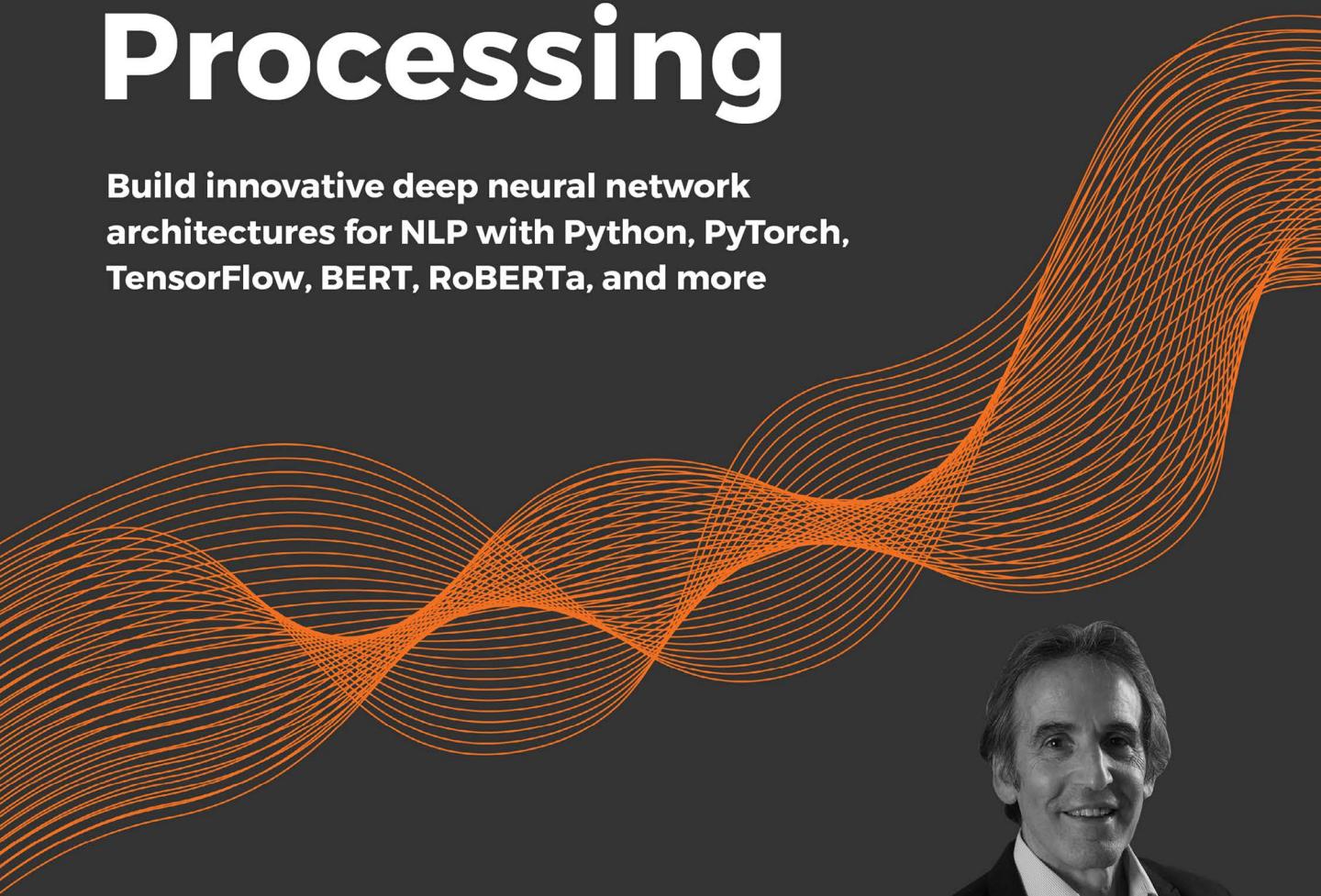


EXPERT INSIGHT

Transformers for Natural Language Processing

**Build innovative deep neural network
architectures for NLP with Python, PyTorch,
TensorFlow, BERT, RoBERTa, and more**



Denis Rothman

Packt

Transformers for Natural Language Processing

Build innovative deep neural network architectures for NLP with Python, PyTorch, TensorFlow, BERT, RoBERTa, and more

Denis Rothman

Packt

BIRMINGHAM - MUMBAI

Transformers for Natural Language Processing

Copyright © 2021 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

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 D'Silva, Divya Mudaliar

Project Editor: Janice Gonsalves

Content Development Editors: Joanne Lovell, Bhavesh Amin

Copy Editor: Safis Editing

Technical Editor: Karan Sonawane

Proofreader: Safis Editing

Indexer: Rekha Nair

Presentation Designer: Ganesh Bhadwalkar

First published: January 2021

Production reference: 1260121

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-80056-579-1

www.packt.com



packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Learn better with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.Packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.Packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Denis Rothman graduated from Sorbonne University and Paris Diderot University, designing one of the very first word2matrix patented embedding and vectorizing systems. He began his career authoring one of the first AI cognitive **Natural Language Processing (NLP)** chatbots applied as an automated language teacher for Moët et Chandon and other companies. He has authored an AI resource optimizer for IBM and apparel producers and an **advanced planning and scheduling (APS)** solution used worldwide.

I want to thank the corporations who trusted me from the start to deliver artificial intelligence solutions and share the risks of continuous innovation. I also thank my family, who believed I would make it big at all times.

About the reviewers

George Mihaila is currently a PhD candidate in computer science at University of North Texas. The main research areas he works on are Deep Learning and **Natural Language Processing (NLP)** with a focus on dialogue generation. His research thesis is on casual dialogue generation with persona.

George is very passionate when it comes to AI and NLP. He always keeps up with the latest language models. Every time a new groundbreaking model comes along, he likes to study the code to better understand its inner workings.

Besides his research, George is also involved in writing tutorials on how to use transformer models in various machine learning tasks. He loves the idea of open source and likes sharing his knowledge and helping others in NLP through his GitHub projects and personal website.

In his free time, George likes to cook and travel with his significant other.

Thanks to everyone on the publishing team and to Denis Rothman for allowing me this opportunity and for making the review process so much fun and easy.

Malte Pietsch is co-founder and CTO at *deepset*, where he builds Haystack – an end-to-end framework for building enterprise search engines fueled by open source and NLP. He holds an M.Sc. with honors from TU Munich and conducted research at Carnegie Mellon University. Before founding *deepset*, he worked as a data scientist for multiple startups. He is an open-source lover, likes reading papers before breakfast, and is obsessed with automating the boring parts of his work.

Carlos Toxtli is a human-computer interaction researcher who studies the impact of artificial intelligence in the future of work. He studied a Ph.D. in computer science at West Virginia University and a master's degree in technological innovation and entrepreneurship at the Monterrey Institute of Technology and Higher Education. He has worked for international organizations such as Google, Microsoft, Amazon, and the United Nations. He was also the technical reviewer on the *Artificial Intelligence By Example, Second Edition* and *Hands-On Explainable AI (XAI) with Python* books. He has also built companies that use artificial intelligence in the financial, educational, customer service, and parking industries. Carlos has published numerous research papers, manuscripts, and book chapters for different conferences and journals in his field.

Table of Contents

Preface	ix
Chapter 1: Getting Started with the Model Architecture of the Transformer	1
The background of the Transformer	2
The rise of the Transformer: Attention Is All You Need	4
The encoder stack	6
Input embedding	8
Positional encoding	11
Sub-layer 1: Multi-head attention	17
Sub-layer 2: Feedforward network	33
The decoder stack	34
Output embedding and position encoding	36
The attention layers	37
The FFN sub-layer, the Post-LN, and the linear layer	37
Training and performance	38
Before we end the chapter	39
Summary	40
Questions	41
References	41
Chapter 2: Fine-Tuning BERT Models	43
The architecture of BERT	44
The encoder stack	44
Preparing the pretraining input environment	47
Pretraining and fine-tuning a BERT model	50
Fine-tuning BERT	53
Activating the GPU	53
Installing the Hugging Face PyTorch interface for BERT	54

Importing the modules	54
Specifying CUDA as the device for torch	55
Loading the dataset	55
Creating sentences, label lists, and adding BERT tokens	57
Activating the BERT tokenizer	57
Processing the data	58
Creating attention masks	59
Splitting data into training and validation sets	59
Converting all the data into torch tensors	60
Selecting a batch size and creating an iterator	60
BERT model configuration	61
Loading the Hugging Face BERT uncased base model	63
Optimizer grouped parameters	65
The hyperparameters for the training loop	66
The training loop	66
Training evaluation	68
Predicting and evaluating using the holdout dataset	69
Evaluating using Matthews Correlation Coefficient	70
The score of individual batches	71
Matthews evaluation for the whole dataset	72
Summary	72
Questions	73
References	73
Chapter 3: Pretraining a RoBERTa Model from Scratch	75
Training a tokenizer and pretraining a transformer	76
Building KantaiBERT from scratch	77
Step 1: Loading the dataset	78
Step 2: Installing Hugging Face transformers	79
Step 3: Training a tokenizer	80
Step 4: Saving the files to disk	82
Step 5: Loading the trained tokenizer files	84
Step 6: Checking resource constraints: GPU and CUDA	85
Step 7: Defining the configuration of the model	86
Step 8: Reloading the tokenizer in transformers	87
Step 9: Initializing a model from scratch	87
Exploring the parameters	89
Step 10: Building the dataset	93
Step 11: Defining a data collator	94
Step 12: Initializing the trainer	94
Step 13: Pretraining the model	95
Step 14: Saving the final model (+tokenizer + config) to disk	96

Step 15: Language modeling with FillMaskPipeline	97
Next steps	98
Summary	99
Questions	100
References	100
Chapter 4: Downstream NLP Tasks with Transformers	101
Transduction and the inductive inheritance of transformers	102
The human intelligence stack	104
The machine intelligence stack	105
Transformer performances versus Human Baselines	106
Evaluating models with metrics	106
Accuracy score	106
F1-score	107
Matthews Correlation Coefficient (MCC)	107
Benchmark tasks and datasets	108
From GLUE to SuperGLUE	108
Introducing higher Human Baseline standards	110
The SuperGLUE evaluation process	111
Defining the SuperGLUE benchmark tasks	113
BoolQ	114
Commitment Bank (CB)	114
Multi-Sentence Reading Comprehension (MultiRC)	115
Reading Comprehension with Commonsense Reasoning Dataset (ReCoRD)	116
Recognizing Textual Entailment (RTE)	117
Words in Context (WiC)	118
The Winograd Schema Challenge (WSC)	118
Running downstream tasks	119
The Corpus of Linguistic Acceptability (CoLA)	120
Stanford Sentiment TreeBank (SST-2)	121
Microsoft Research Paraphrase Corpus (MRPC)	122
Winograd schemas	123
Summary	124
Questions	124
References	125
Chapter 5: Machine Translation with the Transformer	127
Defining machine translation	128
Human transductions and translations	130
Machine transductions and translations	130
Preprocessing a WMT dataset	131
Preprocessing the raw data	131
Finalizing the preprocessing of the datasets	134
Evaluating machine translation with BLEU	138

Geometric evaluations	139
Applying a smoothing technique	141
Chencherry smoothing	142
Translations with Trax	142
Installing Trax	143
Creating a Transformer model	143
Initializing the model using pretrained weights	144
Tokenizing a sentence	144
Decoding from the Transformer	145
De-tokenizing and displaying the translation	145
Summary	146
Questions	147
References	147
Chapter 6: Text Generation with OpenAI GPT-2 and GPT-3 Models	149
The rise of billion-parameter transformer models	151
The increasing size of transformer models	152
Context size and maximum path length	153
Transformers, reformers, PET, or GPT?	154
The limits of the original Transformer architecture	155
Running BertViz	156
The Reformer	159
Pattern-Exploiting Training (PET)	161
The philosophy of Pattern-Exploiting Training (PET)	162
It's time to make a decision	163
The architecture of OpenAI GPT models	164
From fine-tuning to zero-shot models	164
Stacking decoder layers	167
Text completion with GPT-2	168
Step 1: Activating the GPU	169
Step 2: Cloning the OpenAI GPT-2 repository	170
Step 3: Installing the requirements	172
Step 4: Checking the version of TensorFlow	172
Step 5: Downloading the 345M parameter GPT-2 model	173
Steps 6-7: Intermediate instructions	175
Steps 7b-8: Importing and defining the model	176
Step 9: Interacting with GPT-2	178
Training a GPT-2 language model	179
Step 1: Prerequisites	180
Steps 2 to 6: Initial steps of the training process	180
Step 7: The N Shepperd training files	182

Step 8: Encoding the dataset	182
Step 9: Training the model	183
Step 10: Creating a training model directory	184
Context and completion examples	185
Generating music with transformers	189
Summary	189
Questions	190
References	190
Chapter 7: Applying Transformers to Legal and Financial Documents for AI Text Summarization	193
Designing a universal text-to-text model	194
The rise of text-to-text transformer models	195
A prefix instead of task-specific formats	196
The T5 model	198
Text summarization with T5	199
Hugging Face	199
Hugging Face transformer resources	200
Initializing the T5-large transformer model	202
Getting started with T5	203
Exploring the architecture of the T5 model	204
Summarizing documents with T5-large	207
Creating a summarization function	207
A general topic sample	209
The Bill of Rights sample	210
A corporate law sample	211
Summary	212
Questions	213
References	214
Chapter 8: Matching Tokenizers and Datasets	215
Matching datasets and tokenizers	216
Best practices	217
Step 1: Preprocessing	218
Step 2: Post-processing	219
Continuous human quality control	220
Word2Vec tokenization	221
Case 0: Words in the dataset and the dictionary	224
Case 1: Words not in the dataset or the dictionary	225
Case 2: Noisy relationships	227
Case 3: Rare words	228
Case 4: Replacing rare words	229
Case 5: Entailment	230
Standard NLP tasks with specific vocabulary	231
Generating unconditional samples with GPT-2	232

Controlling tokenized data	233
Generating trained conditional samples	236
T5 Bill of Rights Sample	237
Summarizing the Bill of Rights, version 1	238
Summarizing the Bill of Rights, version 2	238
Summary	240
Questions	240
References	241
Chapter 9: Semantic Role Labeling with BERT-Based Transformers	243
Getting started with SRL	244
Defining Semantic Role Labeling	244
Visualizing SRL	245
Running a pretrained BERT-based model	247
The architecture of the BERT-based model	247
Setting up the BERT SRL environment	248
SRL experiments with the BERT-based model	249
Basic samples	249
Sample 1	249
Sample 2	251
Sample 3	253
Difficult samples	256
Sample 4	256
Sample 5	260
Sample 6	262
Summary	262
Questions	263
References	264
Chapter 10: Let Your Data Do the Talking: Story, Questions, and Answers	265
Methodology	267
Transformers and methods	267
Method 0: Trial and error	268
Method 1: NER first	271
Using NER to find questions	271
Location entity questions	274
Person entity questions	277
Method 2: SRL first	278
Question-answering with ELECTRA	279
Project management constraints	281
Using SRL to find questions	282

Next steps	287
Exploring Haystack with a RoBERTa model	289
Summary	290
Questions	291
References	291
Chapter 11: Detecting Customer Emotions to Make Predictions	293
Getting started: Sentiment analysis transformers	294
The Stanford Sentiment Treebank (SST)	294
Sentiment analysis with RoBERTa-large	297
Predicting customer behavior with sentiment analysis	299
Sentiment analysis with DistilBERT	299
Sentiment analysis with Hugging Face's models list	301
DistilBERT for SST	303
MiniLM-L12-H384-uncased	304
RoBERTa-large-mnli	305
BERT-base multilingual model	307
Summary	309
Questions	309
References	310
Chapter 12: Analyzing Fake News with Transformers	311
Emotional reactions to fake news	312
Cognitive dissonance triggers emotional reactions	313
Analyzing a conflictual Tweet	314
Behavioral representation of fake news	317
A rational approach to fake news	319
Defining a fake news resolution roadmap	320
Gun control	321
Sentiment analysis	321
Named entity recognition (NER)	324
Semantic Role Labeling (SRL)	325
Reference sites	329
COVID-19 and former President Trump's Tweets	333
Semantic Role Labeling (SRL)	333
Before we go	336
Looking for the silver bullet	336
Looking for reliable training methods	337
Summary	337
Questions	338
References	338
Appendix: Answers to the Questions	339
Chapter 1, Getting Started with the Model Architecture of the Transformer	339

Chapter 2, Fine-Tuning BERT Models	340
Chapter 3, Pretraining a RoBERTa Model from Scratch	341
Chapter 4, Downstream NLP Tasks with Transformers	342
Chapter 5, Machine Translation with the Transformer	343
Chapter 6, Text Generation with OpenAI GPT-2 and GPT-3 Models	344
Chapter 7, Applying Transformers to Legal and Financial Documents for AI Text Summarization	345
Chapter 8, Matching Tokenizers and Datasets	346
Chapter 9, Semantic Role Labeling with BERT-Based Transformers	347
Chapter 10, Let Your Data Do the Talking: Story, Questions, and Answers	347
Chapter 11, Detecting Customer Emotions to Make Predictions	348
Chapter 12, Analyzing Fake News with Transformers	349
Other Books You May Enjoy	351
Index	355

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.

The global economy has been moving from the physical world to the digital world.

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.

Artificial intelligence-driven language understanding will continue to expand exponentially, as will the volumes of data these activities generate. 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.

Without AI language understanding, it would be incredibly difficult for society to use the Internet.

The Transformer architecture is both revolutionary and disruptive. The Transformer and subsequent transformer architectures and models are revolutionary because they changed the way we think of NLP and artificial intelligence itself. The architecture of the Transformer is not an evolution. It breaks with the past, leaving RNNs and CNNs behind. It takes us closer to seamless machine intelligence that will match human intelligence in the years to come.

The Transformer and subsequent transformer architectures, concepts, and models are disruptive. The various transformers we will explore in this book will progressively replace NLP as we knew it before their arrival.

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, ethical and extract 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 control the millions of messages made per minute manually! 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 leads us to a deeper aspect of artificial intelligence. In a world of exponentially growing data, AI performs more tasks than humans could ever perform. Think of how many translators would be required only to translate one billion online messages, whereas machine translations have no quantitative limits.

This book will show you how to improve language understanding. Each chapter will take you through the key aspects of language understanding from scratch in Python, PyTorch, and TensorFlow.

The demand for language understanding keeps increasing daily 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.

Progress needed to be made. The Transformer, introduced by Google, provides novel approaches to language understanding through a novel self-attention architecture. OpenAI offers transformer technology, and Facebook's AI Research department provides high-quality datasets. Overall, the Internet giants have made transformers available to all, as we will discover in this book.

Transformers can outperform the classical RNN and CNN models in use today. English to French translation and English to German translation transformer models provide better results than ConvS2S (RNN), GNMT (CNN), and SliceNet (CNN), for example.

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.

The book's goal is to give readers the knowledge and tools for Python deep learning that are 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 with Python programming familiarity.
- 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, Getting Started with the Model Architecture of the Transformer, goes through the background of NLP to understand how RNN, LSTM, and CNN architectures were abandoned and how the Transformer architecture 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 2, Fine-Tuning BERT Models, builds on the architecture of the original Transformer. **Bidirectional Encoder Representations from Transformers (BERT)** takes transformers into a vast 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 3, 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.

We will experiment with masked language modeling on an *Immanuel Kant* dataset to explore conceptual NLP representations.

Part II: Applying Transformers for Natural Language Understanding and Generation

Chapter 4, 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 5, 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 6, Text Generation with OpenAI GPT-2 and GPT-3 Models, explores many aspects of OpenAI's GPT-2 transformers. We will first examine GPT-2 and GPT-3 from a project management perspective by looking into alternative solutions such as the Reformer and PET. Then we will explore the novel architecture of OpenAI's GPT-2 and GPT-3 transformer models and run a GPT-2 345M parameter model and interact with it to generate text. We will then train a GPT-2 117M parameter model on a custom dataset and produce customized text completion.

Chapter 7, 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. Finally, 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.

Chapter 8, 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-agnostic method. Finally, 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.

Chapter 9, 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.

Part III: Advanced Language Understanding Techniques

Chapter 10, 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 11, 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.

Chapter 12, 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.

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 1, Getting Started with the Model Architecture of the Transformer*. It contains the description of the Original Transformer, which is built from building blocks 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 this chapter 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.

Download the example code files

The code bundle for the book is hosted on GitHub at <https://github.com/PacktPublishing/Transformers-for-Natural-Language-Processing>. 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/9781800565971_ColorImages.pdf.

Conventions used

There are several text conventions used throughout this book.

CodeInText: Indicates 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:

```
[[0.99987495]] word similarity
[[0.8600013]] positional encoding 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>.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packtpub.com.

1

Getting Started with the Model Architecture of the Transformer

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 **Natural Language Processing (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.

In December 2017, the seminal *Vaswani et al. Attention Is All You Need* article, written by Google Brain members and Google Research, was published. 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. Transformers have become a key component of NLP.

The digital world would never have existed without NLP. Natural Language Processing would have remained primitive and inefficient without artificial intelligence. However, the use of **Recurrent Neural Networks (RNNs)** and **Convolutional Neural Networks (CNNs)** comes at a tremendous cost in terms of calculations and machine power.

In this chapter, we will first start with the background of NLP that led to the rise of the Transformer. We will briefly go from early NLP to RNNs and CNNs. Then we will see how the Transformer overthrew the reign of RNNs and CNNs, which had prevailed for decades for sequence analysis.

Then we will open the hood of the Transformer model described by *Vaswani et al. (2017)* and examine the key 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 background of the Transformer
- 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

Our first step will be to explore the background of the Transformer.

The background of the Transformer

In this section, we will go through the background of NLP that led to the Transformer. The Transformer model invented by Google Research has toppled decades of Natural Language Processing research, development, and implementations.

Let us first see how that happened when NLP reached a critical limit that required a new approach.

Over the past 100+ years, many great minds have worked on sequence transduction and language modeling. 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 my favorite researchers with you to lay the ground 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 **artificial intelligence (AI)** as **Markov Decision Processes (MDPs)**, Markov Chains, and Markov Processes. In 1902, Markov showed that we could predict the next element of a chain, a sequence, using only the last past element of that chain. In 1913, he applied this to a 20,000-letter dataset using past sequences to predict the future letters of a chain. Bear in mind that he had no computer but managed to prove his theory, which is still in use today in AI.

In 1948, Claude Shannon's *The Mathematical Theory of Communication* was published. He cites Andrey Markov's theory multiple times when building his probabilistic approach to sequence modeling. Claude Shannon laid the ground for a communication model based on a source encoder, a transmitter, and a received decoder or semantic decoder.

In 1950, Alan Turing published his seminal article: *Computing Machinery and Intelligence*. Alan Turing based this article on machine intelligence on the immensely successful Turing Machine that decrypted German messages. The expression *artificial intelligence* was first used by John McCarthy in 1956. However, Alan Turing was implementing artificial intelligence in the 1940s to decode encrypted encoded messages in German.

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. However, creating rule lists for the billions of language combinations in our digital world is a challenge yet to be met. For the moment, it seems impossible. But who knows what will happen?

In 1982, John Hopfield introduced **Recurrent Neural Networks (RNNs)**, 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. RNNs evolved, and LSTMs emerged as we know them. An RNN memorizes the persistent states of a sequence efficiently:

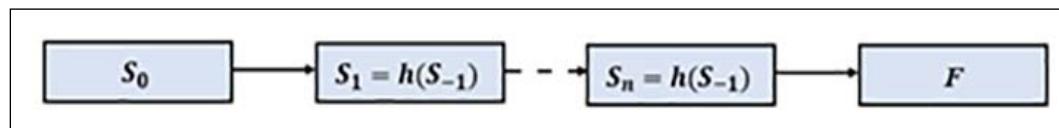


Figure 1.1: The RNN process

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

In the 1980s, **Yann Le Cun** designed the multi-purpose **Convolutional Neural Network (CNN)**. He applied CNNs to text sequences, and they have been widely used for sequence transduction and modeling as well. They are also based on *persistent states* that gather information layer by layer. In the 1990s, summing up several years of work, Yann Le Cun produced LeNet-5, which led to the many CNN models we know today. The CNN's otherwise efficient architecture faces problems when dealing with long-term dependencies in very long 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 that required an increasing amount of computer power, AI developers used more powerful machines and found ways to optimize gradients.

It seemed that nothing else could be done to make more progress. Thirty years passed this way. And then, in December 2017, came the Transformer, the incredible innovation that seems to have come from a distant planet. The Transformer swept everything away, producing impressive scores on standard datasets.

Let's start our exploration of the architecture of the Transformer with the design of this alien NLP/NLU spaceship!

The rise of the Transformer: Attention Is All You Need

In December 2017, Vaswani et al. 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.

In this section, we will look at the Transformer model they built from the outside. 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:

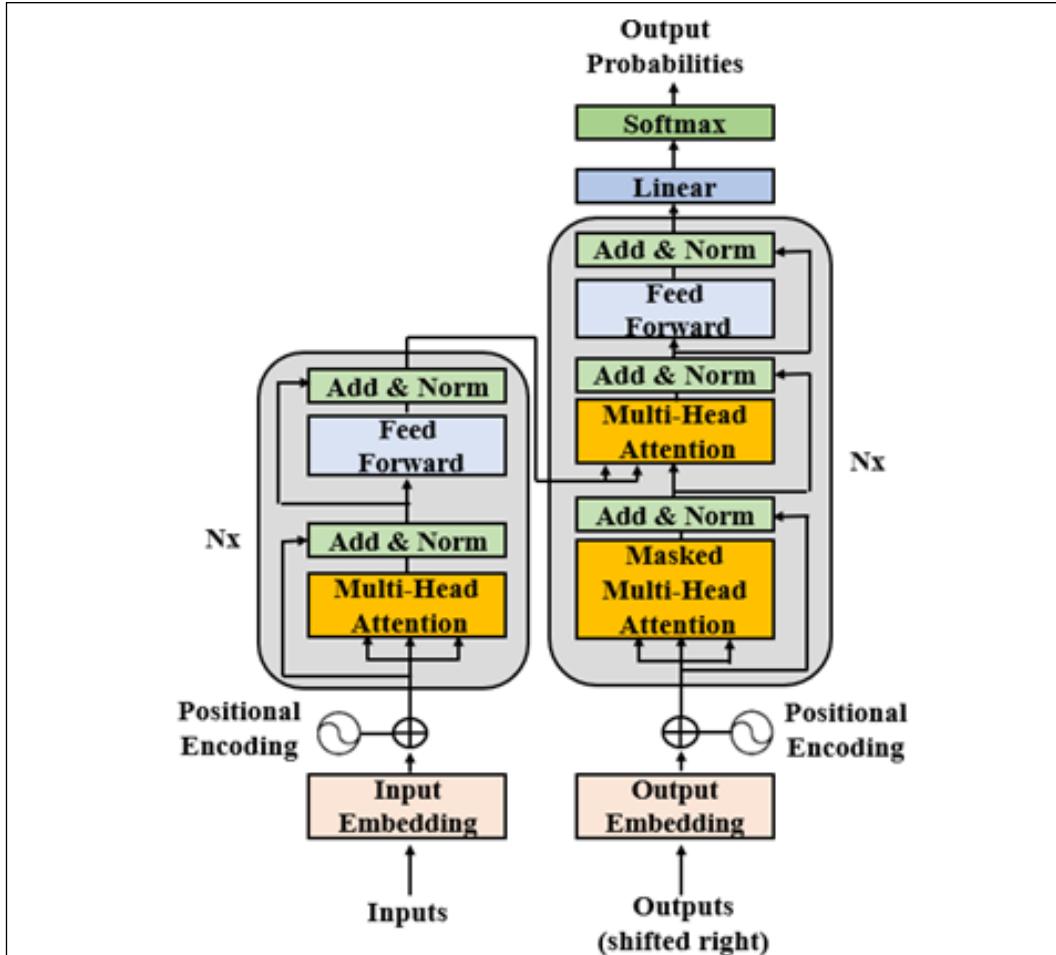


Figure 1.2: The architecture of the Transformer

On the left, the inputs enter the encoder side of the Transformer through an attention sub-layer and **FeedForward Network (FFN)** sub-layer. On the right, the target outputs go into the decoder side of the Transformer through two attention sub-layers and an FFN sub-layer. We immediately notice that there is no RNN, LSTM, or CNN. Recurrence has been abandoned.

Attention has replaced recurrence, which requires an increasing number of operations as the distance between two words increases. The attention mechanism is a "word-to-word" operation. 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 among all the other words, including itself ("cat" and "cat"):

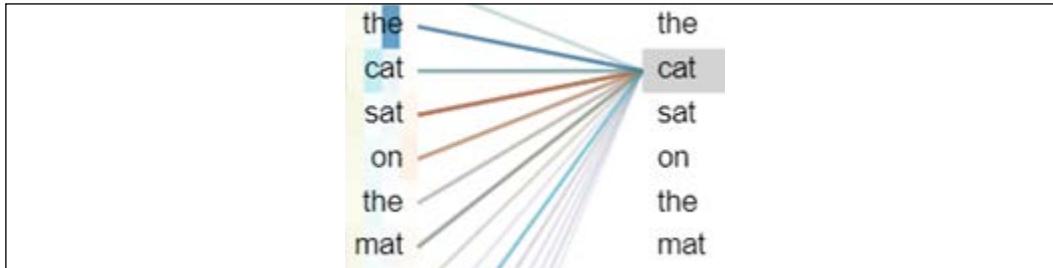


Figure 1.3: Attending to all the words

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

For each attention sub-layer, 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
- The 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 that are as critical as the attention mechanism, as you will see when we look inside the architecture.

We just looked at the Transformer 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:

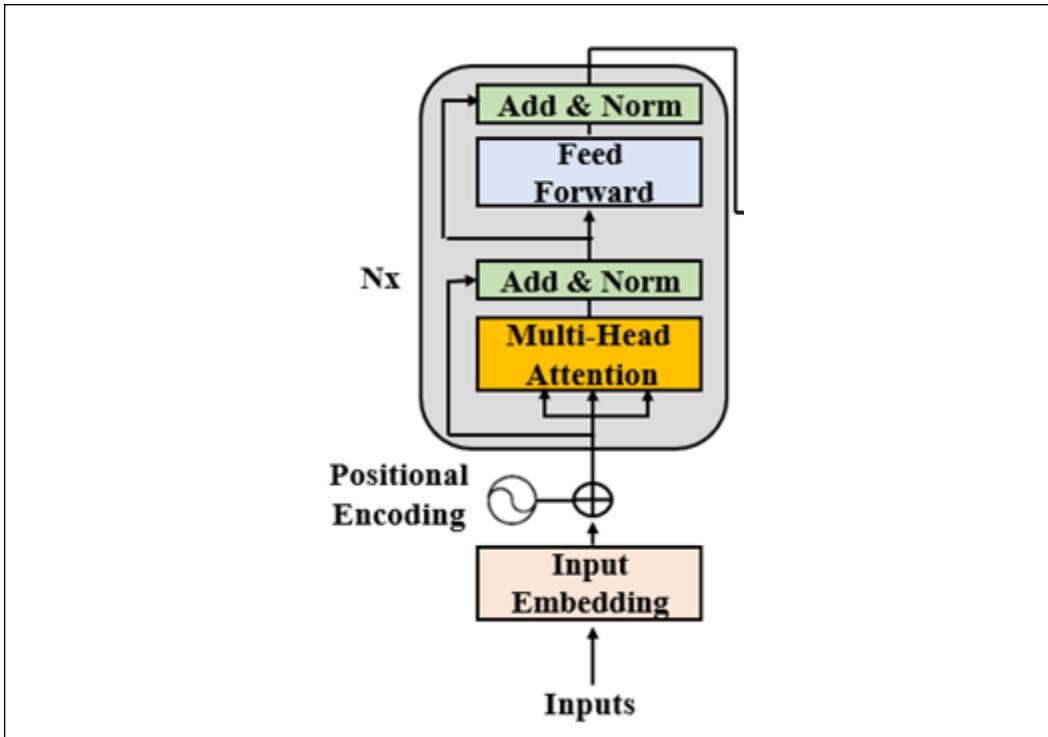


Figure 1.4: A layer of the encoder stack of the Transformer

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

Notice that a residual connection surrounds each main sub-layer, $\text{Sublayer}(x)$, in the Transformer model. These connections transport the unprocessed input x of a sub-layer 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:

$$\text{LayerNormalization}(x + \text{Sublayer}(x))$$

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 sub-layer 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 how 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 sub-layer 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. 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 sub-layers and mechanisms.

We will begin with the embedding sub-layer.

Input embedding

The input embedding sub-layer 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:

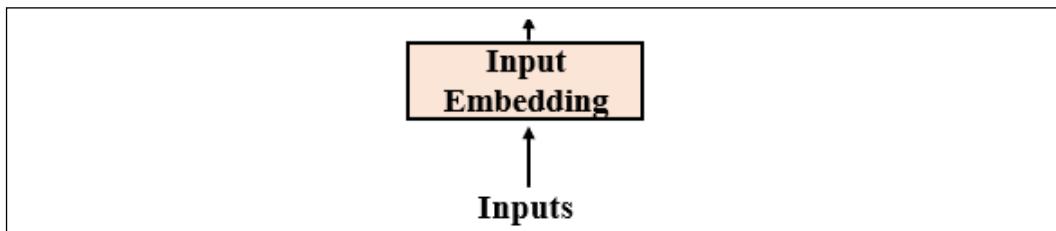


Figure 1.5: The input embedding sub-layer of the Transformer

The embedding sub-layer works like other standard transduction models. A tokenizer will transform a sentence into tokens. Each tokenizer has its methods, but the results are similar. 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', 'a', 'revolutionary', 'n', 'l', 'p',  
'model', '!']
```

You will notice that this tokenizer normalized the string to lower case 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 sub-layer. 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 $\text{word}(i)$ is the center word in a two-step window, a skip-gram model will analyze $\text{word}(i-2)$, $\text{word}(i-1)$, $\text{word}(i+1)$, and $\text{word}(i+2)$. 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 $d_{\text{model}} = 512$ 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 `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 very 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:

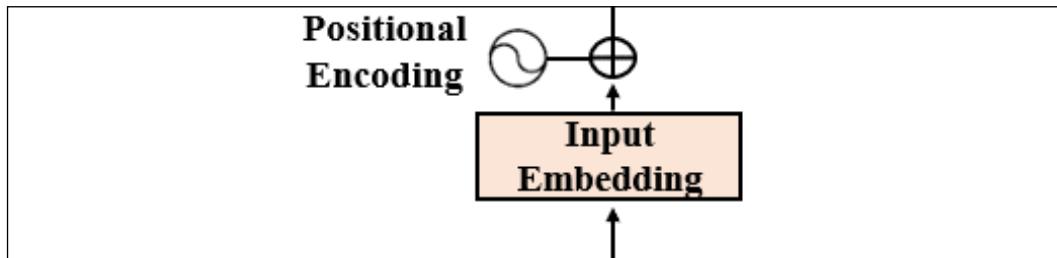


Figure 1.6: Position encoding

We cannot create independent positional vectors that would have a high cost on the training speed of the Transformer and make attention sub-layers very 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.

We also know that 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 sub-layer, we can see that **black** and **brown** may be similar, but they are far apart:

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 this goal. The designers found a clever way to use a unit sphere to represent positional encoding with sine and cosine values that will thus remain small but very 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:

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

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

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 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
```

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:

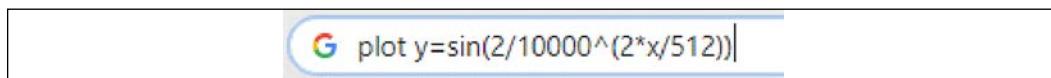


Figure 1.7: Plotting with Google

You will obtain the following graph:

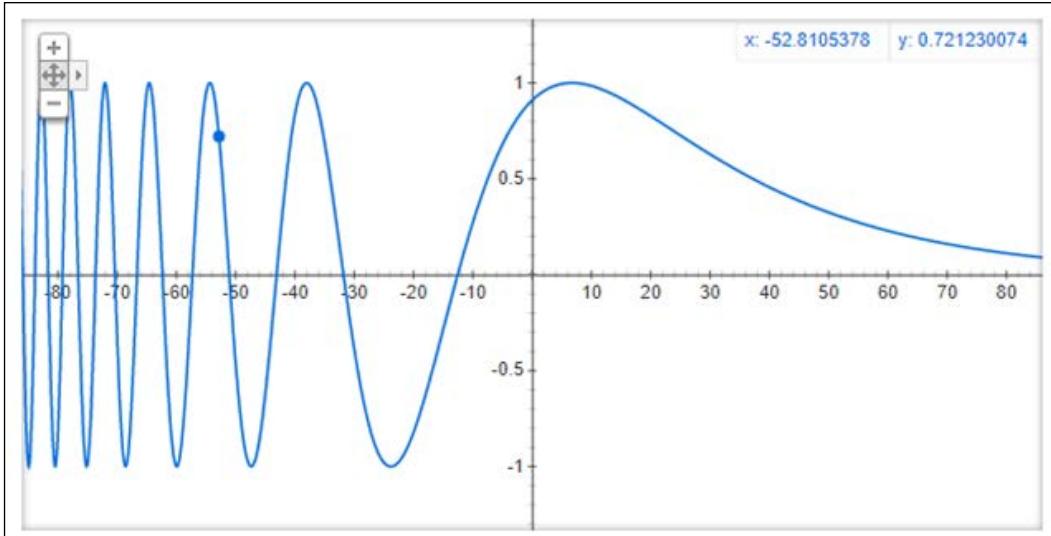


Figure 1.8: The graph

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 now 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:

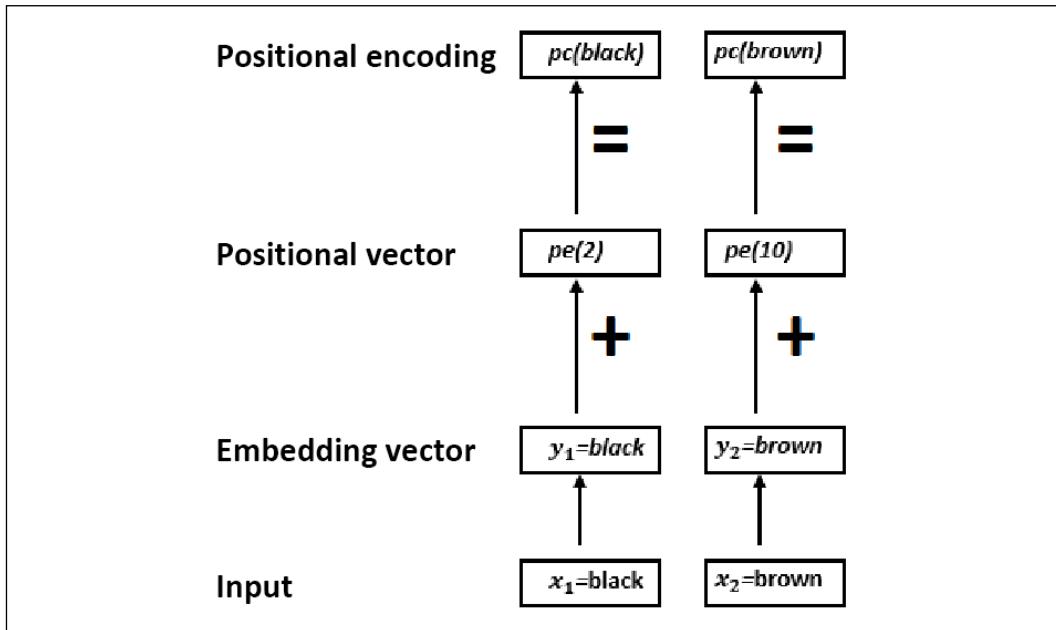


Figure 1.9: Positional encoding

If we go back and take the word embedding of `black`, for example, and name it $y1=\text{black}$, we are ready to add it to the positional vector $pe(2)$ we obtained with positional encoding functions. We will obtain the positional encoding $pc(\text{black})$ of the input word `black`:

$$pc(\text{black}) = y1 + pe(2)$$

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 $y1$ 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`:

$$y_1 * \text{math.sqrt}(d_{\text{model}})$$

We can now add the positional vector to the embedding vector of the word `black`, both of which are 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 $d_{\text{model}} = 512$:

```
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. The output of this algorithm, which is not rule-based, might slightly vary during each run.

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 very 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.

Hugging Face and Google Brain Trax both, among others, provide ready-to-use libraries for functionality we explored in the word embedding section and the present positional encoding section. Thus, you do not need to run the program I used in this chapter to check the Transformer equations, and this section is self-contained. 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.

The output of positional encoding is the multi-head attention sub-layer.

Sub-layer 1: Multi-head attention

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

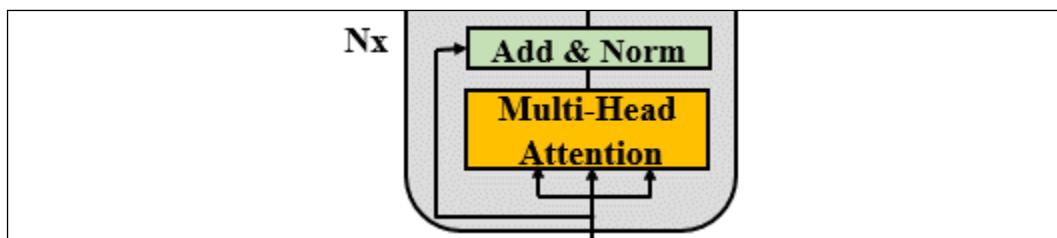


Figure 1.10: Multi-head attention sub-layer

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 sub-layer 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 $d_{model} = 512$:

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

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 of 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:

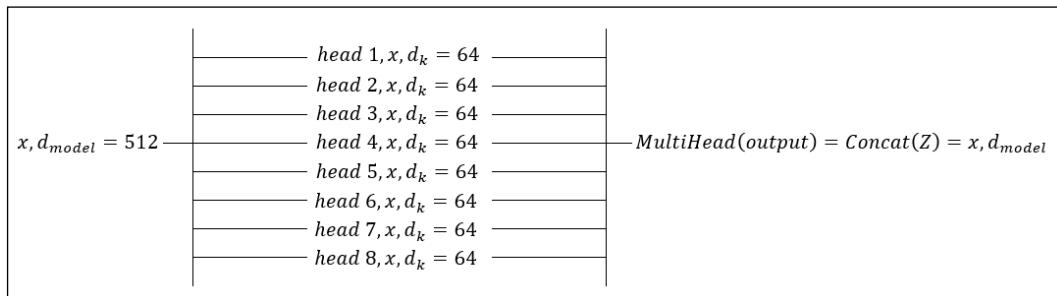


Figure 1.11: Multi-head representations

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 sub-layer is not a sequence of dimensions but one lines of xm^*d_{model} matrix.

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

$$\text{MultiHead}(\text{output}) = \text{Concat}(z_0, z_1, z_2, z_3, z_4, z_5, z_6, z_7) = x, d_{model}$$

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, each word vector has three representations:

- A query vector (Q) that has a dimension of $d_q = 64$, which is activated and trained when a word vector x_n seeks all of the key-value pairs of the other word vectors, including itself in self-attention
- A key vector (K) that has a dimension of $d_k = 64$, which will be trained to provide an attention value
- A value vector (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 :

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

The vectors 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 respective 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 of 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.

Hugging Face and Google Brain Trax, among others, provide ready-to-use frameworks, libraries, and modules 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 in order 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.

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 sub-layer using basic code:

```
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:

```
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:

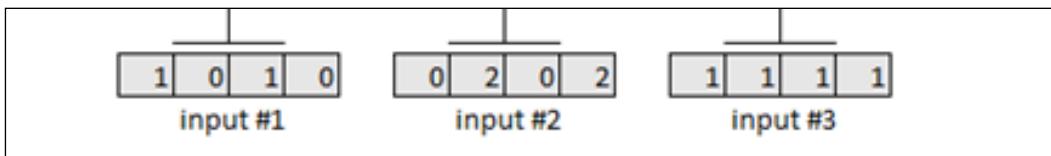


Figure 1.12: Input of a multi-head attention sub-layer

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 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:

```
Step 2: weights 3 dimensions x d_model=4
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:

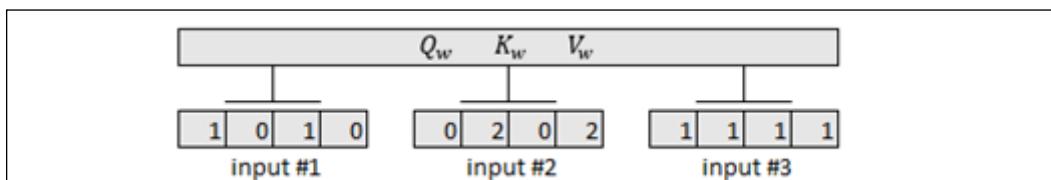


Figure 1.13: Weight matrices added to the model

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

Step 3: Matrix multiplication to obtain Q, K, 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 $Q_1 = [1, 0, 2]$, $Q_2 = [2, 2, 2]$, 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:

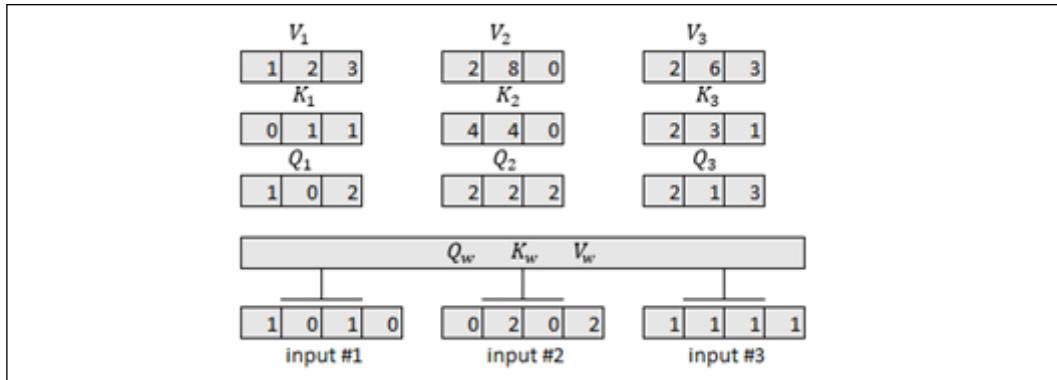


Figure 1.14: Q, K, and V are generated

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:

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

Step 4 focuses on Q and K :

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

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:

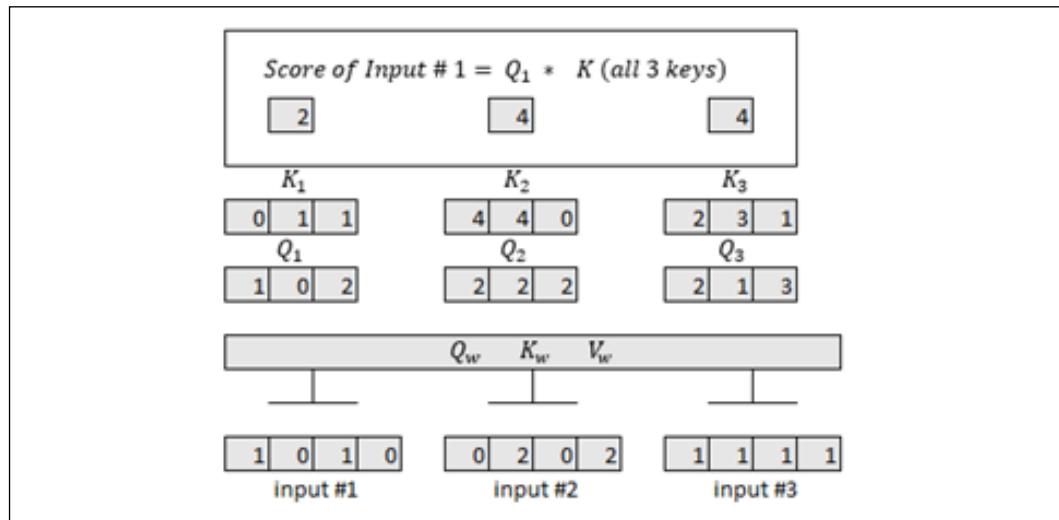


Figure 1.15: Scaled attention scores for input #1

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:

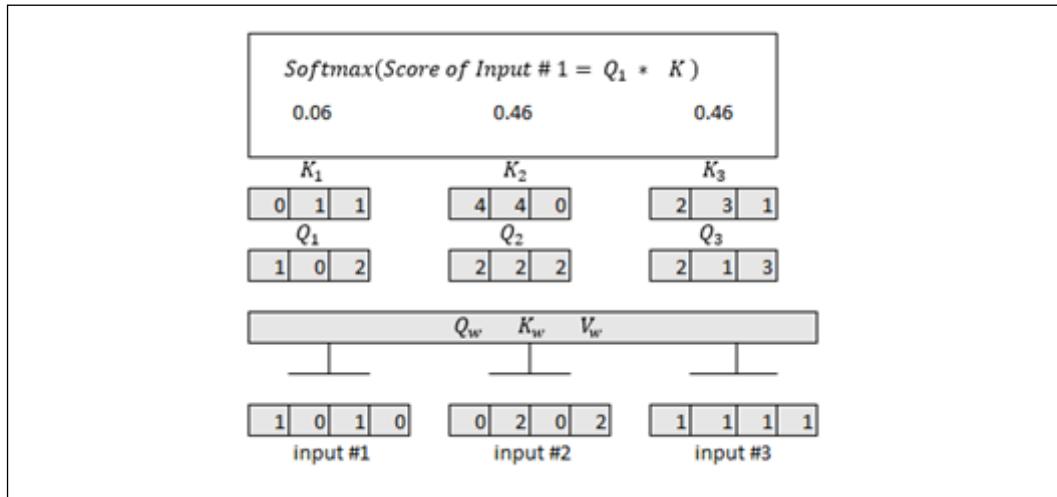


Figure 1.16: Softmax score of input #1 for all of the keys

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:

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

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:

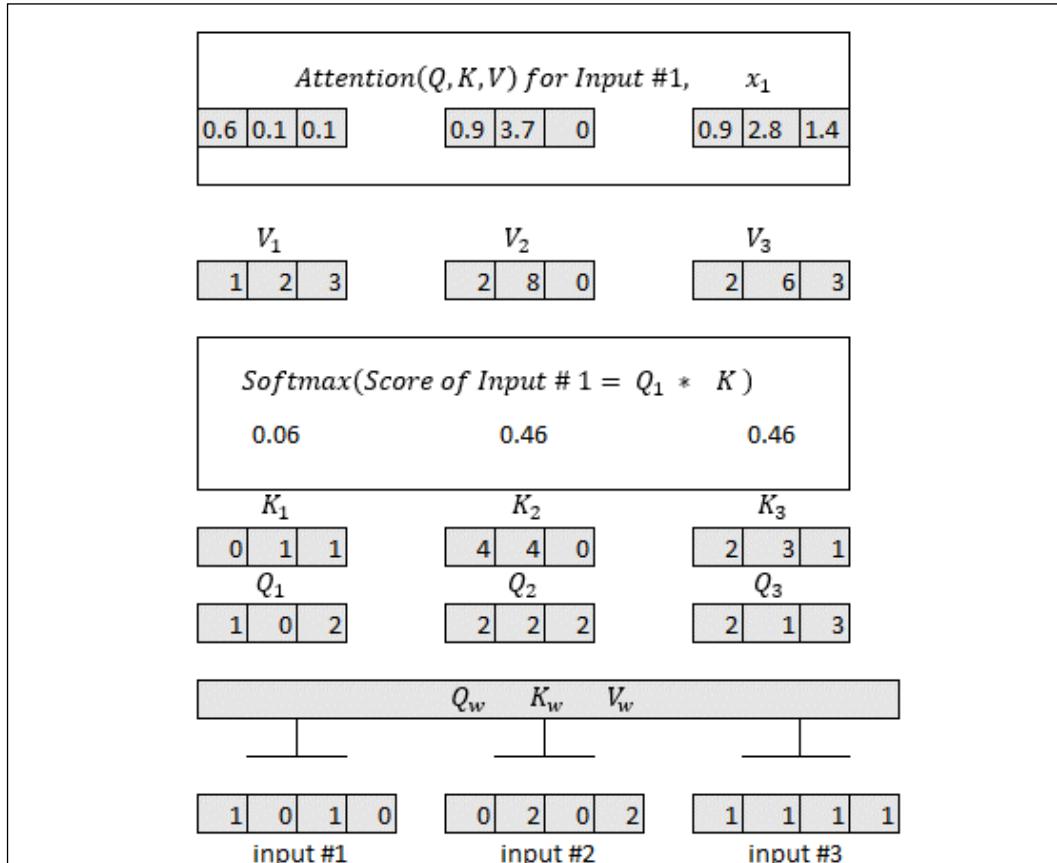


Figure 1.17: Attention representations

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("Step7: 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 1.18*:

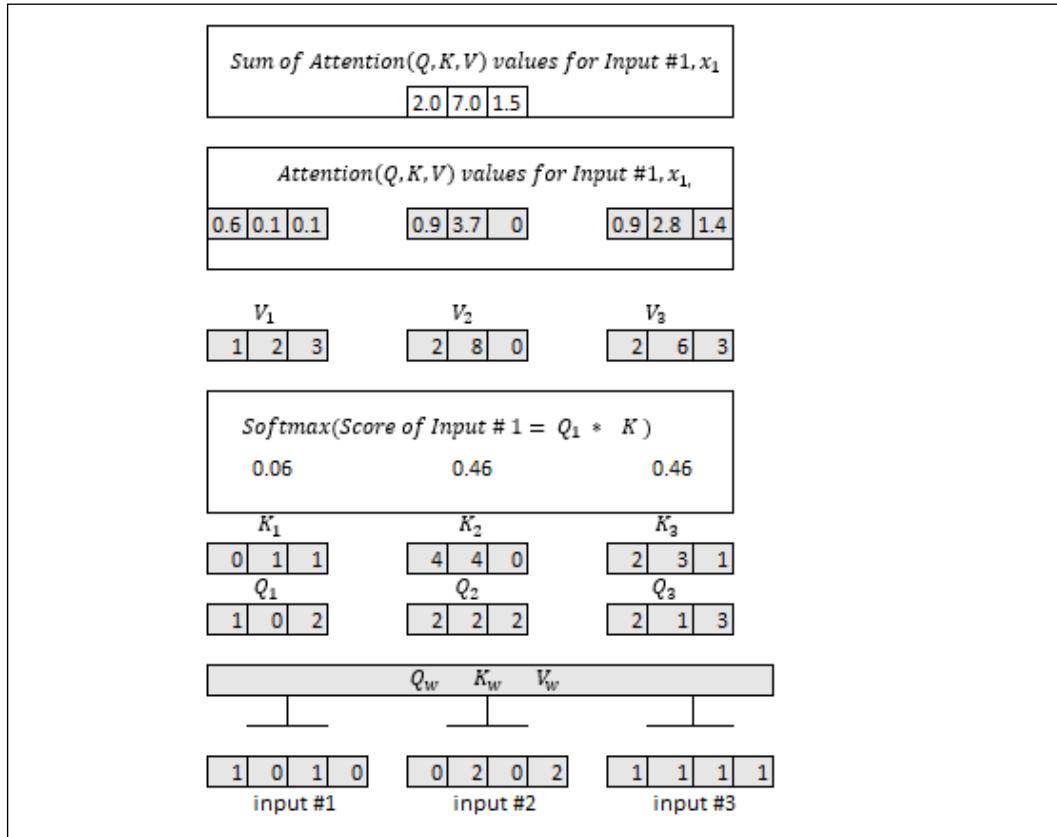


Figure 1.18: Summed results for one input

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 onward, 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 sub-layer'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_o , 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 random generation of the vectors.

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

Step 9: The output of the heads of the attention sub-layer

We assume that we have trained the 8 heads of the attention sub-layer. 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
sub-layer")
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 sub-
layer
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 sub-layer.

Step 10: Concatenation of the output of the heads

The Transformer concatenates the 8 elements of Z:

$$\text{MultiHead}(output) = \text{Concat}(z_0, z_1, z_2, z_3, z_4, z_5, z_6, z_7)W^0 = x, d_{\text{model}}$$

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 is concatenated:

```

print("Step 10: Concatenation 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:

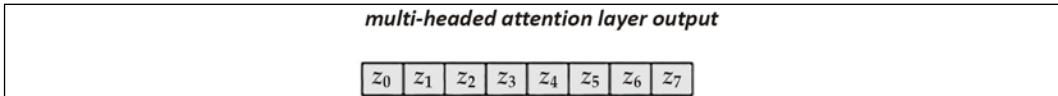


Figure 1.19: Attention sub-layer output

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

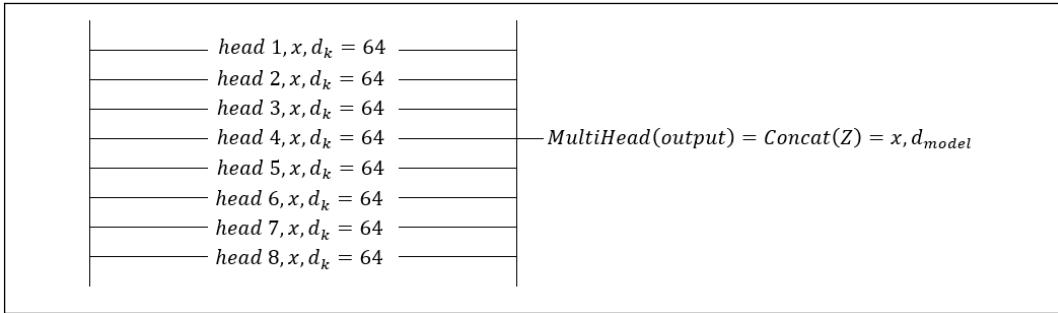


Figure 1.20: Concatenation of the output of the 8 heads

Layer normalization will now process the attention sub-layer.

Post-layer normalization

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

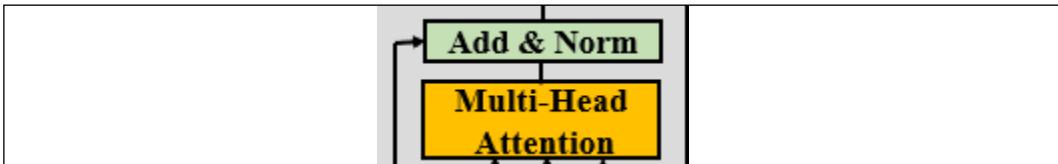


Figure 1.21: Post-layer normalization

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 sub-layer. 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:

$$\text{LayerNorm}(x + \text{Sublayer}(x))$$

$\text{Sublayer}(x)$ is the sub-layer itself. x is the information available at the input step of $\text{Sublayer}(x)$.

The input of *LayerNorm* 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 *LayerNorm*(v):

$$\text{LayerNorm}(v) = \gamma \frac{v - \mu}{\sigma} + \beta$$

The variables are:

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

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

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

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

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

This version of *LayerNorm*(v) shows the general idea of the many possible Post-LN methods.

The next sub-layer can now process the output of the Post-LN or *LayerNorm*(v). In this case, the sub-layer is a feedforward network.

Sub-layer 2: Feedforward network

The input of the FFN is the $d_{model} = 512$ output of the Post-LN of the previous sub-layer:

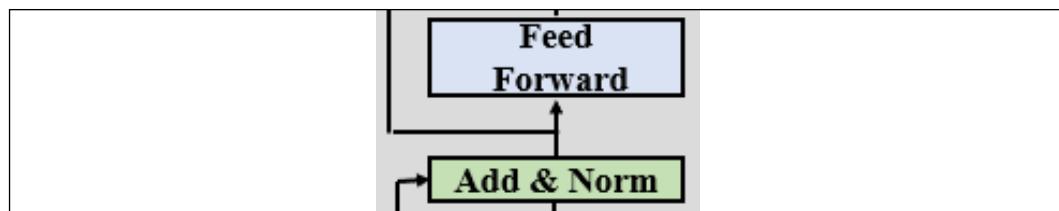


Figure 1.22: Feedforward sub-layer

The FFN sub-layer 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 kernel size 1 convolutions.

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

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

The output of the FFN goes to the 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:

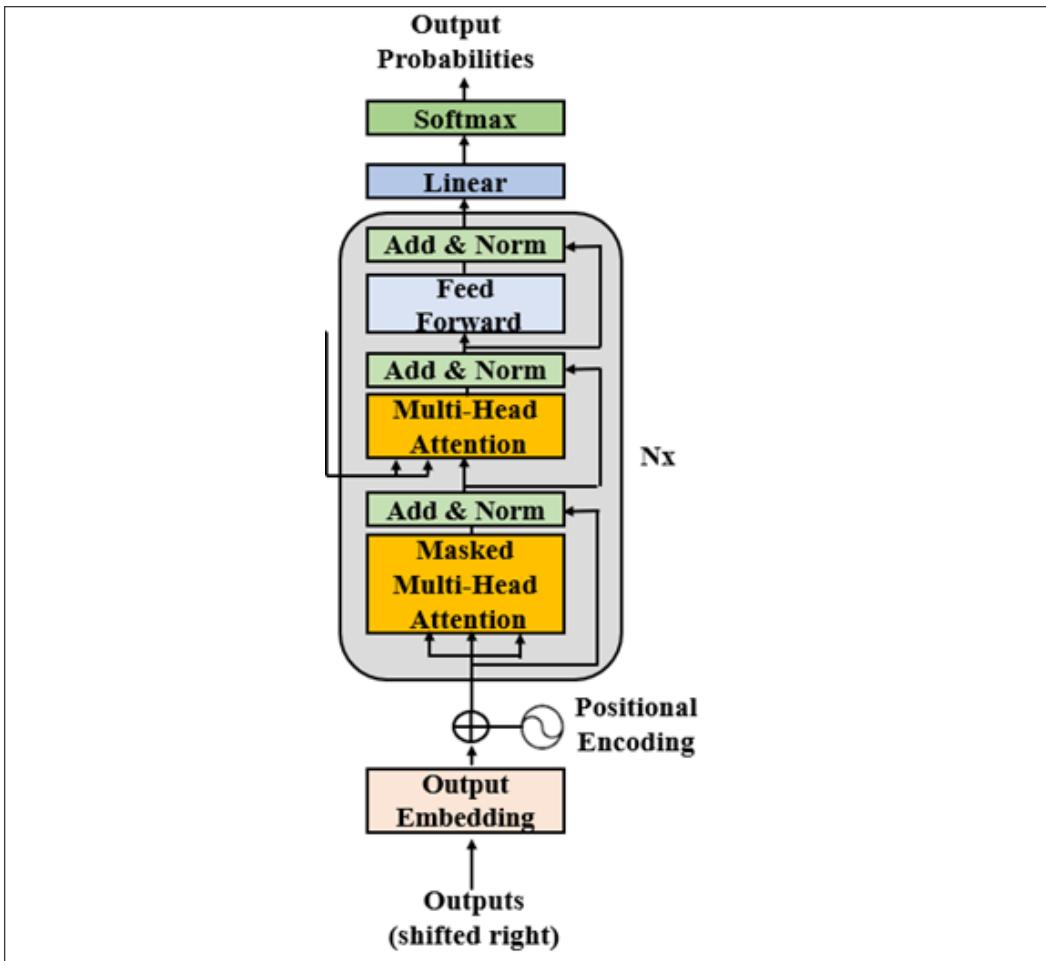


Figure 1.23: A layer of the decoder stack of the Transformer

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 sub-layers: 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 sub-layer, which is the masked multi-head attention mechanism. In this sub-layer 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, $\text{Sublayer}(x)$, surrounds each of the three main sub-layers in the Transformer model like in the encoder stack:

$$\text{LayerNormalization}(x + \text{Sublayer}(x))$$

The embedding layer sub-layer is only present at the bottom level of the stack, like for the encoder stack. The output of every sub-layer 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 sub-layer 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 sub-layers of the decoder is mostly the same as the sub-layers 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 sub-layer 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 sub-layer 1 as in the encoder.

The multi-head attention sub-layer 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 sub-layer 2 draws information from the encoder by taking *encoder* (K, V) into account during the dot-product attention operations. This sub-layer also draws information from the masked multi-head attention sub-layer 1 (masked attention) by also taking sub-layer 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 sub-layer of a decoder as:

$$\text{Input_Attention} = (\text{Output_decoder_sub_layer-1}(Q), \text{Output_encoder_layer}(K, V))$$

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

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

The FFN sub-layer, the Post-LN, and the linear layer

The FFN sub-layer has the same structure as the FFN of the encoder stack. The Post-LN 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:

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

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

$$y = w^*x + b$$

x 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 as 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 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 were applied, such as residual dropout and dropouts, to the sums of embeddings. Also, the Transformer applies label smoothing, which 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 subsequent chapters.

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

Before we end the chapter

Everything you saw in this chapter can be condensed into a ready-to-use Hugging Face transformer model. Bear in mind that Hugging Face, like all other solutions, is evolving at full speed to keep up with the research labs so you might encounter deprecation messages in the future.

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 last two cells.

We first ensure that Hugging Face's transformers are installed:

```
!pip -qq install transformers
```

The first cell imports the Hugging Face pipeline, which 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 that contains several transformer usages. The pipeline 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 avec des 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 transduction 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 sub-layer 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 in more 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 an LSTM or CNN layers. (True/False)
6. Attention examines all of 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, Łukasz 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>
- *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/>
- Google Brain Trax documentation: <https://trax-ml.readthedocs.io/en/latest/>
- 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/>

2

Fine-Tuning BERT Models

In *Chapter 1, Getting Started with the Model Architecture of the Transformer*, 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 are having 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.

In this chapter, we 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-tuning BERT model
- Loading an accessibility judgement 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 Transformer.

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 1, Getting Started with the Model Architecture of the Transformer*. You can consult *Chapter 1* 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 1, Getting Started with the Model Architecture of the Transformer*, is shown in *Figure 2.1*:

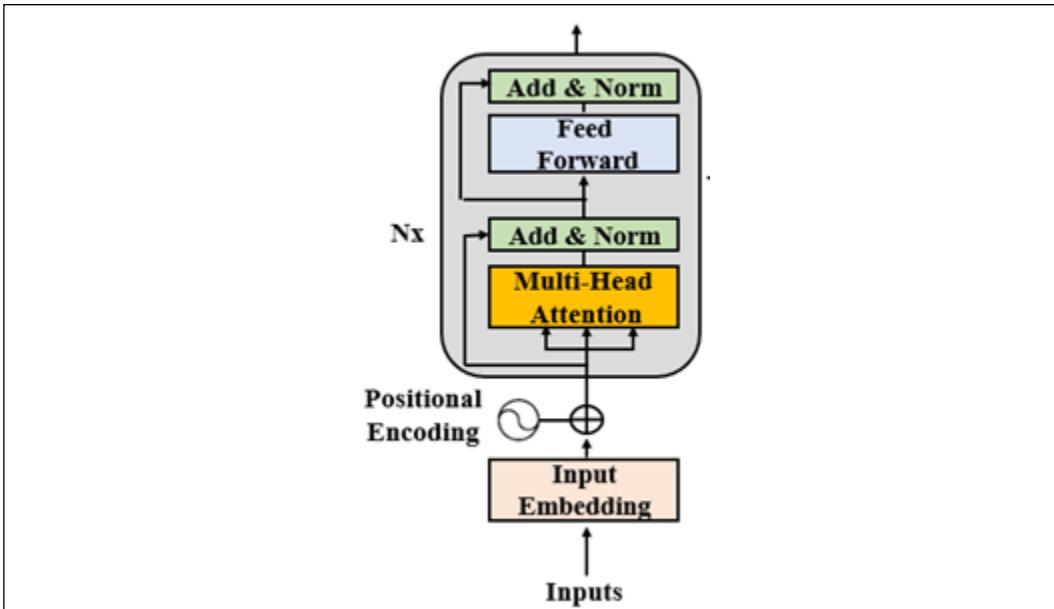


Figure 2.1: The encoder layer

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 $d_{model} = 512$. The number of attention heads of the original Transformer is $A=8$. The dimensions of a head of the original Transformer is:

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

BERT encoder layers are larger than the original Transformer model.

Two BERT models can be built with the encoder layers:

- BERT_{BASE}, which contains a stack of $N=12$ encoder layers. $d_{model} = 768$ and can also be expressed as $H=768$, as in the BERT paper. A multi-head attention sub-layer contains $A=12$ heads. The dimensions 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:

$$\text{output_multi-head_attention} = \{z_0, z_1, z_2, \dots, z_{11}\}$$

- BERT_{LARGE}, which contains a stack of $N=24$ encoder layers. $d_{model} = 1024$. A multi-head attention sub-layer contains $A=16$ heads. The dimensions 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:

$$\text{output_multi-head_attention} = \{z_0, z_1, z_2, \dots, z_{15}\}$$

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

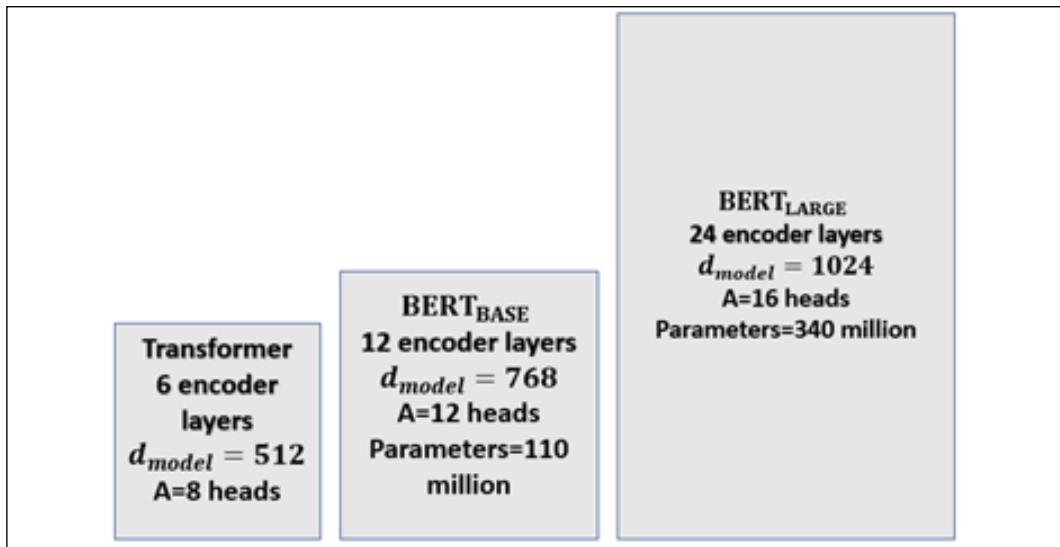


Figure 2.2: Transformer models

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 now 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 goes further and states 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 sub-word segmentation method, tokenization 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.

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 would become one input complete 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 _[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 2.3: 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 all of 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. This forces the model to think harder during the multi-head attention learning process. This makes BERT able to 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 the pretraining, and the second is fine-tuning, as shown in *Figure 2.4*:

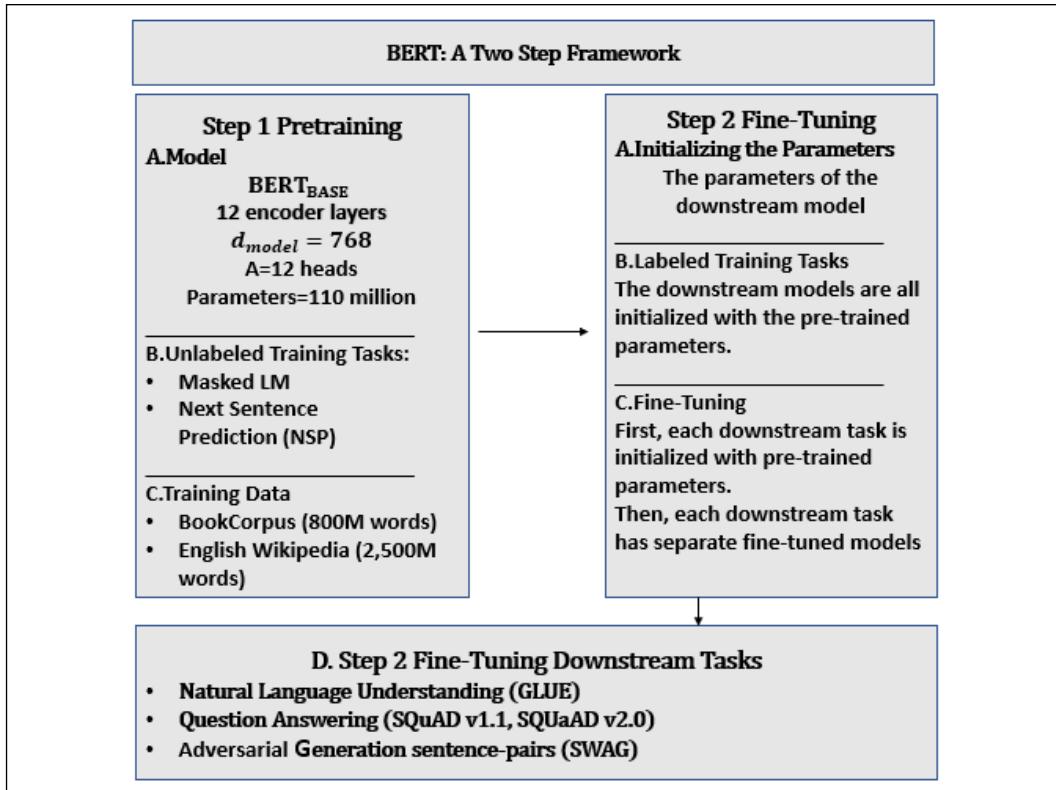


Figure 2.4: The BERT framework

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 that 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 **Masked Language Modeling (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 3, Pretraining a RoBERTa Model from Scratch*, we will pretrain a BERT-like model from scratch in 15 steps. We will even compile our own data, train a tokenizer, and then train the model. The goal of this chapter is to first go through the specific building blocks of BERT and then fine-tune an existing model.
- In *Chapter 4, Downstream NLP Tasks with Transformers*, we will go through many downstream NLP tasks, exploring GLUE, SQuAD v1.1, SQuAD, SWAG, BLEU, 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 6, Text Generation with OpenAI GPT-2 and GPT-3 Models*, we will explore the architecture and usage of Open AI GPT, GPT-2, and GPT-3 transformers. BERT_{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.

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

In this section, we will fine-tune a BERT model to predict the downstream task of *Acceptability Judgements* 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_DR.ipynb` in Google Colab (make sure you have an email account). The notebook is in `Chapter02` of the GitHub repository of this book.

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

Let's start making sure the GPU is activated.

Activating the GPU

Pretraining a multi-head attention transformer model requires the parallel processing GPUs can provide.

The program first starts by checking if the GPU is activated:

```
#@title Activating the GPU
# Main menu->Runtime->Change Runtime Type
import tensorflow as tf
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
    raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))
```

The output should be:

```
Found GPU at: /device:GPU:0
```

The program will be using Hugging Face modules.

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 feels 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  
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
```

No message will be displayed if all goes well, 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 Specify CUDA as device for Torch
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
n_gpu = torch.cuda.device_count()
torch.cuda.get_device_name(0)
```

The VM I ran on Google Colab displayed the following output:

```
'Tesla P100-PCIE-16GB'
```

The output may vary with Google Colab configurations.

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 4, Downstream NLP Tasks with Transformers*, we will explore the key tasks transformers must perform to prove their efficiency.

Use the Google Colab file manager to upload `in_domain_train.tsv` and `out_of_domain_dev.tsv`, which you will find on GitHub in the `Chapter02` directory of the repository of the book.

You should see them appear in the file manager:

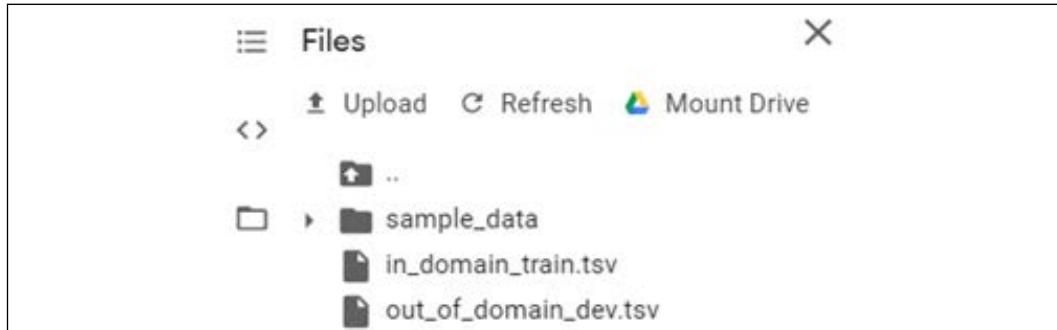


Figure 2.5: Uploading the datasets

Now the program will load the datasets:

```
#@title Loading the Dataset

```

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:

```
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 512 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 on those padded tokens!

The idea is to apply a mask with a value of `1` for each token, which will be followed by `0`s 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 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 huge datasets without crashing the memory of the machine.

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 pooler is there to convert the sequence tensor for classification tasks, which require a fixed dimension to represent the sequence. The pooler will thus convert the sequence tensor to (batch size, hidden size), which are fixed parameters.
- `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 6 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 Hugging Face Bert uncased base model
model = BertForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=2)
model.cuda()
```

This pretrained model can be trained further if necessary. It is interesting to explore the architecture in detail to visualize the parameters of each sub-layer 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/5bfcd0485ece086ebcbe
d2d008813037968a9e58/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 there is a measurement for loss and accuracy, which will be plotted. The training loop uses the dataloader 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:

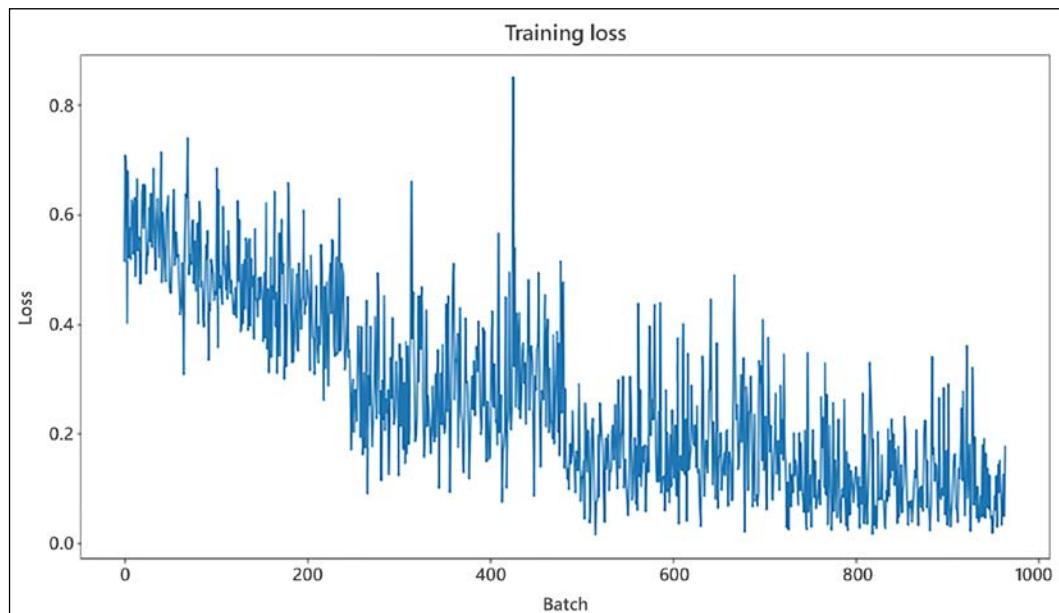


Figure 2.6: Training loss per batch

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 contained 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 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:

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

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 take a look at the scores on the individual batches to get a sense of the variability in the metric between batches.

The score of individual batches

Let's view the score 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)
```

The output confirms that the MCC is positive, which shows that there is a correlation for 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 Judgement* 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 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. OpenAI GPT had worked on this approach before, but BERT took it to another level!

In this chapter, we fine-tuned a BERT model. In the next chapter, *Chapter 3, Pretraining a RoBERTa Model from Scratch*, we will dig deeper into the BERT framework and build a pretraining 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, Łukasz 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: Pre-training 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.

3

Pretraining a RoBERTa Model from Scratch

In this chapter, we will build a RoBERTa model from scratch. The model will take 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 1, Getting Started with the Model Architecture of the Transformer*, we went through the building blocks of the original Transformer. In *Chapter 2, 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* books created for this chapter. We will see how the data was obtained. You will see how you will be able 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.

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

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 2, Fine-Tuning BERT Models*.

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

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

The initial BERT models were undertrained. RoBERTa increases the performance of pretraining transformers for downstream tasks. RoBERTa has improved the mechanics of the pretraining process. For example, it does not use **WordPiece** tokenization but goes down to byte-level **Byte Pair Encoding (BPE)**.

In this chapter, KantaiBERT, like BERT, will be trained using masked language modeling.

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 represent a large number of parameters. However, the parameters are spread over 6 layers and 12 heads, which makes it relatively small. 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. We know that large models provide excellent performance. But what if you want to run a model on a smartphone? Miniaturization has been the key to technological evolution. Transformers will 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, 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.

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 workpiece 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 a masked language modeling 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 then run it on a masked language modeling 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 cells of the notebook, which makes it 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 4, Downstream NLP Tasks with Transformers*, we will explore several datasets. However, the goal of this chapter is to understand the training process of a transformer with notebook cells that could be run in real time without having to wait 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:...

You can load `kant.txt`, which is in the directory of this chapter on GitHub using Colab's file manager. Or you can use `curl` 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/PacktPublishing/
Transformers-for-Natural-Language-Processing/master/Chapter03/kant.txt
--output "kant.txt"
```

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

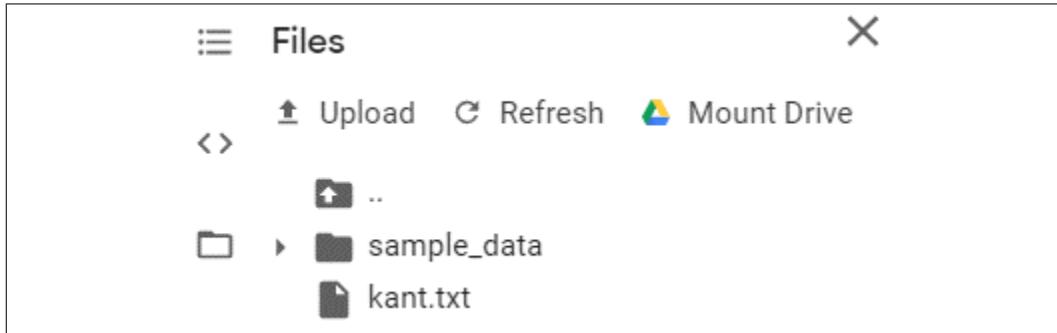


Figure 3.1: Colab file manager

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

The dataset is defined and loaded.



Note: 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 byte-level tokenizer will break a string or word down into a sub-string or sub-word. There are two main advantages 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, and we could obtain "sm" and "all," for example. In any case, the words are broken down into sub-word tokens and smaller units of sub-word parts such as "sm" and "all" instead of simply "small."
- The chunks of strings classified as an 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 sub-string 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:

'the', 'token', 'izer',

The string is now tokenized into tokens with whitespace information.

The next step is to replace them with their indices:

'the'	'token'	'izer'
150	5430	4712

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>"])
```

```
"<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 is 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 sub-strings
- `vocab.json`, which contains the indices of the tokenized sub-strings

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:

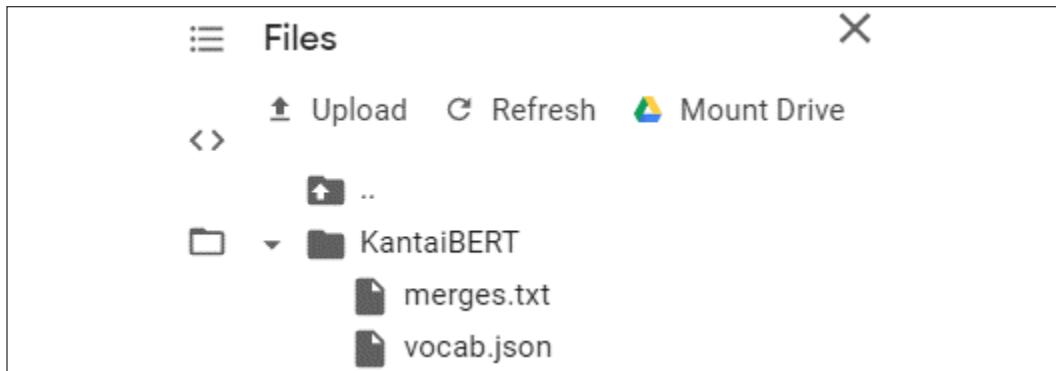


Figure 3.2: Colab file manager

The files in this example are small. You can double-click on them to view their contents.

`merges.txt` contains the tokenized sub-strings as planned:

```
#version: 0.2 - Trained by `huggingface/tokenizers`  
G t  
h e  
G a  
o n  
i n  
G o  
Gt he  
r e  
i t  
Go f
```

`vocab.json` contains the indices:

```
[..., "Gthink":955, "preme":956, "GE":957, "Gout":958, "Gdut":959, "aly":960, "Gexp":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', 'Critique', 'of', 'Pure', 'Reason', '.' ]
```

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', 'Critique', 'of', 'Pure', 'Reason', '.', '</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	P0	63W / 149W	9707MiB / 11441MiB	0%	Default	
Processes:							
GPU	PID	Type	Process name		GPU Memory Usage		

Figure 3.3: Information on the NVIDIA card

We will now check to make sure PyTorch sees CUDA:

```
#@title Checking that PyTorch Sees CUDA
import torch
torch.cuda.is_available()
```

The results should be True:

```
True
```

Compute Unified Device Architecture (CUDA) was developed by NVIDIA to use the parallel computing power of its NVIDIA card.

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 sub layers.

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 add to your understanding of 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 many 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 3.4: LP



Note that all of the numbers we are displaying might vary depending on the version of the transformers module we are using.

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 find out is to try and see if a parameter `p` in the list `LP[p]` has two dimensions or not:

```
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 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 into 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 **Masked Language Modeling (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 the loss, learning rate, epoch, and 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:

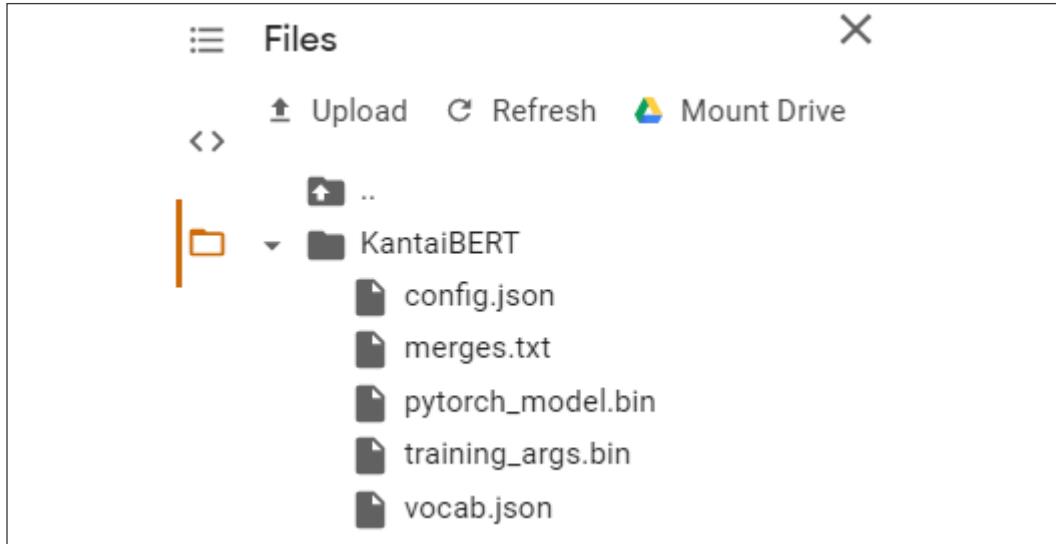


Figure 3.5: Colab file manager

`config.json`, `pytorch_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 masked language modeling:

```
#@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 conceptional 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 at 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 very interesting humanlike 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 the Transformer, 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 construction 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 obtain. 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.

In the next chapter, *Downstream NLP Tasks with Transformers*, we will discover yet another innovative architecture of 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 predicts a word contained in a mask token in a sentence. (True/False)
9. A BERT-like model has no self-attention sub-layers. (True/False)
10. Data collators are helpful for backpropagation. (True/False)

References

- *RoBERTa: A Robustly Optimized BERT Pretraining Approach* by Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyano: <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

4

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 355 million parameter transformer model in action on a range of NLU tasks.

We will begin this chapter with the quest to outperform 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 performances of transformers. Measuring 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, and the University of Washington 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. One day, another model will emerge, and the days of RNNs might be over for NLP tasks.

This chapter covers the following topics:

- Machine versus human intelligence for transduction and induction
- The NLP transduction and induction process
- Measuring transformer performances 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

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.

The following figure, a mind experiment designed in transformer architecture layers and sub-layers, shows the deceptive similarity between humans and machines. Let's study the learning process of transformer models to understand downstream tasks.

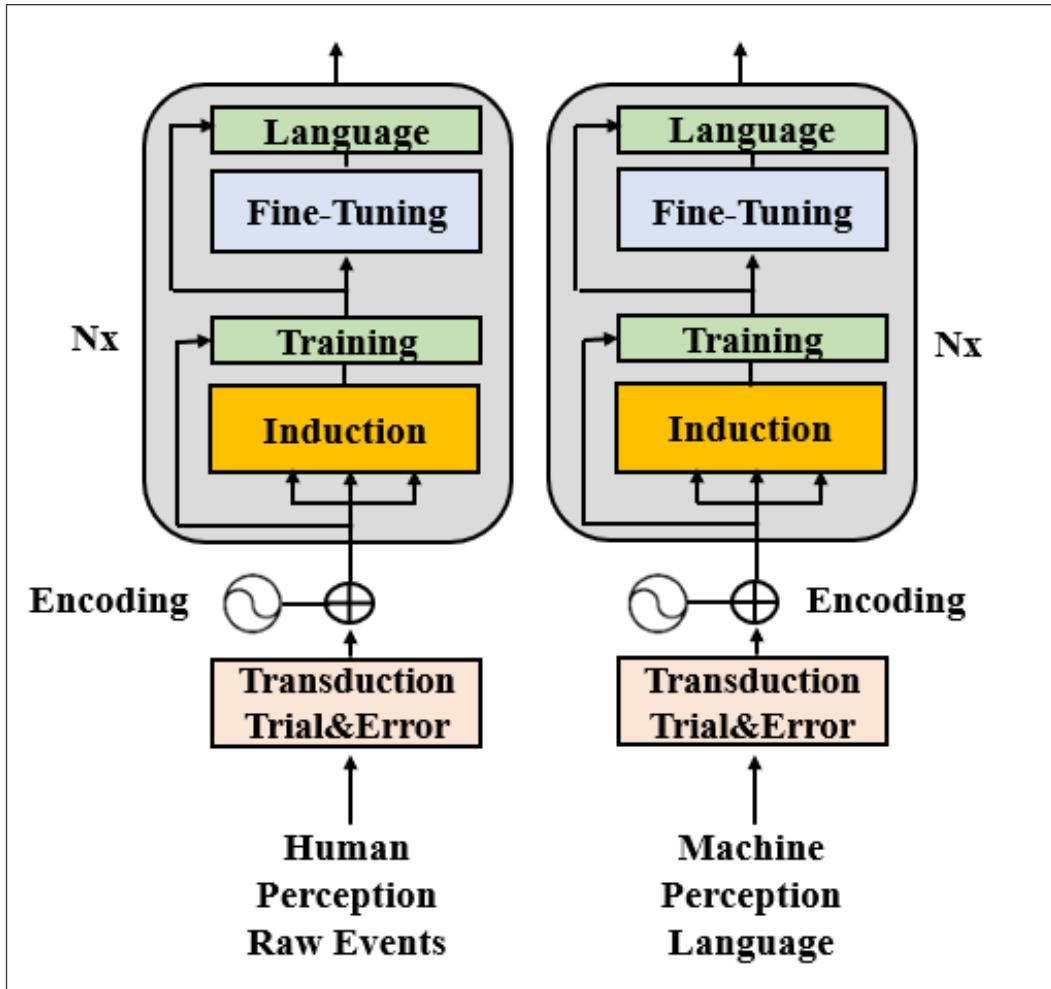


Figure 4.1: Human and machine learning methods

For our example, $N=2$. This conceptual representation has, thus, two layers.

The human intelligence stack

On the left side of *Figure 4.1*, 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 there 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 finetuned knowledge. On top of that, humans perceive massive amounts of events that then go through the transduction, induction, training, and fine-tuning sub-layers along with previous transcribed knowledge.

Our infinite approach loop goes from layer 0 to layer 1 and back to layer 0 with more and more raw and processed information.

The result is fascinating! We do not need to learn (train) 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

On the right side of *Figure 4.1*, 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 (sub-words) 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 sub-layer, which is working hard to produce valid inductions:

- The transformer model excluded the former sequence-based learning operations and used self-attention to heighten the vision of the model
- Attention sub-layers have an advantage over humans at this point: they can process millions of examples for their inductive thinking operations
- Like us, they find patterns through transduction and induction
- They memorize these patterns with 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



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 how we do. They start with a handicap because at 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 enable it 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, tasks can all be pretrained or fine-tuned tasks.

Every NLP model needs to be evaluated with a standard method.

In this section, we 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 y matches the correct predictions \hat{y} for a given subset $samples_i$ of a set of samples or not. The basic function is:

$$\text{Accuracy}(y, \hat{y}) = \frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} 1(\hat{y}_i = y_i)$$

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.

F1-score uses weighted values of precision and recall. It is a weighted average of precision and recall values:

$$\text{F1-score} = 2 * (\text{precision} * \text{recall}) / (\text{precision} + \text{recall})$$

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:

$$P = \frac{T_p}{T_p + F_p}$$

$$R = \frac{T_p}{T_p + F_n}$$

F1-score can thus be viewed as the harmonic mean (reciprocal of the arithmetic mean) of precision (P) and recall (R):

$$\text{F1 score} = \frac{PxR}{P + R}$$

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 2, Fine-Tuning BERT Models*. MCC computes a measurement with true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN).

The MCC can be summarized by the following equation:

$$\frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

MCC provides an excellent binary metric, 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 performances:

- A model
- A dataset-driven task
- A metric as described in the *Evaluating models with metrics* section of this chapter

We will explore 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 applied 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 average humans, as we can see in the GLUE leaderboard (June 2020). The GLUE leaderboard, <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 three leaders and the position of GLUE's Human Baselines:

Rank	Name	Model	URL	Score
1	HFL iFLYTEK	MacALBERT + DKM		90.7
2	Alibaba DAMO NLP	StructBERT + TAPT	↗	90.6
3	PING-AN Omni-Sinotic	ALBERT + DAAF + NAS		90.6
4	ERNIE Team - Baidu	ERNIE	↗	90.4
5	T5 Team - Google	T5	↗	90.3
14	GLUE Human Baselines	GLUE Human Baselines	↗	87.1

Figure 4.2: GLUE Leaderboard – December 2020



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 rank #14, which shows that NLU models have surpassed non-expert humans on GLUE tasks. This is a problem. Without a standard to try 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.

Finally, we can see that Baidu has entered the NLU race with interesting results. Sun et al. (2019) designed a transformer model named ERNIE that introduces continual incremental pretraining and multi-task fine-tuning. The results produced are impressive because of the wide range of pretraining and fine-tuning multi-task methods.



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 Baseline standards

Wang et al. (2019) saw the limits of GLUE. They designed SuperGLUE for more difficult NLU tasks.

SuperGLUE immediately re-established the Human Baseline as 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 4.3: SuperGLUE Leaderboard 2.0 – December 2020

The SuperGLUE leaderboard will evolve as we produce better NLU models. In early 2021, Transformers have already surpassed the Human Baselines and this is only a beginning. Notice the arrival of Huawei Noah's Art Lab with transformer models. Transformers are going global!



AI algorithm rankings will constantly change. 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 not only to 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
Recognizing 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

[DOWNLOAD ALL DATA](#)

Figure 4.4: SuperGLUE tasks

The task list is interactive: <https://super.gluebenchmark.com/tasks>.

Each task contains links to the required information to perform the 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** is available 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 the papers or websites describing the problem to solve. 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 4.5: 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, the top-ranking results are obtained by transformers that are already reaching SuperGLUE's Human Baselines level:

Rank	Name	Model	URL	Score	BoolQ	CB	COPA
1	SuperGLUE Human Baselines	SuperGLUE Human Baselines		89.8	89.0	95.8/98.9	100.0
2	T5 Team - Google	T5		89.3	91.2	93.9/96.8	94.8
3	Huawei Noah's Ark Lab	NEZHA-Plus		86.7	87.8	94.4/96.0	93.6
4	Alibaba PAI&ICBU	PAI Albert		86.1	88.1	92.4/96.4	91.8
5	Tencent Jarvis Lab	RoBERTa (ensemble)		85.9	88.2	92.5/95.6	90.8

Figure 4.6: COPA SuperGLUE transformer performances – December 2020

As incredible as it seems, transformers are climbing the leaderboard ladder in a very short time!

We have seen what is behind one task. 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 multi-tasks using a pretrained model and then applying it to fine-tuned downstream tasks. The Transformer model now leads in all of the GLUE and SuperGLUE tasks. We will continue to focus on SuperGLUE downstream tasks for which the human baseline 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 training dataset, `train.jsonl`, 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 to make. 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 look into two of them. The model has to 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 commonsense 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 if 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 Schemas 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 that 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 if 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 the eight 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 of all of these tasks is us!*

A downstream task is a fine-tuned transformer task that inherited 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 wasn't used to fully pretrain the model. In this section, we will consider all of 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_DR.ipynb` in *Chapter 2, 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 2, 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)

Standford 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 11, 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 a positive and negative movie review 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 if 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 (see the next section on STS-B)

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 the *The Winograd Schema Challenge (WSC)* section of this chapter. 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 word *car* or *garage*. Can a transformer disambiguate this pronoun?

Open `Transformer_tasks.ipynb`, go 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 *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

In this chapter, we analyzed the difference between the human language representation process and the way machine intelligence has to perform transduction. We saw that transformers must rely on the outputs of our incredibly complex thought process 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 the history of NLP. 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 must face 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 5, 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 if one sequence entails another sequence. (True/False)
6. A Winograd schema predicts if a verb is spelled correctly. (True/False)
7. Transformer models now occupy the top ranks of GLUE and SuperGLUE. (True/False)
8. Human Baseline 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 baseline 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 Pre-Training 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>*

5

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 has to 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 WMT dataset. Finally, we will see how to implement machine translations.

This chapter covers the following topics:

- Defining machine translation
- Human transduction
- Machine transduction
- Preprocessing a WMT dataset
- Evaluating machine translation with BLEU
- Geometric evaluations
- Chencherry smoothing
- Enabling eager execution
- 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 to design 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:

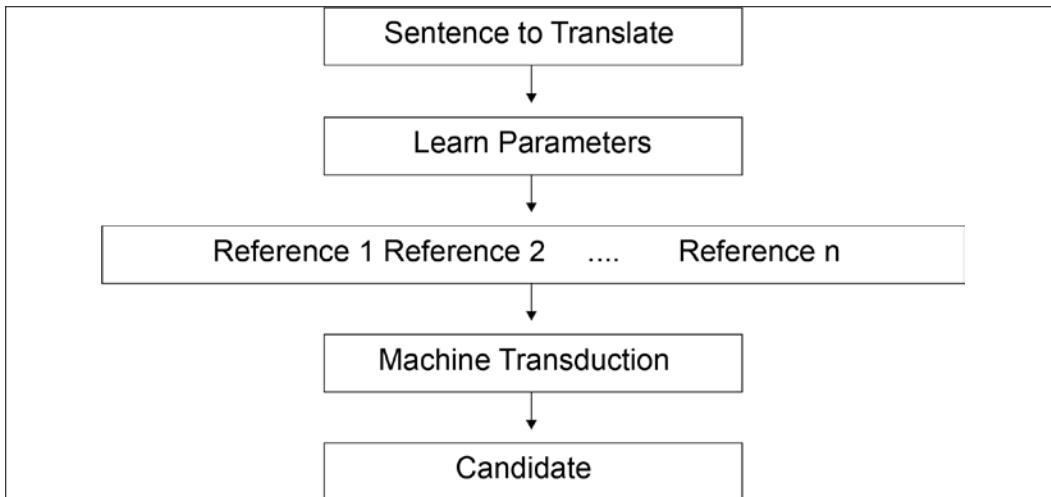


Figure 5.1: Machine translation process

The general idea in *Figure 5.1* 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 millions upon millions of parameters
- Learn the many ways 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 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 was 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 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 that there are several references in the *Machine translation process* described in *Figure 5.1*.

In real life, a human translator will not translate sentence *A* into sentence *B* several times but only once. However, in real life, more than one translator could translate sentence *A*. For example, you can find several English translations of *Les Essais* by Montaigne, written in French. 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:

$$\text{reference} = \{ \text{Reference 1}, \text{Reference 2} \dots \text{Reference } n \}$$

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 of 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.

Evaluation of machine translation proves that NLP has progressed. To determine that a solution is better than another one, each NLP challenger, each 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 **Workshop on Machine Translation (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*. The link is <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.

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 now need to be cleaned to avoid training useless and noisy tokens. The lines are normalized, tokenized on white spaces, and converted to lower case. 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 from 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.

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`. 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 if we use 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 that are 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, then save the output and 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, then save the output and 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 interrompu 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 a *candidate* translation in English 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 if we compared human translation to machine translation word by word, we could obtain efficient results.

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. Bear in mind 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 output 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.

Papineni et al. (2002) came up with a modified unigram approach. The idea was to count the word occurrences in the reference sentence and make sure the word was not over evaluated in the candidate sentence.

Consider the following example *Papineni et al. (2002)* explained:

```
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
```

For example, 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.

Smoothing is a very efficient method. BLEU smoothing can be traced back to label smoothing, applied to softmax outputs in the Transformer.

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 $1 - \varepsilon$.
- Increase the 0 values by $0 + \frac{\varepsilon}{k - 1}$.

We obtain the following result if we apply this approach:

`candidate_smoothed=[0.75, 0.25, 0.25, 0.25]` making the output open to future transformations and changes.

The Transformer uses variants of label smoothing.

A variant for 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 smoothing 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 voussvp 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 seen how a dataset is preprocessed and how BLEU evaluates machine translations.

Let's 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 examples.

Though T2T was a good start, Google Brain 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.

In this section, we 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 a Transformer model

We will create the original Transformer model as described in *Chapter 1, Getting Started with the Model Architecture of the Transformer*.

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 the *Preprocessing a WMT dataset* section of this chapter.

The sentence will now be tokenized:

```
#@title Tokenize 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 the *Defining machine Translation* section of this chapter.

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
Maschinübersicht.
```

The Transformer translated "machine intelligence" into "Maschinübersicht."

If we deconstruct "*Maschiniübersicht*" into "*Maschin*"(machine) + "*übersicht*"(intelligence), we can see that:

- "*über*" literally means "over."
- "*sicht*" means "sight" or "view."

The Transformer is telling 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 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 a very high baseline for machines to reach. We saw that English-French and English-German translations imply quite an amount of 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 to score translations but even modified BLEU has its limits. We thus added a smoothing technique to enhance BLEU.

Finally, 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, *Text Generation with OpenAI GPT-2 and GPT-3 Models*, we will discover another way of assembling a transformer 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://www.aclweb.org/anthology/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, Łukasz Kaiser, 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/>

6

Text Generation with OpenAI GPT-2 and GPT-3 Models

In 2020, *Brown et al.* (2020) described the training of an OpenAI GPT-3 model with 175 billion parameters trained with approximately one trillion words in 50 petaflop/s days. This represents about 50×10^{20} operations per day for 400 billion byte-pair-encoded tokens. At the same time, we learned that OpenAI had access to a tailor-made supercomputer that contained 280,000 CPUs and 10,000 GPUs.

A new era had begun. A battle of giants had begun with the recent ground-breaking intelligence of transformers and the power of supercomputers. Microsoft, Google, Facebook, Baidu, IBM, and others produce game-changing AI resources several times a year. AI project managers and developers need to continually reinvent a way to understand, tame, and implement these mind-blowing innovations.

The machine intelligence of OpenAI GPT-3 and supercomputers' machine power led *Brown et al.* (2020) to zero-shot experiments. The idea was to use a trained model for downstream tasks without training the parameters any further. The goal would be for a trained model to go directly into multi-task production. OpenAI decided to restrict the use of GPT-3 models to specific users. The future of AI could well be limited to cloud users. The size and power of GPT-2 and GPT-3 seem to have taken NLP to another level. However, it might not be the only path to increasing the performance of transformers.

This chapter will first examine the evolution of size and transformer models from a project management perspective. Can we accept a future in which we can only implement artificial intelligence using Cloud AI models? Is this assessment correct? Should we even consider GPT models? We will see if the Reformer architecture or the **Pattern-Exploiting Training (PET)** method challenges the assertion that huge models and supercomputers are the only future we have.

Once we have decided to explore GPT models, we will look into the zero-shot challenge of using trained transformer models with little to no fine-tuning of the model's parameters for downstream tasks. We'll explore the innovative architecture of GPT transformer models.

We will then implement a 345M parameter GPT-2 transformer in TensorFlow using OpenAI's repository. We will interact with the model to produce text completion with standard conditioning sentences. We need to understand GPT-2 like any other transformer model to be able to make the right choices at the right time.

However, we need to go further. So finally, we will build a 117M customized GPT-2 model. We will tokenize the high-level conceptual Kant dataset we used to train the RoBERTa model in *Chapter 3, Pretraining a RoBERTa Model from Scratch*. This time we will train the dataset with GPT-2 models. We will interact with our trained model to obtain rather surprising human baseline level outputs.

By the end of the chapter, you will be able to train GPT-2 models on your custom data and interact as you wish with a machine whose intelligence is growing by the day.

This chapter is based on the knowledge acquired from chapters one through five. Take the necessary time to go through them again to make sure you have the main aspects of the models and the evaluation benchmarks in mind before reading this chapter.

This chapter covers the following topics:

- The limits of the original Transformer architecture
- How the Reformer may solve the limits of the Transformer
- How PET might solve the limits of training transformers
- Defining zero-shot transformer models
- The path from few-shots to one-shot
- GPT-2 and GPT-3 models
- 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

Let's begin by taking a look at the progression of transformers over the last few years.

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 model (GPT)** that 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 do 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 on the SuperGLUE leaderboards as this book is written.

How did this happen so quickly?

To understand how such evolution happened, we will look first at one aspect of this evolution through the models' sizes.

The increasing size of transformer models

From 2017 to 2020 alone, the number of parameters increases from 65M parameters in the original Transformer model to 175B parameters in the GPT-3 model, as shown in *Table 6.1*:

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 6.1: The evolution of the number of parameters of transformers

Table 6.1 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 and the dates. For example, once the original Transformer set the market in motion, transformers emerged from Google Brain and Research, OpenAI, and Facebook AI produced new models in parallel.

Furthermore, some sizes of 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.

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 that is presented to us.

However, the drawback of context size is the distance it takes to understand what a word refers to. The path it takes to analyze long-term dependencies requires a change 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 maximum path length can be summed up as shown in *Table 6.2* in Big O notation:

Layer Type	Maximum Path Length	Context Size
Self-Attention	$O(1)$	1
Recurrent	$O(n)$	12288

Table 6.2: 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 context size of 12,888 tokens has the same maximum length path as the context size of 512 tokens of the Transformer Base model.

A recurrent layer, in an RNN, for example, 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 12,288 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 nearly a trillion words using the Common Crawl dataset.
- Training large transformer models requires machine power that is only available to a limited number of teams in the world. *Vaswani et al. (2017)* trained the Transformer Big model with 213 million parameters consuming 2.3×10^{19} FLOPs. GPT-3 was trained in 50 petaflop/s-days!
- Designing the architecture of transformers requires highly qualified teams that can only be funded by a limited number of organizations in the world.

The size and architecture will continue to evolve and increase. Supercomputers will continue to provide the necessary resources to train transformers.

Before going through the main aspects of OpenAI GPT models, let's pause and look into the choices we have between transformers, reformers, a PET approach, or GPT models.

Transformers, reformers, PET, or GPT?

Before using GPT models, we need to stop and look at transformers from a project management perspective at this point in our book's journey. Which model and which method must we choose for a given NLP project? Should we trust any of them?

Once we consider *cost management, accountability* follows, and choosing a model and a machine become life-and-death decisions for a project. In this section, we will stop and think before entering the world of the recent GPT-2 and huge GPT-3 (and more may come) models.

We have successively gone through:

- The original architecture of the Transformer with an encoder and a decoder stack in *Chapter 1, Getting Started with the Model Architecture of the Transformer*.

- Fine-tuning a pretrained BERT model with only an encoder stack and no decoder stack in *Chapter 2, Fine-Tuning BERT models*.
- Training a RoBERTa-like model with only an encoder stack and no decoder stack in *Chapter 3, Pretraining a RoBERTa Model from Scratch*.
- The main NLP tasks in *Chapter 4, Downstream NLP Tasks with Transformers*.
- The important translation task in *Chapter 5, Machine Translation with the Transformer*.
- And now we are faced with the perspective of using the huge *decoder-stack-only GPT-3 models* on a Cloud AI platform in the future.

Project management best practice compels us to examine the prospect of only using a GPT-3 transformer model and its subsequent versions on a billable cloud AI platform such as Microsoft Azure. A project manager can easily see how billable cloud servers, such as preinstalled VMs, can be convenient to outsource the use of powerful machines at a reasonable price.

However, being forced to abandon the idea of controlling our own transformers and only use a third-party billable GPT transformer, for example, is something to consider before making this decision.

In this section, before using a GPT model, we will examine:

- The limits of the original Transformer model.
- The Reformer solution to the possible limits of the architecture of the Transformer model.
- The PET solution to training a model.

Let's begin by looking into the limits of the original Transformer architecture.

The limits of the original Transformer architecture

The possible limits of the original Transformer model are linked to memory problems leading to more machine power.

Let's visualize the attention heads to get a pragmatic view of the Transformer model's limits.

Open `head_view_bert.ipynb` to implement `BertViz`, designed by *Jesse Vig*, to visualize attention heads in several transformer models, such as the Transformer, BERT, GPT-2, and RoBERTa, and more. We'll run a BERT model since any model will suffice to see the limits of the Transformer model.

Running BertViz

It only takes four steps to visualize transformer attention heads, interact with them, and understand the limits of the Transformer model.

Let's first install `BertViz` and the requirements.

Step 1: Installing BertViz

The notebook installs `BertViz`, Hugging Face transformers, and the other basic requirements to implement the program:

```
#@title Step 1: Installing BertViz and Requirements
import sys
!test -d bertviz_repo && echo "FYI: bertviz_repo directory already
exists, to pull latest version uncomment this line: !rm -r bertviz_
repo"
# !rm -r bertviz_repo # Uncomment if you need a clean pull from repo
!test -d bertviz_repo || git clone https://github.com/jessevig/bertviz
bertviz_repo
if not 'bertviz_repo' in sys.path:
    sys.path += ['bertviz_repo']
!pip install regex
!pip install transformers
```

We will now import the necessary modules.

Step 2: Importing the modules

`BertViz` has made the modules seamless to import:

```
#@title Step 2: Import BertViz Head Views and BERT
from bertviz import head_view
from transformers import BertTokenizer, BertModel
```

And that's it! We are ready to prepare the HTML visualization interface.

Step 3: Defining the HTML function

`BertViz` is now ready to go.

The notebook implements a standard IPython HTML function:

```

#@title Step 3: Defining the HTML Function
def call_html():
    import IPython
    display(IPython.core.display.HTML('''
        <script src="/static/components/requirejs/require.js"></script>
        <script>
            requirejs.config({
                paths: {
                    base: '/static/base',
                    "d3": "https://cdnjs.cloudflare.com/ajax/libs/d3/3.5.8/
d3.min",
                    jquery: '//ajax.googleapis.com/ajax/libs/jquery/2.0.0/
jquery.min',
                },
            });
        </script>
    '''))

```

We are now ready to process and display attention heads.

Step 4: Processing and displaying attention heads

Let's now process a translation to display the attention heads. We could use any transformer model or any task. We would still reach the same conclusions.

The notebook uses a pretrained BERT, processes the sentences to be translated, and displays the attention head activity:

```

#@title Step 4: Processing and Displaying Attention Heads
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)

sentence_a = "The cat sleeps on the mat"
sentence_b = "Le chat dors sur le tapis"
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]
input_id_list = input_ids[0].tolist() # Batch index 0
tokens = tokenizer.convert_ids_to_tokens(input_id_list)
call_html()

head_view(attention, tokens)
```

The output displays an interactive HTML interface showing attention head activity:

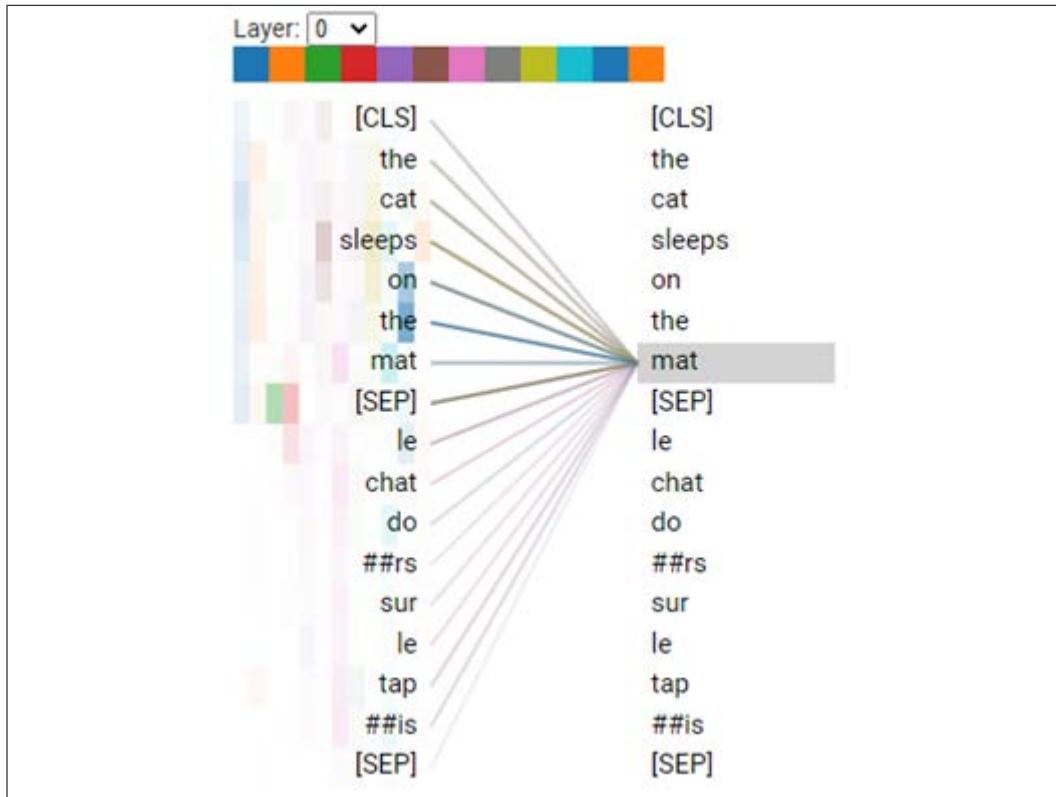


Figure 6.1: Visualizing Attention Heads

Take a few minutes to play around with the attention heads. Go through the 12 layers (select a layer in the drop-down list). Hover over each word, look at the connections and 12 attention heads (the little squares next to each word).

Our empirical experimentation with the attention heads leads to some critical conclusions:

- The attention process takes all possible pairs of words into account to learn the connections between them. The larger the context window, the more pairs will be analyzed.
- If a text is 100K words long, this leads to 100K times 100K word pairs. That translates into 10 billion pairs for each step! The computer power to achieve this process is mind-blowing and requires supercomputers to achieve acceptable performances.
- The number of layers leads to substantial memory requirements to store information, including bloating feedforward layers that reach terabytes for models containing thousands of layers.
- One reason we might analyze long sequences is for transformer music generation, which you can listen to in the *Generating music with transformers* section of this chapter.

Before using substantial computer resources, a standard project management process is to examine several solutions at the algorithm architecture level or training level.

Google AI came up with the Reformer, which might be one of the possible solutions.

The Reformer

Kitaev et al. (2020), <https://arxiv.org/abs/2001.04451>, designed the Reformer to solve the attention issue and the memory issue, adding functionality to the original Transformer model. The approach is interesting though there are other ways to solve the performance issues of transformers using distillation, for example, as explained in the *Pattern-Exploiting Training (PET)* section of this chapter.

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 the $\text{hash}(q) == \text{hash}(p)$. 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 this example) in a process called LSH bucketing, just like 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 this example) to parallelize. Finally, attention will only be applied within the same bucket in its chunk and the previous chunk.

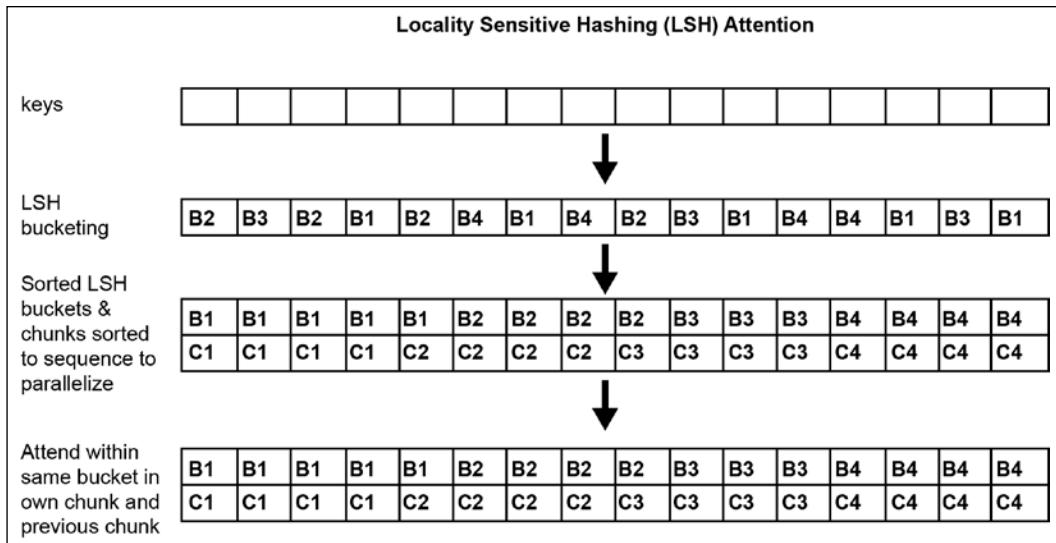


Figure 6.2: LSH attention heads

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.

You can explore documentation and code of the Reformer model further on Hugging Face: https://huggingface.co/transformers/model_doc/reformer.html?highlight=reformer

However, bear in mind that the Reformer is not the silver bullet to the Transformer's limits.

This section went through the Reformer's main concepts that can bring large training models down to a few GB of memory and only one GPU. However, is that the solution? Or is PET a better approach to train transformer models.

Let's first understand what PET is before making a decision.

Pattern-Exploiting Training (PET)

OpenAI created large models such as GPT-3 and Google AI optimized the Transformer with the Reformer. But what if Google AI and OpenAI got it all wrong?

Timo Schick and Hinrich Schütze wrote a paper that seriously challenges Google AI and OpenAI's approach.

The title of the paper *Schick and Schütze* (2020) wrote speaks for itself:

It's Not Just Size That Matters: Small Language Models Are Also Few-Shot Learners:

<https://arxiv.org/abs/2009.07118>

Schick and Schütze contend that a 223 million parameter transformer model outperformed a GPT-3 175 billion parameter model on the SuperGLUE leaderboard. With a transformer model that is only a fraction, about only 0.001, of the gigantic GPT-3 model, *Timo Schick and Hinrich Schütze* obtained good results with a single GPU and 11 GB of RAM.

That is quite a performance on the part of the Center for Information and Language Processing at the Ludwig Maximilian University of Munich! This progress is quite surprising coming from a relatively small research center compared to Google AI and OpenAI backed by Microsoft.

Timo Schick achieved rank #9 on the SuperGLUE leaderboard and GPT-3 only rank #12:

9	Timo Schick	iPET (ALBERT) - Few-Shot (32 Examples)		75.4
10	Adrian de Wynter	Bort (Alexa AI)		74.1
11	IBM Research AI	BERT-mtl		73.5
12	Ben Mann	GPT-3 few-shot - OpenAI		71.8

Figure 6.3: Leaderboard version: 2.0 December 2020

The rankings on the SuperGLUE leaderboard continuously change. But at one point in time, *Timo Schick and Hinrich Schütze* made an exceptionally strong statement in the history of **Language Models (LMs)**.

Note that the bidirectional ALBERT model with PET performs better than the GPT-3 unidirectional model.

PET appears to be a method to be taken into account. Let's go through the basic concepts.

The philosophy of Pattern-Exploiting Training (PET)

PET, as described by *Schick and Schütze (2020)*, relies on one core principle:

Reformulate a training task as a cloze question.

The reformulation of training tasks to optimize the training process enhances the transformer model's performance while reducing the size of both the model and the datasets.

We all encountered cloze tasks in school, such as:

Fill the blank with a noun:

I live in a ____.

The correct answer is: house

Cloze questions are a natural fit for transformers, which train using masked tokens.

Let:

- M be a masked language model named *MLM*
- T be the vocabulary of the *MLM*
- $_ \in T$ be the mask token

The process by which PET maps inputs to outputs requires a set of **pattern verbalizer pairs (PVPs)**.

Each PVP pair contains:

- A *pattern P* that converts (maps) inputs to cloze questions containing a single mask.
- A *verbalizer v* that converts (maps) each output to a single token.

With this information, PET aims to determine an output y is the correct one for an input x .

PET will thus determine the probability of $v(y)$ of being the correct token at the masked position $P(x)$.

PET will use cross-entropy to fine-tune its process.

We can see that this process can be associated with *knowledge distillation*, which takes large models and converts them into smaller ones. A variant of PET is iPET, an iterative process by which the models train using datasets that increase in size each generation using the labels produced by the previous generation.

Distillation through iPET has proven its efficiency with a relatively small ALBERT model that obtained better results than the gigantic GPT-3 model on the SuperGLUE leaderboard.

PET is available on GitHub:

<https://github.com/timoschick/pet>

We can see that PET introduces distillation in the training process, reducing the need for both large transformer models and machine power.

It's worth taking into account when deciding what architecture to design for a project.

It's time to make a decision

What will a project manager's decision be? We have seen the limits of the original Transformer model, which leads to the crossroads where we have to choose a path to:

- Accept the limits of the original Transformer model and move on to huge models requiring huge machine memory and computing power.
- To refuse the limits of the original Transformer and tweak its architecture with reformer-type approaches.
- Use different training methods such as PET, an efficient knowledge distillation approach.
- Use a combination of these approaches.
- Design your own training methods and model architecture.



There are many transformer model methods continuously appearing on the market. Take the necessary time to find the right path for your project.

In real-life project management, each approach will be carefully evaluated using standard evaluation parameters:

- The cost of each solution
- The efficiency of each solution
- The human and machine resources to implement the project
- The time-to-market and time-to-production

We cannot rule out any of the solutions before carrying out a careful study of a project's goals and costs. Each project will have a life of its own, and there is no predestined path.

We might need a full-blown GPT-2 or GPT-3 model to reach our goals.

Let's now go through the main aspects of OpenAI GPT models.

The architecture of OpenAI GPT models

Transformers went from training, fine-tuning, and finally zero-shot models in less than 3 years between the end of 2017 and the first semester 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 understand the motivation of the OpenAI team that designed GPT models. We will begin by going through the fine-tuning to 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.

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. 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, only to fine-tune the model's supervised learning.

OpenAI opted for a 12-layer decoder-only transformer we will describe in the Stacking decoder layers section of this 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 rapidly led Radford et al. (2019) to start thinking of 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:

$$\text{Examples} = (x_1, x_2, x_3, \dots, x_n)$$

The examples are composed of sequences of symbols:

$$\text{Sequences} = (s_1, s_2, s_3, \dots, s_n)$$

This led to a metamodel that can be expressed as a probability distribution for any type of input:

$$p(\text{output} / \text{input})$$

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 share 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 be able to produce excellent results with little to no fine-tuning for downstream tasks:

$$p(\text{output} / \text{multi-tasks})$$

OpenAI was reaching its goal to train a model and then run 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 next 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 yet 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 to force the transformer to think with machine intelligence. Machine intelligence, though not human, is efficient.

We understand the motivations that led to the architecture of GPT models. Let's have a look at the decoder-layer-only GPT model.

Stacking decoder layers

We now understand that the OpenAI team focused on language modeling. It makes sense to keep the masked attention sub-layer. Hence the choice to retain the decoder stacks and exclude the encoder stacks. To reach excellent results, *Brown et al.* (2020) dramatically increased the size of the decoder-only transformer models.

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 1, Getting Started with the Model Architecture of the Transformer*. 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 6.1*:

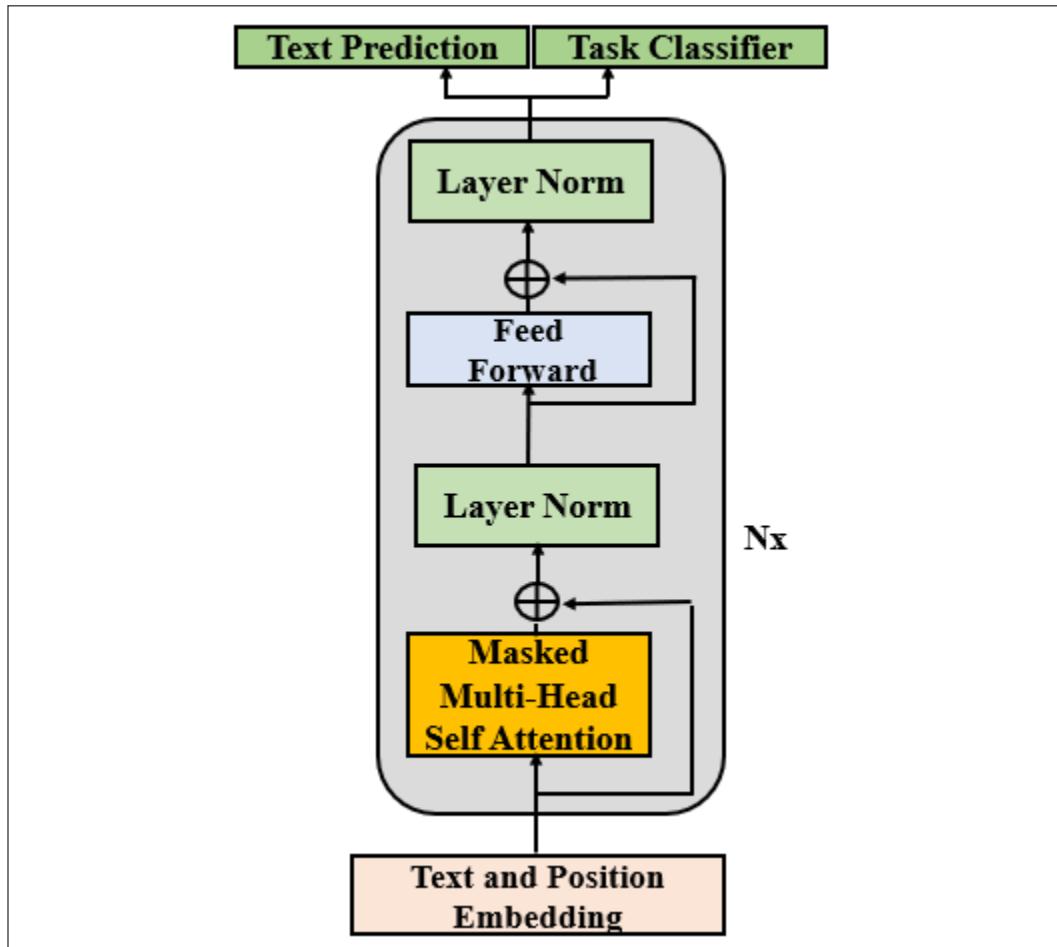


Figure 6.4: GPT decoder-only architecture

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. 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 less than four GPT models, and *Brown et al. (2020)* described no less 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:

$$n_{params} = 175.0B, n_{layers} = 96, d_{model} = 12288, n_{heads} = 96$$

This chapter will first use a trained GPT-2 345M model for text completion with 24 decoder layers with self-attention sub-layers 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.

We have explored the process that led us from fine-tuning to zero-shot GPT-3 models. Although the GPT-3 models are only available to a few users worldwide, 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 then train it. Let's start by interacting with the 345M parameter GPT-2 model.

Text completion with GPT-2

This section 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 9 steps. Open `OpenAI_GPT_2.ipynb` in Google Colaboratory. The notebook is in the chapter of the GitHub repository of this book. You will notice that the notebook is also divided into the same 9 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.*



It is important to note that we are running a low-level GPT-2 model and not a one-line call to obtain a result. We are also avoiding pre-packaged versions. We are getting our hands dirty to understand the architecture of a GPT-2 from scratch. You might get some deprecation messages. However, the effort is worthwhile.

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. We do not have open-source access to OpenAI's larger models such as GPT-3 at the time of this book's writing. OpenAI has begun to offer transformers as a cloud service. However, in the PET section of this chapter, we saw that we might be able to run small models with standard machines without going through a cloud service to run a powerful transformer GPT-3.

We will use the GPT-2 model in this section but will not train it. We could not train large GPT models even if we had access to the source code because most of us lack the computing power to do it. Vaswani et al. (2017) already used 8 P100 GPUs to train the first "big" 213M parameter Transformer model. We would need petaflops with the more recent transformer models!

An average developer does not have access to this level of machine power. Google Cloud, Microsoft Azure, **Amazon Web Services (AWS)**, for example, can rent a certain level of machine resources in teraflops to cloud customers.

If we go a step further, it becomes tougher to train transformers in teraflops. Accessing petaflops calculation power is limited to a restricted number of teams in the world.

However, we will see that the results produced by the limited power of Google Colaboratory VMs for our 345M parameter GPT-2 are quite convincing.

Now that we know more about what it takes to train large present-day transformer models, let's activate the GPU in the **Notebook settings** to get the most out of the VM:

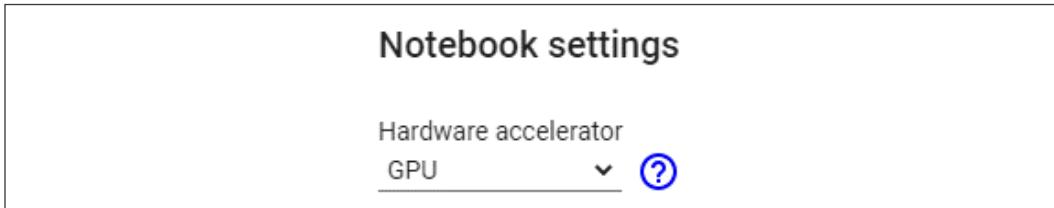


Figure 6.5: The GPU hardware accelerator

We can see that activating the GPU is a prerequisite for better performance that will give us access to the world of GPT transformers. Let's now clone the OpenAI GPT-2 repository.

Step 2: Cloning the OpenAI GPT-2 repository

OpenAI is still letting us download GPT-2. This may be discontinued in the future, or maybe we will have access to more resources. At this point, the evolution of transformers and their usage moves so fast 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:

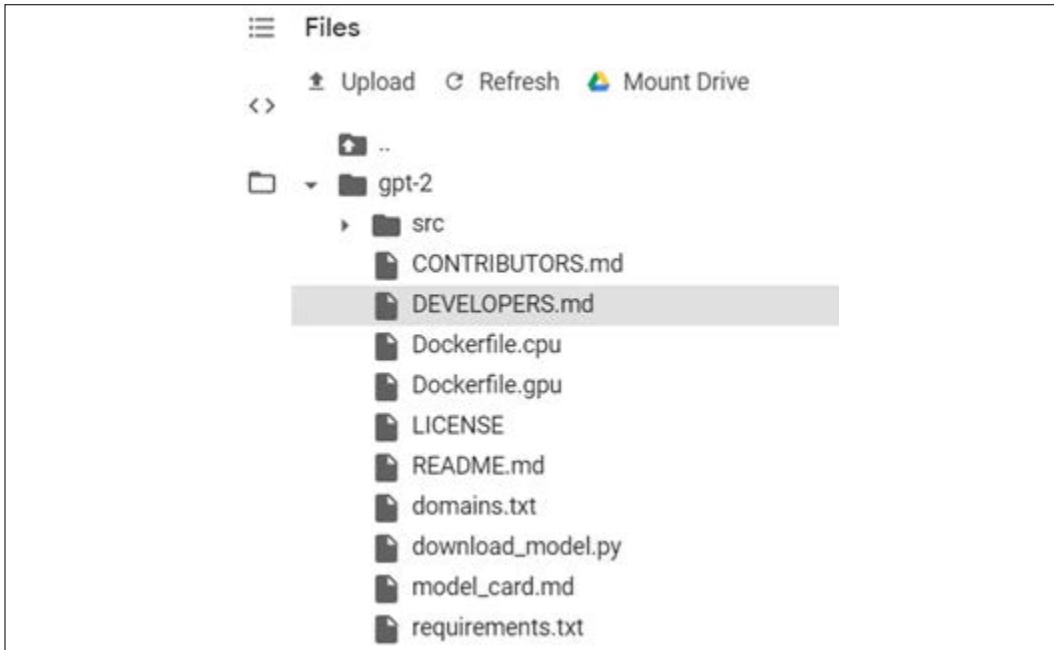


Figure 6.6: Cloned GPT-2 repository

Click on **src**, and you will see that the Python files we need from OpenAI to run our model are installed:

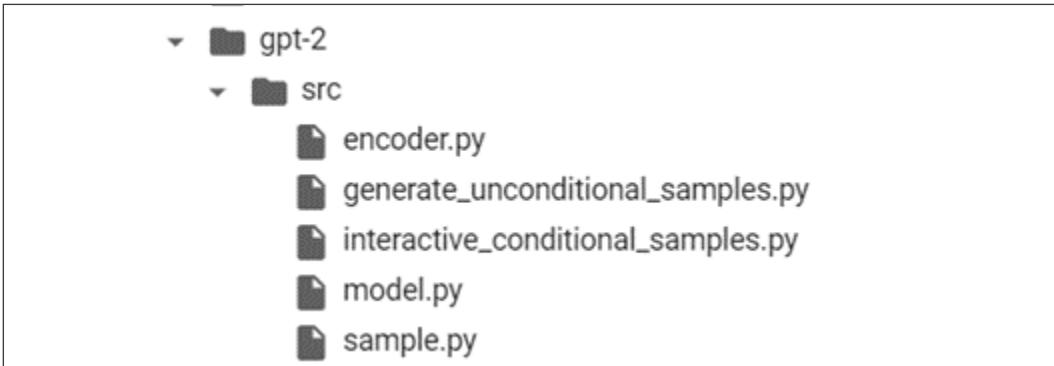


Figure 6.7: The GPT-2 Python files to run a model

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 this chapter.

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 transformer 345M transformer model provided by OpenAI uses TensorFlow 1.x. This will lead to several warnings when running the program. We will ignore them and run at full speed on the thin ice of training GPT models ourselves with our modest machines. In 2020, GPT models have reached 175 billion parameters, making it impossible for us to train them without having access to a supercomputer.

The corporate giants' research labs, such as Facebook AI and OpenAI and Google Research/Brain, are speeding towards super-transformers and are leaving us with what they can for us to learn and understand. They do not have time to go back and update all of the models they share.

This is one of the reasons for which 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 version `tf 1.x` is displayed or not, rerun the cell to make sure, then restart the VM. *Rerun this cell to make sure before continuing.*

If you encounter a TensorFlow 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 explored the GPT models in the *The architecture of OpenAI GPT models* section of this chapter. We will now download a 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:

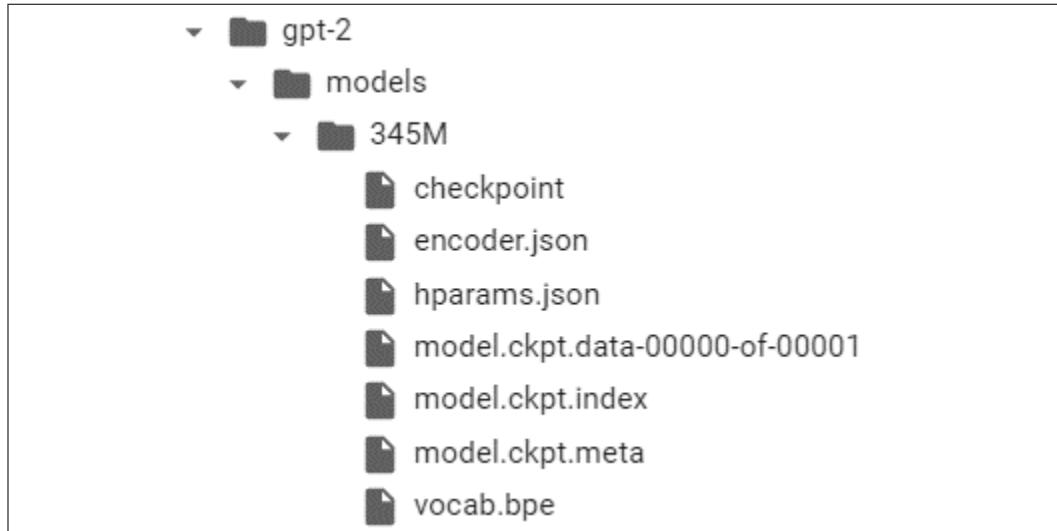


Figure 6.8: The GPT-2 Python files of the 345M parameter model

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 subsection *Step 3: Training a tokenizer* in Chapter 3, *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 *Training a GPT-2 language model* section of this chapter.

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 of 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 this chapter. 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 of 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 seeing an image with 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. 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`, the sample sequence parameters from `sample.py`. It will encode and decode sequences with `encoder.py`.

`interactive_conditional_samples.py` will then restore the checkpoint data defined in *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 117M 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. Anyway, if new GPT models are made available to us, we might need to run them on very powerful cloud computers.

In the meantime, 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 6.9: 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 more or less random since the GPT-2 model was not trained on our dataset.

Let's have a look at the first few lines the GPT model generated:

```
"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 type *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 very rich, and we can observe several facts:

- The context we entered *conditioned* the output generated by the model.
- The context was a demonstration for 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.

Let's see if we can obtain more impressive results by training the model on a customized dataset.

Training a GPT-2 language model

This section 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 3, Pretraining a RoBERTa Model from Scratch*.

This section refers to the code of `Training_OpenAI_GPT_2.ipynb`, which is in this chapter's directory of the book on GitHub.



It is important to note that we are running a low-level GPT-2 model and not a one-line call to obtain a result. We are also avoiding pre-packaged versions. We are getting our hands dirty to understand the architecture of a GPT-2 from scratch. You might get some deprecation messages. However, the effort is worthwhile.

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 chapter directory of the GitHub repository of this book:

- Activate the GPU in the notebook's runtime menu as explained in *Step 1* of the *Text completion with GPT-2* section of this chapter.
- 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 `gpt-2-train_files` directory that is in the GitHub repository of this book.

The *N Shepperd* GitHub repository provides the necessary files to train our GPT-2 model. We will not clone *N Shepperd*'s repository. 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 in-built file manager. The dataset is named `dset.txt` so that you can replace its content without modifying the program with your customized inputs after you have read this chapter.

This dataset is in the `gpt-2-train_files` directory that is in the GitHub repository of this book. It is the `kant.txt` dataset used in *Chapter 3, 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 briefly go through *Steps 2 to 6* since we described them in previous sections of this chapter. We will then copy the dataset and the model to the project directory.

Each step is the same step as the one described in the *Text completion with GPT-2* section of this chapter.

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. Wait until you have checked the TensorFlow version to restart the VM only once during your session. Then restart it if necessary.

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 of 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 section. We will now copy them into our project directory:

```
#@title Step 7: Copying the N Shepperd Training Files
#Reference 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 `encoder.py` to display the file in Google Colaboratory.

`encoder.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)
```

`encoder.py` also loads OpenAI's encoding program, `encoder.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`. `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 the 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 *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 6.10: 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.

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* subsection of the *Text completion with GPT-2* section of this chapter.

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!

Context and completion examples

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 type *Ctrl + M*.

The result is random but makes sense from a grammatical perspective. From a semantic point of view, the result is not as 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=ieKZP4CSg4GVWD
SJtwQczgTWQhAWB07+tKWh0jzz7o6rP41Ey&ssl=cTheory%20issue1&fastSource=po
ts&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".
```

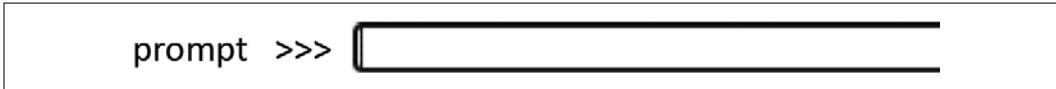
The result of an unconditional text generator is interesting but not convincing.

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 8* of the *Text completion with GPT-2* section of this chapter. The program prompts us to enter the context:



prompt >>>

Figure 6.11: Context input for text completion

Let's enter the same paragraph written by *Emmanuel Kant* as we did in *Step 9* of the *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. 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 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 that is 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 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!

You can also modify the text completion parameters as explained in *Step 8* of the *Text completion with GPT-2* section of this chapter.

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. GPT-2 expects the best out of us, as in real life!

We have successfully interacted with a GPT-2 model. OpenAI's GPT-3 model is much larger and produces exciting results in many domains.

Before we go, let's listen to some music generated by transformers.

Generating music with transformers

Before we go, experience how language modeling leads to quite exciting music generation with transformers.

Google AI's Music Transformer uses transformers to create music.

Click on the following link and take a few minutes to enjoy the samples:

<https://magenta.tensorflow.org/music-transformer>

Amazon's AWS DeepComposer also uses transformers to create music:

<https://aws.amazon.com/blogs/machine-learning/using-transformers-to-create-music-in-aws-deepcomposer-music-studio/>

AWS DeepComposer has a virtual keyboard to input music sequences.

Musicians can now create music and use transformers to explore new horizons, provide inspiration, and enhance their artistic experience.

It's now time to conclude this groundbreaking chapter and explore more transformer territory.

Summary

In this chapter, we discovered the new era of transformer models training 100,000,000,000+ parameters on supercomputers. OpenAI's GPT models are taking NLU beyond the reach of most NLP development teams.

We first examined transformer models from a project management perspective to see if transformers can be designed to use only one GPU, for example, and remain accessible to all. We saw that by optimizing a transformer model's architecture (Reformer) and training methods such as PET, we could reduce the model's size, requiring less machine power.

We then explored the design of GPT models, which are all built on the decoder stack of the original Transformer. 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 extremely 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 go 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.

Finally, we went further and trained a 117M parameter GPT-2 model on a customized dataset. The interactions with this relatively small trained model produced fascinating results. We can easily imagine how zero-shot models will function in the future.

Does this mean that in the future, users will not need AI NLP/NLU developers anymore? Will users simply upload the task definition and input text to cloud Transformer models and download the results?

The truth may lie in outsourcing general multi-tasks to cloud models and working on task-specific models when necessary. 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 input and produce completion content. (True/False)
8. We cannot interact with a 345M GPT parameter 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

- Reference BertViz GitHub Repository by Jesse Vig: <https://github.com/jessevig/bertviz>
- Google AI Blog on the Reformer: <https://ai.googleblog.com/2020/01/reformer-efficient-transformer.html>

- *Nikita Kitaev, Łukasz Kaiser, Anselm Levskaya, 2020, Reformer: The Efficient Transformer:* <https://arxiv.org/abs/2001.04451>
- *Timo Schick, Hinrich Schütze, 2020, It's Not Just Size That Matters: Small Language Models Are Also Few-Shot Learners:* <https://arxiv.org/abs/2009.07118>
- *Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz 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*
- OpenAI GPT-2 GitHub Repository: <https://github.com/openai/gpt-2>
- N Shepperd GitHub Repository: <https://github.com/nshepperd/gpt-2>

7

Applying Transformers to Legal and Financial Documents for AI Text Summarization

During the first six chapters, we explored the architecture of the Transformer and how to train transformers. We also implemented pretrained models that could perform downstream tasks with fine-tuning. Finally, in *Chapter 6, Text Generation with OpenAI GPT-2 and GPT-3 Models*, we discovered that OpenAI has begun to experiment with zero-shot models that require no fine-tuning.

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. 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 provides 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 transformers' 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 Text-To-Text Transfer Transformer. 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. We will explore the limits of transfer learning approaches applied to transformers.

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

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 6, Text Generation with OpenAI GPT-2 and GPT-3 Models, 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-tuning 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) worked on designing a conceptual text-to-text model and then implementing 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." From the start, the Google team working on this approach emphasized that it would not modify the fundamental architecture of the original Transformer.

At that point, *Raffel et al. (2019)* wanted to focus on concepts, not techniques. 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 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) that is followed by another sequence (B). B, in turn, becomes the start sequence leading to another sequence, as shown in *Figure 7.1*:

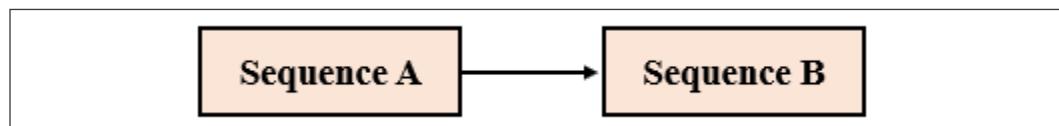


Figure 7.1: A sequence-to-sequence representation of communication

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 that accompanied them!

The technical revolution of attention layers that attend to all of the tokens in a sequence at the same time led to the T5 conceptual revolution.

The T5 model can be summed up as a Text-To-Text Transfer Transformer. 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) offered 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. 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 5, Machine Translation with the Transformer*.
- "cola sentence: + [sequence]" for *The Corpus of Linguistic Acceptability (CoLA)*, as we used in *Chapter 2, Fine-Tuning BERT models*, when we fine-tuned a BERT transformer model.

- "sts_b sentence 1:+[sequence]" for semantic textual similarity benchmarks. Natural language inferences and entailment are similar problems, as described in *Chapter 4, Downstream NLP tasks with Transformers*.
- "summarize + [sequence]" for text summarization problems 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 7.2*:



Figure 7.2: 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 **Text-To-Text Transfer Transformer (T5)**. The input and output of many NLP tasks have been unified, as shown in *Figure 7.3*.

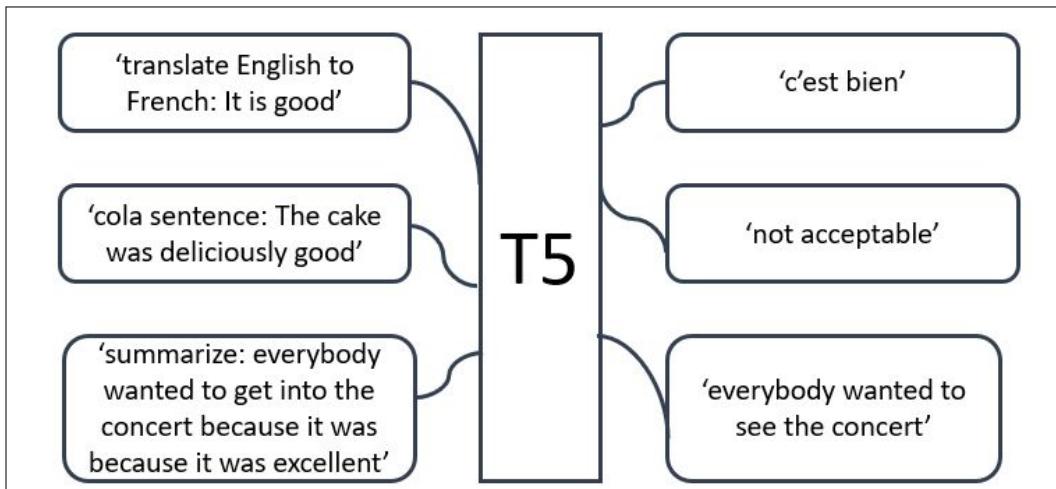


Figure 7.3: The T5 text-to-text framework

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. The energy of the team was focused on defining NLP tasks in a standard format.

They chose to use the original Transformer model we defined in *Chapter 1, Getting Started with the Model Architecture of the Transformer*, as we can see in Figure 7.4:

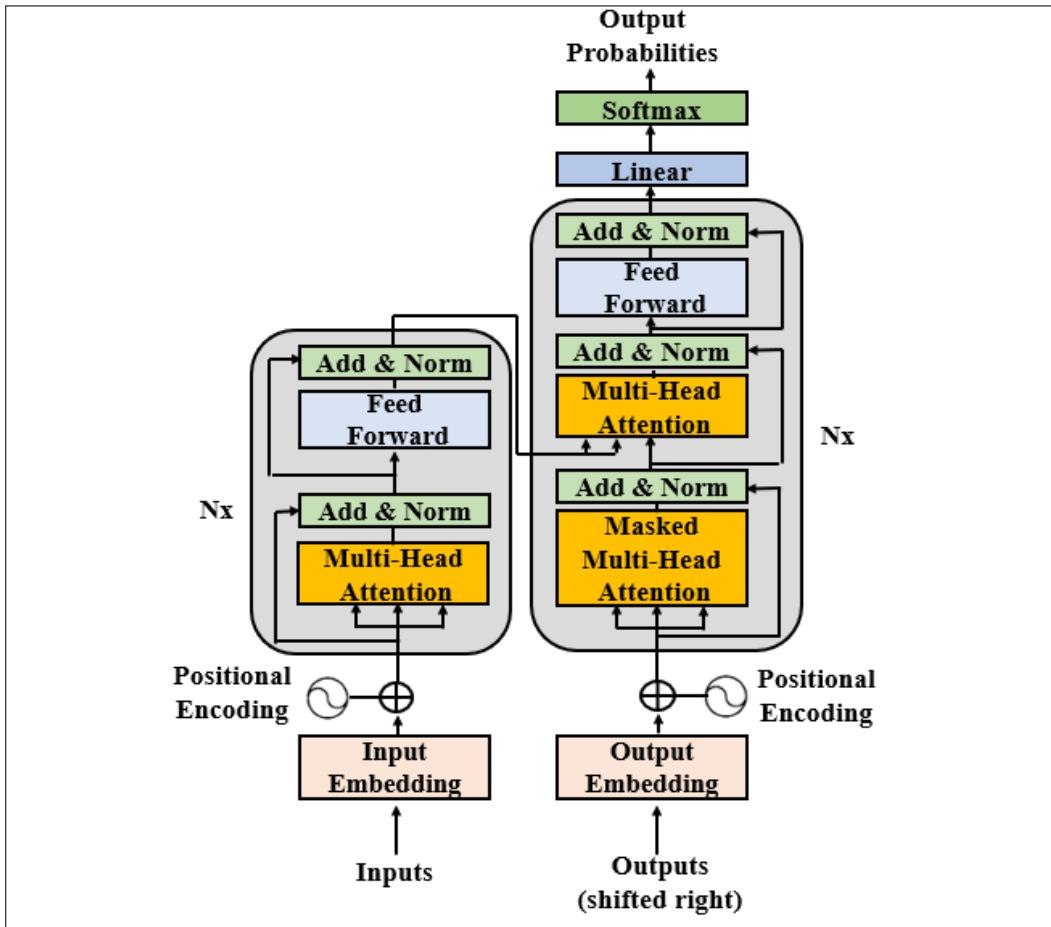


Figure 7.4: The original Transformer model used by T5

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 sub-layers become "sub-components" containing a self-attention layer and a feedforward network. The use of the word "blocks" and "sub-components" 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 1, Getting Started with the Model Architecture of the Transformer*.
- Self-attention is "order-independent," which means that it performs operations on sets.
- 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 comparisons between pairwise relationships. For more, see Shaw et al. (2018) in the *Reference* section of this chapter.
- Positional embeddings are shared and re-evaluated through all of 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. In this section, we 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 type of document, including legal and corporate documents.

Let's begin by using 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 2, Fine-Tuning BERT Models*, and to train a RoBERTa model in *Chapter 3, Pretraining a RoBERTa Model from Scratch*.

However, we needed to explore other approaches, such as Trax, in *Chapter 5, Machine Translation with the Transformer*, and OpenAI's GitHub repository in *Chapter 6, Text Generation with OpenAI GPT-2 and GPT-3 Models*.

In this chapter, we will use Hugging Face's framework again and explain more about the resources made available online.

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 7.5*:

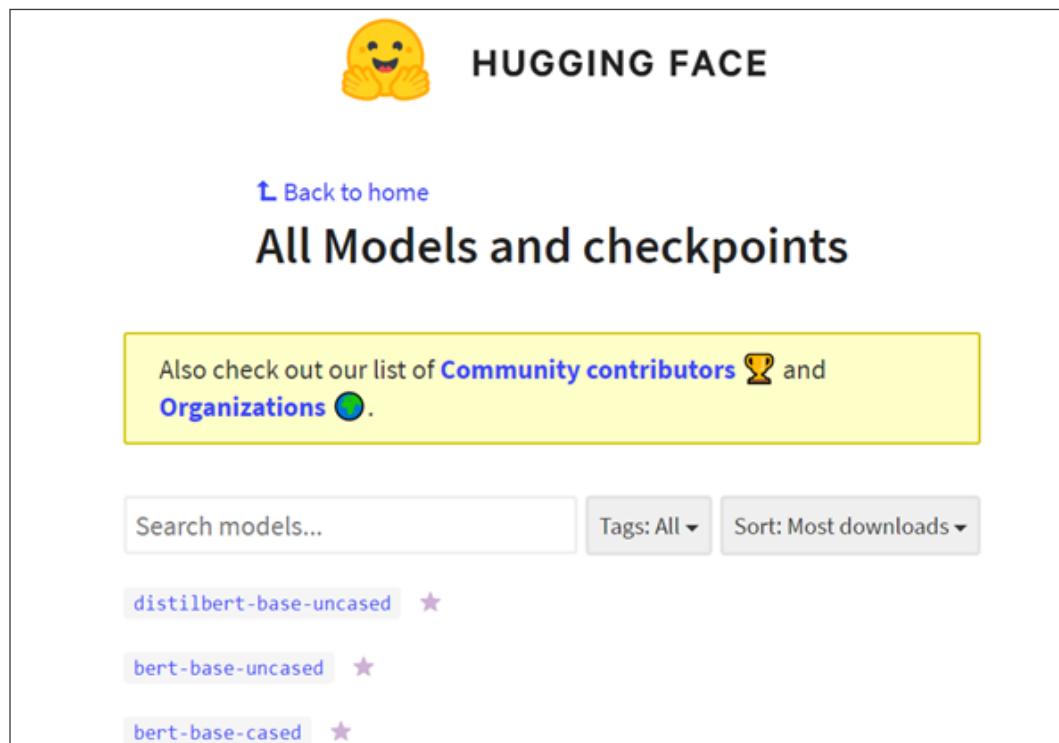


Figure 7.5: Hugging Face models

On this page, <https://huggingface.co/models>, we can search for a model. In our case, we are looking for **t5-large**, a t5-large 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:



Figure 7.6: Searching for a T5 model

We can see that the original five T5 transformers are available:

- **Base**, which is the baseline model. It was designed to be similar to the BERT_{BASE} with 12 layers and around 220 million parameters.
- **Small**, which is a smaller model with 6 layers and 60 million parameters.
- **Large**, which is designed to be similar to BERT_{LARGE} with 12 layers and 770 million parameters.
- **3B** and **11B**, which use 24 layer encoders and decoders with around 2.8 billion parameters and 11 billion parameters.

For more on the description of BERT_{BASE} and BERT_{LARGE}, you can take a few minutes now or later to review these models in *Chapter 2, Fine-Tuning BERT Models*.

In our case, we select **t5-large**:

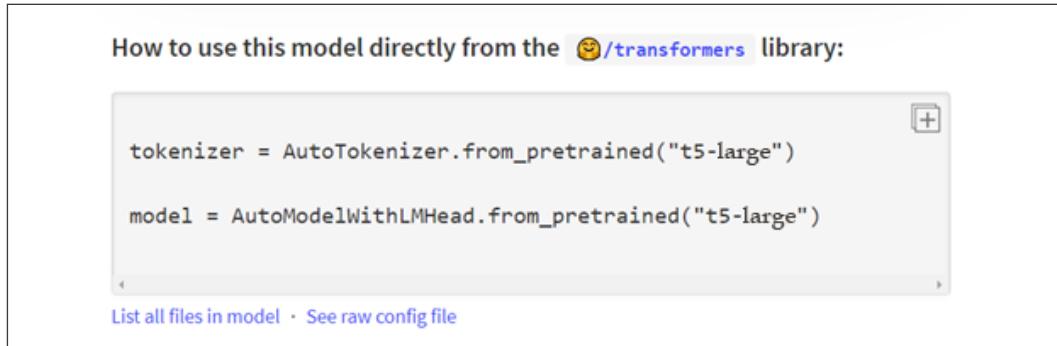


Figure 7.7: How to use a Hugging Face model

Figure 7.7 shows how to use the model in the code we are going to write. We can also look into the list of files in the model and the raw configuration file. We will look into the configuration file when we initialize the model in *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>

In this chapter, we will not implement these datasets or metrics. We will focus on how to implement any type of text to summarize.

Let's start by initializing the T5 transformer model.

Initializing the T5-large transformer model

In this sub-section, 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:

`Summarizing_Text_with_T5.ipynb`

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==4.0.0
```



Version 4.0.0 of Hugging Face transformers is pinned due to the rapid evolution of transformers, leading to changing libraries and modules to adapt to the market.

We also 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 sub-layers 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 that research labs share with us. 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 8, 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 *beam search*, 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. Too little vocabulary will distort the NLP tasks. We will explore this further in *Chapter 8, 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 sub-layers and 4,096 for the inner calculations of the feedforward network sub-layer 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, decoder stacks, the attention sub-layers, and the feedforward sub-layers.

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

In this section, we 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 `m1`, the maximum length of the summarized text. Both parameters are variables you send to the function each time you call it:

```
def summarize(text,m1):
```



Hugging Face, among others, provide 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 tokens 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 some code. You can also load a dataset of your choice and call the summaries in a loop.

The goal of the program 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 expressed the precise 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, exceptin 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)
```

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 any other NLP model faces when faced with a text such as this one. We cannot just present samples that always work and make a user believe that transformers, no matter how innovative they are, have solved all of the NLP challenges we face.

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 difficult text with an NLP model, you will always find documents that the model fails to summarize.

When a model fails on a task, we need to 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 to be improved.

Now, let's try a corporate law sample.

A corporate law sample

Corporate law contains many legal subtleties, which makes a summarizing task 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 try 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 conclude and move on to our next adventure!

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, which indicated the type of NLP problem to solve. This leads to a standard text-to-text format. The **Text-To-Text Transfer Transformer (T5)** was born. We saw that this deceptively 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).

Improving transformers and NLP, in general, requires more research in every aspect of the processing of NLP tasks.

In the next chapter, *Matching Tokenizers and Datasets*, we will explore the limits of tokenizers and define methods to 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. NLP text summarization works for any text. (True/False)
10. Hugging Face is a framework that makes transformers easier to implement. (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, Łukasz 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>*

8

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, trained a GPT-2 model, and implemented a T5 model. 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. Artificial intelligence is data-driven. *Raffel et al. (2019)*, like all of 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 are using (advanced medical language, for example) with words that are 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 word2vector tokenizer to describe the problems we face with any tokenizing method. The limits will be illustrated with a Python program.

We will continue by exploring 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.

Finally, we will go back to the summarizing problem we faced when we tried to summarize the *Bill of Rights* with a T5 model in *Chapter 7, Applying Transformers to Legal and Financial Documents for AI Text Summarization*. We will apply the ideas we discovered in this chapter to improve the T5's summary.

This chapter covers the following topics:

- Basic guidelines to control the output of tokenizers
- Raw data strategies and preprocessing data strategies
- Word2vector tokenization problems and limits
- Creating a Python program to evaluate word2vector tokenizers
- Evaluating GPT-2 tokenizers
- Building a Python program to evaluate the output of byte-level BPE algorithms
- Customizing NLP tasks with specific vocabulary
- Testing a standard T5 conditional input sample
- Improving the datasets

Our first step will be to explore the text-to-text methodology defined by *Raffel et al. (2019)*.

Matching datasets and tokenizers

Downloading benchmarks 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-2-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 has 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 the English language at all. They decided that datasets need to be cleaned before using them.

We will take the recommendations Raffel et al. (2019) made further and apply corporate quality control best practices to the preprocessing and post-processing phases. The examples described, among many other rules to apply, give an idea of the tremendous work required to obtain acceptable real-life project datasets.

Figure 8.1 lists some of the key quality controls processes to apply to datasets:

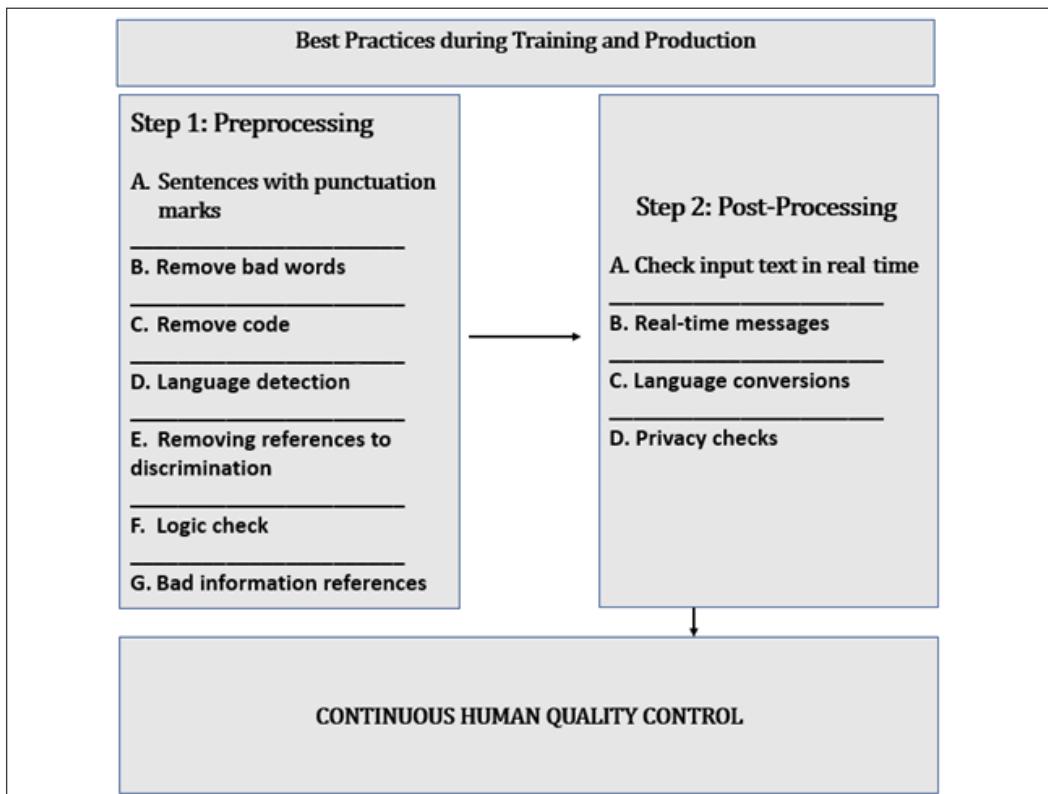


Figure 8.1: Best practices for transformer datasets

Quality control, as shown in *Figure 8.1*, is divided into the preprocessing phase (*Step 1*) when training a transformer, and post-processing 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 to preprocess 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 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/LDN00BW/List-of-Dirty-Naughty-Obscene-and-Otherwise-Bad-Words>

- **Remove code**

This is a tricky one because sometimes code is the content we are looking for. However, in general, it is best to remove code from content for NLP tasks.

- **Language detection**

Sometimes web sites 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 web sites, or persons. This is a tough job, but well worthwhile.

This list contains some of the primary best practices. More is required, such as filtering privacy law violations, and other actions for specific projects.

Once a transformer is trained to learn proper English, for example, we need to help it detect problems in the input texts in the production phase.

Step 2: Post-processing

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. 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. As such, we need to avoid the weaponization of the NLP tasks they perform to run unacceptable tasks.

Let's go through some of the best practices:

- **Check input text in real time**

Do not accept bad information. 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 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 automated 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. The training set will continuously improve, and the transformer will continue to learn.

Figure 8.2 shows how continuous quality control will help the transformer's training dataset grow and increase its performance in production:

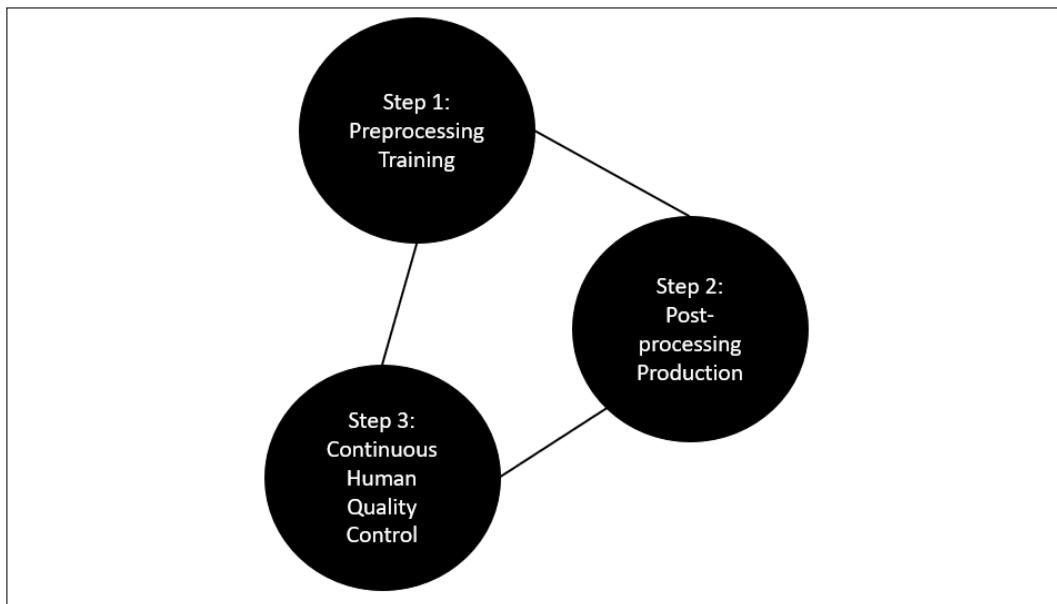


Figure 8.2: Continuous Human Quality Control

We have gone through several best practices described by Raffel et al. (2019), and I

have added some 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. Some word pairs just don't fit together, as we can see in *Figure 8.3*:

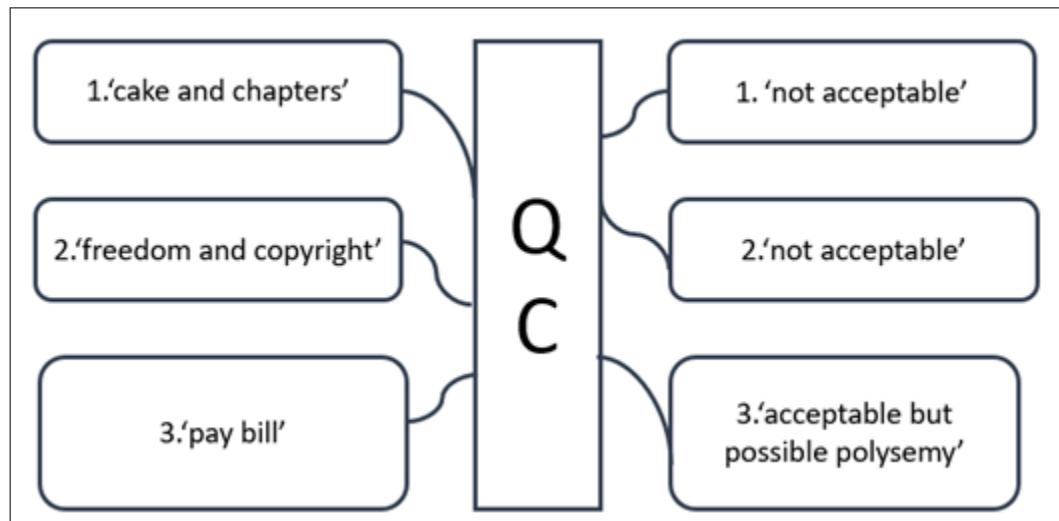


Figure 8.3: Word pairs that tokenizers miscalculated

The examples shown in *Figure 8.3* 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 freedom of speech, for example. "Copyright" refers to a 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. "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 medical, airplane or rocket assembly, or other critical domains).

Open `Tokenizer.ipynb`, based on `positional_encoding.ipynb`, which we created in *Chapter 1, Getting Started with the Model Architecture of the Transformer*.

The pre-requisites are installed and imported first:

```
#@title Pre-Requisites
!pip install --upgrade gensim
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 *Emmanuel 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=0.25:

```
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 0 and 1.

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 6 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.

We can consider this case as acceptable.

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 will distort the transformer model's output if the word was 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 3, 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 the 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 social media standards 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 word2vector 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 one might be tempted to rely on a transformer's ability to make inferences with unseen words.

However, in a strategic project with critical decision making, my recommendation is to run several different quality control methods. 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 exceeds 0.8.

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 *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 see the problem we face with rare words.

Case 3: Rare words

Rare words produce devastating effects of the output of transformers for specific tasks that go beyond trivial 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, or 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 and that are forgotten or that only specialists use.

For example, in this case, we are using the word "justiciar":

```
#@title Case 3: Rare words
word1="justiciar";word2="judgement"
print("Similarity",similarity(word1,word2),word1,word2)
```

The similarity with "judgment" is reasonable but should be higher:

Similarity [[0.6606605]] justiciar judgement

One might think that the word "justiciar" is far fetched. The tokenizer extracted it from the *Magna Carta*, which dates back to the early 13th century.

However, several articles of the *Magna Carta* are still valid in 21st century England! 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 4: Replacing rare words

Replacing rare words represents a project in itself. The work this takes is reserved for specific tasks and projects. If a corporate budget can cover the cost of having a knowledge base in aeronautics, for example, it's 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 3, we stumbled on the word "judiciar." If we go back to its origin, we can see if it comes from the French Normand language and is the root of the French Latin-like word "judicaire."

We could replace the word "judiciar" with "judge," which conveys the same meta-concept:

```
#@title Case 4: Replacing words
word1="judge";word2="judgement"
print("Similarity",similarity(word1,word2),word1,word2)
```

If it produces an interesting result:

```
Similarity [[0.7962761]] judge judgement
```

We could also keep the work "justiciar" but try the modern meaning of the word and compare it to "judge." You could try the following example by adding it to the notebook:

```
word1="justiciar";word2="judge"
print("Similarity",similarity(word1,word2),word1,word2)
```

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. 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. The transformer's performance with NLP tasks would increase, and the knowledge base of the corporation would progressively increase.

Let's see how to use cosine similarity for entailment verification.

Case 5: 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 5: 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 3: Rare words* and *Case 4: Replacing rare words* from the Word2Vec tokenization section of this chapter.

We will use `Training_OpenAI_GPT_2_CH08.ipynb`, a renamed version of the notebook we used to train a dataset in *Chapter 6, Text Generation with OpenAI GPT-2 and GPT-3 Models*.

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_CH08.ipynb` in detail. If necessary, take some time to go back through *Chapter 6, Text Generation with OpenAI GPT-2 and GPT-3 Models*. Make sure you upload the necessary files before beginning, as explained in *Chapter 6*. The files are on GitHub in the `gpt-2-train_files` directory of *Chapter08*. Although we are using the same notebook as in *Chapter 6*, note that the dataset, `dset`, is now named `mdset` in the directory and code.

Let's first generate an unconditional sample with a GPT-2 model trained to understand medical content.

Generating unconditional samples with GPT-2

In *Case 3: Rare Words* and *Case 4: 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, 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 6, Text Generation with OpenAI GPT-2 and GPT-3 Models*. You can run this code cell by cell using *Chapter 6* to guide you. Take special care to follow the instructions to make sure `tf 1.x` is activated.

After training the model on the medical dataset, will you 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'
```

Run the cell. 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. 01 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 are 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. We can always increase the size of the dataset. But will it contain what we are looking for? Could we find bad correlations with more data? 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.

Imagine you have a dataset with billions of words but that 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 here is that it will take a lot of hard work and a solid team to produce reliable results.

Let's take our investigation further and control the tokenized data.

Controlling tokenized data

In this section, we will read the first words the GPT-2 model encoded with its pretrained tokenizer.

We will go to the Additional Tools: Controlling Tokenized Data cell of the Training_OpenAI_GPT_2_CH08.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`:

```
#@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
suggests : 5644
that : 326
```

We can easily recognize the initial tokens preceded by the initial whitespace characters (>). 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 down into sub-words:

```
Am : 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 contains 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:

```
Âm : 716
o : 78
eb : 1765
oid : 1868
Ând : 290
Âmes : 18842
ench : 24421
ym : 4948
al : 282
```

One 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" as of one token with the other "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.

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
```

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 try 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

The result is better but requires more research.

Let's look into another sample that requires careful analysis.

T5 Bill of Rights Sample

The following sample, taken from the *Bill of Rights*, is more difficult because it expresses the precise rights of a person.

Open `Summarizing_Text_V2.ipynb`, a copy of the `Summarizing_Text_with_T5.ipynb` notebook we used in *Chapter 7, Applying Transformers to Legal and Financial Documents for AI Text Summarization*.

We will first run T5 without making any changes.

Summarizing the Bill of Rights, version 1

In this section, we will enter the same text we did in *Chapter 7, Applying Transformers to Legal and Financial Documents for AI Text Summarization*:

```
#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, exceptin
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)
```

As in *Chapter 7, Applying Transformers to Legal and Financial Documents for AI Text Summarization*, we can see that T5 did not really summarize the input text but simply shortened it:

```
Number of characters: 591
Preprocessed and prepared text
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
```

Let's move to version 2 to find out why T5 did not summarize the text correctly.

Summarizing the Bill of Rights, version 2

The words in the excerpt of the *Bill of Rights* seem modern because this is modern English. Although the words are not rare, the grammatical structure of the sentence is complicated and confusing.

The pretrained T5 model is used in modern everyday English. Many books are translated from older English into everyday English. Let's do that. Let's translate the input text into everyday English:

```
#Bill of Rights,V
text ="""
A person must be indicted by a Grand Jury for a capital or infamous
crime.
There are exceptions in time of war for a person in the army, navy, or
national guard.
A person can not be judged twice for the same offense or put in a
situation of double jeopardy of life.
A person can not be asked to be a witness against herself or himself.
A person cannot be deprived of life, liberty or property without due
process of law.
A person must be compensated for property taken for public use.
"""

print("Number of characters:",len(text))
summary=summarize(text,50)
print ("\n\nSummarized text: \n",summary)
```

The result is better though it might vary from one run to another. The summary is not that bad:

```
Number of characters: 485
Preprocessed and prepared text:
summarize: A person must be indicted by a Grand Jury for a capital
Summarized text:
there are exceptions in time of war for a person in the army, navy, or
national guard. no person can be deprived of life, liberty or property
without due process of law. there must be compensation for property
taken for
```

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 on specific topics to become a specialist in that field. The bottom line is that, for a specific project, we will still have to train the transformers on specific datasets.

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.

Let's now conclude this chapter and move on to another NLU exploration.

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 summarize, such as how a tokenizer "thinks" and encodes data.

Finally, we applied the method to a T5 summarizing problem with a certain amount of success. There is still a lot of room for improvement.

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, *Chapter 9, Semantic Role Labeling with BERT-Based Transformers*, we will dig deep into NLU and use a BERT model to ask a transformer to explain a sentence's meaning.

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 a 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, 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* of 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>

9

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 the basic syntax of language before running **Semantic Role Labeling (SRL)**.

Shi and Lin (2019) start 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) suggest that SRL can be considered as sequence labeling and provide a standardized input format. Their BERT-based model produced surprisingly good results.

In this chapter, we 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. 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. 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 stack only 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 TensorFlow 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, transformers, once again, 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* a word or group of words plays in a sentence and the relationship established with the predicate.

A *semantic role* is a role a noun or noun phrase plays in relation to the main verb in a sentence. 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*. In the sentence "Marvin walked in the park," the predicate is "walked" in its restricted form.

The words "in the park" *modifies* the meaning of "walked" and is 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

In this chapter, we 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 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, which we actively share. All of the figures in this chapter were created with the AllenNLP tools.

The Allen Institute provides transformer models that continuously evolve. 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 9.1* is an SRL representation of "Marvin walked in the park":

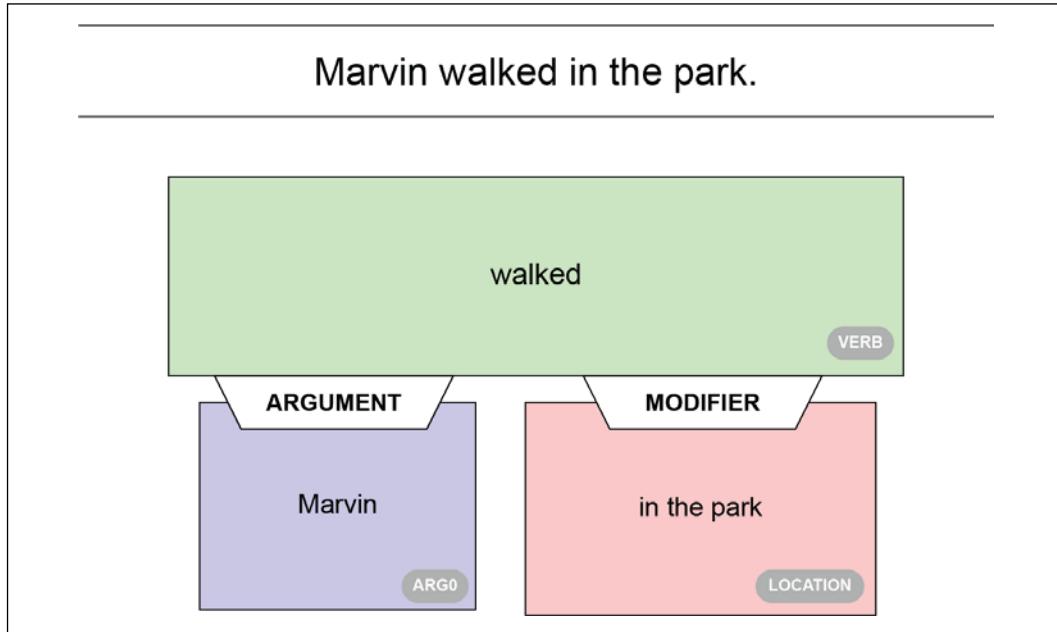


Figure 9.1: The SRL representation of a sentence

We can observe the following labels in *Figure 9.1*:

- **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

In this section, we 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 as 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 2, Fine-Tuning BERT Models*, in general. You can also go straight to the *BERT model configuration* section of that chapter, which describes the usage parameters of the BERT model we are running in this chapter:

- BertForMaskedLM
- 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

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 gone 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, indicates the end of the sentence.
- [SEP] is followed by the predicate identification designed by the authors.
- [SEP] is the second separator, 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/reading-comprehension> under the *Defining Semantic Role Labeling* section.

We will apply the following method:

1. 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. AllenNLP models and transformers, in general, are continuously trained and updated. Also, the datasets using 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 allenlp==1.0.0 allenlp-models==1.0.0
```

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 not easy 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 *Sample 1* cell in `SRL.ipynb`:

```
!echo '{"sentence": "Did Bob really think he could prepare a meal for\n50 people in only a few hours?"}' | \
allennlp predict https://storage.googleapis.com/allennlp-public-models/
bert-base-srl-2020.03.24.tar.gz -
```

The transformer identified the verb "think," for example, as we can see in the following excerpt of the raw output of the cell:

```
prediction: {"verbs": [{"verb": "think", "description": "Did [ARG0:\nBob] [ARGM-ADV: really] [V: think] [ARG1: he could prepare a meal for\n50 people in only a few hours] ?"}],
```

If we run the sample in the AllenNLP online interface, we obtain a visual representation of the SRL task. The first verb identified is "think":

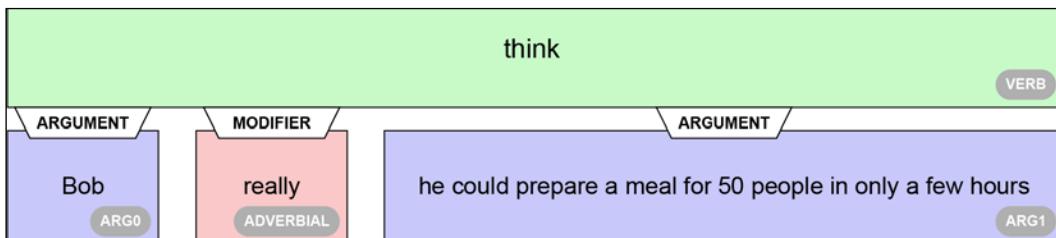


Figure 9.2: Identifying the verb "think"

If we take a close look at this representation, we can detect some interesting properties of the simple BERT-based transformer, which:

- 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 transformer then moved to the verb "prepare," labeled it, and analyzed its context:



Figure 9.3: 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." Both arguments are correctly related to the verb "prepare"
- That "in only a few hours" is a temporal modifier of "prepare"
- That "could" was a modal modifier that indicates the modality of a verb, such as the likelihood of an event

The text output of AllenNLP sums the analysis up:

```
think: Did [ARG0: Bob] [ARGM-ADV: really] [V: think] [ARG1: he could prepare a meal for 50 people in only a few hours] ?  
  
could: Did Bob really think he [V: could] prepare a meal for 50 people in only a few hours ?  
  
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] ?
```

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:

```
!echo '{"sentence": "Mrs. And Mr. Tomaso went to Europe for vacation and visited Paris and first went to visit the Eiffel Tower."}' | \
allennlp predict https://storage.googleapis.com/allennlp-public-models/
bert-base-srl-2020.03.24.tar.gz -
```

The excerpt of the output proves that the transformer correctly identified the verbs in the sentence:

```
prediction: {"verbs": [{"verb": "went", "description": "[ARG0: Mrs. and Mr. Tomaso] [V: went] [ARG4: to Europe] [ARGM-PRP: for vacation]"}]
```

Running the sample on AllenNLP online shows that an argument was identified as the *purpose* of the trip:

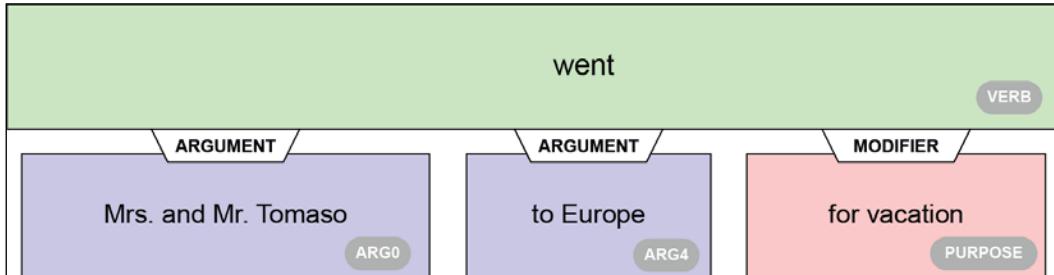


Figure 9.4: Identifying the verb "went," the arguments, and the modifier

We can interpret the arguments of the verb "went." However, the transformer found that the modifier of the verb was the purpose of the trip. 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 "visit" as being related to "Paris":

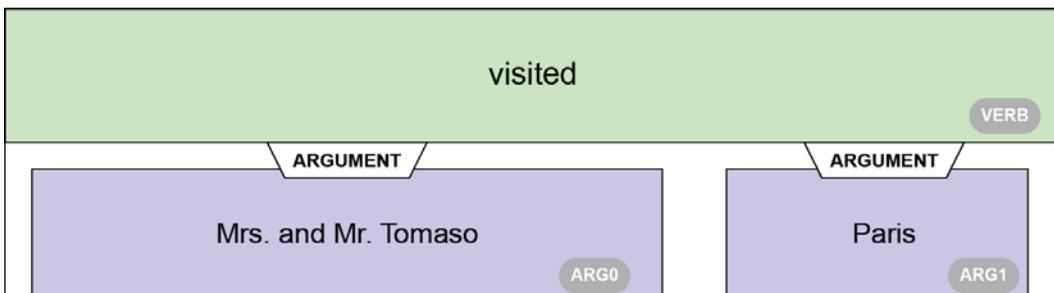


Figure 9.5: Identifying the verb "visited" and the arguments

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 final task we asked the transformer to do was to 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:

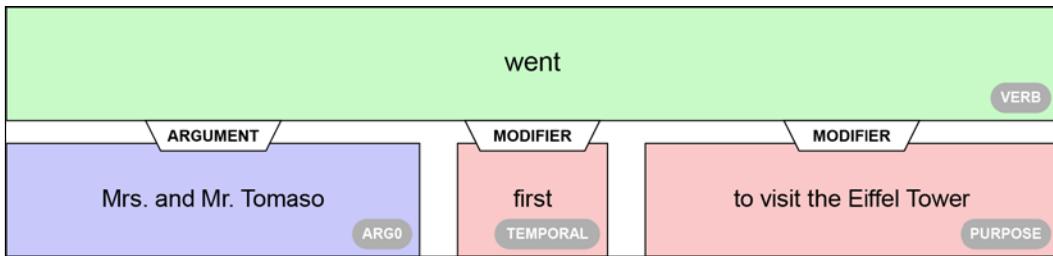


Figure 9.6: Identifying the verb "went," the argument, and the modifiers

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."

The formatted text output of the AllenNLP online interface sums the excellent result obtained for this sample:

```
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 .
```

```
visited: [ARG0: Mrs. and Mr. Tomaso] went to Europe for vacation and [V: visited] [ARG1: Paris] and first went to visit the Eiffel Tower .
```

```
went: [ARG0: Mrs. and Mr. Tomaso] went to Europe for vacation and visited Paris and [ARGM-TMP: first] [V: went] [ARGM-PRP: to visit the Eiffel Tower] .
```

```
visit: [ARG0: Mrs. and Mr. Tomaso] went to Europe for vacation and visited Paris and first went to [V: visit] [ARG1: the Eiffel Tower] .
```

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 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:

```
!echo '{"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."}' | \
allennlp predict https://storage.googleapis.com/allennlp-public-models/ \
bert-base-srl-2020.03.24.tar.gz -
```

The transformer found its way around, as shown in the following excerpts of the raw output that contain the verbs:

```
prediction: {"verbs": [{"verb": "wanted," "description": "[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": "likes," "description": "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": "drank," "description": "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," "description": "John wanted to drink tea , Mary likes to drink coffee but Karim drank some cool water and Faiza [V: would] [ARGM-DIS: like] to drink tomato juice ."}]
```

When we run the sentence on the AllenNLP online interface, we obtain several visual representations. We will examine two of them.

The first one is perfect. it identifies the verb "wanted" and makes the right associations:

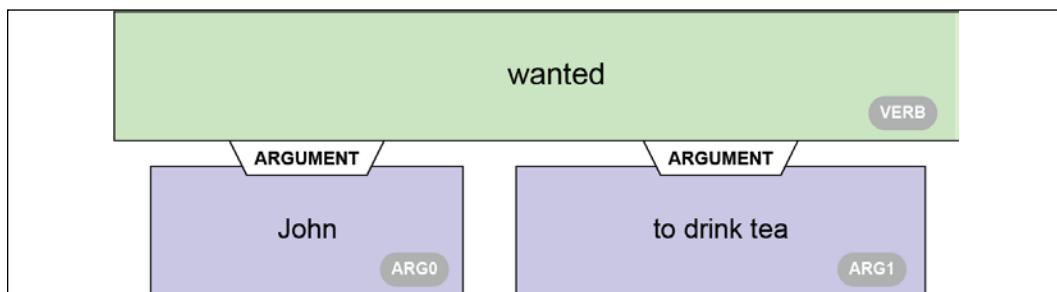


Figure 9.7: Identifying the verb "wanted" and the arguments

However, when it identified the verb "drank," it slipped in "and Faiza" as an argument:

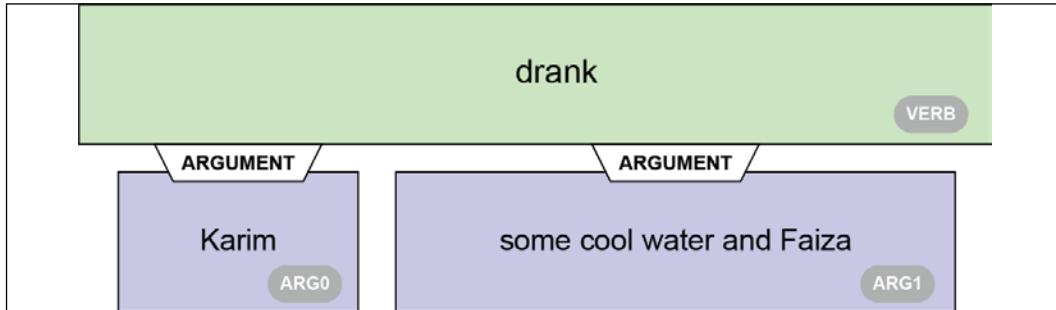


Figure 9.8: Identifying the verb "drank" and the arguments

The sentence meant that "Karim drank some cool water." The presence of "and Faiza" as an argument of "drank" is debatable.

The problem has an impact on "Faiza would like to drink tomato juice":

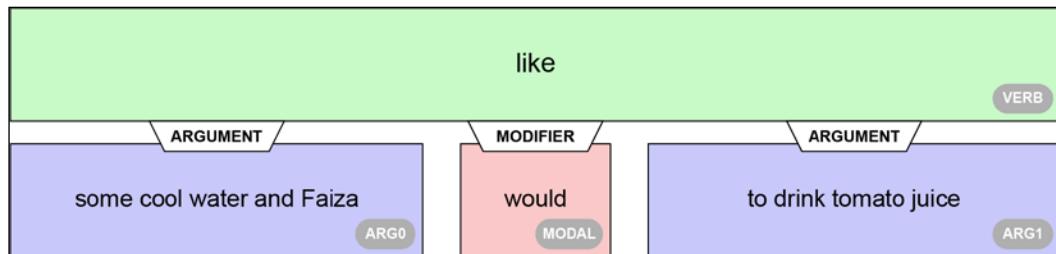


Figure 9.9: Identifying the verb "like," the arguments, and the modifier

The presence of "some cool water and" is not an argument of *like*. Only "Faiza" is an argument of "like."

The text output obtained with AllenNLP confirms the problem:

```
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 .
```

```
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 .
```

```
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 .
```

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 .

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 .

would: John wanted to drink tea , Mary likes to drink coffee but Karim drank some cool water and Faiza [V: would] [ARGM-DIS: like] to drink tomato juice .

like: John wanted to drink tea , Mary likes to drink coffee but Karim drank [ARG0: some cool water and Faiza] [ARGM-MOD: would] [V: like] [ARG1: to drink tomato juice] .

drink: John wanted to drink tea , Mary likes to drink coffee but Karim drank [ARG0: some cool water and Faiza] would like to [V: drink] [ARG1: tomato juice] .

The output is a bit fuzzy. For example, we can see that one of the arguments of the verb "like" is that "Karim drank some cool water and Faiza", which is confusing:

like: John wanted to drink tea , Mary likes to drink coffee but Karim drank [ARG0: some cool water and Faiza] [ARGM-MOD: would] [V: like] [ARG1: to drink tomato juice] .

We found that the BERT-based transformer produces relatively good results on basic samples. Let's try some more difficult ones.

Difficult samples

In this section, we will run samples that contain problems that the BERT-based transformer will first solve. 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, whose husband went jogging every Sunday, 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`:

```
!echo '{"sentence": "Alice, whose husband went jogging every Sunday, liked to go to a dancing class in the meantime."}' | \
allennlp predict https://storage.googleapis.com/allennlp-public-models/
bert-base-srl-2020.03.24.tar.gz -
```

The raw output is quite long, with detailed descriptions. Let's focus on the part we are interested in and see if the model finds the predicate. It did! It found the verb "liked" as shown in this excerpt of the raw output:

```
[ARG0: Alice , whose husband went jogging every Sunday] , [V: liked]
```

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:

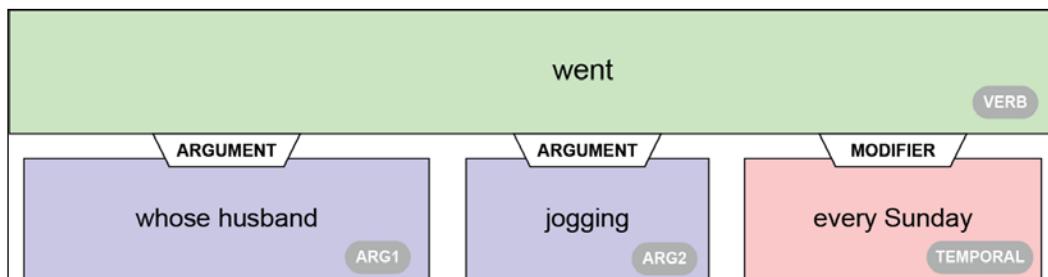


Figure 9.10: The predicate "went" has been identified

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*:

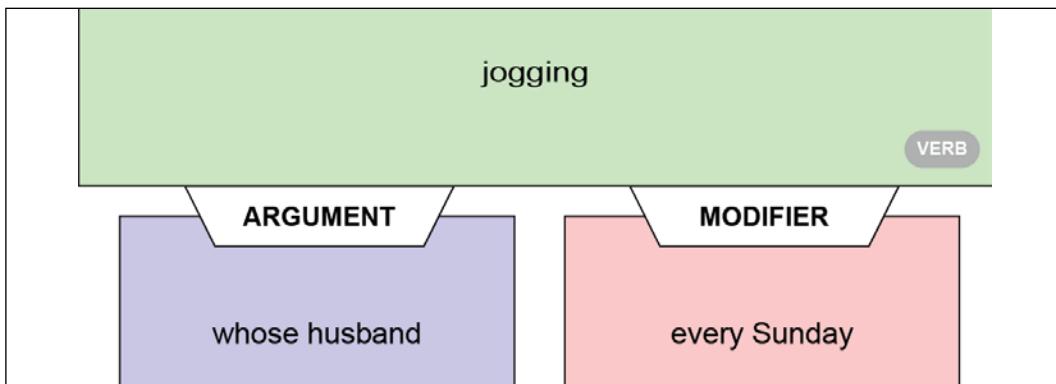


Figure 9.11: SRL detection of the verb "jogging"

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:

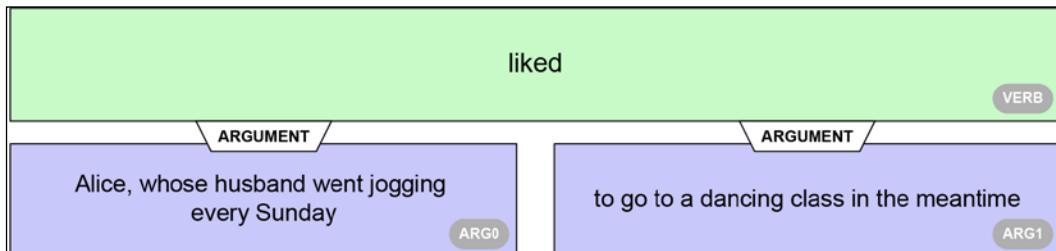


Figure 9.12: Identifying the verb "liked"

The argument describing Alice is a bit long but correct. If we go back to the raw output in our SRL.ipynb notebook, we can see that the raw detail confirms that the analysis is correct:

```
[ARG0: Alice , whose husband went jogging every Sunday] , [V: liked]  
[ARG1: to go to a dancing class in the meantime]
```

The transformer also detects and analyzes the verb "go" correctly:

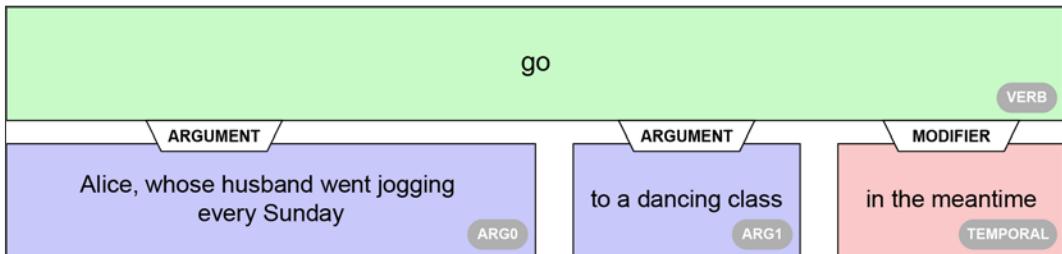


Figure 9.13: Detecting the verb "go," its arguments, and modifier

We can see that the temporal modifier "in the meantime" was identified as well. It is quite a performance when we think of the simple sequence + verb input the BERT-based model was trained with.

Finally, the transformer identifies the last verb, "dancing," as being related to "class":

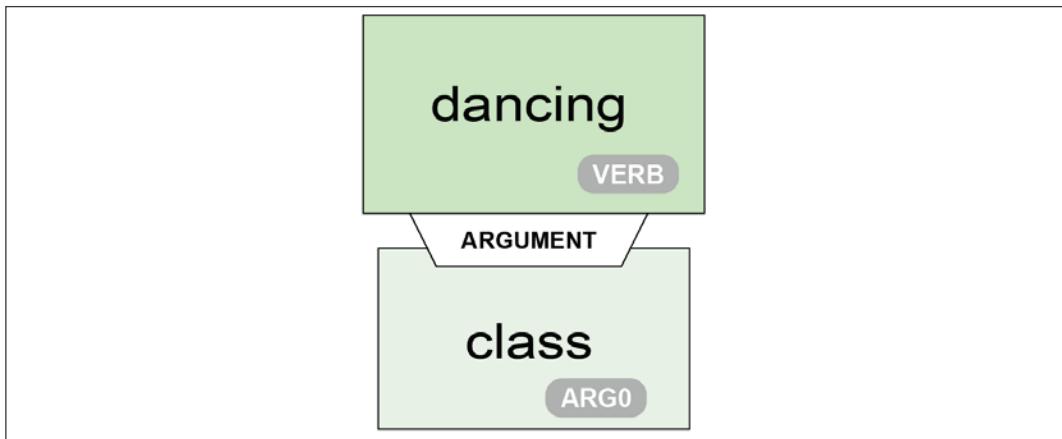


Figure 9.14: Relating the argument "class" to the verb "dancing"

We will now look at the formatted text output produced by the AllenNLP online UI:

```
went: Alice , [ARG1: whose husband] [V: went] [ARG2: jogging] [ARGM-TMP: every Sunday] , liked to go to a dancing class in the meantime .  
jogging: Alice , [ARG0: whose husband] went [V: jogging] [ARGM-TMP: every Sunday] , liked to go to a dancing class in the meantime .  
  
liked: [ARG0: Alice , whose husband went jogging every Sunday] , [V: liked] [ARG1: to go to a dancing class in the meantime] .  
  
go: [ARG0: Alice , whose husband went jogging every Sunday] , liked to [V: go] [ARG4: to a dancing class] [ARGM-TMP: in the meantime] .  
  
dancing: Alice , whose husband went jogging every Sunday , liked to go to a [V: dancing] [ARG0: class] in the meantime .
```

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 that can have multiple functions and meanings. It goes beyond polysemy since the word "round" can have both 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 mean to 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."

Round is used in a sense "to bring to perfection". The best grammatical form would have been "rounded," but the transformer found the right verb, and the sentence sounds rather poetic.

Let's run *Sample 5* in SRL.ipynb:

```
!echo '{"sentence": "The bright sun, the blue sky, the warm sand, the palm trees, everything round off."}' | \  
allennlp predict https://storage.googleapis.com/allennlp-public-models/  
bert-base-srl-2020.03.24.tar.gz -
```

The output shows no verbs. The transformer did not identify the predicate. In fact, it found no verbs at all:

```
prediction: {"verbs": [], "words": ["The", "bright", "sun", ",",
"the", "blue", "sky", ",", "the", "warm", "sand", ",",
"the", "palm",
"trees", ",",
"everything", "round", "off", "."]}
```

Since we like our BERT-based transformer, we will be kind to it. Let's change the sentence from the past tense to the present tense:

"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:

```
!echo '{"sentence": "The bright sun, the blue sky, the warm sand, the
palm trees, everything rounds off."}' | \
allennlp predict https://storage.googleapis.com/allennlp-public-models/
bert-base-srl-2020.03.24.tar.gz -
```

The raw output shows that the predicate was found, as shown in the following excerpt:

```
prediction: {"verbs": [{"verb": "rounds", "description": "[ARG1: The
bright sun .../..."}]
```

If we run the sentence on AllenNLP, we obtain the visual explanation:



Figure 9.15: Detecting the word "round" as a verb

Our BERT-based transformer did well because the word "round" can be found as "rounds" in its plural 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.

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. "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. The "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:

```
!echo '{"sentence": "Now, ice pucks guys!"}' | \
allennlp predict https://storage.googleapis.com/allennlp-public-models/
bert-base-srl-2020.03.24.tar.gz -
```

The transformer fails to find the verb:

```
prediction: {"verbs": [], "words": ["Now", ",", "ice", "pucks",
"guys", "!"]}
```

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!

Try some examples or samples of your own to see what SRL can do and the limits of the approach.

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. 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.

Transformers will continue to improve the standardization of NLP through their distributed architecture and input formats.

In the next chapter, *Chapter 10, 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 Labeling resources: <https://demo.allennlp.org/semantic-role-labeling/MjE4NDE1NA==>*

10

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.

Transformers, no matter how powerful they have become, 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 still impossible. Transformers often use general domain training datasets in a closed question-and-answer environment. For example, critical answers in medical care and law interpretation require additional NLP functionality.

However, transformers cannot answer any question correctly regardless of whether the training environment is closed with preprocessed question-answer sequences or not. If a sequence contains more than one subject and compound propositions, a transformer model can make wrong predictions.

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 that can be 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. 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.

Finally, we will add **Semantic Role Labeling (SRL)** functions to the blueprint of the text generator.

Before we leave the chapter, the *Next steps* section will provide additional ideas to build a reliable question-answering solution, including implementing the Haystack framework.

By the end of the chapter, you will see how to build your own multi-task NLP helpers 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

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 mostly presented as an NLP exercise involving a transformer and a dataset that contains the ready-to-ask questions and provides the answers to 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.

Chapter 6, Text Generation with OpenAI GPT-2 and GPT-3 Models, showed that the **Pattern-Exploiting Training (PET)** method applied to a small ALBERT model exceeded the performance of the much larger GPT-3 *model*.

The method outperforms models in many cases. 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 a question to an NLP bot 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 make 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 in which 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 artificial intelligence NLP projects implementing question-answering for a defense project in a tactical situation for an aerospace corporation, I combined different NLP methods to make sure 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==4.0.0
```

We will now import Hugging Face's pipeline, which contains a vast amount of 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 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, WBG0 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 or 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 methodic approach.

In this chapter:

1. We will continue to run QA.ipynb cell by cell to explore the methods described in each section.

2. 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 **extra-large (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 persons, 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, WBG0 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 edited to 2 decimals to fit the width of the page:

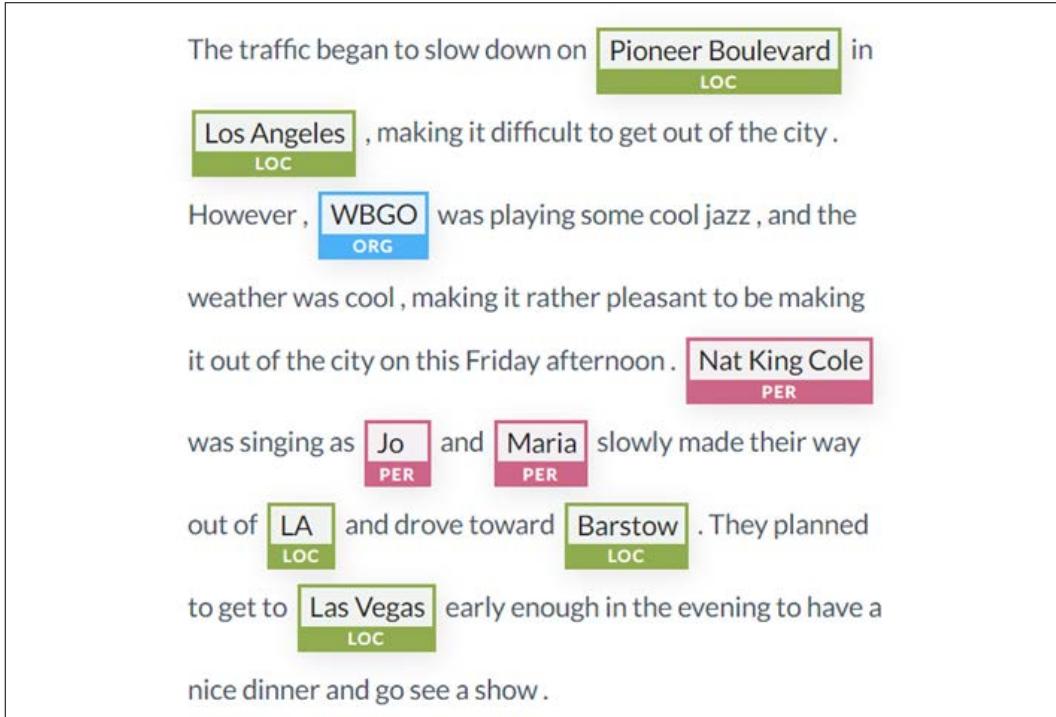
```
[{'word': 'Pioneer', 'score': 0.97, 'entity': 'I-LOC', 'index': 8},  
 {'word': 'Boulevard', 'score': 0.99, 'entity': 'I-LOC', 'index': 9},  
 {'word': 'Los', 'score': 0.99, 'entity': 'I-LOC', 'index': 11},  
 {'word': 'Angeles', 'score': 0.99, 'entity': 'I-LOC', 'index': 12},  
 {'word': 'W', 'score': 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:



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 is 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 in order 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 Barstow"). 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 to implement 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 which 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 found.

The bad news is that implemented transformers or any AI in real-life projects require powerful machines, 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 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 the 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 9, Semantic Role Labeling with BERT-Based Transformers*.

Let's run the same sequence on AllenNLP, <https://demo.allennlp.org/reading-comprehension>, in the **Semantic Role Labeling** section to obtain a visual representation of the verb "drove" in our sequence:

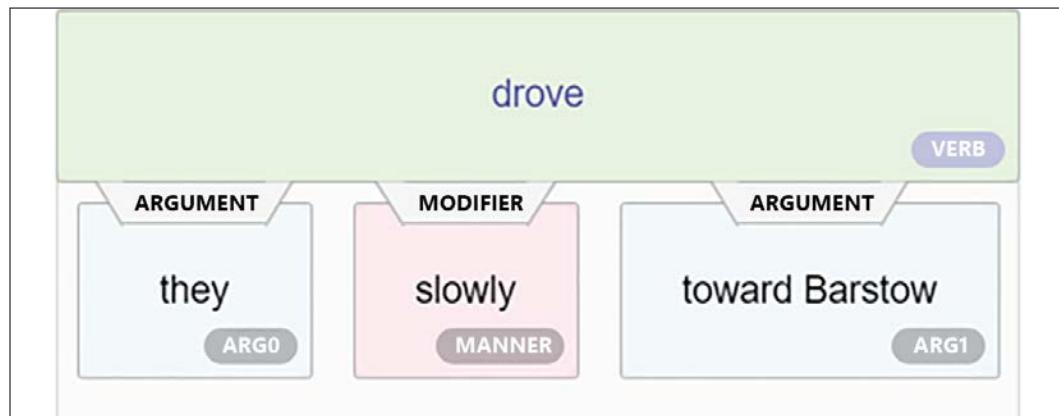


Figure 10.2: EER Semantic Role Labeling (SRL)

We can see the problem. The argument of the verb "driving" is "they." There is no relationship established between "they" and "Jo" and "Maria." It seems that the inference could be made.



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 2, 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) decided to introduce plausible alternatives with a generator network rather than simply use 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 10.3* shows how ELECTRA is trained:

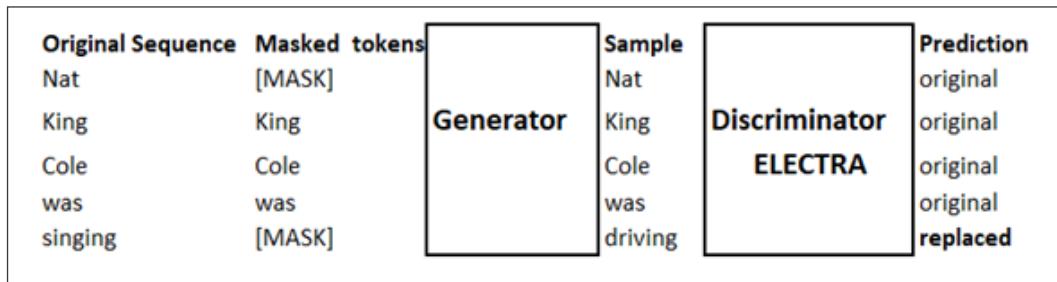


Figure 10.3: ELECTRA is trained as a discriminator

Figure 10.3 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 if 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 even 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 the 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 not satisfactory, a project manager might shut the project down.
- You can also try ready-to-use transformers, although they might not totally fit your need, 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.

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 9, 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/MjYxNDAyNA==>) 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-base model found 12 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:

```
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. It will be too difficult to disambiguate multiple semantic roles of the verb's arguments without further development.

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:

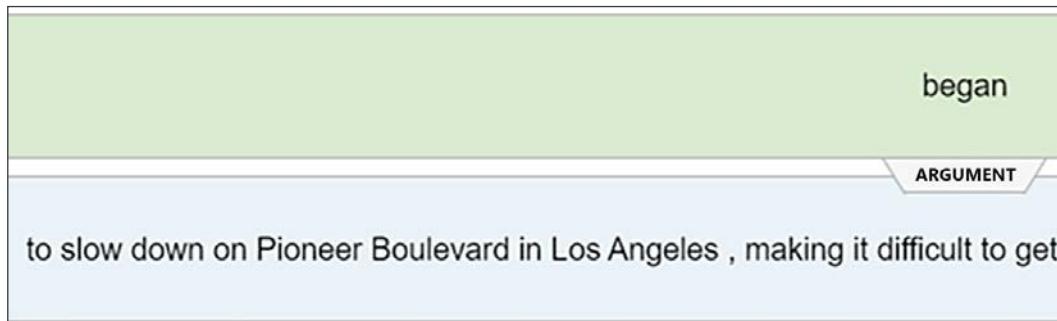


Figure 10.4: SRL applied to the verb "began"

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 maxlen:  
    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:

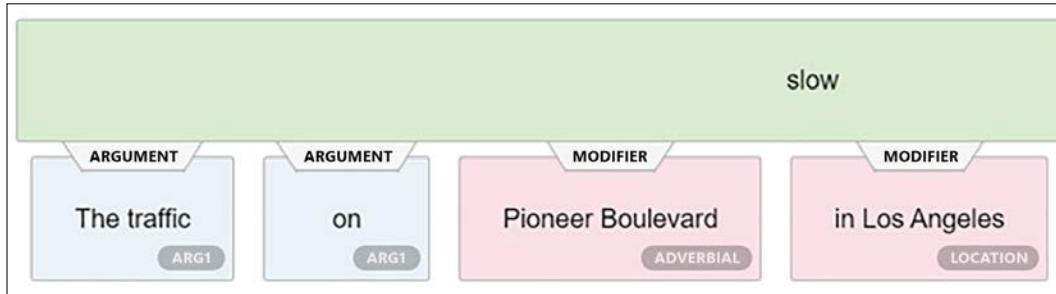


Figure 10.5: SRL applied to the verb "slow"

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.

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:

```
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, implies.

Let's try our approach with the next filter verb: "playing." The visual representation shows that the arguments are short:

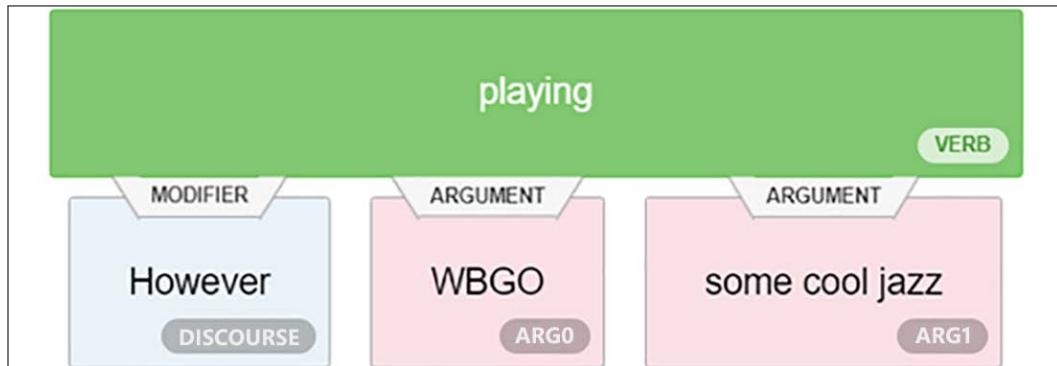


Figure 10.6: SRL applied to the verb "playing"

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]
```

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:

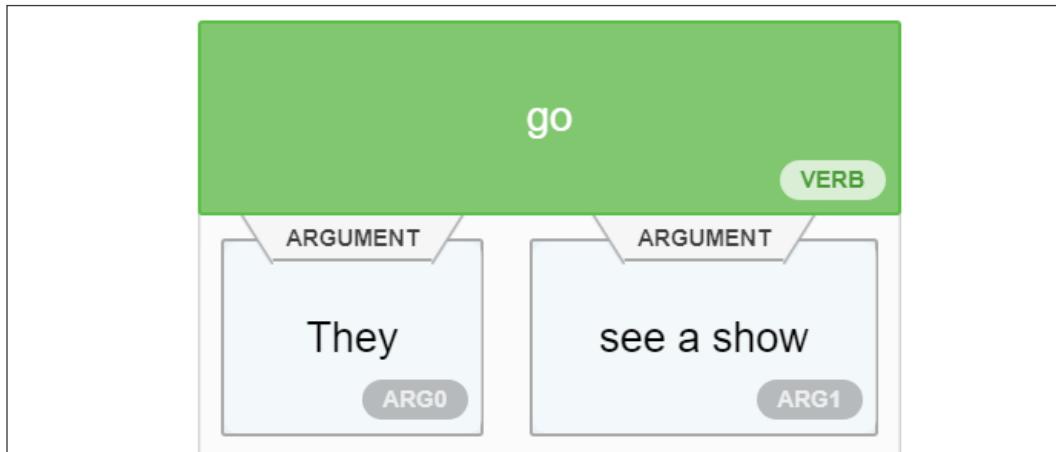


Figure 10.7: SRL applied to the verb "go"

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 could have been a good contribution to help 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 10.8: 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, then let the 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, to train transformers, or even to 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 3, 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.

We have explored some critical aspects of the use of question-answering 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. 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 outputs.

We then had the choice of continuing 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 continue to 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.

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 implemented 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 stand-alone solution.

In the next chapter, *Chapter 11, 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 Institute Allen for reading comprehension resources: <https://demo.allennlp.org/reading-comprehension>
- Kevin Clark, Minh-Thang Luong, Quoc V. Le, Christopher D. Manning, 2020, ELECTRA: Pre-training 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/>

11

Detecting Customer Emotions to Make Predictions

Sentiment analysis relies on the principle of compositionality. If we cannot understand parts of a sentence, how can we understand a whole 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." 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 not only the structure of the sequence but also its logical form.

We will then test several transformer models with complex sentences and some simple sentences. We will find that no matter which model we try, it will not work if it wasn'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 will end the chapter with a nice five-star interface you could implement on your website.

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

Let's begin by going through the SST.

Getting started: Sentiment analysis transformers

In this section, we 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:

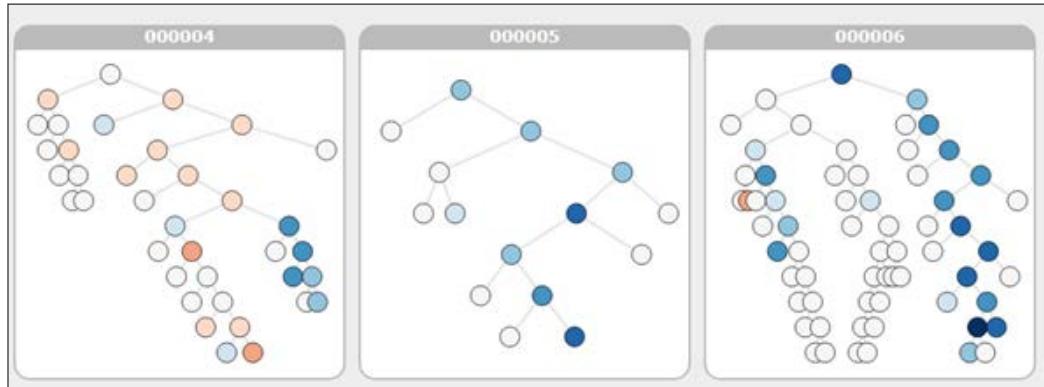


Figure 11.1: Graphs of sentiment trees

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 understanding:

- 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:

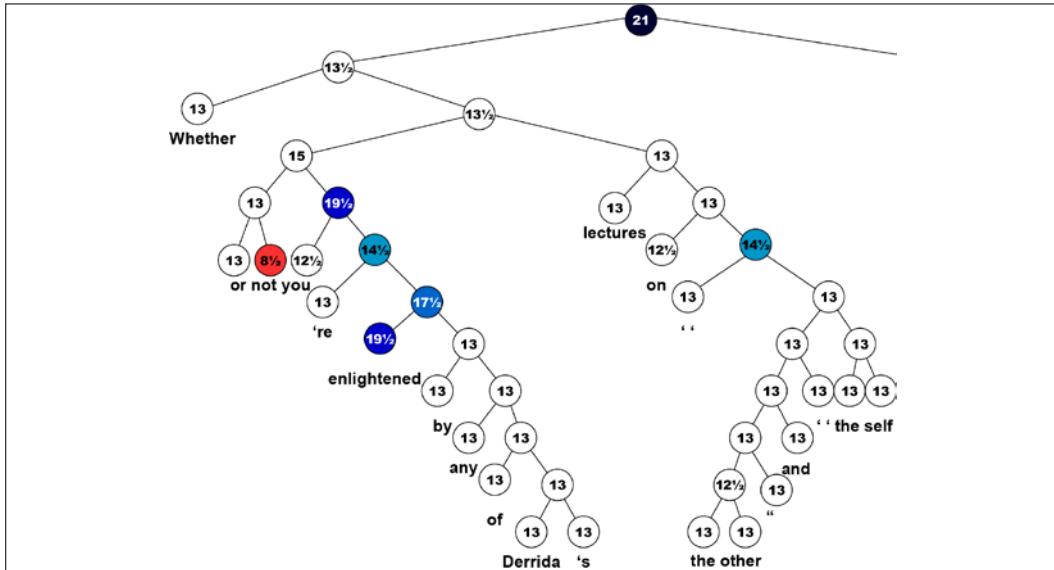


Figure 11.2: The "Whether" segment of a complex sentence

The sentence has been correctly split into two main parts. The second segment is also correct:

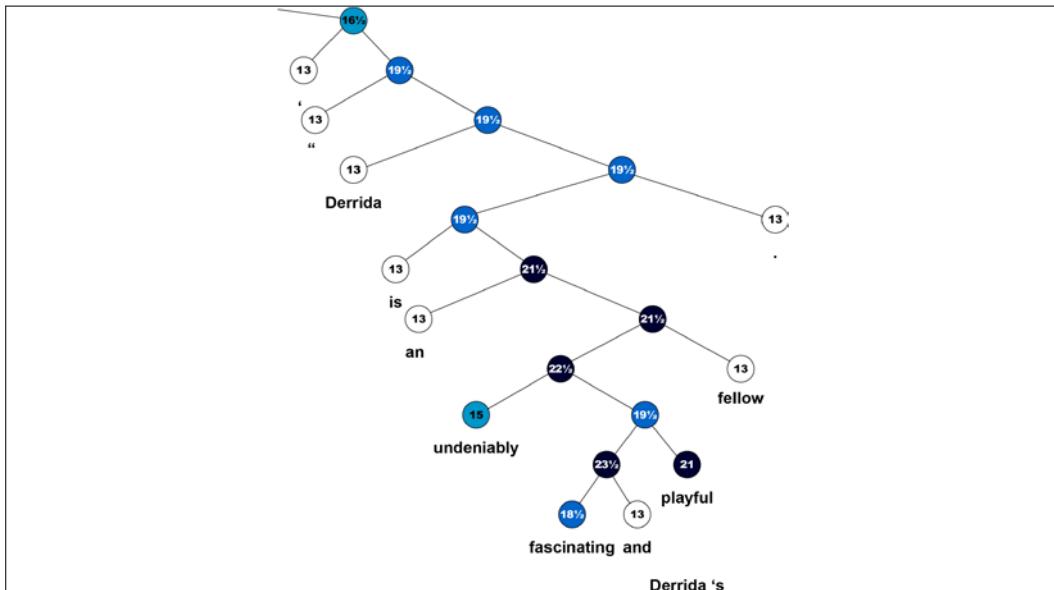


Figure 11.3: The main segment of a complex sentence

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 subtleties 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\nDerrida\'s lectures on the other and the self, Derrida is an undeniably\nfascinating and playful fellow."}' | \
allennlp predict https://storage.googleapis.com/allennlp-public-models/\nsst-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
}
```

You can take a few minutes if necessary to go through the description of a BERT architecture in the *BERT model configuration* section in *Chapter 2, Fine-Tuning BERT Models*, to take full advantage of this model.

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]
```

The output also contains the token IDs (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

In this section, we will run a sentiment analysis task on several Hugging Face transformer models to see which ones produce the best results and the 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  
}
```

You can take a few minutes if necessary to go through the description of a BERT architecture in the *BERT model configuration* section in *Chapter 2, Fine-Tuning BERT Models*, to take full advantage of this model.

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 Model10x  
    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 configuration=1, default=0
CS=classify(sequence,M)
print(CS)

```

In this case, `seq=3` is activated so that we can 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

In this section, we 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 2, Fine-Tuning BERT Models*
- For pretraining, you can refer to *Chapter 3, 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 **Tags: All** drop-down list:

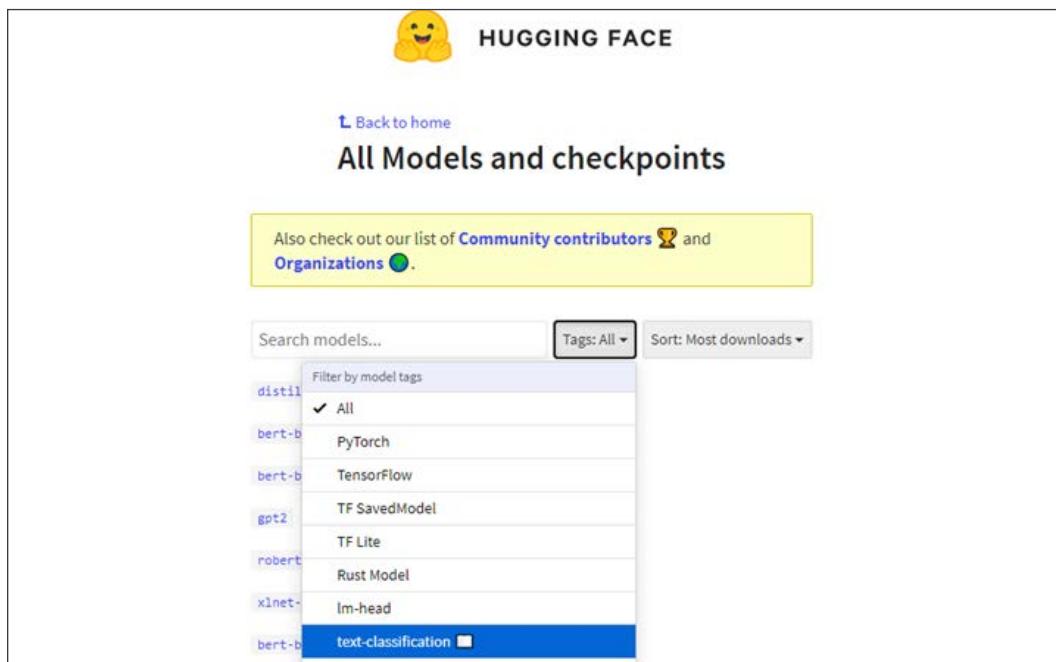


Figure 11.4: Selecting text-classification models

A list of transformer models trained for text classification will appear:



Figure 11.5: 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:



Figure 11.6: 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 this book's writing, 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 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, for example, we will get both false negatives and false positives. *Regular human intervention will still be mandatory for several more years.*

Let's give a MiniLM model a try.

MiniLM-L12-H384-uncased

MiniLM-L12-H384-uncased optimizes the size of the last self-attention layer of the teacher, among other tweaks of a BERT model, to obtain better performances. It has 12 layers, 12 heads, and 33M 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 score:

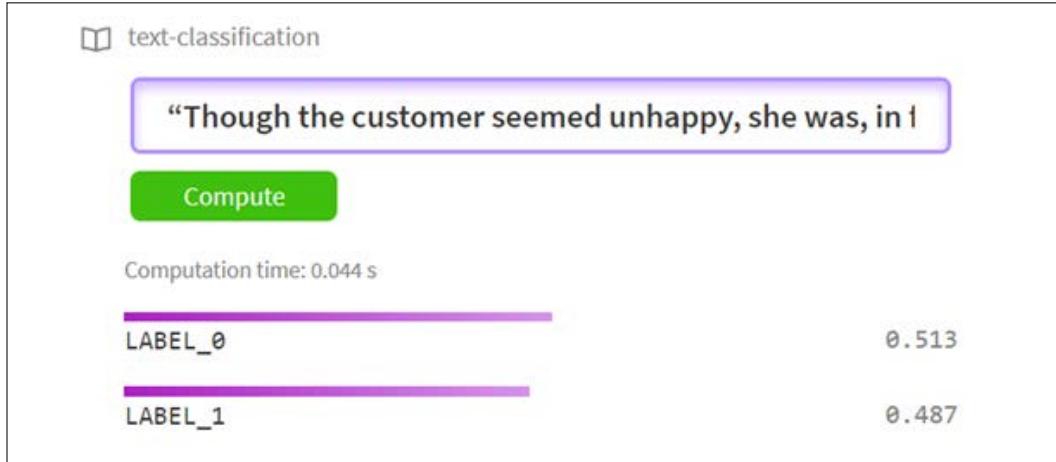


Figure 11.7: Complex sentence sentiment analysis

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 to solve the interpretation of a complex sentence when we are trying to determine what a customer meant. 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 though it remains neutral:



Figure 11.8: 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! It is very well-designed.

Let's run it with a friendly and positive sentence in English:

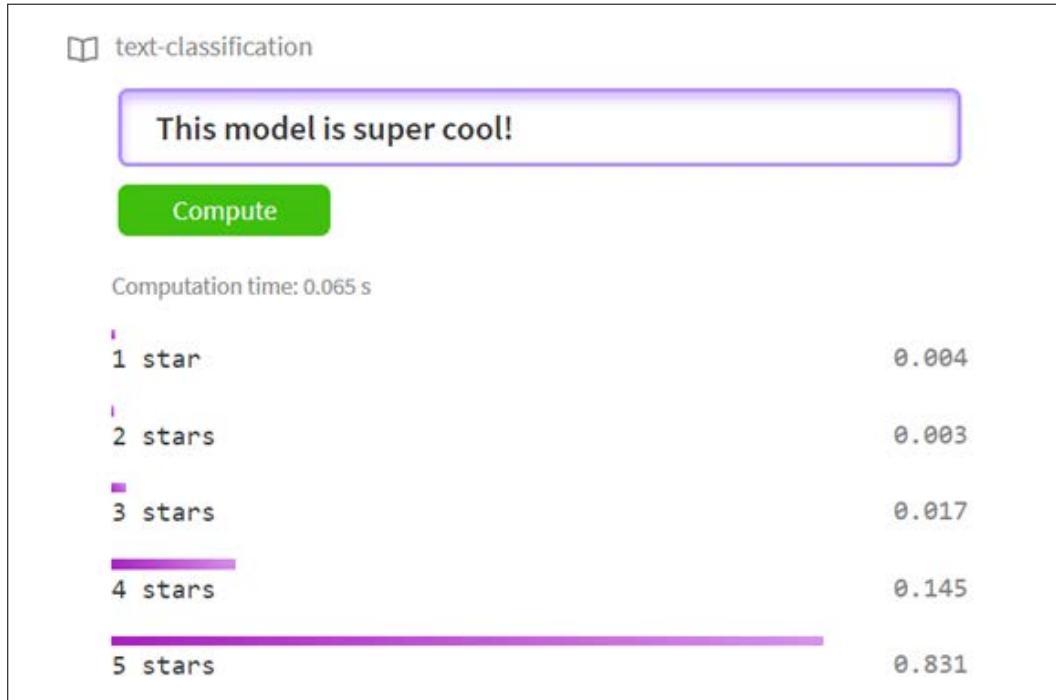


Figure 11.9: Sentiment analysis in English

Let's try it in French with "Ce modèle est super bien!" ("this model is super good," meaning "cool"):



Figure 11.10: 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:

```
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.

Summary

In this chapter, we went through some advanced theory. 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. 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.

Finally, 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.

In the next chapter, *Chapter 12, 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, Veselin Stoyanov, 2019, RoBERTa: A Robustly Optimized BERT Pretraining Approach: <https://arxiv.org/pdf/1907.11692.pdf>*
- The Allen Institute for AI: <https://allenai.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>*

12

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 rotate 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.

In this chapter, we will tackle the hot topics of the day. We will work on checking 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 opinion 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 every day.

We will first 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 not only physically need but also 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 his 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. Many of our decisions are based on emotions, not reason.

Let's translate these concepts into a behavioral flow chart 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. We become nervous, agitated, and it wears us down like a short-circuit in a toaster.

We have many examples to think about in 2021! Should we wear a mask with COVID-19? Are COVID-19 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 that a significant portion of fake news for one side is the truth of the other side!

We are in January 2021, and the US Republicans and Democrats still do not agree on the November 2020 election's outcome! Each side accuses the other of "fake news."

We could go on and find scores of other topics by just opening one newspaper and then reading another view in another opposing one! Some common-sense premises to this chapter can be drawn from these examples:

- Trying to find a transformer model that will detect fake news automatically makes no sense. In the world of social media and multi-cultural 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. Resolving the problem of fake news is exactly like trying to resolve a conflict between two parties or within our own minds.

My recommendation in this chapter and life is to analyze each conflictual tension; deconstruct 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 on the matter.

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 an actual 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 11, Detecting Customer Emotions to Make Predictions*.

We will first install `allenlp-models`:

```
!pip install allenlp==1.0.0 allenlp-models==1.0.0
```

We then run the next cell to analyze the Tweet:

```
!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]
```

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 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** and then click on **Simple Gradients Visualization** to obtain the following representation:

Sentence:

```
<s> Climate change is b og us . It âĢ L s a plot
</s>
```



Visualizing the top 3 most important words.

Figure 12.1: Visualizing the top most important words

Surprisingly, "climate" + "is" + "bogus" mostly influenced the result. The political aspect of "plot" came after.

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 next 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. A mind storm is building up! 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. 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 12.2 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, that does not mean that the news is not fake. Here are some of the reasons 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 people in Europe 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 not fake. If two parties disagree, this will lead to a conflict:



Figure 12.2: 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 more often than not, a person will 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 that 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 the contribution of humans 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. In this section, we 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. 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 12.3 defines a roadmap for a rational fake news analysis process. The process contains transformer NLP tasks and traditional functions:

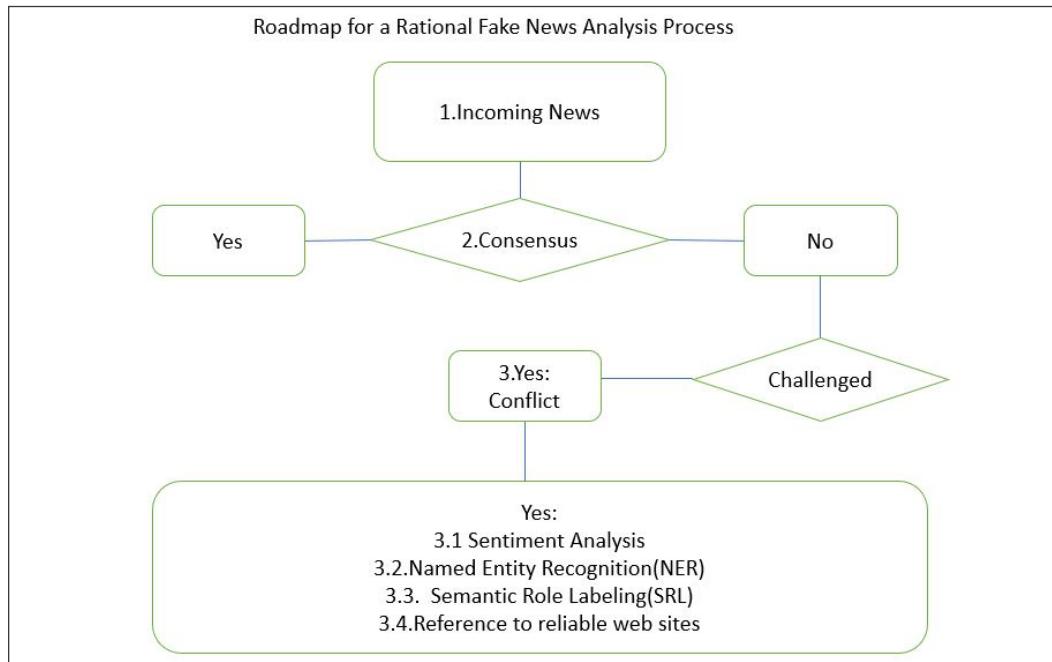


Figure 12.3: Going from emotional reactions to fake news to rational representations

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.org resources to run a RoBERTa-large transformer in our `Fake_News.ipynb` notebook. We will use AllenNLP.org's visual tools to visualize the keywords and explanation. We introduced sentiment analysis in *Chapter 11, 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 10, Let Your Data Do the Talking: Story, Questions, and Answers*. We will use Hugging Face's BERT transformer model for the task. We will use AllenNLP.org's visual tools to visualize the entities and explanation.
- **3.3. Semantic Role Labeling (SRL)** to label verbs from social media messages for *Phase 3.4*. We described SRL in *Chapter 9, 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.

Gun control

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 is 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\na problem. I raised my kids right so they have guns too and never hurt\nanything except rabbits."}' | \nallennlp predict https://storage.googleapis.com/allennlp-public-models/\nsst-roberta-large-2020.06.08.tar.gz -
```

The prediction is positive:

```
prediction: {"logits": [1.9383275508880615, -1.6191326379776],\n"probs": [0.9722791910171509, 0.02772079035639763]}
```

We will now visualize the result on AllenNLP.org. **SmoothGrad Visualization** provides the best explanation:

Sentence:

<s> I have had r ifles and guns for years and never had a problem . I
raised my kids right so they have guns too and never h urt anything
except r abb its . </s>

Figure 12.4: SmoothGrad Visualization of a sentence

The explanation shows that Afirst78 "never" + "problem" + "guns."



Results may vary over time. 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`: "never" + "problem" + "guns" can be extracted and noted for further analysis.

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 **Simple Gradients Visualization** provides the best result:

✓ Simple Gradients Visualization

See saliency map interpretations generated by [visualizing the gradient](#).

Sentence:

<s> I have **heard** gun shots all my life in my ne igh bor hood , have lost many friends , and am **af**raid to go out at night . </s>

Figure 12.5: Simple Gradients Visualization of a sentence

The keywords are "heard" + "afraid" for function 2 of this section:

`Fake_News_FUNCTION_2`: "heard" + "afraid" + "guns" 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: "never" + "problem" + "guns."`
Afirst78 probably lives in a mid-western state in the US. Many of these states have small populations, are very quiet, and enjoy very 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: "heard" + "afraid" + "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 such as those we describe in this chapter. *Better information is the key to less fake news battles.*

We will follow our process and apply named entity recognition to our sentence.

Named entity recognition (NER)

This chapter aims to show that by using several transformer methods, the user will benefit from a broader perception of a message through different angles. In production mode, an HTML page could sum up this chapter's transformer methods and even contain other transformer tasks.

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.

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\na problem. I raised my kids right so they have guns too and never hurt\nanything except rabbits."}' | \\\nallennlp predict https://storage.googleapis.com/allennlp-public-models/\nbert-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\n[V: had] [ARG1: rifles and guns] [ARGM-TMP: for years] and never had a\nproblem ..."}]
```

Now let's go into visual detail on AllenNLP.org 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*:



Figure 12.6: SRL for the verb "had"

The arguments of "had" sum up `Afirst78`'s experience: "I" + "rifles and guns" + "for years."

The arguments of "raised" display Afirst78's parental experience:



Figure 12.7: SRL verb and arguments for the verb "raised"

The arguments explain many pro-gun positions: "my kids" + "have guns" + "never hurt anything."

The verb "hurt" follows the same line of thought as seen in the formatted text version of the SRL task when clicking on **Text**:

```
[ARGM-NEG: never] [V: hurt] [ARG1: anything except rabbits].
```

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" + "never hurt anything"

Now let's explore the gun control message.

Gun control SRL

We will first run the Facebook message in `Fake_New.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.org](https://allenai.org/), then to the **Semantic Labeling Section**. We enter the sentence and run the transformer model. The verb "heard" shows the tough reality of this message:

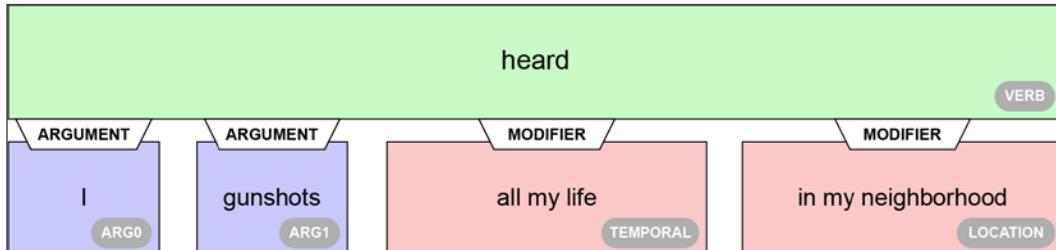


Figure 12.8: SRL representation of the verb "heard"

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:

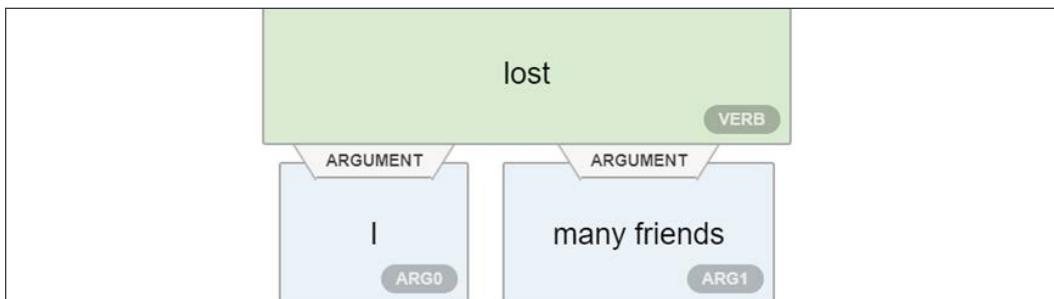


Figure 12.9: SRL representation of the verb "lost"

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:

- **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 quiet large 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. NLP transformers are like a mathematical calculator. They 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 are there to help users understand messages more deeply, not to 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 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 have 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 die.³⁵

www.hsph.harvard.edu › magazine ▾ [Traduire cette page](#)

Guns & Suicide | Harvard Public Health Magazine | 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 12.10: 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 12.11: 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. My recommendation is to never let 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 that no matter what you think, the bottom line with gun possession is 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

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 focus on analyzing 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.org](https://allenai.org/).

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.org, the **Semantic Role Labeling** section, run the sentence, and look at the result. The verb "hospitalized" shows the member is staying close to the facts:

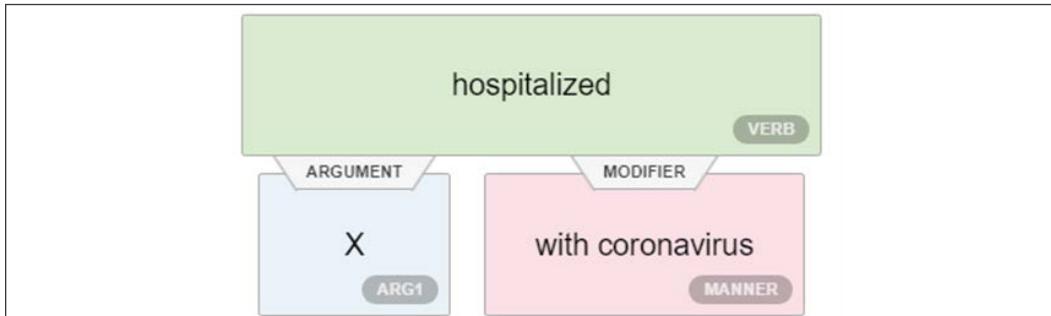


Figure 12.12: SRL arguments of the verb "hospitalized"

The message is simple: "X" + "hospitalized" + "coronavirus."

The verb "requested" shows that the message is becoming political:



Figure 12.13: SRL arguments of the verb "requested"

We don't know if the person requested the former President to pray or he 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 that was banned from Twitter. I took the names out and paraphrased it, and toned it down. Still, when we run it on AllenNLP.org 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:

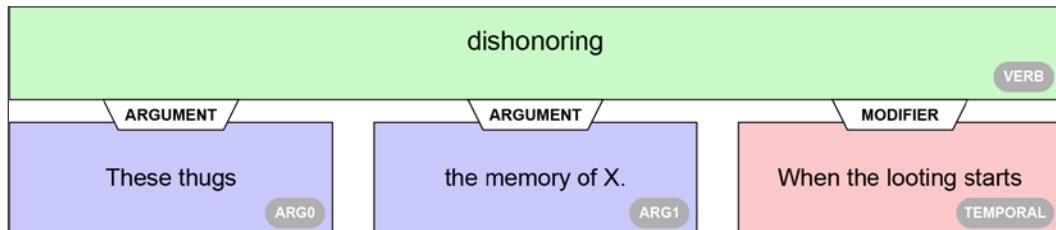


Figure 12.14: SRL arguments of the verb "dishonoring"

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 to listen to others and try to keep 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.

Looking for the silver bullet

Looking for a silver bullet transformer model can be time-consuming or rewarding, depending on how much time and money you want to spend on continually changing models.

For example, a new approach to transformers can be found through *disentanglement*. *Disentanglement* in AI allows you to separate the features of a representation 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 in December 2020 using 1.5B parameters.

Should you stop everything you are doing on transformers and rush to this model, integrate your data, train the model, test it, and implement it?

It is very probable that by the end of 2021, another model will beat this one and so on. Should you change models all of the time in production? That will be your decision.

You can also choose to design better training methods.

Looking for reliable training methods

Looking for reliable training methods with smaller models such as the PET designed by Timo Schick, covered in *Chapter 6, Text Generation with OpenAI GPT-2 and GPT-3 Models*, can also be a solution.

Why? Being in a good position on the SuperGLUE leaderboard does not mean that the model will provide a high quality of decision-making for medical, legal, and other critical areas for sequence predictions.

Looking for customized training solutions for a specific topic could be more productive than trying all the best transformers on the SuperGLUE leaderboard.

Take your time to think about implementing transformers to find the best approach for your project.

We will now conclude the chapter and book.

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 now up to you to take these transformer tools and ideas to implement them to make the world a better place for all!

You will write the next chapter in real life!

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/>

Appendix: Answers to the Questions

Chapter 1, Getting Started with the Model Architecture of the Transformer

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 an LSTM or CNN layers. (True/False)
True.
6. Attention examines all of the tokens in a sequence, not just the last one. (True/False)
True.

7. A transformer uses a positional vector, not positional encoding. (True/False)
False. A transformer uses positional encoding. The original Transformer model does not have an additional positional vector. Positional encoding is added to the input once it is processed.
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 2, 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 with **Masked Language Modeling (MLM)**. (True/False)
True.
6. BERT pretrains with **Next Sentence Predictions (NSP)**. (True/False)
True.
7. BERT pretrains 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 3, 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 6 decoder layers. (True/False)
False. BERT contains 6 encoder layers, not decoder layers.
8. Masked language modeling predicts a word contained in a mask token in a sentence. (True/False)
True.
9. A BERT-like model has no self-attention sub-layers. (True/False)
False. BERT has self-attention layers.
10. Data collators are helpful for backpropagation. (True/False)
False. Data collators are part of the dataset class.

Chapter 4, Downstream NLP Tasks with Transformers

1. Machine intelligence uses the same data as humans to make predictions. (True/False)
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 if one sequence entails another sequence. (True/False)
True.
6. A Winograd Schema predicts if a verb is spelled correctly. (True/False)
False. Winograd schemas mostly 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)
False. Transformer models beat human baselines for GLUE and do the same for SuperGLUE. As we keep increasing the level of SuperGLUE benchmarks, the models will continue to progress and beat the human baseline standards.
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 5, Machine Translation with the Transformer

1. Machine translation has now exceeded human baselines. (True/False)
False. Machine translation is one of the toughest 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 it into another language is not the same process as representing English and then translating it into another language. The language structures are not the same.
7. The original Transformer multi-head attention sub-layer has 2 heads. (True/False)
False. Each attention sub-layer has 8 heads.
8. The original Transformer encoder has 6 layers. (True/False)
True.
9. The original Transformer encoder has 6 layers but only 2 decoder layers. (True/False)
False. There are 6 decoder layers.
10. You can train transformers without decoders. (True/False)
True. The architecture of BERT only contains encoders.

Chapter 6, Text Generation with OpenAI GPT-2 and GPT-3 Models

1. A zero-shot method trains the parameters once. (True/False)
False. No the parameters of the model are first trained through as many episodes as necessary. Zero-shot means that downstream tasks are performed without additional fine-tuning.
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 GPT parameter model on a machine with less than 8 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. We will be able to build models with increasing numbers of parameters and connections.

Chapter 7, 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. NLP text summarization works for any text. (True/False)
False.
10. Hugging Face is a framework that makes transformers easier to implement. (True/False)
True.

Chapter 8, 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 a 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 9, 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 10, 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 11, 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 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 the proper 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 before changing models. (True/False)
True.

Chapter 12, 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.

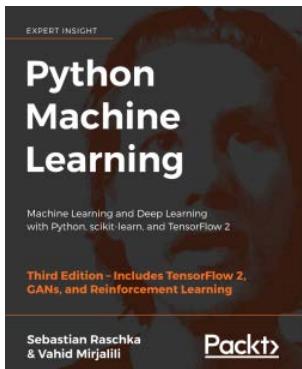
7. A rational approach to fake news can help clarify one's position. (True/False)
True.
8. Connecting transformers to reliable websites can help somebody understand why some news is fake. (True/False)
True.
9. Transformers can make summaries of reliable websites to help us understand some of the topics 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.

Share your experience

Thank you for taking the time to read this book. If you enjoyed this book, help others to find it. Leave a review at: <https://www.amazon.com/dp/1800565798>

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Python Machine Learning - Third Edition

Sebastian Raschka and Vahid Mirjalili

ISBN: 9781789955750

- Master the frameworks, models, and techniques that enable machines to 'learn' from data
- Use scikit-learn for machine learning and TensorFlow for deep learning
- Apply machine learning to image classification, sentiment analysis, intelligent web applications, and more

Other Books You May Enjoy

- Build and train neural networks, GANs, and other models
- 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



Hands-On Explainable AI (XAI) with Python

Denis Rothman

ISBN: 9781800208131

- Plan for XAI through the different stages of the machine learning life cycle
- Estimate the strengths and weaknesses of popular open-source XAI applications
- Examine how to detect and handle bias issues in machine learning data
- Review ethics considerations and tools to address common problems in machine learning data
- Share XAI design and visualization best practices
- Integrate explainable AI results using Python models
- Use XAI toolkits for Python in machine learning life cycles to solve business problems

Index

A

AllenNLP

Amazon Web Services (AWS) 169
associative neutral networks 3

B

benchmark tasks, SuperGLUE

BoolQ 114
Commitment Bank (CB) 114
defining 113
Multi-Sentence Reading Comprehension (MultiRC) 115, 116
Reading Comprehension with Commonsense Reasoning Dataset (ReCoRD) 116, 117
Recognizing Textual Entailment (RTE) 118
Winograd Schema Challenge (WSC) 118, 119
Words in Context (WiC) 118

BERT-based model

RoBERTa 75
DistilBERT 299
DeBERTa 336
KantaiBERT 76
used, for performing SRL experiments 249

BERT-base multilingual model 307, 308

BERT model

fine-tuning 50-52
pretraining 50-52

BERT model, fine-tuning 53

attention masks, creating 59
batch size, selecting 60, 61
BERT tokenizer, activating 57
BERT tokens, creating 57
configuration, initializing 61, 62
CUDA, specifying 55

data, converting into torch tensors 60
data, processing 58
dataset, loading 55-57
data, splitting into training and validation sets 59
evaluation, using holdout dataset 69, 70
evaluation, using Matthews Correlation Coefficient (MCC) 70
GPU, activating 53
Hugging Face PyTorch interface, installing 54
hyperparameters, for training loop 66
individual batches score 71
iterator, creating 60, 61
label lists, creating 57
Matthews evaluation, for whole dataset 72
modules, importing 54, 55
optimizer grouped parameters 65
prediction, using holdout dataset 69
sentences, creating 57
training evaluation, displaying 68
training loop 66, 67

BERT uncased base model

loading 63, 64

BertViz

attention heads, displaying 157-159
attention heads, processing 157, 159
HTML function, defining 156, 157
installing 156
modules, importing 156
running 156

Bidirectional Encoder Representations from Transformers (BERT)

encoder stack 44-46
Hugging Face PyTorch interface, installing 54

Bilingual Evaluation Understudy Score (BLEU) 138-142

machine translation, evaluating with 138
billion-parameter transformer models
content size 153, 159
maximum path length 153
rise 151
size, checking 152
BoolQ 114
Byte Pair Encoding (BPE) 76

C

chencherry smoothing 142
Choice of Plausible Answers (COPA) 112
cognitive dissonance
used, for triggering emotional reactions 313, 314
Commitment Bank (CB) 114
complex sentence 296, 297
Compute Unified Device Architecture (CUDA) 55, 86
conflictual Tweet
analyzing 314-316
Convolutional Neural Network (CNN) 4
Corpus of Linguistic Acceptability (CoLA) 52, 120
customer behavior
predicting, with sentiment analysis 299

D

datasets
and tokenizers, matching 216
DeBERTa
decision making 163, 164
decoder stack, Transformer 34, 36
attention layers 37
FFN sub layer 37, 38
output embedding 36
position embedding 36
post-Linear layer 38
post-LN 37, 38
decoding
from Transformer 145
DistilBERT
for SST 303, 304
using, for sentiment analysis 299-301
document summarization, with T5-large transformer model 207

Bill of Rights sample 210
corporate law sample 211, 212
sample 209
summarization function, creating 207, 208
downstream tasks, running 119
Corpus of Linguistic Acceptability (CoLA) 120
Microsoft Research Paraphrase Corpus (MRPC) 122, 123
Stanford Sentiment TreeBank (SST-2) 121
Winograd schemas 123

E

encoder stack, BERT 44-46
pretraining input environment, preparing 47
encoder stack, Transformer 6-8
Feedforward Network (FFN) 33, 34
input embedding 8-10
positional encoding 11-14
positional encoding, adding to embedding vector 15-17
extra-large (XL) transformer models 270

F

fake news
behavioral representation 317
behavioral representation, phases 317-319
emotional reactions 312
gun control 321
rational approach 319
resolution roadmap, defining 320, 321
FeedForward Network (FFN) 5
Few-Shot (FS) 166
fine-tuning (FT) 166

G

General Language Understanding Evaluation (GLUE) 55
migrating, to SuperGlue 108, 109
URL 108
Generative Pre-Training model (GPT) 151
geometric evaluations 139, 140
tokenized data, controlling 233-236
used, for generating unconditional samples 232, 233
GPT-2 117M model trained, on datasets

context and completion example 185-188

GPT-2 transformer model

dataset, encoding 182
dataset, training 183
GPT-2 117M model 185-188
interacting with 178, 179
tokenized data, controlling 233-236
N Shepperd training files 182
training 179
training model directory, creating 184, 185
training, prerequisites 180
training process, steps 180, 182
used, for generating unconditional samples 232, 233

GPT-3 transformer model 155

Graphics Processing Unit (GPU) 85

GPU, activating 53, 169, 170

H

Haystack

exploring, with RoBERTa model 289, 290

Hopfield networks 3

Hugging Face transformer resources 199-202

Hugging Face models

reference link 301

human baselines

higher standards 110
versus transformer performances 106

human intelligence transduction and induction

versus machine intelligence transduction and induction 104

human quality control 220

human transductions 130

human translations 130

I

interactive sentiment treebank

reference link 295

K

KantaiBERT 76, 77

KantaiBERT, building steps 77

configuration, defining of model 86, 87
data collator, defining 94

dataset, building 93

dataset, loading 78, 79

files, saving to disk 82, 83

final model (+tokenizer + config),
saving to disk 96

Hugging Face transformers, installing 79, 80

language modeling, with FillMaskPipeline 97

model, initializing 87-89

model, pretraining 95, 96

parameters, exploring 89-93

resource constraints, checking 85, 86

tokenizer, reloading in transformers 87

tokenizer, training 80-82

trained tokenizer files, loading 84, 85

trainer, initializing 94, 95

L

Locality Sensitivity Hashing (LSH) 159

location entity questions 274

heuristics, applying 274, 275

project management 276, 277

M

machine intelligence transduction and induction

versus human intelligence transduction and induction 105, 106

machine translations 128-130

machine translation, evaluating with BLEU 138

chencherry smoothing, applying 142

geometric evaluations 139, 140

smoothing technique, applying 141

Masked Language Modeling (MLM) 47, 49, 52, 77, 94, 97, 102, 280

Matthews Correlation Coefficient (MCC) 53

used, for evaluating predictions 70

measurement scoring methods, used by GLUE and SuperGlue

accuracy score 106

F1-score 107

Matthews Correlation Coefficient (MCC) 107

method outperforms models 267

methods, for question-answering

NER 271

SRL 278

trial and error 268

Microsoft Research Paraphrase Corpus (MRPC) 122, 123

MiniLM-L12-H384-uncased 304, 305

Multi-Genre Natural Language Inference (MultiNLI) task

- reference link 305

multi-head attention sub-layer 17

- architecture 17-19
- final representations 26, 28
- input, representing 20
- input vectors, multiplying by weight matrices 23, 24
- output 30
- output concatenation 31, 32
- post-layer normalization (Post-LN) 32, 33
- results, adding 30
- results, summarizing 28, 29
- scaled attention scores 24, 25
- scaled softmax attention scores, for each vector 25
- weight matrices, initializing 21, 22

Multi-Sentence Reading Comprehension (MultiRC) 115

N

Named Entity Recognition (NER) 324, 325

- gun control 328, 329
- pro-guns 326, 327
- used, for finding questions 271-273

Natural Language Inferences (NLI) 218

Natural Language Toolkit (NLTK) 138

- nltk.translate package 142

Next Sentence Prediction (NSP) 49, 50

nltk.translate package

- reference link 142

O

One-Shot (1S) 166

OpenAI GPT models

- architecture 164
- decoder layers, stacking 167, 168
- fine-tuning models, migrating to zero-shot models 164-166

original Transformer

- architecture 4-6

background 2-4

decoder stack 34

decoding from 145

encoder stack 6, 8

limits 115

performance 38

pretraining 76

training 38

used, for music generation 189

Out-Of-Vocabulary (OOV) 135

P

Pattern-Exploiting Training (PET) 161, 162

- philosophy 162, 163
- URL 163

person entity questions 277

positional encoding (PE) 11-14

- adding, to embedding vector 15, 16
- multi-head attention sub-layer 17

post-layer normalization (Post-LN) 32

predicate 245

pretrained BERT-based model

- architecture 247, 248
- running 247
- SRL environment, setting up 248

pretrained weights

- used, for initializing transformer model 144

pretraining input environment preparation 47

- masked language modeling 47, 48
- Next Sentence Prediction (NSP) 48-50

principle of compositionality

Project Gutenberg

- URL 78

project management

- difficult project 276
- easy project 276
- intermediate project 276
- very difficult project 277

Q

question-answering

- ELECTRA, using 279, 280
- implementing, steps 287, 288
- in transformers 267, 268
- methods 268

R

random question
asking 267
Reading Comprehension with Commonsense Reasoning Dataset (ReCoRD) 116
Recurrent Neural Networks (RNNs) 3
Reformer 159, 160
reliable training methods 337
RoBERTa
RoBERTa-large-mnli 305, 306
Robustly Optimized BERT Pretraining Approach (RoBERTa) 76
Haystack, exploring with 289
sentiment analysis 297, 298

S

semantic role 245
Semantic Role Labeling (SRL) 244-335
complex samples 256-262
defining 244, 245
gun control SRL 328, 329
pro-guns SRL 326, 327
project management constraints 281
samples 249-255
used, for finding questions 282-287
visualizing 245, 246
sentiment analysis 321
customer behavior, predicting 299
gun control analysis 323, 324
pro-guns analysis 322
with DistilBERT 299-301
with Hugging Face's models list 301-303
with RoBERTa-large 297, 298
sentiment analysis transformers 294
sentiment trees
graph 295
silver bullet transformer model 336
Situations With Adversarial Generations (SWAG) 52
smoothing technique
applying 141
chencherry smoothing 142
standard NLP tasks
trained conditional samples, generating 236
unconditional samples, generating with GPT-2 232, 233

with specific vocabulary 231
Stanford Sentiment Treebank (SST) 294-299
Stanford Sentiment TreeBank (SST-2) 121
Subject Matter Experts (SMEs) 277
SuperGLUE
benchmark tasks, defining 113
evaluation process, working 111-113

T

Text-To-Text Transformer (T5) 197-199
Bill of Rights sample 237-239
T5-large transformer model 204-207
task-specific formats
unifying 196, 197
Tensor2Tensor (T2T) 142
text completion, with GPT-2 168
345M parameter GPT-2 model,
downloading 173, 174
GPU, activating 169, 170
intermediate instructions 175
model, defining 176, 177
model, importing 176, 177
OpenAI GPT-2 repository, cloning 170, 172
requirements, installing 172
TensorFlow version, checking 172
text summarization, with T5 199
Hugging Face 199
T5-large transformer model, initializing 202
text-to-text transformer models
rise 195, 196
tokenizer
See Tokenizing sentences
Tokenizing sentences
byte-level byte-pair encoding
(byte-level BPE) 77, 226, 231, 235
processing 52
training 76
Word2Vec 221-230
trained conditional samples
generating 236
transduction 104
Transformer
architecture 4-6
background 2-4
decoder stack 34
decoding from 145

encoder stack 6, 8
performance 38
pretraining 76
training 38
used, for music generation 189

transformer datasets, best practices
post-processing 219
preprocessing 218, 219
standard heuristics, applying to 218

transformer model
creating 143, 144
initializing, with pretrained weights 144

transformer performances, versus human baselines 106
benchmark tasks and datasets 108
models with metrics, evaluating 106
SuperGLUE benchmark tasks, defining 113

transformers
inductive inheritance 102

translation
de-tokenizing 145, 146
displaying 145, 146

Trax
installing 143
translations with 142-146

U

universal text-to-text model
designing 194, 195

W

Winograd Schema Challenge (WSC) 123
Winograd schemas 123, 124

WMT dataset
preprocessing 131
raw data, preprocessing 131-134

WMT dataset, preprocessing
finalizing 134-138

Words in Context (WiC) 118

Word2Vec tokenization 221-230
cases 224-230

Workshops on Machine Translation (WMT)
datasets 131
URL 38

Z

Zero-Shot (ZS) 166

