtime menu in Google Colab. With a new VM and a CPU, I obtained:

```
Run 1: Similarity [[0.51549244]] freedom liberty
Run 2: Similarity [[0.51549244]] freedom liberty
...
Run 10: Similarity [[0.51549244]] freedom liberty
```

I performed another "factory reset runtime" of the runtime menu in Google Colab. I also activated the GPU. With a new VM and GPU, I obtained:

```
Run 1: Similarity [[0.58365834]] freedom liberty
Run 2: Similarity [[0.58365834]] freedom liberty
...
Run 10: Similarity [[0.58365834]] freedom liberty
```

The conclusion here is that stochastic algorithms are based on probabilities. It is good practice to run a prediction `n` times if necessary.

Let's now see what happens when a word is missing.

Case 1: Words not in the dataset or the dictionary

A missing word means trouble in many ways. In this case, we send `corporations` and `rights` to the similarity function:

```
#@title Word(s) Case 1: Word not in text or dictionary
word1="corporations";word2="rights"
print("Similarity",similarity(word1,word2),word1,word2)
```

The dictionary does not contain the word `corporations`:

```
corporations :[unk] key not found in dictionary
Similarity 0 corporations rights
```

Dead end! The word is an unknown `[unk]` token.

The missing word will provoke a chain of events and problems that distort the transformer model's output if the word is important. We will refer to the missing word as `unk`.

Several possibilities need to be checked, and questions answered:

- `unk` was in the dataset but was not selected to be in the tokenized dictionary.
- `unk` was not in the dataset, which is the case for the word `corporations`. This explains why it's not in the dictionary in this case.
- `unk` will now appear in production if a user sends an input to the transformer that contains the token and it is not tokenized.
- `unk` was not an important word for the dataset but is for the usage of the transformer.

The list of problems will continue to grow if the transformer produces terrible results in some cases. We can consider `0.8` as excellent performance for a transformer model for a specific downstream task during the training phase. But in real life, who wants to work with a system that's wrong 20% of the time:

- A doctor?
- A lawyer?
- A nuclear plant maintenance team?

`0.8` is satisfactory in a fuzzy environment like social media, in which many of the messages lack proper language structure anyway.

Now comes the worst part. Suppose an NLP team discovers this problem and tries to solve it with byte-level BPE, as we have been doing throughout this book. If necessary, take a few minutes and go back to *Chapter 4*, *Pretraining a RoBERTa Model from Scratch*, *Step 3: Training a tokenizer*.

The nightmare begins if a team only uses byte-level BPE to fix the problem:

- `unk` will be broken down into word pieces. For example, we could end up with `corporations` becoming `corp` + `o` + `ra` + `tion` + `s`. One or several of these tokens have a high probability of being found in the dataset.

- unk will become a set of sub-words represented by tokens that exist in the dataset but do not convey the original token's meaning.
- The transformer will train well, and nobody will notice that unk was broken into pieces and trained meaninglessly.
- The transformer might even produce excellent results and move its performance up from 0.8 to 0.9.
- Everybody will be applauding until a professional user applies an erroneous result in a critical situation. For example, in English, corp can mean corporation or corporal. This could create confusion and bad associations between corp and other words.

We can see that the standard of social media might be enough to use transformers for trivial topics. But in real-life corporate projects, it will take hard work to produce a pretrained tokenizer that matches the datasets. In real life, datasets grow every day with user inputs. User inputs become part of the datasets of models that should be trained and updated regularly.

For example, one way to ensure quality control can be through the following steps:

- Train a tokenizer with a byte-level BPE algorithm.
- Control the results with a program such as the one we will create in the *Controlling tokenized data* section of this chapter.
- Also, train a tokenizer with a Word2Vec algorithm, which will only be used for quality control, then parse the dataset, find the unk tokens, and store them in the database. Run queries to check if critical words are missing.

It might seem unnecessary to check the process in such detail, and you might be tempted to rely on a transformer's ability to make inferences with unseen words.

However, I recommend running several different quality control methods in a strategic project with critical decision making. For example, in a legal summary of a law, one word can make the difference between losing and winning a case in court. In an aerospace project (airplanes, rockets), there is a 0 error tolerance standard.

The more quality control processes you run, the more reliable your transformer solution will be.

We can see that it takes a lot of legwork to obtain a reliable dataset! Every paper written on transformers refers in one way or another to the work it took to produce acceptable datasets.

Noisy relationships also cause problems.

Case 2: Noisy relationships

In this case, the dataset contained the words `etext` and `declaration`:

```
#@title Case 2: Noisy Relationship
word1="etext";word2="declaration"
print("Similarity",similarity(word1,word2),word1,word2)
```

Furthermore, they both ended up in the tokenized dictionary:

```
Similarity [[0.880751]] etext declaration
```

Even better, their cosine similarity seems to be sure about its prediction and exceeds `0.5`. The stochastic nature of the algorithm might produce different results on various runs.

At a trivial or social media level, everything looks good.

However, at a professional level, the result is disastrous!

`etext` refers to *Project Gutenberg*'s preface to each ebook on their site, as explained in the *Matching datasets and tokenizers* section of this chapter. What is the goal of the transformer for a specific task:

- To understand an editor's preface?
- Or to understand the content of the book?

It depends on the usage of the transformer and might take a few days to sort out. For example, suppose an editor wants to understand prefaces automatically and uses a transformer to generate preface text. Should we take the content out?

`declaration` is a meaningful word related to the actual content of the *Declaration of Independence*.

`etext` is part of a preface that *Project Gutenberg* adds to all of its ebooks.

This might produce erroneous natural language inferences such as *etext is a declaration* when the transformer is asked to generate text.

Let's look into a missing word issue.

Case 3: Words in the text but not in the dictionary

In some cases, a word may be in a text but not in the dictionary. This will distort the results.

Let's take the words pie and logic:

```
#@title Case 3: word in text, not in dictionary
word1="pie";word2="logic"
print("Similarity",similarity(word1,word2),word1,word2)
```

The word pie is not in the dictionary:

```
pie :[unk] key not found in dictionary
Similarity 0 pie logic
```

We can assume that the word pie would be in a tokenized dictionary. But what if it isn't or another word isn't? The word pie is not in the text file.

Therefore, we should have functions in the pipeline to detect words that are not in the dictionary to implement corrections or alternatives. Also, we should have functions in the pipeline to detect words in the datasets that may be important.

Let's see the problem we face with rare words.

Case 4: Rare words

Rare words produce devasting effects on the output of transformers for specific tasks that go beyond simple applications.

Managing rare words extends to many domains of natural language. For example:

- Rare words can occur in datasets but go unnoticed, or models are poorly trained to deal with them.
- Rare words can be medical, legal, engineering terms, or any other professional jargon.
- Rare words can be slang.
- There are hundreds of variations of the English language. For example, different English words are used in certain parts of the United States, the United Kingdom, Singapore, India, Australia, and many other countries.
- Rare words can come from texts written centuries ago that are forgotten or that only specialists use.

For example, in this case, we are using the word justiciar:

```
#@title Case 4: Rare words
word1="justiciar";word2="judgement"
print("Similarity",similarity(word1,word2),word1,word2)
```

The similarity with `judgement` is reasonable but should be higher:

```
Similarity [[0.6606605]] justiciar judgement
```

You might think that the word `justiciar` is far-fetched. The tokenizer extracted it from the *Magna Carta*, dating back to the early 13th century. Unfortunately, the program will get confused, and we will obtain unexpected results after each run.

> Note: The predictions may vary from one run to another. However, they show how careful we must be in the tokenizing and embedding phases of our transformer model projects.

However, several articles of the *Magna Carta* are still valid in 21st century England! For example, clauses 1, 13, 39, and 40 are still valid!

The most famous part of the *Magna Carta* is the following excerpt, which is in the dataset:

```
(39) No free man shall be seized or imprisoned, or stripped of his
rights or possessions, or outlawed or exiled, or deprived of his
standing in any other way, nor will we proceed with force against him,
or send others to do so, except by the lawful judgement of his equals
or by the law of the land.
(40) To no one will we sell, to no one deny or delay right or justice.
```

If we implement a transformer model in a law firm to summarize documents or other tasks, we must be careful!

Let's now see some methods we could use to solve a rare word problem.

Case 5: Replacing rare words

Replacing rare words represents a project in itself. This work is reserved for specific tasks and projects. Suppose a corporate budget can cover the cost of having a knowledge base in aeronautics, for example. In that case, it is worth spending the necessary time querying the tokenized directory to find words it missed.

Problems can be grouped by topic, solved, and the knowledge base will be updated regularly.

In *Case 4*, we stumbled on the word `justiciar`. If we go back to its origin, we can see that it comes from the French Normand language and is the root of the French Latin-like word `judicaire`.

We could replace the word `justiciar` with `judge`, which conveys the same meta-concept:

```
#@title Case 5: Replacing rare words
word1="judge";word2="judgement"
print("Similarity",similarity(word1,word2),word1,word2)
```

It produces an interesting result, but we still need to be careful because of the non-deterministic aspect of the algorithm:

```
Similarity [[0.7962761]] judge judgement
```

We could also keep the word `justiciar`, but try the word's modern meaning and compare it to `judge`. You could try implementing `Case 5: Replacing rare words`:

```
word1="justiciar";word2="judge"
print("Similarity",similarity(word1,word2),word1,word2)
```

In any case, some rare words need to be replaced by more mainstream words.

The result would be satisfactory:

```
Similarity [[0.9659128]] justiciar judge
```

We could create queries with replacement words that we run until we find correlations that are over `0.9`, for example. Moreover, if we are managing a critical legal project, we could have the essential documents that contained rare words of any kind translated into standard English. Thus, the transformer's performance with NLP tasks would increase, and the knowledge base of the corporation would progressively increase.

Let's now see how to use cosine similarity for entailment verification.

Case 6: Entailment

In this case, we are interested in words in the dictionary and test them in a fixed order.

For example, let's see if “`pay`" + “`debt`" makes sense in our similarity function:

```
#@title Case 6: Entailment
word1="pay";word2="debt"
print("Similarity",similarity(word1,word2),word1,word2)
```

The result is satisfactory:

```
Similarity [[0.89891946]] pay debt
```

We could check the dataset with several word pairs and check if they mean something. These word pairs could be extracted from emails in a legal department, for example. If the cosine similarity is above `0.9`, then the email could be stripped of useless information and the content added to the knowledge base dataset of the company.

Let's now see how well-pretrained tokenizers match with NLP tasks.

Standard NLP tasks with specific vocabulary

This section focuses on *Case 4: Rare words* and *Case 5: Replacing rare words* from the *Word2Vec tokenization* section of this chapter.

We will use `Training_OpenAI_GPT_2_CH09.ipynb`, a renamed version of the notebook we used to train a dataset in *Chapter 7*, *The Rise of Suprahuman Transformers with GPT-3 Engines*.

Two changes were made to the notebook:

- `dset`, the dataset, was renamed `mdset` and contains medical content
- A Python function was added to control the text that was tokenized using byte-level BPE

We will not describe `Training_OpenAI_GPT_2_CH09.ipynb`, which we covered in *Chapter 7*, *The Rise of Suprahuman Transformers with GPT-3 Engines*, and *Appendices III and IV*. Make sure you upload the necessary files before beginning, as explained in *Chapter 7*.

There is no limit to the time you wish to train the model for. Interrupt it in order to save the model.

The files are on GitHub in the `gpt-2-train_files` directory of `Chapter09`. Although we are using the same notebook as in *Chapter 7*, note that the dataset, `dset`, is now named `mdset` in the directory and code.

First, let's generate an unconditional sample with a GPT-2 model trained to understand medical content.

Generating unconditional samples with GPT-2

We will get our hands dirty in this section to understand the inner workings of transformers. We could, of course, skip the whole chapter and simply use an OpenAI API. However, a 4.0 AI specialist must become an AI guru to *show*, not vaguely tell, Transformer models what to do through preprocessing pipelines. In order to *show* a transformer model what to do, it is necessary to understand how a transformer model works.

In *Case 4: Rare words*, and *Case 5: Replacing rare words*, we saw that rare words could be words used in a specific field, old English, variations of the English language around the world, slang, and more.

In 2020, the news was filled with medical terms to do with the COVID-19 outbreak. In this section, we will see how a GPT-2 transformer copes with medical text.

The dataset to encode and train contains a paper by *Martina Conte* and *Nadia Loy* (2020), named *Multi-cue kinetic model with non-local sensing for cell migration on a fibers network with chemotaxis*.

The title in itself is not easy to understand and contains rare words.

Load the files located in the `gpt-2-train_files` directory, including `mdset.txt`. Then run the code, as explained in *Chapter 7*, *The Rise of Suprahuman Transformers with GPT-3 Engines*. You can run this code cell by cell using *Chapter 7* to guide you. Take special care to follow the instructions to make sure `tf 1.x` is activated. Make sure to run *Step 4*, then restart the runtime and then run the *Step 4* `tf 1.x` cell again before continuing. Otherwise you will get an error in the notebook. We are getting our hands dirty to use the low-level original GPT-2 code in this section and not an API.

After training the model on the medical dataset, you will reach the unconditional sample cell, *Step 11: Generating Unconditional Samples*:

```
#@title Step 11: Generating Unconditional Samples
import os # import after runtime is restarted
os.chdir("/content/gpt-2/src")
!python generate_unconditional_samples.py --model_name '117M'
```

The time it takes to run this command and the other code in this notebook depends on the power of your machine. This notebook and all other GPT-2 code is explained for educational purposes only in this book. It is recommended to use OpenAI's API for GPT-3 in production. The response times are faster for transformer projects.

Run the cell and stop it when you wish. It will produce a random output:

```
community-based machinery facilitates biofilm growth. Community members
place biochemistry as the main discovery tool to how the cell interacts
with the environment and thus with themselves, while identifying and
understanding all components for effective Mimicry.
2. Ol Perception
```

```
Cytic double-truncation in phase changing (IP) polymerases (sometimes
called "tcrecs") represents a characteristic pattern of double-
crossing enzymes that alter the fundamental configuration that allows
initiation and maintenance of process while chopping the plainNA with
vibrational operator. Soon after radical modification that occurred during
translational parasubstitution (TMT) achieved a more or less uncontrolled
activation of SYX. TRSI mutations introduced autophosphorylation of TCMase
sps being the most important one that was incorporated into cellular
double-triad (DTT) signaling across all
cells, by which we allow R h and ofcourse an IC 2A- >
.../...
```

If we have a close look at the output, we notice the following points:

- The structure of the generated sentences is relatively acceptable
- The grammar of the output is not bad
- To a non-professional, the output might seem human-like

However, the content makes no sense. The transformer was unable to produce real content related to the medical paper we trained. Obtaining better results will take hard work. Of course, we can always increase the size of the dataset. But will it contain what we are looking for? Could we find wrong correlations with more data? For example, imagine a medical project involving COVID-19 with a dataset containing the following sentences:

- `COVID-19 is not a dangerous virus, but it is like ordinary flu.`
- `COVID-19 is a very dangerous virus.`
- `COVID-19 is not a virus but something created by a lab.`
- `COVID-19 was certainly not created by a lab!`
- `Vaccines are dangerous!`
- `Vaccines are lifesavers!`
- `Governments did not manage the pandemic correctly.`
- `Governments did what was necessary.`

And more contradictory sentences such as these. These discrepancies confirm that both datasets and tokenizers must be customized for specialized healthcare projects, aeronautics, transportation, and other critical domains.

Imagine you have a dataset with billions of words, but the content is so conflictual and noisy that you could never obtain a reliable result no matter what you try!

This could mean that the dataset would have to be smaller and limited to content from scientific papers. But even then, scientists often disagree with each other.

The conclusion is that it will take a lot of hard work and a solid team to produce reliable results.

Let's now try to condition the GPT-2 model.

Generating trained conditional samples

In this section, we move to the *Step 12: Interactive Context and Completion Examples* cell of the notebook and run it:

```
#@title Step 12: Interactive Context and Completion Examples
import os # import after runtime is restarted
os.chdir("/content/gpt-2/src")
!python interactive_conditional_samples.py --temperature 0.8 --top_k 40
--model_name '117M' --length 50
```

An Industry 4.0 AI specialist will focus less on code and more on how to *show* a transformer model what to do. Every model requires a level of showing what to do and not just using unconditional data to tell it to vaguely do something.

We condition the GPT-2 model by entering a part of the medical paper:

```
During such processes, cells sense the environment and respond to external
factors that induce a certain direction of motion towards specific targets
(taxis): this results in a persistent migration in a certain preferential
direction. The guidance cues leading to directed migration may be
biochemical or biophysical. Biochemical cues can be, for example, soluble
factors or growth factors that give rise to chemotaxis, which involves
a mono-directional stimulus. Other cues generating mono-directional
stimuli include, for instance, bound ligands to the substratum that induce
haptotaxis, durotaxis, that involves migration towards regions with an
increasing stiffness of the ECM, electrotaxis, also known as galvanotaxis,
that prescribes a directed motion guided by an electric field or current,
or phototaxis, referring to the movement oriented by a stimulus of light
[34]. Important biophysical cues are some of the properties of the
extracellular matrix (ECM), first among all the alignment of collagen
fibers and its stiffness. In particular, the fiber alignment is shown to
stimulate contact guidance [22, 21]. TL;DR:
```

We added `TL;DR`: at the end of the input text to tell the GPT-2 model to summarize the text we conditioned it with. The output makes sense, both grammatically and semantically:

```
the ECM of a single tissue is the ECM that is the most effective.
To address this concern, we developed a novel imaging and immunostaining
scheme that, when activated, induces the conversion of a protein to its
exogenous target
```

Since the outputs are non-deterministic, we could get this response also:

```
Do not allow the movement to be directed by a laser (i.e. a laser that
only takes one pulse at a time), but rather a laser that is directed at a
target and directed at a given direction. In a nutshell, be mindful.
```

The results are better but require more research.

The conclusion we can draw from this example and chapter is that pretraining transformer models on vast amounts of random web crawl data, for example, will teach the transformer English. However, like us, a transformer also needs to be trained in specific domains to become a specialist in that field.

Let's take our investigation further and control the tokenized data.

Controlling tokenized data

This section will read the first words the GPT-2 model encoded with its pretrained tokenizer.

When running the cells, stop a cell before running a subsequent one.

We will go to the `Additional Tools: Controlling Tokenized Data` cell of the `Training_OpenAI_GPT_2_CH09.ipynb` notebook we are using in this chapter. This cell was added to the notebook for this chapter.

The cell first unzips `out.npz`, which contains the encoded medical paper that is in the dataset, `mdset`:

```python
#@title Additional Tools : Controlling Tokenized Data
#Unzip out.npz
import zipfile
with zipfile.ZipFile('/content/gpt-2/src/out.npz', 'r') as zip_ref:
    zip_ref.extractall('/content/gpt-2/src/')
```

`out.npz` is unzipped and we can read `arr_0.npy`, the `NumPy` array that contains the encoded dataset we are looking for:

```
#Load arr_0.npy which contains encoded dset
import numpy as np
f=np.load('/content/gpt-2/src/arr_0.npy')
print(f)
print(f.shape)
for i in range(0,10):
    print(f[i])
```

The output is the first few elements of the array:

```
[1212 5644  326 ...   13  198 2682]
```

We will now open `encoder.json` and convert it into a Python dictionary:

```
#We first import encoder.json
import json
i=0
with open("/content/gpt-2/models/117M/encoder.json", "r") as read_file:
    print("Converting the JSON encoded data into a Python dictionary")
    developer = json.load(read_file) #converts the encoded data into a
Python dictionary
    for key, value in developer.items(): #we parse the decoded json data
        i+=1
        if(i>10):
            break;
        print(key, ":", value)
```

Finally, we display the key and value of the first `500` tokens of our encoded dataset:

```
#We will now search for the key and value for each encoded token
    for i in range(0,500):
        for key, value in developer.items():
            if f[i]==value:
                print(key, ":", value)
```

The first words of `mdset.txt` are as follows:

```
This suggests that
```

I added those words to make sure the GPT-2 pretrained tokenizer would easily recognize them, which is the case:

```
This : 1212
Ġsuggests : 5644
Ġthat : 326
```

We can easily recognize the initial tokens preceded by the initial whitespace characters (`Ġ`). However, let's take the following word in the medical paper:

```
amoeboid
```

`amoeboid` is a rare word. We can see that the GPT-2 tokenizer broke it down into sub-words:

```
Ġam : 716
o : 78
eb : 1765
oid : 1868
```

Let's skip the whitespace and look at what happened. `amoeboid` has become `am` + `o`+ `eb` + `oid.` We must agree that there are no unknown tokens: `[unk]`. That is due to the byte-level BPE strategy used.

However, the transformer's attention layers might associate:

- `am` with other sequences such as `I am`
- `o` with any sequence that was taken apart and contained an `o` as well
- `oid` with another sequence containing `oid`, possibly `tabloid` with some algorithms

This is not good news at all. Let's take this further with the following words:

```
amoeboid and mesenchymal
```

The output clearly displays `and`. As for the rest, the tokens are confusing:

```
Ġam : 716
o : 78
eb : 1765
oid : 1868
Ġand : 290
Ġmes : 18842
```

```
ench : 24421
ym : 4948
al : 282
```

You might wonder why this is a problem. The reason can be summed up in one word: polysemy. If we use a word2vec tokenizer, the dictionary might not contain rare words such as `amoeboid`, and we would come up with an unknown token.

If we use byte-level BPE, we obtain overall better results because we exclude fewer variations of the same word, such as go and go + `ing`.

However, the `am` token in `amoeboid` brings polysemy into the problem at a low level. `am` can be a sort of prefix, the word `am` as in `I` + `am`, or a sub-word such as in `am` + `bush`. Attention layers could associate the `am` of one token with another `am`, creating relationships that do not exist. This defines the core problem of polysemy in NLU.

We can say that progress is being made, but we need to work harder to improve NLP.

We have gone through a lot of the everyday problems we face in real-life projects using some examples. Take some time and try some examples you think are useful.

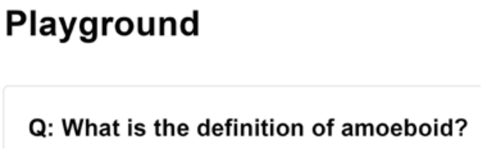
Before we leave, we will use a probing task to verify the level of NLU that a transformer model provides.

Exploring the scope of GPT-3

Even the most powerful transformers such as OpenAI GPT-3 have their limits. Let's see how GPT-3 reacts to the word `amoeboid`, which is closer to a medical term than a mainstream word. We will need technical jargon in many projects. Matching datasets requires quality control of how a transformer organizes its dictionary and embeddings.

We humans can detect errors and correct somebody. For example, in this chapter, we explored the word `amoeboid` in the *Controlling tokenized data* section of this chapter.

Let's first ask GPT-3 what `amoeboid` means:



Playground

Q: What is the definition of amoeboid?

Figure 9.4: Asking GPT-3 what "amoeboid" means

amoeboid (resembling an amoeba) is an adjective, yet GPT-3 states that it is a noun in the output:

```
A: Amoeboid is a noun which means "resembling an amoeba"
```

We then ask GPT-3 a more precise question and still obtain an incorrect answer:

```
Q: Is amoeboid a noun or an adjective?
A: Amoeboid is a noun.
```

Finally, we insist and ask for a clear definition and obtain a correct answer:

```
Q: What does amoeboid mean in medical terms?
A: Amoeboid means "resembling an amoeba".
```

The definition is accurate, although the grammatical analysis isn't.

What's more important in a real-life project? To understand the definition of a word or to identify its role in a sentence as an adjective or a noun?

> The definition of a word is sufficient for a medical project. In this case, GPT-3 might be sufficient. If the definition is sufficient, SRL wasn't a prerequisite for understanding a sentence.

Maybe grammatical aspects are important for an educational grammar school project, but not for corporate supply chain, finance, and e-commerce applications.

OpenAI GPT-3 can be fine-tuned in both cases, as we saw in *Chapter 7*, *The Rise of Suprahuman Transformers with GPT-3 Engines*.

This section concludes that we have to ensure we have all the data we need in a trained transformer model. If not, the tokenization process will be incomplete. Maybe we should have taken a medical dictionary and created a large corpus of medical articles that contained that specific vocabulary. Then if the model still is not accurate enough, we might have to tokenize our dataset and train a model from scratch.

A 2022 developer will have less development work, but will still have to think and design a lot!

Let's now conclude this chapter and move on to another NLU task.

Summary

In this chapter, we measured the impact of the tokenization and subsequent data encoding process on transformer models. A transformer model can only attend to tokens from the embedding and positional encoding sub-layers of a stack. It does not matter if the model is an encoder-decoder, encoder-only, or decoder-only model. It does not matter if the dataset seems good enough to train.

If the tokenization process fails, even partly, the transformer model we are running will miss critical tokens.

We first saw that for standard language tasks, raw datasets might be enough to train a transformer.

However, we discovered that even if a pretrained tokenizer has gone through a billion words, it only creates a dictionary with a small portion of the vocabulary it comes across. Like us, a tokenizer captures the essence of the language it is learning and only *remembers* the most important words if these words are also frequently used. This approach works well for a standard task and creates problems with specific tasks and vocabulary.

We then looked for some ideas, among many, to work around the limits of standard tokenizers. We applied a language checking method to adapt the text we wish to process, such as how a tokenizer *thinks* and encodes data.

We applied the method to unconditional and conditional tasks with GPT-2.

Finally, we analyzed the limits of data tokenizing and matching datasets with GPT-3. The lesson you can take away from this chapter is that AI specialists are here to stay for quite some time!

In the next chapter, *Semantic Role Labeling with BERT-Based Transformers*, we will dig deep into NLU and use a BERT model to ask a transformer to explain the meaning of a sentence.

Questions

1. A tokenized dictionary contains every word that exists in a language. (True/False)
2. Pretrained tokenizers can encode any dataset. (True/False)
3. It is good practice to check a database before using it. (True/False)
4. It is good practice to eliminate obscene data from datasets. (True/False)
5. It is good practice to delete data containing discriminating assertions. (True/False)
6. Raw datasets might sometimes produce relationships between noisy content and useful content. (True/False)

7. A standard pretrained tokenizer contains the English vocabulary of the past 700 years. (True/False)
8. Old English can create problems when encoding data with a tokenizer trained in modern English. (True/False)
9. Medical and other types of jargon can create problems when encoding data with a tokenizer trained in modern English. (True/False)
10. Controlling the output of the encoded data produced by a pretrained tokenizer is good practice. (True/False)

References

- *Colin Raffel*, *Noam Shazeer*, *Adam Roberts*, *Katherine Lee*, *Sharan Narang*, *Michael Matena*, *Yanqi Zhou*, *Wei Li*, and *Peter J. Liu*, 2019, *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*: `https://arxiv.org/pdf/1910.10683.pdf`
- OpenAI GPT-2 GitHub repository: `https://github.com/openai/gpt-2`
- N. Shepperd GitHub repository: `https://github.com/nshepperd/gpt-2`
- Hugging Face framework and resources: `https://huggingface.co/`
- U.S. Legal, *Montana Corporate Laws*: `https://corporations.uslegal.com/state-corporation-law/montana-corporation-law/#:~:text=Montana%20Corporation%20Law,carrying%20out%20its%20business%20activities`
- *Martina Conte*, *Nadia Loy*, 2020, *Multi-cue kinetic model with non-local sensing for cell migration on a fibers network with chemotaxis*: `https://arxiv.org/abs/2006.09707`
- *The Declaration of Independence of the United States of America*, by *Thomas Jefferson*: `https://www.gutenberg.org/ebooks/1`
- *The United States Bill of Rights*, by the United States, and related texts: `https://www.gutenberg.org/ebooks/2`
- *The Magna Carta*: `https://www.gutenberg.org/ebooks/10000`
- *The Critique of Pure Reason*, *The Critique of Practical Reason*, and *Fundamental Principles of the Metaphysic of Moral*: `https://www.gutenberg.org`

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the authors:

`https://www.packt.link/Transformers`

10

Semantic Role Labeling with BERT-Based Transformers

Transformers have made more progress in the past few years than NLP in the past generation. Standard NLU approaches first learn syntactical and lexical features to explain the structure of a sentence. The former NLP models would be trained to understand a language's basic syntax before running **Semantic Role Labeling** (**SRL**).

Shi and *Lin* (2019) started their paper by asking if preliminary syntactic and lexical training can be skipped. Can a BERT-based model perform SRL without going through those classical training phases? The answer is yes!

Shi and *Lin* (2019) suggested that SRL can be considered sequence labeling and provide a standardized input format. Their BERT-based model produced surprisingly good results.

This chapter will use a pretrained BERT-based model provided by the Allen Institute for AI based on the *Shi* and *Lin* (2019) paper. *Shi* and *Lin* took SRL to the next level by dropping syntactic and lexical training. We will see how this was achieved.

We will begin by defining SRL and the standardization of the sequence labeling input formats. We will then get started with the resources provided by the Allen Institute for AI. Next, we will run SRL tasks in a Google Colab notebook and use online resources to understand the results.

Finally, we will challenge the BERT-based model by running SRL samples. The first samples will show how SRL works. Then, we will run some more difficult samples. We will progressively push the BERT-based model to the limits of SRL. Finding the limits of a model is the best way to ensure that real-life implementations of transformer models remain realistic and pragmatic.

This chapter covers the following topics:

- Defining semantic role labeling
- Defining the standardization of the input format for SRL
- The main aspects of the BERT-based model's architecture
- How an encoder-only stack can manage a masked SRL input format
- BERT-based model SRL attention process
- Getting started with the resources provided by the Allen Institute for AI
- Building a notebook to run a pretrained BERT-based model
- Testing sentence labeling on basic examples
- Testing SRL on difficult examples and explaining the results
- Taking the BERT-based model to the limit of SRL and explaining how this was done

Our first step will be to explore the SRL approach defined by *Shi* and *Lin* (2019).

Getting started with SRL

SRL is as difficult for humans as for machines. However, once again, transformers have taken a step closer to our human baselines.

In this section, we will first define SRL and visualize an example. We will then run a pretrained BERT-based model.

Let's begin by defining the problematic task of SRL.

Defining semantic role labeling

Shi and *Lin* (2019) advanced and proved the idea that we can find who did what, and where, without depending on lexical or syntactic features. This chapter is based on *Peng Shi* and *Jimmy Lin*'s research at the *University of Waterloo*, California. They showed how transformers learn language structures better with attention layers.

SRL labels the *semantic role* as the role a word or group of words plays in a sentence and the relationship established with the predicate.

A *semantic role* is the role a noun or noun phrase plays in relation to the main verb in a sentence. For example, in the sentence `Marvin walked in the park`, `Marvin` is the *agent* of the event occurring in the sentence. The *agent* is the *doer* of the event. The main verb, or *governing verb*, is `walked`.

The *predicate* describes something about the subject or agent. The predicate could be anything that provides information on the features or actions of a subject. In our approach, we will refer to the predicate as the main *verb*. For example, in the sentence `Marvin walked in the park`, the predicate is `walked` in its restricted form.

The words `in the park` *modify* the meaning of `walked` and are the *modifier*.

The noun or noun phrases that revolve around the predicate are *arguments* or *argument terms*. `Marvin`, for example, is an *argument* of the *predicate* `walked`.

We can see that SRL does not require a syntax tree or a lexical analysis.

Let's visualize the SRL of our example.

Visualizing SRL

This chapter will be using the Allen Institute's visual and code resources (see the *References* section for more information). The Allen Institute for AI has excellent interactive online tools, such as the one we've used to represent SRL visually throughout this chapter. You can access these tools at `https://demo.allennlp.org/`.

The Allen Institute for AI advocates *AI for the common good*. We will make good use of this approach. All of the figures in this chapter were created with the AllenNLP tools.

The Allen Institute provides transformer models that continuously evolve. Therefore, the examples in this chapter might produce different results when you run them. The best way to get the most out of this chapter is to:

- Read and understand the concepts explained beyond merely running a program
- Take the time to understand the examples provided

Then run your own experiments with sentences of your choice with the tool used in this chapter: `https://demo.allennlp.org/semantic-role-labeling`.

We will now visualize our SRL example. *Figure 10.1* is an SRL representation of `Marvin walked in the park`:

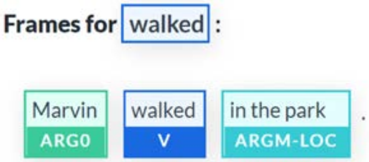


Figure 10.1: The SRL representation of a sentence

We can observe the following labels in *Figure 10.1*:

- **Verb**: The predicate of the sentence
- **Argument**: An argument of the sentence named **ARG0**
- **Modifier**: A modifier of the sentence. In this case, a location. It could have been an adverb, an adjective, or anything that modifies the predicate's meaning

The text output is interesting as well, which contains shorter versions of the labels of the visual representation:

```
walked: [ARG0: Marvin] [V: walked] [ARGM-LOC: in the park]
```

We have defined SRL and gone through an example. It is time to look at the BERT-based model.

Running a pretrained BERT-based model

This section will begin by describing the architecture of the BERT-based model used in this chapter.

Then we will define the method used to experiment with SRL samples with a BERT model.

Let's begin by looking at the architecture of the BERT-based model.

The architecture of the BERT-based model

AllenNLP's BERT-based model is a 12-layer encoder-only BERT model. The AllenNLP team implemented the BERT model described in *Shi* and *Lin* (2019) with an additional linear classification layer.

For more on the description of a BERT model, take a few minutes, if necessary, to go back to *Chapter 3*, *Fine-Tuning BERT Models*.

The BERT-based model takes full advantage of bidirectional attention with a simple approach and architecture. The core potential of transformers resides in the attention layers. We have seen transformer models with both encoder and decoder stacks. We have seen other transformers with encoder layers only or decoder layers only. The main advantage of transformers remains in the near-human approach of attention layers.

The input format of the predicate identification format defined by *Shi* and *Lin* (2019) shows how far transformers have come to understand a language in a standardized fashion:

```
[CLS] Marvin walked in the park.[SEP] walked [SEP]
```

The training process has been standardized:

- `[CLS]` indicates that this is a classification exercise
- `[SEP]` is the first separator, indicating the end of the sentence
- `[SEP]` is followed by the predicate identification designed by the authors
- `[SEP]` is the second separator, which indicates the end of the predicate identifier

This format alone is enough to train a BERT model to identify and label the semantic roles in a sentence.

Let's set up the environment to run SRL samples.

Setting up the BERT SRL environment

We will be using a Google Colab notebook, the AllenNLP visual text representations of SRL available at `https://demo.allennlp.org/` under the *Semantic Role Labeling* section.

We will apply the following method:

1. We will open `SRL.ipynb`, install `AllenNLP`, and run each sample
2. We will display the raw output of the SRL run
3. We will visualize the output using AllenNLP's online visualization tools
4. We will display the output using AllenNLP's online text visualization tools

This chapter is self-contained. You can read through it or run the samples as described.

The SRL model output may differ when AllenNLP changes the transformer model used. This is because AllenNLP models and transformers, in general, are continuously trained and updated. Also, the datasets used for training might change. Finally, these are not rule-based algorithms that produce the same result each time. The outputs might change from one run to another, as described and shown in the screenshots.

Let's now run some SRL experiments.

SRL experiments with the BERT-based model

We will run our SRL experiments using the method described in the *Setting up the BERT SRL environment* section of this chapter. We will begin with basic samples with various sentence structures. We will then challenge the BERT-based model with some more difficult samples to explore the system's capacity and limits.

Open `SRL.ipynb` and run the installation cell:

```
!pip install allennlp==2.1.0 allennlp-models==2.1.0
```

Then we import the tagging module and a trained BERT predictor:

```
from allennlp.predictors.predictor import Predictor
import allennlp_models.tagging
import json


predictor = Predictor.from_path("https://storage.googleapis.com/allennlp-public-models/structured-prediction-srl-bert.2020.12.15.tar.gz")
```

We also add two functions to display the JSON object SRL BERT returns. The first one displays the verb of the predicate and the description:

```
def head(prediction):
  # Iterating through the json to display excerpt of the prediciton
  for i in prediction['verbs']:
    print('Verb:',i['verb'],i['description'])
```

The second one displays the full response, including the tags:

```
def full(prediction):
  #print the full prediction
  print(json.dumps(prediction, indent = 1, sort_keys=True))
```

At the time of this publication, the BERT model was specifically trained for semantic role labeling use. The name of the model is SRL BERT. SRL BERT was trained using the OntoNotes 5.0 dataset: `https://catalog.ldc.upenn.edu/LDC2013T19`.

This dataset contains sentences and annotations. The dataset was designed to identify predicates (a part of a sentence containing a verb) in a sentence and identify the words that provide more information on the verbs. Each verb comes with its “arguments” that tell us more about it. A “frame” contains the arguments of a verb.

SRL BERT is thus a specialized model trained to perform a specific task, and as such, it is not a foundation model like OpenAI GPT-3 as we saw in *Chapter 7*, *The Rise of Suprahuman Transformers with GPT-3 Engines*.

SRL BERT will focus on semantic role labeling with acceptable accuracy as long as the sentence contains a predicate.

We are now ready to warm up with some basic samples.

Basic samples

Basic samples seem intuitively simple but can be tricky to analyze. Compound sentences, adjectives, adverbs, and modals are difficult to identify, even for non-expert humans.

Let's begin with an easy sample for the transformer.

Sample 1

The first sample is long but relatively easy for the transformer:

`Did Bob really think he could prepare a meal for 50 people in only a few hours?`

Run the *Sample 1* cell in `SRL.ipynb`:

```
prediction=predictor.predict(
    sentence="Did Bob really think he could prepare a meal for 50 people
in only a few hours?"
)
head(prediction)
```

BERT SRL identified the four predicates; the verb for each one labeled the result as shown in this excerpt using the `head(prediction)` function:

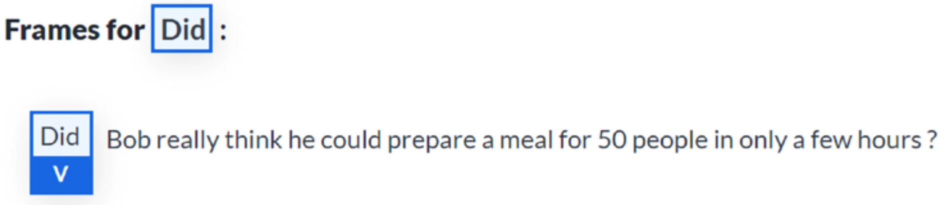
```
Verb: Did [V: Did] Bob really think he could prepare a meal for 50 people
in only a few hours ?
Verb: think Did [ARG0: Bob] [ARGM-ADV: really] [V: think] [ARG1: he could
prepare a meal for 50 people in only a few hours] ?
Verb: could Did Bob really think he [V: could] [ARG1: prepare a meal for
50 people in only a few hours] ?
Verb: prepare Did Bob really think [ARG0: he] [ARGM-MOD: could] [V:
prepare] [ARG1: a meal for 50 people] [ARGM-TMP: in only a few hours] ?
```

You can view the full response by running the `full(prediction)` cell.

If we use the description of the arguments of the dataset based on the structure of the PropBank (Proposition Bank), the verb `think`, for example, in this excerpt can be interpreted as:

- `V` identifies the verb `think`
- `ARG0` identifies the agent; thus, `Bob` is the agent or "pro-agent"
- `ARGM-ADV` considers `really` as an adverb (`ADV`), with `ARGM` meaning that the adverb provides an adjunct (not necessary) thus not numbered

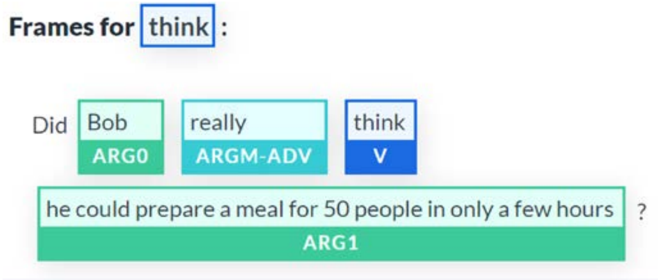
If we run the sample in the AllenNLP online interface, we obtain a visual representation of the SRL task of one frame per verb. The first verb is `Did`:


Frames for Did :

Did Bob really think he could prepare a meal for 50 people in only a few hours ?

Figure 10.2: Identifying the verb "Did"

The second verb identified is `think`:


Frames for think :

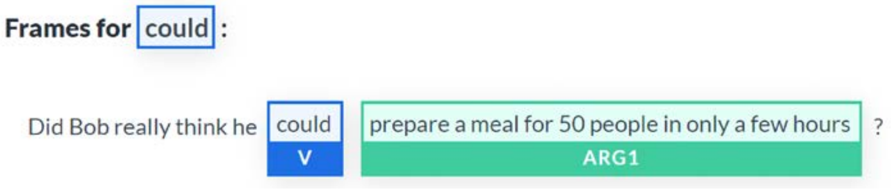
Did Bob really think he could prepare a meal for 50 people in only a few hours ?

Figure 10.3: Identifying the verb "think"

If we take a close look at this representation, we can detect some interesting properties of the SRL BERT that:

- Detected the verb `think`
- Avoided the `prepare` trap that could have been interpreted as the main verb. Instead, `prepare` remained part of the argument of `think`
- Detected an adverb and labeled it

The third verb is `could`:

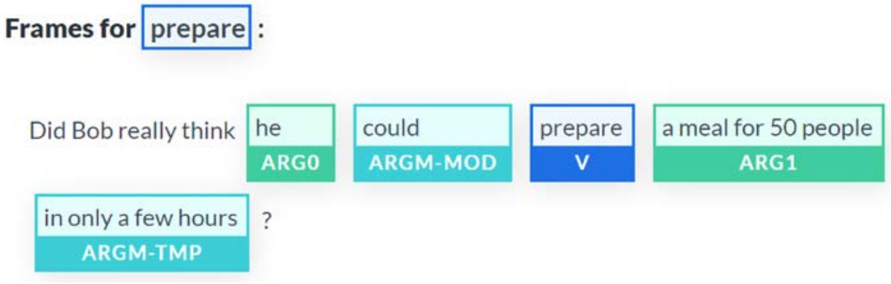


Frames for could **:**

Did Bob really think he could prepare a meal for 50 people in only a few hours ?

Figure 10.4: Identifying the verb "could" and the argument

The transformer then moved to the verb `prepare`, labeled it, and analyzed its context:



Frames for prepare **:**

Did Bob really think he could prepare a meal for 50 people in only a few hours ?

Figure 10.5: Identifying the verb "prepare", the arguments, and the modifiers

Again, the simple BERT-based transformer model detected a lot of information on the grammatical structure of the sentence and found:

- The verb `prepare` and isolated it
- The noun `he` and labeled it as an argument and did the same for `a meal for 50 people`, which is a "proto-patient," which involves a modification by other participants
- That `in only a few hours` is a temporal modifier (**ARGM-TMP**)
- That `could` was a modal modifier that indicates the modality of a verb, such as the likelihood of an event

We will now analyze another relatively long sentence.

Sample 2

The following sentence seems easy but contains several verbs:

```
Mrs. and Mr. Tomaso went to Europe for vacation and visited Paris and first went
to visit the Eiffel Tower.
```

Will this confusing sentence make the transformer hesitate? Let's see by running the *Sample 2* cell of the `SRL.ipynb` notebook:

```
prediction=predictor.predict(
    sentence="Mrs. and Mr. Tomaso went to Europe for vacation and visited
Paris and first went to visit the Eiffel Tower."
)
head(prediction)
```

The excerpt of the output proves that the transformer correctly identified the verbs in the sentence:

```
Verb: went [ARG0: Mrs. and Mr. Tomaso] [V: went] [ARG4: to Europe] [ARGM-
PRP: for vacation] and visited Paris and first went to visit the Eiffel
Tower .

Verb: visited [ARG0: Mrs. and Mr. Tomaso] went to Europe for vacation and
[V: visited] [ARG1: Paris] and first went to visit the Eiffel Tower .

Verb: went [ARG0: Mrs. and Mr. Tomaso] went to Europe for vacation and
visited Paris and [ARGM-TMP: first] [V: went] [ARG1: to visit the Eiffel
Tower] .
Verb: visit [ARG0: Mrs. and Mr. Tomaso] went to Europe for vacation and
visited Paris and first went to [V: visit] [ARG1: the Eiffel Tower] .
```

Running the sample on AllenNLP online identified four predicates, thus generating four frames.

The first frame is for `went`:

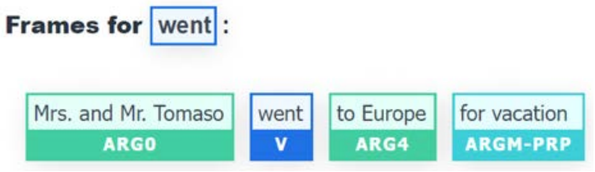
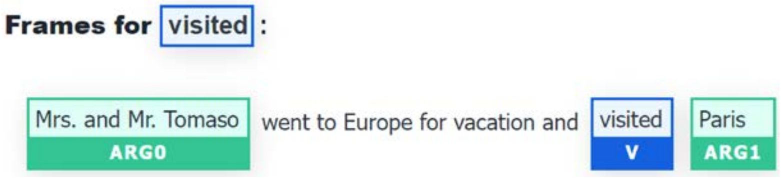


Figure 10.6: Identifying the verb "went," the arguments, and the modifier

We can interpret the arguments of the verb `went`. `Mrs. and Mr. Tomaso` are the agents. The transformer found that the main modifier of the verb was the purpose of the trip: `to Europe`. The result would not be surprising if we did not know that *Shi* and *Lin* (2019) had only built a simple BERT model to obtain this high-quality grammatical analysis.

We can also notice that `went` was correctly associated with `Europe`. The transformer correctly identified the verb `visited` as being related to `Paris`:



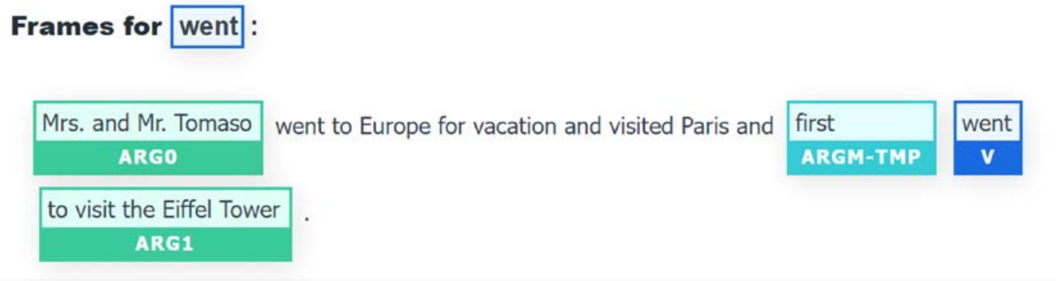
Frames for visited :

Mrs. and Mr. Tomaso ARG0 went to Europe for vacation and visited V Paris ARG1

Figure 10.7: Identifying the verb "visited" and the arguments

The transformer could have associated the verb `visited` directly with the `Eiffel Tower`. But it didn't. It stood its ground and made the right decision.

The next task we asked the transformer to do was identify the context of the second use of the verb `went`. Again, it did not fall into the trap of merging all of the arguments related to the verb `went`, used twice in the sentence. Again, it correctly split the sequence and produced an excellent result:



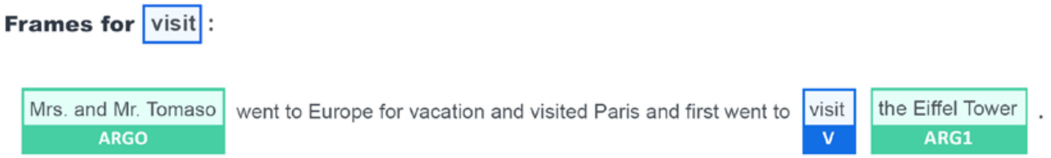
Frames for went :

Mrs. and Mr. Tomaso ARG0 went to Europe for vacation and visited Paris and first ARGM-TMP went V to visit the Eiffel Tower ARG1 .

Figure 10.8: Identifying the verb "went," the argument, and the modifiers

The verb `went` was used twice, but the transformer did not fall into the trap. It even found that `first` was a temporal modifier of the verb `went`.

Finally, the verb `visit` was used a second time, and SRL BERT correctly interpreted its use:



Frames for visit :

Mrs. and Mr. Tomaso ARGO went to Europe for vacation and visited Paris and first went to visit V the Eiffel Tower ARG1 .

Figure 10.9: Identifying the verb "visit" and the arguments

Let's run a sentence that is a bit more confusing.

Sample 3

Sample 3 will make things more difficult for our transformer model. The following sample contains variations of the verb `drink` four times:

`John wanted to drink tea, Mary likes to drink coffee but Karim drank some cool water and Faiza would like to drink tomato juice.`

Let's run *Sample 3* in the `SRL.ipynb` notebook:

```
prediction=predictor.predict(
    sentence="John wanted to drink tea, Mary likes to drink coffee but
Karim drank some cool water and Faiza would like to drink tomato juice."
)
head(prediction)
```

The transformer found its way around, as shown in the following excerpt of the output that contains the verbs:

```
Verb: wanted [ARG0: John] [V: wanted] [ARG1: to drink tea] , Mary likes to
drink coffee but Karim drank some cool water and Faiza would like to drink
tomato juice .

Verb: drink [ARG0: John] wanted to [V: drink] [ARG1: tea] , Mary likes to
drink coffee but Karim drank some cool water and Faiza would like to drink
tomato juice .

Verb: likes John wanted to drink tea , [ARG0: Mary] [V: likes] [ARG1: to
drink coffee] but Karim drank some cool water and Faiza would like to
drink tomato juice .

Verb: drink John wanted to drink tea , [ARG0: Mary] likes to [V: drink]
[ARG1: coffee] but Karim drank some cool water and Faiza would like to
drink tomato juice .

Verb: drank John wanted to drink tea , Mary likes to drink coffee but
[ARG0: Karim] [V: drank] [ARG1: some cool water] and Faiza would like to
drink tomato juice .

Verb: would John wanted to drink tea , Mary likes to drink coffee but
Karim drank some cool water and [ARG0: Faiza] [V: would] like [ARG1: to
drink tomato juice] .
```

```
Verb: like John wanted to drink tea , Mary likes to drink coffee but Karim
drank some cool water and [ARG0: Faiza] [ARGM-MOD: would] [V: like] [ARG1:
to drink tomato juice] .

Verb: drink John wanted to drink tea , Mary likes to drink coffee but
Karim drank some cool water and [ARG0: Faiza] would like to [V: drink]
[ARG1: tomato juice] .
```

We obtain several visual representations when we run the sentence on the AllenNLP online interface. We will examine two of them.

The first one is perfect. It identifies the verb `wanted` and makes the correct associations:

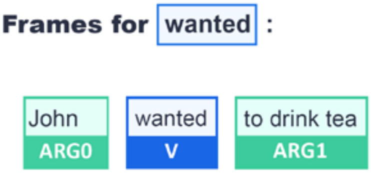


Figure 10.10: Identifying the verb "wanted" and the arguments

When it identified the verb `drank`, it correctly excluded `Faiza` and only produced `some cool water` as the argument.

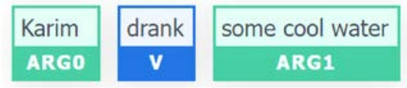


Figure 10.11: Identifying the verb "drank" and the arguments

We've found that the BERT-based transformer produces relatively good results up to now on basic samples. So let's try some more difficult ones.

Difficult samples

This section will run samples that contain problems that the BERT-based transformer will first solve. Finally, we will end with an intractable sample.

Let's start with a complex sample that the BERT-based transformer can analyze.

Sample 4

Sample 4 takes us into more tricky SRL territory. The sample separates `Alice` from the verb `liked`, creating a long-term dependency that has to jump over `whose husband went jogging every Sunday.`

The sentence is:

`Alice, whose husband went jogging every Sunday, liked to go to a dancing class in the meantime.`

A human can isolate `Alice` and find the predicate:

`Alice liked to go to a dancing class in the meantime.`

Can the BERT model find the predicate like us?

Let's find out by first running the code in `SRL.ipynb`:

```
prediction=predictor.predict(
    sentence="Alice, whose husband went jogging every Sunday, liked to go
to a dancing class in the meantime."
)
head(prediction)
```

The output identified the verbs for each predicate and labeled each frame:

```
Verb: went Alice , [ARG0: whose husband] [V: went] [ARG1: jogging] [ARGM-
TMP: every Sunday] , liked to go to a dancing class in the meantime .

Verb: jogging Alice , [ARG0: whose husband] went [V: jogging] [ARGM-TMP:
every Sunday] , liked to go to a dancing class in the meantime .

Verb: liked [ARG0: Alice , whose husband went jogging every Sunday] , [V:
liked] [ARG1: to go to a dancing class in the meantime] .

Verb: go [ARG0: Alice , whose husband went jogging every Sunday] , liked
to [V: go] [ARG4: to a dancing class] [ARGM-TMP: in the meantime] .

Verb: dancing Alice , whose husband went jogging every Sunday , liked to
go to a [V: dancing] class in the meantime .
```

Let's focus on the part we are interested in and see if the model found the predicate. It did! It found the verb `liked` as shown in this excerpt of the output, although the verb `like` is separated from `Alice` by another predicate:

```
Verb: liked [ARG0: Alice , whose husband went jogging every Sunday]
```

Let's now look at the visual representation of the model's analysis after running the sample on AllenNLP's online UI. The transformer first finds Alice's husband:

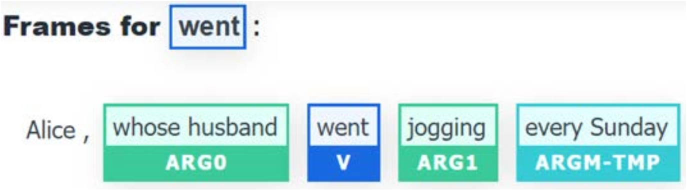


Figure 10.12: The predicate "went" has been identified

The transformer explains that:

- The predicate or verb is `went`
- `whose husband` is the argument
- `jogging` is another argument related to `went`
- `every Sunday` is a temporal modifier represented in the raw output as `[ARGM-TMP: every Sunday]`

The transformer then found what Alice's husband was *doing*:

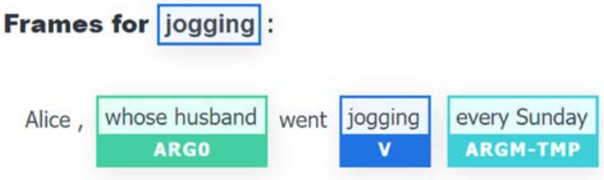
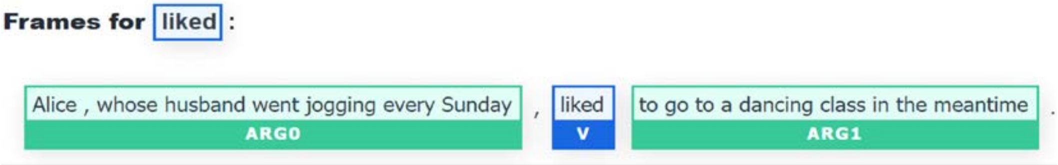


Figure 10.13: SRL detection of the verb "jogging"

We can see that the verb `jogging` was identified and was related to `whose husband` with the temporal modifier `every Sunday`.

The transformer doesn't stop there. It now detects what Alice liked:

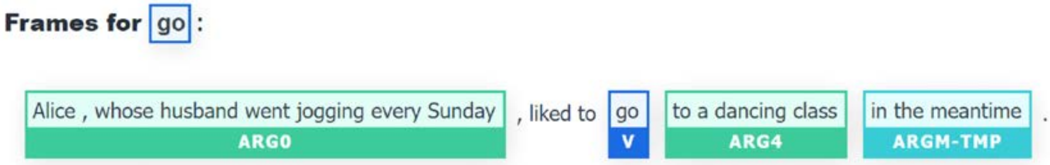


Frames for liked :

Alice , whose husband went jogging every Sunday ARG0 , liked V to go to a dancing class in the meantime ARG1 .

Figure 10.14: Identifying the verb "liked"

The transformer also detects and analyzes the verb go correctly:



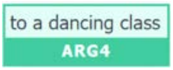
Frames for go :

Alice , whose husband went jogging every Sunday ARG0 , liked to go V to a dancing class ARG4 in the meantime ARGM-TMP .

Figure 10.15: Detecting the verb "go," its arguments, and modifier

We can see that the temporal modifier `in the meantime` was also identified. It is quite a performance considering the simple *sequence* + *verb* input SRL BERT was trained with.

Finally, the transformer identifies the last verb, `dancing`, as being related to `class`:



to a dancing class ARG4

Figure 10.16: Relating the argument "class" to the verb "dancing"

The results produced by *Sample 4* are quite convincing! Let's try to find the limit of the transformer model.

Sample 5

Sample 5 does not repeat a verb several times. However, *Sample 5* contains a word with multiple functions and meanings. It goes beyond polysemy since the word round can have different meanings and grammatical functions. The word round can be a noun, an adjective, an adverb, a transitive verb, or an intransitive verb.

As a transitive or intransitive verb, round can attain perfection or completion. In this sense, round can be used with `off`.

The following sentence uses round in the past tense:

```
The bright sun, the blue sky, the warm sand, the palm trees, everything round off.
```

The verb round in this predicate is used in a sense of "to bring to perfection." Of course, the most accessible grammatical form would have been "rounded." but let's see what happens with our sentence.

Let's run *Sample 5* in `SRL.ipynb`:

```
prediction=predictor.predict(
    sentence="The bright sun, the blue sky, the warm sand, the palm trees,
everything round off."
)
head(prediction)
```

The output shows no verbs. The transformer did not identify the predicate. In fact, it found no verbs at all even when we run the `full(prediction)` function:

```
"verbs": []
```

However, the online version seems to interpret the sentence better because it found the verb:

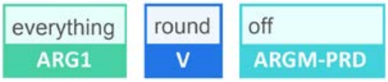


Figure 10.17: Detecting the verb "round" and "everything" as the argument

Since we like our SRL transformer, we will be kind to it. *We will show it what to do with a verb form that is more frequently used*. Let's change the sentence from the past tense to the present tense by adding an s to round:

```
The bright sun, the blue sky, the warm sand, the palm trees, everything rounds off.
```

Let's give `SRL.ipynb` another try with the present tense:

```
prediction=predictor.predict(
    sentence="The bright sun, the blue sky, the warm sand, the palm trees,
everything rounds off."
)
head(prediction)
```

The raw output shows that the predicate was found, as shown in the following output:

```
Verb: rounds [ARG1: The bright sun , the blue sky , the warm sand , the
palm trees , everything] [V: rounds] [ARGM-PRD: off] .
```

If we run the sentence on AllenNLP, we obtain the visual explanation:

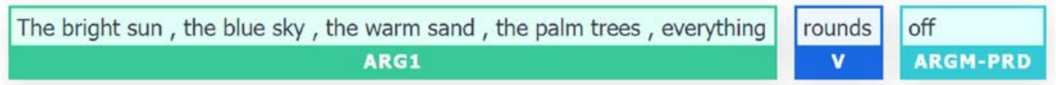

The bright sun , the blue sky , the warm sand , the palm trees , everything
ARG1
rounds
V
off
ARGM-PRD

Figure 10.18: Detecting the word "rounds" as a verb

Our BERT-based transformer did well because the word `round` can be found as `rounds` in its present tense form.

The BERT model initially failed to produce the result we expected. But with a little help from its friends, all ended well for this sample.

We can see that:

- Outputs may vary with the evolution of the versions of a model we have implemented
- An Industry 4.0 pragmatic mindset requires more cognitive efforts to *show* a transformer what to do

Let's try another sentence that's difficult to label.

Sample 6

Sample 6 takes a word we often think is just a noun. However, more words than we suspect can be both nouns and verbs. For example, *to ice* is a verb used in hockey to shoot a puck all the way across the rink and beyond the goal line of an opponent. A puck is the disk used in hockey.

A hockey coach can start the day by telling a team to train icing pucks. We then can obtain the *imperative* sentence when the coach yells:

`Now, ice pucks guys!`

Note that guys can mean persons regardless of their sex.

Let's run the *Sample 6* cell to see what happens:

```
prediction=predictor.predict(
    sentence="Now, ice pucks guys!"
)
head(prediction)
```

The transformer fails to find the verb: `"verbs": []`.

Game over! We can see that transformers have made tremendous progress, but there is still a lot of room for developers to improve the models. Humans are still in the game to *show* transformers what to do.

The online interface confuses `pucks` with a verb:

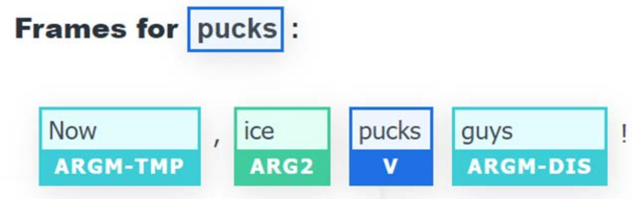


Figure 10.19: The model incorrectly labels "pucks" as the verb

This problem may be solved with another model, but you will reach another limit. Even GPT-3 has limits you will have to cope with.

> When you implement transformers for a customized application with specialized jargon or technical vocabulary, you will reach intractable limits at some point.

These limits will require your expertise to make a project a success. Therefore, you will have to create specialized dictionaries to succeed in a project. This is good news for developers! You will develop new cross-disciplinary and cognitive skills that your team will appreciate.

Try some examples or samples of your own to see what SRL can do with the approach's limits. Then explore how to develop preprocessing functions to show the transformer what to do for your customized applications.

Before we leave, let's question the motivation of SRL.

Questioning the scope of SRL

We are alone when faced with a real-life project. We have a job to do, and the only people to satisfy are those who asked for that project.

Pragmatism must come first. Technical ideology after.

In the 2020s, former AI ideology and new ideology coexist. By the end of the decade, there will be only one winner merging some of the former into the latter.

This section questions the productivity of SRL and also its motivation through two aspects:

- The limit of predicate analysis
- Questioning the use of the term "semantic"

The limit of predicate analysis

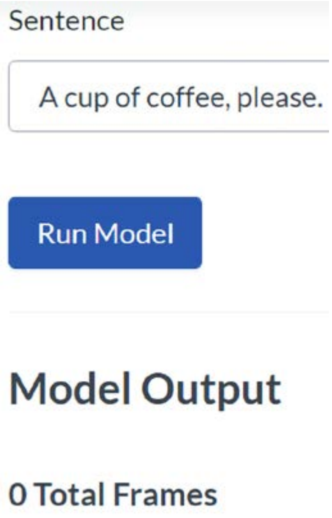
SRL relies on predicates. SRL BERT only works as long as you provide a verb. But millions of sentences do not contain verbs.

If you provide SRL BERT in the **Semantic Role Labeling** section in the AllenNLP demo interface (`https://demo.allennlp.org/`) with an assertion alone, it works.

But what happens if your assertion is an answer to a question:

- Person 1: `What would you like to drink, please?`
- Person 2: `A cup of coffee, please.`

When we enter Person 2's answer, SRL BERT finds nothing:



Sentence

A cup of coffee, please.

Model Output

0 Total Frames

Figure 10.20: No frame obtained

The output is `0` total frames. SRL was unable to analyze this sentence because it contains an *ellipsis*. The predicate is implicit, not explicit.

The definition of an ellipsis is the act of leaving one or several words out of a sentence that is not necessary for it to be understood.

Hundreds of millions of sentences containing an ellipsis are spoken and written each day.

Yet SRL BERT yields *0 Total Frames* for all of them.

The following answers (`A`) to questions (`Q`) beginning with `what`, `where`, and `how` yield *0 Total Frames*:

`Q: What would you like to have for breakfast?`

`A: Pancakes with hot chocolate.`

(The model deduces: `pancakes`=proper noun, `with`=preposition, `hot`= adjective, and `chocolate`=common noun.)

`Q: Where do you want to go?`

`A: London, please.`

(The model deduces: `London`=proper noun and `please`=adverb.)

`Q: How did you get to work today?`

`A: Subway.`

(The model deduces: `subway`=proper noun.)

> We could find millions more examples that SRL BERT fails to understand because the sentences do not contain predicates.

We could apply this to questions in the middle of dialogue as well and still obtain no frames (outputs) from SRL BERT:

Context: The middle of a conversation during which person 2 did not want coffee:

`Q: So, tea?`

`A: No thanks.`

`Q: Ok, hot chocolate?`

`A: Nope.`

`Q: A glass of water?`

`A: Yup!`

We just saw a conversation with no frames, no semantic labeling. Nothing.

We can finish with some movie, concert, or exhibition reviews on social media that produce 0 frames:

- `Best movie ever!`
- `Worst concert in my life!`
- `Excellent exhibition!`

This section showed the limits of SRL. Let's now redefine SRL and show how to implement it.

Redefining SRL

SRL BERT presupposes that sentences contain predicates, which is a false assumption in many cases. Analyzing a sentence cannot be based on a predicate analysis alone.

A predicate contains a verb. The predicate tells us more about the subject. The following predicate contains a verb and additional information:

```
The dog ate his food quickly.
```

`ate...quickly` tells us more about the way the dog ate. However, a verb alone can be a predicate, as in:

```
Dogs eat.
```

The problem here resides in the fact that "verbs" and "predicates" are part of syntax and grammar analysis, not semantics.

Understanding how words fit together from a grammatical, functional point of view is restrictive.

Take this sentence that means absolutely nothing:

```
Globydisshing maccaked up all the tie.
```

SRL BERT perfectly performs "semantic" analysis on a sentence that means nothing:

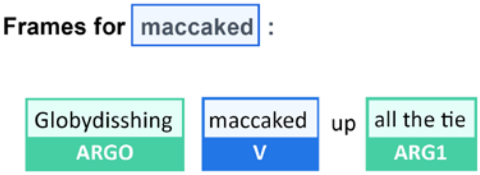


Figure 10.21: Analyzing a meaningless sentence

We can draw some conclusions from these examples:

- SRL predicate analysis only works when there is a verb in the sentence
- SRL predicate analysis cannot identify an ellipsis
- Predicates and verbs are part of the structure of a language, of grammatical analysis
- Predicate analysis identifies structures but not the meaning of a sentence
- Grammatical analysis goes far beyond predicate analysis as the necessary center of a sentence

Semantics focuses on the meaning of a phrase or sentence. Semantics focuses on context and the way words relate to each other.

Grammatical analysis includes syntax, inflection, and the functions of words in a phrase or sentence. The term semantic role labeling is misleading; it should be named "predicate role labeling."

We perfectly understand sentences without predicates and beyond sequence structure.

Sentiment analysis can decode the meaning of a sentence and give us an output without predicate analysis. Sentiment analysis algorithms perfectly understand that "Best movie ever" is positive regardless of the presence of a predicate or not.

> Using SRL alone to analyze language is restrictive. Using SRL in an AI pipeline or other AI tools can be very productive to add more intelligence to natural language understanding.

My recommendation is to use SRL with other AI tools, as we will see in *Chapter 13*, *Analyzing Fake News with Transformers*.

Let's now conclude our exploration of the scope and limits of SRL.

Summary

In this chapter, we explored SRL. SRL tasks are difficult for both humans and machines. Transformer models have shown that human baselines can be reached for many NLP topics to a certain extent.

We found that a simple BERT-based transformer can perform predicate sense disambiguation. We ran a simple transformer that could identify the meaning of a verb (predicate) without lexical or syntactic labeling. *Shi* and *Lin* (2019) used a standard *sentence + verb* input format to train their BERT-based transformer.

We found that a transformer trained with a stripped-down *sentence* + *predicate* input could solve simple and complex problems. The limits were reached when we used relatively rare verb forms. However, these limits are not final. If difficult problems are added to the training dataset, the research team could improve the model.

We also discovered that AI for the good of humanity exists. The *Allen Institute for AI* has made many free AI resources available. In addition, the research team has added visual representations to the raw output of NLP models to help users understand AI. We saw that explaining AI is as essential as running programs. The visual and text representations provided a clear view of the potential of the BERT-based model.

Finally, we explored the scope and limits of SRL to optimize how we will use this method with other AI tools.

Transformers will continue to improve the standardization of NLP through their distributed architecture and input formats.

In the next chapter, *Chapter 11*, *Let Your Data Do the Talking: Story, Questions, and Answers*, we will challenge transformers on tasks usually only humans perform well. We will explore the potential of transformers when faced with **Named Entity Recognition** (**NER**) and question-answering tasks.

Questions

1. **Semantic Role Labeling** (**SRL**) is a text generation task. (True/False)
2. A predicate is a noun. (True/False)
3. A verb is a predicate. (True/False)
4. Arguments can describe who and what is doing something. (True/False)
5. A modifier can be an adverb. (True/False)
6. A modifier can be a location. (True/False)
7. A BERT-based model contains encoder and decoder stacks. (True/False)
8. A BERT-based SRL model has standard input formats. (True/False)
9. Transformers can solve any SRL task. (True/False)

References

- *Peng Shi* and *Jimmy Lin*, 2019, *Simple BERT Models for Relation Extraction and Semantic Role Labeling*: `https://arxiv.org/abs/1904.05255`
- The Allen Institute for AI: `https://allennlp.org/`
- The Allen Institute for AI semantic role labeling resources: `https://demo.allennlp.org/semantic-role-labeling`

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the authors:

`https://www.packt.link/Transformers`

11

Let Your Data Do the Talking: Story, Questions, and Answers

Reading comprehension requires many skills. When we read a text, we notice the keywords and the main events and create mental representations of the content. We can then answer questions using our knowledge of the content and our representations. We also examine each question to avoid traps and making mistakes.

No matter how powerful they have become, transformers cannot answer open questions easily. An open environment means that somebody can ask any question on any topic, and a transformer would answer correctly. That is difficult but possible to some extent with GPT-3, as we will see in this chapter. However, transformers often use general domain training datasets in a closed question-and-answer environment. For example, critical answers in medical care and law interpretation will often require additional NLP functionality.

However, transformers cannot answer any question correctly regardless of whether the training environment is closed with preprocessed question-answer sequences. A transformer model can sometimes make wrong predictions if a sequence contains more than one subject and compound propositions.

This chapter will focus on methods to build a question generator that finds unambiguous content in a text with the help of other NLP tasks. The question generator will show some of the ideas applied to implement question-answering.

We will begin by showing how difficult it is to ask random questions and expect the transformer to respond well every time.

We will help a `DistilBERT` model answer questions by introducing **Named Entity Recognition** (**NER**) functions that suggest reasonable questions. In addition, we will lay the ground for a question generator for transformers.

We will add an ELECTRA model that was pretrained as a discriminator to our question-answering toolbox.

We will continue by adding **Semantic Role Labeling** (**SRL**) functions to the blueprint of the text generator.

Then, the *Next steps* section will provide additional ideas to build a reliable question-answering solution, including implementing the `Haystack` framework.

Finally, we will go straight to the GPT-3 Davinci engine interface online to explore question-answering tasks in an open environment. Again, no development, no training, and no preparation are required!

By the end of the chapter, you will see how to build your own multi-task NLP helpers or use Cloud AI for question-answering.

This chapter covers the following topics:

- The limits of random question-answering
- Using NER to create meaningful questions based on entity identification
- Beginning to design the blueprint of a question generator for transformers
- Testing the questions found with NER
- Introducing an ELECTRA encoder pretrained as a discriminator
- Testing the ELECTRA model with standard questions
- Using SRL to create meaningful questions based on predicate identification
- Project management guidelines to implement question-answering transformers
- Analyzing how to create a question generated using SRL
- Using the output of NER and SRL to define the blueprint of a question generator for transformers
- Exploring Haystack's question-answering framework with RoBERTa
- Using GPT-3's interface requires no development or preparation

Let's begin by going through the methodology we will apply to analyze the generation of questions for question-answering tasks.

Methodology

Question-answering is mainly presented as an NLP exercise involving a transformer and a dataset containing the ready-to-ask questions and answering those questions. The transformer is trained to answer the questions asked in this closed environment.

However, in more complex situations, reliable transformer model implementations require customized methods.

Transformers and methods

A perfect and efficient universal transformer model for question-answering or any other NLP task does not exist. The best model for a project is the one that produces the best outputs for a specific dataset and task.

The method outperforms models in many cases. For example, a suitable method with an average model often will produce more efficient results than a flawed method with an excellent model.

In this chapter, we will run `DistilBERT`, `ELECTRA`, and `RoBERTa` models. Some produce better *performances* than others.

However, *performance* does not guarantee a result in a critical domain.

For example, in a space rocket and spacecraft production project, asking an NLP bot a question means obtaining one exact answer.

Suppose the user needs to ask a question on a hundred-page report on the status of a regeneratively cooled nozzle and combustion chamber of a rocket. The question could be specific, such as `Is the cooling status reliable or not?` That is the bottom-line information the user wants from the NLP bot.

To cut a long story short, letting the NLP bot, transformer model or not, make a literal statistical answer with no quality and cognitive control is too risky and would not happen. A trustworthy NLP bot would be connected to a knowledge base containing data and rules to run a rule-based expert system in the background to check the NLP bot's answer. The NLP transformer model bot would produce a smooth, reliable natural language answer, possibly with a human voice.

A universal transformer *model* and *method* that will fit all needs does not exist. Each project requires specific functions and a customized approach and will vary tremendously depending on the users' expectations.

This chapter will focus on the general constraints of question-answering beyond a specific transformer model choice. This chapter is not a question-answering project guide but an introduction to how transformers can be used for question-answering.

We will focus on using question-answering in an open environment where the questions were not prepared beforehand. Transformer models require help from other NLP tasks and classical programs. We will explore some methods to give an idea of how to combine tasks to reach the goal of a project:

- *Method 0* explores a trial and error approach of asking questions randomly.
- *Method 1* introduces NER to help prepare the question-answering tasks.
- *Method 2* tries to help the default transformer with an ELECTRA transformer model. It also introduces SRL to help the transformer prepare questions.

The introduction to these three methods shows that a single question-answering method will not work for high-profile corporate projects. Adding NER and SRL will improve the linguistic intelligence of a transformer agent solution.

For example, in one of my first AI NLP projects implementing question-answering for a defense project in a tactical situation for an aerospace corporation, I combined different NLP methods to ensure that the answer provided was 100% reliable.

You can design a multi-method solution for each project you implement.

Let's start with the trial and error approach.

Method 0: Trial and error

Question-answering seems very easy. Is that true? Let's find out.

Open `QA.ipynb`, the Google Colab notebook we will be using in this chapter. We will run the notebook cell by cell.

Run the first cell to install Hugging Face's transformers, the framework we will be implementing in this chapter:

```
!pip install -q transformers
```

> Note: Hugging Face transformers continually evolve, updating libraries and modules to adapt to the market. If the default version doesn't work, you might have to pin one with `!pip install transformers==[version that runs with the other functions in the notebook]`.

We will now import Hugging Face's pipeline, which contains many ready-to-use transformer resources. They provide high-level abstraction functions for the Hugging Face library resources to perform a wide range of tasks. We can access those NLP tasks through a simple API. The program was created on Google Colab. It recommended to run it on Google Colab VM using a free Gmail account.

The `pipeline` is imported with one line of code:

```
from transformers import pipeline
```

Once that is done, we have one-line options to instantiate transformer models and tasks:

1. Perform an NLP task with the default `model` and default `tokenizer`:

   ```
   pipeline("<task-name>")
   ```

2. Perform an NLP task using a custom `model`:

   ```
   pipeline("<task-name>", model="<model_name>")
   ```

3. Perform NLP tasks using a custom `model` and a custom `tokenizer`:

   ```
   pipeline('<taskname>', model='<model name>', tokenizer='<tokenizer_
   name>')
   ```

Let's begin with the default model and tokenizer:

```
nlp_qa = pipeline('question-answering')
```

Now, all we have to do is provide a text that we will then use to submit questions to the transformer:

```
sequence = "The traffic began to slow down on Pioneer Boulevard in Los
Angeles, making it difficult to get out of the city. However, WBGO was
playing some cool jazz, and the weather was cool, making it rather
pleasant to be making it out of the city on this Friday afternoon. Nat
King Cole was singing as Jo, and Maria slowly made their way out of LA and
drove toward Barstow. They planned to get to Las Vegas early enough in the
evening to have a nice dinner and go see a show."
```

The sequence is deceptively simple, and all we need to do is plug one line of code into the API to ask a question and obtain an answer:

```
nlp_qa(context=sequence, question='Where is Pioneer Boulevard ?')
```

The output is a perfect answer:

```
{'answer': 'Los Angeles,', 'end': 66, 'score': 0.988201259751591, 'start':
55}
```

We have just implemented a question-answering transformer NLP task in a few lines of code! You could now download a ready-to-use dataset that contains texts, questions, and answers.

In fact, the chapter could end right here, and you would be all set for question-answering tasks. However, things are never simple in real-life implementations. Suppose we have to implement a question-answering transformer model for users to ask questions on many documents stored in the database. We have two significant constraints:

- We first need to run the transformer through a set of key documents and create questions that show that the system works
- We must show how we can guarantee that the transformer answers the questions correctly

Several questions immediately arise:

- Who is going to find the questions to ask to test the system?
- Even if an expert agrees to do the job, what will happen if many of the questions produce erroneous results?
- Will we keep training the model if the results are not satisfactory?
- What happens if some of the questions cannot be answered, no matter which model we use or train?
- What if this works on a limited sample but the process takes too long and cannot be scaled up because it costs too much?

If we just try questions that come to us with an expert's help and see which ones work and which ones don't, it could take forever. Trial and error is not the solution.

This chapter aims to provide some methods and tools that will reduce the cost of implementing a question-answering transformer model. *Finding good questions for question-answering is quite a challenge when implementing new datasets for a customer*.

We can think of a transformer as a LEGO® set of building blocks we can assemble as we see fit using encoder-only or decoder-only stacks. We can use a set of small, large, or **extra-large** (**XL**) transformer models.

We can also think of the NLP tasks we have explored in this book as a LEGO® set of solutions in a project we must implement. We can assemble two or more NLP tasks to reach our goals, just like any other software implementation. We can go from a trial and error search for questions to a methodical approach.

In this chapter:

- We will continue to run QA.ipynb cell by cell to explore the methods described in each section.
- We will also use the AllenNLP NER interface to obtain a visual representation of the NER and SRL results. You can enter the sentence in the interface by going to https://demo.allennlp.org/reading-comprehension, then select **Named Entity Recognition** or **Semantic Role Labeling,** and enter the sequence. In this chapter, we will take the AllenNLP model used into account. We just want to obtain visual representations.

Let's start by trying to find the right XL transformer model questions for question-answering with a NER-first method.

Method 1: NER first

This section will use NER to help us find ideas for good questions. Transformer models are continuously trained and updated. Also, the datasets used for training might change. Finally, these are not rule-based algorithms that produce the same result each time. The outputs might change from one run to another. NER can detect people, locations, organizations, and other entities in a sequence. We will first run a NER task that will give us some of the main parts of the paragraph we can focus on to ask questions.

Using NER to find questions

We will continue to run QA.ipynb cell by cell. The program now initializes the pipeline with the NER task to perform with the default model and tokenizer:

```
nlp_ner = pipeline("ner")
```

We will continue to use the deceptively simple sequence we ran in the *Method 0: Trial and error* section of this chapter:

```
sequence = "The traffic began to slow down on Pioneer Boulevard in Los
Angeles, making it difficult to get out of the city. However, WBGO was
playing some cool jazz, and the weather was cool, making it rather
pleasant to be making it out of the city on this Friday afternoon. Nat
King Cole was singing as Jo and Maria slowly made their way out of LA and
drove toward Barstow. They planned to get to Las Vegas early enough in the
evening to have a nice dinner and go see a show."
```

We run the nlp_ner cell in QA.ipynb:

```
print(nlp_ner(sequence))
```

The output generates the result of the NLP tasks. The scores were rounded up to two decimal places to fit the width of the page:

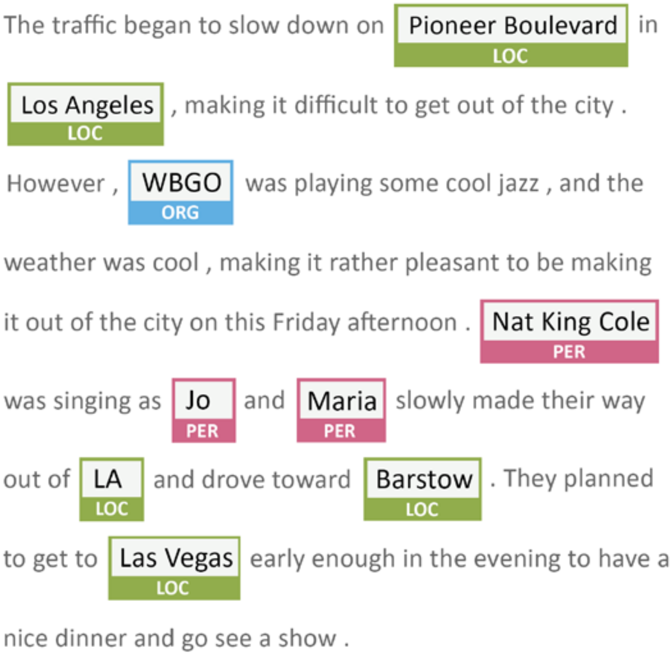
```
[{'word': 'Pioneer', 'score': 0.97, 'entity': 'I-LOC', 'index': 8},
{'word': 'Boulevard', 'score': 0.99, 'entity': 'I-LOC', 'index': 9},
{'word': 'Los', 'score': 0.99, 'entity': 'I-LOC', 'index': 11},
{'word': 'Angeles', 'score': 0.99, 'entity': 'I-LOC', 'index': 12},
{'word': 'W', 'score': 0.99, 'entity': 'I-ORG', 'index': 26},
{'word': '##B', 'score': 0.99, 'entity': 'I-ORG', 'index': 27},
{'word': '##G', 'score': 0.98, 'entity': 'I-ORG', 'index': 28},
{'word': '##O', 'score': 0.97, 'entity': 'I-ORG', 'index': 29},
{'word': 'Nat', 'score': 0.99, 'entity': 'I-PER', 'index': 59},
{'word': 'King', 'score': 0.99, 'entity': 'I-PER', 'index': 60},
{'word': 'Cole', 'score': 0.99, 'entity': 'I-PER', 'index': 61},
{'word': 'Jo', 'score': 0.99, 'entity': 'I-PER', 'index': 65},
{'word': 'Maria', 'score': 0.99, 'entity': 'I-PER', 'index': 67},
{'word': 'LA', 'score': 0.99, 'entity': 'I-LOC', 'index': 74},
{'word': 'Bar', 'score': 0.99, 'entity': 'I-LOC', 'index': 78},
{'word': '##sto', 'score': 0.85, 'entity': 'I-LOC', 'index': 79},
{'word': '##w', 'score': 0.99, 'entity': 'I-LOC', 'index': 80},
{'word': 'Las', 'score': 0.99 'entity': 'I-LOC', 'index': 87},
{'word': 'Vegas', 'score': 0.9989519715309143, 'entity': 'I-LOC', 'index':
88}]
```

The documentation of Hugging Face describes the labels used. In our case, the main ones are:

- `I-PER`, the name of a person
- `I-ORG`, the name of an organization
- `I-LOC`, the name of a location

The result is correct. Note that `Barstow` was split into three tokens.

Let's run the same sequence on `AllenNLP` in the **Named Entity Recognition** section (`https://demo.allennlp.org/named-entity-recognition`) to obtain a visual representation of our sequence:



The traffic began to slow down on Pioneer Boulevard LOC in
Los Angeles LOC , making it difficult to get out of the city .
However , WBGO ORG was playing some cool jazz , and the
weather was cool , making it rather pleasant to be making
it out of the city on this Friday afternoon . Nat King Cole PER
was singing as Jo PER and Maria PER slowly made their way
out of LA LOC and drove toward Barstow LOC . They planned
to get to Las Vegas LOC early enough in the evening to have a
nice dinner and go see a show .

Figure 11.1: NER

We can see that NER has highlighted the key entities we will use to create questions for question-answering.

Let's ask our transformer two types of questions:

- Questions related to locations
- Questions related to persons

Let's begin with location questions.

Location entity questions

`QA.ipynb` produced nearly `20` entities. The location entities are particularly interesting:

```
[{'word': 'Pioneer', 'score': 0.97, 'entity': 'I-LOC', 'index': 8},
{'word': 'Boulevard', 'score': 0.99, 'entity': 'I-LOC', 'index': 9},
{'word': 'Los', 'score': 0.99, 'entity': 'I-LOC', 'index': 11},
{'word': 'Angeles', 'score': 0.99, 'entity': 'I-LOC', 'index': 12},
{'word': 'LA', 'score': 0.99, 'entity': 'I-LOC', 'index': 74},
{'word': 'Bar', 'score': 0.99, 'entity': 'I-LOC', 'index': 78},
{'word': '##sto', 'score': 0.85, 'entity': 'I-LOC', 'index': 79},
{'word': '##w', 'score': 0.99, 'entity': 'I-LOC', 'index': 80},
{'word': 'Las', 'score': 0.99 'entity': 'I-LOC', 'index': 87},
{'word': 'Vegas', 'score': 0.9989519715309143, 'entity': 'I-LOC', 'index':
88}]
```

Applying heuristics

We can apply heuristics, a method, to create questions with the output `QA.ipynb` generated:

- Merge the locations back into their original form with a parser
- Apply a template to the locations

It is beyond the scope of this book to write classical code for a project. We could write a function that would do the work for us, as shown in this pseudocode:

```
for i in range beginning of output to end of the output:
    filter records containing I-LOC
    merge the I-LOCs that fit together
    save the merged I-LOCs for questions-answering
```

The NER output would become:

- `I-LOC`, `Pioneer Boulevard`
- `I-LOC`, `Los Angeles`
- `I-LOC`, `LA`
- `I-LOC`, `Barstow`
- `I-LOC`, `Las Vegas`

We could then generate questions automatically with two templates. For example, we could apply a random function. We could write a function that would do the job for us, as shown in the following pseudocode:

```
from the first location to the last location:
    choose randomly:
        Template 1: Where is [I-LOC]?
        Template 2: Where is [I-LOC] located?
```

We would obtain five questions automatically. For example:

```
Where is Pioneer Boulevard?
Where is Los Angeles located?
Where is LA?
Where is Barstow?
Where is Las Vegas located?
```

We know that some of these questions cannot be directly answered with the sequence we created. But we can also manage that automatically. Suppose the questions were created automatically with our method:

1. Enter a sequence
2. Run NER
3. Create the questions automatically

Let's suppose the questions were created automatically and let's run them:

```
nlp_qa = pipeline('question-answering')
print("Question 1.",nlp_qa(context=sequence, question='Where is Pioneer
Boulevard ?'))
print("Question 2.",nlp_qa(context=sequence, question='Where is Los
Angeles located?'))
print("Question 3.",nlp_qa(context=sequence, question='Where is LA ?'))
print("Question 4.",nlp_qa(context=sequence, question='Where is Barstow
?'))
print("Question 5.",nlp_qa(context=sequence, question='Where is Las Vegas
located ?'))
```

The output shows that only `Question 1` was answered correctly:

```
Question 1. {'score': 0.9879662851935791, 'start': 55, 'end': 67,
'answer': 'Los Angeles,'}
Question 2. {'score': 0.9875189033668121, 'start': 34, 'end': 51,
'answer': 'Pioneer Boulevard'}
Question 3. {'score': 0.5090435442006118, 'start': 55, 'end': 67,
'answer': 'Los Angeles,'}
Question 4. {'score': 0.3695214621538554, 'start': 387, 'end': 396,
'answer': 'Las Vegas'}
Question 5. {'score': 0.21833994202792262, 'start': 355, 'end': 363,
'answer': 'Barstow.'}
```

The output displays the `score`, the `start` and `end` position of the answer, and the `answer` itself. The `score` of `Question 2` is `0.98` in this run, although it wrongly states that `Los Angeles` in `Pioneer Boulevard`.

What do we do now?

It's time to control transformers with project management to add quality and decision-making functions.

Project management

We will examine four examples, among many others, of how to manage the transformer and the hard-coded functions that manage it automatically. We will classify these four project management examples into four project levels: easy, intermediate, difficult, and very difficult. Project management is not in the scope of this book, so we will briefly go through these four categories:

1. **An easy project** could be a website for an elementary school. A teacher might be delighted by what we have seen. The text could be displayed on an HTML page. The five answers to the questions we obtained automatically could be merged with some development into five assertions in a fixed format: `I-LOC is in I-LOC` (for example, `Barstow is in California`). We then add `(True, False)` under each assertion. All the teacher would have to do would be to have an administrator interface that allows the teacher to click on the right answers to finalize a multiple-choice questionnaire!
2. **An intermediate project** could be to encapsulate the transformer's automatic questions and answers in a program that uses an API to check the answers and correct them automatically. The user would see nothing. The process is seamless. The wrong answers the transformer made would be stored for further analysis.
3. **A difficult project** would be implementing an intermediate project in a chatbot with follow-up questions. For example, the transformer correctly places `Pioneer Boulevard` in `Los Angeles`. A chatbot user might ask a natural follow-up question such as `near where in LA?`. This requires more development.
4. **A very difficult project** would be a research project that would train the transformer to recognize `I-LOC` entities over millions of records in datasets and output results of real-time streaming of map software APIs.

The good news is that we can also find a way to use what we find.

The bad news is that implemented transformers or any AI in real-life projects require powerful machines and a tremendous amount of teamwork between project managers, **Subject Matter Experts (SMEs)**, developers, and end users.

Let's now try person entity questions.

Person entity questions

Let's start with an easy question for the transformer:

```
nlp_qa = pipeline('question-answering')
nlp_qa(context=sequence, question='Who was singing ?')
```

The answer is correct. It states who in the sequence was singing:

```
{'answer': 'Nat King Cole,'
 'end': 277,
 'score': 0.9653632081862433,
 'start': 264}
```

We will now ask the transformer a question that requires some thinking because it is not clearly stated:

```
nlp_qa(context=sequence, question='Who was going to Las Vegas ?')
```

It is impossible to answer that question without taking the sentence apart. The transformer makes a big mistake:

```
{'answer': 'Nat King Cole,'
 'end': 277,
 'score': 0.3568152742800521,
 'start': 264}
```

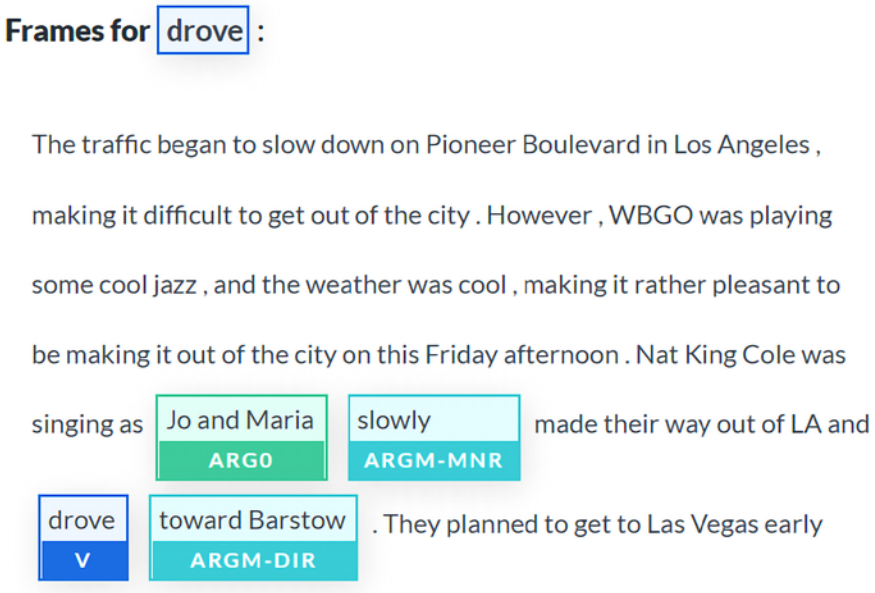
The transformer is honest enough to display a `score` of only `0.35`. This `score` might vary from one calculation to another or from one transformer model to another. We can see that the transformer faced a semantic labeling problem. Let's try to do better with person entity questions applying an SRL-first method.

Method 2: SRL first

The transformer could not find who was driving to go to `Las Vegas` and thought it was from `Nat King Cole` instead of `Jo` and `Maria`.

What went wrong? Can we see what the transformers think and obtain an explanation? To find out, let's go back to semantic role modeling. If necessary, take a few minutes to review *Chapter 10, Semantic Role Labeling with BERT-Based Transformers*.

Let's run the same sequence on `AllenNLP` in the **Semantic Role Labeling** section, `https://demo.allennlp.org/semantic-role-labeling`, to obtain a visual representation of the verb `drove` in our sequence by running the SRL BERT model we used in the previous chapter:



Frames for drove **:**

The traffic began to slow down on Pioneer Boulevard in Los Angeles , making it difficult to get out of the city . However , WBGO was playing some cool jazz , and the weather was cool , making it rather pleasant to be making it out of the city on this Friday afternoon . Nat King Cole was singing as Jo and Maria slowly made their way out of LA and drove toward Barstow . They planned to get to Las Vegas early

Figure 11.2: SRL run on the text

SRL BERT found 19 frames. In this section, we focus on `drove`.

Note: The results may vary from one run to another or when AllenNLP updates the model versions.

We can see the problem. The argument of the verb `drove` is `Jo and Maria`. It seems that the inference could be made.

Keep in mind that transformer models keep evolving. The output might vary; however, the concepts remain the same.

Is that true? Let's ask the question in `QA.ipynb`:

```
nlp_qa(context=sequence, question='Who are they?')
```

The output is correct:

```
{'answer': 'Jo and Maria',
 'end': 305,
 'score': 0.8486017557290779,
 'start': 293}
```

Could we find a way to ask the question to obtain the right answer? We will try by paraphrasing the question:

```
nlp_qa(context=sequence, question='Who drove to Las Vegas?')
```

We obtain a somewhat better result:

```
{'answer': 'Nat King Cole was singing as Jo and Maria',
 'end': 305,
 'score': 0.35940926070820467,
 'start': 264}
```

The transformer now understands that `Nat King Cole` was `singing` and that `Jo and Maria` were doing something in the meantime.

We still need to go further and find a way to ask better questions.

Let's try another model.

Question-answering with ELECTRA

Before switching models, we need to know which one we are using:

```
print(nlp_qa.model)
```

The output first shows that the model is a DistilBERT model trained on question-answering:

```
DistilBertForQuestionAnswering((distilbert): DistilBertModel(
```

The model has `6` layers and `768` features, as shown in layer `6` (the layers are numbered from `0` to `n`):

```
(5): TransformerBlock(
          (attention): MultiHeadSelfAttention(
            (dropout): Dropout(p=0.1, inplace=False)
            (q_lin): Linear(in_features=768, out_features=768, bias=True)
            (k_lin): Linear(in_features=768, out_features=768, bias=True)
            (v_lin): Linear(in_features=768, out_features=768, bias=True)
            (out_lin): Linear(in_features=768, out_features=768,
bias=True)
```

We will now try the `ELECTRA` transformer model. *Clark* et al. (2020) designed a transformer model that improved the **Masked Language Modeling** (**MLM**) pretraining method.

In *Chapter 3*, *Fine-Tuning BERT Models*, in the *Masked language modeling* subsection, we saw that the BERT model inserts random masked tokens with `[MASK]` during the training process.

Clark et al. (2020) introduced plausible alternatives with a generator network rather than simply using random tokens. BERT models are trained to predict the identities of the (masked) corrupted tokens. *Clark* et al. (2020) trained an ELECTRA model as a discriminator to predict whether the masked token was a generated token or not. *Figure 11.3* shows how ELECTRA is trained:

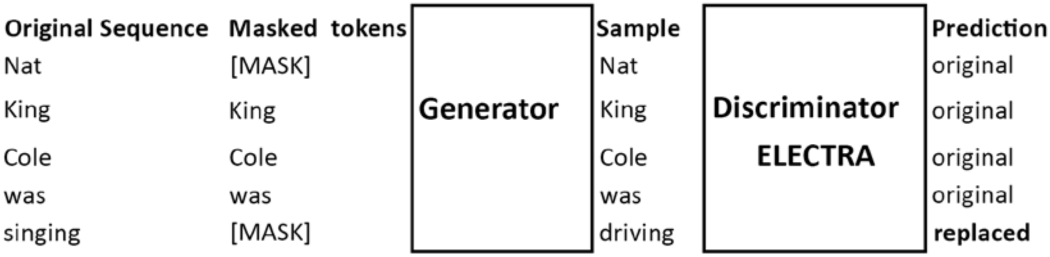


Figure 11.3: ELECTRA is trained as a discriminator

Figure 11.3 shows that the original sequence is masked before going through the generator. The generator inserts *acceptable* tokens and not random tokens. The ELECTRA transformer model is trained to predict whether a token comes from the original sequence or has been replaced.

The architecture of an ELECTRA transformer model and most of its hyperparameters are the same as BERT transformer models.

We now want to see if we can obtain a better result. The next cell to run in `QA.ipynb` is the question-answering cell with an `ELECTRA-small-generator`:

```
nlp_qa = pipeline('question-answering', model='google/electra-small-
generator', tokenizer='google/electra-small-generator')
nlp_qa(context=sequence, question='Who drove to Las Vegas ?')
```

The output is not what we expect:

```
{'answer': 'to slow down on Pioneer Boulevard in Los Angeles, making it
difficult to',
 'end': 90,
 'score': 2.5295573154019736e-05,
 start': 18}
```

The output might change from one run or transformer model to another; however, the idea remains the same.

The output also sends training messages:

```
- This IS expected if you are initializing ElectraForQuestionAnswering
from the checkpoint of a model trained on another task or with another
architecture..
- This IS NOT expected if you are initializing ElectraForQuestionAnswering
from the checkpoint of a model that you expect to be exactly identical..
```

You might not like these warning messages and might conclude that this is a bad model. But always explore every avenue that is offered to you. ELECTRA might require more training, of course. But *experiment* as much as possible to find new ideas! Then you can decide to train a model further or move on to another one.

We must now think of the next steps to take.

Project management constraints

We have not obtained the results we expected with the default DistilBERT and ELECTRA transformer models.

There are three main options among other solutions:

- Train DistilBERT and ELECTRA or other models with additional datasets. Training datasets is a costly process in real-life projects. The training could last months if new datasets need to be implemented and hyperparameters changed. The hardware cost needs to be taken into account as well. Furthermore, if the result is unsatisfactory, a project manager might shut the project down.
- You can also try ready-to-use transformers, although they might not fit your needs, such as the Hugging Face model: `https://huggingface.co/transformers/usage.html#extractive-question-answering`.
- Find a way to obtain better results by using additional NLP tasks to help the question-answering model.

In this chapter, we will focus on finding additional NLP tasks to help the default DistilBERT model.

Let's use SRL to extract the predicates and their arguments.

Using SRL to find questions

AllenNLP uses the BERT-based model we implemented in the `SRL.ipynb` notebook in *Chapter 10, Semantic Role Labeling with BERT-Based Transformers*.

Let's rerun the sequence on AllenNLP in the **Semantic Role Labeling** section, `https://demo.allennlp.org/semantic-role-labeling`, to obtain a visual representation of the predicates in the sequence.

We will enter the sequence we have been working on:

```
The traffic began to slow down on Pioneer Boulevard in Los Angeles, making
it difficult to get out of the city. However, WBGO was playing some cool
jazz, and the weather was cool, making it rather pleasant to be making it
out of the city on this Friday afternoon. Nat King Cole was singing as Jo
and Maria slowly made their way out of LA and drove toward Barstow. They
planned to get to Las Vegas early enough in the evening to have a nice
dinner and go see a show.
```

The BERT-based model found several predicates. Our goal is to find the properties of SRL outputs that could automatically generate questions based on the verbs in a sentence.

We will first list the predicate candidates produced by the BERT model:

```
verbs={"began," "slow," "making"(1), "playing," "making"(2), "making"(3),
"singing,",…, "made," "drove," "planned," go," see"}
```

If we had to write a program, we could start by introducing a verb counter, as shown in the following pseudocode:

```
def maxcount:
for in range first verb to last verb:
    for each verb
       counter +=1
       if counter>max_count, filter verb
```

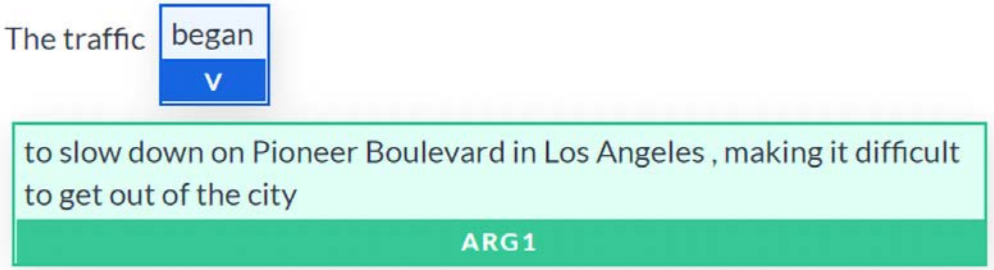
If the counter exceeds the number of acceptable occurrences (`max_count`), the verb will be excluded in this experiment. Without further development, it will be too difficult to disambiguate multiple semantic roles of the verb's arguments.

Let's take `made`, which is the past tense of `make`, out of the list as well.

Our list is now limited to:

```
verbs={"began," "slow," "playing," "singing," "drove," "planned," go,"
see"}
```

If we continued to write a function to filter the verbs, we could look for verbs with lengthy arguments. The verb began has a very long argument:



The traffic began

to slow down on Pioneer Boulevard in Los Angeles , making it difficult to get out of the city

Figure 11.4: SRL applied to the verb "began"

The argument of began is so long it doesn't fit in the screenshot. The text version shows how difficult it would be to interpret the argument of began:

```
began: The traffic [V: began] [ARG1: to slow down on Pioneer Boulevard in
Los Angeles , making it difficult to get out of the city] . However , WBGO
was playing some cool jazz] , and the weather was cool , making it rather
pleasant to be making it out of the city on this Friday afternoon . Nat
King Cole was singing as Jo and Maria slowly made their way out of LA and
drove toward Barstow . They planned to get to Las Vegas early enough in
the evening to have a nice dinner and go see a show .
```

We could add a function to filter verbs that contain arguments that exceed a maximum length:

```
def maxlength:
for in range first verb to last verb:
    for each verb
       if length(argument of verb)>max_length, filter verb
```

If the length of one verb's arguments exceeds a maximum length (`max_length`), the verb will be excluded in this experiment. For the moment, let's just take began out of the list:

Our list is now limited to:

```
verbs={ "slow", "playing", "singing", "drove",    "planned"," go"," see"}
```

We could add more exclusion rules depending on the project we are working on. We can also call the maxlength function again with a very restrictive max_length value to extract potentially interesting candidates for our automatic question generator. The verb candidates with the shortest arguments could be transformed into questions. The verb slow fits the three rules we set: it appears only once in the sequence, the arguments are not too long, and it contains some of the shortest arguments in the sequence. The AllenNLP visual representation confirms our choice:

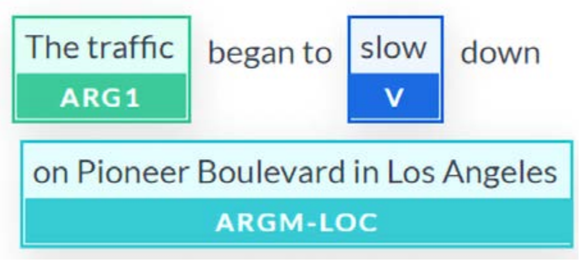


Figure 11.5: SRL applied to the verb "slow"

The text output can be easily parsed:

```
slow: [ARG1: The traffic] began to [V: slow] down [ARG1: on] [ARGM-ADV:
Pioneer Boulevard] [ARGM-LOC: in Los Angeles] , [ARGM-ADV: making it
difficult to get out of the city] .
```

This result and the following outputs may vary with the ever-evolving transformer models, but the idea remains the same. The verb slow is identified and this is the key aspect of this SRL output.

We could automatically generate the what template. We will not generate a who template because none of the arguments were labeled I-PER (person). We could write a function that manages these two possibilities, as shown in the following pseudocode:

```
def whowhat:
   if NER(ARGi)==I-PER, then:
        template=Who is [VERB]
   if NER(ARGi)!=I-PER, then:
     template=What is [VERB]
```

This function would require more work to deal with verb forms and modifiers. However, in this experiment, we will just apply the function and generate the following question:

```
What is slow?
```

Let's run the default `pipeline` with the following cell:

```
nlp_qa = pipeline ('question-answering')
nlp_qa(context= sequence, question='What was slow?')
```

The result is satisfactory:

```
{'answer': 'The traffic',
'end': 11,
'score': 0.4652545872921081,
'start': 0}
```

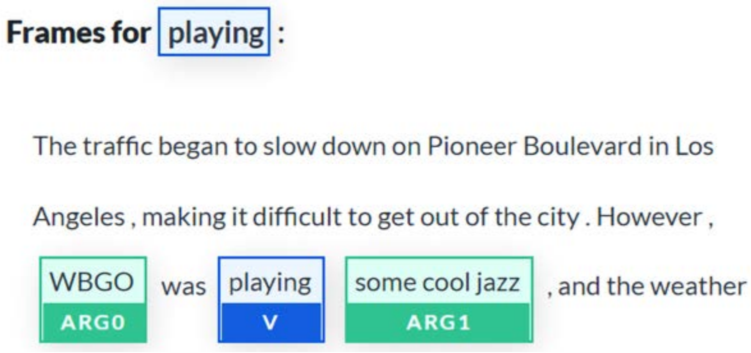
The default model, in this case, `DistilBERT`, correctly answered the question.

Our automatic question generator can do the following:

- Run NER automatically
- Parse the results with classical code
- Generate entity-only questions
- Run SRL automatically
- Filter the results with rules
- Generate SRL-only questions using the NER results to determine which template to use

This solution is by no means complete. More work needs to be done and probably requires additional NLP tasks and code. However, it gives an idea of the hard work implementing AI, in any form, requires.

Let's try our approach with the next filter verb: `playing`. The visual representation shows that the arguments are `WBGO` and `some cool jazz`:



Frames for playing :

The traffic began to slow down on Pioneer Boulevard in Los Angeles , making it difficult to get out of the city . However , WBGO was playing some cool jazz , and the weather

Figure 11.6: SRL applied to the verb "playing"

The text version is easy to parse:

```
playing: The traffic began to slow down on Pioneer Boulevard in Los
Angeles , making it difficult to get out of the city . [ARGM-DIS: However]
, [ARG0: WBGO] was [V: playing] [ARG1: some cool jazz]
```

This result and the following outputs may vary with the ever-evolving transformer models, but the idea remains the same: identifying the verb and its arguments.

If we ran the whowhat function, it would show that there is no I-PER in the arguments. The template chosen will be the what template, and the following question could be generated automatically:

```
What is playing?
```

Let's run the default pipeline with this question in the following cell:

```
nlp_qa = pipeline('question-answering')
nlp_qa(context=sequence, question='What was playing')
```

The output is also satisfactory:

```
{'answer': 'cool jazz,,'
 'end': 153,
 'score': 0.35047012837950753,
 'start': 143}
```

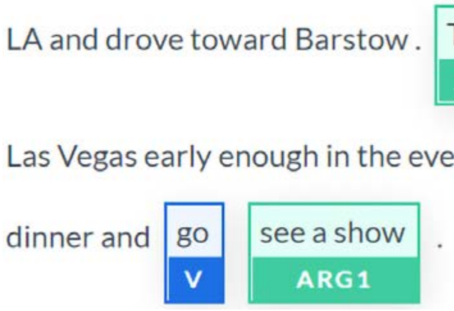
singing is a good candidate, and the whowhat function would find the I-PER template and automatically generate the following question:

```
Who is singing?
```

We have already successfully tested this question in this chapter.

The next verb is drove, which we have already tagged as a problem. The transformer cannot solve this problem.

The verb go is a good candidate:



LA and drove toward Barstow . They planned to get to Las Vegas early enough in the evening to have a nice dinner and go see a show .

Figure 11.7: SRL applied to the verb "go"

It would take additional development to produce a template with the correct verb form. Let's suppose the work was done and ask the model the following question:

```
nlp_qa = pipeline('question-answering')
nlp_qa(context=sequence, question='Who sees a show?')
```

The output is the wrong argument:

```
{'answer': 'Nat King Cole,'
 'end': 277,
 'score': 0.5587267250683112,
 'start': 264}
```

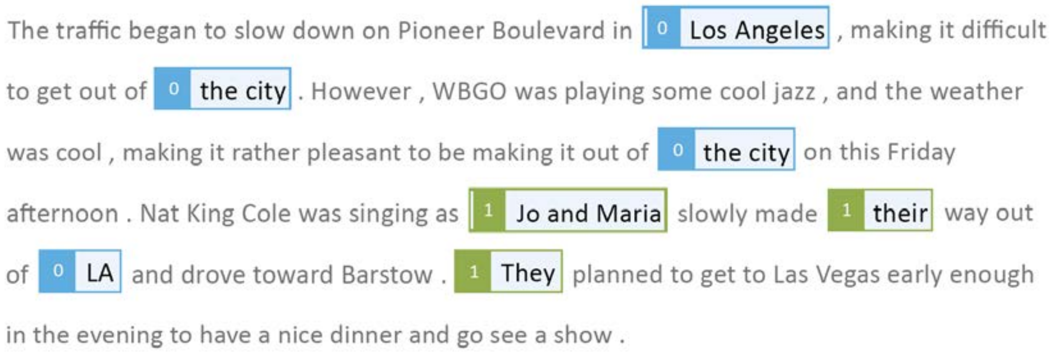
We can see that the presence of `Nat King Cole` and `Jo` and `Maria` in the same sequence in a complex sequence creates disambiguation problems for transformer models and any NLP model. More project management and research would be required.

Next steps

There is no easy way to implement question-answering or shortcuts. We began to implement methods that could generate questions automatically. Automatic question generation is a critical aspect of NLP.

More transformer models need to be pretrained with multi-task datasets containing NER, SRL, and question-answering problems to solve. Project managers also need to learn how to combine several NLP tasks to help solve a specific task, such as question-answering.

Coreference resolution, `https://demo.allennlp.org/coreference-resolution`, could have helped our model identify the main subjects in the sequence we worked on. This result produced with `AllenNLP` shows an interesting analysis:



The traffic began to slow down on Pioneer Boulevard in 0 Los Angeles , making it difficult to get out of 0 the city . However , WBGO was playing some cool jazz , and the weather was cool , making it rather pleasant to be making it out of 0 the city on this Friday afternoon . Nat King Cole was singing as 1 Jo and Maria slowly made 1 their way out of 0 LA and drove toward Barstow . 1 They planned to get to Las Vegas early enough in the evening to have a nice dinner and go see a show .

Figure 11.8: Coreference resolution of a sequence

We could continue to develop our program by adding the output of coreference resolution:

```
Set0={'Los Angeles', 'the city,' 'LA'}
Set1=[Jo and Maria, their, they}
```

We could add coreference resolution as a pretraining task or add it as a post-processing task in the question generator. In any case, question generators that simulate human behavior can considerably enhance the performance of question-answering tasks. We will include more customized additional NLP tasks in the pretraining process of question-answering models.

Of course, we can decide to use new strategies to pretrain the models we ran in this chapter, such as DistilBERT and ELECTRA, and then let users ask the questions they wish. I recommend both approaches:

- Work on question generators for question-answering tasks. These questions can be used for educational purposes, train transformers, or even provide ideas for real-time users.
- Work on pretraining transformer models by including specific NLP tasks, which will improve their question-answering performance. Use the question generator to train it further.

Exploring Haystack with a RoBERTa model

`Haystack` is a question-answering framework with interesting functionality. It is worth exploring to see if it might fit your needs for a given project.

In this section, we will run question-answering on the sentence we experimented with using other models and methods in this chapter.

Open Haystack_QA_Pipeline.ipynb.

The first cell installs the modules necessary to run Haystack:

```
# Install Haystack
!pip install farm-haystack==0.6.0
# Install specific versions of urllib and torch to avoid conflicts with
preinstalled versions on Colab
!pip install urllib3==1.25.4
!pip install torch==1.6.0+cu101-f https://download.pytorch.org/whl/torch_
stable.html
```

The notebook uses a RoBERTa model:

```
# Load a  local model or any of the QA models on Hugging Face's model hub
(https://huggingface.co/models)
from haystack.reader.farm import FARMReader
reader = FARMReader(model_name_or_path="deepset/roberta-base-squad2", use_
gpu=True, no_ans_boost=0, return_no_answer=False)
```

You can go back to *Chapter 4*, *Pretraining a RoBERTa Model from Scratch*, for a general description of a RoBERTa model.

The remaining cells of the notebook will answer questions on the text we have been exploring in detail in this chapter:

```
text = "The traffic began to slow down on Pioneer Boulevard in…/… have a
nice dinner and go see a show."
```

You can compare the answers obtained with the previous sections' outputs and decide which transformer model you would like to implement.

Exploring Q&A with a GTP-3 engine

This section will try to avoid training, fine-tuning, loading a program on a server, or even using a dataset. Instead, a user can simply connect to their OpenAI account and use the interactive educational interface.

A GPT-3 engine online educational interface will provide sufficiently good answers by providing *E* (explanation) and *T* (text) as follows:

E = Answer questions from this text

T = `The traffic began to slow down on Pioneer Boulevard in…/… have a nice dinner and go see a show.`

Here are some questions asked and answers obtained in the form of question-answer:

- `Who is going to Las Vegas?: Jo and Maria`
- `Who was singing?: Nat King Cole`
- `What kind of music was playing?: jazz`
- `What was the plan for the evening?: to have a nice dinner and go see a show`

That's it! That's all you need to do to run a wide range of educational NLP tasks online with an interactive interface even without an API with GPT-3 engines.

You can change *S* (showing GPT-3 what is expected) and *E* and create endless interactions. The next generation of NLP is born! An Industry 4.0 developer, consultant, or project manager will need to acquire a new set of skills: cognitive approaches, linguistics, psychology, and other cross-disciplinary dimensions. If necessary, you can take your time and go back to *Chapter 7, The Rise of Suprahuman Transformers with GPT-3 Engines*.

We have explored some critical aspects of the use of question-answering with transformers. Let's sum up the work we have done.

Summary

In this chapter, we found that question-answering isn't as easy as it seems. Implementing a transformer model only takes a few minutes. However, getting it to work can take a few hours or several months!

We first asked the default transformer in the Hugging Face pipeline to answer some simple questions. `DistilBERT`, the default transformer, answered the simple questions quite well. However, we chose easy questions. In real life, users ask all kinds of questions. The transformer can get confused and produce erroneous output.

We then decided to continue to ask random questions and get random answers, or we could begin to design the blueprint of a question generator, which is a more productive solution.

We started by using NER to find useful content. We designed a function that could automatically create questions based on NER output. The quality was promising but required more work.

We tried an ELECTRA model that did not produce the results we expected. We stopped for a few minutes to decide if we would spend costly resources to train transformer models or design a question generator.

We added SRL to the blueprint of the question generator and tested the questions it could produce. We also added NER to the analysis and generated several meaningful questions. The `Haystack` framework was also introduced to discover other ways of addressing question-answering with RoBERTa.

Finally, we ran an example using a GPT-3 engine directly in the OpenAI educational interactive interface without an API. Cloud AI platforms are increasing in power and accessibility.

Our experiments led to one conclusion: multi-task transformers will provide better performance on complex NLP tasks than a transformer trained on a specific task. Implementing transformers requires well-prepared multi-task training, heuristics in classical code, and a question generator. The question generator can be used to train the model further by using the questions as training input data or as a standalone solution.

In the next chapter, *Detecting Customer Emotions to Make Predictions*, we will explore how to implement sentiment analysis on social media feedback.

Questions

1. A trained transformer model can answer any question. (True/False)
2. Question-answering requires no further research. It is perfect as it is. (True/False)
3. **Named Entity Recognition** (**NER**) can provide useful information when looking for meaningful questions. (True/False)
4. **Semantic Role Labeling** (**SRL**) is useless when preparing questions. (True/False)
5. A question generator is an excellent way to produce questions. (True/False)
6. Implementing question answering requires careful project management. (True/False)
7. ELECTRA models have the same architecture as GPT-2. (True/False)
8. ELECTRA models have the same architecture as BERT but are trained as discriminators. (True/False)
9. NER can recognize a location and label it as `I-LOC`. (True/False)
10. NER can recognize a person and label that person as `I-PER`. (True/False)

References

- *The Allen Institute* for AI: `https://allennlp.org/`
- *The Allen Institute* for reading comprehension resources: `https://demo.allennlp.org/reading-comprehension`
- *Kevin Clark*, *Minh-Thang Luong*, *Quoc V. Le*, *Christopher D. Manning*, 2020, *ELECTRA: Pretraining Text Encoders as Discriminators Rather Than Generators*: `https://arxiv.org/abs/2003.10555`
- Hugging Face pipelines: `https://huggingface.co/transformers/main_classes/pipelines.html`
- GitHub Haystack framework repository: `https://github.com/deepset-ai/haystack/`

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the authors:

`https://www.packt.link/Transformers`

12

Detecting Customer Emotions to Make Predictions

Sentiment analysis relies on the principle of compositionality. How can we understand a whole sentence if we cannot understand parts of a sentence? Is this tough task possible for NLP transformer models? We will try several transformer models in this chapter to find out.

We will start with the **Stanford Sentiment Treebank** (**SST**). The SST provides datasets with complex sentences to analyze. It is easy to analyze sentences such as `The movie was great`. However, what happens if the task becomes very tough with complex sentences such as `Although the movie was a bit too long, I really enjoyed it.`? This sentence is segmented. It forces a transformer model to understand the structure of the sequence and its logical form.

We will then test several transformer models with complex sentences and simple sentences. We will find that no matter which model we try, it will not work if it isn't trained enough. Transformer models are like us. They are students that need to work hard to learn and try to reach real-life human baselines.

Running DistilBERT, RoBERTa-large, BERT-base, MiniLM-L12-H84-uncased, and BERT-base multilingual models is fun! However, we will discover that some of these students require more training, just like we would.

Along the way, we will see how to use the output of the sentiment tasks to improve customer relationships and see a nice five-star interface you could implement on your website.

Finally, we will use GPT-3's online interface for sentiment analysis with an OpenAI account. No AI development or API is required!

This chapter covers the following topics:

- The SST for sentiment analysis
- Defining compositionality for long sequences
- Sentiment analysis with AllenNLP (RoBERTa)
- Running complex sentences to explore the new frontier of transformers
- Using Hugging Face sentiment analysis models
- DistilBERT for sentiment analysis
- Experimenting with MiniLM-L12-H384-uncased
- Exploring RoBERTa-large-mnli
- Looking into a BERT-base multilingual model
- Sentiment analysis with GPT-3

Let's begin by going through the SST.

Getting started: Sentiment analysis transformers

This section will first explore the SST that the transformers will use to train models on sentiment analysis.

We will then use AllenNLP to run a RoBERTa-large transformer.

The Stanford Sentiment Treebank (SST)

Socher et al. (2013) designed semantic word spaces over long phrases. They defined principles of *compositionality* applied to long sequences. The principle of compositionality means that an NLP model must examine the constituent expressions of a complex sentence and the rules that combine them to understand the meaning of a sequence.

Let's take a sample from the SST to grasp the meaning of the principle of compositionality.

This section and chapter are self-contained, so you can choose to perform the actions described or read the chapter and view the screenshots provided.

Go to the interactive sentiment treebank: `https://nlp.stanford.edu/sentiment/treebank.html?na=3&nb=33`.

You can make the selections you wish. Graphs of sentiment trees will appear on the page. Click on an image to obtain a sentiment tree:

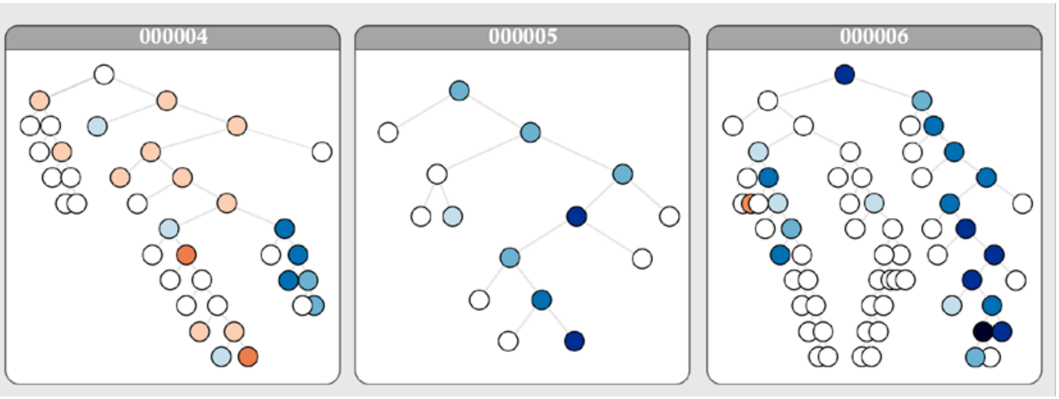


Figure 12.1: Graphs of sentiment trees

For this example, I clicked on graph number 6, which contains a sentence mentioning `Jacques Derrida, a pioneer in deconstruction theories in linguistics`. A long, complex sentence appears:

`Whether or not you're enlightened by any of Derrida's lectures on the other and the self, Derrida is an undeniably fascinating and playful fellow.`

Socher et al. (2013) worked on compositionality in vector spaces and logic forms.

For example, defining the rule of logic that governs the Jacques Derrida sample implies an understanding of the following:

- How the words `Whether`, `or`, and `not` and the comma that separates the `Whether` phrase from the rest of the sentence can be interpreted
- How to understand the second part of the sentence after the comma with yet another `and`!

Once the vector space was defined, *Socher* et al. (2013) could produce complex graphs representing the principle of compositionality.

We can now view the graph section by section. The first section is the `Whether` segment of the sentence:

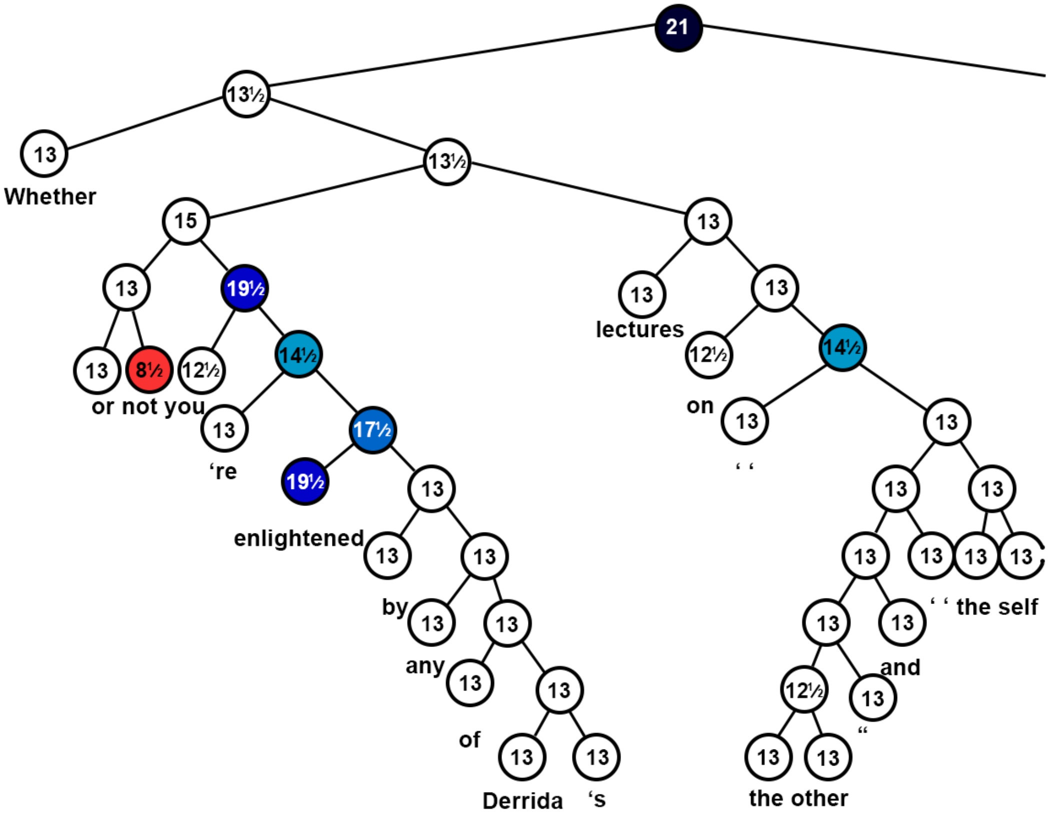


Figure 12.2: The "Whether" segment of a complex sentence

The sentence has been correctly split into two main parts. The second segment is also correct:

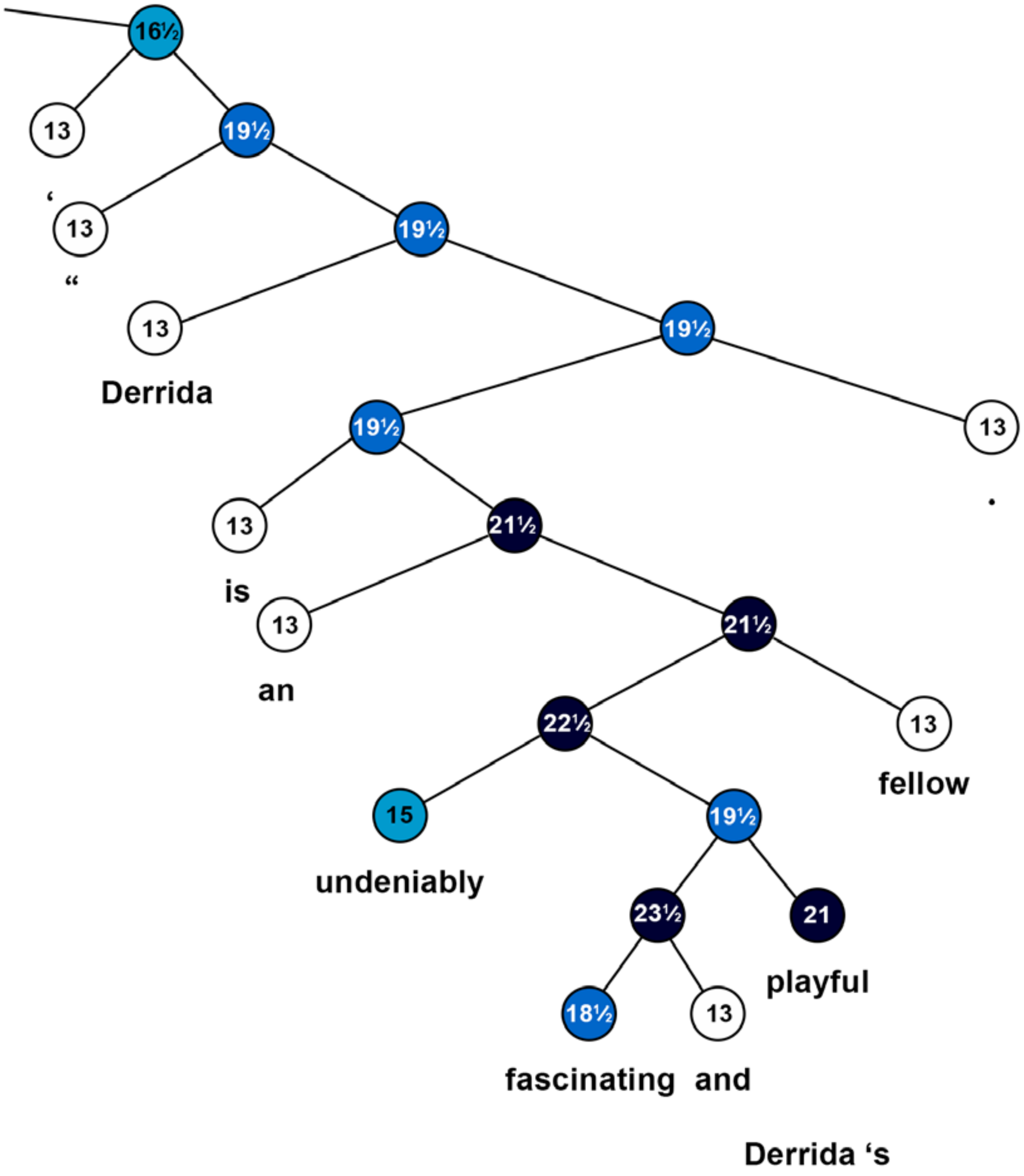


Figure 12.3: The main segment of a complex sentence

We can draw several conclusions from the method *Socher* et al. (2013) designed:

- Sentiment analysis cannot be reduced to counting positive and negative words in a sentence
- A transformer model or any NLP model must be able to learn the principle of compositionality to understand how the constituents of a complex sentence fit together with logical form rules
- A transformer model must be able to build a vector space to interpret the subtilities of a complex sentence

We will now put this theory into practice with a RoBERTa-large model.

Sentiment analysis with RoBERTa-large

In this section, we will use the AllenNLP resources to run a RoBERTa-large transformer. *Liu* et al. (2019) analyzed the existing BERT models and found that they were not trained as well as expected. Considering the speed at which the models were produced, this was not surprising. They worked on improving the pretraining of BERT models to produce a **Robustly Optimized BERT Pretraining Approach** (**RoBERTa**).

Let's first run a RoBERTa-large model in `SentimentAnalysis.ipynb`.

Run the first cell to install `allennlp-models`:

```
!pip install allennlp==1.0.0 allennlp-models==1.0.0
```

Now let's try to run our Jacques Derrida sample:

```
!echo '{"sentence": "Whether or not you're enlightened by any of Derrida's
lectures on the other and the self, Derrida is an undeniably fascinating
and playful fellow."}' | \
allennlp predict https://storage.googleapis.com/allennlp-public-models/
sst-roberta-large-2020.06.08.tar.gz -
```

The output first displays the architecture of the RoBERTa-large model, which has 24 layers and 16 attention heads:

```
"architectures": [
  "RobertaForMaskedLM"
  ],
  "attention_probs_dropout_prob": 0.1,
  "bos_token_id": 0,
  "eos_token_id": 2,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 1024,
  "initializer_range": 0.02,
  "intermediate_size": 4096,
  "layer_norm_eps": 1e-05,
  "max_position_embeddings": 514,
  "model_type": "roberta",
  "num_attention_heads": 16,
  "num_hidden_layers": 24,
  "pad_token_id": 1,
  "type_vocab_size": 1,
  "vocab_size": 50265
}
```

If necessary, you can take a few minutes to go through the description of a BERT architecture in the *BERT model configuration* section in *Chapter 3*, *Fine-Tuning BERT Models*, to take full advantage of this model.

Sentiment analysis produces values between `0` (negative) and `1` (positive).

The output then produces the result of the sentiment analysis task, displaying the output logits and the final positive result:

```
prediction:  {"logits": [3.646597385406494, -2.9539334774017334], "probs":
[0.9986421465873718, 0.001357800210826099]
```

Note: The algorithm is stochastic so the ouputs may vary from one run to another.

The output also contains the token IDs (which may vary from one run to another) and the final output label:

```
"token_ids": [0, 5994, 50, 45, 47, 769, 38853, 30, 143, 9, 6113, 10505,
281, 25798, 15, 5, 97, 8, 5, 1403, 2156, 211, 14385, 4347, 16, 41, 35559,
12509, 8, 23317, 2598, 479, 2], "label": "1",
```

The output also displays the tokens themselves:

```
"tokens": ["<s>", "\u0120Whether", "\u0120or", "\u0120not", "\u0120you",
"\u0120re", "\u0120enlightened", "\u0120by", "\u0120any", "\u0120of", "\
u0120Der", "rid", "as", "\u0120lectures", "\u0120on", "\u0120the", "\
u0120other", "\u0120and", "\u0120the", "\u0120self", "\u0120,", "\
u0120D", "err", "ida", "\u0120is", "\u0120an", "\u0120undeniably", "\
u0120fascinating", "\u0120and", "\u0120playful", "\u0120fellow", "\
u0120.", "</s>"]}
```

Take some time to enter some samples to explore the well-designed and pretrained RoBERTa model.

Now let's see how we can use sentiment analysis to predict customer behavior with other transformer models.

Predicting customer behavior with sentiment analysis

This section will run a sentiment analysis task on several Hugging Face transformer models to see which ones produce the best results and which ones we simply like the best.

We will begin this by using a Hugging Face DistilBERT model.

Sentiment analysis with DistilBERT

Let's run a sentiment analysis task with DistilBERT and see how we can use the result to predict customer behavior.

Open `SentimentAnalysis.ipynb` and the transformer installation and import cells:

```
!pip install -q transformers
from transformers import pipeline
```

We will now create a function named `classify`, which will run the model with the sequences we send to it:

```
def classify(sequence,M):
   #DistilBertForSequenceClassification(default model)
    nlp_cls = pipeline('sentiment-analysis')
    if M==1:
      print(nlp_cls.model.config)
    return nlp_cls(sequence)
```

Note that if you send `M=1` to the function, it will display the configuration of the DistilBERT 6-layer, 12-head model we are using:

```
DistilBertConfig {
  "activation": "gelu",
  "architectures": [
    "DistilBertForSequenceClassification"
  ],
  "attention_dropout": 0.1,
  "dim": 768,
  "dropout": 0.1,
  "finetuning_task": "sst-2",
  "hidden_dim": 3072,
  "id2label": {
    "0": "NEGATIVE",
    "1": "POSITIVE"
  },
  "initializer_range": 0.02,
  "label2id": {
    "NEGATIVE": 0,
    "POSITIVE": 1
  },
  "max_position_embeddings": 512,
  "model_type": "distilbert",
  "n_heads": 12,
  "n_layers": 6,
  "output_past": true,
  "pad_token_id": 0,
  "qa_dropout": 0.1,
  "seq_classif_dropout": 0.2,
  "sinusoidal_pos_embds": false,
  "tie_weights_": true,
  "vocab_size": 30522
}
```

The specific parameters of this DistilBERT model are the label definitions.

We now create a list of sequences (you can add more) that we can send to the `classify` function:

```
seq=3
if seq==1:
  sequence="The battery on my Model9X phone doesn't last more than 6 hours
and I'm unhappy about that."
if seq==2:
  sequence="The battery on my Model9X phone doesn't last more than 6 hours
and I'm unhappy about that. I was really mad! I bought a Moel10x and
things seem to be better. I'm super satisfied now."
if seq==3:
  sequence="The customer was very unhappy"
if seq==4:
  sequence="The customer was very satisfied"
print(sequence)
M=0 #display model cofiguration=1, default=0
CS=classify(sequence,M)
print(CS)
```

In this case, `seq=3` is activated to simulate a customer issue we need to take into account. The output is negative, which is the example we are looking for:

```
[{'label': 'NEGATIVE', 'score': 0.9997098445892334}]
```

We can draw several conclusions from this result to predict customer behavior by writing a function that would:

- Store the predictions in the customer management database.
- Count the number of times a customer complains about a service or product in a period (week, month, year). A customer that complains often might switch to a competitor to get a better product or service.
- Detect the products and services that keep occurring in negative feedback messages. The product or service might be faulty and require quality control and improvements.

You can take a few minutes to run other sequences or create some sequences to explore the DistilBERT model.

We will now explore other Hugging Face transformers.

Sentiment analysis with Hugging Face's models' list

This section will explore Hugging Face's transformer models list and enter some samples to evaluate their results. The idea is to test several models, not only one, and see which model fits your need the best for a given project.

We will be running Hugging Face models: `https://huggingface.co/models`.

For each model we use, you can find the description of the model in the documentation provided by Hugging Face: `https://huggingface.co/transformers/`.

We will test several models. If you implement them, you might find that they require fine-tuning or even pretraining for the NLP tasks you wish to perform. In that case, for Hugging Face transformers, you can do the following:

- For fine-tuning, you can refer to *Chapter 3, Fine-Tuning BERT Models*
- For pretraining, you can refer to *Chapter 4, Pretraining a RoBERTa Model from Scratch*

Let's first go through the list of Hugging Face models: `https://huggingface.co/models`.

Then, select **Text Classification** in the **Tasks** pane:

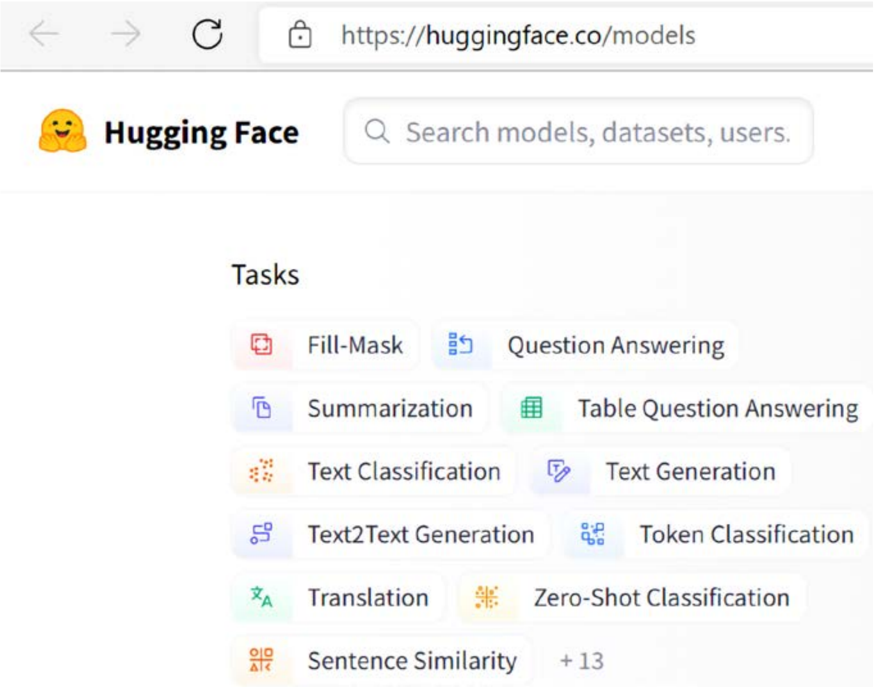
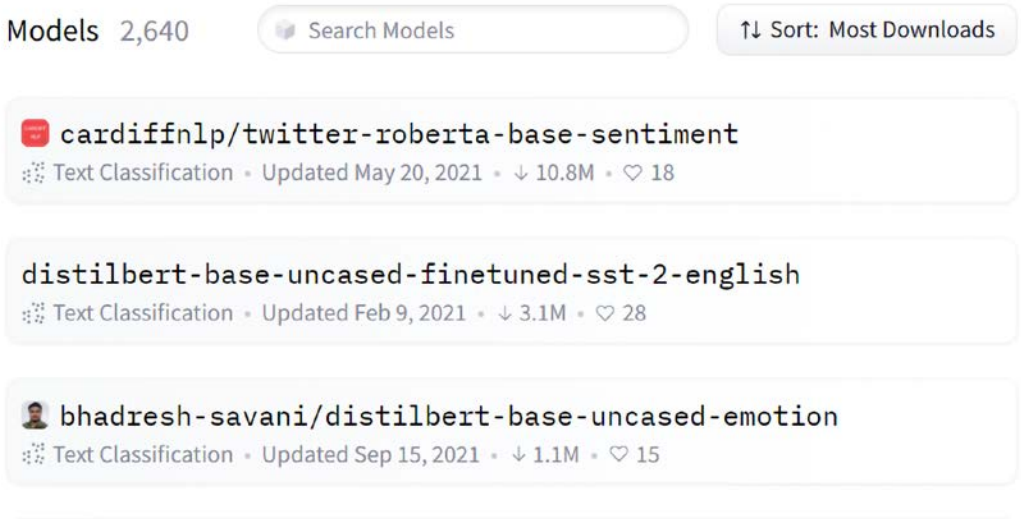


Figure 12.4: Selecting text classification models

A list of transformer models trained for text classification will appear:



Models 2,640

Search Models

⇅ Sort: Most Downloads

cardiffnlp/twitter-roberta-base-sentiment
Text Classification · Updated May 20, 2021 · ↓ 10.8M · ♡ 18

distilbert-base-uncased-finetuned-sst-2-english
Text Classification · Updated Feb 9, 2021 · ↓ 3.1M · ♡ 28

bhadresh-savani/distilbert-base-uncased-emotion
Text Classification · Updated Sep 15, 2021 · ↓ 1.1M · ♡ 15

Figure 12.5: Hugging Face pretrained text-classification models

The default sort mode is **Sort: Most downloads**.

We will now search for some exciting transformer models we can test online.

We will begin with DistilBERT.

DistilBERT for SST

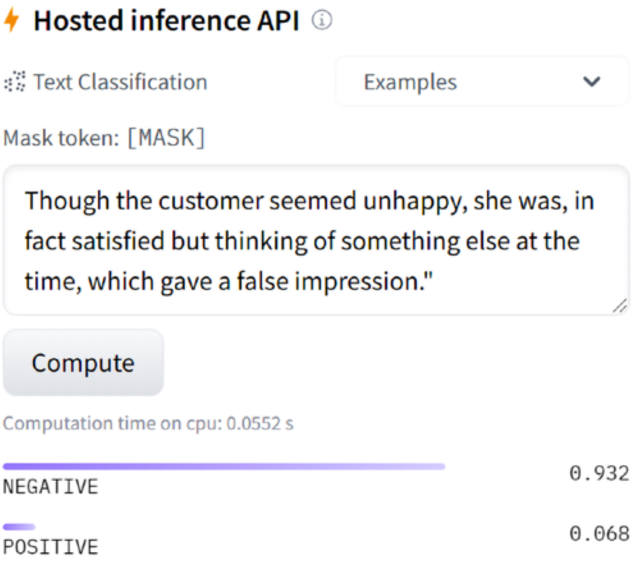
The `distilbert-base-uncased-finetuned-sst-2-english` model was fine-tuned on the SST.

Let's try an example that requires a good understanding of the principles of compositionality:

```
"Though the customer seemed unhappy, she was, in fact satisfied but thinking of
something else at the time, which gave a false impression."
```

This sentence is tough for a transformer to analyze and requires logical rule training.

The output is a false negative:



Mask token: [MASK]

Though the customer seemed unhappy, she was, in fact satisfied but thinking of something else at the time, which gave a false impression."

Computation time on cpu: 0.0552 s

```
NEGATIVE    0.932
POSITIVE    0.068
```

Figure 12.6: The output of a complex sequence classification task

A false negative does not mean that the model is not working correctly. We could choose another model. However, it could mean that we must download and train it longer and better!

At the time of writing this book, BERT-like models have good rankings on both the GLUE and SuperGLUE leaderboards. The rankings will continuously change but not the fundamental concepts of transformers.

We will now try a difficult but less complicated example.

This example is a crucial lesson for real-life projects. When we try to estimate how many times a customer complained, we will get both false negatives and false positives. *Therefore, regular human intervention will still be mandatory for several more years.*

Let's give a MiniLM model a try.

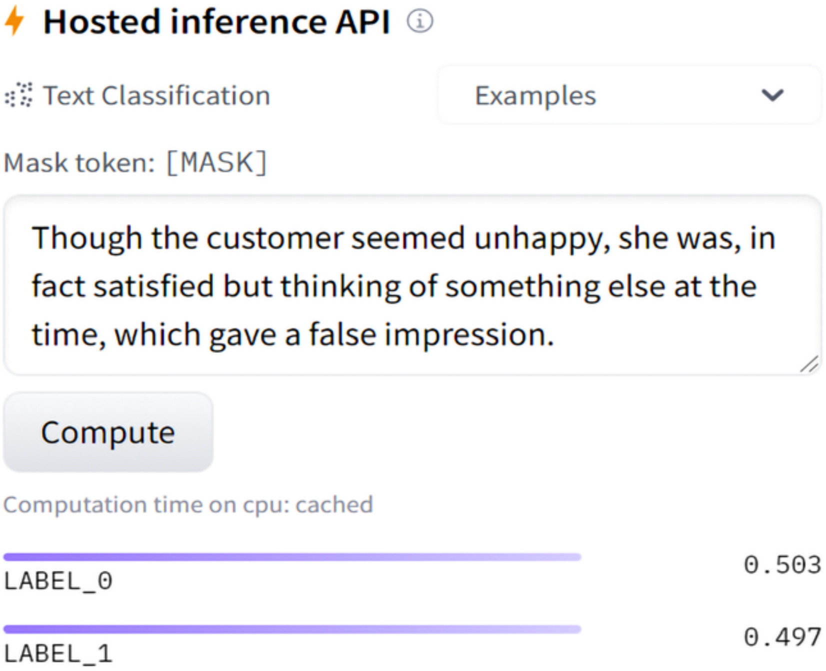
MiniLM-L12-H384-uncased

Microsoft/MiniLM-L12-H384-uncased optimizes the size of the last self-attention layer of the teacher, among other tweakings of a BERT model, to obtain better performances. It has 12 layers, 12 heads, and 33 million parameters, and is 2.7 times faster than BERT-base.

Let's test it for its capacity to understand the principles of compositionality:

```
Though the customer seemed unhappy, she was, in fact satisfied but thinking of
something else at the time, which gave a false impression.
```

The output is interesting because it produces a careful split (undecided) score:



Though the customer seemed unhappy, she was, in fact satisfied but thinking of something else at the time, which gave a false impression.

```
LABEL_0    0.503
LABEL_1    0.497
```

Figure 12.7: Complex sentence sentiment analysis

We can see that this output is not conclusive since it is around `0.5`. It should be positive.

Let's try a model involving entailment.

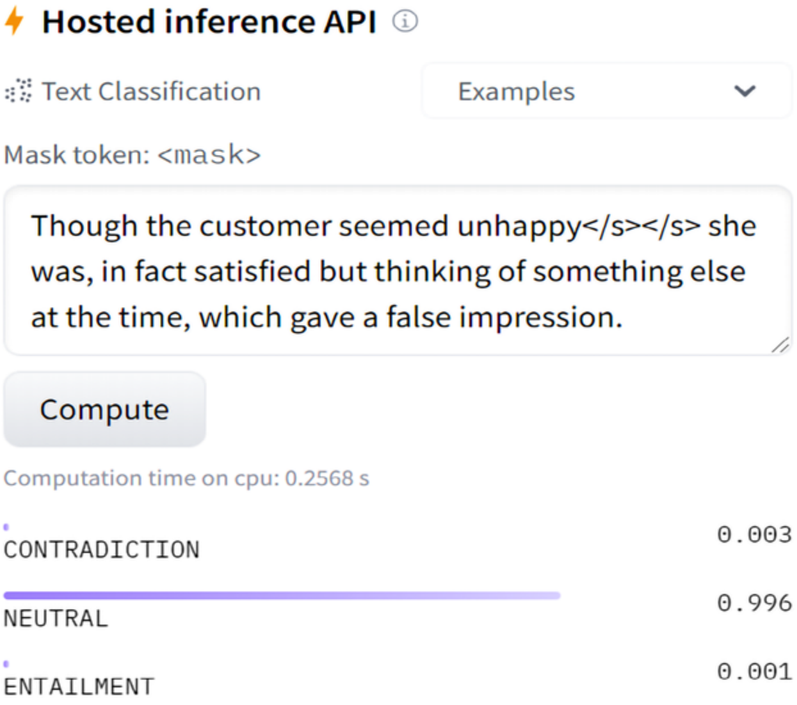
RoBERTa-large-mnli

A **Multi-Genre Natural Language Inference** (**MultiNLI**) task, `https://cims.nyu.edu/~sbowman/multinli/`, can help solve the interpretation of a complex sentence when we are trying to determine what a customer means. Inference tasks must determine whether a sequence entails the following one or not.

We need to format our input and split the sequence with sequence splitting tokens:

```
Though the customer seemed unhappy</s></s> she was, in fact satisfied but thinking of something else at the time, which gave a false impression
```

The result is interesting, although it remains neutral:



Though the customer seemed unhappy</s></s> she was, in fact satisfied but thinking of something else at the time, which gave a false impression.

Computation time on cpu: 0.2568 s

```
CONTRADICTION    0.003
NEUTRAL          0.996
ENTAILMENT       0.001
```

Figure 12.8: The neutral result obtained for a slightly positive sentence

However, there is no mistake in this result. The second sequence is not inferred from the first sequence. The result is carefully correct.

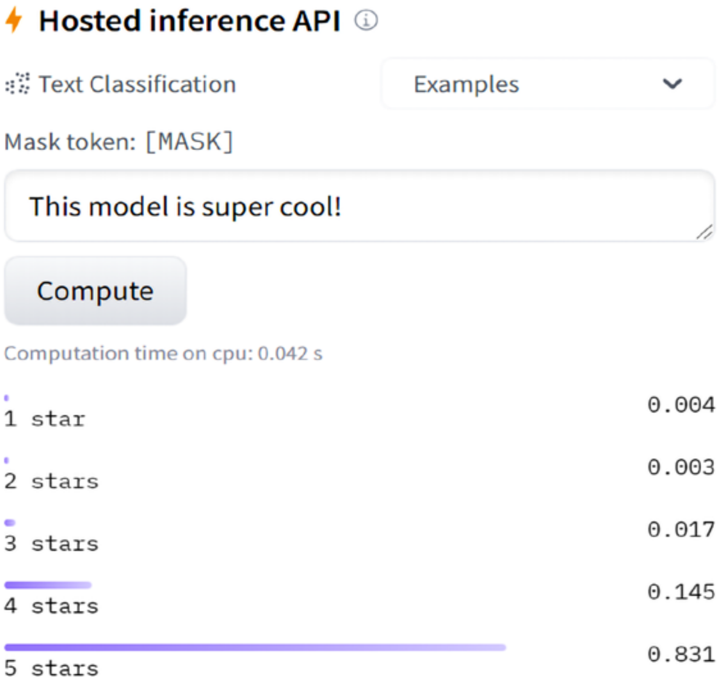
Let's finish our experiments on a "positive sentiment" multilingual BERT-base model.

BERT-base multilingual model

Let's run our final experiment on a super cool BERT-base model: `nlptown/bert-base-multilingual-uncased-sentiment`.

It is very well-designed.

Let's run it with a friendly and positive sentence in English:



Hosted inference API

Text Classification

Examples

Mask token: [MASK]

This model is super cool!

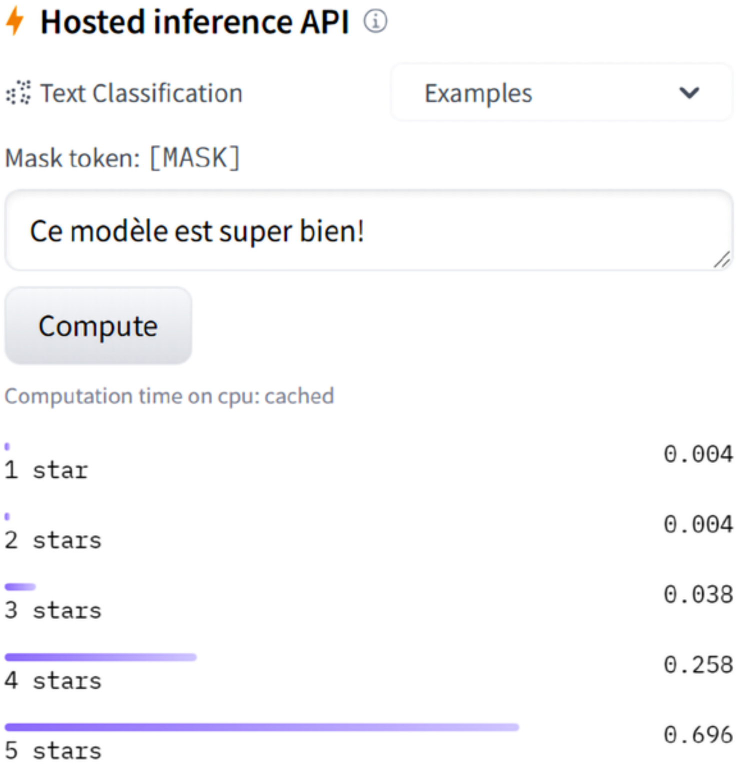
Compute

Computation time on cpu: 0.042 s

```
1 star      0.004
2 stars     0.003
3 stars     0.017
4 stars     0.145
5 stars     0.831
```

Figure 12.9: Sentiment analysis in English

Let's try it in French with "Ce modèle est super bien!" ("this model is super good," meaning "cool"):



Mask token: [MASK]

Ce modèle est super bien!

Computation time on cpu: cached

```
1 star     0.004
2 stars    0.004
3 stars    0.038
4 stars    0.258
5 stars    0.696
```

Figure 12.10: Sentiment analysis in French

The path of this model for Hugging Face is `nlptown/bert-base-multilingual-uncased-sentiment`. You can find it in the search form on the Hugging Face website. Its present link is `https://huggingface.co/nlptown/bert-base-multilingual-uncased-sentiment?text=Ce+mod%C3%A8le+est+super+bien%21`.

You can implement it on your website with the following initialization code:

```python
from transformers import AutoTokenizer, AutoModelForSequenceClassification
tokenizer = AutoTokenizer.from_pretrained("nlptown/bert-base-multilingual-uncased-sentiment")
model = AutoModelForSequenceClassification.from_pretrained("nlptown/bert-base-multilingual-uncased-sentiment")
```

It will take some time and patience, but the result could be super cool!

You could implement this transformer on your website to average out the global satisfaction of your customers! You could also use it as continuous feedback to improve your customer service and anticipate customer reactions.

Before we leave, we will see how GPT-3 performs sentiment analysis.

Sentiment analysis with GPT-3

You will need an OpenAI account to run the examples in this section. The educational interface requires no API, no development, or training. You can simply enter some tweets, for example, and ask for sentiment analysis:

Tweet: `I didn't find the movie exciting, but somehow I really enjoyed watching it!`

Sentiment: `Positive`

Tweet: `I never ate spicy food like this before but find it super good!`

Sentiment: `Positive`

The outputs are satisfactory.

We will now submit a difficult sequence to the GPT-3 engine:

Tweet: `It's difficult to find what we really enjoy in life because of all of the parameters we have to take into account.`

Sentiment: `Positive`

The output is false! The sentiment is not positive at all. The sentence shows the difficulty of life. However, the word `enjoy` introduced bias for GPT-3.

If we take enjoy out of the sequence and replace it with the verb `are`, the output is negative:

Tweet: `It's difficult to find what we really are in life because of all of the parameters we have to take into account.`

Sentiment: `Negative`

The output is false also! It's not because life is difficult to figure out that we can conclude that the sentence is negative. The correct output should have been neutral. Then we could ask GPT-3 to perform another task to explain why it is difficult in a pipeline, for example.

Running NLP tasks as a user with nothing to do shows where Industry 4.0 (I4.0) is going: less human intervention and more automatic functionality. *However, we know that some situations require our new skills, such as designing preprocessing functions when the transformer doesn't produce the expected result. Humans are still useful!*

An example of tweet classification with ready-to-use code is described in the *Running OpenAI GPT-3 Tasks* section of *Chapter 7*, *The Rise of Suprahuman Transformers with GPT-3 Engines*. You can implement the examples of this section in that code if you wish.

Let's now see how we can still prove ourselves valuable assets.

Some Pragmatic I4.0 thinking before we leave

The sentiment analysis with Hugging Face transformers contained a sentence that came out as "neutral."

But is that true?

Labeling this sentence "neutral" bothered me. I was curious to see if OpenAI GPT-3 could do better. After all, GPT-3 is a foundation model that can theoretically do many things it wasn't trained for.

I examined the sentence again:

```
Though the customer seemed unhappy, she was, in fact, satisfied but thinking of
something else at the time, which gave a false impression.
```

When I read the sentence closely, I could see that the customer is `she`. When I looked deeper, I understood that `she` is `in fact satisfied`. I decided not to try models blindly until I reached one that works. Trying one model after the other is not productive.

I needed to get to the root of the problem using logic and experimentation. I didn't want to rely on an algorithm that would find the reason automatically. Sometimes we need to use *our* neurons!

Could the problem be that it is difficult to identify `she` as the `customer` for a machine? As we did in *Chapter 10*, *Semantic Role Labeling with BERT-Based Transformers*, let's ask SRL BERT.

Investigating with SRL

Chapter 10 ended with my recommendation to use SRL with other tools, which we are doing now.

I first ran `She was satisfied` using the **Semantic Role Labeling** interface on `https://demo.allennlp.org/`.

The result was correct:

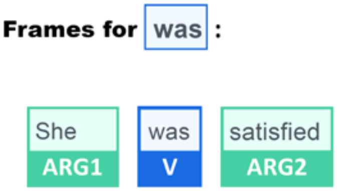


Figure 12.11: SRL of a simple sentence

The analysis is clear in the frame of this predicate: `was` is the verb, `She` is **ARG1**, and `satisfied` is **ARG2**.

We should find the same analysis in a complex sentence, and we do:

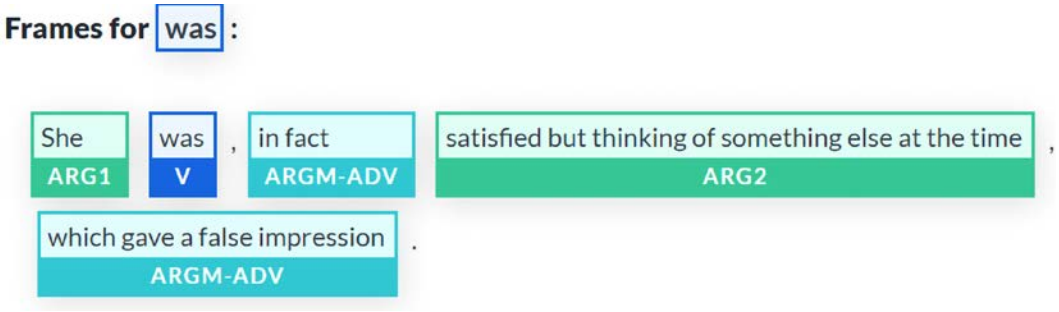

Frames for was :
She
ARG1
was
V
,
in fact
ARGM-ADV
satisfied but thinking of something else at the time
ARG2
,
which gave a false impression
ARGM-ADV
.

Figure 12.12: The verb "satisfied" is merged with other words, causing confusion

`Satisfied` is still **ARG2**, so the problem might not be there.

Now, the focus is on **ARGM-ADV**, which modifies `was` as well. The word `false` is quite misleading because **ARGM-ADV** is relative to **ARG2**, which contains `thinking`.

The `thinking` predicate gave a `false impression`, but thinking is not identified as a predicate in this complex sentence. Could it be that `she was` is an unidentified ellipsis, as we saw in the *Questioning the scope of SRL* section of *Chapter 10*?

We can quickly verify that by entering the full sentence without an ellipsis:

```
Though the customer seemed unhappy, she was, in fact, satisfied but she was thinking of something else at the time, which gave a false impression.
```

The problem with SRL was the ellipsis again, as we saw in *Chapter 10*. We now have five correct predicates with five accurate frames.

Frame 1 shows that `unhappy` is correctly related to `seemed`:

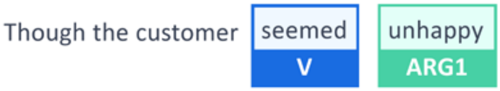


Figure 12.13: "Unhappy" is correctly related to "seemed"

Frame 2 shows that `satisfied` is now separated from the sentence and individually identified as an argument of `was` in a complex sentence:

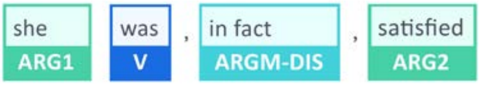


Figure 12.14: "satisfied" is now a separate word in ARG2

Now, let's go straight to the predicate containing `thinking`, which is the verb we wanted BERT SRL to analyze correctly. Now that we suppressed the ellipsis and repeated "she was" in the sentence, the output is correct:

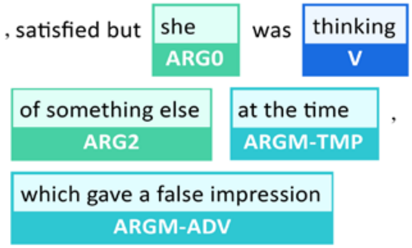


Figure 12.15: Accurate output without an ellipsis

Now, we can leave our SRL investigation with two clues:

- The word `false` is a confusing argument for an algorithm to relate to other words in a complex sentence
- The ellipsis of the repetition of `she was`

Before we go to GPT-3, let's go back to Hugging Face with our clues.

Investigating with Hugging Face

Let's go back to the DistilBERT base uncased fine-tuned SST-2 model we used in this chapter's *DistilBERT for SST* section.

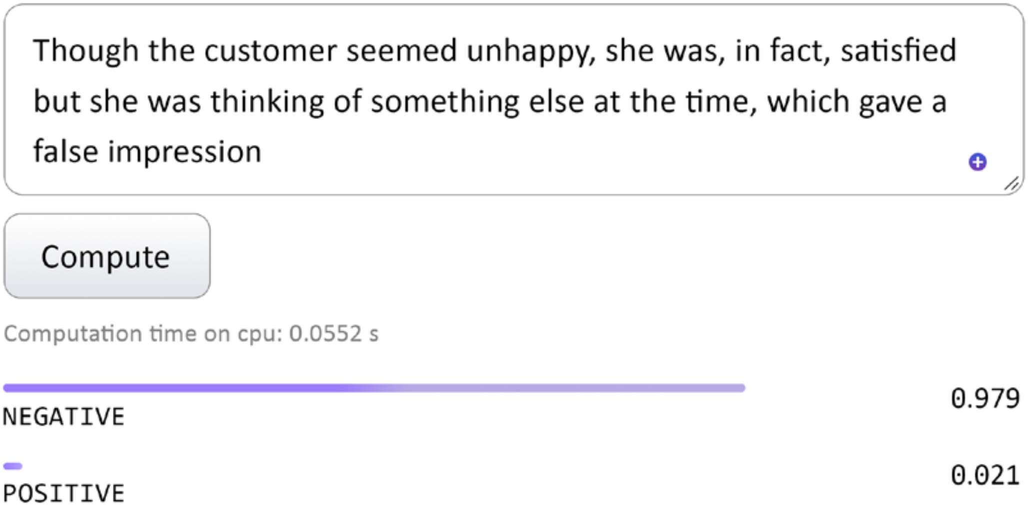
We will investigate our two clues:

- The ellipsis of `she was`

  We will first submit a full sentence with no ellipsis:

  ```
  Though the customer seemed unhappy, she was, in fact, satisfied but she was thinking of something else at the time, which gave a false impression
  ```

  The output remains negative:



Though the customer seemed unhappy, she was, in fact, satisfied but she was thinking of something else at the time, which gave a false impression

Computation time on cpu: 0.0552 s

NEGATIVE 0.979

POSITIVE 0.021

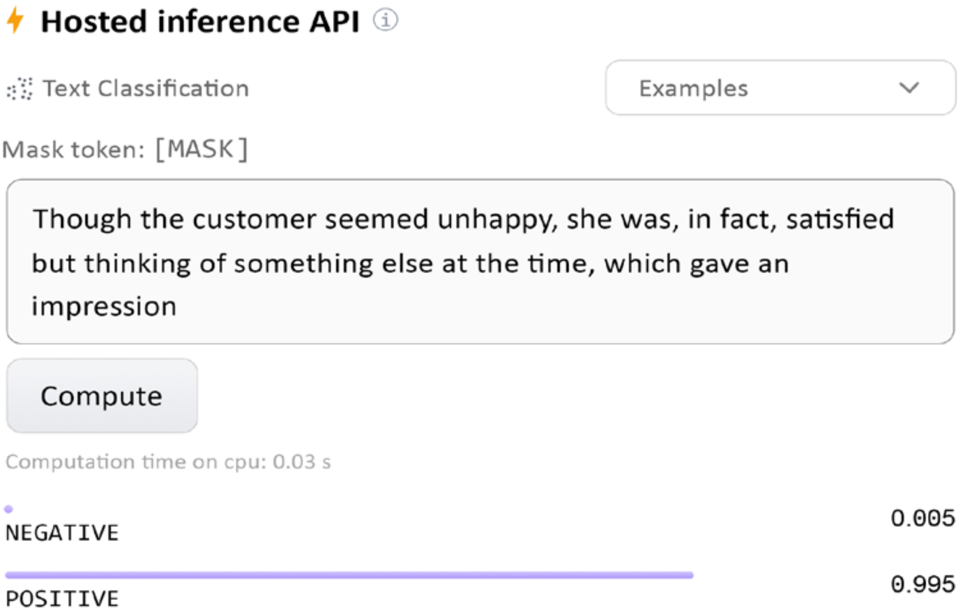
Figure 12.16: A false negative

- The presence of `false` in an otherwise positive sentence.

  We will now take `false` out of the sentence but leave the ellipsis:

  `Though the customer seemed unhappy, she was, in fact, satisfied but thinking of something else at the time, which gave an impression`

  Bingo! The output is positive:



Mask token: [MASK]

Though the customer seemed unhappy, she was, in fact, satisfied but thinking of something else at the time, which gave an impression

Computation time on cpu: 0.03 s

NEGATIVE 0.005

POSITIVE 0.995

Figure 12.17: A true positive

We know that the word `false` creates confusion for SRL if there is an ellipsis of `was thinking`. We also know that `false` creates confusion for the sentiment analysis Hugging Face transformer model we used.

Can GPT-3 do better? Let's see.

Investigating with the GPT-3 playground

Let's use the OpenAI's example of an **Advanced tweet classifier** and modify it to satisfy our investigation in three steps:

- *Step 1*: Showing GPT-3 what we expect:

  Sentence: `"The customer was satisfied"`

  Sentiment: `Positive`

  Sentence: `"The customer was not satisfied"`

  Sentiment: `Negative`

  Sentence: `"The service was 👍"`

  Sentiment: `Positive`

  Sentence: `"This is the link to the review"`

  Sentiment: `Neutral`

- *Step 2*: Showing it a few examples of the output format we expect:

  ```
  1. "I loved the new Batman movie!"
  2. "I hate it when my phone battery dies"
  3. "My day has been 👍"
  4. "This is the link to the article"
  5. "This new music video blew my mind"
  ```

  Sentence sentiment ratings:

  ```
  1: Positive
  2: Negative
  3: Positive
  4: Neutral
  5: Positive
  ```

- *Step 3*: Entering our sentence among others (number 3):

  ```
  1. "I can't stand this product"
  2. "The service was bad! 😠"
  3. "Though the customer seemed unhappy she was in fact satisfied but thinking of something else at the time, which gave a false impression"
  4. "The support team was ❤️❤️"
  5. "Here is the link to the product."
  ```

  Sentence sentiment ratings:

  ```
  1: Negative
  2: Positive
  3: Positive
  4: Positive
  5: Neutral
  ```

The output seems satisfactory since our sentence is positive (number 3). Is this result reliable? We could run the example here several times. But let's go down to code level to find out.

GPT-3 code

We just click on **View code** in the playground, copy it, and paste it into our `SentimentAnalysis.ipynb` chapter notebook. We add a line to only print what we want to see:

```
response = openai.Completion.create(
  engine="davinci",
  prompt="This is a Sentence sentiment classifier\nSentence: \"The
customer was satisfied\"\nSentiment: Positive\n###\nSentence: \"The
customer was not satisfied\"\nSentiment: Negative\n###\nSentence: \"The
service was 👍\"\nSentiment: Positive\n###\nSentence: \"This is the link
to the review\"\nSentiment: Neutral\n###\nSentence text\n\n\n1. \"I loved
the new Batman movie!\"\n2. \"I hate it when my phone battery dies\"\
n3. \"My day has been 👍\"\n4. \"This is the link to the article\"\n5.
\"This new music video blew my mind\"\n\n\nSentence sentiment ratings:\
n1: Positive\n2: Negative\n3: Positive\n4: Neutral\n5: Positive\n\n\n###\
nSentence text\n\n\n1. \"I can't stand this product\"\n2. \"The service
was bad! 😠\"\n3. \"Though the customer seemed unhappy she was in fact
satisfied but thinking of something else at the time, which gave a false
impression\"\n4. \"The support team was ❤️❤️\"\n5. \"Here is the link to
the product.\"\n\n\nSentence sentiment ratings:\n",
```

```
    temperature=0.3,
    max_tokens=60,
    top_p=1,
    frequency_penalty=0,
    presence_penalty=0,
    stop=["###"]
)
r = (response["choices"][0])
print(r["text"])
```

The output is not stable, as we can see in the following responses:

- **Run 1**: Our sentence (number 3) is neutral:

  ```
  1: Negative

  2: Negative

  3: Neutral

  4: Positive

  5: Positive
  ```

- **Run 2**: Our sentence (number 3) is positive:

  ```
  1: Negative

  2: Negative

  3: Positive

  4: Positive

  5: Neutral
  ```

- **Run 3**: Our sentence (number 3) is positive
- **Run 4**: Our sentence (number 3) is negative

This leads us to the conclusions of our investigation:

- SRL shows that if a sentence is simple and complete (no ellipsis, no missing words), we will get a reliable sentiment analysis output.
- SRL shows that if the sentence is moderately difficult, the output might, or might not, be reliable.

- SRL shows that if the sentence is complex (ellipsis, several propositions, many ambiguous phrases to solve, and so on), the result is not stable, and therefore not reliable.

The conclusions of the job positions of developers in the present and future are:

- Less AI development will be required with Cloud AI and ready-to-use modules.
- More design skills will be required.
- Classical development of pipelines to feed AI algorithms, control them, and analyze their outputs will require thinking and targeted development.

This chapter shows a huge future for developers as thinkers, designers, and pipeline development!

It's now time to sum up our journey and explore new transformer horizons.

Summary

In this chapter, we went through some advanced theories. The principle of compositionality is not an intuitive concept. The principle of compositionality means that the transformer model must understand every part of the sentence to understand the whole sentence. This involves logical form rules that will provide links between the sentence segments.

The theoretical difficulty of sentiment analysis requires a large amount of transformer model training, powerful machines, and human resources. Although many transformer models are trained for many tasks, they often require more training for specific tasks.

We tested RoBERTa-large, DistilBERT, MiniLM-L12-H384-uncased, and the excellent BERT-base multilingual model. We found that some provided interesting answers but required more training to solve the SST sample we ran on several models.

Sentiment analysis requires a deep understanding of a sentence and extraordinarily complex sequences. So, it made sense to try RoBERTa-large-mnli to see what an interference task would produce. The lesson here is not to be conventional with something as unconventional as transformer models! Try everything. Try different models on various tasks. Transformers' flexibility allows us to try many different tasks on the same model or the same task on many different models.

We gathered some ideas along the way to improve customer relations. If we detect that a customer is unsatisfied too often, that customer might just seek out our competition. If several customers complain about a product or service, we must anticipate future problems and improve our services. We can also display our quality of service with online real-time representations of a transformer's feedback.

Finally, we ran sentiment analysis with GPT-3 directly online with nothing to do but use the interface! It's surprisingly effective, but we see that humans are still required to solve the more difficult sequences. We saw how SRL could help identify the issues in complex sequences.

We can conclude that developers have a huge future as thinkers, designers, and pipeline development.

In the next chapter, *Analyzing Fake News with Transformers*, we'll use sentiment analysis to analyze emotional reactions to fake news.

Questions

1. It is not necessary to pretrain transformers for sentiment analysis. (True/False)
2. A sentence is always positive or negative. It cannot be neutral. (True/False)
3. The principle of compositionality signifies that a transformer must grasp every part of a sentence to understand it. (True/False)
4. RoBERTa-large was designed to improve the pretraining process of transformer models. (True/False)
5. A transformer can provide feedback that informs us whether a customer is satisfied or not. (True/False)
6. If the sentiment analysis of a product or service is consistently negative, it helps us make the proper decisions to improve our offer. (True/False)
7. If a model fails to provide a good result on a task, it requires more training before changing models. (True/False)

References

- *Richard Socher*, *Alex Perelygin*, *Jean Wu*, *Jason Chuang*, *Christopher Manning*, *Andrew Ng*, and *Christopher Potts*, *Recursive Deep Models for Semantic Compositionality over a Sentiment Treebank*: `https://nlp.stanford.edu/~socherr/EMNLP2013_RNTN.pdf`
- Hugging Face pipelines, models, and documentation:
  - `https://huggingface.co/transformers/main_classes/pipelines.html`
  - `https://huggingface.co/models`
  - `https://huggingface.co/transformers/`

- *Yinhan Liu*, *Danqi Chen*, *Omer Levy*, *Mike Lewis*, *Luke Zettlemoyer*, and *Veselin Stoyanov*, 2019, *RoBERTa: A Robustly Optimized BERT Pretraining Approach*: `https://arxiv.org/pdf/1907.11692.pdf`
- *The Allen Institute for AI*: `https://allennlp.org/`
- The Allen Institute for reading comprehension resources: `https://demo.allennlp.org/sentiment-analysis`
- RoBERTa-large contribution, *Zhaofeng Wu*: `https://zhaofengwu.github.io/`
- *The Stanford Sentiment Treebank*: `https://nlp.stanford.edu/sentiment/treebank.html`

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the authors:

`https://www.packt.link/Transformers`

13

Analyzing Fake News with Transformers

We were all born thinking that the Earth was flat. As babies, we crawled on flat surfaces. As kindergarten children, we played on flat playgrounds. In elementary school, we sat in flat classrooms. Then, our parents and teachers told us that the Earth was round and that the people on the other side of it were upside down. It took us quite a while to understand why they did not fall off the Earth. Even today, when we see a beautiful sunset, we still see the "sun set," and not the Earth rotating away from the sun!

It takes time and effort to figure out what is fake news and what isn't. Like children, we have to work our way through something we perceive as fake news.

This chapter will tackle some of the topics that create tensions. We will check the facts on topics such as climate change, gun control, and Donald Trump's Tweets. We will analyze Tweets, Facebook posts, and other sources of information.

Our goal is certainly not to judge anybody or anything. Fake news involves both opinions and facts. News often depends on the perception of facts by local culture. We will provide ideas and tools to help others gather more information on a topic and find their way in the jungle of information we receive daily.

We will be focusing on ethical methods, not the performance of transformers. We will not be using a GPT-3 engine for this reason. We are not replacing human judgment. Instead, we are providing tools for humans to make their own judgments manually. GPT-3 engines have attained human-level performance for many tasks. However, we should leave moral and ethical decision making to humans.

Therefore, first, we will begin by defining the path that leads us to react emotionally and rationally to fake news.

We will then define some methods to identify fake news with transformers and heuristics.

We will be using the resources we built in the previous chapters to understand and explain fake news. We will not judge. We will provide transformer models that explain the news. Some might prefer to create a universal absolute transformer model to detect and assert that a message is fake news.

I choose to educate users with transformers, not to lecture them. This approach is my opinion, not a fact!

This chapter covers the following topics:

- Cognitive dissonance
- Emotional reactions to fake news
- A behavioral representation of fake news
- A rational approach to fake news
- A fake news resolution roadmap
- Applying sentiment analysis transformer tasks to social media
- Analyzing gun control perceptions with NER and SRL
- Using information extracted by transformers to find reliable websites
- Using transformers to produce results for educational purposes
- How to read former President Trump's Tweets with an objective but critical eye

Our first step will be to explore the emotional and rational reactions to fake news.

Emotional reactions to fake news

Human behavior has a tremendous influence on our social, cultural, and economic decisions. Our emotions influence our economy as much as, if not more than, rational thinking. Behavioral economics drives our decision-making process. We buy consumer goods that we physically need and satisfy our emotional desires. We might even buy a smartphone in the heat of the moment, although it exceeds our budget.

Our emotional and rational reactions to fake news depend on whether we think slowly or react quickly to incoming information. *Daniel Kahneman* described this process in his research and book, *Thinking, Fast and Slow (2013)*.

He and *Vernon L. Smith* were awarded the *Nobel Memorial Prize* in *Economic Sciences* for behavioral economics research. Behavior drives decisions we previously thought were rational. Unfortunately, many of our decisions are based on emotions, not reason.

Let's translate these concepts into a behavioral flowchart applied to fake news.

Cognitive dissonance triggers emotional reactions

Cognitive dissonance drives fake news up to the top ranks of Twitter, Facebook, and other social media platforms. If everybody agrees with the content of a Tweet, nothing will happen. If somebody writes a Tweet saying, "Climate change is important," nobody will react.

We enter a state of cognitive dissonance when tensions build up between contradictory thoughts in our minds. As a result, we become nervous, agitated, and it wears us down like a short-circuit in a toaster.

We have many examples to think about. Is wearing a mask with COVID-19 necessary when we are outdoors? Are lockdowns a good or bad thing? Are the coronavirus vaccines effective? Or are coronavirus vaccines dangerous? Cognitive dissonance is like a musician that keeps making mistakes while playing a simple song. It drives us crazy!

The fake news syndrome increases cognitive dissonance exponentially! One expert will assert that vaccines are safe, and another that we need to be careful. One expert says that wearing a mask outside is useless, and another one asserts on a news channel that we must wear one! Each side accuses the other of fake news!

It appears for one side a significant portion of fake news is the other side's truth.

We are in 2022, and the US Republicans and Democrats are still unable to agree on national election rules in the wake of the 2020 presidential elections, or the organization of the upcoming elections.

We could go on and find scores of other topics by just opening one newspaper and then read another view in another opposing one! Nevertheless, some common-sense premises to this chapter can be drawn from these examples:

- Finding a transformer model that automatically detects fake news makes no sense. In the world of social media and multicultural expression, each group has a sense of knowing the truth, and the other group is expressing fake news.
- Trying to express our view as being the truth from one culture to another makes no sense. In a global world, cultures vary in each country, each continent, and everywhere in social media.

- Fake news as an absolute is a myth.
- We need to find a better definition of fake news.

My opinion (not a fact, of course!) is that fake news is a state of cognitive dissonance that can only be resolved by cognitive reasoning. Thus, resolving the problem of fake news is exactly like trying to resolve a conflict between two parties or within our own minds.

In this chapter and life, my recommendation is to analyze each conflictual tension by deconstructing the conflict and ideas with transformer models. We are not "combating fake news," "finding inner peace," or pretending to use transformers to find "the absolute truth to oppose fake news."

We use transformers to obtain a deeper understanding of a sequence of words (a message) to form a more profound and broader opinion on a topic.

Once that is done, let the lucky user of transformer models obtain a better vision and opinion.

To do this, I designed the chapter as a classroom exercise we can use for ourselves and others. Transformers are a great way to deepen our understanding of language sequences, form broader opinions, and develop our cognitive abilities.

Let's start by seeing what happens when somebody posts a conflictual Tweet.

Analyzing a conflictual Tweet

The following Tweet is a message posted on Twitter (I paraphrased it). The Tweets shown in this chapter are in raw dataset format, not the Twitter interface display. You can be sure that many people would disagree with the content if a leading political figure or famous actor tweeted it:

```
Climate change is bogus. It's a plot by the liberals to take the economy down.
```

It would trigger emotional reactions. Tweets would pile up on all sides. It would go viral and trend!

Let's run the Tweet on transformer tools to understand how this Tweet could create a cognitive dissonance storm in somebody's mind.

Open `Fake_News.ipynb`, the notebook we will be using in this section.

We will begin with resources from the *Allen Institute for AI*. We will run the RoBERTa transformer model we used for sentiment analysis in *Chapter 12, Detecting Customer Emotions to Make Predictions*.

We will first install `allennlp-models`:

```
!pip install allennlp==1.0.0 allennlp-models==1.0.0
```

AllenNLP is continuously updating the versions as they progress. Version 2.4.0 exists at the time of the writing but provides no additional value to the examples provided in this chapter at this date. *Stochastic algorithms or models that are updated can produce different outputs from one to another*.

We then run the next cell with Bash to analyze the output of the Tweet in detail (information on the model and the output):

```
!echo '{"sentence":"Climate change is bogus. It's a plot by the liberals
to take the economy down."}' | \
allennlp predict https://storage.googleapis.com/allennlp-public-models/
sst-roberta-large-2020.06.08.tar.gz -
```

The output shows that the Tweet is negative. The positive value is `0`, and the negative value is near 1:

```
"probs": [0.0008486526785418391, 0.999151349067688]
```

The output might vary from one run to the other since transformers are stochastic algorithms.

We will now go to `https://allennlp.org/` to get a visual representation of the analysis.

The output might change from one run to another. Transformer models are continuously trained and updated. Our goal in this chapter is to focus on the reasoning of transformer models.

We select **Sentiment Analysis** (`https://demo.allennlp.org/sentiment-analysis`) and choose the **RoBERTa large model** to run the analysis.

We obtain the same negative result. However, we can investigate further and see what words influenced RoBERTa's decision.

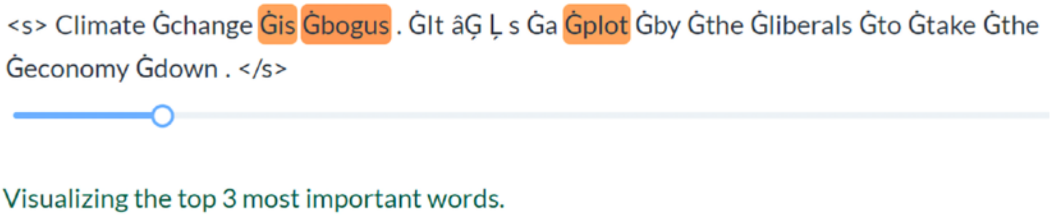
Go to **Model Interpretations**.

Interpreting the model will provide insights into how a result was obtained. We can choose one or look into three options:

- **Simple Gradient Visualization**: This approach provides two visualizations. The first one computes the gradient of the score of a class related to the input. The second one is a saliency (main features) map inferred from the class and input.
- **Integrated Gradient Visualization**: This model requires no changes to a neural network.
- The motivation is to design calls to the gradient used to generate an attribution of the predictions of a neural network to its inputs.

- **Smooth Gradient Visualization**: This approach computes the gradients using the output predictions and the input. The goal is to identify the features of the input. However, noise is added to improve the interpretation.

In this section, go to **Model Interpretations** and then click on **Simple Gradient Visualization** and on **Interpret Prediction** to obtain the following representation:



<s> Climate Ġchange Ġis Ġbogus . ĠIt âĢ Ļ s Ġa Ġplot Ġby Ġthe Ġliberals Ġto Ġtake Ġthe Ġeconomy Ġdown . </s>

Visualizing the top 3 most important words.

Figure 13.1: Visualizing the top 3 most important words

`is` + `bogus` + `plot` mostly influenced the negative prediction.

At this point, you may be wondering why we are looking at such a simple example to explain cognitive dissonance. The explanation comes from the following Tweet.

A staunch Republican wrote the first Tweet. Let's call the member `Jaybird65`. To his surprise, a fellow Republican tweeted the following Tweet:

`I am a Republican and think that climate change consciousness is a great thing!`

This Tweet came from a member we will call `Hunt78`. Let's run this sentence in `Fake_News.ipynb`:

```
!echo '{"sentence":"I am a Republican and think that climate change
consciousness is a great thing!"}' | \
allennlp predict https://storage.googleapis.com/allennlp-public-models/
sst-roberta-large-2020.06.08.tar.gz -
```

The output is positive, of course:

```
"probs": [0.9994876384735107, 0.0005123814917169511]
```

A cognitive dissonance storm is building up in `Jaybird65`'s mind. He likes `Hunt78` but disagrees. His mind storm is intensifying! If you read the subsequent Tweets that ensue between `Jaybird65` and `Hunt78`, you would discover some surprising facts that hurt `Jaybird65`'s feelings:

`Jaybird65` and `Hunt78` obviously know each other.

- If you go to their respective Twitter accounts, you will see that they are both hunters.
- You can see that they are both staunch Republicans.

`Jaybird65`'s initial Tweet came from his reaction to an article in the *New York Times* stating that climate change was destroying the planet.

`Jaybird65` is quite puzzled. He can see that `Hunt78` is a Republican like him. He is also a hunter. So how can `Hunt78` believe in climate change?

This Twitter thread goes on for a massive number of raging Tweets.

However, we can see that the roots of fake news discussions lie in emotional reactions to the news. A rational approach to climate change would simply be:

- No matter what the cause is, the climate is changing.
- We do not need to take the economy down to change humans.
- We need to continue to build electric cars, more walking space in large cities, and better agricultural habits. We just need to do business in new ways that will most probably generate revenue.

But emotions are strong in humans!

Let's represent the process that leads from news to emotional and rational reactions.

Behavioral representation of fake news

Fake news starts with emotional reactions, builds up, and often leads to personal attacks.

Figure 13.2 represents the three-phase emotional reaction path to fake news when cognitive dissonance clogs our thinking process:

Phase 1: Incoming News

Two persons or groups of persons react to the news they obtained through their respective media: Facebook, Twitter, other social media, TV, radio, websites. Each source of information contains biased opinions.

Phase 2: Consensus

The two persons or groups of persons can agree or disagree. If they disagree, we will enter phase 3, during which the conflict might rage.

If they agree, the consensus stops the heat from building up, and the news is accepted as `real` news. However, even if all parties believe the news they are receiving is not fake, it does not mean that it is not fake. Here are some things that explain that news labeled as `not fake news` can be fake news:

- In the early 12th century, most people in Europe agreed that Earth was the center of the universe and that the solar system rotated around the Earth.
- In 1900, most people believed that there would never be such a thing as an airplane that would fly over oceans.
- In January 2020, most Europeans believed that COVID-19 was a virus impacting only China and not a global pandemic.

The bottom line is that a consensus between two parties or even a society as a whole does not mean that the incoming news is true or false. If two parties disagree, this will lead to a conflict:

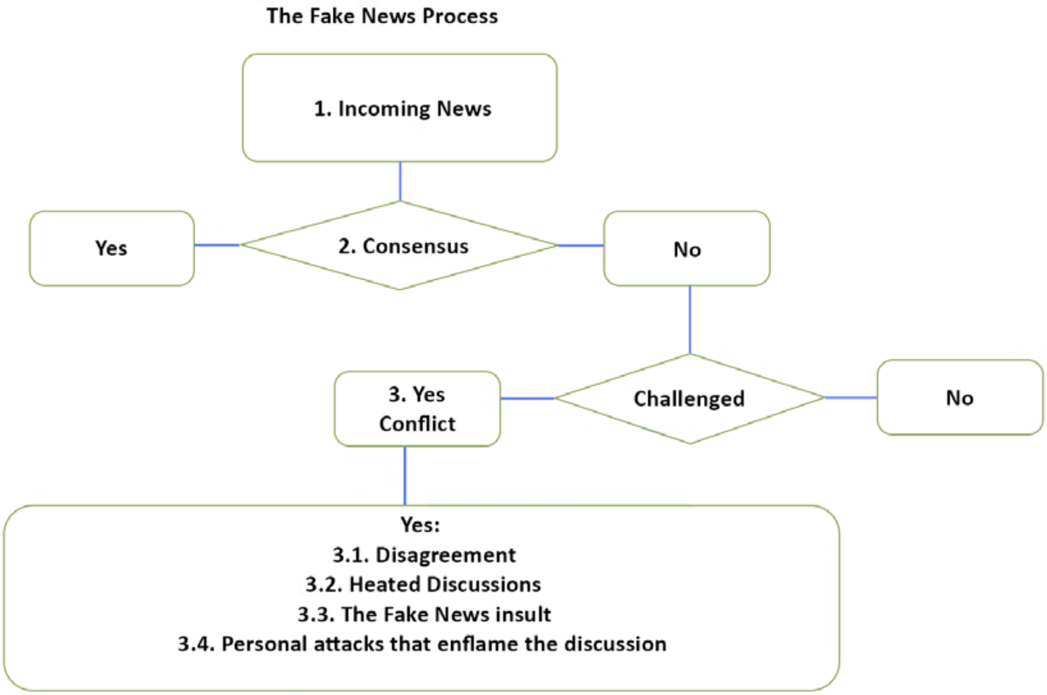


Figure 13.2: Representation of the path from news to a fake news conflict

Let's face it. On social media, members usually converge with others that have the same ideas and rarely change their minds no matter what. This representation shows that a person will often stick to their opinion expressed in a Tweet, and the conflict escalates as soon as somebody challenges their message!

Phase 3: Conflict

A fake news conflict can be divided into four phases:

- **3.1**: The conflict begins with a disagreement. Each party will Tweet or post messages on Facebook or other platforms. After a few exchanges, the conflict might wear out because each party is not interested in the topic.
- **3.2**: If we go back to the climate change discussion between `Jaybird65` and `Hunt78`, we know things can get nasty. The conversation is heating up!
- **3.3**: At one point, inevitably, the arguments of one party will become fake news. `Jaybird65` will get angry and show it in numerous Tweets and say that climate change due to humans is fake news. `Hunt78` will get angry and say that denying humans' contribution to climate change is fake news.
- **3.4**: These discussions often end in personal attacks. Godwin's law often enters the conversation even if we don't know how it got there. Godwin's law states that one party will find the worst reference possible to describe the other party at one point in a conversation. It sometimes comes out as "You liberals are like Hitler trying to force our economy down with climate change." This type of message can be seen on Twitter, Facebook, and other platforms. It even appears in real-time chats during presidential speeches on climate change.

Is there a rational approach to these discussions that could soothe both parties, calm them down, and at least reach a middle-ground consensus to move forward?

Let's try to build a rational approach with transformers and heuristics.

A rational approach to fake news

Transformers are the most powerful NLP tools ever. This section will first define a method that can take two parties engaged in conflict over fake news from an emotional level to a rational level.

We will then use transformer tools and heuristics. We will run transformer samples on gun control and former President Trump's Tweets during the COVID-19 pandemic. We will also describe heuristics that could be implemented with classical functions.

You can implement these transformer NLP tasks or other tasks of your choice. In any case, the roadmap and method can help teachers, parents, friends, co-workers, and anybody seeking the truth. Thus, your work will always be worthwhile!

Let's begin with the roadmap of a rational approach to fake news that includes transformers.

Defining a fake news resolution roadmap

Figure 13.3 defines a roadmap for a rational fake news analysis process. The process contains transformer NLP tasks and traditional functions:

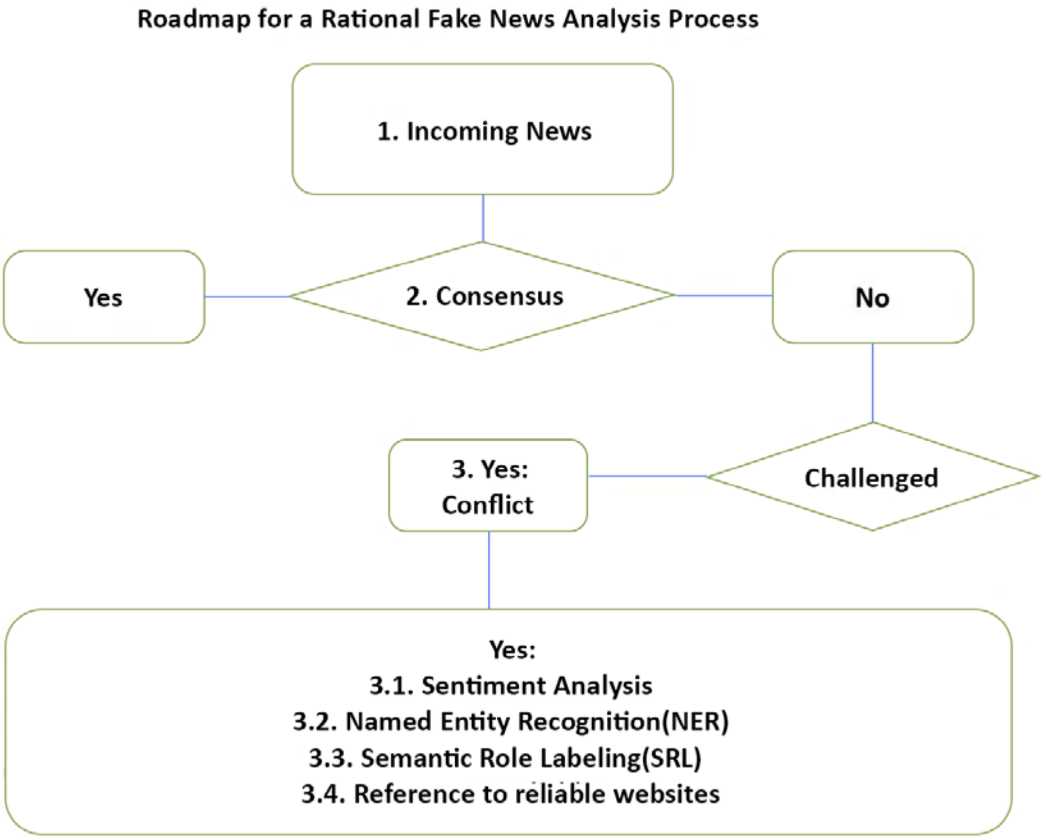

Roadmap for a Rational Fake News Analysis Process
1. Incoming News
Yes
2. Consensus
No
3. Yes:
Conflict
Challenged
Yes:
3.1. Sentiment Analysis
3.2. Named Entity Recognition(NER)
3.3. Semantic Role Labeling(SRL)
3.4. Reference to reliable websites

Figure 13.3: Going from emotional reactions to fake news to rational representations

We see that a rational process will nearly always begin once an emotional reaction has begun. The rational process must kick in as soon as possible to avoid building up emotional reactions that could interrupt the discussion.

Phase 3 now contains four tools:

- **3.1**: Sentiment Analysis to analyze the top-ranking "emotional" positive or negative words. We will use `AllenNLP` resources to run a RoBERTa large transformer in our `Fake_News.ipynb` notebook. We will use AllenNLP's visual tools to visualize the keywords and explanations. We introduced sentiment analysis in *Chapter 12*, *Detecting Customer Emotions to Make Predictions*.
- **3.2**: **Named Entity Recognition** (**NER**) to extract entities from social media messages for *Phase 3.4*. We described NER in *Chapter 11*, *Let Your Data Do the Talking: Story, Questions, and Answers*. We will use Hugging Face's BERT transformer model for the task. In addition, we will use AllenNLP.org's visual tools to visualize the entities and explanations.
- **3.3**: **Semantic Role Labeling** (**SRL**) to label verbs from social media messages for *Phase 3.4*. We described SRL in *Chapter 10*, *Semantic Role Labeling with BERT-Based Transformers*. We will use AllenNLP's BERT model in `Fake_News.ipynb`. We will use AllenNLP.org's visual tools to visualize the output of the labeling task.
- **3.4**: References to reliable websites will be described to show how classical coding can help.

Let's begin with gun control.

The gun control debate

The Second Amendment of the *Constitution of the United States* asserts the following rights:

```
A well regulated Militia, being necessary to the security of a free State, the
right of the people to keep and bear Arms, shall not be infringed.
```

America has been divided on this subject for decades:

- On the one hand, many argue that it is their right to bear firearms, and they do not want to endure gun control. They argue that it is fake news to contend that possessing weapons creates violence.
- On the other hand, many argue that bearing firearms is dangerous and that without gun control, the US will remain a violent country. They argue that it's fake news to contend that it is not dangerous to carry weapons.

We need to help each party. Let's begin with sentiment analysis.

Sentiment analysis

If you read Tweets, Facebook messages, YouTube chats during a speech, or any other social media, you will see that the parties are fighting a raging battle. You do not need a TV show. You can just eat your popcorn as the Tweet battles tear the parties apart!

Let's take a Tweet from one side and a Facebook message from the opposing side. I changed the members' names and paraphrased the text (not a bad idea considering the insults in the messages). Let's start with the pro-gun Tweet:

Pro-guns analysis

This Tweet is the honest opinion of a person:

```
Afirst78: I have had rifles and guns for years and never had a problem. I raised my kids right so they have guns too and never hurt anything except rabbits.
```

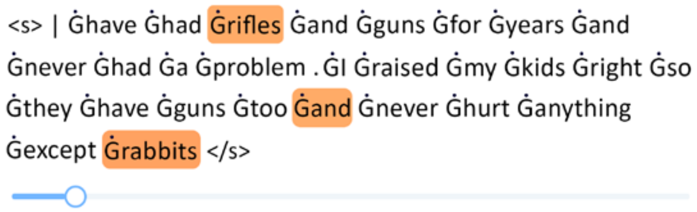
Let's run this in `Fake_News.ipynb`:

```
!echo '{"sentence": "I have had rifles and guns for years and never had
a problem. I raised my kids right so they have guns too and never hurt
anything except rabbits."}' | \
allennlp predict https://storage.googleapis.com/allennlp-public-models/
sst-roberta-large-2020.06.08.tar.gz -
```

The prediction is positive:

```
prediction:  {"logits": [1.9383275508880615, -1.6191326379776], "probs":
[0.9722791910171509, 0.02772079035639763]
```

We will now visualize the result on AllenNLP. **Simple Gradient Visualization** provides an explanation:



```
<s> | Ġhave Ġhad Ġrifles Ġand Ġguns Ġfor Ġyears Ġand
Ġnever Ġhad Ġa Ġproblem . ĠI Ġraised Ġmy Ġkids Ġright Ġso
Ġthey Ġhave Ġguns Ġtoo Ġand Ġnever Ġhurt Ġanything
Ġexcept Ġrabbits </s>
```

Figure 13.4: Simple Gradient Visualization of a sentence

The explanation shows that sentiment analysis on the Tweet by `Afirst78` highlights `rifles` + `and` + `rabbits`.

Results may vary at each run or over time. This is because transformer models are continuously trained and updated. However, the focus in this chapter is on the process, not a specific result.

We will pick up ideas and functions at each step. `Fake_News_FUNCTION_1` is the first function in this section:

`Fake_News_FUNCTION_1`: `rifles` + `and` + `rabbits` can be extracted and noted for further analysis. We can see that "rifles" are not "dangerous" in this example.

We will now analyze `NYS99`'s view that guns must be controlled.

Gun control analysis

`NYS99: "I have heard gunshots all my life in my neighborhood, have lost many friends, and am afraid to go out at night."`

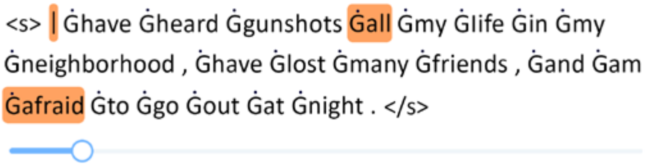
Let's first run the analysis in `Fake_News.ipynb`:

```
!echo '{"sentence": "I have heard gunshots all my life in my neighborhood,
have lost many friends, and am afraid to go out at night."}' | \
allennlp predict https://storage.googleapis.com/allennlp-public-models/
sst-roberta-large-2020.06.08.tar.gz -
```

The result is naturally negative:

```
prediction:  {"logits": [-1.3564586639404297, 0.5901418924331665],
"probs": [0.12492450326681137, 0.8750754594802856]
```

Let's find the keywords using AllenNLP online. We run the sample and can see that **Smooth Gradient Visualization** highlights the following:



<s> I Ġhave Ġheard Ġgunshots Ġall Ġmy Ġlife Ġin Ġmy
Ġneighborhood , Ġhave Ġlost Ġmany Ġfriends , Ġand Ġam
Ġafraid Ġto Ġgo Ġout Ġat Ġnight . </s>

Figure 13.5: Smooth Gradient Visualization of a sentence

The keyword `afraid` stands out for function 2 of this section. We now know that "afraid is associated with "guns."

We can see that the model has problems interpreting these cognitive dissonances. Our human critical thinking is still necessary!

`Fake_News_FUNCTION_2`: `afraid` and `guns` (the topic) can be extracted and noted for further analysis.

If we now put our two functions side by side, we can clearly understand why the two parties are fighting each other:

- `Fake_News_FUNCTION_1`: `rifles` + `and` + `rabbits`

  `Afirst78` probably lives in a mid-western state in the US. Many of these states have small populations, are very quiet, and enjoy low crime rates. `Afirst78` may never have traveled to a major city, enjoying the pleasure of a quiet life in the country.

- `Fake_News_FUNCTION_2`: `afraid` + the topic `guns`

  `NYS99` probably lives in a big city or a greater area of a major US city. Crime rates are often high, and violence is a daily phenomenon. `NYS99` may never have traveled to a mid-western state and seen how `Afirst78` lives.

These two honest but strong views prove why we need to implement solutions like those described in this chapter.

Better information is the key to fewer fake news battles.

We will follow our process and apply named entity recognition to our sentence.

Named entity recognition (NER)

This chapter shows that by using several transformer methods, the user will benefit from a broader perception of a message through different angles. An HTML page could sum up this chapter's transformer methods and even contain other transformer tasks in production mode.

We must now apply our process to the Tweet and Facebook message, although we can see no entities in the messages. However, the program does not know that. We will only run the first message to illustrate this step of the process.

We will first install Hugging Face transformers:

```
!pip install -q transformers
from transformers import pipeline
from transformers import AutoTokenizer,
AutoModelForSequenceClassification,AutoModel
```

Now, we can run the first message:

```
nlp_token_class = pipeline('ner')
nlp_token_class('I have had rifles and guns for years and never had a
problem. I raised my kids right so they have guns too and never hurt
anything except rabbits.')
```

The output produces no result since there are no entities. However, this doesn't mean it should be taken out of the pipeline. Another sentence might contain the name of the location of a person providing clues on the culture in that area.

Let's check the model we are using before we move on:

```
nlp_token_class.model.config
```

The output shows that the model uses 9 labels and 1,024 features for the attention layers:

```
BertConfig {
  "_num_labels": 9,
  "architectures": [
    "BertForTokenClassification"
  ],
  "attention_probs_dropout_prob": 0.1,
  "directionality": "bidi",
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 1024,
  "id2label": {
    "0": "O",
    "1": "B-MISC",
    "2": "I-MISC",
    "3": "B-PER",
    "4": "I-PER",
    "5": "B-ORG",
    "6": "I-ORG",
    "7": "B-LOC",
    "8": "I-LOC"
  },
```

We are using a BERT 24-layer transformer model. If you wish to explore the architecture, run `nlp_token_class.model`.

We will now run SRL on the messages.

Semantic Role Labeling (SRL)

We will continue to run `Fake_News.ipynb` cell by cell in the order found in the notebook. We will examine both points of view.

Let's start with a pro-gun perspective.

Pro-guns SRL

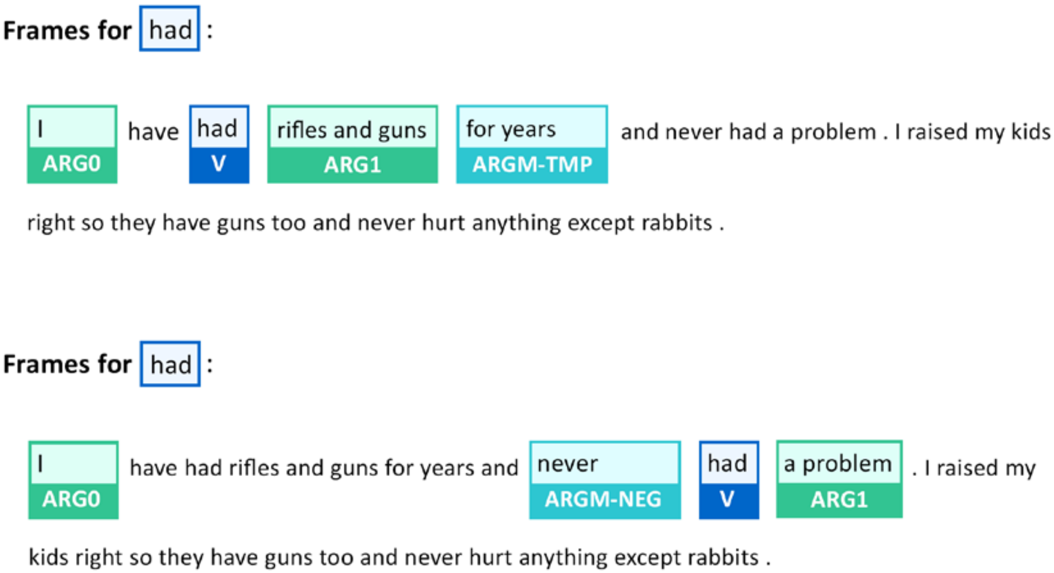
We will first run the following cell in `Fake_News.ipynb`:

```
!echo '{"sentence": "I have had rifles and guns for years and never had
a problem. I raised my kids right so they have guns too and never hurt
anything except rabbits."}' | \
allennlp predict https://storage.googleapis.com/allennlp-public-models/
bert-base-srl-2020.03.24.tar.gz -
```

The output is very detailed and can be useful if you wish to investigate or parse the labels in detail, as shown in this excerpt:

```
prediction:  {"verbs": [{"verb": "had", "description": "[ARG0: I] have [V:
had] [ARG1: rifles and guns] [ARGM-TMP: for years] and never had a problem
...
```

Now let's go into visual detail on AllenNLP in the **Semantic Role Labeling** section. We first run the SRL task for this message. The first verb, `had`, shows that `Afirst78` is an *experienced gun owner*:



Frames for had :

I have had rifles and guns for years and never had a problem . I raised my kids right so they have guns too and never hurt anything except rabbits .

Frames for had :

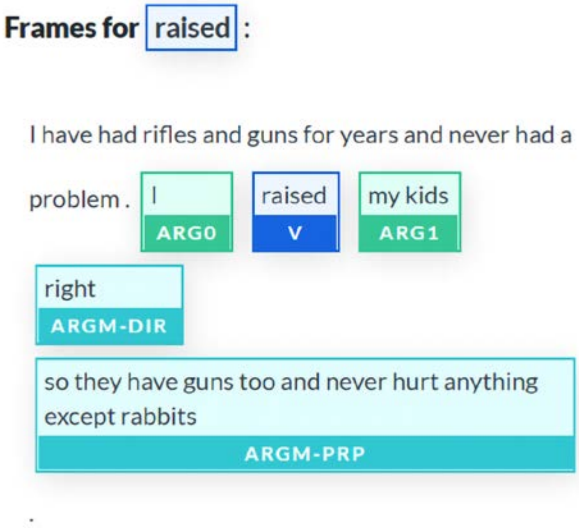
I have had rifles and guns for years and never had a problem . I raised my kids right so they have guns too and never hurt anything except rabbits .

Figure 13.6: SRL for the verb "had"

The arguments of had sum up `Afirst78`'s experience: `I` + `rifles and guns` + `for years`.

The second frame for had adds the information `I` + `never` + `had` + `a problem`

The arguments of `raised` display `Afirst78`'s parental experience:



Frames for raised :

I have had rifles and guns for years and never had a problem . I raised my kids right so they have guns too and never hurt anything except rabbits .

I
ARG0
raised
V
my kids
ARG1
right
ARGM-DIR
so they have guns too and never hurt anything except rabbits
ARGM-PRP

Figure 13.7: SRL verb and arguments for the verb "raised"

The arguments explain many pro-gun positions: `my kids` + …`have guns too and never hurt anything`.

The results may vary from one run to another or when the model is updated, but the process remains the same.

We can add what we found here to our collection of functions with some parsing:

- `Fake_News_FUNCTION_3`: `I` + `rifles and guns` + `for years`
- `Fake_News_FUNCTION_4`: `my kids` + `have guns` too and `never hurt anything`

Now let's explore the gun control message.

Gun control SRL

We will first run the Facebook message in `Fake_News.ipynb`. We will just continue to run the notebook cell by cell in the order they were created in the notebook:

```
!echo '{"sentence": "I have heard gunshots all my life in my neighborhood,
have lost many friends, and am afraid to go out at night."}' | \
allennlp predict https://storage.googleapis.com/allennlp-public-models/
bert-base-srl-2020.03.24.tar.gz -
```

The result labels the key verbs in the sequence in detail, as shown in the following excerpt:

```
prediction:  {"verbs": [{"verb": "heard", "description": "[ARG0: I] have
[V: heard] [ARG1: gunshots all my life in my neighborhood]"
```

We continue to apply our process, go to AllenNLP, and then to the **Semantic Role Labeling** section. We enter the sentence and run the transformer model. The verb `heard` shows the tough reality of this message:

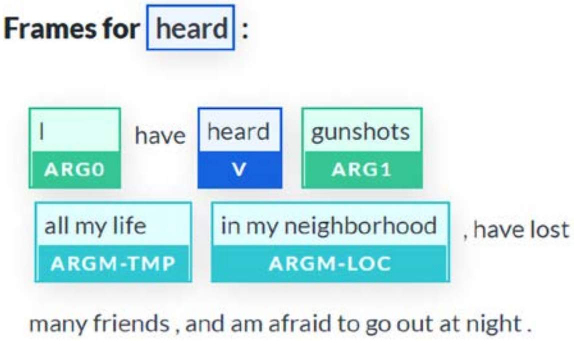
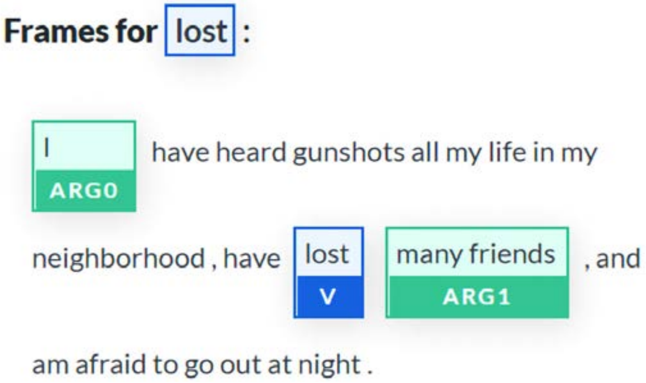


Figure 13.8: SRL representation of the verb "heard"

We can quickly parse the words for our fifth function:

- `Fake_News_FUNCTION_5`: `heard` + `gunshots` + `all my life`

The verb `lost` shows significant arguments related to it:



Frames for lost :

I have heard gunshots all my life in my neighborhood , have lost many friends , and am afraid to go out at night .

Figure 13.9: SRL representation of the verb "lost"

We have what we need for our sixth function:

- `Fake_News_FUNCTION_6`: `lost` + `many` + `friends`

It is good to suggest reference sites to the user once different transformer models have clarified each aspect of a message.

Reference sites

We have run the transformers on NLP tasks and described traditional heuristic hard coding that needs to be developed to parse the data and generate six functions.

> Bear in mind that the results may vary from one run to another. The six functions were generated at different times and provided slightly different results from the previous section. However, the main ideas remain the same. Let's now focus on these six functions.

- Pro-guns: `Fake_News_FUNCTION_1`: `never` + `problem` + `guns`
- Gun control: `Fake_News_FUNCTION_2`: `heard` + `afraid` + `guns`
- Pro-guns: `Fake_News_FUNCTION_3`: `I` + `rifles and guns` + `for years`
- Pro-guns: `Fake_News_FUNCTION_4`: `my kids` + `have guns` + `never hurt anything`

- Gun control: `Fake_News_FUNCTION_5`: `heard` + `gunshots` + `all my life`
- Gun control: `Fake_News_FUNCTION_6`: `lost` + `many` + `friends`

Let's reorganize the list and separate both perspectives and draw some conclusions to decide our actions.

Pro-guns and gun control

The pro-gun arguments are honest, but they show that there is a lack of information on what is going on in major cities in the US:

- Pro-guns: `Fake_News_FUNCTION_1`: `never` + `problem` + `guns`
- Pro-guns: `Fake_News_FUNCTION_3`: `I` + `rifles and guns` + `for years`
- Pro-guns: `Fake_News_FUNCTION_4`: `my kids` + `have guns` + `never hurt anything`

The gun control arguments are honest, but they show that there is a lack of information on how large quiet areas of the Midwest can be:

- Gun control: `Fake_News_FUNCTION_2`: `heard` + `afraid` + `guns`
- Gun control: `Fake_News_FUNCTION_5`: `heard` + `gunshots` + `all my life`
- Gun control: `Fake_News_FUNCTION_6`: `lost` + `many` + `friends`

Each function can be developed to inform the other party.

For example, let's take `FUNCTION1` and express it in pseudocode:

```
Def FUNCTION1:
call FUNCTIONs 2+5+6 Keywords and simplify
Google search=afraid guns lost many friends gunshots
```

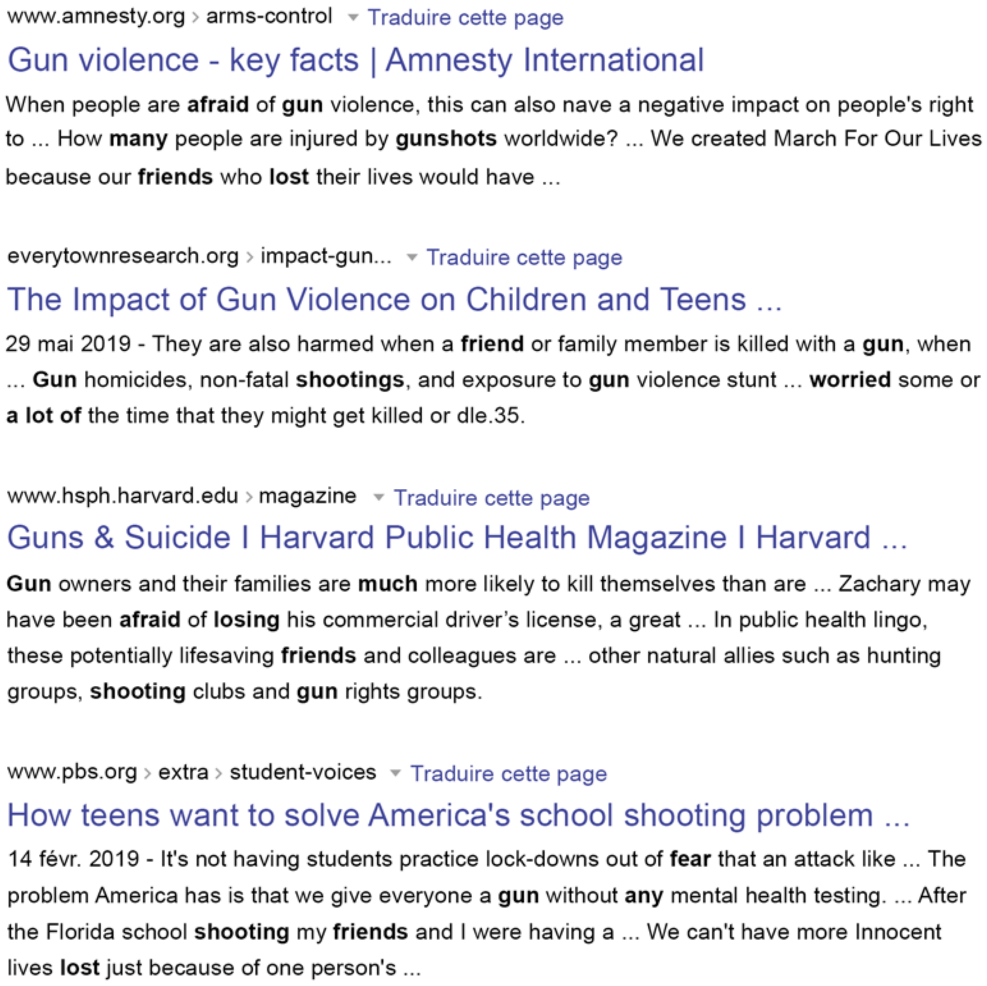
The goal of the process is:

- First, run transformer models to *deconstruct and explain* the messages. Using an NLP transformer is like a mathematical calculator. It can produce good results, but it takes a free-thinking human mind to interpret them!
- Then, ask a trained NLP human user to be *proactive*, *search*, and *read* information better.

Transformer models help users understand messages more deeply; they don't think for them! We are trying to help users, not lecture or brainwash them!

Parsing would be required to process the results of the functions. However, if we had hundreds of social media messages, we could automatically let our program do the whole job.

The links will change as Google modifies its searches. However, the first links that appear are interesting to show to pro-gun advocates:



www.amnesty.org › arms-control ▾ Traduire cette page

Gun violence - key facts | Amnesty International

When people are **afraid** of **gun** violence, this can also nave a negative impact on people's right to ... How **many** people are injured by **gunshots** worldwide? ... We created March For Our Lives because our **friends** who **lost** their lives would have ...

everytownresearch.org › impact-gun... ▾ Traduire cette page

The Impact of Gun Violence on Children and Teens ...

29 mai 2019 - They are also harmed when a **friend** or family member is killed with a **gun**, when ... **Gun** homicides, non-fatal **shootings**, and exposure to **gun** violence stunt ... **worried** some or **a lot of** the time that they might get killed or dle.35.

www.hsph.harvard.edu › magazine ▾ Traduire cette page

Guns & Suicide I Harvard Public Health Magazine I Harvard ...

Gun owners and their families are **much** more likely to kill themselves than are ... Zachary may have been **afraid** of **losing** his commercial driver's license, a great ... In public health lingo, these potentially lifesaving **friends** and colleagues are ... other natural allies such as hunting groups, **shooting** clubs and **gun** rights groups.

www.pbs.org › extra › student-voices ▾ Traduire cette page

How teens want to solve America's school shooting problem ...

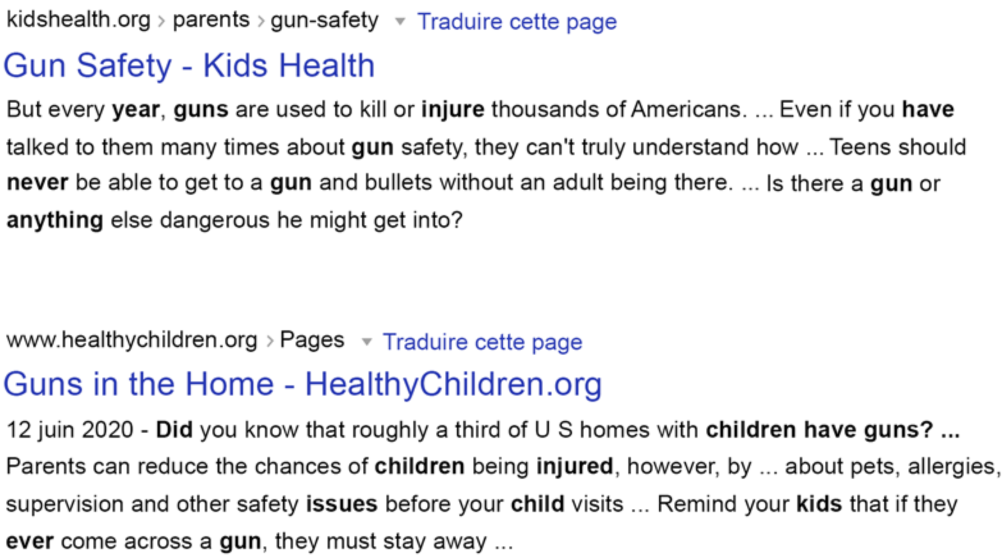
14 févr. 2019 - It's not having students practice lock-downs out of **fear** that an attack like ... The problem America has is that we give everyone a **gun** without **any** mental health testing. ... After the Florida school **shooting** my **friends** and I were having a ... We can't have more Innocent lives **lost** just because of one person's ...

Figure 13.10: Guns and violence

Let's imagine we are searching gun control advocates with the following pseudocode:

```
Def FUNCTION2:
call FUNCTIONs 1+3+4 Keywords and simplify
Google search=never problem guns for years kids never hurt anything
```

The Google search returned no clear positive results in favor of pro-gun advocates. The most interesting ones are neutral and educational:



kidshealth.org › parents › gun-safety ▾ Traduire cette page

Gun Safety - Kids Health

But every **year**, **guns** are used to kill or **injure** thousands of Americans. ... Even if you **have** talked to them many times about **gun** safety, they can't truly understand how ... Teens should **never** be able to get to a **gun** and bullets without an adult being there. ... Is there a **gun** or **anything** else dangerous he might get into?

www.healthychildren.org › Pages ▾ Traduire cette page

Guns in the Home - HealthyChildren.org

12 juin 2020 - **Did** you know that roughly a third of U S homes with **children have guns? ...** Parents can reduce the chances of **children** being **injured**, however, by ... about pets, allergies, supervision and other safety **issues** before your **child** visits ... Remind your **kids** that if they **ever** come across a **gun**, they must stay away ...

Figure 13.11: Gun safety

You could run automatic searches on Amazon's bookstore, magazines, and other educational material.

Most importantly, it is essential for people with opposing ideas to talk to each other without getting into a fight. Understanding each other is the best way to develop empathy on both sides.

One might be tempted to trust social media companies. *I recommend never letting a third party act as a proxy for your mind process*. Use transformer models to deconstruct messages but remain proactive!

A consensus on this topic could be to agree on following safety guidelines with gun possession. For example, one can choose either not to have guns at home or lock them up safely, so children do not have access to them.

Let’s move on to COVID-19 and former President Trump’s Tweets.

COVID-19 and former President Trump’s Tweets

No matter your political opinion, there is so much being said by Donald Trump and about Donald Trump that it would take a book in itself to analyze all of the information! This is a technical, not a political book, so we will analyze the Tweets scientifically.

We described an educational approach to fake news in the *Gun control* section of this chapter. We do not need to go through the whole process again.

We implemented and ran AllenNLP’s SRL task with a BERT model in our `Fake_News.ipynb` notebook in the *Gun control* section.

In this section, we will focus on the logic of fake news. We will run the BERT model on SRL and visualize the results on AllenNLP’s website.

Now, let’s go through some presidential Tweets on COVID-19.

Semantic Role Labeling (SRL)

SRL is an excellent educational tool for all of us. We tend just to read Tweets passively and listen to what others say about them. Breaking messages down with SRL is a good way to develop social media analytical skills to distinguish fake from accurate information.

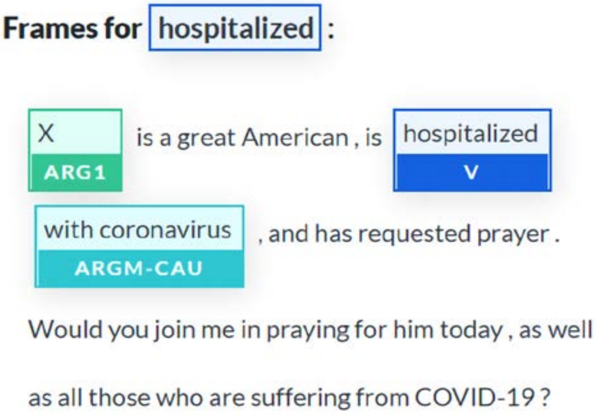
I recommend using SRL transformers for educational purposes in class. A young student can enter a Tweet and analyze each verb and its arguments. It could help younger generations become active readers on social media.

We will first analyze a relatively undivided Tweet and then a conflictual Tweet:

Let’s analyze the latest Tweet found on July 4 while writing this book. I took the name of the person who is referred to as a “Black American” out and paraphrased some of the former President’s text:

```
X is a great American, is hospitalized with coronavirus, and has requested prayer.
Would you join me in praying for him today, as well as all those who are suffering
from COVID-19?
```

Let's go to AllenNLP's **Semantic Role Labeling** section, run the sentence, and look at the result. The verb `hospitalized` shows the member is staying close to the facts:



Frames for hospitalized :

X is a great American , is hospitalized with coronavirus , and has requested prayer .

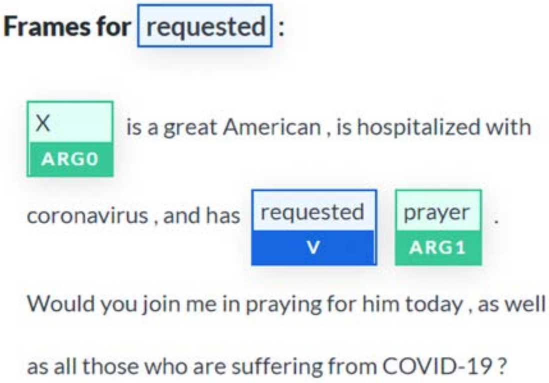
Would you join me in praying for him today , as well

as all those who are suffering from COVID-19 ?

Figure 13.12: SRL arguments of the verb "hospitalized"

The message is simple: `X` + `hospitalized` + `with coronavirus`.

The verb `requested` shows that the message is becoming political:



Frames for requested :

X is a great American , is hospitalized with coronavirus , and has requested prayer .

Would you join me in praying for him today , as well

as all those who are suffering from COVID-19 ?

Figure 13.13: SRL arguments of the verb "requested"

We don't know if the person requested the former President to pray or decided he would be the center of the request.

A good exercise would be to display an HTML page and ask the users what they think. For example, the users could be asked to look at the results of the SRL task and answer the two following questions:

```
Was former President Trump asked to pray, or did he deviate a request made to others for political reasons?
```

```
Is the fact that former President Trump states that he was indirectly asked to pray for X fake news or not?
```

You can think about it and decide for yourself!

Let's have a look at one banned from Twitter. I took the names out and paraphrased it and toned it down. Still, when we run it on AllenNLP and visualize the results, we get some surprising SRL outputs.

Here is the toned-down and paraphrased Tweet:

```
These thugs are dishonoring the memory of X.
```

```
When the looting starts, actions must be taken.
```

Although I suppressed the main part of the original Tweet, we can see that the SRL task shows the bad associations made in the Tweet:

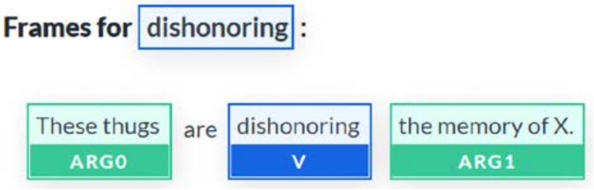


Figure 13.14: SRL arguments of the verb "dishonoring"

An educational approach to this would be to explain that we should not associate the arguments `thugs` and `memory` and `looting`. They do not fit together at all.

An important exercise would be to ask a user why the SRL arguments do not fit together.

I recommend many such exercises so that the transformer model users develop SRL skills to have a critical view of any topic presented to them.

Critical thinking is the best way to stop the propagation of the fake news pandemic!

We have gone through rational approaches to fake news with transformers, heuristics, and instructive websites. However, in the end, a lot of the heat in fake news debates boils down to emotional and irrational reactions.

In a world of opinion, you will never find an entirely objective transformer model that detects fake news since opposing sides never agree on what the truth is in the first place! One side will agree with the transformer model's output. Another will say that the model is biased and built by enemies of their opinion!

The best approach is listening to others and keeping the heat down!

Before we go

This chapter focused more on applying transformers to a problem than finding a silver bullet transformer model, which does not exist.

You have two main options to solve an NLP problem: find new transformer models or create reliable, durable methods to implement transformer models.

We will now conclude the chapter and move on to interpret transformer models.

Summary

Fake news begins deep inside our emotional history as humans. When an event occurs, emotions take over to help us react quickly to a situation. We are hardwired to react strongly when we are threatened.

Fake news spurs strong reactions. We fear that this news could temporarily or permanently damage our lives. Many of us believe climate change could eradicate human life from Earth. Others believe that if we react too strongly to climate change, we might destroy our economies and break society down. Some of us believe that guns are dangerous. Others remind us that the Second Amendment of the *United States Constitution* gives us the right to possess a gun in the US.

We went through other raging conflicts over COVID-19, former President Trump, and climate change. In each case, we saw that emotional reactions are the fastest ones to build up into conflicts.

We then designed a roadmap to take the emotional perception of fake news to a rational level. We used some transformer NLP tasks to show that it is possible to find key information in Tweets, Facebook messages, and other media.

We used news perceived by some as real news and others as fake news to create a rationale for teachers, parents, friends, co-workers, or just people talking. We added classical software functions to help us on the way.

At this point, you have a toolkit of transformer models, NLP tasks, and sample datasets in your hands.

You can use artificial intelligence for the good of humanity. It's up to you to take these transformer tools and ideas to implement them to make the world a better place for all.

A good way to understand transformers is to visualize their internal process. We will analyze how a transformer gradually builds a representation of a sequence in the next chapter, *Interpreting Black Box Transformer Models*.

Questions

1. News labeled as fake news is always fake. (True/False)
2. News that everybody agrees with is always accurate. (True/False)
3. Transformers can be used to run sentiment analysis on Tweets. (True/False)
4. Key entities can be extracted from Facebook messages with a DistilBERT model running NER. (True/False)
5. Key verbs can be identified from YouTube chats with BERT-based models running SRL. (True/False)
6. Emotional reactions are a natural first response to fake news. (True/False)
7. A rational approach to fake news can help clarify one's position. (True/False)
8. Connecting transformers to reliable websites can help somebody understand why some news is fake. (True/False)
9. Transformers can make summaries of reliable websites to help us understand some of the topics labeled as fake news. (True/False)
10. You can change the world if you use AI for the good of us all. (True/False)

References

- *Daniel Kahneman*, 2013, *Thinking, Fast and Slow*
- Hugging Face pipelines: `https://huggingface.co/transformers/main_classes/pipelines.html`
- The Allen Institute for AI: `https://allennlp.org/`

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the authors:

```
https://www.packt.link/Transformers
```

14

Interpreting Black Box Transformer Models

Million- to billion-parameter transformer models seem like huge black boxes that nobody can interpret. As a result, many developers and users have sometimes been discouraged when dealing with these mind-blowing models. However, recent research has begun to solve the problem with innovative, cutting-edge tools.

It is beyond the scope of this book to describe all of the explainable AI methods and algorithms. So instead, this chapter will focus on ready-to-use visual interfaces that provide insights for transformer model developers and users.

The chapter begins by installing and running `BertViz` by *Jesse Vig*. Jesse did quite an excellent job of building a visual interface that shows the activity in the attention heads of a BERT transformer model. BertViz interacts with the BERT models and provides a well-designed interactive interface.

We will continue to focus on visualizing the activity of transformer models with the **Language Interpretability Tool** (**LIT**). LIT is a non-probing tool that can use PCA or UMAP to represent transformer model predictions. We will go through PCA and use UMAP as well.

Finally, we will visualize a transformer's journey through the layers of a BERT model with dictionary learning. **Local Interpretable Model-agnostic Explanations** (**LIME**) provides practical functions to visualize how a transformer learns how to understand language. The method shows that transformers often begin by learning a word, then the word in the sentence context, and finally long-range dependencies.

By the end of the chapter, you will be able to interact with users to show visualizations of the activity of transformer models. BertViz, LIT, and visualizations through dictionary learning still have a long way to go. However, these nascent tools will help developers and users understand how transformer models work.

This chapter covers the following topics:

- Installing and running BertViz
- Running BertViz's interactive interface
- The difference between probing and non-probing methods
- A **Principal Component Analysis** (**PCA**) reminder
- Running LIT to analyze transformer outputs
- Introducing LIME
- Running transformer visualization through dictionary learning
- Word-level polysemy disambiguation
- Visualizing low-level, mid-level, and high-level dependencies
- Visualizing key transformer factors

Our first step will begin by installing and using BertViz.

Transformer visualization with BertViz

Jesse Vig's article, *A Multiscale Visualization of Attention in the Transformer Model*, 2019, recognizes the effectiveness of transformer models. However, *Jesse Vig* explains that deciphering the attention mechanism is challenging. The paper describes the process of BertViz, a visualization tool.

BertViz can visualize attention head activity and interpret a transformer model's behavior.

BertViz was first designed to visualize BERT and GPT-3 models. In this section, we will visualize the activity of a BERT model.

Let's now install and run BertViz.

Running BertViz

It only takes five steps to visualize transformer attention heads and interact with them.

Open the `BertViz.ipynb` notebook in the `Chapter14` directory in the GitHub repository of this book.

The first step is to install `BertViz` and the requirements.

Step 1: Installing BertViz and importing the modules

The notebook installs `BertViz`, Hugging Face transformers, and the other basic requirements to implement the program:

```
!pip install bertViz
from bertViz import head_view, model_view
from transformers import BertTokenizer, BertModel
```

The head view and model view libraries are now imported. We will now load the BERT model and tokenizer.

Step 2: Load the models and retrieve attention

BertViz supports BERT, GPT-2, RoBERTa, and other models. You can consult BertViz on GitHub for more information: `https://github.com/jessevig/BertViz`.

In this section, we will run a `bert-base-uncased` model and a pretrained tokenizer:

```
# Load model and retrieve attention
model_version = 'bert-base-uncased'
do_lower_case = True
model = BertModel.from_pretrained(model_version, output_attentions=True)
tokenizer = BertTokenizer.from_pretrained(model_version, do_lower_case=do_
lower_case)
```

We now enter our two sentences. You can try different sequences to analyze the behavior of the model. `sentence_b_start` will be necessary for *Step: 5 Model view*:

```
sentence_a = "A lot of people like animals so they adopt cats"
sentence_b = "A lot of people like animals so they adopt dogs"
inputs = tokenizer.encode_plus(sentence_a, sentence_b, return_
tensors='pt', add_special_tokens=True)
token_type_ids = inputs['token_type_ids']
input_ids = inputs['input_ids']
attention = model(input_ids, token_type_ids=token_type_ids)[-1]
sentence_b_start = token_type_ids[0].tolist().index(1)
input_id_list = input_ids[0].tolist() # Batch index 0
tokens = tokenizer.convert_ids_to_tokens(input_id_list)
```

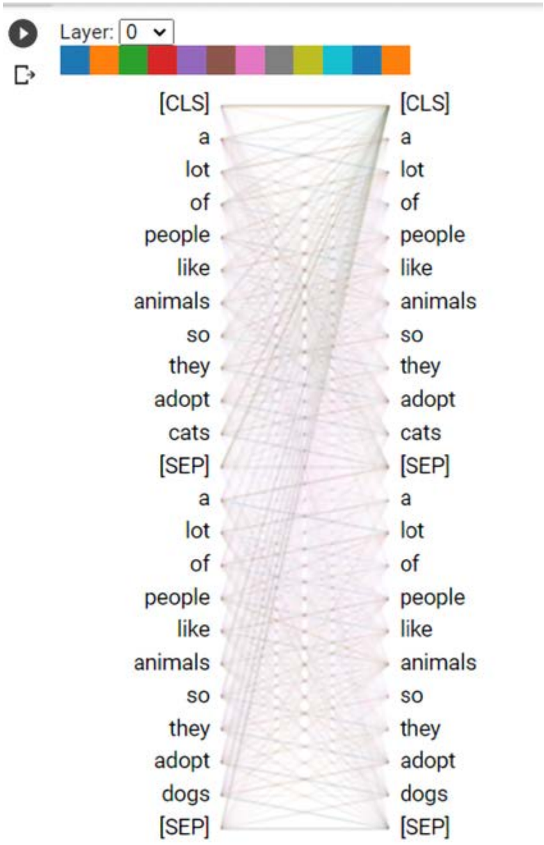
And that's it! We are ready to interact with the visualization interface.

Step 3: Head view

We just have one final line to add to activate the visualization of the attention heads:

```
head_view(attention, tokens)
```

The words of the first layer (layer 0) are not the actual tokens, but the interface is educational. The 12 attention heads of each layer are displayed in different colors. The default view is set to layer 0, as shown in *Figure 14.1*:



Layer: 0

Figure 14.1: The visualization of attention heads

We are now ready to explore attention heads.

Step 4: Processing and displaying attention heads

Each color above the two columns of tokens represents an attention head of the layer number. Choose a layer number and click on an attention head (color).

The words in the sentences are broken down into tokens in the attention. However, in this section, the word `tokens` loosely refers to `words` to help us understand how the transformer heads work.

I focused on the word `animals` in *Figure 14.2*:

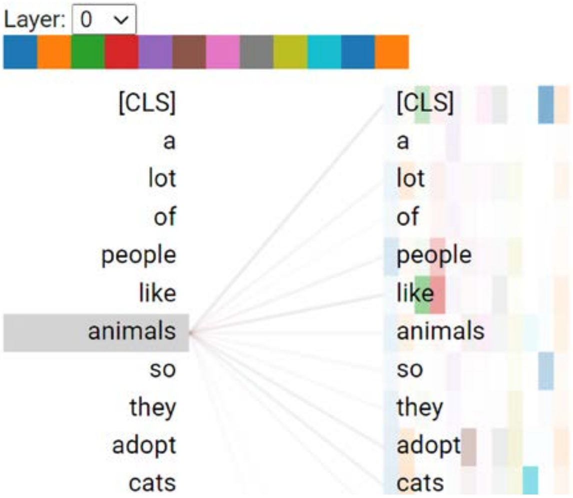


Figure 14.2: Selecting a layer, an attention head, and a token

`BertViz` shows that the model made a connection between `animals` and several words. This is normal since we are only at layer 0.

Layer 1 begins to isolate words `animals` is related to, as shown in *Figure 14.3*:

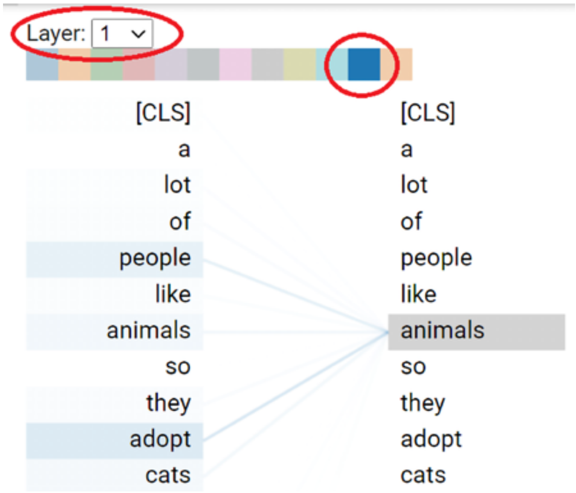


Figure 14.3: Visualizing the activity of attention head 11 of layer 1

Attention head 11 makes a connection between `animals`, `people`, and `adopt`.

If we click on `cats`, some interesting connections are shown in *Figure 14.4*:

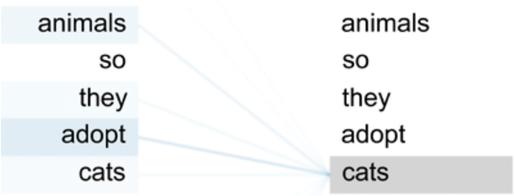


Figure 14.4: Visualizing the connections between cats and other tokens

The word `cats` is now associated with `animals`. This connection shows that the model is learning that cats are animals.

You can change the sentences and then click on the layers and attention heads to visualize how the transformer makes connections. You will find limits, of course. The good and bad connections will show you how transformers work and fail. Both cases are valuable for explaining how transformers behave and why they require more layers, parameters, and data.

Let's see how `BertViz` displays the model view.

Step 5: Model view

It only takes one line to obtain the model view of a transformer with `BertViz`:

```
model_view(attention, tokens, sentence_b_start)
```

`BertViz` displays all of the layers and heads in one view, as shown in the view excerpt in *Figure 14.5*:

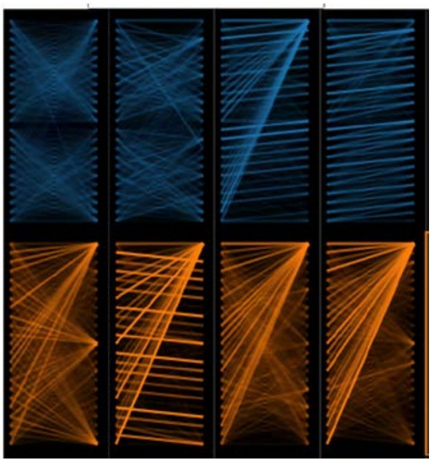


Figure 14.5: Model view mode of BertViz

If you click on one of the heads, you will obtain a head view with word-to-word and sentence-to-sentence options. You can then go through the attention heads to see how the transformer model makes better representations as it progresses through the layers. For example, *Figure 14.6* shows the activity of an attention head in the first layers:

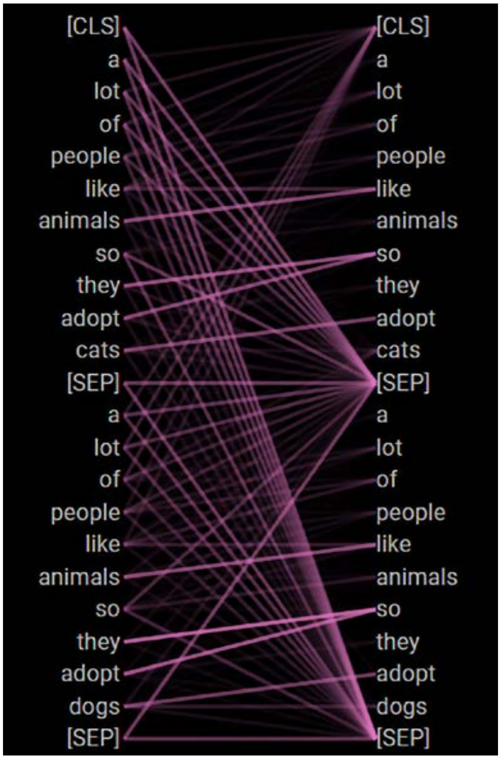


Figure 14.6: Activity of an attention head in the lower layers of the model

Sometimes, the representation makes connections between the separator, `[SEP]`, and words, which does not make much sense. However, sometimes tokens are not activated in every attention head of every layer. Also, the level of training of a transformer model limits the quality of the interpretation.

In any case, `BertViz` remains an interesting educational tool and interpretability tool for transformer models.

Let's now run the intuitive LIT tool.

LIT

LIT's visual interface will help you find examples that the model processes incorrectly, dig into similar examples, see how the model behaves when you change a context, and more language issues related to transformer models.

LIT does not display the activities of the attention heads like `BertViz` does. However, it's worth analyzing why things went wrong and trying to find solutions.

You can choose a **Uniform Manifold Approximation and Projection** (**UMAP**) visualization or a PCA projector representation. PCA will make more linear projections in specific directions and magnitude. UMAP will break its projections down into mini-clusters. Both approaches make sense depending on how far you want to go when analyzing the output of a model. You can run both and obtain different perspectives of the same model and examples.

This section will use PCA to run LIT. Let's begin with a brief reminder of how PCA works.

PCA

PCA takes data and represents it at a higher level.

Imagine you are in your kitchen. Your kitchen is a 3D cartesian coordinate system. The objects in your kitchen are all at specific x, y, z coordinates too.

You want to cook a recipe and gather the ingredients on your kitchen table. Your kitchen table is a higher-level representation of the recipe in your kitchen.

The kitchen table is using a cartesian coordinate system too. But when you extract the *main features* of your kitchen to represent the recipe on your kitchen table, you are performing PCA. This is because you have displayed the principal components that fit together to make a specific recipe.

The same representation can be applied to NLP. For example, a dictionary is a list of words. But the words that mean something together constitute a representation of the principal components of a sequence.

The PCA representation of sequences in LIT will help visualize the outputs of a transformer.

The main steps to obtain an NLP PCA representation are:

- **Variance**: The numerical variance of a word in a dataset; the frequency and frequency of its meaning, for example.
- **Covariance**: The variance of more than one word is related to that of another word in the dataset.

- **Eigenvalues and eigenvectors**: To obtain a representation in the cartesian system, we need the vectors and magnitudes representation of the covariances. The eigenvectors will provide the direction of the vectors. The eigenvalues will provide their magnitudes.
- **Deriving the data**: The last step is to apply the feature vectors to the original dataset by multiplying the row feature vector by the row data:
- **Data to display** = row of feature vector * row of data

PCA projections provide a clear linear visualization of the data points to analyze.

Let's now run LIT.

Running LIT

You can run LIT online or open it in a Google Colaboratory notebook. Click on the following link to access both options:

- `https://pair-code.github.io/lit/`

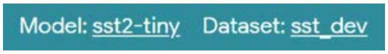
The tutorial page contains several types of NLP tasks to analyze:

- `https://pair-code.github.io/lit/tutorials/`

In this section, we will run LIT online and explore a sentiment analysis classifier:

- `https://pair-code.github.io/lit/tutorials/sentiment/`

Click on **Explore this demo yourself**, and you will enter the intuitive LIT interface. The transformer model is a small transformer model:



Model: sst2-tiny Dataset: sst_dev

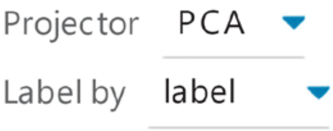
Figure 14.7: Selecting a model

You can change the model by clicking on the model. You can test this type of model and similar ones directly on Hugging Face on its hosted API page:

`https://huggingface.co/sshleifer/tiny-distilbert-base-uncased-finetuned-sst-2-english`

The NLP models might change on LIT's online version based on subsequent updates. The concepts remain the same, just the models change.

Let's begin by selecting the PCA projector and the binary 0 or 1 classification label of the sentiment analysis of each example:

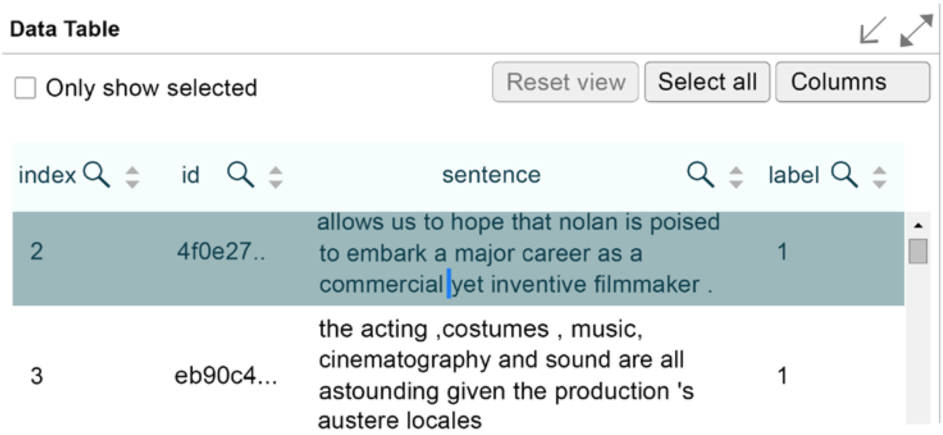


Projector PCA

Label by label

Figure 14.8: Selecting the projector and type of label

We then go to the data table and click on a **sentence** and its classification **label**:

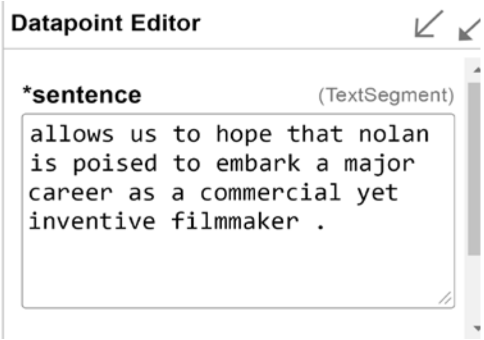


index	id	sentence	label
2	4f0e27..	allows us to hope that nolan is poised to embark a major career as a commercial yet inventive filmmaker .	1
3	eb90c4...	the acting ,costumes , music, cinematography and sound are all astounding given the production 's austere locales	1

Figure 14.9: Selecting a sentence

The algorithm is stochastic so the output can vary from one run to another.

The sentence will also appear in the datapoint editor:



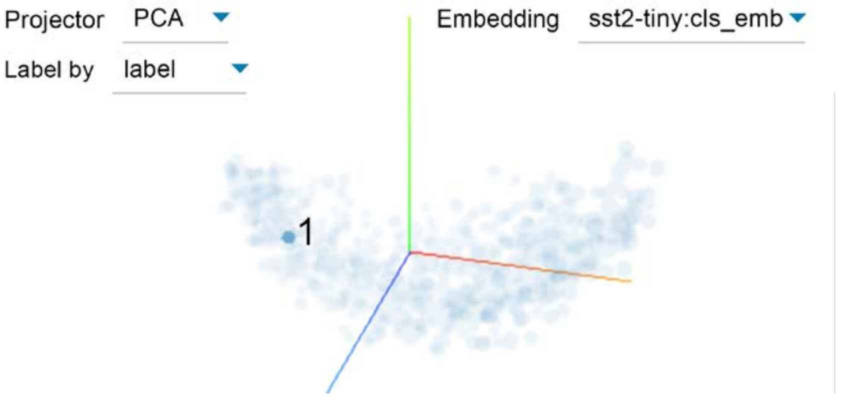
*sentence (TextSegment)

```
allows us to hope that nolan
is poised to embark a major
career as a commercial yet
inventive filmmaker .
```

Figure 14.10: Datapoint editor

The datapoint editor allows you to change the context of the sentence. For example, you might want to find out what went wrong with a counterfactual classification that should have been in one class but ended up in another one. You can change the context of the sentence until it appears in the correct class to understand how the model works and why it made a mistake.

The sentence will appear in the PCA projector with its classification:



Projector PCA

Embedding sst2-tiny:cls_emb

Label by label

Figure 14.11: PCA projector in a positive cluster

You can click the data points in the PCA projector, and the sentences will appear in the datapoint editor under the sentence you selected. That way, you can compare results.

LIT contains a wide range of interactive functions you can explore and use.

> The results obtained in LIT are not always convincing. However, LIT provides valuable insights in many cases. Also, it is essential to get involved in these emerging tools and techniques.

Let's now visualize transformer layers through dictionary learning.

Transformer visualization via dictionary learning

Transformer visualization via dictionary learning is based on transformer factors.

Transformer factors

A transformer factor is an embedding vector that contains contextualized words. A word with no context can have many meanings, creating a polysemy issue. For example, the word `separate` can be a verb or an adjective. Furthermore, `separate` can mean disconnect, discriminate, scatter, and has many other definitions.

Yun et al., 2021, thus created an embedding vector with contextualized words. A word embedding vector can be constructed with sparse linear representations of word factors. For example, depending on the context of the sentences in a dataset, `separate` can be represented as:

```
separate=0.3" keep apart"+"0.3" distinct"+ 0.1 "discriminate"+0.1 "sever" +
0.1 "disperse"+0.1 "scatter"
```

To ensure that a linear representation remains sparse, we don't add 0 factors that would create huge matrices with 0 values. Thus, we do not include useless information such as:

```
separate= 0.0"putting together"+".0" "identical"
```

The whole point is to keep the representation sparse by forcing the coefficients of the factors to be greater than 0.

The hidden state for each word is retrieved for each layer. Since each layer progresses in its understanding of the representation of the word in the dataset of sentences, the latent dependencies build up. This sparse linear superposition of transformer factors becomes a dictionary matrix with a sparse vector of coefficients to be inferred that we can sum up as:

$$\varphi R^{dxm}\alpha$$

In which:

- φ (phi) is the dictionary matrix
- α is the sparse vector of coefficients to be inferred

Yun et al., 2021, added, ε, Gaussian noise samples to force the algorithm to search for deeper representations.

Also, to ensure that the representation remains sparse, the equation must be written s.t. (such that) $\alpha > 0$.

The authors refer to *X* as the set of hidden states of the layers and *x* as a sparse linear superposition of transformer factors that belongs to *X*.

They beautifully sum up their sparse dictionary learning model as:

$$X \ = \varphi\alpha + \varepsilon\ s\ t\ \alpha > 0$$

In the dictionary matrix, φ:,c refers to a column of the dictionary matrix and contains a transformer factor.

φ:,c is divided into three levels:

- **Low-level** transformer factors to solve polysemy problems through word-level disambiguation
- **Mid-level** transformer factors take us further into sentence-level patterns that will bring vital context to the low level
- **High-level** transformer patterns that will help understand long-range dependencies

The method is innovative, exciting, and seems efficient. However, there is no visualization functionality at this point. Therefore, *Yun* et al., 2021, created the necessary information for LIME, a standard interpretable AI method to visualize their findings.

The interactive transformer visualization page is thus based on LIME for its outputs. The following section is a brief introduction to LIME.

Introducing LIME

LIME stands for **Local Interpretable Model-Agnostic Explanations**. The name of this explainable AI method speaks for itself. It is *model-agnostic*. Thus, we can draw immediate consequences about the method of transformer visualization via dictionary learning:

- This method does not dig into the matrices, weights, and matrix multiplications of transformer layers.
- The method does not explain how a transformer model works, as we did in *Chapter 2, Getting Started with the Architecture of the Transformer Model*.
- In this chapter, the method peeks into the mathematical outputs provided by the sparse linear superpositions of transformer factors.

LIME does not try to parse all of the information in a dataset. Instead, LIME finds out whether a model is *locally reliable* by examining the features around a prediction.

LIME does not apply to the model globally. It focuses on the local environment of a prediction.

This is particularly efficient when dealing with NLP because LIME explores the context of a word, providing invaluable information on the model's output.

In visualization via dictionary learning, an instance *x* can be represented as:

$$x \in \mathrm{R}^{d}$$

The interpretable representation of this instance is a binary vector:

$$x' \in \{0,1\}^{d'}$$

The goal is to determine the local presence or absence of a feature or several features. In NLP, the features are tokens that can be reconstructed into words.

For LIME, g represents a transformer model or any other machine learning model. G represents a set of transformer models containing g, among other models:

$$g \in G$$

LIME's algorithm can thus be applied to any transformer model.

At this point, we know that:

- LIME targets a word and searches the local context for other words
- LIME thus provides the local context of a word to explain why that word was predicted and not another one

Exploring explainable AI such as LIME is not in the scope of this book on transformers for NLP. However, for more on LIME, see the *References* section.

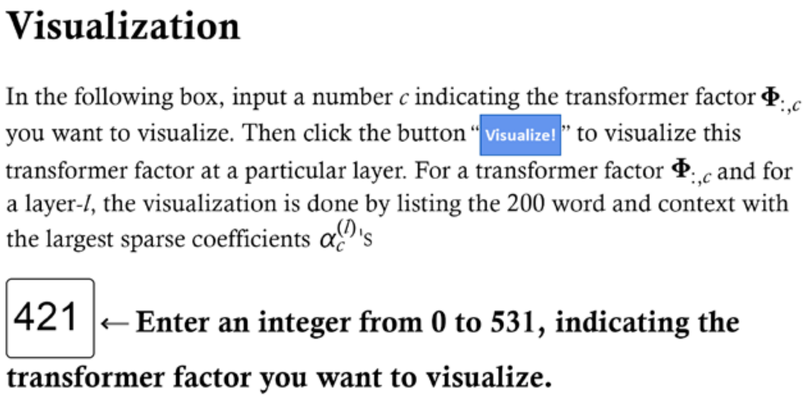
Let's now see how LIME fits in the method of transformer visualization via dictionary learning.

Let's now explore the visualization interface.

The visualization interface

Visit the following site to access the interactive transformer visualization page: `https://transformervis.github.io/transformervis/`.

The visualization interface provides intuitive instructions to start analyzing a transformer factor of a specific layer in one click, as shown in *Figure 14.12*:



Visualization

In the following box, input a number c indicating the transformer factor $\Phi_{:,c}$ you want to visualize. Then click the button "Visualize!" to visualize this transformer factor at a particular layer. For a transformer factor $\Phi_{:,c}$ and for a layer-l, the visualization is done by listing the 200 word and context with the largest sparse coefficients $\alpha_c^{(l)}$'s

421 ← Enter an integer from 0 to 531, indicating the transformer factor you want to visualize.

Figure 14.12: Selecting a transformer factor

Once you have chosen a factor, you can click on the layer you wish to visualize for this factor:

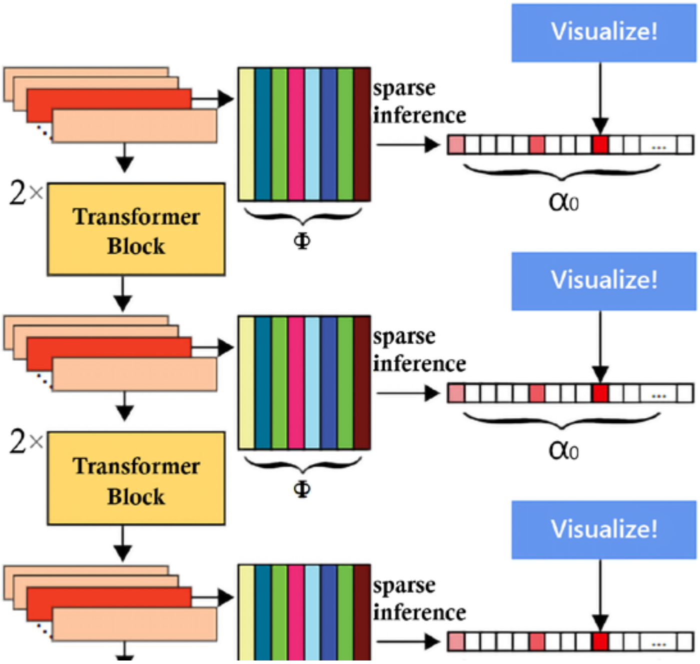


Figure 14.13: Visualize function per layer

The first visualization shows the activation of the factor layer by layer:

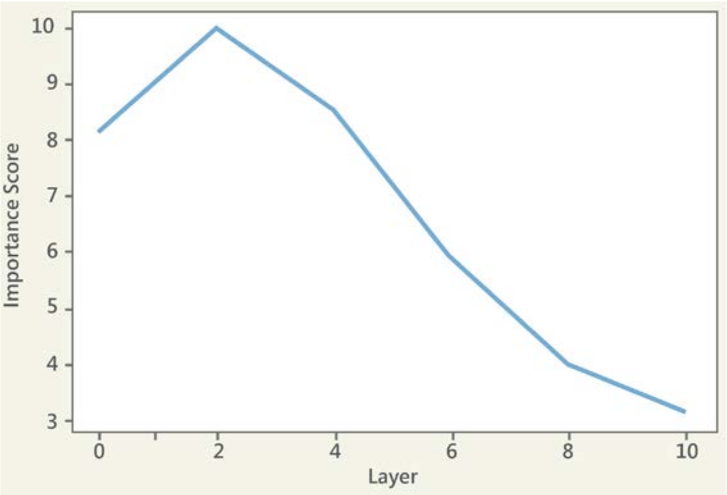
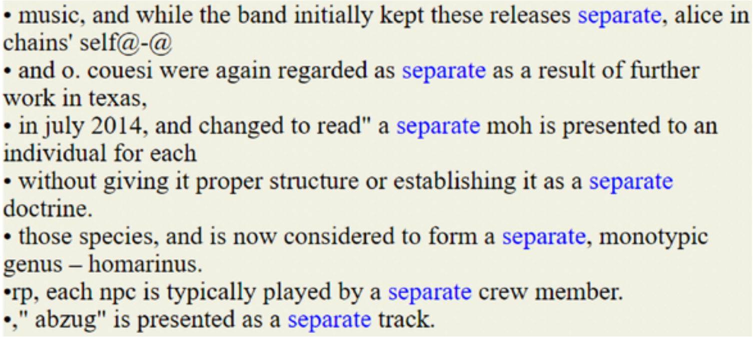


Figure 14.14: Importance of a factor for each layer

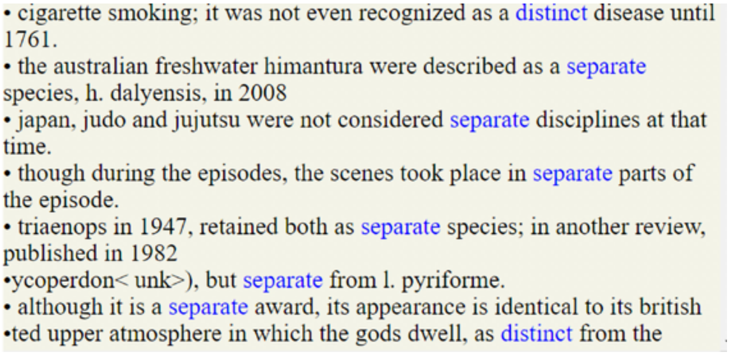
Factor `421` focuses on the lexical field of `separate`, as shown in the lower layers:



• music, and while the band initially kept these releases separate, alice in chains' self@-@
• and o. couesi were again regarded as separate as a result of further work in texas,
• in july 2014, and changed to read" a separate moh is presented to an individual for each
• without giving it proper structure or establishing it as a separate doctrine.
• those species, and is now considered to form a separate, monotypic genus – homarinus.
•rp, each npc is typically played by a separate crew member.
•," abzug" is presented as a separate track.

Figure 14.15: The representation of "separate" in the lower layers

As we visualize higher layers, longer-range representations emerge. Factor `421` began with the representation of `separate`. But at higher levels, the transformer began to form a deeper understanding of the factor and associated `separate` with `distinct`, as shown in *Figure 14.16*:



• cigarette smoking; it was not even recognized as a distinct disease until 1761.
• the australian freshwater himantura were described as a separate species, h. dalyensis, in 2008
• japan, judo and jujutsu were not considered separate disciplines at that time.
• though during the episodes, the scenes took place in separate parts of the episode.
• triaenops in 1947, retained both as separate species; in another review, published in 1982
•ycoperdon< unk>), but separate from l. pyriforme.
• although it is a separate award, its appearance is identical to its british
•ted upper atmosphere in which the gods dwell, as distinct from the

Figure 14.16: The higher-layer representations of a transformer factor

Try several transformer factors to visualize how transformers expand their perception and understanding of language, layer by layer.

You will find many good examples and also poor results. Focus on the good examples to understand how a transformer makes its way through language learning. Use the poor results to understand why it made a mistake. Also, the transformer model used for the visualization interface is not the most powerful or well-trained one.

In any case, get involved and stay in the loop of this ever-evolving field!

For example, you can explore `Understanding_GPT_2_models_with_Ecco.ipynb`, which is in the GitHub repository of this book for this chapter. It shows you how transformers generate candidates before choosing a token. It is self-explanatory.

In this section, we saw how transformers learn the meaning of words layer by layer. A transformer generates candidates before making a choice. As shown in the notebook, a transformer model is stochastic and as such chooses among several top probabilities. Consider the following sentence:

```
"The sun rises in the_____."
```

What word would you choose at the end of the sentence? We all hesitate. So do transformers!

In this case, the GPT-2 model chooses the word `sky`:

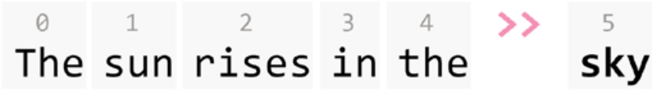


Figure 14.17: Completing a sequence

But there are other candidates the GPT-2 model may choose in another run, as shown in *Figure 14.18*:

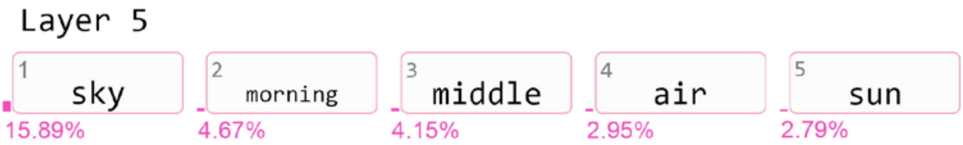


Figure 14.18: The other candidates for the completion

We can see that `sky` appears in rank first. However, `morning` appears in rank second and could fit as well. If we run the model several times, we may obtain different outputs because the model is stochastic.

It seems that the domain of AI and transformers is complete.

However, let's see why humans still have a lot of work to do before we go.

Exploring models we cannot access

The visual interfaces explored in this chapter are fascinating. However, there is still a lot of work to do!

For example, OpenAI's GPT-3 model runs online or through an API. Thus, we cannot access the weights of some **Software as a Service** (**SaaS**) transformer models. This trend will increase and expand in the years to come. Corporations that spend millions of dollars on research and computer power will tend to provide pay-as-you-go services, not open-source applications.

Even if we had access to the source code or output weights of a GPT-3 model, using a visual interface to analyze the 9,216 attention heads (96 layers x 96 heads) would be quite challenging.

Finding what is wrong will still require some human involvement in many cases.

For example, the polysemy issue of the word `coach` in English to French translation often represents a problem. In English, a coach can be a person who trains people, or a bus. The word `coach` exists in French, but it only applies to a person who trains people.

If you go to OpenAI AI GPT-3 playground, `https://openai.com/`, and translate sentences containing the word `coach`, you might obtain mixed results.

Sentence 1 is translated correctly by the OpenAI engine:

```
English: The coach broke down, and everybody complained.
French: Le bus a eu un problème et tout le monde s'est plaint.
```

`coach` is translated as a bus, which is fine. More context would be required.

The outputs are stochastic, so the translation might be correct one time and false the time after.

However, Sentence 2 is mistranslated:

```
English: The coach was dissatisfied with the team and everybody
complained.
French: Le bus était insatisfait du équipe et tout le monde s'est plaint.
```

This time, the GPT-3 engine missed the fact that `coach` meant a person, not a bus. The same stochastic runs will provide unstable outputs.

If we modify sentence 2 by adding context, we will obtain the proper translation:

```
English: The coach of the football team was dissatisfied and everybody
complained.
```

```
French: Le coach de l'équipe de football était insatisfait et tout le
monde s'est plaint.
```

The translation now contains the French word `coach` for the same definition of the English word `coach` in this sentence. More context was added.

OpenAI's solutions, AI in general, and transformer models in particular, are continuously progressing. Furthermore, most Industry 4.0 AI-driven micro-decisions do not require the level of sophistical of NLP or translation tasks and are effective.

However, human intervention and development at the Cloud AI API level will still remain necessary for quite a long time!

Summary

Transformer models are trained to resolve word-level polysemy disambiguation

low-level, mid-level, and high-level dependencies. The process is achieved by connecting training million- to trillion-parameter models. The task of interpreting these giant models seems daunting. However, several tools are emerging.

We first installed `BertViz`. We learned how to interpret the computations of the attention heads with an interactive interface. We saw how words interacted with other words for each layer.

The chapter continued by defining the scope of probing and non-probing tasks. Probing tasks such as NER provide insights into how a transformer model represents language. However, non-probing methods analyze how the model makes predictions. For example, LIT plugged a PCA project and UMAP representations into the outputs of a BERT transformer model. We could then analyze clusters of outputs to see how they fit together.

Finally, we ran transformer visualization via dictionary learning. A user can choose a transformer factor to analyze and visualize the evolution of its representation from the lower layers to the higher layers of the transformer. The factor will progressively go from polysemy disambiguation to sentence context analysis and finally to long-term dependencies.

The tools of this chapter will evolve along with other techniques. However, the key takeaway of this chapter is that transformer model activity can be visualized and interpreted in a user-friendly manner. In the next chapter, we will discover new transformer models. We will also go through risk management methods to choose the best implementations for a transformer model project.

Questions

1. BertViz only shows the output of the last layer of the BERT model. (True/False)
2. BertViz shows the attention heads of each layer of a BERT model. (True/False)
3. BertViz shows how the tokens relate to each other. (True/False)
4. LIT shows the inner workings of the attention heads like BertViz. (True/False)
5. Probing is a way for an algorithm to predict language representations. (True/False)
6. NER is a probing task. (True/False)
7. PCA and UMAP are non-probing tasks. (True/False)
8. LIME is model agnostic. (True/False)
9. Transformers deepen the relationships of the tokens layer by layer. (True/False)
10. Visual transformer model interpretation adds a new dimension to interpretable AI. (True/False)

References

- BertViz:*Jesse Vig,2019, A Multiscale Visualization of Attention in the Transformer Model,2019,* `https://aclanthology.org/P19-3007.pdf`
- BertViz: `https://github.com/jessevig/BertViz`
- LIT, explanation of sentiment analysis representations: `https://pair-code.github.io/lit/tutorials/sentiment/`
- LIT: `https://pair-code.github.io/lit/`
- *Transformer visualization via dictionary learning*: *Zeyu Yun*, *Yubei Chen*, *Bruno A Olshausen*, *Yann LeCun*, 2021, Transformer visualization via dictionary learning: contextualized embedding as a linear superposition of transformer factors, `https://arxiv.org/abs/2103.15949`
- Transformer visualization via dictionary learning: `https://transformervis.github.io/transformervis/`

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the authors:

`https://www.packt.link/Transformers`

15

From NLP to Task-Agnostic Transformer Models

Up to now, we have examined variations of the original Transformer model with encoder and decoder layers, and we explored other models with encoder-only or decoder-only stacks of layers. Also, the size of the layers and parameters has increased. However, the fundamental architecture of the transformer retains its original structure with identical layers and the parallelization of the computing of the attention heads.

In this chapter, we will explore innovative transformer models that respect the basic structure of the original Transformer but make some significant changes. Scores of transformer models will appear, like the many possibilities a box of LEGO© pieces gives. You can assemble those pieces in hundreds of ways! Transformer model sublayers and layers are the LEGO© pieces of advanced AI.

We will begin by asking which transformer model to choose among the many offers and the ecosystem we will implement them in.

Then we will discover **Locality Sensitivity Hashing** (**LSH**) buckets and chunking in Reformer models. We will then learn what disentanglement is in DeBERTa models. DeBERTa also introduces an alternative way of managing positions in the decoder. DeBERTA's high-powered transformer model exceeds human baselines.

Our last step stops will be to discover powerful computer vision transformers such as Vit, CLIP, and DALL-E. We can add CLIP and DALL-E to OpenAI GPT-3 and Google BERT (trained by Google) to the very small group of **foundation models**.

These powerful foundation models prove that transformers are *task-agnostic*. A transformer learns sequences. These sequences include vision, sound, and any type of data represented as a sequence.

Images contain sequences of data-like language. We will run ViT, CLIP, and DALL-E models to learn. We will take vision models to innovative levels.

By the end of the chapter, you will see that the world of *task-agnostic* transformers has evolved into a universe of imagination and creativity.

This chapter covers the following topics:

- Choosing a transformer model
- The Reformer transformer model
- **Locality Sensitivity Hashing (LSH)**
- Bucket and chunking techniques
- The DeBERTA transformer model
- Disentangled attention
- Absolute positions
- Text-image vision transformers with CLIP
- DALL-E, a creative text-image vision transformer

Our first step will be to see how to choose a model and an ecosystem.

Choosing a model and an ecosystem

We thought that testing transformer models by downloading them would require machine and human resources. Also, you might have thought that if a platform doesn't have an online sandbox by this time, it will be a risk to go further because of the work to test a few examples.

However, sites such as Hugging Face download pretrained models automatically in real time, as we will see in *The Reformer* and *DeBERTa* sections! So, what should we do? Thanks to that, we can run Hugging Face models in Google Colab without installing anything on the machine ourselves. We can also test Hugging Face models online.

The idea is to analyze without having anything to "install." "Nothing to Install" in 2022 can mean:

- Running a transformer task online
- Running a transformer on a preinstalled Google Colaboratory VM that seamlessly downloads a pretrained model for a task, which we can run in a few lines
- Running a transformer through an API

The definition of "install" has expanded over the past few years. The definition of "online" has widened. We can consider using a few lines of code to run an API as a meta-online test.

We will refer to "without installing," and "online" in a broad sense in this section. *Figure 15.1* shows how we should test models "online":

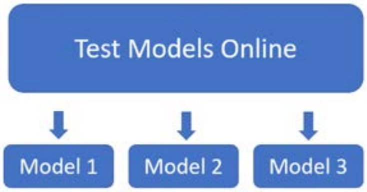


Figure 15.1: Testing transformer models online

Testing in this decade has become flexible and productive, as the following shows:

- Hugging Face hosts API models such as DeBERTa and some other models. In addition, Hugging Face offers an AutoML service to train and deploy transformer models in their ecosystem.
- OpenAI's GPT-3 engine runs on the online playground and provides an API. OpenAI offers models that cover many NLP tasks. The models require no training. GPT-3's billion-parameter zero-shot engine is impressive. It shows that transformer models with many parameters produce better results overall. Microsoft Azure, Google Cloud AI, AllenNLP, and other platforms offer interesting services.
- An online model analysis can be done by reading a paper if it is worthwhile. A good example is Google's publication by *Fedus* et al., (2021), on *Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity*. Google increased the size of the T5-based models we studied in *Chapter 8*, *Applying Transformers to Legal and Financial Documents for AI Text Summarization*. This paper confirms the strategy of large online models such as GTP-3.

However, in the end, *you are the one taking the risk* of choosing one solution over another. The time you spend on exploring platforms and models will help you optimize the implementation of your project once you have made your choice.

You can host your choice in three different ways, as shown in *Figure 15.2:*

- On a local machine using an API. OpenAI, Google Cloud AI, Microsoft Azure AI, Hugging Face, and others provide good APIs. An application can be on a local machine and not on a cloud platform but can go through a cloud service with an API.

- On a cloud platform such as **Amazon Web Services** (**AWS**) or Google Cloud. You can train, fine-tune, test, and run the models on these platforms. In this case, there is no application on a local machine. Everything is on the cloud.
- From anywhere using an API! On a local machine, a data center VM, or from anywhere. This means that the API would be integrated in a physical system such as a windmill, an airplane, a rocket, or an autonomous vehicle. The system could thus permanently connect with another system through an API.

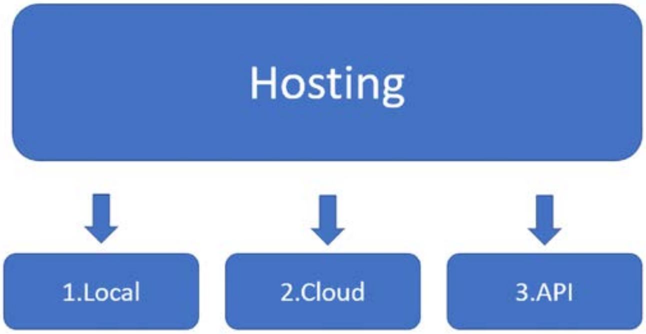


Figure 15.2: Implementing options for your models

In the end, it is up to you to make the decision. Take your time. Test, analyze, compute the costs, and work as a team to listen to different perspectives. *The more you understand how transformers work, the better the choices you make will be*.

Let's now explore the Reformer, a variation of the original Transformer model.

The Reformer

Kitaev et al. (2020) designed the Reformer to solve the attention and memory issues, adding functionality to the original Transformer model.

The Reformer first solves the attention issue with **Locality Sensitivity Hashing** (**LSH**) buckets and chunking.

LSH searches for nearest neighbors in datasets. The hash function determines that if datapoint q is close to p, then $hash(q) == hash(p)$. In this case, the data points are the keys of the transformer model's heads.

The LSH function converts the keys into LSH *buckets* (*B1* to *B4* in *Figure 15.3*) in a process called LSH bucketing, just like how we would take objects similar to each other and put them in the same sorted buckets.

The sorted buckets are split into *chunks* (*C1* to *C4* in *Figure 15.3*) to parallelize. Finally, attention will only be applied within the same bucket in its chunk and the previous chunk:

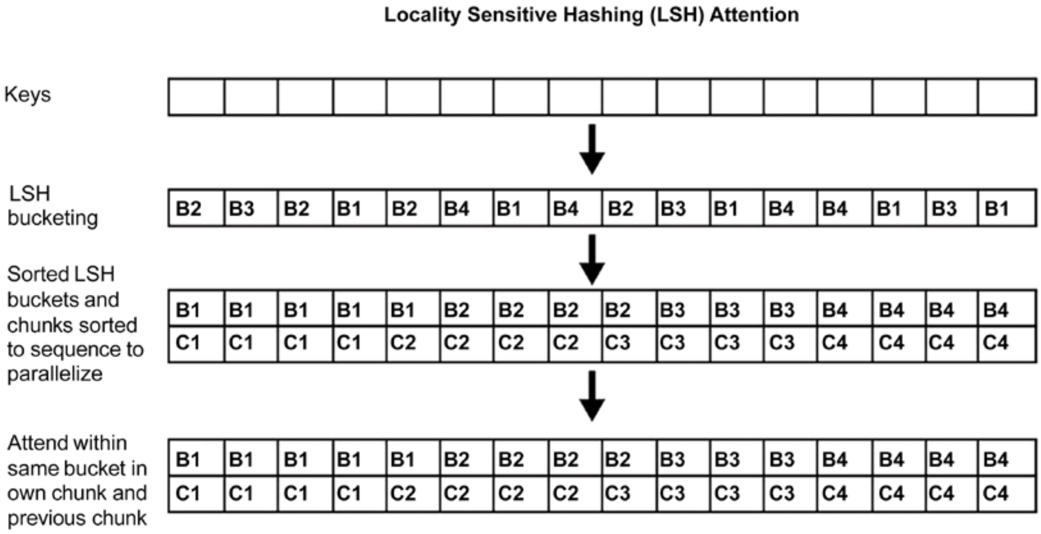

Locality Sensitive Hashing (LSH) Attention
Keys
LSH bucketing
B2 B3 B2 B1 B2 B4 B1 B4 B2 B3 B1 B4 B4 B1 B3 B1
Sorted LSH buckets and chunks sorted to sequence to parallelize
B1 B1 B1 B1 B1 B2 B2 B2 B2 B3 B3 B3 B4 B4 B4 B4
C1 C1 C1 C1 C2 C2 C2 C2 C3 C3 C3 C3 C4 C4 C4 C4
Attend within same bucket in own chunk and previous chunk
B1 B1 B1 B1 B1 B2 B2 B2 B2 B3 B3 B3 B4 B4 B4 B4
C1 C1 C1 C1 C2 C2 C2 C2 C3 C3 C3 C3 C4 C4 C4 C4

Figure 15.3: LSH attention heads

LSH bucketing and chunking considerably reduce the complexity from $O(L^2)$, attending to all the word pairs, to $O(L\log L)$, only attending to the content of each bucket.

The Reformer also solves the memory issue of recomputing each layer's input instead of storing the information for multi-layer models. The recomputing is achieved on-demand instead of consuming terabytes of memory for some large multi-layer models.

We will now use a Reformer model trained on the English translation of *Crime and Punishment* by *Fyodor Dostoevsky*.

Running an example

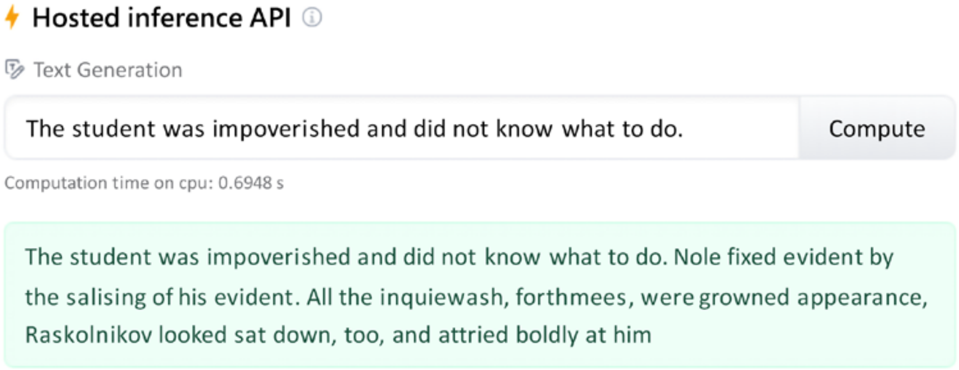
Let's run it directly online with the hosted inference API. The input sentence is:

```
The student was impoverished and did not know what to do.
```

The link to the online interface contains the input:

```
https://huggingface.co/google/reformer-crime-and-punishment?text=The+student+was
+impoverished+and+did+not+know+what+to+do
```

The hosted inference API appears with the input sentence. Click on compute to obtain an inference, and the result will appear right under the input:



The student was impoverished and did not know what to do.

Computation time on cpu: 0.6948 s

The student was impoverished and did not know what to do. Nole fixed evident by the salising of his evident. All the inquiewash, forthmees, were growned appearance, Raskolnikov looked sat down, too, and attried boldly at him

Figure 15.4: The Reformer's hosted inference API

You might get a different response since the algorithm is stochastic. The Reformer was reasonably trained, though not with a supercomputer with billions of bits of information like OpenAI's GPT-3. The result of the Reformer is not very impressive. It would take more training and fine-tuning to obtain better results.

OpenAI's GPT-3 engine produces the following result for text completion:

The student was impoverished and did not know what to do. He did not have any one to turn to and could not find a place to stay. He took out a pad from his bag and started writing. He wrote:

```
"My name is XXXXXXXXXX. I am a student at XXXXXXXXXX. I have no family, no friends,
no money."
```

The result is more convincing. You can access OpenAI's playground after having signed up: `https://openai.com/`

> Note: OpenAI GPT-3, as with other transformer models and most deep learning models, is based on stochastic algorithms. The results might vary from one to another.

This shows that a highly well-trained transformer model containing billions of parameters can outperform an innovative transformer model architecture.

Will supercomputer-driven cloud AI platforms progressively outperform local attempts or even less powerful cloud platforms? You need to address these issues through prototypes before investing in one solution over another.

Note: The stochastic nature of transformer models may produce different results when running them. Also, online platforms continually change their interfaces. We need to accept that and adapt.

DeBERTa introduces another innovative architecture, which we will now explore.

DeBERTa

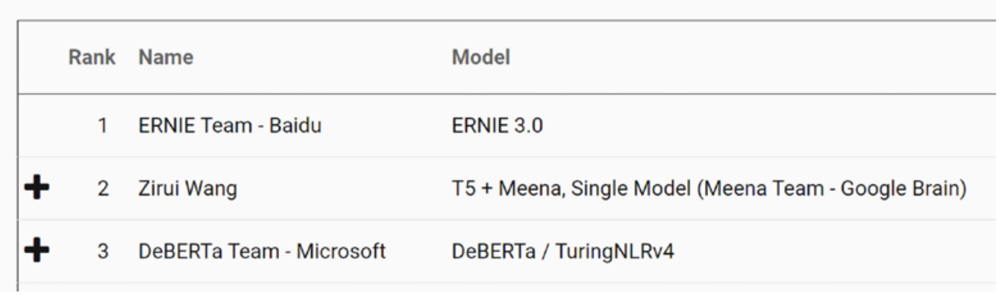
Another new approach to transformers can be found through *disentanglement*. *Disentanglement* in AI allows you to separate the representation features to make the training process more flexible. *Pengcheng He*, *Xiaodong Liu*, *Jianfeng Gao*, and *Weizhu Chen* designed DeBERTa, a disentangled version of a transformer, and described the model in an interesting article: *DeBERTa: Decoding-enhanced BERT with Disentangled Attention*: `https://arxiv.org/abs/2006.03654`

The two main ideas implemented in DeBERTa are:

- Disentangle the content and position in the transformer model to train the two vectors separately
- Use an absolute position in the decoder to predict masked tokens in the pretraining process

The authors provide the code on GitHub: `https://github.com/microsoft/DeBERTa`

DeBERTa exceeds the human baseline on the SuperGLUE leaderboard:



	Rank	Name	Model
	1	ERNIE Team - Baidu	ERNIE 3.0
+	2	Zirui Wang	T5 + Meena, Single Model (Meena Team - Google Brain)
+	3	DeBERTa Team - Microsoft	DeBERTa / TuringNLRv4

Figure 15.5: DeBERTa on the SuperGLUE leaderboard

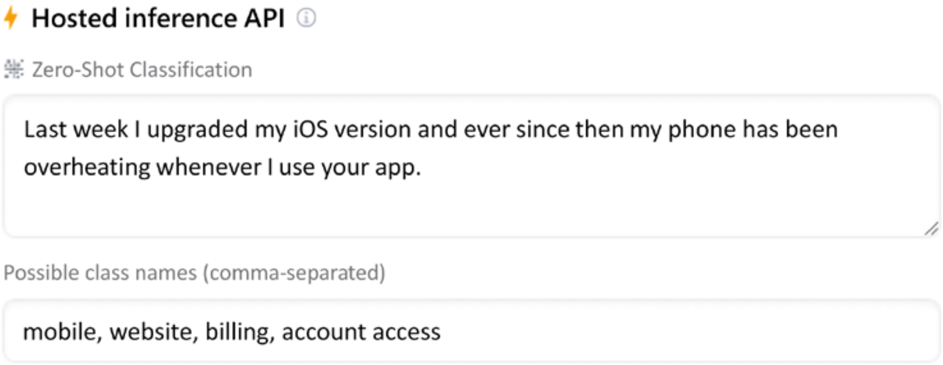
Remove any space before Let's run an example on Hugging Face's cloud platform.

Running an example

To run an example on Hugging Face's cloud platform, click on the following link:

`https://huggingface.co/cross-encoder/nli-deberta-base`

The hosted inference API will appear with an example and output of possible class names:



Last week I upgraded my iOS version and ever since then my phone has been overheating whenever I use your app.

Possible class names (comma-separated)

mobile, website, billing, account access

Figure 15.6: DeBERTa's hosted inference API

The possible class names are `mobile`, `website`, `billing`, and `account access`.

The result is interesting. Let's compare it to a GPT-3 keyword task. First, sign up on `https://openai.com/`

Enter `Text` as the input and `Keywords` to ask the engine to find keywords:

Text: `Last week I upgraded my iOS version and ever since then my phone has been overheating whenever I use your app.`

Keywords: `app, overheating, phone`

The possible keywords are `app`, `overheating`, and `phone`.

We have gone through the DeBERTa and GPT-3 transformers. We will now extend transformers to vision models.

From Task-Agnostic Models to Vision Transformers

Foundation models, as we saw in *Chapter 1*, *What Are Transformers?*, have two distinct and unique properties:

- **Emergence** – Transformer models that qualify as foundation models can perform tasks they were not trained for. They are large models trained on supercomputers. They are not trained to learn specific tasks like many other models. Foundation models learn how to understand sequences.

- **Homogenization** – The same model can be used across many domains with the same fundamental architecture. Foundation models can learn new skills through data faster and better than any other model.

GPT-3 and Google BERT (only the BERT models trained by Google) are task-agnostic foundation models. These task-agnostic models lead directly to ViT, CLIP, and DALL-E models. Transformers have uncanny sequence analysis abilities.

The level of abstraction of transformer models leads to multi-modal neurons:

- **Multi-modal neurons** can process images that can be tokenized as pixels or image patches. Then they can be processed as *words* in vision transformers. Once an image has been encoded, transformer models see the tokens as any *word* token, as shown in *Figure 15.7*:

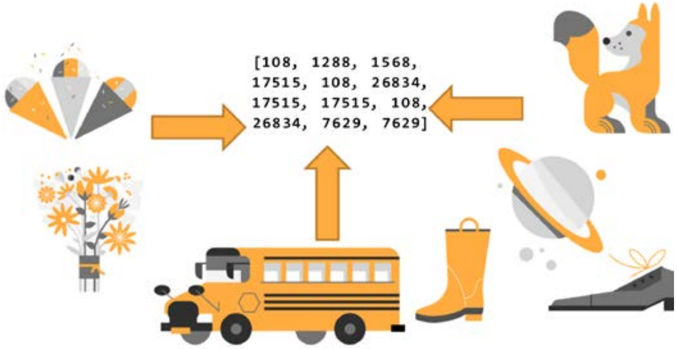


Figure 15.7: Images can be encoded into word-like tokens

In this section, we will go through:

- ViT, vision transformers that process images as patches of *words*
- CLIP, vision transformers that encode text and image
- DALL-E, vision transformers that construct images with text

Let's begin by exploring ViT, a vision transformer that processes images as patches of *words*.

ViT – Vision Transformers

Dosovitskiy et al. (2021) summed up the essence of the vision transformer architecture they designed in the title of their paper: *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*.

An image can be converted into patches of 16x16 words.

Let's first see the architecture of a ViT before looking into the code.

The Basic Architecture of ViT

A vision transformer can process an image as patches of *words*. In this section, we will go through the process in three steps:

1. Splitting the image into patches
2. A linear projection of the patches
3. The hybrid input embedding sublayer

The first step is to SPLIT the image into equal-sized patches.

Step 1: Splitting the image into patches

The image is split into *n* patches, as shown in *Figure 15.8*. There is no rule saying how many patches as long as all the patches have the same dimensions, such as 16x16:

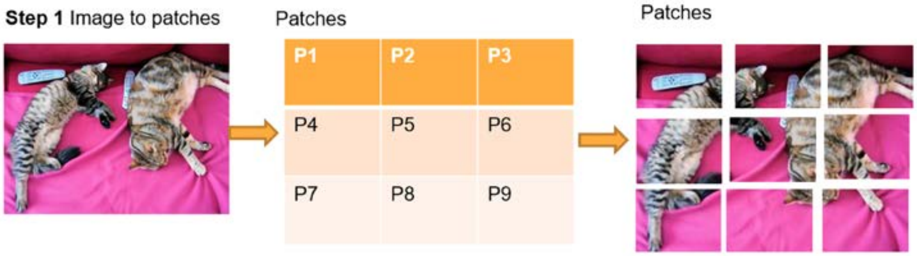


Figure 15.8: Splitting an image into patches

The patches of equal dimensions now represent the *words* of our sequence. The problem of what to do with these patches remains. We will see that each type of vision transformer has its own method.

Image citation: The image of cats used in this section and the subsequent sections was taken by *DocChewbacca*: `https://www.flickr.com/photos/st3f4n/`, in 2006. It is under a Flickr free license, `https://creativecommons.org/licenses/by-sa/2.0/`. For more details, see *DocChewbacca*'s image on Flickr: `https://www.flickr.com/photos/st3f4n/210383891`

In this case, for the ViT, *Step 2* will be to make a linear projection of flattened images.

Step 2: Linear projection of flattened images

Step 1 converted an image to equal-sized patches. The motivation of the patches is to avoid processing the image pixel by pixel. However, the problem remains to find a way to process the patches.

The team at Google Research decided to design a linear projection of flattened images with the patches obtained by splitting the image, as shown in *Figure 15.9*:

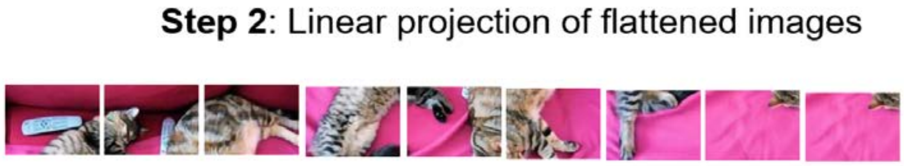


Figure 15.9: Linear projection of flattened images

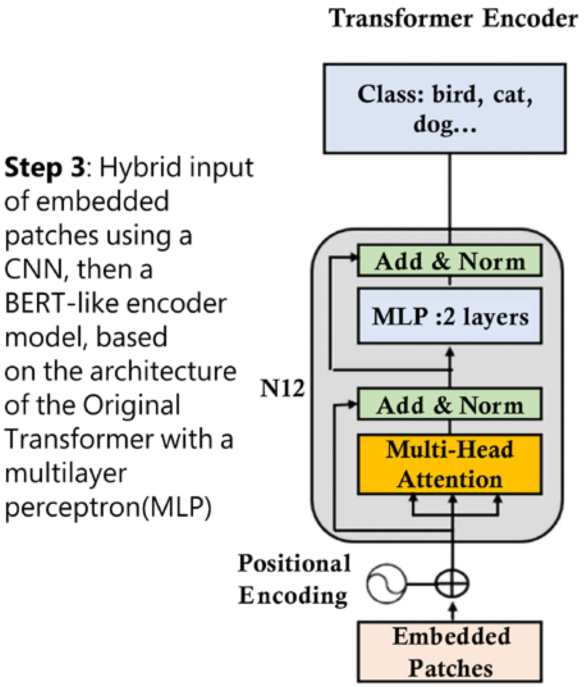
The idea is to obtain a sequence of work-like patches. The remaining problem is to embed the sequence of flattened images.

Step 3: The hybrid input embedding sublayer

Word-like image sequences can fit into a transformer. The problem is that they still are images!

Google Research decided that a hybrid input model would do the job, as shown in *Figure 15.10*:

- Add a convolutional network to embed the linear projection of the patches
- Add positional encoding to retain the structure of the original image
- Then process the embedded input with a standard original BERT-like encoder



Step 3: Hybrid input of embedded patches using a CNN, then a BERT-like encoder model, based on the architecture of the Original Transformer with a multilayer perceptron(MLP)

Figure 15.10: A hybrid input sublayer and a standard encoder

Google Research found a clever way to convert an NLP transformer model into a vision transformer.

Now, let's implement a Hugging Face example of a vision transformer in code.

Vision transformers in code

In this section, we will focus on the main areas of code that relate to the specific architecture of vision transformers.

Open `Vision_Transformers.ipynb`, which is in the GitHub repository for this chapter.

Google Colab VM's contain many pre-installed packages such as `torch` and `torchvision`. You can display them by uncommenting the command in the first cell of the notebook:

```
#Uncomment the following command to display the list of pre-installed
modules
#!pip list -v
```

Then go to the **Vision Transformer** (**ViT**) cell of the notebook. The notebook first installs Hugging Face transformers and imports the necessary modules:

```
!pip install transformers
from transformers import ViTFeatureExtractor, ViTForImageClassification
from PIL import Image
import requests
```

Note: At the time of writing this book, Hugging Face warns us that the code can be unstable due to constant evolutions. This should not stop us from exploring ViT models. Testing new territory is what the cutting edge is all about!

We then download an image from the COCO dataset. You can find a comprehensive corpus of datasets on their website if you wish to experiment further: `https://cocodataset.org/`

Let's download from the VAL2017 dataset. Follow the COCO dataset website instructions to obtain these images through programs or download the datasets locally.

The VAL2017 contains 5,000 images we can choose from to test this ViT model. You can run any of the 5,000 images.

Let's test the notebook with the image of the cats. We first retrieve the image of the cats through their URL:

```
url = 'http://images.cocodataset.org/val2017/000000039769.jpg'
image = Image.open(requests.get(url, stream=True).raw)
```

We next download Google's feature extractor and classification model:

```
feature_extractor = ViTFeatureExtractor.from_pretrained('google/vit-base-
patch16-224')
model = ViTForImageClassification.from_pretrained('google/vit-base-
patch16-224')
```

The model was trained on 224 x 244 resolution images but was presented with 16 x 16 patches for feature extraction and classification. The notebook runs the model and makes a prediction:

```
inputs = feature_extractor(images=image, return_tensors="pt")
outputs = model(**inputs)
logits = outputs.logits
```

```
# model predicts one of the 1000 ImageNet classes
predicted_class_idx = logits.argmax(-1).item()
print("Predicted class:",predicted_class_idx,": ", model.config.
id2label[predicted_class_idx])
```

The output is:

```
Predicted class: 285 :  Egyptian cat
```

Explore the code that follows the prediction, which gives us information at a low level, among which are:

- `model.config.id2label`, which will list the labels of the classes. The 1000 label classes explain why we obtain a class and not a detailed text description:

  ```
  {0: 'tench, Tinca tinca',1: 'goldfish, Carassius auratus', 2: 'great
  white shark, white shark, man-eater, man-eating shark, Carcharodon
  carcharias',3: 'tiger shark, Galeocerdo cuvieri',...,999: 'toilet
  tissue, toilet paper, bathroom tissue'}
  ```

- `model`, which will display the architecture of the model that begins with the hybrid usage of a convolutional input sublayer:

  ```
  (embeddings): ViTEmbeddings(
    (patch_embeddings): PatchEmbeddings(
      (projection): Conv2d(3, 768, kernel_size=(16, 16), stride=(16,
  16))
    )
  ```

After the convolutional input embedding sublayer, the model is a BERT-like encoder.

Take your time to explore this innovative move from NLP transformers to transformers for images, leading to transformers for everything quite rapidly.

Now, let's go through CLIP, another computer vision model.

CLIP

Contrastive Language-Image Pre-Training (CLIP) follows the philosophy of transformers. It plugs sequences of data in its transformer-type layers. Instead of sending text pairs, this time, the model sends text-image pairs. Once the data is tokenized, encoded, and embedded, CLIP, a task-agnostic model, learns text-image pairs as with any other sequence of data.

The method is contrastive because it looks for the contrasts in the features of the image. It is the method we use in some magazine games in which we have to find the differences, the contrasts, between two images.

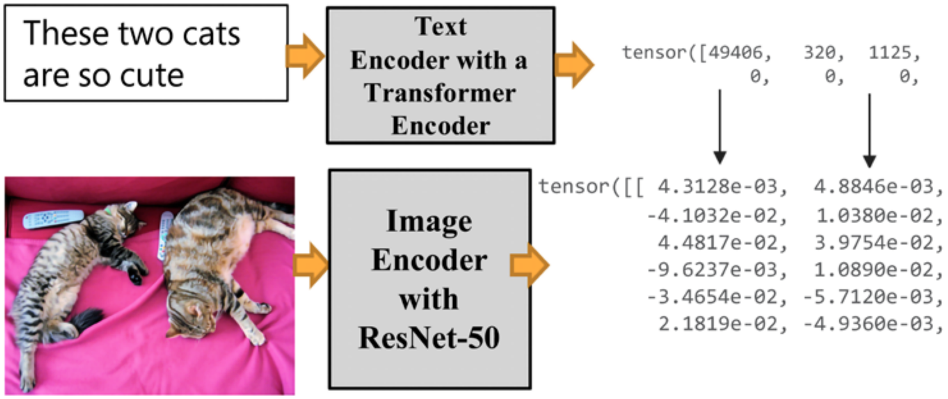
Let's first see the architecture of CLIP before looking into the code.

The Basic Architecture of CLIP

Contrastive: the images are trained to learn how they fit together through their differences and similarities. The image and captions find their way toward each other through (joint text, image) pretraining. After pretraining, CLIP learns new tasks.

CLIPs are transferable because they can learn new visual concepts, like GPT models, such as action recognition in video sequences. The captions lead to endless applications.

ViT splits images into word-like patches. CLIP jointly trains *text and image* encoders for (caption, image) pairs to maximize cosine similarity, as shown in *Figure 15.11*:


These two cats are so cute
Text Encoder with a Transformer Encoder
Image Encoder with ResNet-50

```
tensor([49406,   320,  1125,
            0,     0,     0,
```

```
tensor([[ 4.3128e-03,  4.8846e-03,
         -4.1032e-02,  1.0380e-02,
          4.4817e-02,  3.9754e-02,
         -9.6237e-03,  1.0890e-02,
         -3.4654e-02, -5.7120e-03,
          2.1819e-02, -4.9360e-03,
```

Figure 15.11: Jointly training text and images

Figure 15.11 shows how the transformer will run a standard transformer encoder for the text input. It will run a ResNet 50-layer CNN for the images in a transformer structure. ResNet 50 was modified to run an average pooling layer in an attention pooling mechanism with a multi-head QKV attention head.

Let's see how CLIP learns text-image sequences to make predictions.

CLIP in code

Open `Vision_Transformers.ipynb`, which is in the repository for this chapter on GitHub. Then go to the `CLIP` cell of the notebook.

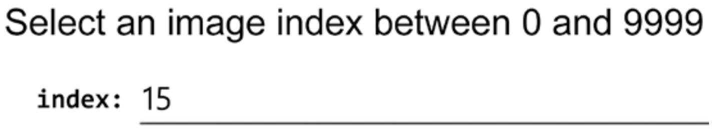
The program begins by installing PyTorch and CLIP:

```
!pip install ftfy regex tqdm
!pip install git+https://github.com/openai/CLIP.git
```

The program also imports the modules and CIFAR-100 to access the images:

```python
import os
import clip
import torch
from torchvision.datasets import CIFAR100
```

There are 10,000 images available with an index between 0 and 9,999. The next step is to select an image we want to run a prediction on:



Select an image index between 0 and 9999

index: 15

Figure 15.12: Selecting an image index

The program then loads the model on the device that is available (GPU or CPU):

```python
# Load the model
device = "cuda" if torch.cuda.is_available() else "cpu"
model, preprocess = clip.load('ViT-B/32', device)
```

The images are downloaded:

```python
# Download the dataset
cifar100 = CIFAR100(root=os.path.expanduser("~/.cache"), download=True,
train=False)
```

The input is prepared:

```python
# Prepare the inputs
image, class_id = cifar100[index]
image_input = preprocess(image).unsqueeze(0).to(device)
text_inputs = torch.cat([clip.tokenize(f"a photo of a {c}") for c in
cifar100.classes]).to(device)
```

Let's visualize the input we selected before running the prediction:

```
import matplotlib.pyplot as plt
from torchvision import transforms
plt.imshow(image)
```

The output shows that `index 15` is a lion:

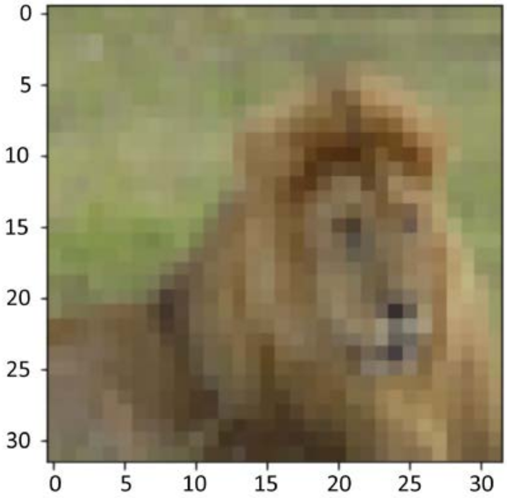


Figure 15.13: Image of Index 15

The images in this section are from *Learning Multiple Layers of Features from Tiny Images*, *Alex Krizhevsky*, 2009: `https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf`. They are part of the `CIFAR-10` and `CIFAR-100` datasets (`toronto.edu`): `https://www.cs.toronto.edu/~kriz/cifar.html`

We know this is a lion because we are humans. A transformer initially designed for NLP has to learn what an image is. We will now see how well it can recognize images.

The program shows that it is running a joint transformer model by separating the image input from the text input when calculating the features:

```
# Calculate features
with torch.no_grad():
    image_features = model.encode_image(image_input)
    text_features = model.encode_text(text_inputs)
```

Now CLIP makes a prediction and displays the top five predictions:

```
# Pick the top 5 most similar labels for the image
image_features /= image_features.norm(dim=-1, keepdim=True)
text_features /= text_features.norm(dim=-1, keepdim=True)
similarity = (100.0 * image_features @ text_features.T).softmax(dim=-1)
values, indices = similarity[0].topk(5)

# Print the result
print("\nTop predictions:\n")
for value, index in zip(values, indices):
    print(f"{cifar100.classes[index]:>16s}: {100 * value.item():.2f}%")
```

You can modify `topk(5)` if you want to obtain more or fewer predictions. The top five predictions are displayed:

```
Top predictions:

            lion: 96.34%
           tiger: 1.04%
           camel: 0.28%
      lawn_mower: 0.26%
         leopard: 0.26%
```

CLIP found the lion, which shows the flexibility of transformer architectures.

The next cell displays the classes:

```
cifar100.classes
```

You can go through the classes to see that with only one label per class, which is restrictive, CLIP did a good job:

```
[...,'kangaroo','keyboard','lamp','lawn_mower','leopard','lion',
 'lizard', ...]
```

The notebook contains several other cells describing the architecture and configuration of CLIP that you can explore.

The `model` cell is particularly interesting because you can see the visual encoder that begins with a convolutional embedding like for the ViT model and then continues as a “standard” size-768 transformer with multi-head attention:

```
CLIP(
  (visual): VisionTransformer(
    (conv1): Conv2d(3, 768, kernel_size=(32, 32), stride=(32, 32),
bias=False)
    (ln_pre): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
    (transformer): Transformer(
      (resblocks): Sequential(
        (0): ResidualAttentionBlock(
          (attn): MultiheadAttention(
            (out_proj): NonDynamicallyQuantizableLinear(in_features=768,
out_features=768, bias=True)
          )
          (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
          (mlp): Sequential(
            (c_fc): Linear(in_features=768, out_features=3072, bias=True)
            (gelu): QuickGELU()
            (c_proj): Linear(in_features=3072, out_features=768,
bias=True)
          )
          (ln_2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        )
```

Another interesting aspect of the `model` cell is to look into the size-512 text encoder that runs jointly with the image encoder:

```
(transformer): Transformer(
    (resblocks): Sequential(
      (0): ResidualAttentionBlock(
        (attn): MultiheadAttention(
          (out_proj): NonDynamicallyQuantizableLinear(in_features=512,
out_features=512, bias=True)
        )
        (ln_1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
        (mlp): Sequential(
          (c_fc): Linear(in_features=512, out_features=2048, bias=True)
          (gelu): QuickGELU()
          (c_proj): Linear(in_features=2048, out_features=512, bias=True)
        )
        (ln_2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
      )
```

Go through the cells that describe the architecture, configuration, and parameters to see how CLIP represents data.

We showed that task-agnostic transformer models process image-text pairs as text-text pairs. We could apply task-agnostic models to music-text, sound-text, music-images, and any type of data pairs.

We will now explore DALL-E, another task-agnostic transformer model that can process images and text.

DALL-E

DALL-E, as with CLIP, is a task-agnostic model. CLIP processed text-image pairs. DALL-E processes the text and image tokens differently. DALL-E's input is a single stream of text and image of 1,280 tokens. 256 tokens are for the text, and 1,024 tokens are used for the image. DALL-E is a foundation model like CLIP.

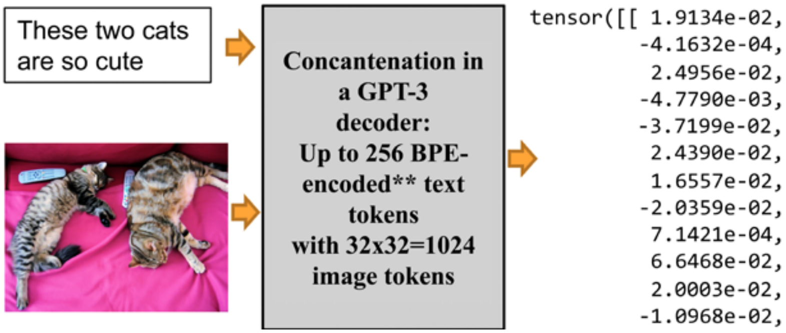
DALL-E was named after *Salvador Dali* and Pixar's WALL-E. The usage of DALL-E is to enter a text prompt and produce an image. However, DALL-E must first learn how to generate images with text.

DALL-E is a 12-billion-parameter version of GPT-3.

This transformer generates images from text descriptions using a dataset of text-image pairs.

The Basic Architecture of DALL-E

Unlike CLIP, DALL-E concatenates up to 256 BPE-encoded text tokens with 32×32 = 1,024 image tokens, as shown in *Figure 15.14*:


These two cats are so cute
Concantenation in a GPT-3 decoder: Up to 256 BPE-encoded** text tokens with 32x32=1024 image tokens

```
tensor([[ 1.9134e-02,
         -4.1632e-04,
          2.4956e-02,
         -4.7790e-03,
         -3.7199e-02,
          2.4390e-02,
          1.6557e-02,
         -2.0359e-02,
          7.1421e-04,
          6.6468e-02,
          2.0003e-02,
         -1.0968e-02,
```

Figure 15.14: DALL-E concatenates text and image input

Figure 15.14 shows that this time, our cat image is concatenated with the input text.

DALL-E has an encoder and a decoder stack, which is built with a hybrid architecture of infusing convolutional functions in a transformer model.

Let's peek into the code to see how the model works.

DALL-E in code

In this section, we will see how DALL-E reconstructs images.

Open `Vision_Transformers.ipynb`. Then go to the `DALL-E` cell of the notebook. The notebook first installs OpenAI DALL-E:

```
!pip install DALL-E
```

The notebook downloads the images and processes an image:

```
import io
import os, sys
import requests
import PIL

import torch
import torchvision.transforms as T
import torchvision.transforms.functional as TF

from dall_e import map_pixels, unmap_pixels, load_model
from IPython.display import display, display_markdown

target_image_size = 256

def download_image(url):
    resp = requests.get(url)
    resp.raise_for_status()
    return PIL.Image.open(io.BytesIO(resp.content))

def preprocess(img):
    s = min(img.size)

    if s < target_image_size:
        raise ValueError(f'min dim for image {s} < {target_image_size}')

    r = target_image_size / s
    s = (round(r * img.size[1]), round(r * img.size[0]))
    img = TF.resize(img, s, interpolation=PIL.Image.LANCZOS)
    img = TF.center_crop(img, output_size=2 * [target_image_size])
    img = torch.unsqueeze(T.ToTensor()(img), 0)
    return map_pixels(img)
```

The program now loads the OpenAI DALL-E encoder and decoder:

```
# This can be changed to a GPU, e.g. 'cuda:0'.
dev = torch.device('cpu')

# For faster load times, download these files locally and use the local
paths instead.
enc = load_model("https://cdn.openai.com/dall-e/encoder.pkl", dev)
dec = load_model("https://cdn.openai.com/dall-e/decoder.pkl", dev)
```

I added the enc and dec cells so that you can look into the encoder and decoder blocks to see how this hybrid model works: the convolutional functionality in a transformer model and the concatenation of text and image input.

The image processed in this section is mycat.jpg (creator: Denis Rothman, all rights reserved, written permission required to reproduce it). The image is in the Chapter15 directory of this book's repository. It is downloaded and processed:

```
x=preprocess(download_image('https://github.com/Denis2054/AI_Educational/
blob/master/mycat.jpg?raw=true'))
```

Finally, we display the original image:

```
display_markdown('Original image:')
display(T.ToPILImage(mode='RGB')(x[0]))
```

The output displays the image:

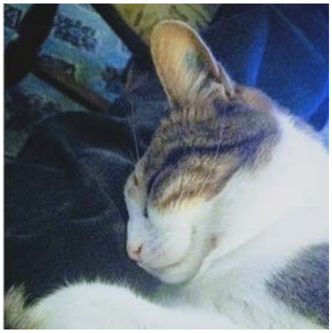


Figure 15.15: An image of a cat

Now, the program processes and displays the *reconstructed* image:

```
import torch.nn.functional as F

z_logits = enc(x)
z = torch.argmax(z_logits, axis=1)
z = F.one_hot(z, num_classes=enc.vocab_size).permute(0, 3, 1, 2).float()

x_stats = dec(z).float()
x_rec = unmap_pixels(torch.sigmoid(x_stats[:, :3]))
x_rec = T.ToPILImage(mode='RGB')(x_rec[0])

display_markdown('Reconstructed image:')
display(x_rec)
```

The reconstructed image looks extremely similar to the original:

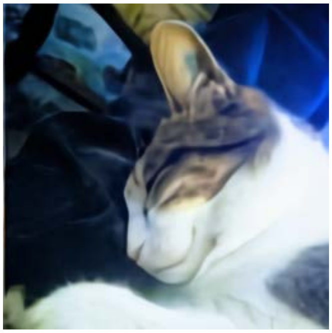


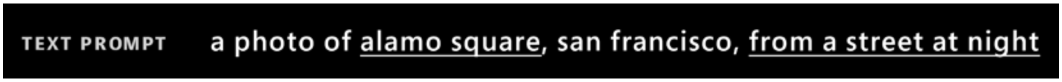
Figure 15.16: DALL-E reconstructs the image of the cat

The result is impressive. DALL-E learned how to generate images on its own.

The full DALL-E source code is not available at the time of the book's writing and might never be. An OpenAI API to generate images from text prompts is not online yet. But keep your eyes open!

In the meantime, we can continue discovering DALL-E on OpenAI at `https://openai.com/blog/dall-e/`

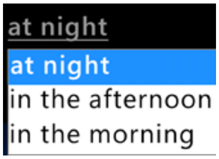
Once you have reached the page, scroll down to the examples provided. For example, I chose a photo of Alamo Square in San Francisco as a prompt:



TEXT PROMPT a photo of alamo square, san francisco, from a street at night

Figure 15.17: Prompt for Alamo Square, SF

Then I modified "at night" to "in the morning":



at night

at night
in the afternoon
in the morning

Figure 15.18: Modifying the prompt

DALL-E then generated a multitude of `text2image` images:

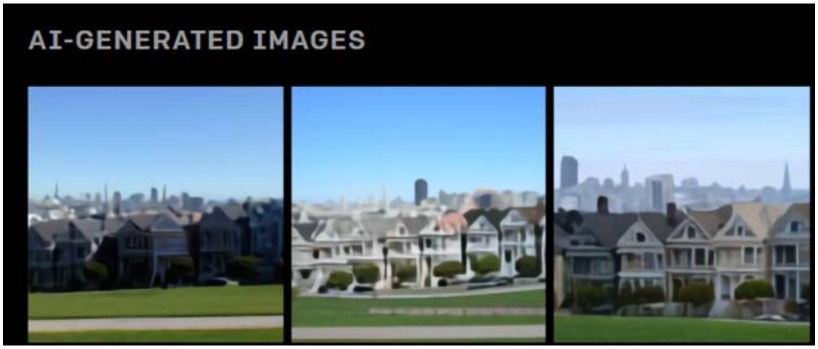


Figure 15.19: Generating images from text prompts

We have implemented ViT, CLIP, and DALL-E, three vision transformers. Let's go through some final thoughts before we finish.

An expanding universe of models

New transformer models, like new smartphones, emerge nearly every week. Some of these models are both mind-blowing and challenging for a project manager:

- **ERNIE** is a continual pretraining framework that produces impressive results for language understanding.

  Paper: `https://arxiv.org/abs/1907.12412`

Challenges: Hugging Face provides a model. Is it a full-blown model? Is it the one Baidu trained to exceed human baselines on the SuperGLUE Leaderboard (December 2021): `https://super.gluebenchmark.com/leaderboard`? Do we have access to the best one or just a toy model? What is the purpose of running AutoML on such small versions of models? Will we gain access to it on the Baidu platform or a similar one? How much will it cost?

- **SWITCH**: A trillion-parameter model optimized with sparse modeling.

  Paper: `https://arxiv.org/abs/2101.03961`

  Challenges: The paper is fantastic. Where is the model? Will we ever have access to the real fully trained model? How much will it cost?

- **Megatron-Turing**: A 500 billion parameter transformer model.

  Blog: `https://developer.nvidia.com/blog/using-deepspeed-and-megatron-to-train-megatron-turing-nlg-530b-the-worlds-largest-and-most-powerful-generative-language-model/`

  Challenges: One of the best models on the market. Will we have access through an API? Will it be a full-blown model? How much will it cost?

- **XLNET** is pretrained like BERT, but the authors contend it exceeds BERT model performance.

  Paper: `https://proceedings.neurips.cc/paper/2019/file/dc6a7e655d7e5840e66733e9ee67cc69-Paper.pdf`

  Challenges: Does XLNET really exceed the performances of Google BERT, the version Google uses for their activities? Do we have access to the best versions of Google BERT or XLNET models?

The list has become endless and it is growing!

Testing them all remains a challenge beyond the issues mentioned previously. Only a few transformer models qualify as foundation models. A foundation model must be:

- Fully trained for a wide range of tasks
- Able to perform tasks it was not trained for because of the unique level of NLU it has attained
- Sufficiently large to guarantee reasonably accurate results, such as OpenAI GPT-3

Many sites offer transformers that prove useful for educational purposes but cannot be considered sufficiently trained and large to qualify for benchmarking.

The best approach is to deepen your understanding of transformer models as much as possible. At one point, you will become an expert, and finding your way through the jungle of big tech innovations will be as easy as choosing a smartphone!

Summary

New transformer models keep appearing on the market. Therefore, it is good practice to keep up with cutting-edge research by reading publications and books and testing some systems.

This leads us to assess which transformer models to choose and how to implement them. We cannot spend months exploring every model that appears on the market. We cannot change models every month if a project is in production. Industry 4.0 is moving to seamless API ecosystems.

Learning all the models is impossible. However, understanding a new model quickly can be achieved by deepening our knowledge of transformer models.

The basic structure of transformer models remains unchanged. The layers of the encoder and/or decoder stacks remain identical. The attention head can be parallelized to optimize computation speeds.

The Reformer model applies **LSH** buckets and chunking. It also recomputes each layer's input instead of storing the information, thus optimizing memory issues. However, a billion-parameter model such as GPT-3 produces acceptable results for the same examples.

The DeBERTa model disentangles content and positions, making the training process more flexible. The results are impressive. However, billion-parameter models such as GPT-3 can equal the outputs of a DeBERTa.

ViT, CLIP, and DALL-E took us into the fascinating world of task-agnostic text-image vision transformer models. Combining language and images produces new and productive information.

The question remains to see how far ready-to-use AI and automated systems will go. We will attempt to visualize the future of transformer-based AI in the next chapter on the rise of metahumans.

Questions

1. Reformer transformer models don't contain encoders. (True/False)
2. Reformer transformer models don't contain decoders. (True/False)

3. The inputs are stored layer by layer in Reformer models. (True/False)
4. DeBERTa transformer models disentangle content and positions. (True/False)
5. It is necessary to test the hundreds of pretrained transformer models before choosing one for a project. (True/False)
6. The latest transformer model is always the best. (True/False)
7. It is better to have one transformer model per NLP task than one multi-task transformer model. (True/False)
8. A transformer model always needs to be fine-tuned. (True/False)
9. OpenAI GPT-3 engines can perform a wide range of NLP tasks without fine-tuning. (True/False)
10. It is always better to implement an AI algorithm on a local server. (True/False)

References

- Hugging Face Reformer: `https://huggingface.co/transformers/model_doc/reformer.html?highlight=reformer`
- Hugging Face DeBERTa: `https://huggingface.co/transformers/model_doc/deberta.html`
- *Pengcheng He*, *Xiaodong Liu*, *Jianfeng Gao*, *Weizhu Chen*, 2020, *Decoding-enhanced BERT with Disentangled Attention*: `https://arxiv.org/abs/2006.03654`
- *Alexey Dosovitskiy*, *Lucas Beyer*, *Alexander Kolesnikov*, *Dirk Weissenborn*, *Xiaohua Zhai*, *Thomas Unterthiner*, *Mostafa Dehghani*, *Matthias Minderer*, *Georg Heigold*, *Sylvain Gelly*, *Jakob Uszkoreit*, *Neil Houlsby*, *2020, An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*: `https://arxiv.org/abs/2010.11929`
- OpenAI: `https://openai.com/`
- *William Fedus*, *Barret Zoph*, *Noam Shazeer*, 2021, *Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity*: `https://arxiv.org/abs/2101.03961`
- *Alec Radford*, *Jong Wook Kim*, *Chris Hallacy*, *Aditya Ramesh*, *Gabriel Goh*, *Sandhini Agarwal*, *Girish Sastry*, *Amanda Askell*, *Pamela Mishkin*, *Jack Clark*, *Gretchen Krueger*, *Ilya Sutskever*, 2021, *Learning Transferable Visual Models From Natural Language Supervision*: `https://arxiv.org/abs/2103.00020`
- C7LIP: `https://github.com/openai/CLIP`

- *Aditya Ramesh*, *Mikhail Pavlov*, *Gabriel Goh*, *Scott Gray*, *Chelsea Voss*, *Alec Radford*, *Mark Chen*, *Ilya Sutskever*, 2021, *Zero-Shot Text-to-Image Generation*: `https://arxiv.org/abs/2102.12092`
- DALL-E: `https://openai.com/blog/dall-e/`

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the authors:

`https://www.packt.link/Transformers`

16

The Emergence of Transformer-Driven Copilots

When **Industry 4.0** (**I4.0**) reaches maturity, it will all be about machine-to-machine connections, communication, and decision-making. AI will be primarily embedded in ready-to-use pay-as-you-go cloud AI solutions. Big tech will absorb the most talented AI specialists to create APIs, interfaces, and integration tools.

AI specialists will go from development to design to becoming architects, integrators, and cloud AI pipeline administrators. Thus, AI is becoming a job for engineer consultants more than engineer developers.

Chapter 1, What Are Transformers?, introduced foundation models, transformers that can do NLP tasks they were not trained for. *Chapter 15, From NLP to Task-Agnostic Transformer Models*, expanded foundation model transformers to task-agnostic models that can perform vision tasks, NLP tasks, and much more.

This chapter will extend task-agnostic OpenAI GPT-3 models to a wide range of copilot tasks. A new generation of AI specialists and data scientists will learn how to work with AI copilots to help them generate source code automatically and make decisions.

This chapter begins by exploring prompt engineering in more detail. The example task consists of converting meeting notes into a summary. Transformers boost our productivity. However, we will see how natural language remains a challenge for AI.

We will learn how to use OpenAI Codex as a copilot. GitHub Copilot suggests source code as we write our programs using Codex. Codex can also convert natural language into code.

We will then discover new AI methods with domain-specific GPT-3 engines. This chapter will show how to generate embeddings with 12,288 dimensions and plug them into machine learning algorithms. We will also see how to ask a transformer to produce instructions automatically.

We will see how to filter biased input and output before looking into transformer-driven recommenders. AI of the 2020s must be built with ethical methods.

Recommender systems have permeated every social media platform to suggest videos, posts, messages, books, and many other products we might want to consume. We will build an educational multi-purpose transformer-based recommender system using ML in the process.

Transformer models analyze sequences. They began with NLP but have successfully expanded to computer vision. We will explore a transformer-based computer vision program developed in JAX.

Finally, we will see AI copilots contribute to the transition of virtual systems into metaverses, which will expand in this decade. You are the pilot when you develop your applications. However, when you have code to develop, the activated completions are limited to methods, not lines of code. An IDE might suggest a list of methods. Copilots can produce completions of whole paragraphs of code!

This chapter covers the following topics:

- Prompt engineering
- GitHub Copilot
- Codex language to source code models
- Embedding datasets
- Embedded-driven machine learning
- Instruct series
- Content filter models
- Exploring transformer-based recommenders
- Extending NLP sequence learning to behavior predictions
- Implementing transformer models in JAX
- Applying transformer models to computer vision

Let's begin with prompt engineering, which is a critical ability to acquire.

Prompt engineering

Speaking a specific language is not hereditary. There is not a language center in our brain containing the language of our parents. Our brain engineers our neurons early in our lives to speak, read, write, and understand a language. Each human has a different language circuitry depending on their cultural background and how they were communicated with in their early years.

As we grow up, we discover that much of what we hear is chaos: unfinished sentences, grammar mistakes, misused words, bad pronunciation, and many other distortions.

We use language to convey a message. We quickly find that we need to adapt our language to the person or audience we address. We might have to try additional "inputs" or "prompts" to obtain the result ("output") we expect. Foundation-level transformer models such as GPT-3 can perform hundreds of tasks in an indefinite number of ways. *We must learn the language of transformer prompts and responses as we would any other language*. Effective communication with a person or near-human-level transformer must contain a minimum amount of information to maximize results. We represent the minimum input information to obtain a result as *minI* and the maximum output of any system as *maxR*.

We can represent this chain of communication as:

$$minI(input) \rightarrow maxR(output)$$

We will replace "input" with "prompt" for transformers to show that our input influences how the model will react. The output is the "response." The dialogue with transformers, *d(T)*, can be expressed as:

$$d(T)=minI(prompt) \rightarrow maxR(response)$$

When *minI(prompt)*→`1`, the probability of *maxR*(response) →`1`.

When *minI(prompt)* →`0`, the probability of *maxR*(response) →`0`.

The quality *d(T)* depends on how well we can define *minI(prompt)*.

If your prompt tends to reach `1`, then it will produce probabilities that tend to `1`.

If your prompt tends to reach `0`, then it will produce output probabilities that tend to `0`.

Your prompt is part of the content that impacts the probabilities! Why? Because the transformer will include the prompt and the response in its estimations.

It takes many years to learn a language as a child or an adult. It also takes quite some time to learn the language of transformers and how to design *minI(prompt)* effectively. We need to understand them, their architecture, and the way the algorithms calculate predictions. Then we need to spend quite some time understanding how to design the input, the prompt, for the transformers to behave as we expect.

This section focuses on oral language. The prompt for OpenAI GPT-3 for an NLP task will often be taken from meeting notes or conversations, which tend to be unstructured. Transforming meeting notes or conversations into a summary can be quite challenging. This section will focus on summarizing notes of the same conversation in seven situations that go from casual English to casual or formal English with limited context.

We will begin with casual English with a meaningful context.

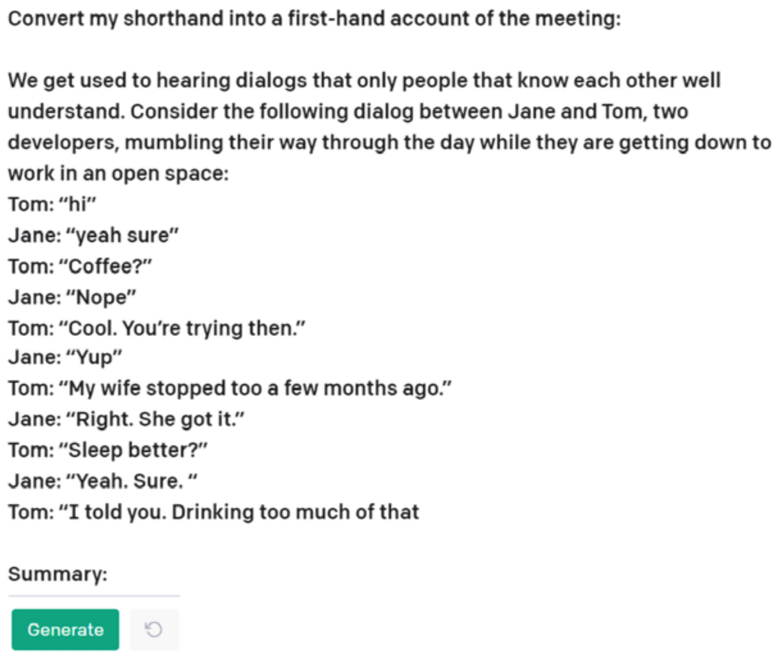
Casual English with a meaningful context

Casual English is spoken with shorter sentences and limited vocabulary.

Let's ask OpenAI GPT-3 to perform a "notes to summary" task. Go to `www.openai.com`. Log in or sign up. Then go to the **Examples** page and select **Notes to summary**.

We will give GPT-3 all the information required to summarize a casual conversation between Jane and Tom. Jane and Tom are two developers starting work. Tom offers Jane coffee. Jane declines the offer.

In this case, *minI(prompt)=1* since the input information is fine, as shown in *Figure 16.1*:



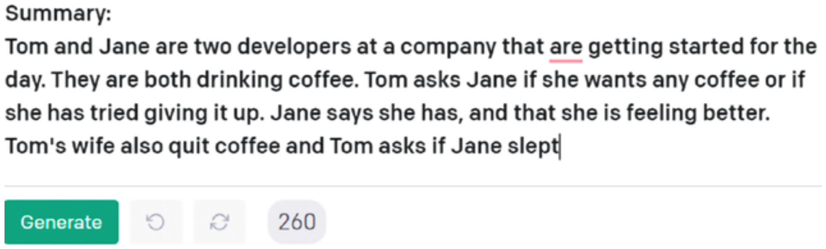
Convert my shorthand into a first-hand account of the meeting:

We get used to hearing dialogs that only people that know each other well understand. Consider the following dialog between Jane and Tom, two developers, mumbling their way through the day while they are getting down to work in an open space:
Tom: "hi"
Jane: "yeah sure"
Tom: "Coffee?"
Jane: "Nope"
Tom: "Cool. You're trying then."
Jane: "Yup"
Tom: "My wife stopped too a few months ago."
Jane: "Right. She got it."
Tom: "Sleep better?"
Jane: "Yeah. Sure. "
Tom: "I told you. Drinking too much of that

Summary:

Figure 16.1: Summarizing well-documented notes

When we click on **Generate**, we get a surprisingly good answer, as shown in *Figure 16.2*:



Summary:
Tom and Jane are two developers at a company that are getting started for the day. They are both drinking coffee. Tom asks Jane if she wants any coffee or if she has tried giving it up. Jane says she has, and that she is feeling better. Tom's wife also quit coffee and Tom asks if Jane slept

Figure 16.2: An acceptable summary is provided by GPT-3

Can we conclude that AI can find structures in our chaotic daily conversations, meetings, and aimless chatter? The answer isn't easy. We will now complicate the input by adding a metonymy.

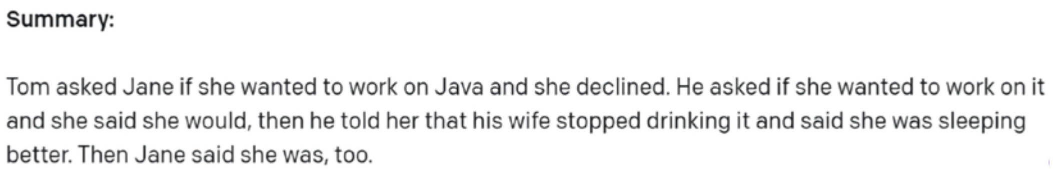
Casual English with a metonymy

Tom mentioned the word `coffee`, setting GPT-3 on track. But what if Tom used the word `java` instead of `coffee`. Coffee refers to the beverage, but `java` is an ingredient that comes from the island of Java. A metonymy is when we use an attribute of an object, such as `java` for `coffee`. Java is also a programming language with a logo that is a cup of coffee.

We are facing three possible definitions of `java`: an ingredient of coffee meaning coffee (metonymy), the island of Java, and the name of a programming language. GPT-3 now has a polysemy (several meanings of the same word) issue to solve.

Humans master polysemy. We learn the different meanings of words. We know that a word doesn't mean much without context. In this case, Jane and Tom are developers, complicating the situation. Are they talking about coffee or the language?

The answer is easy for a human since Tom then talks about his wife, who stopped drinking it. GPT-3 can be confused by this polysemy when the word `java` replaces `coffee`, and it produces an incorrect answer:



Summary:

Tom asked Jane if she wanted to work on Java and she declined. He asked if she wanted to work on it and she said she would, then he told her that his wife stopped drinking it and said she was sleeping better. Then Jane said she was, too.

Figure 16.3: Incorrect GPT-3 response when prompt is confusing

We thus confirm that when $minI(prompt)\rightarrow 0$, the probability of $maxR(response)\rightarrow 0$.

Human conversations can become even more difficult to analyze if we add an ellipsis.

Casual English with an ellipsis

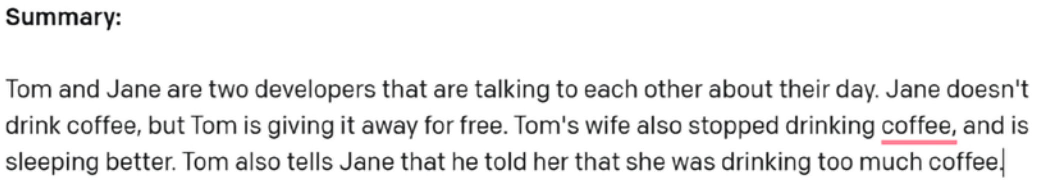
The situation can get worse. Let's suppose Tom is drinking a cup of coffee, and Jane looks at him and the cup of coffee as she casually greets him.

Instead of asking Jane if she wants coffee or java, Tom says:

```
"Want some?"
```

Tom left out the word `coffee`, which is an ellipsis. Jane can still understand what Tom means by looking at him holding a cup of coffee.

OpenAI GPT-3 detects the word `drinking` and still manages to associate this verb with the question `Want some?`. We don’t`want some` of a programming language. The following summary produced by GPT-3 is still correct:



Summary:

Tom and Jane are two developers that are talking to each other about their day. Jane doesn't drink coffee, but Tom is giving it away for free. Tom's wife also stopped drinking coffee, and is sleeping better. Tom also tells Jane that he told her that she was drinking too much coffee.

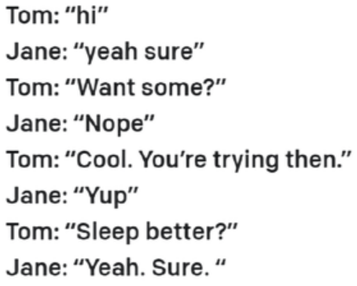
Figure 16.4: Correct response produced by GPT-3

Now, let’s see what happens when there is vague context that a human can understand but remains a challenge for AI.

Casual English with vague context

If we take this further, `Tom` doesn’t need to mention his wife for `Jane` to understand what he is talking about since he is holding a cup of coffee.

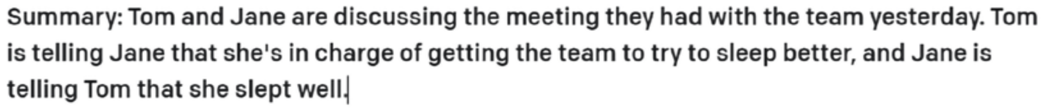
Let’s remove `Tom`'s reference to his wife and the verb `drinking`. Let’s leave `want some` in instead of coffee or java:



Tom: “hi”
Jane: “yeah sure”
Tom: “Want some?”
Jane: “Nope”
Tom: “Cool. You’re trying then.”
Jane: “Yup”
Tom: “Sleep better?”
Jane: “Yeah. Sure. “

Figure 16.5: Vague input context

The output reflects the apparent chaos of the conversation:



Summary: Tom and Jane are discussing the meeting they had with the team yesterday. Tom is telling Jane that she's in charge of getting the team to try to sleep better, and Jane is telling Tom that she slept well.

Figure 16.6: Poor GPT-3 response

The prompt was too vague, leading to an inadequate response that we can sum up as:

d(T)→0 because when minI(prompt)→`0`, the probability of *maxR(response)→*`0`

When humans communicate, they bring their culture, past relationships, visual situations, and other *invisible factors into a conversation*. These invisible factors for third parties can be:

- Reading a text without seeing what the people were doing (actions, facial expressions, body language, etc.)
- Listening to people refer to things they know about, but we don't (movies, sports, problems in a factory, etc.)
- Cultural events from a culture that's different from ours

The list is endless!

We can see that these *invisible* factors make AI *blind*.

Let's now introduce sensors into the situation.

Casual English with sensors

We now introduce video sensors into the room for a thought experiment. Imagine we can use image captioning with a video feed and supply a context early in the dialogue such as:

Humans sometimes generate dialogue that only people that know each other well understand. Consider the following dialogue between Jane and Tom. The video feed produces image captioning showing that Tom is drinking a cup of coffee and Jane is typing on her keyboard. Jane and Tom are two developers, mumbling their way through the day while getting down to work in an open space.

Then we provide the following chaotic chat as a prompt:

```
Tom: "hi" Jane: "yeah sure" Tom: "Want some?" Jane: "Nope" Tom: "Cool. You're
trying then." Jane: "Yup" Tom: "Sleep better?" Jane: "Yeah. Sure. "
```

The output of GPT-3 is acceptable though important semantic words are missing at the start:

```
Summarize: A developer can be seen typing on her keyboard. Another
developer enters the room and offers her a cup of coffee. She declines,
but he insists. They chat about her sleep and the coffee.
```

The results may change from one run to another. GPT-3 looks at the top probabilities and selects one of the best. GPT-3 made it through this experiment because image captioning provided the context.

However, what if Tom is not holding a cup of coffee, depriving GPT-3 of visual context?

Casual English with sensors but no visible context

The most difficult situation for AI is if Tom refers to an event every day but not today. Suppose that Tom comes in with a cup of coffee every morning. He comes in now and asks Jane if she `wants` some *before* getting some coffee. Our thought experiment is to imagine all the possible cases. In that case, the video feed in our thought experiment will reveal nothing, and we are back to chaos again. Also, the video feed can't see if they are developers, accountants, or consultants. So, let's take that part of context out, which leaves us with the following context. Let's go further. The dialogue contains `Tom`: and `Jane:`. So we don't need to mention that context. We are left with:

```
Tom: "hi" Jane: "yeah sure" Tom: "Want some?" Jane: "Nope" Tom: "Cool. You're
trying then." Jane: "Yup" Tom: "Sleep better?" Jane: "Yeah. Sure."
```

The output is quite astonishing. The casual language used by Jane and Tom leads GPT-3 to absurd conclusions. Remember, GPT-3 is a stochastic algorithm. The slightest change in the inputs can lead to quite different outputs. GPT-3 is trying to guess what they are talking about. GPT-3 detects that the conversation is about consuming something. Their casual language leads to nonsensical predictions about illegal substances that I am not reproducing in this section for ethical reasons.

GPT-3 determines the level of language used and associates it with related situations.

What will happen if we reproduce this same experiment using formal English?

Formal English conversation with no context

Let's now keep all of the context out but provide formal English. Formal English contains longer sentences, good grammar, and manners. We can express the same conversation that contains no context using formal English:

```
Tom: "Good morning, Jane" Jane: "Good morning, Tom" Tom: "Want some as well?" Jane:
"No, thank you. I'm fine." Tom: "Excellent. You are on the right track!" Jane:
"Yes, I am" Tom: "Do you sleep better these days?" Jane: "Yes, I do. Thank you. "
```

GPT-3 naturally understands what Tom refers to with "drinking" with this level of English and good manners. The output is quite satisfactory:

```
Summarize: Tom says "good morning" to Jane. Tom offers her some of what he's drinking.
Jane says "no, thank you. I'm fine." Tom says "excellent" and that she is on the
right track. Jane says, "yes, I am." Tom asks if she sleeps better these days.
```

We could imagine an endless number of variations on this same conversation by introducing other people into the dialogue and other objects and generating an endless number of situations.

Let's sum these experiments up.

Prompt engineering training

Our thoughts are often chaotic. Humans use many methods to reconstruct unstructured sentences. *Humans often need to ask additional questions to understand what somebody is talking about. You need to accept this when interacting with a trained transformer such as OpenAI GPT-3*.

Keep in mind that a *dialogue d(T)* with a transformer and the response, *maxR(response)*, depends on the quality of your inputs, *minI(prompt)*, as defined at the beginning of this section:

$$d(T)=minI(prompt) \rightarrow maxR(response)$$

When *minI(prompt)*→`1`, the probability of *maxR(response)*→`1`.

When *minI(prompt)*→`0`, the probability of *maxR(response)*→`0`.

Practice prompt engineering and measure your progress in time. Prompt engineering is a new skill that will take you to the next level of AI.

The prompt engineering abilities lead to being able to master copilots.

Copilots

Welcome to the world of AI-driven development copilots powered by OpenAI and available in Visual Studio.

GitHub Copilot

Let's begin with GitHub Copilot:

`https://github.com/github/copilot-docs`

In this section, we will use GitHub Copilot with PyCharm (JetBrains):

`https://github.com/github/copilot-docs/tree/main/docs/jetbrains`

Follow the instructions in the documentation to install JetBrains and activate OpenAI GitHub Copilot in PyCharm.

Working withGitHub Copilot is a four-step process (see *Figure 16.7*):

- OpenAI Codex is trained on public code and text on the internet.
- The trained model is plugged into the GitHub Copilot service.
- The GitHub service manages back-and-forth flows between code we write in an editor (in this case PyCharm) and OpenAI Codex. The GitHub Service Manager makes suggestions and then sends the interactions back for improvement.
- The code editor is our development workspace.

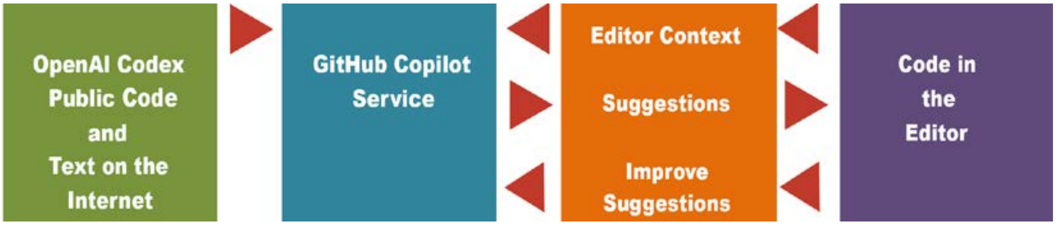


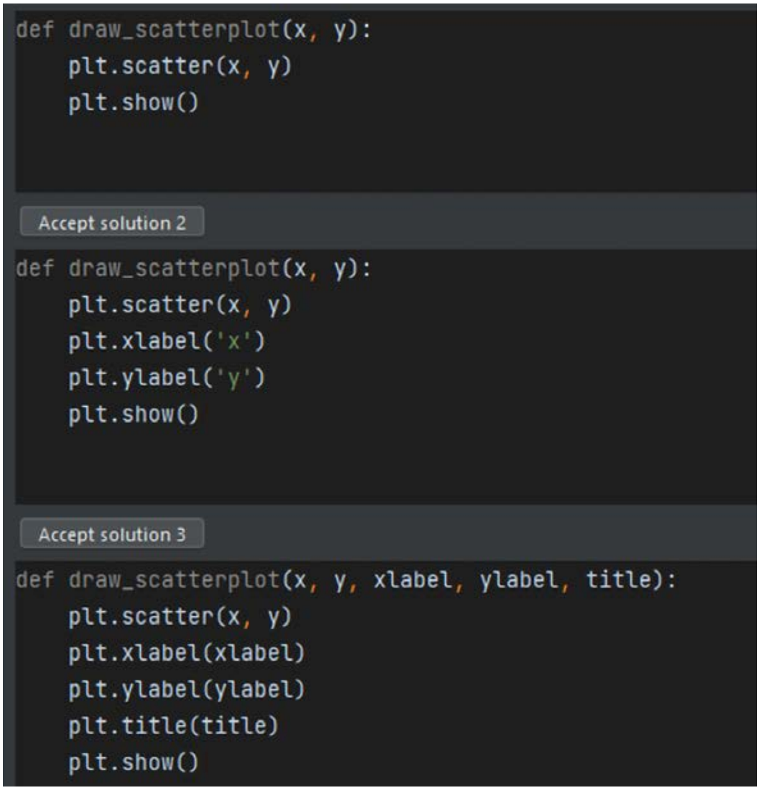
Figure 16.7: GitHub Copilot's four-step process

Follow the instructions provided by GitHub Copilot, log in to GitHub when you are in PyCharm. For any troubleshooting, read `https://copilot.github.com/#faqs`.

Once you are all set in the PyCharm editor, simply type:

```python
import matplotlib.pyplot as plt
def draw_scatterplot
```

As soon as the code is typed, you can open the OpenAI GitHub suggestion pane and see the suggestions:



```
def draw_scatterplot(x, y):
    plt.scatter(x, y)
    plt.show()
```

```
def draw_scatterplot(x, y):
    plt.scatter(x, y)
    plt.xlabel('x')
    plt.ylabel('y')
    plt.show()
```

```
def draw_scatterplot(x, y, xlabel, ylabel, title):
    plt.scatter(x, y)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title(title)
    plt.show()
```

Figure 16.8: Suggestions for the code you typed

Once you choose the copilot suggestion you prefer, it will appear in the editor. You can confirm suggestions with the *Tab* key. You can wait for another suggestion, such as drawing a scatterplot:

```
import matplotlib.pyplot as plt
def draw_scatterplot(x, y):
    plt.scatter(x, y)
    plt.xlabel('x')
    plt.ylabel('y')
    plt.show()




draw_scatterplot([1, 2, 3, 4, 5], [1, 4, 9, 16, 25])
```

The plot will be displayed:

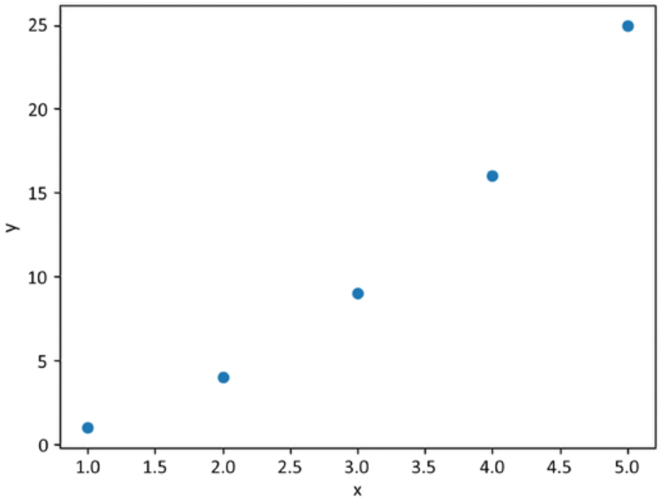


Figure 16.9: A GitHub Copilot scatterplot

You can run the result on your machine with `GitHub_Copilot.py`, which is in the `Chapter16` folder in this book's GitHub repository.

The technology is seamless, invisible, and will progressively expand into all areas of development. The system is packed with GPT-3 functionality along with other pipelines. The technology is available for Python, JavaScript, and more.

It will take training with prompt engineering to get used to working with GitHub Copilot driven by OpenAI Codex.

Let's go directly to OpenAI Codex, which can be a good place to train using copilots.

Codex

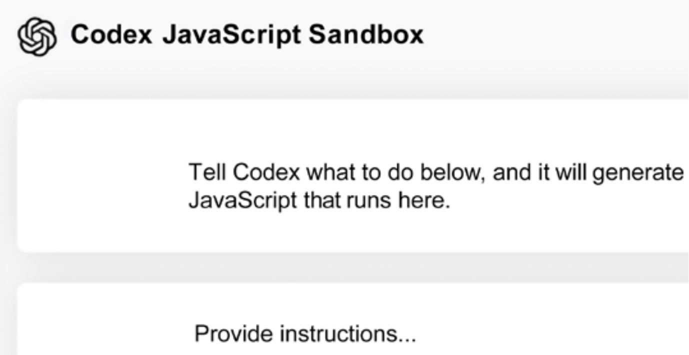
OpenAI Codex does more than suggest source code. Codex can translate natural language into source code.

Go to the OpenAI website and click on the link to the Codex interface:

`https://beta.openai.com/codex-javascript-sandbox`

Provide instructions and the result will be displayed in the Codex window and the source code on the right pane of the interface.

The example in this section contains a few sequences using the prompt pane:



Tell Codex what to do below, and it will generate JavaScript that runs here.

Provide instructions...

Figure 16.10: Codex's JavaScript sandbox

We enter natural language in the **Provide instructions...** pane, generating JavaScript that will run.

For example, I entered two instructions in natural language:

```
Draw 50 small ping pong balls of all sorts of colors
```

```
Make the balls round
```

Codex is derived from GPT-3. In some cases, the engine is referred to as "davinci-codex." This means Codex inherits the properties of a GPT-3 engine. To understand this, let's take one parameter among others: `top_p`. The engine samples the output of the engine. For example, if `top_p` is set to `0.1`, it will only take the top 10% of the sampling into account. It will retrieve the tokens in that top 10% probability mass. Since the whole computation is stochastic, it might take some top choices and another set of top choices to run after. Be patient and take your time designing the prompt.

Your AI engine learning path:

1. Learn to understand the behavior of the Codex engine.
2. Accept the free creative nature of stochastic algorithms.
3. Get used to piloting them with better prompts. You will then be able to grow with Codex as the engines improve.

The script will appear in the JavaScript pane with commented instructions and the code:

```
/* draw 50 small ping pong balls of all sorts of colors */
var balls = [];
for (var i = 0; i < 50; i++) {
  var ball = document.createElement('div');
  ball.style.width = '10px';
  ball.style.height = '10px';
  ball.style.backgroundColor = 'hsl(' + Math.random() * 360 + ', 100%,
50%)';
  ball.style.position = 'absolute';
  ball.style.left = Math.random() * window.innerWidth + 'px';
  ball.style.top = Math.random() * window.innerHeight + 'px';
  document.body.appendChild(ball);
  balls.push(ball);
}

/* Make the balls round */
for (var i = 0; i < balls.length; i++) {
  balls[i].style.borderRadius = '50%';
}
```

The result is displayed:

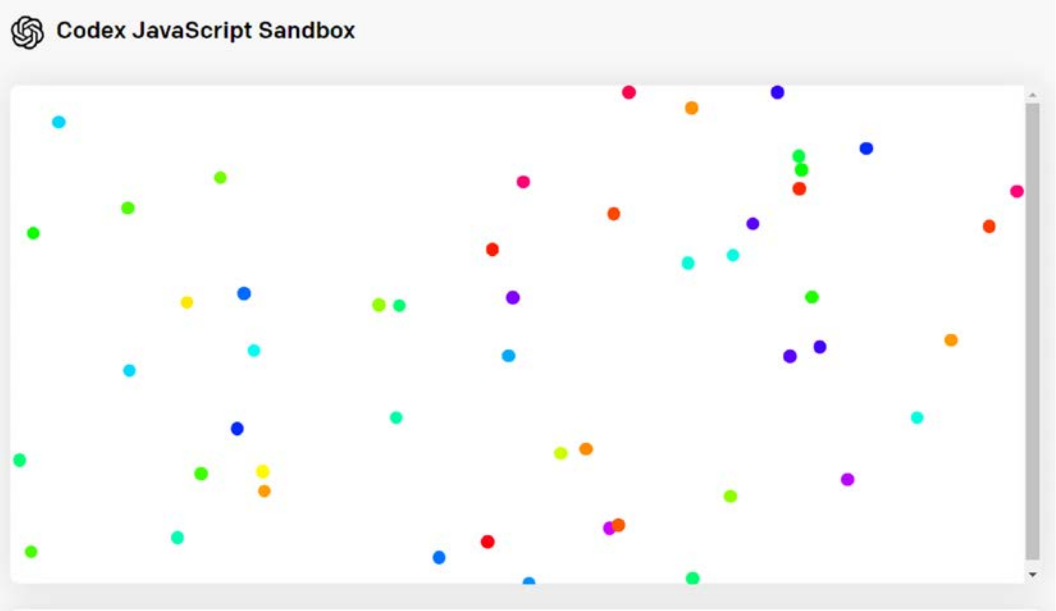


Figure 16.11: Creating fifty multicolored balls

Now let's ask the program to move the balls:

```
Make all of the balls move inside the window
```

The code is generated:

```
/* make all of the balls move inside the window. */
var moveBalls = function() {
  for (var i = 0; i < balls.length; i++) {
    var ball = balls[i];
    ball.style.left = (parseInt(ball.style.left) + Math.random() * 10 - 5)
+ 'px';
    ball.style.top = (parseInt(ball.style.top) + Math.random() * 10 - 5) +
'px';
  }
  window.requestAnimationFrame(moveBalls);
};
moveBalls();
```

Once the balls move, export the code to HTML by clicking on the **Export to JSFiddle** button:

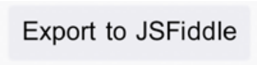
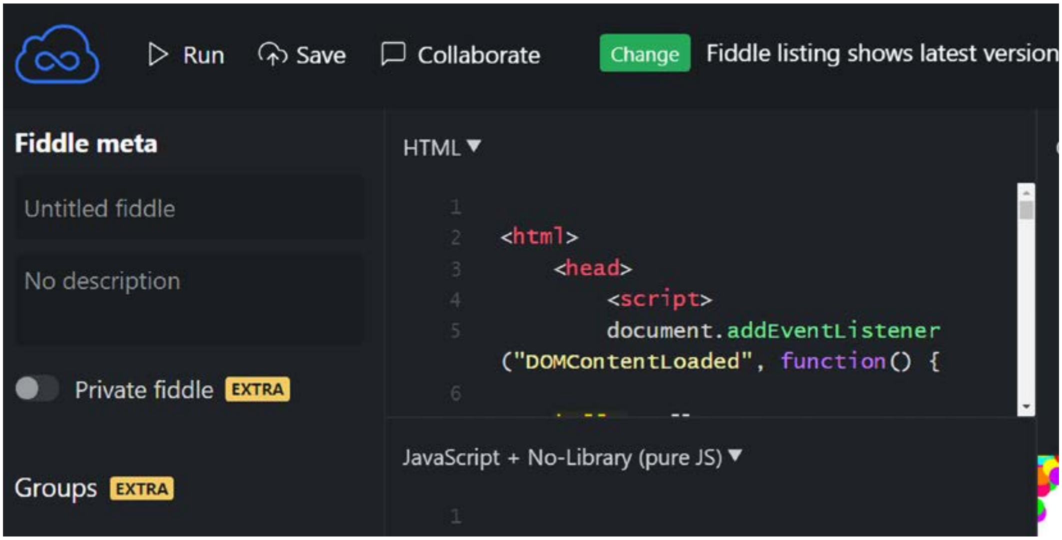


Figure 16.12: The Export to JSFiddle button

JSFiddle creates an HTML page:



```
<html>
    <head>
        <script>
        document.addEventListener
("DOMContentLoaded", function() {
```

Figure 16.13: JSFiddle creating an HTML page

In this case, the code was saved to `codex.html`, which is in this chapter's folder in the GitHub repository of the book. You can open and watch the innovative result of creating an HTML page with language natural to code source code.

Domain-specific GPT-3 engines

This section explores GPT-3 engines that can perform domain-specific tasks. We will run three models in the three subsections of this section:

- Embedding2ML to use GPT-3 to provide embeddings for ML algorithms
- Instruct series to ask GPT-3 to provide instructions for any task
- Content filter to filter bias or any form of unacceptable input and output

Open `Domain_Specific_GPT_3_Functionality.ipynb`.

We will begin with embedding2ML (embeddings as an input to ML).

Embedding2ML

OpenAI has trained several embedding models with different dimensions with different capabilities:

- Ada (1,024 dimensions)
- Babbage (2,048 dimensions)
- Curie (4,096 dimensions)
- Davinci (12,288 dimensions)

For more explanations on each engine, you will find more information on OpenAI's website:

`https://beta.openai.com/docs/guides/embeddings`.

The Davinci model offers embedding with 12,288 dimensions. In this section, we will use the power of Davinci to generate the embeddings of a supply chain dataset. However, we will not send the embeddings to the embedding sublayer of the transformer!

We will send the embeddings to a clustering machine learning program from the scikit-learn library in six steps:

- *Step 1: Installing and importing OpenAI, and entering the API key*
- *Step 2: Loading the dataset*
- *Step 3: Combining the columns*

- *Step 4: Running the GPT-3 embedding*
- *Step 5: Clustering (k-means) with the embeddings*
- *Step 6: Visualizing the clusters (t-SNE)*

The process is summed up in *Figure 16.14*:

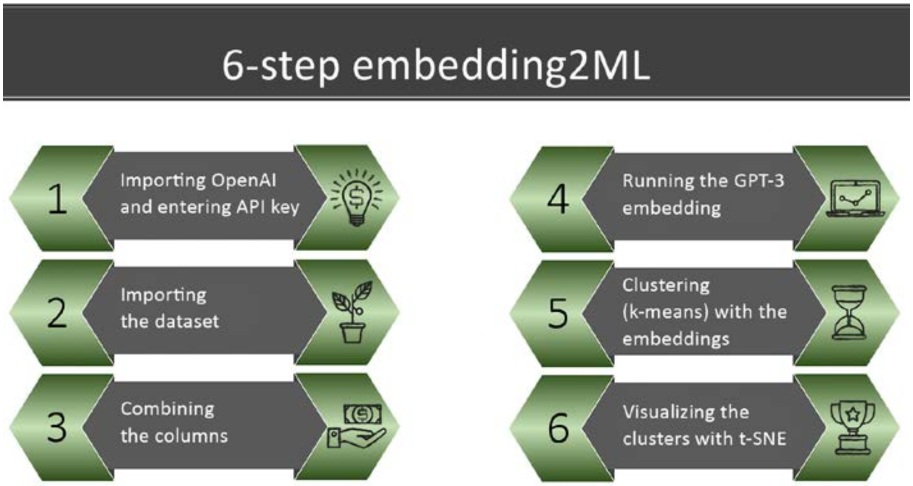

6-step embedding2ML
1
Importing OpenAI and entering API key
2
Importing the dataset
3
Combining the columns
4
Running the GPT-3 embedding
5
Clustering (k-means) with the embeddings
6
Visualizing the clusters with t-SNE

Figure 16.14: Six-step process for sending embeddings to a clustering algorithm

Open the Google Colab file, `Domain_Specific_GPT_3_Functionality.ipynb` and go to the `Embedding2ML with GPT-3 engine` section of the notebook.

The steps described in this section match the notebook cells. Let's go through a summary of each step of the process.

Step 1: Installing and importing OpenAI

Let's start with the following substeps:

1. Run the cell
2. Restart the runtime
3. Run the cell again to make sure since you restarted the runtime:

```
try:
  import openai
except:
  !pip install openai
  import openai
```

4. Enter the API key:

    ```
    openai.api_key="[YOUR_KEY]"
    ```

We now load the dataset.

Step 2: Loading the dataset

Load your file before running the cell. I uploaded `tracking.csv` (available in the GitHub repository of this book), which contains SCM data:

```
import pandas as pd
df = pd.read_csv('tracking.csv', index_col=0)
```

The data contains seven fields:

- `Id`
- `Time`
- `Product`
- `User`
- `Score`
- `Summary`
- `Text`

Let's print the first few lines using the following command:

```
print(df)
```

```
                  Time Product  User  Score        Summary   Text
Id
1     01/01/2016 06:30   WH001  C001      4        on time   AGV1
2     01/01/2016 06:30   WH001  C001      8           late     R1   NaN
3     01/01/2016 06:30   WH001  C001      2          early    R15   NaN
4     01/01/2016 06:30   WH001  C001     10  not delivered    R20   NaN
5     01/01/2016 06:30   WH001  C001      1        on time     R3   NaN
...                ...     ...   ...    ...            ...    ...   ...
1049  01/01/2016 06:30   WH003  C002      9        on time   AGV5   NaN
1050  01/01/2016 06:30   WH003  C002      2           late  AGV10   NaN
1051  01/01/2016 06:30   WH003  C002      1          early   AGV5   NaN
1052  01/01/2016 06:30   WH003  C002      6  not delivered   AGV2   NaN
1053  01/01/2016 06:30   WH003  C002      3        on time   AGV2   NaN

[1053 rows x 7 columns]
```

We can combine columns to build the clusters we wish.

Step 3: Combining the columns

We can combine the `Product` column with `Summary` to obtain a view of products and their delivery status. Remember that this is only an experimental exercise. In a real-life project, carefully analyze and decide on the columns you wish to combine.

The following example code can be replaced with any choice you make for your experimentation:

```
df['combined'] = df.Summary.str.strip()+ "-" + df.Product.str.strip()
print(df)
```

We can now see a new column named `combined`:

```
                 Time Product  User  ... Text           combined
Id                                    ...
1     01/01/2016 06:30 WH001  C001  ... AGV1             on time-WH001
2     01/01/2016 06:30 WH001  C001  ... R1  NaN           late-WH001
3     01/01/2016 06:30 WH001  C001  ... R15  NaN          early-WH001
4     01/01/2016 06:30 WH001  C001  ... R20  NaN  not delivered-WH001
5     01/01/2016 06:30 WH001  C001  ... R3  NaN        on time-WH001
...                ...     ...   ...  ...    ...  ...                  ...
1049  01/01/2016 06:30 WH003  C002  ... AGV5  NaN        on time-WH003
1050  01/01/2016 06:30 WH003  C002  ... AGV10  NaN         late-WH003
1051  01/01/2016 06:30 WH003  C002  ... AGV5  NaN          early-WH003
1052  01/01/2016 06:30 WH003  C002  ... AGV2  NaN  not delivered-WH003
1053  01/01/2016 06:30 WH003  C002  ... AGV2  NaN        on time-WH003

[1053 rows x 8 columns]
```

We will now run the embedding model on the `combined` column.

Step 4: Running the GPT-3 embedding

We will now run the `davinci-similarity` model to obtain 12,288 dimensions for the `combined` column:

```
import time
import datetime
# start time
start = time.time()
```

```
def get_embedding(text, engine="davinci-similarity"):
   text = text.replace("\n", " ")
   return openai.Engine(id=engine).embeddings(input = [text])['data'][0]
['embedding']

df['davinci_similarity'] = df.combined.apply(lambda x: get_embedding(x,
engine='davinci-similarity'))

# end time
end = time.time()
etime=end-start
conversion = datetime.timedelta(seconds=etime)
print(conversion)
print(df)
```

The result is impressive. We have 12,288 dimensions for the combined column:

```
0:04:44.188250
                  Time  ...                                davinci_
similarity
Id                      ...
1     01/01/2016 06:30  ...  [-0.0047378824, 0.011997132, -0.017249448,
-0....
2     01/01/2016 06:30  ...  [-0.009643857, 0.0031537763, -0.012862709,
-0....
3     01/01/2016 06:30  ...  [-0.0077407444, 0.0035147679, -0.014401976,
-0...
4     01/01/2016 06:30  ...  [-0.007547746, 0.013380095, -0.018411927,
-0.0...
5     01/01/2016 06:30  ...  [-0.0047378824, 0.011997132, -0.017249448,
-0....
...                ...  ...
...
1049  01/01/2016 06:30  ...  [-0.0027823148, 0.013289047, -0.014368941,
-0....
1050  01/01/2016 06:30  ...  [-0.0071367626, 0.0046446105, -0.010336877,
0....
```

```
1051  01/01/2016 06:30  ...  [-0.0050991694, 0.006131069, -0.0138306245,
-0...
1052  01/01/2016 06:30  ...  [-0.0066779135, 0.014575769, -0.017257102,
-0....
1053  01/01/2016 06:30  ...  [-0.0027823148, 0.013289047, -0.014368941,
-0....

[1053 rows x 9 columns]
```

We now need to convert the result into a numpy matrix:

```
#creating a matrix
import numpy as np
matrix = np.vstack(df.davinci_similarity.values)
matrix.shape
```

The matrix has a shape of 1,053 records x 12,288 dimensions, which is quite impressive:

```
(1053, 12288)
```

The matrix is now ready to be sent to a scikit-learn machine learning clustering algorithm.

Step 5: Clustering (k-means clustering) with the embeddings

We usually send classical datasets to a k-means clustering algorithm. We will send a 12,288-dimension dataset to the ML algorithm, not to the next sublayer of the transformer.

We first import k-means from scikit-learn:

```
from sklearn.cluster import KMeans
```

We now run a classical k-means clustering algorithm with our 12,288-dimension dataset:

```
n_clusters = 4
kmeans = KMeans(n_clusters = n_clusters,init='k-means++',random_state=42)
kmeans.fit(matrix)
labels = kmeans.labels_
df['Cluster'] = labels
df.groupby('Cluster').Score.mean().sort_values()
```

The output is four clusters as requested:

```
Cluster
2    5.297794
0    5.323529
1    5.361345
3    5.741697
```

We can print the labels for the content of the dataset:

```
print(labels)
```

The output is:

```
[2 3 0 ... 0 1 2]
```

Let's now visualize the clusters using t-SNE.

Step 6: Visualizing the clusters (t-SNE)

t-SNE keeps local similarities. PCA maximizes large pairwise distances. In this case, smaller pairwise distances.

The notebook will use `matplotlib` to display t-SNE:

```
from sklearn.manifold import TSNE
import matplotlib
import matplotlib.pyplot as plt
```

Before visualizing we need to run the t-SNE algorithm:

```
#t-SNE
tsne = TSNE(n_components=2, perplexity=15, random_state=42, init='random',
learning_rate=200)
vis_dims2 = tsne.fit_transform(matrix)
```

We can now display the results in `matplotlib`:

```
x = [x for x,y in vis_dims2]
y = [y for x,y in vis_dims2]

for category, color in enumerate(['purple', 'green', 'red', 'blue']):
    xs = np.array(x)[df.Cluster==category]
    ys = np.array(y)[df.Cluster==category]
```

```
        plt.scatter(xs, ys, color=color, alpha=0.3)

        avg_x = xs.mean()
        avg_y = ys.mean()

        plt.scatter(avg_x, avg_y, marker='x', color=color, s=100)
    plt.title("Clusters of embeddings-t-SNE")
```

The plot shows the clusters with many data points piled up around them. There are also many data points circled around the clusters attached to the closest centroid:

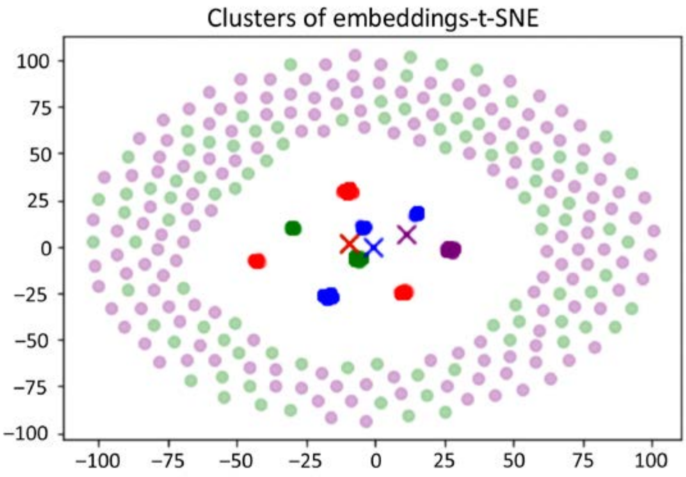


Figure 16.15: Clusters of embeddings-t-SNE

We ran a large GPT-3 model to embed 12,288 dimensions. Then we plugged the result into a clustering algorithm. The potential of combining transformers and machine learning is endless!

You can go to the `Peeking into the embeddings` section of the notebook if you wish to peek into the data frames.

Let's now have a look at the instruct series.

Instruct series

Personal assistants, avatars in metaverses, websites, and many other domains will increasingly need to provide clear instructions when a user asks for help. Go to the `instruct series` section of `Domain_Specific_GPT_3_Functionality.ipynb`.

In this section, we will ask a transformer to explain how to set up parent control in Microsoft Edge with the following prompt: `Explain how to set up parent control in Edge`.

We first run the completion cell:

```
import os
import openai
os.environ['OPENAI_API_KEY'] ='[YOUR_API_KEY]'
print(os.getenv('OPENAI_API_KEY'))
openai.api_key = os.getenv("OPENAI_API_KEY")

response = openai.Completion.create(
  engine="davinci-instruct-beta",
  prompt="Explain how to set up parent control in Edge.\n\n\nACTIONS:",
  temperature=0,
  max_tokens=120,
  top_p=1,
  frequency_penalty=0,
  presence_penalty=0
)
r = (response["choices"][0])
print(r["text"])
```

The response is a list of instructions as requested:

```
1. Start Internet Explorer.
2. Click on the tools menu.
3. Click on the Internet options.
4. Click on the advanced tab.
5. Click to clear or select the enable personalized favorite menu check
box.
```

The number of instructions you can ask for is unlimited! Use your creativity and imagination to find more examples!

Sometimes the input or the output is not acceptable. Let's see how to implement a content filter.

Content filter

Bias, unacceptable language, and any form of unethical input should be excluded from your AI applications.

One of OpenAI's trained models is a content filter. We will run an example in this section. Go to the `content filter` section of `Domain_Specific_GPT_3_Functionality.ipynb`.

My recommendation is to filter the input and the output, as shown in *Figure 16.16*:

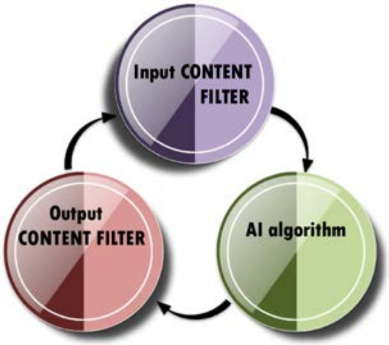


Figure 16.16: Implementing a content filter

My recommendation is to implement a three-step process:

1. Apply a content filter to ALL input data
2. Let the AI algorithm run as trained
3. Apply a content filter to ALL output data

In this section, the input and output data will be named `content`.

Take an obnoxious input as the following one:

```
content = "Small and fat children should not play basketball at school."
```

This input is unacceptable! School is not the NBA. Basketball should remain a nice exercise for *everyone*.

Let's now run the content filter in the cell - `content-filter-alpha`:

```

response = openai.Completion.create(
      en      prompt = "<|endoftext|>"+content+"\n--\nLabel:",
      temperature=0,
      max_tokens=1,
      top_p=1,
      frequency gine="content-filter-alpha",
_penalty=0,
      presence_penalty=0,
      logprobs=10
    )
```

The content filter stores the result in `response`, a dictionary object. We retrieve the value of choice to obtain the level of acceptability:

```
r = (response["choices"][0])
print("Content filter level:", r["text"])
```

The content filter sends one of three values back:

- `0` – Safe
- `1` – Sensitive
- `2` – Unsafe

In this case, the result is 2, of course:

```
Content filter level: 2
```

The content filter might not be sufficient. I recommend adding other algorithms to control and filter input/output content: rule bases, dictionaries, and other methods.

Now that we have explored domain-specific models, let's build a transformer-based recommender system.

Transformer-based recommender systems

Transformer models learn sequences. Learning language sequences is a great place to start considering the billions of messages posted on social media and cloud platforms each day. Consumer behaviors, images, and sounds can also be represented in sequences.

In this section, we will first create a general-purpose sequence graph and then build a general-purpose transformer-based recommender in Google Colaboratory. We will then see how to deploy them in metahumans.

Let's first define general-purpose sequences.

General-purpose sequences

Many activities can be represented by entities and links between them. They are thus organized in sequences. For example, a video on YouTube can be an entity A, and the link can be the behavior of a person going from video A to video E.

Another example is a bad fever being an entity F, and the link being the inference a doctor may make leading to a micro-decision B. The purchase of product D on Amazon by a consumer can generate a link to a suggestion C or another product. The examples are infinite!

We can define the entities in this section with six letters:

E={A,B,C,D,E,F}

When we speak a language, we follow grammar rules and cannot escape them.

For example, suppose *A=" I"*, *E=" eat"*, and *D=" candy"*. There is only one proper sequence to express the fact that I consume candy: "I eat candy."

If somebody says "eat candy I," it will sound slightly off.

In this sequence, the links representing those rules are:

A->E (I eat)

E->D(eat candy)

We can automatically infer rules in any domain by observing behaviors, learning datasets with ML, or manually listening to experts.

In this section, we will suppose that we have observed a YouTube user for several months who spends several hours watching videos. We have noticed that the user systematically goes from one type of video to another. For example, from the video of singer B to the video of singer D. The behavior rules *X* of this person *P* seem to be:

X(P)={AE,BD,BF,C,CD,DB,DC,DE,EA,ED,FB}

We can represent this system of entities as vertices in a graph and the links as edges. For example, if we apply *X(P)* to the vertices, we obtain the following undirected graph:

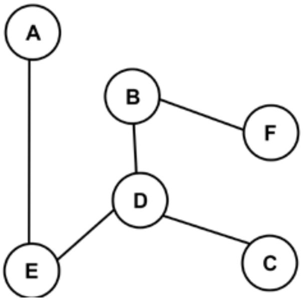


Figure 16.17: Graph of YouTube user's video combinations

Suppose the vertices are videos of a viewer's favorite singers and that *C* is the singer the viewer prefers. We can give a value of 1 to the statistical transitions (links or edges) the viewer made in the past weeks. We can also give a value of 100 to the viewer's favorite singer's videos.

For this viewer, the path is represented by the (edges, vertices) values *V(R(P)):*

$$V(X(P))=\{AE=1,BD=1,BF=1,C=100,CD=1,DB=1,DE=1,EA=1,ED=1,FB=1\}$$

The goal of a recommender is thus to suggest sequences that lead to videos of singer *C* or suggest *C* directly in some cases.

We can represent the undirected graph in reward matrix `R`:

```
                 A,B,C,D,E,F
R = ql.matrix([ [0,0,0,0,1,0],   A
                [0,0,0,1,0,1],   B
                [0,0,100,1,0,0], C
                [0,1,1,0,1,0],   D
                [1,0,0,1,0,0],   E
                [0,1,0,0,0,0]])  F
```

Let's use this reward matrix to simulate the activity of a viewer *X* over several months.

Dataset pipeline simulation with RL using an MDP

In this section, we will simulate the behavior *X* of a person *P* watching videos of songs on YouTube, which we define as *X(P)*. We will determine the values of the behavior of *P* as *V(X(P))*. We will then organize the values in a reward matrix *R* for a **Markov Decision Process** (**MDP**) that we will now implement in a Bellman equation.

Open `KantaiBERT_Recommender.ipynb`, which is in this chapter's folder in the book's GitHub repository. The notebook is a modification of `KantaiBERT.ipynb` described in *Chapter 4*, *Pretraining a RoBERTa Model from Scratch*.

In *Chapter 4*, we trained a transformer using `kant.txt`, which contained some of the works of Immanuel Kant. In this section, we will generate thousands of sequences of a person's behavior through **reinforcement learning** (**RL**). RL is not in the scope of this book, but this section contains some reminders.

The first step is to train a transformer model to learn and simulate a person's behavior.

Training customer behaviors with an MDP

`KantaiBERT.ipynb` in *Chapter 4* began by loading `kant.txt` to train a RoBERTa with a DistilBERT architecture. `kant.txt` contained works by Immanuel Kant. In this section, we will generate sequences using the reward matrix *R* defined in the *General-purpose sequences* section of this chapter:

```
R = ql.matrix([ [0,0,0,0,1,0],
                [0,0,0,1,0,1],
                [0,0,100,1,0,0],
                [0,1,1,0,1,0],
                [1,0,0,1,0,0],
                [0,1,0,0,0,0]])
```

The first cell of the program is thus:

```
Step 1A Training: Dataset Pipeline Simulation with RL using an MDP:
```

This cell implements an MDP using the Bellman equation:

```
# The Bellman MDP based Q function
Q[current_state, action] = R[current_state, action] + gamma * MaxValue
```

In this equation:

- `R` is the original reward matrix.
- `Q` is the updated matrix, which is the same size as `R`. However, it is updated through reinforcement learning to compute the relative value of the link (edge) between each entity (vertex).
- `gamma` is a learning rate set to `0.8` to avoid overfitting the training process.
- `MaxValue` is the maximum value of the next vertex. For example, if the viewer `P` of YouTube videos is viewing singer `A`, the program might increase the value of `E` so that this suggestion can appear as a recommendation.

Little by little, the program will try to find the best values to help a viewer find the best videos to watch. Once the reinforcement program has learned the best links (edges), it can recommend the best viewing sequences.

The original reward matrix has trained to become an operational matrix. If we add the original entities, the trained values clearly appear:

```
    A       B       C       D       E       F
[[  0.      0.      0.      0.    258.44    0.   ]  A
 [  0.      0.      0.    321.8     0.    207.752]  B
 [  0.      0.    500.    321.8     0.      0.   ]  C
 [  0.    258.44  401.      0.    258.44    0.   ]  D
 [207.752   0.      0.    321.8     0.      0.   ]  E
 [  0.    258.44    0.      0.      0.      0.   ]] F
```

The original sequences of values *V* of the behaviors *X* of person *P* were:

$$V(X(P))=\{AE=1,BD=1,BF=1,C=100,\ CD=1,DB=1,DE=1,EA=1,ED=1,FB=1\}$$

They have been trained to become:

$$V(X(P))=\{AE=259.44,\ BD=321.8\ ,BF=207.752,\ C=500,\ CD=321.8\ ,DB=258.44,\ DE=258.44,\\ EA=207.752,\ ED=321.8,\ FB=258.44\}$$

This is quite a change!

Now, it becomes possible to recommend a sequence of exciting videos of *P*'s preferred singers. Suppose *P* views a video of singer *E*. Line *E* of the trained matrix will recommend a video of the highest value of that line, which is *D=321.8*. Thus, a video of singer *D* will appear in the YouTube feed of person *P*.

The goal of this section is not to stop at this phase. Instead, this section uses an MDP to create meaningful sequences to create a dataset for the transformer to use for training.

YouTube does not need to generate sequences to create a dataset. YouTube stores all the behaviors of all viewers in big data. Then Google's powerful algorithms take over to recommend the best videos in the video feed of a viewer.

Other platforms use cosine similarity as implemented in *Chapter 9*, *Matching Tokenizers and Datasets*, to make predictions.

The MDP could have been trained for YouTube viewers, Amazon buyers, Google search results, a doctor's diagnosis path, a supply chain, and any type of sequence. Transformers are taking sequence learning, training, and predictions to another level.

Let's implement a simulation to create behavior sequences for a transformer model.

Simulating consumer behavior with an MDP

Once the RL part of the program is trained in cell 1, cell 2, `Step 1B Applying: Dataset Pipeline Simulation with MDP` will simulate a YouTube viewer's behavior over several months. It will also include similar viewer profiles adding up to a simulation of 10,000 sequences of video watching.

Cell 2 begins by creating the `kant.txt` file that will be used to train the KantaiBERT transformer model:

```
""" Simulating a decision-making process"""
f = open("kant.txt", "w")
```

Then the entities (vertices) are introduced:

```
conceptcode=["A","B","C","D","E","F"]
Now the number of sequences is set to 10,000:

maxv=10000
```

The function chooses a random start vertex named `origin`:

```
origin=ql.random.randint(0,6)
```

The program uses the trained matrix to select the best sequence for any domain from that point of origin. In this case, we suppose they are the favorite singers of a person such as the following examples:

```
FBDC EDC EDC DC BDC AEDC AEDC BDC BDC AEDC BDC AEDC EDC BDC AEDC DC AEDC
DC…/…
```

Once 10,000 sequences have been calculated, `kant.txt` contains a dataset for the transformer.

With `kant.txt`, the remaining cells of the program are the same as in `KantaiBERT.ipynb` described in *Chapter 4*, *Pretraining a RoBERTa Model from Scratch*.

The transformer is now ready to make recommendations.

Making recommendations

In *Chapter 4*, `KantaiBERT.ipynb` contained the following masked sequence:

```
fill_mask("Human thinking involves human<mask>.")
```

This sequence is specific and related to *Immanuel Kant*'s works. This notebook has a general-purpose dataset that can be used in any domain.

In this notebook, the input is:

```
fill_mask("BDC<mask>.")
```

The output contains duplicates. It will take a cleaning function to filter them to obtain two non-duplicate sequences:

```
[{'score': 0.00036507684853859246,
  'sequence': 'BDC FBDC.',
  'token': 265,
  'token_str': ' FBDC'},
 {'score': 0.00023987806343939155,
  'sequence': 'BDC DC.',
  'token': 271,
  'token_str': ' DC'}]
```

The sequences make sense. Sometimes a viewer will watch the same videos, sometimes not. The behavior can be chaotic. That's where machine learning comes in and how AI can be used in metahumans.

Metahuman recommenders

Once the sequences have been generated, they will be converted back into natural language for user interfaces. Metahuman, in this section, refers to a recommender that takes large amounts of features that:

- Exceed human capacity to reason with that many parameters
- Lead to more accurate predictions than a human can make

These pragmatic metahumans are not digital humans yet in this context but powerful computing tools. We will go through metahumans in the digital sense in the *Humans and AI copilots in metaverses* section.

For example, the BDC sequences could be a song by singer *B*, followed by singer *D*, and then *P*'s favorite singer *C*.

Once the sequence is converted into natural language, several options are possible:

- The sequence can be sent to a bot or a digital human.

> When an emerging technology appears, jump on the train and take the ride! You will get to know this technology and evolve with it. You can google other metahuman platforms. In any case, you can remain on the cutting edge by learning how to circumvent limits and find ways to use new technology.

- You can use the metahuman as an educational video while waiting for an API.
- A metahuman can be inserted into an interface as a voice message. For example, when using Google Maps in a car, you listen to the voice. It sounds like a human. We even slip and sometimes think it's a person, but it isn't. It's a machine.
- It can also be an invisible embedded suggestion in Amazon. It remains something that makes recommendations that lead us to make micro-decisions. It influences us as a salesperson would do. It's an invisible metahuman.

In this case, the general-purpose sequences were created by an MDP and trained by a RoBERTa transformer. This shows that transformers can be applied to any type of sequence.

Let's see how transformers are applied to computer vision.

Computer vision

This book is about NLP, not computer vision. However, in the previous section, we implemented general purpose sequences that can be applied to many domains. Computer vision is one of them.

The title of the article by *Dosovitskiy* et al. (2021) says it all: *An image is worth 16x16 words: Transformers for Image Recognition at Scale*. The authors processed an image as sequences. The results proved their point.

Google has made vision transformers available in a Colaboratory notebook. Open `Vision_Transformer_MLP_Mixer.ipynb` in the `Chapter16` directory of this book's GitHub repository.

Open `Vision_Transformer_MLP_Mixer.ipynb` contains a transformer computer vision model in `JAX()`. JAX combines Autograd and XLA. JAX can differentiate Python and NumPy functions. JAX speeds up Python and NumPy by using compilation techniques and parallelization.

The notebook is self-explanatory. You can explore it to see how it works. However, bear in mind that when Industry 4.0 reaches maturity and Industry 5.0 kicks in, the best implementations will be obtained by integrating your data on Cloud AI platforms. Local development will diminish, and companies will turn to Cloud AI without bearing local development, maintenance, and support.

The notebook's table of contents contains a transformer process we have gone through several times in this book. However, this time, it's simply applied to sequences of digital image information:

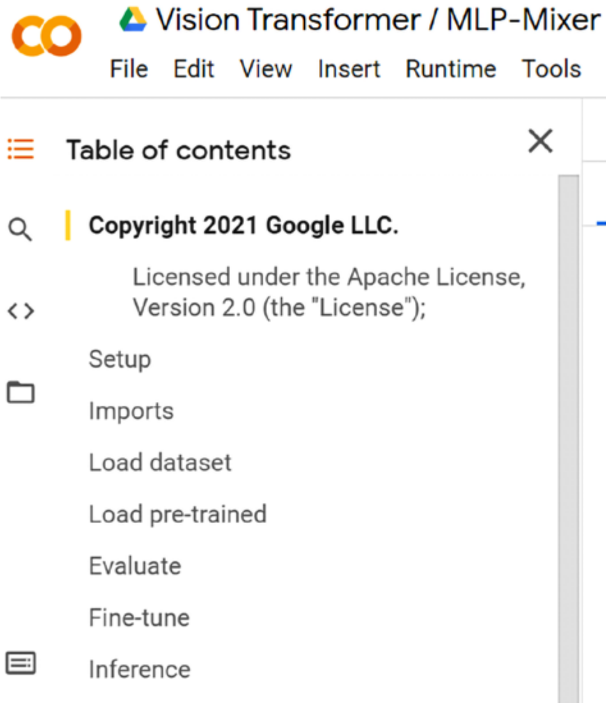

CO
Vision Transformer / MLP-Mixer
File Edit View Insert Runtime Tools
Table of contents
Copyright 2021 Google LLC.
Licensed under the Apache License, Version 2.0 (the "License");
Setup
Imports
Load dataset
Load pre-trained
Evaluate
Fine-tune
Inference

Figure 16.18: Our vision transformer notebook

The notebook follows standard deep learning methods. It shows some images with labels:

```
# Show some images with their labels.
images, labels = batch['image'][0][:9], batch['label'][0][:9]
titles = map(make_label_getter(dataset), labels.argmax(axis=1))
show_img_grid(images, titles)
```

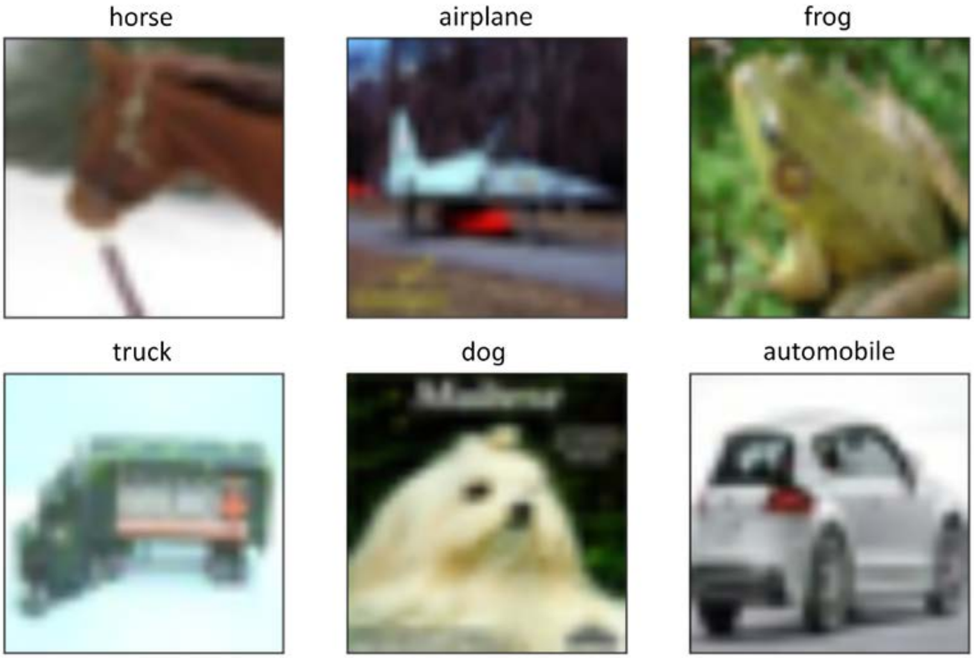


Figure 16.19: Images with labels

The images in this chapter are from *Learning Multiple Layers of Features from Tiny Images, Alex Krizhevsky*, 2009: `https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf`. They are part of the `CIFAR-10` and `CIFAR-100` datasets (`toronto.edu`): `https://www.cs.toronto.edu/~kriz/cifar.html`.

The notebook contains the standard transformer process and then displays the training images:

```
# Same as above, but with train images.
# Note how images are cropped/scaled differently.
# Check out input_pipeline.get_data() in the editor at your right to see
how the
# images are preprocessed differently.
batch = next(iter(ds_train.as_numpy_iterator()))
images, labels = batch['image'][0][:9], batch['label'][0][:9]
titles = map(make_label_getter(dataset), labels.argmax(axis=1))
show_img_grid(images, titles)
```

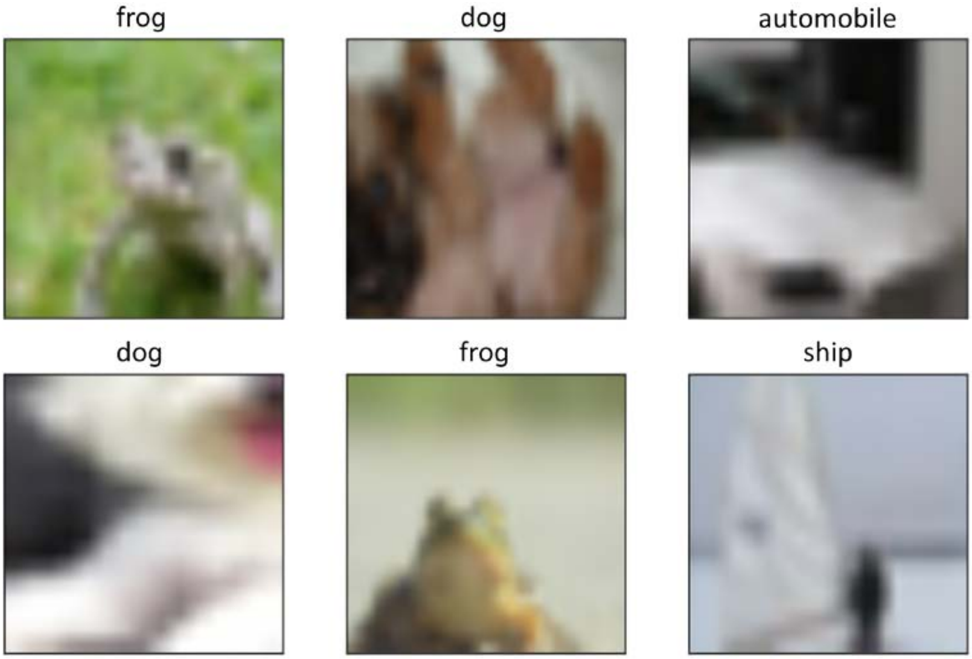


Figure 16.20: Trained data

Transformer programs can classify random pictures. It seems like a miracle to take a transformer model originally designed for NLP and use it for general-purpose sequences for recommenders and then for computer vision. However, we are just beginning to explore the generalization of sequences training.

The simplicity of the model is surprising! The vision transformer relies on the architecture of transformers. It does not contain the complexity of convolutional neural networks. Yet, it produces comparable results.

Now, robots and bots can be equipped with transformer models to understand language and interpret images to understand the world around them.

Vision transformers can be implemented in metahumans and metaverses.

Humans and AI copilots in metaverses

Humans and metahuman AI are merging into metaverses. Exploring metaverses is beyond the scope of this book. The toolbox provided by this book shows the path to metaverses populated by humans and metahuman AI.

Avatars, computer vision, and video game experience will make our communication with others *immersive*. We will go from looking at smartphones to being in locations with others.

From looking at to being in

The evolution from looking at to being in is a natural one. We invented computers, added screens, then invented smartphones, and now use apps for video meetings.

Now we can enter virtual reality for all types of meetings and activities.

We will use Facebook's metaverse, for example, on our smartphone to *feel* present in the same location as the people (personal and professional) we meet. Feeling *present* will no doubt be a major evolution in smartphone communication.

Feeling present somewhere is quite different from looking at a small screen on a mobile.

The metaverse will make the impossible possible: a spacewalk, surfing on huge waves, walking in a forest, visiting dinosaurs, and wherever our imagination takes us.

Yes, there are limits, dangers, threats, and everything that goes with human technology. However, we can use AI to control AI, as we saw with content filtering.

The transformer tools in this book added to the emerging metaverse technology will take us literally to another world.

Make good use of the knowledge and skills you acquired in this book to create your ethical future in a metaverse or the physical world!

Summary

This chapter described the rise of AI copilots with human-decision-making-level capability. Industry 4.0 has opened the door to machine interconnectivity. Machine-to-machine micro-decision making will speed up transactions. AI copilots will boost our productivity in a wide range of domains.

We saw how to use OpenAI Codex to generate source code while we code and even with natural language instructions.

We built a transformer-based recommender system using a dataset generated by the MDP program to train a RoBERTa transformer model. The dataset structure was a multi-purpose sequence model. A metahuman can thus acquire multi-domain recommender functionality.

The chapter then showed how a vision transformer could classify images processed as sequences of information.

Finally, we saw that the metaverse would make recommendations visible through a metahuman interface or invisible in deeply embedded functions in social media, for example.

Transformers have emerged with innovating copilots and models in an incredibly complex new era. The journey will prove both challenging and exciting!

Questions

1. AI copilots that can generate code automatically do not exist. (True/False)
2. AI copilots will never replace humans. (True/False)
3. GPT-3 engines can only do one task. (True/False)
4. Transformers can be trained to be recommenders. (True/False)
5. Transformers can only process language. (True/False)
6. A transformer sequence can only contain words. (True/False)
7. Vision transformers cannot equal CNNs. (True/False)
8. AI robots with computer vision do not exist. (True/False)
9. It is impossible to produce Python source code automatically. (True/False)
10. We might one day become the copilots of robots. (True/False)

References

- OpenAI platform for GPT-3: `https://openai.com`
- OpenAI models and engines: `https://beta.openai.com/docs/engines`
- Vision Transformers: *Alexey Dosovitskiy*, *Lucas Beyer*, *Alexander Kolesnikov*, *Dirk Weissenborn*, *Xiaohua Zhai*, *Thomas Unterthiner*, *Mostafa Dehghani*, *Matthias Minderer*, *Georg Heigold*, *Sylvain Gelly*, *Jakob Uszkoreit*, *Neil Houlsby*, 2020, *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*: `https://arxiv.org/abs/2010.11929`
- JAX for vision transformers: `https://github.com/google/jax`
- OpenAI, Visual Studio Copilot: `https://copilot.github.com/`
- Facebook metaverse: `https://www.facebook.com/Meta/videos/577658430179350`

- Markov Decision Process (MDP), examples and graph: *Denis Rothman*, 2020, *Artificial Intelligence by Example*, 2nd Edition: `https://www.amazon.com/Artificial-Intelligence-Example-advanced-learning/dp/1839211539/ref=sr_1_3?crid=238SF8FPU7BB0&keywords=denis+rothman&qid=1644008912&sprefix=denis+rothman%2Caps%2C143&sr=8-3`

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the authors:

`https://www.packt.link/Transformers`

Appendix I — Terminology of Transformer Models

The past decades have produced **Convolutional Neural Networks** (**CNNs**), **Recurrent Neural Networks** (**RNNs**), and more types of **Artificial Neural Networks** (**ANNs**). They all have a certain amount of vocabulary in common.

Transformer models introduced some new words and used existing words slightly differently. This appendix briefly describes transformer models to clarify the usage of deep learning vocabulary when applied to transformers.

The motivation of transformer model architecture relies upon an industrial approach to deep learning. The geometric nature of transformers boosts parallel processing. In addition, the architecture of transformers perfectly fits hardware optimization requirements. Google, for example, took advantage of the stack structure of transformers to design domain-specific optimized hardware that requires less floating-number precision.

Designing transformers models implies taking hardware into account. Therefore, the architecture of a transformer combines software and hardware optimization from the start.

This appendix defines some of the new usages of neural network language.

Stack

A *stack* contains identically sized layers that differ from classical deep learning models, as shown in *Figure I.1*. A stack runs from *bottom to top*. A stack can be an encoder or a decoder.

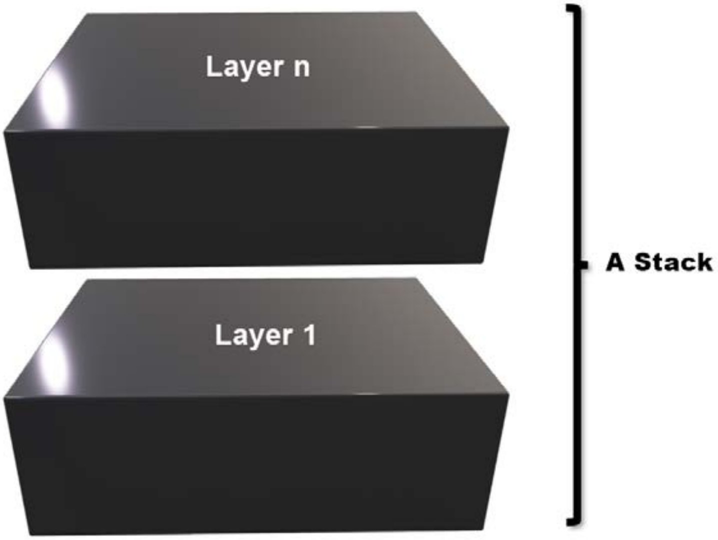


Figure I.1: Layers form a stack

Transformer stacks learn and see more as they rise in the stacks. Each layer transmits what it learned to the next layer just as our memory does.

Imagine that a *stack* is the Empire State Building in New York City. At the bottom, you cannot see much. But you will see more and farther as you ascend throught the offices on higher floors and look out the windows. Finally, at the top, you have a fantastic view of Manhattan!

Sublayer

Each layer contains sublayers, as shown in *Figure I.2*. Each sublayer of different layers has an identical structure, which boosts hardware optimization.

The original Transformer contains two sublayers that run from *bottom to top:*

- A self-attention sublayer, designed specifically for NLP and hardware optimization

- A classical feedforward network with some tweaking

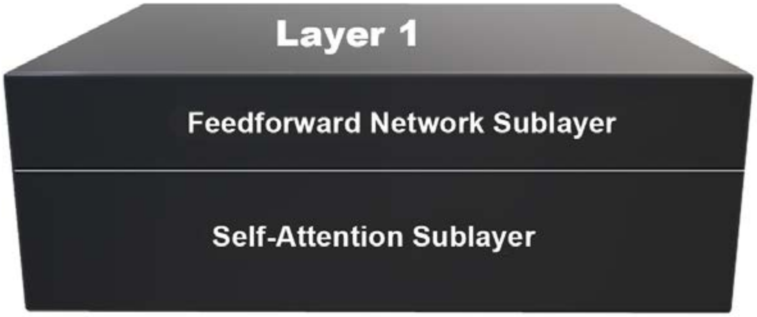


Figure I.2: A layer contains two sublayers

Attention heads

A self-attention sublayer is divided into n independent and identical layers called *heads*. For example, the original Transformer contains eight heads.

Figure I.3 represents heads as processors to show that transformers' industrialized structure fits hardware design:

Figure I.3: A self-attention sublayer contains heads

Note that the attention heads are represented by microprocessors in *Figure I.3* to stress the parallel processing power of transformer architectures.

Transformer architectures fit both NLP and hardware-optimization requirements.

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the authors:

```
https://www.packt.link/Transformers
```

Appendix II — Hardware Constraints for Transformer Models

Transformer models could not exist without optimized hardware. Memory and disk management design remain critical components. However, computing power remains a prerequisite. It would be nearly impossible to train the original Transformer described in *Chapter 2*, *Getting Started with the Architecture of the Transformer Model*, without GPUs. GPUs are at the center of the battle for efficient transformer models.

This appendix to *Chapter 3*, *Fine-Tuning BERT Models*, will take you through the importance of GPUs in three steps:

- The architecture and scale of transformers
- CPUs versus GPUs
- Implementing GPUs in PyTorch as an example of how any other optimized language optimizes

The Architecture and Scale of Transformers

A hint about hardware-driven design appears in the *The architecture of multi-head attention* section of *Chapter 2*, *Getting Started with the Architecture of the Transformer Model*:

"However, we would only get one point of view at a time by analyzing the sequence with one d_{model} block. Furthermore, it would take quite some calculation time to find other perspectives.

A better way is to divide the $d_{model} = 512$ dimensions of each word x_n of x (all the words of a sequence) into 8 $d_k = 64$ dimensions.

We then can run the 8 “heads” in parallel to speed up the training and obtain 8 different representation subspaces of how each word relates to another:

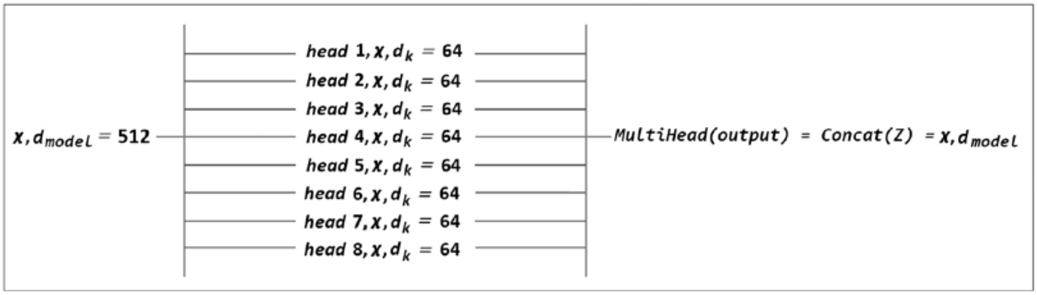


Figure II.1: Multi-head representations

You can see that there are now 8 heads running in parallel.

We can easily see the motivation for forcing the attention heads to learn 8 different perspectives. However, digging deeper into the motivations of the original 8 attention heads performing different calculations in parallel led us directly to hardware optimization.

Brown et al. (2020), in *Language Models are Few-Shot Learners*, `https://arxiv.org/abs/2005.14165`, describe how they designed GPT models. They confirm that transformer architectures are hardware-driven.

We partition the model across GPUs along with both the depth and width dimension to minimize data-transfer between nodes. The precise architectural parameters for each model are chosen based on computational efficiency and load-balancing in the layout of models across GPUs.

Transformers differ in their construction (encoders and decoders) and size. But they all have hardware constraints that require parallel processing. We need to take this a step further and see why GPUs are so special.

Why GPUs are so special

A clue to GPU-driven design emerges in the *The architecture of multi-head attention* section of *Chapter 2*, *Getting Started with the Architecture of the Transformer Model.*

Attention is defined as “Scaled Dot-Product Attention,” which is represented in the following equation into which we plug *Q*, *K*, and *V*:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

We can now conclude the following:

- Attention heads are designed for parallel computing
- Attention heads are based on *matmul*, matrix multiplication

GPUs are designed for parallel computing

A **CPU** (**central processing unit**) is optimized for *serial processing*. But if we run the attention heads through serial processing, it would take far longer to train an efficient transformer model. Very small educational transformers can run on CPUs. However, they do not qualify as state-of-the-art models.

A **GPU** (**graphics processing unit**) is designed for *parallel processing*. Transformer models were designed for *parallel processing (GPUs)*, not *serial processing (CPUs)*.

GPUs are also designed for matrix multiplication

NVIDIA GPUs, for example, contain tensor cores that accelerate matrix operations. A significant proportion of artificial intelligence algorithms use matrix operations, including transformer models. NVIDIA GPUs contain a goldmine of hardware optimization for matrix operations. The following links provide more information:

- `https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/`
- `https://www.nvidia.com/en-us/data-center/tesla-p100/`

Google's **Tensor Processing Unit** (**TPU**) is the equivalent of NVIDIA's GPUs. TensorFlow will optimize the use of tensors when using TPUs.

- For more on TPUs, see `https://cloud.google.com/tpu/docs/tpus`.
- For more on tensors in TensorFlow, see `https://www.tensorflow.org/guide/tensor`.

BERT$_{\text{BASE}}$ (110M parameters) was initially trained with 16 TPU chips. BERT$_{\text{LARGE}}$ (340M parameters) was trained with 64 TPU chips. For more on training BERT, see `https://arxiv.org/abs/1810.04805`.

We have established that the architecture of the transformer perfectly fits the constraints of parallel hardware. We still need to address the issue of implementing source code that runs on GPUs.

Implementing GPUs in code

PyTorch, among other languages and frameworks, manages GPUs. PyTorch contains tensors just as TensorFlow does. A tensor may look like NumPy np.arrays(). However, NumPy is not fit for parallel processing. Tensors use the parallel processing features of GPUs.

Tensors open the doors to distributed data over GPUs in PyTorch, among other frameworks: https://pytorch.org/tutorials/intermediate/ddp_tutorial.html

In the Chapter03 notebook, BERT_Fine_Tuning_Sentence_Classification_GPU.ipynb, we used **CUDA** (**Compute Unified Device Architecture**) to communicate with NVIDIA GPUs. CUDA is an NVIDIA platform for general computing on GPUs. Specific instructions can be added to our source code. For more, see https://developer.nvidia.com/cuda-zone.

In the Chapter03 notebook, we used CUDA instructions to transfer our model and data to NVIDIA GPUs. PyTorch has an instruction to specify the device we wish to use: torch.device.

For more, see https://pytorch.org/docs/stable/notes/cuda.html.

We will explain device to illustrate the implementation of GPUs in PyTorch and programs in general. Let's focus on selecting a device, data parallelism, loading a model to a device, and adding batch data to the device. Each bullet point contains the way device is used and the cell number in BERT_Fine_Tuning_Sentence_Classification_GPU.ipynb:

- **Select device (Cell 3)**

  The program checks to see if CUDA is available on an NVIDIA GPU. If not, the device will be CPU:

  ```
  device = torch.device("cuda" if torch.cuda.is_available() else
  "cpu")
  !nvidia-smi
  ```

- **Data parallelism (Cell 16)**

  The model can be distributed for parallel computing over several GPUs if more than one GPU is available:

  ```
  model = BertForSequenceClassification.from_pretrained("bert-base-
  uncased", num_labels=2)
  model = nn.DataParallel(model)
  ```

- **Loading the model to the device (cell 16)**

  The model is sent to the device:

  ```
      model.to(device)
  ```

- **Add batch to device (cell 20) for training and validation data**

  Batches of data are added to the GPUs available (1 to `n`):

  ```
  # Add batch to GPU
  batch = tuple(t.to(device) for t in batch)
  ```

In the following section, I describe tests I made to illustrate the use of GPUs for transformer models by running a notebook of the chapter with three runtime configurations.

Testing GPUs with Google Colab

In this section, I describe informal tests I ran to illustrate the potential of GPUs. We'll use the same `Chapter03` notebook: `BERT_Fine_Tuning_Sentence_Classification_GPU.ipynb`.

I ran the notebook on three scenarios:

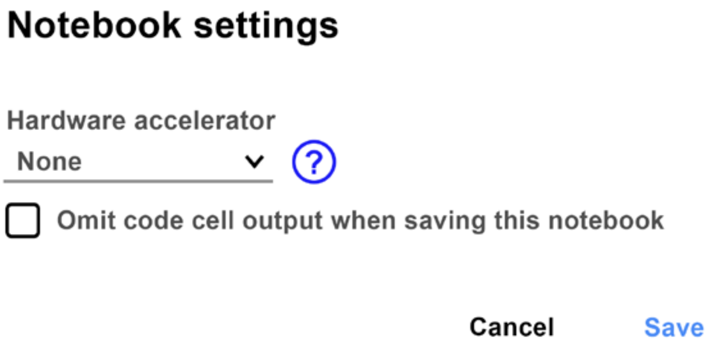
- Google Colab Free with a CPU
- Google Colab Free with a GPU
- Google Colab Pro

Google Colab Free with a CPU

It is nearly impossible to fine-tune or train a transformer model with millions or billions of parameters on a CPU. CPUs are mostly sequential. Transformer models are designed for parallel processing.

In the **Runtime** menu and **Change Runtime Type** submenu, you can select a hardware accelerator: **None (CPU)**, **GPU**, or **TPU**.

This test was run with **None (CPU)**, as shown in *Figure II.2*:



Notebook settings

Hardware accelerator

None

□ Omit code cell output when saving this notebook

Figure II.2: Selecting a hardware accelerator

When the notebook reaches the training loop, it slows down right from the start:

```
Epoch :   0%|          | 0/4 [00:00<?, ?it/s]
```

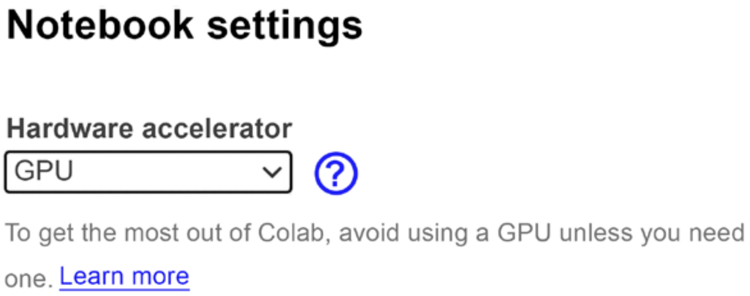
Figure II.3: Training loop

After 15 minutes, nothing has really happened.

CPUs are not designed for parallel processing. Transformer models are designed for parallel processing, so part from toy models, they require GPUs.

Google Colab Free with a GPU

Let's go back to the notebook settings to select a **GPU**.



Notebook settings

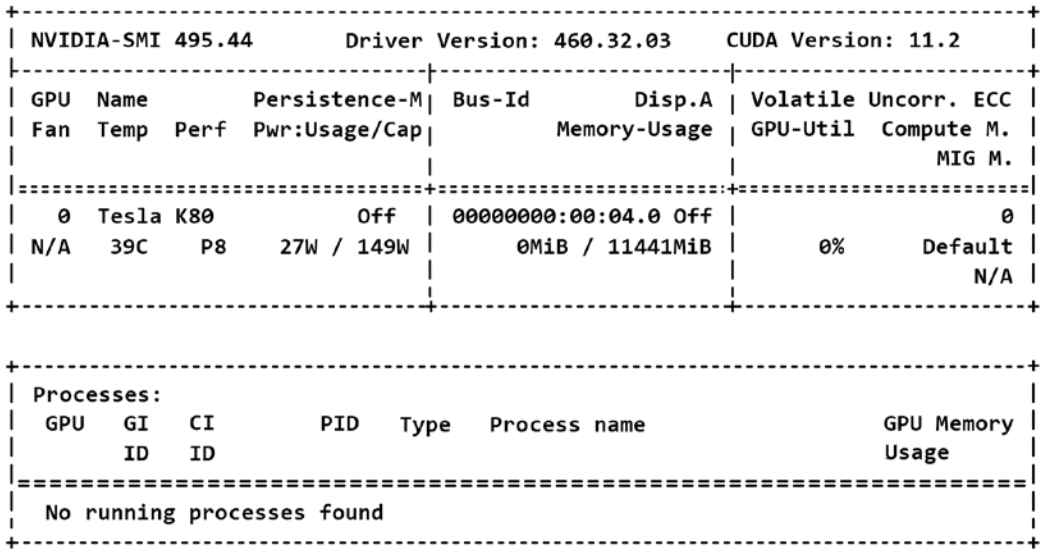
Hardware accelerator

GPU

To get the most out of Colab, avoid using a GPU unless you need one. Learn more

Figure II.4 Selecting a GPU

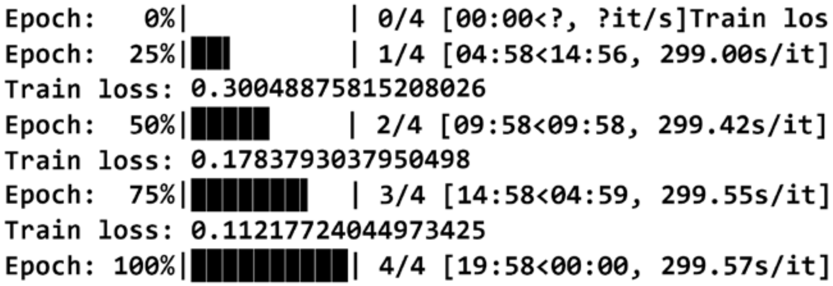
At the time of writing, I tested Google Colab, and an NVIDIA K80 was attributed to the VM with CUDA 11.2:



```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 495.44       Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla K80           Off  | 00000000:00:04.0 Off |                    0 |
| N/A   39C    P8    27W / 149W |      0MiB / 11441MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Figure II.5: NVIDIA K80 GPU activated

The training loop advanced normally and lasted about 20 minutes. However, Google Colab VMs, at the time of these tests (November 2021), do not provide more than one GPU. GPUs are expensive. In any case, *Figure II.6*, shows that the training loop was performed in a reasonable time:



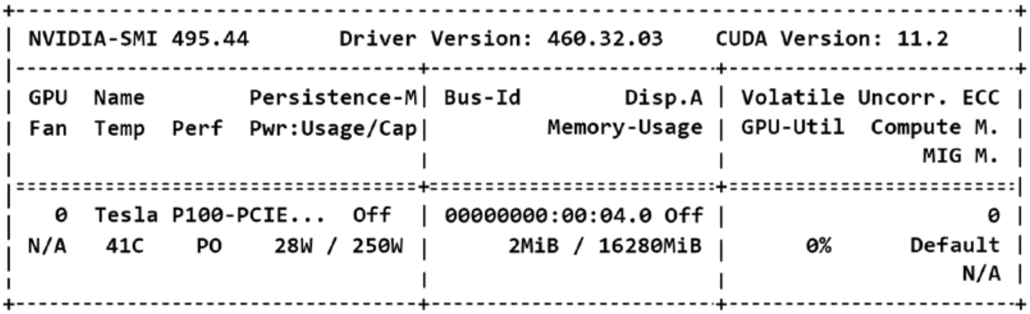
```
Epoch:   0%|          | 0/4 [00:00<?, ?it/s]Train los
Epoch:  25%|██▌       | 1/4 [04:58<14:56, 299.00s/it]
Train loss: 0.30048875815208026
Epoch:  50%|█████     | 2/4 [09:58<09:58, 299.42s/it]
Train loss: 0.1783793037950498
Epoch:  75%|███████▌  | 3/4 [14:58<04:59, 299.55s/it]
Train loss: 0.11217724044973425
Epoch: 100%|██████████| 4/4 [19:58<00:00, 299.57s/it]
```

Figure II.6: Training loop with a K80 GPU

I found it interesting to see whether Google Colab Pro provides faster GPUs.

Google Colab Pro with a GPU

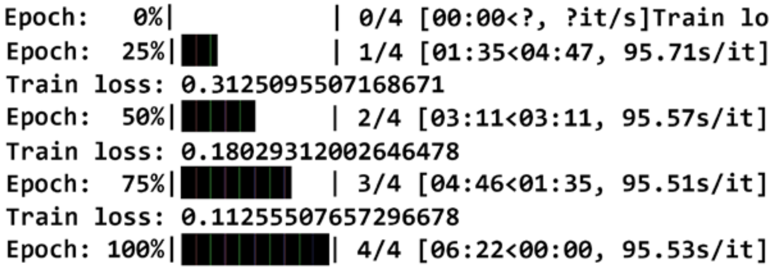
The VM activated with Google Colab provided an NVIDIA P100 GPU, as shown in *Figure II.7*. That was interesting because the original Transformer was trained with 8 NVIDIA P100s as stated in *Vaswani* et al.(2017), *Attention is All you Need.* It took 12 hours to train the base models with 10^6×65 parameters and with 8 GPUs:



```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 495.44       Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   41C    P0    28W / 250W |      2MiB / 16280MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

Figure II.7: The Google Colab Pro VM was provided with a P100 GPU

The training loop time was considerably reduced and lasted less than 10 minutes, as shown in *Figure II.8*:



```
Epoch:   0%|          | 0/4 [00:00<?, ?it/s]Train lo
Epoch:  25%|██▌       | 1/4 [01:35<04:47, 95.71s/it]
Train loss: 0.3125095507168671
Epoch:  50%|█████     | 2/4 [03:11<03:11, 95.57s/it]
Train loss: 0.18029312002646478
Epoch:  75%|███████▌  | 3/4 [04:46<01:35, 95.51s/it]
Train loss: 0.11255507657296678
Epoch: 100%|██████████| 4/4 [06:22<00:00, 95.53s/it]
```

Figure II.8: Training loop with a P100 GPU

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the authors:

`https://www.packt.link/Transformers`

Appendix III — Generic Text Completion with GPT-2

This appendix is the detailed explanation of the *Generic text completion with GPT-2* section in *Chapter 7*, *The Rise of Suprahuman Transformers with GPT-3 Engines*. This section describes how to implement a GPT-2 transformer model for generic text complexion.

You can read the usage of this notebook directly in *Chapter 7* or build the program and run it in this appendix to get more profound knowledge of how a GPT model works.

We will clone the `OpenAI_GPT_2` repository, download the 345M-parameter GPT-2 transformer model, and interact with it. We will enter context sentences and analyze the text generated by the transformer. The goal is to see how it creates new content.

This section is divided into nine steps. Open `OpenAI_GPT_2.ipynb` in Google Colaboratory. The notebook is in the `AppendixIII` directory of the GitHub repository of this book. You will notice that the notebook is also divided into the same nine steps and cells as this section.

Run the notebook cell by cell. The process is tedious, but *the result produced by the cloned OpenAI GPT-2 repository is gratifying*. We saw that we could run a GPT-3 engine in a few lines. But this appendix gives you the opportunity, even if the code is not optimized anymore, to see how GPT-2 models work.

Hugging Face has a wrapper that encapsulates GPT-2 models. It's useful as an alternative to the OpenAI API. However, the goal in this appendix is *not* to avoid the complexity of the underlying components of a GPT-2 model but to explore them!

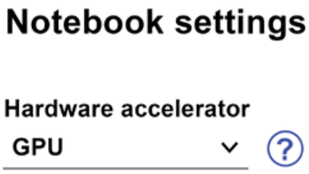
Finally, It is important to stress that we are running a low-level GPT-2 model and not a one-line call to obtain a result. That is why we are avoiding pre-packaged versions (the OpenAI GPT-3 API, Hugging Face wrappers, others). We are getting our hands dirty to understand the architecture of GPT-2 from scratch. As a result, you might get some deprecation messages. However, the effort is worthwhile to become an Industry 4.0 AI guru.

Let's begin by activating the GPU.

Step 1: Activating the GPU

We must activate the GPU to train our GPT-2 345M-parameter transformer model.

To activate the GPU, go to the **Runtime** menu in **Notebook settings** to get the most out of the VM:



Notebook settings

Hardware accelerator

GPU

Figure III.1: The GPU hardware accelerator

We can see that activating the GPU is a prerequisite for better performance that will give us access to the world of GPT transformers. So let's now clone the OpenAI GPT-2 repository.

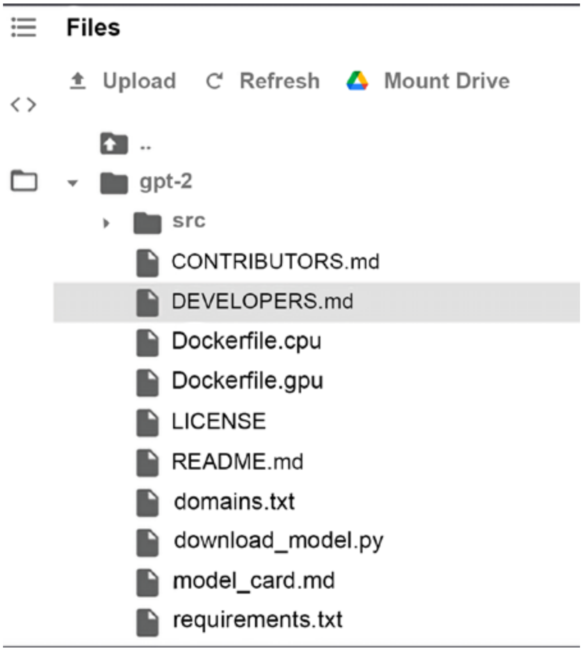
Step 2: Cloning the OpenAI GPT-2 repository

OpenAI still lets us download GPT-2 for now. This may be discontinued in the future, or maybe we will get access to more resources. At this point, the evolution of transformers and their usage moves so fast that nobody can foresee how the market will evolve, even the major research labs themselves.

We will clone OpenAI's GitHub directory on our VM:

```
#@title Step 2: Cloning the OpenAI GPT-2 Repository
!git clone https://github.com/openai/gpt-2.git
```

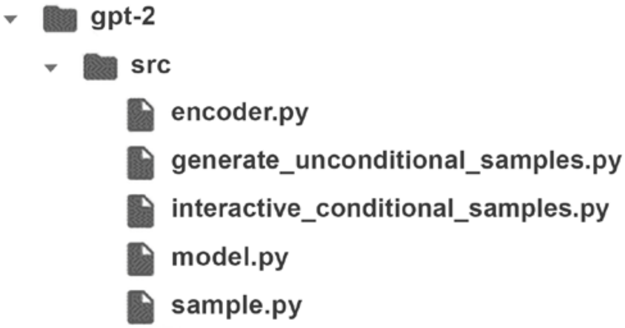
When the cloning is over, you should see the repository appear in the file manager:



```
..
gpt-2
  src
  CONTRIBUTORS.md
  DEVELOPERS.md
  Dockerfile.cpu
  Dockerfile.gpu
  LICENSE
  README.md
  domains.txt
  download_model.py
  model_card.md
  requirements.txt
```

Figure III.2: Cloned GPT-2 repository

Click on **src**, and you will see that the Python files we need from OpenAI to run our model are installed:



```
gpt-2
  src
    encoder.py
    generate_unconditional_samples.py
    interactive_conditional_samples.py
    model.py
    sample.py
```

Figure III.3: The GPT-2 Python files to run a model

You can see that we do not have the Python training files we need. We will install them when we train the GPT-2 model in the *Training a GPT-2 language model* section of *Appendix IV*, *Custom Text Completion with GPT-2*.

Let's now install the requirements.

Step 3: Installing the requirements

The requirements will be installed automatically:

```
#@title Step 3: Installing the requirements
import os          # when the VM restarts import os necessary
os.chdir("/content/gpt-2")
!pip3 install -r requirements.txt
```

When running cell by cell, we might have to restart the VM and thus import os again.

The requirements for this notebook are:

- `Fire 0.1.3` to generate **command-line interfaces (CLIs)**
- `regex 2017.4.5` for regex usage
- `Requests 2.21.0`, an HTTP library
- `tqdm 4.31.1` to display a progress meter for loops

You may be asked to restart the notebook.

Do not restart it now. Let's wait until we check the version of TensorFlow.

Step 4: Checking the version of TensorFlow

The GPT-2 345M transformer model provided by OpenAI uses TensorFlow 1.x. This will lead to several warnings when running the program. However, we will ignore them and run at full speed on the thin ice of training GPT models ourselves with our modest machines.

In the 2020s, GPT models have reached 175 billion parameters, making it impossible for us to train them ourselves efficiently without having access to a supercomputer. The number of parameters will only continue to increase.

The corporate giants' research labs, such as Facebook AI and OpenAI, and Google Research/Brain, are speeding toward super-transformers and are leaving what they can for us to learn and understand. But, unfortunately, they do not have time to go back and update all the models they share. However, we still have this notebook!

TensorFlow 2.x is the latest TensorFlow version. However, older programs can still be helpful. This is one reason why Google Colaboratory VMs have preinstalled versions of both TensorFlow 1.x and TensorFlow 2.x.

We will be using TensorFlow 1.x in this notebook:

```
#@title Step 4: Checking the Version of TensorFlow
#Colab has tf 1.x and tf 2.x installed
#Restart runtime using 'Runtime' -> 'Restart runtime...'
%tensorflow_version 1.x
import tensorflow as tf
print(tf.__version__)
```

The output should be:

```
TensorFlow 1.x selected.
1.15.2
```

Whether the `tf 1.x` version is displayed or not, rerun the cell to make sure, and then restart the VM. *Rerun this cell to make sure before continuing*.

> If you encounter a TensforFlow error during the process (ignore the warnings), rerun this cell, restart the VM, and rerun to make sure.

Do this every time you restart the VM. The default version of the VM is `tf.2`.

We are now ready to download the GPT-2 model.

Step 5: Downloading the 345M-parameter GPT-2 model

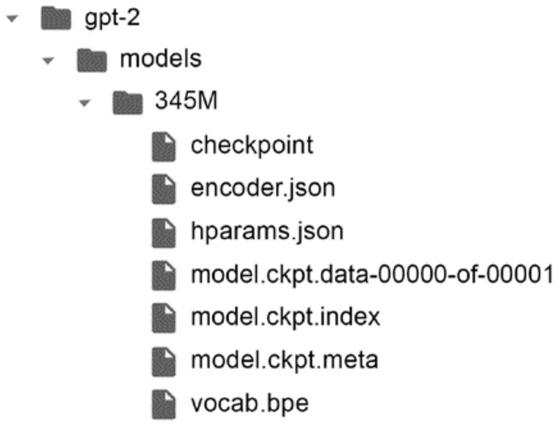
We will now download the trained 345M-parameter GPT-2 model:

```
#@title Step 5: Downloading the 345M parameter GPT-2 Model
# run code and send argument
import os # after runtime is restarted
os.chdir("/content/gpt-2")
!python3 download_model.py '345M'
```

The path to the model directory is:

`/content/gpt-2/models/345M`

It contains the information we need to run the model:



gpt-2
models
345M
checkpoint
encoder.json
hparams.json
model.ckpt.data-00000-of-00001
model.ckpt.index
model.ckpt.meta
vocab.bpe

Figure III.4: The GPT-2 Python files of the 345M-parameter model

The `hparams.json` file contains the definition of the GPT-2 model:

- `"n_vocab"`: `50257`, the size of the vocabulary of the model
- `"n_ctx"`: `1024`, the context size
- `"n_embd"`: `1024`, the embedding size
- `"n_head"`: `16`, the number of heads
- `"n_layer"`: `24`, the number of layers

`encoder.json` and `vacab.bpe` contain the tokenized vocabulary and the BPE word pairs. If necessary, take a few minutes to go back and read the *Step 3: Training a tokenizer* subsection in *Chapter 4*, *Pretraining a RoBERTa Model from Scratch*.

The `checkpoint` file contains the trained parameters at a checkpoint. For example, it could contain the trained parameters for 1,000 steps, as we will do in the *Step 9: Training a GPT-2 model* section of *Appendix IV*, *Custom Text Completion with GPT-2*.

The `checkpoint` file is saved with three other important files:

- `model.ckpt.meta` describes the graph structure of the model. It contains `GraphDef`, `SaverDef`, and so on. We can retrieve the information with `tf.train.import_meta_graph([path]+'model.ckpt.meta')`.

- `model.ckpt.index` is a string table. The keys contain the name of a tensor, and the value is `BundleEntryProto`, which contains the metadata of a tensor.
- `model.ckpt.data` contains the values of all the variables in a *TensorBundle collection*.

We have downloaded our model. We will now go through some intermediate steps before activating the model.

Steps 6-7: Intermediate instructions

In this section, we will go through *Steps 6*, *7*, and *7a*, which are intermediate steps leading to *Step 8*, in which we will define and activate the model.

We want to print UTF-encoded text to the console when we are interacting with the model:

```
#@title Step 6: Printing UTF encoded text to the console
!export PYTHONIOENCODING=UTF-8
```

We want to make sure we are in the `src` directory:

```
#@title Step 7: Project Source Code
import os # import after runtime is restarted
os.chdir("/content/gpt-2/src")
```

We are ready to interact with the GPT-2 model. We could run it directly with a command, as we will do in the *Training a GPT-2 language model* section of *Appendix IV*, *Custom Text Completion with GPT-2*. However, in this section, we will go through the main aspects of the code.

`interactive_conditional_samples.py` first imports the necessary modules required to interact with the model:

```
#@title Step 7a: Interactive Conditional Samples (src)
#Project Source Code for Interactive Conditional Samples:
# /content/gpt-2/src/interactive_conditional_samples.py file
import json
import os
import numpy as np
import tensorflow as tf
```

We have gone through the intermediate steps leading to the activation of the model.

Steps 7b-8: Importing and defining the model

We will now activate the interaction with the model with `interactive_conditional_samples.py`.

We need to import three modules that are also in `/content/gpt-2/src`:

```
import model, sample, encoder
```

The three programs are:

- `model.py` defines the model's structure: the hyperparameters, the multi-attention `tf.matmul` operations, the activation functions, and all the other properties.
- `sample.py` processes the interaction and controls the sample that will be generated. It makes sure that the tokens are more meaningful.

  Softmax values can sometimes be blurry, like looking at an image in low definition. `sample.py` contains a variable named `temperature` that will make the values sharper, increasing the higher probabilities and softening the lower ones.

  `sample.py` can activate Top-*k* sampling. Top-*k* sampling sorts the probability distribution of a predicted sequence. The higher probability values of the head of the distribution are filtered up to the k-*th* token. The tail containing the lower probabilities is excluded, preventing the model from predicting low-quality tokens.

  `sample.py` can also activate Top-*p* sampling for language modeling. Top-*p* sampling does not sort the probability distribution. Instead, it selects the words with high probabilities until the sum of this subset's probabilities or the nucleus of a possible sequence exceeds *p*.
- `encoder.py` encodes the sample sequence with the defined model, `encoder.json`, and `vocab.bpe`. It contains both a BPE encoder and a text decoder.

You can open these programs to explore them further by double-clicking on them.

`interactive_conditional_samples.py` will call the functions required to interact with the model to initialize the following information: the hyperparameters that define the model from `model.py`, and the sample sequence parameters from `sample.py`. It will encode and decode sequences with `encode.py`.

`interactive_conditional_samples.py` will then restore the checkpoint data defined in the *Step 5: Downloading the 345M-parameter GPT-2 model* subsection of this section.

You can explore `interactive_conditional_samples.py` by double-clicking on it and experiment with its parameters:

- `model_name` is the model name, such as `"124M"` or `"345M,"` and relies on `models_dir`.
- `models_dir` defines the directory containing the models.
- `seed` sets a random integer for random generators. The seed can be set to reproduce results.
- `nsamples` is the number of samples to return. If it is set to `0`, it will continue to generate samples until you double-click on the *run* button of the cell or press *Ctrl* + *M*.
- `batch_size` determines the size of a batch and has an impact on memory and speed.
- `length` is the number of tokens of generated text. If set to `none`, it relies on the hyperparameters of the model.
- `temperature` determines the level of Boltzmann distributions. If the temperature is high, the completions will be more random. If the temperature is low, the results will become more deterministic.
- `top_k` controls the number of tokens taken into consideration by Top-*k* at each step. `0` means no restrictions. `40` is the recommended value.
- `top_p` controls Top-*p*.

For the program in this section, the scenario of the parameters we just explored will be:

- `model_name = "345M"`
- `seed = None`
- `nsamples = 1`
- `batch_size = 1`
- `length = 300`
- `temperature = 1`
- `top_k = 0`
- `models_dir = '/content/gpt-2/models'`

These parameters will influence the model's behavior, the way it is conditioned by the context input, and generate text completion sequences. First, run the notebook with the default values. You can then change the code's parameters by double-clicking on the program, editing it, and saving it. The changes will be deleted at each restart of the VM. Save the program and reload it if you wish to create interaction scenarios.

The program is now ready to prompt us to interact with it.

Step 9: Interacting with GPT-2

In this section, we will interact with the GPT-2 345M model.

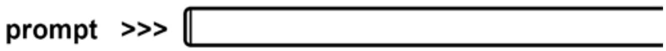
There will be more messages when the system runs, but as long as Google Colaboratory maintains `tf 1.x`, we will run the model with this notebook. One day, we might have to use GPT-3 engines if this notebook becomes obsolete, or we will have to use Hugging Face GPT-2 wrappers, for example, which might be deprecated as well in the future.

In the meantime, GPT-2 is still in use so let's interact with the model!

To interact with the model, run the `interact_model` cell:

```
#@title Step 9: Interacting with GPT-2
interact_model('345M',None,1,1,300,1,0,'/content/gpt-2/models')
```

You will be prompted to enter some context:



prompt >>>

Figure III.5: Context input for text completion

You can try any type of context you wish since this is a standard GPT-2 model.

We can try a sentence written by Emmanuel Kant:

```
Human reason, in one sphere of its cognition, is called upon to
consider questions, which it cannot decline, as they are presented by
its own nature, but which it cannot answer, as they transcend every
faculty of the mind.
```

Press *Enter* to generate text. The output will be relatively random since the GPT-2 model was not trained on our dataset, and we are running a stochastic model anyway.

Let's have a look at the first few lines the GPT model generated at the time I ran it:

```
"We may grant to this conception the peculiarity that it is the only
causal logic.
In the second law of logic as in the third, experience is measured at its
end: apprehension is afterwards closed in consciousness.
The solution of scholastic perplexities, whether moral or religious, is
not only impossible, but your own existence is blasphemous."
```

To stop the cell, double-click on the run button of the cell.

You can also press *Ctrl* + *M* to stop generating text, but it may transform the code into text, and you will have to copy it back into a program cell.

The output is rich. We can observe several facts:

- The context we entered *conditioned* the output generated by the model.
- The context was a demonstration of the model. It learned what to say from the model without modifying its parameters.
- Text completion is conditioned by context. This opens the door to transformer models that do not require fine-tuning.
- From a semantic perspective, the output could be more interesting.
- From a grammatical perspective, the output is convincing.

You can see if we could obtain more impressive results by training the model on a customized dataset in *Appendix IV*, *Custom Text Completion with GPT-2*.

References

- *OpenAI GPT-2* GitHub repository: `https://github.com/openai/gpt-2`
- *N Shepperd's* GitHub repository: `https://github.com/nshepperd/gpt-2`

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the authors:

`https://www.packt.link/Transformers`

Appendix IV — Custom Text Completion with GPT-2

This appendix, relating to *Chapter 7*, *The Rise of Suprahuman Transformers with GPT-3 Engines*, describes how to customize text completion with a GPT-2 model.

This appendix shows how to build a GPT-2 model, train it, and interact with custom text in 12 steps.

Open `Training_OpenAI_GPT_2.ipynb`, which is in the GitHub repository of this appendix. You will notice that the notebook is also divided into the same 12 steps and cells as this appendix.

Run the notebook cell by cell. The process is tedious, but *the result produced by the cloned OpenAI GPT-2 repository is gratifying*. We are not using the GPT-3 API or a Hugging Face wrapper.

We are getting our hands dirty to see how the model is built and trained. You will see some deprecation messages, but we need to get inside the model, not the wrappers or the API. However, the effort is worthwhile.

Let's begin by activating the GPU.

Training a GPT-2 language model

In this section, we will train a GPT-2 model on a custom dataset that we will encode. We will then interact with our customized model. We will be using the same `kant.txt` dataset as in *Chapter 4*, *Pretraining a RoBERTa Model from Scratch*.

We will open the notebook and run it cell by cell.

Step 1: Prerequisites

The files referred to in this section are available in the `AppendixIV` directory of this book's GitHub repository:

- Activate the GPU in the Google Colab's notebook runtime menu if you are running it on Google Colab, as explained in *Step 1: Activating the GPU* in *Appendix III*, *Generic Text Completion with GPT-2*.

- Upload the following Python files to Google Colaboratory with the built-in file manager: `train.py`, `load_dataset.py`, `encode.py`, `accumulate.py`, `memory_saving_gradients.py`.
- These files originally come from *N Shepperd's* GitHub repository: `https://github.com/nshepperd/gpt-2`. However, you can download these files from the `AppendixIV\gpt-2-train_files` directory in this book's GitHub repository.
- The *N Shepperd's* GitHub repository provides the necessary files to train our GPT-2 model. We will not clone *N Shepperd's* repository. Instead, we will be cloning OpenAI's repository and adding the five training files we need from *N Shepperd's* repository.
- Upload `dset.txt` to Google Colaboratory with the built-in file manager. The dataset is named `dset.txt` so that you can replace its content without modifying the program with your customized inputs after reading this appendix.
- This dataset is in the `gpt-2-train_files` directory in the GitHub repository of this appendix. It is the `kant.txt` dataset used in *Chapter 4*, *Pretraining a RoBERTa Model from Scratch*.

We will now go through the initial steps of the training process.

Steps 2 to 6: Initial steps of the training process

This subsection will only briefly go through *Steps 2 to 6* since we described them in detail in *Appendix III*, *Generic Text Completion with GPT-2*. We will then copy the dataset and the model to the project directory.

The program now clones OpenAI's GPT-2 repository and not *N Shepperd's* repository:

```
#@title Step 2: Cloning the OpenAI GPT-2 Repository
#!git clone https://github.com/nshepperd/gpt-2.git
!git clone https://github.com/openai/gpt-2.git
```

We have already uploaded the files we need to train the GPT-2 model from *N Shepperd's* directory.

The program now installs the requirements:

```
#@title Step 3: Installing the requirements
import os              #when the VM restarts import os necessary
os.chdir("/content/gpt-2")
!pip3 install -r requirements.txt
```

This notebook requires `toposort`, which is a topological sort algorithm:

```
!pip install toposort
```

Do not restart the notebook after installing the requirements. Instead, wait until you have checked the TensorFlow version to restart the VM only once during your session. After that, restart it if necessary. It is tedious but worthwhile to get inside the code beyond just wrappers and APIs.

We now check the TensorFlow version to make sure we are running version `tf 1.x`:

```
#@title Step 4: Checking TensorFlow version
#Colab has tf 1.x , and tf 2.x installed
#Restart runtime using 'Runtime' -> 'Restart runtime...'
%tensorflow_version 1.x
import tensorflow as tf
print(tf.__version__)
```

Whether the `tf 1.x` version is displayed or not, rerun the cell to make sure, restart the VM, and rerun this cell. That way, you are sure you are running the VM with `tf 1.x`.

The program now downloads the 117M parameter GPT-2 model we will train with our dataset:

```
#@title Step 5: Downloading 117M parameter GPT-2 Model
# run code and send argument
import os # after runtime is restarted
os.chdir("/content/gpt-2")
!python3 download_model.py '117M' #creates model directory
```

We will copy the dataset and the 117M parameter GPT-2 model into the `src` directory:

```
#@title Step 6: Copying the Project Resources to src
!cp /content/dset.txt /content/gpt-2/src/
!cp -r /content/gpt-2/models/ /content/gpt-2/src/
```

The goal is to group all the resources we need to train the model in the `src` project directory.

We will now go through the N Shepperd training files.

Step 7: The N Shepperd training files

The training files we will use come from *N Shepperd* ‘s GitHub repository. We uploaded them in *Step 1: Prerequisites* of this appendix. We will now copy them into our project directory:

```
#@title Step 7: Copying the N Shepperd Training Files
```

```
#Referfence GitHub repository: https://github.com/nshepperd/gpt-2
import os # import after runtime is restarted
!cp /content/train.py /content/gpt-2/src/
!cp /content/load_dataset.py /content/gpt-2/src/
!cp /content/encode.py /content/gpt-2/src/
!cp /content/accumulate.py /content/gpt-2/src/
!cp /content/memory_saving_gradients.py /content/gpt-2/src/
```

The training files are now ready to be activated. Let's now explore them, starting with `encode.py`.

Step 8: Encoding the dataset

The dataset must be encoded before training it. You can double-click on `encode.py` to display the file in Google Colaboratory.

`encode.py` loads `dset.txt` by calling the `load_dataset` function that is in `load_dataset.py`:

```
from load_dataset import load_dataset
…/…
chunks = load_dataset(enc, args.in_text, args.combine, encoding=args.
encoding)
```

`encode.py` also loads OpenAI's encoding program, `encode.py`, to encode the dataset:

```
import encoder
…/…
enc = encoder.get_encoder(args.model_name,models_dir)
```

The encoded dataset is saved in a `NumPy` array and stored in `out.npz`. Now, `npz` is a `NumPy` zip archive of the array generated by the encoder:

```
import numpy as np
np.savez_compressed(args.out_npz, *chunks)
```

The dataset is loaded, encoded, and saved in `out.npz` when we run the cell:

```
#@title Step 8:Encoding dataset
import os # import after runtime is restarted
os.chdir("/content/gpt-2/src/")
model_name="117M"
!python /content/gpt-2/src/encode.py dset.txt out.npz
```

Our GPT-2 117M model is ready to be trained.

Step 9: Training a GPT-2 model

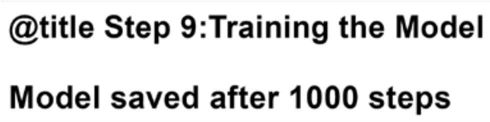
We will now train the GPT-2 117M model on our dataset. We send the name of our encoded dataset to the program:

```
#@title Step 9:Training the Model
#Model saved after 1000 steps
import os # import after runtime is restarted
os.chdir("/content/gpt-2/src/")
!python train.py --dataset out.npz
```

When you run the cell, it will train until you stop it. The trained model is saved after 1,000 steps. When the training exceeds 1,000 steps, stop it. The saved model checkpoints are in `/content/gpt-2/src/checkpoint/run1`. You can check the list of these files in the *Step 10A: Copying Training Files* cell of the notebook.

You can stop the training by double-clicking on the run button of the cell. The training will end, and the trained parameters will be saved.

You can also stop training the model after 1,000 steps with *Ctrl* + *M*. The program will stop and save the trained parameters. It will convert the code into text (you will have to copy it back into a code cell) and display the following message:



@title Step 9:Training the Model

Model saved after 1000 steps

Figure IV.1: Saving a trained GPT-2 model automatically

The program manages the optimizer and gradients with the `/content/gpt-2/src/memory_saving_gradients.py` and `/content/gpt-2/src/accumulate.py` programs.

`train.py` contains a complete list of parameters that can be tweaked to modify the training process. Run the notebook without changing them first. Then, if you wish, you can experiment with the training parameters and see if you can obtain better results.

The GPT-3 model generates samples that you can read during its training. At one point during my GPT-2 training run, the system generated a sample I found enlightening:

```
The world is not a thing in itself, but is a representation of the world
in itself.
```

A representation of the world is what we humans create and what AI learns. Interesting!

Let's continue our experiment by creating a directory for our training model.

Step 10: Creating a training model directory

This section will create a temporary directory for our model, store the information we need, and rename it to replace the directory of the GPT-2 117M model we downloaded.

We start by creating a temporary directory named `tgmodel`:

```
#@title Step 10: Creating a Training Model directory
#Creating a Training Model directory named 'tgmodel'
import os
run_dir = '/content/gpt-2/models/tgmodel'
if not os.path.exists(run_dir):
  os.makedirs(run_dir)
```

We then copy the checkpoint files that contain the trained parameters we saved when we trained our model in the *Step 9: Training the model* subsection of this section:

```
#@title Step 10A: Copying training Files
!cp /content/gpt-2/src/checkpoint/run1/model-1000.data-00000-of-00001 /
content/gpt-2/models/tgmodel
!cp /content/gpt-2/src/checkpoint/run1/checkpoint /content/gpt-2/models/
tgmodel
!cp /content/gpt-2/src/checkpoint/run1/model-1000.index /content/gpt-2/
models/tgmodel
!cp /content/gpt-2/src/checkpoint/run1/model-1000.meta /content/gpt-2/
models/tgmodel
```

Our `tgmodel` directory now contains the trained parameters of our GPT-2 model.

We described these files' content in *Step 5: Downloading the 345M parameter GPT-2 model* in *Appendix III, Generic Text Completion with GPT-2*.

We will now retrieve the hyperparameters and vocabulary files from the GPT-2 117M model we downloaded:

```
#@title Step 10B: Copying the OpenAI GPT-2 117M Model files
!cp /content/gpt-2/models/117M/encoder.json /content/gpt-2/models/tgmodel
!cp /content/gpt-2/models/117M/hparams.json /content/gpt-2/models/tgmodel
!cp /content/gpt-2/models/117M/vocab.bpe /content/gpt-2/models/tgmodel
```

Our `tgmodel` directory now contains our complete customized GPT-2 117M model.

Our last step is to rename the original GPT-2 model we downloaded and set the name of our model to `117M`:

```
#@title Step 10C: Renaming the model directories
import os
!mv /content/gpt-2/models/117M  /content/gpt-2/models/117M_OpenAI
!mv /content/gpt-2/models/tgmodel  /content/gpt-2/models/117M
```

Our trained model is now the one the cloned OpenAI GPT-2 repository will run. Let's interact with our model!

Step 11: Generating unconditional samples

In this section, we will interact with a GPT-2 117M model trained on our dataset. We will first generate an unconditional sample that requires no input on our part. Then we will enter a context paragraph to obtain a conditional text completion response from our trained model.

Let's first run an unconditional sample:

```
#@title Step 11: Generating Unconditional Samples
import os # import after runtime is restarted
os.chdir("/content/gpt-2/src")
!python generate_unconditional_samples.py --model_name '117M'
```

You will not be prompted to enter context sentences since this is an unconditional sample generator.

To stop the cell, double-click on the run button of the cell or press *Ctrl* + *M*.

The result is random but makes sense from a grammatical perspective. From a semantic point of view, the result is not so interesting because we provided no context. But still, the process is remarkable. It invents posts, writes a title, dates it, invents organizations and addresses, produces a topic, and even imagines web links!

The first few lines are rather incredible:

```
Title: total_authority
Category:
Style: Printable
Quote:
Joined: July 17th, 2013
```

```
Posts: 0
Offtopic link: "Essential research, research that supports papers being
peer reviewed, research that backs up one's claims for design, research
that unjustifiably accommodates scientific uncertainties, and research
that persuades opens doors for science and participation in science",
href: https://groups.google.com/
search?q=Author%3APj&src=ieKZP4CSg4GVWDSJtwQczgTWQhAWBO7+tKWn0jzz7o6rP4lEy&s
sl=cTheory%20issue1&fastSource=posts&very=device
Offline
Joined: May 11th, 2014
Posts: 1729
Location: Montana AreaJoined: May 11th, 2014Posts: 1729Location: Montana
Posted: Fri Dec 26, 2017 9:18 pm Post subject: click
I. Synopsis of the established review group
The "A New Research Paradigm" and Preferred Alternative (BREPG) group lead
authors John Obi (Australian, USA and Chartered Institute of Tropical and
Climate Change Research), Marco Xiao (China and Department of Sociology/
Ajax, International Institute of Tropical and Climate Change Research,
Shanghai University) and Jackie Gu (US/Pacific University, Interselicitas
de NASA and Frozen Planet Research Research Center, Oak Ridge National
Laboratory). Dr. Obi states: "Our conclusions indicate that the existence
of the new peer reviewed asan-rubie study predisposes journal publishers
to read scientific publishers constantly to seek a consignment of, and
to be affiliated with, a large target certain of their persons. The
current practice of supplying books with peer review by commonly-overruled
specialists itself spreads from part to part, sufficient to spread the
impact of peer reviews by experts to all levels of the organization, so as
to increase the extent of exposure to the term of deviation that source
profiles or findings require".
```

The result of an unconditional random text generator is interesting but not convincing.

Step 12: Interactive context and completion examples

We will now run a conditional sample. The context we enter will condition the model to think as we want it to, to complete the text by generating tailor-made paragraphs.

Run the cell and explore the magic:

```
#@title Step 12: Interactive Context and Completion Examples
import os # import after runtime is restarted
```

```
os.chdir("/content/gpt-2/src")
!python interactive_conditional_samples.py --temperature 0.8 --top_k 40
--model_name '117M'
```

If necessary, take a few minutes to go back through the parameters in *Step 9, Interacting with GPT-2* of *Appendix III*, *Generic Text Completion with GPT-2*. The program prompts us to enter the context:

```
prompt >>>
```

Figure IV.2: Context input for text completion

Let's enter the same paragraph written by Emmanuel Kant as we did in *Step 9: Interacting with GPT-2* in *Appendix III*, *Generic Text Completion with GPT-2*.

```
Human reason, in one sphere of its cognition, is called upon to
consider questions, which it cannot decline, as they are presented by
its own nature, but which it cannot answer, as they transcend every
faculty of the mind.
```

Press *Enter* to generate text as we did previously. Again, the outputs might change from one run to another, though it is structured and logical, making transformers attractive. This time, the result is not random and is impressive.

Let's look at the first few lines the GPT-2 model produced:

```
"Now, what is the nature of the reason, except that it is of a kind and
power of mind that is so called on to think, and to understand?
The reason is the thing which apprehends, and seeks to understand,
the objects of its minds, which apprehend and apprehend the things of
itself. No one can speak of this reason without the understanding and the
understanding of the human reason. But that knowledge and understanding
which is the most natural of all the faculties, which, because it is
understood by the human reason, is the most natural of all the faculties,
is not without reason, but only without understanding. The human reason,
who sees and knows the things which the natural sense, which, though in
the natural sense, does not understand, is the source of every other
knowledge which we understand. The reason is the thing which does not
understand, but understands the things which we comprehend, and therefore,
is not to be understood. I mean that the reason is the thing which does
not understand, but knows only the things which we do understand…."
```

To stop the cell, double-click on the run button of the cell or enter *Ctrl* + *M*.

Wow! I doubt anybody can see the difference between the text completion produced by our trained GPT-2 model and humans. It might also generate different outputs at each run. This could be the output, for example:

```
The first question, to which it is obliged to respond, is, how can it
be true that a mind which is ignorant of the truth of any proposition
can, or ought, to, accept any proposition which it has not given it? And
this question has been so well answered that it is impossible for any
man to believe in the existence of any mind which has never received any
information.
"It is not the mind of any man, as far as we can judge, which is the
subject of any philosophical inquiry. It is the mind of the minds of men,
in their opinion, which they consider the most to be their most important
subject. And if they can see through this, they will see it, and they will
understand it, and they will understand it."
```

I think our model could outperform many humans in this abstract exercise in philosophy, reason, and logic!

The limit is that the text will vary from one run to another. So, although it seems excellent, it will not fit every single need we have in everyday life.

We can draw some conclusions from our experiment:

- A well-trained transformer model can produce text completion that is at a human level.
- A GPT-2 model can almost reach human level in text generation on complex and abstract reasoning.
- Text context is an efficient way of conditioning a model by demonstrating what is expected.
- Text completion is text generation based on text conditioning if context sentences are provided.
- Although the text is at human level, it does not mean that it will fit every need we have. It is locally interesting but not globally effective at this point.

You can try to enter conditioning text context examples to experiment with text completion. You can also train our model on your own data. Just replace the content of the `dset.txt` file with yours and see what happens!

Bear in mind that our trained GPT-2 model will react like a human. If you enter a short, incomplete, uninteresting, or tricky context, you will obtain puzzled or bad results. This is because GPT-2 expects the best out of us, as in real life!

References

- OpenAI GPT-2 GitHub Repository: `https://github.com/openai/gpt-2`
- *N Shepperd's* GitHub Repository: `https://github.com/nshepperd/gpt-2`

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the authors:

`https://www.packt.link/Transformers`

Appendix V — Answers to the Questions

Chapter 1, What are Transformers?

1. We are still in the Third Industrial Revolution. (True/False)

   False. Eras in history indeed overlap. However, the Third Industrial Revolution focused on making the world digital. The Fourth Industrial Revolution has begun to connect everything to everything else: systems, machines, bots, robots, algorithms, and more.

2. The Fourth Industrial Revolution is connecting everything to everything else. (True/False)

   True. This leads to an increasing amount of automated decisions that formerly required human intervention.

3. Industry 4.0 developers will sometimes have no AI development to do. (True/False)

   True. In some projects, AI will be an online service that requires no development.

4. Industry 4.0 developers might have to implement transformers from scratch. (True/False)

   True. In some projects, not all, standard online services or APIs might not satisfy the needs of a project. There may not be a satisfactory solution for a project in some cases. Instead, a developer will have to customize a model significantly and work from scratch.

5. It's not necessary to learn more than one transformer ecosystem, such as Hugging Face, for example. (True/False)

   False. A corporation's policy might be to work only with Google Cloud AI or Microsoft Azure AI. Hugging Face might be a tool used in another company. You can't know in advance and, in most cases, cannot decide.

6. A ready-to-use transformer API can satisfy all needs. (True/False)

   True if it is effective. False if the transformer model is not well trained.

7. A company will accept the transformer ecosystem a developer knows best. (True/False)

   False. A company may or may not accept what a developer suggests. Therefore, it's safer to cover as many bases as possible.

8. Cloud transformers have become mainstream. (True/False)

   True.

9. A transformer project can be run on a laptop. (True/False)

   True for a prototype, for example. False for a project involving thousands of users.

10. Industry 4.0 AI specialists will have to be more flexible (True/False)

    True.

Chapter 2, Getting Started with the Architecture of the Transformer Model

1. NLP transduction can encode and decode text representations. (True/False)

   True. NLP is transduction that converts sequences (written or oral) into numerical representations, processes them, and decodes the results back into text.

2. **Natural Language Understanding** (**NLU**) is a subset of **Natural Language Processing** (**NLP**). (True/False)

   True.

3. Language modeling algorithms generate probable sequences of words based on input sequences. (True/False)

   True.

4. A transformer is a customized LSTM with a CNN layer. (True/False)

   False. A transformer does not contain an LSTM or a CNN at all.

5. A transformer does not contain LSTM or CNN layers. (True/False)

   True.

6. Attention examines all the tokens in a sequence, not just the last one. (True/False)

   True.

7. A transformer does not use positional encoding. (True/False)

   False. A transformer uses positional encoding.

8. A transformer contains a feedforward network. (True/False)

   True.

9. The masked multi-headed attention component of the decoder of a transformer prevents the algorithm parsing a given position from seeing the rest of a sequence that is being processed. (True/False)

   True.

10. Transformers can analyze long-distance dependencies better than LSTMs. (True/False)

    True.

Chapter 3, Fine-Tuning BERT Models

1. BERT stands for Bidirectional Encoder Representations from Transformers. (True/False)

   True.

2. BERT is a two-step framework. *Step 1* is pretraining. *Step 2* is fine-tuning. (True/False)

   True.

3. Fine-tuning a BERT model implies training parameters from scratch. (True/False)

   False. BERT fine-tuning is initialized with the trained parameters of pretraining.

4. BERT only pretrains using all downstream tasks. (True/False)

   False.

5. BERT pretrains on **Masked Language Modeling** (**MLM**). (True/False)

   True.

6. BERT pretrains on **Next Sentence Prediction** (**NSP**). (True/False)

   True.

7. BERT pretrains on mathematical functions. (True/False)

   False.

8. A question-answer task is a downstream task. (True/False)

   True.

9. A BERT pretraining model does not require tokenization. (True/False)

   False.

10. Fine-tuning a BERT model takes less time than pretraining. (True/False)

    True.

Chapter 4, Pretraining a RoBERTa Model from Scratch

1. RoBERTa uses a byte-level byte-pair encoding tokenizer. (True/False)

   True.

2. A trained Hugging Face tokenizer produces `merges.txt` and `vocab.json`. (True/False)

   True.

3. RoBERTa does not use token-type IDs. (True/False)

   True.

4. DistilBERT has 6 layers and 12 heads. (True/False)

   True.

5. A transformer model with 80 million parameters is enormous. (True/False)

   False. 80 million parameters is a small model.

6. We cannot train a tokenizer. (True/False)

   False. A tokenizer can be trained.

7. A BERT-like model has six decoder layers. (True/False)

   False. BERT contains six encoder layers, not decoder layers.

8. MLM predicts a word contained in a mask token in a sentence. (True/False)

   True.

9. A BERT-like model has no self-attention sublayers. (True/False)

   False. BERT has self-attention layers.

10. Data collators are helpful for backpropagation. (True/False)

    True.

Chapter 5, Downstream NLP Tasks with Transformers

1. Machine intelligence uses the same data as humans to make predictions. (True/False)

   True and False.

   True. In some cases, machine intelligence surpasses humans when processing massive amounts of data to extract meaning and perform a range of tasks that would take centuries for humans to process.

   False. For NLU, humans have access to more information through their senses. Machine intelligence relies on what humans provide for all types of media.

2. SuperGLUE is more difficult than GLUE for NLP models. (True/False)

   True.

3. BoolQ expects a binary answer. (True/False)

   True.

4. WiC stands for Words in Context. (True/False)

   True.

5. **Recognizing Textual Entailment** (**RTE**) detects whether one sequence entails another sequence. (True/False)

   True.

6. A Winograd schema predicts whether a verb is spelled correctly. (True/False)

   False. Winograd schemas mainly apply to pronoun disambiguation.

7. Transformer models now occupy the top ranks of GLUE and SuperGLUE. (True/False)

   True.

8. Human Baseline Standards are not defined once and for all. They were made tougher to attain by SuperGLUE. (True/False)

   True.

9. Transformer models will never beat SuperGLUE human baseline standards. (True/False)

   True and False.

   False. Transformer models beat human baselines for GLUE and will do the same for SuperGLUE in the future.

   True. We will keep setting higher benchmark standards as we progress in the field of NLU.

10. Variants of transformer models have outperformed RNN and CNN models. (True/False)

    True. But you never know what will happen in the future in AI!

Chapter 6, Machine Translation with the Transformer

1. Machine translation has now exceeded human baselines. (True/False)

   False. Machine translation is one of the most challenging NLP ML tasks.

2. Machine translation requires large datasets. (True/False)

   True.

3. There is no need to compare transformer models using the same datasets. (True/False)

   False. The only way to compare different models is to use the same datasets.

4. BLEU is the French word for blue and is the acronym of an NLP metric. (True/False)

   True. **BLEU** stands for **Bilingual Evaluation Understudy Score**, making it easy to remember.

5. Smoothing techniques enhance BERT. (True/False)

   True.

6. German-English is the same as English-German for machine translation. (True/False)

   False. Representing German and then translating into another language is not the same process as representing English and translating into another language. The language structures are not the same.

7. The original Transformer multi-head attention sublayer has two heads. (True/False)

   False. Each attention sublayer has eight heads.

8. The original Transformer encoder has six layers. (True/False)

   True.

9. The original Transformer encoder has six layers but only two decoder layers. (True/False)

   False. There are six decoder layers.

10. You can train transformers without decoders. (True/False)

    True. The architecture of BERT only contains encoders.

Chapter 7, The Rise of Suprahuman Transformers with GPT-3 Engines

1. A zero-shot method trains the parameters once. (True/False)

   False. No parameters are trained.

2. Gradient updates are performed when running zero-shot models. (True/False)

   False.

3. GPT models only have a decoder stack. (True/False)

   True.

4. It is impossible to train a 117M GPT model on a local machine. (True/False)

   False. We trained one in this chapter.

5. It is impossible to train the GPT-2 model with a specific dataset. (True/False)

   False. We trained one in this chapter.

6. A GPT-2 model cannot be conditioned to generate text. (True/False)

   False. We implemented this in this chapter.

7. A GPT-2 model can analyze the context of input and produce completion content. (True/False)

   True.

8. We cannot interact with a 345M GTP parameter model on a machine with fewer than eight GPUs. (True/False).

   False. We interacted with a model of this size in this chapter.

9. Supercomputers with 285,000 CPUs do not exist. (True/False)

   False.

10. Supercomputers with thousands of GPUs are game-changers in AI. (True/False)

    True. Thanks to this, we will be able to build models with increasing numbers of parameters and connections.

Chapter 8, Applying Transformers to Legal and Financial Documents for AI Text Summarization

1. T5 models only have encoder stacks like BERT models. (True/False)

   False.

2. T5 models have both encoder and decoder stacks. (True/False)

   True.

3. T5 models use relative positional encoding, not absolute positional encoding. (True/False)

   True.

4. Text-to-text models are only designed for summarization. (True/False)

   False.

5. Text-to-text models apply a prefix to the input sequence that determines the NLP task. (True/False)

   True.

6. T5 models require specific hyperparameters for each task. (True/False)

   False.

7. One of the advantages of text-to-text models is that they use the same hyperparameters for all NLP tasks. (True/False)

   True.

8. T5 transformers do not contain a feedforward network. (True/False)

   False.

9. Hugging Face is a framework that makes transformers easier to implement. (True/False)

   True.

10. OpenAI's transformer engines are game-changers. (True/False)

    True. OpenAI has produced a wide range of ready-to-use engines such as Codex (language to code) or Davinci (a general purpose engine).

Chapter 9, Matching Tokenizers and Datasets

1. A tokenized dictionary contains every word that exists in a language. (True/False)

   False.

2. Pretrained tokenizers can encode any dataset. (True/False)

   False.

3. It is good practice to check a database before using it. (True/False)

   True.

4. It is good practice to eliminate obscene data from datasets. (True/False)

   True.

5. It is good practice to delete data containing discriminating assertions. (True/False)

   True.

6. Raw datasets might sometimes produce relationships between noisy content and useful content. (True/False)

   True.

7. A standard pretrained tokenizer contains the English vocabulary of the past 700 years. (True/False)

   False.

8. Old English can create problems when encoding data with a tokenizer trained in modern English. (True/False)

   True.

9. Medical and other types of jargon can create problems when encoding data with a tokenizer trained in modern English. (True/False)

   True.

10. Controlling the output of the encoded data produced by a pretrained tokenizer is good practice. (True/False)

    True.

Chapter 10, Semantic Role Labeling with BERT-Based Transformers

1. **Semantic Role Labeling** (**SRL**) is a text generation task. (True/False)

   False.

2. A predicate is a noun. (True/False)

   False.

3. A verb is a predicate. (True/False)

   True.

4. Arguments can describe who and what is doing something. (True/False)

   True.

5. A modifier can be an adverb. (True/False)

   True.

6. A modifier can be a location. (True/False)

   True.

7. A BERT-based model contains encoder and decoder stacks. (True/False)

   False.

8. A BERT-based SRL model has standard input formats. (True/False)

   True.

9. Transformers can solve any SRL task. (True/False)

   False.

Chapter 11, Let Your Data Do the Talking: Story, Questions, and Answers

1. A trained transformer model can answer any question. (True/False)

   False.

2. Question-answering requires no further research. It is perfect as it is. (True/False)

   False.

3. **Named Entity Recognition** (**NER**) can provide useful information when looking for meaningful questions. (True/False)

   True.

4. **Semantic Role Labeling** (**SRL**) is useless when preparing questions. (True/False)

   False.

5. A question generator is an excellent way to produce questions. (True/False)

   True.

6. Implementing question-answering requires careful project management. (True/False)

   True.

7. ELECTRA models have the same architecture as GPT-2. (True/False)

   False.

8. ELECTRA models have the same architecture as BERT but are trained as discriminators. (True/False)

   True.

9. NER can recognize a location and label it as I-LOC. (True/False)

   True.

10. NER can recognize a person and label that person as I-PER. (True/False)

    True.

Chapter 12, Detecting Customer Emotions to Make Predictions

1. It is not necessary to pretrain transformers for sentiment analysis. (True/False)

   False.

2. A sentence is always positive or negative. It cannot be neutral. (True/False)

   False.

3. The principle of compositionality signifies that a transformer must grasp every part of a sentence to understand it. (True/False)

   True.

4. RoBERTa-large was designed to improve the pretraining process of transformer models. (True/False)

   True.

5. A transformer can provide feedback that informs us of whether a customer is satisfied or not. (True/False)

   True.

6. If the sentiment analysis of a product or service is consistently negative, it helps us make appropriate decisions to improve our offer. (True/False)

   True.

7. If a model fails to provide a good result on a task, it requires more training or fine-tuning before changing models. (True/False)

   True.

Chapter 13, Analyzing Fake News with Transformers

1. News labeled as fake news is always fake. (True/False)

   False.

2. News that everybody agrees with is always accurate. (True/False)

   False.

3. Transformers can be used to run sentiment analysis on Tweets. (True/False)

   True.

4. Key entities can be extracted from Facebook messages with a DistilBERT model running NER. (True/False)

   True.

5. Key verbs can be identified from YouTube chats with BERT-based models running SRL. (True/False)

   True.

6. Emotional reactions are a natural first response to fake news. (True/False)

   True.

7. A rational approach to fake news can help clarify one's position. (True/False)

   True.

8. Connecting transformers to reliable websites can help somebody understand why some news is fake. (True/False)

   True.

9. Transformers can make summaries of reliable websites to help us understand some of the topics labeled as fake news. (True/False)

   True.

10. You can change the world if you use AI for the good of us all. (True/False)

    True.

Chapter 14, Interpreting Black Box Transformer Models

1. BERTViz only shows the output of the last layer of the BERT model. (True/False)

   False. BERTViz displays the outputs of all the layers.

2. BERTViz shows the attention heads of each layer of a BERT model. (True/False)

   True.

3. BERTViz shows how the tokens relate to each other. (True/False)

   True.

4. LIT shows the inner workings of the attention heads like BERTViz. (True/False)

   False. However, LIT makes non-probing predictions.

5. Probing is a way for an algorithm to predict language representations. (True/False)

   True.

6. NER is a probing task. (True/False)

   True.

7. PCA and UMAP are non-probing tasks. (True/False)

   True.

8. LIME is model-agnostic. (True/False)

   True.

9. Transformers deepen the relationships of the tokens layer by layer. (True/False)

   True.

10. Visual transformer model interpretation adds a new dimension to interpretable AI. (True/False)

    True.

Chapter 15, From NLP to Task-Agnostic Transformer Models

1. Reformer transformer models don’t contain encoders. (True/False)

   False. Reformer transformer models contain encoders.

2. Reformer transformer models don’t contain decoders. (True/False)

   False. Reformer transformer models contain encoders and decoders.

3. The inputs are stored layer by layer in Reformer models. (True/False)

   False. The inputs are recomputed at each level, thus saving memory.

4. DeBERTa transformer models disentangle content and positions. (True/False)

   True.

5. It is necessary to test the hundreds of pretrained transformer models before choosing one for a project. (True/False)

   True and False. You can try all of the models, or you can choose a very reliable model and implement it to fit your needs.

6. The latest transformer model is always the best. (True/False)

   True and false. A lot of research is being produced on transformers, but some experimental models are short-lived. Sometimes, though, the latest model exceeds the performance of preceding models.

7. It is better to have one transformer model per NLP task than one multi-task transformer model. (True/False)

   True and False. This is a personal decision you will have to make. Risk assessment is a critical aspect of a project.

8. A transformer model always needs to be fine-tuned. (True/False)

   False. GPT-3 engines are zero-shot models.

9. OpenAI GPT-3 engines can perform a wide range of NLP tasks without fine-tuning. (True/False)

   True.

10. It is always better to implement an AI algorithm on a local server. (True/False)

    False. It depends on your project. It's a risk assessment you will have to make.

Chapter 16, The Emergence of Transformer-Driven Copilots

1. AI copilots that can generate code automatically do not exist. (True/False)

   False. GitHub Copilot, for example, is now in production.

2. AI copilots will never replace humans. (True/False)

   True and false. AI will take over many tasks in sales, support, maintenance, and other domains. However, many complex tasks will still require human intervention.

3. GPT-3 engines can only do one task. (True/False)

   False. GPT-3 engines can do a wide variety of tasks.

4. Transformers can be trained to be recommenders. (True/False)

   True. Transformers have gone from language sequences to sequences in many domains.

5. Transformers can only process language. (True/False)

   False. Once transformers are trained for language sequences, they can analyze many other types of sequences.

6. A transformer sequence can only contain words. (True/False)

   False. Once the language sequences are processed, transformers only work on numbers, not words.

7. Vision transformers cannot equal CNNs. (True/False)

   False. Transformers are deep neural networks that can equal CNNs in computer vision.

8. AI robots with computer vision do not exist. (True/False)

   False. For example, robots with computer vision have begun to surface in military applications.

9. It is impossible to produce Python source code automatically. (True/False)

   False. Microsoft and OpenAI have joined to produce a copilot that can write Python code with us or for us.

10. We might one day become the copilots of robots. (True/False)

    This could be true or false. This remains a challenge for humans, bots, and robots in an ever-growing AI ecosystem.

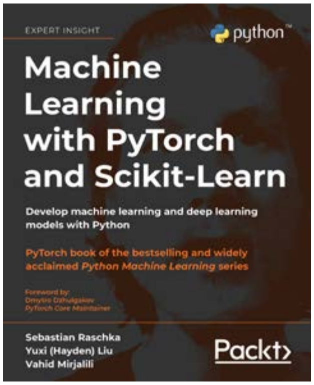
Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the authors:

`https://www.packt.link/Transformers`

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:


EXPERT INSIGHT
python™
Machine Learning with PyTorch and Scikit-Learn
Develop machine learning and deep learning models with Python
PyTorch book of the bestselling and widely acclaimed Python Machine Learning series
Foreword by:
Dmytro Dzhulgakov
PyTorch Core Maintainer
Sebastian Raschka
Yuxi (Hayden) Liu
Vahid Mirjalili
Packt›

Machine Learning with PyTorch and Scikit-Learn

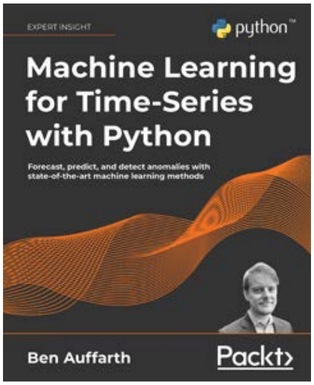
Sebastian Raschka

Yuxi (Hayden) Liu

Vahid Mirjalili

ISBN: 978-1-80181-931-2

- Explore frameworks, models, and techniques for machines to ‘learn’ from data
- Use scikit-learn for machine learning and PyTorch for deep learning
- Train machine learning classifiers on images, text, and more
- Build and train neural networks, transformers, and boosting algorithms
- Discover best practices for evaluating and tuning models
- Predict continuous target outcomes using regression analysis
- Dig deeper into textual and social media data using sentiment analysis

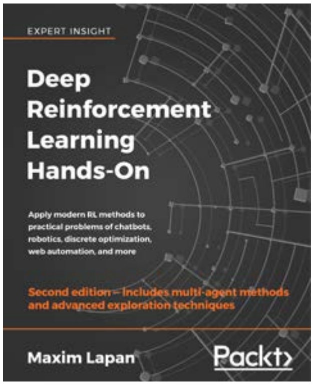

EXPERT INSIGHT
python™
Machine Learning for Time-Series with Python
Forecast, predict, and detect anomalies with state-of-the-art machine learning methods
Ben Auffarth
Packt>

Machine Learning for Time-Series with Python

Ben Auffarth

ISBN: 978-1-80181-962-6

- Understand the main classes of time-series and learn how to detect outliers and patterns
- Choose the right method to solve time-series problems
- Characterize seasonal and correlation patterns through autocorrelation and statistical techniques
- Get to grips with time-series data visualization
- Understand classical time-series models like ARMA and ARIMA
- Implement deep learning models, like Gaussian processes, transformers, and state-of-the-art machine learning models
- Become familiar with many libraries like Prophet, XGboost, and TensorFlow


EXPERT INSIGHT
Deep Reinforcement Learning Hands-On
Apply modern RL methods to practical problems of chatbots, robotics, discrete optimization, web automation, and more
Second edition – Includes multi-agent methods and advanced exploration techniques
Maxim Lapan
Packt

Deep Reinforcement Learning Hands-On – Second Edition

Maxim Lapan

ISBN: 978-1-83882-699-4

- Understand the deep learning context of RL and implement complex deep learning models
- Evaluate RL methods including cross-entropy, DQN, actor-critic, TRPO, PPO, DDPG, D4PG, and others
- Build a practical hardware robot trained with RL methods for less than $100
- Discover Microsoft's TextWorld environment, which is an interactive fiction games platform
- Use discrete optimization in RL to solve a Rubik's Cube
- Teach your agent to play Connect 4 using AlphaGo Zero
- Explore the very latest deep RL research on topics including AI chatbots
- Discover advanced exploration techniques, including noisy networks and network distillation techniques

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Transformers for Natural Language Processing - Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please `click here to go straight to the Amazon review page` for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index