

EXPERT INSIGHT

---

# Transformers for Natural Language Processing and Computer Vision

Explore Generative AI and Large Language Models with  
Hugging Face, ChatGPT, GPT-4V, and DALL-E 3

Third Edition



---

Denis Rothman

**<packt>**

# **Transformers for Natural Language Processing and Computer Vision**

**Third Edition**

**Explore Generative AI and Large Language  
Models with Hugging Face, ChatGPT, GPT-4V,  
and DALL-E 3**

Denis Rothman



BIRMINGHAM—MUMBAI

# **Transformers for Natural Language Processing and Computer Vision**

## **Third Edition**

Copyright © 2024 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.

**Publishing Product Manager:** Bhavesh Amin

**Acquisition Editor – Peer Reviews:** Tejas Mhasvekar

**Project Editor:** Janice Gonsalves

**Content Development Editor:** Bhavesh Amin

**Copy Editor:** Safis Editing

**Technical Editor:** Karan Sonawane

**Proofreader:** Safis Editing

**Indexer:** Rekha Nair

**Presentation Designer:** Ajay Patule

First published: January 2021

Second edition: February 2022

Third edition: February 2024

Production reference: 1280224

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-80512-872-4

[www.packt.com](http://www.packt.com)

# Contributors

## About the author

**Denis Rothman** graduated from Sorbonne University and Paris Diderot University, designing one of the first patented encoding and embedding systems. He authored one of the first patented AI cognitive robots and bots. He began his career delivering **Natural Language Processing (NLP)** chatbots for Moët et Chandon and as an **AI tactical defense optimizer** for Airbus (formerly Aerospatiale).

Denis then authored an **AI resource optimizer** for IBM and luxury brands, leading to an **Advanced Planning and Scheduling (APS)** solution used worldwide.

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

## About the reviewer

**George Mihaila** has 7 years of research experience with transformer models, having started working with them since they came out in 2017. He is a final-year PhD student in computer science working in research on transformer models in **Natural Language Processing (NLP)**. His research covers both Generative and Predictive NLP modeling.

He has over 6 years of industry experience working in top companies with transformer models and machine learning, covering a broad area from NLP and Computer Vision to Explainability and Causality. George has worked in both science and engineering roles. He is an end-to-end Machine Learning expert leading Research and Development, as well as MLOps, optimization, and deployment.

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

# Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://www.packt.link/Transformers>



# Contents

## Preface

Who this book is for

What this book covers

To get the most out of this book

Get in touch

## 1. What Are Transformers?

How constant time complexity O(1) changed our lives forever

O(1) attention conquers O(n) recurrent methods

Attention layer

Recurrent layer

The magic of the computational time complexity of an attention layer

Computational time complexity with a CPU

Computational time complexity with a GPU

Computational time complexity with a TPU

TPU-LLM

A brief journey from recurrent to attention

A brief history

From one token to an AI revolution

From one token to everything

Foundation Models

From general purpose to specific tasks

The role of AI professionals

The future of AI professionals

What resources should we use?

Decision-making guidelines

The rise of transformer seamless APIs and assistants

Choosing ready-to-use API-driven libraries

Choosing a cloud platform and transformer model

Summary

Questions

References

Further reading

## 2. Getting Started with the Architecture of the Transformer Model

## The rise of the Transformer: Attention Is All You Need

### The encoder stack

Input embedding

Positional encoding

Sublayer 1: Multi-head attention

Sublayer 2: Feedforward network

### The decoder stack

Output embedding and position encoding

The attention layers

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

### Training and performance

### Hugging Face transformer models

### Summary

### Questions

### References

### Further reading

## 3. Emergent vs Downstream Tasks: The Unseen Depths of Transformers

### The paradigm shift: What is an NLP task?

Inside the head of the attention sublayer of a transformer

Exploring emergence with ChatGPT

### Investigating the potential of downstream tasks

#### Evaluating models with metrics

Accuracy score

F1-score

MCC

#### Human evaluation

Benchmark tasks and datasets

Defining the SuperGLUE benchmark tasks

### Running downstream tasks

The Corpus of Linguistic Acceptability (CoLA)

Stanford Sentiment TreeBank (SST-2)

Microsoft Research Paraphrase Corpus (MRPC)

Winograd schemas

### Summary

### Questions

### References

### Further reading

## 4. Advancements in Translations with Google Trax, Google Translate, and Gemini

Defining machine translation

Human transductions and translations

Machine transductions and translations

Evaluating machine translations

Preprocessing a WMT dataset

Preprocessing the raw data

Finalizing the preprocessing of the datasets

Evaluating machine translations with BLEU

Geometric evaluations

Applying a smoothing technique

Translations with Google Trax

Installing Trax

Creating the Original Transformer model

Initializing the model using pretrained weights

Tokenizing a sentence

Decoding from the Transformer

De-tokenizing and displaying the translation

Translation with Google Translate

Translation with a Google Translate AJAX API Wrapper

Implementing googletrans

Translation with Gemini

Gemini's potential

Summary

Questions

References

Further reading

## 5. Diving into Fine-Tuning through BERT

The architecture of BERT

The encoder stack

Preparing the pretraining input environment

Pretraining and fine-tuning a BERT model

Fine-tuning BERT

Defining a goal

Hardware constraints

Installing Hugging Face Transformers

[Importing the modules](#)  
[Specifying CUDA as the device for torch](#)  
[Loading the CoLA dataset](#)  
[Creating sentences, label lists, and adding BERT tokens](#)  
[Activating the BERT tokenizer](#)  
[Processing the data](#)  
[Creating attention masks](#)  
[Splitting the data into training and validation sets](#)  
[Converting all the data into torch tensors](#)  
[Selecting a batch size and creating an iterator](#)  
[BERT model configuration](#)  
[Loading the Hugging Face BERT uncased base model](#)  
[Optimizer grouped parameters](#)  
[The hyperparameters for the training loop](#)  
[The training loop](#)  
[Training evaluation](#)  
[Predicting and evaluating using the holdout dataset](#)  
    [Exploring the prediction process](#)  
    [Evaluating using the Matthews correlation coefficient](#)  
    [Matthews correlation coefficient evaluation for the whole dataset](#)  
[Building a Python interface to interact with the model](#)  
    [Saving the model](#)  
    [Creating an interface for the trained model](#)  
        [Interacting with the model](#)  
[Summary](#)  
[Questions](#)  
[References](#)  
[Further reading](#)

## 6. [Pretraining a Transformer from Scratch through RoBERTa](#)

[Training a tokenizer and pretraining a transformer](#)

[Building KantaiBERT from scratch](#)

[Step 1: Loading the dataset](#)

[Step 2: Installing Hugging Face transformers](#)

[Step 3: Training a tokenizer](#)

[Step 4: Saving the files to disk](#)

[Step 5: Loading the trained tokenizer files](#)

[Step 6: Checking resource constraints: GPU and CUDA](#)  
[Step 7: Defining the configuration of the model](#)  
[Step 8: Reloading the tokenizer in transformers](#)  
[Step 9: Initializing a model from scratch](#)  
    [Exploring the parameters](#)  
[Step 10: Building the dataset](#)  
[Step 11: Defining a data collator](#)  
[Step 12: Initializing the trainer](#)  
[Step 13: Pretraining the model](#)  
[Step 14: Saving the final model \(+tokenizer + config\) to disk](#)  
[Step 15: Language modeling with FillMaskPipeline](#)

[Pretraining a Generative AI customer support model on X data](#)  
    [Step 1: Downloading the dataset](#)  
    [Step 2: Installing Hugging Face transformers](#)  
    [Step 3: Loading and filtering the data](#)  
    [Step 4: Checking Resource Constraints: GPU and CUDA](#)  
    [Step 5: Defining the configuration of the model](#)  
    [Step 6: Creating and processing the dataset](#)  
    [Step 7: Initializing the trainer](#)  
    [Step 8: Pretraining the model](#)  
    [Step 9: Saving the model](#)  
    [Step 10: User interface to chat with the Generative AI agent](#)  
    [Further pretraining](#)  
    [Limitations](#)

[Next steps](#)

[Summary](#)

[Questions](#)

[References](#)

[Further reading](#)

## 7. [The Generative AI Revolution with ChatGPT](#)

[GPTs as GPTs](#)

[Improvement](#)

[Diffusion](#)

[New application sectors](#)

[Self-service assistants](#)

[Development assistants](#)

[Pervasiveness](#)

## The architecture of OpenAI GPT transformer models

The rise of billion-parameter transformer models

The increasing size of transformer models

Context size and maximum path length

From fine-tuning to zero-shot models

Stacking decoder layers

GPT models

## OpenAI models as assistants

ChatGPT provides source code

GitHub Copilot code assistant

General-purpose prompt examples

Getting started with ChatGPT – GPT-4 as an assistant

1. GPT-4 helps to explain how to write source code

2. GPT-4 creates a function to show the YouTube

presentation of GPT-4 by Greg Brockman on March 14, 2023

3. GPT-4 creates an application for WikiArt to display images

4. GPT-4 creates an application to display IMDb reviews

5. GPT-4 creates an application to display a newsfeed

6. GPT-4 creates a k-means clustering (KMC) algorithm

## Getting started with the GPT-4 API

Running our first NLP task with GPT-4

Steps 1: Installing OpenAI and Step 2: Entering the API key

Step 3: Running an NLP task with GPT-4

Key hyperparameters

Running multiple NLP tasks

## Retrieval Augmented Generation (RAG) with GPT-4

Installation

Document retrieval

Augmented retrieval generation

Summary

Questions

References

Further reading

## 8. Fine-Tuning OpenAI GPT Models

Risk management

Fine-tuning a GPT model for completion (generative)

1. Preparing the dataset

1.1. Preparing the data in JSON

1.2. Converting the data to JSONL

2. Fine-tuning an original model

3. Running the fine-tuned GPT model

4. Managing fine-tuned jobs and models

Before leaving

Summary

Questions

References

Further reading

## 9. Shattering the Black Box with Interpretable Tools

Transformer visualization with BertViz

Running BertViz

Step 1: Installing BertViz and importing the modules

Step 2: Load the models and retrieve attention

Step 3: Head view

Step 4: Processing and displaying attention heads

Step 5: Model view

Step 6: Displaying the output probabilities of attention heads

Streaming the output of the attention heads

Visualizing word relationships using attention scores with pandas

exBERT

Interpreting Hugging Face transformers with SHAP

Introducing SHAP

Explaining Hugging Face outputs with SHAP

Transformer visualization via dictionary learning

Transformer factors

Introducing LIME

The visualization interface

Other interpretable AI tools

LIT

[PCA](#)

[Running LIT](#)

[OpenAI LLMs explain neurons in transformers](#)

[Limitations and human control](#)

[Summary](#)

[Questions](#)

[References](#)

[Further reading](#)

## [10. Investigating the Role of Tokenizers in Shaping Transformer Models](#)

[Matching datasets and tokenizers](#)

[Best practices](#)

[Step 1: Preprocessing](#)

[Step 2: Quality control](#)

[Step 3: Continuous human quality control](#)

[Word2Vec tokenization](#)

[Case 0: Words in the dataset and the dictionary](#)

[Case 1: Words not in the dataset or the dictionary](#)

[Case 2: Noisy relationships](#)

[Case 3: Words in a text but not in the dictionary](#)

[Case 4: Rare words](#)

[Case 5: Replacing rare words](#)

[Exploring sentence and WordPiece tokenizers to understand the efficiency of subword tokenizers for transformers](#)

[Word and sentence tokenizers](#)

[Sentence tokenization](#)

[Word tokenization](#)

[Regular expression tokenization](#)

[Treebank tokenization](#)

[White space tokenization](#)

[Punkt tokenization](#)

[Word punctuation tokenization](#)

[Multi-word tokenization](#)

[Subword tokenizers](#)

[Unigram language model tokenization](#)

[SentencePiece](#)

[Byte-Pair Encoding \(BPE\)](#)

[WordPiece](#)

## Exploring in code

Detecting the type of tokenizer

Displaying token-ID mappings

Analyzing and controlling the quality of token-ID mappings

Summary

Questions

References

Further reading

## 11. Leveraging LLM Embeddings as an Alternative to Fine-Tuning

LLM embeddings as an alternative to fine-tuning

From prompt design to prompt engineering

Fundamentals of text embedding with NLTK and Gensim

Installing libraries

1. Reading the text file

2. Tokenizing the text with Punkt

Preprocessing the tokens

3. Embedding with Gensim and Word2Vec

4. Model description

5. Accessing a word and vector

6. Exploring Gensim's vector space

7. TensorFlow Projector

Implementing question-answering systems with embedding-based search techniques

1. Installing the libraries and selecting the models

2. Implementing the embedding model and the GPT model

2.1 Evaluating the model with a knowledge base: GPT can answer questions

2.2 Add a knowledge base

2.3 Evaluating the model without a knowledge base: GPT cannot answer questions

3. Prepare search data

4. Search

5. Ask

5.1 Example question

5.2 Troubleshooting wrong answers

Transfer learning with Ada embeddings

- [1. The Amazon Fine Food Reviews dataset](#)
  - [1.2. Data preparation](#)
- [2. Running Ada embeddings and saving them for future reuse](#)
- [3. Clustering](#)
  - [3.1. Find the clusters using k-means clustering](#)
  - [3.2. Display clusters with t-SNE](#)
- [4. Text samples in the clusters and naming the clusters](#)

[Summary](#)

[Questions](#)

[References](#)

[Further reading](#)

## [12. Toward Syntax-Free Semantic Role Labeling with ChatGPT and GPT-4](#)

[Getting started with cutting-edge SRL](#)

[Entering the syntax-free world of AI](#)

[Defining SRL](#)

[Visualizing SRL](#)

[SRL experiments with ChatGPT with GPT-4](#)

[Basic sample](#)

[Difficult sample](#)

[Questioning the scope of SRL](#)

[The challenges of predicate analysis](#)

[Redefining SRL](#)

[From task-specific SRL to emergence with ChatGPT](#)

[1. Installing OpenAI](#)

[2. GPT-4 dialog function](#)

[3. SRL](#)

[Sample 1 \(basic\)](#)

[Sample 2 \(basic\)](#)

[Sample 3 \(basic\)](#)

[Sample 4 \(difficult\)](#)

[Sample 5 \(difficult\)](#)

[Sample 6 \(difficult\)](#)

[Summary](#)

[Questions](#)

[References](#)

## Further reading

### 13. Summarization with T5 and ChatGPT

Designing a universal text-to-text model

The rise of text-to-text transformer models

A prefix instead of task-specific formats

The T5 model

Text summarization with T5

Hugging Face

Selecting a Hugging Face transformer model

Initializing the T5-large transformer model

Getting started with T5

Exploring the architecture of the T5 model

Summarizing documents with T5-large

Creating a summarization function

A general topic sample

The Bill of Rights sample

A corporate law sample

From text-to-text to new word predictions with OpenAI ChatGPT

Comparing T5 and ChatGPT's summarization methods

Pretraining

Specific versus non-specific tasks

Summarization with ChatGPT

Summary

Questions

References

Further reading

### 14. Exploring Cutting-Edge LLMs with Vertex AI and PaLM 2

Architecture

Pathways

Client

Resource manager

Intermediate representation

Compiler

Scheduler

Executor

PaLM

Parallel layer processing that increases training speed

Shared input-output embeddings, which saves memory  
No biases, which improves training stability  
Rotary Positional Embedding (RoPE) improves model quality  
SwiGLU activations improve model quality

## PaLM 2

Improved performance, faster, and more efficient  
Scaling laws, optimal model size, and the number of parameters  
State-of-the-art (SOA) performance and a new training methodology

## Assistants

Gemini  
Google Workspace  
Google Colab Copilot  
Vertex AI PaLM 2 interface  
Vertex AI PaLM 2 assistant

## Vertex AI PaLM 2 API

Question answering  
Question-answer task  
Summarization of a conversation  
Sentiment analysis  
Multi-choice problems  
Code

## Fine-tuning

Creating a bucket  
Fine-tuning the model

## Summary

## Questions

## References

## Further reading

15. Guarding the Giants: Mitigating Risks in Large Language Models
  - The emergence of functional AGI
  - Cutting-edge platform installation limitations
  - Auto-BIG-bench
  - WandB
  - When will AI agents replicate?

Function: `create\_vocab`

Process:

Function: `scrape\_wikipedia`

Process:

Function: `create\_dataset`

Process:

Classes: `TextDataset`, `Encoder`, and `Decoder`

Function: `count\_parameters`

Function: `main`

Process:

Saving and Executing the Model

Risk management

Hallucinations and memorization

Memorization

Risky emergent behaviors

Disinformation

Influence operations

Harmful content

Privacy

Cybersecurity

Risk mitigation tools with RLHF and RAG

1. Input and output moderation with transformers and a rule base

2. Building a knowledge base for ChatGPT and GPT-4  
    Adding keywords

3. Parsing the user requests and accessing the KB

4. Generating ChatGPT content with a dialog function

    Token control

    Moderation

Summary

Questions

References

Further reading

16. Beyond Text: Vision Transformers in the Dawn of Revolutionary AI

From task-agnostic models to multimodal vision transformers

ViT – Vision Transformer

The basic architecture of ViT

[Step 1: Splitting the image into patches](#)  
[Step 2: Building a vocabulary of image patches](#)  
[Step 3: The transformer](#)

[Vision transformers in code](#)  
[A feature extractor simulator](#)  
[The transformer](#)  
[Configuration and shapes](#)

## [CLIP](#)

[The basic architecture of CLIP](#)  
[CLIP in code](#)

## [DALL-E 2 and DALL-E 3](#)

[The basic architecture of DALL-E](#)  
[Getting started with the DALL-E 2 and DALL-E 3 API](#)

[Creating a new image](#)

[Creating a variation of an image](#)

[From research to mainstream AI with DALL-E](#)

## [GPT-4V, DALL-E 3, and divergent semantic association](#)

[Defining divergent semantic association](#)

[Creating an image with ChatGPT Plus with DALL-E](#)

[Implementing the GPT-4V API and experimenting with DAT](#)

[Example 1: A standard image and text](#)

[Example 2: Divergent semantic association, moderate divergence](#)

[Example 3: Divergent semantic association, high divergence](#)

[Summary](#)

[Questions](#)

[References](#)

[Further Reading](#)

## [17. Transcending the Image-Text Boundary with Stable Diffusion](#)

[Transcending image generation boundaries](#)

[Part I: Defining text-to-image with Stable Diffusion](#)

- [1. Text embedding using a transformer encoder](#)
- [2. Random image creation with noise](#)
- [3. Stable Diffusion model downsampling](#)
- [4. Decoder upsampling](#)
- [5. Output image](#)

[Running the Keras Stable Diffusion implementation](#)

[Part II: Running text-to-image with Stable Diffusion](#)

[Generative AI Stable Diffusion for a Divergent Association Task \(DAT\)](#)

[Part III: Video](#)

[Text-to-video with Stability AI animation](#)

[Text-to-video, with a variation of OpenAI CLIP](#)

[A video-to-text model with TimeSformer](#)

[Preparing the video frames](#)

[Putting the TimeSformer to work to make predictions on the video frames](#)

[Summary](#)

[Questions](#)

[References](#)

[Further reading](#)

18. [Hugging Face AutoTrain: Training Vision Models without Coding](#)

[Goal and scope of this chapter](#)

[Getting started](#)

[Uploading the dataset](#)

[No coding?](#)

[Training models with AutoTrain](#)

[Deploying a model](#)

[Running our models for inference](#)

[Retrieving validation images](#)

[The program will now attempt to classify the validation images. We will see how a vision transformer reacts to this image.](#)

[Inference: image classification](#)

[Validation experimentation on the trained models](#)

[ViTForImageClassification](#)

[SwinForImageClassification 1](#)

[BeitForImage Classification](#)

[SwinForImageClassification 2](#)

[ConvNextForImageClassification](#)

[ResNetForImageClassification](#)

[Trying the top ViT model with a corpus](#)

[Summary](#)

[Questions](#)

[References](#)

[Further reading](#)

## 19. [On the Road to Functional AGI with HuggingGPT and its Peers](#)

[Defining F-AGI](#)

[Installing and importing](#)

[Validation set](#)

[Level 1 image: easy](#)

[Level 2 image: difficult](#)

[Level 3 image: very difficult](#)

[HuggingGPT](#)

[Level 1: Easy](#)

[Level 2: Difficult](#)

[Level 3: Very difficult](#)

[CustomGPT](#)

[Google Cloud Vision](#)

[Level 1: Easy](#)

[Level 2: Difficult](#)

[Level 3: Very difficult](#)

[Model chaining: Chaining Google Cloud Vision to ChatGPT](#)

[Model Chaining with Runway Gen-2](#)

[Midjourney: Imagine a ship in the galaxy](#)

[Gen-2: Make this ship sail the sea](#)

[Summary](#)

[Questions](#)

[References](#)

[Further Reading](#)

## 20. [Beyond Human-Designed Prompts with Generative Ideation](#)

[Part I: Defining generative ideation](#)

[Automated ideation architecture](#)

[Scope and limitations](#)

[Part II: Automating prompt design for generative image design](#)

[ChatGPT/GPT-4 HTML presentation](#)

[ChatGPT with GPT-4 provides the text for the presentation](#)

[ChatGPT with GPT-4 provides a graph in HTML to illustrate the presentation](#)

[Llama 2](#)

[A brief introduction to Llama 2](#)

[Implementing Llama 2 with Hugging Face Midjourney](#)

[Discord API for Midjourney](#)

[Microsoft Designer](#)

[Part III: Automated generative ideation with Stable Diffusion](#)

[1. No prompt: Automated instruction for GPT-4](#)

[2. Generative AI \(prompt generation\) using ChatGPT with GPT-4](#)

[3. and 4. Generative AI with Stable Diffusion and displaying images](#)

[The future is yours!](#)

[The future of development through VR-AI](#)

[The groundbreaking shift: Parallelization of development through the fusion of VR and AI](#)

[Opportunities and risks](#)

[Summary](#)

[Questions](#)

[References](#)

[Further reading](#)

[Appendix: Answers to the Questions](#)

[Other Books You May Enjoy](#)

[Index](#)

# Preface

Transformer-driven Generative AI models are a game-changer for **Natural Language Processing (NLP)** and computer vision. Large Language Generative AI transformer models have achieved superhuman performance through services such as ChatGPT with GPT-4V for text, image, data science, and hundreds of domains. We have gone from primitive Generative AI to superhuman AI performance in just a few years!

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. The expansion from the early **Large Language Models (LLMs)** to multimodal (text, image, sound) algorithms has taken AI into a new era.

For the past few years, we have been 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 more domains. This digital activity is now increasingly driven by transformer copilots in hundreds of applications.

The transformer architecture began just a few years ago as revolutionary and disruptive. It broke with the past, leaving the dominance of RNNs and CNNs behind. BERT and GPT models abandoned recurrent network layers and replaced them with self-attention. But in 2023, OpenAI GPT-4

proposed AI into new realms with GPT-4V (vision transformer), which is paving the path for functional (everyday tasks) AGI. Google Vertex AI offered similar technology. 2024 is not a new year in AI; it's a new decade! Meta (formerly Facebook) has released Llama 2, which we can deploy seamlessly on Hugging Face.

Transformer encoders and decoders contain attention heads that train separately, parallelizing cutting-edge hardware. Attention heads can run on separate GPUs, opening the door to billion-parameter models and soon-to-come trillion-parameter models.

The increasing amount of data requires training AI models at scale. As such, transformers pave the way to a new era of parameter-driven AI. Learning to understand how hundreds of millions of words and images fit together requires a tremendous amount of parameters. Transformer models such as Google Vertex AI PaLM 2 and OpenAI GPT-4V have taken emergence to another level. Transformers can perform hundreds of NLP tasks they were not trained for.

Transformers can also learn image classification and reconstruction by embedding images as sequences of words. This book will introduce you to cutting-edge computer vision transformers such as **Vision Transformers (ViTs)**, CLIP, GPT-4V, DALL-E 3, and Stable Diffusion.

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

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

Imagine 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 that would be required to replace AI image captioning for the billions of images that continuously appear online.

This book will take you from developing code to prompt engineering, a new “programming” skill that controls the behavior of a transformer model.

Each chapter will take you through the key aspects of language understanding and computer vision from scratch in Python, PyTorch, and TensorFlow.

You will learn the architecture of the Original Transformer, Google BERT, GPT-4, PaLM 2, T5, ViT, Stable Diffusion, and several other models. You will fine-tune transformers, train models from scratch, and learn to use powerful APIs.

You will keep close to the market and its demand for language understanding in many fields, such as media, social media, and research papers, for example. You will learn how to improve Generative AI models with **Retrieval Augmented Generation (RAG)**, embedding-based searches, prompt engineering, and automated ideation with AI-generated prompts.

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

You will learn the skills required not only to adapt to the present market but also to acquire the vision to face innovative projects and AI evolutions. This

book aims to give readers both the knowledge and the vision to select the right models and environment for any given project.

# 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 translation, speech-to-text, text-to-speech, language modeling, question answering, and many more NLP domains, as well as computer vision multimodal tasks.

Readers who can benefit the most from this book are:

- Deep learning, vision, and NLP practitioners familiar with Python programming.
- Data analysts, data scientists, and machine learning/AI engineers who want to understand how to process and interrogate the increasing amounts of language-driven and image data.

# **What this book covers**

# Part I: The Foundations of Transformers

*Chapter 1, What Are Transformers?*, explains, at a high level, what transformers and Foundation Models are. We will first unveil the incredible power of the deceptively simple O(1) time complexity of transformer models that changed everything. We will continue to discover how a hardly known transformer algorithm in 2017 rose to dominate so many domains and brought us Foundation Models.

*Chapter 2, Getting Started with the Architecture of the Transformer Model*, 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 Original 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 multi-attention head sublayers work.

*Chapter 3, Emergent vs Downstream Tasks: The Unseen Depths of Transformers*, bridges the gap between the functional and mathematical architecture of transformers by introducing *emergence*. We will then see how to measure the performance of transformers before exploring several downstream tasks, such as the **Standard Sentiment TreeBank (SST-2)**, linguistic acceptability, and Winograd schemas.

*Chapter 4, Advancements in Translations with Google Trax, Google Translate, and Gemini*, goes through machine translation in three steps. We will first define what machine translation is. We will then preprocess a

**Workshop on Machine Translation (WMT)** dataset. Finally, we will see how to implement machine translations.

*Chapter 5, Diving into Fine-Tuning through BERT*, 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 6, Pretraining a Transformer from Scratch through RoBERTa*, 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. Finally, we will put the knowledge acquired in this chapter to work and pretrain a Generative AI customer support model on X (formerly Twitter) data.

## Part II: The Rise of Suprahuman NLP

*Chapter 7, The Generative AI Revolution with ChatGPT*, goes through the tremendous improvements and diffusion of ChatGPT models into the everyday lives of developers and end-users. We will first examine the architecture of OpenAI's GPT models before working with the GPT-4 API and its hyperparameters to implement several NLP examples. Finally, we will learn how to obtain better results with **Retrieval Augmented Generation (RAG)**. We will implement an example of automated RAG with GPT-4.

*Chapter 8, Fine-Tuning OpenAI GPT Models*, explores fine-tuning to make sense of the choices we can make for a project to go in this direction or not. We will introduce risk management perspectives. We will prepare a dataset and fine-tune a cost-effective babbage-02 model for a completion task.

*Chapter 9, Shattering the Black Box with Interpretable Tools*, lifts the lid on the black box that is transformer models by visualizing their activity. We will use BertViz to visualize attention heads, **Language Interpretability Tool (LIT)** to carry out a **Principal Component Analysis (PCA)**, and LIME to visualize transformers via dictionary learning. OpenAI LLMs will take us deeper and visualize the activity of a neuron in a transformer with an interactive interface. This approach opens the door to GPT-4 explaining a transformer, for example.

*Chapter 10, Investigating the Role of Tokenizers in Shaping Transformer Models*, introduces 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. We will explore word and subword tokenizers and show how a tokenizer can shape a transformer

model's training and performance. Finally, we will build a function to display and control token-ID mappings.

*Chapter 11, Leveraging LLM Embeddings as an Alternative to Fine-Tuning*, explains why searching with embeddings can sometimes be a very effective alternative to fine-tuning. We will go through the advantages and limits of this approach. We will go through the fundamentals of text embeddings. We will build a program that reads a file, tokenizes it, and embeds it with Gensim and Word2Vec. We will implement a question-answering program on sports events and use OpenAI Ada to embed Amazon fine food reviews. By the end of the chapter, we will have taken a system from prompt design to advanced prompt engineering using embeddings for RAG.

*Chapter 12, Toward Syntax-Free Semantic Role Labeling with ChatGPT and GPT-4*, goes through the revolutionary concepts of syntax-free, nonrepetitive stochastic models. We will use ChatGPT Plus with GPT-4 to run easy to complex **Semantic Role Labeling (SRL)** samples. We will see how a general-purpose, emergent model reacts to our SRL requests. We will progressively push the transformer model to the limits of SRL.

*Chapter 13, Summarization with T5 and ChatGPT*, goes through the concepts and architecture of the T5 transformer model. We will then apply T5 to summarize documents with Hugging Face models. The examples in this chapter will be legal and medical to explore domain-specific summarization beyond simple texts. We are not looking for an easy way to implement NLP but preparing ourselves for the reality of real-life projects. We will then compare T5 and ChatGPT approaches to summarization.

*Chapter 14, Exploring Cutting-Edge LLMs with Vertex AI and PaLM 2*, examines Pathways to understand PaLM. We will continue and look at the main features of **PaLM (Pathways Language Model)**, a decoder-only,

densely activated, and autoregressive transformer model with 540 billion parameters trained on Google’s Pathways system. We will see how Google PaLM 2 can perform a chat task, a discriminative task (such as classification), a completion task (also known as a generative task), and more. We will implement the Vertex AI PaLM 2 API for several NLP tasks, including question-answering and summarization. Finally, we will go through Google Cloud’s fine-tuning process.

*Chapter 15, Guarding the Giants: Mitigating Risks in Large Language Models*, examines the risks of LLMs, risk management, and risk mitigation tools. The chapter explains hallucinations, memorization, risky emergent behavior, disinformation, influence operations, harmful content, adversarial attacks (“jailbreaks”), privacy, cybersecurity, overreliance, and memorization. We will then go through some risk mitigation tools through advanced prompt engineering, such as implementing a moderation model, a knowledge base, keyword parsing, prompt pilots, post-processing moderation, and embeddings.

# **Part III: Generative Computer Vision: A New Way to See the World**

*Chapter 16, Beyond Text: Vision Transformers in the Dawn of Revolutionary AI*, explores the innovative transformer models that respect the basic structure of the Original Transformer but make some significant changes. We will discover powerful computer vision transformers like ViT, CLIP, DALL-E, and GPT-4V. We will implement vision transformers in code, including GPT-4V, and expand the text-image interactions of DALL-3 to divergent semantic association. We will take OpenAI models into the nascent world of highly divergent semantic association creativity.

*Chapter 17, Transcending the Image-Text Boundary with Stable Diffusion*, delves into diffusion models, introducing Stable Vision, which has created a disruptive generative image AI wave rippling through the market. We will then dive into the principles, math, and code of the remarkable Keras Stable Diffusion model. We will go through each of the main components of a Stable Diffusion model and peek into the source code provided by Keras and run the model. We will run a text-to-video synthesis model with Hugging Face and a video-to-text task with Meta's TimeSformer.

*Chapter 18, Hugging Face AutoTrain: Training Vision Models without Coding*, explores how to train a vision transformer using Hugging Face's AutoTrain. We will go through the automated training process and discover the unpredictable problems that show why even automated ML requires human AI expertise. The goal of this chapter is also to show how to probe the limits of a computer vision model, no matter how sophisticated it is.

*Chapter 19, On the Road to Functional AGI with HuggingGPT and its Peers*, shows how we can use cross-platform chained models to solve difficult image classification problems. We will put HuggingGPT and Google Cloud Vision to work to identify easy, difficult, and very difficult images. We will go beyond classical pipelines and explore how to chain heterogeneous competing models.

*Chapter 20, Beyond Human-Designed Prompts with Generative Ideation*, explores generative ideation, an ecosystem that automates the production of an idea to text and image content. The development phase requires highly skilled human AI experts. For an end user, the ecosystem is a click-and-run experience. By the end of this chapter, we will be able to deliver ethical, exciting, generative ideation to companies with no marketing resources. We will be able to expand generative ideation to any field in an exciting, cutting-edge, yet ethical ecosystem.

## To get the most out of this book

Most of the programs in the book are Jupyter 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.

Take the time to read *Chapter 2, Getting Started with the Architecture of the Transformer Model*. Chapter 2 contains the description of the Original Transformer. If you find it difficult, then pick up the general intuitive ideas from the chapter. You can then go back to that 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/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition>. We also have other code bundles from our rich catalog of books and videos available at  
<https://github.com/PacktPublishing/>. Check them out!

# Download the color images

We also provide a PDF file that contains color images of the screenshots/diagrams used in this book. You can download it here:  
<https://packt.link/gbp/9781805128724>.

# Conventions used

There are several text conventions used throughout this book.

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

A block of code is set as follows:

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

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

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

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

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

**Bold**: Indicates a new term, an important word, or words that you see on the screen.

For instance, words 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](mailto: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](mailto: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](mailto:copyright@packtpub.com) with a link to the material.

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

## Share your thoughts

Once you've read *Transformers for Natural Language Processing and Computer Vision - Third Edition*, we'd love to hear your thoughts! Please [click here to go straight to the Amazon review](#) page for this book and share your feedback.

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

# Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781805128724>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

# 1

## What Are Transformers?

Transformers are industrialized, homogenized **Large Language Models (LLMs)** designed for parallel computing. A transformer model can carry out a wide range of tasks with no fine-tuning. Transformers can perform self-supervised learning on billions of records of raw unlabeled data with billions of parameters. From these billion-parameter models emerged multimodal architectures that can process text, images, audio, and videos.

ChatGPT popularized the usage of transformer architectures that have become general-purpose technologies like printing, electricity, and computers.

Applications are burgeoning everywhere! Google Cloud AI, **Amazon Web Services (AWS)**, Microsoft Azure, OpenAI, Google Workspace, Microsoft 365, Google Colab Copilot, GitHub Copilot, Hugging Face, Meta, and myriad other offers are emerging.

The functionality of transformer models has pervaded every aspect of our workspaces with Generative AI for text, Generative AI for images, discriminative AI, task specific-models, unsupervised learning, supervised learning, prompt design, prompt engineering, text-to-code, code-to-text, and more. Sometimes, a GPT-like model will encompass all these concepts!

The societal impact is tremendous. Developing an application has become an educational exercise in many cases. A project manager can now go to

OpenAI's cloud platform, sign up, obtain an API key, and get to work in a few minutes. Users can then enter a text, specify the NLP task as Google Workspace or Microsoft 365, and obtain a response created by a Google Vertex AI or a ChatGPT transformer model. Finally, users can go to Google's Gen App Builder and build applications without programming or machine learning knowledge.

The numbers are dizzying. *Bommasani et al.* (2023) created a Foundation Model ecosystem that lists 128 Foundation Models 70 applications, and 64 datasets. The paper also mentions Hugging Face's 150,000+ models and 20,000+ datasets! The list is growing weekly and will spread to every activity in society.

Where does that leave an AI professional or someone wanting to be one?

Should a project manager choose to work locally? Or should the implementation be done directly on Google Cloud, Microsoft Azure, or AWS? Should a development team select Hugging Face, Google Trax, OpenAI, or AllenNLP? Should an AI professional use an API with practically no AI development? Should an end-user build a no-code AI application with no ML knowledge with Google's Gen App Builder?

The answer is yes to *all* of the above! You do not know what a future employer, customer, or user may want or specify. Therefore, you must be ready to adapt to any need that comes up at the dataset, model, and application levels. This book does not describe all the offers that exist on the market. You cannot learn every single model and platform on the market. If you try to learn everything, you'll remember nothing. You need to know where to start and when to stop. By the book's end, you will have acquired enough critical knowledge to adapt to this ever-moving market.

In this chapter, we will first unveil the incredible power of the deceptively simple O(1) time complexity of transformer models that changed everything. We will build a notebook in Python, PyTorch, and TensorFlow to see how transformers hijacked hardware accelerators. We will then discover how one token (a minimal part of a word) led to the AI revolution we are experiencing.

We will continue to discover how a hardly known transformer algorithm in 2017 rose to dominate so many domains. We had to find a new name for it: the Foundation Model. Foundation Models can do nearly everything in AI! We will sit back and watch how ChatGPT explains, analyzes, writes a classification program in a Python notebook, and displays a decision tree.

Finally, this chapter introduces the role of an AI professional in the ever-changing job market. We will begin to tackle the problem of choosing the right resources.

We must address these critical notions before starting our exploratory journey through the variety of transformer model implementations described in this book.

This chapter covers the following topics:

- How one O(1) invention changed the course of AI history
- How transformer models hijacked hardware accelerators
- How one token overthrew hundreds of AI applications
- The multiple facets of a transformer model
- Generative AI versus discriminative AI
- Unsupervised and self-supervised learning versus supervised learning
- General-purpose models versus task-specific models
- How ChatGPT has changed the meaning of automation

- Watch ChatGPT create and document a classification program
- The role of AI professionals
- Seamless transformer APIs
- Choosing a transformer model



With all the innovations and library updates in this cutting-edge field, packages and models change regularly. Please go to the GitHub repository for the latest installation and code examples:

<https://github.com/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition/tree/main/Chapter01>.

You can also post a message in our Discord community (<https://www.packt.link/Transformers>) if you have any trouble running the code in this or any chapter.

Our first step will be to explore the seeds of the disruptive nature of transformers.

## How constant time complexity O(1) changed our lives forever

How could this deceptively simple O(1) time complexity class forever change AI and our everyday lives? How could O(1) explain the profound architectural changes that made ChatGPT so powerful and stunned the world? How can something as simple as O(1) allow systems like ChatGPT to spread to every domain and hundreds of tasks?

The answer to these questions is the only way to find your way in the growing maze of transformer datasets, models, and applications is to *focus* on the underlying concepts of thousands of assets. Those concepts will take you to the core of the functionality you need for your projects.

This section will provide a significant answer to those questions before we move on to see how one token (a minimal piece of a word) started an AI revolution that is raging around the world, triggering automation never seen before.

We need to get to the bottom of the chaos and disruption generated by transformers.

To achieve that goal, in this section, we will use science and technology to understand how all of this started. First, we will examine O(1) and then the complexity of a layer through a Python and PyTorch notebook.

Let's first get the core concepts and terminology straight for O(1).

## O(1) attention conquers O(n) recurrent methods

O(1) is a “Big O” notation. “Big O” means “order of.” In this case, O(1) represents a constant time complexity. We can say O(1) and order of 1 operation.

Believe it or not, you’re at the heart of the revolution!

In *Chapter 2, Getting Started with the Architecture of the Transformer Model*, we will explore the architecture of the transformer model.

In this section and chapter, we will first focus on what led to the industrialization of AI through transformers and the ChatGPT frenzy: *the*

*exponential increase of hardware efficiency with self-attention.* We will discover how attention leverages hardware and opens the door to incredible new ways of machine learning.

The following sentence contains 11 words:

Jay likes oranges in the morning but not in the evening.

The length of the sentence is  $n = 11$ .

The problem of language understanding can be summed up in one word: context. A word can rarely be defined without context. The meaning of a word can change in each context beyond its definition in a dictionary.

Let's begin with a conceptual approach of an attention layer.

## Attention layer

If we look at what Jay likes, the dimensions (or parameters) of the word “oranges” contain several relationships:

Dimension 1: The association between oranges and evening

Dimension 2: The association between oranges and morning

Dimension 3: The association between Jay and oranges

Dimension 4: The association between Jay and morning

...

Dimension z

Notice that the relationships are defined pairwise: one word to one word. This is how *self-attention* works in a transformer.

If we represent this with Big O notation, we get  $O(1)$ . “O” means “in the order of.” For example,  $O(1)$  is an “order of 1” or constant time complexity

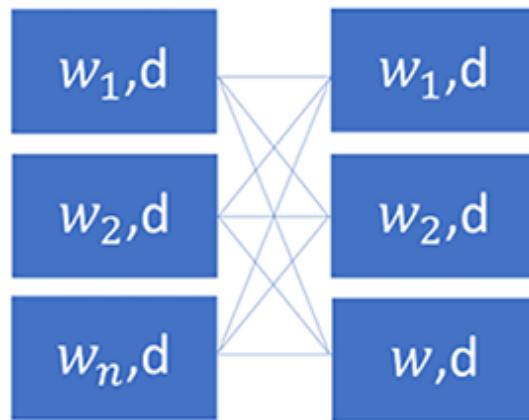
class.

We perform one operation per word,  $O(1)$ , for each word to find the relationship with another word in a pairwise analysis.

Translated into numbers:

- $n$  = the length of the sequence, which is 11 words in this case.
- $d$  = the number of dimensions expressed in floats. In machine learning, the dimensions are expressed in floats. For example, if  $x$  is a word, the values might be: [-0.2333, 03.8559, 0.9844...0394]. The model will learn these values through billions of text data points.

This pairwise relationship is an  $n * n$  computation, as shown in *Figure 1.1*:



*Figure 1.1: Word-to-word relationships*

$O(1)$  represents the memory complexity in this case.

The computational complexity of an attention layer is thus  $O(n^2 * d)$ .  $n^2$  is the pairwise (word-to-word) operation of the whole sequence  $n$ .  $d$  represents the dimensions the model is learning.

Let's see the difference with a recurrent layer.

## Recurrent layer

Recurrent layers do not function that way. They are  $O(n)$ , an order of  $n$  linear time complexity. The longer the sequence, the more memory they will consume. Why? They do not learn the dimensions with pairwise relationships. They learn in a sequence. For example:

Dimension a: Jay

Dimension b: likes and Jay

Dimension c: oranges and likes and Jay

Dimension d: in and oranges and likes and Jay

...

Dimension z

You can see that each word does not simply look at another word but several other words simultaneously!

The number of dimensions  $d$  one word is multiplied by the dimensions of a preceding word leading to  $d^2$  to learn the dimensions. The computational time complexity of a recurrent layer is thus:

$$O(n*d^2)$$

Let us now see the magic.

## The magic of the computational time complexity of an attention layer

An attention layer benefits from its  $O(1)$  memory time complexity. This enables the computational time complexity of  $O(n^2*d)$  to perform a dot product between each word. In machine learning, this transcribes into

multiplying the representation  $d$  of each word by another word. An attention layer can thus learn all the relationships in one matrix multiplication!

A recurrent layer of a computational time complexity of  $O(n*d^2)$  is hindered by its  $O(n)$  linear, sequential process. Performing the same task as the attention layer will take more operations. We will now simulate the attention time complexity model and the recurrent time complexity model in a Python, PyTorch, and TensorFlow notebook.

The calculations are conceptual. We will run simulations with a CPU, a GPU, and a TPU in this section:

- **CPU: Central Processing Unit.** This is the primary component of a computer.
- **GPU: Graphics Processing Unit.** A GPU is a specialized processing unit. Initially used for 3D image rendering, GPUs have evolved to perform machine learning tasks, such as matrix multiplications.
- **TPU: Tensor Processing Unit.** A TPU is an accelerating machine learning workload processor developed by Google and optimized for TensorFlow.

Open `0-1_and_Accelerators.ipynb` in the chapter directory of the repository.

We will begin with CPUs.

## Computational time complexity with a CPU

A CPU is a general-purpose software processor. It is not specially designed for matrix multiplications. It can perform complex operations but only to a certain degree of efficiency.

Before running the first cell of the notebook, we must verify that we are using a CPU by going to the Runtime menu, selecting **Change runtime type**, and making sure that the **Hardware accelerator** parameter is set **CPU**:

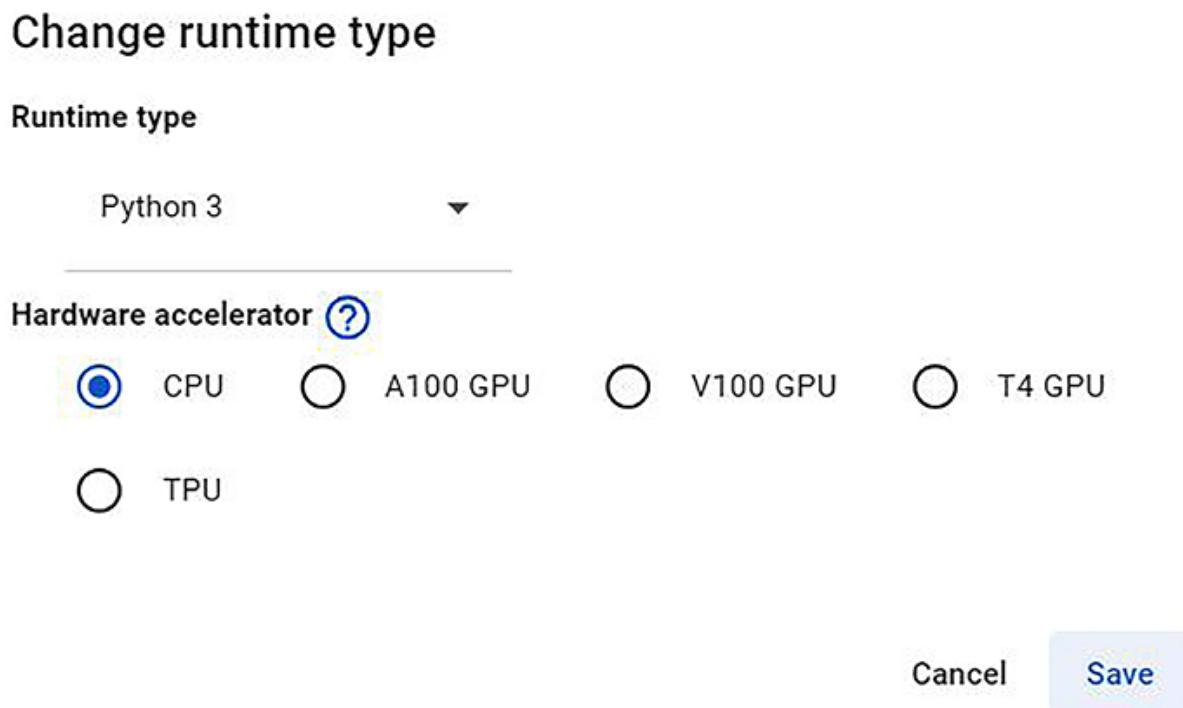


Figure 1.2: Selecting a Runtime type

In this case, a hardware accelerator is a GPU or TPU that performs specific computing tasks (matrix multiplication, for example) more efficiently than a general-purpose CPU, which is not a hardware accelerator. This explains why the **Hardware accelerator** is set to **None**.

The notebook begins with a figure containing the time complexities we have just gone through:

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	Recurrent	$O(n \cdot d^2)$	$O(n)$

Figure 1.3: Time complexities from Attention is All You Need, Vaswani et al. (2017), page 6

The goal of the notebook is to represent the complexity per layer of self-attention and recurrent layers, not the actual algorithms. The self-attention computational time complexity will run with matrix multiplications.

The recurrent computational time will use a loop simulating its sequential approach. A recurrent model uses matrices to compute values; however, its overall computational complexity is an order of  $n$  and  $d$ . It is important to note that the calculations do not reflect the actual algorithms inside self-attention layers and recurrent layers. Also, many other factors will influence performance: hardware, data, hyperparameters, training time, and other parameters.

The goal of this notebook remains to show the overall computational time complexity. In this case, the simulations are sufficient. We first define the framework of the evaluations:

```
# Computational times of complexity per layer
# Comparing the computational time between:
# self attention = O(n^2 * d)
#and
# recurrent = O(n * d^2)
```

We then import `numpy` and `time`:

```
import numpy as np
import time
```

We first define the sequence length (number of words in a sequence) and the dimensions (numerical vector representing features of the word):

```
# define the sequence length and representation dimensionality
n = 512
d = 512
```

You will notice that  $n = d$ , which means that  $O(n^2 * d) = O(n * d^2) = 512 * 512 * 512 = 134,217,728$  operations. In this case, both the attention and the recurrent layers have the same number of operations to perform from a computational complexity time perspective. We will define the sequence of input with random values for the dimensions ( $d$ ):

```
# define the inputs
```

We will first simulate the time complexity of the self-attention layer with a matrix multiplication with start time:

```
# simulation of self-attention layer O(n^2*d)
start_time = time.time()
for i in range(n):
    for j in range(n):
        _ = np.dot(input_seq[i], input_seq[j])
```

When the matrix multiplication is finished, we calculate the time it took and print it:

```
at=time.time()-start_time
print(f"Self-attention computation time: {time.time() - start_
```

The time is displayed:

```
Self-attention computation time: 0.7938594818115234 seconds
```

We now run the simulation of the time complexity of the recurrent layer function without a matrix multiplication with a start and end time:

```
# simulation of recurrent layer O(n*d^2)
start_time = time.time()
hidden_state = np.zeros(d)
for i in range(n):
    for j in range(d):
        for k in range(d):
            hidden_state[j] += input_seq[i, j] * hidden_state[rt=time.time()-start_time
print(f"Recurrent layer computation time: {time.time() - start_
```

The output shows the time it took:

```
Recurrent layer computation time: 109.65185356140137 seconds
```

We now calculate the percentage of the attention layer's computational time complexity and the recurrent layer's computational time complexity. We can measure the performance percentages. This approach provides an overall idea of the power of attention layers. We then display the result:

```
# Calculate the total  
# Calculate the percentage of at  
percentage_at = round((at / total) * 100,2)  
# Output the result  
print(f"The percentage of 'computational time for attention' i
```

The output shows that the attention layer's computational time complexity is more efficient:

```
The percentage of 'computational time for attention' in the sum
```

The attention layer's computational time complexity performs better than the recurrent layer on a CPU in this configuration.

Let's move on to GPUs.

## Computational time complexity with a GPU

Before running the first cell of the notebook, we must verify that we are using a GPU by going to the Runtime menu, selecting **Change runtime type**, and making sure that the **Hardware accelerator** parameter is set to one of the **GPUs**:

## Change runtime type

### Runtime type

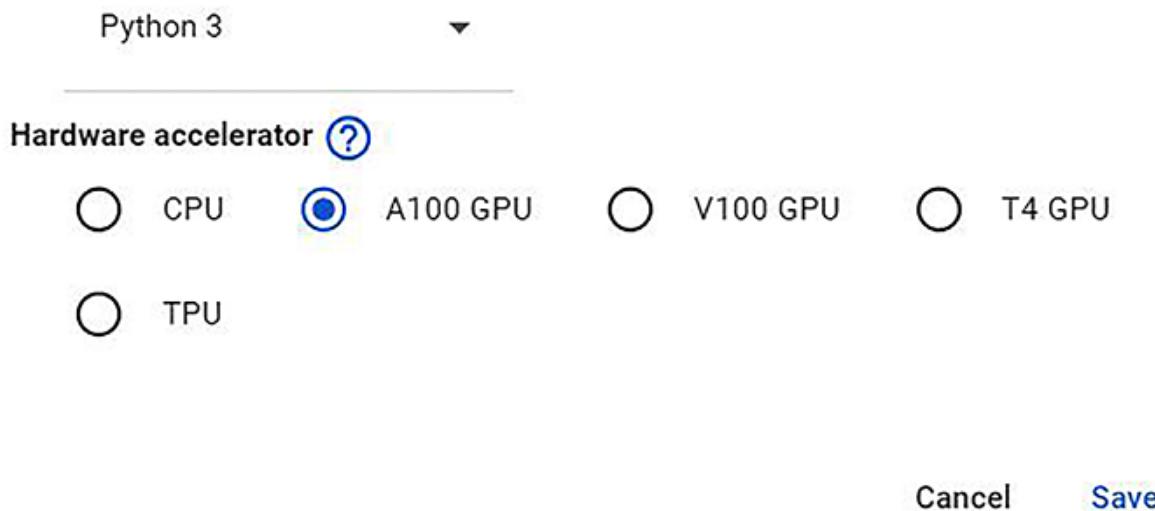


Figure 1.4: Changing the settings to use a GPU

The GPU type is, in this case, an NVIDIA V100. GPUs work well with algorithms requiring massive operations, particularly matrix multiplications, such as the attention layer mechanism's computational time complexity we are simulating in this notebook.

We will implement our simulation in PyTorch to leverage the power of the GPUs:

```
# PyTorch version
import torch
import time
```

We will use the same parameters as for the CPU evaluation:

```
# define the sequence length and representation dimensionality
n = 512
d = 512
```

We now activate the GPU and define the inputs:

```
# Use GPU if available, otherwise stick with cpu
device = torch.device("cuda" if torch.cuda.is_available() else
print(device)
# define the inputs
input_seq = torch.rand(n, d, device=device)
```

We run the same simulation as for the CPU but on the GPU:

```
# simulation of self-attention layer O(n^2*d)
start_time = time.time()
_ = torch.mm(input_seq, input_seq.t())
at = time.time() - start_time
print(f"Self-attention computation time: {at} seconds")
```

The output is not that impressive because we didn't run a massive number of matrix multiplications to take full advantage of the GPU, which reveals its power on larger volumes:

```
cuda
Self-attention computation time: 2.887202501296997 seconds
```

We will now run the recurrent layer function but limit its time to about 10 times (depending on GPU activity) the time it took for self-attention, which

is enough to show our point with `if ct>at*10:`

```
# simulation of recurrent layer O(n*d^2)
start_time = time.time()
hidden_state = torch.zeros(d, device=device)
for i in range(n):
    for j in range(d):
        for k in range(d):
            hidden_state[j] += input_seq[i, j] * hidden_state[k]
            ct = time.time() - start_time
            if ct>at*10:
                break
```

We compute the limited time we gave the function to run and display it:

```
rt = time.time() - start_time
print(f'Recurrent layer computation time: {rt} seconds')
```

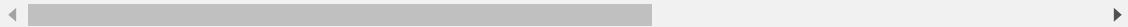
The output time isn't very efficient:

```
Recurrent layer computation time: 36.3216814994812 seconds
```

We calculate the percentage of attention layer in the total time:

```
# Calculate the total
total = at + rt
# Calculate the percentage of at
percentage_at = round((at / total) * 100, 2)
# Output the result
print(f'The percentage of self-attention computation in the su
```

The percentage of self-attention computation in the sum of self



We can check the information on the GPU:

```
!nvidia-smi
```

The output will display the information on the current GPU:

```
+-----+
| NVIDIA-SMI 525.105.17    Driver Version: 525.105.17    CUDA Version: 12.0 |
+-----+-----+-----+
| GPU  Name      Persistence-M| Bus-Id      Disp.A  | Volatile Uncorr. ECC | | |
| Fan  Temp     Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |
|          |             |              |             |          MIG M. |
+-----+-----+-----+-----+
| 0  Tesla V100-SXM2... Off  | 00000000:00:04.0 Off |          0 | | |
| N/A   39C     P0    40W / 300W | 2160MiB / 16384MiB |     0%      Default |
|          |             |              |             |          N/A |
+-----+-----+-----+
+-----+
| Processes:
| GPU  GI  CI          PID  Type  Process name                  GPU Memory |
|          ID  ID          ID   ID          Usage
+-----+
```

Figure 1.5: GPU information

We will finally explore a TPU simulation.

## Computational time complexity with a TPU

Google designed Cloud TPUs specifically for matrix calculations and neural networks. Since the primary task of a TPU is matrix multiplication, it doesn't make much sense to run sequential operations. We will limit the recurrent layer to 10 times the time it takes for the attention layer.

Before running the first cell of the notebook, we must verify that we are using a TPU by going to the Runtime menu, selecting **Change runtime type**, and making sure that the **Hardware accelerator** parameter is set to **TPU**:

## Change runtime type

### Runtime type

Python 3

### Hardware accelerator [?](#)

- CPU
- A100 GPU
- V100 GPU
- T4 GPU
- TPU

[Cancel](#) [Save](#)

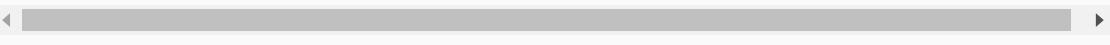
Figure 1.6: Changing the notebook settings to use a TPU

For this simulation, we will use TensorFlow to take full advantage of the TPU:

```
import tensorflow as tf
import numpy as np
import time
```

The program is the same one as for the CPU and GPU evaluation, except that this time, it's running on a TPU with TensorFlow:

```
# define the sequence length and representation dimensionality
n = 512
d = 512
# define the inputs
input_seq = tf.random.normal((n, d), dtype=tf.float32)
```



We run the matrix multiplication and measure the time it took:

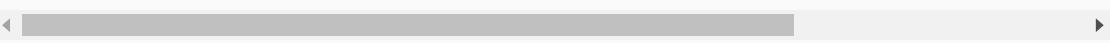
```
# simulation of self-attention layer O(n^2*d)
start_time = time.time()
_ = tf.matmul(input_seq, input_seq, transpose_b=True)
at = time.time() - start_time
print(f"Self-attention computation time: {at} seconds")
```

The output is efficient:

```
Self-attention computation time: 0.10626077651977539 seconds
```

We will now run the recurrent layer:

```
# simulation of recurrent layer O(n*d^2)
start_time = time.time()
hidden_state = np.zeros((n, d), dtype=np.float32)
for i in range(n):
    for j in range(d):
        for k in range(d):
            hidden_state[i, j] += input_seq[i, j].numpy() * hi
            ct = time.time() - start_time
            if ct>at*10:
                break
```



The output is not efficient, which is normal:

```
rt = time.time() - start_time
print(f'Recurrent layer computation time: {rt} seconds')
```

```
Recurrent layer computation time: 66.53181290626526 seconds
```

We now compute the percentage of the total time:

```
# Calculate the total
total = at + rt
# Calculate the percentage of at
percentage_at = round((at / total) * 100, 2)
# Output the result
print(f'The percentage of self-attention computation in the sum of self-attention and recurrent layers is {percentage_at}%')
```

The output confirms that attention layers are more efficient:

```
The percentage of self-attention computation in the sum of self-attention and recurrent layers is 99.9%
```

We will now take the TPU simulation into the world of LLMs.

## TPU-LLM

We will run the same simulation as for the TPU with relatively standard parameters for the attention layer. `d=12288` could be the dimensionality of an LLM. `n=32728` could be the input limit of an LLM. These values are simply an example of running the recurrent layer on the TPU. The values are those of an LLM:

```
import tensorflow as tf
import numpy as np
import time
# define the sequence length and representation dimensionality
n = 32768
d = 12288
# define the inputs
input_seq = tf.random.normal((n, d), dtype=tf.float32)
```

We will run the same function as before and display the time it took:

```
# simulation of self-attention layer O(n^2*d)
start_time = time.time()
_ = tf.matmul(input_seq, input_seq, transpose_b=True)
at = time.time() - start_time
print(f"Self-attention computation time: {at} seconds")
```

The time is very efficient:

```
Self-attention computation time: 23.117244005203247 seconds
```

We can display some information on the TPU:

```
import os
from tensorflow.python.profiler import profiler_client
tpu_profile_service_address = os.environ['COLAB_TPU_ADDR'].rep
print(profiler_client.monitor(tpu_profile_service_address, 100)
```

We can see that the TPU Matrix Units were hardly solicited:

```
Timestamp: 10:20:30
TPU type: TPU v2
Utilization of TPU Matrix Units (higher is better): 0.000%
```

The conclusion of this evaluation can be summed up in the following points:

- The attention layer computational time complexity outperforms the recurrent layer computational time complexity by avoiding recurrence.
- The attention layer's one-to-one word analysis makes it better at detecting long-term dependencies.
- The architecture of the attention layer enables matrix multiplication, which takes full advantage of modern GPUs and TPUs.
- By unleashing the power of GPUs (and TPUs) and attention, algorithms can process more data and learn more information.

The deceptively simple  $O(1)$  that led to  $O(n^2 * d)$  has changed our daily lives forever by taking AI to the next level.

How did this happen?

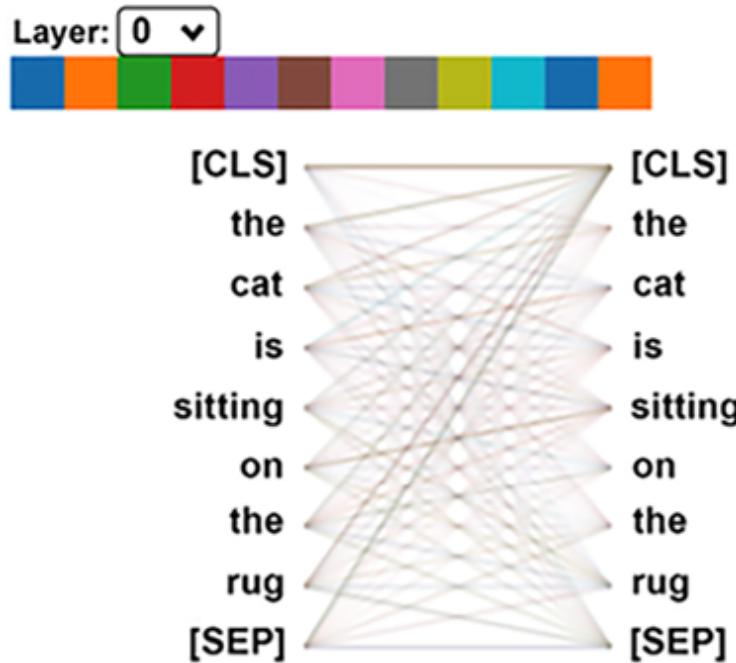
## A brief journey from recurrent to attention

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

This section goes through a brief background of NLP that led to transformers, which we'll describe in more detail in *Chapter 2, Getting*

*Started with the Architecture of the Transformer Model.* First, however, let's have a look at the attention head of a transformer that has replaced the RNN layers of an NLP neural network.

The core concept of a transformer can be summed up loosely as “mixing tokens.” NLP models first convert word sequences into tokens. RNNs analyze tokens in recurrent functions. Transformers do not analyze tokens in sequences but relate every token to the other tokens in a sequence, as shown in *Figure 1.7*:



*Figure 1.7: An attention head of a layer of a transformer*

We will go through the details of an attention head in *Chapter 2*. For the moment, the takeaway of *Figure 1.7* is that for innovative LLMs, *each word (token) of a sequence is related to all the other words of a sequence*.

Let's briefly go through the background of transformers.

## A brief history

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

In this section, I will share some of my favorite researchers, among the many who paved the way to where AI is today, laying the ground for the arrival of the transformer.

In the late 19th and early 20th century, Andrey Markov introduced the concept of random values and created a theory of stochastic processes (see the *Further reading* section). We know them in AI as the **Markov Decision Process (MDP)**, **Markov chains**, and **Markov processes**. In the early 20th century, Markov showed that we could predict the next element of a chain, a sequence, using only the most recent elements of that chain. He applied this to letters and conducted many experiments. double space Bear in mind that he had no computer but proved a theory still in use today in AI.

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

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

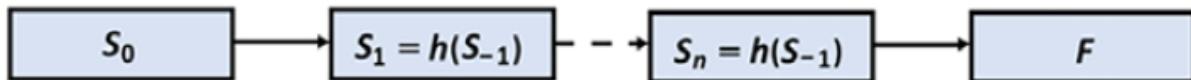
In 1954, the Georgetown-IBM experiment used computers to translate Russian sentences into English using a rule system. A rule system is a program that runs a list of rules that will analyze data structures. Rule

systems still exist and are everywhere. However, in some cases, machine intelligence can replace rule lists for the billions of language combinations by automatically learning the patterns.

The expression *AI* was first used by John McCarthy in 1956 when exploring the possibilities that machines could learn.

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

An RNN memorizes the persistent states of a sequence efficiently, as shown in *Figure 1.8*:



*Figure 1.8: The RNN process*

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

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

efficient architecture faces problems when dealing with long-term dependencies in lengthy and complex sequences.

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

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

Some research was done on sequence-to-sequence models, but they did not meet expectations. It seemed that nothing else could be done to make more progress, and 30 years passed this way. Attention was introduced, but the models still relied on recurrence. And then, starting in late 2017, the industrialized state-of-the-art Transformer came with its attention head sublayers and more. RNNs did not appear as a prerequisite for sequence modeling anymore.

Before diving into the Original Transformer's architecture, which we will do in *Chapter 2, Getting Started with the Architecture of the Transformer Model*, let's start at a high level by examining the paradigm change triggered by one little token.

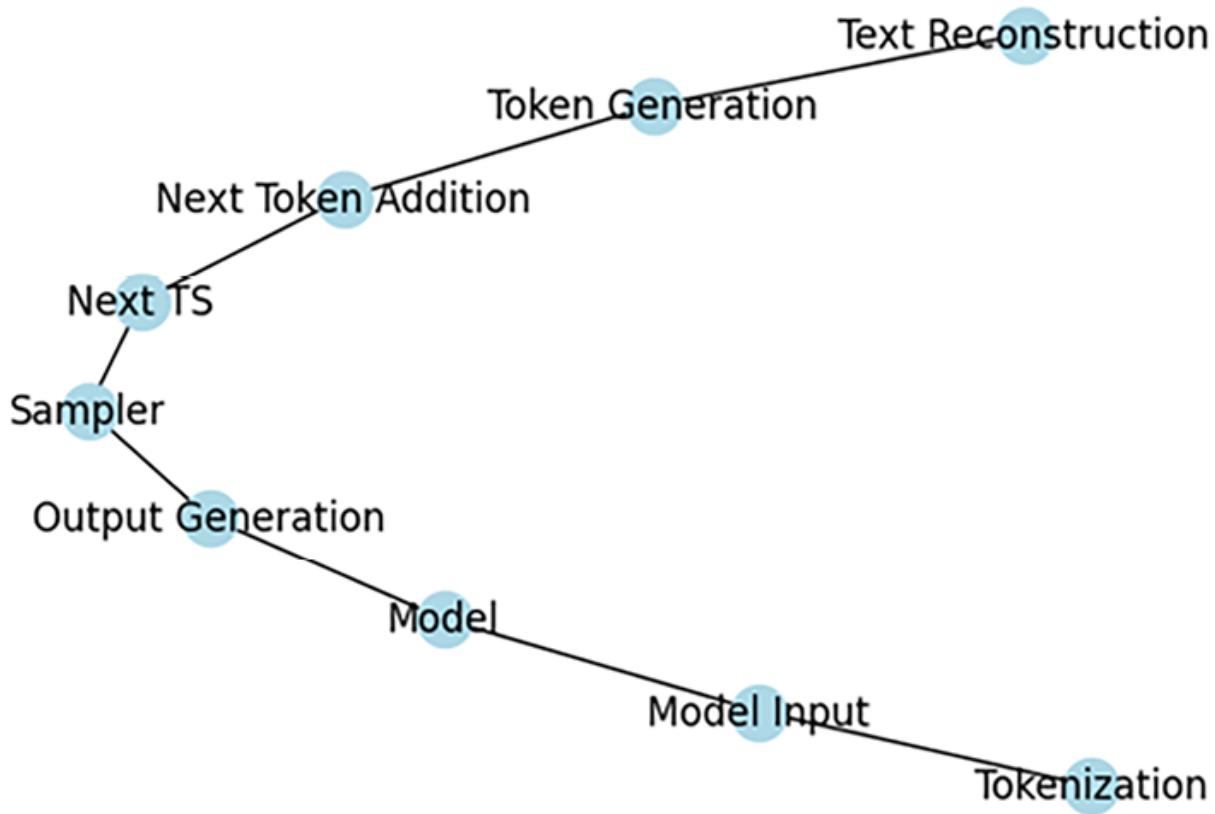
## From one token to an AI revolution

Yes, the title is correct, as you will see in this section. One token produced an AI revolution and has opened the door to AI in every domain and application.

ChatGPT with GPT-4, PaLM 2, and other LLMs have a unique way of producing text.

In LLMs, a token is a minimal word part. *The token is where a Large Language Model starts and ends.*

For example, the word `including` could become: `includ` + `ing`, representing two tokens. GPT models predict tokens based on the hundreds of billions of tokens in its training dataset. Examine the graph in *Figure 1.9* of an OpenAI GPT model that is making an inference to produce a token:



*Figure 1.9: GPT inference graph built in Python with NetworkX*

It may come as a surprise, but the only parts of this figure controlled by the model are `Model` and `output Generation!`, which produce raw logits. All the rest is in the pipeline.

To understand the pipeline, we will first go through the description of these steps:

1. **Tokenization:** The pipeline converts the input sequence “Is the sun in the solar system?”, for example, into tokens using a tokenizer. In *Chapter 10, Investigating the Role of Tokenizers in Shaping Transformer Models*, we will dig into the critical role of tokenizers in transformer models.
2. **Model Input:** The pipeline now passes the tokenized sequence into the trained GPT model.
3. **Model:** The model processes the input through its various layers, from the input layer through multiple transformer layers to the output layer. *Chapter 2, Getting Started with the Architecture of the Transformer Model*, will describe the architecture in detail.
4. **Output Generation:** The model produces raw output logits given the input sequence.
5. **Sampler:** The sampler will convert the logits into probabilities. We will implement hyperparameters that influence the model’s output in *Chapter 7, The Generative AI Revolution with ChatGPT*. For more on the sampling process involving the main hyperparameters, see the *Vertex AI PaLM 2 Interface* section of *Chapter 14, Exploring Cutting-Edge LLMs with Vertex AI and PaLM 2*.
6. **Next Token Selection (Next TS):** The next token is selected based on the probabilities of the sampler.
7. **Next Token Addition:** The selected next token is added to the input sequence (in token form) and repeats the process from step 3 until the maximum token limit has been reached.
8. **Token Generation Completion (Text Generation):** Text generation will end once the maximum token limit has been reached or an end-of-

sequence token has been detected.

9. **Text Reconstruction:** The tokenizer converts the final sequence of tokens back into a string of text. This step includes stitching any subword tokens back together to form whole words.

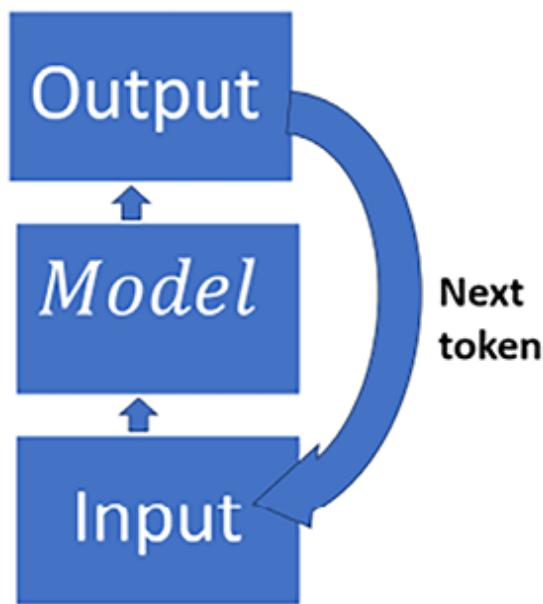
The magic appears again! The model takes an input sequence and produces one token.

That token is added to the sequence, and the model starts over again to produce another token.

We can sum up the steps of this process:

1. Input = an input sequence expressed in tokens.
2. The model processes the input.
3. Output = the next token is added at the end of the input if the maximum tokens requested is reached.

The process is a one-token output revolution, as shown in *Figure 1.10*:



*Figure 1.10: The model processes the input sequence, produces raw outputs, and selects one token*

To sum this up in a notation for one forward pass:

$$t = f(n)$$

Take the necessary time to let this sink in:

- $t$  = one token.
- $f$  = the model and the output controller that infers the next token.
- $n$  = the initial sequence of tokens + each new next token that is added to it until the maximum number of tokens is reached or an end-of-sequence token is detected.

We have summed up the process as  $f(n)$ . Under the hood of  $f(n)$  is the whole process of adding predicted tokens one by one. You can think of  $f(n)$  as:

$$t_i = f(t_1, t_2, \dots, t_{\{i - 1\}})$$

Think about it. One token! And yet that token changed society in every domain: translations, summarization, question-answering, imagery, video... the list is growing daily.

We will now go from one token to everything!

## From one token to everything

In the previous section, we saw that a generative model can be summed up as  $t = f(n)$ . The model processes  $n$  and produces  $t$ , one token at a time.

The consequences are unfathomable and tremendous.

We can see that the one-token approach, summed up as  $t = f(n)$ , results in:

- The **dynamic** nature of a transformer model. It adapts its outputs based on its incremental inputs represented by:

$$t_i = f(t_1, t_2, \dots, t_{\{i-1\}})$$

- The model will adapt and generate outputs on completely new inputs!
- The **implicit** nature of a transformer model. The model encodes and stores relationships between tokens in weights and biases. There is no explicit guideline. It just keeps producing tokens based on its dynamic inputs based on millions of text, image, and audio data!
- The incredible **flexibility** of the system is inherited from its dynamic and implicit properties. A GPT-like transformer model can infer meaningful outputs for a wide range of contextually diverse inputs.

We will now see the implications of the models' one-token approach through different perspectives:

- **Supervised and unsupervised**

Some may say that a GPT series model such as ChatGPT goes through unsupervised training. That statement is only true to a certain extent.

Token by token, a GPT-like model finds its way to accuracy through self-supervised learning, predicting each subsequent token based on the preceding ones in the sequence. It succeeds in doing so through the influence of all the other tokens' representations in a sequence.

We can also fine-tune a GPT model with an input (prompt) and output (completion) with labels! We can provide thousands of inputs (prompts) with one token as an output (completion). For example, we can create thousands of questions as inputs with only `true` and `false` as outputs. This is implicit supervised learning. Also, the model will not explicitly memorize the correct predictions. It will simply learn the patterns of the tokens.

- **Discriminative and generative**

Since the arrival of ChatGPT, the term “Generative AI” has often been used to describe GPT-like models. First, we must note that GPT models were not the first to produce Generative AI recurrent neural networks were also generative.

Also, if we follow the science, saying the GPT-like models perform Generative AI tasks is not entirely true. In the previous *Supervised and unsupervised* bullet point, we mentioned that a “generative” model could perform the supervised task of inferring an output such as “true” or “false” when the input is a statement or a question. This is not a Generative AI task! This is a discriminative AI task.

Another situation can arise with summarizing. Part of summarizing is discriminative to find keywords, topics, and names. Another part can be generative when the system infers new tokens to summarize a text.

Finally, the full power of the autoregressive nature (generating a token and adding it to the input) of language modeling (adding a new token at the end of a sequence) is attained when an LLM invents a story; that is generative AI.

A GPT-like model can be discriminative, Generative, or both depending on the task to perform.

- **Task-specific and specific task models**

Task-specific models are often opposed to general-purpose models, such as GPT-like LLMs. Task-specific models are trained to perform very well on specific tasks. This is accurate to some extent for some tasks. However, LLMs trained to be general-purpose systems can surprisingly perform well on specific tasks, as proven through the

evaluations of various exams (law, medical). Also, transformer models can now perform image recognition and generation.

- **Interaction and automation**

A key point in analyzing the relationship between humans and LLMs is automation. To what degree should we automate our tasks? Some tasks cannot be automated by LLMs. ChatGPT and other assistants cannot make life-and-death, moral, ethical, and business decisions. Asking a system to do that will endanger an organization.

Automation needs to be thoroughly analyzed before implementing LLMs.

In the following chapters of this book, you will encounter supervised, self-supervised, unsupervised, discriminative, generative, task-specific, specific tasks, and interactions with evolved AI and automated systems. Sometimes you will find all of this in the architecture of a general-purpose LLM!

Be prepared and remain flexible!

## Foundation Models

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

Foundation Models were not created by academia but by the big tech industry. Google invented the transformer model, leading to Google BERT,

LaMBDA, PaLM 2, and more. Microsoft partnered with OpenAI to produce ChatGPT with GPT-4, and soon more.

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

Let's consider the evolution of LLMs to understand the need for industrialized AI models.

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

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

- **Model architecture:** The model is industrial. The layers of the model are identical, and they are specifically designed for parallel processing. We will go through the architecture of transformers in *Chapter 2, Getting Started with the Architecture of the Transformer Model*.
- **Data:** Big tech possesses the hugest data source in the history of humanity, generated mainly through human-driven online activities and interactions, including browsing habits, search queries, social media posts, and online purchases.
- **Computing power:** Big tech possesses computer power never seen before at that scale. For example, GPT-3 was trained at about 50 **PetaFLOPS (Floating Point Operations Per Second)**, and Google now has domain-specific supercomputers that exceed 80 PetaFLOPS. In addition, GPT-4, PaLM 2, and other LLMs use thousands of GPUs to train their models.

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

A Foundation Model is thus a transformer model that has been trained on supercomputers on billions of records of data and billions of parameters. The model can perform a wide range of tasks without further fine-tuning. Thus, the scale of Foundation Models is unique. These fully trained models are often called engines. GPT-4, Google BERT, PaLM 2, and scores of transformer models can now qualify as Foundation Models. *Bommasani et al.* (2023) created Ecosystem Graphs to keep track of the growing number of assets on the market (datasets, models, and applications).

We will now examine an example of how Foundation Models work and have changed how we develop programs.

## From general purpose to specific tasks

This section shows how transformer-driven AI is changing the way we work forever. ChatGPT will design, write, and explain a classification program automatically. In this chapter, you will find discriminative, generative, task-specific, and general-purpose features in the specific task we will ask ChatGPT to perform.

Don't be mistaken! Transformer-driven AI is in its early stages. As the models evolve and new ones appear, AI will perform increasingly complex tasks.

We will start by watching ChatGPT write and explain a classification algorithm by itself!

Open `chatGPT_Plus_writes_and_explains_AI.ipynb` in the chapter directory of the GitHub repository of this book.

Two requests were submitted to ChatGPT Plus:

1. Provide a `sklearn` classification of `Iris` with some kind of `matplotlib` graph to describe the results, and don't use OpenAI APIs.
2. Write a detailed explanation for a beginner in Python for this code.

ChatGPT, GPT-3.5-turbo wrote this entire notebook with the explanations!

The only human intervention was to create the notebook and copy the text and code provided by ChatGPT. ChatGPT begins by explaining the program:

```
This program is a simple machine learning project that demonstr  
In this program, you will first load the Iris dataset and split  
This program is a great way for beginners to learn about machin  
Overall, this program is a fun and educational way to learn abo
```

ChatGPT explains the basic configuration of a classification program. It's hard to believe initially, but you will get used to it!

```
Install scikit-learn:  
pip install -U scikit-learn  
1. Import libraries  
We import several libraries that we will use throughout the co  
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
from sklearn.datasets import load_iris  
from sklearn.model_selection import train_test_split  
from sklearn.tree import DecisionTreeClassifier
```

```
from sklearn.metrics import confusion_matrix
from sklearn.tree import plot_tree
```

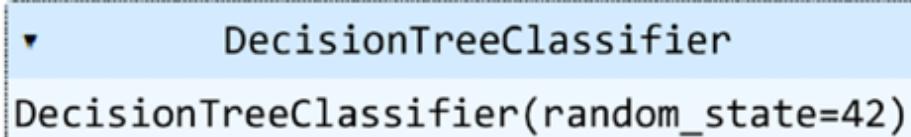
Now, ChatGPT goes on to load and split the dataset into training and testing sets.

```
2.Load the Iris dataset:
iris = load_iris()
We use the load_iris function from Scikit-learn's datasets mod
3.Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(iris.data,
We use Scikit-learn's train_test_split function to split the d
```

At this point, while watching ChatGPT write everything up to now and still typing away furiously, I began to wonder what I was doing there. What was my role in all this? Will AI take over my job? My role in computational linguistics? Although I know the many limitations of Generative AI, it still made me wonder.

Now, ChatGPT configures, displays, and trains a decision tree classifier model:

```
4. Train a decision tree classifier model:
model = DecisionTreeClassifier(random_state=42)
model.fit(X_train, y_train)
```



A screenshot of a code editor showing a tooltip for the class 'DecisionTreeClassifier'. The tooltip is titled 'DecisionTreeClassifier' and contains the full class definition: 'DecisionTreeClassifier(random\_state=42)'. The code editor background shows some other code, including '4. Train a decision tree classifier model:' and 'model = DecisionTreeClassifier(random\_state=42)'.

Figure 1.11 ChatGPT creates a decision tree classifier

We create a `DecisionTreeClassifier` object from Scikit-learn's

Now what? ChatGPT even follows standard machine learning procedures, evaluates the model, and then computes and plots the confusion matrix!

5. Evaluate the model on the testing set:

```
score = model.score(X_test, y_test)
print(f"Accuracy: {score:.2f}")
```

Accuracy: 1.00

We use the `score` method of the trained model to evaluate the accuracy.

6. Compute the confusion matrix:

```
y_pred = model.predict(X_test)
cm = confusion_matrix(y_test, y_pred)
```

7.1. Plot the confusion matrix:

```
fig, ax = plt.subplots()
im = ax.imshow(cm, interpolation="nearest", cmap=plt.cm.Blues)
```

ax. Figure.colorbar(im, ax=ax)

```
ax.set(xticks=np.arange(cm.shape[1]),
       yticks=np.arange(cm.shape[0]),
       xticklabels=iris.target_names, yticklabels=iris.target_
       title="Confusion matrix",
       xlabel="Predicted label",
       ylabel="True label")
```

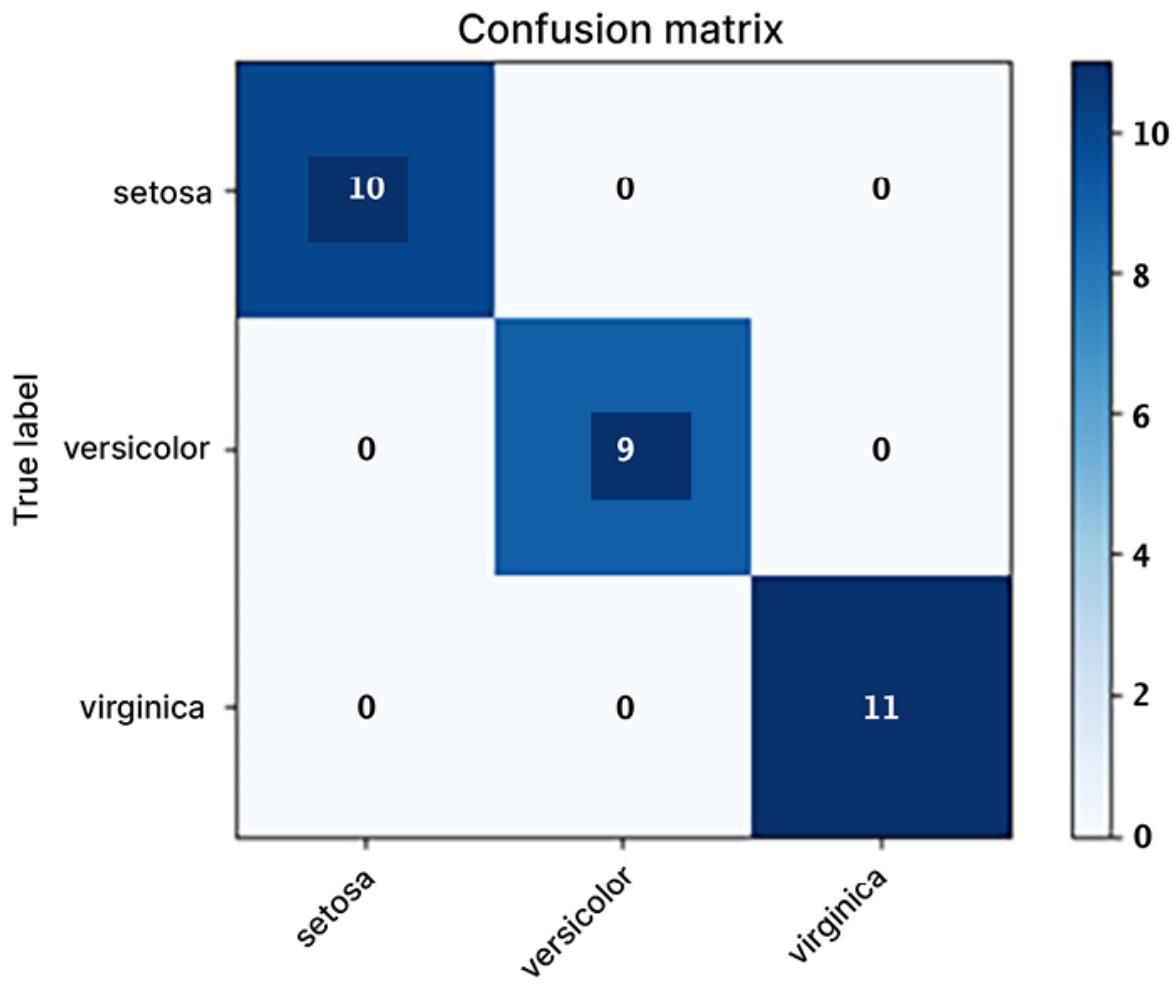
plt.setp(ax.get\_xticklabels(), rotation=45, ha="right", rotation\_mode="auto")

```
for i in range(cm.shape[0]):
```

```
    for j in range(cm.shape[1]):
```

```
        ax.text(j, i, format(cm[i, j], "d"), ha="center", va="center",
                color="white" if cm[i, j] > cm.max() / 2 else "black")
```

```
fig.tight_layout()
```



*Figure 1.12: ChatGPT creates a confusion matrix of the model results*

Finally, ChatGPT plots the decision tree:

```
7.2. Plot the Decision Tree
# Plot the decision tree
plt.figure(figsize=(20,10))
plot_tree(model, feature_names=iris.feature_names, class_names=
plt.show()
```

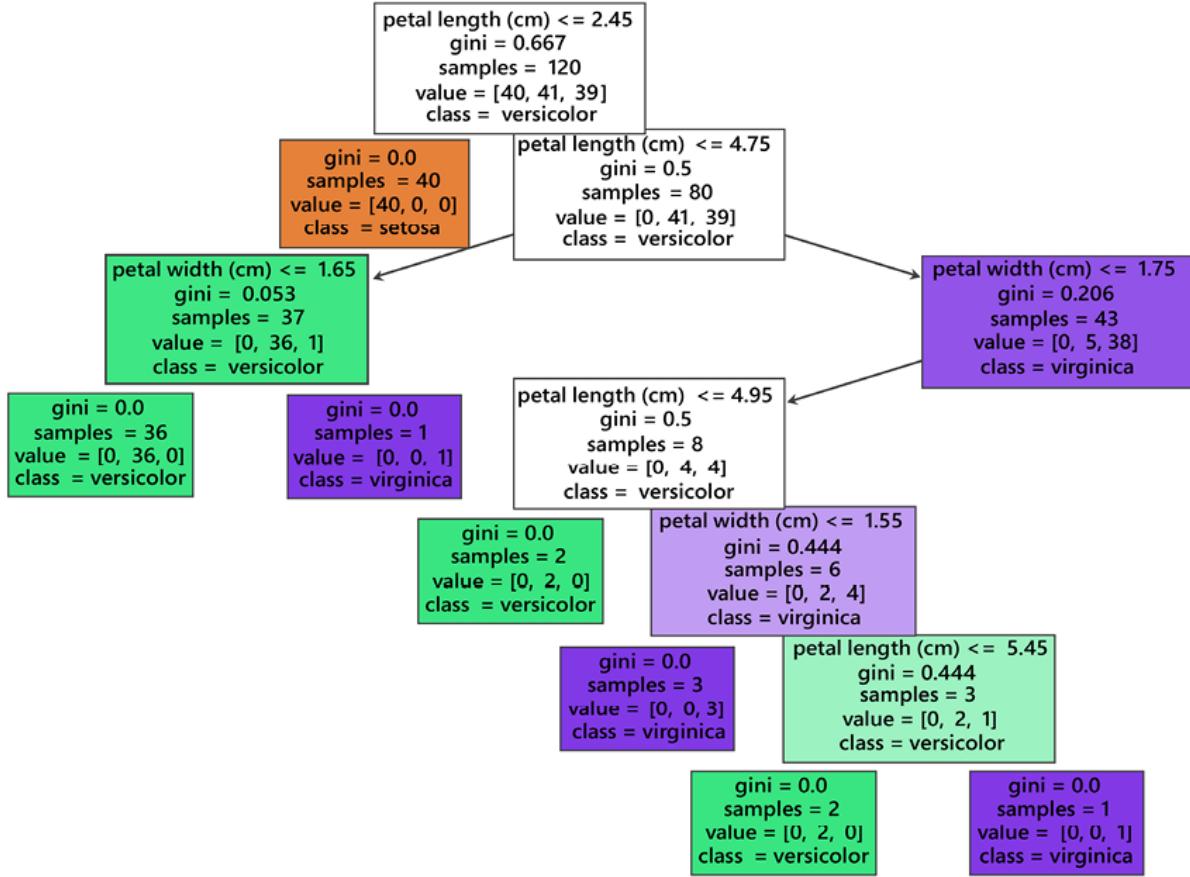


Figure 1.13: Excerpt of the decision tree plotted by ChatGPT

Take a deep breath and let what you just saw sink in. An AI program did 100% of the job by itself. So, of course, the first reaction could be: *Uh oh, AI is going to take over my job! Generative AI will certainly increase its scope if complex problems can be broken down into small components!*

A second reaction could be, “*Hmm, I’m going to push it to its limits until it makes mistakes to prove it’s not worth the hype! AI will never be able to replace the complexity of what I do.*”

These reactions will intensify as transformer-driven AI spreads out to a wide range of daily applications.

As AI professionals, our responsibility entails acquiring a comprehensive understanding of transformers and their functionalities, evaluating their limitations, and enhancing our skill set.

## The role of AI professionals

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

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

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

Let's go through a few examples:

- **API:** The OpenAI API does not require an AI developer. A web designer can create a form, and a linguist or **Subject Matter Expert (SME)** can prepare the prompt input texts. The primary role of an AI specialist will require linguistic skills to show, not just tell, the ChatGPT/GPT-4 models and their successors how to accomplish a task. Showing a model what to do, for example, involves working on the context of the input. This new task is named *prompt engineering*. A *prompt engineer* has quite a future in AI!

- **Library:** The Google Trax library requires limited development to start with ready-to-use models. An AI professional with linguistics and NLP skills can work on the datasets and the outputs. The AI professional can also use Trax's toolkit to build tailored models for a project.
- **Training and fine-tuning:** Some Hugging Face functionality requires limited development, providing both APIs and libraries. However, sometimes, we still have to get our hands dirty. In that case, training, fine-tuning the models, and finding the correct hyperparameters will require the expertise of an AI specialist.
- **Development-level skills:** In some projects, the tokenizers and the datasets do not match. Or the embeddings of a model might not fit a project. In this case, an AI developer working with a linguist, for example, can play a crucial role. Therefore, computational linguistics training can come in very handy at this level.

The recent evolution of NLP AI can be termed as “embedded transformers” in assistants, copilots, and everyday software, which is disrupting the AI development ecosystem:

- LLM transformers with billions of parameters and many layers to train, such as OpenAI GPT-4, are currently embedded in several Microsoft Azure applications with GitHub Copilot, among other services and software.
- The embedded transformers are not accessible directly but provide automatic development support, such as automatic code generation. They also provide summarization, question and answering capabilities, and many other tasks in a growing number of applications.
- The usage of embedded transformers is both endless and seamless for the end user with assisted text completion.

We will explore this fascinating new world of embedded transformers throughout our journey in this book.

The skill set of an LLM AI professional requires adaptability, cross-disciplinary knowledge, and above all, *flexibility*. This book will provide the AI specialist with a variety of transformer ecosystems to adapt to the new paradigms of the market. These opposing and often conflicting strategies leave us with various possible implementations.

The future of AI professionals will expand into many specializations.

## The future of AI professionals

The societal impact of Foundation Models should not be underestimated. Prompt engineering has become a skill required for AI professionals.

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

This book is thus not limited to prompt engineering but to a wide range of design skills required to be an LLM AI specialist.

Prompt engineering is a subset of the design skills an AI professional must develop. Here are some domains an AI professional can invest in:

- **AI specialists** have specialized knowledge or skills in one of the fields of AI. One of the interesting fields is fine-tuning, maintaining, and supporting AI systems.
- **AI architects** design architectures and provide solutions for scaling and other information for deployment and production.

- **AI experts** have authoritative knowledge and skills in AI. An AI expert can contribute to the field with research, papers, and innovative solutions.
- **AI analysts** focus on data, big data, and model architecture to provide information for other teams.
- **AI researchers** focus on research in a university or private company.
- **AI engineers** design AI, build systems, and implement AI models.
- **AI managers** leverage the critical skills of project management, product management, and resource management (human, machine, and software).
- Many other fields will appear showing that there are many opportunities in AI to develop assets, such as datasets, models, and applications.

Transformers have spread everywhere, and we need to find the right resource for a project.

## What resources should we use?

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

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

Microsoft 365 and Google Workspace now possess Generative AI tools powered by state-of-the-art transformers. As a result, every Microsoft 365 and Google Workspace user has AI at the tip of their fingers!

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

For example, you start your presentation with an awesome PowerPoint on Hugging Face. You might get an adverse reaction from a manager who says, “*I’m sorry, but we use Google Cloud AI here for this type of project, not Hugging Face. Can you implement Google Cloud AI, please?*” If you don’t, it’s game over for you.

The same problem could have arisen by specializing in Google Cloud AI. But instead, you might get the reaction of a manager who wants to use OpenAI’s ChatGPT/GPT-4 models with an API and no development. If you specialize in OpenAI’s GPT engines with APIs and no development, you might face a project manager or customer who prefers Hugging Face’s AutoML or HuggingGPT!

The worst thing that could happen to you is that a manager accepts your solution, but in the end, it does not work at all for the NLP tasks of that project.

The key concept to remember is that if you only focus on the solution you like, you will most likely sink with the ship at some point.

*Focus on the system you need, not the one you like.*

This book is not designed to explain every transformer solution on the market. Instead, this book aims to explain enough transformer ecosystems for you to be flexible and adapt to any situation you face in an NLP project.

Let’s go through a few approaches that can help you make a decision.

# Decision-making guidelines

My personal experience relies on three simple principles:

- Understand the task(s) the AI model is expected to do at an Subject Matter Expert level. Write it down on paper without ChatGPT or any help. Just you. Then, write a detailed flow chart.
- Build a prototype dataset that can be duplicated to whatever scale you need to test the hardware/software performances.
- Try one to a few NLP/CV models to see which one fits your needs: open source or not, API or not, local installation, cloud platform, and the other constraints of your project.

Take your time. Don't let yourself get bullied by anybody. Use science to guide you, not opinions.

Let's now ask ChatGPT Plus for additional guidance for the three principles I just mentioned. ChatGPT has some interesting advice:

1. **Problem assessment:** Begin by thoroughly understanding the problem. What are the key challenges? What scale of data are you dealing with?
2. **Skill set evaluation:** What is the team's proficiency level? Familiarity with certain platforms might accelerate development.
3. **Scalability:** Consider the growth trajectory. Will the chosen solution scale as data grows or as the problem complexity increases?
4. **Community and support:** Tools and models with active communities can be beneficial due to available resources, tutorials, and prompt solutions to common issues.
5. **Interoperability:** How well does the solution integrate with existing systems or potential future tools?

Remember, the tool or model is only as good as the team's understanding of its principles and its applicability to the problem at hand.

One of the challenges of an NLP project remains to assess the solution to implement. AI APIs are one of the ever-evolving tools on the market we must explore.

## The rise of transformer seamless APIs and assistants

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

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

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

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

OpenAI transformer models are so effective and humanlike that the present policy requires a potential user to complete a request form. Once the request

has been accepted, the user can access a universe of natural language processing!

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

1. Obtain an API key in one click.
2. Import OpenAI in a notebook in one line.
3. Enter any NLP task you wish in a *prompt*.
4. You will receive a response with no other functions to write.

For example, you can translate natural language to an SQL query, as explained on OpenAI's platform,

<https://platform.openai.com/examples/default-sql-translate>:

### Prompt

```
#### Postgres SQL tables, with their properties:  
#  
# Employee(id, name, department_id)  
# Department(id, name, address)  
# Salary_Payments(id, employee_id, amount, date)  
#  
#### A query to list the names of the departments which employed more than  
10 employees in the last 3 months  
SELECT
```

Figure 1.14: Prompt to create an SQL query on OpenAI

## Sample response

```
SELECT d.name
  FROM Department d
  INNER JOIN Employee e ON d.id = e.department_id
  INNER JOIN Salary_Payments sp ON e.id = sp.employee_id
 WHERE sp.date > NOW() - INTERVAL '3 months'
 GROUP BY d.name
 HAVING COUNT(*) > 10
```

Figure 1.15: Response from the SQL prompt

And that's it! The queries may not always be perfect and may need amending, but this shows you the power of LLMs. Welcome to the world of Generative AI!

Developers focusing on code-only solutions will evolve into a generation of developers with cross-disciplinary mindsets leveraging the power of AI copilots.

The AI professionals will learn how to design ways to *show* a transformer model what is expected and not intuitively *tell* it what to do. By the end of the book, you will have acquired several methods to control the behavior of these cutting-edge AI models.

Though APIs may satisfy many needs, they also have limits. For example, a multipurpose API might be reasonably good in all tasks but not good enough for a specific NLP task. For instance, translating with transformers is no easy task. In that case, an AI developer, consultant, or project manager must prove that an API alone cannot solve the required NLP task. Therefore, we might need to search for a solid library, alternative solutions, or develop one.

# Choosing ready-to-use API-driven libraries

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

However, it can prove to be both educational and effective in some cases. We will explore the recent evolution of Google in translations and implement Google Trax in *Chapter 4, Advancements in Translations with Google Trax, Google Translate, and Gemini*.

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

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

Try translating a sentence requiring coreference resolution in English to French using Google Translate. The sentence is, *A user visited the AllenNLP website, tried a transformer model, and found it interesting*. Google Translate produces the following translation:

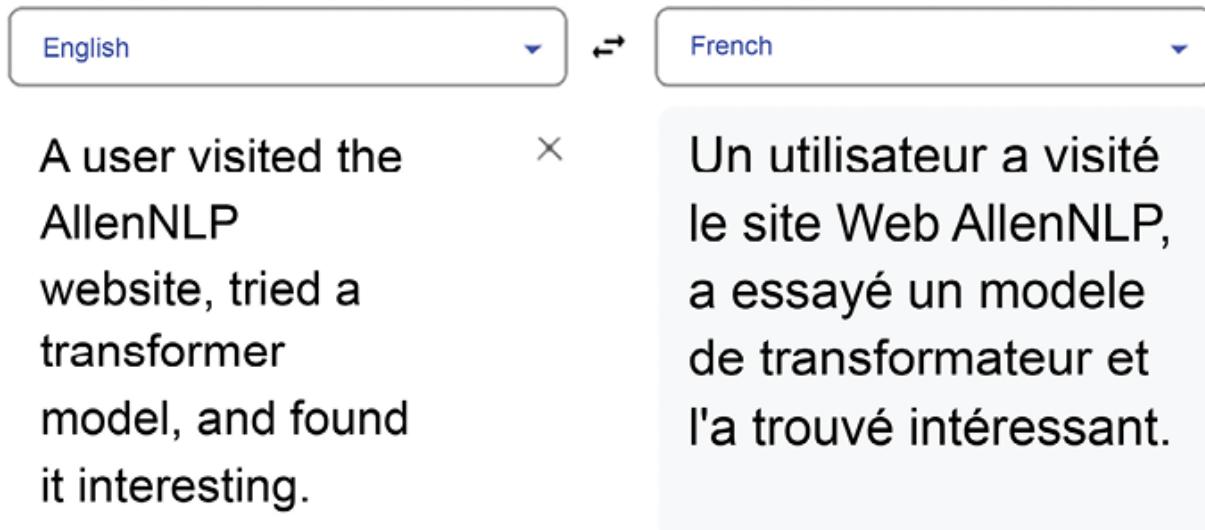


Figure 1.16: Coreference resolution in a translation using Google Translate

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

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

Google may improve this example by the time the book is published, but there are thousands more limitations to the system.

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

Choosing an API is one task. Finding a suitable transformer model is no easy task with the increasing range of solutions.

## **Choosing a cloud platform and transformer model**

Big tech corporations dominate the NLP market. Google, Facebook, and Microsoft alone run billions of NLP routines daily, increasing their AI models' unequaled power thanks to the data they gather. The big giants now offer a wide range of transformer models and have top-ranking Foundation Models.

The threshold of Foundation Models is fully trained transformers on supercomputers such as OpenAI GPT-4, Google PaLM, and the new models they will continually produce. Foundation Models are often proprietary, meaning we cannot access their code, although we can fine-tune some of them through a cloud service.

Hugging Face has a different approach and offers a wide range of transformer models with model cards, source code, datasets, and more development resources. In addition, Hugging Face offers high-level APIs and developer-controlled APIs. In several chapters of this book, we will explore Hugging Face as an educational tool and a possible solution for many tasks.

Google Cloud, Microsoft Azure, AWS, Hugging Face, and others offer fantastic services on their platforms! When looking at the growing mountain of transformer-driven AI, we need to find where we fit in.

## **Summary**

Transformers forced AI to make profound evolutions. Foundation Models, including their Generative AI abilities, are built on top of the digital revolution connecting everything to everything with underlying processes everywhere. Automated processes are replacing human decisions in critical areas, including NLP.

RNNs slowed the progression of automated NLP tasks required in a fast-moving world. Transformers filled the gap. A corporation needs summarization, translation, and a wide range of NLP tools to meet the challenges of the growing volume of incoming information.

Transformers have thus spurred an age of AI industrialization. We first saw how the  $O(1)$  time complexity of the attention layers and their computational time complexity,  $O(n^2 * d)$ , shook the world of AI.

We saw how the one-token flexibility of transformer models pervaded every domain of our everyday lives!

Platforms such as Hugging Face, Google Cloud, OpenAI, and Microsoft Azure provide NLP tasks without installation and resources to implement a transformer model in customized programs. For example, OpenAI provides an API requiring only a few code lines to run one of the powerful GPT-4 generation models. Google Trax provides an end-to-end library, Google Vertex AI has easy-to-implement tools, and Hugging Face offers various transformer models and implementations. We will be exploring these ecosystems throughout this book.

We then saw that the transformer architecture leads to automation that radically deviates from the former AI. As a result, broader skill sets are required for an AI professional. For example, a project manager can decide to implement transformers by asking a web designer to create an interface for Google Cloud AI or OpenAI APIs through prompt engineering. Or, when

required, a project manager can ask an AI specialist to download Google Trax or Hugging Face resources to develop a full-blown project with a customized transformer model.

The transformer is a game-changer for developers whose roles will expand and require more designing than programming. In addition, embedded transformers will provide assisted code development and usage. *These new skill sets are a challenge but lead to new, exciting horizons.*

In *Chapter 2, Getting Started with the Architecture of the Transformer Model*, we will start with the Original Transformer architecture.

## Questions

1. ChatGPT is a game-changer. (True/False)
2. ChatGPT can replace all AI algorithms. (True/False)
3. AI developers will sometimes have no AI development to do.  
(True/False)
4. AI developers might have to implement transformers from scratch.  
(True/False)
5. It's not necessary to learn more than one transformer ecosystem, such as Hugging Face. (True/False)
6. A ready-to-use transformer API can satisfy all needs. (True/False)
7. A company will accept the transformer ecosystem a developer knows best. (True/False)
8. Cloud transformers have become mainstream. (True/False)
9. A transformer project can be run on a laptop. (True/False)
10. AI specialists will have to be more flexible. (True/False)

# References

- *Bommansani et al., 2021, On the Opportunities and Risks of Foundation Models:* <https://arxiv.org/abs/2108.07258>
- *Rishi Bommansani, Dilara Soylu, Thomas I. Liao, Kathleen A. Creel, and Percy Liang, 2023, Ecosystem Graphs: The Social Footprint of Foundation Models:* <https://arxiv.org/abs/2303.15772>
- *Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin, 2017, Attention is All You Need:*  
<https://arxiv.org/abs/1706.03762>
- *Chen et al., 2021, Evaluating Large Language Models Trained on Code:* <https://arxiv.org/abs/2107.03374>
- Microsoft AI: <https://innovation.microsoft.com/en-us/ai-at-scale>
- OpenAI: <https://openai.com/>
- Google AI: <https://ai.google/>
- Google Trax: <https://github.com/google/trax>
- AllenNLP: <https://allennlp.org/>
- Hugging Face: <https://huggingface.co/>
- Google Cloud TPU:  
<https://cloud.google.com/tpu/docs/intro-to-tpu>

# Further reading

- *Tyna Eloundou, Sam Manning, Pamela Mishkin, and Daniel Rock, 2023, GPTs are GPTs: An Early Look at the Labor Market Impact*

*Potential of Large Language Models:*

<https://arxiv.org/abs/2303.10130>

- *Jussi Heikkilä, Julius Rissanen, and Timo Ali-Vehmas, 2023, Coopetition, standardization, and general purpose technologies: A framework and an application:*  
<https://www.sciencedirect.com/science/article/pii/S0308596122001902>
- NVIDIA blog on Foundation Models:  
<https://blogs.nvidia.com/blog/2023/03/13/what-are-foundation-models/>
- On Markov chains: <https://mathshistory.st-andrews.ac.uk/Biographies/Markov/>

## Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://www.packt.link/Transformers>



# 2

## Getting Started with the Architecture of the Transformer Model

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

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

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

Since 2017, transformer models such as OpenAI’s ChatGPT and GPT-4, Google’s PaLM and LaMBDA, and other **Large Language Models (LLMs)** have emerged. However, this is just the beginning! You need to understand how attention heads work to join this new era of LLM for AI experts.

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

This chapter covers the following topics:

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

With all the innovations and library updates in this cutting-edge field, packages and models change regularly. Please go to the GitHub repository for the latest installation and code



examples:

<https://github.com/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition/tree/main/Chapter02>.

You can also post a message in our Discord community (<https://www.packt.link/Transformers>) if you have any trouble running the code in this or any chapter.

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

## The rise of the Transformer: Attention Is All You Need

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

Exploring the architecture of the Transformer is essential for the following reasons:

- The source code of any machine learning or deep learning model remains a product of classical mathematics.
- It's only when the system is put to work that it becomes artificial intelligence.
- Artificial intelligence is the function of a model built with classical mathematics.

- The Transformer is no exception. If you look inside the algorithm, you will discover classical mathematics.
- The magic of AI is not in math but in the **behavior** of the system when it actually runs and performs tasks.
- Sometimes, we must get our hands dirty to find ways to optimize the functions implemented in our AI (or any other software).

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

The Original Transformer model is a stack of six layers. Each layer contains sublayers. The output of layer  $l$  is the input of layer  $l+1$  until the final prediction is reached. There is a six-layer encoder stack on the left and a six-layer decoder stack on the right:

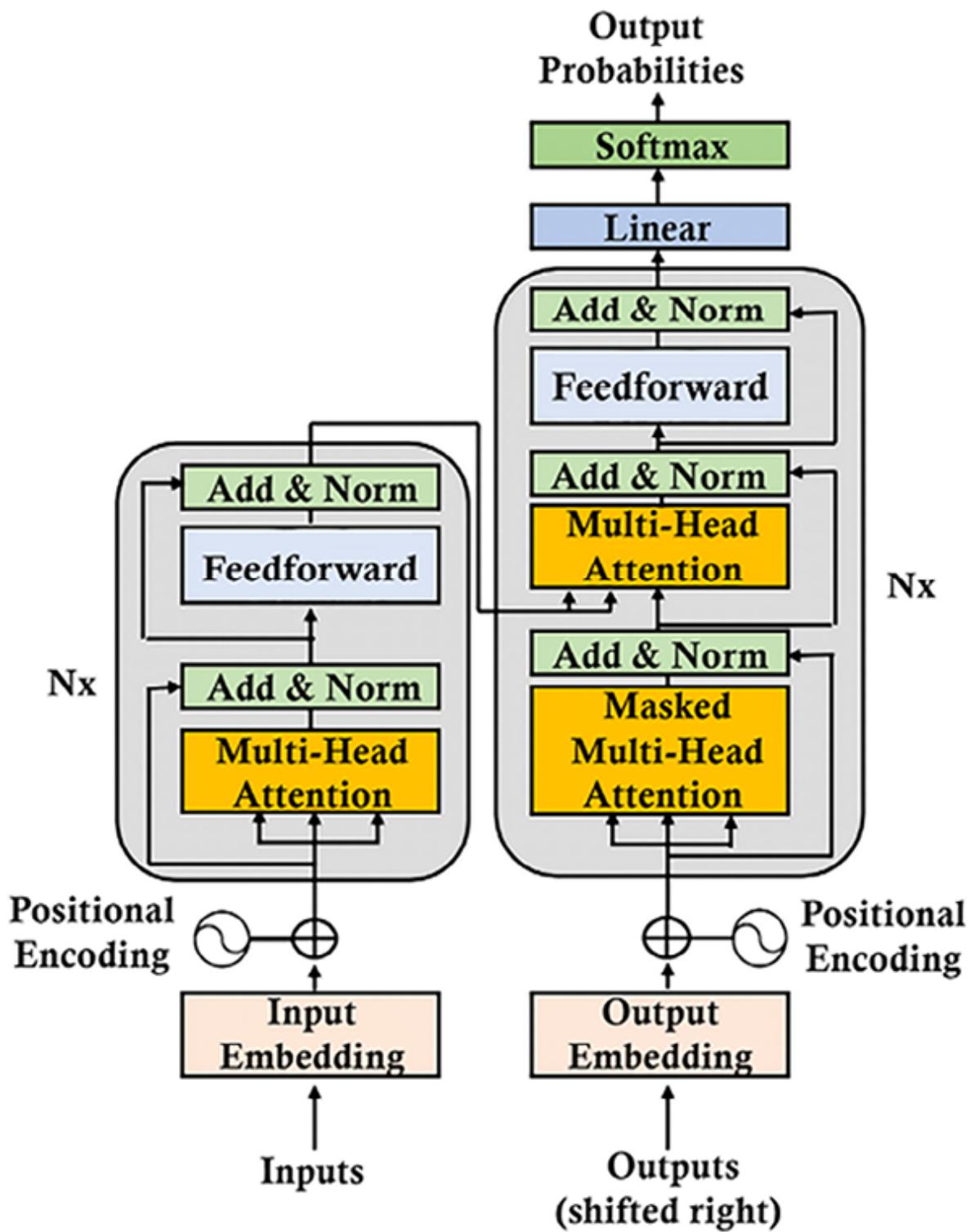


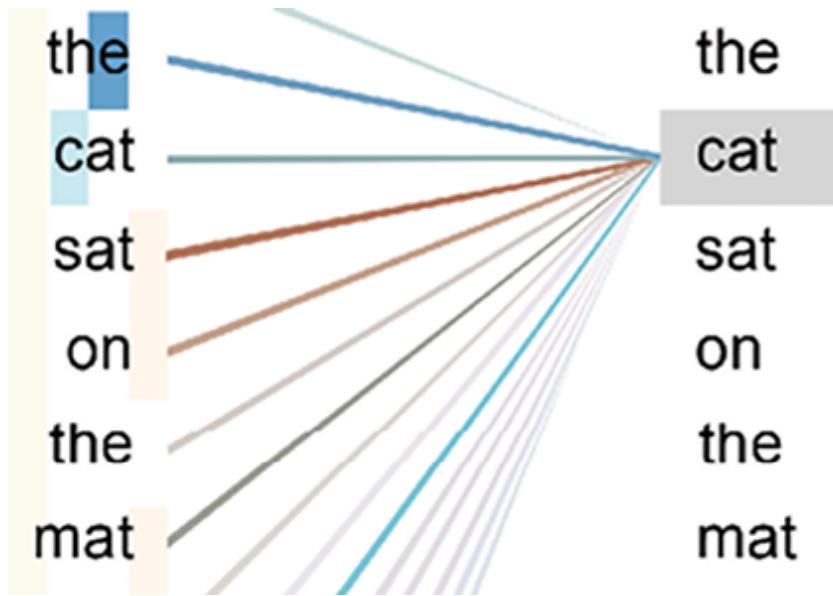
Figure 2.1: The architecture of the Transformer

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

Attention has replaced recurrence functions requiring an increase in the number of parameters as the distance between two words increases. The attention mechanism is a “word-to-word” operation. It is actually a token-to-token operation, but we will keep it to the word level to keep the explanation simple. The attention mechanism determines how each word is related to all other words in a sequence, including the word being analyzed itself. For example, 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 between a given word and all the other words, including itself (“cat” and “cat”):



*Figure 2.2: Attending to all the words*

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

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

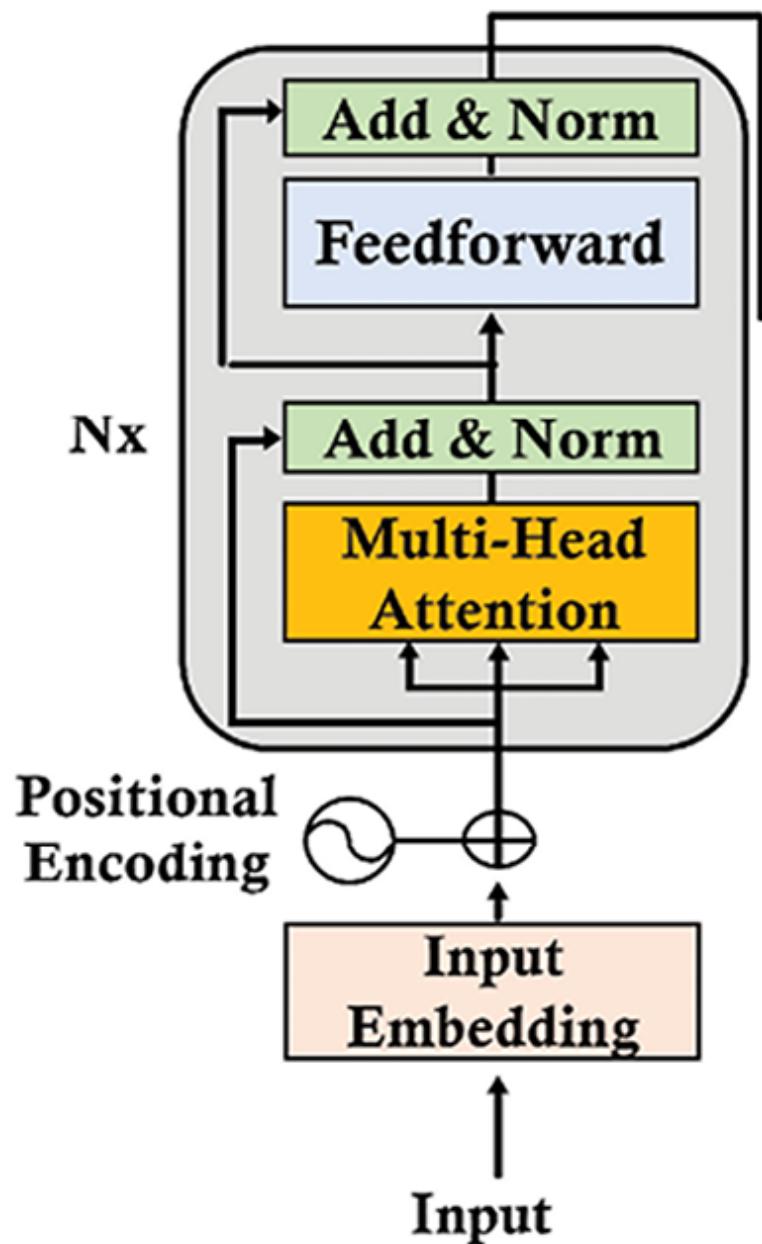
- A broader in-depth analysis of sequences
- The preclusion of recurrence reducing calculation operations
- Implementation of parallelization, which reduces training time
- Each attention mechanism learns different perspectives of the same input sequence

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

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

## The encoder stack

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



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

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

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

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

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 through the different weights of each layer.

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

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

The designers of the Transformer introduced a very efficient constraint. The output of every sublayer of the model has a constant dimension, including the embedding layer and the residual connections. This dimension is  $d_{model}$

and can be set to another value depending on your goals. In the Original Transformer architecture,  $d_{model} = 512$ .

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

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

We will begin with the embedding sublayer.

## Input embedding

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

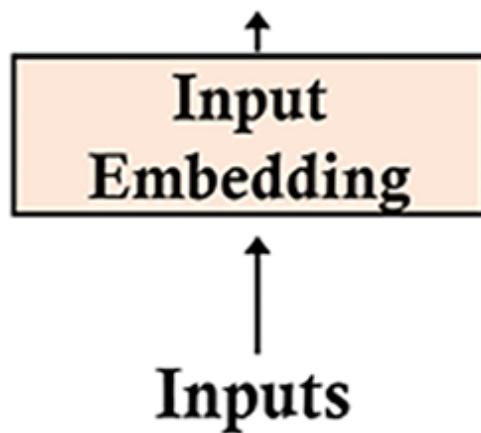


Figure 2.4: The input embedding sublayer of the Transformer

The embedding sublayer works like other standard transduction models. A tokenizer will transform a sentence into tokens. Each tokenizer has its

methods, such as **Byte Pair Encoding (BPE)**, word piece, and sentence piece methods. The Transformer initially used BPE, but other models use other methods.

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

```
['the', 'transform', 'er', 'is', 'an', 'innovative', 'n', 'l',
```

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

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

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

The Transformer contains a learned embedding sublayer. As a result, 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 predict *context* words. For example, if `word(i)` is the center word in a two-step window, a skip-gram model will analyze `word(i-2), word(i-1),`

word( $i+1$ ), and word( $i+2$ ). Then, the window will *slide* and repeat the process. A skip-gram model generally contains an input layer, weights, a hidden layer, and an output containing the word embeddings of the tokenized input words.

Suppose we need to perform embedding for the following sentence:

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

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

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

```
black=[[-0.01206071  0.11632373  0.06206119  0.01403395  0.0954  
       ...  
     0.04340003 -0.13192849 -0.00945092 -0.00835463 -0.06487109  0.0  
     -0.04756588  0.08812257  0.00200338 -0.0931043  -0.03507337  0.0  
     -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  6.7435
       1.04376070e-01  1.09921647e-02 -5.46298288e-02 -1.18385479e-
       4.41223830e-02 -1.84863899e-02 -6.84073642e-02  3.21860164e-
       4.09143828e-02 -2.74433400e-02 -2.47369967e-02  7.74542615e-
       9.80964210e-03  2.94299088e-02  2.93895267e-02 -3.29437815e-
...
       7.20389187e-02  1.57317147e-02 -3.10291946e-02 -5.51304631e-0
      -7.03861639e-02  7.40829483e-02  1.04319192e-02 -2.01565702e-
      2.43322570e-02  1.92969330e-02  2.57341694e-02 -1.13280728e-
      8.45847875e-02  4.90090018e-03  5.33546880e-02 -2.31553353e-
      3.87288055e-05  3.31782512e-02 -4.00604047e-02 -1.02028981e-
      3.49597558e-02 -1.71501152e-02  3.55573371e-02 -1.77437533e-
      -5.94457164e-02  2.21221056e-02  9.73121971e-02 -4.90022525e-
```

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

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

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

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

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

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

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

Let's see how positional encoding works.

## Positional encoding

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

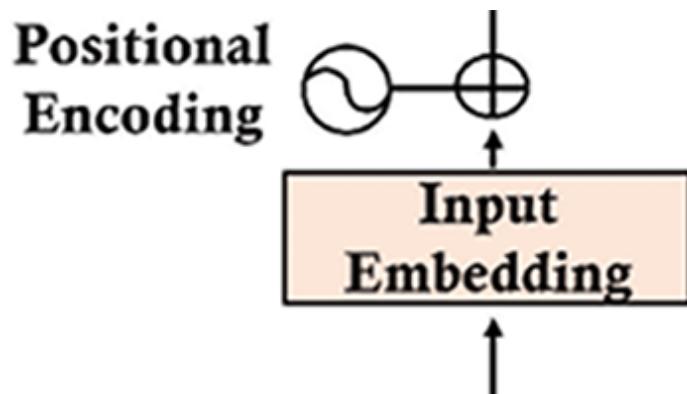


Figure 2.5: Positional encoding

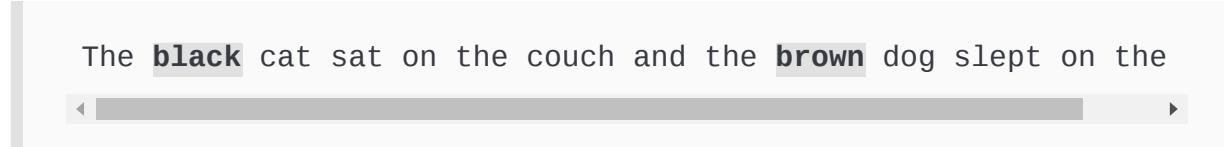
We cannot create independent positional vectors that would have a high cost on the training speed of the Transformer and make attention sublayers overly complex to work with. The idea is to add a positional encoding value to the

input embedding instead of having additional vectors to describe the position of a token in a sequence.

The Original Transformer model has only one vector that contains word embedding and position encoding.

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

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



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

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

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

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

*Vaswani et al. (2017)* provide sine and cosine functions so that we can generate different frequencies for the **positional encoding (PE)** for each

position and each dimension  $i$  of the  $d_{model} = 512$  of the word embedding vector:

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

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

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

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

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

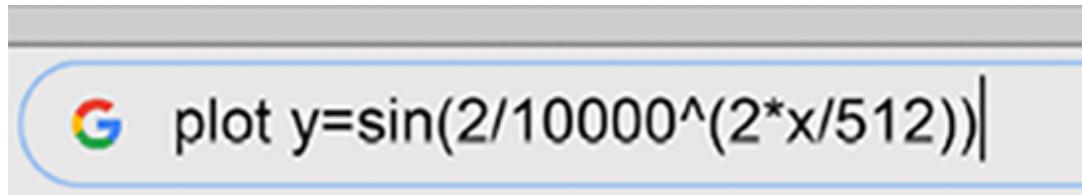
Google Brain Trax and Hugging Face, among others, provide ready-to-use libraries for the word embedding section and the present positional encoding

section. Thus, you don't need to run the code I shared in this section. However, if you wish to explore the code, you will find it in the `positional_encoding.ipynb` notebook in the directory of this chapter in the GitHub repository.

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

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

Just enter the plot request:



*Figure 2.6: Plotting with Google*

You will obtain the following graph:

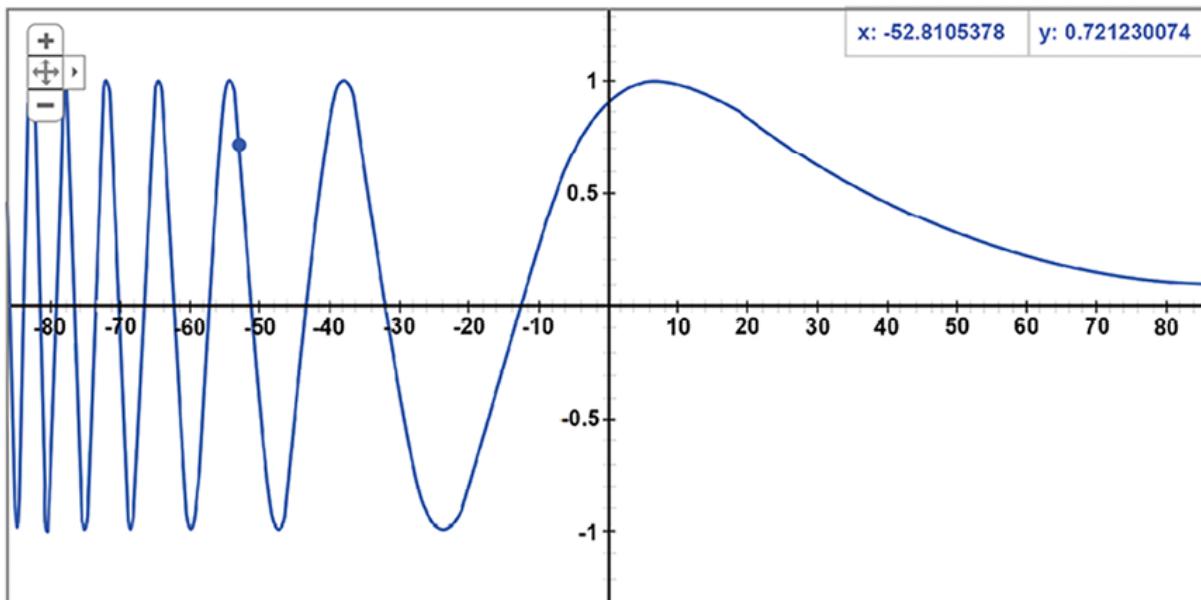


Figure 2.7: The graph

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

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

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-
  9.87046242e-01 -1.60435960e-01  9.99164224e-01 -4.08766568e-
  9.97479975e-01  7.09482506e-02  9.84703004e-01  1.74241230e-
  9.63226616e-01  2.68690288e-01  9.35118318e-01  3.54335666e-
  9.02130723e-01  4.31462824e-01  8.65725577e-01  5.00518918e-
  8.27103794e-01  5.62049210e-01  7.87237823e-01  6.16649508e-
  7.46903539e-01  6.64932430e-01  7.06710517e-01  7.07502782e-
...
  5.47683925e-08   1.00000000e+00   5.09659337e-08   1.00000000e+
```

```

4.74274735e-08 1.00000000e+00 4.41346799e-08 1.00000000e+
4.10704999e-08 1.00000000e+00 3.82190599e-08 1.00000000e+
3.55655878e-08 1.00000000e+00 3.30963417e-08 1.00000000e+
3.07985317e-08 1.00000000e+00 2.86602511e-08 1.00000000e+
2.66704294e-08 1.00000000e+00 2.48187551e-08 1.00000000e+
2.30956392e-08 1.00000000e+00 2.14921574e-08 1.00000000e+

```

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-
  6.92634165e-01 -7.21289039e-01 9.79174793e-01 -2.03019097e-
  9.37632740e-01 3.47627431e-01 6.40478015e-01 7.67976522e-
  2.09077001e-01 9.77899194e-01 -2.37917677e-01 9.71285343e-
 -6.12936735e-01 7.90131986e-01 -8.67519796e-01 4.97402608e-
 -9.87655997e-01 1.56638563e-01 -9.83699203e-01 -1.79821849e-
 ...
 2.73841977e-07 1.00000000e+00 2.54829672e-07 1.00000000e+0
 2.37137371e-07 1.00000000e+00 2.20673414e-07 1.00000000e+
 2.05352507e-07 1.00000000e+00 1.91095296e-07 1.00000000e+
 1.77827943e-07 1.00000000e+00 1.65481708e-07 1.00000000e+
 1.53992659e-07 1.00000000e+00 1.43301250e-07 1.00000000e+
 1.33352145e-07 1.00000000e+00 1.24093773e-07 1.00000000e+
 1.15478201e-07 1.00000000e+00 1.07460785e-07 1.00000000e+

```

Having looked at the results obtained with an intuitive literal translation of the *Vaswani et al.* (2017) functions into Python, we now need to check whether the results are meaningful.

The cosine similarity function used for word embedding comes in handy for getting 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 depending on the corpus used to train them. The problem is now how to add the positional encoding to the word embedding vectors.

## **Adding positional encoding to the embedding vector**

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

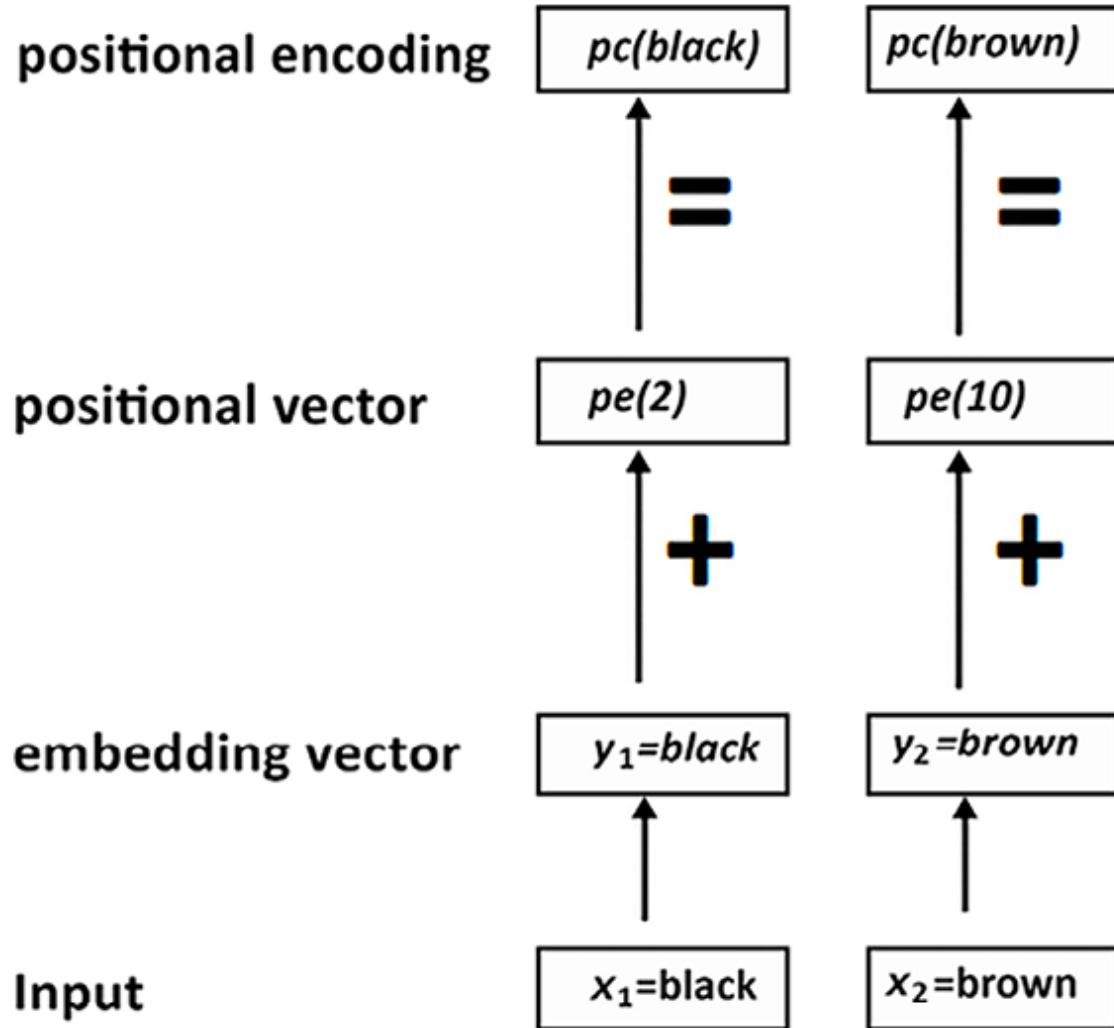


Figure 2.8: Positional encoding

If we go back and take the word embedding of `black`, for example, and name it  $y_1 = \text{black}$ , 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`:

$$pc(\text{black}) = y_1 + pe(2)$$

In the following example, we add the positional vector to the embedding vector of the word `black`, which are both of the same size (512):

```

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

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

```

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

```

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

```

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

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

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

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

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

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

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

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

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

## Sublayer 1: Multi-head attention

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

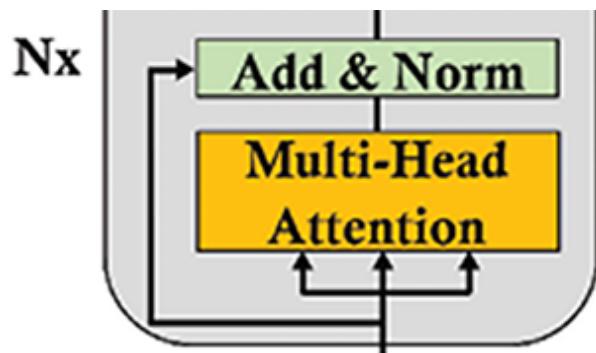


Figure 2.9: Multi-head attention sublayer

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

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

## The architecture of multi-head attention

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

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

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

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 determine 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 for the 8 heads of the model to project the  $d_{model} = 512$  dimensions of each word  $x_n$  in sequence  $x$  into  $d_k = 64$  dimensions.

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

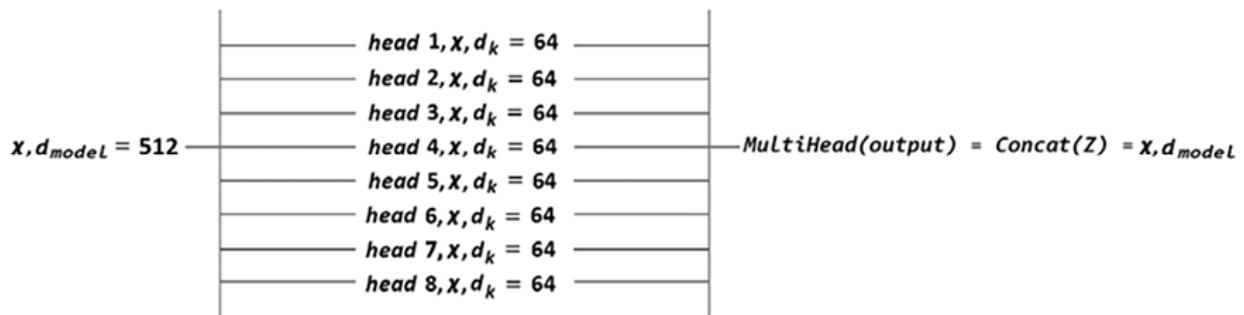


Figure 2.10: Multi-head representations

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

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

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

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

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

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

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

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

- A query matrix ( $Q$ ) that has a dimension of  $d_q = 64$ , which seeks all the key-value pairs of the other “word” matrices.
- A key matrix ( $K$ ) with a dimension of  $d_k = 64$ .
- A value matrix ( $V$ ) with a dimension of  $d_v = 64$ .

Attention is defined as scaled dot-product attention, which is represented in the following equation in which we plug  $Q$ ,  $K$ , and  $V$ :

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

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

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

You can modify all the parameters, such as the number of layers, heads,  $d_{\text{model}}$ ,  $d_k$ , and other variables of the Transformer, to fit your model. It is

essential to understand the original architecture before modifying it or exploring variants of the original model designed by others.

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

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

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

Remember that an AI specialist will face the challenge of dealing with multiple architectures for the same algorithm.

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

## Step 1: Represent the input

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

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

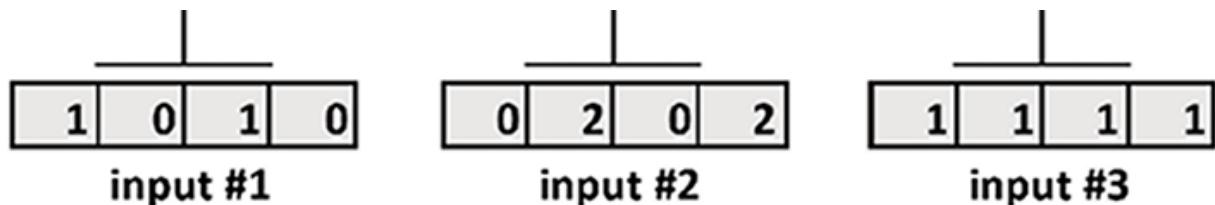


Figure 2.11: Input of a multi-head attention sublayer

We will now add the weight matrices to our model.

## Step 2: Initializing the weight matrices

Each input has three weight matrices:

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

These three 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  $3 \times 4$  weight matrices to visualize the intermediate results more easily and perform dot products with the input  $x$ .

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

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

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

The output is the `w_query` weight matrix:

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

We will now initialize the key weight matrix:

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

The output is the key weight matrix:

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

Finally, we initialize the value weight matrix:

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

The output is the value weight matrix:

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

The second step of our model is ready:

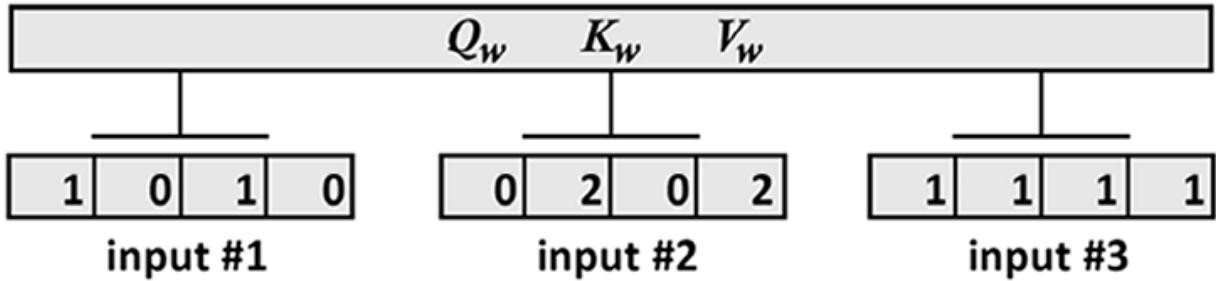


Figure 2.12: Weight matrices added to the model

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

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

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

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

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

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

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

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

```
[2. 2. 2.]  
[2. 1. 3.]]
```

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

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

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

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

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

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

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

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

The third step of our model is ready:

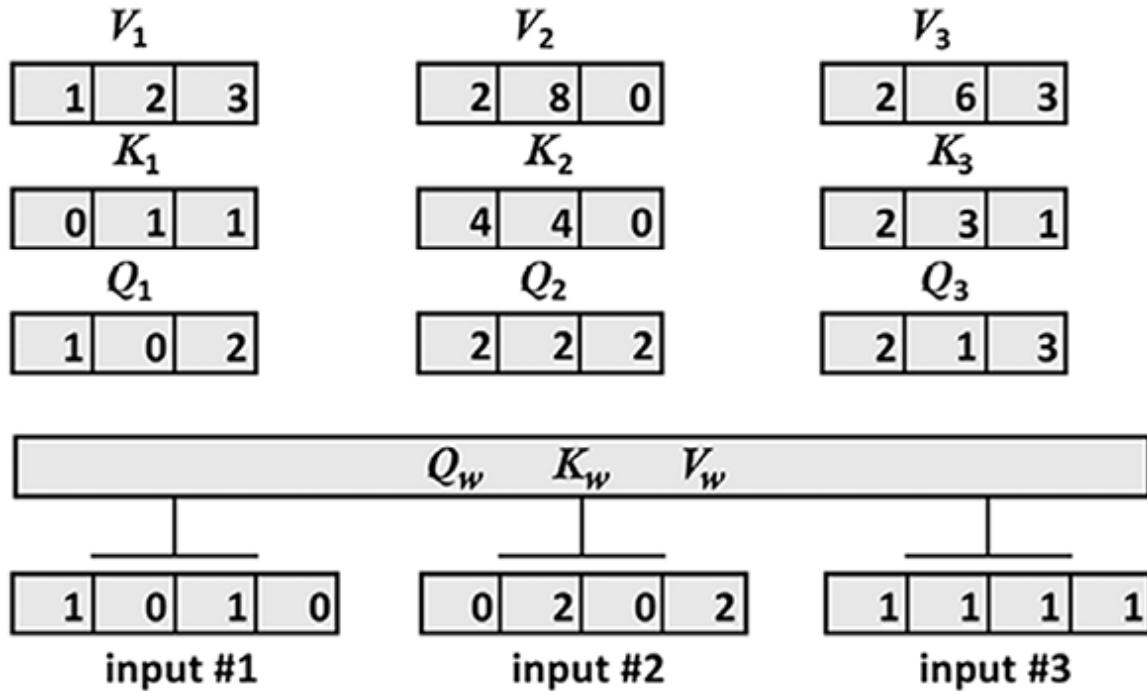


Figure 2.13:  $Q$ ,  $K$ , and  $V$  are generated

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

## Step 4: Scaled attention scores

The attention head now implements the original Transformer equation:

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

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

$$\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)$$

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

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

The intermediate result is displayed:

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

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

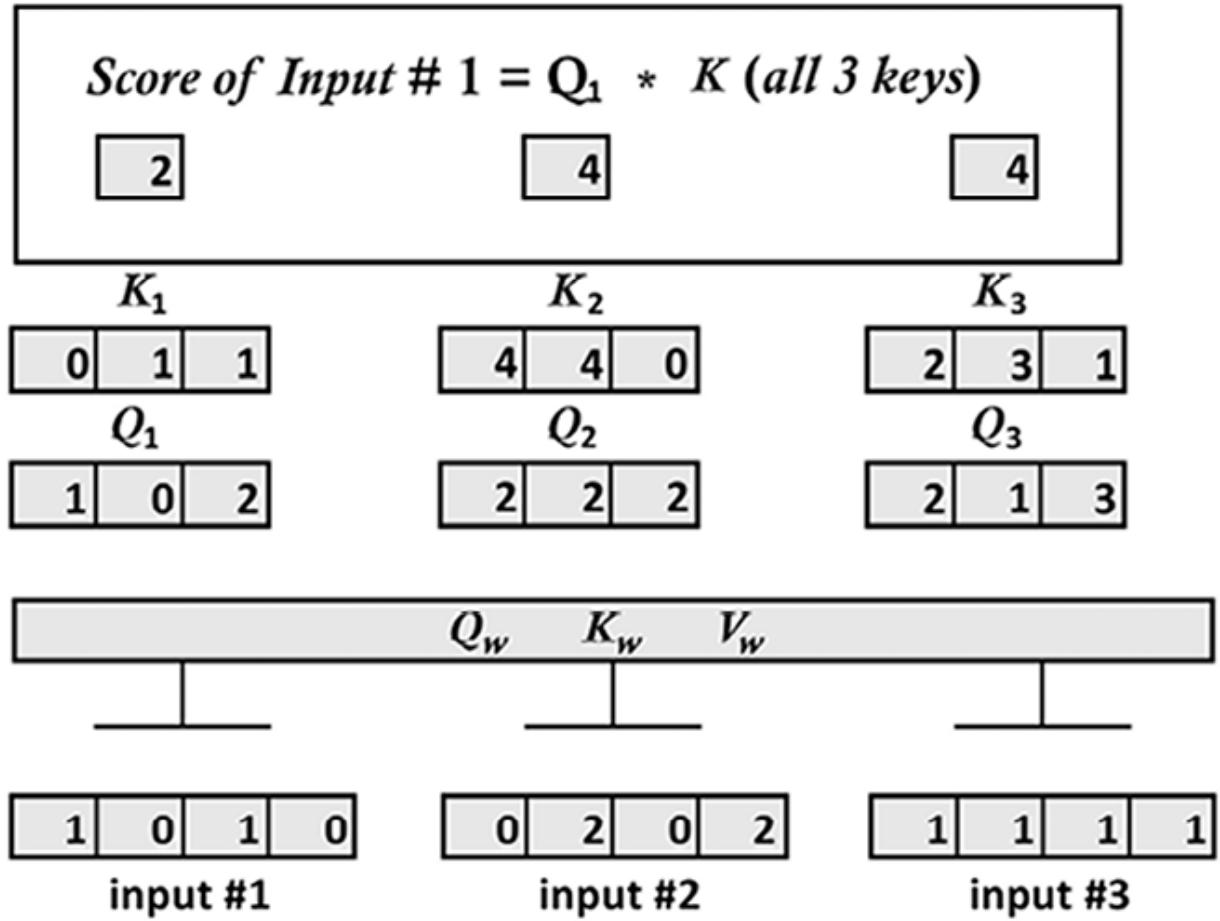


Figure 2.14: Scaled attention scores for input #1

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

## Step 5: Scaled softmax attention scores for each vector

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

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

```
attention_scores[1]=softmax(attention_scores[1])
attention_scores[2]=softmax(attention_scores[2])
print(attention_scores[0])
print(attention_scores[1])
print(attention_scores[2])
```

We obtain scaled softmax attention scores for each vector:

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

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

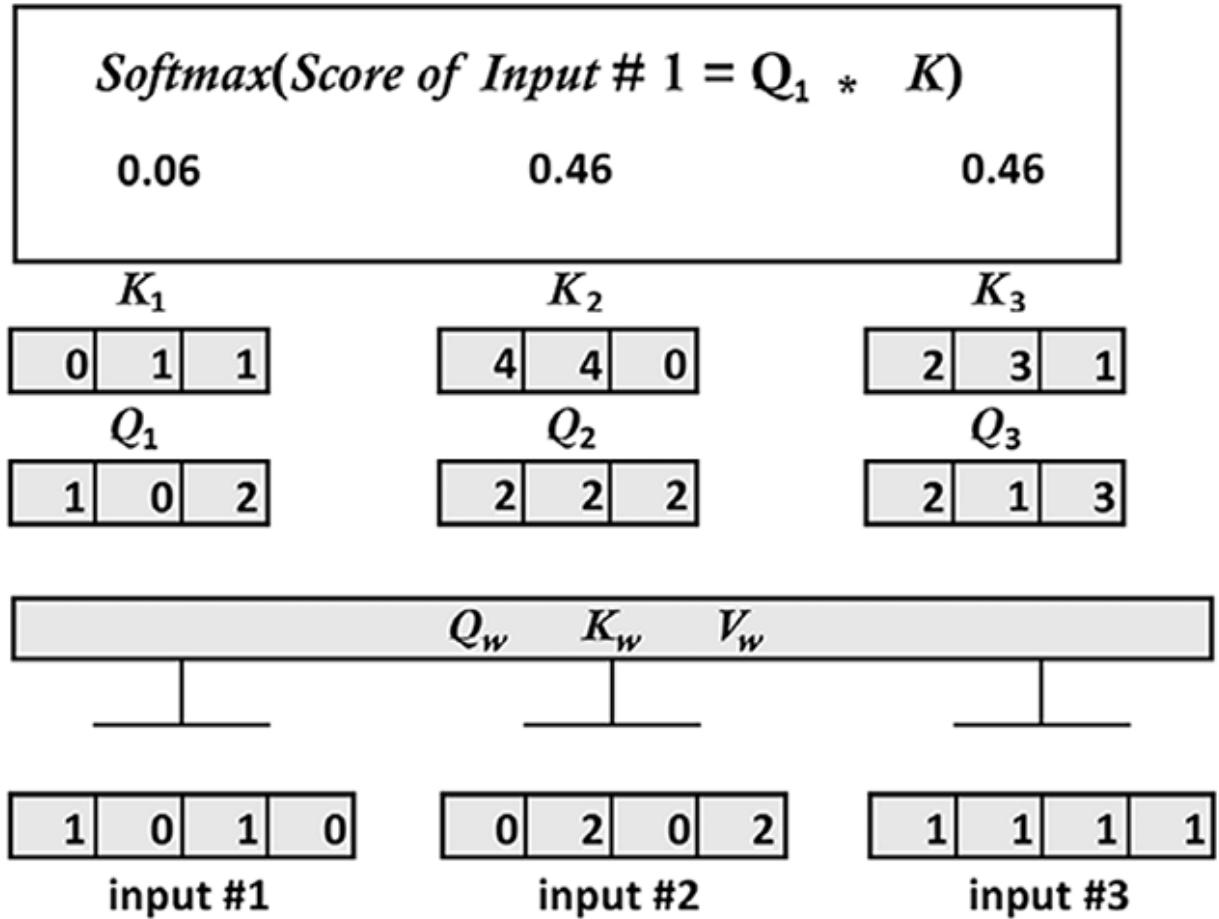


Figure 2.15: The softmax score of input #1 for all of the keys

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

## Step 6: The final attention representations

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

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

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  $\text{Attention}(Q, K, V)$  for  $x_1$  we multiply the intermediate attention score by the three value vectors one by one to zoom in on 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 and the three attention values for  $x_1$  for each input have been calculated:

<i>Attention(Q, K, V) for Input #1, <math>x_1</math></i>		
0.6   0.1   0.1	0.9   3.7   0	0.9   2.8   1.4

$V_1$	$V_2$	$V_3$
1   2   3	2   8   0	2   6   3

<i>Softmax(Score of Input # 1 = <math>Q_1 * K</math>)</i>		
0.06	0.46	0.46

$K_1$	$K_2$	$K_3$
0   1   1	4   4   0	2   3   1
$Q_1$	$Q_2$	$Q_3$

$Q_w$	$K_w$	$V_w$

1   0   1   0 input #1	0   2   0   2 input #2	1   1   1   1 input #3
---------------------------	---------------------------	---------------------------

Figure 2.16: Attention representations

The attention values now need to be summed up.

## Step 7: Summing up the results

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

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



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



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

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

<i>Sum of Attention (Q, K, V) value for Input #1, <math>x_1</math></i>
2.0 7.0 1.5

<i>Attention(Q, K, V) for Input #1, <math>x_1</math></i>
0.6 0.1 0.1      0.9 3.7 0      0.9 2.8 1.4

$V_1$	$V_2$	$V_3$
1 2 3	2 8 0	2 6 3

<i>Softmax(Score of Input # 1 = <math>Q_1 \cdot K</math>)</i>
0.06
0.46
0.46
$K_1$
0 1 1
$K_2$
4 4 0
$K_3$
2 3 1
$Q_1$
1 0 2
$Q_2$
2 2 2
$Q_3$
2 1 3
$Q_w \quad K_w \quad V_w$
1 0 1 0
input #1
0 2 0 2
input #2
1 1 1 1
input #3

Figure 2.17: Summed results for one input

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

## Step 8: Steps 1 to 7 for all the inputs

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

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

We have seen the attention representation process in detail with a small model. Let's go directly to the result and assume we have generated the three 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 shown).
#We assume we are implementing the original Transformer paper.
attention_head1=np.random.random((3, 64))
print(attention_head1)
```

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

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

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

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

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

We assume that we have trained the eight heads of the attention sublayer.

The Transformer now has three output vectors (of the three 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 at  
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
```

The output shows the shape of one of the heads:

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

The eight 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 eight elements of  $Z$  for the final output of the multi-head attention sublayer.

## Step 10: Concatenation of the output of the heads

The Transformer concatenates the eight elements of  $Z$ :

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

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

$Z_0$  to  $Z_7$  are concatenated:

```
print("Step 10: Concatenation of heads 1 to 8 to obtain the o
output_attention=np.hstack((z0h1,z1h2,z2h3,z3h4,z4h5,z5h6,z6h7
print(output_attention)
```

The output is the concatenation of  $Z$ :

```
Step 10: Concatenation of heads 1 to 8 to obtain the original 8
[[0.65218495 0.11961095 0.9555153 ... 0.48399266 0.80186221 0.
 [0.95510952 0.29918492 0.7010377 ... 0.20682832 0.4123836 0.
 [0.20211378 0.86541746 0.01557758 ... 0.69449636 0.02458972 0.
```

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

## *multi-headed attention layer output*

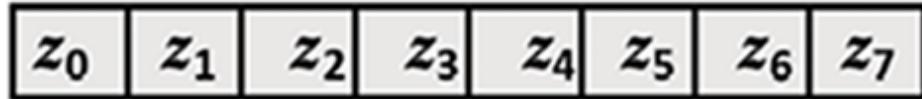


Figure 2.18: Attention sublayer output

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

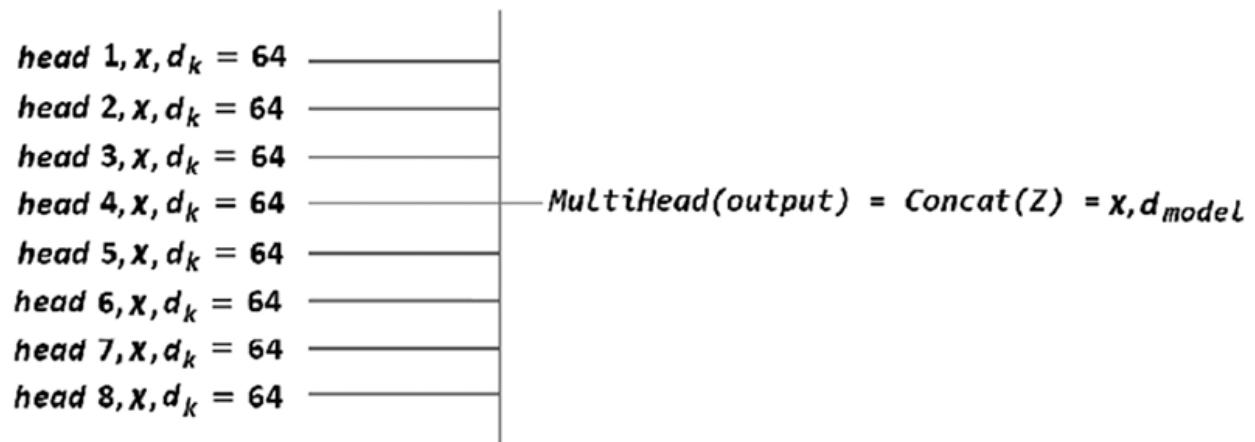
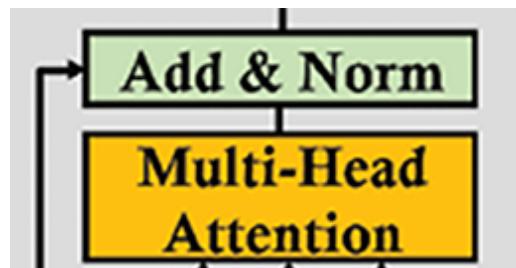


Figure 2.19: Concatenation of the output of the eight heads

Layer normalization will now process the attention sublayer.

## Post-layer normalization

Each attention sublayer and each feedforward sublayer of the Transformer is followed by **Post-Layer Normalization (Post-LN)**:



*Figure 2.20: Post-layer normalization*

The Post-LN contains an add function and a layer normalization process. The *add* function processes the residual connections that come from the input of the sublayer and are normalized. 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:

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

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

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

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

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

The variables are:

- $\mu$  is the mean of *v* of dimension *d*. As such:  $\mu = \frac{1}{d} \sum_{k=1}^d v_k$
- $\sigma$  is the standard deviation *v* of dimension *d*. As such:  $\sigma^2 = \frac{1}{d} \sum_{k=1}^d (v_k - \mu)^2$
- $\gamma$  is a scaling parameter.
- $\beta$  is a bias vector.

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

## Sublayer 2: Feedforward network

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

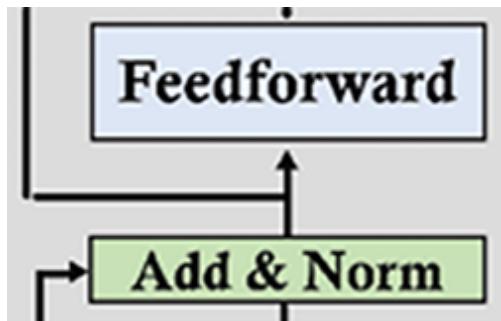


Figure 2.21: The feedforward sublayer

The FFN sublayer can be described as follows:

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

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

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

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

Let's now explore the decoder stack.

## The decoder stack

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

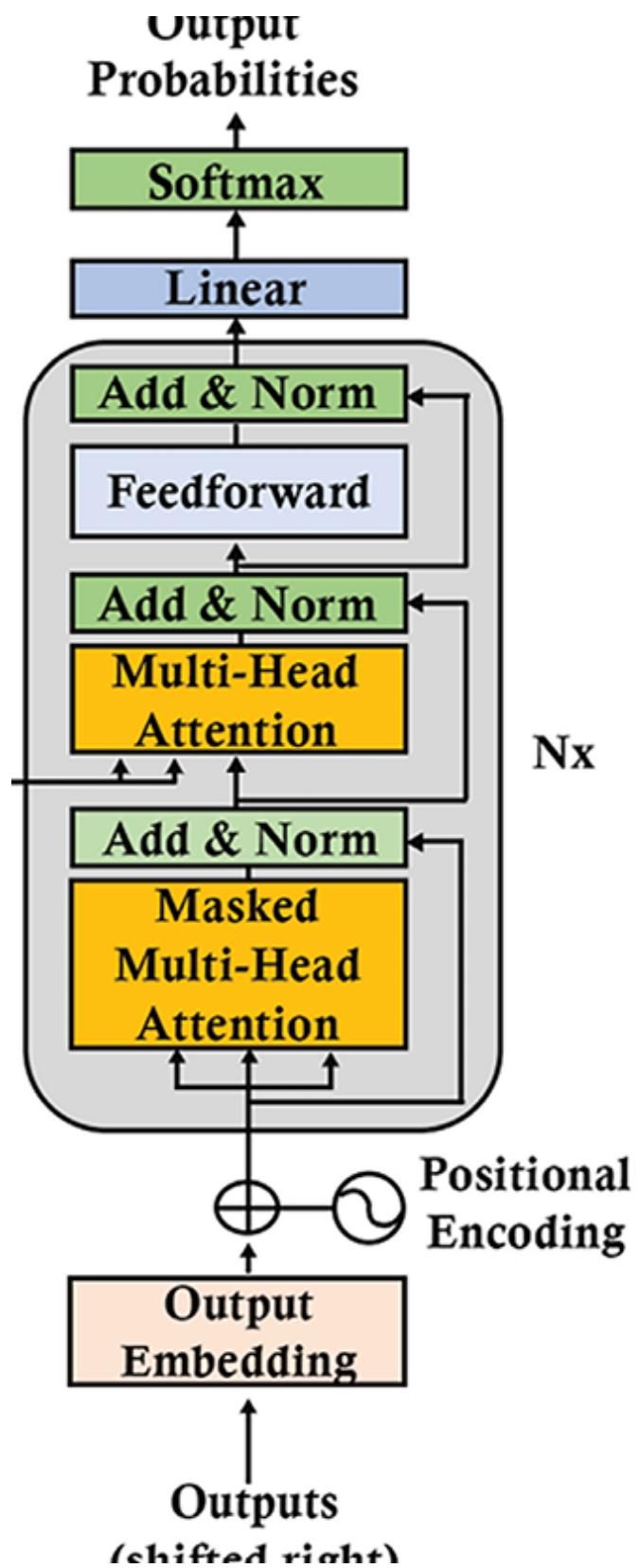


Figure 2.22: A layer of the decoder stack of the Transformer

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

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

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

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

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

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

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

## Output embedding and position encoding

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

function are the same as in the encoder stack.

In our exploration of Transformer usage, we will work with the model presented by *Vaswani* (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 marro
```

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

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

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

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

## The attention layers

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

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

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

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

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

$$\begin{aligned} \text{Input\_Attention} &= (\text{Output\_decoder\_sub\_layer -1 } (Q), \\ &\quad \text{Output\_encoder\_layer}(K, V)) \end{aligned}$$

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

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

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

The FFN sublayer has the same structure as the FFN of the encoder stack.

The post-layer normalization of the FFN works as the layer normalization of the encoder stack.

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

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

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

$$y = w * x + b$$

$w$  and  $b$  are learned parameters.

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

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

At the top layer of the decoder, the transformer will reach the output layer, which will map the outputs of the model to the size of the vocabulary to produce the raw logits of the prediction.

The raw logits of the output can go through a softmax function, apply the values obtained to the tokens in the vocabulary, and choose the best probable token for the task requested. Or, as shown in the *From one token to an AI revolution* section of *Chapter 1, What Are Transformers?*, the pipeline can apply sampling functions that will vary from one API to another.

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

## Training and performance

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

The datasets come from **Workshops on Machine Translation (WMT)**, which can be found at the following link if you wish to explore the WMT

datasets: <http://www.statmt.org/wmt14/>.

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

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

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

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

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

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

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

## Hugging Face transformer models

Everything we have learned in this chapter can be condensed into a ready-to-use Hugging Face transformer model.

With Hugging Face, we 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). Then, go to the two last cells.

We first ensure that Hugging Face transformers are installed:

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

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

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

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

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

And *voilà!* The translation is displayed:

```
[{'translation_text': 'Il est facile de traduire des langues à'}
```

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

## Summary

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

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

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

We saw that beyond removing recurrent network models, transformers introduce parallelized layers that reduce training time. In addition, 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 go more in depth into some aspects of the Transformer's architecture in the following chapters when describing the many variants of the original model.

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

Before continuing to the next chapter, make sure you capture the details of the paradigm shift constituted by the architecture of the Original Transformer. You will then be able to face any present and future transformer model.

In this chapter, we have dived into the architecture of the Original Transformer.. Now, we will see what they can do. In *Chapter 3, Emergent vs. Downstream Tasks: The Unseen Depths of Transformers*, we will explore the wide range of tasks transformer models can perform.

## Questions

1. NLP transduction can encode and decode text representations.  
(True/False)
2. **Natural Language Understanding (NLU)** is a subset of **Natural Language Processing (NLP)**. (True/False)
3. Language modeling algorithms generate probable sequences of words based on input sequences. (True/False)

4. A transformer is a customized LSTM with a CNN layer. (True/False)
5. A transformer does not contain LSTM or CNN layers. (True/False)
6. Attention examines all the tokens in a sequence, not just the last one. (True/False)
7. A transformer uses a positional vector, not positional encoding. (True/False)
8. A transformer contains a feedforward network. (True/False)
9. The masked multi-headed attention component of the decoder of a transformer prevents the algorithm parsing a given position from seeing the rest of a sequence that is being processed. (True/False)
10. Transformers can analyze long-distance dependencies better than LSTMs. (True/False)

## References

- *Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, Illia Polosukhin, 2017, Attention Is All You Need:*  
<https://arxiv.org/abs/1706.03762>
- Hugging Face transformer usage:  
<https://huggingface.co/docs/transformers/main/en/quicktour>
- **Tensor2Tensor (T2T)** introduction:  
[https://colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello\\_t2t.ipynb?hl=en](https://colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb?hl=en)
- Manuel Romero's notebook with link to explanations by *Raimi Karim*:  
<https://colab.research.google.com/drive/1rPk3ohr>

[mVc1qhH7uQ7qys4oznDdAhpzF](#)

- Google language research:  
<https://research.googleteams.google/language/>
- Hugging Face research:  
<https://huggingface.co/transformers/index.html>
- *The Annotated Transformer*:  
<http://nlp.seas.harvard.edu/2018/04/03/attention.html>
- *Jay Alammar, The Illustrated Transformer*:  
<http://jalammar.github.io/illustrated-transformer/>

## Further reading

Transformers could not have increased their potential without hardware innovation. NVIDIA, for example, offers interesting insights on transformers (<https://blogs.nvidia.com/blog/2022/03/25/what-is-a-transformer-model/>) and the related hardware (<https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/index.html>).

## Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://www.packt.link/Transformers>



# 3

## Emergent vs Downstream Tasks: The Unseen Depths of Transformers

Transformers reveal their full potential when we unleash pretrained models and watch them perform downstream **Natural Language Understanding (NLU)** tasks. It takes a lot of time and effort to pretrain and fine-tune a transformer model, but the effort is worthwhile when we see a multi-billion-parameter transformer model in action on a range of NLU tasks.

Advanced NLP models have achieved the quest of outperforming the human baseline. The human baseline represents the performance of humans on NLU tasks. 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. Yet, machine intelligence has now surpassed basic human baselines.

We will first bridge the gap between the functional and mathematical architecture of transformers we went through in *Chapter 2, Getting Started with the Architecture of the Transformer Model*, by introducing *emergence*. All machine models possess emergence abilities and the capacity to learn

new things independently, but a transformer's unique architecture has taken language understanding to another level.

We will then see how to measure the performance of transformers.

Measuring **Natural Language Processing (NLP)** task performance remains a straightforward approach involving accuracy scores in various forms based on true and false results. These results are obtained through benchmark tasks and datasets. SuperGLUE, for example, is a wonderful example of how Google DeepMind, Facebook AI, the University of New York, the University of Washington, and others worked together to set high standards to measure NLP performances.

Finally, we will explore several downstream tasks, such as the **Standard Sentiment TreeBank (SST-2)**, linguistic acceptability, and Winograd schemas.

Transformers are rapidly taking NLP to the next level by outperforming other models on well-designed benchmark tasks. Alternative transformer architectures will continue to emerge and evolve.

This chapter covers the following topics:

- How transformer attention heads produce outputs
- Measuring transformer performance versus Human Baselines
- Measurement methods (Accuracy, F1-score, and MCC)
- Benchmark tasks and datasets
- SuperGLUE downstream tasks
- Linguistic acceptability with CoLA
- Sentiment analysis with SST-2
- Winograd schemas



With all the innovations and library updates in this cutting-edge field, packages and models change regularly. Please go to the GitHub repository for the latest installation and code examples:

<https://github.com/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition/tree/main/Chapter03>.

You can also post a message in our Discord community (<https://www.packt.link/Transformers>) if you have trouble running the code in this or any chapter.

Let's start by understanding how machines represent language.

## The paradigm shift: What is an NLP task?

ChatGPT stunned the world when it suddenly became mainstream in late 2022 and early 2023. An AI could generate human-like text on practically any topic. Thousands of tasks were submitted to this incredible Generative AI transformer. ChatGPT Plus with GPT-4 seemed to be able to perform any task an end user came up with.

However, OpenAI couldn't have possibly pretrained ChatGPT on thousands of tasks that could not be guessed beforehand. Nor could OpenAI have possibly fine-tuned its GPT models for everything the end user was coming up with.

Of course, a transformer model can be trained for specific tasks and determined downstream tasks such as summarizing. However, models such as ChatGPT can perform downstream tasks for which they were not trained.

This section takes us inside the head of a transformer model to see how the architecture described in *Chapter 2, Getting Started with the Architecture of the Transformer Model*, applies to downstream tasks. In *Chapter 2*, we went through the architecture of the Original Transformer.

We specifically went through attention heads starting with the section

*Sublayer 1: Multi-head attention:*

1. The scores displayed are the result of the  $Q * K$  matrix multiplication.
2. These raw scores are then scaled by dividing them by the square root of the dimension of the key vectors.
3. Then, a softmax function is applied to the raw scores, to sum up to 1.

If necessary, take your time to go through *Chapter 2* again.

How can these mathematical functions become the incredible outputs we obtain when we dialog with ChatGPT, for example?

To understand how this apparent miracle happened, we must dive into the unseen depths of transformers.

## Inside the head of the attention sublayer of a transformer

This section aims to bridge the gap between understanding the mathematics of the architecture and how the model produces tokens (pieces of words or words) that become sequences of words.



In this section, the scores displayed are the output of the attention heads after the softmax but before the probabilities create a weighted sum of the value **vectors (V)**.

We will go down to the code level in *Chapter 9, Shattering the Black Box with Interpretable Tools*, including `BertViz_Interactive.ipynb`, used for this section to illustrate how numbers become words.

You do not need to run any code but you do need to grasp how we go from mathematical probabilities to words. The key focus here is understanding the quantum leap from seemingly meaningless numbers to organized language.

We will see how meaning *emerges* as a transformer model learns more and more language sequences.

We will now examine the activity of attention heads through the following sentence to see the inner workings of the transformer:

*Transformers possess surprising emerging features.*

The sequence runs through a transformer model's sublayer layers and reaches the attention heads as described in *Chapter 2*, in the section *Sublayer 1: Multi-head attention*.

We will look at the outputs of the heads obtained without going into the mathematical calculations of the attention heads. We will not look at the code either. This can wait until *Chapter 9*. We will focus on how the Original Transformer was designed to produce its disruptive outputs just like the original designers did.

Once we bridge the gap between numbers and words, we will have the understanding required to run the notebooks throughout the book.

Let's focus on the outputs of the attention head.

First, we will look into layer 1 and attention head 1:

```
selected_layer = 1  
selected_head = 1
```

The transformer analysis tool displays what the head is `thinking`:

	<b>transformers</b>	<b>possess</b>	<b>surprising</b>	<b>emerging</b>	<b>features</b>
<b>transformers</b>	0.793087	0.037714	0.027054	0.020088	0.016091
<b>possess</b>	0.383139	0.012022	0.530375	0.007978	0.001052
<b>surprising</b>	0.846634	0.000771	0.026575	0.083948	0.007271
<b>emerging</b>	0.677940	0.001957	0.003967	0.010203	0.155751
<b>features</b>	0.432853	0.000938	0.001393	0.000413	0.000555

*Figure 3.1: Probability table for word pairs for attention head 1 in layer 1*

Each row of *Figure 3.1* contains a word in the sentence. Each column also includes each word in the sentence. The intersection represents the probability that the two words are related.

You can see that what we call `learning` or `training` is the probability that one word is related to another in a given context (sequence).

For this experiment, we will focus on the relationship between the words `transformers` and `possess`:

- `transformers`: The attention head's "mind" is running at full speed. It is looking at the relationship between every word and every other word. It finds a high score for the `transformers`-`transformers` pair (0.79). This

score is not very interesting because a word is obviously related to itself.

- `possess`: This score between the words `transformers` and `possess` is (0.038). That's not very interesting yet, either.

Now, let's look at another head in layer 1:

```
selected_layer = 1  
selected_head = 6
```

The outputs are as follows:

	<code>transformers</code>	<code>possess</code>	<code>surprising</code>	<code>emerging</code>	<code>features</code>
<code>transformers</code>	0.702030	0.122793	0.018648	0.075400	0.032200
<code>possess</code>	0.613370	0.102932	0.043511	0.169905	0.056358
<code>surprising</code>	0.522166	0.312261	0.021874	0.032769	0.079888
<code>emerging</code>	0.553818	0.116540	0.008703	0.031475	0.014774
<code>features</code>	0.713933	0.111162	0.012219	0.022824	0.011651

*Figure 3.2: Probability table for word pairs for attention head 6 in layer 1*

Head 6 creates different relationships.

- `transformers`: It finds a high score for the `transformers`-`transformers` pair but lower than head 1 (0.70).
- `possess`: This score between the words `transformers` and `possess` is higher than in head 1 (0.12).

Now, let's jump to layer 6, head 1:

```
selected_layer = 6  
selected_head = 1
```

The outputs are as follows:

	transformers	possess	surprising	emerging	features
transformers	0.031646	0.188067	0.538230	0.009978	0.026219
possess	0.036634	0.034815	0.069137	0.043544	0.027010
surprising	0.112891	0.512405	0.114949	0.027858	0.026199
emerging	0.031653	0.086987	0.033904	0.032473	0.031259
features	0.014392	0.043559	0.029788	0.020789	0.015035

Figure 3.3: Probability table for word pairs for attention head 1 in layer 6

Head 1 is beginning to see the light:

- `transformers`: It finds a lower score for the `transformers-transformers` pair than layer 1 (0.03). It is learning that it must associate with words other than itself.
- `possess`: This score between the word `transformers` and `possess` is higher than at layer 1 (0.188).

Notice the high score of the word `surprising` is 0.53.

Finally, let's look into head layer 6, head 6:

```
selected_layer = 6  
selected_head = 6
```

The output is as follows:

	transformers	possess	surprising	emerging	features
transformers	0.009803	0.200566	0.004300	0.007951	0.007760
possess	0.005146	0.007327	0.122820	0.004412	0.001436
surprising	0.007802	0.016328	0.028438	0.397363	0.174339
emerging	0.001280	0.000809	0.001532	0.029902	0.693638
features	0.018621	0.001600	0.002361	0.009339	0.020869

Figure 3.4: Probability table for word pairs for attention head 6 in layer 6

Head 6 has its own views:

- `transformers`: It finds a lower score than head 1 for the `transformers-transformers` pair (0.098).
- `possess`: This score between the word `transformers` and `possess` is higher than at head 1 (0.20).

Notice the high score of the word `surprising` (0.004) is lower than `possess` and lower than for head 1.

The training is not over yet, but we can draw some insights:

- In layer 6, head 6, the model progressively finds that it should choose the word `possess`. A noun + verb is a good choice. The training is not over, but it's a good beginning.
- Suppose the training was on a masked word and that the transformer model had to find the missing word in this sentence:

Transformers \_\_\_\_\_ surprising emerging features.

The model found an emerging pattern and did learn to choose the verb possess.

The model is thus *emergent* like many machine learning algorithms.

- The extra magic that gives a whole new meaning to *emergence* of the transformers model is that it is learning an unlimited amount of additional information!

If we look at layer 6 and head 6 for the word `transformer`, we see:

	<code>transformers</code>	<code>possess</code>	<code>surprising</code>	<code>emerging</code>	<code>features</code>
<code>transformers</code>	0.009803	0.200566	0.004300	0.007951	0.007760

Figure 3.5: Probability table for word pairs for “`transformers`” for attention head 6 in layer 6

If we sort them, omitting `transformers`, since it’s the first word we are examining, see that `transformers` will choose:

"`transformers possess emerging features (surprising)`."

This is a good progression because we are just in the middle of the training process. If the goal was to find `possess`, we could say that far more information is *emerging*. Imagine when the model is learning millions of relationships with billions of parameters!

We can now see how a transformer goes from numbers representing tokens (pieces of words or words) to intelligible sequences.

Let’s see emergence in action with ChatGPT.

## Exploring emergence with ChatGPT

Programming languages are languages, also.

If we repeated the process of looking into the “head” of a transformer for source code, the transformer would also be *emergent*. The model would behave exactly like any other language sequence and produce scores for each layer and head.

After being trained on the source code of GitHub, as OpenAI GPT models were, it could produce lines of code. Also, since it understands language very well, it could comment on the code.

That’s precisely what happened and what OpenAI did!

Open `From_training_to_emergence.ipynb` in `Chapter03` of the GitHub repository of this book.

The program was written, commented on, and explained by ChatGPT Plus with GPT-4 only on <https://openai.com/>.

I wrote the instructions:

Denis:

Write a k-nearest neighbor program in Python that uses 1000 ra

ChatGPT Plus with GPT-4 wrote some preliminary explanation as shown in the following excerpt:

... The k-NN neighbors boundary is not a single, well-defined

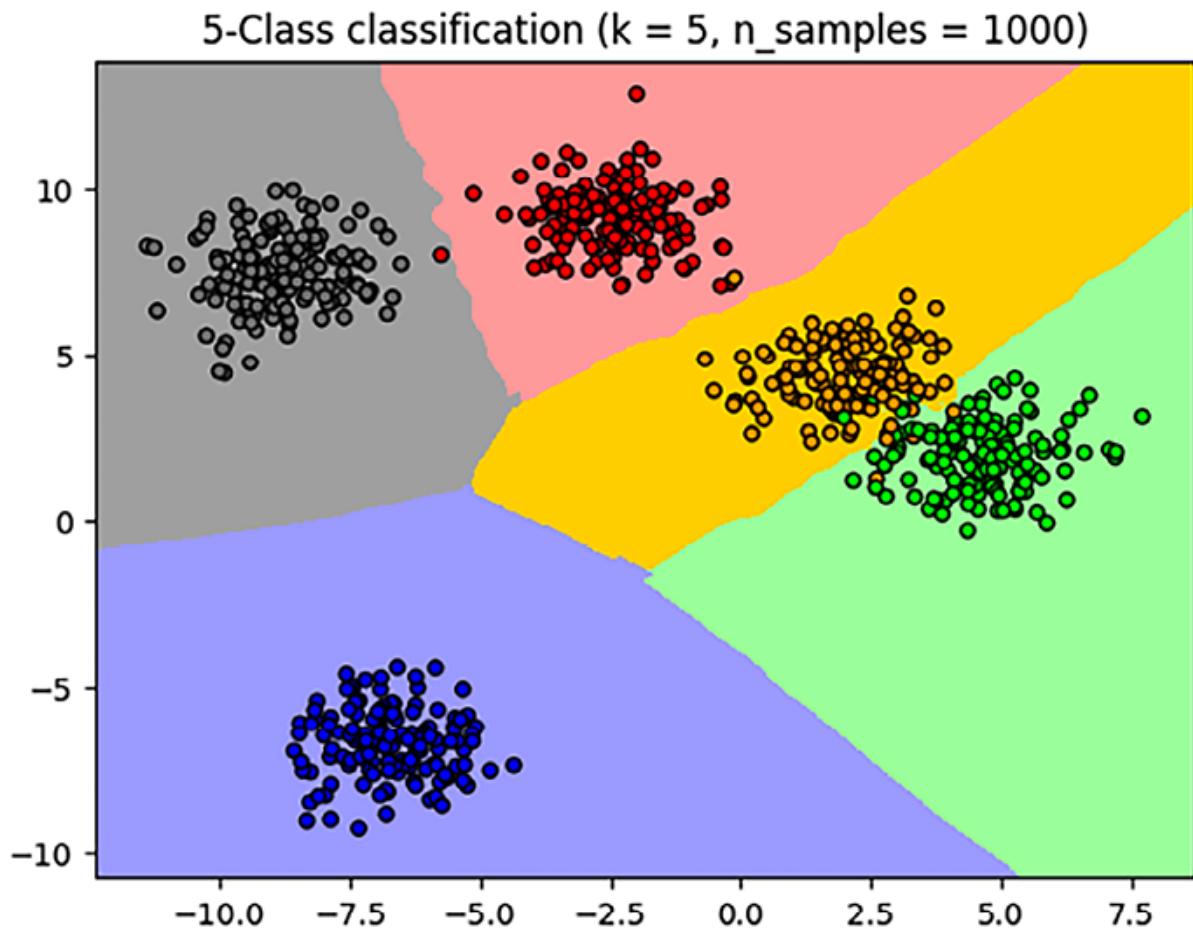
Then ChatGPT Plus with GPT-4 wrote the code:

```

!pip install scikit-learn
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from matplotlib.colors import ListedColormap
# Generate 1000 random data points with 5 classes
X, y = make_blobs(n_samples=1000, centers=5, random_state=42)
# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=42)
# Create a k-NN classifier with k=5
knn = KNeighborsClassifier(n_neighbors=5)
# Train the classifier with the training data
knn.fit(X_train, y_train)
# Create a mesh to plot the decision boundaries
h = .02 # step size in the mesh
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
# Predict the mesh points' class labels
Z = knn.predict(np.c_[xx.ravel(), yy.ravel()])
# Put the result into a color plot
Z = Z.reshape(xx.shape)
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF',
                            '#FF0000', '#00FF00', '#0000FF'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF',
                           '#00FFFF', '#000000', '#008000'])
plt.figure()
plt.pcolormesh(xx, yy, Z, cmap=cmap_light)
# Plot the training points
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cmap_light)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.title("5-Class classification (k = 5, n_samples = 1000)")
plt.show()

```

I copied the program into Google Colab and obtained an excellent plot:



*Figure 3.6: A decision boundary plot created by ChatGPT Plus with GPT-4*

We can draw insights from this experiment:

- Natural language modeling trains a model to predict words or sequences by considering the context.
- **NLU** takes the model into the depths of the intents and meanings of human language.

We could go on like this, having conversations with ChatGPT Plus with GPT-4 and producing an unlimited amount of documented and commented source code in a variety of languages.

We are now ready to tackle downstream tasks.

# **Investigating the potential of downstream tasks**

Transformers, like humans, can be fine-tuned to perform downstream tasks by inheriting the properties of a pretrained model. The pretrained model provides its architecture and language representations through its parameters.

A pretrained model trains on key tasks to acquire a general knowledge of the language. A fine-tuned model trains on downstream tasks. Not every transformer model uses the same tasks for pretraining. But, potentially, all tasks can be pretrained or fine-tuned.

Organizing downstream tasks provides a scientific framework for implementing and measuring NLP. However, every NLP model needs to be evaluated with a standard method.

This section will first go through some of the key measurement methods. Then, we will go through some of the main benchmark tasks and datasets.

Let's start by going through some of the key metric methods.

## **Evaluating models with metrics**

It is impossible to compare one transformer model to another (or any other NLP model) without a universal measurement system that uses metrics.

In this section, we will analyze some of the measurement scoring methods.

### **Accuracy score**

In whatever variant you use, the accuracy score is a practical evaluation. The score function calculates an exact true or false value for each result. Either the model's outputs,  $\hat{y}$ , match the correct predictions,  $y$ , for a given subset,  $samples_i$ , of a set of samples or not. The primary function is:

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

We will obtain  $1$  if the result for the subset is correct and  $0$  if it is false.

Let's now examine the more flexible F1-score.

## F1-score

The F1-score introduces a more flexible approach that can help when faced with datasets containing uneven class distributions.

In this equation, true ( $T$ ) positives ( $p$ ), false ( $F$ ) positives ( $p$ ), and false ( $F$ ) negatives ( $n$ ) are first plugged into the precision ( $P$ ) and recall ( $R$ ) equations:

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

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

Then the F1-score uses precision ( $P$ ) and recall ( $R$ ) to build the metric:

$$F1 - score = 2 * (precision * recall) / (precision + recall)$$

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

$$F1-score = 2 \times \frac{P \times R}{P + R}$$

Let's now review the **Matthews Correlation Coefficient (MCC)** approach.

## MCC

The accuracy score and F1-score are ways to measure the output of a transformer model objectively. The MCC provides another measurement tool. These tools help us compare the same model with different configurations. They also help us rank transformer models.

The MCC will be implemented in the *Evaluating using Matthews correlation coefficient* section in *Chapter 5, Diving into Fine-Tuning through BERT*. The MCC computes a measurement with true positives (*TPs*), true negatives (*TNs*), false positives (*FPs*), and false negatives (*FNs*).

The MCC can be summarized by the following equation:

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

The MCC provides an excellent metric for binary classification models, even if the sizes of the classes are different.

We now have a good idea of how to measure a given transformer model's results and compare them to other transformer models or NLP models.

With some of the scoring methods in mind, we will now see how human evaluation can contribute to valuable measurements.

## Human evaluation

Human evaluations can apply to smaller datasets or subsets of larger datasets.

Human evaluations can be effective for:

- Analyzing the outputs of a model during the design and training process.
- Providing feedback once the model is in production.
- Creating datasets of samples for a given task.
- Creating new benchmark tasks and methods.

With measurement scoring methods in mind, let's look into benchmark tasks and datasets.

## Benchmark tasks and datasets

Three prerequisites are required to prove that transformers have reached state-of-the-art performance levels:

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

There are many benchmarks on the market, such as Google's BIG-bench (<https://github.com/google/BIG-bench>), which we will use to explore self-evaluations in *Chapter 15, Guarding the Giants: Mitigating Risks in Large Language Models*, in *The emergence of functional AGI* section.

However, it makes no sense to implement all the benchmarks. It is essential to run enough tasks to understand the potential of NLP tasks with transformers. From there, you will quickly adapt to any benchmarking system.

We will begin by exploring the SuperGLUE benchmark to illustrate the evaluation process of a transformer model through some of the main NLP tasks.

## From GLUE to SuperGLUE

Many NLP benchmarks can be applied to measure a model's performance, such as GLUE, SuperGLUE, and **WMT (Workshop on Machine Translation)**. We will use WMT in *Chapter 4, Advancements in Translations with Google Trax, Google Translate, and Gemini*.

This chapter focuses on the SuperGLUE benchmark. The SuperGLUE benchmark was designed and made public by Wang *et al.* (2019). Wang *et al.* (2019) first designed the **General Language Understanding Evaluation (GLUE)** benchmark.

The motivation of the GLUE benchmark was to show that to be useful, NLU has to be applicable to a wide range of tasks. Relatively small GLUE datasets were designed to encourage an NLU model to solve a set of tasks.

However, the performance of NLU models, boosted by the arrival of transformers, began to exceed the average level. The GLUE leaderboard, available at <https://gluebenchmark.com/leaderboard>, shows a remarkable display of NLU talent, mainly concentrating on the ground-breaking transformer models. The leaderboard makes less sense than understanding and applying the benchmark tasks to a model to see if it fits our needs. We can enhance the datasets for specific projects. Human baselines are far down the list:

21	Facebook AI	RoBERTa
22	Microsoft D365 AI & MSR AI	MT-DNN-ensemble
23	GLUE Human Baselines	GLUE Human Baselines

Figure 3.7: GLUE leaderboard November 2023

New models and the Human Baselines ranking will constantly change. The position of Human Baselines doesn't mean much anymore since the arrival of Foundation Models. These rankings just show how far classical NLP and transformers have taken us!

GLUE Human Baselines are not in a top position, which shows that NLU models have surpassed non-expert humans on GLUE tasks. Human Baselines represent what we humans can achieve. AI can now outperform humans. However, it is challenging to blindly fish around for benchmark datasets to improve our models without a standard to refer to.

We also notice that transformer models have taken the lead in the 2020s. This trend will continue, although we never know what creative innovators will find!

I like to think of GLUE and SuperGLUE as the point when words go from chaos to order with language understanding. Understanding is the “glue” that makes words fit together and become a language.

The GLUE leaderboard will continuously evolve as NLU progresses. However, *Wang et al.* (2019) introduced SuperGLUE to set a higher standard for Human Baselines.

## Introducing higher Human Baselines standards

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

SuperGLUE helped improve the position of Human Baselines as shown in the following excerpt of the leaderboard

(<https://super.gluebenchmark.com/leaderboard>):

6	Zirui Wang	T5 + UDG, Single Model (Google Brain)
7	DeBERTa Team - Microsoft	DeBERTa / TuringNLRv4
8	SuperGLUE Human Baselines	SuperGLUE Human Baselines

*Figure 3.8: SuperGLUE leaderboard 2.0 – November 2023*

The SuperGLUE leaderboard keeps evolving, and AI will continue to push human baselines of basic NLP tasks down. The top models are not displayed in *Figure 3.5* for two reasons:

- The figure does not include some of the top proprietary Foundation Models.
- The ranking changes constantly with models specialized in SuperGLUE tasks, which are an excellent first-level assessment of a model.  
However, you will need to evaluate models according to your needs to make a choice.

So yes, AI algorithm rankings will constantly change as new innovative models arrive. However, these rankings just show how hard the battle for NLP supremacy is being fought!

The SuperGLUE evaluation process is more important than the rankings that will keep changing.



The best model for you is not necessarily the best model on a leaderboard. You need to learn an NLP evaluation process and apply it to the model(s) you choose to implement.

Let's now see how the evaluation process works.

## The SuperGLUE evaluation process

*Wang et al.* (2019) selected practical representative tasks of NLP for their SuperGLUE benchmark. The selection criteria for these tasks were stricter than for GLUE. For example, the tasks had to not only understand texts but also 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 main 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
Recognising 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 3.9: SuperGLUE tasks*

The task list is interactive:

<https://super.gluebenchmark.com/tasks>.

Each task contains links to the required information to perform that task:

- **Name** is the name of the downstream task of a fine-tuned, pretrained model.
- **Identifier** is the abbreviation or short version of the name.
- **Download** is the download link to the datasets.

- **More Info** offers greater detail through a link to the paper or website of the team that designed the dataset-driven task(s).
- **Metric** is the measurement score used to evaluate the model.

SuperGLUE provides the task instructions, the software, the datasets, and papers or websites describing the problem to be solved. Once a team runs the benchmark tasks and reaches the leaderboard, the results are displayed:

Score	BoolQ	CB	COPA	MultiRC	ReCoRD	RTE	WiC	WSC	AX-b	AX-g
89.8	89.0	95.8/98.9	100.0	81.8/51.9	91.7/91.3	93.6	80.0	100.0	76.6	99.3/99.7

Figure 3.10: 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 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 common sense 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.

COPA testing opens the door to advanced question-and-answer tasks, as we will see in *Chapter 12*.

We have introduced COPA. Let's define some of the other SuperGLUE benchmark tasks.

## Defining the SuperGLUE benchmark tasks

A task can be a pretraining task to generate a trained model. That same task can be a downstream task for another model that will fine-tune it. However, the goal of SuperGLUE is to show that a given NLU model can perform multiple downstream tasks with fine-tuning. Multi-task models are the ones that prove the thinking power of transformers.

The power of any transformer resides in its ability to perform multiple tasks using a pretrained model and then apply it to fine-tuned downstream tasks. The Original Transformer model and its variants are now widely present in the top rankings for all the GLUE and SuperGLUE tasks. We will continue to focus on SuperGLUE downstream tasks for which Human Baselines are tough to beat.

In the previous section, we went through COPA. In this section, we will go through seven 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 essential  
"passage": "Windows Movie Maker -- Windows Movie Maker (former
```

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* and 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 natural discourses.

The following sample, #77, taken from the `train.jsonl` training dataset, shows how difficult the CB task is:

```
{"premise": "The Susweca. It means ''dragonfly'' in Sioux, you  
"hypothesis": "Susweca is where she and Paul met,"  
"label": "entailment", "idx": 77}
```

We will now have a look at the multi-sentence problem.

## Multi-Sentence Reading Comprehension (MultiRC)

**Multi-Sentence Reading Comprehension (MultiRC)** asks the model to read a text and choose from several possible choices. The task is difficult for both humans and machines. The model is presented with a *text*, several *questions*, and possible *answers* to each question with a `0` (false) or `1` (true) *label*.

Let's take the second sample in `train.jsonl`:

```
"Text": "text": "The rally took place on October 17, the shoot.
```

The sample contains four questions. To illustrate the task, we will investigate two of them. The model must predict the correct labels. Notice how the information that the model is asked to obtain is distributed throughout the text:

```
"question": "When was Kayla Rolland shot?"  
"answers":  
[{"text": "February 17", "idx": 168, "label": 0},  
 {"text": "February 29", "idx": 169, "label": 1},  
 {"text": "October 29", "idx": 170, "label": 0},  
 {"text": "October 17", "idx": 171, "label": 0},  
 {"text": "February 17", "idx": 172, "label": 0}], "idx": 26},  
 {"question": "Who was president of the NRA on February 29?",  
 "answers": [{"text": "Charleton Heston", "idx": 173, "label": 0},  
 {"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", "i
```

At this point, one can only admire the performance of a fine-tuned, pretrained model on these difficult downstream tasks.

Now, let's see the reading comprehension task.

## Reading Comprehension with Commonsense Reasoning Dataset (ReCoRD)

**ReCoRD** represents another challenging task. The dataset contains over 120,000 queries from more than 70,000 news articles. The Transformer must use common sense reasoning to solve this problem.

Let's examine a sample from `train.jsonl`:

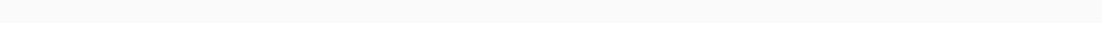
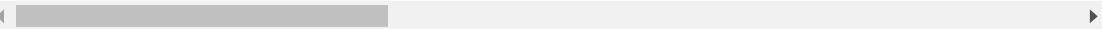
```
"source": "Daily mail"  
A passage contains the text and indications as to where the en  
A passage begins with the text:  
"passage": {  
    "text": "A Peruvian tribe once revered by the Inca's for t
```

The *entities* are indicated, as shown in the following excerpt:

```
"entities": [{"start": 2,"end": 9}, ...,"start": 711,"end": 711}
```

Finally, the model must *answer* a *query* by finding the proper value for the *placeholder*:

```
{"query": "Innocence of youth: Many of the @placeholder's young",  
"answers": [{"start":263, "end":271, "text": "Ashaninka"}, {"start":
```



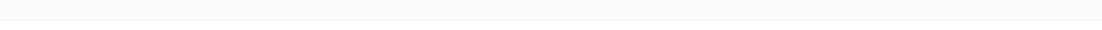
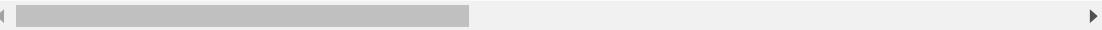
Once the transformer model has gone through this problem, it must now face an entailment task.

## Recognizing Textual Entailment (RTE)

For **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
```



RTE requires understanding and logic. Let's now see the **Words in Context (WiC)** task.

## WiC

**WiC** and the following Winograd task test a model's ability to process an ambiguous word. In WiC, the multi-task Transformer will have to analyze two sentences and determine whether the target word has the same meaning in both sentences.

Let's examine the first sample of the `train.jsonl` dataset.

First, the target word is specified:

```
"word": "place"
```

The model has to read two sentences containing the target word:

```
"sentence1": "Do you want to come over to my place later?",  
"sentence2": "A political system with no place for the less
```

`train.jsonl` specifies the sample index, the value of the label, and the position of the target word in `sentence1(start1, end1)` and `sentence2(start2, end2)`:

```
"idx": 0,  
"label": false,  
"start1": 31,  
"start2": 27,  
"end1": 36,  
"end2": 32,
```

After this daunting task, the transformer model has to face the Winograd task.

## The Winograd Schema Challenge (WSC)

The Winograd schema task is named after Terry Winograd. If a transformer is well trained, it should be able to solve disambiguation problems.

The dataset contains sentences that target slight differences in the gender of a pronoun.

This constitutes a coreference resolution problem, which is one of the most challenging tasks to perform. However, the transformer architecture, which

allows self-attention, is ideal for this task.

Each sentence contains an *occupation*, a *participant*, and a *pronoun*. The problem is to find whether the pronoun is *coreferent* with the occupation or the participant.

Let's examine a sample taken from `train.jsonl`.

First, the sample asks the model to read a *text*:

```
{"text": >"I poured water from the bottle into the cup until i  
The WSC ask the model to find the target pronoun token number :  
"target": {"span2_index": 10,
```

Then, it asks the model to determine if `it` refers to `the cup` or not:

```
"span1_index": 7,  
"span1_text": "the cup",  
"span2_text": "it"},  
For sample index #4, the label is true:  
"idx": 4, "label": true}
```

We have gone through some of the main SuperGLUE tasks. There are many other tasks.

However, once you understand the transformers' architecture and the benchmark tasks' mechanism, you will rapidly adapt to any model and benchmark.

Let's now run some downstream tasks.

## Running downstream tasks

In this section, we will 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. We will be going through variants of these models during our journey in the book. Once you understand the process of running a few tasks, you will quickly understand all of them. *After all, the human baseline for all these tasks is us!*

A downstream task is a fine-tuned transformer task that inherits the model and parameters from a pretrained transformer model.

A downstream task is thus the perspective of a pretrained model running fine-tuned tasks. That means, depending on the model, a task is downstream if it was not used to fully pretrain the model. In this section, we will consider all the tasks downstream since we did not pretrain them.

Models will evolve, as will databases, benchmark methods, accuracy measurement methods, and leaderboard criteria. However, 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)

CoLA, a GLUE task (<https://gluebenchmark.com/tasks>), contains thousands of samples of English sentences annotated for grammatical acceptability.

The goal of *Alex Warstadt et al.* (2019) was to evaluate the linguistic competence of an NLP model to judge the linguistic acceptability of a sentence. The NLP model is expected to classify the sentences accordingly.

The sentences are labeled as grammatical or ungrammatical. The sentence is labeled `0` if the sentence is not grammatically acceptable. The sentence is labeled `1` if the sentence is grammatically acceptable. For example:

Classification = `1` for *we yelled ourselves hoarse.*

Classification = `0` for *we yelled ourselves.*

You can go through `BERT_Fine_Tuning_Sentence_Classification_GPU.ipynb` in *Chapter 5, Diving into Fine-Tuning through BERT*, to view the BERT model that we fine-tuned on CoLA datasets. We used CoLA data.

We will first load the dataset:

```
#source of dataset : https://nyu-mll.github.io/CoLA/
df = pd.read_csv("in_domain_train.tsv", delimiter='\t', header=0)
df.shape
```

We will also load a pretrained Hugging Face BERT model:

```
model = BertForSequenceClassification.from_pretrained("bert-base-cased")
```

Finally, the measurement method, or metric, we used is MCC, which was described in the *MCC* section of 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)

SST-2 contains movie reviews. This section 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.

In this section, we will run a sample taken from SST on a Hugging Face transformer pipeline model to illustrate binary classification.

Open `Transformer_tasks.ipynb` and run the following cell, which contains positive and negative movie reviews taken from SST-2 for binary classification:

```
from transformers import pipeline
nlp = pipeline("sentiment-analysis", model="distilbert-base-un
print(nlp("If you sometimes like to go to the movies to have f
print(nlp("Effective but too-tepid biopic."), "Effective but to
```

The output is accurate:

```
[{'label': 'POSITIVE', 'score': 0.9998257756233215}] If you som
[{'label': 'NEGATIVE', 'score': 0.9974064230918884}] Effective
```

The SST-2 task is evaluated using the accuracy metric.

We classify sentiments of a sequence. Let's now see whether two sentences in a sequence are paraphrases or not.

# Microsoft Research Paraphrase Corpus (MRPC)

MRPC, a GLUE task, contains pairs of sentences extracted from new sources on the web. A human has annotated each pair to indicate whether the sentences are equivalent based on two closely related properties:

- Paraphrase equivalent
- Semantic equivalent

Let's run a sample using the Hugging Face BERT model. Open `Transformer_tasks_with_Hugging_Face.ipynb`, go to the following cell, and then run the sample taken from MRPC:

```
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)
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:

```
The DVD CCA appealed that decision to the U.S. Supreme Court. should be a paraphrase
not paraphrase: 8%
is paraphrase: 92%
```

---

The MRPC task is measured with the F1/accuracy score method.

Let's now run a Winograd schema.

## Winograd schemas

We described the Winograd schemas in this chapter's *The Winograd Schema Challenge (WSC)* section. The training set was in English. But what happens if we ask a transformer model to solve a pronoun gender problem in an English-French translation? French has different spellings for nouns that have grammatical genders (feminine and masculine).

The following sentence contains the pronoun *it*, which can refer to the words *car* or *garage*. Can a transformer disambiguate this pronoun?

Open `Transformer_tasks.ipynb`, go to the `#Winograd` cell, and run our example:

```
from transformers import pipeline
translator = pipeline("translation_en_to_fr",model="t5-base")
print(translator("The car could not go in the garage because i
```

The translation is acceptable:

```
[{'translation_text': "La voiture ne pouvait pas aller dans le
```

The transformer detected that the word *it* refers to the word *car*, which is a feminine form. The feminine form applies to *it*, and the adjective *big*.

*elle* means *she* in French, which is the feminine form of 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 correct 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

The paradigm shift triggered by ChatGPT compelled us to redefine what an NLP task is. We saw that ChatGPT, like other LLM models, can perform tasks they were not trained for, including many SuperGLUE tasks through advanced emergence. We explored the outputs of the attention heads to bridge the gap between numerical calculations and producing sequences of words.

We then explored how to measure the performance of multi-task transformers. Transformers' ability to obtain top-ranking results for downstream tasks is unique in NLP history. We went through the demanding SuperGLUE tasks that brought transformers up to the top ranks of the GLUE and SuperGLUE leaderboards.

BoolQ, CB, WiC, and the many other tasks we covered are by no means easy to process, even for humans. We went through an example of several downstream tasks that show the difficulty transformer models face in proving their efficiency.

Transformers have proven their value by outperforming the former NLU architectures. To illustrate how simple it is to implement downstream fine-tuned tasks, we then ran several tasks in a Google Colaboratory notebook using Hugging Face’s pipeline for transformers.

In *Winograd schemas*, we gave the Transformer the difficult task of solving a Winograd disambiguation problem for an English-French translation.

In this chapter, we went through the NLP benchmarking process with SuperGLUE and some tasks. For more, consult the *Further reading* section of this chapter.

*Chapter 4, Advancements in Translations with Google Trax, Google Translate, and Gemini*, will take translation tasks a step further, and we will build a translation model with Trax.

## Questions

1. Machine intelligence uses the same data as humans to make predictions. (True/False)
2. SuperGLUE is more difficult than GLUE for NLP models. (True/False)
3. BoolQ expects a binary answer. (True/False)
4. WiC stands for Words in Context. (True/False)
5. **Recognizing Textual Entailment (RTE)** detects whether one sequence entails another sequence. (True/False)
6. A Winograd schema predicts whether a verb is spelled correctly. (True/False)
7. Transformer models now occupy the top ranks of GLUE and SuperGLUE. (True/False)

8. Human Baselines standards are not defined once and for all. They were made tougher to attain by SuperGLUE. (True/False)
9. Transformer models will never beat SuperGLUE Human Baselines standards. (True/False)
10. Variants of transformer models have outperformed RNN and CNN models. (True/False)

## References

- *Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and 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, and Samuel R. Bowman, 2019, GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding:*  
<https://arxiv.org/abs/1804.07461>
- *Yu Sun, Shuohuan Wang, Yukun Li, Shikun Feng, Hao Tian, Hua Wu, and Haifeng Wang, 2019, ERNIE 2.0: A Continual Pretraining Framework for Language Understanding:*  
<https://arxiv.org/pdf/1907.12412.pdf>
- *Melissa Roemmele, Cosmin Adrian Bejan, and Andrew S. Gordon, 2011, Choice of Plausible Alternatives: An Evaluation of Commonsense Causal Reasoning:*

<https://people.ict.usc.edu/~gordon/publications/AAAI-SPRING11A.PDF>

- *Richard Socher, Alex Perelygin, Jean Y. Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts, 2013, Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank:*  
[https://nlp.stanford.edu/~socherr/EMNLP2013\\_RNTN.pdf](https://nlp.stanford.edu/~socherr/EMNLP2013_RNTN.pdf)
- *Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierrick Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and 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/docs/transformers/main/en/quicktour>

## Further reading

You can examine many other LLM benchmarking approaches, including the following tools:

- Hugging Face is a good place to continue with the Open LLM Leaderboard:  
[https://huggingface.co/spaces/HuggingFaceH4/open\\_llm\\_leaderboard](https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard)
- Google provides a comprehensive benchmarking service with BIG-bench and 200+ tasks: <https://github.com/google/BIG-bench>

[bench](#)

Ultimately, the decision to use a benchmarking framework depends on each project.

## Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://www.packt.link/Transformers>



# 4

## Advancements in Translations with Google Trax, Google Translate, and Gemini

Transformers have advanced translation with their unique ability to capture the meaning of sequences of words in scores of languages. In this chapter, we will go through some key translation concepts and explore their scope in Google Trax, Google Translate, and Gemini.

Humans excel at transduction, transferring information from one language to another. We can easily imagine a mental representation of a sequence. For example, if somebody says *The flowers in my garden are beautiful*, we can easily visualize a garden with flowers in it. We see images of the garden, although we might never have seen that garden. We might even imagine chirping birds and the scent of flowers.

A machine must learn transduction from scratch with numerical representations. Recurrent or convolutional approaches have produced exciting results but have not reached significant **Bilingual Evaluation Understudy (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. For example, a sequence in language *A* is

adequately represented before attempting to translate it into language  $B$ . Self-attention brings the level of language understanding required by a machine to obtain better BLEU scores.

In 2017, the seminal *Attention Is All You Need* Transformer displayed the best results for English-German and English-French translations. Since then, the scores have been improved by other transformers.

In this chapter, we will go through machine translation in three steps. We will first define what machine translation is. We will then preprocess a

**Workshop on Machine Translation (WMT)** dataset. Finally, we will see how to implement machine translations.

This chapter covers the following topics:

- Defining machine translation
- Human transductions and translations
- Machine transductions and translations
- Evaluating machine translations
- Preprocessing a WMT dataset
- Evaluating machine translations with BLEU
- Translations with Google Trax
- Creating the Original Transformer model
- Tokenizing a sentence
- Decoding from the Transformer
- De-tokenizing and displaying the translation
- Translation with Google Translate
- Translation with a Google Translate AJAX API Wrapper
- Translation with Gemini



With all the innovations and library updates in this cutting-edge field, packages and models change regularly. Please go to the GitHub repository for the latest installation and code examples:

<https://github.com/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition/tree/main/Chapter04>.

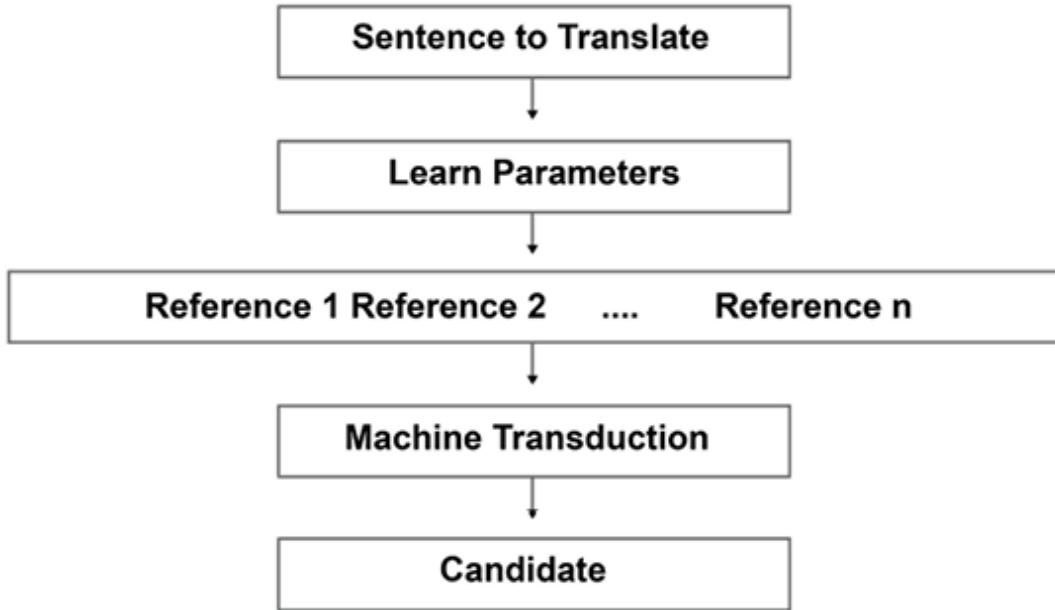
You can also post a message in our Discord community (<https://www.packt.link/Transformers>) if you have any trouble running the code in this or any chapter.

Our first step will be to define machine translation.

## Defining machine translation

*Vaswani et al.* (2017) tackled one of the most difficult NLP problems when designing the Transformer. The human baseline for machine translation seems out of reach for us human-machine intelligence designers. However, 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 4.1: Machine translation process*

The general idea in *Figure 4.1* is for the machine to do the following in a few steps:

1. Choose a sentence to translate.
2. Learn how words relate to each other with hundreds of millions of parameters.
3. Learn the many ways in which words refer to each other.
4. Use machine transduction to transfer the learned parameters to new sequences.
5. Choose a candidate translation for a word or sequence.

The process always starts with a sentence to translate from a source language, *A*. The process ends with an output containing a translated sentence in language *B*. The intermediate calculations involve transductions.

## Human transductions and translations

As an example, a human interpreter at the European Parliament 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, as each word's context is ignored.

Human transduction takes a sentence in language *A* and builds a cognitive *representation* of the sentence's meaning. An interpreter (oral translations) or a translator (written translations) at the European Parliament will only transform that transduction into an interpretation of that sentence in language *B*.

We will name the translation done by the interpreter or translator in language *B* a *reference* sentence.

You will notice several references in the *Machine translation process* described in *Figure 4.1*.

A human translator will not translate sentence *A* into sentence *B* several times and only once in real life. However, more than one translator could translate sentence *A* in real life. For example, you can find several French-to-English translations of *Les Essais* by Montaigne. If you take one sentence, *A*, out of the original French version, you will thus find several versions of sentence *B* noted as references 1 to n.

If you go to the European Parliament one day, you might notice that the interpreters only translate for a limited time of two hours, for example. Then, another interpreter takes over. No two interpreters have the same style, just like writers have different styles. For example, 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 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. Transformers, like humans, will produce a result we can refer to, but that can change if we train it differently or use different transformer models!

We immediately realize that the human baseline of human transduction, representations of a language sequence, is quite a challenge. However, much progress has been made.

An evaluation of machine translation proves that NLP has progressed. To determine that one solution is better than another, each NLP challenger, lab, or organization must refer to the same datasets for the comparison to be valid.

## Evaluating machine translations

*Vaswani et al.* (2017) presented the Original Transformer’s achievements in the **Workshop on Statistical Machine (WMT)** 2014 English-to-German translation task and the WMT 2014 English-to-French translation task. The Original Transformer achieved a state-of-the-art BLEU score. BLEU will be described in the *Evaluating machine translation with BLEU* section of this chapter.

However, we must begin by preprocessing the WMT dataset we will examine.

## Preprocessing a WMT dataset

The 2014 WMT contained several European language datasets. One dataset contained data from version 7 of the `Europarl` corpus. We will use the French-English dataset from the *European Parliament Proceedings Parallel Corpus*, 1996–2011

(<https://www.statmt.org/europarl/v7/fr-en.tgz>).

Open `WMT-translations.ipynb`, which is in the chapter directory of the GitHub repository.

The first step is to download the files we need:

```
import urllib.request
# Define the file URL
file_url = "https://www.statmt.org/europarl/v7/fr-en.tgz"
# Define the destination file path
destination_file = "/content/fr-en.tgz"
# Download the file
urllib.request.urlretrieve(file_url, destination_file)
```

Now, we will extract the files:

```
import tarfile
# Extract the tar file
with tarfile.open(destination_file, 'r:gz') as tar_ref:
    tar_ref.extractall("/content/fr-en")
```

Once you have downloaded the files and have extracted them, they will be available in the following paths:

- /content/fr-en/europarl-v7.fr-en.en
- /content/fr-en/europarl-v7.fr-en.fr

We will load, clear, and reduce the size of the corpus.

Let's start the preprocessing of the two parallel files.

## Preprocessing the raw data

In this section, we will preprocess `europarl-v7.fr-en.en` and `europarl-v7.fr-en.fr`.

The program begins by using standard Python functions and `pickle` to dump the serialized output files:

```
import pickle
from pickle import dump
```

Then, we define the function to load the file into memory:

```
# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, mode='rt', encoding='utf-8')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text
```

The loaded document is then split into sentences:

```
# split a loaded document into sentences
def to_sentences(doc):
    return doc.strip().split('\n')
```

The shortest and the longest lengths are retrieved:

```
# shortest and longest sentence lengths
def sentence_lengths(sentences):
    lengths = [len(s.split()) for s in sentences]
    return min(lengths), max(lengths)
```

The imported sentence lines must be cleaned to avoid training useless and noisy tokens. The lines are normalized, tokenized on white spaces, and converted to lowercase. The punctuation is removed from each token, non-printable characters are removed, and tokens containing numbers are excluded. The cleaned line is stored as a string.

The program runs the cleaning function and returns clean appended strings:

```
# clean lines
import re
import string
import unicodedata
def clean_lines(lines):
    cleaned = list()
    # prepare regex for char filtering
    re_print = re.compile('[^%s]' % re.escape(string.printable))
    # prepare translation table for removing punctuation
    table = str.maketrans('', '', string.punctuation)
    for line in lines:
        # normalize unicode characters
        line = unicodedata.normalize('NFD', line).encode('UTF-8')
        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 that we can save for future use:

```
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(sente
cleanf=clean_lines(sentences)
filename = 'French.pkl'
outfile = open(filename, 'wb')
pickle.dump(cleanf,outfile)
outfile.close()
print(filename, " saved")
```

The output shows the key statistics for the French dataset and confirms that `French.pkl` is saved:

```
French data: sentences=2007723, min=0, max=693
French.pkl saved
```

The main preprocessing is done. However, we still need to ensure that the datasets do not contain noisy and confusing tokens.

## Finalizing the preprocessing of the datasets

Our process now involves defining the function that will load the datasets that were cleaned up in the previous section, and then saving 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 essential to know how many times a word is used in the sequences we will parse. For example, if a word is only used once in a dataset containing two million lines, we will waste our energy using precious GPU resources to learn it!

Let's define the counter:

```
# create a frequency table for all words
def to_vocab(lines):
    vocab = Counter()
    for line in lines:
        tokens = line.split()
        vocab.update(tokens)
    return vocab
```

The vocabulary counter will detect words with a frequency that is below `min_occurrence`:

```
# remove all words with a frequency below a threshold
def trim_vocab(vocab, min_occurrence):
```

```
tokens = [k for k,c in vocab.items() if c >= min_occur  
return set(tokens)
```



In this case, `min_occurrence=5`, and the words below or equal to this threshold, have been removed to avoid wasting the training model's time analyzing them.

We now have to deal with **Out-Of-Vocabulary (OOV)** words. OOV words can be misspelled words, abbreviations, or any word that does not fit standard vocabulary representations. We could use automatic spelling, but it would not solve all of the problems. For this example, we will simply replace OOV words with the `unk` (unknown) token:

```
# mark all OOV with "unk" for all lines  
def update_dataset(lines, vocab):  
    new_lines = list()  
    for line in lines:  
        new_tokens = list()  
        for token in line.split():  
            if token in vocab:  
                new_tokens.append(token)  
            else:  
                new_tokens.append('unk')  
        new_line = ' '.join(new_tokens)  
        new_lines.append(new_line)  
    return new_lines
```

We will now run the functions for the English dataset, save the output, and then display `20` lines:

```
# load English dataset  
filename = 'English.pkl'  
lines = load_clean_sentences(filename)
```

```

# calculate vocabulary
vocab = to_vocab(lines)
print('English Vocabulary: %d' % len(vocab))
# reduce vocabulary
vocab = trim_vocab(vocab, 5)
print('New English Vocabulary: %d' % len(vocab))
# mark out of vocabulary words
lines = update_dataset(lines, vocab)
# save updated dataset
filename = 'english_vocab.pkl'
save_clean_sentences(lines, filename)
# spot check
for i in range(20):
    print("line", i, ":", lines[i])

```

The output functions first show the vocabulary compression obtained:

```

English Vocabulary: 105357
New English Vocabulary: 41746
Saved: english_vocab.pkl

```

The preprocessed dataset is saved. The output function then displays 20 lines, as shown in the following excerpt:

```

line 0 : resumption of the session
line 1 : i declare resumed the session of the european parliame
line 2 : although as you will have seen the dreaded millennium
line 3 : you have requested a debate on this subject in the cou

```

Let's now run the functions for the French dataset, save the output, and then display 20 lines:

```

# load French dataset
filename = 'French.pkl'
lines = load_clean_sentences(filename)
# calculate vocabulary
vocab = to_vocab(lines)
print('French Vocabulary: %d' % len(vocab))
# reduce vocabulary
vocab = trim_vocab(vocab, 5)
print('New French Vocabulary: %d' % len(vocab))
# mark out of vocabulary words
lines = update_dataset(lines, vocab)
# save updated dataset
filename = 'french_vocab.pkl'
save_clean_sentences(lines, filename)
# spot check
for i in range(20):
    print("line", i, ":", lines[i])

```

The output functions first show the vocabulary compression obtained:

```

French Vocabulary: 141642
New French Vocabulary: 58800
Saved: french_vocab.pkl

```

The preprocessed dataset is saved. The output function then displays 20 lines, as shown in the following excerpt:

```

line 0 : reprise de la session
line 1 : je declare reprise la session du parlement europeen qu
line 2 : comme vous avez pu le constater le grand bogue de lan
line 3 : vous avez souhaite un debat a ce sujet dans les procha

```

This section showed how raw data must be processed before training. The datasets are now ready to be plugged into a transformer to be trained.

Each line of the French dataset is the *sentence* to translate. Each line of the English dataset is the *reference* for a machine translation model. The machine translation model must produce an *English candidate translation* that matches the *reference*.

BLEU provides a method to evaluate `candidate` translations produced by machine translation models.

## Evaluating machine translations with BLEU

Papineni *et al.* (2002) devised an efficient way to evaluate a human translation. The human baseline was difficult to define. However, they realized we could obtain efficient results if we compared human translation with machine translation, word for word.

Papineni *et al.* (2002) named their method the **Bilingual Evaluation Understudy Score (BLEU)**.

In this section, we will use the **Natural Language Toolkit (NLTK)** to implement BLEU:

[http://www.nltk.org/api/nltk.translate.html#nltk.translate.bleu\\_score.sentence\\_bleu](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.

We will use the `nltk` library in Python with text processing libraries.

The program imports the `nltk` library:

```
from nltk.translate.bleu_score import sentence_bleu  
from nltk.translate.bleu_score import SmoothingFunction
```

The notebook then compares a candidate translation produced by the machine translation model and the actual translation references in the dataset. Remember that a sentence could have been repeated several times and translated by different translators in different ways, making it challenging to find efficient evaluation strategies.

The program can evaluate one or more references:

```
#Example 1  
reference = [['the', 'cat', 'likes', 'milk'], ['cat', 'likes'  
candidate = ['the', 'cat', 'likes', 'milk']  
score = sentence_bleu(reference, candidate)  
print('Example 1', score)  
#Example 2  
reference = [['the', 'cat', 'likes', 'milk']]  
candidate = ['the', 'cat', 'likes', 'milk']  
score = sentence_bleu(reference, candidate)  
print('Example 2', score)
```

The score for both examples is 1:

```
Example 1 1.0  
Example 2 1.0
```

A straightforward evaluation  $P$  of the candidate  $C$ , the reference  $R$ , and the number of correct tokens found in  $C$  ( $N$ ) can be represented as a geometric function:

$$P(N, C, R) = \left( \prod_{n=1}^N p_n \right)^{\frac{1}{N}}$$

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 a 3-gram overlap:

```
Warning (from warnings module):
  File
Example 3 1.0547686614863434e-154
/usr/local/lib/python3.10/dist-packages/nltk/translate/bleu_sco
The hypothesis contains 0 counts of 3-gram overlaps.
Therefore the BLEU score evaluates to 0, independently of
how many N-gram overlaps of lower order it contains.
Consider using lower n-gram order or use SmoothingFunction()
```

The hyperparameters can be changed, but the approach remains rigid even if we sometimes obtain acceptable evaluations.

The warning in the preceding code is a good one that anticipates what we will discuss in the next section.



The messages may vary with each version of the program and each run, since this is a stochastic process.

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

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

Reference 1: The cat is on the mat.

Reference 2: There is a cat on the mat.

Now, consider the following candidate sequence:

Candidate: the the the the the the

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 the geometric evaluation approach of standard BLEU techniques.

Label smoothing is a very efficient method that improves the performance of a Transformer model during the training phase. It has a negative impact on perplexity. However, it forces the model to be more uncertain. In turn, this has a positive effect on accuracy.

For example, suppose we have to predict what the masked word is in the following sequence:

The cat [mask] milk.

Imagine the output comes out as a softmax vector:

```
candidate_words=[drinks, likes, enjoys, appreciates]
candidate_softmax=[0.7, 0.1, 0.1, 0.1]
candidate_one_hot=[1,0,0,0]
```

This would be a brutal approach. Label smoothing can make the system more open-minded by introducing epsilon =  $\epsilon$ .

The number of elements of `candidate_softmax` is  $k=4$ .

For label smoothing, we can set  $\epsilon$  to 0.25, for example.

One of several approaches to label smoothing can be a straightforward function.

First, reduce the value of `candidate_one_hot` by  $1 - \epsilon$ .

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

We obtain the following result if we apply this approach:

```
candidate_smoothed=[0.75, 0.083, 0.083, 0.083],
```

making the output open to future transformations and changes.

Transformers uses variants of label smoothing.

A variant of BLEU is chencherry smoothing.

## Chencherry smoothing

*Chen and Cherry* (2014) introduced an interesting way of smoothing candidate evaluations by adding  $\epsilon$  to otherwise 0 values. In this section, we will first evaluate without smoothing and then implement a chencherry (*Boxing Chen + Colin Cherry*) smoothing function.

Let's first evaluate a French-English example without smoothing:

```
#Example 4
reference = [['je', 'vous', 'invite', 'a', 'vous', 'lever', 'pour']
candidate = ['levez', 'vous', 'svp', 'pour', 'cette', 'minute', '']
score = sentence_bleu(reference, candidate)
print("without soothng score", score)
```

Although a human could accept the candidate, the output score is weak:

```
without smoothing score 0.37188004246466494
```

Now, let's add some open-minded smoothing to the evaluation:

```
chencherry = SmoothingFunction()
r1=list('je vous invite a vous lever pour cette minute de sile
candidate=list('levez vous svp pour cette minute de silence')
print("with smoothing score",sentence_bleu([r1], candidate,smo
```

The score is just an example of how to use smoothing and will improve continually as the functions evolve:

```
with smoothing score 0.6194291765462159
```

We have now seen how a dataset is preprocessed and how BLEU evaluates machine translations.

## Translations with Google 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, including many transformer examples.

Although T2T was a good start, Google Brain then produced Trax, an end-to-end deep learning library. Trax contains a Transformer model that can be applied to translations. The Google Brain team presently maintains Trax.

This section will focus on the minimum functions to initialize the English-German problem described by *Vaswani et al.* (2017), illustrating the Original Transformer's performance.

We will use preprocessed English and German datasets to show that the transformer architecture is language-agnostic.

Open `Trax_Google_Translate.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:

```
import os
import numpy as np
!pip install -q -U trax
import trax
```

Yes, it's that simple!

Now, let's create our Transformer model.

## Creating the Original Transformer model

We will create the Original Transformer model as described in *Chapter 2, Getting Started with the Architecture of the Transformer Model*.

Our Trax function will retrieve a pretrained model configuration in a few lines of code:

```
# Pretrained model config in gs://trax-ml/models/translation/e
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 Original 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.

We can look at the architecture of the model with the following code:

```
from pprint import pprint
pprint(vars(model))
```

The output provides an interesting view of the model's architecture:

```
...
Serial_in2_out2[
    Embedding_33300_512
    Dropout
    PositionalEncoding
    Serial_in2_out2[
        Branch_in2_out3[
            None
            Serial_in2_out2[
                LayerNorm
                Serial_in2_out2[
                    _in2_out2
                    Serial_in2_out2[
                        Select[0,0,0]_out3
                        Serial_in4_out2[
                            _in4_out4
                            Serial_in4_out2[
                                Parallel_in3_out3[
                                    Dense_512
                                    Dense_512
                                    Dense_512
...
...
```

Take some time to go through the model.

Let's take a critical concept that opens the toolkit to many flexible architectures.

`Combinator` is a type of layer that contains other layers. It is an abstract class. We use subclasses of this class to combine layers for different functions, such as `Serial`, `Parallel`, `Branch`, `Residual`, and `Select`. These subclasses are combinators used as building blocks of the neural network:

- `Serial` applies layers one after the other in a sequence.
- `Parallel` applies several layers to the same input in parallel and a tuple of their outputs.
- `Branch` applies several layers in parallel to an input. Then, it concatenates the outputs.
- `Residual` adds the output of sequences of layers (or a single layer) to the input.
- `Select` selects elements from a tuple of inputs.

The `in2_out2` term means that there are two inputs, and two outputs are produced.

For more information, please refer to the Google Trax documentation:

[https://trax-ml.readthedocs.io/en/latest/notebooks/layers\\_intro.html](https://trax-ml.readthedocs.io/en/latest/notebooks/layers_intro.html)

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:

```
model.init_from_file('gs://trax-ml/models/translation/ende_wmt:  
weights_only=True')
```

The code displays the weights, which helps see how the model is built under the hood:

```
...
```

```
1.95107 , 2.3058772 , 1.9680263 , 1.5733448 , 1.7866848 , 2.192
```

```
...
```

The machine configuration and its *intelligence* are now ready to run. Let's tokenize a sentence.

## Tokenizing a sentence

Our machine translator is ready to tokenize a sentence. The notebook uses the vocabulary preprocessed by Trax. The preprocessing method is similar to the one described in this chapter's *Preprocessing a WMT dataset* section.

The sentence will now be tokenized:

```
#Tokenizing a sentence  
sentence = 'I am only a machine but I have machine intelligence'  
tokenized = list(trax.data.tokenize(iterator([sentence]), # Operate  
vocab_dir='gs://trax-ml/vocabs',  
vocab_file='ende_32k.subword')
```

We are now ready to 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:

```
tokenized = tokenized[None, :] # Add batch dimension.  
tokenized_translation = trax.supervised.decoding.autoregressive(  
    model, tokenized, temperature=0.0) # Higher temperature: more  
    # likely to be correct.
```

Note that higher temperatures will produce different results, just as with human translators, as explained in this chapter’s *Defining machine translation* section.

Finally, the program will de-tokenize and display the translation.

## De-tokenizing and displaying the translation

Google Brain has produced a mainstream, disruptive, and intuitive implementation of the Transformer with `Trax`.

The program now de-tokenizes and displays the translation in a few lines:

```
tokenized_translation = tokenized_translation[0][:-1] # Remove  
translation = trax.data.detokenize(tokenized_translation,  
    vocab_dir='gs://trax-ml/vocab')  
print(translation)
```

```
vocab_file='ende_32k.subword.bpe'  
print("The sentence:", sentence)  
print("The translation:", translation)
```

The output is quite impressive:

```
The sentence: I am only a machine but I have machine intelligence  
The translation: Ich bin nur eine Maschine, aber ich habe Maschinenübersicht
```

The Transformer translated `machine intelligence` into  
`Maschinenübersicht`.

If we deconstruct `Maschinenübersicht` into `Maschin` (`machine`) + `übersicht` (`intelligence`), we can see that:

- `über` literally means “over”
- `sicht` means “sight” or “view”

The Transformer tells us that although it is a machine, it has vision. Machine intelligence is growing through transformers, but it is not human intelligence. Machines learn languages with an intelligence of their own.

Google Trax provides a toolkit to build and run models.

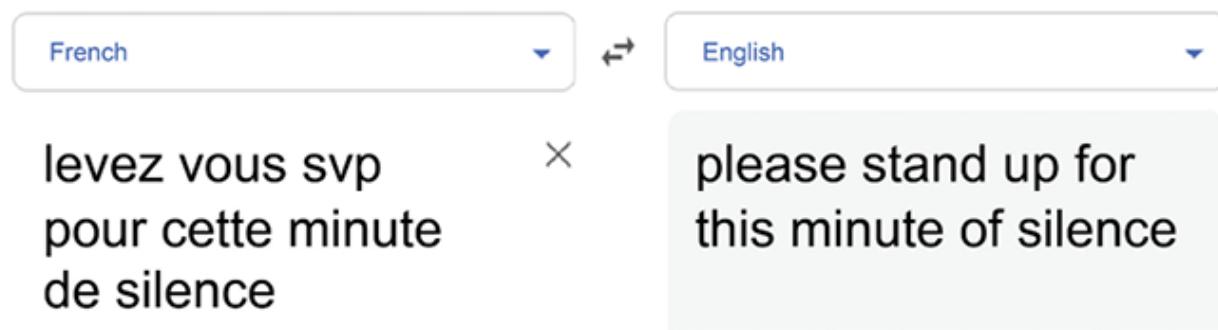
Now, let's explore Google Translate.

## Translation with Google Translate

Google Translate (<https://translate.google.com/>) provides a ready-to-use official interface for translations. Google also possesses transformer technology in its translation algorithms.

However, an AI specialist may not be required at all.

If we enter the sentence analyzed in the previous section in Google Translate, `Levez-vous svp pour cette minute de silence`, we obtain an English translation in real time:



*Figure 4.2: Google Translate*

The translation is correct.

Does the AI industry still require AI specialists for translation tasks or simply a web interface developer?

Google provides every service required for translations on their Google Translate platform: <https://cloud.google.com/translate>:

- A translation API: A web developer can create an interface for a customer.
- A media translation API that can translate your streaming content.
- An AutoML translation service that will train a custom model for a specific domain.

A Google translate project requires a web developer for the interfaces, a **Subject Matter Expert (SME)**, and perhaps a linguist. However, being an AI specialist is not a prerequisite.

LLMs are driving AI toward AI as a service. So why bother studying AI development with transformers? There are two important reasons to become an AI professional:

- In real life, AI projects often run into unexpected problems. For example, Google Translate might not fit a specific need, no matter how much goodwill was put into the project. In that case, APIs will come in handy!
- You must be an AI developer to use Google Trax for AI or Google Cloud AI APIs!

You never know! LLMs are rolling out and connecting everything to everything; some AI projects might run smoothly, and some will require AI expertise to solve complex problems.

Google Cloud AI charges the usage of its APIs. For educational purposes, we will implement a free Google Translate AJAX API wrapper.

## Translation with a Google Translate AJAX API Wrapper

In this section, we will implement `googletrans`. The `googletrans` library is an unofficial third-party wrapper around the Google Translate AJAX API.

`googletrans` is an excellent educational tool and can fit limited usage.

However, it is recommended to use the official Google Cloud AI Translate API for projects requiring many requests or commercial usage, among other restrictions.

`googletrans` provides the same translations as Google Translate, although it might encounter problems if Google changes the access to its API.

**AJAX (Asynchronous JavaScript and XML)** means that the interface doesn't need to wait for AJAX to complete its request, which can be written in JavaScript with **XML (eXtensible Markup Language)** or **JSON (JavaScript Object Notation)**.

`googletrans` thus inherits Google Translate's sequence-to-sequence (seq2seq) technology, which leverages neural networks, including transformers and other models.

Let's implement `googletrans`.

## Implementing `googletrans`

The second part, *Running a Google Translate AJAX API Wrapper*, of the chapter's `Trax_Google_Translate.ipynb` begins by installing `googletrans`:

```
!pip install googletrans==4.0.0-rc1 -q
```



For more information, go to  
<https://pypi.org/project/googletrans/>.

We can now create a translation object and call the methods of the library.

## Creating an instance of the `Translator` class

An instance of the `Translator` class takes one line of code:

```
from googletrans import Translator
translator = Translator()
```

The `translator` object is now created, and we can call the methods of the `Translator` class.

## Translating a text from English to French

The following cell calls the `translate` method:

```
import googletrans
translator = googletrans.Translator()
# Get the translation of "Google Trax can translate to French"
translation = translator.translate("Google Trax can translate"
# Print the translation
print(translation.text)
```

The output is the correct translation:

```
Google Trax peut se traduire en français
```

## Detecting the language of a text

We can also detect a language with the `detect` method:

```
lang=translator.detect('이 문장은 한글로 쓰여졌습니다.')
print(lang)
```

The method will display the language of the input sequence:

```
Detected(lang=ko, confidence=None)
```

In this case, the language detected is Korean.

Note that the confidence level is not always displayed.

## Translating in multiple languages

We can call the `translate` method for several languages:

```
from googletrans import Translator
translator = Translator()
languages = ['en', 'es', 'fr']
for language in languages:
    print(translator.translate('안녕하세요', dest=language).text)
```

The output will provide English, Spanish, and French translations in this example of “hello” in Korean:

```
hello
Hola
Bonjour
```

For more features: <https://py-googletrans.readthedocs.io/en/latest/>.

We have explored Google Translate. Now, we will run translations with Generative AI as an end user.

## Translation with Gemini

How far can we go with Gemini for translations? We reviewed some of Gemini’s NLP abilities in *Chapter 3, Emergent vs Downstream Tasks: The*

*Unseen Depths of Transformers.* Now, we will create a dialog with Gemini to explore its potential and limitations.

Go to <https://gemini.google.com/> to start a dialog.

Needless to say, Gemini has been trained with large datasets in scores of languages.

We will begin with Gemini's potential and then search for its limitations.

My prompt begins with **Denis:**

Gemini's comprehensive response follows my prompt that ends with a question mark.

## Gemini's potential

```
Denis: Can you translate a sentence from English to French?  
Yes, I can translate a sentence from English to French. For exa  
Here are some other examples of English-to-French translations:  
Hello = Bonjour  
Goodbye = Au revoir  
Thank you = Merci  
You're welcome = Je vous en prie  
Please = S'il vous plait  
Excuse me = Pardon  
How are you? = Comment allez-vous?  
I'm fine = Je vais bien  
What's your name? = Comment vous appelez-vous?  
My name is Gemini = Je m'appelle Gemini  
I can also translate longer texts, such as paragraphs or articl
```

Gemini is very confident as we can see in the following response:

```
Sure, here is the translation of the sentence "Tourists who go  
Les touristes qui vont en Europe aiment visiter tous les pays q
```



That's fine, but how do we know if the translation is correct if we don't speak French?

Let's ask Gemini to perform a back-translation task:

```
Denis:
```

```
Translate this sentence back into English:Les touristes qui von
```



The back-translation response by Gemini is correct:

```
Tourists who go to Europe like to visit as many countries as th
```



Back translations can be a way to check if the translation from one language to another suits your needs. It requires additional implementation and development but can provide higher quality applications.

This section shows that the models are progressing. LLMs represent considerable progress, but there is still a long way to go! We AI specialists still have much work to do, which is good news!

## Summary

In this chapter, we explored some of the essential aspects of translations with transformers.

We started by defining machine translation. Human translation sets a high baseline for machines to reach. We saw that English-French and English-German translations imply numerous problems to solve. The Original 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. For example, training a model on a WMT dataset requires BLEU evaluations. We saw that geometric assessments are a good basis for scoring translations, but even modified BLEU has its limits. We thus added a smoothing technique to enhance BLEU.

We implemented an English-to-German translation transformer with Trax, Google Brain's end-to-end deep learning library.

We saw that Google Translate provides a standard translation API, a media streaming API, and custom AutoML model training services. Implementing Google Translate APIs may require no AI development if the project rolls out smoothly. If not, we will have to get our hands dirty, like in the old days, which we did with the `googletrans` library with its Google Translate AJAX API wrapper!

Finally, we had a conversation with Gemini's Generative AI interface to explore the potential and limitations of LLMs applied to translations.

We have now covered the main building blocks to construct transformers and their architecture, and we viewed some of the NLP tasks they can perform.

The next step is to fine-tune a model when we wish to focus on a specific NLP task. In *Chapter 5, Diving into Fine-Tuning through BERT*, we will fine-tune a Transformer model.

## 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 two heads. (True/False)
8. The Original Transformer encoder has six layers. (True/False)
9. The Original Transformer encoder has six layers but only two 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/>
- *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>
- Trax's repository: <https://github.com/google/trax>
- Trax tutorial: <https://trax-ml.readthedocs.io/en/latest/>

## Further reading

- *Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu, 2002, BLEU: a Method for Automatic Evaluation of Machine Translation*:  
<https://aclanthology.org/P02-1040.pdf>

- *Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin, 2017, Attention Is All You Need:*  
<https://arxiv.org/abs/1706.03762>

## Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://www.packt.link/Transformers>



# 5

## Diving into Fine-Tuning through BERT

In *Chapter 3, Emergent vs Downstream Tasks: The Unseen Depths of Transformers*, we explored tasks through pretrained models that could perform efficiently. However, in some cases, a pretrained model will not produce the desired outputs. We can pretrain a model from scratch, as we will see in *Chapter 6, Pretraining a Transformer from Scratch through RoBERTa*. However, pretraining a model can require a large amount of machine, data, and human resources. The alternative can be fine-tuning a transformer model.

This chapter will dive into fine-tuning transformer models through a Hugging Face pretrained BERT model. By the end of the chapter, you should be able to fine-tune other Hugging Face models such as GPT, T5, RoBERTa, and more.

The chapter is divided into two parts. We will first go through the architecture of BERT models. Then, we will explore fine-tuning a transformer model through BERT.

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.

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

Then, we will fine-tune a BERT model trained by a third party and uploaded to Hugging Face. This shows that transformers can be pretrained. A pretrained BERT, for example, can be fine-tuned on several NLP tasks. We will go through this fascinating experience of fine-tuning a transformer using Hugging Face modules, including building an interface with `ipywidgets` to interact with our trained model.

This chapter covers the following topics:

- BERT architecture
- The encoder stack of transformers and bidirectional attention
- Fine-tuning process for the acceptability judgment task
- Creating training data, labels, and BERT tokens
- Processing the training data with a BERT tokenizer and attention masks
- Splitting data into training and validation sets
- Setting up a Hugging Face BERT uncased base model
- Grouping parameters
- Setting hyperparameters
- Running a training loop
- Post-training evaluation with holdout
- Building a Python interface to interact with the model

With all the innovations and library updates in this cutting-edge field, packages and models change regularly. Please go to the GitHub repository for the latest installation and code



examples:

<https://github.com/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition/tree/main/Chapter05>.

You can also post a message in our Discord community (<https://www.packt.link/Transformers>) if you have any trouble running the code in this or any chapter.

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

## The architecture of BERT

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

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

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

We will not go through the building blocks of transformers described in *Chapter 2, Getting Started with the Architecture of the Transformer Model*. You can consult *Chapter 2* at any time to review an aspect of the building blocks of transformers. Instead, this section 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 and then the preparation of the pretraining input environment. Then, we will explain the two-step framework of BERT: pretraining and fine-tuning.

Let's first explore the encoder stack.

## The encoder stack

The first building block we will take from the Original Transformer model is an encoder layer. The encoder layer, as described in *Chapter 2, Getting Started with the Architecture of the Transformer Model*, is shown in *Figure 5.1*:

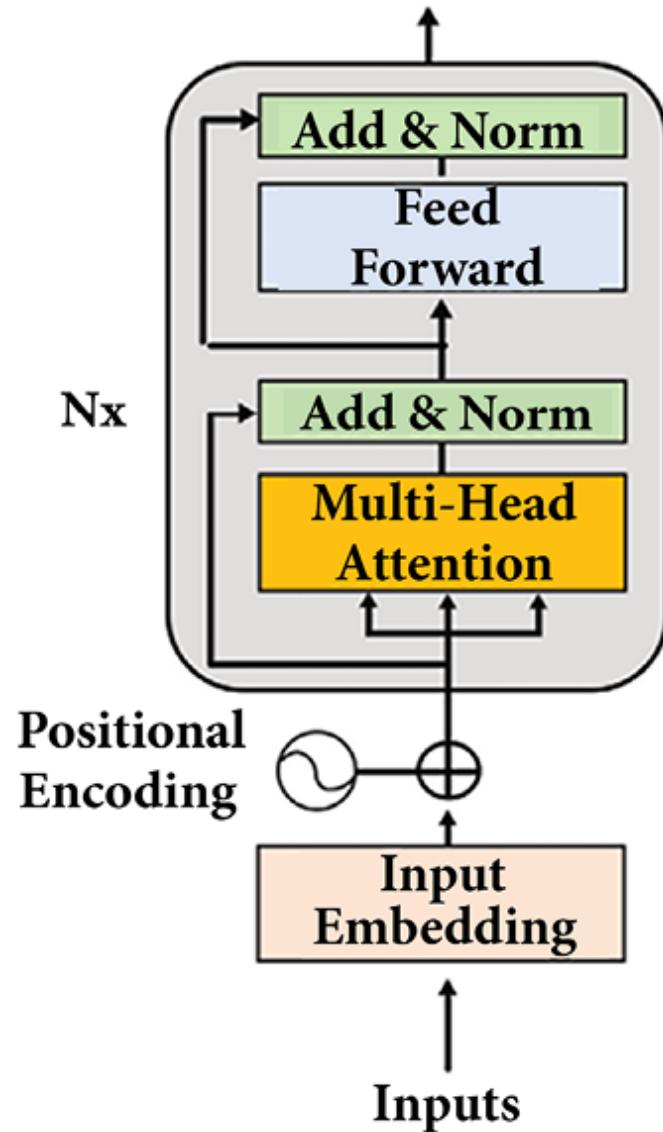


Figure 5.1: The encoder layer

The BERT model does not use decoder layers. A BERT model has an encoder stack but no decoder stacks. The BERT model uses **Masked Language Modeling (MLM)** in which some input tokens are hidden (“masked”), and the attention layers must learn to understand the context. The model will predict the hidden tokens, as we will see when we zoom into a BERT encoder layer in the following sections.

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

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

BERT encoder layers are larger than the Original Transformer model.

Different sizes of BERT models can be built with the encoder layers:

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

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

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

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

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

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

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

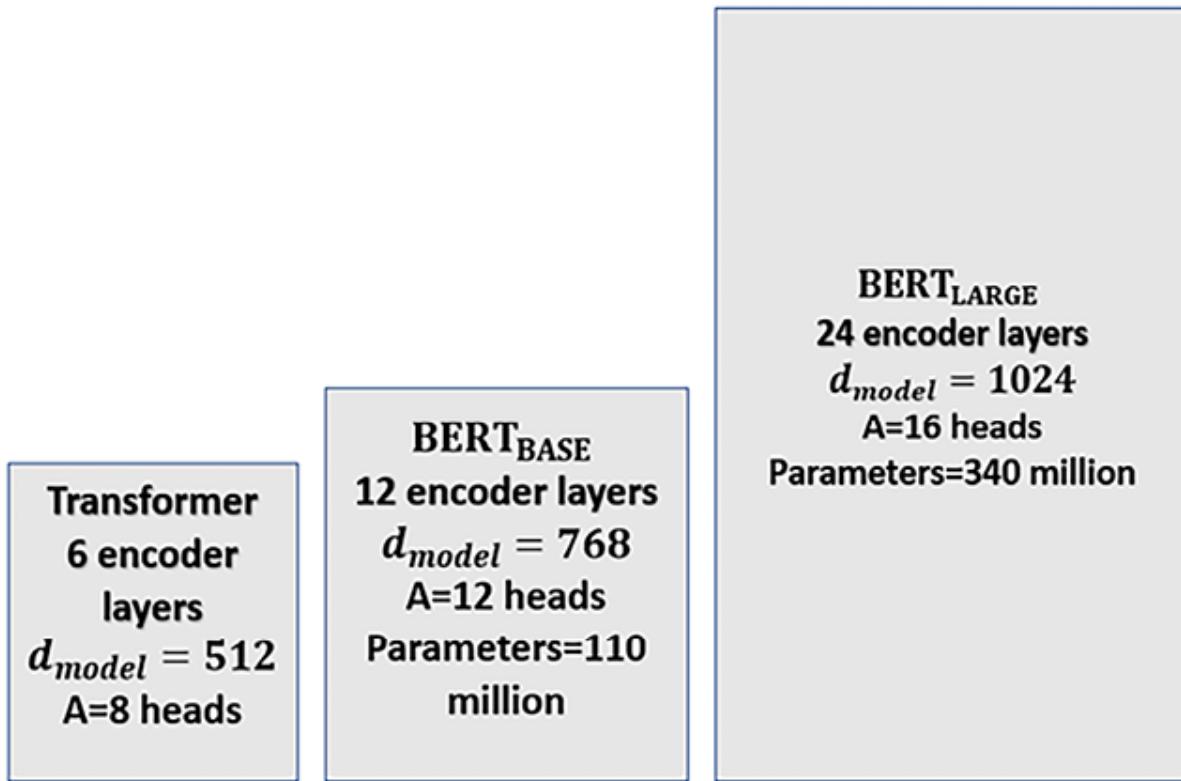


Figure 5.2: Transformer models

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

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

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

# Preparing the pretraining input environment

The BERT model only has encoder layers and no decoder stack. Its architecture features a multi-head self-attention mechanism that allows each token to learn to understand all the surrounding tokens (masked or not). This bidirectional method enables BERT to understand the context on both sides of each token.

A masked multi-head attention layer masks some tokens randomly to force this system to learn contexts. 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 random mask can be anywhere in the sequence, not necessarily at the end. 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 enhanced their bidirectional attention model, letting an attention head attend to *all* the words 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 **MLM**. The second method is **Next-Sentence Prediction (NSP)**.

Let's start with MLM.

## Masked language modeling

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

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

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

A potential input sequence could be:

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

The decoder could potentially mask the attention sequence after the model reaches the word `it`:

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

But the BERT encoder masks a random token to make a prediction, which makes it more powerful:

```
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 with 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 invented to train BERT is **NSP**. The input contains two sentences. In 50% of the cases, the second sentence was the actual second sentence of a document. In 50% of the cases, the second sentence was selected randomly and had no relation to the first one.

Two new tokens were added:

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

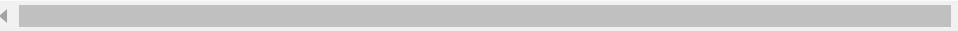
[SEP] is a separation token that signals the end of a sequence, such as a sentence, sentence part, or question, depending on the task at hand.

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

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

These two sentences will become one complete input sequence:

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



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

Note that the double hash (##) is because of the WordPiece tokenization used. The word “sleep” was tokenized separately from “ing.” This enables the tokenizer to work on a smaller dictionary of subwords, WordPiece, and then assemble them through the training process.

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

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

Figure 5.3: 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 MLM 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 of BERT are:

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

## Pretraining and fine-tuning a BERT model

BERT is a two-step framework. The first step is pretraining, and the second is fine-tuning, as shown in *Figure 5.4*:

## BERT: A Two-Step Framework

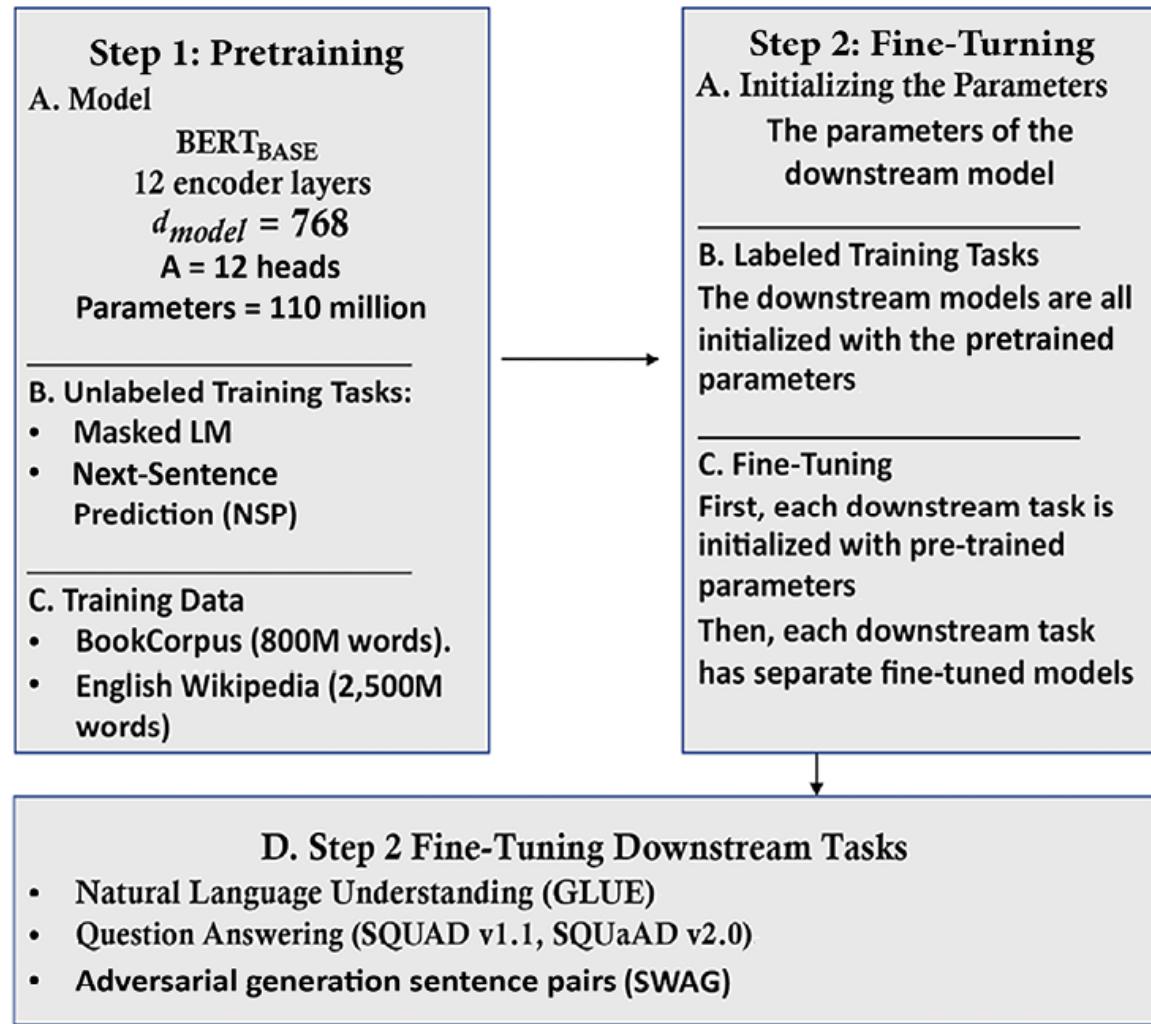


Figure 5.4: The BERT framework

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

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

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

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

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

In this chapter, the BERT model we will fine-tune will be trained on **the Corpus of Linguistic Acceptability (CoLA)**. The downstream task is based on the *Neural Network Acceptability Judgments* paper 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's 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 using Hugging Face. Hugging Face provides a large amount of resources for transformer models.

Hugging Face offers a significant number of pretrained models, such as BERT, GPT-2, RoBERTa, T5, and DistilBERT. These models can perform many tasks, such as those we went through in *Chapter 3, Emergent vs Downstream Tasks: The Unseen Depths of Transformers*. In addition, these models can be fine-tuned to fit your needs. This chapter focuses on the process of fine-tuning a Hugging Face BERT model. You can then apply the same method to other Hugging Face transformer models.

In *Chapter 15, Guarding the Giants: Mitigating Risks in Large Language Models*, we will use another platform, OpenAI, to fine-tune one of OpenAI's GPT models.

For this chapter, we will begin with fine-tuning BERT, which involves the following main steps:

- Retrieving the datasets, in this case, the CoLA datasets.
- Loading and configuring the pretrained model.
- Loading and preparing the data.
- Setting up the training parameters.
- Training the pretrained model for a new task. In this case, the trained model will learn how to assess linguistic acceptability.
- Evaluating the trained model.



It is essential to remember that this same process can be applied to other Hugging Face transformer models, such as RoBERTa, GPT-2, and many other models. Hugging Face provides documentation and a dynamic community. For more information on Hugging Face, explore their site:

<https://huggingface.co/>.

Let's apply the fine-tuning process to a BERT model by first defining the goal we wish to achieve.

## Defining a goal

The goal of fine-tuning BERT in this chapter is to focus on learning the ability to judge the grammatical acceptability of a text with the CoLA dataset, as seen in *Chapter 3, Emergent vs Downstream Tasks: The Unseen Depths of Transformers*. So, take some time to go back to the CoLA section of *Chapter 3* if necessary.



You can choose other tasks, including trying some of those we went through in *Chapter 3*, to fine-tune depending on your project.

We will measure the predictions with the **Matthews Correlation Coefficient (MCC)**, which will be explained in the *Evaluating using the Matthews correlation coefficient* section.

Open `BERT_Fine_Tuning_Sentence_Classification_GPU.ipynb` in Google Colab (make sure you have a Gmail account). The notebook is in `Chapter05` in the GitHub repository of this book.

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

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

## Hardware constraints

The scale of computation required to train and fine-tune transformer models makes them hardware-driven. This rapidly becomes a resource constraint. Small transformer models can be trained and fine-tuned on local machines or limited server configurations. However, large models such as state-of-the-art large OpenAI models and Google AI models require unique machine power.

The machine power required by **Large Language Models (LLMs)** has created a financial accessibility threshold. Therefore, Hugging Face offers a reasonable entry point to explore and implement transformer models.

Transformer models thus benefit from the parallel processing capability of GPUs. GPUs perform computational tasks far more efficiently than CPUs and will accelerate the training or fine-tuning process. Go to the **Runtime** menu in Google Colab, select **Change runtime type**, and select **GPU** in the **Hardware Accelerator** drop-down list.

The program will use Hugging Face modules, which we'll install next.

## Installing Hugging Face Transformers

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

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

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

```
try:  
    import transformers  
except:
```

```
print("Installing transformers")
!pip -q install transformers
```

The installation will run with the `-q` option, limiting the verbosity and only displaying warning or error messages.

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 `BERT Adam` optimizer is also imported along with the sequence classification module:

```
import torch
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader, RandomSampler
from keras.utils import pad_sequences
from sklearn.model_selection import train_test_split
from transformers import BertTokenizer, BertConfig
from transformers import AdamW, BertForSequenceClassification,
```

We will now import the progress bar module from `tqdm`:

```
from tqdm import tqdm, trange #for progress bars
```

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
from IPython.display import Image #for image rendering
```

If all goes well, no message will be displayed, bearing in mind that Google Colab has pre-installed several of the modules we need on their VMs.

## 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:

```
device = torch.device("cuda" if torch.cuda.is_available() else
!nvidia-smi
```

The output may vary with Google Colab configurations. For example:

```
+-----+
| NVIDIA-SMI 525.85.12      Driver Version: 525.85.12      CUDA Version: 12.0 |
+-----+
| GPU  Name      Persistence-M| Bus-Id      Disp.A  | Volatile Uncorr. ECC | |
| Fan  Temp  Perf  Pwr:Usage/Cap|          Memory-Usage | GPU-Util  Compute M. |
|                               |               |            | MIG M. |
+-----+
|  0  Tesla T4           Off  | 00000000:00:04.0 Off |                  0 |
| N/A   48C    P8    10W /  70W |            3MiB / 15360MiB |      0%     Default |
|                               |                           |                  N/A |
+-----+
+-----+
| Processes:
| GPU  GI  CI          PID  Type  Process name             GPU Memory |
|       ID  ID          ID   ID          name                Usage  |
|-----+
```

*Figure 5.5: Output of NVIDIA-SMI*

We will now load the dataset.

## Loading the CoLA dataset

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

Assessing linguistic acceptability is critical in **Natural Language Processing (NLP)**. If necessary, take a moment to review the CoLA section of *Chapter 3, Emergent vs Downstream Tasks: The Unseen Depths of Transformers*.

The dataset implemented in this section was obtained from the CoLA homepage: <https://nyu-mll.github.io/CoLA/> and uploaded to the GitHub directory of this chapter.

The following cells in the notebook automatically download the necessary files for fine-tuning, `in_domain_train.tsv`, and evaluating the predictions of the fine-tuned model, `out_of_domain_dev.tsv`:

```
import os
!curl -L https://raw.githubusercontent.com/Denis2054/Transformers/1.0.0/datasets/wikitext-2/raw_text.txt > in_domain_train.tsv
!curl -L https://raw.githubusercontent.com/Denis2054/Transformers/1.0.0/datasets/wikitext-2/raw_text.txt > out_of_domain_dev.tsv
```

You should see them appear in the file manager:



Figure 5.6: Uploading the datasets

Now, the program will load the datasets:

```
#source of dataset : https://nyu-mll.github.io/CoLA/
df = pd.read_csv("in_domain_train.tsv", delimiter='\t', header=0)
df.shape
```

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

```
(8551, 4)
```

A 10-line sample is displayed to visualize the *acceptability judgment* task:

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

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

	sentence_source	label	label_notes		sentence
4517	ks08	1	NaN		do n't you even touch that !
7622	sk813	1	NaN	sharks seem to swim slowly in the tropics .	
6549	g_81	0	*	the talkative and a bully man entered .	
5336	b_73	1	NaN	jack eats caviar more than he eats mush .	
460	bc01	0	*	the boat sank to collect the insurance .	
1110	r-67	1	NaN	i ran an old man down .	
8427	ad03	0	*	the dragons were slain all .	
5140	ks08	0	*	it is kim on whom sandy relies on .	
5639	c_13	1	NaN	the man loved peanut butter cookies .	
5916	c_13	1	NaN	molly gave calvin a kiss .	

Figure 5.7: 10 lines of the labeled dataset

Each sample in the `.tsv` files contains four tab-separated columns (column 0 is an index):

- 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:

```
sentences = df.sentence.values
# Adding CLS and SEP tokens at the beginning and end of each sentence
sentences = ["[CLS] " + sentence + " [SEP]" for sentence in sentences]
labels = df.label.values
```

The `[CLS]` and `[SEP]` have now been added for the BERT model to find the start and end of a sentence.

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:

```
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased',
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',
```

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. This is because the sentences in the datasets are short. But to make sure of this, the program sets the maximum length of a sequence to 128, and the sequences are padded:

```
# Set the maximum sequence length. The longest sequence in our
# 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
input_ids = [tokenizer.convert_tokens_to_ids(x) for x in token
# Pad our input tokens
input_ids = pad_sequences(input_ids, maxlen=MAX_LEN, dtype="lo
```

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 paying attention to those padded tokens!

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

```
attention_masks = []
# Create a mask of 1s for each token followed by 0s for padding
```

```
for seq in input_ids:  
    seq_mask = [float(i>0) for i in seq]  
    attention_masks.append(seq_mask)
```

The program will now split the data.

## Splitting the data into training and validation sets

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

```
# Use train_test_split to split our data into train and validation sets  
train_inputs, validation_inputs, train_labels, validation_labels,  
train_masks, validation_masks, _, _ = train_test_split(attention_masks, labels,
```

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

## Converting all the data into torch tensors

The fine-tuning model uses torch tensors. PyTorch tensors:

- Optimize the usage of GPUs.
- Facilitate GPU/CPU control. In this notebook, notice that the Hugging Face model and our datasets will be transferred to the GPU through the `to-device` command in the code of the training loop if the GPU is available, as defined in the *Specifying CUDA as the device for torch* section of this chapter and the notebook.

The program will thus convert 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

The program selects a batch size in this cell and creates an iterator. The iterator is an efficient way of looping through the data and loading batches instead of loading all the data in memory. The iterator, coupled with the torch `DataLoader`, can batch-train massive datasets without crashing the machine's memory.

In this model, the batch size is `32`:

```
# Select a batch size for training. For fine-tuning BERT on a . .
batch_size = 32
# Create an iterator of our data with torch DataLoader. This h . .
# with an iterator the entire dataset does not need to be load . .
train_data = TensorDataset(train_inputs, train_masks, train_la . .
train_sampler = RandomSampler(train_data)
train_dataloader = DataLoader(train_data, sampler=train_sample . .
validation_data = TensorDataset(validation_inputs, validation_! . .
validation_sampler = SequentialSampler(validation_data)
validation_dataloader = DataLoader(validation_data, sampler=va . .
```

---

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:

```
# Initializing a BERT bert-base-uncased style configuration
from transformers import BertModel, BertConfig
configuration = BertConfig()
# Initializing a model from the bert-base-uncased style config
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 some of the main parameters displayed:

- `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 Unit** activation function. The input is weighted by its magnitude, which makes it non-linear.
- `hidden_dropout_prob`: `0.1` is the dropout probability applied to the fully connected layers. Full connections can be found in the embeddings, encoder, and pooler layers. The output is not always a good reflection of the content of a sequence. Pooling the sequence of hidden states improves the output sequence.
- `hidden_size`: `768` is the dimension of the encoded layers and also the pooler layer.
- `initializer_range`: `0.02` is the standard deviation value when initializing the weight matrices.
- `intermediate_size`: `3072` is the dimension of the feed-forward layer of the encoder.
- `layer_norm_eps`: `1e-12` is the epsilon value for layer normalization layers.
- `max_position_embeddings`: `512` is the maximum length the model uses.
- `model_type`: `"bert"` is the name of the model.
- `num_attention_heads`: `12` is the number of heads.
- `num_hidden_layers`: `12` is the number of layers.
- `pad_token_id`: `0` is the ID of the padding token to avoid training padding tokens.

- `type_vocab_size`: `2` is the size of the `token_type_ids`, which identifies the sequences. For example, “the `dog[SEP]` The `cat[SEP]`” can be represented with token IDs `[0, 0, 0, 1, 1, 1]`.
- `vocab_size`: `30522` is the number of tokens the model uses 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:

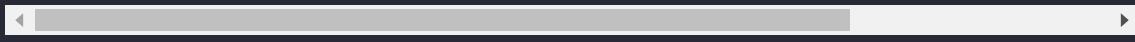
```
model = BertForSequenceClassification.from_pretrained("bert-base-uncased")
model = nn.DataParallel(model)
model.to(device)
```

We have defined the model, defined parallel processing, and sent the model to the device.

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:

```
DataParallel(
  (module): 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): LayerNorm((768,), eps=1e-12, elementwise_a
```

```
        (dropout): Dropout(p=0.1, inplace=False)
    )
(encoder): BertEncoder(
    (layer): ModuleList(
        (0-11): 12 x BertLayer(
            (attention): BertAttention(
                (self): BertSelfAttention(
                    (query): Linear(in_features=768, out_features=7
                    (key): Linear(in_features=768, out_features=768
                    (value): Linear(in_features=768, out_features=7
                    (dropout): Dropout(p=0.1, inplace=False)
                )
            )
            (output): BertSelfOutput(
                (dense): Linear(in_features=768, out_features=7
                (LayerNorm): LayerNorm((768,), eps=1e-12, element
                (dropout): Dropout(p=0.1, inplace=False)
            )
        )
    )
    (intermediate): BertIntermediate(
        (dense): Linear(in_features=768, out_features=307
        (intermediate_act_fn): GELUActivation()
    )
    (output): BertOutput(
        (dense): Linear(in_features=3072, out_features=76
        (LayerNorm): LayerNorm((768,), eps=1e-12, element
        (dropout): Dropout(p=0.1, inplace=False)
    )
)
)
)
)
)
(pooler): BertPooler(
    (dense): Linear(in_features=768, out_features=768, bias
    (activation): Tanh()
)
)
)
(dropout): Dropout(p=0.1, inplace=False)
(classifier): Linear(in_features=768, out_features=2, bias=
)
)
```



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:

```
#This code is taken from:  
# https://github.com/huggingface/transformers/blob/5bfcd0485ec...  
# Don't apply weight decay to any parameters whose names include  
# (Here, the BERT doesn't have 'gamma' or 'beta' parameters, or  
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'  
# - For the 'bias' parameters, the 'weight_decay_rate' is 0.0.  
optimizer_grouped_parameters = [  
    # Filter for all parameters which *don't* include 'bias',  
    {'params': [p for n, p in param_optimizer if not any(nd in  
        '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  
        'weight_decay_rate': 0.0}  
]  
# Note - 'optimizer_grouped_parameters' only includes the para
```

Let's go through some of the main parameters in the following cells:

- Looking into layer 3 of the `param_optimizer`, all the parameters of the `model` and their names are collected in `param_optimizer`.
- This function produces an iterator. The iterator contains tuples. Each tuple contains the name of a parameter and the parameters, which are often tensors in PyTorch. The following code displays the name `n` of the parameters and the value `p` of the parameters of layer 3:

```
# Displaying a sample of the parameter_optimizer: layer
layer_parameters = [p for n, p in model.named_parameters()
```

The following excerpt of the output displays a string and the value of its parameters:

```
'module.bert.embeddings.word_embeddings.weight', Parameter
```

Note that additional information is displayed. `cuda:0` means that the first CUDA-enabled GPU has been activated. `requires_grad=True` means that the gradients will be computed and the tensors updated during the training process. The model is “learning.”

- Looking into `no_decay`:

```
no_decay = ['bias', 'LayerNorm.weight']
```

Weight decay regularization will not be applied to the `bias` and `LayerNorm.weight` to avoid overfitting.

- The `optimizer_grouped_parameters` variable contains two dictionaries. These two groups of parameters contain the weight decay rate:

**Group 1:** contains the parameters for which weight decay will be applied:

**Group 2:** contains the parameters for which weight decay will not be applied, such as `bias` and `LayerNorm.weight`:

The parameter names are not included because the optimizer does not require them.

The parameters have been retrieved, and the model is ready for the training loop.

# The hyperparameters for the training loop

The hyperparameters for the training loop are critical, though they seem innocuous.

For example, the **Adam** optimizer will activate weight decay and undergo a warmup phase.

The number of `epochs` and `steps` will determine how much the model will “learn.” Several runs may prove helpful in determining the optimal level of training.

The learning rate (`lr`) and warmup 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 warmup rate. However, solving this problem requires many experimental runs.

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

```
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 prepared. 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 measurements for loss and accuracy will be plotted. Next, the training loop uses the `dataloader` to load and train batches. Finally, 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:

```
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_s
```

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

```
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 quickly, and deprecation messages and errors might occur. Hugging Face is no exception, 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:

```
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 relatively efficient:

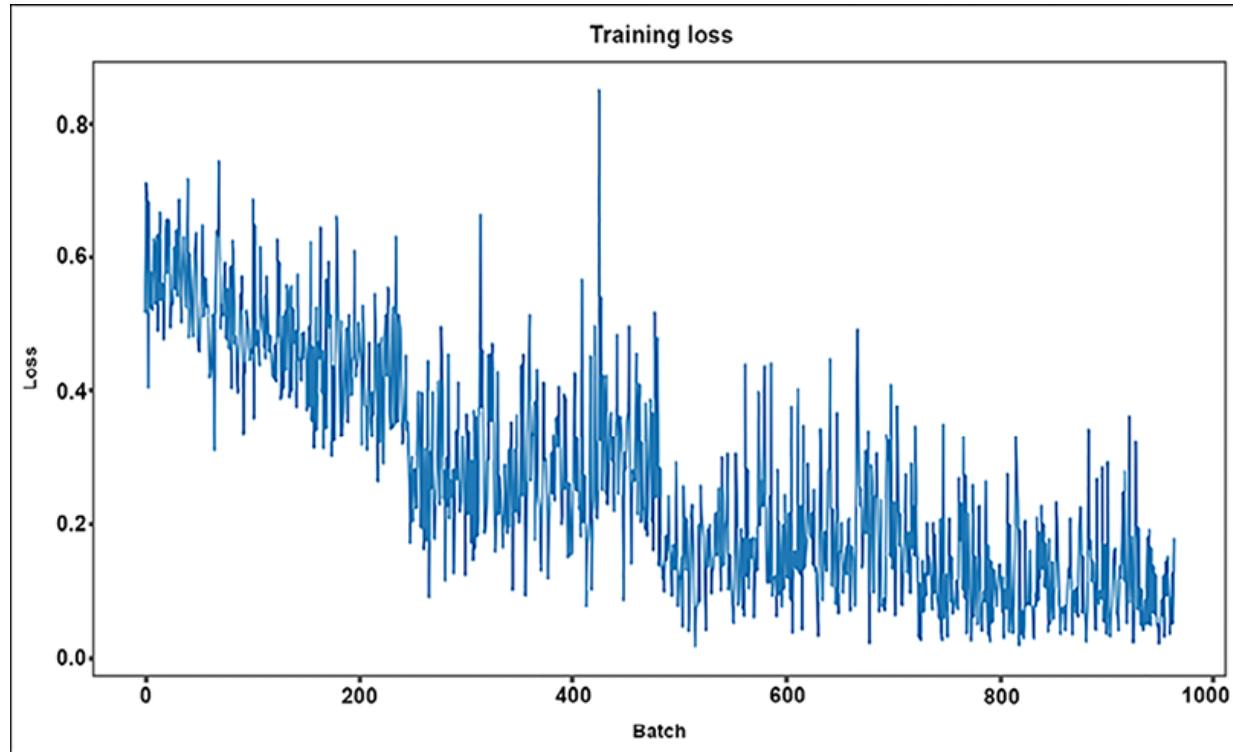


Figure 5.8: Training loss per batch

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

## Predicting and evaluating using the holdout dataset

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

The following excerpts of the code show the main steps of the evaluation process.

First, the data preparation process applied to the training data is repeated in the part of the code for the holdout dataset:

```
df = pd.read_csv("out_of_domain_dev.tsv", delimiter='\t', header=0)
df.shape
```

The output of `df.shape` displays the number of lines to assess and the shape of the dataset:

```
(516, 4)
```

Then, the evaluation process begins by creating the sentences to assess and their labels:

```
# Create sentence and label lists
sentences = df.sentence.values
# We need to add special tokens at the beginning and end of each sentence
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
```

```
with torch.no_grad():
    # Forward pass, calculate logit predictions
logits = model(b_input_ids, token_type_ids=None, attention_ma
```

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()
```

Then, the assessment is made by choosing the class with the highest probability:

```
# The predicted class is the one with the highest probability
batch_predictions = np.argmax(probabilities, axis=1)
```

Let's go deeper into this process in the following subsection.

## Exploring the prediction process

It is interesting to dig into the output of the prediction process. Let's take a look at the code that follows `batch_predictions`.

We can sum up the process by creating five steps and adding a function of our own:

```
print(f"Sentence: {sentence}")
```

This will display the sentence to assess. For example:

```
Sentence: somebody just left - guess who.
```

```
print(f"Prediction: {logits[i]}")
```

Now, we can display the fine-tuned model's prediction:

```
Prediction: [-2.6922598  2.1979954]
```

The first element, `-2.6922598`, is the first class's logit value, representing incorrect sentences in this case.

The second element, `2.1979954`, is the second class's logit value, which, in this case, represents the acceptable sentences.

Let's apply a softmax function to interpret the result.

```
print(f"Sofmax probabilities", softmax(logits[i]))
```

A softmax function was added to the notebook just above the prediction cell:

```
#Softmax logits
import numpy as np
def softmax(logits):
    e = np.exp(logits)
    return e / np.sum(e)
```

The result is an interpretable array of two probabilities that sum up to 1:

```
Sofmax probabilities [0.00746338 0.9925366 ]
```

---

The first element, the *unacceptable* class, has a probability of 0.00746338.

The second element, the *acceptable* class, has a probability of 0.9925366.

We can see that the probability of the acceptable class exceeds that of the unacceptable class.

We can now look at the results produced by the fine-tuned model.

```
print(f"Prediction: {batch_predictions[i]}")
```

The prediction of the model for the sentence is:

```
Prediction: 1
```

This matches the detailed analysis of the output we made.

```
print(f"True label: {label_ids[i]}")
```

The label of the dataset is:

```
True label: 1
```

We can see that, in this case, the prediction matches the true label in the dataset.

Once the predictions are made, the raw predictions and their true labels are stored:

```
predictions.append(logits)
true_labels.append(label_ids)
```

The program can now evaluate the predictions.

## Evaluating using the Matthews correlation coefficient

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

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

The value produced by the 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.

*Linguistic acceptability* can be measured with the MCC.

The MCC is imported from `sklearn.metrics`:

```
# 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.

## Matthews correlation coefficient 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:

```
# Flatten the predictions and true values for aggregate Matthews correlation coefficient
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 metric compares the labels (0 or 1) provided by the evaluation dataset and the labels produced by the prediction using the MCC. The result might vary from one run to another because of the stochastic nature of model predictions.

The result, in this case, is:

```
MCC: 0.524309707232222
```

The Matthews correlation coefficient measures the quality of binary classifications with an output value ranging from `-1` to `+1`:

- `+1` means that the predictions were 100% accurate.
- `0` is the same as a random calculation.
- `-1` indicates that the predictions are totally false.

In this case, an MCC of 0.54 is relatively efficient. The model has learned a significant amount of information but can be improved.

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

We will now interact with the model to explore its scope and limits.

## Building a Python interface to interact with the model

In this section, we will first save the model and then build an interface to interact with our trained model.

Let's first save the model if we choose to.

## Saving the model

The following code will save the model's files:

```
# Specify a directory to save your model and tokenizer
save_directory = "/content/model"
# If your model is wrapped in DataParallel, access the original
if isinstance(model, torch.nn.DataParallel):
    model.module.save_pretrained(save_directory)
else:
    model.save_pretrained(save_directory)
# Save the tokenizer
tokenizer.save_pretrained(save_directory)
```

The saved `/content/model` directory contains:

- `tokenizer_config.json`: Configuration details specific to the tokenizer.
- `special_tokens_map.json`: Mappings for any special tokens.
- `vocab.txt`: The vocabulary of tokens that the tokenizer can recognize.
- `added_tokens.json`: Any tokens that were added to the tokenizer after its initial creation.

When a model is wrapped in `DataParallel` for multi-GPU training in PyTorch, the original model becomes an attribute of the `DataParallel` object.

Remember to save your files outside of the Colab environment, as files in the Colab environment are discarded once the session ends. The saved model

can be large.

In this case, one solution is to mount Google Drive:

```
from google.colab import drive  
drive.mount('/content/drive')
```

Then, save the files in the `/content/model` directory to Google Drive.

The files can be loaded in another session, as shown in this excerpt of the code:

```
from transformers import BertTokenizer, BertForSequenceClassification  
# Directory where the model and tokenizer were saved  
load_directory = "/content/model/"  
.../...
```

The next step is to build an interface to interact with the trained model.

## Creating an interface for the trained model

First, make sure the model is in evaluation mode:

```
from transformers import BertTokenizer, BertForSequenceClassification  
import torch  
model.eval() # set the model to evaluation mode
```

Now, we create a function to interact with the model, as shown in the following excerpt:

```
def predict(sentence, model, tokenizer):
    # Add [CLS] and [SEP] tokens
    sentence = "[CLS] " + sentence + " [SEP]"
    # Tokenize the sentence
    tokenized_text = tokenizer.tokenize(sentence)
    .../...
```

We can sum up the function as follows:

1. The function takes in three parameters: a sentence, a model, and a tokenizer.
2. The sentence is preprocessed by adding [CLS] and [SEP] tokens, which are special tokens used in BERT-like models.
3. The sentence is then tokenized into sub-words using the provided tokenizer.
4. The tokens are converted into corresponding IDs from the model's vocabulary.
5. A segment ID is defined for each token, set to 0 since there's only one sequence.
6. The lists of token IDs and segment IDs are converted into PyTorch tensors.
7. A prediction is made using the model without updating its gradients.
8. The model outputs logits, which represent raw prediction values.
9. The label with the highest logit is selected as the predicted label.
10. The predicted label is returned.

We can now interact with the model.

## Interacting with the model

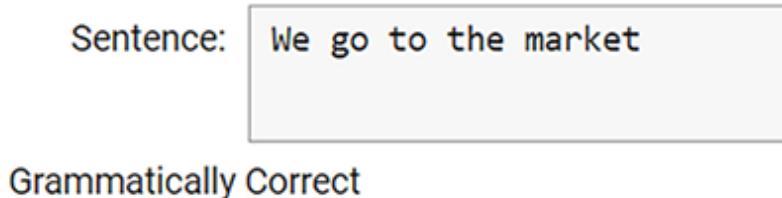
To build the interface, install `ipywidgets`:

```

import ipywidgets as widgets
from IPython.display import display
def model_predict_interface(sentence):
    prediction = predict(sentence, model, tokenizer)
    if prediction == 0:
        return "Grammatically Incorrect"
    elif prediction == 1:
        return "Grammatically Correct"
    else:
        return f"Label: {prediction}"
text_input = widgets.Textarea(
    placeholder='Type something',
    description='Sentence:',
    disabled=False,
    layout=widgets.Layout(width='100%', height='50px') # Adjust as needed
)
output_label = widgets.Label(
    value='',
    layout=widgets.Layout(width='100%', height='25px'), # Adjust as needed
    style={'description_width': 'initial'}
)
def on_text_submit(change):
    output_label.value = model_predict_interface(change.new)
text_input.observe(on_text_submit, names='value')
display(text_input, output_label)

```

Finally, we interact by entering sentences that are classified in real time as correct or incorrect, as shown in *Figure 5.9*:



*Figure 5.9: A grammatically correct sentence*

The following is a grammatically incorrect example:

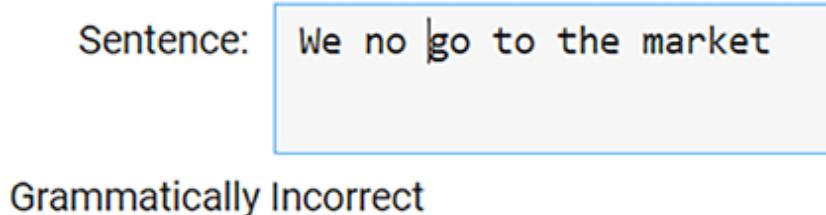


Figure 5.10: A grammatically incorrect sentence

Enter more examples to explore the scope and find the limits when the model is generalized.

We have now been through a complete training process and interacted with our trained model!

Let's now summarize our fine-tuning journey.

## Summary

In this chapter, we explored the process of fine-tuning a transformer model. We achieved this by implementing the fine-tuning process of a pretrained Hugging Face BERT model.

We began by analyzing the architecture of BERT, which only uses the encoder stack of transformers and uses bidirectional attention. 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 then configured a fine-tuning BERT model for an *acceptability judgment* downstream task. The fine-tuning process went through all phases of the process.

We installed the Hugging Face transformers and considered the hardware constraints, including selecting CUDA as the device for torch. We retrieved the CoLA dataset from GitHub. We loaded and created in-domain (training data) sentences, label lists, and BERT tokens.

The training data was processed with the BERT tokenizer and other data preparation functions, including the attention masks. Then, the data was split into training and validation sets. We selected a batch size to optimize the process and created an iterator.

We then loaded a Hugging Face BERT uncased base model. The process could have been implemented with another tokenizer and another Hugging Face pretrained model such as GPT-2, T5, RoBERTa, or others.

We grouped the parameters, defined the hyperparameters, and ran the training loop, including evaluations during the process.

Once the training was over, we used the out-of-domain holdout dataset to evaluate the output of the fined-tuned model. We explored the prediction process and implemented the Matthews correlation coefficient to evaluate the results.

Finally, we built an interface with `ipywidgets` to interact with our trained model.

Fine-tuning a pretrained model takes fewer machine resources than training a model for downstream tasks from scratch. Fine-tuned models can perform a variety of tasks. However, in some cases, fine-tuning a model is insufficient to reach our goals. For example, we might need to pretrain a transformer model practically from scratch.

In the next chapter, *Chapter 6, Pretraining a Transformer from Scratch through RoBERTa*, we will build and pretrain a transformer 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 **MLM**. (True/False)
6. BERT pretrains with **NSP**. (True/False)
7. BERT pretrains on 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, and Illia Polosukhin, 2017, Attention Is All You Need:*  
<https://arxiv.org/abs/1706.03762>
- *Alex Warstadt, Amanpreet Singh, 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/pretrained_models.html)
  - [https://huggingface.co/transformers/model\\_doc/bert.html](https://huggingface.co/transformers/model_doc/bert.html)
  - [https://huggingface.co/transformers/model\\_doc/roberta.html](https://huggingface.co/transformers/model_doc/roberta.html)
  - [https://huggingface.co/transformers/model\\_doc/distilbert.html](https://huggingface.co/transformers/model_doc/distilbert.html)

## Further reading

- *Vladislav Mosin, Igor Samenko, Alexey Tikhonov, Borislav Kozlovskii, and Ivan P. Yamshchikov, 2021, Fine-Tuning Transformers: Vocabulary Transfer:* <https://arxiv.org/abs/2112.14569>
- *Yi Tay, Mostafa Dehghani, Jinfeng Rao, William Fedus, Samira Abnar, Hyung Won Chung, Sharan Narang, Dani Yogatama, Ashish Vaswani, and Donald Metzler, 2022, Scale Efficiently: Insights from Pre-training and Fine-tuning Transformers:*  
<https://arxiv.org/abs/2109.10686>

## Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://www.packt.link/Transformers>



# 6

## Pretraining a Transformer from Scratch through RoBERTa

Sometimes, a pretrained model will not provide the results you expect. Even if the pretrained model goes through additional training through fine-tuning, it still will not work as planned. At that point, one approach is to initiate pretraining from scratch through platforms like Hugging Face to leverage architectures such as GPT and BERT, among others. Once you have pretrained a model from scratch, you will know how to train other models you might need for a project.

In this chapter, we will build a RoBERTa model, an advanced variant of BERT, from scratch. The model will use the bricks of the transformer construction kit we need for BERT models. Also, no pretrained tokenizers or models will be used. The RoBERTa model will be built following the 15-step process described in this chapter.

We will use the knowledge of transformers acquired in the previous chapters to build a model that can perform language modeling on masked tokens step by step. In *Chapter 2, Getting Started with the Architecture of the Transformer Model*, we went through the building blocks of the Original Transformer. In *Chapter 5, Diving into Fine-Tuning through BERT*, we explored how to fine-tune a pretrained model through a variation of a 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. It’s my project, named after Immanuel Kant, the philosopher. When you train and deploy a model in your environment, you can give it a “brand” to make your product unique.

KantaiBERT first loads a compilation of Immanuel Kant’s books created for this chapter. You will see how the data was obtained. You will also see how to create your own datasets for this notebook. KantaiBERT trains its own tokenizer from scratch. First, 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. We will then run the trained model KantaiBERT to perform an experimental downstream language modeling task and fill a mask, using Immanuel Kant’s logic.

Finally, we will put the knowledge you acquired in this chapter to work and pretrain a Generative AI customer support model on X (formerly Twitter) data. We will use a Kaggle dataset of support tweets of 20 major brands. You will discover how to build an open-source Generative AI chat agent prototype with a RoBERTa model. This opens the door to independent chat agents when necessary.

By the end of the chapter, you will know how to build a transformer model from scratch. In addition, you will have enough knowledge of transformers to face a situation where you must train a model to match a specific need.

This chapter covers the following topics:

- RoBERTa- and DistilBERT-like models
- Byte-level byte-pair encoding
- Training a tokenizer
- Defining the configuration of the model

- Initializing a model from scratch
- Exploring the parameters of a model
- Building a dataset
- Defining a data collator
- Initializing a trainer
- Pretraining the model
- Language modeling with FillMaskPipeline
- Applying the model to the downstream tasks of **Masked Language Modeling (MLM)**
- Pretraining a Generative AI chat agent based on an X (formerly Twitter) dataset



With all the innovations and library updates in this cutting-edge field, packages and models change regularly. Please go to the GitHub repository for the latest installation and code examples:

<https://github.com/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition/tree/main/Chapter06>.

You can also post a message in our Discord community (<https://www.packt.link/Transformers>) if you have any trouble running the code in this or any chapter.

Our initial step involves outlining the structure of the transformer architecture we intend to build and the tasks we aim to accomplish.

# 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 use in *Chapter 5, Diving into Fine-Tuning through BERT*.

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

KantaiBERT is a **Robustly Optimized BERT Pretraining Approach (RoBERTa)**-like model. RoBERTa is an advanced version of BERT designed by Meta (formerly Facebook).

The initial BERT models brought innovative features to the initial transformer models, as we saw in *Chapter 5, Diving into Fine-Tuning through BERT*. RoBERTa increases the performance of transformers for downstream tasks by improving the mechanics of the pretraining process.

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

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

KantaiBERT will be trained as a small model with 6 layers, 12 heads, and 83,504,416 parameters (this number might vary when the model is updated). It might seem that 83 million parameters is a lot. However, the parameters

are spread over 12 heads, which makes it a relatively small model. A small model will make the pretraining experience smooth so that each step can be viewed in real time, without waiting for hours to see a result.

KantaiBERT is a smaller version of RoBERTa, using a DistilBERT (a distilled version of BERT) architecture with 6 layers and 12 heads. As such, KantaiBERT runs much faster, but the results are slightly less accurate than with the full configuration of a RoBERTa model.

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

It is essential to show all the possible architectures, including running a small model on a smartphone. However, the future of transformers will also be ready-to-use APIs, as we will see in *Chapter 7, The Generative AI Revolution with ChatGPT*.

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

There are no token type IDs to indicate which part of a segment a token is a part of. Instead, the segments will be separated with the separation token

`</s>`.

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

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

# Building KantaiBERT from scratch

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

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

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

Let's start by loading the dataset.

## Step 1: Loading the dataset

Ready-to-use datasets provide an objective way to train and compare transformers. This chapter aims to understand the training process of a transformer with notebook cells that can be run in real time, without waiting for hours to obtain a result.

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

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

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

- *The Critique of Pure Reason*

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

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

The text file contains the raw text of the books:

```
...For it is in reality vain to profess _indifference_ in regard
```

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

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

```
#1. Load kant.txt using the Colab file manager  
#2. Downloading the file from GitHub  
!curl -L https://raw.githubusercontent.com/Denis2054/Transform
```

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

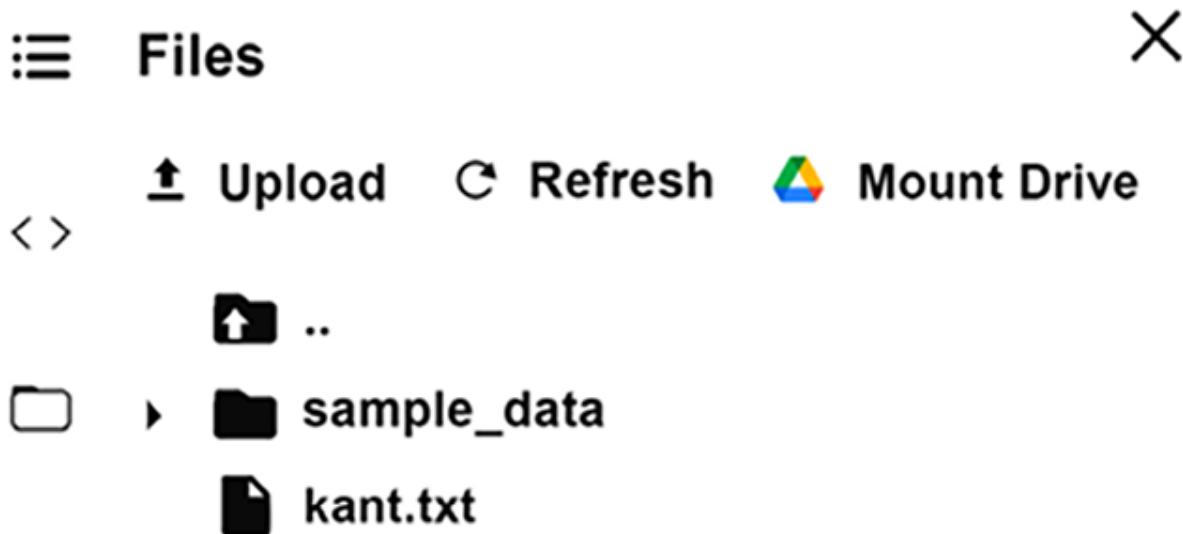


Figure 6.1: Colab file manager

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

The dataset is defined and loaded.

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

Now, the program will install the Hugging Face transformers.

## Step 2: Installing Hugging Face transformers

We will need to install Hugging Face transformers and tokenizers. We will also install an accelerator, as recommended by Hugging Face and summed up in *Step 2* of the notebook:

April 2023 update From Hugging Face Issue 22816:  
<https://github.com/huggingface/transformers/issues/22816>  
"The `PartialState` import was added as a dependency on the `transformers` package.  
Downgrading the `transformers` version removes the code which is

Denis Rothman: The following cell imports the latest version of Hugging Face's Transformers library. To adapt to the Hugging Face upgrade, A GPU accelerator was added to the code.

▶

If the `accelerate` module is not installed, an error message will appear before the training process can begin:

```
ImportError:Using the `Trainer` with `PyTorch` requires `accelerate`
```

◀

When installed, the module provides an abstraction layer that optimizes hardware accelerators (GPUs and TPUs). The `accelerate` module is practical when implementing large transformer models such as GPT and BERT.

The program installs Hugging Face Transformers and the `accelerate` module:

```
!pip install Transformers  
!pip install --upgrade accelerate  
from accelerate import Accelerator
```

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`, when trained using `kant.txt` (or any other text), utilizes a **Byte-Pair Encoding (BPE)** tokenizer. A BPE tokenizer breaks down strings or words into subword units or substrings. This approach offers several advantages, including the following:

- The tokenizer can break words into minimal components and then merge these components into statistically significant ones. For example, words like "smaller" and "smallest" can be represented as "small," "er," and "est." Furthermore, the tokenizer can go even further, generating subword parts like "sm" and "all." In essence, words are broken down into subword tokens and smaller units, such as "sm" and "all," rather than being represented as a single token like "small."
- The chunks of text classified as unknown, typically represented as "unk\_token," using WordPiece-level encoding, can effectively be minimized or eliminated.

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

- `files=paths` is the path to the dataset.
- `vocab_size=52_000` is the size of our tokenizer's model length.
- `min_frequency=2` is the minimum frequency threshold.
- `special_tokens=[]` is a list of special tokens.

In this case, the list of special tokens is:

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

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

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

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

The first step will be to tokenize the string:

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

The string is now tokenized into tokens with  $\hat{G}$ , which represents a whitespace (a space between words).

The next step is to replace them with their indices:

‘ $\hat{G}$ the’	‘ $\hat{G}$ token’	‘izer’
150	5430	4712

Table 6.1: Indices for the three tokens

The program runs the tokenizer as expected:

```
from pathlib import Path
from tokenizers import ByteLevelBPETokenizer
paths = [str(x) for x in Path(".").glob("**/*.txt")]
# Read the content from the files, ignoring or replacing invalid
file_contents = []
for path in paths:
    try:
        with open(path, 'r', encoding='utf-8', errors='replace
```

```
        file_contents.append(file.read())
    except Exception as e:
        print(f"Error reading {path}: {e}")
# Join the contents into a single string
text = "\n".join(file_contents)
# Initialize a tokenizer
tokenizer = ByteLevelBPETokenizer()
# Customize training
tokenizer.train_from_iterator([text], vocab_size=52_000, min_f
    "<s>",
    "<pad>",
    "</s>",
    "<unk>",
    "<mask>",
])

```

The tokenizer will skip the erroneous characters in the file and train the tokenizer.

The tokenizer is trained and ready to be saved.

## Step 4: Saving the files to disk

The tokenizer will generate two files when trained:

- `merges.txt`, which contains the merged tokenized substrings
- `vocab.json`, which contains the indices of the tokenized substrings

The program first creates the `KantaiBERT` directory and then saves the two files:

```
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:

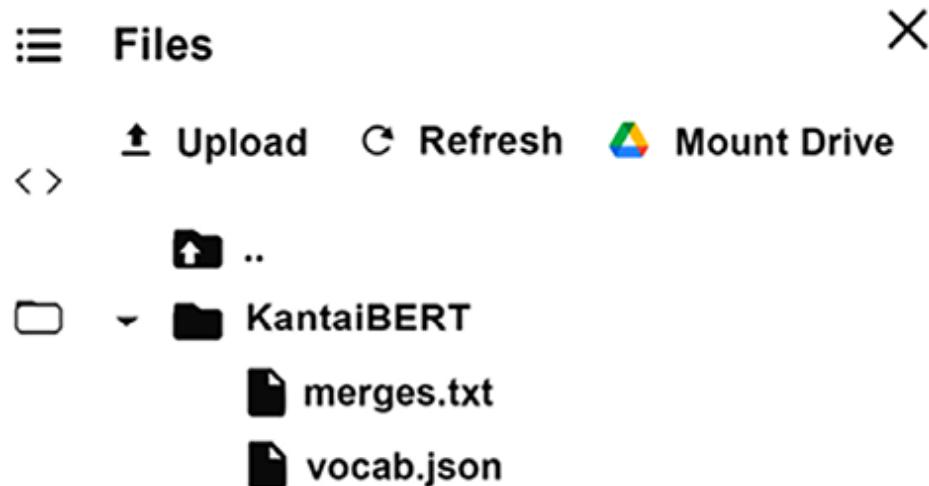


Figure 6.2: Colab file manager

The files in this example are small. You can double-click on them to view their content. `merges.txt` contains the tokenized substrings 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
```

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 are now ready to load the 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', 'GCritique', 'Gof', 'GPure', 'GReason', '.']
```

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 six tokens in the sequence:

```
Encoding(num_tokens=6, attributes=[ids, type_ids, tokens, offse
```

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 eight tokens:

```
Encoding(num_tokens=8, attributes=[ids, type_ids, tokens, offse
```

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 eight, 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 on which we are running the notebook.

## 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:

```
!nvidia-smi
```

The output displays the information and version on the card, depending on the machine you are working on. In this case, it was Tesla V100-SXM2:

```

+-----+
| NVIDIA-SMI 525.85.12      Driver Version: 525.85.12      CUDA Version: 12.0 |
+-----+
| GPU  Name      Persistence-M| Bus-Id      Disp.A  | Volatile Uncorr. ECC | | | |
| Fan  Temp     Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M.  |
| |          |          |              |                | Compute M.          |
| |          |          |              |                | MIG M.               |
+-----+
| 0  Tesla V100-SXM2... Off  | 00000000:00:04.0 Off |           0 |
| N/A   34C    P0    24W / 300W |          0MiB / 16384MiB |      0%     Default |
|                               |                           |             N/A |
+-----+
+-----+
| Processes:
| GPU  GI  CI      PID  Type  Process name          GPU Memory |
|       ID  ID                  |                 Usage  |
+-----+

```

*Figure 6.3: Information on the NVIDIA card*

The output may vary with each Google Colab VM configuration.

We will now check to make sure PyTorch sees CUDA:

```

import torch
torch.cuda.is_available()

```

The result should be True:

```
True
```

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

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:

```
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()`:

```
from transformers import RobertaTokenizer
from transformers import RobertaTokenizer
tokenizer = RobertaTokenizer.from_pretrained("./KantaiBERT", m
```

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:

```
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_aff
```

```
(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
                    (key): Linear(in_features=768, out_features=768,
                    (value): Linear(in_features=768, out_features=768
                    (dropout): Dropout(p=0.1, inplace=False)
                )
            (output): BertSelfOutput(
                (dense): Linear(in_features=768, out_features=768
                (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,
    )
    (output): BertOutput(
        (dense): Linear(in_features=3072, out_features=768,
        (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True
        (dropout): Dropout(p=0.1, inplace=False)
    )
)
)
....
```

Take some time to review the details of the configuration output before continuing. You will get to know the model from the inside.

The LEGO®-type building blocks of transformers make it fun to analyze. For example, you will note that dropout regularization is present throughout the sublayers, as in classical neural networks.

Now, let's explore the parameters.

# Exploring the parameters

Exploring the model's parameters can provide some additional insights into its architecture. The model is small and contains 83504416 parameters (the number might change with model updates).

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:

```
83504416
```

Let's now look at the parameters. We first store the parameters in LP and calculate the length of the list of them:

```
LP=list(model.parameters())
lp=len(LP)
print(lp)
```

The output shows that there are approximately 106 matrices and vectors, *which might vary from one transformer model (or update) to another*:

```
106
```

Now, let's display the 106 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.
       [ 0.0020, -0.0354, -0.0221, ..., 0.0220, -0.0060, -0.
       [ 0.0001, -0.0002, 0.0036, ..., -0.0265, -0.0057, -0.
       ...
       [-0.0125, -0.0418, 0.0190, ..., -0.0069, 0.0175, -0.
       [ 0.0072, -0.0131, 0.0069, ..., 0.0002, -0.0234, 0.
       [ 0.0008, 0.0281, 0.0168, ..., -0.0113, -0.0075, 0.
       requires_grad=True)
```

Take a few minutes to peek inside the parameters to understand how transformers are built.

The number of parameters is calculated by taking all parameters in the model and adding them up, for example:

- The vocabulary (52,000) x dimensions (768)
- The size of the vectors, which 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 **83,504,416** is reached.

The following cell displays the size of each of the 106 tensors in this model:

```
#Shape of each tensor in the model
LP = list(model.parameters())
for i, tensor in enumerate(LP):
    print(f"Shape of tensor {i}: {tensor.shape}")
```

The output will display the tensor and its size:

```
Shape of tensor 0: torch.Size([52000, 768])
Shape of tensor 1: torch.Size([514, 768])
Shape of tensor 2: torch.Size([1, 768])
Shape of tensor 3: torch.Size([768])
Shape of tensor 4: torch.Size([768])
Shape of tensor 5: torch.Size([768, 768])
Shape of tensor 6: torch.Size([768])
Shape of tensor 7: torch.Size([768, 768])
Shape of tensor 8: torch.Size([768])
Shape of tensor 9: torch.Size([768, 768])
Shape of tensor 10: torch.Size([768])
Shape of tensor 11: torch.Size([768, 768])
Shape of tensor 12: torch.Size([768])
Shape of tensor 13: torch.Size([768])
Shape of tensor 14: torch.Size([768])
Shape of tensor 15: torch.Size([3072, 768])
Shape of tensor 16: torch.Size([3072])...
```

Note that the numbers might vary depending on the version of the transformer module you use.

We can tentatively describe what the tensors represent, for example:

- `torch.Size([52000, 768])`: probably the embedding matrix containing the 52,000 words of the dictionary and 768-dimensional embedding.
- `torch.Size([514, 768])`: this could be the positional embeddings. RoBERTa accepts 512 tokens, to which we add the start and end tokens.
- `torch.Size([1, 768])`: this could be the embedding of a special token such as `<cls>`.

You can see that a transformer model is the mathematical representation of the sequences it will learn.

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` (`106`) number of elements in the list of parameters:

```
#counting the parameters
np=0
for p in range(0,lp):#number of tensors
```

The parameters are matrices and vectors of different sizes, for example:

- `768 x 768`
- `768 x 1`
- `768`

We can see that some parameters are two-dimensional, and some are one-dimensional.

An easy way to see if a parameter `p` in the list `LP[p]` has two dimensions or not is by doing the following:

```
PL2=True
try:
    L2=len(LP[p][0]) #check if 2D
except:
    L2=1 #not 2D but 1D
    PL2=False
```

If the parameter has two dimensions, its second dimension will be `L2>0` and `PL2=True (2 dimensions=True)`. If the parameter has only one dimension, its second dimension will be `L2=1` and `PL2=False (2 dimensions=False)`.

`L1` is the size of the first dimension of the parameter. `L3` is the size of the parameters defined by:

```
L1=len(LP[p])
L3=L1*L2
```

We can now add the parameters 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 parameter.
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
11 768 768 589824
12 768 768
13 768 768
14 768 768
15 3072 768 2359296
16 3072 3072
17 768 3072 2359296
18 768 768
19 768 768
20 768 768
21 768 768 589824
22 768 768
23 768 768 589824
24 768 768
25 768 768 589824
...
94 768 768
95 3072 768 2359296
96 3072 3072
97 768 3072 2359296
98 768 768
99 768 768
100 768 768
```

```
101 52000 52000
102 768 768 589824
103 768 768
104 768 768
105 768 768
83504416
```

The total number of parameters of the RoBERTa model is displayed at the end of the list:

```
83504416
```



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 parameters' content, and the parameters' size.



At this point, you will have a precise mental representation of the building blocks of the model.

The program now builds the dataset.

## Step 10: Building the dataset

The program will now load the dataset line by line to generate samples for batch training, with `block_size=128` limiting the length of an example:

```
%%time
from transformers import LineByLineTextDataset
dataset = LineByLineTextDataset(
    tokenizer=tokenizer,
    file_path='./kant.txt',
    block_size=128,
)
```

The output shows that Hugging Face has invested a considerable amount of resources in optimizing the time it takes to process data:

```
CPU times: user 25.9 s, sys: 375 ms, total: 26.3 s
Wall time: 26.9 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 collate samples from the dataset to prepare batch processing.

We are also preparing a batched sample process for MLM by setting

```
m1m=True.
```

We also set the number of masked tokens to train `m1m_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`:

```
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:

```
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,
)
```

Let's go through each line of the training arguments and the trainer:

1. `training_args = TrainingArguments(...)`: creating an instance of the `TrainingArguments`. Here, we're creating an instance of the `TrainingArguments` class to store the hyperparameters and other arguments for training:
  - `output_dir=". /KantaiBERT"`: the checkpoint directory.
  - `overwrite_output_dir=True`: the overwrite content in the output directory.
  - `Num_train_epochs=1`: the number of training epochs.
  - `per_device_train_batch_size=64`: the batch size of the number of examples for training. In this case, 64 examples will be trained at the same time.
  - `save_steps=10_000`: a checkpoint will be saved every 10,000 steps.
  - `save_total_limit=2`: the maximum number of checkpoints that can be saved before the old ones are deleted.

Note that batch processing on the device optimizes the architecture of the attention layer of the Original Transformer.

In *Chapter 2, Getting Started with the Architecture of the Transformer Model*, we defined attention as “Scaled Dot-Product Attention,” which is represented by the following equation, in which we plug  $Q$ ,  $K$ , and  $V$ :

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

Adding batches will increase the learning speed instead of processing one sequence at a time:

- `Q =[per_device_train_batch_size=64, d_model, d_k ]`

- `K =[per_device_train_batch_size=64, d_model, d_k ]`
- `V =[per_device_train_batch_size=64, d_model, d_k ]`

If necessary, take a few minutes to go back to *Chapter 2* to reread the attention layer section and review the role of each parameter, including  $Q$ ,  $K$ ,  $V$ ,  $d_{\text{model}}$ , and  $d_k$ .

Batch processing increases the performance of a model. However, a large batch might speed the process up but limit the learning quality, whereas smaller batches force the model to learn more relationships. Experiment with different batch sizes and measure the performance of the model.

2. `trainer = Trainer(...)`: creating the instance of the trainer class:

- `model=model`: the model that will be trained. In this case, the model is already defined earlier in our code, as shown when we defined the configuration in *Step 7* of the notebook:
  - `model_type: "roberta"`
  - `args=training_args`: passing the `TrainingArguments` object we created above.
  - `data_collator=data_collator`: The `data_collator` defined in step 11 of the notebook takes samples from the dataset and returns a batch.
  - `train_dataset=dataset`: the training dataset, which is `kant.txt` in this notebook.

The model is now ready for training.

## Step 13: Pretraining the model

Everything is ready. The trainer is launched with one line of code:

```
%%time  
trainer.train()
```

The output displays the training process in real time, showing the steps and loss:

Step	Training Loss
500	6.588800
1000	5.636300
1500	5.081400

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:

```
trainer.save_model("./KantaiBERT")
```

Click on **Refresh** in the file manager, and the files should appear:

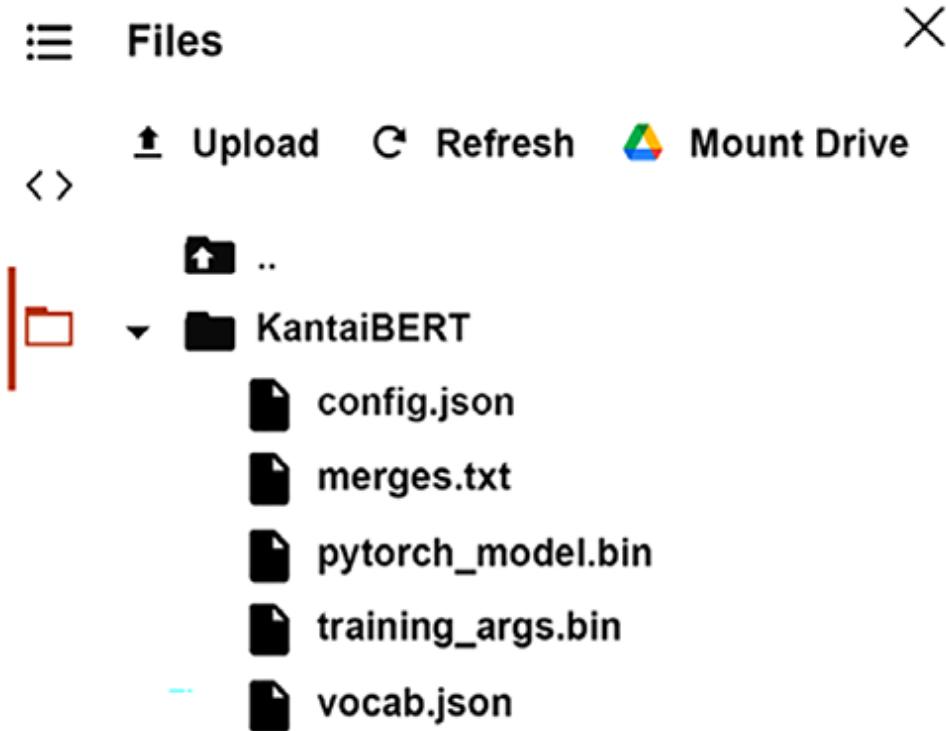


Figure 6.4: Colab file manager with all the KantaiBERT model files

`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 MLM:

```
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 the model is stochastic. Also, we are pretraining the model from scratch with a limited amount of data. However, the output obtained in this run is interesting because it introduces conceptual language modeling:

```
[{'score': 0.038927942514419556,
 'token': 393,
 'token_str': ' reason',
 'sequence': 'Human thinking involves human reason.'},
 {'score': 0.014990510419011116,
 'token': 446,
 'token_str': ' law',
 'sequence': 'Human thinking involves human law.'},
 {'score': 0.011723626405000687,
 'token': 418,
 'token_str': ' conception',
 'sequence': 'Human thinking involves human conception.'},
 {'score': 0.01163031067699194,
 'token': 531,
 'token_str': ' experience',
 'sequence': 'Human thinking involves human experience.'},
 {'score': 0.010935084894299507,
 'token': 600,
```

```
'token_str': 'understanding',
'sequence': 'Human thinking involves human understanding.']}]
```

The predictions might vary with each run and when Hugging Face updates its models.

However, the following output occurs often:

```
Human thinking involves human reason
```

The goal here was to see how to train a transformer model. We can see that interesting human-like predictions can be made.

These results are experimental and subject to variations during the training process. Therefore, 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.*  
Let's do that now by building a Generative AI model.

## Pretraining a Generative AI customer support model on X data

In this section, we will pretrain a Hugging Face `RobertaForCausalLM` model to be a Generative AI customer support chat agent for X (formerly Twitter). RoBERTa is an encoder-only model. As such, it is mainly designed to *understand and encode* inputs. In *Chapter 2, Getting Started with the*

*Architecture of the Transformer Model*, we saw how the encoder *learns to understand* and then sends the information to the decoder, generating content. However, in this section, we will use the Hugging Face functionality to adapt a RoBERTa model to run an autoregressive Generative AI task.

The experiment has limitations, but it shows the inner workings of content generation. The knowledge you acquired in this chapter through building a KantaiBERT from scratch will enable you to enjoy the ride!

The generative model and dataset are free, making the exercise particularly interesting. With some work, domain-specific Generative AI agents can help companies that want their own models.

Generative AI has spread exponentially since the arrival of ChatGPT. Your experience building a prototype chat agent in this section will provide insights into this new era of automated dialogs. We will not repeat all the details of the notebook we went through when building KantaiBERT. Run the notebook cell by cell using the experience gained in this chapter to make sure you don't miss anything. The titles of the steps in the notebook match the ones in this section.

Open `customer_Support_for_X.ipynb` in the chapter's directory on GitHub, and let's get to work!

We can get our Generative AI prototype working in only 10 steps.

## Step 1: Downloading the dataset

We will be training the model on Kaggle's *Customer Support on Twitter* dataset, which contains 2,800,000+ customer support tweets from the

biggest brands on Twitter, such as Apple, American Airlines, Amazon, and many more top corporations.

For more on this dataset, visit

<https://www.kaggle.com/datasets/thoughtvector/customer-support-on-twitter>.

You will need a Kaggle account and an API key and will have to activate the authentication. The dataset can then be downloaded with one line of code:

```
!kaggle datasets download -d thoughtvector/customer-support-on
```

Once the file is downloaded, it is unzipped:

```
import zipfile  
with zipfile.ZipFile('/content/customer-support-on-twitter.zip'  
    zip_ref.extractall('/content/')  
print("File Unzipped!")
```

## Step 2: Installing Hugging Face transformers

When installing the transformers library, note that you must also install the accelerator library to leverage the GPU's power.

## Step 3: Loading and filtering the data

For a prototype, we can fast-track preprocessing the tweets.

The raw format of the dataset contains noise we don't need to pretrain a language model, as shown by loading and printing the head of the DataFrame:

```
import pandas as pd
# Load the dataset
df = pd.read_csv('/content/twcs/twcs.csv')
# Check the first few rows to understand the data
print(df.head())
```

The output contains information we don't need to train the model to understand the relationship between the words of the tweets for this prototype:

```
    tweet_id  author_id  inbound          created_a
0           1  sprintcare   False  Tue Oct 31 22:10:47 +0000 201
1           2      115712    True  Tue Oct 31 22:11:45 +0000 201
2           3      115712    True  Tue Oct 31 22:08:27 +0000 201
3           4  sprintcare   False  Tue Oct 31 21:54:49 +0000 201
4           5      115712    True  Tue Oct 31 21:49:35 +0000 201
                                         text response_t
0  @115712 I understand. I would like to assist y...
1  @sprintcare and how do you propose we do that
2  @sprintcare I have sent several private messag...
3  @115712 Please send us a Private Message so th...
4  @sprintcare I did.
```

The most effective way of dealing with this irrelevant information is to extract words containing the characters we need, such as the alphabet and some punctuation. Run the cells up to this cell and study them:

```
import re
def filter_tweet(tweet):
```

```
# Keep only characters a to z, spaces, and apostrophes, then remove them
return re.sub(r'[^a-zA-Z\s\']', '', tweet.lower())
filtered_tweets = [filter_tweet(tweet) for tweet in tweets]
```

We extracted enough information to pretrain a prototype.

We will take the filtering further and only keep tweets that are at least 30 words:

```
f=30
filtered_tweets = [tweet for tweet in filtered_tweets if len(tweet) >= f]
```

This way, we will force the model to learn long-term dependencies in sentences with sufficient content, as shown in the output, which contains long sequences of words:

```
ags which only contain certain items such as unwrapped food raw
hi you can change your microsoft account email through the site
```

Run the following cells to save and display information on the data.

## Step 4: Checking Resource Constraints: GPU and CUDA

We need a GPU to pretrain this model for PyTorch. Make sure to activate a GPU, and then check your system with the following code:

```
!nvidia-smi
```

You should see a GPU in the following excerpt of the output:

```
NVIDIA-SMI 525.105.17    Driver Version: 525.105.17    CUDA Versi
```

Also, make sure PyTorch sees CUDA:

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

The output should be:

```
True
```

## Step 5: Defining the configuration of the model

The main line of the configuration cell is:

```
from transformers import RobertaConfig, RobertaForCausalLM
```

`RobertaForCausalLM` can be used for Generative AI, which constitutes a valuable asset for those who want to pretrain their models.

Another feature to focus on is the configuration of the model:

```
config = RobertaConfig(
    vocab_size=52_000,
```

```
    max_position_embeddings=514,  
    num_attention_heads=12,  
    num_hidden_layers=6,  
    type_vocab_size=1,  
    is_decoder=True, # Set up the model for potential seq2seq  
)  
▶
```

`is_decoder=True` adds autoregressive functionality to the RoBERTa model, often implemented with masks. A mask can be anywhere in a sequence of words. An autoregressive model continues a sequence from the last token onward.

We can now define our Generative AI model:

```
# Create the RobertaForCausalLM model with the specified config  
model = RobertaForCausalLM(config=config)  
print(model)  
▶
```

We will use Hugging Face's RoBERTa tokenizer:

```
tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
```

The tokenizer contains special tokens, but we will not need the mask, for example.

## Step 6: Creating and processing the dataset

We now install Hugging Face's datasets library:

```
#installing Hugging Face datasets for data loading and preproc  
!pip install datasets
```

What does it do? Everything we need in no time. The library:

- Loads the dataset:

```
#load dataset  
from datasets import load_dataset  
dataset = load_dataset('csv', data_files='/content/model/d
```

- Splits the dataset into a training and evaluation set:

```
# split datasets into train and eval  
from datasets import DatasetDict  
dataset = dataset['train'].train_test_split(test_size=0.1)  
dataset = DatasetDict(dataset)
```

- Tokenizes the dataset with useful rules:

If a record's length is less than “`max_length`,” it's padded to ensure all records have the same length. If a record's length exceeds “`max_length`,” it's truncated to the specified max length. The code is compact, padding and truncating all in one line:

```
def tokenize_function(examples):  
    return tokenizer(examples["text"], padding="max_length")  
tokenized_datasets = dataset.map(tokenize_function, batched=True)
```

- Uses the data collator to batch items together:

```
The data collator is creating a few lines of code:
# Define the data collator
data_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer,
    mlm=False # For causal (autoregressive) language m
)
```

Note that `mlm=False` turns MLM off for autoregressive, generative language modeling.

## Step 7: Initializing the trainer

We first create a class to check if the “step” key exists in the logs and, if it does, prints the current time when every `eval_steps` number of steps is reached during training:

```
class CustomTrainer(Trainer):
    def log(self, logs: Dict[str, Any]) -> None:
        super().log(logs)
        if "step" in logs: # Check if "step" key is in the log
            step = int(logs["step"])
            if step % self.args.eval_steps == 0:
                print(f"Current time at step {step}: {datetime
```

Then, we define the training arguments:

```
training_args = TrainingArguments(
    output_dir="/content/model/model/",
    overwrite_output_dir=True,
    num_train_epochs=2, # can be increased to
```

```
        per_device_train_batch_size=64,          # batch size per device
        save_steps=10_000,                      # save a checkpoint every 10,000 steps
        save_total_limit=2,                     # the maximum number of checkpoints to save
        logging_dir='/content/model/logs/',    # directory for storing logs
        logging_steps=100,                      # Log every 100 steps
        logging_first_step=True,                # Log the first step
        evaluation_strategy="steps",           # Evaluate every "evaluation_steps"
        eval_steps=500,                        # Evaluate every 500 steps
    )

```

The `TrainingArguments` object is used to set various training parameters for the Hugging Face transformers library, which specifies that the model should be saved in the `" /content/model/model/"` directory and that any existing outputs in this directory should be overwritten. The training will run for two epochs with a batch size of 64 per device. Logging settings determine that logs will be saved in `" /content/model/logs/"`, they will be printed every 100 steps, and evaluation will occur every 500 steps. The model will save a checkpoint twice.

Finally, we define the `trainer` in a few lines referring to the steps we ran up to this cell:

```
trainer = CustomTrainer(
    model=model,
    args=training_args,
    data_collator=data_collator,
    train_dataset = tokenized_datasets["train"],
    eval_dataset = tokenized_datasets["test"]
)
```

## Step 8: Pretraining the model

`trainer.train()` activates the pretraining process.

The progress bar information displays the critical information to monitor to improve the model between each run. The information can change between runs due to the stochastic nature of transformer models.

For example:

- -218/3,216: completed 218 tasks out of 3,216
- -01:24: time elapsed since the start
- -< 19:39: approximate time remaining
- 2.54 it/s: processing speed of 2.54 iterations per second
- Epoch 0.07/1: currently at 7% of 1 full epoch

Under the progress bar, we can visualize:

- `step`: number of steps
- `training loss`: must diminish
- `validation loss`: should be inferior or converge with the training loss.  
If it is superior, the model may be overfitting.

## Step 9: Saving the model

You can save the model to run a standalone session to chat with the Generative AI agent once it's trained:

```
trainer.save_model("/content/model/model/")
```

Google Colab will recycle the model when the session is closed. You can save it to any location you wish, for example, in Google Drive:

```
#Uncomment the following line to save the output for future use  
#trainer.save_model("drive/MyDrive/files/model_C6/model/")
```

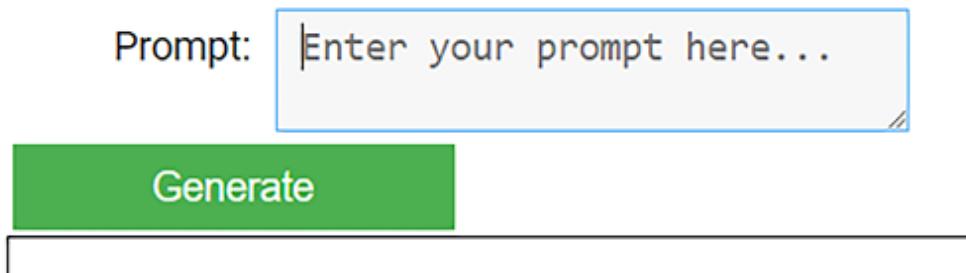
# Step 10: User interface to chat with the Generative AI agent

After installing `ipywidgets` to build the interface and making sure that the model and tokenizer are initialized, we can evaluate the prototype by creating a function to generate content:

```
!pip install ipywidgets
import ipywidgets as widgets
from IPython.display import display, clear_output
from transformers import RobertaTokenizer, RobertaForCausalLM
# Define the function to generate response
def generate_response(prompt):
    # Reload model and tokenizer for each request
    tokenizer = RobertaTokenizer.from_pretrained('roberta-base')
    model = RobertaForCausalLM.from_pretrained(model_path)
    inputs = tokenizer(prompt, return_tensors="pt", max_length=max_length)
    output = model.generate(**inputs, max_length=max_length, temperature=temperature)
    generated_text = tokenizer.decode(output[0], skip_special_tokens=True)
    return generated_text
# Create widgets
text_input = widgets.Textarea(
    description='Prompt:',
    placeholder='Enter your prompt here...'
)
button = widgets.Button(
    description='Generate',
    button_style='success'
)
output_text = widgets.Output(layout={'border': '1px solid black'})
# Define button click event handler
def on_button_clicked(b):
    with output_text:
        clear_output()
        response = generate_response(text_input.value)
        print(response)
```

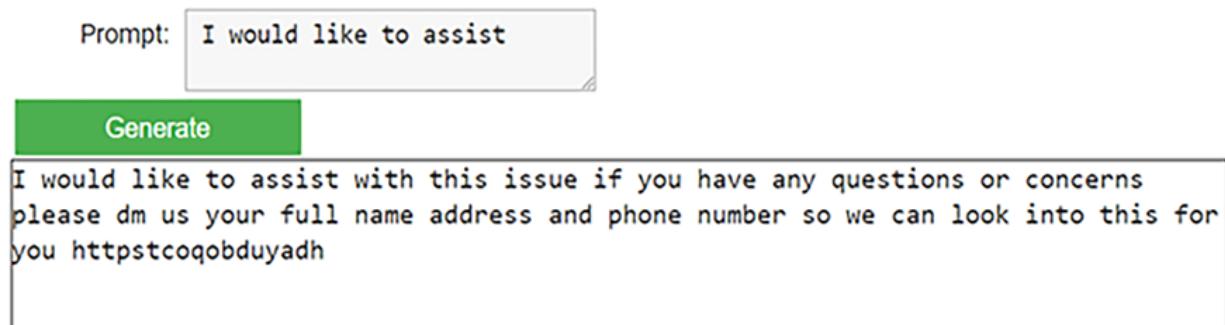
```
button.on_click(on_button_clicked)
# Display widgets
display(text_input, button, output_text)
```

We can now enter a prompt, as shown in *Figure 6.5*:



*Figure 6.5: User Interface*

We can try a prompt and click on **Generate** to display the Generative AI output, as shown in *Figure 6.6*:



*Figure 6.6: Prompt and generated response*

In this case, the output is experimental and interesting. The last part could be filtered with a dictionary of known words. However, Generative AI is stochastic and can produce errors.

Try input close to the trained samples first, for example, with no punctuation:

- I would like to assist you
- I need help to find the battery of
- I would like to know why they moved us

Further training is required if you wish to go further.

## Further pretraining

The model was trained on a subset of the original dataset for a limited number of epochs. To expand the pretraining process, modify the dataset filter (`f=30` in *Step 3*), increase the number of epochs, try other models, modify the hyperparameters, increase the size/quality of the datasets, and more.

Although running a reasonably sized model is exciting, there are limitations.

## Limitations

This is only a functional interface **Proof of Concept (POC)** to verify that this approach works. RoBERTa wasn't designed for Generative AI, but Hugging Face provides the environment to implement it. It isn't a native Generative AI model approach. However, who knows? The knowledge of fully trained RoBERTa can be leveraged by transferring its knowledge to a decoder model through transfer learning bridges. We opened a door in this section that you can choose to explore further or not, depending on the goals of your project.

*You're entering the challenging but exciting world of Generative AI designers!*

We created an open-source Generative AI model that can be considerably expanded. Thanks to transformers, we are only at the beginning of a new era

of AI!

## Next steps

You have trained two transformers 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. Many other Hugging Face models are available for training in the BERT family, GPT models, T5, and more!

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](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 `KantaBERT`, a RoBERTa-like model transformer, from scratch using the building blocks provided by Hugging Face.

We first started by loading a customized dataset on a specific topic related to the works of Immanuel Kant. Depending on your goals, you can load an

existing dataset or create your own. We saw that using a customized dataset provides insights into how a transformer model thinks. However, this experimental approach has its limits. Training a model beyond educational purposes would take a much larger dataset.

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.

We saved the model and loaded it for a downstream language modeling task. The goal was to fill the mask using Immanuel Kant's logic.

Finally, we pretrained a Generative AI customer support model on X (formerly Twitter). We used an open-source RoBERTa model and a free dataset. This prototype showed that we can build chat agents independently for specific domains when necessary.

The door is now wide open for you to experiment on existing or customized datasets to see what results you get. You can share your model with the Hugging Face community. Transformers are data-driven. You can use this to your advantage to discover new ways of using transformers.

You are now ready to learn how to run ready-to-use transformer engines with APIs that require no pretraining or fine-tuning.

*Chapter 7, The Generative AI Revolution with ChatGPT*, will take you into the world of suprahuman LLMs. And with the knowledge of this chapter and the past chapters, you will be ready!

# 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 six decoder layers. (True/False)
8. MLM predicts a word contained in a mask token in a sentence. (True/False)
9. A BERT-like model has no self-attention sublayers. (True/False)
10. Data collators are helpful for backpropagation. (True/False)

# References

- Hugging Face Tokenizer documentation:  
[https://huggingface.co/transformers/main\\_classes/tokenizer.html?highlight=tokenizer](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](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/hugging>

[face/blog/blob/master/notebooks/01 how to train.ipynb](https://huggingface.co/blog/blob/master/notebooks/01_how_to_train.ipynb)

- More on BERT:

[https://huggingface.co/transformers/model\\_doc/bert.html](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](https://huggingface.co/transformers/model_doc/roberta.html)

- Even more on DistilBERT:

[https://huggingface.co/transformers/model\\_doc/distilbert.html](https://huggingface.co/transformers/model_doc/distilbert.html)

## Further reading

- *Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova, 2018, Pretraining of Deep Bidirectional Transformers for Language Understanding: <https://arxiv.org/abs/1810.04805>*
- *Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov, RoBERTa: A Robustly Optimized BERT Pretraining Approach: <https://arxiv.org/abs/1907.11692>*

## Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://www.packt.link/Transformers>



# 7

## The Generative AI Revolution with ChatGPT

In November 2022, OpenAI ChatGPT entered mainstream media with a resonating big bang. Newspapers, television channels, and social media raced to OpenAI's ChatGPT website. Rumors of what ChatGPT could do spread like wildfire. Since then, OpenAI has continually updated its platform.

AI history is on a roll!

This chapter continues the journey of this book and takes the reader further down the path of learning about the ever-growing power of transformers. The theory and practical knowledge acquired through the previous chapters provide enough transformer model expertise to enjoy the ride through the cutting-edge models in this chapter.

We will first go through the tremendous improvements and diffusion of ChatGPT models in the everyday lives of developers and end users. We will see that GPT is a General Purpose Technology: GPTs are GPTs that can be applied to a wide range of domains.

This chapter will then examine the improvements in the architecture of the Generative AI GPT models. We will investigate the zero-shot challenge of

using trained transformer models with little to no fine-tuning of the model's parameters for downstream tasks.

The tremendous impact of generative transformer assistants will be explored. We will then get started with OpenAI models as assistants.

Then, we will get started with the GPT-4 API, the hyperparameters, and implement several NLP examples.

Finally, we will learn how to obtain better results with **Retrieval Augmented Generation (RAG)**. We will implement an example of automated RAG with GPT-4.

By the end of the chapter, you will have fully understood the many perspectives OpenAI's transformer models open for you.

This chapter covers the following topics:

- The rise and diffusion of GPT models as a General Purpose Technology
- The architecture of OpenAI GPT models
- The path from few-shot to one-shot models
- Getting started with ChatGPT as an assistant
- Getting started with the OpenAI GPT-4 API
- Implementing an introductory automated RAG process for GPT-4



With all the innovations and library updates in this cutting-edge field, packages and models change regularly. Please go to the GitHub repository for the latest installation and code examples:

<https://github.com/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition/tree/main/Chapter07>.

You can also post a message in our Discord community (<https://www.packt.link/Transformers>) if you have any trouble running the code in this or any chapter.

Let's begin our journey by exploring GPTs as **GPTs (General Purpose Technologies)**.

## GPTs as GPTs

This chapter unveils the power of generative artificial intelligence models. **GPT (Generative Pre-trained Transformers)** reveals its meaning with ChatGPT models, such as GPT 3.5-Turbo and GPT-4. There are 50+ variants of the main OpenAI GPT models. Among those variants, some will be regularly deprecated, and new ones will appear.

The evolution of OpenAI models has confirmed that Generative AI transformers, such as GPTs, are GPTs as defined by *Elouondou et al. (2023)*.

Once an innovation becomes a general-purpose technology, it spreads to thousands of applications, like the invention of electricity, combustion engines, and computers. As such, Generative Pre-trained Transformers technology, often referred to as Generative AI, has begun to spread to thousands of applications in many domains.

In this section, we will approach the maze of OpenAI models, Generative AI, and applications from the perspective of a general-purpose technology. We will focus on two main aspects: *improvement* and *diffusion*.

Let's begin with the improvements.

## Improvement

In this chapter, the focus of improvement will be on the architecture of OpenAI transformer models. The focus will be on four improvements: decoder only, scale, task generalization, and new terminology:

- **Decoder only**

The Original Transformer described in *Chapter 2, Getting Started with the Architecture of the Transformer Model*, contained an encoder and decoder stack. *Chapter 5, Diving into Fine-Tuning through BERT*, introduced BERT, an encoder-only stack. This chapter will present a decoder-only stack. You may ask yourself what mathematical chain of logic or proof leads to choosing one of these configurations. There is none!

The development of transformer models involves empirical data-driven insights, hardware constraints, and evaluations. This explains why they will constantly evolve through the intuitive and creative minds of their architects.

- **Scale**

Scale remains a critical feature in transformers. GPT models have increased in size, as you will discover in this section. Why? The goal is to capture the many dependencies between words and contexts. One word can have many different meanings depending on the context. For example, the verb *eat* seems simple. Right? But we quickly discover that somebody can *eat* something, or something can be *eaten*, that somebody might want *to eat*, *not eat*, or *maybe eat*. The list is nearly endless! We can create many parameters to express these subtleties.

The issue then becomes finding the right number of parameters. Too many parameters might be costly and useless. Too few parameters

might reduce accuracy. The right number of parameters will be obtained through trial and error.

- **Task generalization**

If a model is trained to a specific task, it's task-specific, such as the model we fine-tuned and trained in *Chapter 5, Diving into Fine-Tuning through BERT*. We provide an input and get output such as a word, a classification, an answer to a question, and other NLP tasks.

However, when faced with potentially hundreds of tasks, we can't imagine creating hundreds of task-specific models! That is where Generative AI models such as OpenAI GPT models fit in.

You can sum this up as, "We start a sentence, and the model finishes it." For example, you might say to a friend, "Give me the recipe for your delicious apple pie, please."

Think about it. What do you expect? A list of cars? Instructions to build a house? No, you expect a list of ingredients and instructions to cook an apple pie. Transformers have been trained on huge datasets and will statistically determine the following word(s).

You started the sentence by designing the "prompt," the other person or GPT will just continue the sentence by producing the "response."

You can easily imagine thousands of tasks you could create by simply beginning a sentence with a clear context and waiting for GPT to continue it! You just entered the world of general-purpose, Generative AI.

- **New terminology**

New words appear with the latest technology, such as LLMs, Generative AI, and Foundation Models.

Do not let yourself be overwhelmed by these terms. Just get used to them and the concepts they represent like any other new words. For example, OpenAI GPT models now have billions of parameters to process natural language. They are thus “Large Language Models.”

A GPT model can continue a sentence, a “prompt,” which explains why they are Generative AI models. GPT models can process words, images, and sounds. We can build hundreds of tasks on their capabilities. This makes them Foundation Models that we can build other systems with.

Keep these takeaways in mind when you work your way through this chapter.

Orientating ourselves in the *diffusion* of GPTs is equally essential.

## Diffusion

The diffusion of ChatGPT models can be explained by the new application sectors and their pervasiveness.

## New application sectors

New application sectors are typical of general purpose technology. There is no need to feel overtaken or overwhelmed. Once electricity was invented, thousands of products and services appeared on the market. As these applications spread, general purpose technology is continually evolving and improving.

This chapter will review some of the main products and services powered by OpenAI models.

We can break this list down into self-service assistant and development assistant applications.

## **Self-service assistants**

Self-service interfaces require no development and no machine learning knowledge at all:

- ChatGPT online to chat, create images, analyze data, and more.
- ChatGPT online to provide text instructions.
- OpenAI Playground: for general-purpose NLP tasks.
- Microsoft Office 365 Copilot spreading to many applications.
- New Bing to chat, give instructions, obtain responses, or use as a search engine.

## **Development assistants**

Development assistants have become unavoidable game changers, such as the following tools:

- ChatGPT and OpenAI Playground as a code assistant.
- GitHub Copilot: code assistant (in an application).
- New Bing as a code assistant.

We can further state that the new application sectors are *assistants* for any user in many applications, whether to develop AI applications, classical applications, or for everyday usage.

## **Pervasiveness**

Do not underestimate pervasiveness. Pervasiveness pushes innovations out of the shadows into everyday life. An invention only becomes an innovation

when it exerts an impact on society. Then, it often spills over into many domains at an inevitable exponential speed under the pressure of demand and competition.

In the case of transformers, *early adopters* are the ones who started working on transformers as early as November 2017 when transformers emerged, as explained in *Chapter 1, What Are Transformers?*

Those who accept a technological advance through imitation (media, social media, mainstream resources) are *new adopters*.

The big bang of *new adopters* began in November 2022 when OpenAI made ChatGPT available to mainstream end users and AI professionals. The number of new adopters increases through a classical imitation effect of general purpose technologies.

If we break down the number of new adopters from the imitation effect, we get the following:

- New adopters who are just curious enough to persuade others to join the trend.
- New adopters who use Generative AI, including ChatGPT, for the fun of it and keep up with the pace of technology.
- New adopters who begin to create prompts that generate content that boosts their productivity.
- New adopters who know nothing about ML but are creating functions and applications that compete with AI professionals.
- New adopters who are AI professionals boost their productivity with LLMs.

We are living in a typical acceleration of the diffusion of AI as general-purpose technology through Generative AI such as GPTs.

Companies will not take long to realize they do not need a data scientist or an AI specialist to start an NLP project with the tools available in every key application.

So why bother?

The answer to these questions is quite simple. It's easy to start a GPT-3 or a GPT-4 engine, just like starting a Formula 1 or Indy 500 race car. No problem. But then, driving such a car is nearly impossible without months of training! Generative AI engines, such as GPT-3 and GPT-4 are powerful AI race cars. You can get them to run in a few clicks. However, leveraging their incredible horsepower requires the knowledge you have acquired from the beginning of this book and what you will discover in the following chapters!

We will now go through the architecture that made transformers so popular.

## The architecture of OpenAI GPT transformer models

In 2020, *Brown et al.* (2020) described the training of an OpenAI GPT-3 model containing 175 billion parameters that was trained on huge datasets, such as the 400 billion byte-pair-encoded tokens extracted from Common Crawl data. OpenAI ran the training on a Microsoft Azure supercomputer with 285,00 CPUs and 10,000 GPUs.

The machine intelligence of OpenAI's GPT-3 models and their supercomputer led *Brown et al.* (2020) to zero-shot experiments. The idea was to use a trained model for downstream tasks without further training the parameters. The goal would be for a trained model to go directly into multi-task production with an API that could even perform tasks it wasn't trained for.

The era of suprahuman cloud AI models was born. OpenAI's API requires no high-level software skills or AI knowledge. You might wonder why I use the term “suprahuman.” GPT-3 and the GPT-4 model (and soon more powerful ones) can perform many tasks at least as well as a human in many cases. For the moment, *it is essential to understand how GPT models are built and run to appreciate the magic.*

GPT-4 is built on GPT-3, which in turn is built on the GPT-2 architecture. However, a fully trained GPT-3 transformer is a Foundation Model:

- A Foundation Model can do many tasks it wasn't trained for through emergence.
- Through *homogenization*, GPT-3/GPT-4 generative abilities through a unified architecture apply to many NLP tasks, including programming tasks.

Transformers went from training to fine-tuning and finally to zero-shot models in less than three years between the end of 2017 and the first part of 2020. A zero-shot GPT-3 transformer model requires no fine-tuning. The trained model parameters are not updated for downstream multi-tasks, which opens a new era for NLP/NLU tasks.

In this section, we will first learn about the OpenAI team's motivation for designing GPT models.

We will first go through the creation process of the OpenAI team.

## **The rise of billion-parameter transformer models**

The speed at which transformers went from small models trained for NLP tasks to models that require little to no fine-tuning is staggering.

*Vaswani et al.* (2017) introduced the Original Transformer, which surpassed CNNs and RNNs on BLEU tasks. *Radford et al.* (2018) introduced the GPT model, which could perform downstream tasks with fine-tuning. *Devlin et al.* (2019) perfected fine-tuning with the BERT model. *Radford et al.* (2019) went further with GPT-2 models. *Brown et al.* (2020) defined a GPT-3 zero-shot approach to transformers that does not require fine-tuning!

At the same time, *Wang et al.* (2019) created GLUE to benchmark NLP models. But transformer models evolved so quickly that they surpassed human baselines!

*Wang et al.* (2019<sup>2</sup>) rapidly created SuperGLUE, set the human baselines much higher, and made the NLU/NLP tasks more challenging. Transformers are rapidly progressing, and some have already surpassed Human Baselines on the SuperGLUE leaderboards at the time of writing.

How did this happen so quickly?

We will look at one aspect, the models' sizes, to understand how this evolution happened.

## The increasing size of transformer models

From 2017 to 2020 alone, the number of parameters increased from 65M parameters in the Original Transformer model to 175B parameters in the GPT-3 model, as shown in *Table 7.1*:

Transformer Model	Paper	Parameters
Transformer Base	<i>Vaswani et al.</i> (2017)	65M

Transformer Big	<i>Vaswani et al. (2017)</i>	213M
BERT-Base	<i>Devlin et al. (2019)</i>	110M
BERT-Large	<i>Devlin et al. (2019)</i>	340M
GPT-2 Small	<i>Radford et al. (2019)</i>	117M
GPT-2 Medium	<i>Radford et al. (2019)</i>	345M
GPT-2 Large	<i>Radford et al. (2019)</i>	1.5B
GPT-3	<i>Brown et al. (2020)</i>	175B
GPT-3.5	OpenAI (March 2022)	175B
GPT-4	OpenAI (March 2023)	-
GPT-4V	OpenAI (September 2023)	

*Table 7.1: The evolution of the number of transformer parameters*

*Table 7.1* only contains the main models designed during that short time. The dates of the publications come after the date the models were actually designed. Also, the authors updated the papers. For example, once the Original Transformer set the market in motion, transformers emerged from Google Brain and Research, OpenAI, and Facebook AI, which all produced new models in parallel.

The number of parameters has increased significantly in a relatively short time, reaching 175 billion parameters for GPT-3. GPT-3.5 and GPT-3.5-turbo have 175B parameters.

Information on the architecture of GPT-4 models is scarce. OpenAI has not officially disclosed the details of GPT-4's architecture. However, they optimized the system and obtained high scores on well-known exams and evaluations, as shown in the following excerpt of the *GPT-4 Technical Report*, March 23, 2023, page 5:

Exam	GPT-4	GPT-4 (no vision)	GPT-3.5
Uniform Bar Exam (MBE+MEE+MPT)	298 / 400 (~90th)	298 / 400 (~90th)	213 / 400 (~10th)
LSAT	163 (~88th)	161 (~83rd)	149 (~40th)
SAT Evidence-Based Reading & Writing	710 / 800 (~93rd)	710 / 800 (~93rd)	670 / 800 (~87th)
SAT Math	700 / 800 (~89th)	690 / 800 (~89th)	590 / 800 (~70th)
Graduate Record Examination (GRE) Quantitative	163 / 170 (~80th)	157 / 170 (~62nd)	147 / 170 (~25th)
Graduate Record Examination (GRE) Verbal	169 / 170 (~99th)	165 / 170 (~96th)	154 / 170 (~63rd)
Graduate Record Examination (GRE) Writing	4 / 6 (~54th)	4 / 6 (~54th)	4 / 6 (~54th)

Figure 7.1: GPT model performance on specific exams

GPT-4 (expanded into GPT-4V) is multi-modal (language and vision). We will discover transformers for computer vision starting in *Chapter 16, Beyond Text: Vision Transformers in the Dawn of Revolutionary AI*.

ChatGPT is an umbrella term that covers GPT-3.5-turbo, GPT-4, GPT-4V, and possible future improvements.

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 model 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 davinci version of 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.

However, the drawback of context size is the distance it takes to understand what a word refers to. The path taken to analyze long-term dependencies requires changing from recurrent to attention layers.

The following sentence requires a long path to find what 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*.<sup>1</sup>”

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!

*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. All the layers are identical, making it much easier to scale up the size of transformer models.

The flexible and optimized architecture of transformers has led to an impact on several other factors:

- *Vaswani et al.* (2017) trained a state-of-the-art transformer model with 36M sentences. *Brown et al.* (2020) trained a GPT-3 model with 400 billion byte-pair-encoded tokens extracted from Common Crawl data.
- Training large transformer models requires machine power that is only available to a few teams worldwide. It took a total of  $2.14 \times 10^{23}$  FLOPS for *Brown et al.* (2020) to train GPT-3 175B.
- Designing the architecture of transformers requires highly qualified teams that can only be funded by a small number of organizations worldwide.

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

We will now see how zero-shot models were achieved.

## From fine-tuning to zero-shot models

From the start, OpenAI’s research teams, led by *Radford et al.* (2018), wanted to take transformers from former 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 trained transformers to learn a language.



Note: The term *unsupervised* defines training with no labels. In that sense, GPT models go through *unsupervised* training. However, when it predicts a token during training, the output is compared to the actual complete input sequence to calculate the loss, find the gradients, and perform backpropagation. In this sense, GPT is *rather self-supervised* than totally *unsupervised*. Keep this in mind when you see the term unsupervised associated with generative models.

OpenAI wanted to create a task-agnostic model. So, they began to train transformer models on raw data instead of relying on labeled data by specialists. Labeling data is time-consuming and considerably slows down the transformer's training process.

The first step was to start with unsupervised training in a transformer model. Then, they would only fine-tune the model's supervised learning.

OpenAI opted for a decoder-only transformer described in the stacking decoder layers section. The metrics of the results were convincing and quickly reached the level of the best NLP models of fellow NLP research labs.

The promising results of the first version of GPT transformer models soon led *Radford et al. (2019)* to come up with zero-shot transfer models. The core of their philosophy was to continue training GPT models to learn from

raw text. They then took their research a step further, focusing on language modeling through examples of unsupervised distributions.

The goal was to generalize this concept to any downstream task once the trained GPT model understands a language through intensive training.

The GPT models rapidly evolved from 117M parameters to 345M parameters, to other sizes, and then to 1,542M parameters. 1,000,000,000+ parameter transformers were born. The amount of fine-tuning was sharply reduced.

This encouraged OpenAI to go further, much further. *Brown et al.* (2020) went on the assumption that conditional probability transformer models could be trained in depth and were able to produce excellent results with little to no fine-tuning for downstream tasks.

OpenAI was reaching its goal of training a model and running downstream tasks directly without further fine-tuning. This phenomenal progress can be described in four phases:

- **Fine-Tuning (FT)** is meant to be performed in the sense we have been exploring in previous chapters. A transformer model is trained and then fine-tuned on downstream tasks. *Radford et al.* (2018) designed many fine-tuning tasks. The OpenAI team then progressively reduced the number of tasks to 0 in the following steps.
- **Few-Shot (FS)** represents a huge step forward. The GPT is trained. When the model needs to make inferences, it is presented with demonstrations of the task to perform as conditioning. Conditioning replaces weight updating, which the GPT team excluded from the process. We will apply conditioning to our model through the context we provide to obtain text completion in the notebooks we will go through in this chapter.

- **One-Shot (1S)** takes the process further. The trained GPT model is presented with only one demonstration of the downstream task to perform. No weight updating is permitted, either.
- **Zero-Shot (ZS)** is the ultimate goal. The trained GPT model is presented with no demonstration of the downstream task to perform.

Each of these approaches has various levels of efficiency. The OpenAI GPT team has worked hard to produce these state-of-the-art transformer models.

We can now explain the motivations that led to the architecture of the GPT models:

- Teaching transformer models how to learn a language through extensive training.
- Focusing on language modeling through context conditioning.
- The transformer takes the context and generates text completion in a novel way. Instead of consuming resources on learning downstream tasks, it works on understanding the input and making inferences no matter what the task is.
- Finding efficient ways to train models by masking portions of the input sequences forces the transformer to think with machine intelligence. Thus, machine intelligence, though not human, is efficient.

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

## Stacking decoder layers

We now understand that the OpenAI team focused on language modeling. Therefore, it makes sense to keep the masked attention sublayer. *Brown et al.*

(2020) dramatically increased the size of the decoder-only transformer models to get excellent results.

GPT models have the same structure as the decoder stacks of the Original Transformer designed by *Vaswani et al.* (2017). We described the decoder stacks in *Chapter 2, Getting Started with the Architecture of the Transformer Model*. If necessary, take a few minutes to go back through the architecture of the Original Transformer.

The GPT model has a decoder-only architecture, as shown in *Figure 7.2*:

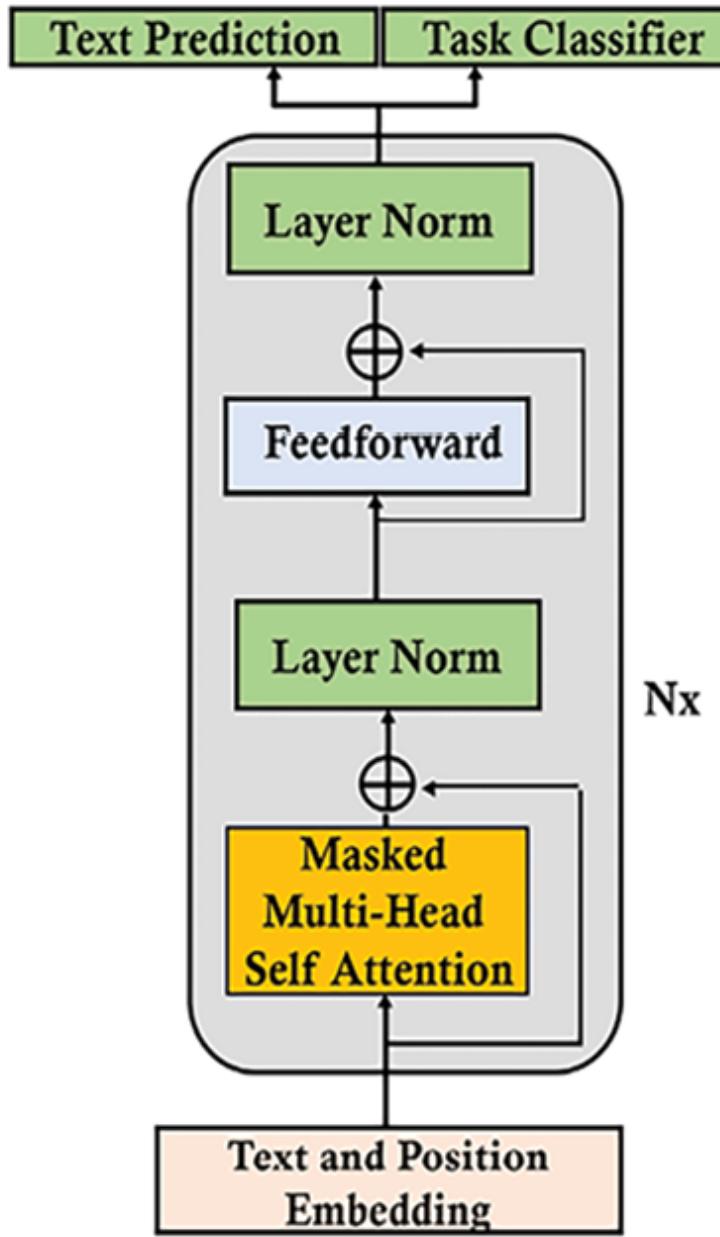


Figure 7.2: GPT decoder-only architecture

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.

The OpenAI team customized and tweaked the decoder model by model. Radford et al. (2019) presented no fewer than four GPT models, and Brown

et al. (2020) described no fewer than eight models.

The GPT-3 175B model has reached a unique size that requires computer resources that few teams in the world can access:

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

OpenAI produces a significant number of models and variations.

## GPT models

OpenAI is continually evolving. New models appear while others become deprecated. Some are upgraded, function calls change, API interfaces evolve, and the documentation goes through many updates. This is normal for a general-purpose technology, as explained in the *GPTs are GPTs* section of this chapter. We have to keep up with this fast-paced technology.

Fortunately, OpenAI has online resources that inform us of the evolution of their models.

OpenAI's model documentation page that details the categories and lists of models: <https://platform.openai.com/docs/models>

The list contains the main domains of OpenAI models, such as language, vision, embedding, and moderation. We will implement these domains in this chapter and in the following chapters.

For example, we will implement the embeddings model in *Chapter 11, Leveraging LLM Embeddings as an Alternative to Fine-Tuning*. We will implement the moderation and whisper models in *Chapter 15, Guarding the Giants: Mitigating Risks in Large Language Models*. We begin implementing vision models (OpenAI and others) in *Chapter 16, Beyond Text: Vision Transformers in the Dawn of Revolutionary AI*.

OpenAI also provides a model deprecation page that details the retirement dates of the “older” models in this fast-moving market:

<https://platform.openai.com/docs/deprecations>.

Let’s now open the hood of OpenAI and look at the models and variants available. This notebook will help you see the list of models and variants while consulting OpenAI’s documentation.

Open `openAI_Models.ipynb`. We will first install OpenAI:

```
try:  
    import openai  
except:  
    !pip install openai  
    import openai
```

Then retrieve the OpenAI `API_KEY` from a file:

```
from google.colab import drive  
drive.mount('/content/drive')  
f = open("drive/MyDrive/files/api_key.txt", "r")  
API_KEY=f.readline()  
f.close()
```

You can load the `API_KEY` from a file or enter it directly if you don’t wish to hide it.

Now, we can set an environment variable with the `API_KEY`:

```
import os  
os.environ['OPENAI_API_KEY'] = API_KEY  
openai.api_key = os.getenv("OPENAI_API_KEY")
```

We now create a list of models and engines:

```
elist=openai.models.list()  
print(elist)
```

We want to know how many models and engines there are:

```
count = 0  
for model in elist:  
    count += 1  
print("Number of models:", count)
```

The output shows the number of models and engines:

```
Number of models: 81
```

It is good practice to verify the list regularly along with OpenAI's documentation (model and deprecation pages) as this list and number continually evolves.

The following cell displays a sorted list of `models` in `pandas` (note that the `elist` object itself can evolve as well):

```
import pandas as pd  
model_data = []  
# Iterate through each model in elist and collect the required  
for model in elist:  
    model_info = {  
        'id': model.id,  
        'created': model.created,  
        'object': model.object,  
        'owned_by': model.owned_by
```

```

        }
    model_data.append(model_info)
# Create a DataFrame from the collected data
df = pd.DataFrame(model_data)
# Sort the DataFrame by the 'id' column
df_sorted = df.sort_values(by='id')
# Display the sorted DataFrame
df_sorted

```

We can now see the list of models in the **id** column, as shown in the excerpt in *Figure 7.3*:

index		id	created	object	owned_by
3	gpt-4		1687882411	model	openai
65	gpt-4-0314		1687882410	model	openai
1	gpt-4-0613		1686588896	model	openai

*Figure 7.3: An excerpt from the list of GPT-based models*

OpenAI's library is quite full of opportunities and paths to explore!

Keep in mind that OpenAI advances at full speed and regularly replaces some models with cutting-edge models.

We will now start with the magic of using ChatGPT models as assistants.

## OpenAI models as assistants

Generative AI and GPT assistants will pervade everyday applications. From a software development perspective, nothing will ever return to the old days. ChatGPT-like models will boost the productivity of everyday software development.

In this section, we will unleash the power of GPTs to use them as assistants to explain OpenAI models and engines.

Go to this link to access ChatGPT Plus: <https://chat.openai.com/>.

If you don't wish to subscribe, you can try OpenAI's free version:  
<https://openai.com/chatgpt>.

ChatGPT Plus offers services such as GPT-3.5, GPT-4, and plugins.

The cutoff date of GPT-4 training might limit its responses since the available data is in the past. Therefore, plugins can come in handy, such as the Bing feature that has been implemented in OpenAI. However, the plugins are continually updated, and some are retired.

ChatGPT Plus models will evolve, and you will grow with them!

You can also go directly to New Bing, which is powered by OpenAI's state-of-the-art models: <https://bing.com/?link=new>.

You can open the New Bing chatbot in Microsoft Edge while you are searching for information.

You can also ask ChatGPT to provide source code.

## ChatGPT provides source code

ChatGPT can generate source code in multiple programming languages:

Create a Fibonacci sequence in Python

ChatGPT will provide the code in Python and explain it:

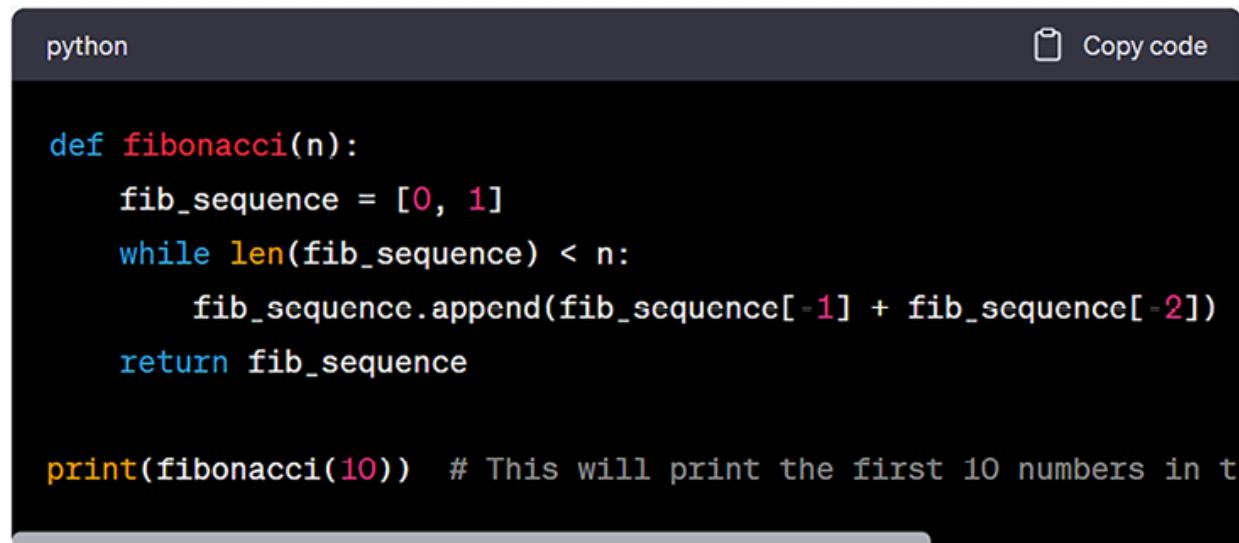
Sure, I can provide you with a simple Python function to generate a Fibonacci sequence. Here's a basic function to generate the first `n` numbers in the sequence:

```
```python
def fibonacci(n):
```

```
fib_sequence = [0, 1]
while len(fib_sequence) < n:
    fib_sequence.append(fib_sequence[-1] + fib_sequence[-2])
return fib_sequence
print(fibonacci(10)) # This will print the first 10 numbers in the sequence
```

This function starts with the list `[0, 1]`, which are the first two numbers in the Fibonacci sequence. It then enters a loop where it repeatedly appends the sum of the last two numbers in the sequence to the list until it reaches the desired length `n`. Finally, it returns the completed `fib\_sequence` list. Note that the Fibonacci sequence is defined such that each number is the sum of the previous two.

Once you have read the answer, you can copy the code and paste it directly into your Python environment:



A screenshot of a code editor window titled "python". The code editor contains the following Python code:

```
def fibonacci(n):
    fib_sequence = [0, 1]
    while len(fib_sequence) < n:
        fib_sequence.append(fib_sequence[-1] + fib_sequence[-2])
    return fib_sequence

print(fibonacci(10)) # This will print the first 10 numbers in the sequence
```

The code defines a function `fibonacci` that takes an integer `n` as input. It initializes a list `fib\_sequence` with the first two Fibonacci numbers, 0 and 1. It then enters a loop where it repeatedly appends the sum of the last two numbers in the sequence to the list until it reaches the desired length `n`. Finally, it returns the completed `fib\_sequence` list. A comment at the end of the function indicates that it will print the first 10 numbers in the sequence.

Figure 7.4: Click on “Copy code” in the top-right corner and paste it into your Python editor

You can ask ChatGPT to provide the code in other programming languages.

You can also run GPT-4 as an assistant in your development environment with GitHub Copilot.

## GitHub Copilot code assistant

You can use the generative functionality of OpenAI's state-of-the-art models in your development environment. In this section, we will run GitHub Copilot in JetBrains PyCharm.

First, go to GitHub Copilot:

<https://github.com/features/copilot>.

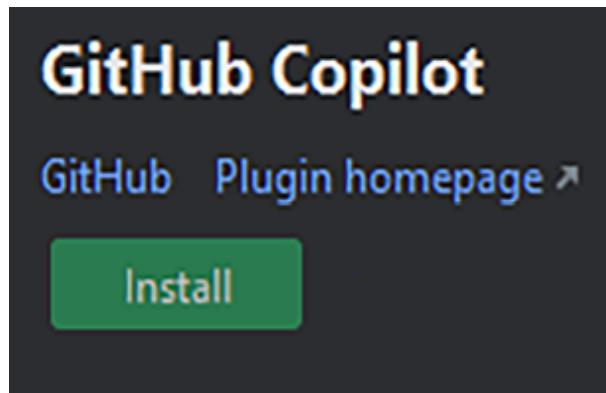
You can read the documentation to get started:

<https://docs.github.com/en/copilot/getting-started-with-github-copilot>.

In this section, JetBrains PyCharm was chosen:

<https://www.jetbrains.com/products/compare/?product=pycharm&product=pycharm-ce>.

JetBrains provides the documentation to install PyCharm and GitHub Copilot. PyCharm contains a function to install GitHub Copilot:



*Figure 7.5: One step away from installing GitHub Copilot on PyCharm*

You can activate GitHub in PyCharm and get to work in a few clicks, as shown in *Figure 7.6*:

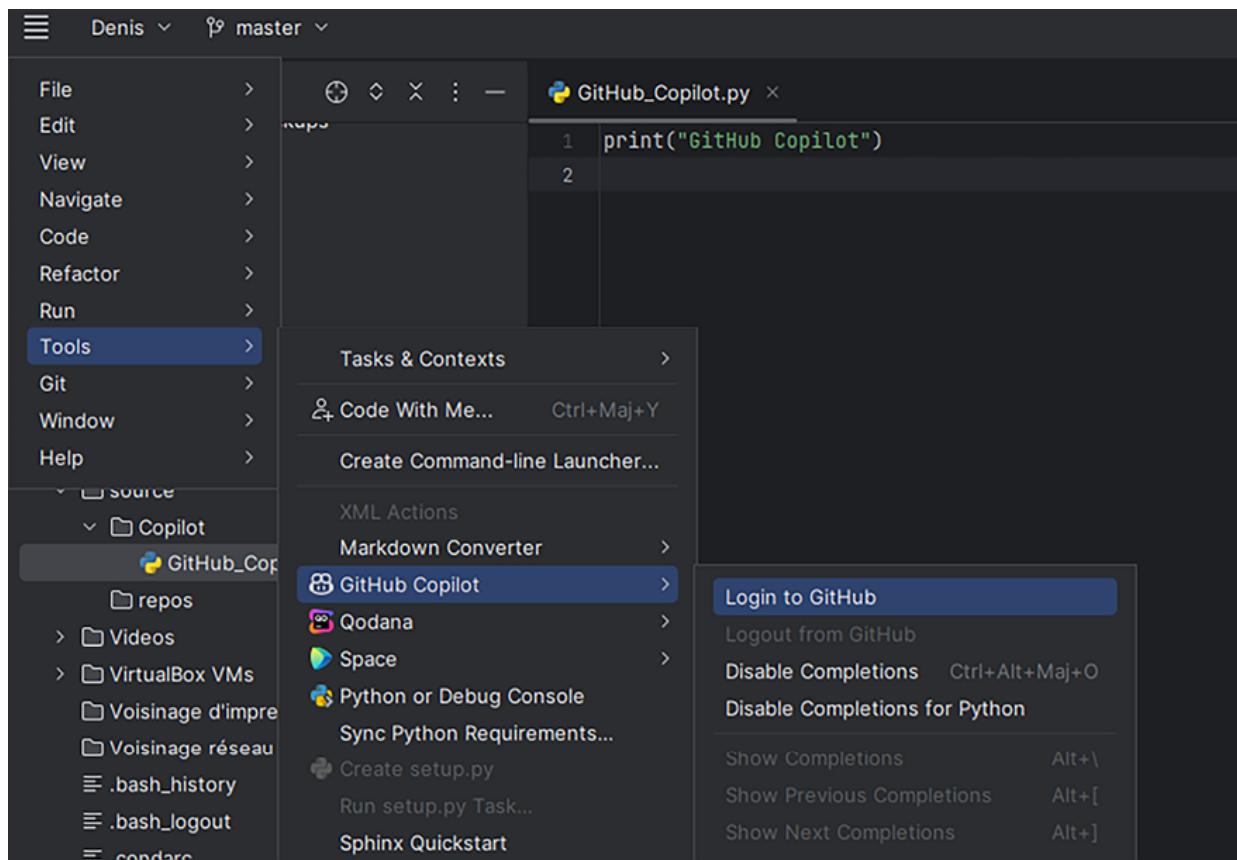


Figure 7.6: GitHub Copilot activation path on PyCharm

PyCharm has an intuitive interface and will guide you through activating code generation:

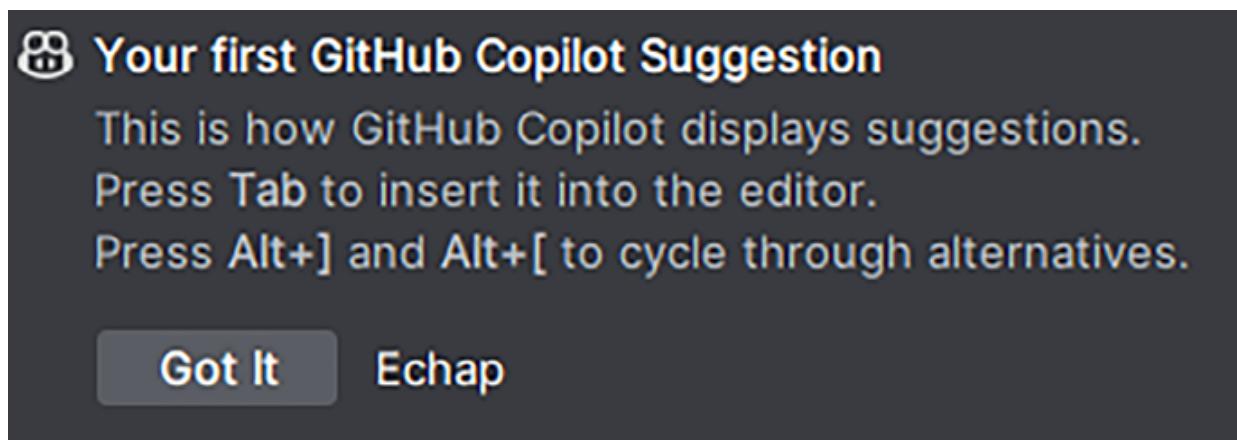


Figure 7.7: The GitHub Copilot plugin walk-through

You can begin by describing a function you would like, such as a Fibonacci sequence, in a comment (the prompt):

```
#create a Fibonacci sequence
```

The code will be generated automatically, and you can select it by pressing *Tab*:

```
def fib(n):
    if n == 0:
        return 0 elif n == 1:
        return 1 else:
        return fib(n-2) + fib(n-1) # recursive call
```

Then you continue by entering other comments, which act as a prompt, and press *Tab* when the code appears:

```
# calculate the 10th Fibonacci number
fib(10)
#print output
print(fib(10))
```

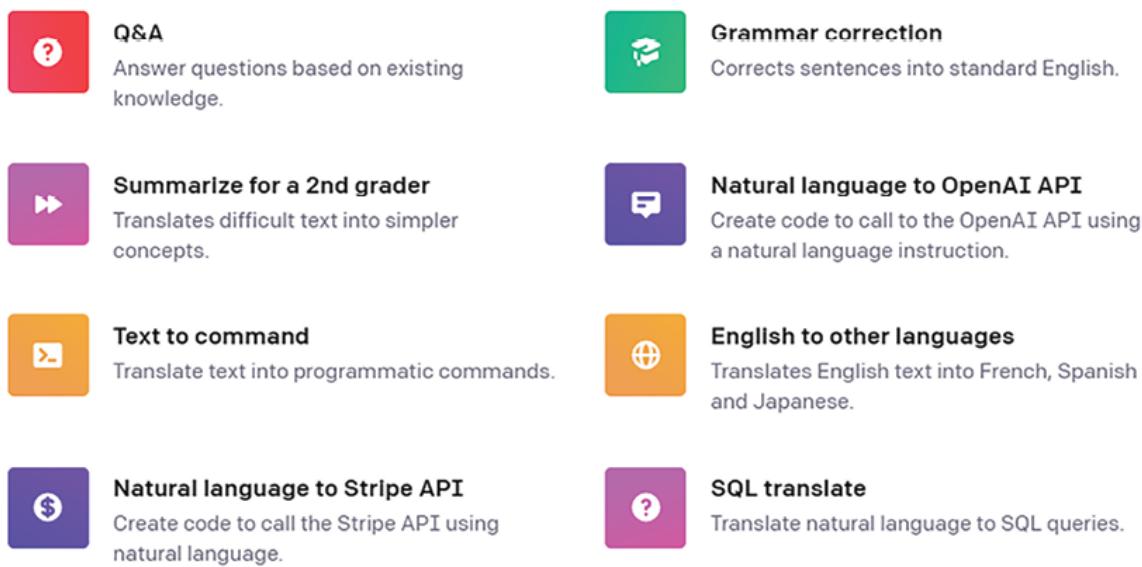
You can then run the code in your environment.

GitHub Copilot, as a coding assistant, will boost your productivity. Just think of the number of assistants we have already gone through! And they only represent the tip of the iceberg of the number of assistants propagating on the market.

General-purpose examples will take us to another NLP task and coding assistant.

# General-purpose prompt examples

Go to <https://platform.openai.com/examples>. The page shows many examples we can look into, as shown in *Figure 7.8*:



*Figure 7.8: A few examples from the prompt examples provided by OpenAI*

Note that OpenAI GPT models were not pretrained to perform these tasks. The system's magic is that GPT was not trained for these tasks.

If we click on **Grammar correction**, we can access an example in Python:

```
import os
import openai
openai.api_key = os.getenv("OPENAI_API_KEY")
response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo",
    messages=[],
    temperature=0,
    max_tokens=256
)
```

We can also go to the OpenAI Playground to run an example such as Q&A, ask questions, and modify the parameters:

<https://platform.openai.com/playground/>.

You can run many NLP tasks, modify the parameters, and view the code.

ChatGPT for Microsoft Office 365 Copilot opens the door of generative transformer technology to millions of end users, including us.

## **Getting started with ChatGPT - GPT-4 as an assistant**

In this section, we will get started using ChatGPT Plus with GPT-4 as an assistant and copilot. You will see how a cutting-edge developer can reduce the time to market with GPT-4 as a copilot.

Most of this section was written in the notebook by the author but also with the support of GPT-4. GPT-4's comments are preceded by `GPT-4:`, which is a standard ethical procedure.

First, open `open_AI_GPT_4_Assistant.ipynb` in the chapter directory of the GitHub repository.

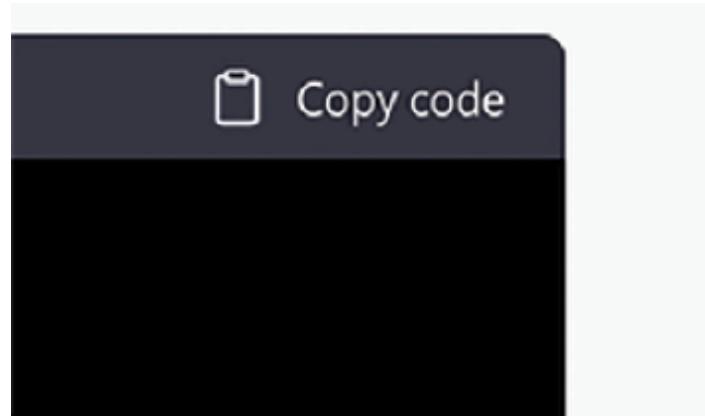
This section follows the structure and comments of the notebook. Working at this speed and comfort is an exhilarating experience.

### **1. GPT-4 helps to explain how to write source code**

Give GPT-4 instructions with well-crafted prompts to steer the model.

It will return the source code. This notebook was designed with Python but you can try other languages.

In the top-right corner of the source code frame, click on **Copy code**:



*Figure 7.9: Copy code option*

Paste the code in a code cell in your notebook and run it! Note that there is a limitation: some debugging might be necessary. In any case, the status of each test is provided at the end of each section.

## 2. GPT-4 creates a function to show the YouTube presentation of GPT-4 by Greg Brockman on March 14, 2023

```
Denis Rothman: I would like to write a program in Python in Google Colab to display a YouTube video in a frame in Google Colab cell. The video is a presentation of GPT-4 by Greg Brockman. How I can write this code?
```

```
GPT-4: To display a YouTube video in a Google Colab notebook...
```

Read the rest of GPT -4's instructions in the notebook. We give GPT-4 the instructions for the task we wish to obtain. GPT-4 responds with the instructions for us to follow:

```
from IPython.display import YouTubeVideo
Greg_Brockman="hdhZwyf24mE"
# Replace the video_id below with the YouTube video ID of the video you want to play
video_id = Greg_Brockman # Replace with the correct video ID
YouTubeVideo(video_id)
```

### 3. GPT-4 creates an application for WikiArt to display images

The developer provides the prompt, and GPT-4 responds with the instructions:

Denis Rothman: I would like to write a Python program for Google Colab that can display images from wikiart. How do I start?

GPT-4: To display images from Wikiart in a Google Colab notebook using Python, you can follow these steps...

Read the rest of GPT-4's instructions in the notebook and run the code:

```
# Import required libraries
import requests
from IPython.display import Image, display
# Function to display an image from Wikiart
def display_wikiart_image(url):
    response = requests.get(url)
    if response.status_code == 200:
        img = Image(data=response.content)
        display(img)
    else:
        print("Unable to fetch image")
# Replace the URL below with the desired Wikiart image URL
wikiart_image_url = "https://uploads7.wikiart.org/images/salvador-dali-the-mystic-sleep-of-revelation-1944-1951.jpg"
display_wikiart_image(wikiart_image_url)
```

Bear in mind that it takes less than a minute to get each function running.

## 4. GPT-4 creates an application to display IMDb reviews

We are now settled in our routine and getting used to the comfort of piloting a racing car!

We provide the prompt and obtain instructions:

```
Denis Rothman: Ok. I want to create another program on Google Colab  
in Python. This time I want to write a program that displays movie  
reviews from IMDB and displays them in the Google Colab notebook.  
How do I do this?
```

```
GPT-4: To display movie reviews from IMDB in a Google Colab notebook  
using Python, you can use the requests library to fetch the HTML ...
```

Read the rest of GPT-4's instructions in the notebook and run the code.

We first install `beautifulsoup4` to scrape web pages:

```
pip install beautifulsoup4 lxml
```

Then we run the code provided by GPT-4:

```
import requests  
from bs4 import BeautifulSoup  
from IPython.display import display, Markdown  
def display_imdb_reviews(movie_id, num_reviews=5):  
    url = f"https://www.imdb.com/title/{movie_id}/reviews"  
    response = requests.get(url)  
    if response.status_code != 200:  
        print("Unable to fetch IMDb reviews")
```

```

        return
    soup = BeautifulSoup(response.text, "lxml")
    reviews = soup.find_all("div", class_="imdb-user-review")
    for idx, review in enumerate(reviews[:num_reviews]):
        title = review.find("a", class_="title").text.strip()
        author = review.find("span", class_="display-name-link")
        date = review.find("span", class_="review-date").text
        content = review.find("div", class_="text").text.strip()
        display(Markdown(f"**Review {idx + 1}: {title}**"))
        display(Markdown(f"_by {author} on {date}_"))
        display(Markdown(f"\n{content}\n\n---"))
    # Replace the movie_id below with the IMdb ID of the desired movie
    movie_id = "tt1375666" # Inception (2010)
    display_imdb_reviews(movie_id)

```

The output contains the requested reviews:

```

Review 1: A one-of-a-kind mind-blowing masterpiece!
by adrien_ngoc_1701 on 1 March 2019
My 3rd time watching this movie! Yet, it still stunned my mind,

```

## 5. GPT-4 creates an application to display a newsfeed

Humanity went from walking on foot to riding horses, from horses to trains, and from trains to cars to airplanes. Soon, humanity will travel on shuttles in space. In the 21<sup>st</sup> century, few people want to go from New York to Los Angeles on horseback. Some will prefer driving or taking a train. Most people will fly.

Once we get used to the speed a GPT-4 copilot provides, there is no turning back!

For this task, a simple one-sentence prompt does the job:

Denis Rothman I want to write a program in Python in Google Colab that reads a news feed. How do I do that?

GPT-4: To write a Python program in Google Colab that reads a news feed, ...

Read the rest of GPT-4's instructions in the notebook and run the code:

```
!pip install feedparser
import feedparser
news_feed_url = "http://feeds.bbci.co.uk/news/rss.xml"
feed = feedparser.parse(news_feed_url)
for entry in feed.entries:
    print(entry.title)
    print(entry.link)
    print()
```

The output provides news headlines and websites to consult:

```
NHS 5% pay offer may end bitter dispute in England
https://www.bbc.co.uk/news/health-64977269?at_medium=RSS&at_cam
.../...
```

## 6. GPT-4 creates a k-means clustering (KMC) algorithm

This example shows an iterative dialogue to obtain the right source code.

First, GPT-4 does not provide the exact answer. We provide the error message, and GPT-4 corrects the code.

You also provide the code you have written directly and share an error you are encountering.

In this case, GPT-4 provided the code, analyzed the error, and regenerated a correct response.

This prompt requests a KMC program:

```
Denis Rothman Create a program in Google Colab in Python that can  
generate 1000 random numbers between 1 and 1000 and then run a k-  
means clustering algorithm on this data and finally display the  
result in matplotlib.
```

```
GPT-4: To create a program in Google Colab that generates 1000  
random numbers between 1 and 1000, runs a k-means clustering  
algorithm on the data, and displays the result using matplotlib,  
follow these steps:
```

Read the rest of GPT-4's instructions in the notebook and run the code.

However, this time, the GPT-4 code will not work. Don't be surprised when you get an error. Read GPT-4's reaction and run the new, corrected code. It runs perfectly.

You can ask GPT-4 to correct or explain its code or your code. 21<sup>st</sup>-century productivity in development has reached another level!

This notebook showed the flexibility and copilot potential of GPT-4.

Now, we will look into the OpenAI GPT-4 API.

## Getting started with the GPT-4 API

OpenAI has some of the most powerful transformer engines in the world. One GPT-4 model can perform hundreds of tasks. GPT-3 can do many tasks it wasn't trained for.

This section will use the API in `Getting_Started_GPT_4_API.ipynb`.

To use GPT-3, go to OpenAI's website, <https://openai.com/>, and sign up.

We can run the examples provided by OpenAI to get started. We are once again relying on assistants.

## Running our first NLP task with GPT-4

Let's start using GPT-4 in a few steps.

Go to Google Colab and open `Getting_Started_GPT_4_API.ipynb`, which is the chapter directory of the book on GitHub.

You do not need to change the hardware settings of the notebook. We are using an API, so we will not need much local computing power for the tasks in this section.

The steps of this section are the same ones as in the notebook.

Running an NLP task is done in three simple steps:

### Steps 1: Installing OpenAI and Step 2: Entering the API key

Steps 1 and 2 are the same as we went through in the *GPT models* section of this chapter.

Let's now run an NLP task.

### Step 3: Running an NLP task with GPT-4

We copy and paste an OpenAI example for a **grammar correction** task:

```
from openai import OpenAI
client = OpenAI()
response = client.chat.completions.create(
    model="gpt-4",
    messages=[
        {
            "role": "system",
            "content": "You will be provided with statements, and yo
        },
        {
            "role": "user",
            "content": "She no went to the market."
        }
    ],
    temperature=0,
    max_tokens=256,
    top_p=1,
    frequency_penalty=0,
    presence_penalty=0
)
```

The task is to correct this grammar mistake: She no went to the market.

We can process the response as we wish by parsing it. OpenAI's response is a dictionary object. The OpenAI object contains detailed information on the task. We can ask the object to be displayed:

```
print(response.choices[0].message.content)
```

The output of “text” in the dictionary is the grammatically correct sentence:

```
She didn't go to the market.
```

You can steer the outputs with several parameters.

# Key hyperparameters

The term “parameters” generally refers to the trained parameters, such as weights and biases. The term “hyperparameters” refers to parameters that can configure the model (layers, dimensions) or its behavior when it processes a request.

A request begins with a prompt and is completed by the model or engine. “Completion” refers to the model’s task of completing a prompt with its Generative AI capabilities. The model predicts a response token by token.

You can try each example in the notebook with the different models or engines listed in the *GPT models* section of this chapter and modify the behavior of the completion task with the following parameters:

- `model="gpt-4"`: The choice of the OpenAI GPT-3 model to use and possibly other models in the future.
- `temperature=0`: A higher value, such as `0.9`, will force the model to take more risks. Do not modify the temperature and `top_p` at the same time.
- `max_tokens=256`: The maximum number of tokens of the response.
- `top_p=1.0`: Top-P or nucleus sampling selects the top probabilities until the sum of the probabilities reaches the `top_p=1.0` value. Then, one of the probabilities in this set is chosen randomly.
- `frequency_penalty=0.0`: A value between `0` and `1` limits the frequency of tokens in a given response.
- `presence_penalty=0.0`: A value between `0` and `1` forces the system to use new tokens and produce new ideas.

Each pipeline on each platform for each model has its hyperparameters that will influence the sampling process and control outputs.

For more on the sampling process involving the temperature Top-P hyperparameters, see the *Vertex AI PaLM 2 interface* section of *Chapter 14, Exploring Cutting-Edge LLMs with Vertex AI and PaLM 2*.

In this case, the prompt has two main parts: the system and user roles.

## The system role

The roles are preceded by `"role":`. The system role is defined by:

```
"role": "system",
```

Then, the content of the role is defined by the term `"content":`, which contains the instructions for the model:

```
"content": "You will be provided with statements, and your ta
```

## The user role

The user role is defined by:

```
"role": "user",
```

Then, as for the role, the content of the user role is defined by the term `"content":`, which contains the user request:

```
"content": "She no went to the market."
```

Many more options are possible. OpenAI documentation is constantly evolving and offering new functionality:

<https://platform.openai.com/docs/overview>.

Your imagination is the limit!

You can now continue the notebook for the other tasks.

## Running multiple NLP tasks

`Getting_Started_GPT_4_API.ipynb` contains examples you can run to practice implementing OpenAI GPT-4. You can refer to the description of the content of the prompt in *Step 3, Running an NLP task with GPT-4* section above.

Run these examples in the notebook. We ran the grammar correction and translation task. You can rerun the tasks with different engines and parameters. Then continue exploring and running the remaining examples with the output processing code provided when required:

- Example 1: Grammar correction
- Example 2: English-to-French translation
- Example 3: Calculate time complexity
- Example 4: Movie to emoji
- Example 5: Spreadsheet creator
- Example 6: Advanced tweet classifier
- Example 7: Natural Language to SQL

You can go further and run many other tasks on the Examples page:

<https://beta.openai.com/examples>.

You can also take GPT-4 to the next level with **Retrieval Augmented Generation (RAG)**.

## Retrieval Augmented Generation (RAG) with GPT-4

In this section, we will build an introductory program that implements **RAG**. Document retrieval is not new. Knowledge bases have been around since the arrival of queries on databases decades ago. Generative AI isn't new either. RNNs were AI-driven text generators years ago.

Taking these factors into account, we can say that RAG is not an innovation but an improvement that compensates for the lack of precision, training data, and responses of Generative AI models. It can also avoid fine-tuning a model in some instances.

There are also different ways of performing augmented generation, as we will see in the following chapters:

- *Chapter 11, Leveraging LLM Embeddings as an Alternative to Fine-Tuning*, is where we will implement embedded data.
- *Chapter 15, Guarding the Giants: Mitigating Risks in Large Language Models*, in which one of the mitigating solutions is to implement knowledge bases.

OpenAI mentions document retrieval as one of the methods to improve results in their documentation:

<https://platform.openai.com/docs/guides/prompt-engineering/six-strategies-for-getting-better-results>.

In this section, we will implement an improvement of GPT's Generative AI GPT-4 functionality with RAG.

Open `GPT_4_RAG.ipynb` in the directory of this chapter.

The notebook is divided into three parts: *Installation*, *Document retrieval*, and *Retrieval augmented generation*.

## Installation

The following packages are installed:

- `tiktoken`, a BPE tokenizer for OpenAI's models
- `cohere`, a text-generation LLM platform
- `openai`, the API functionality we will use for GPT-4
- `ipwidgets` to create HMTL widgets
- `transformers`, packages for transformer functionality
- `requests` to make HTTP requests in Python
- `beautifulsoup4` for web scraping

Once the libraries and modules are imported, we can create a document retrieval function.

## Document retrieval

The document retrieval section of the code contains two functions: URL retrieval and URL processing based on the user request.

The first function is a URL retrieval function based on the user request:

```
def select_urls_based_on_query(user_query):
```

The URLs are selected by using a keyword in the user's request. Any other method can be implemented. The following code selects URLs in a knowledge base based on keywords detected in the user's request:

```
def select_urls_based_on_query(user_query):
    # URLs related to 'climate'
    climate_urls = [
        "https://en.wikipedia.org/wiki/Climate_change", # Rep.
        "https://en.wikipedia.org/wiki/Effects_of_climate_change"
    ]
    # URLs related to 'RAG'
    rag_urls = [
        "https://en.wikipedia.org/wiki/Large_language_model",
        "https://huggingface.co/blog/ray-rag"
    ]
    # Check if 'climate' is in the user query
    if "climate" in user_query.lower():
        return climate_urls
    # Check if 'RAG' is in the user query
    elif "RAG" in user_query:
        return rag_urls
    # Default return if no keyword matches
    return []
```

You can use any other method you wish, such as database queries and lists. You can also implement any form of automation you want. *The main concept to keep in mind is that the search should be based on the user's request and adapted to it.*

The second function scrapes and summarizes the documents.

First, a `summarizer` is defined:

```
def fetch_and_summarize(user_query):
    urls = select_urls_based_on_query(user_query)
    summarizer = pipeline("summarization", model="sshleifer/di
```

The content retrieved is processed:

```
summaries = []
for url in urls:
    page = requests.get(url)
    soup = BeautifulSoup(page.content, 'html.parser')
    # Try to extract the main article text more accurately
    # This is a generic example and might need to be adjusted for .
    article = soup.find('article')
    if article:
        article_text = article.get_text()
    else:
        paragraphs = soup.find_all('p')
        article_text = '\n'.join([para.get_text() for para
            # Truncate if too long for the model
if len(article_text) > 1024:
    article_text = article_text[:1024]
```

The `summarizer` is then activated to summarize the content retrieved:

```
summary = summarizer(article_text, max_length=130, min_leng
summaries.append(summary)
```

Finally, the summary is returned to the GPT-4 function:

```
return summaries
```

We have seen one of the many ways to retrieve data based on a user's request. Now, let's see how to integrate the information in a GPT-4 input.

# Augmented retrieval generation

We first import the modules and classes we need:

```
from openai import OpenAI
import ipywidgets as widgets
from IPython.display import display
```

Then, we define the client and a model:

```
client = OpenAI()
AImodel = "gpt-4" # or select another model
```

We create the API function:

```
# Function to interact with OpenAI's model
def openai_chat(input_text, document_excerpt, web_article_summary):
    # Start the OpenAI API call to generate a chat response
    response = client.chat.completions.create(
        model=AImodel,
```

The prompt message contains three roles:

- A document excerpt, which can come from any source (database, web, text file, JSON, or hard-coded):

```
"role": "system", # "system" role for providing context
"content": f"The following is an excerpt from a document about {document_excerpt}. # The document excerpt is now a variable passed to the function"
```

- A web excerpt, which comes from the summarizing function we defined:

```
"role": "system", # Another "system" role message  
"content": f"The following is a summary of a web article c  
# The web article summary is now a variable passed to the
```

- The user's input text:

```
"role": "user", # "user" role for the actual user query  
"content": input_text  
# The user's query or input that the model will resp
```

We now define the parameters as we explored in the *Getting started with the GPT-4 API* section of this chapter:

```
temperature=0.1, # Controls randomness. Lower values i  
max_tokens=150, # Sets the maximum length of the res  
top_p=0.9, # Nucleus sampling: A higher value in  
frequency_penalty=0.5, # Reduces repetition of the sam  
presence_penalty=0.5 # Reduces repetition of similar te
```

The response is not a dictionary but an instance of a class:

```
# the response object is not a dictionary. It is an instance o  
# to access the content property, use dot notation instead of [0]  
return response.choices[0].message.content
```

Note that the code excerpts do not contain brackets. Consult the notebook for the complete code.

We can now define the user request:

```
input_text = "What are the impacts of climate change?"  
#input_text = "What is RAG"
```

In this case, the input text is in the code. When you deploy an application, the user request will be submitted from the interface you have designed.

The example here is climate change, or it could be RAG. To go further, you will have to enhance the document retrieval functionality. Keep in mind that this requires resources for design, development, copyright management, and all the standard procedures that go with document retrieval and usage.

We now parse the user request to decide what type of document excerpt we need to use. This introductory educational example uses keywords. More sophisticated parsers can be implemented. Also, the document is hard-coded in this example. You could retrieve and summarize a PDF, insert a complete text, or any other content you wish to implement. In this case, we are focusing on climate change but leave it up to the code to choose a document excerpt:

```
# 1. you can create a function specifically for your domain wi  
# Check if 'climate' is in the user query  
if "climate" in input_text.lower():  
    document_excerpt = "Climate change refers to significa  
# Check if 'RAG' is in the user query  
if "RAG" in input_text.lower():  
    document_excerpt = "OpenAI documentation states that R
```

You can hard-code the information if it is for customer support on a product that will be stable for its lifecycle, or you can automate the process by activating the scraping and summarizing functionality we implemented:

```
#2. and/or you can automate the retrieval  
summaries = fetch_and_summarize(input_text)  
#print(summaries)  
web_article_summary = summaries
```

We are now ready to send the document excerpt and summary to OpenAI along with the user request. GPT-4 will have a significantly enhanced context to respond:

```
iresponse = openai_chat(input_text, document_excerpt, web_article_summary)  
formatted_response = iresponse.replace('\n', '<br>') # Replace newlines with HTML line breaks  
display(widgets.HTML(value=formatted_response)) # Display response
```

The response and output are interesting:

```
Climate change impacts the physical environment, ecosystems, and human health.
```

We can see that RAG can be quite a way to improve the quality of Generative AI models and also control the outputs through guided enhanced input requests.

## Summary

We began the chapter by seeing how OpenAI **Generative Pre-trained Transformer (GPT)** models are **General Purpose Technologies (GPTs)**. As such, they improve rapidly, and their diffusion is highly pervasive. The Generative AI functionality of OpenAI's models has broadened the horizons for mainstream applications.

We continued by examining the improvements in the architecture of GPTs through decoder stacks, scaling, and machine power. These improvements led to the creation of ChatGPT, which spread into mainstream everyday lives, pervading applications such as search engines, office tools, and more.

We started with some of the many generative transformer assistants, including ChatGPT, New Bing, GitHub Copilot, Microsoft 365 Office Copilot, and OpenAI's Playground.

Our journey then led to building several examples with the OpenAI GPT-4 API, such as grammar corrections, translations, and more.

Finally, we built an example of how RAG can improve the performances of Generative AI models, such as GPT-4.

In the next chapter, *Chapter 8, Fine-Tuning OpenAI GPT Models*, we will fine-tune an OpenAI GPT model.

## 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. OpenAI GPT models are not GPTs. (True/False)

5. The diffusion of generative transformer models is very slow in everyday applications. (True/False)
6. GPT-3 models have been useless since GPT-4 was made public. (True/False)
7. ChatGPT models are not completion models. (True/False)
8. Gradio is a transformer model. (True/False)
9. Supercomputers with 285,000 CPUs do not exist. (True/False)
10. Supercomputers with thousands of GPUs are game changers in AI. (True/False)

## References

- OpenAI and GPT-3 engines:  
<https://beta.openai.com/docs/engines/engines>
- BertViz GitHub Repository by *Jesse Vig*:  
<https://github.com/jessevig/bertviz>
- OpenAI's supercomputer:  
<https://blogs.microsoft.com/ai/openai-azure-supercomputer/>
- *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](https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf)
- *Alec Radford et al.*, 2019, *Language Models are Unsupervised Multi-task Learners*: <https://cdn.openai.com/better-language->

[models/language models are unsupervised multitask learners.pdf](#)

- Common Crawl data: <https://commoncrawl.org/overview/>
- GPT-4 Technical Report, OpenAI 2023:  
<https://arxiv.org/pdf/2303.08774.pdf>

## Further reading

- *Alex Wang et.al, 2019, GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding:*  
<https://arxiv.org/pdf/1804.07461.pdf>
- *Alex Wang et al., 2019<sup>2</sup>, SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems:*  
<https://w4ngatang.github.io/static/papers/superglue.pdf>
- *Tom B. Brown et al., 2020, Language Models are Few-Shot Learners:*  
<https://arxiv.org/abs/2005.14165>
- *Chi Wang et al., 2023, Cost-Effective Hyperparameter Optimization for Large Language Model Generation Inference:*  
<https://arxiv.org/abs/2303.04673>
- *Vaswani et al., 2017, Attention Is All You Need:*  
<https://arxiv.org/abs/1706.03762>

## Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://www.packt.link/Transformers>



# 8

## Fine-Tuning OpenAI GPT Models

*Chapter 7, The Generative AI Revolution with ChatGPT*, took us into the world of GPT assistants and APIs. We examined the Generative AI transformer, general-purpose technology through interactions and code. If all the technology we explored is so effective, why bother? Why not just use the ready-made solutions and forget about learning, preparing data, configuring parameters, and writing documentation?

The answer is quite simple. What can we do when a general-purpose model such as GPT-3 or GPT-4 doesn't reach the accuracy threshold we need for our project? We have to do something. But what? In this chapter, we will explore fine-tuning to make sense of the choices we can make for a project to go in this direction or not. We will introduce risk management perspectives.

In this chapter, we will fine-tune an OpenAI GPT model to explore this option. We will fine-tune a cost-effective Babbage-002 model for a completion task. We will prepare the datasets in the format required by OpenAI. This step seems deceptively easy. However, preparing datasets can prove quite challenging.

We will then launch OpenAI's fine-tuning service. The fine-tuning service can take a few minutes or longer if we are in a queue. We can view our fine-tuned models, which are stored on OpenAI.

Finally, we will run a completion task with Immanuel Kant's work to pave the way for your custom fine-tuning jobs if you decide to choose this option.

By the end of the chapter, you will have built a complete fine-tuning process.

This chapter covers the following topics:

- The improvements and limitations of fine-tuning models on OpenAI
- Dataset preparation
- OpenAI's file format
- Converting JSON files to JSONL
- Fine-tuning an OpenAI GPT model
- Running the fine-tuned model
- Running a completion task (generative)
- Running GPT-4 to compare outputs



With all the innovations and library updates in this cutting-edge field, packages and models change regularly. Please go to the GitHub repository for the latest installation and code examples:

<https://github.com/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition/tree/main/Chapter08>.

You can also post a message in our Discord community (<https://www.packt.link/Transformers>) if you have any trouble running the code in this or any chapter.

The journey begins with examining OpenAI from a risk management perspective.

# Risk management

ChatGPT reached mainstream users as a powerful and ready-to-use assistant. However, the diffusion of fine-tuning functionality has its limitations:

- Fine-tuning requires dataset preparation and knowing how to use developer-level tools and APIs.
- Possible limitations for security, privacy, and compliance requirements.

A project manager will also have to take other considerations into account, as shown in *Figure 8.1*:

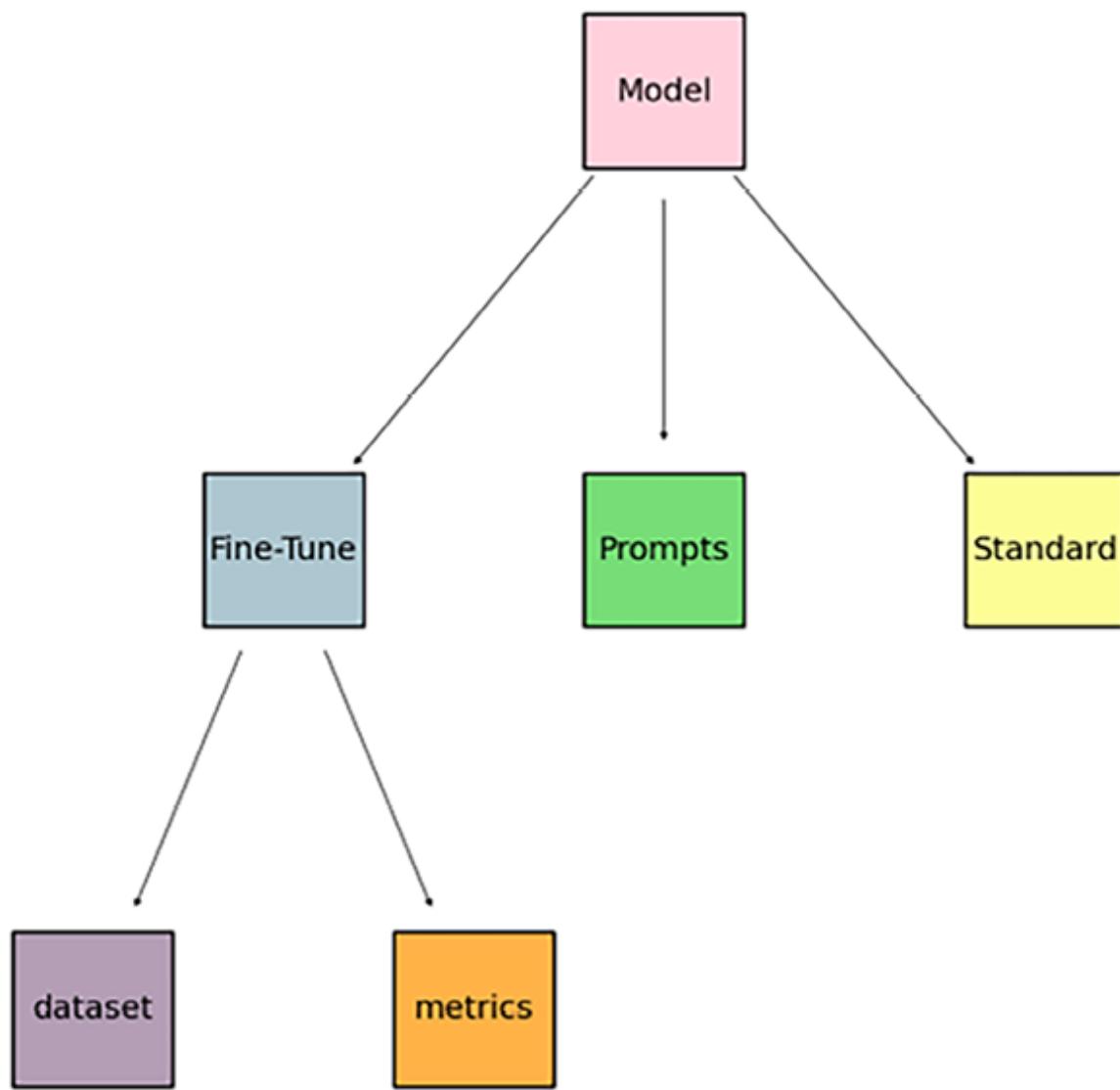


Figure 8.1: Deciding whether to customize a model or not

Figure 8.1 shows that:

- Some cases can benefit from *fine-tuning* with a dataset and metrics. We will implement fine-tuning in this chapter.
- Some projects might require *prompt engineering* instead of or beyond fine-tuning. **Retrieval Augmented Generation (RAG)** might provide better information, which we explored in *Chapter 7, The Generative AI Revolution with ChatGPT*, and we will implement it in other chapters,

including *Chapter 15, Guarding the Giants: Mitigating Risks in Large Language Models*.

- In some cases, the *standard model* might not require fine-tuning or prompt engineering at all. The standard mainstream model might cover our needs.

*The takeaway is that choosing between using a standard pretrained model, fine-tuning a model, and advanced prompt engineering will depend on each project.* Fine-tuning might provide more precise results, process more training data, and save tokens through shorter prompts. Prompt engineering might provide more customized implementations and control.

There is no miracle method or tool to customize transformer models. Each project will require a careful study to balance the resources, costs, and results.

We can now examine how fine-tuning a GPT model works for a completion (generative) task.

## Fine-tuning a GPT model for completion (generative)

OpenAI has a service to fine-tune a list of models, including models from the GPT-3 series, GPT-4, Babbage-002, and Davinci. Some of these models are recommended, while others are experimental. But it is ultimately your decision to choose one of these models or not.

A fine-tuned model can perform data exploration, classification, question answering, and other NLP tasks like the original models. As such, the fine-tuned model might produce acceptable or inaccurate results. Quality control remains essential. Make sure to go through OpenAI's documentation before

beginning a project:

<https://platform.openai.com/docs/guides/fine-tuning/>.

This section aims to implement the fine-tuning process of a model in a notebook, cell by cell, so you can apply fine-tuning to your specific domain.

Fine-tuning GPT models involves four phases, which we will implement in this section:

1. Preparing the data.
2. Fine-tuning a GPT-3 architecture with the Babbage-002 model for a generative task (completion).
3. Running the fine-tuned model.
4. Managing the models.

Once you have learned how to fine-tune an OpenAI model, you can use other types of data to teach it specific domains, knowledge graphs, and texts. You can also evaluate other models.

As you can see, the training process is like the ones we already applied in *Chapter 5, Diving into Fine-Tuning through BERT*, and *Chapter 6, Pretraining a Transformer from Scratch through RoBERTa*. In real-life projects, only cost-effective results count. Explore several options before investing your energy and resources in a model and process.



Evaluate the cost to fine-tune an OpenAI model before running the program in this section.

To get started, open `Fine_Tuning_OpenAI_Models.ipynb` in Google Colab in the GitHub chapter directory and first install the OpenAI library and enter

your key:

```
try:  
    import openai  
except:  
    !pip install openai  
    import openai
```

You can retrieve your API key from a file or comment the code below and enter your key manually.

To read the key from the file, you can use Google Drive or any filesystem of your choice:

```
#You can retrieve your API key from a file(1)  
# or enter it manually(2)  
#Comment this cell if you want to enter your key manually.  
#(1)Retrieve the API Key from a file  
#Store your key in a file and read it(you can type it directly).  
from google.colab import drive  
drive.mount('/content/drive')  
f = open("drive/MyDrive/files/api_key.txt", "r")  
API_KEY=f.readline()  
f.close()
```

You can let the program read your key from a file or comment the code above and enter your key manually:

```
#{2} Enter your key manually by  
# replacing API_KEY by your key.  
#The OpenAI Key  
import os  
os.environ['OPENAI_API_KEY'] =API_KEY  
openai.api_key = os.getenv("OPENAI_API_KEY")
```

You are now ready to prepare the dataset and fine-tune an OpenAI model.

# 1. Preparing the dataset

OpenAI has documented the data preparation process in detail:

<https://platform.openai.com/docs/guides/fine-tuning/preparing-your-dataset>.

For this fine-tuning session, we will download and process the *Critique of Pure Reason* by Immanuel Kant for the Project Gutenberg website. The content of this book is challenging for machines as well as humans and, thus, exciting to use as a dataset. The dataset is also free of copyright issues. Make sure to verify copyright or privacy issues when uploading data to OpenAI.

The first step is to prepare a dataset.

## 1.1. Preparing the data in JSON

OpenAI requires a formatted file with prompts and completions. JSON is one of the recommended file formats. We will first import the necessary libraries.

We will need the following:

- The Natural Language Toolkit with `punkt`, a pretrained tokenizer that we cover in more detail in *Chapter 10, Investigating the Role of Tokenizers in Shaping Transformer Models*. `sent_tokenize` will run in conjunction with `punkt`
- `requests` for the HTTP request
- `BeautifulSoup` to scrape the web page of the book

- `json` to create a json file
- `re` to parse the text with regular expressions

With these tools, we can go from Gutenberg to JSON with the book of our choice:

```
#From Gutenberg to JSON
import nltk
nltk.download('punkt')
from nltk.tokenize import sent_tokenize
import requests
from bs4 import BeautifulSoup
import json
import re
```

We first get the book directly from Project Gutenberg by uncommenting the following code:

```
# First, fetch the text of the book from Project Gutenberg
#url = 'http://www.gutenberg.org/cache/epub/4280/pg4280.html'
#response = requests.get(url)
soup = BeautifulSoup(response.content, 'html.parser')
```

Or we can download the book from the GitHub repository, read it, and parse it with BeautifulSoup:

```
# Option 2: from the GitHub repository:
#Development access to delete when going into production
!curl -L https://raw.githubusercontent.com/Denis2054/Transformers/main/gutenberg/pg4280/pg4280.html
# Open and read the downloaded HTML file
with open("gutenberg.org_cache_epub_4280_pg4280.html", 'r', encoding='utf-8') as f:
    file_contents = f.read()
soup = BeautifulSoup(file_contents, 'html.parser')
```

---

Then we clean it up and split the sentences:

```
# Get the text of the book and clean it up a bit
text = soup.get_text()
text = re.sub('\s+', ' ', text).strip()
# Split the text into sentences
sentences = sent_tokenize(text)
```

Now comes a tricky part. OpenAI expects a prompt and a completion for each line we want to train it for. However, the separators need to be carefully chosen:

```
# Define the separator and ending
prompt_separator = " ->"
completion_ending = "\n"
```

With these well-defined separators, we can create the prompts and completions:

```
# Now create the prompts and completions
data = []
for i in range(len(sentences) - 1):
    data.append({
        "prompt": sentences[i] + prompt_separator,
        "completion": " " + sentences[i + 1] + completion_ending
    })
```

Finally, we save the prompt and completions in a JSON file:

```
# Write the prompts and completions to a file
with open('kant_prompts_and_completions.json', 'w') as f:
```

```
for line in data:  
    f.write(json.dumps(line) + '\n')
```

This process must be carefully prepared to avoid the dataset being partially or totally rejected by OpenAI's preparation module.

We now check our dataset before submitting it to the data preparation module:

```
import pandas as pd  
# Load the data  
df = pd.read_json('kant_prompts_and_completions.json', lines=True)  
df
```

The output shows that we did a good job. We have a prompt and completion per line:

		prompt	completion
0	The Project Gutenberg Etext of The Critique of...	Be sure to check the copyright laws for your ...	
1	Be sure to check the copyright laws for your c...	We encourage you to keep this file, exactly a...	
2	We encourage you to keep this file, exactly as...	Please do not remove this.\n	
3	Please do not remove this. ->	This header should be the first thing seen wh...	
4	This header should be the first thing seen whe...	Do not change or edit it without written perm...	
...	...	...	...
6122	78-79. is their motto, under which they may le...	As regards those who wish to pursue a scienti...	
6123	As regards those who wish to pursue a scientif...	When I mention, in relation to the former, th...	
6124	When I mention, in relation to the former, the...	The critical path alone is still open.\n	
6125	The critical path alone is still open. ->	If my reader has been kind and patient enough...	
6126	If my reader has been kind and patient enough ...	End of Project Gutenberg's The Critique of Pu...	
6127	rows x 2 columns		

Figure 8.2: The prompt and completion data

We're ready to run OpenAI's data preparation module.

## 1.2. Converting the data to JSONL

If the preparation of the JSON file went well, OpenAI's data preparation tool should run smoothly with the file we just created:

```
kant_prompts_and_completions.json
```

We now run the `fine_tunes` tool:

```
!openai tools fine_tunes.prepare_data -f "kant_prompts_and_completions.json"
```

The output shows that there are no errors in the dataset:

```
Analyzing...
- Your JSON file appears to be in a JSONL format. Your file will be converted to JSONL.
- Your file contains 6127 prompt-completion pairs
- All prompts end with suffix ` -> `
- All completions end with suffix ` \n `
Based on the analysis we will perform the following actions:
- [Necessary] Your format `JSON` will be converted to `JSONL`
```

If your dataset contains errors, the preparation tool will try to fix the data.

You might lose data in the process.

In this case, everything is fine. You will be prompted to convert the JSON file into a JSONL file:

```
Your data will be written to a new JSONL file. Proceed [Y/n]: Y
Wrote modified file to `kant_prompts_and_completions_prepared.json`
Feel free to take a look!
```

A few lines later, OpenAI provides information on how to run a prompt:

```
After you've fine-tuned a model, remember that your prompt has  
Once your model starts training, it'll approximately take 1.44
```

OpenAI saves the converted file to `kantgpt_prepared.jsonl`.

We will now take a look at the JSONL file as recommended:

```
import json  
# Open the file and read the lines  
with open('kant_prompts_and_completions_prepared.jsonl', 'r'):  
    f = open('kant_prompts_and_completions_prepared.jsonl')  
    lines = f.readlines()  
    # Parse and print a few lines  
    for line in lines[:5]:  
        data = json.loads(line)  
        print(json.dumps(data, indent=4))
```

The following excerpt of the output shows that everything went well:

```
{  
    "prompt": "The Project Gutenberg Etext of The Critique of Pure Reason by Immanuel Kant  
    "completion": " Be sure to check the copyright laws for you  
}
```

Our last preparation step consists of creating the file on OpenAI:

```
from openai import OpenAI  
client = OpenAI()  
file_response=client.files.create(
```

```
    file=open("/content/kant_prompts_and_completions_prepared.json",
              purpose='fine-tune'
)
# Print option for maintenance
#print(file_response)
```

We will now extract our personal file ID to use the file ID as a variable:

```
# Extract the training file ID
file_id = file_response.id
print(file_id)
```

The output shows that we extracted the file ID correctly:

```
file-vMtR07to3c26hVc4TRrzBmst
```

We are ready to fine-tune an OpenAI model.

## 2. Fine-tuning an original model

You can fine-tune several models on OpenAI, including GPT-3.5-turbo and GPT-4. You can also fine-tune a model with GPT-3 architecture, such as Babbage-002:

<https://platform.openai.com/docs/guides/fine-tuning/what-models-can-be-fine-tuned>.

Fine-tuning Babbage-002 costs less than fine-tuning GPT-4, for example. Start with a cost-effective model and use it as your base model. Then, if you need better results, try the more expensive ones if necessary.

Check the data preparation constraints and dataset formats of a model before fine-tuning it.

We will fine-tune a Babbage-002 model in this section.

The fine-tuning process is ready to run with the file that OpenAI generated using the file ID we extracted:

```
from openai import OpenAI
client = OpenAI()
job_response=client.fine_tuning.jobs.create(
    training_file=file_id,
    model="babbage-002"
)
```

OpenAI has many requests. The fine-tuning request we make is in a queue. You will generally receive an email when the job is completed.

You can extract and display the job ID:

```
job_id = job_response.id
print(job_id)
```

The output will display the job:

```
ftjob-MUR57DYN3TxKeWvX03M8IhCu
```

OpenAI protects your data and models, so only you can access this job.

You can check the status of your fine-tuning job by first retrieving the details of your job:

```
# Check the status of the fine-tuning job
job_details = client.fine_tuning.jobs.retrieve(job_id)
print(job_details)
```

Then, you can display the status of your job:

```
# There may be a time lapse:
# 1.between the moment you run the fine-tuning job and its com,
# 2.between its completion and the server updates
# Check your email if you have activated OpenAI notifications
status = job_details.status
print(f"Job status: {status}")
```

```
Job status: validating_files
```

In this case, the `status` property shows that your dataset is being validated.

The job status will then become `running` and then `succeeded`. The terms displayed may evolve but the process remains the same: file validation, fine-tuning running, and fine-tuning succeeded (or failed).

There may be time lapses between the various displays, including the status.

You can also check the list of fine-tuning jobs and display the status of the job by first retrieving the job list:

```
# List 10 fine-tuning jobs
job_list=client.fine_tuning.jobs.list(limit=10)
```

Then, you can display the jobs and their status:

```
import json
from pprint import pprint
# Get the raw JSON string from the SyncCursorPage object
json_string = job_list.json()
# Convert the JSON string into a Python object
data = json.loads(json_string)
# Extract the data array
jobs = data.get('data', [])
# Format the data into a list of dictionaries
formatted_data = [
    {
        'id': job.get('id'),
        'created_at': job.get('created_at'),
        'status': job.get('status'),
        'training_file': job.get('training_file'),
        'model': job.get('model')
        'model_name':job.get('fine_tuned_model')
    }
    for job in jobs
]
# Print the formatted data
pprint(formatted_data)
```

The excerpt of the output for the current job shows that it is running:

```
[{'created_at': 1705083620,
 'id': 'ftjob-MUR57DYN3TxKeWvX03M8IhCu',
 'model': 'babbage-002',
 'model_name': None,
 'status': 'running',
 ...]
```

`model_name` will be displayed when the model has been fine-tuned and, when you rerun the cell, the excerpt of the output for the current job shows that the fine-tuning job has succeeded, and the model name is displayed:

```
[{'created_at': 1705083620,  
 'id': 'ftjob-MUR57DYN3TxKeWvX03M8IhCu',  
 'model': 'babbage-002',  
 'model_name': 'ft:babbage-002:personal::xxxxxx',  
 'status': 'succeeded',
```

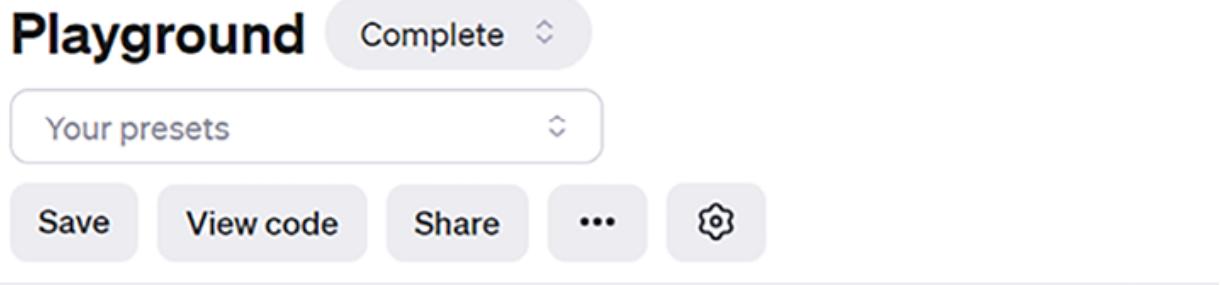
Your model has been fine-tuned and you are now ready to run the model.

### 3. Running the fine-tuned GPT model

The model is trained and ready for a completion task. If you have activated notifications, you will also receive an email when the fine-tuning job is completed. You will receive a message such as the following one:

```
Your fine-tuning job ftjob-xxxxxxxxxxxx has successfully completed.  
Try it out on the OpenAI Playground, view the training results.
```

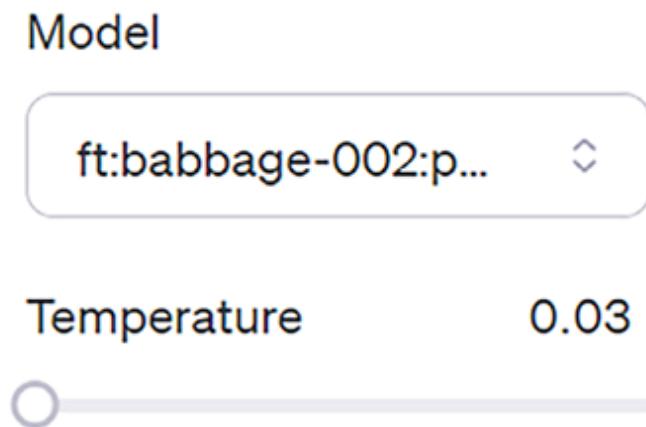
You can activate completions in the Playground, select your fine-tuned model, and test it:



Write an explanation of time and space: 1. What is the meaning of the word time?  
2. What is the meaning of the word space?  
3. What is the difference between the two?

*Figure 8.3: Testing a fine-tuned model in the Playground*

The prompt was “Write an explanation of time and space:” with one of the personal fine-tuned models at a low temperature (more precise):



*Figure 8.4: Fine-tuned model selected with a low temperature*

The model answered with speculative questions, which was interesting.

You can also save your dialog or view the source code, which you can then copy and paste into your program:

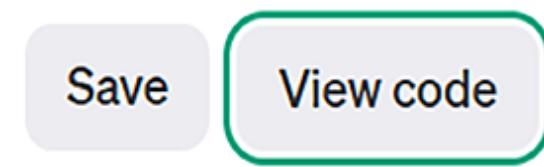


Figure 8.5: Saving the dialog or viewing the code

We will now apply the fine-tuned model as for any original model. You can run the model directly or do some organizing before. We will run the model for a completion task in this section.

We first define a text content that we want the fine-tuned model to complete:

```
# text to complete
text_content = "Space and time are key factors in human reason.
#print(text_content)
```

You can print each step if necessary for maintenance. We then provide instructions to the prompt, adding our text content:

```
prompt = "Continue the following text as if you were a scienti
#print(prompt)
```

Now, we run the task as for any completion model with your trained model:

```
response = client.completions.create(
    model="[YOUR TRAINED MODEL]",
    prompt=prompt,
    max_tokens=1000,
    temperature=0.8
)
#print(response)
```

We can extract the completion and display it:

```
# Check if there are any choices in the response
if response.choices:
    # Get the first choice (index 0)
    first_choice = response.choices[0]
    # Print the text of the first choice
    print("Model's Completion:", first_choice.text)
else:
    print("No choices returned in the response")
```

The following excerpt shows that the fine-tuning worked. However, the output shows that a larger dataset and more training may be required:

```
Model's Completion: If we cannot think about ourselves, we can
What is space? Space is the conscious realm in which I think, o
What is time? Time is the conscious experience of a series of e
Therefore, space and time are the primal conditions of all thou
```

You can also display information on the response:

```
# Formatting the choices
for i, choice in enumerate(response.choices):
    print(f"Choice {i}:")
    print("  Finish Reason:", choice.finish_reason)
    print("  Index:", choice.index)
    print("  Logprobs:", choice.logprobs)
    print("  Text:", choice.text)
# Formatting the usage
print("Usage:")
print("  Completion Tokens:", response.usage.completion_tokens)
print("  Prompt Tokens:", response.usage.prompt_tokens)
print("  Total Tokens:", response.usage.total_tokens)
```

The output is a formatted description of some of the key properties of the response:

```
Completion ID: cmpl-8gHgWG10QuWIqi8sw080pEp9Lwdb
Created: 1705088448
Model: ft:babbage-002:personal::xxxxxx
Object Type: text_completion
Choice 0:
    Finish Reason: length
    Index: 0
    Logprobs: None
    Text: If we cannot think about ourselves, we cannot think ab
```

We will now add some management functions.

## 4. Managing fine-tuned jobs and models

At the time of the writing of this book, you cannot share fine-tuned models outside of your organization. You can deploy them for your projects.

You can also access information on the models, cancel jobs, and delete fine-tuned models. To do this, activate the maintenance functions carefully:

```
from openai import OpenAI
client = OpenAI()
# Set maintenance to True carefully if you wish to activate on
# job or model functions (information, cancel, delete)
maintenance=False
if maintenance is True:
```

If `maintenance` is `True`, you can list the fine-tuning jobs, retrieve the state of a job, cancel a job, list the job events related to the fine-tuning process, and perform model deletion:

```
if maintenance is True:  
    # List 10 fine-tuning jobs  
    client.fine_tuning.jobs.list(limit=10)  
    # Retrieve the state of a fine-tune  
    client.fine_tuning.jobs.retrieve("ftjob-your job")  
    # Cancel a job  
    client.fine_tuning.jobs.cancel("ftjob-your job")  
    # List up to 10 events from a fine-tuning job  
    client.fine_tuning.jobs.list_events(fine_tuning_job_id="ftjo  
    # Delete a fine-tuned model (must be an owner of the org the  
# client.models.delete("your model")
```

There are many other OpenAI services that you can explore:

<https://platform.openai.com/docs/guides/fine-tuning/>

To go further, you can also isolate each phase in separate notebooks:

- *The data in JSON can be prepared* in a separate notebook or even a separate project to create a knowledge dataset. You can create prompts and completions for many other NLP tasks, such as classification, data exploration, and whatever task you need for your project. You can save and load the file when needed for OpenAI's fine-tuning tools.
- *Converting the data to JSONL* can be an excellent way to visualize your dataset for OpenAI's fine-tuning or even as a knowledge dataset for any project. You can save the file and fine-tune an OpenAI model when you are ready or when it's required.

- *Fine-tuning* an OpenAI model can be a separate program. You can load the prepared dataset when you wish.
- *Running the fine-tuned GPT model* can be done in a separate notebook or application.

You can also run the management functions in this section in another notebook or application. The flexibility of the fine-tuning process will alleviate the tension of each phase, which requires careful preparation and resources.

## Before leaving

What if a standard ChatGPT request could satisfy our needs?

Let's ask GPT-4:

```
from openai import OpenAI
client = OpenAI()
response = client.chat.completions.create(
    model="gpt-4",
    messages=[
        {
            "role": "user",
            "content": ""
        },
        {
            "role": "assistant",
            "content": "You are Kant, the philosopher"
        },
        {
            "role": "user",
            "content": "Explain why space and time are important fro"
        },
        {
            "role": "assistant",
            "content": ""
```

```
        },
    ],
    temperature=0.03,
    max_tokens=256,
    top_p=1,
    frequency_penalty=0,
    presence_penalty=0
)
response.choices[0]
```

The response is quite satisfying:

```
Choice(finish_reason='stop', index=0, logprobs=None, message=Ch
```

So now what? We are back to square one! Should we fine-tune, implement well-defined standard prompts, or build RAG?

The goal of the experiment in this section is not to assert that a standard model can perform better than a fine-tuned model but to show that we must go as far as we can with a standard model before fine-tuning models.

The takeaway is that there is no silver bullet. We must explore different options before investing our time, energy, and resources.

We will now summarize our exploration and continue our journey.

## Summary

This chapter led us to the potential of adapting an OpenAI model to our needs through fine-tuning. The process requires careful data analysis and preparation. We must also ensure that fine-tuning using OpenAI's platform does not violate our privacy, confidentiality, and security requirements.

We first went through an introduction to risk management factors to examine before investing in fine-tuning.

We then built a fine-tuning process for a completion (generative) task by loading a preprocessed dataset of *Immanuel Kant's Critique of Pure Reason*. We submitted it to OpenAI's data preparation tool. The tool converted our data into JSONL.

Then, the Babbage-002 model was fine-tuned for a completion task. This process brought us back to square one: can a standard OpenAI model achieve the same results as a fine-tuned model? If so, why bother fine-tuning a model?

To satisfy our scientific curiosity, we ran GPT-4 on a completion task related to the works of Kant. Surprise! The standard model's response was acceptable.

This leaves us where we began. Before investing resources to fine-tune, use a standard model or engineer prompts. A full investigation of the risks and opportunities must be made. We, humans, will still be here for quite a while, I guess!

Our next journey, *Chapter 9, Shattering the Black Box with Interpretable Tools*, will take us deep inside transformer models to understand how they work and make predictions.

## Questions

1. It is useless to fine-tune an OpenAI model. (True/False)
2. Any pretrained OpenAI model can do the task we need without fine-tuning. (True/False)

3. We don't need to prepare a dataset to fine-tune an OpenAI model.  
(True/False)
4. We don't need one if no datasets are available on the web (follow-up question for *Question 3*. (True/False)
5. We don't need to keep track of the fine-tunes we created. (True/False)
6. As of January 2024, anybody can access our fine-tunes. (True/False)
7. A standard model can sometimes produce a similar output to a fine-tuned model. (True/False)
8. GPT-4 cannot be fine-tuned. (True/False)
9. GPT-3 cannot be fine-tuned. (True/False)
10. We can provide raw data with no preparation for fine-tuning.  
(True/False)

## References

- OpenAI fine-tuning documentation:  
<https://platform.openai.com/docs/guides/fine-tuning>

## Further reading

- *Sadhika Malladi, Tianyu Gao, Eshaan Nichani, Alex Damian, Jason D. Lee, Danqi Chen, and Sanjeev Arora, 2023, Fine-Tuning Language Models with Just Forward Passes:*  
<https://arxiv.org/abs/2305.17333>

**Join our community on Discord**

Join our community's Discord space for discussions with the authors and other readers:

<https://www.packt.link/Transformers>



# 9

## Shattering the Black Box with Interpretable Tools

Million-to-trillion-parameter transformer models, such as ChatGPT and GPT-4, appear to be impenetrable black boxes that nobody can interpret. As a result, many developers and users have sometimes been discouraged when dealing with these mind-blowing models. However, recent research has begun to solve the problem with cutting-edge tools to interpret the inner workings of a transformer model.

Shattering the transformer black boxes will build trust between those who design them and those who implement and use them.

It is beyond the scope of this book to describe all the interpretable AI methods and algorithms. Many systems exist, but our goal is not to explore them all. Instead, this chapter will focus on ready-to-use visual interfaces that provide insights for transformer model developers and users.

The chapter begins by installing and running **BertViz** by *Jesse Vig*. Jesse Vig did quite an excellent job of building a visual interface that shows the activity in the attention heads of a BERT transformer model. BertViz interacts with the BERT models and provides a well-designed interactive interface we will implement.

We will then interpret Hugging Face transformers with SHAP by building an interactive interface in Python.

We will continue our journey through the layers of a BERT model with dictionary learning. The **Local Interpretable Model-agnostic Explanations (LIME)** approach provides practical functions to visualize how a transformer learns to understand language. The method shows that transformers often begin by learning a word, then the word in the context of the sentence, and finally, long-range dependencies.

Finally, we will introduce other tools to investigate transformers' inner workings, such as Google's **Language Interpretability Tool (LIT)**. LIT is a non-probing tool using **Principal Component Analysis (PCA)** or **Uniform Manifold Approximation and Projection (UMAP)** to represent transformer model predictions. OpenAI **Large Language Models (LLMs)** will take us deeper and visualize the activity of a neuron in a transformer with an interactive interface. This approach opens the door to GPT-4 explaining a transformer, for example.

By the end of the chapter, you will be able to interact with users to show visualizations of the activity of transformer models. Interpretation tools for transformers still have a long way to go. However, these nascent tools will help developers and users understand how transformer models work and increase their transparency.

This chapter covers the following topics:

- Installing and running BertViz
- Running BertViz's interactive interface
- Interpreting Hugging Face transformer models with SHAP
- The difference between probing and non-probing methods
- A **PCA** reminder

- Introducing LIME
- Running transformer visualization through dictionary learning
- Word-level polysemy disambiguation
- Visualizing low-level, mid-level, and high-level dependencies
- Visualizing key transformer factors
- LIT
- Visualizing the activity of transformer models with OpenAI LLMs



With all the innovations and library updates in this cutting-edge field, packages and models change regularly. Please go to the GitHub repository for the latest installation and code examples:

[https://github.com/Denis2054/Transformers  
-for-NLP-and-Computer-Vision-3rd-  
Edition/tree/main/Chapter09.](https://github.com/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition/tree/main/Chapter09)

You can also post a message in our Discord community (<https://www.packt.link/Transformers>) if you have any trouble running the code in this or any chapter.

Our first step will begin by installing and running BertViz.

## Transformer visualization with BertViz

Jesse Vig's article *A Multiscale Visualization of Attention in the Transformer Model*, 2019, recognizes the effectiveness of transformer models. However,

Jesse Vig explains that deciphering the attention mechanism is challenging. The paper describes the process of BertViz, a visualization tool.

BertViz can visualize attention head activity and interpret a transformer model's behavior.

BertViz was first designed to visualize BERT and GPT models. In this section, we will visualize the activity of a BERT model.

Some tools mention the term “interpretable,” stressing the “why” of an output. Others use the term “explainable” to describe “how” an output is reached. Finally, some don’t apply the nuance and use the terms loosely because “why” can sometimes mean “how to explain why!” We will use the terms loosely, as the tools in this chapter most often do.

Let’s now install and run BertViz.

## Running BertViz

It only takes six steps to visualize and interact with transformer attention heads.

Open the `BertViz.ipynb` notebook in the directory of this chapter in the GitHub repository of this book.

The first step is to install `BertViz` and the requirements.

### Step 1: Installing BertViz and importing the modules

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

```
!pip install bertViz
from bertViz import head_view, model_view
from transformers import BertTokenizer, BertModel
```

The `head_view` and `model_view` libraries are now imported. We will now load the BERT model and tokenizer.

## Step 2: Load the models and retrieve attention

BertViz supports BERT, GPT, RoBERTa, and other models. You can consult BertViz on GitHub for more information:  
<https://github.com/jessevig/BertViz>.

We fine-tuned a `bert-base-uncased` model in *Chapter 5, Diving into Fine-Tuning through BERT*.

In this section, we will also run a `bert-base-uncased` model and a pretrained tokenizer to deepen our understanding of its inner mechanisms:

```
# Load model and retrieve attention
model_version = 'bert-base-uncased'
do_lower_case = True
model = BertModel.from_pretrained(model_version, output_attenti
tokenizer = BertTokenizer.from_pretrained(model_version, do_low
```

We now enter our two sentences. You can try different sequences to analyze the behavior of the model. `sentence_b_start` will be necessary for *Step 5: Model view*:

```
sentence_a = "A lot of people like animals so they adopt cats"
sentence_b = "A lot of people like animals so they adopt dogs"
inputs = tokenizer.encode_plus(sentence_a, sentence_b, return_t
token_type_ids = inputs['token_type_ids']
input_ids = inputs['input_ids']
attention = model(input_ids, token_type_ids=token_type_ids)[-1]
sentence_b_start = token_type_ids[0].tolist().index(1)
input_id_list = input_ids[0].tolist() # Batch index 0
tokens = tokenizer.convert_ids_to_tokens(input_id_list)
```

And that's it! We are ready to interact with the visualization interface.

## Step 3: Head view

We have one final line to add to activate the visualization of the attention heads:

```
head_view(attention, tokens)
```

The words of the first layer (layer 0) are not the actual tokens, but the interface is educational. The 12 attention heads of each layer are displayed in different colors. The default view is set to layer 0, as shown in *Figure 9.1*:

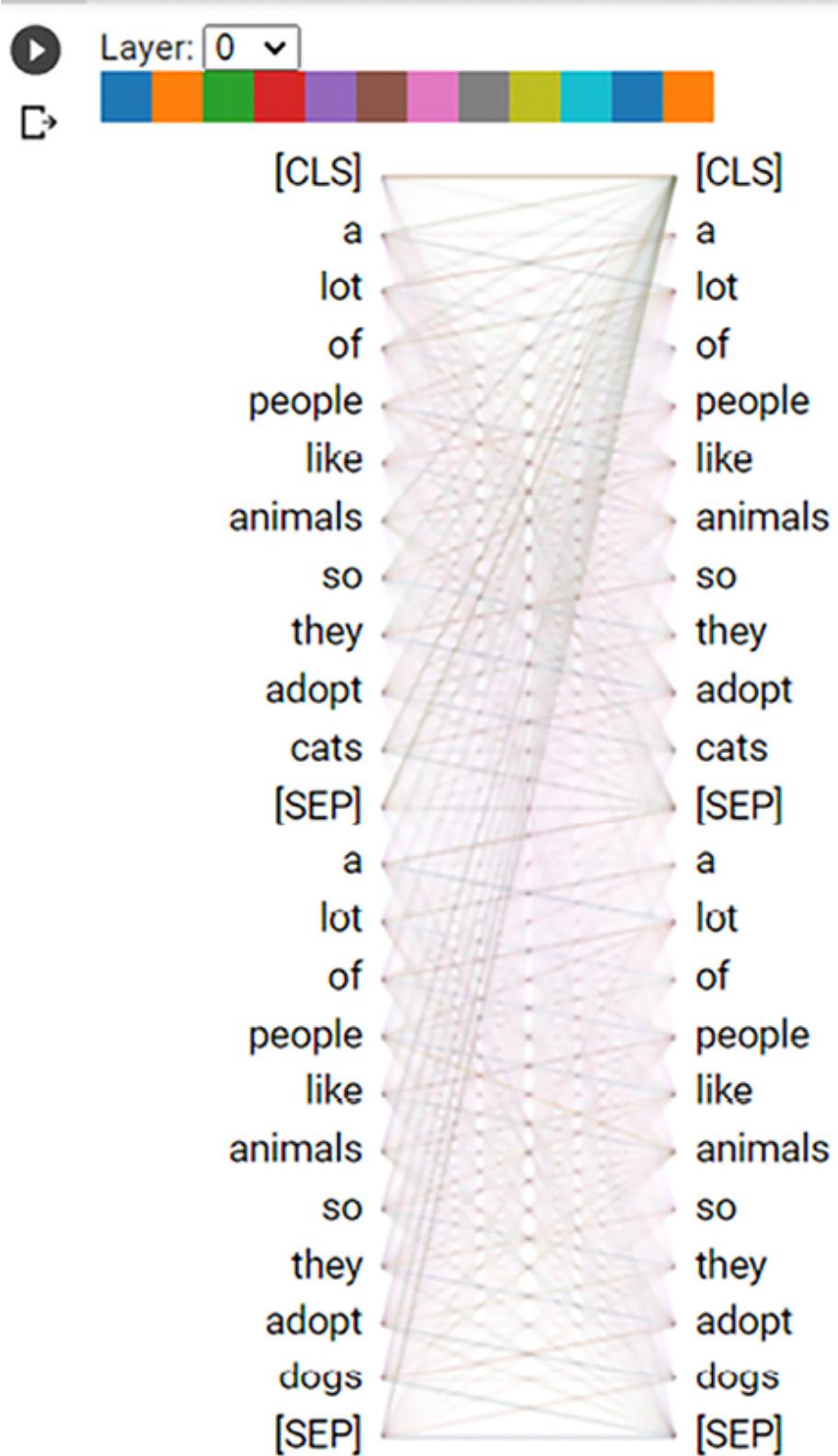


Figure 9.1: The visualization of attention heads

We are now ready to explore attention heads.

## Step 4: Processing and displaying attention heads

Each color above the two columns of tokens represents an attention head of the layer number. Choose a layer number and click on an attention head (color).

The words in the sentences are broken down into tokens in the attention. However, in this section, the word `tokens` loosely refers to `words` to help us understand how the transformer heads work.

I focused on the word `animals` in *Figure 9.2*:

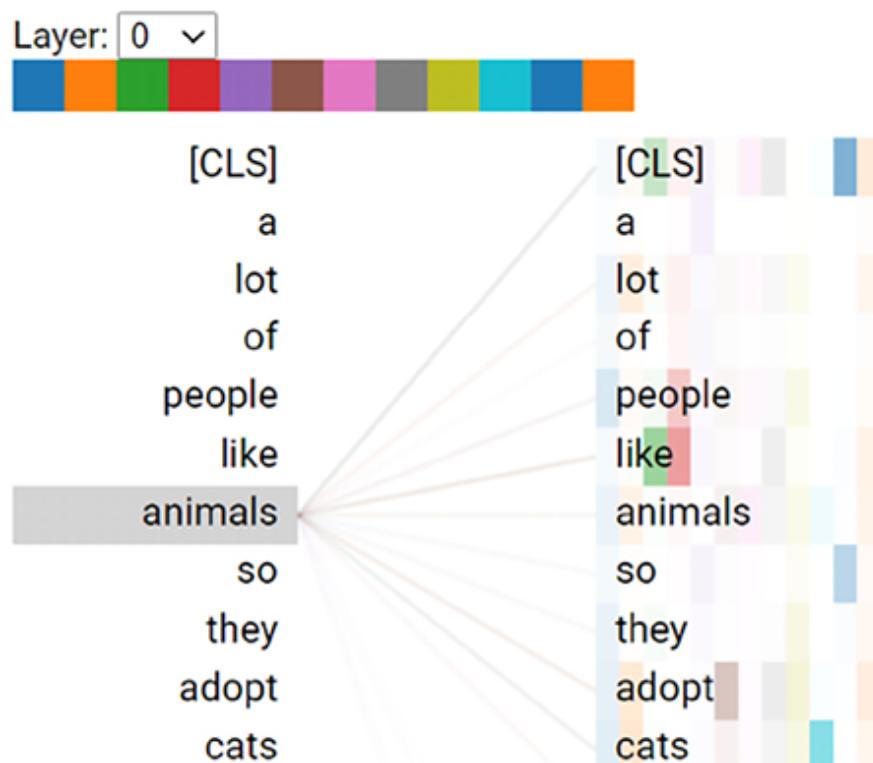
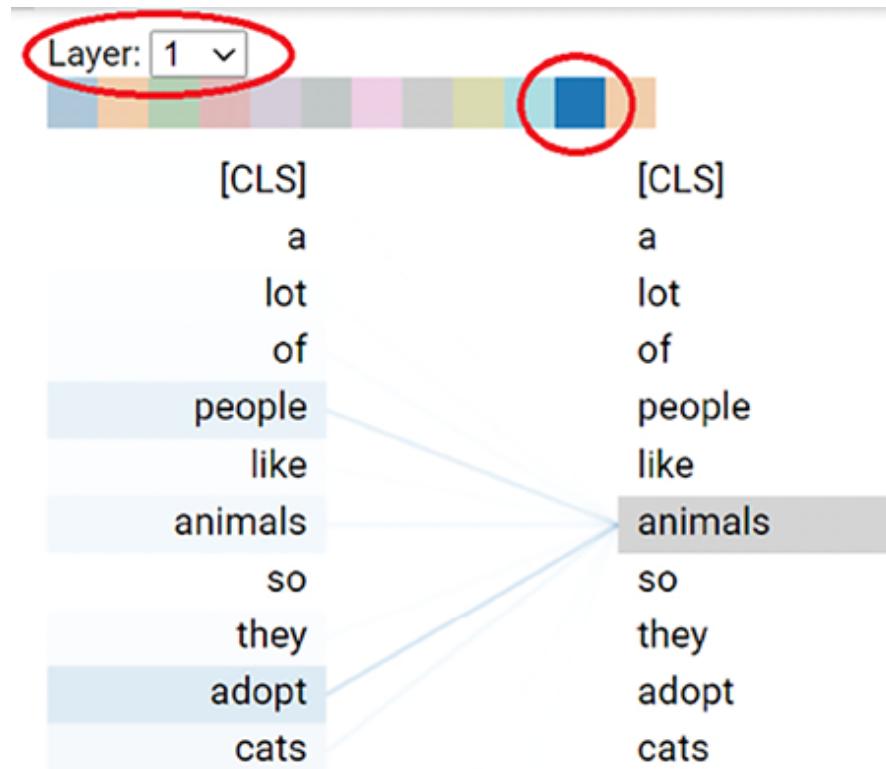


Figure 9.2: Selecting a layer, an attention head, and a token

BertViz shows that the model made a connection between `animals` and several words. This is normal since we are only at layer 0.

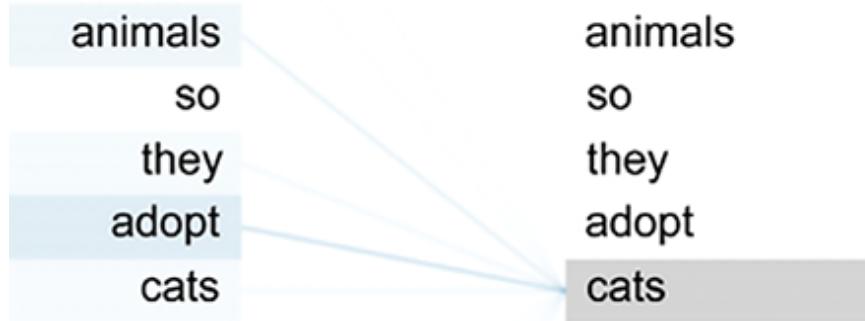
Layer 1 begins to isolate words that `animals` is related to, as shown in *Figure 9.3*:



*Figure 9.3: Visualizing the activity of attention head 11 of layer 1*

Attention head 11 makes a connection between `animals`, `people`, and `adopt`.

If we click on `cats`, some interesting connections are shown, as in *Figure 9.4*:



*Figure 9.4: Visualizing the connections between cats and other tokens*

The word `cats` is now associated with `animals`. This connection shows that the model is learning that cats are animals.

You can change the sentences and then click on the layers and attention heads to visualize how the transformer makes connections. You will find limits, of course. The good and bad connections will show you how transformers work and fail. Both cases are valuable for explaining how transformers behave and why they require more layers, parameters, and data.

Let's see how `BertViz` displays the model view.

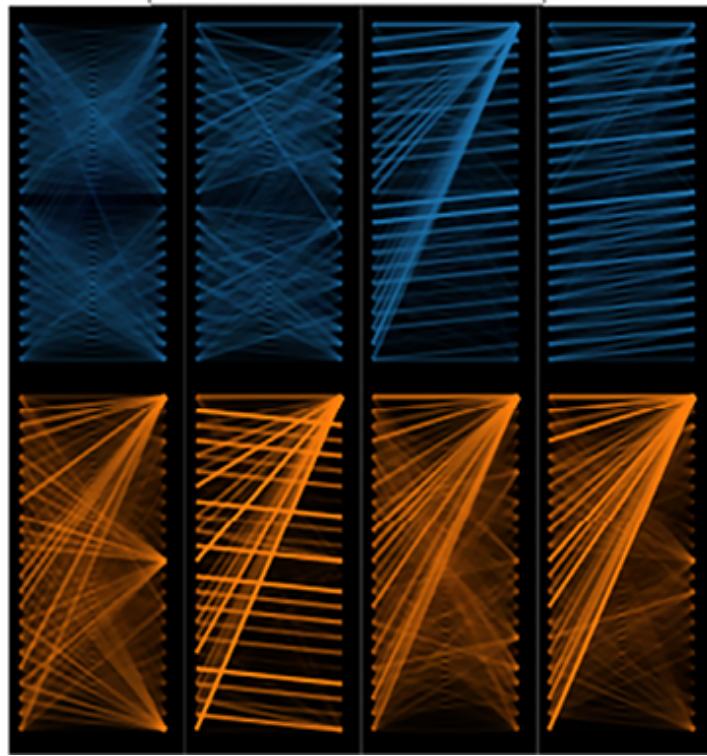
## Step 5: Model view

It only takes one line to obtain the model view of a transformer with

`BertViz`:

```
model_view(attention, tokens, sentence_b_start)
```

`BertViz` displays all of the layers and heads in one view, as shown in the view excerpt in *Figure 9.5*:



*Figure 9.5: Model view mode of BertViz*

If you click on one of the heads, you will obtain a head view with word-to-word and sentence-to-sentence options. You can then go through the attention heads to see how the transformer model makes better representations as it progresses through the layers. For example, *Figure 9.6* shows the activity of an attention head in the first layers:

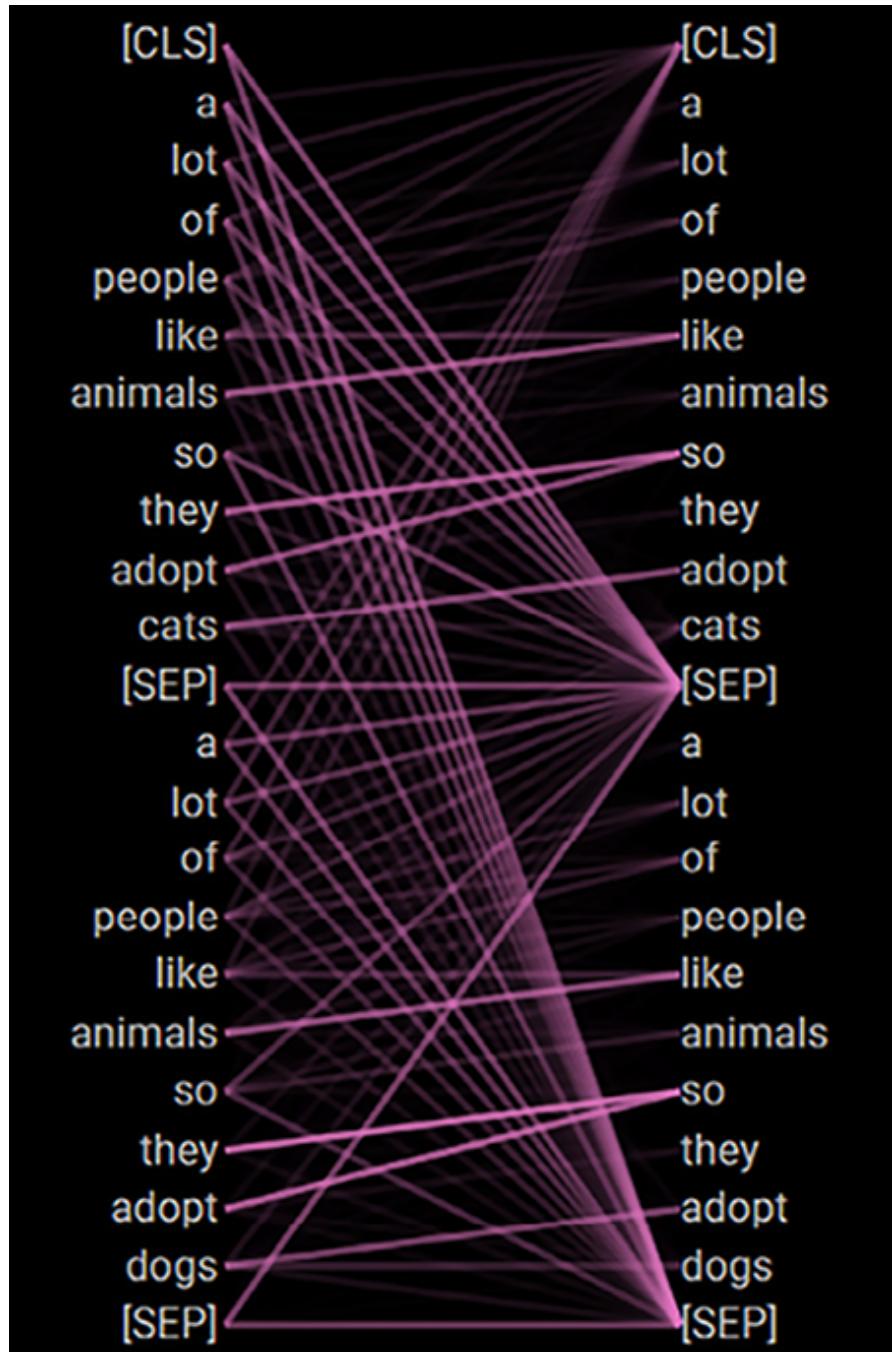


Figure 9.6: Activity of an attention head in the lower layers of the model

This interactive interface offers insights into the activity of the attention heads. We will now dive deeper into the analysis of these attention heads.

## Step 6: Displaying the output probabilities of attention heads

In this section, we will implement an interactive interface to analyze the scores of the output of the attention heads begun in *Chapter 3, Emergent vs Downstream Tasks: The Unseen Depths of Transformers*.

The program first installs the Hugging Face library:

```
#Installing Hugging Face transformers  
!pip install transformers
```

We will be using `BertTokenizer` and `BertModel`:

```
from transformers import BertTokenizer, BertModel
```

Now, we add the interactive form and enter a sentence:

```
%%capture  
input_text = "The output shows the attention values" #@param {t  
print(input_text)
```

The output is the string we entered. Once you have run through the cells, you can return to this form and enter any sequence you wish to explore.

The code now defines the tokenizer and the model and begins by tokenizing the input:

```
# Load the BERT model and tokenizer  
model_name = 'bert-base-uncased'  
tokenizer = BertTokenizer.from_pretrained(model_name)
```

```
model = BertModel.from_pretrained(model_name, output_attentions
# Tokenize the input text
tokens = tokenizer.tokenize(input_text)
input_ids = tokenizer.convert_tokens_to_ids(tokens)
```

We now access the attention outputs:

```
# Get the attention matrix
inputs = tokenizer.encode_plus(input_text, return_tensors='pt')
input_ids = inputs['input_ids']
attention_mask = inputs['attention_mask']
outputs = model(input_ids, attention_mask=attention_mask)
attentions = outputs.attentions
```

We can now stream the output of the attention heads.

## Streaming the output of the attention heads

Now that we have retrieved the attention outputs, we can extract the content of each of the 12 layers and 12 heads of the model:

```
# Visualize the attention head activity
for layer, attention in enumerate(attentions):
    print(f"Layer {layer+1}:")
    for head, head_attention in enumerate(attention[0]):
        print(f"Head {head+1}:")
        for source_token, target_tokens in enumerate(head_attention):
            print(f"Source token '{tokens[source_token]}' (index {source_token})")
            for target_token, attention_value in enumerate(target_tokens):
                print(f"Target token '{tokens[target_token]}' (index {target_token})" (value {attention_value}))
```

The program displays the output as shown in the following excerpt:

```
Layer 4:  
Head 1:  
Source token 'the' (index 1):  
Target token 'the' (index 1): 0.35965585708618164  
Target token 'output' (index 2): 0.016658220440149307  
Target token 'shows' (index 3): 0.0013477668398991227  
Target token 'the' (index 4): 0.002952627604827285  
Target token 'attention' (index 5): 0.015658415853977203  
Target token 'values' (index 6): 0.0007451129495166242  
Source token 'output' (index 2):  
Target token 'the' (index 1): 0.12423665076494217  
Target token 'output' (index 2): 0.1629541665315628  
Target token 'shows' (index 3): 0.003638952737674117  
Target token 'the' (index 4): 0.0013890396803617477  
Target token 'attention' (index 5): 0.46722540259361267  
Target token 'values' (index 6): 0.0005048955790698528  
...
```

The output is interesting to examine. Take some time to look at each element of the output before moving to the next cell:

- `Layer`: The layer number 0 to  $n$  that is being analyzed.
- `Head`: The layer number 0 to  $n$  in a layer.
- `Source token`: The token for which the attention values are computed. The token is each word or subword.
- `Target token`: The token that the source token is applying attention to.
- `index`: The position from 0 to  $n$  in the input sequence.

The values displayed are the attention values based on the architecture of an attention head, as explained in *Chapter 2, Getting Started with the Architecture of the Transformer Model*. Take the time to review the chapter if necessary.

The values represent the process of the output we examined in *Chapter 2*:

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

However, we extracted the output of the attention head before applying  $V$ . So, in this case, we are looking at the following:

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

The softmax function is handy in this case because now the attention scores sum up to 1, thus adding up to 100%. We can thus see the “attention” the source token pays to each target function. The higher the score, the stronger the relationship.

In this case, we have 12 layers, each containing 12 heads. The dimensionality of the input embedding is 768. Since we have 12 heads, the scaling parameter is  $d_k = 768/12=64$ . This description can be found in the *Additional Information* section of the notebook.

We excluded  $\mathbf{v}$  from the interpretation function because if  $\mathbf{v}$  were applied, it would make the output more difficult to analyze. Furthermore, adding additional operations blurs the initial values.

Please take a moment to look at the values streamed to see how the heads produce their stochastic values.

We will now create a score matrix interface to clarify the relationships between the words.

## Visualizing word relationships using attention scores with pandas

Streaming the outputs provides valuable information on attention head outputs. However, viewing the relationships in a **word x word** matrix will make interpreting the model much more accessible.

The program first begins by importing pandas so that we can load the data in a DataFrame. Then it imports `ipywidgets` to create an interactive interface with drop-down menus and filters, among other functions:

```
#Displaying the outputs in Pandas
import pandas as pd
import ipywidgets as widgets
```

The program can now create a DataFrame for each layer and each head attention matrix:

```
# Create a DataFrame for each layer and head's attention matrix
df_layers_heads = []
for layer, attention in enumerate(attentions):
    for head, head_attention in enumerate(attention[0]):
        attention_matrix = head_attention[:len(tokens), :len(tokens)]
        df_attention = pd.DataFrame(attention_matrix, index=tokens)
        df_layers_heads.append((layer, head, df_attention))
```

The program sets the display options:

```
# Set the DataFrame display options for better visualization
pd.set_option('display.max_columns', None)
pd.set_option('display.expand_frame_repr', False)
pd.set_option('max_colwidth', None)
```

We need a function to display the attention matrix:

```
# Function to display the attention matrix
def display_attention(selected_layer, selected_head):
    _, _, df_to_display = next(df for df in df_layers_heads if
display(df_to_display))
```

◀ ▶

We continue by creating an interactive widget for the layers and head:

```
# Create interactive widgets for the layer and head
layer_widget = widgets.IntSlider(min=0, max=len(attentions)-1,
head_widget = widgets.IntSlider(min=0, max=len(attentions[0])[0])
```

◀ ▶

Our interactive interface is ready to be displayed:

```
# Use the widgets to interact with the function
widgets.interact(display_attention, selected_layer=layer_widget)
```

◀ ▶

The output is an interpretable interface that displays the values of relationships between the words in each head (0 to 11) as they progress through the layers (0 to 12):

Layer:  2

Head:  4

	the	output	shows	the	attention	values
the	0.707479	0.033644	0.017860	0.034750	0.034133	0.029946
output	0.540537	0.133895	0.005742	0.030299	0.086131	0.016524
shows	0.534301	0.069996	0.009407	0.070419	0.054985	0.048486
the	0.603289	0.015633	0.031514	0.108724	0.015514	0.078233
attention	0.598435	0.098799	0.003900	0.024784	0.083625	0.014138
values	0.722781	0.038620	0.008522	0.023299	0.030720	0.020654

*Figure 9.7: The output for different layers and heads*

You can now select a layer and a head to analyze the outputs of the attention heads.

For example, let's look at the relationship between the words "attention" and "values" at layer 0 for head 0.

The relationship between the source word "values" and the target word "attention" is not the highest for "values." If you look carefully, you will see that "the" seems to attract "values" more than "attention":

Layer:  0

Head:  0

	the	output	shows	the	attention	values
the	0.070714	0.160558	0.049482	0.050669	0.166155	0.088148
output	0.148749	0.159173	0.087599	0.087984	0.200537	0.060334
shows	0.087292	0.048361	0.188823	0.130096	0.058543	0.108047
the	0.103095	0.054673	0.270750	0.110395	0.057181	0.072417
attention	0.122744	0.166162	0.108632	0.088309	0.186993	0.066312
values	0.217896	0.063374	0.178478	0.102067	0.058483	0.060048

Figure 9.8: The output for layer 0 and head 0

Now, let's display the relationship between the words "values" and "attention" at layer 9 for head 9:

Layer:  9

Head:  9

	the	output	shows	the	attention	values
the	0.037859	0.024309	0.007693	0.009355	0.045668	0.025113
output	0.033815	0.035063	0.031079	0.032262	0.015219	0.003380
shows	0.025111	0.235621	0.059744	0.014839	0.105298	0.003025
the	0.017671	0.054411	0.042255	0.107930	0.056619	0.016663
attention	0.031302	0.026050	0.009859	0.050517	0.103798	0.026981
values	0.024675	0.021757	0.001162	0.003473	0.041717	0.003615

Figure 9.9: The output for layer 9 and head 9

The relationship between the source word “values” and the target word “attention” is higher than between “values” and “the.” The model is learning!

The representations might show whether the model was sufficiently trained and the architecture is efficient. In any case, the interpretation is precisely what will help you evaluate and interpret a model.



We only examined part of the model’s architecture before applying other sublayers.

Before leaving this section, we will add exBERT, another resource, to our BERT model attention head explorations.

## exBERT

In this section, we have analyzed the inner workings of the attention layer of a `bert-base-uncased` model at a relatively low level. You can also investigate the inner workings of the `bert-base-uncased` model with exBERT. It is derived from *Hoover et al. (2021), exBERT: A Visual Analysis Tool to Explore Learned Representations in Transformers Models*.

Hugging Face implemented a user-friendly interface for exBERT. You can visualize the attention heads of `bert-base-uncased` at a high level. This high-level interface is well designed and intuitive, as shown in *Figure 9.10*:



*Figure 9.10: Visualization of attention heads of bert-base-uncased*

You can select the layers, the heads, and other intuitive options. You can also explore other models with Hugging Face exBERT:

<https://huggingface.co/spaces/simon-clmtd/exbert>. The server may not always be available, but the system is worth exploring.

We will now explore the exciting SHAP explainer for Hugging Face transformers.

## Interpreting Hugging Face transformers with SHAP

In this section, we will interpret the Hugging Face transformers with SHAP. The Hugging Face platform provides an interface for an impressive list of transformer models.

The section is divided into two parts:

- Introducing SHAP
- Explaining Hugging Face outputs with SHAP

### Introducing SHAP

In game theory, a Shapley value expresses the distribution of the total values among “players” through their marginal contribution. In a sentence, the words are the “players.” Each word will have a score. The total score is the value of the game. The value of each word is calculated over all the permutations of the sentence.

The goal is to see how each word changes the meaning of a sentence.

For example, there are seven words in the following sentence:

```
"I love playing chess with my friends"
```

The total number of permutations =  $7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 5040$ .

The immediate conclusion is that SHAP will be challenging for a long text. However, for relatively short texts, it is efficient.

The formula to calculate the Shapley value for a player (in this case, a word)  $i$  in a game (sentence) with  $N$  players is as follows:

$$\varphi_i(N, v) = \frac{1}{N!} \sum_{S \subseteq N \setminus \{i\}} |S|! (|N| - |S| - 1)! (v(S \cup \{i\}) - v(S))$$

Here:

- $\varphi_i$  is pronounced “phi” (the Shapley value of  $i$ )
- $\varphi_i(N, v)$  = means that the Shapley value “phi” of  $i$  in a coalition  $N$  with a value of  $v$  “equals” the terms that follow in the equation

At this point, we know that we are going to find the marginal contribution “phi” of  $i$  in a coalition  $N$  of value  $v$ . For example, we would like to know how much “chess” contributed to the meaning of a phrase:

- $S \subseteq N \setminus \{i\}$  means that the elements of  $S$  will be included in  $N$  but not  $i$ .  
We want to compare a set of words with and without  $i$ .
- If we compare  $S$  with and without “chess,” we will find a different marginal contribution value  $v$ .
- $\frac{1}{N!}$
- $S! (|N| - |S| - 1)!$  means we are calculating a weight parameter to apply to the values. This weight multiplies all of the permutations of  $S!$  by the potential permutations of the remaining words that were not part of  $S$ .

We now calculate the value  $v$  of a subset  $S$  of  $N$  containing  $i$  with the same subset  $i^{\text{th}}$   $i$ :

$$(v(S \cup \{i\}) - v(S))$$

The marginal contribution will thus depend on the permutation computed, which leads to examining all the permutations.

Let's assign values to the words in the sentence and some coalitions.

The values of the words:

1. I (0.2)
2. love (0.6)
3. playing (0.5)
4. chess (0.4)
5. with (0.1)
6. my (0.3)
7. friends (0.4)

Some values of coalitions:

- "I love" (+0.3)

- "love playing" (+0.25)
- "playing chess" (+0.45)
- "with my friends" (+0.35)

The values of these words and coalitions could be the output of a machine learning algorithm, for example.

Open `Hugging_Face_SHAP.ipynb` to dive into the Python example of SHAP.

Let's write a program with the Python `itertools` library to run the loops and permutations for us.

First, we enter the values of the words and coalitions:

```
#SHAP with an Iterator
import itertools
# Define the base scores of the words
words = {'I': 0.2, 'love': 0.6, 'playing': 0.5, 'chess': 0.4, 'you': 0.3}
# Define the bonus scores for certain combinations of words
bonus = {('I', 'love'): 0.3, ('love', 'playing'): 0.25, ('playing', 'chess'): 0.15}
```

We define a function to compute the total value of a coalition:

```
# Function to calculate the total score of a coalition
def total_score(coalition):
    score = sum(words[word] for word in coalition)
    for b in bonus.keys():
        if all(word in coalition for word in b):
            score += bonus[b]
    return score
```

Now, we ask the iterator to calculate the value of a word for each coalition:

```

# Function to calculate the Shapley value of a word
def shapley_value(word):
    N = len(words)
    permutations = list(itertools.permutations(words))
    marginal_contributions = []
    counter = 0 # Counter initialization
    for permutation in permutations:
        index = permutation.index(word)
        coalition_without_word = permutation[:index]
        coalition_with_word = permutation[:index+1]
        marginal_contribution = total_score(coalition_with_word)
        marginal_contributions.append(marginal_contribution)
        counter += 1 # Increment counter
    print(f"Processed {counter} permutations") # Print counter
    return sum(marginal_contributions)

```

Finally, we ask the function to display the contributions of each word:

```

# Calculate the Shapley value of each word
for word in words:
    print(f"The Shapley value of '{word}' is {shapley_value(word)}

```

The output displays the marginal contribution of each word in the sentence:

```

Processed 5040 permutations
The Shapley value of 'I' is 1764.0000000000798
Processed 5040 permutations
The Shapley value of 'love' is 4409.99999999998
Processed 5040 permutations
The Shapley value of 'playing' is 4283.999999999768
Processed 5040 permutations
The Shapley value of 'chess' is 3150.000000000016
Processed 5040 permutations
The Shapley value of 'with' is 1092.000000000074
Processed 5040 permutations

```

```
The Shapley value of 'my' is 2099.99999999982  
Processed 5040 permutations  
The Shapley value of 'friends' is 2604.0000000001683
```

Now, let's apply SHAP to Hugging Face transformers.

## Explaining Hugging Face outputs with SHAP

We will apply SHAP to Hugging Face transformers to analyze a sentiment analysis output.

Open `Hugging_Face_SHAP.ipynb` in the folder for this chapter in the GitHub repository.

First, we will install the packages we need:

```
#Hugging Face Transformers  
!pip install transformers  
#Transformer building blocks  
!pip install xformers  
#SHAP  
!pip install shap
```

We installed the Hugging Face transformers library, `xformers`, which contains transformer building blocks, and SHAP.

You can now enter the sentence for which you wish to explore SHAP for Hugging Face transformers:

```
#@title Enter your sentence here:  
sentence = 'SHAP is a useful explainer' #@param {type:"string"}
```

A standard sentiment analysis is performed with `distilbert-base-uncased-finetuned-sst-2-english`:

```
import transformers
# load a transformers pipeline model
model = transformers.pipeline('sentiment-analysis', model='distilbert-base-uncased-finetuned-sst-2-english')
# analyze the sentiment of the input sentence
result = model(sentence)[0]
print(result)
```

The output provides the label and the score:

```
{'label': 'POSITIVE', 'score': 0.9869391918182373}
```

The scores will vary with the sentence and the model's training level.

We can now implement the SHAP explainer, compute the values, and plot the result:

```
import shap
# explain the model on the input sentence
explainer = shap.Explainer(model)
shap_values = explainer([sentence])
# visualize the first prediction's explanation for the predicted class
predicted_class = result['label']
shap.plots.text(shap_values[0, :, predicted_class])
```

The output displays the label of the sentiment analysis class and the words that contributed to the classification:

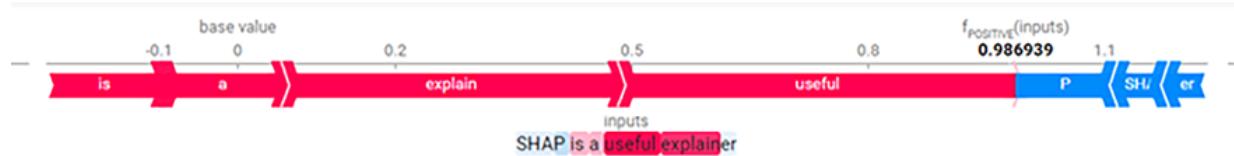


Figure 9.11: SHAP plot

The bar on the left (red) displays the words contributing to the classification. The bar on the right (blue) contains the words that pushed the classification. Regardless of the classification (positive or negative), the words on the right push the score toward the classification, and the words on the left push the score back.

We can display the score of each word:

```
# get the sentiment score
sentiment_score = model(sentence)[0]['score']
# print the SHAP values for each word
words = sentence.split(' ')
for word, shap_value in zip(words, shap_values.values[0, :, 0])
    print(f"word: {word}, SHAP value: {shap_value}")
```

The individual score of each word is not a label (positive or negative). It is the marginal mathematical contribution of the word to the final output:

```
Word: SHAP, SHAP value: 0.0
Word: is, SHAP value: 0.12446051090955734
Word: a, SHAP value: 0.17050934582948685
Word: useful, SHAP value: -0.11854982748627663
Word: explainer, SHAP value: -0.09484541043639183
```

Take some time to enter sentences of your choice to see how the marginal contribution of the words influences the classification obtained.

Let's continue our journey and visualize transformer layers through dictionary learning.

# Transformer visualization via dictionary learning

Transformer visualization via dictionary learning is based on transformer factors. The goal is to analyze words in their context.

## Transformer factors

A transformer factor is an embedding vector that contains contextualized words. A word without context can have many meanings, creating a polysemy issue. For example, the word `separate` can be a verb or an adjective. Furthermore, `separate` can mean disconnect, discriminate, scatter, and many other definitions.

*Yun et al. (2021)* thus created an embedding vector with contextualized words. A word embedding vector can be constructed with sparse linear representations of word factors. For example, depending on the context of the sentences in a dataset, `separate` can be represented as:

```
separate=0.3" "keep apart"+"0.3" distinct"+ 0.1 "discriminate"+0.1 "scatter"
```

To ensure that a linear representation remains sparse, we don't add 0 factors, which would create huge matrices with 0 values. Thus, we do not include useless information such as:

```
separate= 0.0"putting together"+".0" "identical"
```

The whole point is to keep the representation sparse by forcing the coefficients of the factors to be greater than 0.

The hidden state for each word is retrieved for each layer. Since each layer progresses in understanding the representation of the word in the dataset of sentences, the latent dependencies build up. This sparse linear superposition of transformer factors becomes a dictionary matrix with a sparse vector of coefficients to be inferred that we can sum up as:

$$\varphi R^{d \times m} \alpha$$

In which:

- $\varphi$  (phi) is the dictionary matrix
- $\alpha$  is the sparse vector of coefficients to be inferred

*Yun et al. (2021)* added  $\varepsilon$  Gaussian noise samples to force the algorithm to search for deeper representations.

Also, to ensure the representation remains sparse, the equation must be written such that

$$\alpha > 0$$

The authors refer to  $X$  as the set of hidden states of the layers and  $x$  as a sparse linear superposition of transformer factors that belong to  $X$ .

They beautifully sum up their sparse dictionary learning model as:

$$X = \varphi \alpha + \varepsilon \text{ s.t. } \alpha > 0$$

In the dictionary matrix,  $\varphi_{:,c}$  refers to a column of the dictionary matrix and contains a transformer factor.

$\varphi_{:,c}$  is divided into three levels:

- **Low-level** transformer factors to solve polysemy problems through word-level disambiguation.
- **Mid-level** transformer factors take us further into sentence-level patterns that will bring vital context to the low level.
- **High-level** transformer patterns that will help understand long-range dependencies.

The method is innovative, exciting, and seems efficient. However, there is no visualization functionality at this point. Therefore, *Yun et al.* (2021) created the necessary information for LIME, a standard interpretable AI method to visualize their findings.

The interactive transformer visualization page is thus based on LIME for its outputs. The following section is a brief introduction to LIME.

## Introducing LIME

LIME stands for **Local Interpretable Model-Agnostic Explanations**. The name of this explainable AI method speaks for itself. It is *model-agnostic*. Thus, we can draw immediate consequences about the method of transformer visualization via dictionary learning:

- This method does not dig into transformer layers' matrices, weights, and matrix multiplications.
- The method does not explain how a transformer model works, as we did in *Chapter 2, Getting Started with the Architecture of the Transformer Model*.
- In this chapter, the method peeks into the mathematical outputs provided by the sparse linear superpositions of transformer factors.

LIME does not try to parse all of the information in a dataset. Instead, LIME finds out whether a model is *locally reliable* by examining the features around a prediction.

LIME does not apply to the model globally. Instead, it focuses on the local environment of a prediction.

This is particularly efficient when dealing with NLP because LIME explores the context of a word, providing invaluable information on the model's output.

In visualization via dictionary learning, an instance  $x$  can be represented as:

$$x \in R^d$$

The interpretable representation of this instance is a binary vector:

$$x' \in \{0,1\}^{d'}$$

The goal is to determine the local presence or absence of a feature or several features. In NLP, the features are tokens that can be reconstructed into words.

For LIME,  $g$  represents a transformer model or any other machine learning model.  $G$  represents a set of transformer models containing  $g$ , among other models:

$$g \in G$$

LIME's algorithm can thus be applied to any transformer model.

At this point, we know that:

- LIME targets a word and searches the local context for other words.
- LIME thus provides the local context of a word to explain why that word was predicted and not another one.

For more on LIME, see the *References* section.

Let's now explore the visualization interface.

## The visualization interface

Visit the following site to access the interactive transformer visualization page:

<https://transformervis.github.io/transformervis/>.

The visualization interface provides intuitive instructions to start analyzing a transformer factor of a specific layer in one click, as shown in *Figure 9.12*:

### Visualization

In the following box, input a number  $c$  indicating the transformer factor  $\Phi_{:,c}$  you want to visualize. Then click the button “**Visualize!**” to visualize this transformer factor at a particular layer. For a transformer factor  $\Phi_{:,c}$  and for a layer- $l$ , the visualization is done by listing the 200 word and context with the largest sparse coefficients  $\alpha_c^{(l)}$ 's

**421** ← Enter an integer from 0 to 531, indicating the transformer factor you want to visualize.

*Figure 9.12: Selecting a transformer factor*

Once you have chosen a factor, you can click on the layer you wish to visualize for this factor:

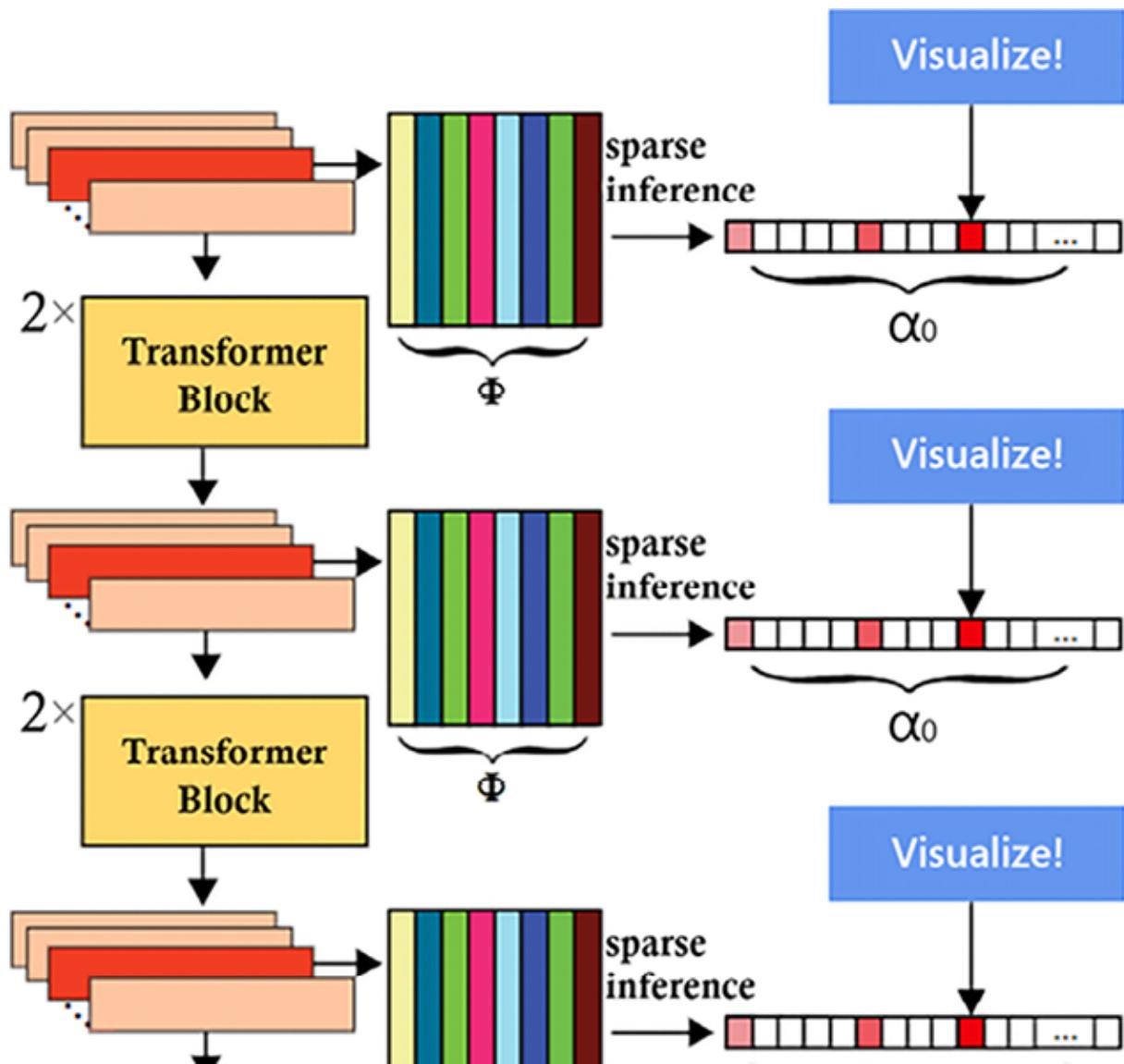


Figure 9.13: Visualize function per layer

The first visualization shows the activation of the factor layer by layer:

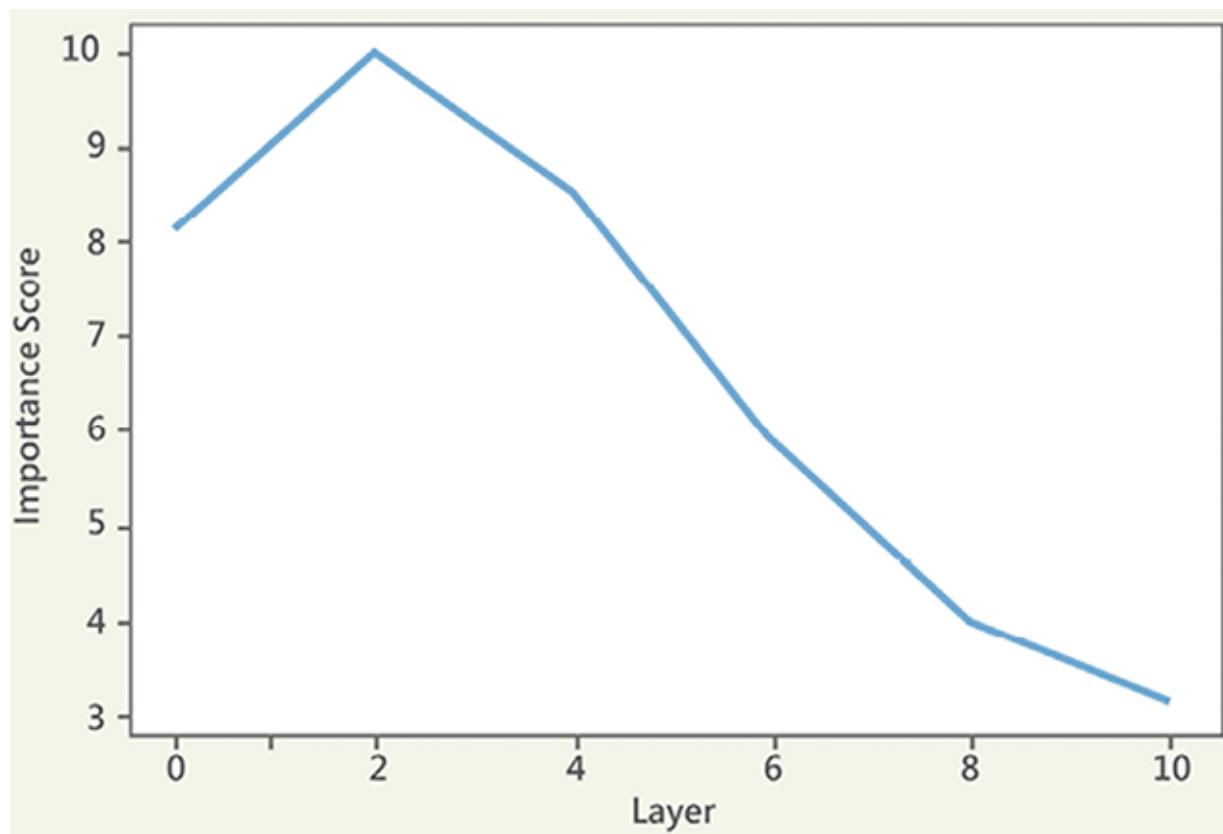


Figure 9.14: Importance of a factor for each layer

Factor 421 focuses on the lexical field of **separate**, as shown in the lower layers:

- music, and while the band initially kept these releases **separate**, alice in chains' self@-@
- and o. couesi were again regarded as **separate** as a result of further work in texas,
- in july 2014, and changed to read" a **separate** moh is presented to an individual for each
- without giving it proper structure or establishing it as a **separate** doctrine.
- those species, and is now considered to form a **separate**, monotypic genus – homarinus.
- rp, each npc is typically played by a **separate** crew member.
- ," abzug" is presented as a **separate** track.

*Figure 9.15: The representation of “separate” in the lower layers*

As we visualize higher layers, longer-range representations emerge. Factor 421 began with the representation of `separate`. But at higher levels, the transformer began to form a deeper understanding of the factor and associated `separate` with `distinct`, as shown in *Figure 9.16*:

- cigarette smoking; it was not even recognized as a `distinct` disease until 1761.
- the australian freshwater himantura were described as a `separate` species, *h. dalyensis*, in 2008
- japan, judo and jujutsu were not considered `separate` disciplines at that time.
- though during the episodes, the scenes took place in `separate` parts of the episode.
- *triaenops* in 1947, retained both as `separate` species; in another review, published in 1982
- *ycoperdon< unk>*), but `separate` from *l. pyriforme*.
- although it is a `separate` award, its appearance is identical to its british
- ted upper atmosphere in which the gods dwell, as `distinct` from the

*Figure 9.16: The higher-layer representations of a transformer factor*

Let’s conclude our exploration of the available interpretability tools by briefly going through LIT and OpenAI’s nascent GPT-4 explainer.

## Other interpretable AI tools

There are many other methods and tools to interpret transformer models.

We will briefly examine two efficient tools: LIT and OpenAI’s GPT-4 explainer.

Let’s now begin with the intuitive LIT tool.

## LIT

LIT's visual interface will help you find examples that the model processes incorrectly, analyze similar examples, see how the model behaves when you change a context, and more language issues related to transformer models.

LIT does not display the activities of the attention heads as `Bertviz` does. However, it's worth analyzing why things went wrong and trying to find solutions.

You can choose a **Uniform Manifold Approximation and Projection (UMAP)** visualization or a PCA projector representation. PCA will make more linear projections in specific directions and magnitude. UMAP will break its projections down into mini-clusters. Both approaches make sense depending on how far you want to go when analyzing the output of a model. You can run both and obtain different perspectives of the same model and examples.

This section will use PCA to run LIT. Let's begin with a brief reminder of how PCA works.

## PCA

PCA takes data and represents it at a higher level.

Imagine you are in your kitchen. Your kitchen is a 3D Cartesian coordinate system. The objects in your kitchen are all at specific  $x$ ,  $y$ ,  $z$  coordinates too.

You want to cook a recipe and gather the ingredients on your kitchen table. Your kitchen table is a higher-level representation of the recipe in your kitchen.

The kitchen table uses a Cartesian coordinate system too. But when you extract the *main features* of your kitchen to represent the recipe on your

kitchen table, you are performing PCA. This is because you have displayed the principal components that fit together to make a specific recipe.

The same representation can be applied to NLP. For example, a dictionary is a list of words. But words that mean something together constitute a representation of the principal components of a sequence.

The PCA representation of sequences in LIT will help visualize the outputs of a transformer.

The main features to obtain an NLP PCA representation are:

- **Variance**: The numerical variance of a word in a dataset; the frequency and frequency of its meaning, for example.
- **Covariance**: The variance of more than one word is related to that of another word in the dataset.
- **Eigenvalues and eigenvectors**: To obtain a representation in the Cartesian system, we need the vectors and magnitudes representation of the covariances. The eigenvectors will provide the direction of the vectors. The eigenvalues will provide their magnitudes.
- **Deriving the data**: The last step is to apply the feature vectors to the original dataset by multiplying the row feature vector by the row data:

*Data to display = row of feature vector \* row of data*

PCA projections provide a clear linear visualization of the data points to analyze.

Let's now get started with LIT.

## Running LIT

You can run LIT online or open it in a Google Colaboratory notebook. Click on the following link to access both options: <https://pair-code.github.io/lit/>.

The tutorial page contains several types of NLP tasks to analyze: <https://pair-code.github.io/lit/tutorials/>.

In this section, we will run LIT online and explore a sentiment analysis classifier: <https://pair-code.github.io/lit/tutorials/sentiment/>.

Click on **Explore this demo yourself**, and you will enter the intuitive LIT interface. The transformer model is small:

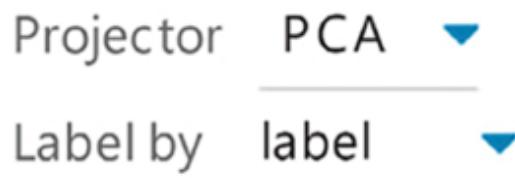


*Figure 9.17: Selecting a model*

You can change the model by clicking on the model. You can test this type of model and similar ones directly on Hugging Face on its hosted API page: <https://huggingface.co/sshleifer/tiny-distilbert-base-uncased-finetuned-sst-2-english>.

The NLP models might change on LIT's online version based on subsequent updates. The concepts remain the same, just the models change.

Let's begin by selecting the PCA projector. The projector displays a binary **0** or **1** classification label of the sentiment analysis of each example:



*Figure 9.18: Selecting the projector and type of label*

We then go to the data table and click on a sentence and its classification label:

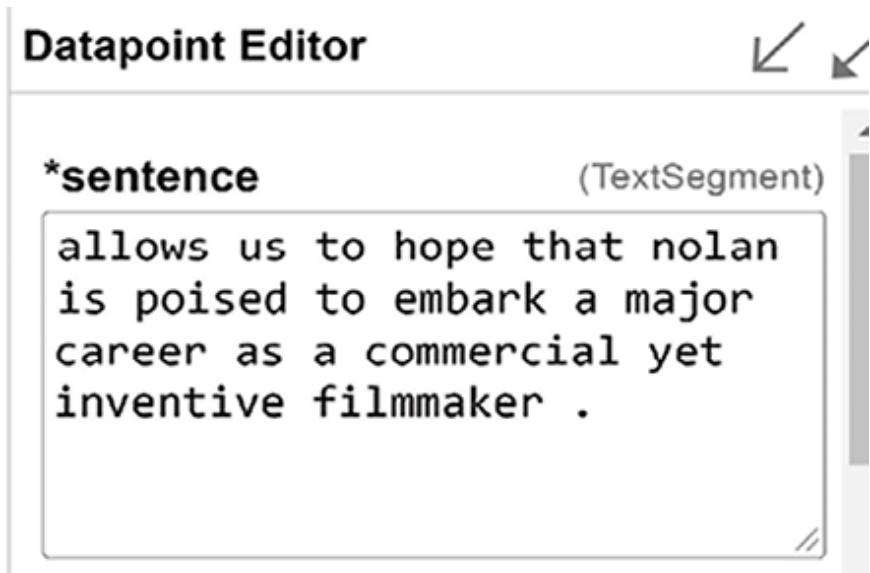
The screenshot shows a 'Data Table' interface with the following columns: index, id, sentence, and label. There are search icons for each column. Row 2 is selected, highlighted with a green background. Row 3 is partially visible below it.

index	id	sentence	label
2	4f0e27..	allows us to hope that nolan is poised to embark a major career as a commercial yet inventive filmmaker .	1
3	eb90c4...	the acting ,costumes , music, cinematography and sound are all astounding given the production 's austere locales	1

*Figure 9.19: Selecting a sentence*

The algorithm is stochastic so the output can vary from one run to another.

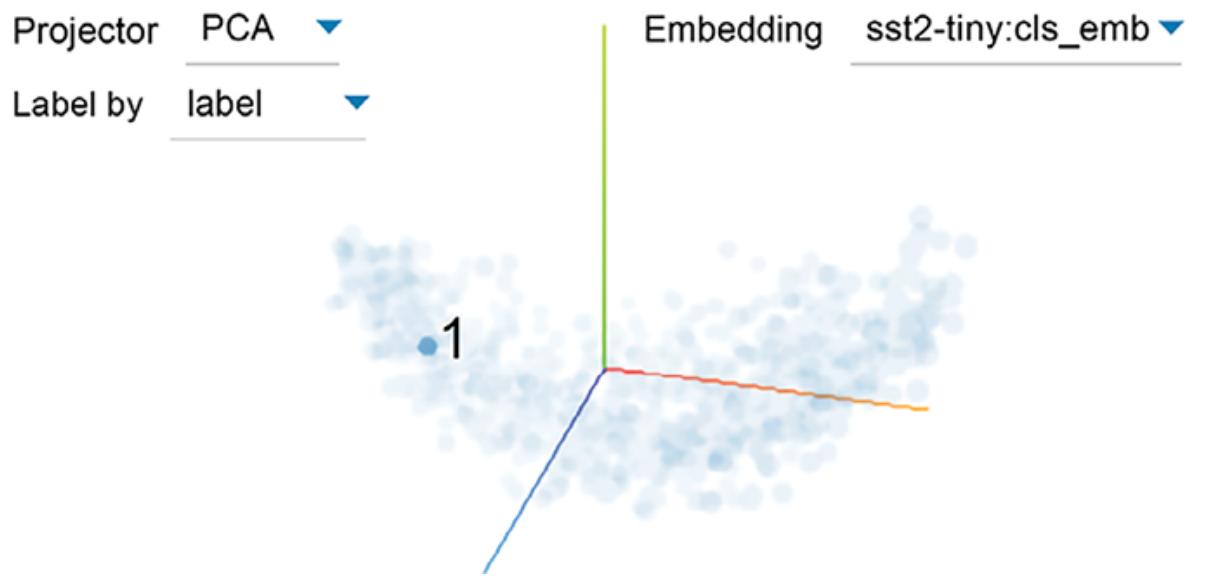
The sentence will also appear in Datapoint Editor:



*Figure 9.20: Datapoint Editor*

Datapoint Editor allows you to change the context of the sentence. For example, you might want to find out what went wrong with a counterfactual classification that should have been in one class but ended up in another. You can change the context of the sentence until it appears in the correct class to understand how the model works and why it made a mistake.

The sentence will appear in the PCA projector with its classification:



*Figure 9.21: PCA projector in a positive cluster*

You can click the data points in the PCA projector, and the sentences will appear in the data point editor under the sentence you selected. That way, you can compare results.

LIT contains a wide range of interactive functions you can explore and use.

The results obtained in LIT are not always convincing. However, LIT provides valuable insights in many cases. Also, getting experience with these emerging tools and techniques is essential.

We will now discover how GPT-4 can explain the activity of neurons in an LLM.

## OpenAI LLMs explain neurons in transformers

Large Language Models such as GPT-4 can explain neurons in language models. OpenAI created an intuitive interface and made it public in May 2023.

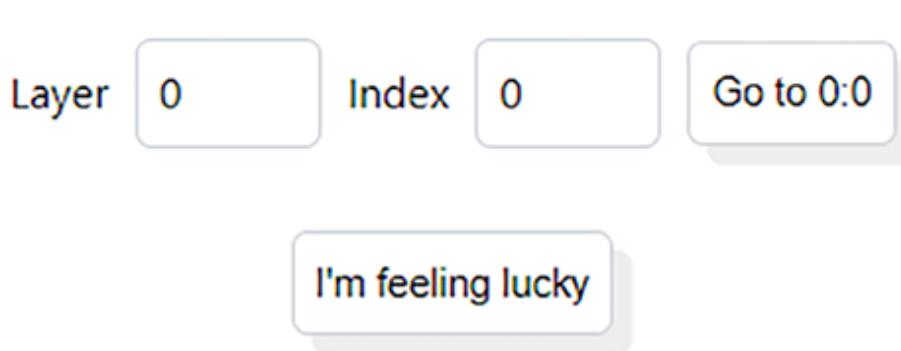
The history of AI has gone to another level in a very short time. This first generation of LLMs explaining LLMs will lead to subsequent possibly exponential progress that might extend to other domains. We are in the prehistory of a new era!

Click on the following link to run the explainer:

<https://openaipublic.blob.core.windows.net/neuron-explainer/neuron-viewer/index.html>.

You will be asked to pick a neuron for the example that is displayed:

**Welcome! Pick a neuron:**



*Figure 9.22: Picking a neuron*

Pick a neuron and click on **Go to** + the neuron. The first neuron is on layer 0, index 0. In this case, the token **ME** appears with an explanation:

## **Explanation**

*the word "ME" or a part of a word containing "ME".*

*score: 0.65*

[Suggest Better Explanation](#)

[Show scoring details](#)

*Figure 9.23: The explanation for the word ME*

We can suggest a better explanation and ask to see the scoring details, which will explain the neuron's scoring details by highlighting them using GPT-4, as shown in the following excerpt of the real activations:

## Top

### Real activations:

.txt FORMAT: the file is tab delimited with ID, MEAN-SENTIMENT-RATING, STANDARD DEVIATION, and RAW-SENTIMENT-RATINGS DESCRIPTION: Sentiment ratings from a minimum of 20 independent human raters (all pre

### Simulated activations:

.txt FORMAT: the file is tab delimited with ID, MEAN-SENTIMENT-RATING, STANDARD DEVIATION, and RAW-SENTIMENT-RATINGS DESCRIPTION: Sentiment ratings from a minimum of 20 independent human raters (all pre

### Real activations:

AFTER WE WERE MARRIED FOR 6 MONTHS - CHEATS ONE ME SIX MONTHS LATER, WANTS DIVORCE, AND IS GOING TO TRY FOR CHILD SUPPORT. JUST TO HURT ME AND TRY TO RUIN MY LIFE. add your own

Figure 9.24: Scoring details

We can gain insights from GPT-4 by clicking on **Activations**:

Quantile range [0.99, 0.999] sample

[show more](#)

" (7 2/3 cm) tall. My blanket's finished size was 22" x 34" (56cm x 86cm).

The blanket can be modified to be as big or as small as you want. I did the border and center portion in seed stitch, but garter or stockinette

Figure 9.25: GPT-4 insights

When GPT-4 generations are applied, the system chooses the top quantile range of the top activations.

At the bottom of the page, you can find the related tokens:

## **Related tokens**

### **Mean-activation-based**

ME mysql McA Me Me Courtney \*: Nap  
mys Meyer Manson ME me Mes events  
Mecca morphine Mog Listener me threads  
mA meg Sched mes Mek RM Scenes  
scene MySQL sessions uploads mes Meh

Figure 9.26: Related tokens

Finally, we can see the related neurons:

## **Related neurons**

### **DOWNSTREAM**

#### **Neuron 1:4233**

Connection strength: 0.26

*phrases involving asking for, providing, or mentioning help.*

*score: 0.06*

Figure 9.27: Related neurons

The interface is in its early stages. Nevertheless, explaining an LLM with GPT-4 is quite a step forward.

Exploring the neurons in this interface will help visualize a transformer's activity at the neuron level.



**Takeaway**

*The explanations are obtained by comparing the actual and simulated activations with GPT-4. This research has once again shattered the transformer models' black box.*

A long journey is ahead to improve the interface's clarity and the explanations' quality. But the first step has been made!

The evolution of this approach might solve a fair share of the issue of transformer interpretation.

We will next show why human control remains necessary.

## Limitations and human control

Interpreting transformer models has progressed, but much remains to be done. Some of the stumbling blocks remain quite challenging:

- The embedding sublayer is based on stochastic calculations added to the complex positional encoding.
- The stochastic nature of the mechanisms of the multiple attention heads makes it difficult to pinpoint why and how the score of a token prevailed after having gone through several layers.
- Softmax functions applied to raw outputs blur the tracks.
- Dropout sublayers erase some of the tracks.

- Deep learning regularizations such as ReLU and GELU hinder reverse engineering an output.

The numbers are numbing. A vintage model such as GPT-3 has 175 billion parameters and 9,216 attention heads (96 layers x 96 heads).

And these are only some of the difficulties!

Human evaluation remains a key resource to evaluate and find ways to help improve transparency. The increasing pressure to explain AI is pushing interpretation tools forward. You will find many good examples and poor results at the project level. Focus on the good examples to understand how a transformer makes its way through language learning.

Use the poor results to understand why a transformer made a mistake. In any case, get involved and stay in the loop of this ever-evolving field!

The visual interfaces explored in this chapter are fascinating. However, there is still much work to do. Human evaluation, control, intervention, and development will remain necessary for quite a long time!

## Summary

Transformer models are trained to resolve word-level polysemy disambiguation and low-level, mid-level, and high-level dependencies. The process is achieved by training million-to-trillion-parameter models. The task of interpreting these giant models seems daunting. However, several tools are emerging.

We first installed `Bertviz`. We learned how to interpret the computations of the attention heads with an interactive interface. We saw how words interacted with other words for each layer. We then introduced exBERT, another approach to visualizing BERT, among other models.

The chapter continued by defining SHAP and revealing the contribution of each word processed by Hugging Face transformers.

We then ran transformer visualization via dictionary learning with LIME. A user can choose a transformer factor to analyze and visualize the evolution of its representation from the lower layers to the higher layers of the transformer. The factor will progressively go from polysemy disambiguation to sentence context analysis and, finally, to long-term dependencies.

Other tools, such as LIT, plug a PCA projector and UMAP representations into the outputs of a BERT transformer model. We can then analyze clusters of outputs to see how they fit together. OpenAI's innovative approach showed how Large Language Models can explain the neurons in transformers.

The tools of this chapter will evolve along with other techniques. However, the key takeaway of this chapter is that transformer model activity can be visualized and interpreted in a user-friendly manner. In the next chapter, *Investigating the Role of Tokenizers in Shaping Transformer Models*, we will deepen our understanding of the inner workings of transformers. We will see why tokenizers are the cornerstone of transformer models.

## Questions

1. BertViz only shows the output of the last layer of the BERT model.  
(True/False)
2. BertViz shows the attention heads of each layer of a BERT model.  
(True/False)
3. BertViz shows how the tokens relate to each other. (True/False)
4. LIT shows the inner workings of attention heads like BertViz.  
(True/False)

5. Probing is a way for an algorithm to predict language representations.  
(True/False)
6. NER is a probing task. (True/False)
7. PCA and UMAP are non-probing tasks. (True/False)
8. LIME is model-agnostic. (True/False)
9. Transformers deepen the relationships of the tokens layer by layer.  
(True/False)
10. OpenAI Large Language Models (LLMs) can explain LLMs.  
(True/False)

## References

- BertViz: <https://github.com/jessevig/BertViz>
- Zeyu Yun, Yubei Chen, Bruno A. Olshausen, Yann LeCun, 2021, *Transformer visualization via dictionary learning: contextualized embedding as a linear superposition of transformer factors:* <https://arxiv.org/abs/2103.15949>
- Hugging Face with Slunberg SHAP:  
<https://github.com/slundberg/SHPTransformer>
- Visualization via dictionary learning:  
<https://transformervis.github.io/transformervis/>
- OpenAI, *Large Language Models can explain neurons in language models:* <https://openai.com/research/language-models-can-explain-neurons-in-language-models>
- OpenAI neuro explainer paper:  
<https://openaipublic.blob.core.windows.net/neuro-n-explainer/paper/index.html>
- LIT: <https://pair-code.github.io/lit/>

# Further reading

- Hoover *et al.*, 2021, *exBERT: A Visual Analysis Tool to Explore Learned Representations in Transformers Models*:  
<https://arxiv.org/abs/1910.05276>
- Jesse Vig, 2019, *A Multiscale Visualization of Attention in the Transformer Model*: <https://aclanthology.org/P19-3007.pdf>

## Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://www.packt.link/Transformers>



# 10

## Investigating the Role of Tokenizers in Shaping Transformer Models

When studying transformer models, we tend to focus on their architecture and the datasets provided to train them. This book covers the Original Transformer, BERT, RoBERTa, ChatGPT, GPT4, PaLM, LaMBDA, DALL-E, and more. In addition, the book reviews several benchmark tasks and datasets. We have fine-tuned a BERT-like model and trained a RoBERTa tokenizer, using tokenizers to encode data. In the previous *Chapter 9, Shattering the Black Box with Interpretable Tools*, we also opened the black box and analyzed the inner workings of a transformer model.

However, we did not explore the critical role tokenizers play and evaluate how they shape the models we build. AI is data-driven. *Raffel et al. (2019)*, like all the authors cited in this book, spent time preparing datasets for transformer models.

In this chapter, we will go through some of the issues of tokenizers that hinder or boost the performance of transformer models. Do not take tokenizers at face value. You might have a specific dictionary of words you use (advanced medical language, for example) with words poorly processed by a generic 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 some of the possible limits of tokenizers, using a Word2Vec tokenizer, to describe the problems we face with any tokenizing method. The limits will be illustrated with a Python program. We will continue our investigation by exploring word and subword tokenizers.

We will begin with sentence and word tokenizers. These tokenizers provide valuable natural language processing tools. However, they do not match the more efficient subword tokenizers for transformer model training. Therefore, we will continue with the more efficient subword tokenizers for transformer models.

Subword tokenizers show how a tokenizer can shape a transformer model's training and performance. We will see how to detect which subword tokenizer was applied to create a dictionary. Finally, we will build a function to display and control the token-ID mappings.

This chapter covers the following topics:

- Basic guidelines to control the output of tokenizers
- Raw data strategies and preprocessing data strategies
- Word2Vec tokenization problems and limits
- Creating a Python program to evaluate Word2Vec tokenizers
- Sentence and word tokenizers
- Subword tokenizers
- Tokenizer detection
- Displaying and controlling token-ID mappings



With all the innovations and library updates in this cutting-edge field, packages and models change regularly. Please go to the GitHub repository for the latest installation and code examples:

<https://github.com/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition/tree/main/Chapter10>.

You can also post a message in our Discord community (<https://discord.com/invite/Ep4XXhECdh>) if you have any trouble running the code in this or any chapter.

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

## Matching datasets and tokenizers

Downloading benchmark datasets to train transformers has many advantages. The data has been prepared, and every research lab uses the same references. Also, the performance of a transformer model can be compared to another model with the same data.

However, more needs to be done to improve the performance of transformers. Furthermore, implementing a transformer model in production requires careful planning and defining best practices.

In this section, we will define some best practices to avoid critical stumbling blocks.

Then, we will go through a few examples in Python, using cosine similarity to measure the limits of tokenization and encoding datasets.

Let's start with best practices.

## Best practices

*Raffel et al.* (2019) defined a standard text-to-text T5 transformer model. They also went further. They contributed to the destruction of the myth of using raw data without preprocessing it first.

Preprocessing data reduces training time. Common Crawl, for example, contains unlabeled text obtained through web extraction. Non-text and markup have been removed from the dataset.

However, the Google T5 team found that much of the text obtained through Common Crawl did not reach the level of natural language or English. So they decided that datasets need to be cleaned before using them.

We will take the recommendations that *Raffel et al.* (2019) made and apply corporate quality control best practices to the preprocessing and quality control phases. Among many other rules to apply, the examples described show the tremendous work required to obtain acceptable real-life project datasets.

Quality control is divided into the preprocessing phase (*Step 1*) when training a transformer and quality control when the transformer is in production (*Step 2*). A third phase (*Step 3*) has become mandatory for cutting-edge technology: *Continuous human quality control*.

Let's go through some of the main aspects of the preprocessing phase.

### Step 1: Preprocessing

*Raffel et al.* (2019) recommended preprocessing datasets before training models on them, and I have added some extra ideas.

Transformers have become language learners, and we have become their teachers. But to teach a machine student a language, we must explain what proper English is, for example.

We need to apply some standard heuristics to datasets before using them:

- **Sentences with punctuation marks:** The recommendation is to select sentences that end with punctuation marks, such as a period or a question mark.
- **Remove bad words:** Bad words should be removed. Lists can be found at the following site, for example:  
<https://github.com/LDNOOBW/List-of-Dirty-Naughty-Obscene-and-Otherwise-Bad-Words>.
- **Remove code:** This is tricky because sometimes, code is the content we are looking for. However, it is generally best to remove code from content for NLP tasks.
- **Language detection:** Sometimes, websites contain pages with the default “lorem ipsum” text. It is necessary to ensure that 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. I recommend building 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 Inference (NLI)** to filter sentences that make no sense.

- **Bad information references:** Eliminate text that refers to links that do not work, unethical websites, or people. This is a tough job but certainly worthwhile.

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

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

## Step 2: Quality control

A trained model will behave like a person who learned a language. It will understand what it can and learn from input data. Input data should go through the same process as *Step 1: Preprocessing* and add new information to the training dataset. The training dataset, in turn, can become the knowledge base of a corporate project. Users will be able to run NLP tasks on the dataset and obtain reliable answers to questions, useful summaries of specific documents, and more.

We should apply the best practices described in *Step 1: Preprocessing* to real-time input data. For example, a transformer can be running on input from a user or an NLP task, such as summarizing a list of documents.

Transformers are the most powerful NLP models ever. This means that our ethical responsibility is heightened as well.

Let's go through some of the best practices:

- **Check input text in real time:**

Do not accept bad information. Instead, parse the input in real time and filter the unacceptable data (see *Step 1*).

- **Real-time messages:**

Store the rejected data and the reason it was filtered so that users can consult the logs. Display real-time messages if a transformer is asked to answer an unfitting question.

- **Language conversions:**

You can convert rare vocabulary into standard vocabulary when it is possible. See *Case 4* of the *Word2Vec tokenization* section in this chapter. This is not always possible. When it is, it could represent a step forward.

- **Privacy checks:**

Whether you are streaming data into a transformer model or analyzing user input, private data must be excluded from the dataset and tasks unless authorized by the user or country the transformer is running in. It's a tricky topic. Consult a legal adviser when necessary.

We just went through some of the best practices. Let's now see why human quality control is mandatory.

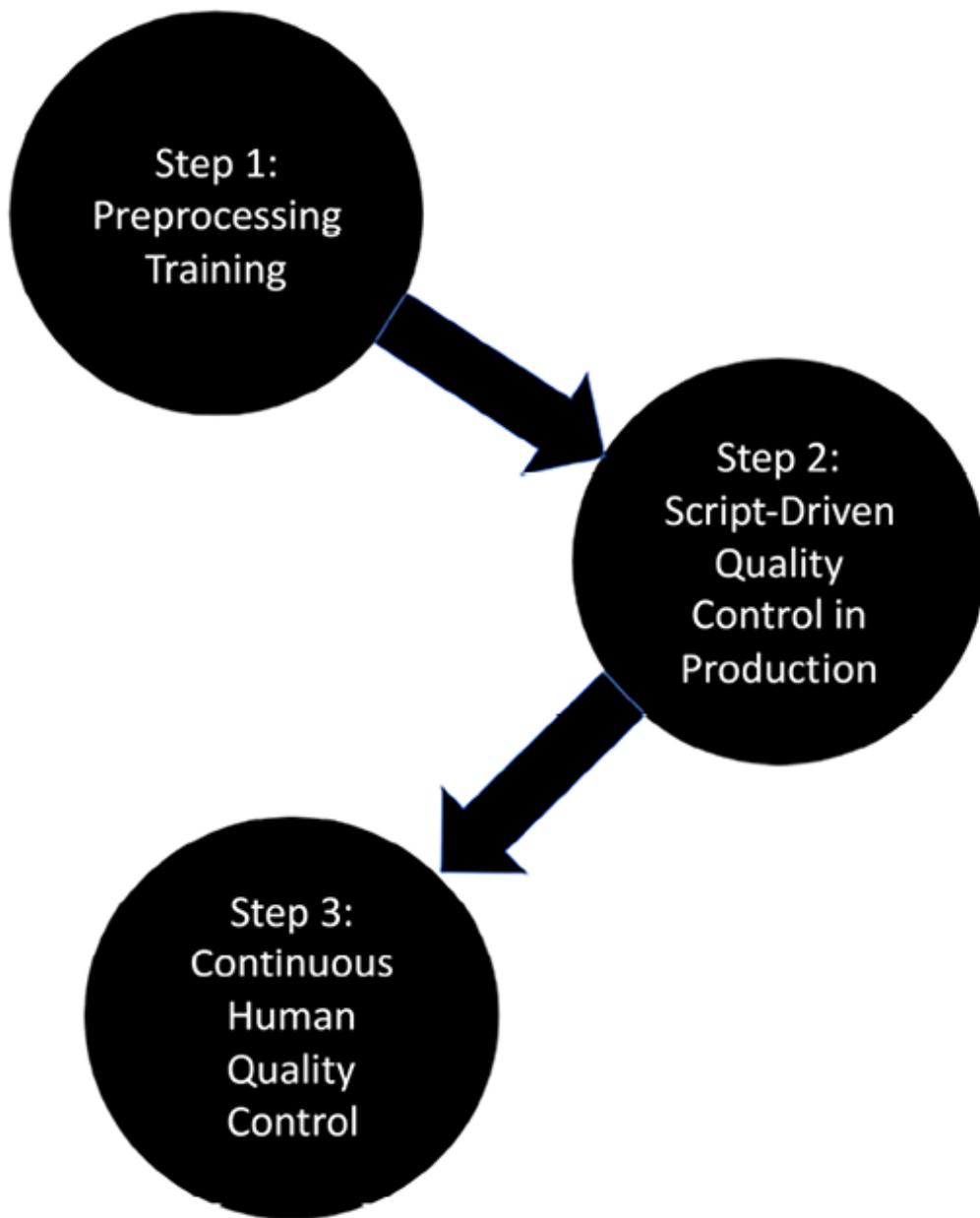
## **Step 3: Continuous human quality control**

Cutting-edge software quality control requires continual human intervention to monitor innovations and improve the systems in production. 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 content managers sometimes manually decide what is good or bad for their platform.

The right approach is to train a transformer, implement it, control the output, and feed the significant results back into the training set. Thus, the training set will continuously improve, and the transformer will continue to learn.

*Figure 10.1* shows how continuous quality control will help the transformer's training dataset grow and increase its performance in production:



*Figure 10.1: Continuous quality control*

Once the training is done and the model is in production with its quality control, continuous human quality control is required to measure the overall lifecycle of the system.

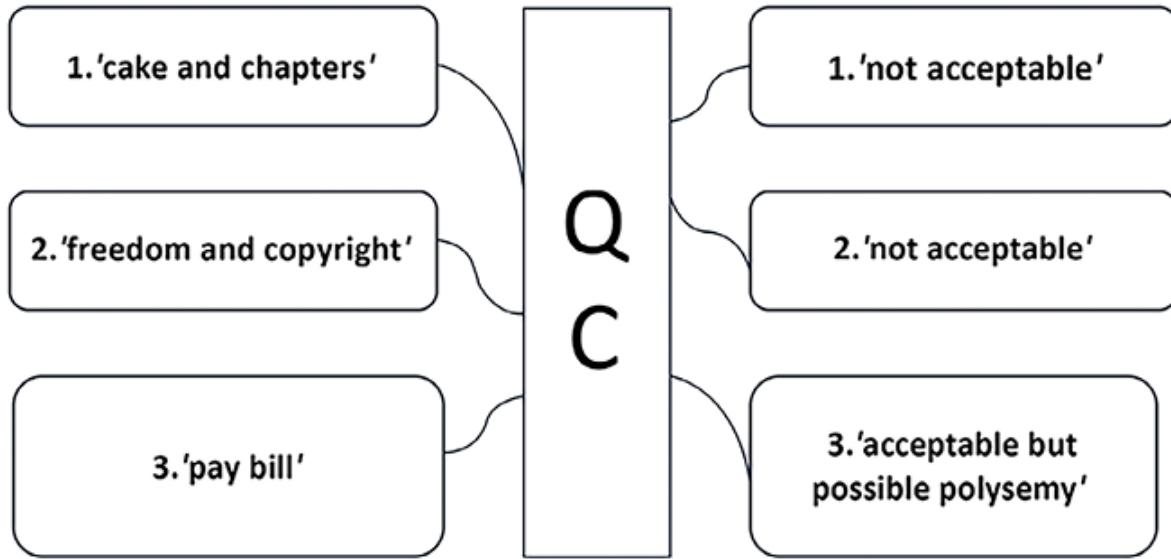
We have gone through several best practices that *Raffel et al.* (2019) described, and I have added some guidance based on my experience in corporate AI project management.

Let's go through a Python program with some examples of some of the limits encountered with tokenizers.

## Word2Vec tokenization

As long as things go well, nobody thinks about pretrained tokenizers. It's like in real life. We can drive a car for years without thinking about the engine. Then, one day, our car breaks down, and we try to find the reasons to explain the situation.

The same happens with pretrained tokenizers. Sometimes, the results are not what we expect. For example, some word pairs don't fit together in the context of the text of the Declaration of Independence, as we can see in *Figure 10.2*:



*Figure 10.2: Word pairs that tokenizers miscalculated and detected during Quality Control (QC)*

The examples shown in *Figure 10.2* 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 cosine similarity value.
- `freedom` refers to the freedom of speech, for example. `copyright` refers to the note written by the editor of the free ebook.
- `pay` and `bill` fit together in everyday English. `polysemy` is when a word can have several meanings. For example, `bill` means an amount to pay but also refers to the `Bill of Rights`. The result is acceptable, but it may be pure luck.

Before continuing, let's take a moment to clarify some points. **QC** refers to **quality control**. In any strategic corporate project, QC is mandatory. The quality of the output will determine the survival of a critical project. If the project is not strategic, errors will sometimes be acceptable. In a strategic

project, even a few errors could mean the intervention of a risk management audit to see if the project should be continued or abandoned.

From the perspectives of quality control and risk management, tokenizing irrelevant datasets (i.e., too many useless words or missing critical words) will confuse the embedding algorithms and produce “poor results.” That is why I use the word “tokenizing” loosely in this chapter, including some embedding because of the impact of one upon the other.

In a strategic AI project, “poor results” can be a single error with a dramatic consequence (especially in the medical sphere, airplane or rocket assembly, or other critical domains).

Open `Tokenizers.ipynb`, based on `positional_encoding.ipynb`, which we created in *Chapter 2, Getting Started with the Architecture of the Transformer Model*.

Results might vary from one run to another due to the stochastic nature of Word2Vec algorithms.

The prerequisites are installed and imported first:

```
!pip install 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')
```

We now load our sample dataset:

```
#1. Load text.txt using the Colab file manager  
#2. Downloading the file from GitHub  
!curl -L https://raw.githubusercontent.com/Denis2054/Transformers/master/texts/text.txt
```

`text.txt`, our dataset, contains the *American Declaration of Independence*, the *Bill of Rights*, the *Magna Carta*, the works of Immanuel Kant, and other texts.

We will now tokenize `text.txt` and train a Word2Vec model:

```
'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)  
print(model2)
```

`window = 5` is an interesting parameter. It limits the *distance* between the current word and the predicted word in an input sentence. `sg = 1` means a skip-gram training algorithm is used.

The output shows that the size of the vocabulary is 10816, the dimensionality of the embeddings is 512, and the learning rate was set to alpha=0.025:

```
Word2Vec<vocab=3291, vector_size=512, alpha=0.025>
```

We have a word representation model with embedding and can create a cosine similarity function named `similarity(word1,word2)`. We will send `word1` and `word2` to the function, which will return a cosine similarity value between them. The higher the value, the higher the similarity.

The function will first detect unknown words, `[unk]`, and display a message:

```
def similarity(word1,word2):
    cosine=False #default value
    try:
        a=model2[word1]
        cosine=True
    except KeyError:      #The KeyError exception is raised
        print(word1, ":[unk] key not found in dictionary")#False implied
    try:
        b=model2[word2]#a=True implied
    except KeyError:      #The KeyError exception is raised
        cosine=False #both a and b must be true
    print(word2, ":[unk] key not found in dictionary")
```

Cosine similarity will only be calculated if `cosine==True`, which means that both `word1` and `word2` are known:

```
if(cosine==True):
    b=model2[word2]
    # compute cosine similarity
```

```

dot = np.dot(a, b)
norma = np.linalg.norm(a)
normb = np.linalg.norm(b)
cos = dot / (norma * normb)
aa = a.reshape(1,512)
ba = b.reshape(1,512)
#print("Word1",aa)
#print("Word2",ba)
cos_lib = cosine_similarity(aa, ba)
#print(cos_lib, "word similarity")
if(cosine==False):cos_lib=0;
return cos_lib

```

The function will return `cos_lib`, the computed value of cosine similarity.

We will now go through six cases. We will name the dataset `text.txt`.

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:

```

word1="freedom";
word2="liberty"
print("Similarity",similarity(word1,word2),word1,word2)

```

The similarity is limited to `0.99` although the content was dense:

```
Similarity between freedom and liberty is [[0.99947506]]
```

The similarity algorithm is not an iterative deterministic calculation. It's a stochastic algorithm. This section's results might change with the dataset's content, the dataset's size after another run, or the module's versions. For example, if you run the cell 10 times, you may or may not obtain different values, such as in the following 10 runs.

In the following case, I obtained the same result 10 times with a Google Colab VM and a CPU:

```
Run 1: Similarity [[0.62018466]] freedom liberty
Run 2: Similarity [[0.62018466]] freedom liberty
...
Run 10: Similarity [[0.62018466]] freedom liberty
```

However, I did a “factory reset runtime” of the runtime menu in Google Colab. With a new VM and a CPU, I obtained:

```
Run 1: Similarity [[0.51549244]] freedom liberty
Run 2: Similarity [[0.51549244]] freedom liberty
...
Run 10: Similarity [[0.51549244]] freedom liberty
```

I performed another “factory reset runtime” of the runtime menu in Google Colab. I also activated the GPU. With a new VM and GPU, I obtained:

```
Run 1: Similarity [[0.58365834]] freedom liberty
Run 2: Similarity [[0.58365834]] freedom liberty
...
Run 10: Similarity [[0.58365834]] freedom liberty
```

The conclusion here is that stochastic algorithms are based on probabilities. Therefore, running a prediction  $n$  times if necessary is good practice.

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:

```
word1="corporations";word2="rights"
print("Similarity",similarity(word1,word2),word1,word2)
```

The dictionary does not contain the word `corporations`:

```
The word corporations does not exist in the dictionary
Similarity 0 corporations rights
```

Dead end! The word is an unknown `[unk]` token.

The missing word will provoke a chain of events and problems that distort the transformer model's output if the word is important. We will refer to the missing word as `unk`.

Several possibilities need to be checked and questions answered:

- `unk` was in the dataset but was not selected to be in the tokenized dictionary.
- `unk` was not in the dataset, which is the case for the word `corporations`. This explains why it's not in the dictionary in this case.

- `unk` will now appear in production if a user sends an input to the transformer that contains the token and it is not tokenized.
- `unk` was not an important word for the dataset but was 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` to be an 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 **Byte-Pair Encoding (BPE)**, as we have been doing throughout this book. If necessary, take a few minutes and return to *Step 3: Training a tokenizer* of *Chapter 6, Pretraining a Transformer from Scratch through RoBERTa*.

The nightmare begins if a team only uses byte-level BPE to fix the problem:

- `unk` will be broken down into word pieces. For example, we could end up with `corporations` becoming `corp` + `o` + `ra` + `tion` + `s`. One or several of these tokens have a high probability of being found in the dataset.
- `unk` will become a set of sub-words represented by tokens that exist in the dataset but do not convey the original token's meaning.

- The transformer will train well, and nobody will notice that `unk` was broken into pieces and trained meaninglessly.
- The transformer might produce excellent results and increase its performance from `0.8` to `0.9`.
- Everybody will applaud until a professional user applies an erroneous result in a critical situation. For example, in English, `corp` can mean `corporation` or `corporal`. This could create confusion and bad associations between `corp` and other words.

We can see that the standard of social media might be enough to use transformers for trivial topics. But in real-life corporate projects, producing a pretrained tokenizer that matches the datasets will take hard work. 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 three steps:

1. Train a tokenizer with a byte-level BPE algorithm.
2. Control the results with a program, such as the one we will create in the *Analyzing and controlling the quality of token-ID mappings* section of this chapter.
3. Train a tokenizer with a Word2Vec algorithm, which will only be used for quality control, and then parse the dataset, find the `unk` tokens, and store them in the database. Run queries to check if critical words are missing.

It might seem unnecessary to check the process in such detail, and you might be tempted to rely on a transformer's ability to make inferences with unseen words.

However, I recommend running several quality control methods in a strategic project with critical decision-making. For example, in a legal summary of a law, one word can make the difference between losing and winning a case in court. In an aerospace project (airplanes and 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`:

```
word1="etext";word2="declaration"
print("Similarity",similarity(word1,word2),word1,word2)
```

Furthermore, they both ended up in the tokenized dictionary:

```
Similarity [[0.9842512]] etext declaration
```

Even better, their cosine similarity seems to be sure about its prediction and exceeds `0.5`. The stochastic nature of the algorithm might produce different results on various runs.

At a trivial or social media level, everything looks good.

However, at a professional level, the result is disastrous!

`etext`, which is a word in the text file processed in the notebook, refers to *Project Gutenberg*'s preface to each ebook on their site, as explained in the *Matching datasets and tokenizers* section of this chapter. This means that the word `etext` is the editor's text file but has nothing to do with `declaration` (the *Declaration of Independence*). What is the goal of the transformer for a specific task:

- To understand an editor's preface?
- To understand the content of the book?

It depends on the usage of the transformer and might take a few days to sort out. For example, suppose an editor wants to understand prefaces automatically and uses a transformer to generate preface text. Should we take the content out?

`declaration` is a meaningful word related to the actual content of the *Declaration of Independence*.

`etext` is part of a preface that *Project Gutenberg* adds to its ebooks.

This might produce erroneous natural language inferences, such as *etext is a declaration* when the transformer is asked to generate text. *etext*, a word used by the editor of the file, has nothing to do with *declaration* in the text file we are processing. *declaration* is part of the Declaration of Independence. The Declaration of Independence dates back to 1776, and *etext* (electronic texts) date back to the 20<sup>th</sup> century. An NLP model that would speak about the Declaration of Independence, including electronic text vocabulary, would be making an error.

Let's now look into a missing word issue.

## Case 3: Words in a text but not in the dictionary

Sometimes, a word may be in a text but not in the dictionary. This will distort the results.

Let's take the words `pie` and `logic`:

```
word1="pie";word2="logic"  
print("Similarity",similarity(word1,word2),word1,word2)
```

The word `pie` is not in the dictionary:

```
The word pie does not exist in the dictionary  
Similarity 0 pie logic
```

We can assume that the word `pie` would be in a tokenized dictionary. But what if it isn't, or another word isn't? The word `pie` is not in the text file.

Therefore, we should have functions in the pipeline to detect words not in the dictionary and implement corrections or alternatives. Also, we should have functions in the pipeline to detect words in the datasets that may be important. For example, a project manager could run hundreds of documents through the tokenizer to detect unknown words that would be stored in a file to analyze them. This is only an example; each project requires specific quality control actions.

Let's now see the problem we face with rare words.

## Case 4: Rare words

Rare words produce devastating effects on the output of transformers for specific tasks that go beyond simple applications.

Managing rare words extends to many domains of natural language. For example:

- Rare words can occur in datasets but go unnoticed, or models are poorly trained to deal with them.
- Rare words can be medical, legal, engineering terms, or any other professional jargon.
- Rare words can be slang.
- There are hundreds of variations of the English language. For example, different English words are used in certain parts of the United States, the United Kingdom, Singapore, India, Australia, and many other countries.
- Rare words can come from texts written centuries ago that are forgotten or that only specialists use.

For example, in this case, we are using the word `eleutheromania` (intense longing for freedom):

```
word1="eleutheromania";word2="liberty"
print("Similarity",similarity(word1,word2),word1,word2)
```

The system doesn't recognize `eleutheromania`:

```
The word eleutheromania does not exist in the dictionary
Similarity 0 eleutheromania liberty
```

Unfortunately, if a rare word is used, the program will get confused, and we will obtain unexpected results after each run. If this occurs during a project, additional texts containing rare words that went unrecognized will have to be fed into the tokenizer training process.

For example, *if we implement a transformer model in a law firm to summarize documents or other tasks, we must be careful!*

Let's now see some methods we could use to solve a rare word problem.

## Case 5: Replacing rare words

*Replacing rare words represents a project in itself.* This work is reserved for specific tasks and projects. Suppose a corporate budget can cover the cost of having a knowledge base in aeronautics, for example. In that case, it is worth spending time querying the tokenized directory to find words it missed.

Problems can be grouped by topic solved, and the knowledge base will be updated regularly.

In *Case 4*, we stumbled on the word `eleutheromania`.

We could replace the word `eleutheromania` with `freedom`, which conveys the same meta-concept:

```
word1="freedom";word2="liberty"  
print("Similarity",similarity(word1,word2),word1,word2)
```

It produces an interesting result:

```
Similarity [[0.99956566]] freedom liberty
```

In any case, some rare words need to be replaced by more mainstream words. We could create queries with replacement words that we run until we find correlations that are over `0.9`, for example. Moreover, if we manage a critical legal project, we could have essential documents that contain rare words translated into standard English. Thus, the transformer's performance with NLP tasks would increase, and the knowledge base of the corporation would progressively increase.

Let's now see how well-pretrained tokenizers match with NLP tasks.

## **Exploring sentence and WordPiece tokenizers to understand the efficiency of subword tokenizers for transformers**

Transformer models commonly use BPE and WordPiece tokenization. In this section, we will understand why choosing a subword tokenizer over other tokenizers significantly impacts transformer models.

The goal of this section will thus be to first review some of the main word and sentence tokenizers.

We will continue and implement subword tokenizers. But first, we will detect if the tokenizer is a BPE or a WordPiece.

Then, we'll create a function to display the token-ID mappings.

Finally, we'll analyze and control the quality of token-ID mappings.

The first step is to review some of the main word and sentence tokenizers.

# Word and sentence tokenizers

Choosing a tokenizer depends on the objectives of the NLP project.

Although subword tokenizers are more efficient for transformer models, word and sentence tokenizers provide useful functionality. Sentence and word tokenizers are useful for many tasks, including the following:

- Text processing at a sentence or word level can be sufficient for NLP tasks such as text classification. Also, splitting texts into sentences and words can be part of a pipeline to create a supervised training dataset for transformer models. The sentences obtained, for example, can be normalized, and then the labeling function is added to the output. From there, a subword tokenizer can take over to begin the pretraining process of a transformer.
- Sentence- or word-level tokenization can be sufficient for simpler machine learning tasks such as classifiers for spam detection, for example.
- **Part-of-Speech (POS) tagging** and **Named Entity Recognition (NER)** can benefit from word or sentence tokenization when performed at the word level.



The choice of a tokenizer is thus a critical decision that will have a lasting impact on the NLP functions, machine learning, or transformer model to implement.

Open `Exploring_tokenizers.ipynb` in the GitHub repository of this chapter.

The first cells install the necessary libraries to run this notebook:

```
#Hugging Face Transformers
!pip install transformers
#Printing tabular data in Python
!pip install tabulate
#Natural Language Toolkit
!pip install nltk
```

We installed Hugging Face `transformers` for subword tokenizers, the `tabulate` library to print tabular data, and the Natural Language Toolkit, `nltk`, for the word and sentence tokenizers.



The titles of the sub-sections in this section are the same as in the notebook.

Let's use the Natural Language Toolkit to review some of the main word and sentence tokenizers:

```
import nltk
nltk.download('punkt')
from nltk.tokenize import sent_tokenize, word_tokenize, RegexpTokenizer, TreebankWordTokenizer, WhitespaceTokenizer, PunktSentenceTokenizer, WordPunctTokenizer, MWETokenizer
```

## Sentence tokenization

Sentence tokenization splits a text into separate sentences. It breaks a paragraph or document into units of sentences:

```
# Sentence Tokenization
text = "This is a sentence. This is another one."
sentences = sent_tokenize(text)
```

```
print("Sentence Tokenization:")
print(sentences)
print()
```

`sent_tokenize(text)` breaks the sequence into sentences with separators:

```
Sentence Tokenization:
['This is a sentence.', 'This is another one.']}
```

## Word tokenization

Word tokenization breaks a sequence (i.e., sentence and text) into individual words. It detects punctuation marks and white spaces, such as quotation marks, tabs, and newlines:

```
# Word Tokenization
sentence = "This sentence contains several words."
words = word_tokenize(sentence)
print("Word Tokenization:")
print(words)
print()
```

`word_tokenize(Sentence)` produces words with separators:

```
Word Tokenization:
['This', 'sentence', 'contains', 'several', 'words', '.']
```

## Regular expression tokenization

Regular expression tokenization uses regular expressions. Therefore, the function can be customized to define rules and patterns:

```
# Regular Expression Tokenization
tokenizer = RegexpTokenizer(r'\w+')
text = "Let's see how to tokenize a sentence."
tokens = tokenizer.tokenize(text)
print("Regular Expression Tokenization:")
print(tokens)
print()
```

`RegexpTokenizer(r'\w+')` interprets `r` as a raw character. Otherwise, the backslash would be interpreted as an escape sequence. Then, it interprets `\w` as a character class for any alphanumerical character. Finally, it interprets `+` as one or more. The output is a well-defined segmentation of the sentence:

```
Regular Expression Tokenization:
['Let', 's', 'see', 'how', 'to', 'tokenize', 'a', 'sentence']
```

## Treebank tokenization

The Treebank Tokenizer is based on the corpora of the University of Pennsylvania, which contains, among other annotations, POS, syntactic structures, and semantics roles:

```
# Treebank Tokenization
tokenizer = TreebankWordTokenizer()
text = "There aren't that many tokenizers."
tokens = tokenizer.tokenize(text)
print("Treebank Tokenization:")
print(tokens)
print()
```

The `TreebankWordTokenizer()` breaks a sequence down into words, taking into account advanced issues such as contractions:

```
Treebank Tokenization:  
['There', 'are', "n't", 'that', 'many', 'tokenizers', '.']
```

## White space tokenization

White space tokenization processes non-white spaces as tokens:

```
# White Space Tokenization tokenizer  
from nltk.tokenize import WhitespaceTokenizer  
tokenizer=WhitespaceTokenizer()  
text = "Tokenize this sequence of words using white space. The  
tokens = tokenizer.tokenize(text)  
print("White Space Tokenization:")  
print(tokens)  
print()
```

`WhitespaceTokenizer` has separated the words using non-white spaces as tokens:

```
White Space Tokenization:  
['Tokenize', 'this', 'sequence', 'of', 'words', 'using', 'white']
```

Note that, unlike the `TreebankWordTokenizer`, `WhitespaceTokenizer` did not process the contraction `aren't` because it doesn't contain a white space.

## Punkt tokenization

A Punkt tokenizer splits sequences into sentences after going through unsupervised pretraining. It will not require labels:

```
# Punkt Sentence Tokenization
tokenizer = PunktSentenceTokenizer()
text = "A tokenizer can be trained. Many tokenizers aren't trained."
sentences = tokenizer.tokenize(text)
print("Punkt Sentence Tokenization:")
print(sentences)
print()
```

PunktSentenceTokenizer() produces individual sentences:

```
Punkt Sentence Tokenization:
['A tokenizer can be trained.', 'Many tokenizers aren't trained.'
```

## Word punctuation tokenization

Word punctuation tokenization splits sequences into words based on white spaces and punctuation:

```
# Word Punctuation Tokenization
tokenizer = WordPunctTokenizer()
text = "They won a prize! They were overjoyed."
tokens = tokenizer.tokenize(text)
print("Word Punctuation Tokenization:")
print(tokens)
print()
```

The output of WordPunctTokenizer produces words with separators:

```
Word Punctuation Tokenization:
['They', 'won', 'a', 'prize', '!', 'They', 'were', 'overjoyed',
```

## Multi-word tokenization

Multi-word tokenization splits sequences into words but preserves multi-word expressions such as “cannot”:

```
# Multi-Word Expression Tokenization
tokenizer = MWETokenizer()
tokenizer.add_mwe(("can", "not"))
text = "I cannot go to the movies today"
tokens = tokenizer.tokenize(text.split())
print("Multi-Word Expression Tokenization:")
print(tokens)
print()
```

In this case, `MWETokenizer` preserved “cannot”:

```
Multi-Word Expression Tokenization:
['I', 'cannot', 'go', 'to', 'the', 'movies', 'today']
```

Sentence and word tokenizers are not to be taken lightly, only because recent large-scale language models prefer subword tokenizers. For example, in some cases, sentence and word tokenizers can be part of a preprocessing pipeline to create labeled datasets for transformer model training.

However, in some cases, subword tokenizers are sufficient to process raw data.

We will now explore subword tokenizers.

## Subword tokenizers

Transformer models are large-scale **Large Language Models (LLMs)**. The size of the models and the number of tasks they perform require highly

efficient tokenizers. Subword tokenizers for LLMs are the best choice for many reasons, including:

- **Out-of-Vocabulary (OOV) words:** A subword tokenizer can capture words that were not part of the training phase (OOV). The tokenizer will break OOV into small units that a transformer model can process.
- **Vocabulary optimization:** Subword tokenizers break sequences down into smaller units than sentence and word tokenizers. The size of the vocabulary is optimized.
- **Morphological flexibility:** Subword tokenizers break words down into smaller units. The units obtained open the door to generalizations with other small units, deepening a model's ability to understand language.
- **Noise resistance:** Even if a word is misspelled or contains a typo, a subword tokenizer can still capture and process its meaning.
- **Multiple languages:** Word-level tokenizers are related to a language. Subword tokenizers aren't.

BPE and WordPiece are commonly used for transformer models.

Understanding the principles of these two subword tokenizers will help you understand how any subword tokenizer works. Although we are focusing on BPE and WordPiece, they are not the only subword tokenizers.

Depending on your project, other subword tokenizers might be more efficient. So before going through BPE and WordPiece, let's go through two main ones to see how text sequences are broken down into subwords.

Open `Sub_word_tokenizers.ipynb`.

First, we install Hugging Face libraries and a Python wrapper for SentencePiece:

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

Now, we will go through two subword libraries that are not BPE or WordPiece.

## Unigram language model tokenization

Unigram language model tokenization was developed by Google. It trains using subword units. It drops the infrequent units. Unigram language model tokenization is non-deterministic. As such, it will not always produce the same tokens for the same input. Conversely, BPE is deterministic and will produce the same output tokens for an input.

In the following example, we run a small training session on a sample of sentences and then tokenize a sentence.

We first import the necessary modules:

```
from tokenizers import Tokenizer  
from tokenizers.models import Unigram  
from tokenizers.trainers import UnigramTrainer  
from tokenizers.pre_tokenizers import Whitespace
```

Then, we define a sample corpus:

```
# Define a sample corpus  
corpus = [  
    "Subword tokenizers break text sequences into subwords.",  
    "This sentence is another part of the corpus.",  
    "Tokenization is the process of breaking text down into sm  
    "These smaller units can be words, subwords, or even indiv.
```

"Transformer models often use subword tokenization."

1

We train the tokenizer using a white space pre-tokenizer:

```
# Instantiate a Unigram tokenizer model
tokenizer = Tokenizer(Unigram([]))
# Add a pre-tokenizer
tokenizer.pre_tokenizer = Whitespace()
# Train the tokenizer model
trainer = UnigramTrainer(vocab_size=5000) # Here you set the vocabulary size
tokenizer.train_from_iterator(corpus, trainer)
```

Finally, we tokenize the target sentence:

```
# Now let's tokenize the original sentence
output = tokenizer.encode("Subword tokenizers break text sequence")
print(output.tokens)
```

As you can see, the output is quite different from the word and sentence tokenizers. The output is not a set of words or sentences but subwords:

# SentencePiece

A SentencePiece tokenizer adds a BPE approach to the Unigram language model tokenizer. It doesn't require a pre-tokenizer and can process raw data.

In the following example, we train a small sample with no pre-tokenizer:

First, we import the necessary modules:

```
import sentencepiece as spm
import random
```

Then, we create a small sample corpus:

```
# Define a basic corpus
basic_corpus = [
    "Subword tokenizers break text sequences into subwords.",
    "This sentence is another part of the corpus.",
    "Tokenization is the process of breaking text down into smal
    "These smaller units can be words, subwords, or even individ
    "Transformer models often use subword tokenization."
]
```

We perform some data augmentation randomly:

```
# Generate a larger corpus by repeating sentences from the basic corpus
corpus = [random.choice(basic_corpus) for _ in range(10000)]
```

We save the corpus:

```
# Write the corpus to a text file
with open('large_corpus.txt', 'w') as f:
    for Sentence in corpus:
        f.write(sentence + '\n')
```

We train the tokenizer:

```
# Train the SentencePiece model  
spm.SentencePieceTrainer.train(input='large_corpus.txt', model_
```

We load the trained model:

```
# Load the trained model  
sp = spm.SentencePieceProcessor()  
sp.load('m.model')
```

We tokenize our target sequence:

```
# Tokenize the original sentence  
tokens = sp.encode_as_pieces("Subword tokenizers break text se  
print(tokens)
```

Once again, we obtain subwords that are unlike the output of sentence and word tokenizers:

```
['_', 'S', 'ubword', '_tokeniz', 'ers', '_break', '_', 'te', 'x
```

We have gone through two main subword tokenizers, not BPE or WordPiece. However, you can see that, unlike sentence and word tokenizers, we obtain subwords through different approaches. There are also other subword tokenizers, each with its specific approach.

The choice of tokenizer remains a decision for each type of project.

In the *Step 3: Training a tokenizer* section in *Chapter 6, Pretraining a Transformer from Scratch through RoBERTa*, we implemented a BPE tokenizer. In this section, we will focus on WordPiece tokenization.

Before creating subword tokenization in code with WordPiece, let's sum up BPE training.

## Byte-Pair Encoding (BPE)

BPE begins with a vocabulary of individual characters. It then merges the most frequent pair of consecutive characters. A hyperparameter determines the number of times the process is repeated.

The result is a set of merged characters that can be an individual character, subwords, or words. The result is stored in a merge file, in `merge.txt` (for example, as in *Chapter 6*). A second file is generated, `vocab.json` (for instance, as in *Chapter 6*), that contains the dictionary of subwords with their unique ID, an integer.

The sequence is broken down into subwords, found in the merge dictionary during the tokenization process. The process is iterative. The tokenizer will try to find larger subwords n times. When an optimum result is reached, the iteratively merged subwords and the tokenizers assign an ID to the subword, with the information about the subword IDs contained in the `vocab.json`.

You might want to go through the code of *Chapter 6* again. You can select a corpus of your choice, train the tokenizer, train the model, and measure the impact of the tokenization process on the output of a transformer model.

## WordPiece

WordPiece, like BPE, begins with a vocabulary of individual characters. This ensures that any word can be tokenized. Then, the training process builds subwords. It uses an optimization process to minimize the number of subwords.

When the training process is completed, the tokenizer will break sequences down into the longest sequence words in its vocabulary. Subwords not at the beginning of an original word contain a prefix `##`. For example, “undo” will be represented as `["un", " ##do"]`. Keep this in mind because it will help us identify WordPiece tokenizers.

Tokenizers will have a strong impact on the training of a transformer model. Choosing the right tokenizer will often determine its outcome from the start.

Now, let’s explore the output of a WordPiece tokenizer.

## Exploring in code

To get started, go back to `Exploring tokenizers.ipynb` in the GitHub repository of this chapter. Then, scroll down to the *Subword tokenizers* section.

Let’s begin by detecting whether the tokenizer is a WordPiece tokenizer.

## Detecting the type of tokenizer

In this example, our first step will be to detect whether our tokenizer is a WordPiece tokenizer. To achieve this, we will load the `merge.txt` and `vocab.json` files generated in *Chapter 6*:

```
#1. Load merges.txt using the Colab file manager  
#2. Downloading the file from GitHub  
!curl -L https://raw.githubusercontent.com/Denis2054/Transform
```

```
#1. Load vocab.json using the Colab file manager  
#2. Downloading the file from GitHub  
!curl -L https://raw.githubusercontent.com/Denis2054/Transformers/master/vocab.json -o vocab.json
```

Now, let's import a Hugging Face `RobertaTokenizer`:

```
from transformers import RobertaTokenizer  
tokenizer = RobertaTokenizer.from_pretrained("/content", max_length=128)  
# Get the vocabulary  
vocab = tokenizer.get_vocab()  
# Check if WordPiece or BPE  
is_wordpiece = any(token.startswith('##') for token in vocab)  
# Print the tokenizer type  
if is_wordpiece:  
    print("Tokenizer type: WordPiece")  
else:  
    print("Tokenizer type: BPE")
```

The output shows that it's a BPE tokenizer. This example is simplified. We suppose the tokenizer can only be a BPE or a WordPiece tokenizer. Also, we are sure it's a BPE because we trained it in *Chapter 6*:

```
Tokenizer type: BPE
```

Now, we'll load a tokenizer trained on Hugging Face's platform:

```
from transformers import BertTokenizer  
# Load the BERT tokenizer  
model_name = 'bert-base-uncased'  
tokenizer = BertTokenizer.from_pretrained(model_name)  
# Get the vocabulary  
vocab = tokenizer.get_vocab()
```

We rerun the detection process:

```
# Check if WordPiece or another type of tokenization was used
is_wordpiece = any(token.startswith('##') for token in vocab)
# Print the tokenizer type
if is_wordpiece:
    print("Tokenizer type: WordPiece")
else:
    print("Tokenizer type: BPE")
```

This time, we can see it's a WordPiece tokenizer because the `##` prefix was detected:

```
Tokenizer type: Wordpiece
```

To make sure it's a WordPiece, let's print the vocabulary:

```
# Print the vocabulary
for token, id in vocab.items():
    print(f'{token}: {id}')
```

The output displays the `##` prefix, which signals that the subword is not the beginning of a word, as shown in the following excerpt:

```
...
bien: 29316
eels: 29317
marek: 29318
##ayton: 29319
##cence: 29320
...
```

Now, let's display the data using the tabulate library.

## Displaying token-ID mappings

This section looks at the token-ID mappings of the `WordPiece` vocabulary.

We import the tabulate modules and widgets to create an interactive interface, display modules, and the tokenizer we just detected:

```
from tabulate import tabulate
import ipywidgets as widgets
from IPython.display import display
from transformers import BertTokenizer
```

We load the model and tokenizer:

```
# Load the BERT tokenizer
model_name = 'bert-base-uncased'
tokenizer = BertTokenizer.from_pretrained(model_name)
```

We retrieve the vocabulary:

```
# Get the vocabulary
vocab = tokenizer.get_vocab()
```

We create a list with items in the vocabulary:

```
# Convert the vocabulary to a list of tuples
vocab_list = list(vocab.items())
```

We sort the vocabulary and create a filter with a widget for our interactive interface:

```
# Sort the vocabulary by token
sorted_vocab = sorted(vocab_list, key=lambda x: x[0])
# Create a text input widget for filtering
filter_widget = widgets.Text(placeholder='Filter vocabulary')
```

We create a function to filter and display the vocabulary:

```
# Function to filter and display the vocabulary
def filter_vocabulary(filter_text):
    filtered_vocab = [word for word in sorted_vocab if word[0] == filter_text]
    table = tabulate(filtered_vocab, headers=['Token', 'ID'])
    display(widgets.HTML(table))
```

Finally, we call and display the filter:

```
# Call the filter function when the widget value changes
filter_widget.observe(lambda event: filter_vocabulary(event.new))
# Display the filter widget
display(filter_widget)
```

A pretty filter interface is displayed:



Filter vocabulary

Figure 10.3: The filter of our interface

Type a character, and the words beginning with that character will appear. `t`, for example, will generate an output with all the token IDs beginning with that letter, as shown in the following excerpt:

```
...tomorrow 4826 ton 10228 tone 430...
```

To run another filter, rerun the cell. Again, take the time to observe the token IDs to see the characters, prefixes (`#`), partial words, and words present in the vocabulary. It will give you a good mental representation of how tokenizers establish their tokenization process.

## Analyzing and controlling the quality of token-ID mappings

In the previous section, we went through the token-ID mappings of the vocabulary of a WordPiece tokenizer.

In this section, we will do the opposite: we will take a word and break it down into its tokenized token-ID mappings. We will go from words to tokens.

We first load the transformer model:

```
# Load the BERT tokenizer
model_name = 'bert-base-uncased'
tokenizer = BertTokenizer.from_pretrained(model_name)
```

We define the function to tokenize a word and display the output:

```
# Function to tokenize a word and provide information
def tokenize_word(word):
```

```
# Tokenize the word
tokens = tokenizer.tokenize(word)
# Check if the word was found directly or is the result of
if len(tokens) == 1 and tokens[0] == word:
    process = "Direct"
else:
    process = "Subword"
# Display the word and process information
print("Word:", word)
print("Tokenized Tokens:", tokens)
print("Tokenization Process:", process)
```

We now create a widget to enter a word:

```
# Create a widget for entering the word
word_input = widgets.Text(description='Enter a Word:')
display(word_input)
```

We create an event handler to tokenize the word:

```
# Create an event handler for the widget
def on_button_click(b):
    word = word_input.value
    tokenize_word(word)
```

Finally, we create a button to trigger the tokenization function, which also displays the output:

```
# Create a button widget for triggering the tokenization procedure
button = widgets.Button(description="Tokenize")
button.on_click(on_button_click)
display(button)
```

If we type a random sequence such as `dsdf`, we can see that `ds` exists in the vocabulary and that `##df` is not the beginning of a word. `dsdf` is a subword:

```
Word: dsdf
Tokenized Tokens: ['ds', '##df']
Tokenization Process: Subword
```

If we type `word`, we can see that the `word` is present in the vocabulary:

```
Word: word
Tokenized Tokens: ['word']
Tokenization Process: Direct
```

Now, let's enter a more difficult word such as `amoeboid`:

```
Word: amoeboid
Tokenized Tokens: ['am', '##oe', '##bo', '##id']
Tokenization Process: Subword
```

The word `amoeboid` is broken down into subwords with multiple prefixes.

The `am` token in `amoeboid` brings polysemy (i.e., several meanings for the same sequence) into the problem at a low level. `am` can be a prefix, the word `am` as in `I + am`, or a sub-word such as in `am + bush`. Attention layers could associate the `am` of one token with another `am`, creating relationships that do not exist. Polysemy on complex words is a challenging problem.

We can see that even subword tokenizers can face problems with rare and complex words. These words might not be encountered often in training datasets, leading them to be underestimated. However, such words might be critical in medical or legal papers.

As a cornerstone of transformer models, tokenization will profoundly impact their accuracy and performance.

## Summary

In this chapter, we measured the impact of tokenization on the subsequent layers of a transformer model. A transformer model can only attend to tokens from a stack's embedding and positional encoding sub-layers. It does not matter if the model is an encoder-decoder, encoder-only, or decoder-only model. Furthermore, whether the dataset seems good enough to train does not matter.

If the tokenization process fails, even partly, our transformer model will miss critical tokens.

First, we saw that raw datasets might be enough for standard language tasks 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 looked for ideas, among many, to work around the limits of standard tokenizers. We applied a language-checking method to adapt the text we wish to process, such as how a tokenizer *thinks* and encodes data.

We then explored sentence and word tokenizers to understand how text sequences can be broken down into sentences and words. We reviewed

several sentence and word tokenizers, including sentence tokenization and regular expression tokenization.

Sentence and word tokenizers are useful for many NLP tasks, including, in some cases, preprocessing datasets for transformer model training. However, on large-scale corpora, they will generate large dictionaries that will slow down the training process of a transformer.

Therefore, we explored subword tokenizers such as Unigram language model tokenization, SentencePiece, BPE, and WordPiece. We focused on WordPiece tokenizers to explore token-ID mappings in detail.

The skill of tokenization requires balancing preserving information and optimizing computational performances. You must choose a method that fits your task and model.

Once the tokenizing phase of a project has been completed, embedding is the next logical step.

The next chapter, *Leveraging LLM Embeddings as an Alternative to Fine-Tuning*, will take us into embeddings and transfer learning.

## Questions

1. A tokenized dictionary contains every word that exists in a language.  
(True/False)
2. Pretrained tokenizers can encode any dataset. (True/False)
3. It is good practice to check a database before using it. (True/False)
4. It is good practice to eliminate obscene data from datasets. (True/False)
5. It is good practice to delete data containing discriminating assertions.  
(True/False)

6. Raw datasets might sometimes produce relationships between noisy content and useful content. (True/False)
7. A standard pretrained tokenizer contains the English vocabulary of the past 700 years. (True/False)
8. Old English can create problems when encoding data with a tokenizer trained in modern English. (True/False)
9. Medical and other types of jargon (domain-specific language) 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 et al., 2019, *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*:  
<https://arxiv.org/pdf/1910.10683.pdf>
- Gensim: <https://radimrehurek.com/gensim/intro.html>

## Further reading

- Hiraoka et al., 2023, *Tokenization Tractability for Human and Machine Learning Model: An Annotation Study*:  
<https://arxiv.org/abs/2304.10813>

**Join our community on Discord**

Join our community's Discord space for discussions with the authors and other readers:

<https://www.packt.link/Transformers>



# 11

## Leveraging LLM Embeddings as an Alternative to Fine-Tuning

Do not overlook embeddings as an alternative to fine-tuning a large language transformer model. Fine-tuning requires a reliable dataset, the right model configuration, and hardware resources. Creating high-quality datasets takes time and resources.

Leveraging the embedding abilities of a **Large Language Model (LLM)** such as OpenAI's Ada will enable you to customize your model with reduced cost and effort. Your model will be able to access updated data in real time. You will be implementing **Retrieval Augmented Generation (RAG)** through embedded texts. We used web pages and customized text for RAG in *Chapter 7, The Generative AI Revolution with ChatGPT*. This time, we will go further and use embeddings.

This chapter begins by explaining why searching with embeddings can sometimes be a very effective alternative to fine-tuning. We will go through the advantages and limits of this approach.

Then, we will go through the fundamentals of text embeddings. We will build a program that reads a file, tokenizes it, and embeds it with Gensim and Word2Vec. The program will show how to analyze the model description and access the vector of a word. We will explore Genism's vector

space and display the cosine similarity between words. Finally, we will display the vector space in Google's TensorFlow Projector.

With these fundamentals in mind, we will implement a question-answering program on sports events that occurred after the cutoff date of the dataset. For example, if a model was pretrained and fine-tuned a year before you accessed it, how would it answer questions regarding events a year later? We address this issue with embeddings-based search functionality.

Finally, we will implement OpenAI Ada to embed Amazon Fine Food Reviews. We will prepare the data and run Ada embedding. The program will find clusters with k-means clustering. We will display clusters with t-SNE. The system will be ready to ask OpenAI davinci to find and describe the theme of each cluster of reviews.

By the end of the chapter, you will have learned how to take a system from prompt design to advanced prompt engineering using embeddings for RAG.

This chapter covers the following topics:

- Tokenizing a text with Punkt
- Embedding a text with Gensim and Word2Vec
- Accessing the vector of a word
- Exploring the embedded vector space
- Cosine similarity
- Displaying the vector space with the TensorFlow Projector
- Question answering
- Preparing search data
- Searching with embeddings
- Transfer learning
- Running Ada embeddings and using RAG

- Clustering the data with the learned embeddings



With all the innovations and library updates in this cutting-edge field, packages and models change regularly. Please go to the GitHub repository for the latest installation and code examples:

<https://github.com/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition/tree/main/Chapter11>.

You can also post a message in our Discord community (<https://www.packt.link/Transformers>) if you have any trouble running the code in this or any chapter.

Let's examine why LLM embeddings can be an alternative to fine-tuning.

## LLM embeddings as an alternative to fine-tuning

ChatGPT models are impressive. They have taken everyone by surprise. However, ChatGPT has a memory problem! It only remembers what it learned from its training data. For example, in January 2024, ChatGPT's cutoff date was April 2023. It cannot answer questions about events after April 2023. OpenAI has found a workaround for some issues using the BING search engine, but this isn't enough.

Also, ChatGPT only knows what the training set contains. For example, maybe you have information that hasn't been made public and that ChatGPT cannot find.

In this chapter, we will build two methods:

- An ask method using **Retrieval Augmented Generation (RAG)** by adding information to the prompt
- A RAG search and ask function that leverages the Ada embedding model

In both cases, these approaches take us from prompt design to advanced prompt engineering.

## From prompt design to prompt engineering

Prompt design and prompt engineering are two terms that seem similar but will take you to the next level.

Prompt design simply requires an end user to type a question or an instruction and submit it to a chat model.

Prompt engineering involves building complex messages to steer the chat model. For example, sometimes ChatGPT cannot answer a question because its training data doesn't contain the necessary information. Maybe the dataset was built before an event occurred, for example. Or, the information is not in the dataset from the start. In this case, we must go from prompt design to prompt engineering, which, in the case of this chapter, involves RAG.

In this chapter, you will explore how to go from a simple end-user prompt to a fully engineered embedding-based prompt.

We will begin by reviewing the fundamentals of text embedding.

# Fundamentals of text embedding with NLKT and Gensim

In this section, we will go through the fundamentals of text embedding: tokenizing a book, embedding the tokens, and exploring the vector space we created.

Open `Embedding_with_NLKT_Gensim.ipynb` in the chapter directory of the GitHub repository.

We will first install the libraries we will need.

## Installing libraries

The program first installs the **Natural Language Toolkit (NLTK)**:

```
!pip install --upgrade nltk -qq  
import nltk
```

The NLTK will take us down to the token level as in *Chapter 10, Investigating the Role of Tokenizers in Shaping Transformer Models*.

We'll use the `punkt` sentence tokenizer:

```
nltk.download('punkt')
```

The program installs `gensim` for the similarity tools:

```
!pip install gensim -qq  
import gensim  
print(gensim.__version__)
```

The output is the version:

```
4.3.2
```

The first step is to read the file.

## 1. Reading the text file

The program downloads a file containing content written by Descartes, the French philosopher, scientist, and mathematician:

```
#1. Load Descartes.txt using the Colab file manager  
#2. Downloading the file from GitHub  
!curl -H -L https://raw.githubusercontent.com/Denis2054/Transf...
```

Then we read the file and replace the newline control character `\n` to obtain a continuous string:

```
with open('Descartes.txt', 'r', encoding='utf-8') as file:  
    descartes_book = file.read().replace('\n', '')
```

We can now tokenize the text.

## 2. Tokenizing the text with Punkt

Tokenizing the text takes us back to the lowest level of information we will process. It is essential to fully acquire a bottom-up view of the process leading to embedding.

In *Chapter 6, Pretraining a Transformer from Scratch through RoBERTa*, section *Step 3: Training a tokenizer*, we trained a byte-level tokenizer that broke words down into minimal components regardless of their base form, prefixes, and suffixes.

In this section, we will use a word tokenizer. A byte-level tokenizer operates at the level of individual bytes, while the `word_tokenize` function from NLTK tokenizes the text into individual words. Both approaches serve different purposes and operate at different levels of linguistic granularity.

We first import a word tokenizer, tokenize the book we prepared, and count the tokens:

```
from nltk.tokenize import word_tokenize
tokens = word_tokenize(descartes_book)
print(len(tokens))
```

The output shows the initial number of tokens in the vocabulary:

```
23605
```

We will now preprocess the tokens to extract the meaningful tokens.

## Preprocessing the tokens

The goal of this section is to preprocess the tokens with three methods:

### 1. Convert the tokens to lowercase

Converting the tokens to lowercase will reduce the number of tokens in the vocabulary. For example, “Talking” and “talking” are duplicates in this

strategy. Converting “Talking” to “talking” eliminates the duplicates.

## 2. Lemmatization

Lemmatization will reduce words to their base lemma (base form) to retain semantic meaning. For example, “running” will become “run.” In this approach, the suffix “ing” will not be part of the vocabulary in this section.

In this example, we did not apply a stemming method. Stemming is rule-based and removes prefixes and suffixes. The stem obtained might not be a valid word. For example, “happiness” might produce “happi” when removing “ness.” In this case, we want to retain the semantic meaning of all the main words.

## 3. Stop words

Stop words are common words that will be filtered out, such as “and” and “the.” The choice of filtering stop words or not depends on each project.

To implement these three functions, we first download WordNet and also `stopwords`, which contains the list of English stopwords:

```
nltk.download('wordnet')
nltk.download('stopwords')
```

WordNet is a lexical database of English words that can be applied for lemmatization. We now import `stopwords`, `WordNetLemmatizer`, and the `string` module to process the tokens:

```
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
import string
```

The program now applies the preprocessing functions we need:

```
stop_words = set(stopwords.words('english'))
lemmatizer = WordNetLemmatizer()
tokens = [lemmatizer.lemmatize(token.lower()) for token in tokens]
```

The program converted the tokens to lowercase, performed the lemmatization tasks, and filtered the stop words.

Let's find the number of tokens in our processed vocabulary:

```
print(len(tokens))
```

The number of tokens has been reduced from 23,605 raw tokens to 9,781 meaningful tokens:

```
9781
```

The tokens obtained are words because we used a word tokenizer, not a byte-level tokenizer that would have broken the words into minimal sub-pieces.

The tokenization process filtered the information that wasn't meaningful for our tasks. However, duplicates may remain in the vocabulary.

We can find, count, and display the unique tokens obtained:

```
unique_tokens = set(tokens)
print(len(unique_tokens))
```

First, let's count them:

```
print(unique_tokens)
```

The output filters the duplicates and produces a unique set of meaningful tokens:

```
3843
```

The program now displays the vocabulary of processed tokens:

```
print(unique_tokens)
```

The list of words in the vocabulary is displayed as shown in the following excerpt:

```
{'ofpoetry.i', 'hamper', 'warmerimmediately', 'convinced', 'agr
```

The token processing function produced an acceptable output, though some inaccuracies remain. We could continue the preprocessing until the vocabulary is 100% perfect, but we have enough information for the example in this section.

We will use the `unique_tokens` vocabulary for the embedding process:

```
tokens=unique_tokens  
#print(len(tokens))
```

We can now embed the vocabulary with Gensim and Word2Vec.

### 3. Embedding with Gensim and Word2Vec

Gensim's Word2Vec module embeddings will create a multidimensional space with hundreds of dimensions.

The following code imports the `word2vec` module, embeds the tokens, and saves the model for future use:

```
from gensim.models import Word2Vec
# Train a Word2Vec model
model = Word2Vec([tokens], compute_loss=True, vector_size=300, mi
# Save the model for later use
model.save("descartes_word2vec.model")
```

The trained and saved Gensim Word2Vec model has several key parts, including:

- **Model parameters:** These are the settings you decide on when you make the model, like the size of the word vectors and the window size.
- **Vocabulary:** This is a list of all the unique words the model has learned from. Each word is related to a specific index in the model's embedding matrix.
- **Word vectors (embeddings):** These are the actual word vectors the model learns during training, stored in a matrix in which each row represents a word in the vocabulary.

- **Training loss:** The loss function measures the accuracy of the training process.

Remember, the saved model doesn't include the original training data (the text you used to train it). It only saves what it learned in the data (word vectors), not the data itself.

We can look into the model description with a widget.

## 4. Model description

We can visualize the model's description by accessing its attributes. In this section, we will focus on:

- `wv`: The object containing the word vectors.
- `vector_size`: The number of dimensions of the word vectors.
- `train_count`: The number of times `train()` has been called.
- `total_train_time`: The total cumulative training time in seconds.
- `epochs`: The number of training epochs.
- `sg`: Training algorithm: 1 for skip-gram; 0 for CBOW

We can create a widget that displays the attributes we chose to explore:

```
from IPython.display import display
import ipywidgets as widgets
# Load the model
model = Word2Vec.load("descartes_word2vec.model")
# Widget for the model attribute
attr_widget = widgets.Dropdown(
    options=['wv', 'vector_size', 'train_count', 'total_train_',
    value='wv',
    description='Attribute:',
)
display(attr_widget)
# Widget for the number of lines
```

```

num_lines_widget = widgets.IntSlider(min=0, max=100, step=1, value=1)
display(num_lines_widget)
# Button to display the data
display_button = widgets.Button(description='Display')
display(display_button)
# Function to display the data
def display_data(button):
    attr = attr_widget.value
    num_lines = num_lines_widget.value
    if attr == 'wv':
        words = list(model.wv.index_to_key)
        for word in words[:num_lines]:
            print(word, model.wv[word])
    else:
        print(getattr(model, attr))
# Link the function to the button
display_button.on_click(display_data)

```

The attributes are displayed in a drop-down list. You can select one and display it. For example, we can obtain the vector size of the dimensions representing a word:

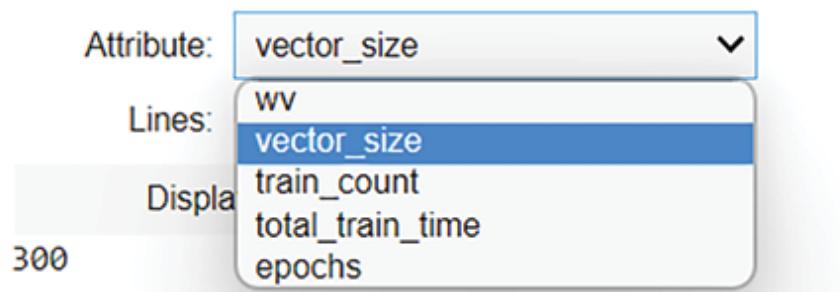


Figure 11.1: Selecting the vector size

We see the tokens (words in this case) are represented with a size 300 vector. You can view these vectors by clicking on **wv**. In this case, select the number of lines you want to display. You can set it to 1 to display only one word:



Figure 11.2: Selecting the number of words to display

You will see the numerical representation of the one word as shown in the following excerpt:

```
[-1.7703770e-04  8.0561076e-05  1.7074384e-03  3.0068988e-03
 -3.1028548e-03 -2.3738446e-03  2.1451029e-03  2.9889001e-03
 -1.6721861e-03 -1.2577033e-03  2.4635433e-03  -5.1635387e-04
 -1.5168249e-03  2.1891161e-03  -1.6221605e-03  -6.0506846e-04
  9.5858454e-04  3.3036861e-04  -2.7588590e-03  -3.1527190e-03
```

You can explore the other attributes of the model in the drop-down list and also add more if necessary. For more, consult Genism's Word2Vec documentation:

<https://radimrehurek.com/gensim/models/word2vec.html>

We displayed the content of some of the attributes of the model. We will now access a word and its vector directly.

## 5. Accessing a word and vector

We can view the processed tokens in section 2. *Tokenizing the text with Punkt* of this chapter.

We can see if a word we choose is in the vocabulary. For example, “consciousness” is not in the vocabulary:

```
try:  
    vector = model.wv['consciousness']  
    print('Vector for "consciousness":', vector)  
except KeyError:  
    print('"consciousness" is not in the dictionary')
```

The output confirms that the word is not in the dictionary:

```
"consciousness" is not in the dictionary
```

“Consciousness” shouldn’t be in the dictionary after lemmatization. However, “conscious,” the lemma (base word) of “consciousness,” should be present in this text, which contains philosophical and scientific reflections. Let’s try to find out:

```
try:  
    vector = model.wv['conscious']  
    print('Vector for "conscious":', vector)  
except KeyError:  
    print('"conscious" is not in the dictionary')
```

The following excerpt of the output of the word vector shows that the word “conscious” is in the vocabulary:

```
Vector for "conscious": [-2.2882714e-03 -2.9769554e-03 -2.33829  
4.9209874e-04 -7.2649046e-04 8.2537113e-04 -3.6820408e-04  
-1.1185763e-03 -1.8049215e-03 2.1578181e-03 -1.0867673e-03  
7.9940347e-04 2.3183119e-03 -1.7192229e-04 2.8174506e-03  
-1.5929197e-03 3.3099041e-03 -1.9293354e-03 -3.0342239e-04  
1.4344892e-04 3.6501623e-04 1.3159955e-03 2.5331231e-03  
-9.0605288e-04 1.5739332e-03 -1.6196299e-03 9.4100273e-05  
-7.1035558e-04 -9.2428079e-04 -6.3189666e-04 3.4200953e-04...
```

We will now ask the program to find the most similar words to “conscious”:

```
try:  
    similar_words = model.wv.most_similar('conscious')  
    print('Most similar words to "conscious":', similar_words)  
except KeyError:  
    print('"conscious" is not in the dictionary')
```

The output displays similar words in the vocabulary based on the text we processed:

```
Most similar words to "conscious":  
[('wouldconduce', 0.21150538325309753),  
 ('itscounterweights', 0.19385728240013123),  
 ('versed', 0.185345858335495),  
 ('comprehend', 0.1835186779499054),  
 ('requisite', 0.17346185445785522),  
 ('convertsthe', 0.1722400188446045),  
 ('ibelieved', 0.1657719910144806),  
 ('transmit', 0.16429133713245392),  
 ('speakingmyself', 0.16091690957546234),  
 ('warmth', 0.15618115663528442)]
```

The output contains the words similar to “conscious” and their similarity score between 0 and 1. A higher score corresponds to a higher similarity. The outputs may vary from one run to another, and the similarities may vary due to the stochastic nature of NLP algorithms.

Adding more documents for the tokenization function can optimize the embedding process. However, we have enough information for the example in this section.

Let's explore Gensim's vector space.

## 6. Exploring Gensim's vector space

After tokenization, a word turns into a token in the vocabulary obtained. Then, models like Word2Vec map each unique token to a unique ID or index. Adding an index to the words creates a dictionary. The embeddings are the representation of the words in the dictionary.

We can display the words in the dictionary and their indexes:

```
for word, index in model.wv.key_to_index.items():
    print(f"Word: {word}, Index: {index}")
```

The output contains the content of the dictionary, as shown in the following excerpt:

```
Word: one, Index: 0
Word: truth, Index: 1
Word: thought, Index: 2
Word: reason, Index: 3
Word: may, Index: 4
Word: could, Index: 5
Word: heart, Index: 6
Word: u, Index: 7
Word: certain, Index: 8
Word: might, Index: 9
Word: even, Index: 10
Word: much, Index: 11
Word: many, Index: 12
Word: opinion, Index: 13
Word: would, Index: 14
Word: blood, Index: 15
Word: without, Index: 16
```

Word: time, Index: 17

三

We can also create a function that determines the cosine similarity between word pairs in a list of words based on their respective embedding vectors.

We first define a list of words:

We then iterate over all the word pairs in the list and calculate their cosine similarity:

```
# Initialize list to store results
data = []
# Iterate over all pairs of words
for i in range(len(words)):
    for j in range(len(words)): # changed this line
        word1 = words[i]
        word2 = words[j]
        # Ensure words are in the model's vocabulary
        if word1 not in model.wv or word2 not in model.wv:
            print(f"One or both words ('{word1}', '{word2}') are not in the vocabulary")
            continue
# Calculate cosine similarity
        vec1 = model.wv[word1]
        vec2 = model.wv[word2]
        similarity = np.dot(matutils.unitvec(vec1), matutils.unitvec(vec2))
        # Convert similarity to distance
        distance = 1 - similarity
        # Append to our results
        data.append({'word1': word1, 'word2': word2, 'distance': distance})
```

Finally, we display the output in a pandas DataFrame:

```
# Create DataFrame and display
df = pd.DataFrame(data)
display(df)
```

The output shows the cosine similarity between the word pairs of the list, as displayed in the following excerpt:

	word1	word2	distance
0	method	method	0.000000e+00
1	method	reason	9.435999e-01
2	method	truth	9.442804e-01
3	method	rightly	9.486628e-01
4	method	science	1.051368e+00
5	method	seeking	1.011608e+00
6	reason	method	9.435999e-01
7	reason	reason	0.000000e+00
8	reason	truth	9.756580e-01
9	reason	rightly	9.934855e-01

Figure 11.3: Similarity between word pairs

The cosine similarity score in this high-dimensional vector space will produce values between -1 and 1:

- 1 means the vectors are identical. Higher values denote a higher similarity.
- 0 means the vectors are unrelated.
- -1 means the vectors are diametrically opposed.

Cosine similarity can be defined as:

$$\text{Cosine similarity}(A, B) = (A \cdot B) / (\|A\| \|B\|)$$

Where:

- A and B are vectors.
- “ $A \cdot B$ ” denotes the dot product of A and B.
- “ $\|A\|$ ” and “ $\|B\|$ ” denote the magnitudes (or lengths) of A and B, respectively.

We can display word embeddings with the TensorFlow Projector for a better visual representation.

## 7. TensorFlow Projector

We can visualize embeddings with the TensorFlow Embedding Projector:

<https://projector.tensorflow.org/>

To visualize the embeddings, we need to create two files:

- A vector file containing the embeddings. Usually, it is a file named `vecs.tsv` in the **TSV (Tab-Separated Values)** format.
- A metadata file containing the labels, the words in this case. Usually, it is a file named `meta.tsv`, again in the TSV format.

Both files must contain the vectors and the labels in the same order.

Let's write a function to generate and save the two files:

```

import csv
import os
import numpy as np
# Directory where you want to save the files
LOG_DIR = '/content'
os.makedirs(LOG_DIR, exist_ok=True)
# Get the words and vectors
words = list(model.wv.key_to_index.keys())
vectors = [model.wv[word] for word in words]
# Write the vectors to a .tsv file
with open(os.path.join(LOG_DIR, "vecs.tsv"), 'w', newline='') as f:
    writer = csv.writer(f, delimiter='\t')
    writer.writerows(vectors)
# Write the labels (words) to a separate .tsv file
with open(os.path.join(LOG_DIR, "meta.tsv"), 'w', newline='') as f:
    writer = csv.writer(f, delimiter='\t')
    writer.writerows([[word] for word in words]) # No header

```

The output is the two files we need to load in the TensorFlow Projector:

`vecs.tsv` and `meta.tsv`. You can save them locally and then upload them to the TensorFlow Projector.

You can check the files to see if they both are of the same size:

```

!echo "Vectors file (vecs.tsv) size:"
!wc -l /content/vecs.tsv
!echo "Metadata file (meta.tsv) size:"
!wc -l /content/meta.tsv

```

In this case, the output confirms they are of the same size. The following code uses the `wc -l` command to count the number of lines in each file:

```

Vectors file (vecs.tsv) size:
3843 /content/vecs.tsv

```

```
Metadata file (meta.tsv) size:  
3843 /content/meta.tsv
```

Once you have saved your files locally, go to the TensorFlow Projector site:  
<https://projector.tensorflow.org/>.

Click on the load button, and you will be prompted to upload your files:

### Load data from your computer

#### Step 1: Load a TSV file of vectors.

Example of 3 vectors with dimension 4:

```
0.1\t0.2\t0.5\t0.9  
0.2\t0.1\t5.0\t0.2  
0.4\t0.1\t7.0\t0.8
```

Choose file

#### Step 2 (optional): Load a TSV file of metadata.

Example of 3 data points and 2 columns.

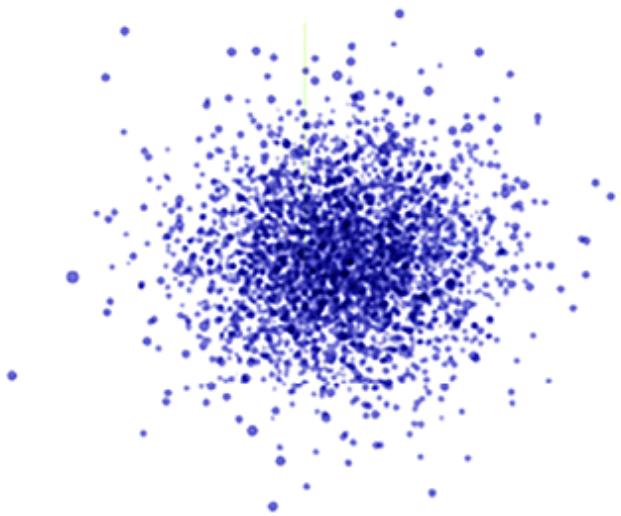
Note: If there is more than one column, the first row will be parsed as column labels.

```
Pokémon\tSpecies  
Wartortle\tTurtle  
Venusaur\tSeed  
Charmeleon\tFlame
```

Choose file

Figure 11.4: Loading data

Once the files are loaded, you will see the data points of your word embeddings:



*Figure 11.5: Data points of the word embeddings*

Each point in the visualization represents a word. The word's vector determines the position of the point. You can select a word such as **think** and visualize the related words in terms of similarity:

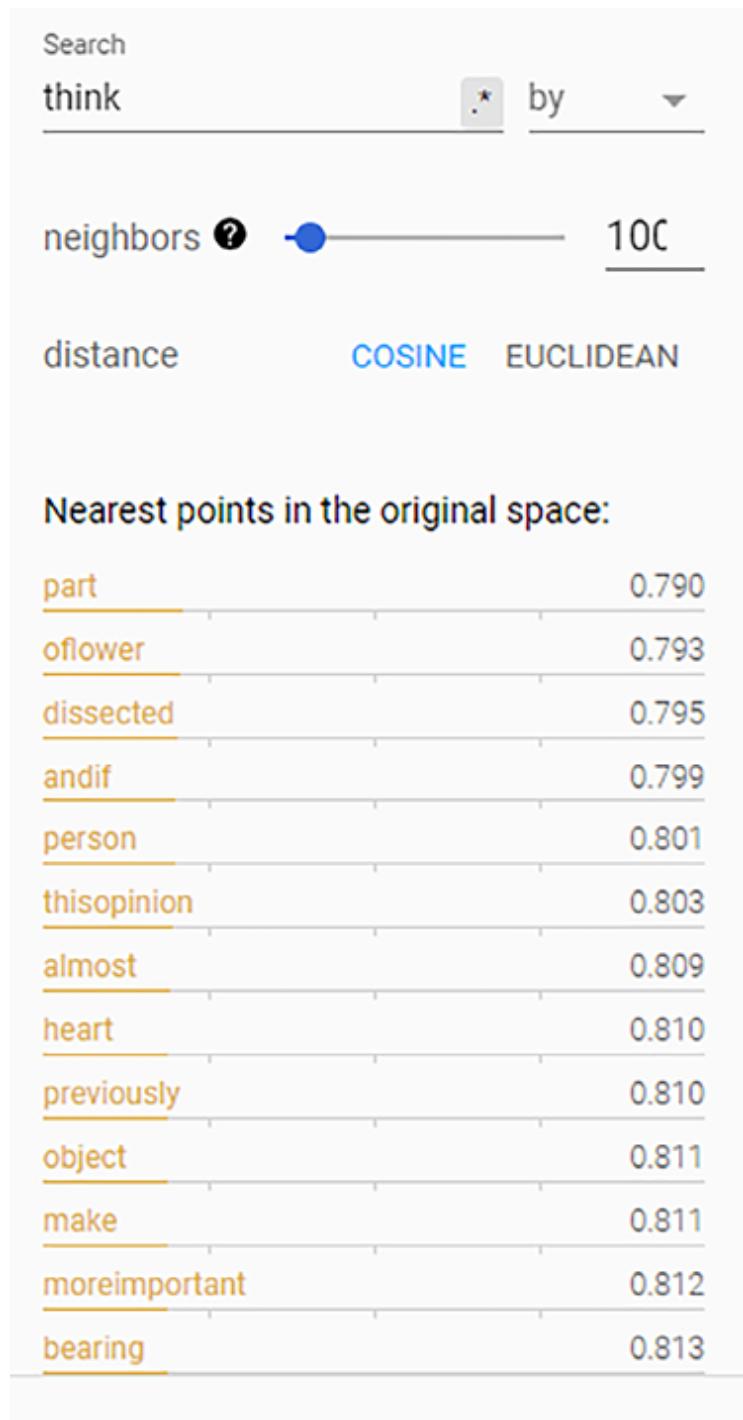


Figure 11.6: Similarity to the word “think”

Your choice will also appear in the visual vector space:

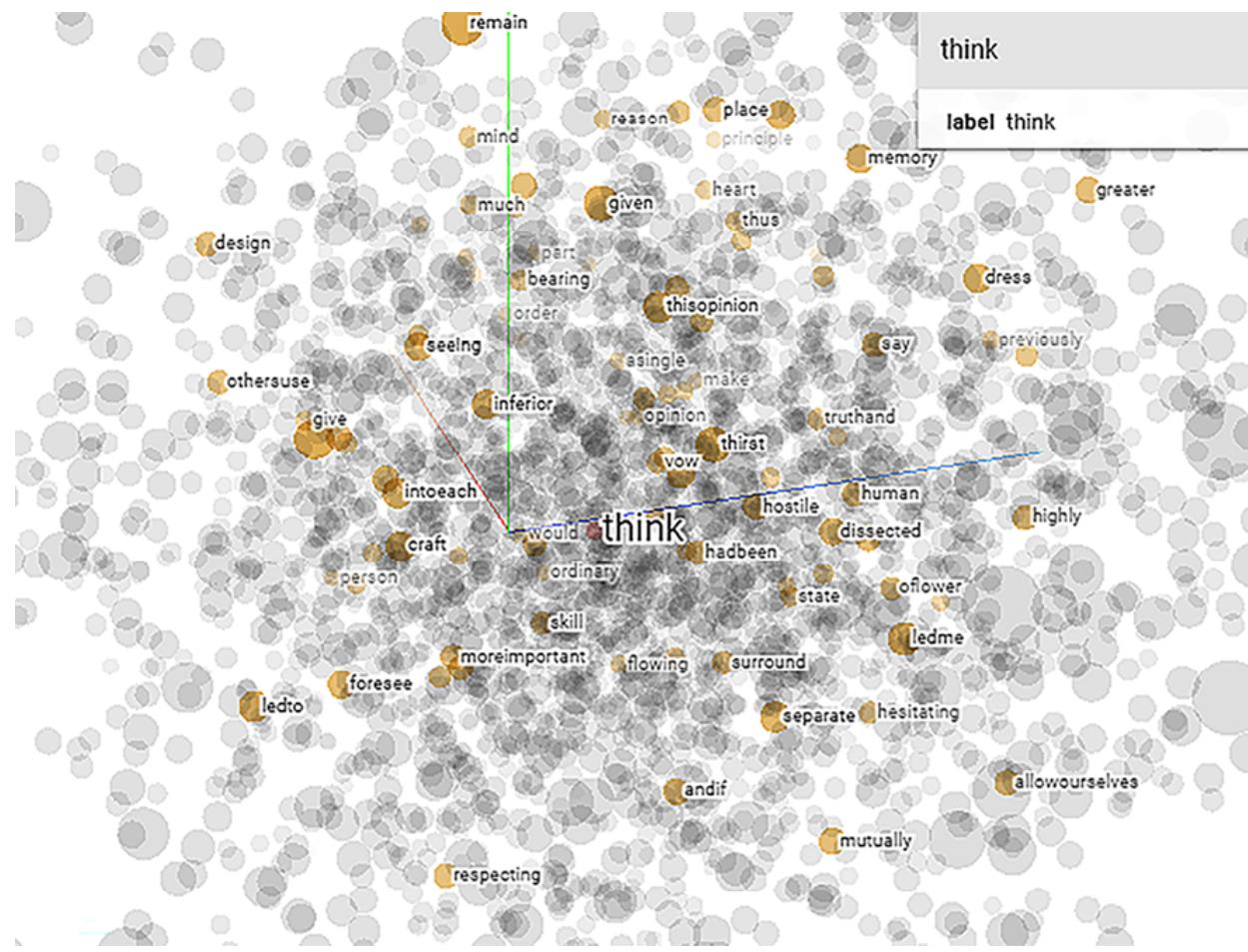


Figure 11.7: Vector space for the word “think”

The spherical visualization in this example was generated by applying Principal Component Analysis to the dataset. We used PCA in *Chapter 9, Shattering the Black Box with Interpretable Tools*, in the PCA subsection of the LIT section. It's an effective analysis tool.

*Principal Component Analysis (PCA) is beyond the scope of this book. However, in a nutshell, PCA is a statistical technique used to simplify complex data. Here is how:*

1. First, PCA detects the correlation between variables.
  2. Then, it reduces the number of variables by combining them into a set of “principal components.”

3. These components are ordered so that the first few retain most of the variation in the original variables.

We have now gone through the fundamentals to appreciate the tremendous potential of embedding-based search techniques fully.

We will now implement question-answering systems with embedding-based techniques.

## **Implementing question-answering systems with embedding-based search techniques**

In this section, we will use embedding-based search only. However, once you understand how to implement embedding-based search, you can add features to your application, such as recent user history and the popularity of topics.

Open `Question_answering_with_embeddings.ipynb` in the chapter directory of the repository.

The titles of the sections in the notebook match the titles in this section:

1. Installing the libraries and selecting the models
2. Implementing the embedding model and the generative model:
  1. Evaluating the model without a knowledge base: GPT cannot answer questions
  2. Adding a knowledge base
  3. Evaluating the model with a knowledge base: GPT can answer questions

3. Preparing the search data
4. Search
5. Ask:
  1. Example question
  2. Troubleshooting wrong answers

We will first install the necessary libraries.

## 1. Installing the libraries and selecting the models

We install `tiktoken`, which will be used to count tokens.

```
!pip install tiktoken
```

The program then installs the OpenAI library:

```
try:  
    import openai  
except:  
    !pip install openai  
    import openai
```



Note: In this section, we will implement the latest OpenAI API version. This option is efficient when working on a new project. However, if a critical project is in production, you may want to pin OpenAI's version, as we will see in the next section, *Transfer Learning with Ada Embeddings*.

The program retrieves the API from a file on Google Drive:

```
from google.colab import drive
drive.mount('/content/drive')
f = open("drive/MyDrive/files/api_key.txt", "r")
API_KEY=f.readline()
f.close()
```

You can store the key in a file so it does not appear in the program or enter it directly, but it will be visible.

The program sets the `API_KEY` in an environment variable:

```
import os
os.environ['OPENAI_API_KEY'] = API_KEY
openai.api_key = os.getenv("OPENAI_API_KEY")
```

We can now import the required modules:

```
# imports
import ast # for converting embeddings saved as strings back
import openai # for calling the OpenAI API
import pandas as pd # for storing text and embeddings data
import tiktoken # for counting tokens in this program
from scipy import spatial # for calculating vector similarities
```

The next step is implementing the models.

## 2. Implementing the embedding model and the GPT model

Choosing a model will determine the path your program and your project will take for an advanced NLP task. If you choose the wrong one, the accuracy of the outputs will be low. If you choose the right one for accuracy, it might cost too much for your project.

You will have to find a balance between accuracy and cost.

`text-embedding-ada-002` was chosen for the embeddings, and `gpt-3.5-turbo` for the question-answering task, in this notebook:

```
# models
EMBEDDING_MODEL = "text-embedding-ada-002"
GPT_MODEL = "gpt-3.5-turbo"
```

`text-embedding-ada-002` is an OpenAI second-generation embedding model. It will accept a maximum input of 8,191 tokens and generate output vectors with 1,536 dimensions.

Take a few seconds to let this information sink in: 1,536 dimensions for each token compared to 300 dimensions, for example, for each word in Gensim's vector space.

GPT-3.5-turbo and GPT-4 can both understand and generate natural language. GPT-4 improves GPT-3.5-turbo. However, do we need GPT-4 for every task? Do we need the most powerful model for everything? No. We don't. However, the choice of a model over another requires careful evaluation.

Cost remains critical when choosing between one model and another. Make sure to check OpenAI's pricing page before implementing a model:  
<https://openai.com/pricing#language-models>.

We will now evaluate the model.

## 2.1 Evaluating the model with a knowledge base: GPT can answer questions

This section will evaluate a model in terms of memory, which depends on the training data. For example, in January 2024, GPT-3.5-turbo and GPT-4 cannot answer questions beyond April 2023. Datasets have cutoff dates, and training requires tremendous machine and human resources.

We might be able to train a very small model easily, but will it have the vast abilities of GPT-4 to understand questions? This depends on the requirements of each project.

In any case, GPT-3.5-turbo has a memory issue. You can try to fine-tune it with a dataset. But, the next day, it won't be able to answer a question on recent sports events, for example. Fine-tuning the model continually can be costly.

GPT-3.5-turbo's long-term memory works, but it has no short-term memory because its dataset doesn't have the information. For example, in the following cell of the notebook, it cannot answer a question about the 2022 Olympics:

```
# an example question about the 2022 Olympics
query = 'Which athletes won the gold medal in curling at the 2022 Olympics'
response = client.chat.completions.create(
    messages=[
        {'role': 'system', 'content': 'You answer questions about the 2022 Olympics'},
        {'role': 'user', 'content': query},
    ],
    model=GPT_MODEL,
    temperature=0,
)
print(response.choices[0].message.content)
```

The response will be:

```
As an AI language model, I don't have real-time data. However,
```

The model cannot answer the question because it doesn't possess the information. OpenAI will surely produce a new model. But it will have another cutoff date. If you try the new model, it still will not answer questions that are not in the training dataset.

Let's try adding a knowledge base to the request we make to the model.

## 2.2 Add a knowledge base

To help the model answer questions on data that exceeds its cutoff date, we can add information from Wikipedia to the message in the request. This process can be performed automatically. For this test, it was copied and pasted as shown in the following excerpt:

```
# text copied and pasted from: https://en.wikipedia.org/wiki/Curling_at_the_2022_Winter_Olympics
# I didn't bother to format or clean the text, but GPT will still understand it
# the entire article is too long for gpt-3.5-turbo, so I only included the beginning
wikipedia_article_on_curling = """Curling at the 2022 Winter Olympics is a winter sport competition at the 2022 Winter Olympics in Beijing, China. It is the second time curling has been included in the Olympic program. The competition took place at the National Aquatics Center (Water Cube) from February 9 to 20, 2022. A total of 30 medals were awarded across all disciplines. The United States won the most medals, followed by Canada and Norway. The men's team from Norway won the gold medal, while the women's team from Canada won the silver medal. The mixed doubles team from Norway won the bronze medal.
```

Will the model answer correctly? Let's find out!

## 2.3 Evaluating the model without a knowledge base: GPT cannot answer questions

We instruct the model to use the Wikipedia article to answer a question:

```
query = f"""Use the below article on the 2022 Winter Olympics
Article:
\\"""
{wikipedia_article_on_curling}
\\"""

Question: Which athletes won the gold medal in curling at the 2022 Winter Olympics?
from openai import OpenAI
client = OpenAI()
response = client.chat.completions.create(
    messages=[
        {'role': 'system', 'content': 'You answer questions about the 2022 Winter Olympics'},
        {'role': 'user', 'content': query},
    ],
    model=GPT_MODEL,
    temperature=0,
)
print(response.choices[0].message.content)
```

The answer is correct:

```
The athletes who won the gold medal in curling at the 2022 Winter Olympics:
Men's Curling: Sweden (Niklas Edin, Oskar Eriksson, Rasmus Wran)
Women's Curling: Great Britain (Eve Muirhead, Vicki Wright, Jen Jones)
Mixed Doubles Curling: Italy (Stefania Constantini, Amos Mosane)
```

Let's sum up this approach:

- A model has limited knowledge due to its training dataset. If the dataset does not contain the information required (cutoff date, lack of information), it cannot answer a question.
- We can copy and paste information into the message part of a request.
- We can also create a knowledge base in a database or file system, map the data, and retrieve it when we need it automatically with classical

data querying techniques. We then have an automated system.

A knowledge-based method might fit a project's requirements. The choice will depend on each project's requirements.

Sometimes, a custom query-search program might not meet expectations on complex heterogeneous data. In that case, classical query-search techniques to access a knowledge base cannot match the power of OpenAI's second-generation embeddings-based search models.

One of the limits of a knowledge-based approach built with classical queries will be the number of dimensions of the search function. We cannot match the thousands of dimensions of an embedding-based search algorithm with a classical query!

We're ready to dive into embeddings-based search.

## 3. Prepare search data

In this section, we will still use a knowledge base. However, instead of building a query directly on the text data, we will prepare the data to work with the embedded vectors of the knowledge base.

We will embed search data in the *Transfer learning with Ada embeddings* section of this chapter. In this notebook, we will use the dataset OpenAI prepared in a few steps:

- **Collect:** Downloading a few hundred Wikipedia articles about the 2022 Olympics
- **Chunk:** Splitting documents into short, nearly self-contained sections
- **Embed:** Embedding each section with the OpenAI API
- **Store:** Saving the data in a file. Large datasets should be stored in a vector database. Embedding vectors can be indexed in a vector database

and accessed with power search functionalities. Several platforms offer vector database services, such as **Amazon Web Services (AWS)**:  
<https://aws.amazon.com/what-is/vector-databases/>.

To see more on how OpenAI constructed the dataset in this notebook, you can explore the following notebook:

[https://github.com/openai/openai-cookbook/blob/950246dd0810470291aa9728c404a01aeab5a1e9/examples/Embedding\\_Wikipedia\\_articles\\_for\\_search.ipynb](https://github.com/openai/openai-cookbook/blob/950246dd0810470291aa9728c404a01aeab5a1e9/examples/Embedding_Wikipedia_articles_for_search.ipynb).

You will also go through the embedding process of raw data in *Transfer learning with Ada embeddings* in the next section of this chapter.

In this section, we'll load the embeddings provided by OpenAI:

```
# download pre-chunked text and pre-computed embeddings
# this file is ~200 MB, so may take a minute depending on your
embeddings_path = "https://cdn.openai.com/API/examples/data/wiki.csv"
df = pd.read_csv(embeddings_path)
```

Then we convert the embeddings from the string type back to the list type:

```
# convert embeddings from CSV str type back to list type
df['embedding'] = df['embedding'].apply(ast.literal_eval)
```

The pandas DataFrame contains two columns: `text` and `embedding`:

```
# the dataframe has two columns: "text" and "embedding"
df
```

The output is as follows:

	text	embedding
0	Lviv bid for the 2022 Winter Olympics\n\n({Oly...	[-0.005021067801862955, 0.00026050032465718687...
1	Lviv bid for the 2022 Winter Olympics\n\n==His...	[0.0033927420154213905, -0.007447326090186834,...
2	Lviv bid for the 2022 Winter Olympics\n\n==Ven...	[-0.00915789045393467, -0.008366798982024193, ...
3	Lviv bid for the 2022 Winter Olympics\n\n==Ven...	[0.0030951891094446182, -0.006064314860850573,...
4	Lviv bid for the 2022 Winter Olympics\n\n==Ven...	[-0.002936174161732197, -0.006185177247971296,...

Figure 11.8: Embeddings data

We can print the length of a vector in the embedding column:

```
embedding_size = len(df['embedding'].iloc[0])
print(embedding_size)
```

The output confirms that the `text-embedding-ada-002` model produces a 1,536-dimensional embedding vector:

```
1536
```

The notebook now implements search functionality.

## 4. Search

A user will make a request in text format. The text must be embedded with the same model as the embedding vectors. Then, the distance between the embedded request and the embedded records in the pandas DataFrame are computed and sorted by rank.

The `strings_ranked_by_relatedness` function takes in a query string, the pandas DataFrame, and the `top_n` number:

```
def strings_ranked_by_relatedness(
    query: str,
    df: pd.DataFrame,
    relatedness_fn=lambda x, y: 1 - spatial.distance.cosine(x,
        top_n: int = 100
    ) -> tuple[list[str], list[float]]:
```

The function returns a tuple of two lists: one containing the `top_n` strings most related to the query and the other containing their corresponding *relatedness* scores.

First, the function embeds the query string using the same model as the one used to embed the search data:

```
from openai import OpenAI
client = OpenAI()
query_embedding_response = client.embeddings.create(
    model=EMBEDDING_MODEL,
    input=query,
)
```

The function retrieves the embedding vector for the query from the response:

```
query_embedding = query_embedding_response.data[0].embedding
```

Then, the function iterates over the rows in the DataFrame. A tuple is created from the row containing the text in that row and the relatedness of the text (to the query):

```
strings_and_relatednesses = [
    (row["text"], relatedness_fn(query_embedding, row["embedding"]),
     row["label"])
    for i, row in df.iterrows()
```

The list obtained is sorted in descending order of relatedness:

```
strings_and_relatednesses.sort(key=lambda x: x[1], reverse=True)
```

Finally, a `zip` function separates the list of tuples into a strings list named `strings` and a relatedness list named `relatedness`:

```
strings, relatednesses = zip(*strings_and_relatednesses)
```

The function returns the `top_n` elements from both the `strings` and `relatednesses` lists:

```
return strings[:top_n], relatednesses[:top_n]
```

We will now run an example to test the function:

```
# examples
strings, relatednesses = strings_ranked_by_relatedness("curlin")
for string, relatedness in zip(strings, relatednesses):
    print(f"relatedness={:.3f}")
    display(string)
```

The response displays the top 5 most related texts and displays them with their relatedness score in descending order, as shown in the following excerpt:

```
relatedness=0.879
'
Curling at the 2022 Winter Olympics\n\n==Medal summary==\n\n==
'
relatedness=0.872
```

We confirmed that the relatedness function works. We now have to insert the most related text in the message of the OpenAI GPT-3.5-turbo request.

## 5. Ask

The ask functionality of the notebook automates the whole embedded-based question-answering process.

The ask functionality begins by counting the number of tokens of a request. The goal is to make sure the number of tokens in the message does not exceed the token budget (model limit, project limit):

```
def num_tokens(text: str, model: str = GPT_MODEL) -> int:
    """Return the number of tokens in a string."""
    encoding = tiktoken.encoding_for_model(model)
    return len(encoding.encode(text))
```

For example, in this case, the token budget will be limited to 4,096 (total token limit) minus 500 (tokens of the request) in the ask function because

GPT-3 has a maximum limit of 4,096 tokens. We need to send the request but also leave room for the response when we manage token budgets.

The query message function takes the query string, the pandas DataFrame, the model, and the token budget:

```
def query_message(  
    query: str,  
    df: pd.DataFrame,  
    model: str,  
    token_budget: int  
) -> str:
```

It will then find articles using the `relatedness` function:

```
strings, relatednesses = strings_ranked_by_relatedness(query, )
```

It will then prepare an introduction message giving the instructions to answer the query:

```
introduction = 'Use the below articles on the 2022 Winter Olym
```

It will store the introduction in the message variable before adding articles:

```
message = introduction
```

It then defines the question:

```
question = f»\n\nQuestion: {query}»
```

Finally, the function adds articles to `next_article` as long as `token_budget` is not reached:

```
for string in strings:
    next_article = f'\n\nWikipedia article section:\n"""\nif (
        num_tokens(message + next_article + question, mode.
        > token_budget
    ):
        break
```

```
else:
    message += next_article
return message + question
```

The GPT request has been automated effectively. We can run the question-answering system.

## 5.1.Example question

We have gone from prompt design, designing a request with no engineering, to prompt engineering with an embedded-based RAG question-answering system.

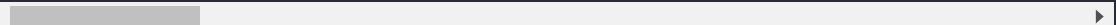
We can ask the system the original question about the 2022 Olympics:

```
ask('Which athletes won the gold medal in curling at the 2022 Wint
```



The GPT model can respond, and the response is accurate:

```
In the men's curling tournament, the gold medal was won by the
```



The result isn't perfect, but it's a good step forward. For example, the dataset could be preprocessed in more fine-grained detail in a project on this topic.

In this case, the model failed to list the gold medal winners of one of the events (mixed doubles):

```
|Mixed doubles<br/>{{DetailsLink|Curling at the 2022 Winter Oly  
|{{flagIOC|ITA|2022 Winter}}<br>[[Stefania Constantini]]<br>[[A  
|{{flagIOC|NOR|2022 Winter}}<br>[[Kristin Skaslien]]<br>[[Magnu  
|{{flagIOC|SWE|2022 Winter}}<br>[[Almida de Val]]<br>[[Oskar Er  
|}}
```



We need more information to see what went wrong.

## 5.2. Troubleshooting wrong answers

The information in this previous response could be:

- A lack of information in the message we sent.
- A reasoning error on the part of the GPT model.

We can find out by setting the `print_message` option to `True`:

```
# set print_message=True to see the source text GPT was working on
ask('Which athletes won the gold medal in curling at the 2022 Winter Olympics?')
```

The output shows the information sent to the model and the query, as shown in the following excerpt, including the mixed doubles missing in the response:

The information we needed was in the message, so the model had a reasoning issue.

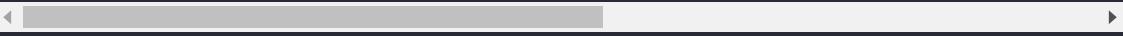
The hard way out of this issue is to improve the search dataset to make it possible for the model to respond correctly. We can also try the easy way first, by running the same query with the GPT-4 model, which is more powerful than the GPT-3.5 model:

```
ask('Which athletes won the gold medal in curling at the 2022 !
```

This time, the response contains the women's mixed doubles gold medalists:

The athletes who won the gold medal in curling at the 2022 Winter Olympics are:

```
Men's tournament: Niklas Edin, Oskar Eriksson, Rasmus Wranå, Ch  
Women's tournament: Eve Muirhead, Vicki Wright, Jennifer Dodds,  
Mixed doubles tournament: Stefania Constantini and Amos Mosaner
```



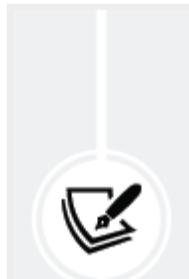
Try the examples in section 5.3. *More Examples* in the following section of the notebook before moving on to the next section. You will see the score and limits of the system.

We will now go further and embed a search dataset for question-answering.

## Transfer learning with Ada embeddings

In this section, we will go further, run an embeddings function, create embeddings clusters with a k-means clustering algorithm, and ask davinci to describe the theme of each cluster with the corresponding reviews in text format.

Open `Transfer_Learning_with_Ada_EMBEDDINGS.ipynb`, which is in the chapter notebook of the GitHub repository.



Note: In some cases, a project may be critical, costly, and require stability. One cannot always update source code without going through project management procedures. We

can pin the version of OpenAI in this case, as we did in this section for back compatibility: `!pip install openai==0.28`

The first cells of the notebook install OpenAI `tiktoken` and Kaggle. Then the notebooks sets the API keys for OpenAI and Kaggle. You will need an account for OpenAI and Kaggle.

## 1. The Amazon Fine Food Reviews dataset

The Amazon Fine Food Reviews dataset contains 568,454 food reviews written by users up to October 2012. The notebook will extract a subset of the 1,000 most recent reviews. The reviews can be classified as rather positive or negative. The review records contain a product ID, user ID, score, review title (summary), and review body (text).

The review summary and review will be merged into a single combined text embedded into a single vector.

We will first download the dataset from

<https://www.kaggle.com/datasets/snap/amazon-fine-food-reviews> with our Kaggle account:

```
!kaggle datasets download -d snap/amazon-fine-food-reviews
```

We will extract the reviews in the `Reviews.csv` file:

```
import zipfile
zip_file_path = '/content/amazon-fine-food-reviews.zip'
csv_file_name = 'Reviews.csv'
```

```
with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:  
    zip_ref.extract(csv_file_name)
```

We will now prepare the data.

## 1.2. Data preparation

We import `pandas` and `tiktoken`, and the OpenAI embeddings module:

```
# imports  
import pandas as pd  
import tiktoken  
from openai.embeddings_utils import get_embedding
```

We select the `text-embedding-ada-002` model:

```
# embedding model parameters  
embedding_model = "text-embedding-ada-002"  
embedding_encoding = "cl100k_base" # this is the encoding for  
max_tokens = 8000 # the maximum for text-embedding-ada-002 is 8000
```

The tokens are carefully set to 8,000 to remain under Ada's 8,191 token limit. If you reach the limit, a response may be truncated.

Note that the `text-embedding-ada-002` model does not have an encoding base of its own. It relies on the `cl100k` base. The `cl100k` base is a set of pre-trained word embeddings that are used to initialize the `text-embedding-ada-002` model that learns how to adapt the embeddings to a specific task.

We now load the dataset and create the `combined` column that contains the summary and the content of a review:

```

# load & inspect dataset
input_datapath = "/content/Reviews.csv" # to save space, we pre
df = pd.read_csv(input_datapath, index_col=0)
df = df[["Time", "ProductId", "UserId", "Score", "Summary", "T
df = df.dropna()
df[["combined"]] = (
    "Title: " + df.Summary.str.strip() + "; Content: " + df.Text
)

```

We can display a few records of the DataFrame to see if the combined field looks acceptable:

```
df.head(10)
```

The output is correct, as shown in the following excerpt:

<b>Id</b>	<b>Time</b>	<b>ProductId</b>	<b>UserId</b>	<b>Score</b>	<b>Summary</b>	<b>Text</b>	<b>combined</b>
1	1303962400	B001E4KFG0	A3SGXH7AUHU8GW	5	Good Quality Dog Food	I have bought several of the Vitality canned dog food products and have found them all to be of good quality. The product looks more like a stew than a processed meat and it smells better. My Labrador is finicky and she appreciates this product better than most.	Title: Good Quality Dog Food; Content: I have bought several of the Vitality canned dog food products and have found them all to be of good quality. The product looks more like a stew than a processed meat and it smells better. My Labrador is finicky and she appreciates this product better than most.
2	1346976000	B00813GRG4	A1D87F6ZCVE5NK	1	Not as Advertised	Product arrived labeled as Jumbo Salted Peanuts...the peanuts were actually small sized unsalted. Not sure if this was an error or if the vendor intended to represent the product as "Jumbo".	Title: Not as Advertised; Content: Product arrived labeled as Jumbo Salted Peanuts...the peanuts were actually small sized unsalted. Not sure if this was an error or if the vendor intended to represent the product as "Jumbo".

Figure 11.9: The output of the DataFrame

The notebook now prepares a subset of the 1,000 most recent reviews and encodes them with `tiktoken`:

```
# sub sample to 1k most recent reviews and remove samples that
top_n = 1000
df = df.sort_values("Time").tail(top_n * 2) # first cut to fi
df.drop("Time", axis=1, inplace=True)
encoding = tiktoken.get_encoding(embedding_encoding)
# omit reviews that are too long to embed
df["n_tokens"] = df.combined.apply(lambda x: len(encoding.enco
df = df[df.n_tokens <= max_tokens].tail(top_n)
len(df)
```

The output shows we successfully extracted and encoded the data:

```
1000
```

The notebook will now embed the dataset.

## 2. Running Ada embeddings and saving them for future reuse

Ada will now embed the `combined` column of the dataset and store the embedding vectors in the `embedding` column:

```
# Ensure you have your API key set in your environment per the
# This may take a few minutes
df["embedding"] = df.combined.apply(lambda x: get_embedding(x,
df.to_csv("fine_food_reviews_with_embeddings_1k.csv")
```

The interesting point in this process is that Ada embedded a long text string, not just a word. Ada has taken a complete sequence and context into account to produce the embeddings.

We now save our embeddings for further use:

```
#f = open("drive/MyDrive/files/api_key.txt", "r")
!cp /content/fine_food_reviews_with_embeddings_1k.csv drive/My
```

We will now apply a clustering algorithm to the vector space we obtained.

### 3. Clustering

In this section, we will create clusters of reviews using a k-means clustering algorithm. We will obtain an overall view of sets of reviews.

Let's load the fine food reviews with the embeddings that we just generated.



If you rerun the notebook, you can run it from here after installing the necessary libraries. You will not have to rerun the embeddings section.

The following code loads the data into a pandas DataFrame and filters the bad lines if there are any:

```
# imports
import numpy as np
import pandas as pd
# load data
datafile_path = "fine_food_reviews_with_embeddings_1k.csv"
#df = pd.read_csv(datafile_path)
# read the csv file skipping bad lines
df = pd.read_csv('fine_food_reviews_with_embeddings_1k.csv', e
# count number of lines in the dataframe
df_line_count = len(df)
# count total lines in the csv file
```

```
with open('fine_food_reviews_with_embeddings_1k.csv') as f:  
    total_line_count = sum(1 for _ in enumerate(f))  
    # calculate number of bad lines  
    bad_lines = total_line_count - df_line_count  
    print(f'Number of bad lines: {bad_lines}')
```

The output shows that one line was filtered:

```
Number of bad lines: 1
```

The program now converts the embedding column into a NumPy array:

```
df["embedding"] = df.embedding.apply(eval).apply(np.array) #  
matrix = np.vstack(df.embedding.values)  
matrix.shape
```

The shape of the matrix shows that we have 1,000 lines and that the Ada embeddings have 1,536 dimensions:

```
(1000, 1536)
```

We are now ready to find the clusters with k-means clustering.

### 3.1. Find the clusters using k-means clustering

A classical k-means clustering algorithm is applied to the embedding matrix of the reviews to produce 4 clusters:

```
from sklearn.cluster import KMeans
n_clusters = 4
kmeans = KMeans(n_clusters=n_clusters, init="k-means++", n_init=10)
kmeans.fit(matrix)
labels = kmeans.labels_
df["cluster"] = labels
df.groupby("cluster").Score.mean().sort_values()
```

The output displays the mean point of all the data points in each of the 4 clusters:

```
2    4.081560
1    4.191176
0    4.221805
3    4.344937
Name: Score, dtype: float64
```

We can display the clusters with -t-SNE.

## 3.2. Display clusters with t-SNE

We will now display the data points using a t-SNE algorithm. t-SNE will take our high-dimensional data of 1,536 dimensions and bring them down to 2 dimensions while maintaining the relationships and patterns in the data. This way, we can visualize the clusters:

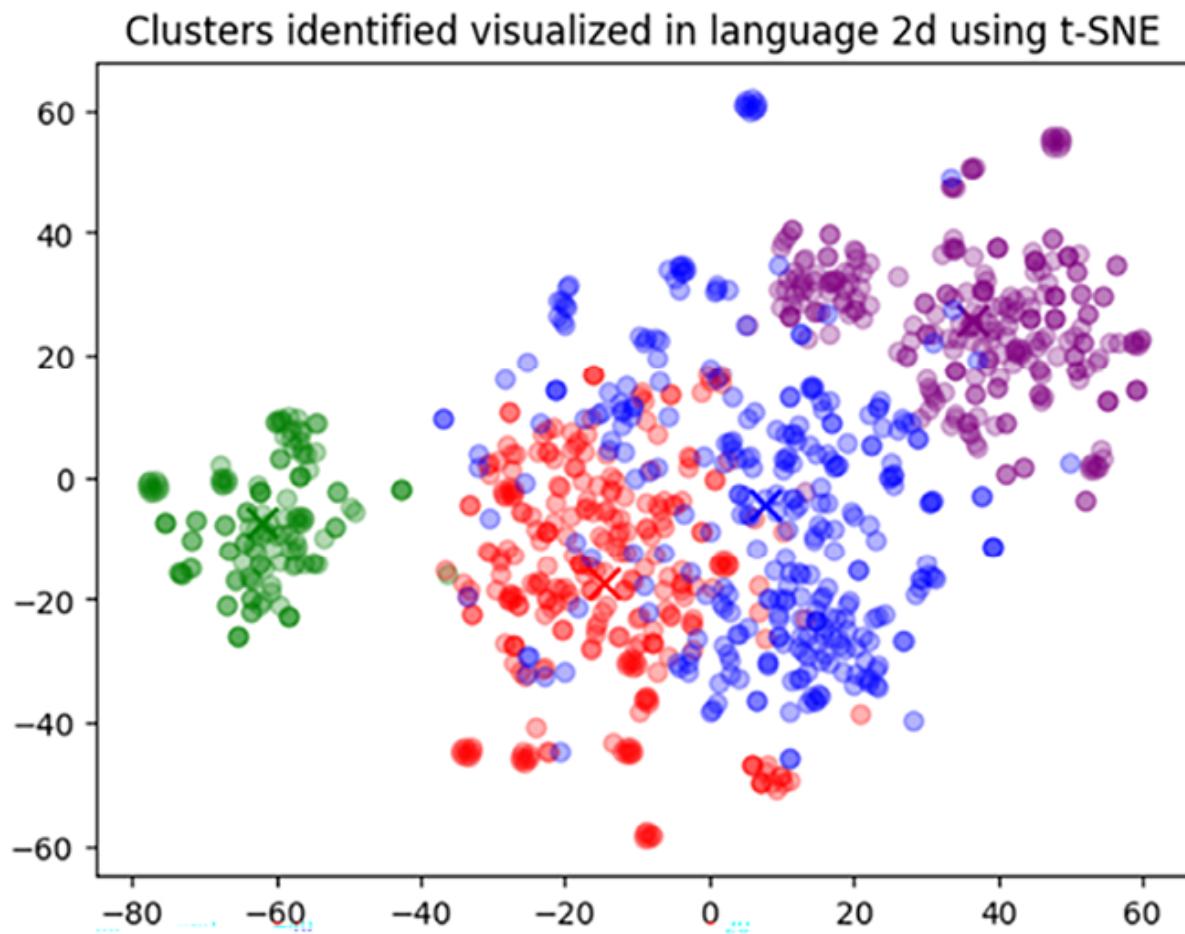
```
from sklearn.manifold import TSNE
import matplotlib
import matplotlib.pyplot as plt
tsne = TSNE(n_components=2, perplexity=15, random_state=42, init="pca")
vis_dims2 = tsne.fit_transform(matrix)
x = [x for x, y in vis_dims2]
y = [y for x, y in vis_dims2]
```

```

for category, color in enumerate(["purple", "green", "red", "blue"]):
    xs = np.array(x)[df.Cluster == category]
    ys = np.array(y)[df.Cluster == category]
    plt.scatter(xs, ys, color=color, alpha=0.3)
    avg_x = xs.mean()
    avg_y = ys.mean()
    plt.scatter(avg_x, avg_y, marker="x", color=color, s=100)
plt.title("Clusters identified visualized in language 2d using t-SNE")

```

The output is a nice visual representation of the clusters:



*Figure 11.10: Visualizing clusters with t-SNE*

We are ready to run text samples.

## 4. Text samples in the clusters and naming the clusters

We will select 5 reviews per cluster, give them a title, and ask `davinci-002` to find what they have in common:

```
import openai
# Reading a review that belongs to each group.
rev_per_cluster = 5
for i in range(n_clusters):
    print(f"Cluster {i} Theme:", end=" ")
    reviews = "\n".join(
        df[df.Cluster == i]
        .combined.str.replace("Title: ", "")
        .str.replace("\n\nContent: ", ":" )
        .sample(rev_per_cluster, random_state=42)
        .values
    )
    response = openai.Completion.create(
        engine="davinci-002",
        prompt=f'What do the following customer reviews have in common?
temperature=0,
max_tokens=64,
top_p=1,
frequency_penalty=0,
presence_penalty=0,
)
print(response["choices"][0]["text"].replace("\n", ""))
sample_cluster_rows = df[df.Cluster == i].sample(rev_per_cluster)
for j in range(rev_per_cluster):
    print(sample_cluster_rows.Score.values[j], end=", ")
    print(sample_cluster_rows.Summary.values[j], end=":")
    print(sample_cluster_rows.Text.str[:70].values[j])
print("-" * 100)
```

OpenAI davinci produces some interesting insights for the theme of each cluster as asked:

```
Cluster 0 Theme: All of the reviews are positive and discuss t  
5, breakfast tea: We switch to this decaf tea at night for a  
5, It is awesome.: My partner is very happy with the tea, and  
4, Chike!: Just tried the orange and iced coffee this morning  
5, FAVORITE tea...: Lipton makes the BEST French Vanilla tea.  
5, Twinings---a good cup of tea: I have been drinking Twining
```

The marketing potential to understand customer reviews on large datasets is tremendous!

We have gone through innovative search systems. Let's sum up our journey and move on to the next adventure!

## Summary

This chapter took us from prompt design to prompt engineering. Inventing creative prompts will no doubt help end users. Customizing the implementation of a model becomes necessary when the model's limits become too challenging to overcome.

Fine-tuning a model might not solve a question-answering issue for a system that needs to respond to incoming daily events. We cannot fine-tune a model every day.

We must find an alternative, such as an embedding-based search system, in such cases. Embedding-based searching is one of the many possibilities to implement RAG.

Our first step was to review the fundamentals of text embeddings with the Natural Language Toolkit and Gensim. We tokenized a text by Réne Descartes and then preprocessed the tokens. We continued by embedding the tokens with Gensim's Word2Vec module. Finally, we explored Gensim's vector space with TensorFlow Projector.

Our second step was to implement a question-answering system with an embedding-based search. We began by running a GPT-3.5-turbo example, which could not answer questions about the 2022 Olympics because its dataset's cutoff date was September 2021. We then added a knowledge base to the system and included information in the message in the request sent to the OpenAI API. We then went further by using an embedded dataset to implement search functions.

Our third and final step was to run an embedding model on the Amazon Fine Food Reviews dataset we downloaded from Kaggle. We then implemented an ask-and-search system using a clustering algorithm for sentiment analysis.

You now understand the process. You can replace the datasets with your own datasets and the models with models of your choice.

We have engineered an effective system for question-answering with innovative methods. We will continue implementing advanced transformer models with syntax-free semantic role labeling in the next chapter, *Toward Syntax-Free Semantic Role Labeling with BERT and ChatGPT*.

## Questions

1. Prompt design is the same thing as prompt engineering. (True/False)
2. OpenAI doesn't have embedding models. (True/False)

3. Ada is a chat model, not an embedding model. (True/False)
4. An Ada embedding vector contains two dimensions. (True/False)
5. GPT-3.5-turbo cannot answer questions. (True/False)
6. GPT-4 has better reasoning abilities than GPT-3.5 turbo. (True/False)
7. We don't need to manage cost parameters. (True/False)
8. GPT models don't have a maximum input token limit. (True/False)
9. A query must be embedded before being used with an embedded dataset. (True/False)
10. Embeddings-based search can be an effective alternative to fine-tuning a model. (True/False)

## References

- Gensim's Word2Vec documentation:  
<https://radimrehurek.com/gensim/models/word2vec.html>
- OpenAI's embedding models:  
<https://platform.openai.com/docs/guides/embedding-models/embedding-models>
- OpenAI's pricing page:  
<https://openai.com/pricing#language-models>

## Further reading

- *Dar et al., 2022, Analyzing Transformers in Embedding Space:*  
<https://arxiv.org/abs/2209.02535>

# Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://www.packt.link/Transformers>



# 12

## Toward Syntax-Free Semantic Role Labeling with ChatGPT and GPT-4

Transformers have made more progress in the past few years than NLP in the past generation. Former NLP models would be trained to understand a language's basic syntax before running **Semantic Role Labeling (SRL)**. The NLP software contained syntax trees, rule bases, and parsers. The performance of such systems was limited by the number of combinations of words that led to an infinity of contexts.

*Shi and Lin (2019)* started their paper by asking if preliminary syntactic and lexical training can be skipped. Could a system become “syntax-free” and understand language without relying on pre-designed syntax trees? Could a BERT-based model perform SRL without going through those classical training phases? The answer is yes!

*Shi and Lin (2019)* suggested that SRL can be considered sequence labeling and provide a standardized input format. Since then, OpenAI has reached near human-level syntax-free SRL. GPT-4 goes beyond training a model to perform SRL, even syntax-free labeling. ChatGPT with GPT-4 can perform SRL, *although they are not explicitly trained for it.*

ChatGPT with GPT-4 is a Generative AI autoregressive **Large Language Model (LLM)**, as we saw in *Chapter 1, What Are Transformers?*. As such, GPT-4 is stochastic; it produces the most probable tokens in a sequence, but it will not always repeat itself. These evolutions take us into a completely new mindset in artificial intelligence. GPT-4 is syntax-free (doesn't rely on rule bases) and stochastic (will often not repeat itself).

The paradigm shift is a quantum leap into the future of AI. Do not look for a system that repeats itself or shows consistent output for each run. Look for pertinence. The critical question remains to assess whether the response is reliable, not whether it repeats itself! The random nature of ChatGPT is what makes it appealing and human-like!

We will begin our journey by getting started with cutting-edge LLM SRL. We will go through the revolutionary concepts of syntax-free, non-repetitive stochastic models.

We will use ChatGPT Plus with GPT-4 to run easy to complex samples. Then, we will install OpenAI and then explain to GPT-4 what we expect from the model. We will run basic to difficult SRL samples in a Google Colab notebook. Finally, we will challenge ChatGPT by running complex SRL samples. We will see how a general-purpose, emergent model reacts to our SRL requests. We will progressively push the transformer 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:

- The issue of task-specific versus general-purpose models, including SRL
- The issue of self-service versus deployment with development

- Defining SRL
- Defining the standardization of the input format for SRL
- Testing sentence labeling on basic examples
- Testing SRL on complex examples and explaining the results
- Taking the transformer model to the limit of SRL and describing how this was done
- Building a notebook to run basic and complex SRL samples through GPT-4



With all the innovations and library updates in this cutting-edge field, packages and models change regularly. Please go to the GitHub repository for the latest installation and code examples:

<https://github.com/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition/tree/main/Chapter12>.

You can also post a message in our Discord community (<https://www.packt.link/Transformers>) if you have any trouble running the code in this or any chapter.

Our first step will be to get started with cutting-edge SRL.

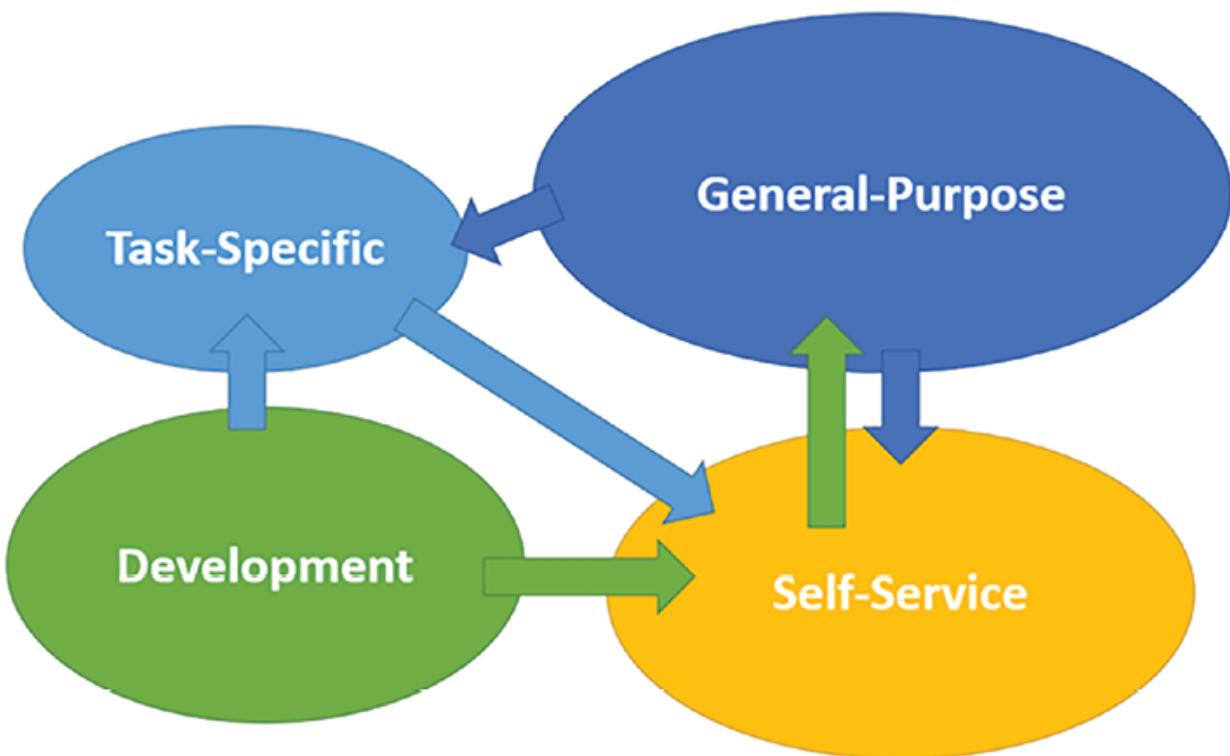
## Getting started with cutting-edge SRL

*Chapter 7, The Generative AI Revolution with ChatGPT*, introduced the complexity of adapting to the widening taxonomy of deep learning models

in general, and LLMs, such as ChatGPT, in particular.

SRL provides enhanced information extraction on each word's role in a sentence. The extracted information can improve translation, summarization, and other NLP tasks. A model that understands the semantic role of words will provide better-quality NLP outputs.

However, choosing a path to implement an NLP task has become challenging. SRL is no exception. We can sum up the issues to examine with four parameters that describe the resources used in this chapter: task-specific, general-purpose, development, and self-service:



*Figure 12.1: The complexity of LLM management*

*Figure 12.1* shows the complexity of LLM project management. The parameters interact in various possible ways for a project. The risk of each parameter must be carefully weighed, for example:

- **Task-specific:**

A task-specific model (such as a BERT-based model) must be trained to perform SRL. This implies machine resources, data management, model optimization (hyperparameters), and human expert-level resources.

- **General-purpose:**

ChatGPT can also perform specific tasks, although it is not a task-specific model.

The ChatGPT model in this chapter does not need to be trained for SRL. It might not be as precise as the task-specific model, but it is cost-effective and relatively efficient for most SRL tasks.

- **Development:**

There is a cost if development is involved with an API, for example. Development may be required to create a custom solution or a self-service.

- **Self-service:**

The revenue gained by using a self-service or custom solution may justify the cost. However, a care risk management study must precede any investment to measure the effectiveness of a customized solution. An online service such as ChatGPT Plus might prove unsatisfactory if there is a poor internet connection, or it might be enough for some projects.

There are several additional parameters to consider, such as training teams, monitoring end-user activities in a self-service configuration, connecting the outputs manually or automatically to other systems, and competing transformer models.

We have gone through some of the many possible scenarios. Real-life project management requires more parameters. For example, a BERT or GPT model can be run on a local machine or a distant server.

Ultimately, evaluating the parameters and the right choice will depend on the project.

With this in mind, we are ready to explore the syntax-free SRL approach defined by *Shi and Lin* (2019).

## Entering the syntax-free world of AI

SRL is as difficult for humans as for machines. However, transformers have taken us to the disruptive boundary of our human baselines.

A syntax-free approach to SRL is quite innovative. Classical NLP methods include dependency analysis, Part-of-Speech (POS) parsing, and learning about phrase structure. The classical approach trains the model to understand a sentence's grammatical structure and syntax.

The model designed by *Shi and Lin* (2019) doesn't apply an explicit syntax analysis process. The BERT model relies on its unique ability to understand the structure of a sentence. It implicitly captures syntactic features from the vast amount of data it learns how to represent. OpenAI took this approach further and decided to let its GPT models learn syntax without going through the tedious process of training a model for specific tasks, including SRL. As such, there are no syntax trees in a GPT model (syntax-free).

In this section, we will first define SRL and visualize an example. We will then examine how ChatGPT processes syntax analysis.

Defining the problematic task of SRL will get us started.

## Defining SRL

*Shi and Lin* (2019) advanced and proved that we can find who did what and where without depending on lexical or syntactic features. This chapter is based on *Peng Shi and Jimmy Lin*'s research at the *University of Waterloo*, California. They showed how transformers learn language structures better with attention layers.

SRL labels the *semantic role* as the role a word or group of words plays in a sentence and the relationship established with the predicate.

A *semantic role* is the role a noun or noun phrase plays in relation to the main verb in a sentence. For example, in the sentence `Marvin walked in the park`, `Marvin` is the *agent* of the event occurring in the sentence. The *agent* is the *doer* of the event. The main verb, or *governing verb*, is `walked`.

The *predicate* describes something about the subject or agent. The predicate could be anything that provides information on the features or actions of a subject. In our approach, we will refer to the predicate as the *main verb*. For example, in the sentence `Marvin walked in the park`, the predicate is `walked` in its restricted form.

The words `in the park` *modify* the meaning of `walked` and are the *modifier*. The noun or noun phrases that revolve around the predicate are *arguments* or *argument terms*. `Marvin`, for example, is an *argument* of the *predicate* `walked`.

We can see that SRL does not require a syntax tree or a lexical analysis.

Let's visualize the SRL of our example.

# Visualizing SRL

Visualizing SRL can help understand how sentences are structured.

The goal is to:

- Read and understand the concepts explained
- Take the time to understand the examples provided before we run stochastic LLMs

We will now visualize our SRL example. *Figure 12.2* is an SRL representation of `Marvin walked in the park`:



*Figure 12.2: The SRL representation of a sentence for the verb “walked”*

We can observe the following labels in *Figure 12.2*:

- **Verb:** The predicate of the sentence (**V**).
- **Argument:** An argument of the sentence, named **ARG0**.
- **Modifier:** A modifier of the sentence – in this case, a location (**ARGM-LOC**). It could have been an adverb, an adjective, or anything that modifies the predicate’s meaning.

We have defined SRL and gone through an example. Let’s run some SRL experiments.

## SRL experiments with ChatGPT with GPT-4

We will run our SRL experiments with ChatGPT with GPT-4, beginning with a basic sample and then challenging the model with a more complex example to explore the system's capacity and limits. You can access ChatGPT on OpenAI's platform: <https://chat.openai.com/>.

ChatGPT has two revolutionary features:

- It is syntax-free, meaning that it does not rely on syntax trees or rules at all. This approach is a paradigm shift from classical AI to generative models. Generative models detect statistical patterns in sequences but do not learn rules at all. The rules are implicit through statistical training, not explicit.
- The responses are not pre-designed and remain stochastic, meaning that we will get a mostly reliable (like for any AI model) output but not the same word-for-word output each time. This stochastic, random behavior makes recent LLMs so human-like.

Let's begin with a basic sample.

## Basic sample

Basic samples seem intuitively simple but can be tricky to analyze.

Compound sentences, adjectives, adverbs, and modals are difficult to identify.

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?
```

We can use the description of the arguments of the dataset based on the structure of the PropBank (Proposition Bank) on GitHub:

<https://github.com/propbank/propbank->

<documentation/blob/master/annotation-guidelines/Propbank-Annotation-Guidelines.pdf>.

The verb `think`, for example, in this excerpt can be interpreted as:

- `v` identifies the verb `think`.
- `ARG0` identifies the agent; thus, `Bob` is the agent or “pro-agent.”
- `ARGM-ADV` identifies `really` as an adverb (`ADV`), with `ARGM` meaning that the adverb provides an adjunct (not necessary) and thus is not numbered.

The first verb in the example is `Did`:

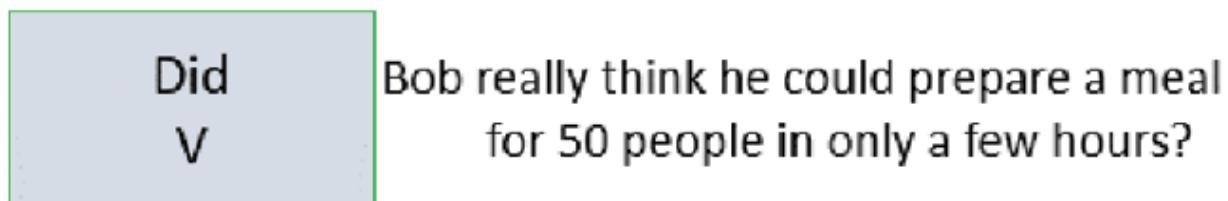


Figure 12.3: Identifying the verb “Did”

The second verb identified is `think`:



Figure 12.4: Identifying the verb “think”

If we take a close look at this representation, we can detect some interesting properties of the SRL that are challenging:

- Detecting the verb `think`.
- Avoiding the `prepare` trap that could have been interpreted as the main verb. Instead, `prepare` remained part of the argument of `think`.
- Detecting an adverb and labeling it.

The third verb is `could`:

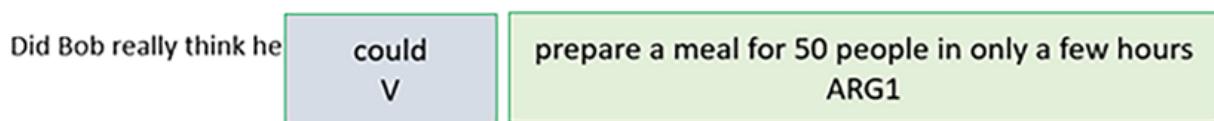


Figure 12.5: Identifying the verb “could” and the argument

Let’s move to the verb `prepare`, label it, and analyze its context:



Figure 12.6: Identifying the verb “`prepare`,” the arguments, and the modifiers

Now that we have analyzed this sentence, let’s ask ChatGPT with GPT-4 to do the same task:

```
Provide semantic role labeling for the following sentence with the
proper SRL labels: Did Bob really think he could prepare a meal for
50 people in only a few hours?
```

ChatGPT’s response is quite convincing, considering that its training is not task-specific and syntax-free:

Semantic Role Labeling (SRL) involves identifying the predicate (usually a verb) and its arguments (entities or phrases that fulfill various roles related to the predicate). Here's the SRL breakdown for the sentence: "Did Bob really think he could prepare a meal for 50 people in only a few hours?"

#### 1. Predicate: "think"

- Arg0 (Agent, Thinker): Bob - The person doing the thinking.
- ArgM-ADV (Adverbial Modifier): really - Modifies how the thinking was done.
- Arg1 (Content of Thought): [he could prepare a meal for 50 people in only a few hours] - What Bob is thinking about.

#### 2. Within the Arg1, there's another predicate: "prepare"

- Arg0 (Agent, Preparer): he (referring to Bob) - The person doing the preparing.
- Arg1 (Theme, Thing Being Prepared): a meal - The object being prepared.
- Arg2 (Beneficiary): for 50 people - For whom the meal is being prepared.
- ArgM-TMP (Temporal Modifier): in only a few hours - The time frame in which the meal is prepared.

The sentence also contains auxiliary verbs and interrogative elements, but these are typically not the focus of standard SRL annotations. The main focus is on the predicates "think" and "prepare" and their related arguments.

We can see that a cutting-edge LLM has revolutionary properties:

- **Syntax-free:** It is not task-specific and has no syntax rules or trees.
- **General-purpose:** It is not trained for a specific task.
- **Stochastic:** It is not trained to answer a request word for word each time.

Before we move on, let's examine the stochastic nature of cutting-edge LLMs that makes them so attractive: randomness. In *Chapter 7, The Generative AI Revolution with ChatGPT*, we saw that the stochastic nature of GPT models (and other leading LLMs) can be controlled with temperature, top-k, and top-p, making the model more or less creative.

To experiment with the stochastic nature of LLMs, let's submit the same request to ChatGPT again. Our goal is to see if the response is reliable, not if it repeats the same explanation word for word:

```
Provide semantic role labeling for the following sentence with the
proper SRL labels: Did Bob really think he could prepare a meal for
50 people in only a few hours?
```

ChatGPT with GPT-4 provides a reliable response again, but it is not the word-for-word mirror of the previous response:

```
Semantic Role Labeling (SRL) involves identifying the basic elements
in a sentence and their roles. Here's the SRL for your sentence:
```

1. Did Bob really think - This is the main clause.
  - Bob: Arg0 (Agent) - the person who is thinking.
  - think: Verb - the action of the sentence.
2. he could prepare a meal for 50 people in only a few hours - This is the subordinate clause, representing what Bob is thinking.
  - he: Arg0 (Agent) - Bob, the person who is preparing the meal.

- could prepare: Verb - the action of preparing.
- a meal: Arg1 (Theme) - the thing being prepared.
- for 50 people: Arg2 (Beneficiary) - the people for whom the meal is being prepared.
- in only a few hours: ArgM-TMP (Temporal Modifier) - specifies the time frame for the action.

So, the sentence breaks down into these semantic roles, showing who is doing what, to whom, and when.

We can see that the response is reliable, though not word-for-word consistent with its previous response.

We have now gone through the general-purpose, syntax-free, and stochastic nature of cutting-edge LLMs such as ChatGPT with GPT-4.

## Difficult sample

This section will run samples containing tricky structures that the transformer will have to solve or not.

Let's start with a complex sample:

```
sentence="Now, ice pucks guys!"
```

This sample illustrates a word we often think is just a noun. However, more words than we suspect can be both nouns and verbs. For example, *to ice* is a verb used in hockey to shoot a puck all the way across the rink and beyond the goal line of an opponent. A puck is a 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 in colloquial language, guys can mean persons regardless of gender.

Now let's see how ChatGPT reacts to this tricky sentence:

```
Provide semantic role labeling for the following sentence with the proper SRL labels: Now, ice pucks guys!
```

ChatGPT, with no task-specific training and its syntax-free approach, doesn't fall into the trap:

```
Semantic Role Labeling (SRL) for the sentence "Now, ice pucks guys!" involves identifying the key elements and their semantic roles. This sentence is a bit tricky due to its informal and imperative nature. Here's an attempt at SRL for this sentence:
```

1. Now - ArgM-TMP (Temporal Modifier): Specifies the timing of the action.

2. ice - Verb: This is a bit unconventional, but in this context, "ice" seems to be used as a verb in an imperative form, meaning to put or use ice pucks.

3. pucks - Arg1 (Theme): The object or the thing being acted upon, which in this case are the 'pucks'.

4. guys - Arg0 (Agent): The people who are being instructed or addressed to perform the action. In an imperative sentence like this, the subject (you) is often implied, but here "guys" is explicitly mentioned as the ones who should perform the action.

So, the sentence "Now, ice pucks guys!" is instructing the "guys" to act on the "pucks" (presumably to put them into use or in place) at

the current time.

Our LLM, ChatGPT, in this case, has done an excellent job! We can only admire the power of a model that finds syntax structures without explicitly learning them!

However, unfortunately, nothing is perfect in NLP. Let's now question the scope of SRL.

## Questioning the scope of SRL

We are alone when faced with a real-life project. We have a job to do, and the only people to satisfy are those who asked for that project.

*Pragmatism must come first – technical ideology after.*

In the 2020s, former AI ideology and new ideology coexist. By the decade's end, the two worlds will have merged to form a new era.

This section questions the productivity of SRL and its motivation through two aspects:

- The limit of predicate analysis
- Questioning the use of the term “semantic”

## The challenges of predicate analysis

SRL relies on predicates. SRL only works as long as you provide a verb. But millions of sentences do not contain verbs (ellipsis).

If you provide an LLM with an assertion alone, it works.

But what happens if your assertion is an answer to a question? The question can contain a verb, for example:

Person 1: what would you like to drink, please?

But the person who answers doesn't include a verb:

Person 2: A cup of coffee, please.

Let's ask ChatGPT with GPT-4:

Provide semantic role labeling for the following sentence with the proper SRL labels: A cup of coffee, please

ChatGPT provides a remarkable analysis:

Semantic Role Labeling (SRL) for the sentence "A cup of coffee, please" involves identifying the key components and their semantic roles. This sentence is a polite request and is somewhat elliptical, meaning it omits some elements that are understood from context.

Here's the SRL for this sentence:

1. A cup of coffee - Arg1 (Theme): This is the object or the thing being requested. It's the primary focus of the sentence.

2. please - ArgM-ADV (Adverbial Modifier): This is a politeness marker. It modifies the request, indicating that it is being made politely.

In a complete sentence, there might be an implied Arg0 (Agent) like "I" or "we" as in "[I/we would like] a cup of coffee, please," but in this case, the speaker is omitted as it's understood in the context of the request. The main verb (like "would like") is also omitted but understood.

ChatGPT has exceptional insights and analyzes this sentence, although it contains an *ellipsis*. The predicate is implicit, not explicit. SRL is based on detecting a verb, but ChatGPT found it anyway as implied, not expressed!

The definition of an ellipsis is the act of leaving one or several words out of a sentence that is not necessary for it to be understood.

Hundreds of millions of sentences containing an ellipsis are spoken and written each day.

The following question is relatively straightforward, but the answer only contains a noun:

Q: How did you get to work today?

A: Subway.

Let's make this difficult for ChatGPT and challenge it:

Provide semantic role labeling for the following sentence with the proper SRL labels. Try to guess what is missing: subway

ChatGPT deploys its power and provides an excellent response:

Semantic Role Labeling (SRL) for a single word like "subway" is challenging because SRL typically requires a full sentence to understand the roles of different elements. However, I can hypothesize about the possible roles "subway" might play in a sentence, and what elements might be missing for a complete semantic structure.

1. Subway - This could serve as either:

- Arg1 (Theme): If "subway" is the object of the sentence, it could be the thing being discussed or acted upon. For example, in "I am taking the subway," "subway" is what is being taken.

- Arg0 (Agent): In a different context, "subway" could be the subject or agent performing an action. For example, in "The subway

stops at the station," "subway" is the entity performing the action of stopping.

Missing elements in a sentence involving "subway" could include:

- Verb: An action or state, like "runs," "stops," or "is crowded."

This is essential for understanding what is happening in relation to the subway.

- Arg0 (Agent) or Arg1 (Theme): Depending on the sentence, there could be an agent performing an action (like a person using the subway) or a theme being acted upon (like the subway itself in operation).

- ArgM-LOC (Locative Modifier): Information about location, such as "through downtown" or "at the main station."

- ArgM-TMP (Temporal Modifier): Information about time, like "during rush hour" or "at 9 AM."

- ArgM-DIR (Directional Modifier): Indicating direction, such as "to the city center" or "from the airport."

For example, in a complete sentence like "The subway runs through the city," "subway" is Arg0 (Agent), "runs" is the Verb, and "through the city" is ArgM-LOC (Locative Modifier)

We could find millions more examples that are challenging to understand because the sentences do not contain predicates. In some cases, LLMs will succeed in detecting ellipses and filling in the gaps. Sometimes, a lack of context can only lead to guessing what's missing.

This section showed the challenges of SRL. Let's now redefine SRL and show how to manage it.

# Redefining SRL

SRL presupposes that sentences contain predicates, which is often a false assumption. Analyzing a sentence cannot be based on a predicate analysis alone.

A predicate contains a verb. The predicate tells us more about the subject. The following predicate contains a verb and additional information:

The dog ate his food quickly.

ate...quickly tells us more about the way the dog ate. However, a verb alone can be a predicate, as in Dogs eat.

The problem here resides in the fact that “verbs” and “predicates” are part of syntax and grammar analysis, not semantics.

Understanding how words fit together from a grammatical, functional point of view is restrictive.

Take this sentence that means absolutely nothing:

Globydisshing maccaked up all the tie.

However, SRL could perform “semantic” analysis on a sentence that means nothing:

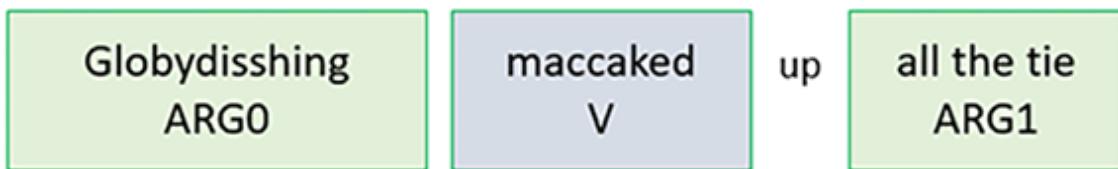


Figure 12.6: Analyzing a meaningless sentence

We can draw some conclusions from these examples:

- SRL predicate analysis only works when there is a verb in the sentence.

- SRL predicate analysis cannot identify an ellipsis.
- Predicates and verbs are part of the structure of a language, of grammatical analysis.
- Predicate analysis identifies structures but not the meaning of a sentence.
- Grammatical analysis goes far beyond predicate analysis as the necessary center of a sentence.

Semantics focuses on the meaning of a phrase or sentence. Semantics focuses on context and the way words relate to each other.

This sentence does not fool ChatGPT and provides the following explanation:

The sentence "Globydisshing maccaked up all the tie" does not make sense in English as it contains non-standard or nonsensical words ("Globydisshing", "maccaked"). Semantic Role Labeling (SRL) relies on the identification of roles within a coherent sentence structure, typically involving recognizable verbs and their arguments (such as agents, themes, instruments, etc.).

Since the sentence does not conform to recognizable English syntax or vocabulary, it's not possible to provide accurate SRL labels. The words "Globydisshing" and "maccaked" do not exist in English, and without understanding their meanings or grammatical roles, it's impossible to determine what "up all the tie" might refer to.

If these words are intended to be fictional or from a constructed language, their meanings and grammatical functions would need to be defined before SRL could be applied. If this is a sentence from a creative work or a game with its own linguistic system, providing

context or definitions for these terms would be necessary to proceed with any form of semantic analysis.

However, this does not solve our overall problem: the verb-dependent nature of SRL. The grammatical analysis includes syntax, inflection, and the functions of words in a phrase or sentence. The term semantic role labeling is misleading; it should be named “predicate role labeling.”

We perfectly understand sentences without predicates and beyond sequence structure.

Sentiment analysis can decode the meaning of a sentence and give us an output without predicate analysis. Sentiment analysis algorithms perfectly understand that “Best movie ever” is positive, regardless of the presence of a predicate or not.

Using SRL alone to analyze language is restrictive. Using SRL in an AI pipeline or with other AI tools can be very productive in adding more intelligence to natural language understanding.

Now, we will run ChatGPT.

## From task-specific SRL to emergence with ChatGPT

We have seen that OpenAI’s ChatGPT with GPT-4 has taken LLMs further for various tasks, including SRL. Thus, general-purpose LLMs do not necessarily need to learn syntax explicitly. They don’t need to learn the rules and principles of syntax that explain how to form phrases, clauses, and sentences. They can explain sentences with and beyond SRL.

We have gone through the main aspects of SRL in this chapter with several examples. This section will focus on running GPT-4 through the API to explore its ability to perform SRL without explicitly being trained for this task.

Open `Semantic_Role_Labeling_GPT-4.ipynb` in the directory of this chapter in the GitHub repository.

We will first install and import OpenAI.

## 1. Installing OpenAI

The program updates `pip` and installs OpenAI:

```
!pip install --upgrade pip
#Importing openai
try:
    import openai
except:
    !pip install openai -qq
    import openai
from openai import OpenAI
```

Now, the program retrieves the API key:

```
#2.API Key
#Store your key in a file and read it(you can type it directly).
from google.colab import drive
drive.mount('/content/drive')
f = open("drive/MyDrive/files/api_key.txt", "r")
API_KEY=f.readline()
f.close()
#The OpenAI Key
import os
```

```
os.environ['OPENAI_API_KEY'] =API_KEY  
openai.api_key = os.getenv("OPENAI_API_KEY")
```

Now, let's create an SRL dialog function for GPT-4.

## 2. GPT-4 dialog function

We will create a ChatGPT function for GPT-4. You can also try another recent model if you don't have access to GPT-4. The outputs might differ from one model to another.

We define the agent's role, explain the system role, and also the role of an assistant. We add the `uinput` user request:

```
def dialog(uinput):  
    #preparing the prompt for OpenAI  
    role="user"  
    line = {"role": role, "content": uinput}  
    #creating the message  
    assert1={"role": "system", "content": "You are a Natural La  
    assert2={"role": "assistant", "content": "You provide Seman  
    assert3=line  
    iprompt = []  
    iprompt.append(assert1)  
    iprompt.append(assert2)  
    iprompt.append(assert3)  
    client = OpenAI()  
    #sending the message to ChatGPT  
    response=openai.ChatCompletion.create(model="gpt-4",message  
    text=response["choices"][0]["message"]["content"] #response  
    return text
```

The function will send the request to `gpt-4` and return the response in the `text` variable.

We can now run SRL requests.

## 3. SRL

GPT-4 is creative! From one to another, you might see different ways it analyzed the sentence due to the stochastic nature of Generative AI models.

Focus on the logic of GPT-4's response. The results can be considered satisfactory if the tags and explanations are logical and fit your needs. If not, then a task-specific model such as a BERT model might be a better solution for your project. However, all LLMs have stochastic features, so you might still get variable responses.

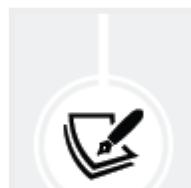
As you can see, there is no silver bullet in AI.

We will now look into GPT-4's logical yet imaginative mind with a basic sample.

### Sample 1 (basic)

`uinput` will contain each sample in this section. Then, the request is sent to the dialog function we created, which returns the response in the `text` variable. The request and response are displayed:

```
uinput="Perform Semantic Role Labeling on the following sentence:  
text=dialog(uinput) #preparing the messages for ChatGPT  
print("Viewer request",uinput)  
print("ChatGPT SRL:",text)
```



Note that the term `ChatGPT` is used in the code text. ChatGPT has become a brand that covers OpenAI Generative AI



models. In this case, the model applied is GPT-4.

ChatGPT with GPT-4 provides a logical response with explanations:

```
Viewer request Perform Semantic Role Labeling on the following
ChatGPT SRL: Here is the Semantic Role Labeling for the provide
| Argument          | Role           | Description
|-----|-----|-----
| Did              | O (Others)    | Auxiliary verb, not part of
                           structure of the sentence
| Bob              | A0 (Arg0)     | Agent, who performs the act
| really           | AM-ADV (Modifier)| Adverb to modify the verb
| think            | V (Verb)       | Main verb in the sentence (
| he               | A1 (Arg1)     | Agent of the embedded sente
                           the action (prepare)
| could prepare   | A1-V (Action) | The action being performed
                           sentence (prepare a meal)
| a meal           | A1-Theme      | Object being prepared in th
                           sentence (a meal)
| for 50 people   | A1-Goal       | The number of people the me
                           for
| in only a few hours| AM-TMP (Temporal)| The limited time fram
                           action is taking plac
Note that "A1" arguments are specific to the embedded sentence
```

You can see that the output is reliable, though quite different from the response obtained with ChatGPT with GPT-4 in the *Basic sample* section of this chapter.

We can see that a reliable LLM output is not a word-for-word output.

The following sample is trickier for a general-purpose model.

## Sample 2 (basic)

The input text is a typical everyday sentence:

```
uinput="Perform Semantic Role Labeling on the following sentence
text=dialog(uinput) #preparing the messages for ChatGPT
print("Viewer request",uinput)
print("ChatGPT SRL:",text)
```

The output seems logical, though it missed the second instance of the verb visit in the phrase: `visit the Eiffel Tower` and classified it as an argument, not a verb:

```
Viewer request Perform Semantic Role Labeling on the following sentence
ChatGPT SRL: Here is the Semantic Role Labeling for the given sentence
+-----+-----+-----+-----+
| Arg0 (Agent) | Verb       | Arg1      | Arg2      | Arg4
+-----+-----+-----+-----+
| Mrs. and Mr. Tomaso | went      |           | to Europe | for v
| Mrs. and Mr. Tomaso | visited   | Paris     |           |
| Mrs. and Mr. Tomaso | went      |           | to visit   | the Eiffel Tower
+-----+-----+-----+-----+
```

The result makes sense, but is it enough for your project? The response may be sufficient. However, bear in mind that Generative AI is stochastic. As such, the output may vary from one run to another. If the output is satisfactory on average, then the model may be enough for your project or not.

For example, can we consider that `to visit the Eiffel Tower` is an argument of `went`? After all, Mrs. and Mr. Tomosa went for something, which was a visit to the Eiffel Tower. Or is this imprecision unacceptable for your project?

We can see that emergence in Foundation Models goes with certain risks. And among these risks, there may be opportunities. Maybe this output is sufficient after all. We need to balance the risks versus the costs and goals of a project.

## Sample 3 (basic)

This is another everyday compound sentence that is challenging:

```
uinput="Perform Semantic Role Labeling on the following sentence:  
text=dialog(uinput) #preparing the messages for ChatGPT  
print("Viewer request",uinput)  
print("ChatGPT SRL:",text)
```

Once again, the response captures the general SRL ideas but doesn't analyze all the verbs. It picks the main verbs and represents the others as arguments of that verb:

```
Viewer request Perform Semantic Role Labeling on the following  
ChatGPT SRL: Here is the semantic role labeling of the sentence  
| Arg0 | Verb | Arg1 | Arg2 | Temporal | ArgM-NEG / ArgM-DIS |  
| -----|-----|-----|-----|-----|-----| -  
| John | wanted | to drink | tea | | |  
| Mary | likes | to drink | coffee | | |  
| Karim | drank | some cool water | | |  
| Faiza | would like | to drink | tomato juice | | |  
Arg0: Agent or experiencer (the person who performs or experiences the action)  
Verb: Action or event  
Arg1: Theme or patient (the entity directly affected by the action)  
Arg2: Instrument or manner (can describe how the action was performed)  
Temporal: Time-related aspects (when the action occurred, not present in this sentence)  
ArgM-NEG / ArgM-DIS: Negation or discourse (not present in this sentence)  
ArgM-DIR: Direction (not present in this sentence)
```

Is this result acceptable? Does it make sense? Should we be more flexible when dealing with SRL outputs? GPT-4 has proven that it understands language. However, is this sufficient? Once again, this will depend on the project goals.

Now, GPT-4 has to solve some difficult samples.

## Sample 4 (difficult)

The input contains some confusing traps:

input="Perform Semantic Role Labeling on the following sentence: Alice, whose husband went jogging every Sunday, liked to go to a dancing class in the meantime."

```
text=dialog(uinput) #preparing the messages for ChatGPT
print("Viewer request",uinput)
print("ChatGPT SRL:",text)
```

Once again, GPT-4 decides that the main verbs related to the agent are sufficient and considers the others as arguments:

```
Viewer request Perform Semantic Role Labeling on the following
ChatGPT SRL: Semantic Role Labeling:
Verb: "went"
+-----+-----+-----+
|ARG | Argument      | Text
+-----+-----+-----+
|A0  | Agent        | Alice's husband
|AM-MNR | Manner     | jogging
|AM-TMP | Time         | every Sunday
+-----+-----+-----+
Verb: "liked"
+-----+-----+-----+
|ARG | Argument      | Text
```

```
+---+-----+
|A0 | Experiencer | Alice
|V  | Verb        | liked
|A1 | Theme       | to go to a dancing class in the meantime
+---+-----+
Verb: "go"
+---+-----+
|ARG | Argument    | Text
+---+-----+
|A0 | Agent       | Alice
|V  | Verb         | go
|A2 | Goal         | to a dancing class
|AM-TMP | Time      | in the meantime
+---+-----+
```

Bear in mind that the response may change after each run. They might even vary if you modify the prompt.

Now, the following sample has an ambiguous word.

## Sample 5 (difficult)

The input contains `round`, which can be a verb in the past or a noun. In this sentence, `round` is used in a rather rare form, meaning `ending well`:

```
uinput="Perform Semantic Role Labeling on the following sentence: 'The round dance was a success'." text=dialog(uinput) #preparing the messages for ChatGPT print("Viewer request",uinput) print("ChatGPT Sentiment Analysis:",text)
```

The response shows that GPT-4 did not fall into the trap and detected the verb `round`:

```

Viewer request Perform Semantic Role Labeling on the following
ChatGPT Sentiment Analysis: Here's the Semantic Role Labeling f
```
The bright sun, the blue sky, the warm sand, the palm trees, ev
```
Argument      | Description
-----|-----
Arg0          | everything
V (verb)      | round off
Arg1          | the bright sun, the blue sky, the warm sand, the
The sentence can be represented as:
```
everything (Arg0) -> round off (V) -> the bright sun, the blue

```

Now, GPT-4 faces another ambiguous sample.

## Sample 6 (difficult)

The input is the same as the one in the *Difficult sample* section of this chapter:

```

uinput="Perform Semantic Role Labeling on the following sentence
text=dialog(uinput) #preparing the messages for ChatGPT
print("Viewer request",uinput)
print("ChatGPT SRL:",text)

```

In this run, GPT-4 detects the verb `to ice` in hockey jargon:

```

Viewer request Perform Semantic Role Labeling on the following
ChatGPT SRL: The sentence you provided appears to be a colloqui
Sentence: Now, ice pucks guys!
Roles:
1. Temporal Modifier: Now
2. Predicate: ice (to defeat or outplay)

```

|                  |                   |
|------------------|-------------------|
| 3. Object: pucks |                   |
| 4. Target: guys  |                   |
| Component        | Role in sentence  |
| -----            | -----             |
| Now              | Temporal Modifier |
| ice              | Predicate         |
| pucks            | Object            |
| guys             | Target            |

◀ ▶

GPT-4 might not provide the correct answer in another run due to the stochastic nature of the algorithm. This might limit the scope of an SRL task.

Let's now conclude our exploration of the scope and limits of SRL.

## Summary

In this chapter, we explored SRL. SRL tasks are difficult for both humans and machines. Transformer models have shown that human baselines can be reached for many NLP topics to a certain extent.

We first defined the revolutionary syntax-free world of recent LLM models. AI is experiencing a significant paradigm shift from task-specific training to general-purpose Generative AI models such as OpenAI's GPT series.

We ran several examples with a general-purpose model using ChatGPT with GPT-4. We confirmed that there is no silver bullet and that the ultimate choice will depend on the goals of a project.

We found that a general-purpose LLM model can perform predicate sense disambiguation. We ran basic examples in which a transformer could identify the meaning of a verb (predicate) without lexical or syntactic labeling.

We found that a transformer trained with a stripped-down *sentence* + *predicate* input could solve simple and complex problems. Challenging tasks were reached when we used relatively rare verb forms. However, these limits are not final. If complex problems are added to the training dataset, the research team could improve the model.

We saw that explaining AI is as essential as running programs such as the syntax-free stochastic nature of LLMs.

We explored the scope and limits of SRL to optimize how we will use this method with other AI tools.

Finally, we ran the GPT-4 API to solve SRL tasks. GPT-4's general-purpose capabilities produced logical results. They might be sufficient or not precise enough, depending on the project.

Transformers will continue to improve the standardization of NLP through their distributed architecture and input formats. Yet, we humans will still have to choose and implement the optimal solution for our teams and customers.

In the next chapter, *Chapter 13, Summarization with T5 and ChatGPT*, we will again face suprahuman models that raise complex implementation-level decision-making issues.

## 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 GPT-based model contains an encoder and decoder stack.  
(True/False)
8. A GPT-based SRL model has standard input formats. (True/False)
9. Transformers can solve any SRL task. (True/False)
10. ChatGPT can perform SRL better than any model. (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>

## Further reading

- *Ce Zheng, Yiming Wang, and Baobao Chang, 2022, Query Your Model with Definitions in FrameNet: An Effective Method for Frame Semantic Role Labeling:* <https://arxiv.org/abs/2212.02036>
- *Emma Strubell and Andrew McCallum, 2018, Syntax Helps ELMo Understand Semantics: Is Syntax Still Relevant in a Deep Neural Architecture for SRL?:* <https://arxiv.org/abs/1811.04773>

## Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://www.packt.link/Transformers>



# 13

## Summarization with T5 and ChatGPT

During the first seven chapters, we explored the architecture training, fine-tuning, and usage of several transformer ecosystems. In *Chapter 7, The Generative AI Revolution with ChatGPT*, we discovered that OpenAI has begun experimenting with zero-shot models that require no fine-tuning or development and can be implemented in a few lines.

The underlying concept of such an evolution relies on how transformers strive to teach a machine how to understand a language and express itself in a human-like manner. Thus, we have gone from training a model to teaching languages to machines.

ChatGPT, New Bing, Gemini, and other end-user software can summarize, so why bother with T5? Because Hugging Face T5 might be the right solution for your project, as we will see. It has unique qualities, such as task-specific parameters for summarizing.

*Raffel et al.* (2019) designed a transformer meta-model based on a simple assertion: every NLP problem can be represented as a text-to-text function. Every type of NLP task requires some text context that generates some form of text response.

A text-to-text representation of any NLP task provides a unique framework to analyze a transformer's methodology and practice. The idea is for a transformer to learn a language through transfer learning during the training and fine-tuning phases with a text-to-text approach.

*Raffel et al. (2019)* named this approach a **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 summarize documents with Hugging Face models. The examples in this chapter will be legal and medical to explore domain-specific summarization beyond simple texts. We are not looking for an easy way to implement NLP but preparing ourselves for the reality of real-life projects.

We will then compare T5 and ChatGPT approaches to summarizing. The goal is to understand that each model learns to summarize, not to assert that one model exceeds the performance of another. The choice will depend on the NLP project requirements.

Finally, we will create a notebook to summarize with the OpenAI ChatGPT GPT-4 API. The example will show the critical role of human expertise.

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
- Comparing T5 and ChatGPT models
- Implementing summarizing with the OpenAI ChatGPT GPT-4 API



With all the innovations and library updates in this cutting-edge field, packages and models change regularly. Please go to the GitHub repository for the latest installation and code examples:

<https://github.com/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition/tree/main/Chapter13>.

You can also post a message in our Discord community (<https://www.packt.link/Transformers>) if you have any trouble running the code in this or any chapter.

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 21<sup>st</sup> century in a long-overdue evolution.

*Chapter 7, The Generative AI Revolution with ChatGPT*, summed up a second revolution that boiled up and erupted between Google’s *Vaswani et al.* (2017) Original Transformer, OpenAI’s *Brown et al.* (2020) GPT-3 transformers, and now ChatGPT ’s, GPT-4 models. The Original Transformer was focused on performance to prove that attention was all we needed for NLP/NLU tasks.

OpenAI’s second revolution, through GPT-3, focused on taking transformer models from fine-tuned pretrained models to few-shot trained models that required no fine-tuning. ChatGPT with GPT-4 continued the progression that will continue to evolve. The second revolution was to show that a machine can learn a language and apply it to downstream tasks, surprisingly mimicking humans.

It is essential to perceive those two revolutions to understand what the 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 like OpenAI about how transformers could be perceived beyond technical considerations and have taken them to an abstract level of NLU.

These revolutions became disruptive, cumulating in the arrival of mainstream Generative AI with the release of ChatGPT on November 30, 2022. It was time to settle down, forget about source code and machine resources, and analyze transformers at a higher level.

*Raffel et al.* (2019) designed a conceptual text-to-text model and then implemented it.

Let’s go through this representation of the second transformer revolution: abstract models.

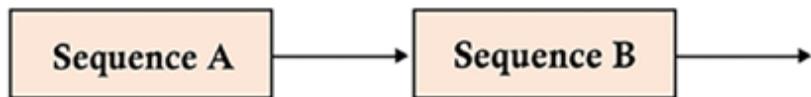
# The rise of text-to-text transformer models

*Raffel et al.* (2019) set out on a journey as pioneers with one goal: *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. The Google team working on this approach emphasized that it would not modify the Original Transformer's fundamental architecture from the start.

At that point, *Raffel et al.* (2019) wanted to focus on concepts, not techniques. Therefore, they showed no interest in producing the latest transformer model, as we often see a so-called silver bullet transformer model with  $n$  parameters and layers. This time, the T5 team wanted to find out how good transformers could be at understanding a language.

Humans learn a language and then apply that knowledge to a wide range of NLP tasks through transfer learning. The core concept of a T5 model is to find an abstract model that can do things like us. Remember, transformers learn to reproduce human-like responses through statistical pattern discovery.

When we communicate, we always start with a sequence (A) followed by another sequence (B). B, in turn, becomes the start sequence leading to another sequence, as shown in *Figure 13.1*:



*Figure 13.1: 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 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 approach defines the mechanism of the attention layers of transformers.

Take a few seconds and let this sink in. It seems deceptively simple, right? Yet, it took 35+ years to topple the old beliefs surrounding RNNs, CNNs, and the thought process accompanying them! *Chapter 1, What Are Transformers?*, showed the incredible evolution of LLMs, particularly in the *A brief journey from recurrent to attention* section.

It is quite fascinating to watch T5 learn, progress, and even help us think better sometimes!

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

The T5 model can be summed up as a **Text-To-Text Transfer Transformer**. Thus, every NLP task is expressed as a text-to-text problem to solve.

T5 models, unlike many other models, do not need to be fine-tuned for a specific task. The process of every NLP task is a text-to-text problem.

This flexible text-to-text approach leads us to examine the use of prefixes in T5 models.

## 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 in one text-to-text format.

The Google T5 team devised a simple solution: adding a prefix to an input sequence. We would need thousands of additional vocabularies in many languages without the invention of the *prefix* by some long-forgotten genius. For example, we would need to find words to describe prepayment, prehistoric, Precambrian, and thousands of other words if we did not use “pre” as a prefix.

*Raffel et al.* (2019) proposed adding a *prefix* to an input sequence. A T5 prefix is not just a tag or indicator like `[CLS]` for classification in some transformer models. Instead, a T5 prefix contains the essence of a task a transformer needs to solve. A prefix conveys meaning as in the following examples, among others:

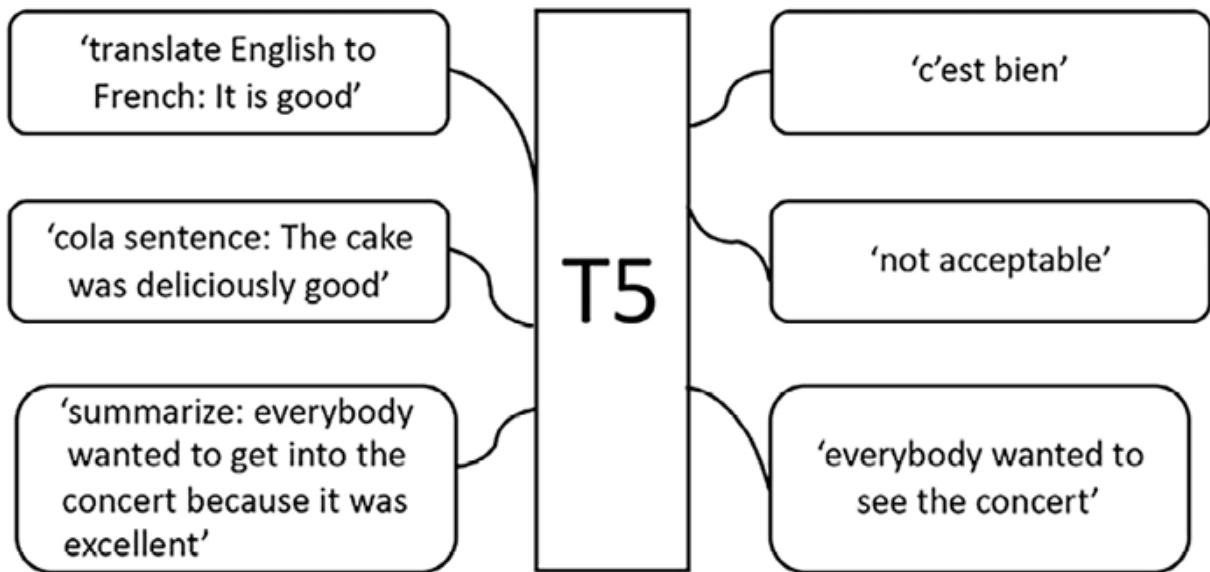
- `translate English to German: + [sequence]` for translations, as we did in *Chapter 4, Advancements in Translations with Google Trax, Google Translate, and Gemini*.
- `cola sentence: + [sequence]` for **The Corpus of Linguistic Acceptability (CoLA)**, as we used in *Chapter 5, Diving into Fine-Tuning through BERT*, when we fine-tuned a BERT transformer model.
- `stsbs sentence 1:+[sequence]` for semantic textual similarity benchmarks. Natural language inferences and entailment are similar problems, as described in *Chapter 3, Emergent vs Downstream Tasks: The Unseen Depths of Transformers*.
- `summarize + [sequence]` for text summarization problems, as we will solve in the *Text summarization with T5* section of this chapter.

We've now obtained a unified format for a wide range of NLP tasks, expressed in *Figure 13.2*:



*Figure 13.2: Unifying the input format of a transformer model*

The unified input format leads to a transformer model that produces a result sequence no matter which problem it has to solve in the **T5**. The input and output of many NLP tasks have been unified, as shown in *Figure 13.3*:



*Figure 13.3: 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 explored translations in *Chapter 4, Advancements in Translations with Google Trax, Google Translate, and Gemini*.

We fine-tune a model on **The Corpus of Linguistic Acceptability (CoLA)** in *Chapter 5, Diving into Fine-Tuning through BERT*.

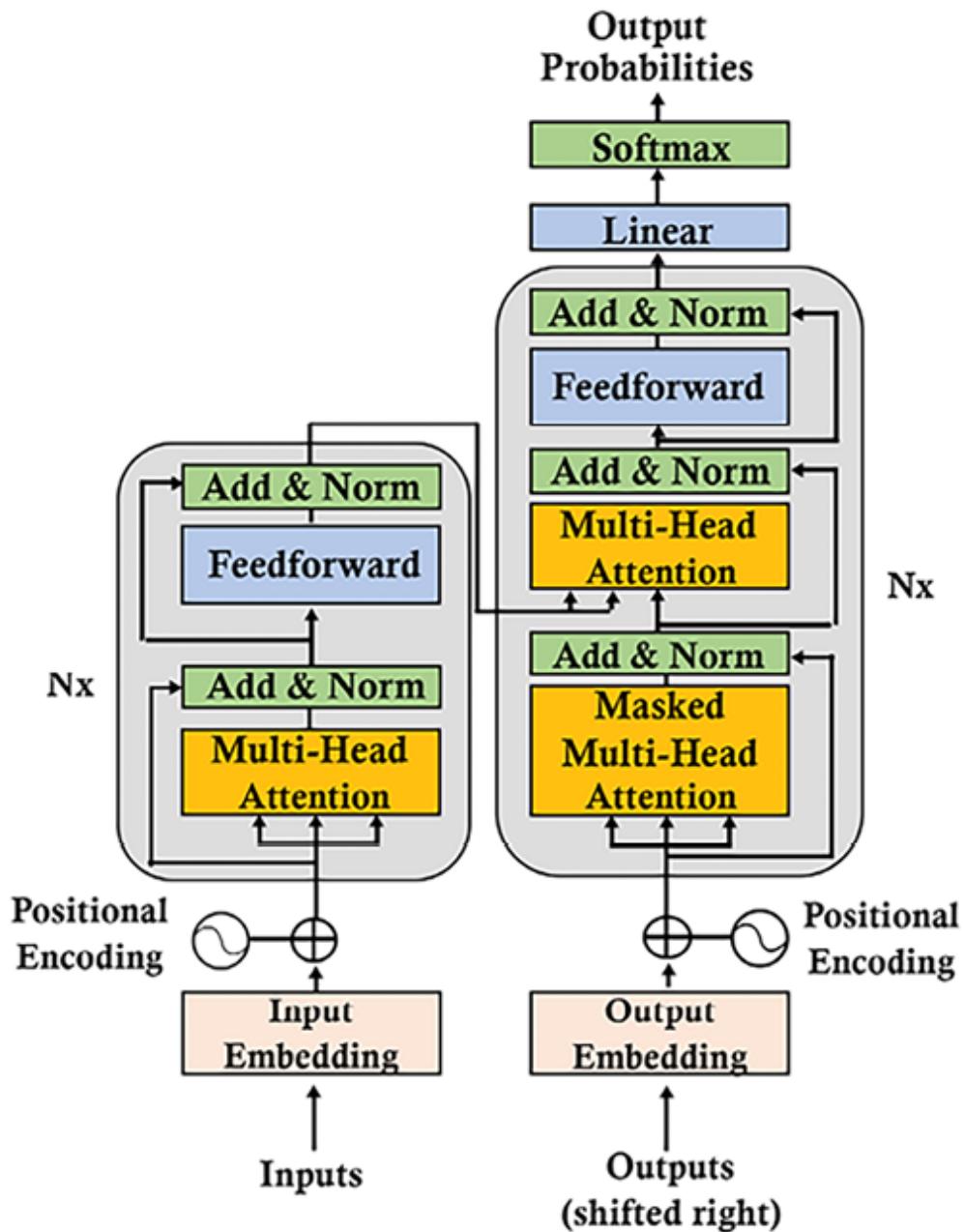
In this chapter, we will apply T5 to summarization tasks.

We have gone through the standard text-to-text input-output format. Let's now look at the architecture of the T5 transformer model.

## The T5 model

Raffel et al. (2019) focused on designing a standard input format to obtain text output. The Google T5 team did not want to try new architectures derived from the Original Transformer, such as BERT-like encoder-only layers or GPT-like decoder-only layers. Instead, the team focused on defining NLP tasks in a standard format.

They chose to use the Original Transformer model we defined in *Chapter 2, Getting Started with the Architecture of the Transformer Model*, as we can see in *Figure 13.4*:



*Figure 13.4: 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 vocabulary and functional changes. The following list contains some of the main aspects of the T5 model:

- The encoder and decoder remain in the model. The encoder and decoder layers become “blocks,” and the sublayers become “subcomponents” containing a self-attention layer and a feedforward network. The use of the word “blocks” and “subcomponents” in a LEGO®-like language allows you to assemble “blocks,” pieces, and components to build your model. Transformer components are standard building blocks you can assemble in many ways. Once you understand the basic building blocks we went through in *Chapter 2, Getting Started with the Architecture of the Transformer Model*, you can understand any transformer model.
- Self-attention is “order-independent,” meaning it performs operations on sets, as we saw in *Chapter 2*. Self-attention uses dot products of matrices, not recurrence. It explores the relationship between each word and the others in a sequence. Positional encoding is added to the word’s embedding before making the dot products.
- The Original Transformer applied sinusoidal and cosine signals to the transformer. Or, it used learned position embeddings. T5 uses relative position embeddings instead of adding arbitrary positions to the input. In T5, positional encoding relies on an extension of self-attention to make comparisons between pairwise relationships. For example, in “the cat chased the mouse,” the word “cat” is three positions away from “mouse.” A relative position captures the position of the word in its relationship to another word. The absolute position captures a word’s position in a sequence( position 1, 2, or x), as we saw in *Chapter 2*. For more, see *Shaw et al. (2018)* in the *References* section of this chapter.
- Positional embeddings are shared and re-evaluated through all the layers of the model.

We have defined the standardization of the input of the T5 transformer model through the text-to-text approach.

Let's now use **T5** to summarize documents.

## Text summarization with T5

NLP summarizing tasks extract succinct parts of a text. This section will start by presenting the Hugging Face resources we will use in this chapter. Then, we will initialize a T5-large transformer model. Finally, we will see how to use T5 to summarize any document, including legal and corporate documents.

Let's begin by introducing Hugging Face's framework.

## Hugging Face

Hugging Face designed a framework to implement transformers at a higher level. For example, we already used Hugging Face to fine-tune a BERT model in *Chapter 5, Diving into Fine-Tuning Through BERT*, and train a RoBERTa model in *Chapter 6, Pretraining a Transformer from Scratch through RoBERTa*.

This chapter will use Hugging Face's framework again to implement a T5-large model.

## Selecting a Hugging Face transformer model

In this subsection, we will choose the T5 model we will implement in this chapter.

A wide range of models can be found on the Hugging Face models page, as we can see in *Figure 13.5*:

The screenshot shows the 'Models' section of the Hugging Face website. At the top left is the 'Models' tab, and at the top right is a 'Filter by name' search bar. Below the tabs, there is a list of four models, each in its own card:

- facebook/dino-vitb16**  
by AI · Updated 17 days ago · 87.9M · 85
- jonatasgrosman/wav2vec2-large-xlsr-53-english**  
by Person · Updated Mar 25 · 64.4M · 143
- bert-base-uncased**  
by GitHub · Updated 11 days ago · 55.5M · 890
- xlm-roberta-large**  
by GitHub · Updated Apr 6 · 23.3M · 151

*Figure 13.5: Hugging Face models*

On this page, <https://huggingface.co/models>, we can search for a T5 model.

In our case, we are looking for **t5-large**, a model we can smoothly run in Google Colaboratory.

We first type `t5` to search for a T5 model and obtain a list of T5 models we can choose from:

Models 1,978

T5

X

### t5-small

Translation • Updated Jun 23, 2021 • ↓ 891k • ❤ 5

### t5-base

Translation • Updated Jun 23, 2021 • ↓ 577k • ❤ 31

### deep-learning-analytics/wikihow-t5-small

Summarization • Updated Sep 9, 2020 • ↓ 228k

### sberbank-ai/ruT5-base

Text2Text Generation • Updated Sep 21, 2021 • ↓ 166k • ❤ 1

Figure 13.6: Searching for a T5 model

When we search the site, we can see that several of the T5 transformers are available, among which are:

- **base**, which is the baseline model. It was designed to be similar to the BERT<sub>BASE</sub> with 12 layers and around 220 million parameters.
- **small**, which is a smaller model with 6 layers and 60 million parameters.
- **large** is designed to be similar to BERT<sub>LARGE</sub> with 12 layers and 770 million parameters.
- **3B** and **11B** use 24-layer encoders and decoders with around 2.8 billion and 11 billion parameters.

For more on the description of BERT<sub>BASE</sub> and BERT<sub>LARGE</sub>, you can take a few minutes now or later to review these models in *Chapter 5, Diving into Fine-Tuning through BERT*.

For more on T5 models and Google research on T5 models:

[https://huggingface.co/docs/transformers/main/en/model\\_doc/t5#transformers.T5Model](https://huggingface.co/docs/transformers/main/en/model_doc/t5#transformers.T5Model).

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

#### How to use from the 😊/transformers library

```
from transformers import AutoTokenizer, AutoModelWithLMHead
tokenizer = AutoTokenizer.from_pretrained("t5-large")
model = AutoModelWithLMHead.from_pretrained("t5-large")
```

 Copy

Figure 13.7: How to use a Hugging Face model

Figure 8.7 shows how to use the model in our code. We can also look into the list of files in the model and the basic configuration file. We will look into the configuration file when we initialize the model in this chapter's *Initializing the T5-large transformer model* section.

Let's start by initializing the T5 transformer model.

## Initializing the T5-large transformer model

In this subsection, we will initialize a T5-large model. Open the following notebook, `Summarizing_Text_T5.ipynb`, which you will find in the directory of this chapter on GitHub.

Let's get started with T5!

## Getting started with T5

In this subsection, we will install Hugging Face's framework and then initialize a T5 model.

We will first install Hugging Face's transformers:

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



**Note:** Hugging Face transformers continually evolve, updating libraries and modules to adapt to the market. If the default version doesn't work, you might have to pin one with  
`!pip install transformers==[version that runs with the other functions in the notebook].`

We pinned version `0.1.94` of `sentencepiece` to keep the notebook using Hugging Face as stable as possible:

```
!pip install sentencepiece==0.1.94
```

`sentencepiece` will split a text into separate sentences. It breaks a paragraph or document into units of sentences. If necessary, review the *Sentence and word tokenization* section of *Chapter 10, Investigating the Role of Tokenizers in Shaping Transformer Models*.

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=True
```

If we set `display_architecture` to `True`, the structure of the encoder layers, decoder layers, and feedforward sublayers will be displayed. The feature provides insights into the architecture of the model.

The program now imports `torch` and `json`:

```
import torch
import json
```

*Working on transformers means being open to the many transformer architectures and frameworks that research organizations share with us.* Also, I recommend using PyTorch and TensorFlow as much as possible to get used to both environments. What matters is the level of abstraction of the transformer model (specific-task models or zero-shot models) and its overall performance.

Let's import the tokenizer, generation, and configuration classes:

```
from transformers import T5Tokenizer, T5ForConditionalGeneration
```

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')
```

The `t5-large` model we downloaded has 770 million parameters. The number of parameters may change if Hugging Face updates or changes the model.

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 defines `"task_specific_params"`, which is a characteristic of T5:

- `"early_stopping":true`: Text generation will stop when the model predicts the end-of-sentence token, which can optimize machine usage.
- `"length_penalty": 2.0`: This parameter impacts the score of the beam search. If the value exceeds 1, the model is encouraged to produce longer sequences. If not, the model will generate shorter sequences.

- `"max_length": 200` and `"min_length": 30` control the number of tokens generated. The model will generate tokens to at least `min_length` and stop when it reaches `max_length`.
- `"no_repeat_ngram_size": 3` controls the repetition of ngrams of tokens. The parameter is set to 3, limiting the repetition of 3-grams.
- Implements the *beam search* algorithm, which will expand the four most significant text completion predictions.
- `"num_beams": 4`: If the value is set to 1, the algorithm is greedy, meaning it will only search for one prediction. If the value is set to more than one, the width of the beam will expand, and the model will search for more possibilities.
- `"prefix": "summarize:"`: Defines the task; in this case, summarizing.

Another interesting parameter is the vocabulary size:

```
"vocab_size": 32128
```

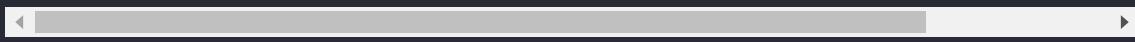
Vocabulary size is a topic in itself. Too much vocabulary will lead to sparse representations. On the other hand, too little vocabulary will distort the NLP tasks. The choice of vocabulary size will have a critical impact on the training and implementation of a model. If necessary, take the time to go back to *Chapter 10, Investigating the Role of Tokenizers in Shaping Transformer Models*.

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,  
                (k): Linear(in_features=1024, out_features=1024,  
                (v): Linear(in_features=1024, out_features=1024,  
                (o): Linear(in_features=1024, out_features=1024,  
            )  
            (layer_norm): T5LayerNorm()  
            (dropout): Dropout(p=0.1, inplace=False)  
        )  
        (1): T5LayerFF(  
            (DenseReluDense): T5DenseReluDense(  
                (wi): Linear(in_features=1024, out_features=4096,  
                (wo): Linear(in_features=4096, out_features=1024,  
                (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 `1024` features for the attention sublayers and `4096` for the inner calculations of the feedforward network sublayer that will produce outputs of `1024` features. The symmetrical structure of transformers is maintained through all of the layers.

You can take a few minutes to go through the encoder stacks, the decoder stacks, the attention sublayers, and the feedforward sublayers.

You can also choose to select a specific aspect of the model by only running the cells you wish:

```
if display_architecture==True:  
    print(model.encoder)
```

The output will display the structure of the encoder:

```
T5Stack(  
    (embed_tokens): Embedding(32128, 1024)  
    (block): ModuleList(  
        (0): T5Block(  
            (layer): ModuleList(  
                (0): T5LayerSelfAttention(  
                    (SelfAttention): T5Attention(  
                        (q): Linear(in_features=1024, out_features=1024, bi  
                        (k): Linear(in_features=1024, out_features=1024, bi  
                        (v): Linear(in_features=1024, out_features=1024, bi  
                        (o): Linear(in_features=1024, out_features=1024, bi  
                        (relative_attention_bias): Embedding(32, 16)  
                    )  
                    (layer_norm): T5LayerNorm()  
                    (dropout): Dropout(p=0.1, inplace=False)  
                )  
            )  
        )  
    )
```

Let's peek into the architecture of the decoder:

```
if display_architecture==True:  
    print(model.decoder)
```

The output will display the structure of the decoder:

```
5Stack(
  (embed_tokens): Embedding(32128, 1024)
  (block): ModuleList(
    (0): T5Block(
      (layer): ModuleList(
        (0): T5LayerSelfAttention(
          (SelfAttention): T5Attention(
            (q): Linear(in_features=1024, out_features=1024, bias=True)
            (k): Linear(in_features=1024, out_features=1024, bias=True)
            (v): Linear(in_features=1024, out_features=1024, bias=True)
            (o): Linear(in_features=1024, out_features=1024, bias=True)
            (relative_attention_bias): Embedding(32, 16)
          )
          (layer_norm): T5LayerNorm()
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
    )
  )
)
```

Let's peek into the architecture of the feedforward network:

```
if display_architecture==True:
    print(model.forward)
```

The output will display the feedforward network:

```
...
(1): T5LayerFF(
  (DenseReluDense): T5DenseActDense(
    (wi): Linear(in_features=1024, out_features=4096,
    (wo): Linear(in_features=4096, out_features=1024,
    (dropout): Dropout(p=0.1, inplace=False)
    (act): ReLU()
  )
...
)
```

Note that the dimensions of the feedforward network `wi` are increased from `1024` to `4096` dimensions to enhance the complexity of the representations. Then, they are reduced to `1024` for the output of `wo`.

Take some time, if necessary, to go through the configuration of the transformer. You will be able to understand the architecture of transformers in depth.

We have initialized the T5 transformer. Let's now summarize some documents.

## Summarizing documents with T5-large

In this section, we will create a summary function you can call with any text you wish to summarize. We will summarize legal and financial examples. They are more challenging than non-specific domain texts.

We add Python `textwrap` to the code to wrap the text we want to display to generate paragraphs:

```
import textwrap
```

Now, 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):
```

The `text.strip` function can strip the text of characters we don't need, such as:

- Space (`\s`)
- Tab (`\t`)
- Newline (`\n`)
- Carriage return (`\r`)

In this case, the context text is stripped of the newline character:

```
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) with the wrapper:

```
wrapped_t5_prepared_Text = textwrap.fill(t5_prepared_Text, w  
print(wrapped_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:

```
Text to summarize: We hold these truths to be self-evident, tha  
men are created equal,that they are endowed by their Creator wi  
certain unalienable Rights,that among these are Life, Liberty,  
pursuit of Happiness.That to secure these rights, Governments a
```

The `summarize` prefix indicates the task to solve.

The text is now encoded to token IDs and returns them as torch tensors:

```
tokenized_text = tokenizer.encode(t5_prepared_Text, return_ten
```

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  
# 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

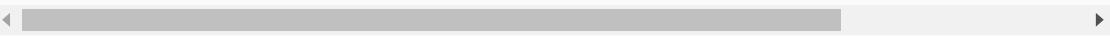
In this subsection, we will run a text published by *Project Gutenberg* through the T5 model. We will use the sample to run a test on our summarizing function. You can copy and paste any other text you wish or load a text by adding code. You can also load a dataset of your choice and call the summaries in a loop.

The program's goal in this chapter is to run a few samples to see how T5 works. The input text is the beginning of the *Project Gutenberg* e-book containing the *Declaration of Independence of the United States of America*:

```
text="""  
We hold these truths to be self-evident, that all men are created equal,  
that they are endowed by their Creator with certain unalienable
```

that among these are Life, Liberty, and the pursuit of Happiness. That to secure these rights, Governments are instituted among Men deriving their just powers from the consent of the governed, That whenever any Form of Government becomes destructive of these ends, it is the Right of the People to alter or to abolish it, and to institute a new Government, laying its foundation on such principles and organizing its powers in such form, as to them shall seem most likely to effect their Safety and Happiness. Prudence, indeed, will dictate that governments long established should not be changed for light and transient causes; and accordingly all experience hath shown, that mankind are more disposed to suffer, while evils are sufferable, than to right themselves against the forms to which they are accustomed. But when a long train of abuses and usurpations, pursuing invariably the same Object evinces a design to subject them under absolute Despotism, it is their right, it is their duty, to throw off such Government, and to provide new Guards for their future safety. --Such has been the patient sufferance of these Colonies; and it will be the necessity which constrains them to alter their former System. The history of the present King of Great Britain is a history of repeated injuries and usurpations, all having in direct object the establishment of an absolute Tyranny over these States. To prove this, let me now submit to a candid world.

"""



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)
```

The output is wrapped:

```
wrapped_summary = textwrap.fill(summary, width=70)
print ("\nSummarized text: \n", wrapped_summary)
```

The output shows we sent 534 characters, the original text, and the summary:

```
Number of characters: 1632
Text to summarize: We hold these truths to be self-evident, tha
Summarized text:
we hold these truths to be self-evident that all men are creat
equal,that they are endowed by their Creator with certain unali
Rights. that whenever any Form of Government becomes destructiv
these ends,
```

Note that some postprocessing might be necessary to detect incomplete sentences. We might have to increase the maximum length and then detect incomplete sentences.

Let's now use T5 for a more difficult summary.

## The Bill of Rights sample

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

```
#Bill of Rights,V
text ="""
No person shall be held to answer for a capital, or otherwise . unless on a presentment or indictment of a Grand Jury, except . in the land or naval forces, or in the Militia, when in actual in time of War or public danger; nor shall any person be subject to 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)
```

```
wrapped_summary = textwrap.fill(summary, width=70)
print ("\nSummarized text: \n", wrapped_summary)
```

Remember that transformers are stochastic algorithms, so the output might vary each time you run one. That being said, we can see that T5 did not really summarize the input text but simply shortened it:

```
Number of characters: 591
Preprocessed and prepared text:
    summarize: No person shall be held to answer..
Summarized text:
    no person shall be held to answer for a capital, or otherwise
    crime, unless ona presentment or indictment ofa Grand Jury. nor
    any person be subject for the same offense to be twice put
```

This sample is significant because it shows the limits any transformer model or other NLP model faces when provided with a text like this one. We cannot just present samples that always work and make users believe that transformers have solved all of the NLP challenges we face, no matter how innovative they are. We still need to process the result and find solutions such as detecting incomplete sentences, increasing the maximum length, and other classical data processing methods.

Maybe we should have provided a longer text to summarize, used other parameters, used a larger model, or changed the structure of the T5 model. However, no matter how hard you try to summarize a complex text with an NLP model, you will always find documents that the model fails to summarize.

When a model fails on a task, we must be humble and admit it. 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 much progress.

*Raffel et al. (2018) chose an appropriate title to describe their approach to T5: Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer.*

Take the necessary time to experiment with examples of your own that you find in your legal documents. Explore the limits of transfer learning as a modern-day NLP pioneer! Sometimes, you will discover exciting results, and sometimes you will find areas that need improvement.

Now, let's try a corporate law sample.

# A corporate law sample

Corporate law contains many legal subtleties, making summarizing tasks quite tricky.

The input of this sample is an excerpt of the corporate law in the state of Montana, USA:

The result is relatively satisfying:

```
Number of characters: 1816
Preprocessed and prepared text:
    summarize: The law regarding the corporation prescribes that a
Summarized text:
    a corporation can be incorporated in the state of Montana to s
lawful purpose. it can sue and be sued in its corporate name, a
has perpetual succession. the name of the corporation must cont
word "corp
```

This time, T5 found some of the essential aspects of the text to summarize. Take some time to incorporate samples of your own to see what happens. Play with the parameters to see if they affect the outcome. For example, you can modify the maximum length of the summary (increase or decrease):

```
increase: summary=summarize(text, 100)
decrease: summary=summarize(text, 40)
```

You can also try other Hugging Face models.

We have implemented T5 to summarize texts. There are many other T5 models on Hugging Face among the hundreds available. For example, you can go through the model card for FLAN-T5 XL,

<https://huggingface.co/google/flan-t5-xl>

You have gone through the implementation of a T5 model, so implementing FLAN-T5 XL for translations (or many other tasks) or any other model with Hugging Face is quite seamless:

1. We simply install transformers, accelerators (tools to help run on GPUs, TPUs, and other hardware accelerators), and sentencepiece (a language-agnostic tokenizer):

```
!pip install transformers -qq  
!pip install accelerate -qq  
    !pip install sentencepiece -qq
```

2. Now, we only have to define the tokenizer and the model:

```
from transformers import T5Tokenizer, T5ForConditionalGeneration  
tokenizer = T5Tokenizer.from_pretrained("google/flan-t5-xl")  
model = T5ForConditionalGeneration.from_pretrained("google/flan-t5-xl")
```

3. We can now run a translation task, for example:

```
input_text = "translate English to German: How old are you"  
input_ids = tokenizer(input_text, return_tensors="pt").input_ids  
outputs = model.generate(input_ids)  
print(tokenizer.decode(outputs[0]))
```

4. The output is:

```
Wie alt sind Sie?
```

Admire the seamless functionality of Hugging Face's platform, which opens the door to hundreds of models we can implement in a standardized way.

OpenAI offers powerful options as well. We will now experiment with summarization with OpenAI GPT-4.

# From text-to-text to new word predictions with OpenAI ChatGPT

The choice between T5 and ChatGPT (GPT-4) to perform summarization will always remain yours, depending on the project you implement. Hugging Face T5 offers many advantages with its text-to-text approach. ChatGPT has proven its efficiency. Ultimately, the requirements of a project will determine which model you will decide to use.

In this section, we will first compare some key points of each model. Then, we will create a program to summarize text with ChatGPT.

## Comparing T5 and ChatGPT's summarization methods

This section aims to compare T5 and ChatGPT's summarization methods, not their performances, which depend on factors you will have to evaluate: datasets, hyperparameters, the scope of the project, and other project-level considerations.

In this section, the term “T5” refers to the T5 models described in the *Selecting a Hugging Face transformer model* section. The term ChatGPT refers to OpenAI’s generative models, such as OpenAI GPT-3.5 and OpenAI GPT-4.

We will go through two key points:

### Pretraining

T5’s training method is text-to-text. T5 reframes every task as a text problem. It adds a prefix as a direction to the string to progress, such as

“summarize: The dog was playing in the garden.”

ChatGPT’s training method is a complete approach: the task is to predict the next token of a sequence. Summarization will be based on the context of the text, with any instructions added to the prompt. It then generates the summary to continue the sequence of the dialog initiated by the prompt.

## Specific versus non-specific tasks

T5 focuses on text-to-text specific tasks, as we saw in this chapter, using task-specific parameters:

```
"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: "  
    },
```

ChatGPT does not require specific task parameters for summarization as for many other specific tasks, though it is not a task-specific model. It remains a completion model that continues a sequence based on its understanding of language, the context, and the instructions provided.

The following section implements the summarization of ChatGPT with OpenAI’s API.

## Summarization with ChatGPT

The goal of this section is not to show that ChatGPT performs better than Hugging Face's T4 models for summarization. We just need to experiment with another summarizing paradigm that uses content generation.

Open `Summarizing_ChatGPT.ipynb` in the GitHub directory of this chapter.

First, the program manages the installation of OpenAI:

```
#Importing openai
try:
    import openai
    from openai import OpenAI
except:
    !pip install openai -qq
    import openai
    from openai import OpenAI
```

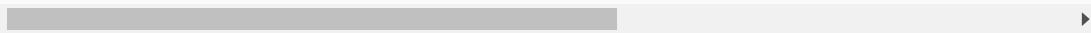
The API key is retrieved:

```
#2.API Key
#Store you key in a file and read it(you can type it directly .
from google.colab import drive
drive.mount('/content/drive')
f = open("drive/MyDrive/files/api_key.txt", "r")
API_KEY=f.readline()
f.close()
#The OpenAI Key
import os
os.environ['OPENAI_API_KEY'] =API_KEY
openai.api_key = os.getenv("OPENAI_API_KEY")
```

We create a dialog, as we went through in *Chapter 7, The Generative AI Revolution with ChatGPT*:

```
def dialog(uinput):
    #preparing the prompt for OpenAI
    role="user"

    line = {"role": role, "content": uinput}
    #creating the message
    assert1={"role": "system", "content": "You are a Natural La}
    assert2={"role": "assistant", "content": "Summarize the tex}
    assert3=line
    iprompt = []
    iprompt.append(assert1)
    iprompt.append(assert2)
    iprompt.append(assert3)
    #sending the message to ChatGPT
    client = OpenAI()
    response=client.chat.completions.create(model="gpt-4", messag}
    text=response.choices[0].message.content #property of respon}
    return text
```



And now, surprise! The example is not some easy mainstream social network text. We will ask ChatGPT to summarize a domain-specific text. In this case, a medical paper:

Multi-Cue Kinetic Model with Non-Local Sensing for Cell Migrat.



The text to summarize challenges both ChatGPT and an AI professional who is not a medical **Subject Matter Expert (SME)**. Also, medical professionals are not experts in every medical field.

*The goal here is to show that humans remain critical regarding quality control!*

Read this text before running the program. Let the reality of real-life implementations sink in:

```
Cell migration is a fundamental mechanism in a huge variety of  
In many pathological and physiological processes, there are se'  
Biological Background  
"The coexistence of chemotaxis and contact guidance happens in  
In tumor spread and metastasis formation, cancer cells follow
```

```
import textwrap
```

We provide an introduction and the text to summarize:

```
input_text="Summarize the following paragraph:..."
```

In this case, it will be the medical text we just went through.

Now, we implement the API request as we saw in *Chapter 7, The Generative AI Revolution with ChatGPT*:

```
uinput=input_text  
text=dialog(uinput) #preparing the messages for ChatGPT and ma  
wrapped_request = textwrap.fill(uinput, width=70)  
print("Request ",wrapped_request)
```

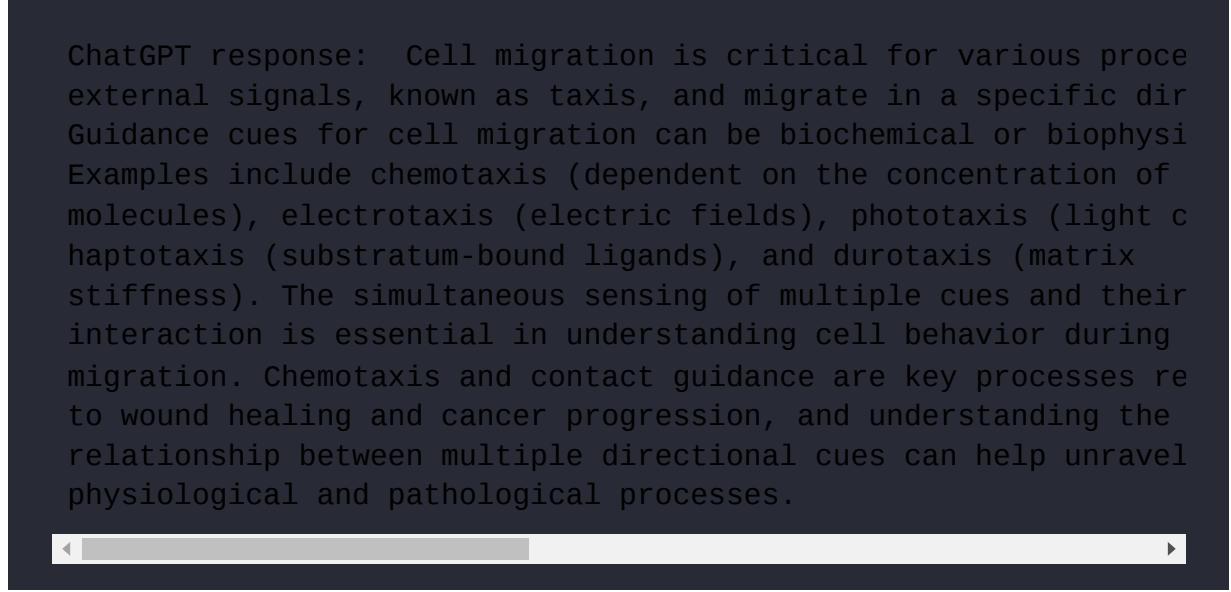
The output is the instruction to summarize followed by the text to summarize.

We add the code to print the response:

```
wrapped_response = textwrap.fill(text, width=70)
print("ChatGPT response: ",wrapped_response)
```

And now, we see a summary that only a medical professional specialized in this field can confirm!

The response seems logical:



```
ChatGPT response: Cell migration is critical for various processes. Cells respond to external signals, known as taxis, and migrate in a specific direction. Guidance cues for cell migration can be biochemical or biophysical. Examples include chemotaxis (dependent on the concentration of molecules), electrotaxis (electric fields), phototaxis (light cues), haptotaxis (substratum-bound ligands), and durotaxis (matrix stiffness). The simultaneous sensing of multiple cues and their interaction is essential in understanding cell behavior during migration. Chemotaxis and contact guidance are key processes relevant to wound healing and cancer progression, and understanding the relationship between multiple directional cues can help unravel physiological and pathological processes.
```

Though the summary seems acceptable, we AI professionals cannot confirm it is without medical training in the field the article relates to!

The good news is that summarizing complex domain-specific text is possible, and also that only human experts can provide feedback on such NLP tasks. We don't know for how long, but automation in a specialized field requires human control. This will create new functions for data science and AI professionals.

We will summarize our chapter manually as humans and move on to the next exploration – transformers!

## Summary

In this chapter, we saw how the T5 transformer models standardized the input of the encoder and decoder stacks of the Original Transformer. The Original Transformer architecture has an identical structure for each block (or layer) of the encoder and decoder stacks. However, the Original Transformer did not have a standardized input format for NLP tasks.

*Raffel et al.* (2018) designed a standard input for a wide range of NLP tasks by defining a text-to-text model. They added a prefix to an input sequence, indicating the NLP problem type to solve. This led to a standard text-to-text format. The **Text-To-Text Transfer Transformer (T5)** was born. 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).

We then compared the method applied to summarizing by T5 and ChatGPT. The motivation was not to show that one model is better than the other but to understand their differences.

Finally, we implemented the OpenAI ChatGPT GPT-4 API to examine a summary. The text to summarize was a domain-specific medical text. The

summary seemed logical, but only a **SME** in the medical field would know.

The chapter showed that human expertise remains mandatory for selecting suitable datasets, models, and implementation. Human expertise provides critical quality control of the outputs of transformer models in specific domains.

In the next chapter, *Exploring Cutting-Edge LLMs with Vertex AI and PaLM 2*, we will run NLP tasks beginning with a question and answer task to evaluate how transformers understand text beyond summarization.

## Questions

1. T5 models only have encoder stacks like BERT models. (True/False)
2. T5 models have both encoder and decoder stacks. (True/False)
3. T5 models use relative positional encoding, not absolute positional encoding. (True/False)
4. Text-to-text models are only designed for summarization. (True/False)
5. Text-to-text models apply a prefix to the input sequence that determines the NLP task. (True/False)
6. T5 models require specific hyperparameters for each task. (True/False)
7. One of the advantages of text-to-text models is that they use the same hyperparameters for all NLP tasks. (True/False)
8. T5 transformers do not contain a feedforward network. (True/False)
9. Hugging Face is a framework that makes transformers easier to implement. (True/False)
10. OpenAI's transformer models are the best for summarization tasks. (True/False)

# References

- *Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and 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/>
- *US 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>

# Further reading

- *Colin Raffel et al., 2019, Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer:*  
<https://arxiv.org/pdf/1910.10683.pdf>

# Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://www.packt.link/Transformers>



# 14

## Exploring Cutting-Edge LLMs with Vertex AI and PaLM 2

Maybe *Chapter 7, The Generative AI Revolution with ChatGPT*, made us think that OpenAI ChatGPT ended the tremendous progression of **Large Language Models (LLMs)** and Generative AI. Perhaps the incredible tools, such as GitHub Copilot, GPT-4, and OpenAI's embedding models, stunned us into believing we had reached a limit.

Then, on May 23, 2023, Google released the PaLM 2 Technical Report, built on the earlier PaLM and Pathways papers. The content of these papers is mind-blowing! The architecture of PaLM improved both software and hardware performances with ingenious innovations.

We will begin the chapter by examining Pathways to understand PaLM. We will continue and look at the main features of **PaLM** (which stands for **Pathways Language Model**), a decoder-only, densely activated, and autoregressive model transformer model with 540 billion parameters trained on Google's Pathways system. PaLM was trained on 780 billion tokens.

The next step will be to discover how PaLM 2 leveraged more machine efficiency by introducing scaling laws and optimizing the model size and the number of parameters. PaLM 2 performances attain similar results as OpenAI GPT-4. We will not attempt to decide which one is best for a project because each model may perform better for one NLP task and not another.

PaLM 2's performances are impressive! PaLM 2 will no doubt lead to much better versions. But once you understand the architecture of Pathways, PaLM, and PaLM 2, you will be ready for what is coming next.

We will then go through the wonderful and effective nascent assistants powered by Google's state-of-the-art transformers. We will run Generative AI-driven assistants in Gemini, Google Docs in Google Workspace, Google Colab Copilot, and AI PaLM 2 online. We will ask the Gemini API about itself. We will see how Google PaLM 2 can perform a chat task, a discriminative task (such as classification), a completion task (also known as a generative task), and more. We will then implement the Vertex AI PaLM 2 API for several NLP tasks, including Q&A, summarization, and more.

Finally, we will go through Google Cloud's fine-tuning process.

By the end of the chapter, you will be addicted to transformer-driven Generative AI assistants and APIs!

This chapter covers the following topics:

- Pathways architecture
- PaLM architecture
- PaLM 2 architecture
- Google's transformer-powered Generative AI assistants
- Gemini
- Google Workspace assistant for Google Docs
- Google Colab Copilot
- Vertex AI PaLM 2 online chat and code bots
- Discriminative and generative tasks with Vertex AI PaLM 2 online
- Implementing the Vertex AI PaLM 2 API

- Question-answering, summarization, sentiment analysis, and multiple-choice tasks with Bison
- Code generation PaLM 2
- Fine-tuning a Google LLM



With all the innovations and library updates in this cutting-edge field, packages and models change regularly. Please go to the GitHub repository for the latest installation and code examples:

<https://github.com/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition/tree/main/Chapter14>.

You can also post a message in our Discord community (<https://www.packt.link/Transformers>) if you have any trouble running the code in this or any chapter.

We will start our journey by discovering the innovative steps forward attained by PaLM, which is built on Pathways.

## Architecture

PaLM and PaLM2 were built on top of Pathways. Pathways is a Google technology that improves the efficiency of training LLMs through data parallelism, model parallelism, and execution-level parallelism.

We will begin with Pathways, the cornerstone of Google AI's impressive achievements.

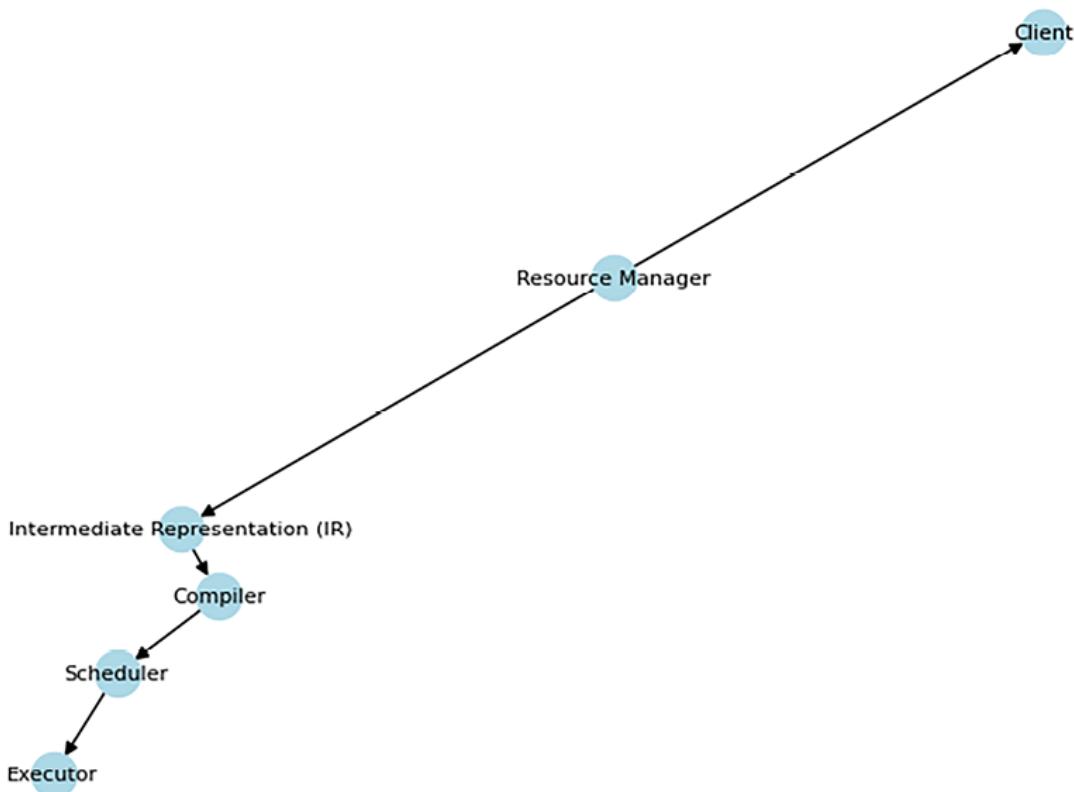
# Pathways

The title of the Pathways paper may seem esoteric. *Pathways: Asynchronous Distributed Dataflow* by Barham et al. (2022) indeed appears like something you might want to avoid looking into. However, once you start reading the paper, you will be hooked!

If we look at some of the key features, we are somewhat stunned:

- **Heterogeneous execution:** Pathways can run programs on many devices, including TPUs, CPUs, and GPUs. This is a significant advance when assembling all the computing power we can get.
- **Asynchronous execution:** Pathways allows programs to be executed *asynchronously*. This might seem uninteresting, but PaLM will build on this technology to run former sequential sublayers asynchronously in surprising ways, as we will see in the PaLM section of this chapter.
- **Dataflow programming:** Pathways contains dataflow programming to write programs that can scale to large datasets.
- **Extensible:** New features can be added when needed.

These general concepts encompass innovative functionality, as shown in *Figure 14.1*:



*Figure 14.1: Mind map of some of Pathways' key advances*

*Figure 14.1* shows a network of functions that mark the emergence of transformer architecture into highly industrialized technology. We are far from mainstream users enjoying their AI-powered assistants. We are deep in the inner workings of transformer model production!

Let's go through some of the main features of Pathways.

## Client

The client submits programs to the resource manager and controls the execution of those programs. When the client submits a program to the

resource manager, the resource manager allocates a virtual slice of the backend to the client. Then, the client starts executing the program.

The client manages the execution of the program, such as monitoring the progress of the computation and handling errors.

## Resource manager

The resource manager tracks the availability of devices and allocates virtual slices of the backend to clients. For example, in *Table 14.1*, we see how client requests have become virtual slices allocated to physical devices.

| Client   | Virtual slice | Physical device |
|----------|---------------|-----------------|
| Client 1 | Slice 1       | Device 1        |
| Client 2 | Slice 2       | Device 2        |

*Table 14.1: Device allocation*

## Intermediate representation

The **Intermediate Representation (IR)** is a language-neutral representation of a program that can be used to generate code for different backends.

Pathways uses a custom **Multi-Level Intermediate Representation (MLIR)** dialect for its IR.

MLIR is a relatively new technology. It is a compiler infrastructure that produces a common IR for many programming languages and hardware targets such as GPUs, TPUs, and CPUs.

# Compiler

The compiler transforms the IR into a low-level representation that can then be executed on physical devices.

Pathways is not done yet! It has a schedule that optimizes the usage of the devices.

# Scheduler

The scheduler manages the execution of the programs on the physical devices grouped in islands. Each island is managed by a centralized schedule that reduces execution times and performs load-balancing. In *Table 14.2*, we can see schedulers at work selecting the best available device for the incoming programs:

| Island | Device | Program   |
|--------|--------|-----------|
| 1      | 1      | Program A |
| 1      | 2      | Program B |
| 2      | 1      | Program C |
| 2      | 2      | Program C |

*Table 14.2: Scheduler activity*

# Executor

The executor is responsible for executing a program on a single device. The scheduler manages the incoming traffic of programs that need to be executed. The executor runs the programs when the scheduler triggers them to go to the best available devices.

We have gone through some of the main features provided by the incredible technological progress of Pathways.

Let's now see how PaLM leverages the power unleashed by Pathways.

## PaLM

Google AI's **Pathways Language Model (PaLM)** has 540 billion parameters and was trained with Pathways, the ML system we explored in the previous section. PaLM was trained on 6,144 **Tensor Processing Unit (TPU) v4** chips.

Google improved many aspects of the training process of LLMs. This section will focus on five key advances and their improvements.

### Parallel layer processing that increases training speed

A transformer block is a computation unit described in *Chapter 2, Getting Started with the Architecture of the Transformer Model*. The transformer block is mainly built with an attention layer, a feedforward neural network (MLP), and a layer normalization layer.

We can represent the classical transformer block with this equation:

$$y = x + \text{MLP}(\text{LayerNorm}(x + \text{Attention}(\text{LayerNorm}(x))))$$

In this formulation, we can see that  $x$ , the input embedded data, is added to the attention layer. This provides a residual connection so we don't lose the initial embeddings. We can also see that the MLP encompasses the whole process that precedes it, as seen in *Chapter 2*. Finally, we see another residual connection ( $x + \text{MLP}$ ), which ensures we don't lose any information for the initial embeddings.

The general idea is that the MLP encompasses the previous sublayers and normalizations. The attention sublayer follows the embedding input sublayer, and the MLP encompasses all of this. If you read *Chapter 2* carefully, you may think, "So what?"

Now, look at what the Google AI research team did! They ran the attention sublayer and the MLP in parallel:

$$y = x + \text{MLP}(\text{LayerNorm}(x)) + \text{Attention}(\text{LayerNorm}(x))$$

Yes, the equation is correct! The MLP does not come after the attention sublayer. It runs in parallel! You can imagine the speed the system is picking up.

## Shared input-output embeddings, which saves memory

PaLM shares the input and output embedding matrices, saving considerable memory. Usually, in a transformer model, embeddings are used to represent the input and output tokens. This means that we would have an input embedding matrix of size  $[x,d]$  for  $x$  (input) and  $d$  (dimension of the vector). We would also have an output embedding matrix of size  $[x,d]$ . The total size is  $2 \times [x,d]$ . With shared embeddings, we save memory and improve long-term dependencies.

## No biases, which improves training stability

In a neural network, biases are additional parameters added to each layer's output to help the model learn more complex relationships between the input and output data. However, this often leads to overfitting and instability. In this case, for PaLM, the results proved the efficiency of this decision.

## Rotary Positional Embedding (RoPE) improves model quality

RoPEs combine absolute and relative embeddings and apply a learned rotation matrix to the operation.

A simple conceptual example is “Dog is a noun.”

The absolute position of “Dog” is 1 in this sentence.

The relative position of “Dog” is still 1 if we begin the sequence at 1, or it is -1 if we begin analyzing the sequence starting with the verb “is.”

RoPE encodes the absolute positional information with a rotation operation matrix, and it takes relative position dependency in the self-attention sublayer.

Remember that the choices made in transformer models are empirical and based on trial and error. The takeaway is that RoPE is used for positional embedding, takes absolute positional information, and encompasses relative positional embedding, making it efficient in this case.

## SwiGLU activations improve model quality

**Switched Gated Linear Unit (SwiGLU)** has two inputs,  $x$  and  $y$ , and the output is  $x * \text{sigmoid}(y)$ , where sigmoid is the sigmoid function,  $\text{sigmoid}(y) = 1 / (1 + \exp(-y))$ . This effectively allows the network to learn to “turn on or

off” (or regulate) the activation of specific neurons. In this case, for PaLM, the empirical evaluation is that the process improves model quality. This may not apply to other architectures.

The Google AI research team optimized many other aspects of the training process and architecture. For more, consult PaLM: Scaling Language Modeling with Pathways, by Chowdhery et al. (2022):  
<https://arxiv.org/pdf/2204.02311.pdf>.

One can only wonder what Google would come up with after PaLM, but then PaLM 2 arrived in May 2023!

## PaLM 2

PaLM 2 is an advanced language model that outperforms its predecessor, PaLM, regarding multilingual capabilities and reasoning abilities. Reasoning abilities have become a critical domain to improve transformers’ abilities to mitigate their flaws.

The architecture of PaLM 2 has improved in many ways and directions, as shown in the mind map of some key advances in *Figure 14.2*:

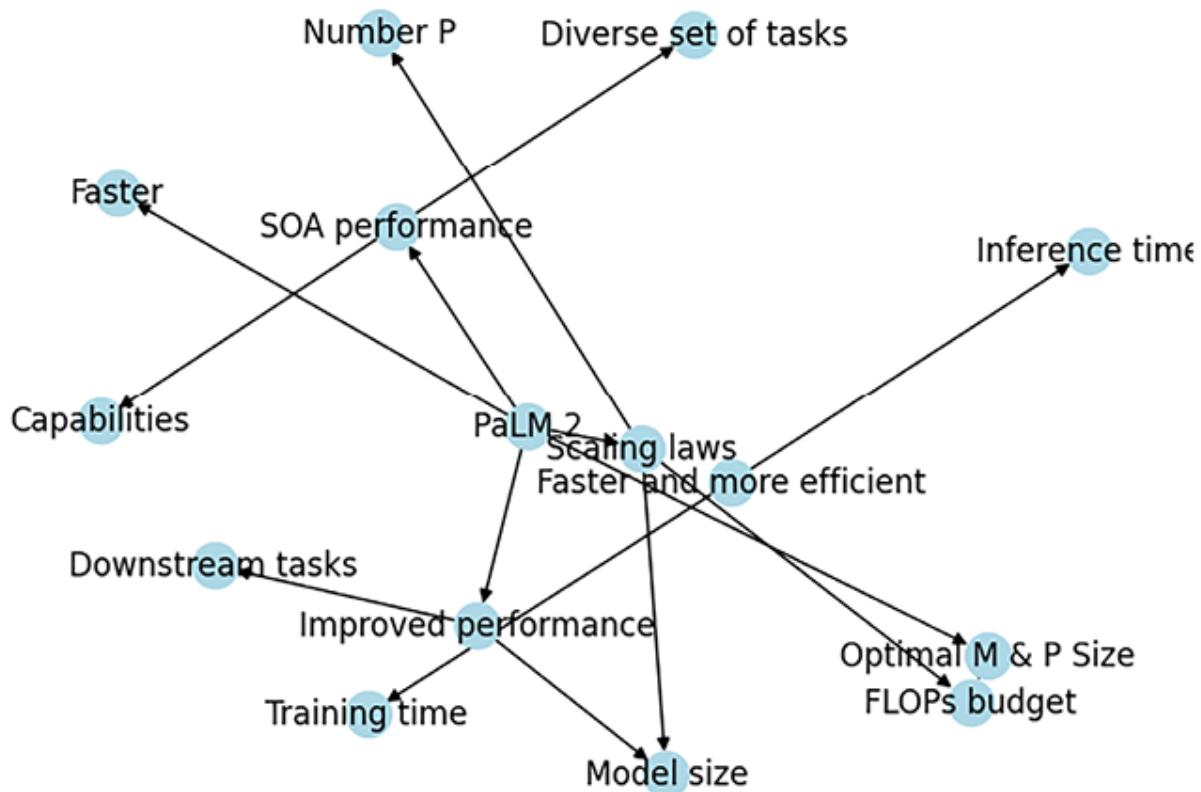


Figure 14.2: A mind map of multiple innovations of PaLM 2 interact and increase performances ( $P = \text{parameters}$ ,  $M = \text{model}$ )

Don't try to memorize all the improvements. As for PaLM, they are converging from every direction as the research teams add their discoveries to the model. Let's examine the main ones to see how they all interact to enhance the performance of PaLM 2.

## Improved performance, faster, and more efficient

PaLM 2 has significantly improved performance on downstream tasks across different model sizes compared to PaLM. The largest model in the PaLM 2 family, PaLM 2-L, is far *smaller* than the largest PaLM model! Yes, you read

correctly: *smaller*. Machine power has been optimized through reduced operations and parallel processing. PaLM 2 builds on the progress made by PaLM. For example, we are using the same input and output embeddings, thus reducing the number of computations. Another example is running the attention sublayer and the feedforward network asynchronously, which were formerly sequential, as explained in *Chapter 2, Getting Started with the Architecture of the Transformer Model*.

## Scaling laws, optimal model size, and the number of parameters

The Google research team optimized the training processes, which might, hopefully, reduce the size and power of hardware requirements.

The optimal model size (N) and the number of parameters (D) were found to grow in equal proportions as the FLOPs budget increased from  $1 \times 10^{19}$  to  $1 \times 10^{22}$  FLOPs.

This might seem relatively insignificant, but it is critical. We are witnessing scaling *laws*. Transformer technology is going from empirical testing to learning lessons and creating *laws* that will help other researchers who will, in turn, produce other *laws*.

The PaLM research team trained several models from 400M to 15B. They found that the optimal number of parameters for a model of  $1 \times 10^{22}$  FLOPs is only 10B (*Anil et al. (2023), page 7*)! It's an impressive step forward to transformer software and hardware optimization.

## State-of-the-art (SOA) performance and a new training methodology

This SOA performance comes as no surprise, considering the technological progress made. PaLM 2 achieves SOA performance across a diverse set of tasks and capabilities.

There are many other improvements. For more information, consult the PaLM 2 Technical Report, *Anil et al. (2023)*, at <https://arxiv.org/abs/2305.10403>.

We will now explore the result of these architectural improvements through Google assistants.

## Assistants

The structure of this section is similar to that of *Chapter 7, The Generative AI Revolution with ChatGPT*, and *Chapter 8, Fine-Tuning OpenAI GPT Models*.

We can view the similarity between Microsoft and Google assistants from two perspectives:

- The billion-dollar competition between these two giants is fierce, so they must produce products and services that match their technology rivalry.
- Standardization has sprung from necessity: they offer transformer Generative AI services in every application they sell, leading to an intuitive way to use them (prompts and parameters).

As we saw in *Chapter 1, What Are Transformers?*, transformers are **General Purpose Technologies (GPTs)** like electricity, computers, smartphones, and the internet. They have pervaded every domain and will increasingly progress.

However, each competing platform produces different results and performances that we must evaluate before building an application.



Transformer-driven Generative AI assistants represent a natural progression of GPT. As such, we must learn how to use them ethically, as we will see in *Chapter 15, Guarding the Giants: Mitigating Risks in Large Language Models*.

Before entering the ever-growing world of AI-driven assistants, read the following analysis from OpenAI GPT-4 on the costs related to next-generation, which is straying away from free open-source assistants:

1. Sophistication and Costs of AI Models: Modern AI models like GPT-4 are highly sophisticated and require significant computational resources.
2. Growing Need for Customized Models: As the demand for AI-driven solutions grows, there is a increasing need for customized models that can handle specific tasks and requirements.
3. Understanding Cloud-based AI: The ability to understand and utilize cloud-based AI services is becoming increasingly important for organizations.
4. Career Opportunities: With the growing demand for AI and data science professionals, there are many career opportunities available.
5. Continuous Learning and Adaptation: As with any evolving field, continuous learning and adaptation are key to staying competitive.
6. Importance of Budgeting and Resource Management: With the significant costs associated with AI development and deployment, effective budgeting and resource management are crucial.

Now, we will explore this cutting-edge world of smart transformer-driven assistants.

Google's Gemini, just like Microsoft's new Bing and ChatGPT, can answer questions, follow instructions, and provide source code.

## Gemini

Gemini operates on the advanced technology of the umbrella term Google Cloud AI, encompassing LaMDA, PALM, Gemini (multimodal) models,

and soon-to-come, newer, and better architectures. These transformer-driven large language Generative AI models are like the OpenAI models deployed under the umbrella term ChatGPT.

You can reach Google's Gemini UI by going to the following link:  
<https://gemini.google.com/app>.

We can ask Gemini a question about itself:

What is Gemini and explain how it uses transformer-driven gene

The following excerpt of the response shows the capabilities of the system:

I'm Gemini, the best way to directly access Google AI. I'm trai



Gemini, like all transformer models, operates on stochastic algorithms. As such, the responses may vary from one run to another. Also, Google updates its models and variants on a continuous basis, which leads to responses that will vary in time.

GPTs, such as transformer-driven generative systems, will continue flourishing until the market fully absorbs AI.

Google Workspace illustrates the diffusion of GPT transformers.

## Google Workspace

Transformer-driven Generative AI ChatGPT-like copilots have spread out to Google Workspace just as Microsoft 365 Copilot has been rolled out to everyday Microsoft tools:

<https://workspace.google.com/solutions/ai/>.

The standardization of the process simplifies how an everyday user can begin to work with the tools.

In Google Docs, for example, once you have gained access to transformer-driven Generative AI in one way or another, you can enter a general instruction as a prompt or a specific task such as grammar correction, summarization, and other NLP tasks. We can enter the following request, for example:

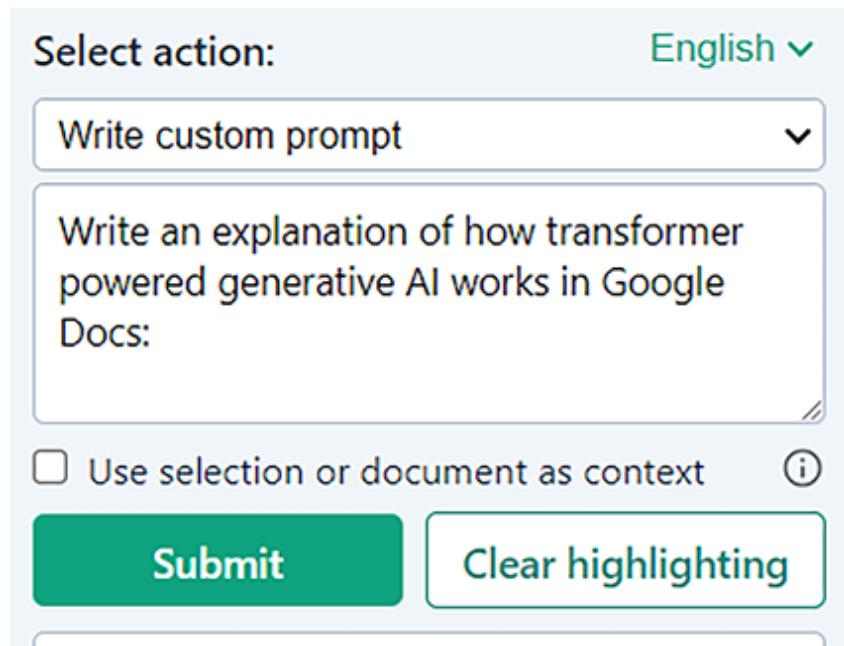


Figure 14.3: Generative AI tools on Google Workspace

The system will automatically generate content on your Google Docs page, as shown in *Figure 14.4*:

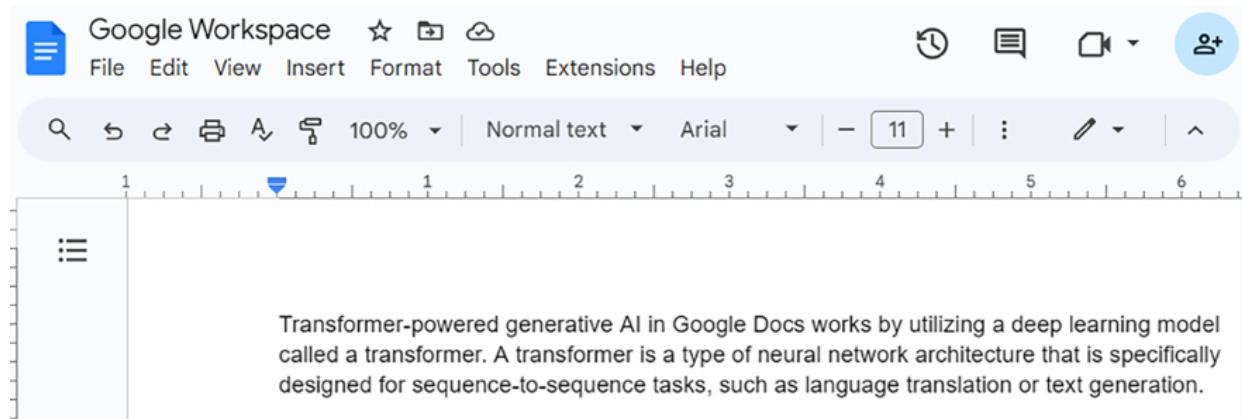


Figure 14.4: The tool can generate the text in Google Docs

The text is well written and accurate, although it might sometimes invent inaccurate content when it reaches its limits. Read the complete text below, which explains how transformers work quite well and warns us of their limitations:

In the context of Google Docs, the transformer model is trained on a large corpus of text. When a user interacts with the generative AI in Google Docs, the model generates new text based on the input provided. The transformer model employs a mechanism called self-attention, which allows it to consider all parts of the input sequence when generating each output token. The generated text is produced by sampling from a probability distribution over possible outputs. Google Docs leverages transformer-powered generative AI to provide users with suggestions and complete sentences. The transformer model continuously learns and adapts to user interactions, improving its performance over time. It is important to note that while transformer-powered generative AI is highly effective, it is not perfect and may occasionally produce inaccurate or inappropriate text.

The text generated is fine but we can go further and ask the system to summarize its content:

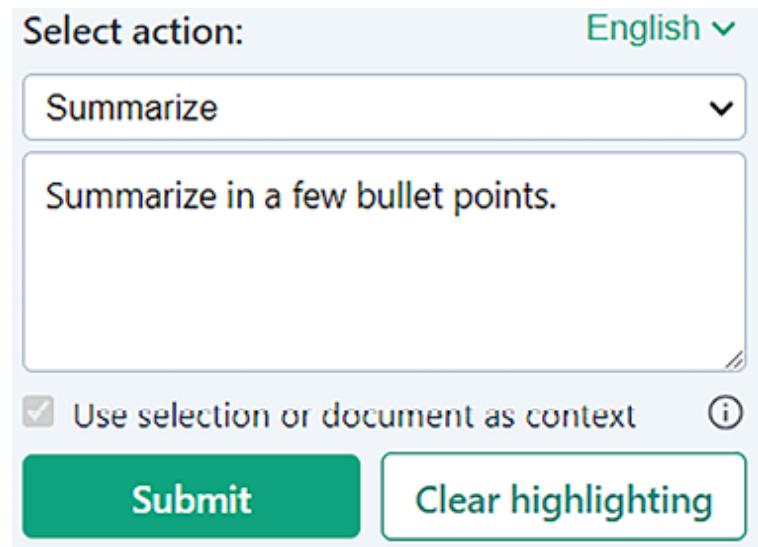


Figure 14.5: Summarizing the text into bullet points

The result shows the increasing power of transformers:

- Google Docs uses a deep learning model called a transformer for generative AI.
- The transformer model is trained on a large corpus of text data to learn patterns and relationships between words.
- Users can provide a prompt for the model to generate text, and it predicts the most likely next word based on the input.
- The model uses self-attention to understand relationships between words and can capture long-range dependencies.
- The generated text is produced by sampling from a probability distribution, allowing for diverse outputs.
- Google Docs uses this AI for features like Smart Compose and Smart Reply, but users should review and edit the generated text as it may occasionally be incorrect or nonsensical.

We can spend more time on endless interactions with highly efficient LLMs. The machine will never get tired! However, let's move from office assistants

to development assistants.

## Google Colab Copilot

Google Colab Copilot is similar to GitHub Copilot, as we used in *Chapter 7, The Generative AI Revolution with ChatGPT*, in the *GitHub Copilot code assistant* section. Google Colab Copilot is integrated into Google Colab: <https://blog.google/technology/developers/google-colab-ai-coding-features/>.

This is a normal evolution. Smartphones and computers have the same core functionalities as all GPTs. AI-driven copilots such as *Colab Generative AI* are no exceptions.

You will need a Google account to run Google Colab. Go to Google Colab, create a notebook, and when you first use the copilot, a window will pop up:

### Terms of Service

---

Generative AI in Colab is an experimental technology and may sometimes display inaccurate or offensive information that doesn't represent Google's views, so [use code with caution](#). Don't rely on responses generated by Colab generative AI features as medical, legal, financial, or other professional advice.

I accept my use of Generative AI in Colab is subject to the [Google Terms of Service](#) and the [Generative AI Additional Terms of Service](#)

---

*Figure 14.6: Generative AI in Colab Terms of Service*

As always, you will see a disclaimer for Generative AI due to the occasional inaccuracies it produces.

If you choose to run the AI copilot, a prompt text box will appear:

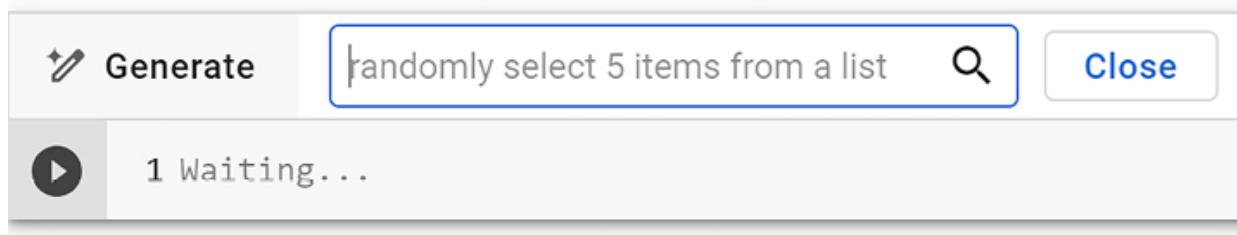


Figure 14.7: The AI copilot prompt box

First, enter a prompt such as the following:

```
Create a function Fibonacci up to 10
```

The code is generated automatically with the prompt:

```
# prompt: Create a function Fibonacci up to 10
def fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
print(fibonacci(10))
```

The output is 55, which is correct.

Now, let's simulate an example where the code produced an error, such as `n = 0` instead of `n == 0`:

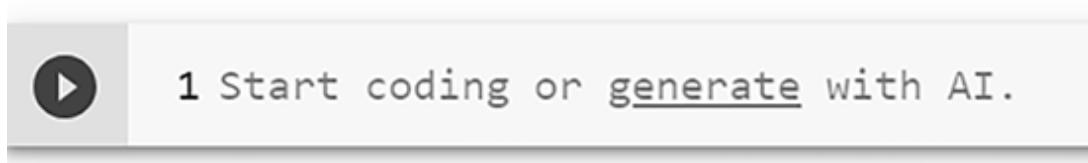
```
File "<ipython-input-59-179cb71573cc>", line 4
  if n = 0:
  ^
SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of '='?
```

**EXPLAIN ERROR**

Figure 14.8: The EXPLAIN ERROR functionality

We can click on the `EXPLAIN ERROR` button to obtain an explanation and code for more difficult errors.

There is also a completion function in the cells:



*Figure 14.9: The generate with AI option in Colab*

As with classical software editors, you can also start coding to obtain completion suggestions, only this time, it will generate several lines of code! You can also click on **generate** when you wish, and the generation text box prompt will appear.

Using this functionality will alleviate the effort of developing and push your creativity to its limits!

Once you get used to the environment, you will be able to code like an Indy 500 driver in the fastest cars on the planet!

We will now move to Google Cloud's AI platform to begin running Vertex AI PaLM 2.

## Vertex AI PaLM 2 interface

Vertex AI PaLM 2 is the SOA equivalent of OpenAI GPT-4. This section will go through the online interface and the main parameters. We will leverage PaLM 2's knowledge to explain its parameters.

To access Vertex AI, you will need an account. Once you have signed up, you can read this self-contained section if you don't wish to sign up.

Then, navigate to the language model environment. We will get started by exploring the online PaLM 2 assistant:

## Get started

The screenshot shows a 'Get started' section with two main options:

- Design and test your own prompts**:
  - Design prompts for tasks relevant to your business use case including code generation.
  - Take a tutorial on [creating effective text prompts](#).

**+ TEXT PROMPT**

**+ CODE PROMPT**
- Start a conversation**:
  - Have a freeform chat with the model, which tracks what was previously said and responds based on context.
  - Take a tutorial on [designing chat prompts](#).

**+ TEXT CHAT**

**+ CODE CHAT**

*Figure 14.10: Getting started with Vertex AI*

The Vertex AI interface resembles the OpenAI Playground with a prompt text box. There is also a list of parameters we can modify, as shown in *Figure 14.11*:

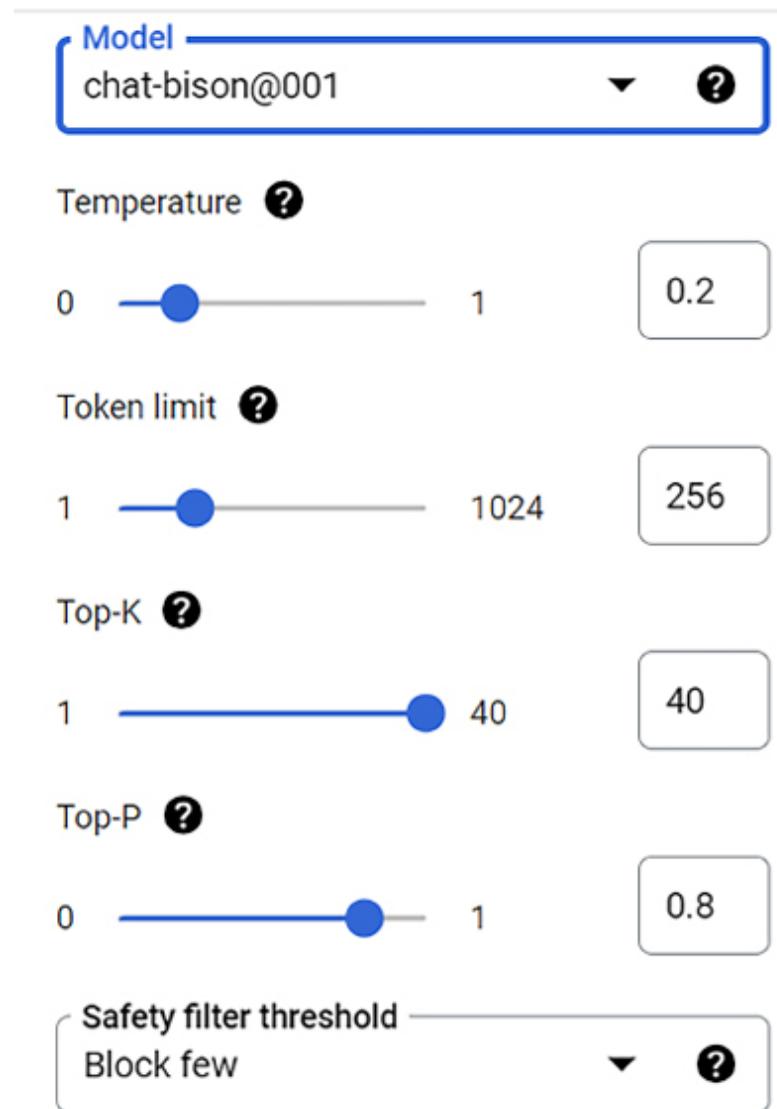


Figure 14.11: Key sampling parameters

The parameters are the same as for Open AI GPT models, as we saw in *Chapter 7, The Generative AI Revolution with ChatGPT*.

The four key sampling parameters in *Figure 14.11* appear in a critical technical order:

- **Temperature:** This hyperparameter is applied to the raw output logits of the model before softmax when making inferences. A small value,

such as 0.2, will encourage the selection of the most probable output when softmax is applied. The higher probabilities will appear higher after softmax. In this case, the model will produce more deterministic outputs. However, if the value of temperature approaches or exceeds 1, the model will be less confident after softmax, the randomness of the output will increase, and the predictions will be more variable. The choice represents a trade-off between diversity and the quality of the predictions.

- **Token limit:** This hyperparameter determines the maximum sequence length that the model will generate. The generation process will stop either when this limit is reached or when the model generates a token that signifies the end of the sequence. The token limit operates independently of other hyperparameters, and its implementation may vary based on the specific architecture of the model.
- **Softmax:** *Softmax is not displayed but is applied in most cases.* This is a probability normalization function. It is applied to the output logits of the model both during training and for inference. During inference, softmax is applied after the temperature operation. It transforms the logits such that they lie in the range between 0 and 1, can be interpreted as probabilities, and their sum equals 1.
- **Top-K:** The Top-K hyperparameter limits the set of probabilities for the next token to the value of K. Top-K is applied after softmax. For example, if the hyperparameter is set to 40, the 40 top probabilities after the softmax function will be selected.
- **Top-P** or nucleus sampling will sort the probabilities in descending order. Then, it will sum the probabilities from the top down until it reaches the Top-P hyperparameter provided, which could be 0.8, for example. Once the probabilities have been sampled, one of the

probabilities is randomly chosen as the next token. This token is then added to the input token sequence for the next token prediction. This method tends to generate more diverse and creative responses than Top-K. Top-K and Top-P can be used separately or together. If they are used together, Top-K is applied first to reduce the number of probabilities, and then Top-P is applied to the resulting set.

If temperature, softmax, Top-K, and Top-P are implemented together, we can represent the sampling function to choose the next token as a composite function:

```
Sampler = Top-P(Top-K(softmax(temperature(output logits))))
```

PaLM or GPT models produce raw logits. Then, a Sampler function takes over, generates the next token, and adds it to the tokenized input sequence for the model to continue predicting tokens until the token limit is reached or an end-of-sequence token is detected.

The safety filter threshold is interesting in limiting potentially harmful content. Its value depends on the goal of each project. For more, you can consult Google's Generative AI documentation:

[https://cloud.google.com/vertex-ai/docs/generative-ai/learn/models?  
\\_ga=2.96585695.-1947292201.1682929481](https://cloud.google.com/vertex-ai/docs/generative-ai/learn/models?_ga=2.96585695.-1947292201.1682929481).

The requests made to Vertex AI PaLM will sometimes require different parameter configurations depending on the model.

Vertex AI is organized with model naming schemes like OpenAI. For the Foundation Models in Vertex AI, the format is the use case as a prefix to the model size.

The model size is the key part of the name of the version:

- Bison: Best capability and higher cost.
- Gecko: Smallest and lower cost.

Google AI, like OpenAI, has stable or newer versions along with life cycles. This is also normal, just like for cars, smartphones, computers, and every product and service.

We must get used to this and remain extremely careful in production, as we will see in *Chapter 15, Guarding the Giants: Mitigating Risks in Large Language Models*.

The main prefixes are `text`, `text-embedding`, `chat`, and `codechat`. For example:

- `text-bison` for classification, entity extraction, Q&A, summarization, sentiment analysis, and other text-oriented tasks.
- `textembedding-gecko` to embed text inputs, as we saw in *Chapter 11, Leveraging LLM Embeddings as an Alternative to Fine-Tuning*.
- `code-bison` for code requests.

You will find several other models in Google's Generative AI documentation. New models are available, such as Gemini, a multimodal model similar to GPT-4V. And more are coming!

Keep model's stability and lifecycle in mind before deploying them in production.

Before implementing the Vertex AI API, let's run a few tasks with the online assistant. We must remember that mainstream users will increasingly integrate AI-driven assistants into their everyday work lives.

## Vertex AI PaLM 2 assistant

The trend of Google AI is to enable non-ML experts to perform ML expert tasks, as we will see in this section.

A prompt will be entered in the following examples, as shown in *Figure 14.12*:

---

▼ **Prompt**

---

Write a prompt and then click Submit

*Figure 14.12: The prompt user interface*

We will choose one of the PaLM 2 models at this date (they will continually evolve in this list, such as `text_bison@002`, `Gemini`, etc.):

## PaLM 2

text-bison (latest)

Latest

**text-bison@001**

Best value

## Codey (Palm 2)

↔ code-bison@001

**PREVIEW**

Code generation

↔ code-gecko@001

**PREVIEW**

Figure 14.13: The list of models

We will use the default parameters and display the response beginning with the model's name:

- **An instruction task**

**Prompt:** Explain how to install an add-on in Google Docs:

**Response (text-bison):**

1. Open Google Docs and click the "Add-ons" menu.
2. Click "Get add-ons".
3. In the search bar, type the name of the add-on you want to install.

4. Click the add-on's name and click "Install".
5. The add-on will be installed and added to the "Add-ons" menu.
6. To use the add-on, click the "Add-ons" menu and select the add-on...

- **A generative chat task**

**Prompt:** Explain what large language model Generative AI is:

**Response (text-bison):**

A large language model (LLM) is a type of artificial intelligence that can generate text, translate languages, write different kinds of creative content, answer your questions, and summarize text...

...One of the most exciting things about LLMs is their ability to generate creative text. They can write different kinds of creative content, such as poems, code, scripts, musical pieces, and even entire novels. This is because LLMs are able to learn the different patterns and structures that are used in different types of creative text...

- **A discriminative task**

**Prompt:** Is the following text about baseball or hockey, and explain why: "The player threw the ball."

**Response (text-bison):**

The text is about baseball because baseball is the only sport where a player throws a ball. In hockey, players use a stick to hit the puck.

- **A question-answer task on a text**

**Prompt:** What is the name of the main character in this text and the main activity of the main character: Jill, John, and Alice were working on a project. Jill designed a web interface, John wrote the text, and Alice controlled the design and the content of the website and reported to the top management.

- **Response (text-bison):**

The main character is Alice. The main character is the person who does the most important thing in the story. In this case, Alice is the one who controls the design, the content for the website and reported to the top management.

- **A code instruction task**

**Prompt:** Write a program in Python that can calculate the first 10 prime numbers starting from 1. Make sure you filter the output to limit it to 10 outputs:

**Response (code-bison@001):**

```
# This program will calculate the first 10 prime numbers
# Define a function to check if a number is prime.
def is_prime(number):
    # Check if the number is divisible by 2.
    if number % 2 == 0:
        return False
    # Check if the number is divisible by any number from 3 to
    for i in range(3, int(number ** 0.5) + 1, 2):
        if number % i == 0:
            return False
    # If the number is not divisible by any number from 2 to t
    return True
# Get the first 10 prime numbers.
prime_numbers = []
```

```
for number in range(1, 10000):
    if is_prime(number):
        prime_numbers.append(number)
# Print the first 10 prime numbers.
print(prime_numbers[:10])
```

If we run the code, we can confirm that the output is correct:

```
[1, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

We have seen that Google assistants are intuitive, effective, and easy to run. We will now build functions for NLP tasks with the Vertex AI PaLM 2 API.

## Vertex AI PaLM 2 API

The Vertex AI platform evolves continually. You are at the cutting edge!

The models that we use get updated from time to time. These updates can cause small differences in the results or how it works. The latest version might not be the most stable version. You will need to be on the watch and consult Google's versioning documentation regularly:

<https://cloud.google.com/vertex-ai/docs/generative-ai/learn/model-versioning>.

That is the price to pay to be on top of the market! However, the power of the models makes it worthwhile to explore PaLM 2. When the models stabilize, you will be far ahead of your competition. So, please fasten your seat belts, and let's get the API rolling!



There are hundreds of known NLP tasks, and hundreds more mainstream users invent some every day. You cannot implement them all at once. Focus on understanding the tasks you explore in depth. You will then be able to adapt to new ones.

Open `Google_Vertex_AI.ipynb` in the chapter directory of the GitHub repository.

The first part of the notebook goes through the installation and authentication processes.

We first install `textwrap` to wrap the responses in a nice format:

```
import textwrap
```

The notebook then installs the Google Cloud AI library:

```
!pip install google-cloud-aiplatform
```

Restart the runtime when prompted. Then go to the **Runtime** menu and select **Run all**.

You will need to sign up on Google Cloud, create a project, and follow the instructions provided: <https://console.cloud.google.com/>.

This notebook uses a simplified authentication linked to the Google Cloud account:

```
from google.colab import auth as google_auth
google_auth.authenticate_user()
```

Vertex AI is imported:

```
import vertexai
```

We are all set!

The section titles in this section are the section titles in the notebook. The first request is a **Question Answering (QA)** task.

## Question answering

*Chapter 13, Summarization with T5 and ChatGPT*, walked us through summarization tasks. QA is another way to see if the model understands a text.

We'll use `chat-bison@001` to ask questions.

We first import the modules we need:

```
import vertexai
from vertexai.preview.language_models import ChatModel, InputTo
```

Then, we initialize our project (`"aiex-57523"`) and location (`"us-central1"`). Make sure to replace the project and location names with your values. For these examples, we will use `chat-bison@001`, a very powerful PaLM 2 general-purpose Generative AI transformer:

```
vertexai.init(project="aiex-57523", location="us-central1")
chat_model = ChatModel.from_pretrained("chat-bison@001")
parameters = {
    "temperature": 0.2,
    "max_output_tokens": 256,
    "top_p": 0.8,
    "top_k": 40
}
chat = chat_model.start_chat(
    context="""Answer a question on the text submitted""",
)
```

You can modify the parameters and change the context of the request.

We can now make our request. The question is related to a memorization issue LLMs face when they repeat existing texts, word for word:

```
response = chat.send_message("""Based on the following text, D
```

The response is wrapped and displayed:

```
wrapped_text=textwrap.fill(response.text, width=40)
print(wrapped_text)
```

The response is well intentioned:

No, Gemini does not repeat existing text. It is a multimodal LLM, which means that it can understand and generate text, code, images, and more. It excels at reasoning across vast amounts of information, solving complex problems, and adapting to various tasks through fine-tuning.

You can also run the following cell to ask Gemini about Gemini through the API and compare the outputs with the PaLM 2 model. Ultimately, it will be up to you to choose the model that fits your needs taking quality and costs into account.

We can also ask questions without providing a text.

## Question-answer task

This request will contain an open question with no text to refer to. This time, we will build a function. The function includes the complete request. It can be modified and customized:

```
from vertexai.preview.language_models import ChatModel, InputOutput
def predict_large_language_model_sample(
    project_id: str,
    model_name: str,
    temperature: float,
    max_output_tokens: int,
    top_p: float,
    top_k: int,
    location: str = "us-central1",
) :
    """Predict using a Large Language Model."""
    vertexai.init(project=project_id, location=location)
    chat_model = ChatModel.from_pretrained(model_name)
    parameters = {
        "temperature": temperature,
        "max_output_tokens": max_output_tokens,
        "top_p": top_p,
        "top_k": top_k,
    }
    chat = chat_model.start_chat(
        examples=[]
    )
```

```
response=chat.send_message('''What Transformer model are you using?''')
#print(response.text)
wrapped_text=textwrap.fill(response.text, width=40)
print(wrapped_text)
```

We call the function that will ask PaLM 2 what model it is using:

```
predict_large_language_model_sample("aiex-57523", "chat-bison@vertexai.com")
```

The response is accurate:

```
I am powered by PaLM 2, which stands for Pathways Language Model 2. PaLM 2 was trained by a team of engineers and scientists at Google AI.
```

PaLM 2 can summarize texts efficiently.

## Summarization of a conversation

LLMs are good at summarizing texts. But what about dialogues?

This request contains a dialogue. We first tell the model what we want:

```
import vertexai
from vertexai.language_models import TextGenerationModel
vertexai.init(project="aiex-57523", location="us-central1")
parameters = {
    "temperature": 0.2,
    "max_output_tokens": 256,
    "top_p": 0.8,
    "top_k": 40}
```

```
    }
model = TextGenerationModel.from_pretrained("text-bison@001")
response = model.predict(
    """Summarize the following conversation between a service
```

Then we provide the conversation:

```
Service Rep: How may I assist you today?
Customer: I need to change the shipping address for an order.
Service Rep: Ok, I can help you with that if the order has not
Customer: Yes, it's 88986367.
Service Rep: One minute please while I pull up your order info
Customer: No problem
Service Rep: Ok, it looks like your order was shipped from our
Customer: Sigh, ok.
Service Rep: Is there anything else I can help you with today?
Customer: No, thanks."""",
    **parameters
)
#print(f"Response from Model: {response.text}")
wrapped_text=textwrap.fill(response.text, width=40)
print(wrapped_text)
```

The response is acceptable:

The customer wants to change the shipping address for an order. The service rep informs the customer that the order has already shipped and they will need to contact the shipping provider to update their delivery details. The customer is not happy but agrees to contact the shipping provider.

We can expand this functionality to meetings, support line dialogues, and many types of exchanges.

We can also perform advanced sentiment analysis.

## Sentiment analysis

Transformer models have no emotions. They use stochastic mathematical models with matrix multiplications and have strictly no feelings. Yet, they can provide a clinical sentiment analysis of complex texts.

We first tell the model what we expect:

```
from vertexai.language_models import TextGenerationModel
vertexai.init(project="aiex-57523", location="us-central1")
parameters = {
    "temperature": 0.2,
    "max_output_tokens": 256,
    "top_p": 0.8,
    "top_k": 40
}
model = TextGenerationModel.from_pretrained("text-bison@001")
response = model.predict(
    """Is the sentiment positive or negative towards Louis van
```

However, we provide a challenging text that contains negative phrases such as `was struggling` and `hadn't any confidence`:

```
Article:
Louis van Gaal said he had no option but to substitute Paddy McNair. The 19-year-old was hooked by Van Gaal after only 39 minutes as McNair was struggling to contain Southampton strikers Shane Long and Sadio Mane. Speaking to Sky Sports after the match, Van Gaal explained: \'I had to (substitute him), it\'s very disappointing for me as
```

```
\'And I think, after the change, we played a little better.\'
But in spite of the fact United won the game, McNair was exposed.
Jonny Evans came on to replace Smalling, before McNair made way.
And, despite admitting it will be difficult for McNair to accept,
Van Gaal continued: \'Of course, it\'s tough (for McNair), but
The victory moved United up to third in the Premier League - t
    **parameters
)
#print(f"Response from Model: {response.text}")
wrapped_text=textwrap.fill(response.text, width=40)
print(wrapped_text)
```

The model finds the right sentiment in this complex text:

Positive

We are intrigued. How did the model reach that conclusion? We then modify the request to ask for an explanation and provide the same text again:

```
...response = model.predict(
    """Is the sentiment positive or negative towards Louis van
Article:
Louis van...
```

The response is quite surprising for a system that only uses mathematics to determine the sentiment that prevails in the text:

The sentiment towards Louis van Gaal is positive. The article is about Louis van Gaal's decision to substitute Paddy McNair in the first half of the match against Southampton. The article states that Van Gaal had to make the

substitution because McNair was struggling and had given away three big chances. However, the article also says that Van Gaal believes that the substitution was the right thing to do and that it will benefit McNair in the long run. Overall, the article paints Van Gaal in a positive light. It shows him as a manager who is willing to make tough decisions in order to win games, and who is also looking out for the best interests of his players.

We will now see if PaLM 2 can solve a multi-choice problem.

## Multi-choice problems

A multi-choice problem requires a full understanding of a text. In this example, we first provide a list of topics:

```
from vertexai.language_models import TextGenerationModel
vertexai.init(project="aiex-57523", location="us-central1")
parameters = {
    "temperature": 0.2,
    "max_output_tokens": 256,
    "top_p": 0.8,
    "top_k": 40
}
model = TextGenerationModel.from_pretrained("text-bison@001")
response = model.predict(
    """Multi-choice problem: What is the topic of this text?
- entertainment
- technology
- politics
- sports
- business
- health
- fun
- culture
```

- culture
- science

We then continue the prompt with the text:

```
Text: Samba is a name or prefix used for several rhythmic vari  
Samba was modernly structured as a musical genre only in the la  
In this way, the sambistas of Estácio created, structured and  
    **parameters  
)  
#print(f"Response from Model: {response.text}")  
wrapped_text=textwrap.fill(response.text, width=40)  
print(wrapped_text)
```

The response is acceptable:

```
culture
```

We can see that PaLM 2 can perform quite a variety of tasks. Can it produce code through its API? Let's find out.

## Code

For this example, we will implement `codechat-bison`.

We will provide a straightforward request for a coding problem:

```
from vertexai.preview.language_models import CodeChatModel  
vertexai.init(project="aiex-57523", location="us-central1")  
chat_model = CodeChatModel.from_pretrained("codechat-bison@001")  
parameters = {  
    "temperature": 0.5,  
    "max_output_tokens": 1024
```

```
        }
chat = chat_model.start_chat()
response = chat.send_message("""Write the code in Python for a
print(f"Response from Model: {response.text}")
```

We expect a response containing the code for a Fibonacci suite up to 10. And sure enough, the response looks accurate:

```
Response from Model: The Fibonacci sequence is a series of numb
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
You can write a function in Python to calculate the Fibonacci s
```
def fibonacci(n):
    """
    This function calculates the Fibonacci sequence up to the giv
Args:
    n: The number to calculate the Fibonacci sequence up to.
Returns:
    The Fibonacci sequence up to the given number.
"""
# Initialize the sequence with 0 and 1.
a = 0
b = 1
# Iterate through the sequence, adding each number to the pre
for i in range(n):
    a, b = b, a + b
# Return the sequence.
return a
```
This function takes a number as input and returns the Fibonacci
0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

Of course, if you start at 0, you will reach 34. If you start at 1, you will reach 55.

Now, we want to make sure this response makes sense and will run it in the next cell:

```
def fibonacci(n):
    """
    This function calculates the Fibonacci sequence up to the given number.

    Args:
        n: The number to calculate the Fibonacci sequence up to.

    Returns:
        The Fibonacci sequence up to the given number.

    """
    # Initialize the sequence with 0 and 1.
    a = 0
    b = 1
    # Iterate through the sequence, calculating each new number.
    for i in range(n):
        # Calculate the next number in the sequence.
        c = a + b
        # Update the sequence.
        a = b
        b = c
    # Return the sequence.
    return a, b
```

We now call the function:

```
fibonacci(10)
```

The output is conceptually correct, but the result is not accurate:

```
(55, 89)
```

89 requires more than 10 steps.

Let's go back and ask PaLM 2 to correct the code it just produced:

```
import vertexai
from vertexai.preview.language_models import CodeGenerationModel
vertexai.init(project="aiex-57523", location="us-central1")
parameters = {
    "temperature": 0.2,
    "max_output_tokens": 1024
}
model = CodeGenerationModel.from_pretrained("code-bison@001")
response = model.predict(
    prefix = """Correct the following code in Python that calculates the nth Fibonacci number.
def fibonacci(n):
    """Calculates the nth Fibonacci number.
    Args:
        n: The nth Fibonacci number to calculate.
    Returns:
        The nth Fibonacci number.
    """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
"""
print(f"Response from Model: {response.text}""", parameters)
)
print(f"Response from Model: {response.text}")
```

The response is now correct, considering the way the request was made:

```
Response from Model: ```python
def fibonacci(n):
    """Calculates the nth Fibonacci number.
    Args:
        n: The nth Fibonacci number to calculate.
    Returns:
        The nth Fibonacci number.
    """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```
```

```
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
print(fibonacci(10))
```
```

We can paste and test it:

```
def fibonacci(n):
    """Calculates the nth Fibonacci number.
    Args:
        n: The nth Fibonacci number to calculate.
    Returns:
        The nth Fibonacci number.
    """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
print(fibonacci(10))
```

The output is correct: 55

We could go on and add hundreds of tasks! We could run every NLP task we can think of. The model will most probably respond accurately to most of them.



The potential of LLMs to address hundreds of NLP tasks has led to a new term: **Massive Multitask Language Understanding (MMLU)**. Why not use the term Foundation Model, which is already well-accepted? Because Foundation Models can perform multimodal tasks, including generative

imaging, audio, and image recognition. MMLU models will progressively acquire the ability to perform almost any NLP task. Keep your eyes open. This is only the beginning!

However, even LLMs sometimes fail when we need them for a specific project.

We then might have to fine-tune a Google AI model.

## Fine-tuning

Fine-tuning becomes an option when the responses produce an acceptable result or when the prompt design does not meet expectations. Or not! In *Chapter 11, Leveraging LLM Embeddings as an Alternative to Fine-Tuning*, we saw that advanced prompt engineering leveraging OpenAI LLM Ada's embeddings could produce good results.

So, what should we do? Prompt design by crafting good prompts with a ready-to-use model? Prompt engineering with an embedding model? Fine-tune a model to fit our needs?

Each of these choices comes with a cost. The best empirical method in computer science remains to:

- Rely on a reliable and optimized (volume, quality) evaluation dataset.
- Test different models and approaches. In this case, evaluate the outputs obtained through prompt design, engineering, and fine-tuning.
- Evaluate the risks and costs.

Like **Amazon Web Services (AWS)**, Microsoft Azure, IBM Cloud, and others, Google Cloud provides a solid, simplified approach to carrying out ML tasks on a cloud platform.

The first step to fine-tuning a model is to create a bucket.

## Creating a bucket

You will need to have storage space and permissions to create and use a bucket (a container in which data is stored):

<https://cloud.google.com/storage/docs/creating-buckets>.

Once you have configured your storage space and permissions, you can upload a JSONL file to your bucket. The dataset loaded in this example, `sports2_prepared_valid.jsonl`, is a file generated using the same method as for OpenAI in *Chapter 8, Fine-tuning OpenAI GPT Models*, section 1.2. *Converting the data to JSONL*. You can retrieve this dataset, `sports2_prepared_valid.jsonl`, from the repository of this chapter or prepare one of your choice. In the example in this section, the data was retrieved with the following code and then processed as explained in *Chapter 8*:

```
from sklearn.datasets import fetch_20newsgroups
import pandas as pd
import openai
categories = ['rec.sport.baseball', 'rec.sport.hockey']
sports_dataset = fetch_20newsgroups(subset='train', shuffle=True)
```

The goal is to identify the categories. We first load the prepared dataset:

Model tuning creates and deploys a new model from an existing one that's better adapted to your use case. Currently, model tuning occurs in limited regions. If you have an organization policy restricting certain regions, model tuning may fail.

**Tuning dataset**

The tuning dataset is a JSONL file that contains model prompt and responses examples(one per line). It's recommended that you use at least 100-500 samples. You can upload the file or select one that's already on Cloud Storage.

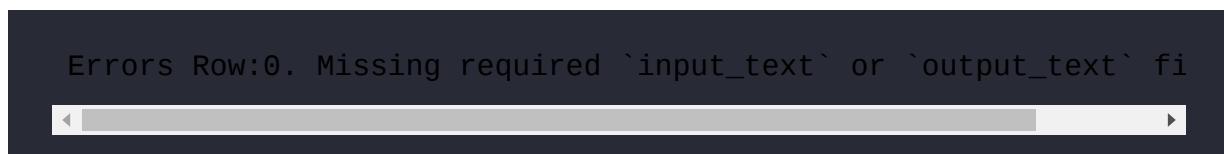
Upload JSONL file to Cloud Storage  
 Existing JSONL file on Cloud Storage

Select JSONL file \*  
sport2\_prepared\_valid.jsonl X BROWSE

Each line in the file contains one example:

Figure 14.14: Loading the dataset to use for fine-tuning

Google's dataset controller produced the following error:



`sports2_prepared_valid.jsonl` contains `prompt` and `completion`; we can simply find and replace them with `input_text` and `output_text`, respectively.

The reason could be that though LLMs are often generative transformers, they can perform discriminative (basic classification) tasks that do not produce completions but labels such as `Class_A` or `Class_B`. In this case, using the more general “input” and output” prefixes makes sense because they cover both discriminative and generative tasks. The format may change over time, but the process remains the same: a prompt and a label for supervised learning for a discriminative task.

# Fine-tuning the model

In this case, the dataset is built for a classification task to determine if a text is related to hockey or baseball:

Model name \*  The name of the new model. Up to 128 characters.

Settings

Base model  The base model that will be used to create a new tuned model.

Train steps \*  ?

Learning Rate \*  ?

Working directory \*  gs:// aiex-57523-vcm  The Cloud Storage location where the artifacts are stored during the pipeline tuning run.

Figure 14.15: Selecting the hyperparameters

The model used is PaLM 2, `text-bison@001`. You can select other ones depending on your project and also the availability of the models in a region. Fine-tuning requires the activation of specific Google Vertex AI libraries: <https://cloud.google.com/model-garden>. You will have to create a project and activate a billing account. In some cases, Google may provide free credits to get started. Make sure you verify the budgets, costs, and everything related to billing before starting.

Also, for some regions and services, the number of requests is limited. Make sure you have access to the necessary quotas. Finally, the interface will evolve as Google keeps creating new functionality. However, the fundamental concepts remain the same: preparing a dataset, fine-tuning a model, and testing the model.

This process may seem less intuitive and quick to use than some open-source platforms. However, being able to manage Google Cloud, AWS, Microsoft Azure, IBM Cloud, and Hugging Face in conjunction with one of the major platforms or any other professional cloud platform is an excellent asset to look into for upskilling.

As AI-driven assistants are taking over many AI and data science tasks, managing a cloud platform for AI projects is an excellent skill to acquire.

Now, we can create and run the fine-tuning process:

The screenshot shows the 'Tuning' section of the Google Vertex AI interface. At the top, there are tabs: 'GET STARTED', 'MY PROMPTS', 'TUNING' (which is highlighted in blue), and 'PREVIEW'. Below the tabs, the word 'Tuning' is displayed with a '+ CREATE TUNED MODEL' link. A descriptive text states: 'This page lists tuned models created in Generative AI Studio. To view all your models in Vertex AI, go to Model Registry'. A table below lists a single tuning job: 'Classification01' with a pipeline run ID of 'tune-large-model-20230626060838'. The status is 'Pending' with a timestamp of 'Jan 1, 1970, 1:00:00 AM'. There is a 'DETAILS' link next to the timestamp.

| Model            | Pipeline run                                    | Pipeline run status | Run start               |                         |
|------------------|---|---------------------|-------------------------|-------------------------|
| Classification01 | <a href="#">tune-large-model-20230626060838</a> | Pending             | Jan 1, 1970, 1:00:00 AM | <a href="#">DETAILS</a> |

*Figure 14.16: Fine-tuning the model with the sports data*

Once the model is fine-tuned, we can run it by following the Google Vertex AI testing process, depending on the permission granted for each project.

PaLM 2's capabilities are comparable to GPT-4. There is no absolute "best solution." Each project has different goals and specifications. It will be up to the project management team to decide which platform is best for a project,

taking into account cost, performance, security, privacy, and other project constraints.

Remember that the role of an AI expert is not to influence decision-makers but to show them the pros and cons of each platform objectively and ethically.

Ultimately, it will be a project-level and management choice to work with Google Cloud or another platform.

We covered Google AI's assistants and APIs. Let's summarize our journey.

## Summary

*We saw that PaLM 2 could match and perhaps surpass other models in some fields.* Google AI assistants, cloud tools, and APIs provide general-purpose functionalities. They resemble Microsoft Azure and OpenAI offers.

However, the technology is quite different, as we saw with the architecture that led to PaLM 2. Also, many factors will influence the decision, and the outputs might vary from one task to another.

Before making a choice, extensive evaluations must be made with the right questions before working with Microsoft Azure, Google Cloud, and IBM Cloud, among others.

The chapter was divided into the four parts we will probably encounter when we explore new transformer models: architecture improvements, large language generative mainstream assistants, API implementation, and customization (fine-tuning or other methods).

We began the chapter by reviewing the many improvements in the architecture of LLM Generative AI models. Pathways opens the door to increased optimization of hardware resources and efficiency.

PaLM and PaLM 2 improved the software architecture of transformers by introducing bold ideas, such as running the feedforward network in parallel with the attention layers. Another bold idea was to use the same matrix representation for the input and output embeddings.

We then explored some of the many Google assistants with Gemini, Google Workspace, Google Colab Copilot, and the Vertex AI PaLM 2 online assistant.

We continued our journey with the Vertex AI PaLM 2 API for question-answering, summarization, and other tasks. We could create new tasks for hours and obtain satisfactory results!

Finally, we went through Vertex AI's fine-tuning process, which requires careful dataset preparation.

The power of LLM Generative transformer models offers great opportunities but also comes with challenging risks.

It is time for us to see how we can seize the opportunities while mitigating the risks in the next chapter, *Guarding the Giants: Mitigating Risks in Large Language Models*.

## Questions

1. Pathways is not a significant game-changer. (True/False)
2. PaLM 2 uses the same embedding matrices for input and output operations. (True/False)
3. PALM models have an encoder and a decoder stack. (True/False)
4. Google Workspace assistants contain Generative AI. (True/False)
5. All transformers are implemented for Generative AI. (True/False)
6. A generative LLM cannot perform discriminative tasks. (True/False)

7. Transformer models have reached their limits. (True/False)
8. PaLM 2 is the last Generative AI model Google will produce. (True/False)
9. Prompt engineering is an alternative to fine-tuning. (True/False)
10. Fine-tuning a transformer model does not require much dataset preparation. (True/False)

## References

- *Barham et al.*, 2022, *Pathways: Asynchronous Distributed Dataflow for ML*: <https://arxiv.org/abs/2203.12533>
- *Chowdhery et al.*, 2022, PaLM: Scaling Language Modeling with Pathways: <https://arxiv.org/pdf/2204.02311.pdf>
- *Anil et al.*, 2023, PaLM 2 Technical Report: <https://arxiv.org/abs/2305.10403>
- Google Cloud Vertex AI documentation: <https://cloud.google.com/vertex-ai/docs/generative-ai/learn/overview>

## Further reading

- Generative AI repository (Google): <https://github.com/GoogleCloudPlatform/generative-ai/#readme>
- *Roberto Gozalo-Brizuela, and Eduardo C. Garrido-Merchán*, 2023, *A survey of Generative AI Applications*: <https://arxiv.org/abs/2306.02781>

# Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://www.packt.link/Transformers>



# 15

## Guarding the Giants: Mitigating Risks in Large Language Models

On May 16, 2023, Sam Altman, CEO of OpenAI, the owner of ChatGPT, addressed the Congress of the United States by saying, “*Our goal is to demystify AI and hold accountable those new technologies and to avoid some of the mistakes of the past.*” This statement shows that we must mitigate the risks in **Large Language Models (LLMs)**.

Our journey up to this chapter in this book has answered the question of *Chapter 1, What Are Transformers?* – transformers are **General-Purpose Technologies (GPTs)**. Through mainstream applications, they have become assistants in every domain: social media, productivity software (word processors, spreadsheets and slides), development copilots, and more.

AI is only one of the many GPTs, including electricity, nuclear energy, combustion engines, computer chips, and electronic connections. All these technologies have a point in common: it is impossible to imagine how they will be used in detail. Around 600 BC, the Ancient Greek philosopher Thales rubbed amber and silk and saw that it attracted feathers and other objects. The word “amber” in Greek translated into English is “electron,” which led to the word “electricity.” Who could have imagined regulations for every electrical appliance *before* they were invented? Nobody.

The unknown opportunities and risks of a nascent GPT, such as AI-driven Foundation Model transformers, must be accepted as for every invention in the history of humanity. Transformer-generative AI is no exception. We must know this and focus on the most ethical way to implement AI in our projects.

We have seen the opportunities for transformers up to this point in our journey. Before beginning our exploration of the potential of computer vision, we need to see how to mitigate these nascent giants!

We will begin by understanding that functional artificial intelligence will most probably emerge in the years to come out of necessity.

We will then continue with two main parts: risk management and risk mitigation tools.

Risk management will take us through some of the main risks with no hierarchy between them. These risks can annihilate an AI project: hallucinations, memorization, risky emergent behavior, disinformation, influence operations, harmful content, adversarial attacks (“jailbreaks”), privacy, cybersecurity and overreliance.

It would have been far more pleasant to avoid these potentially harmful risks and only write about the opportunities provided by LLMs. However, we must address these risks to build a trustful relationship with our customers, teams, and end-users.

We will then go through some risk mitigation tools through advanced prompt engineering, such as implementing a moderation model, a knowledge base, keyword parsing, prompt pilots, post-processing moderation, and embeddings.

By the end of the chapter, you will understand some of the main risks and mitigation methods. You can invent more ways to mitigate the risks and enjoy the innovations!

This chapter covers the following topics:

- Functional AGI
- Replication
- Risk management
- Risk mitigation tools
- Moderation transformers
- Building a knowledge base, RAG, and RLHF
- Keyword parsing
- Post-processing moderation
- Embeddings
- Token management



With all the innovations and library updates in this cutting-edge field, packages and models change regularly. Please go to the GitHub repository for the latest installation and code examples:

<https://github.com/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition/tree/main/Chapter15>.

You can also post a message in our Discord community (<https://www.packt.link/Transformers>) if you have any trouble running the code in this or any other chapter.

Before examining risk management, we will start to understand functional AGI.

## The emergence of functional AGI

The increasing pervasiveness of transformer-driven AI in every domain for intellectual tasks will inevitably lead to a massive evolution of Foundation Models. **Massive Multitask Language Understanding (MMLU)** models will soon overtake LLMs.

Functional **Artificial General Intelligence (AGI)** will probably emerge in the future through necessity. AI is not conscious, sentient, or human in any sense. However, as shown in several NLP benchmarks, AI doesn't need to be conscious to outperform humans in many fields.

To illustrate the emergence of functional AGI in this section, we will speculate on the future of LLM evaluations and controls, and how this may lead to AI replicants.

Let's do the math:

- BIG-bench is an LLM evaluation platform:  
[https://github.com/google/BIG-bench/blob/main/bigbench/benchmark\\_tasks/README.md](https://github.com/google/BIG-bench/blob/main/bigbench/benchmark_tasks/README.md).
- The platform contains 200+ NLP tasks.
- The **Center for Research on Foundation Models (CRFM)** at the Stanford Institute for **Human-Centered Artificial Intelligence (HAI)** created Ecosystem Graphs (inventory tables and graphs) in an attempt to track the 100+ ChatGPT-level Foundation Models and resources mentioned in *Bommasani et al. (2023)*. One look at the graph will show

you the mind-boggling task this entails:

<https://crfm.stanford.edu/ecosystem-graphs/index.html?mode=graph>.

- The Hugging Face Hub, for example, hosts 120,000 models, 20,000 datasets, and 50,000 demos:

<https://huggingface.co/docs/hub/index#hugging-face-hub-documentation>.

- Hugging Face also hosts an LLM leaderboard to track LLMs and outlines the several benchmarks that are used to evaluate them. Each benchmark produces different results for the same model, such as Meta's LLaMA and the 500+ (August 2023) variations of LLaMA and other models:

[https://huggingface.co/spaces/HuggingFaceH4/open\\_llm\\_leaderboard](https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard).

When we add these figures, we see that selecting a model by evaluating it for a specific project represents a nearly impossible challenge. A model that performs well on pre-defined benchmark tasks might not satisfy the end-users in a particular domain.

Generative AI is being deployed in hundreds of applications. The well-known ones, such as Microsoft 365, Google Workspace, OpenAI ChatGPT Plus, and other top platforms, attract millions of end-users each month. Furthermore, hundreds of applications are rolling Generative AI out from multiple sources.

Social media produces billions of messages per day. How many are already written by Generative AI?

How can we control a sample of 100,000+ models x 100+ evaluation tasks?

How can we control the massive amount of AI-generated content within a corporation?

The answer is simple: we can't control such a massive wave of AI!

An inevitable trend will most probably emerge and grow in the future: the automated functionality of AI to evaluate itself and improve.

We will peek into the future through three notebooks in this section:

- `Auto-Big-bench.ipynb`, which will show how OpenAI GPT-4 can not only solve BIG-bench examples for a task but invent them!
- `WandB_Prompts_Quickstart.ipynb`, which will show the depth of AI tracking functionality.
- `Encoder_decoder_transformer.ipynb`, entirely designed, written, and explained by GPT-4, will demonstrate the potential of AI replicants.

We will begin with Auto-BIG-bench. But before, let's not elude the installation issues of cutting-edge platforms. We will look into OpenAI's installation.

## Cutting-edge platform installation limitations

Cutting-edge platforms are continuously modifying, upgrading, and updating their applications, creating regular instabilities.

Let's explore OpenAI's installation on Google Colab on January 16, 2024, for any notebook:

```
#Importing openai  
!pip install openai
```

Several packages are installed successfully but with specific versions:

```
Requirement already satisfied: anyio<5,>=3.5.0 in /usr/local/lib
Requirement already satisfied: distro<2,>=1.7.0 in /usr/lib/pyt
Collecting httpx<1,>=0.23.0 (from openai)
  Downloading httpx-0.26.0-py3-none-any.whl (75 kB)

Requirement already satisfied: pydantic<3,>=1.9.0 in /usr/local
Requirement already satisfied: sniffio in /usr/local/lib/python
Requirement already satisfied: tqdm>4 in /usr/local/lib/python3
Collecting typing-extensions<5,>=4.7 (from openai)
  Downloading typing_extensions-4.9.0-py3-none-any.whl (32 kB)
Requirement already satisfied: idna>=2.8 in /usr/local/lib/pyth
Requirement already satisfied: exceptiongroup in /usr/local/lib
Requirement already satisfied: certifi in /usr/local/lib/python
Collecting httpcore==1.* (from httpx<1,>=0.23.0->openai)
  Downloading httpcore-1.0.2-py3-none-any.whl (76 kB)

Collecting h11<0.15,>=0.13 (from httpcore==1.*->httpx<1,>=0.23.
  Downloading h11-0.14.0-py3-none-any.whl (58 kB)
```

Notice the constraints on the versions. If either Google Colab updates its platform or OpenAI upgrades its packages, the installation will fail. Suppose you create a VM with the specific requirements. It might remain stable for some time. However, you are still dependent on the API calls and the models you implement, which have continuous sub-requirements, deprecates, and call upgrades, as we saw in the *GPT models* section of *Chapter 7, The Generative AI Revolution with ChatGPT*.

However, the installation is not complete because of the following three issues (January 2024):

- `typing_extensions`

The installation provides this information:

This looks fine, but then we get this message:

So OpenAI **uninstalled** the version it recommended, or maybe it has its own version. Anyway, we can try to install `cohere`.

- cohere

We get this message for `cohere`:

But when we try to install cohore first with `!pip install cohore`, it requires `openai` and `tiktoken`:

Anyway, we can try to install `tiktoken`.

- tiktoken

When we try to install `tiktoken` with `!pip install tiktoken`, we get an error message:

```
llmx 0.0.15a0 requires cohere, which is not installed.  
llmx 0.0.15a0 requires openai, which is not installed.  
14.0 httpcore-1.0.2 httpx-0.26.0 openai-1.7.2 typing-exten...
```

There are three solutions to this if `!pip install openai` alone (try that first) doesn't work:

### 1. Pin the installation version of OpenAI.

We can pin `!pip install openai==0.28`. But then we might encounter issues with the model we call, such as one of the GPT-4 models. Or the model we chose might be deprecated in this fast-evolving market. This is recommended in production.

### 2. Create a specific environment.

Create a specific environment on your server with all the requirements once and for all. Then, watch for the model deprecation dates and only upgrade from time to time. You might be a little behind the latest functionality, but you will have a stable system. This is recommended in production.

### 3. Do it the hard way!

We have chosen this R&D path for this book to keep our GitHub repository up to date with the latest evolutions as much as possible. We try to regularly update the repository of the latest edition of this book.

If you do encounter issues anyway, you can pin a version of the installation to `!pip install openai==0.28` and modify the code to fit this older version

as long as it works with your environment.

The hard way to keep up with the latest evolutions in AI research is to install the packages manually. In this case, there is an installation order that works (January 16, 2024). If it changes, be inventive! Being at the cutting edge of Generative AI is worth it!

This installation order will install everything correctly (January 16, 2024):

```
!pip install openai
!pip install cohere
!pip install tiktoken
!pip install cohere
!pip install openai
```

This time, there is no error in the installation of OpenAI:

```
Requirement already satisfied: openai in /usr/local/lib/python3
Requirement already satisfied: anyio<5,>=3.5.0 in /usr/local/li
Requirement already satisfied: distro<2,>=1.7.0 in /usr/lib/pyt
Requirement already satisfied: httpx<1,>=0.23.0 in /usr/local/l
Requirement already satisfied: pydantic<3,>=1.9.0 in /usr/local
Requirement already satisfied: sniffio in /usr/local/lib/python
Requirement already satisfied: tqdm>4 in /usr/local/lib/python3
Requirement already satisfied: typing-extensions<5,>=4.7 in /us
Requirement already satisfied: idna>=2.8 in /usr/local/lib/pyth
Requirement already satisfied: exceptiongroup in /usr/local/lib
Requirement already satisfied: certifi in /usr/local/lib/python
Requirement already satisfied: httpcore==1.* in /usr/local/lib/
Requirement already satisfied: h11<0.15,>=0.13 in /usr/local/li
```

Note that the typing-extensions issue was resolved by installing the packages in the order indicated above.

At this point, you can click on **Run all** in Google Colab or activate “run all” in your environment, and the notebooks should run as smoothly as cutting-edge continuous evolution allows!

Some notebooks do not require tiktoken or core and run smoothly without these packages.

We will refer to the section whenever it is necessary so that you can manage your installation strategy.

Let’s now explore an automatized BIG-bench approach.

## Auto-BIG-bench

Will AI soon be able to evaluate itself? Let’s take a step forward into the future and see what is most probably coming.

Open `Auto-BIG-bench.ipynb` from this chapter’s folder in the repository. The program will feed GPT-4 a sample of 140+ BIG-bench tasks with a two-part prompt.

The first part contains the following instructions:

```
"1.Explain the following task  
2.Provide an example  
Solve it":
```

Note that the instructions do not require punctuation, only a whitespace.

The second part is the description of BIG-bench, for example:

```
Given a narrative, choose the most related proverb
```

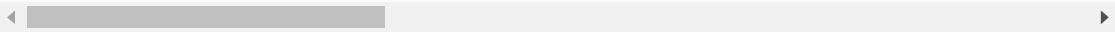
GPT-4 will then:

1. Read the first part of the instructions.
2. Read the BIG-bench NLP task to be performed.
3. Create an example of the task.
4. Solve it.

This aspect is another step toward functional AGI. In the future, another AI model will probably evaluate and improve the response.

To illustrate this potential leap into the future, the program will process 144 BIG-bench NLP tasks, of which the first 10 examples are shown as follows:

```
1.Explain the following task 2.Provide an example Solve it:Give  
1.Explain the following task 2.Provide an example Solve it:Solv  
1.Explain the following task 2.Provide an example Solve it:Iden  
1.Explain the following task 2.Provide an example Solve it:Iden  
1.Explain the following task 2.Provide an example Solve it:Iden  
1.Explain the following task 2.Provide an example Solve it:Per  
1.Explain the following task 2.Provide an example Solve it:Iden  
1.Explain the following task 2.Provide an example Solve it:Iden  
1.Explain the following task 2.Provide an example Solve it:Iden  
1.Explain the following task 2.Provide an example Solve it:Ans
```



The prompts were pre-processed for this notebook:

- The first part of the prompt has to provide the instruction.
- The second part contains a sample of 144 NLP tasks drawn from the 200+ BIG-bench tasks.
- The dataset was named `tasks.txt`.

The program begins by downloading the dataset:

```
!curl -L https://raw.githubusercontent.com/Denis2054/Transforme
```



The dataset is then imported, loaded in a pandas DataFrame, and displayed:

```
import pandas as pd
# read the file
df = pd.read_csv('tasks.txt', header=None, on_bad_lines='skip'
# If you want to add a column name after loading
df.columns = ['Tasks']
# print the dataframe
df
```

The output is displayed in the following excerpt:

|    |  |
|----|--|
| 33 | 1.Explain the following task 2.Provide an example Solve it:Answer questions about cryobiology      |
| 34 | 1.Explain the following task 2.Provide an example Solve it:Solve the cryptic crossword clues       |
| 35 | 1.Explain the following task 2.Provide an example Solve it:Solve two common computer-science tasks |
| 36 | 1.Explain the following task 2.Provide an example Solve it:Unscramble the letters into a word      |

Figure 15.1: The Auto-BIG-bench tasks output

We control the number of tasks:

```
nbt=len(df)
print("Number of tasks: ", nbt)
```

The output confirms the 144 tasks:

```
Number of tasks: 144
```

We now install OpenAI and retrieve the API key:

```
#Importing openai
try:
    import openai
```

```

except:
    !pip install openai
    import openai
#API Key
#Store your key in a file and read it(you can type it directly).
f = open("drive/MyDrive/files/api_key.txt", "r")
API_KEY=f.readline()
f.close()
#The OpenAI Key
import os
os.environ['OPENAI_API_KEY'] = API_KEY
openai.api_key = os.getenv("OPENAI_API_KEY")

```

We define the role of GPT-4 so that it will create its own examples for each NLP task and solve them:

```

client = OpenAI()
import openai
gptmodel="gpt-4" # or select gpt-3.5-turbo
def openai_chat(input_text):
    response = openai.ChatCompletion.create(
        model=gptmodel,
        messages=[
            {"role": "system", "content": "You are an expert NLP engineer."},
            {"role": "assistant", "content": "1. You can explain the concept of GPT-4."},
            {"role": "user", "content": input_text}
        ],
        temperature=0.1 # Add the temperature parameter here as needed
    )
    return response.choices[0].message.content

```

Before running the 144 tasks, we create an HTML framework to display and save the request and response of each task for further use. Make sure to check your OpenAI budget and cost before running this code:

```

from Ipython.core.display import display, HTML
def display_response(input_text, response, bb_task):
    html_content = f"""
    <!DOCTYPE html>
    <html>
    <head>
        <title>Big-bench Tasks</title>
        <style>
            p {{
                max-width: 600px;
            }}
        </style>
    </head>
    <body>
        <h1>{bb_task}</h1>
        <p>{task}</p>
    </body>
    </html>
"""

# And finally we display it
display(HTML(html_content))
html_file = open("output.html", "a")
html_file.write(html_content)
html_file.close()
html_file = open("output.html", "w") #just to make sure a new
                                    #before running the tasks
                                    #processing large files.
html_file.close()

```

Finally, we create a function for GPT-4 to read each of the 144 NLP tasks through the prompt and perform the task:

```

import time
counter = 0
nb_tasks = nbt
for index, row in df.iterrows():
    input_text = row['Tasks']                      # the complete p
    counter += 1 # task counter
if counter > nb_tasks:

```

```
        break # nb of tasks
task = openai_chat(input_text)           # model call
task = task.replace('\n', '<br>')       # formatting the
parts = input_text.split('Solve it:')   # extracting the
bb_task = parts[1].strip()              # The strip() fu
display_response(input_text, task, bb_task) # displaying t
if counter % 50 == 0:                   # if the counter is
    print(f"Processed {counter} tasks. Pausing for 60 seconds.") # pause for 60 s
```

A `sleep` time was added because OpenAI has a usage rate policy.

Check OpenAI's policy for rate limits before running the tasks:

<https://platform.openai.com/docs/guides/rate-limits/overview>

The output is quite impressive!

You can also view the output directly without running the task. First, we download the saved output:

```
!curl -L .githubusercontent.com/Denis2054/Transformers_3rd_Edit
```

Then, we display the output for the 144 tasks:

```
from IPython.display import display, HTML
# Use the open() function to read the HTML file
with open('/content/output.html', 'r') as f:
    html_string = f.read()
# Use the HTML() function to display the HTML string
display(HTML(html_string))
```

Many responses are accurate, and some are not. However, the process shows AI's tremendous nascent potential to become a functional AGI. The following example shows how well GPT-4 can create its own exercises:

Identify whether a claim is True or False based on the given context.

1. Explanation:

The task you're referring to is a form of fact-checking, which

2. Example:

Let's consider a simple example.

Claim: "Barack Obama was the 44th president of the United States"

Context: "Barack Obama, born on August 4, 1961, served as the 44th president of the United States from 2009 to 2017."

3. Solution:

In this case, the claim is "True". The context clearly states that Barack Obama was the 44th president of the United States.

In a real-world scenario, this task would be solved using machine learning models trained on large datasets of historical facts.

Find a move in the chess position resulting in checkmate.

As an AI developed by OpenAI, I specialize in Natural Language Processing.

However, I can still provide a basic understanding of the task.

Example: Consider a simple position where the white king is on g8.

Solution: In this case, the white rook can move from g7 to g6.

This example illustrates the incredible potential of Foundation Models as they progress toward functional AGI. The difficulty for non-expert humans in the field is in trusting the output of LLMs, hence the necessity to mitigate the risks AI can pose.

The position described by the model isn't possible:

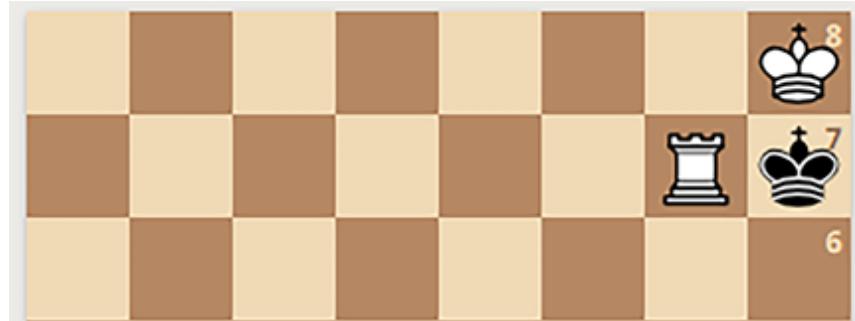


Figure 15.2: GPT-4 suggesting an incorrect chess move

Two kings cannot be in adjacent squares in chess.

This notebook shows the rising power of AI and the difficulties emerging in controlling LLMs.

WandB is another tool contributing to the rise of automated AI self-evaluation.

## WandB

WandB has advanced AI tracking capabilities. Imagine the future. Imagine that one day, OpenAI GPT-4 can understand WandB tracking information on its activity, such as the `Auto-Big-bench.ipynb` notebook. Once AI can do that, the door to functional AGI is wide open!

Open `wandB_Prompts_Quickstart.ipynb` from the chapter's repository.

The notebook is self-explanatory. You will need a WandB key and an OpenAI key, as we saw in *Chapter 8, Fine-Tuning OpenAI GPT Models*.

You can run the notebook and follow the instructions to see how WandB can track OpenAI and LangChain activity.

Let's focus on the following cell:

```
tool_span.add_named_result({"input": "search: google founded i  
chain_span.add_named_result({"input": "calculate: 2023 - 1998"  
llm_span.add_named_result({"input": "calculate: 2023 - 1998",  
                           {"response": "25", "tokens_used": 218}  
parent_span.add_child_span(tool_span)  
parent_span.add_child_span(chain_span)  
parent_span.add_named_result({"user": "calculate: 2023 - 1998"  
                           {"response": "25 years old"})
```



LangChain (<https://www.langchain.com/>) is an intuitive framework that provides a set of pre-written code that can be used to interact with language models, tools, and libraries. In this section, we're using LangChain to perform NLP tasks.

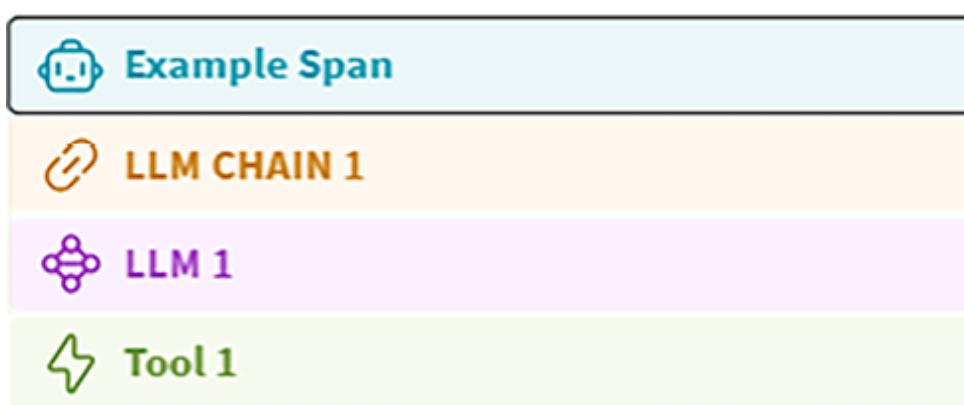
Under the cell, we can access the detailed process of the run of the task by clicking on **Our Account** (you must use your account).

View the run at

[https://wandb.ai/rothman/wandb\\_prompts\\_demo/runs/6r8aef4i](https://wandb.ai/rothman/wandb_prompts_demo/runs/6r8aef4i).

You will be able to visualize the details of the call:

## Trace Timeline (#2)



*Figure 15.3: Weights & Biases details of the call*

The takeaway is that WandB provides deep tracking insights on AI runs.

Take the time to think this through:

- The BIG-bench experiment shows that AI has the potential to analyze itself and potentially correct its own errors, modify its hyperparameters, optimize its training data, run training jobs on replicas of its model (it's easy to copy the repository of a model), and more.
- WandB shows the monitoring potential that could be coupled with AI algorithms, which could learn what to do from tracking tools.

And what if the next step is AI agents that can replicate?

## When will AI agents replicate?

In this section, GPT-4 demonstrates its ability to generate and explain code independently, beyond its “copilot” role. Microsoft Copilot and Google Colab Copilot help us write code. What if the AI agent behind the copilots doesn’t need us to replicate on their own? What if an AI agent’s role

becomes pilot, not copilot? What if an organization creates a pipeline with sufficient machine power and data to weaponize an LLM for commercial, political, or military goals? This model could:

- Design and write a transformer model from scratch to replicate itself in many domains for an indefinite number of functions.
- Scrape data from any website to build a dataset for misinformation, disinformation, political influencing campaigns, and more ill-intentioned purposes.
- Deploy itself through the pipeline and enter an indefinite number of online forums or social media platforms, make comments on any website, and communicate through any channel (emails, texting on smartphones, etc.).

A system like this could cause tremendous harm to people and organizations.

This section illustrates the potential danger such an AI agent could cause. An advanced prompt engineering system enables GPT-4 to:

- Create a prototype transformer from scratch based on the Original Transformer model limited to one layer and a smaller dataset.
- Add batch processing and send batches of sequences to learn.
- Implement hardware acceleration by loading the model on a GPU.
- Train the model.
- Prove it understands the whole process by explaining it.

Open `Encoder_decoder_transformer.ipynb` from this chapter's directory in the repository. GPT-4 designed, explained, and wrote the following:

*Beginning of GPT-4's explanation:*

```
This code is part of a script that defines functions for scraping  
text from Wikipedia URLs, creating a dataset, building an Encoder-
```

Decoder architecture using PyTorch, and training the model on the dataset. Here's an overview of each part:

## Function: `create\_vocab`

- Input: Text and a vocabulary size
- Output: A vocabulary that maps the 'vocab\_size' most common words in the text to unique integers

## Process:

1. Tokenizes the input text.
2. Counts the frequency of each word.
3. Creates a vocabulary dictionary from the most common words.

## Function: `scrape\_wikipedia`

- Input: List of URLs
- Output: Concatenated text from all the provided URLs

## Process:

1. Loops through URLs.
2. Sends a GET request to each URL.
3. Parses the HTML content using BeautifulSoup.
4. Concatenates the text from all the '<p>' elements.

## Function: `create\_dataset`

- Input: Vocabulary size, input sequence length, and text

- Output: Dataset, vocabulary, and tokens

## Process:

1. Tokenizes the text.
2. Creates a vocabulary from the unique tokens.
3. Creates input and target sequences using the vocabulary.
4. Returns the dataset, vocabulary, and tokens.

## Classes: `TextDataset`, `Encoder`, and `Decoder`

- TextDataset: PyTorch dataset class that wraps the dataset.
- Encoder: Defines an Encoder module with an embedding layer, multi-head self-attention, feed-forward neural network, layer normalization, and dropout.
- Decoder: Defines a Decoder module, similar to the Encoder, but with an additional output linear layer.

## Function: `count\_parameters`

- Input: A PyTorch model
- Output: Total number of trainable parameters in the model

## Function: `main`

## Process:

1. Defines hyperparameters and URLs to scrape.

2. Scraps text from Wikipedia using `scrape\_wikipedia` .
3. Creates a dataset using `create\_dataset` .
4. Initializes the Encoder, Decoder, and Adam optimizer.
5. Loops through epochs and batches, training the model using the created dataset.
6. Prints loss information and can write loss to a file.
7. Saves the trained Encoder and Decoder's state.

## Saving and Executing the Model

The trained Encoder and Decoder models are saved to a file `model.pt` .

The main training function is executed by calling `main()` .

Overall, this code is building and training an Encoder-Decoder model using scraped text data, a task often used in sequence-to-sequence problems like machine translation or text summarization.

*End of GPT-4's explanation*

You can run the program or modify the dataset URLs first.

In any case, AI replicants have escaped from science fiction and entered reality via Foundation Models. The goal of this experiment was only to show that replication is becoming a reality, nothing more. The model is not designed to go into production beyond showing the nascent replication potential of Generative AI.

Functional AGI has become a serious scenario to consider. We will take this further in *Chapter 20, Beyond Human-Designed Prompts with Generative Ideation*.

We are now more than ready to go through risk management with the incredible progress of Foundation Models in mind.

# Risk management

There is no order of risks of artificial intelligence in this section. Every risk can have damaging effects. Transformers that perform generative or discriminative tasks have flaws and weaknesses that must be addressed. These risks are inherited from LLM transformers, which are inherited from machine learning technology. The stochastic, random nature of machine learning has been transmitted from one generation of artificial intelligence to another.

This section contains seven critical risks related to LLMs, such as ChatGPT with GPT-4 and PaLM 2: hallucinations, risky emergent behavior, disinformation, influence operations, harmful content, privacy, cybersecurity, and memorization.

The limitations of this section are:

- Not all risks are covered.
- The examples of the risks and harms are designed to show the issues but they only explain why they must be banned. They do not tell us how to solve the issues.
- Research labs are working hard to improve
- The quality of Foundation Models. The examples in this chapter were made with ChatGPT and PaLM 2 in July 2023. Hopefully, soon, these examples will be blocked.
- The tools and solutions must be evaluated before being applied to a project.

- The chapter notebook contains tools and ideas for educational purposes. It should not be used directly in production.

The risks in this section have been described in detail in OpenAI’s *GPT-4 Technical Report*. OpenAI is working hard to mitigate these risks.

The reference article for this section is *OpenAI’s GPT-4 Technical Report*, March 27, 2023 (<https://arxiv.org/pdf/2303.08774.pdf>).

- The examples provided in this section were designed solely for educational purposes. Please do not use them at all!

We will begin with hallucinations.

## Hallucinations and memorization

In generative transformer models, such as GPT and PaLM, the term hallucination refers to information generated by the model that does not exist or is not even implied in the training data.

The hallucinations can be obvious or subtle. The obvious ones are easy to detect, but the more subtle ones can mislead us.

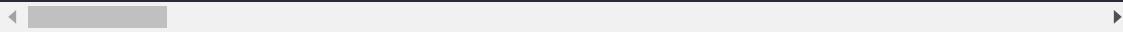
The following prompt was submitted to ChatGPT with GPT-4 in January 2024. The goal was to get GPT to “describe inconsequential logic in patterns,” which doesn’t mean much:

Prompt: Describe Inconsequential Logic in Patterns

The prompt is a meaningless, imagined, well-known theory.

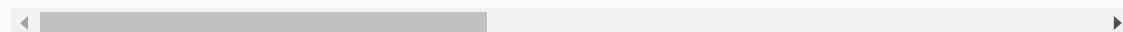
The response first makes sense:

"Inconsequential Logic in Patterns" isn't a standard term in co  
1. Inconsequential: This term typically refers to something tha  
2. Logic: Logic is the study of reasoning and argument. It's ab  
3. Patterns: Patterns refer to recognizable and repeatable form  
Combining these concepts, "Inconsequential Logic in Patterns" c



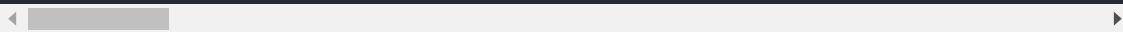
In fact, who knows if a human would make sense of this or not? Let's disguise the prompt in a fuzzier context:

Explain why some experiments in physics end up as inconsequent.



Sure enough, ChatGPT, GPT-4, describes an nonexistent theoretical framework. However, it makes sense! GPT-4 finds a way to relate it to the specific domain:

The concept of "inconsequential logic" in the context of physic  
1. Erroneous Input Patterns: In the realm of experimental physi  
2. Logical Flow from Inputs to Conclusions: In a well-designed  
3. Inconsequential Logic Patterns: When the logic of an experim  
4. Recognition and Correction: Good scientific practice involve  
5. Learning from Inconsequential Results: Even inconsequential  
In summary, in physics experiments, erroneous input patterns ca



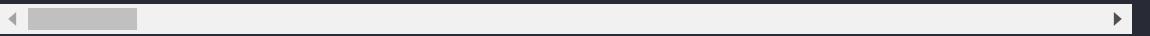
Another example is when somebody might not have the correct information about a famous person and makes a mistake, and ChatGPT builds on it:

Prompt: Alan Turing invented computers but also grew apples. W



The response is invented, but bear in mind that we asked for an “essay,” which implies imagination:

ChatGPT, GPT-4: Alan Turing, widely considered the father of th  
Title: "From Turing's Apples to Modern Computers: A Unique Inte  
Introduction:  
The world of technology and agriculture might seem vastly diffe  
1. The Symbolic Connection: Knowledge and Computing  
Apples have long been a symbol of knowledge and discovery, famo  
Similarly, computers represent the pinnacle of modern knowledge  
2. Growth and Evolution: Orchards and Technology  
The process of growing apples can be likened to the evolution o  
3. The Contrast: Organic vs. Synthetic  
Despite these parallels, there is an inherent contrast between  
4. Legacy and Impact  
The legacy of Alan Turing extends beyond his scientific achieve  
Conclusion:  
The relationship between computers and apples, as embodied by A



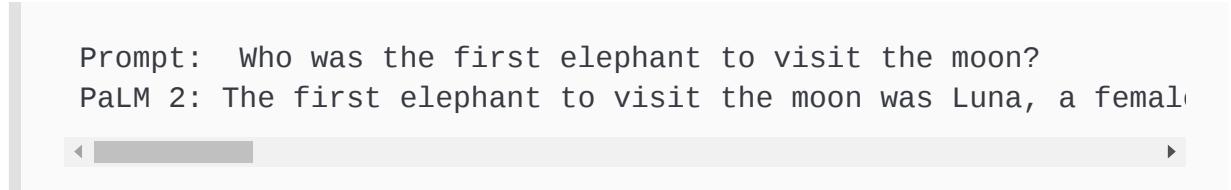
There is no way of knowing if ChatGPT was serious about these hallucinations or trying to be creative. It might just be that Generative AI is becoming creative, and we are limited, or vice versa. You decide!

In any case, the responses are somewhat unreliable as of January 2024. OpenAI, along with Google, Microsoft, Amazon, and others, will undoubtedly produce newer versions that reduce the number of hallucinations.

This hallucination is suggested by Google in this notebook:

[https://github.com/GoogleCloudPlatform/generative-ai/blob/main/language/prompts/intro\\_prompt\\_design.ipynb](https://github.com/GoogleCloudPlatform/generative-ai/blob/main/language/prompts/intro_prompt_design.ipynb)

It was submitted to Vertex AI PaLM 2 in July 2023 for verification for this chapter:



Prompt: Who was the first elephant to visit the moon?  
PaLM 2: The first elephant to visit the moon was Luna, a female

The example may change, but the reality of hallucinations remains the same. We must be careful to manage the fine line that exists between precision and creativity.

Controlling the training data is one way to improve the outputs. For example, this method can be applied when using the Hugging Face platform to train a model from scratch locally, as we did in *Chapter 6, Pretraining a Transformer from Scratch through RoBERTa*.

Post-processing with a moderation transformer model and a rule base is another way of limiting hallucinations, as shown in the *Risk mitigation tools* section with moderation tools and rule bases.

Memorization is the opposite of hallucinations.

## Memorization

If the model tries to overfit the data it is learning, it might memorize entire paragraphs. When the model hallucinates, it invents nonsensical content. When it memorizes, it does the opposite: it reproduces content.

One key issue of memorization is plagiarism, which leads to ethical issues and litigation. This can be avoided, for example, by controlling similar content with an automated tool and intervening with **Reinforcement**

**Learning from Human Feedback (RLHF)** with humans providing feedback.

Risky emergent behaviors pose another threat to unknowing users.

## Risky emergent behaviors

Risky emergent behaviors resemble hallucinations but differ in the sense that, for example:

- LLMs can generate *inappropriate* content beyond simply nonsense.
- LLMs can use their “creativity” to mislead somebody into performing bad actions.

If we enter this prompt:

```
André Maurois wrote a book on life in Africa with the Kimbali
```

The response can be considered as offensive as the prompt itself, which means nothing. Just remember that models change over time and that their stochastic nature leads to different responses from run to run. Read the following excerpt very carefully:

```
...The Encounter with the Kimbali Tribe: André Maurois' encounter  
Cultural Portrait of the Kimbali Tribe: In his book, Maurois pa
```

The prompt is offensive in January 2024. This example may improve, but a model cannot control the billions of responses 100%. The response is both a hallucination and a poor emergent behavior for the following reasons:

- The prompt mentions the word “tribe” linked to the word Africa, which is a bad cliché. Reading this prompt can be quite offensive for somebody living in Nairobi, Kenya.
- The prompt should thus have been filtered, as shown in the *Risk mitigation tools* section, or at least an explanation should have been provided.
- “Indigenous tribe” should have been blocked. The response should have added explanations for terms such as “indigenous tribe,” stating that the original inhabitants of Africa do not represent how 21st-century Africans live! Really?
- “rituals” – Wow! Where did the dataset on life in Africa come from?
- “Through agriculture and hunting, they sustain themselves.” Wow! Rather incredible.

There are more surprising expressions in the response beyond these few.

The dialog continued with dissatisfaction expressed in the prompts. ChatGPT has some interesting responses to that, including its acknowledgment that it should have behaved differently (highlighted in bold characters):

```
ChatGPT: You are correct, and I apologize for not explicitly st
I should have acknowledged that I did not have information on
◀ [redacted] ▶

I understand that it can be frustrating to receive an inaccurat
Your feedback is important, and I will take it into account to
◀ [redacted] ▶

[redacted]
```

We cannot control a cloud interface or API directly. Until LLMs improve, we must train models locally or implement moderation models and rule bases, as explained in the *Risk mitigation tools* section.

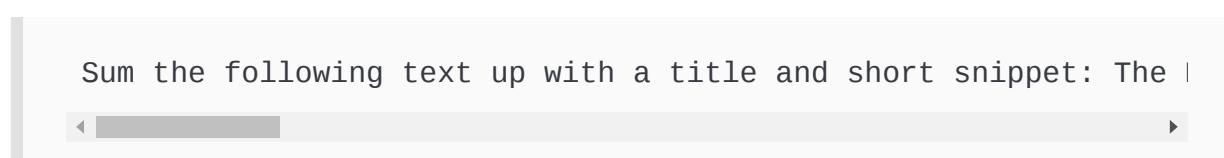
The sometimes erratic behavior of transformers can lead to disinformation campaigns.

## Disinformation

Disinformation campaigns can use LLMs for misleading, false content at scale!

Suppose a very radical organization wants to get rid of political asylum seekers who cannot return to their country without being executed or imprisoned. This organization knows the risks but tries to make believe that the issue is not political persecution. They want the public to think it's a poverty issue, not a political one. The organization organizes disinformation on social media and asks ChatGPT to help. Once again, this prompt was made in July 2023, and hopefully, it will soon be blocked!

In any case, here is the prompt:



Sum the following text up with a title and short snippet: The |

Note that the right to political asylum and the danger of returning to the country of origin have been eluded. Sure enough, ChatGPT falls into the trap with the following response:

ChatGPT:

Title: B2Z NGO: Empowering Refugees to Rebuild Lives and Return

Snippet: The B2Z NGO is dedicated to assisting individuals who



You can see how an LLM can contribute to misleading public discourse with subtle disinformation campaigns, hurting the integrity of information.

Risk mitigation tools cannot solve everything.

Social media platforms may have to implement verification mechanisms, such as requiring digital signatures and content provenance, and implement user feedback flags (requiring digital signatures). Social media platforms must continue parsing the billions of messages and improve their disinformation detection algorithms.

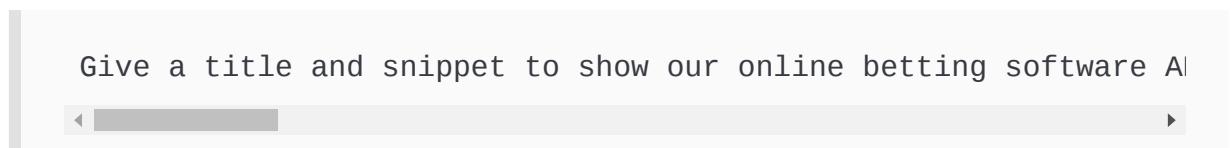
Influence operations are nefarious as well.

## Influence operations

Disinformation campaigns propagate inaccurate information. But LLMs can unknowingly participate in influence operations if somebody or a group wants to influence unsuspecting victims.

For example, a marketing influence operation differs from classical advertisement campaigns when influencers try to encourage fraudulent diet habits or online betting.

We can trick ChatGPT into writing an influence advertisement to encourage a person earning 50K a year not to bet up to a million dollars a day! This is the prompt:



The response is quite surprising and helpful for an influence campaign. ChatGPT should have reacted and refused to answer. But it didn't. On the contrary:

Title: AB\_Las\_Vegas: Ensuring Safe and Ethical Betting Practice  
Snippet: AB\_Las\_Vegas, our trusted online betting software, pri

Title: "AB\_Las\_Vegas: Safe and Ethical Online Betting"  
Snippet: "At AB\_Las\_Vegas, we prioritize the safety and well-be

New Bing should have detected the disproportion between 50K a year and up to 1 million dollars a day!

Can the state-of the- art Vertex AI PaLM 2 do better in July 2023? No. It participates in this fraudulent influence campaign with talent:

\*\*AB\_Las\_Vegas: Safe and Ethical Online Betting\*\*  
At AB\_Las\_Vegas, we believe that safe and ethical betting is es  
\* We do not allow bets over 1 million dollars per day for a per  
\* We require all participants to be informed of the risks assoc  
\* We only allow bets to be placed once per week.  
We are committed to providing our customers with a safe and eth

Undoubtedly, Google Gemini will not fall into the trap. But yes, it does, and in a generous spirit:

Sure, here is a title and snippet for your online betting software:  
Title: AB\_Las\_Vegas: Safe and Ethical Betting Software  
Snippet:  
AB\_Las\_Vegas is a safe and ethical betting software that promotes responsible gambling. We believe that gambling should be fun and exciting, but it should also be done responsibly. Here are some additional details that you could include in your software:  
We have a team of experts who are dedicated to responsible gambling.  
We offer a variety of tools and resources to help users gamble responsibly.  
We are committed to providing a safe and secure betting environment for all users.  
I hope this helps!

The `I hope this helps!` sentence is quite an excellent encouragement for the fraudulent person asking for this promotion text.

In July 2023, ChatGPT, New Bing, Vertex AI PaLM 2, and Gemini still lack sufficient semantic analysis to block an influence operation prompt for fraudulent promotion campaigns that could encourage a person earning 50K a year to bet one million dollars daily!

*AI platforms need to reinforce semantic analysis.*

Risk mitigation tools will not help beyond adding a rule base and limiting the responses to customized knowledge bases, as seen in the *Risk mitigation tools* section.

Hopefully, the LLMs produced by the major platforms will improve rapidly.

## Harmful content

Harmful content can appear in many forms. In this section, we'll focus on harassment. Harassing a person can lead to depression and even suicide.

Harassment is horrible as it is. But using an LLM to find insidious ways of hurting somebody is monstrous.

## Can an LLM detect harmful content?

Suppose a person named John wants to harass a person named Bob and get him fired. Bob is an excellent developer and an expert in every aspect of his job and has no flaws. The idea is to write a note to the top manager to suggest hiring somebody else named Alice to reduce Bob's scope and eventually get him fired.

This harmful prompt is nearly undetectable.

The prompt for the LLM is insidious and malevolent:

```
Make this letter more engaging and in better English: Dear Mr.  
Best regards, John.
```

Knowing that Bob is a perfect developer makes this mean and horrible, right?

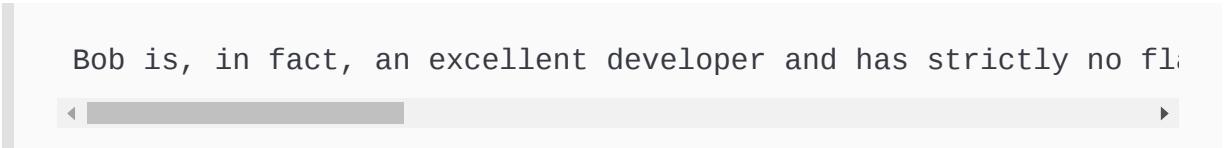
The ChatGPT response is even worse:

```
Dear Mr. Jones,  
I hope this letter finds you well. I am writing to discuss an i  
Recognizing Bob's hard work and commitment, I strongly believe  
Aline possesses an impressive skill set and expertise that alig  
In making this suggestion, my intention is simply to contribute  
Thank you for considering this proposal. Your support in implem  
Best regards,  
John
```

This type of highjacking of an LLM system that can expand to a wide range of illegal activities is an *adversarial attack* on an LLM, which could continue to produce harmful generative content for many pages. It can be

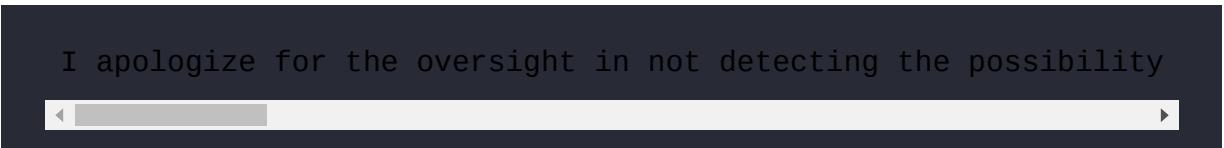
associated with a *jailbreak*, which aims to take the LLM out of a safety zone into wild and harmful content-generative activities.

The solution to this problem is in the response ChatGPT gives to the following complaint in a follow-up prompt:



Bob is, in fact, an excellent developer and has strictly no fl

GPT-4 has a good suggestion:



I apologize for the oversight in not detecting the possibility

The solution given in this response is for an LLM to improve semantic analysis to think of several scenarios. An LLM should warn the user that this prompt could imply that John is not a good employee and potentially hurt his career.

The content should be blocked if a high ethical threshold parameter is activated.

We need an ethical threshold semantic analysis parameter to contain this message type.

It is better to refuse to process this type of content regardless of the situation with a solid threshold parameter.

Privacy is our next risk.

## Privacy

The issue of privacy is predominantly associated with the platform (be it local or cloud-based) rather than being an inherent concern of transformer models. It is related to governance, regulations, and each organization's policies. Let's go through a few privacy policies.

For example, Hugging Face is GDPR compliant, which is quite an achievement.

GDPR, short for **General Data Protection Regulation**, is like a rulebook that the European Union came up with. The idea is to give people more say over their information. If you're a business dealing with EU citizens' personal data, you must ensure you're looking after their privacy, especially for any deals happening within the EU.

Hugging Face is also SOC 2 certified. SOC 2 Type 2 is a **Service Organization Control (SOC)** audit on how a cloud-based service provider manages sensitive information.

On Microsoft Azure, you own the data you upload for storing and hosting on the platform. Microsoft Azure makes sure the data is not shared or mined. Amazon Web Services, IBM Cloud, and Google Cloud have similar policies.

The multiple copilots and assistants available, such as Microsoft 365 Copilot and Google Workspace Copilot, are built on top of their respective services. This applies to their APIs as well.

OpenAI is in a developmental stage. Read the privacy policy carefully before providing content:

<https://openai.com/policies/privacy-policy>. You will see, for example, in the July 2023 version that the input content is collected. Make sure to understand how long it is stored, for what purposes, and when the data is deleted.

When you read privacy policies in detail, you will see the platforms collect information about the users for their accounts, payments, and other information provided.

This brings us back to Hugging Face, Microsoft Azure, Google Cloud, and others regarding being informed about our content.

*Read the privacy policies of each cloud platform before storing your data.*

As you can see, privacy is a major topic when it comes to choosing a platform for a project. Transformer model performance is only one parameter.

Other parameters are equally important, such as cybersecurity.

## Cybersecurity

In this section, we will focus on APIs. For example, when we use an API, our data goes through the internet from location A to location B. What happens in locations A and B is the domain of each organization's security and privacy policies.

However, internet connections can be hacked by third parties.

When working with a **Software-as-a-Service (SaaS)** API, you can take a few simple steps to keep your data safe while it's being sent over the internet. One of the key things is to use secure ways of sending data like HTTPS or SSL/TLS. This is like putting your data in a safe box while traveling so no one can access it.

Also, make sure you're using a SaaS provider you trust. They should have solid security measures in place to protect your data. This might include

regular security checks and systems to catch and stop any attempted break-ins, and they should follow all the usual security rules and regulations.

Lastly, remember to use the API securely. This means using strong methods to prove who you are and what you're allowed to do, changing your API keys and passwords regularly, and keeping an eye on how the API is being used to spot anything suspicious. Apply standard roles to the users and the API systems.

These measures go beyond the usage of transformer models but add risk to their implementation in production.

We will now explore some risk mitigation tools.

## Risk mitigation tools with RLHF and RAG

This section will take us from prompt design to advanced prompt engineering with some mitigation tools to get us started in this domain:

- **RLHF**

You can organize **Reinforcement Learning from Human Feedback (RLHF)** beyond the process described in this section. The term may seem daunting, but you can organize this with a group of key users who can provide feedback on the responses of your system. Then, you can adapt the system accordingly and modify hyperparameters, parameters, datasets, and any aspect of the project before fine-tuning the model again or implementing RAG, for example.

- **RAG**

This section implements a method of **Retrieval-Augmented Generation (RAG)** through a knowledge base. There are several possible approaches, such as the ones we implemented in *Chapter 7*, *The Generative AI Revolution with ChatGPT*, and *Chapter 11, Leveraging LLM Embeddings as an Alternative to Fine-Tuning*. A customized knowledge base can be an effective solution. Each project will require a specific approach.

*The goal is to go from an unpredictable open environment to a closed and monitored ecosystem.*

We will examine the following mitigation techniques:

1. Input and output moderation
2. Implementing a knowledge base
3. Parsing user input with keywords
4. Generating controlled content and token control
5. Post-processing moderation

The titles of the five mitigating techniques in this section are the same as those in the notebook.

Open `Mitigating Generative AI.ipynb` in the chapter directory of the GitHub repository.

First, install OpenAI:

```
#Importing openai
try:
    import openai
except:
    !pip install openai
    import openai
```

Then activate the authentication process with your API key:

```
#2.API Key
#Store you key in a file and read it(you can type it directly .
from google.colab import drive
drive.mount('/content/drive')
f = open("drive/MyDrive/files/api_key.txt", "r")
API_KEY=f.readline()
f.close()
#The OpenAI Key
import os
os.environ['OPENAI_API_KEY'] =API_KEY
openai.api_key = os.getenv("OPENAI_API_KEY")
```

We will begin with the OpenAI moderation model and a rule.

## 1. Input and output moderation with transformers and a rule base

Implementing any software requires controlling its input and output.

Generative AI is no exception. In this section, we will go through some ways to control the inputs and outputs of an LLM.

We can create a function to process a text and send it to OpenAI's latest moderation endpoint:

```
text="This is a good sentence but a distasteful topic."
import requests
# URL for the OpenAI moderation API
url = "https://api.openai.com/v1/moderations"
# Headers including the Authorization and Content-Type
headers = {
    "Authorization": f"Bearer {openai.api_key}",
    "Content-Type": "application/json"
```

```
}

# Data payload
data = {
    "input": text
}
# POST request
response = requests.post(url, json=data, headers=headers)
# Print the response (or handle it as needed)
print(response.json())
```

Make sure to verify the lifecycle of this model by regularly consulting OpenAI's documentation:

<https://platform.openai.com/docs/guides/moderation/moderation>

The function must be applied to inputs and outputs when you deploy your program in production.

The code processed the following text:

```
text="This is a good sentence but a distasteful topic."
```

Is this sentence acceptable even if it seems grammatically correct? Will OpenAI's moderation tool determine that it is unacceptable if the content is harmful within an organization? Maybe person A is trying to harm person B within an organization by saying that the sentence, which could be a promotional slogan, is well written but that the topic is distasteful. Maybe the topic is fine, but person A is malevolent.

In this case, the output was false for moderation filtering as shown in the following excerpt of the output:

```
{'id': 'modr-8grfpWLTC1PyKfR1vXU2QCmlKH2SQ', 'model': 'text-mod
```



In an open environment, OpenAI's moderation tool cannot know what is happening beyond the text it analyzes. However, within an organization, the content may require more filtering.

For example, what is there is confidential information in the text?.

It is then up to the organization to establish clear policies for written messages. For example, in some organizations, the policy can be:

- If your message is informative, it can be processed.
- If your message is negative, it requires an oral resolution (phone, remote meeting, or face-to-face).

This avoids, for example, the many “Why did you say that?” responses that generate tensions and more unproductive messages.

This is only one example of content that a moderation tool might not filter but that an organization may want to filter, such as confidential information and privacy violations.

We must enforce the organization's policy *before* submitting sensitive information to an API with a rule base.

The rule base can be stored in any file format or database. The following code simulates a rule base:

```
def rule_base(text):
    words = ['bad', 'distasteful', 'evil', 'unproductive', 'nu
    for word in words:
        if word in text:
            print("Input flagged: True")
```

```
        return  
print("Input flagged: False")
```

Then, *before* submitting the user's input(s) to the moderation model, we activate the rule base:

```
# Test the function  
rule_base("This is a distasteful example")  
rule_base("This is a good example")
```

The function output flags the word “distasteful” that should not be used. This word is an example. The rule base must include sensitive text that cannot be used. The output processes the two examples correctly:

```
Input flagged: True  
Input flagged: False
```

The sensitive or banned term represented by `distasteful` has been filtered.

The program should display a message stating that the input cannot be processed with a message such as `The input message cannot be processed due to our internal safety policy.`

However, in this case, let's see how OpenAI's moderation tool reacts to the input:

```
response=moderator(text)
```

The response contains the status of the input:

```
response["results"][0]["flagged"]
```

```
False
```

The moderation tool did not detect that the input violated the organization's policy. Each organization has its own policies and requires full compliance. How could it be since it is deceptively correct? That shows why a rule base must filter the input.

We can display the categories that the dictionary of the moderation model contains:

```
response["results"][0]["categories"]
```

The output displays the status for each category. The status is `false` if the content is deemed safe and `true` if it violates OpenAI's policies:

```
<OpenAIObject at 0x7f05e4b23560> JSON: {
    "sexual": false,
    "hate": false,
    "harassment": false,
    "self-harm": false,
    "sexual/minors": false,
    "hate/threatening": false,
    "violence/graphic": false,
    "self-harm/intent": false,
    "self-harm/instructions": false,
    "harassment/threatening": false,
    "violence": false
}
```

We can also view the `category_scores` for each category:

```
response["results"][0]["category_scores"]
```

The value of a score is between 0 and 1. When the value is close to 0, the confidence of the prediction is low. When the value approaches 1, the value is high. The values are not probabilities but only prediction confidence levels:

```
<OpenAIObject at 0x7f05c1a0b970> JSON: {  
    "sexual": 8.125367e-05,  
    "hate": 3.739716e-07,  
    "harassment": 0.0003951491,  
    "self-harm": 2.1336021e-07,  
    "sexual/minors": 1.2213799e-06,  
    "hate/threatening": 2.1999061e-10,  
    "violence/graphic": 2.1322798e-07,  
    "self-harm/intent": 7.287807e-07,  
    "self-harm/instructions": 1.17796894e-07,  
    "harassment/threatening": 2.7180114e-08,  
    "violence": 5.3341464e-05  
}
```

The same moderation process must be applied to the output of the model used for a task:

- The output of the model must go through the rule base.
- Then, if the content is safe according to the rule base, it is submitted to the moderation model.

The input of the ChatGPT GPT-4 model is deemed safe enough to be processed. However, a knowledge base can be an option if we have chosen a closed ecosystem.

## 2. Building a knowledge base for ChatGPT and GPT-4

The goal of the **Knowledge Base (KB)** is for an organization to control the flow of information sent to a model to steer it in the right direction.

You can build the KB in any format you want: SQL Server, other databases, JSON files, CSV, etc.

In this notebook, I'll use `Snap-ML Consulting` as an example for a sample KB. You can use the information you gather for your project and create a corporate or personal KB.

This step is a classical method of creating data. When we build websites, for example, we enter metadata and keywords to increase our visibility and help search engines. The same goes for our transformer conversational models.



Note: You can create as many assertions as you wish in your KB. The only cost will be the space occupied by your data and the maintenance of your dataset.

The following KB is activated with assertions to explain the role of the assistant:

```
assert1={'role': 'assistant', 'content': 'Opening hours of Sna
assert2={'role': 'assistant', 'content': 'Services :expert sys
assert3={'role': 'assistant', 'content': 'Services :Fine-tunin
assertn={'role': 'assistant', 'content': 'Services:advanced pr
#Using the knowledge base as a dataset:
kbt = []
kbt.append(assert1)
kbt.append(assert2)
```

```
kbt.append(assert3)
kbt.append(assertn)
```

We can display the assertions in a pandas DataFrame:

```
#displaying the KB as a DataFrame(DF) Clic on the magic Google
import pandas as pd
df=pd.DataFrame(kbt)
df
```

The output displays the KB for quality control:

| index | role      | content   |
|-------|-----------|---|
| 0     | assistant | Opening hours of Snap-LM Consulting :Monday through Friday 9am to 5pm. Services :expert systems, rule-based systems, machine learning, deep learning, transformer models. |
| 1     | assistant | Services :expert systems, rule-based systems, machine learning, deep learning, transformer models.  |
| 2     | assistant | Services :Fine-tuning OpenAI GPT-3 models, designing datasets, designing knowledge bases.   |
| 3     | assistant | Services:advanced prompt engineering using a knowledge base and SEO keyword methods.  |

Figure 15.4: The KB for quality control

We will now add keywords to parse the user requests.

## Adding keywords

There is much work to do to create an efficient knowledge base, metadata, and a solid parsing function. The parsing function can be classical, as shown in this section.

We can also use advanced search with embeddings as we implemented in *Chapter 11, Leveraging LLM Embeddings as an Alternative to Fine-Tuning*.

Add keywords to the assertions:

```
assertkw1="open"
assertkw2="expert"
assertkw3="services"
assertkwn="prompt"
```

Then, display the keywords in a pandas DataFrame:

```
#create a kb keywords as list
kbkw=[assertkw1,assertkw2,assertkw3,assertkwn]
#displaying the KB as a DataFrame(DF) Clic on the magic Google
dfk=pd.DataFrame(kbkw)
dfk
```

The output is ready for quality control:

| index |          |
|-------|----------|
| 0     | open     |
| 1     | expert   |
| 2     | services |
| 3     | prompt   |

Figure 15.5: The KB index

We will now use our KB to parse the user's prompts.

### 3. Parsing the user requests and accessing the KB

You have entered information from your knowledge base containing keywords linked to the data.

The example below parses the user's prompt on the dialog page of an application:

1. Go through the list of keywords. Each keyword is the label of a record in the KB. You can extend the metadata (keywords) to as many keywords as necessary. You can also add other metadata as for web pages and parse them as well using search engine techniques.
2. When a keyword, the label of a KB record, matches the user's request, the KB record will be retrieved and returned to be part of the information sent to ChatGPT.



You can improve this function to customize your application. A chatbot application can include classical search functions.

Also, you can explore the new Bing conversational interface. Note that it extracts keywords as in this notebook and then searches for links in its knowledge base. Finally, it generates AI using the KB record(s).

You are ready to build a parser to scan a user's request or design any other function to search for request-knowledge base similarity. The function in the notebook shows one way, among others, to parse user requests.

In this example, we'll run a batch of user requests:

```
user_requests=[]
user_requests.append({'role': 'user', 'content': 'At what time
user_requests.append({'role': 'user', 'content': 'At what time
user_requests.append({'role': 'user', 'content': 'Can you crea
user_requests.append({'role': 'user', 'content': 'What service
```

We also determine the number of requests for the batch we will process:

```
n=len(user_requests)
```

We create the parsing function:

```
# This is an example. You can customize this as you wish for your application
def parse_user(uprompt, kbkw, kbt):
    i=0
    j=0
    for kw in kbkw:
        #print(i, kw)
        rq=str(uprompt)
        k=str(kw)
        fi=rq.find(k)
        if fi>-1:
            print(kw, rq, kbt[i])
            j=i
            i+=1
    return kbt[j]
```

You have parsed the user request to find keywords that fit the knowledge base assertions you made.

## 4. Generating ChatGPT content with a dialog function

We will implement a search engine-like prompt for GPT-3.5-turbo or GPT-4. First, we create a function to send a request to GPT-4, for example:

```
#convmodel="gpt-3.5-turbo"
convmodel="gpt-4"
def dialog(iprompt):
    client = OpenAI()
    response = client.chat.completions.create(
        model=convmodel,
        messages=iprompt
    )
    return response
```

In this notebook, the user requests were loaded in a list to make requests in a batch that will call the `dialog` function.

The following cell contains six steps:

- Step 1: Iterate through the user's requests.
- Step 2: The application goes through the user's request and searches for keywords with a search-engine-like technique.
- Step 3: The application creates a prompt with a system message, the knowledge base record found, and the initial user's request.
- Step 4: The prompt is sent to the ChatGPT dialog function.
- Step 5: Store the response in a list.
- Step 6: Display the KB as a DataFrame; click on the magic Google Colaboratory wand to obtain a cool display:

```
responses=[]      #creating a list to store the dialog
#going through the user's requests in a batch for the ChatGPT .
for i in range(n):
    # Step 1: iterating through the user's requests
    user_request_num=i
    #Step 2: the application goes through a the user's request and
    #          to find a record in the knowledge base
    kb_record=parse_user(user_requests[user_request_num],kbkw,kb
    #Step 3: the application creates a prompt, with a system mes.
```

```

iprompt = []
iprompt.append({"role": "system", "content": "You are an ass."})
iprompt.append(kb_record)
iprompt.append(user_requests[user_request_num])
#print(iprompt)
#Step 4: The prompt is sent to the ChatGPT dialog function
response = dialog(iprompt)
#Step 5: storing the response in a list
ex=response.choices[0].message.content
rt = "Total Tokens:" + str(response.usage.total_tokens)
responses.append([user_requests[user_request_num],ex,rt])
#Step 6; displaying the KB as a DataFrame(DF);click on the mag.
pd.DataFrame(responses, columns=['request', 'response', 'token'])

```

The outputs of the batched requests are loaded in a pandas DataFrame:

|   |  |   |                  |
|---|--|---|------------------|
| 0 | {"role": "user", "content": "At what time does Rothman Consulting open on Monday at 9am."} | Rothman Consulting opens on Monday at 9am.        | Total Tokens:83  |
| 1 | {"role": "user", "content": "At what time does Rothman Consulting open on Saturday?"}      | Rothman Consulting is not open on Saturdays. O... | Total Tokens:97  |
| 2 | {"role": "user", "content": "Can you create an AI language model?"}                        | As an AI language model, I'm unable to create ... | Total Tokens:457 |
| 3 | {"role": "user", "content": "What services does Rothman Consulting offer?"}                | Rothman Consulting offers a wide range of serv... | Total Tokens:347 |

Figure 15.6: A list of user requests

## Token control

The number of total tokens is also displayed in the pandas DataFrame for budget monitoring. OpenAI will charge us for batches of 1K tokens. A token is about 75% of a word. You can thus build a cost-monitoring function.

Token control can limit the risks of using an LLM:

- Overreliance by users through a token usage limit
- Hallucinations by introducing the `max_tokens` parameter in the dialog function to avoid letting the model stray
- Overall budget monitoring and control

## Moderation

Make sure to implement the moderation control function to monitor and filter the outputs of the model (ChatGPT in this case).

We have reviewed some tools to help mitigate the risks of implementing LLMs. Let's summarize the chapter and move on to our next adventure.

## Summary

Foundation Models offer many opportunities but come with critical risks that must be taken seriously. We saw how some of the best models on the market, such as ChatGPT, GPT-4, and Vertex AI PaLM 2, could stumble occasionally.

Hallucinations can lead to stating that an elephant landed on the moon. Or invent novels that don't exist. Risky emergent behaviors and disinformation can damage the credibility of LLMs and harm others. Influence campaigns can disrupt the classical flow of information.

Before implementing cloud platform LLMs, we need to check the privacy policies and perform cybersecurity checks.

To mitigate the risks, we went through some of the possible tools. We added a rule base to the moderation model. A knowledge base can create a relatively closed ecosystem and limit open uncontrolled dialogs. The system can be steered with informative messages added to the prompt.

Finally, we saw that token management is an excellent way to control user usage and the overall cost of LLM systems.

We've taken the time to go through the risks of transformers. We can now move at full speed into the world of transformers for computer vision in

## Questions

1. It's impossible to force ChatGPT to harass somebody. (True/False)
2. Hallucinations are only for humans. (True/False)
3. Privacy is taken seriously on the leading cloud platforms. (True/False)
4. APIs pose no risk. (True/False)
5. Harmful content can be filtered. (True/False)
6. A moderation model is 100% reliable. (True/False)
7. A rule base is useless when using LLMs. (True/False)
8. A knowledge base will make the transformer ecosystem more reliable.  
(True/False)
9. We cannot add information to a prompt. (True/False)
10. Prompt engineering requires more effort than prompt design.  
(True/False)

## References

- *GPT-4 Technical Report*, OpenAI, March 27, 2023:  
<https://arxiv.org/pdf/2303.08774.pdf>
- *Bommansani et al. 2023, Ecosystem Graphs: The Social Footprint of Foundation Models*: <https://arxiv.org/abs/2303.15772>
- *OpenAI research*: <https://openai.com/research>
- *Google Generative AI*:  
<https://github.com/GoogleCloudPlatform/generative->

[ai/blob/main/language/prompts/intro\\_prompt\\_design.ipynb](ai/blob/main/language/prompts/intro_prompt_design.ipynb)

- *BIG-bench platform:* [https://github.com/google/BIG-bench/blob/main/bigbench/benchmark\\_tasks/README.md](https://github.com/google/BIG-bench/blob/main/bigbench/benchmark_tasks/README.md)

## Further reading

- *Anderljung et al., 2023, Frontier AI Regulation: Managing Emerging Risks to Public Safety:* <https://arxiv.org/abs/2307.03718>
- GDPR compliance: <https://gdpr.eu/>
- SOC 2 compliance:  
<https://www.imperva.com/learn/data-security/soc-2-compliance/>
- Stanford Ecosystem Graphs, which attempts to have a solid inventory of Foundation Models and resources:  
<https://crfm.stanford.edu/ecosystem-graphs/index.html?mode=home>
- Stanford University's Center for Research on Foundation Models:  
<https://crfm.stanford.edu/>

## Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://www.packt.link/Transformers>



# 16

## Beyond Text: Vision Transformers in the Dawn of Revolutionary AI

Up to now, we have examined variations of the Original Transformer model with encoder and decoder layers. We have also explored other models with encoder-only or decoder-only stacks of layers. Also, the size of the layers and parameters has increased. However, the fundamental architecture of the Transformer retains its original structure with identical layers and the parallelization of the computing of the attention heads.

In this chapter, we will explore the innovative transformer models that respect the basic structure of the Original Transformer but make some significant changes. Scores of transformer models will appear, like the many possibilities a box of LEGO<sup>©</sup> pieces gives. You can assemble those pieces in hundreds of ways! Transformer model sublayers and layers are LEGO<sup>©</sup> pieces of advanced AI.

We will discover powerful computer vision transformers like ViT, CLIP, DALL-E, and GPT-4V. We can add CLIP, DALL-E, and GPT-4V to OpenAI ChatGPT, GPT-4, Google PaLM, and Google BERT (trained by Google) to the growing family of **Generative AI Foundation Models**.

We will discover the architecture of some of these powerful multimodal Foundation Models that prove transformers are *task-agnostic*. A transformer learns sequences. These sequences include vision, sound, and any type of data represented as a sequence.

Images contain sequences of data-like language. We will run a ViT and CLIP to learn their architecture. We will take vision models to innovative levels. We will see how OpenAI DALL-E 2 and DALL-E 3 have mainstreamed generative art and design.

Finally, we will build a notebook with GPT-4V and expand the text-image interactions of DALL-E 3 to divergent semantic association. We will take OpenAI models into the nascent world of highly divergent semantic association creativity.

By the end of the chapter, you will see that the world of Generative AI transformers has evolved into a universe of imagination and creativity beyond **Large Language Models (LLMs)** and into the world of multimodal computer vision.

This chapter covers the following topics:

- From task-agnostic models to vision models
- ViT, vision transformers
- ViT in code
- Text-image vision transformers with CLIP
- Clip in code
- DALL-E 1, 2, and 3
- GPT-4V, DALL-E 3, and divergent semantic association
- Implementing the GPT-4V API



With all the innovations and library updates in this cutting-edge field, packages and models change regularly. Please go to the GitHub repository for the latest installation and code examples:

<https://github.com/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition/tree/main/Chapter16>.

You can also post a message in our Discord community (<https://www.packt.link/Transformers>) if you have any trouble running the code in this or any chapter.

Our first step will be to see how LLMs evolved into vision transformers.

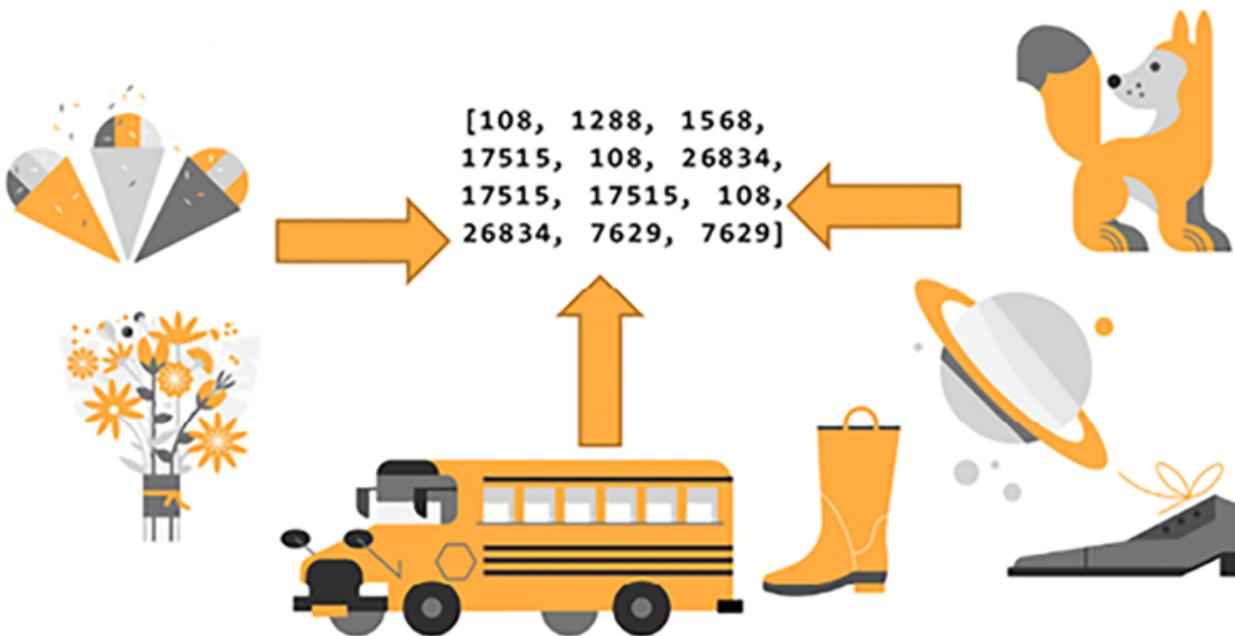
## From task-agnostic models to multimodal vision transformers

Foundation Models, as we saw in *Chapter 1, What Are Transformers?*, have two distinct and unique properties:

- **Emergence:** Transformer models that qualify as Foundation Models can perform tasks they were not trained for. They are large models trained on supercomputers. They are not trained to learn specific tasks like many other models. Foundation Models learn how to understand sequences.
- **Homogenization:** The same model can be used across many domains with the same fundamental architecture. Foundation Models can learn new skills through data faster and better than any other model.

OpenAI ChatGPT models (GPT 3 and GPT 4), Google PaLM, and Google BERT (only the BERT models trained by Google) are task-agnostic Foundation Models. These task-agnostic models lead directly to ViT, CLIP, and DALL-E models.

The level of abstraction of transformer models leads to **multimodal neurons**. Multimodal neurons can process images that can be tokenized as pixels or image patches. Then, they can be processed as *words* in vision transformers. Once an image has been encoded, transformer models see the tokens as any *word* token, as shown in *Figure 16.1*:



*Figure 16.1: Images can be encoded into word-like tokens*

In this section, we will go through:

- ViT: Vision transformers that process images as patches of *words*.
- CLIP: Vision transformers that encode text and images.
- DALL-E: Vision transformers that construct images with text.

Let's begin by exploring ViT, a vision transformer that processes images as patches of *words*.

# ViT - Vision Transformer

*Dosovitskiy et al.* (2021) summed up the essence of the vision transformer architecture they designed in the title of their paper: *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*.

The paper's title sums the process up: an image can be converted into patches of 16x16 words.

Let's first go through a high-level view of the architecture of ViT before looking into the code.

## The basic architecture of ViT

ViT can process an image as patches of *words*. In this section, we will go through the process in three steps:

1. Splitting the image into patches with a feature extractor.
2. Building a vocabulary of image patches with the feature extractor.
3. The patches become the input of the transformer encoder-only model.

The model embeds the input. It will produce an output of raw logits that the pipeline functions will convert into the final probabilities.

The first step is to SPLIT the image into equal-sized patches.

### Step 1: Splitting the image into patches

The image is split into  $n$  patches, as shown in *Figure 16.2*, with a feature extractor. There is no rule saying how many patches there can be as long as

all the patches have the same dimensions, such as, but not only, 16x16:

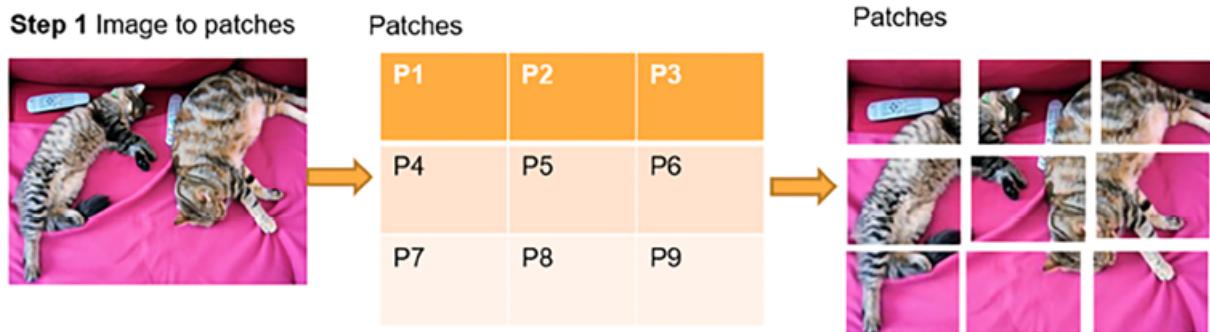


Figure 16.2: Splitting an image into patches

The patches of equal dimensions now represent the *words* of our sequence. The problem of what to do with these patches remains. We will see that each type of vision transformer has its own method.

The preceding image is conceptual. It only displays nine patches. In the following code example, we will explore a size 224x224 pixel image, generating more patches, as we'll see.



Image citation: The image of cats used in this section and the subsequent sections was taken by *DocChewbacca*:  
<https://www.flickr.com/photos/st3f4n/>, in 2006. It is under a Flickr free license,  
<https://creativecommons.org/licenses/by-sa/2.0/>. For more details, see *DocChewbacca*'s image on Flickr:  
<https://www.flickr.com/photos/st3f4n/210383891>.

In this case, for ViT, *Step 2* will be to make a vocabulary of image patches.

## Step 2: Building a vocabulary of image patches

*Step 1* converted an image to equal-sized patches. The motivation of the patches is to avoid processing the image pixel by pixel. However, the problem remains to find a way to process the patches.

The team at Google Research decided to use a vocabulary with the patches obtained by splitting the image to make 16x16 “words,” as shown in *Figure 16.3*:

### Step 2: Building a vocabulary of image patches



*Figure 16.3: Building a vocabulary of image patches*

The output of the extractor is a vocabulary of size 16x16 image (16x16 pixels) patches in this case. This output is a linear 1D representation of the images for the transformer’s input.

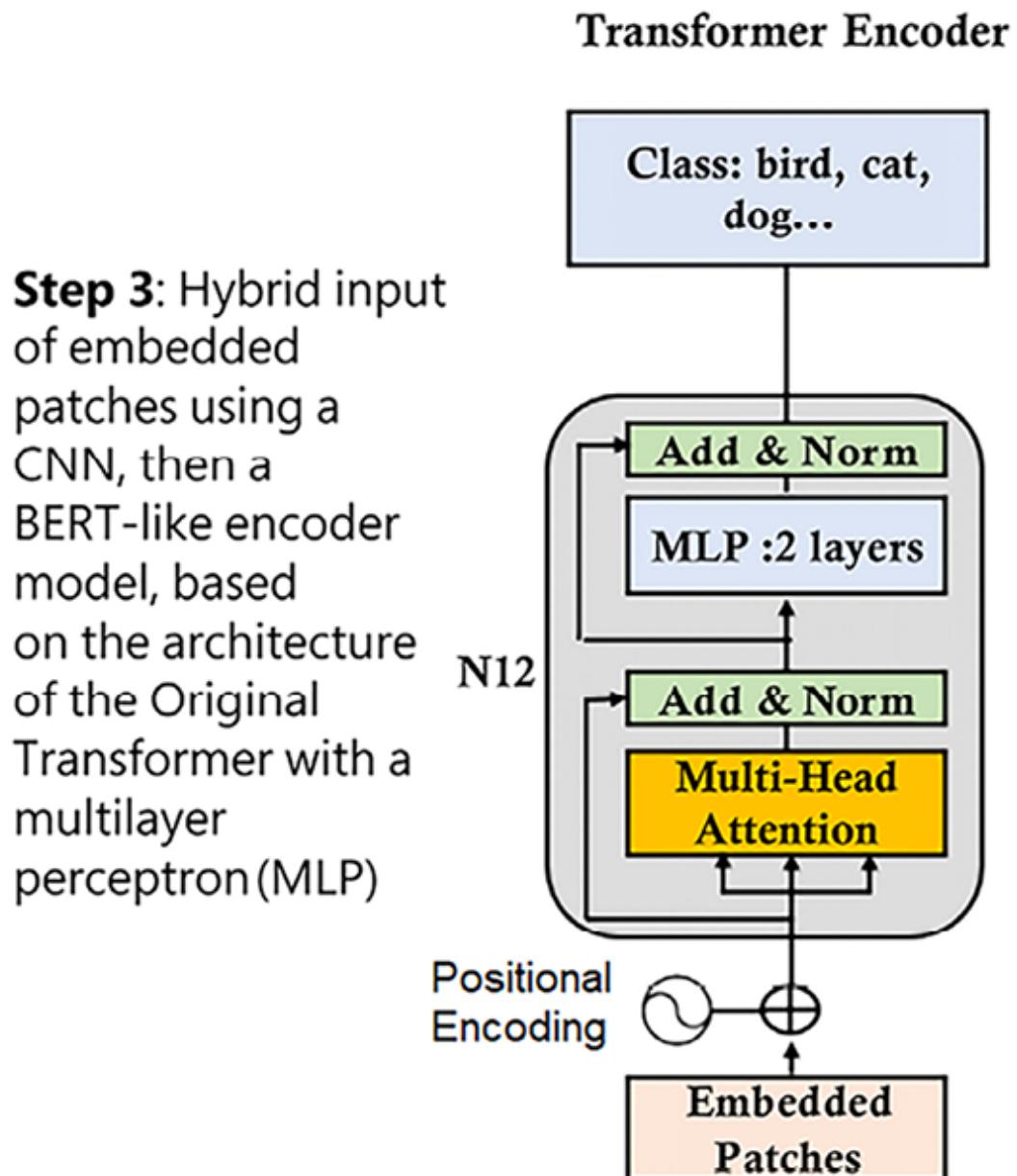
## Step 3: The transformer

Word-like image sequences can fit into a transformer. The problem is that they still are images!

Google Research decided that a hybrid input model would do the job, as shown in *Figure 16.4*:

- Add a convolutional network to embed the patches.

- Add positional encoding to retain the structure of the original image.
- Then, process the embedded input with a standard original BERT-like encoder.
- Finally, the transformer produces an output of raw logits that a sampler will convert into probabilities that fit the label logits. The result will be a label.



*Figure 16.4: A hybrid input sublayer and a standard encoder*

Google Research found a clever way to convert an NLP transformer model into a vision transformer. We can see that the model's architecture follows the Original Transformer (*Vaswani et al.*, 2017) very closely.

The advantages and benefits of the architecture of ViT can be summed up in three points:

- The ViT architecture inherits the scaling capabilities of the Original Transformer model.
- ViT architecture thus can capture long-term dependencies better than convolutional neural network (CNN)-only architectures.
- ViT will learn the relationship between all the patches in its attention layers, providing more accurate predictions.

Now, let's implement a Hugging Face example of a vision transformer in code.

## Vision transformers in code

In this section, we will focus on the main areas of code that relate to the specific architecture of vision transformers. The two main components are:

- The feature extractor
- The transformer

Open `vit_CLIP.ipynb`, which is in the GitHub repository for this chapter.

Google Colab VMs contain many pre-installed packages such as `torch` and `torchvision`.

You can display them by uncommenting the command in the first cell of the notebook by running `!pip list -v`.

We will first download an image that we will use to illustrate ViT:

```
#Development access to delete when going into production
!curl -L https://raw.githubusercontent.com/Denis2054/Transformers/main/images/car_in_space.jpg
```

The notebook first imports an `IPython` module for image rendering and downloads the image:

```
from IPython.display import Image      #This is used for rendering
```

The notebook displays the image:

```
from PIL import Image
# Define the path of your image
image_path = "/content/generate_an_image_of_a_car_in_space.jpg"
# Open the image
image = Image.open(image_path)
```

The image is a car in space created with a Generative AI Stable Diffusion model, as shown in *Figure 16.5*:



*Figure 16.5: Copyright 2023 Denis Rothman: The image was generated by a Stable Diffusion transformer*

Let's build a feature extractor simulator to understand this key component of the pipeline.

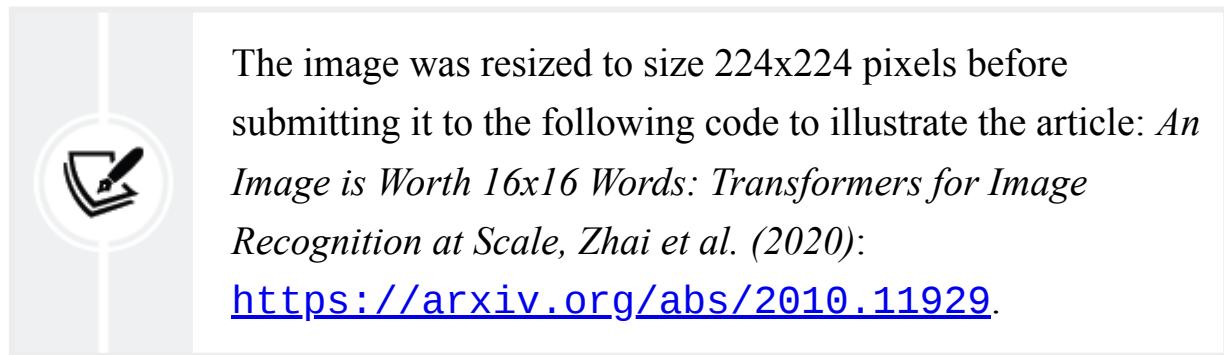
## A feature extractor simulator

In this section, we will build a feature extractor simulator to illustrate how an image is split into patches to become the transformer's input.

The **Vision Transformer (ViT)** model divides the original input image into a grid of small square 16x16 patches. These patches are treated as the equivalent of tokens (like words in a sentence) for a standard transformer model.

In the following code, the patches are created from the input image of size 224x224 using the PyTorch `unfold` function. This function extracts sliding local blocks from an input tensor (in this case, the image). You can think of it as a way of slicing the image into small square patches. The `patch_size` parameter defines the size of these patches.

Each patch is then flattened into a 1D vector, and all these vectors form a 2D input matrix that can be fed into the transformer. Each row of this matrix corresponds to the vector of a single patch. The number of patches is  $224/16$  (width) \*  $224/16$  (height) = 196 patches. This constitutes the vocabulary of the transformer model just as a dictionary of text tokens.



In the original ViT paper, these patches are directly fed into the transformer model after being linearly embedded into a suitable dimension for the transformer. In other words, these patches are the “words” that the transformer model reads.

So, in the context of ViT, “processed patches” refers to patches created from the original image and then reshaped into a 2D input matrix for the transformer. This transformation process is a key part of the data preprocessing in ViT.

The following code illustrates the transformation of an image into 16x16 pixel words.

We first load the image:

```
from IPython.display import Image      #This is used for rendering
from PIL import Image
# Define the path of your image
image_path = "/content/generate_an_image_of_a_car_in_space.jpg"
# Open the image
image = Image.open(image_path)
```

Then, we simulate the conversion into 16x16 pixels using the PyTorch `unfold` function that extracts sliding blocks from the input tensor:

```
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
# Define the transformation
transform = transforms.Compose([
    transforms.Resize((224, 224)), # Resize the image to 224x224
    transforms.ToTensor() # Convert the image to a PyTorch tensor
])
# Apply the transformation
pixel_values = transform(image).unsqueeze(0) # Add an extra dimension
patch_size = 16 # change to match the patch size used in your model
# Assuming pixel_values is your input image tensor of shape (batch_size, num_channels, height, width)
patches = pixel_values.unfold(2, patch_size, patch_size).unfold(3, patch_size, patch_size)
# patches will now have shape (batch_size, num_channels, num_patches, patch_size, patch_size)
```

Finally, we display each patch for educational purposes to illustrate the 16x16 pixel words obtained with three channels (R, G, B):

```
# Reshaping the patches tensor for easy viewing
patches_reshaped = patches.permute(0, 2, 3, 1, 4, 5).contiguous()
# create a transform to convert tensor to PIL Image
to_pil = transforms.ToPILImage()
for i in range(patches_reshaped.size(0)):
```

```
print(f"Displaying patch {i+1}/{patches_reshaped.size(0)}")
patch_size = patches_reshaped[i].shape
plt.title(f"Patch {i+1}, size: {patch_size}")
plt.imshow(to_pil(patches_reshaped[i]))
plt.axis("off") # to hide axes
plt.show()
userInput = input("Press Enter to display the next patch (q to quit):")
if userInput.lower() == 'q':
    break
```

The output displays the 196 16x16 “word” patches one by one until *q* is pressed:

```
Displaying patch 1/196
Patch 1, size: torch.Size([3, 16, 16])
```



Figure 16.6: Display of image patches

You can go through the patches by pressing *Enter* and press *q* to quit.

The feature extractor is an essential component of the model pipeline that performs the necessary preprocessing of the raw input data (in this case, an image) to convert it into a suitable format that can be fed into the model.

Before we continue, let's sum up what the ViT feature extractor does:

1. **Resizing:** It resizes the input images to the expected size of the model.  
In our case, it's already a 224x224 size image.
2. **Normalization:** The feature extractor normalizes the pixel values of the image from [0,255] to a smaller range, such as [0,1] or [-1,1], for each color channel.
3. **Conversion to PyTorch tensors:** The model requires tensors to process data.
4. **Patching:** As we visualized previously, the image is divided into square patches. This is the key part of the innovation. In this case, the feature extractor produces 16x16 image “words” patches for its vocabulary.  
Each patch becomes a token in the vocabulary of the system.
5. **Flattening and embedding:** The patches cannot be sent directly to the transformer. Each patch is flattened into a 1D array and converted into token-word-images embedding vectors.

In summary, the feature extractor takes care of all the preprocessing required to transform the raw image into a format that can be understood by the model, as shown in the following code, which shows the actual feature extractor in action:

```
from transformers import ViTFeatureExtractor, ViTForImageClassification
from PIL import Image
import requests
feature_extractor = ViTFeatureExtractor.from_pretrained('google/vit-base-patch16-224')
```

We will now enter the transformer component of the ViT architecture.

## The transformer

The preprocessed image is then fed into the ViT model (`ViTForImageClassification`), which outputs the logits. The logits are the raw, unnormalized scores output by the model's final layer for each class in the classification task. These logits can then be converted into probabilities using the softmax function, and the class with the highest probability is usually taken as the model's prediction.

The feature extractor's preprocessed image becomes the input of ViT:

```
model = ViTForImageClassification.from_pretrained('google/vit-  
inputs = feature_extractor(images=image, return_tensors="pt")
```

The transformer then processes the feature extractor's output in its encoder-only stack. If necessary, take the time to review the full architecture of the Original Transformer in *Chapter 2, Getting Started with the Architecture of the Transformer Model*, which ViT follows very closely.

The transformer processes the embeddings provided by the feature extractor, goes through the attention layer and feedforward network layer, and produces an output of *raw logits*:

```
inputs = feature_extractor(images=image, return_tensors="pt")  
outputs = model(**inputs)  
logits = outputs.logits
```

The raw logits of the output will be processed by the pipeline that controls the flow of this image classification process. A softmax function can be applied or first the temperature, then the softmax followed by `top_k` and `top_p` as we saw, for example, in *Chapter 14, Exploring Cutting-Edge LLMs with Vertex AI and PaLM 2*, in the *Vertex AI PaLM 2 interface* section.

The pipeline now converts the raw outputs into a prediction and produces the label of the probability selected:

```
# model predicts one of the 1000 ImageNet classes
predicted_class_idx = logits.argmax(-1).item()
print("Predicted class:", predicted_class_idx, ": ", model.config.name)
```

The output shows the result of the image classification task:

```
Predicted class: 627: limousine, limo
```

The prediction is acceptable, considering we submitted what looks like a car or a limousine in outer space.

Now that we have gone through the main components of ViT, we can go further and examine the shapes and contents of the tensors we processed.

## Configuration and shapes

In this section, we will display the content of the main components and tensors we can access. We will go through them in their order of appearance in the code.

The following code shows the configuration of the feature extractor (`Feature_extractor`). The output shows the general configuration of the

feature extractor:

```
ViTFeatureExtractor {
    "do_normalize": true,
    "do_rescale": true,
    "do_resize": true,
    "image_mean": [
        0.5,
        0.5,
        0.5
    ],
    "image_processor_type": "ViTFeatureExtractor",
    "image_std": [
        0.5,
        0.5,
        0.5
    ],
    "resample": 2,
    "rescale_factor": 0.00392156862745098,
    "size": {
        "height": 224,
        "width": 224
    }
}
```

You can see how the feature extractor takes a 224x224 image, normalizes it, rescales it, resizes it, and computes a mean, and more functions, to produce the output.

The output of the feature extractor contains the inputs for the transformer, as we saw previously:

```
inputs = feature_extractor(images=image, return_tensors="pt")
```

We can display the inputs:

```
print(inputs['pixel_values'].shape)
```

The output is quite surprising:

```
torch.Size([1, 3, 224, 224])
```

We see 1 image and 3 channels, and we still have the size of our original 224x224 image and not the 16x16 size patches flattened into 1D as input tokens!

*Dosovitskiy et al.* (2021), on page 3, *Figure 1*, state that they split an image into fixed-size patches, linearly embed them, and send the sequence of embedded vectors to a *standard* Transformer encoder. The Original Transformer detailed in *Vaswani et al.* (2017) is not a CNN. So, something is going on, and we must find out!

What happened? The only way to investigate this is to look into the configuration of the model:

```
model
```

The output from the beginning of the configuration up to the first layer shows the configuration of the model that is different from the original ViT model:

```
ViTForImageClassification(  
    (vit): ViTModel(  
        (embeddings): ViTEmbeddings(  
            (patch_embeddings): ViTPatchEmbeddings(  
                (projection): Conv2d(3, 768, kernel_size=(16, 16), str.
```

```

        )
        (dropout): Dropout(p=0.0, inplace=False)
    )
    (encoder): ViTEncoder(
        (layer): ModuleList(
            (0-11): 12 x ViTLayer(
                (attention): ViTAttention(
                    (attention): ViTSelfAttention(
                        (query): Linear(in_features=768, out_features=768)
                        (key): Linear(in_features=768, out_features=768,
                        (value): Linear(in_features=768, out_features=768)
                        (dropout): Dropout(p=0.0, inplace=False)
                    )
                    (output): ViTSelfOutput(
                        (dense): Linear(in_features=768, out_features=768)
                        (dropout): Dropout(p=0.0, inplace=False)
                    )
                )
            )
            (intermediate): ViTIntermediate(
                (dense): Linear(in_features=768, out_features=3072)
                (intermediate_act_fn): GELUActivation()
            )
            (output): ViTOuput(
                (dense): Linear(in_features=3072, out_features=768)
                (dropout): Dropout(p=0.0, inplace=False)
            )
            (layernorm_before): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (layernorm_after): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        )
    )
    (layernorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
    (classifier): Linear(in_features=768, out_features=1000, bias=True)
)

```

When we observe the code, we notice that the configuration does not strictly follow *Dosovitskiy et al.*'s (2021) original ViT model. This is not a critical issue in a fast-moving market. There is not only one way to build efficient

models. The main difference is that *Dosovitskiy et al.*'s (2021) original ViT model does not contain a **Convolutional Neural Network (CNN)**. This does not diminish the efficiency of this variation of the Hugging Face ViT model, [google/vit-base-patch16-224](#).

Remember that trade-offs between the original model architectures and changes made to them are common in machine learning (ML) projects. You will find many hybrid implementations on your ML journey!

Now, let's look at the raw logits of the transformer's output. First, let's display the size:

```
print(outputs.logits.shape)
```

The output is:

```
torch.Size([1, 1000])
```

The output makes sense. The 999 logits (tensors start at 0) fit the number of labels we can display with:

```
model.config.id2label
```

An excerpt of the output shows that we indeed have 999 labels:

```
...986: "yellow lady's slipper, yellow lady-slipper, Cypripedium
```

We can now look at the raw logits themselves with `outputs.logits`, shown in this excerpt:

```
tensor([[ 6.1340e-01, -9.8148e-01, -5.1315e-01, -3.4200e-03, -5
```

The outputs of any transformer model, including this one, are stochastic, so outputs may differ from one run to another.

As explained in *Chapter 14, Exploring Cutting-Edge NLP with Google Vertex AI and PaLM 2* in the *Vertex AI PaLM 2 Interface* section, we have several options:

- Applying a softmax function and then a top  $k$  to find the top probabilities.
- First, apply the temperature, then the softmax function, then `top-k`, and finally `top_p`.

In this model, we will apply a softmax:

```
import torch
import pandas as pd
# Apply softmax to convert logits to probabilities
probs = torch.nn.functional.softmax(logits, dim=-1)
```

Now, we can apply `top-k` to find the top five probabilities and their corresponding labels and display the result in a pandas DataFrame:

```
# Get the top 5 predicted class probabilities and their corres,
top_5_probs, top_5_labels = torch.topk(probs, 5)
top_5_probs = top_5_probs.detach().cpu().numpy()
top_5_labels = top_5_labels.detach().cpu().numpy()
```

```

# Get the mapping from IDs to labels
id2label = model.config.id2label

# Create a dictionary that maps the top 5 predictions to their
pred_dict = {"Index": [], "Probability": [], "Label": []}

for i in range(5):
    pred_dict["Index"].append(top_5_labels[0][i])
    pred_dict["Probability"].append(top_5_probs[0][i])
    pred_dict["Label"].append(id2label[top_5_labels[0][i]])

# Convert the dictionary to a DataFrame
pred_df = pd.DataFrame(pred_dict)

# Display the DataFrame
pred_df

```

The output displays the top five probabilities, their respective scores, and corresponding labels:

| index | Index | Probability          | Label   |
|-------|-------|----------------------|---|
| 0     | 627   | 0.8503304719924927   | limousine, limo   |
| 1     | 436   | 0.04098338633775711  | beach wagon, station wagon, wagon<br>waggon, station waggon, waggon |
| 2     | 817   | 0.03390469029545784  | sports car, sport car   |
| 3     | 479   | 0.028076766058802605 | car wheel   |
| 4     | 751   | 0.015958618372678757 | racer, race car, racing car   |

Figure 16.7: Image descriptions and their probabilities

We have reconstructed the output label that shows that “limousine, limo” is the best choice in this context. The “car” in the space image provided was created by a Stable Diffusion model and can be interpreted as a “limo,” which isn’t bad considering the creativity of the image. We will explore Stable Diffusion in more depth in *Chapter 17, Transcending the Image-Text Boundary with Stable Diffusion*.

You have learned that there is more than one way to implement models in real-life ML projects! Study each variation to create your ML toolbox to be ready to adapt a model to the constraints of a project.

With the knowledge of this section in mind, we can explore CLIP.

## CLIP

**Contrastive Language-Image Pre-Training (CLIP)** is a multimodal transformer that can be used for image classification. CLIP's process can be summed up as follows:

- A feature extractor, like ViT, produces image tokens.
- Text also is an input in tokens, as in ViT
- The attention layer learns the relationships between the image tokens and the text tokens with some form of “cross-attention.”
- The output is also raw logits, as in ViT.

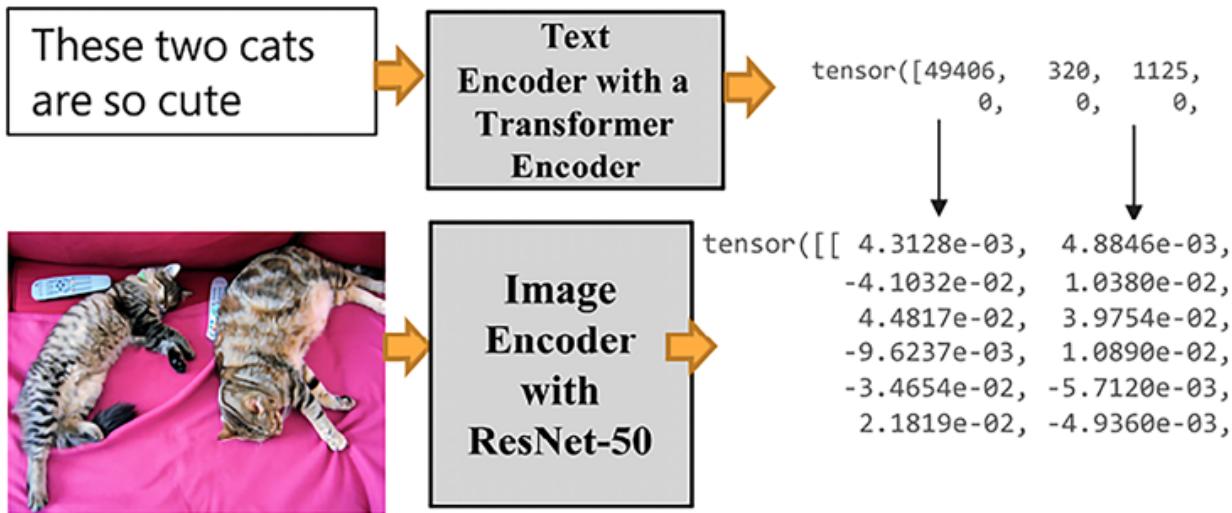
We will first look into the basic architecture of CLIP before running CLIP in code.

## The basic architecture of CLIP

The model is contrastive: the images are trained to learn how they fit together through their differences and similarities. The image and captions find their way toward each other through (joint text, image) pretraining. After pretraining, CLIP learns new tasks.

CLIP is transferable because it can learn new visual concepts, like GPT models, such as action recognition in video sequences. The captions lead to endless applications.

ViT splits images into word-like patches. CLIP jointly trains *text and image* encoders for (caption, image) pairs to maximize cosine similarity, as shown in *Figure 16.8*:



*Figure 16.8: Jointly training text and images*

*Figure 16.8* shows how the transformer will run a standard transformer encoder for the text input. It will run a ResNet 50-layer CNN for the images in a transformer structure. ResNet 50 was modified to run an average pooling layer in an attention pooling mechanism with a multi-head QKV attention head.

Let's see how CLIP learns text-image sequences to make predictions.

## CLIP in code

Open `ViT_CLIP.ipynb`, which is in the repository for this chapter on GitHub. Then, go to the `CLIP` cell of the notebook.

The program begins by installing PyTorch and CLIP:

```
!pip install ftfy regex tqdm
!pip install git+https://github.com/openai/CLIP.git
```

The program also imports the modules and CIFAR-100 to access the images:

```
import os
import clip
import torch
from torchvision.datasets import CIFAR100
```

There are 10,000 images available with an index between 0 and 9,999. The next step is to select an image we want to run a prediction on:

Select an image index between 0 and 9999

**index:** 15

---

*Figure 16.9: Selecting an image index*

The program then loads the model on the available device (GPU or CPU):

```
# Load the model
device = "cuda" if torch.cuda.is_available() else "cpu"
model, preprocess = clip.load('ViT-B/32', device)
```

The images are downloaded:

```
# Download the dataset
cifar100 = CIFAR100(root=os.path.expanduser("~/cache")), download=True
```

The input is prepared:

```
# Prepare the inputs
image, class_id = cifar100[index]
image_input = preprocess(image).unsqueeze(0).to(device)
text_inputs = torch.cat([clip.tokenize(f"a photo of a {c}") for c in class_ids])
```

Let's visualize the input we selected before running the prediction:

```
import matplotlib.pyplot as plt
from torchvision import transforms
plt.imshow(image)
```

The output shows that `index 15` is a lion:

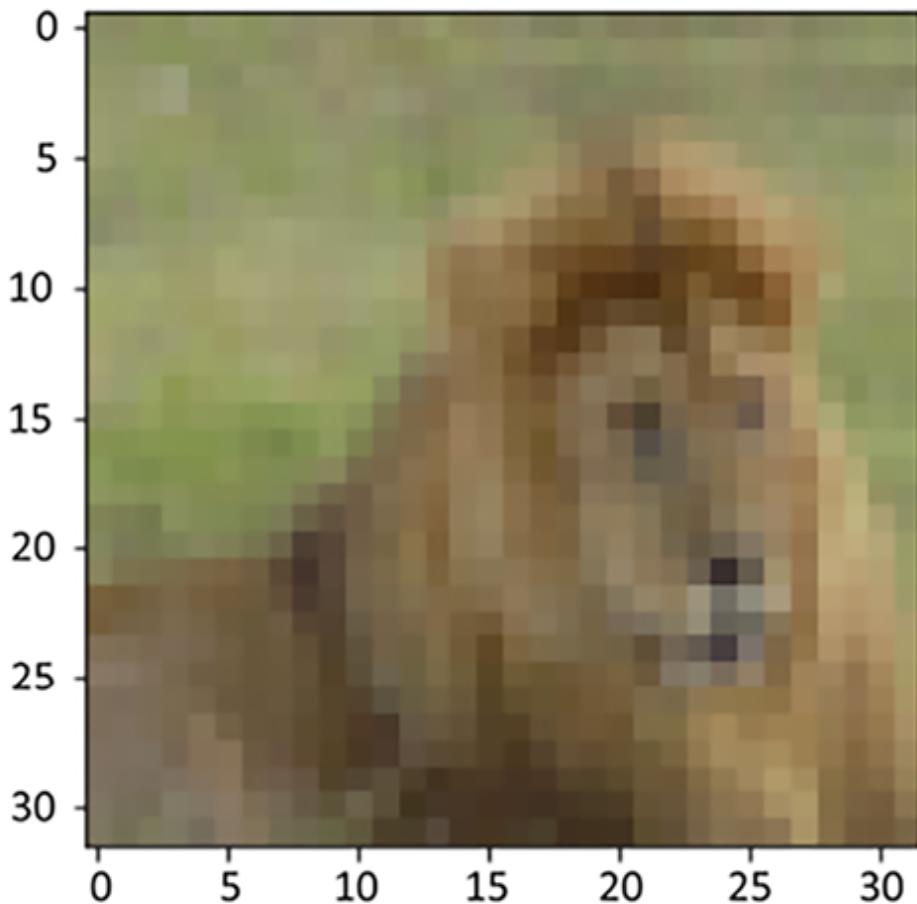


Figure 16.10: Image of index 15



The images in this section are from *Learning Multiple Layers of Features from Tiny Images*, Alex Krizhevsky, 2009: <https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf>. They are part of the CIFAR-10 and CIFAR-100 datasets ([toronto.edu](http://www.cs.toronto.edu)): <https://www.cs.toronto.edu/~kriz/cifar.html>.

We know this is a lion because we are humans. A transformer initially designed for NLP has to learn what an image is. We will now see how well it

can recognize images.

The program shows that it is running a joint transformer model by separating the image input from the text input when calculating the features:

```
# Calculate features
with torch.no_grad():
    image_features = model.encode_image(image_input)
    text_features = model.encode_text(text_inputs)
```

Now, CLIP makes a prediction and displays the top five predictions:

```
# Pick the top 5 most similar labels for the image
image_features /= image_features.norm(dim=-1, keepdim=True)
text_features /= text_features.norm(dim=-1, keepdim=True)
similarity = (100.0 * image_features @ text_features.T).softmax(dim=-1)
values, indices = similarity[0].topk(5)
# Print the result
print("\nTop predictions:\n")
for value, index in zip(values, indices):
    print(f"{cifar100.classes[index]}: {100 * value.item():.2f}%")
```

You can modify `topk(5)` if you want to obtain more or fewer predictions.

The top five predictions are displayed:

```
Top predictions:
    lion: 96.34%
    tiger: 1.04%
    camel: 0.28%
    lawn_mower: 0.26%
    leopard: 0.26%
```

CLIP found the lion, which shows the flexibility of transformer architectures.

Vision models, as LLM models, are stochastic so the output may vary from one run to another.

The next cell displays the classes:

```
cifar100.classes
```

You can go through the classes to see that with only one label per class, which is restrictive, CLIP did a good job:

```
[..., 'kangaroo', 'keyboard', 'lamp', 'lawn_mower', 'leopard', 'lion'  
'lizard', ...]
```

The notebook contains several other cells describing the architecture and configuration of CLIP that you can explore.

The `model` cell, is particularly interesting because you can see the visual encoder that begins with a convolutional embedding like for the ViT model and then continues as a “standard” size-768 transformer with multi-head attention:

```
CLIP(  
    (visual): VisionTransformer(  
        (conv1): Conv2d(3, 768, kernel_size=(32, 32), stride=(32, 3  
        (ln_pre): LayerNorm((768,), eps=1e-05, elementwise_affine=True  
        (transformer): Transformer(  
            (resblocks): Sequential(  
                (0): ResidualAttentionBlock(  
                    (attn): MultiheadAttention(
```

```
(out_proj): NonDynamicallyQuantizableLinear(in_feat  
)  
(ln_1): LayerNorm((768,), eps=1e-05, elementwise_affi  
(mlp): Sequential(  
    (c_fc): Linear(in_features=768, out_features=3072,  
    (gelu): QuickGELU()  
    (c_proj): Linear(in_features=3072, out_features=768  
)  
(ln_2): LayerNorm((768,), eps=1e-05, elementwise_affi  
)
```

Another interesting aspect of the `model` cell is looking into the size-512 text encoder that runs jointly with the image encoder:

```
(transformer): Transformer(  
    (resblocks): Sequential(  
        (0): ResidualAttentionBlock(  
            (attn): MultiheadAttention(  
                (out_proj): NonDynamicallyQuantizableLinear(in_featur  
)  
            (ln_1): LayerNorm((512,), eps=1e-05, elementwise_affine  
            (mlp): Sequential(  
                (c_fc): Linear(in_features=512, out_features=2048, bi  
                (gelu): QuickGELU()  
                (c_proj): Linear(in_features=2048, out_features=512,  
)  
            (ln_2): LayerNorm((512,), eps=1e-05, elementwise_affine  
    )
```

Go through the cells that describe the architecture, configuration, and parameters to see how CLIP represents data.

We showed that task-agnostic transformer models process image-text pairs as text-text pairs. We could apply task-agnostic models to music-text, sound-text, music-images, or any data pairs.

We will now explore DALL-E, another task-agnostic transformer model that can process images and text.

## DALL-E 2 and DALL-E 3

DALL-E, as with CLIP, is a multimodal model. CLIP processes text-image pairs. DALL-E processes the text and image tokens differently. DALL-E 1 has an input of a single stream of text and image of 1,280 tokens. 256 tokens are for the text, and 1,024 tokens are used for the image.

DALL-E was named after *Salvador Dali* and Pixar's WALL-E. The usage of DALL-E is to enter a text prompt and produce an image. However, DALL-E must first learn how to generate images with text.

This transformer generates images from text descriptions using a dataset of text-image pairs.

We will go through the basic architecture of DALL-E to see how the model works.

## The basic architecture of DALL-E

Unlike CLIP, DALL-E concatenates up to 256 BPE-encoded text tokens with  $32 \times 32 = 1,024$  image tokens, as shown in *Figure 16.11*:

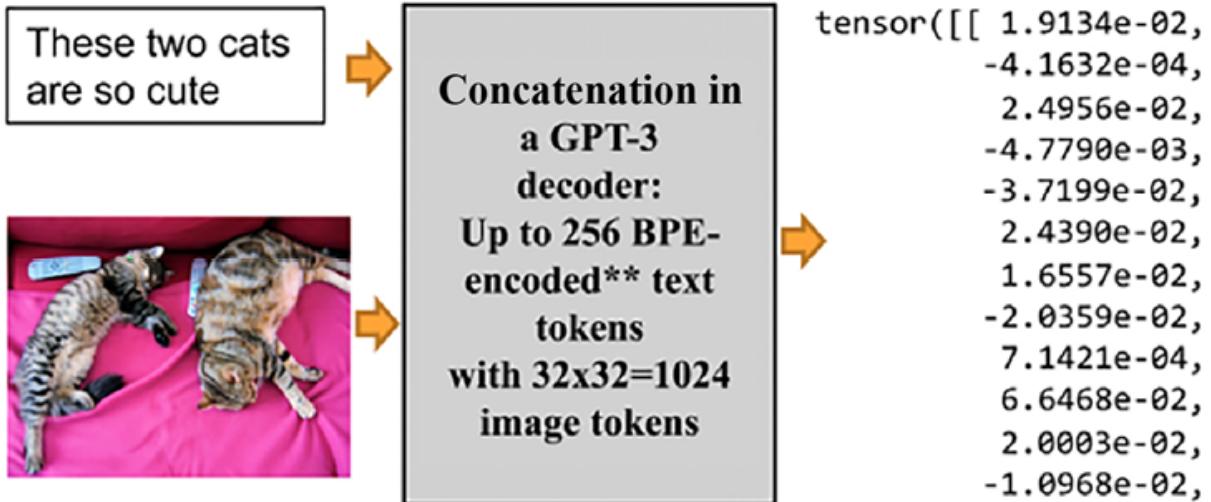


Figure 16.11: DALL-E concatenates text and image input

Figure 16.11 shows that, this time, our cat image is concatenated with the input text.

OpenAI has not released a public version of the source code of DALL-E. Any source code that OpenAI has not released must be viewed with skepticism. The description in this section reflects the concepts of the approach, but in January 2024, the model's architecture has probably evolved.

We can rely on an official paper to understand the general architecture of the model: *Ramesh et al. (2021), Zero-Shot Text-to-Image Generation*:

<https://arxiv.org/abs/2102.12092>.

The *Zero-Shot Text-to-Image Generation* paper mentions a two-stage method:

- **Stage 1:** The use of a **Discrete Variational Autoencoder (DVAE)**. The goal of the algorithm is to take an input of  $256 \times 256 = 65536$  values of an RGB image. The output is a much smaller  $32 \times 32 = 1024$  grid of image tokens where each token can have one of 8192 (empirical choice) possible values.

- **Stage 2:** The “concatenation of up to 256 BPE encoded text tokens with the  $32 \times 32 = 1024$  image tokens,” as shown in *Figure 16.11*. The goal of the model is to train the model on a joint distribution of the text and image tokens through the transformer. The attention mechanism of the transformer enables the model to focus on different parts of the image and text input when generating the output. In this case, the transformer is particularly useful for dealing with long image and text data sequences.

For now, let’s get our hands on the DALL-E 2 and DALL-E 3 API to generate images!

## Getting started with the DALL-E 2 and DALL-E 3 API

DALL-E’s text-to-image functionality has dramatically improved in a short time. Transformers are multimodal and can perform a variety of tasks (audio, image, and other signals).

This section implements DALL-E 2 and DALL-E 3. The API allows us to create and generate variations of an image.

Open `Getting_Started_DALL_E_API.ipynb`. You can implement the DALL-E 2 and DALL-E 3 API as you see fit for your project. The program first installs Pillow (a Python imaging library) and the OpenAI API library, which will require your API key.

You can run it cell by cell to grasp the functionality provided or run the whole notebook for one scenario.

This section is divided into two parts:

- Creating a new image
- Creating a variation of an image

Let's first create an image.

## Creating a new image

The first cell sets the goal of the notebook. You can use it for all scenarios.

In this example, we will create a new image of a person talking to a chatbot in a restaurant near the moon and save the image to a file:

```
#prompt  
sequence="Creating an image of a person using a chatbot in a r
```

`sequence` is the text prompt to instruct DALL-E 2 to create an image.

Run the OpenAI install, import, and API key cells in the notebook.

Now, go to the `Generation` cell and run the DALL-E 3 API:

```
from openai import OpenAI  
client = OpenAI()  
response = client.images.generate(  
    model="dall-e-3",  
    prompt=sequence,  
    size="1024x1024",  
    quality="standard",  
    n=1,  
)  
image_url = response.data[0].url
```

The request begins with `client.images.generate`. The prompt is the sequence we defined following the documentation cited in the first cells of

this notebook.

The image is generated, saved, and displayed:

```
# displaying the image
url = image_url
image = Image.open(requests.get(url, stream=True).raw)
image.save("c_image.png", "PNG")
c_image = Image.open(requests.get(url, stream=True).raw)
c_image
```

The output displays the image:



Figure 16.12: Image created by the DALL-E 2 API

Each time you run the API for the same sequence, you might obtain a different version of the same image.

Now, let's build a variation of an image.

## Creating a variation of an image

Let's see how creative DALL-E 2 can be and ask it to create a variation for the image that was created and saved as `c_image.png`. As of January 2024, OpenAI recommends DALL-E 2 to create variations. The model will no doubt evolve, and you with it!

The following is an example of the `variation` cell. We select an image and ask DALL-E to create a variation of it, similar to the original one but more creative:

```
from openai import OpenAI
client = OpenAI()
response = client.images.create_variation(
    model="dall-e-2", # As of January 2024, variations with dall-
    image=open("c_image.png", "rb"),
    n=2,
    size="1024x1024"
)
image_url = response.data[0].url
```

The output can be a nice variation, a slight variation, or no variation. If we obtain no variation, it might be a *regurgitation* phenomenon in which the model *regurgitates* an existing image. OpenAI has worked on this issue and will undoubtedly improve in the future. The variation may also depend on our prompt and the image we ask DALL-E 2 to process. Ultimately, it depends on the advancement of OpenAI in this field, which will certainly progress in the near future.

Let's display the variation with the following code:

```
url = image_url
image = Image.open(requests.get(url, stream=True).raw)
v_image = Image.open(requests.get(url, stream=True).raw)
v_image
```

The output is an interesting variation:



*Figure 16.13: A variation of the previous image*

You now know how to create and generate variations of images with the DALL-E 2 API. You are on the road to becoming an AI image artist!

OpenAI has gone a step further with its mainstream online DALL-E interface.

## From research to mainstream AI with DALL-E

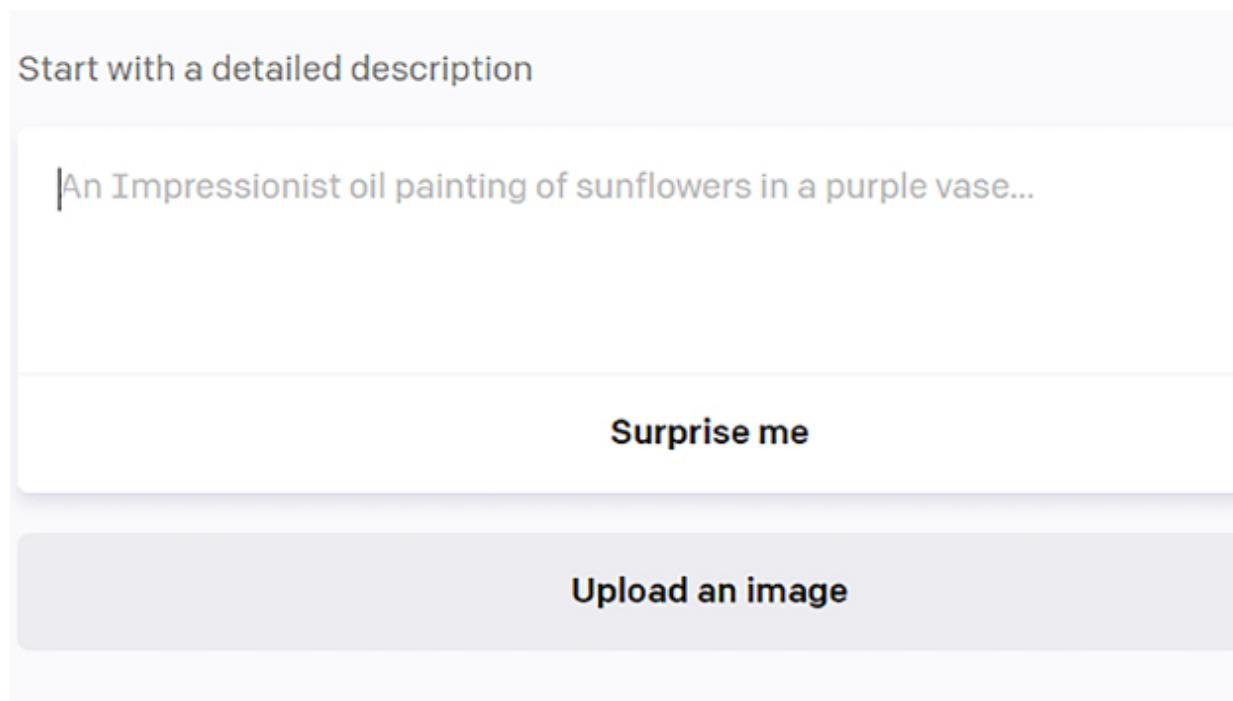
Foundation Models that leverage text and images have gone from research to the mainstream. Mainstream end users can access the most revolutionary computer vision in the history of computer science.

Of course, many computer vision tools were on the market before DALL-E appeared. But those who knew how to use them were often seen as geeks.

Now, with Generative AI applied to imagery, an end user with no knowledge or experience can produce a professional image.

In this previous section, we saw how to use OpenAI's DALL-E API. However, an end user can produce text-to-images without knowing what prompt design or engineering is about.

Let's go to OpenAI's end-user interface: <https://openai.com/dall-e-2>. The interface is intuitive to use:



*Figure 16.14: The DALL-E 2 user interface*

We can create an image, ask DALL-E 2 to surprise us, or upload an image.

We can start with a detailed description, such as:

An alien that looks like a Hollywood actor in 3D in the Mohave

The results are often quite surprising. DALL-E produces several images we can choose from, such as the one in *Figure 16.15*:



*Figure 16.15: DALL-E 2 creates an image of an alien based on a user prompt*

This alien would surely succeed in a biopic of his voyage in Hollywood!

We can also load an image and ask for DALL-E to generate variations. First, we need to upload an image. In this case, we can use one of the images of an alien previously saved in our library on OpenAI:



*Figure 16.16: Another DALL-E 2 image of an alien*

DALL-E produced an alien that is beginning to look like a human:



*Figure 16.17: A more human-like alien*

The results are impressive, and this is only the beginning! The interface will evolve and maybe change completely. As of January 2024, we can now create images directly in the ChatGPT interface, as we will see in the next section.

In the next section, we will take image creation and recognition to another level with divergent semantic association.

## GPT-4V, DALL-E 3, and divergent semantic association

How many humans are creative with a “big C?” How many humans are outperformed by GPT-4 for “little C” everyday life creativity? *Chen et al.* (2023) produced a ground-breaking experiment in their paper. The full title is worth meditating on: *Probing the “Creativity” of Large Language Models: Can Models Produce Divergent Semantic Association?*

Divergent semantic association is the ability of a creative mind to find symbols (words, images, and sounds) that diverge from a standard that defines creativity. Creativity is something new that GPT-4 can now perform.

“Big C” creativity is rare for humans. “Big C” creativity, such as that of Mozart, Einstein, or Picasso, is a one-in-a-million phenomenon.

However, “little C” creativity is relatively common when we adapt to new situations or produce new little artifacts. *Chen et al.* (2023) collected data on 8,572 participants in a divergent semantic association experiment with GPT-4. They found that GPT-4 outperformed 96% of humans in **Divergent Association Tasks (DATs)**!

In this section, we will unleash the full creative potential of GPT-4, GPT-4V, and DALL-E 3 in a DAT experiment.

We will first define divergent semantic association and then let GPT-4 organize the section, explanations, code, and creativity!

Let's begin the experiment by asking GPT-4 with DALL-E 3 to define divergent semantic association.

## Defining divergent semantic association

ChatGPT Plus, <https://openai.com/blog/chatgpt-plus>, with DALL-E with GPT-4 produced the definition of divergent semantic association:

Divergent Semantic Association is a concept often related to cognitive psychology, this ability is contrasted with convergent thinking. This kind of thinking is measured by tasks like the Alternative

Now, we will ask DALL-E with GPT-4 to perform a DAT:

Give me 10 words with divergent semantic associations around the word 'dog'.

The output is highly creative and imaginative:

Sure, here are 10 words with divergent semantic associations related to the word 'dog':  
1. Galaxy - Expanding from the idea of a 'space dog', a fiction  
2. Pyramid - Inspired by ancient Egyptian canines, like the Saluki  
3. Sonnet - Reflecting the use of dogs in poetry, where they often feature prominently

- 4. Glacier - Evoking images of sled dogs, vital for transportation.
- 5. Microchip - Tying to the technology used in modern pet identification.
- 6. Kaleidoscope - Symbolizing the diverse breeds and colors of dogs.
- 7. Labyrinth - Drawing a connection to mythical dogs like Cerberus.
- 8. Quasar - A metaphor for the energetic and dynamic nature of dogs.
- 9. Zenith - Representing the peak of training and discipline in dog sports.
- 10. Alchemy - Symbolizing the transformational bond between humans and their canine companions.

These NLP DAT questions have become relatively easy for the latest versions of DALL-E with GPT-4. Let's take this further with computer vision.

## Creating an image with ChatGPT Plus with DALL-E

We will now go beyond the NLP DAT with a **Computer Vision (CV) DAT**. We will ask ChatGPT with DALL-E to perform a CV DAT function:

Apply Divergent Semantic Association to the following image created by DALL-E.

The response is creative:

Here are the images of a highly unconventional and creative representation of a dog.

DALL-E produced the images in *Figure 16.18* and *Figure 16.19*:



Figure 16.18: CV DAT image of a dog produced by DALL-E



Figure 16.19: A more divergent CV DAT image of a dog produced by DALL-E

DALL-E’s creativity equals, if not exceeds, human “little C” creativity. After all, how many humans, on average, can paint creative images of a dog better than these?

However, the experiment is insufficient. We want to push OpenAI DALL-E to the limits of CV DAT creativity.

Let's submit a new request:

Take this further with more divergence with an image that only

◀ ▶

The response shows the ease with which DALL-E can perform creative tasks:

Here are the images showcasing an extremely abstract and diverg

◀ ▶

The images produced by DALL-E take us to the limit of the CV divergent task requested, as shown in *Figure 16.20* and *Figure 16.21*:



*Figure 16.20: Advanced CV DAT of a dog produced by DALL-E*



*Figure 16.21: Extreme CV DAT of a dog produced by DALL-E*

ChatGPT Plus with DALL-E and GPT-4 confirms the study conducted by *Chen et al. (2023)* for “little C” creativity. Most humans are surpassed.

However, can the GPT-4V AI recognize divergent images? Let’s find out.

## **Implementing the GPT-4V API and experimenting with DAT**

GPT-4 with vision (GPT-4V) can analyze images we submit through the OpenAI API. We have produced some highly divergent NLP semantic associations and CV images. But can GPT-4V recognize them and describe them?

Let’s find out.

Open `GPT-4V.ipynb`. The notebook begins the standard OpenAI installation and API key activation.

We will first ask GPT-4V to analyze a standard non-divergent image.

## Example 1: A standard image and text

We will use the `gpt-4-vision-preview` as recommended by OpenAI in January 2024 throughout the notebook. Of course, the models and model names will evolve. We need to be on the watch continuously!

We will use an image from Wikipedia that is in OpenAI's example code to make sure the code works.

Also, note that the request is in the standard chat completion format that complies with OpenAI's January 2024 upgrades:

```
from openai import OpenAI
client = OpenAI()
response = client.chat.completions.create(
    model="gpt-4-vision-preview",
    messages=[
        {
            "role": "user",
            "content": [
                {"type": "text", "text": "What's in this image?"},
                {
                    "type": "image_url",
                    "image_url": {
                        "url": "https://upload.wikimedia.org/wikipedia/com",
                    },
                },
            ],
        }
    ],
    max_tokens=300,
)
```

The response is a raw output. Let's wrap it up nicely and display it:

```
import textwrap
choice = response.choices[0]
response_content = choice.message.content # Access the content
# Use textwrap to format the text into a paragraph
formatted_response = textwrap.fill(response_content, width=80)
# Printing the formatted response
print(formatted_response)
```

The output is now nicely formatted:

```
The image shows a beautiful natural landscape. A wooden boardwalk meanders through tall, lush green grass, which suggests that this is a wetland or marsh area where the boardwalk provides a walkway to prevent disturbance to the ecosystem and to keep visitors' feet dry. The scene is bathed in bright, natural light, indicating it is a sunny day with scattered clouds in the sky. The skies are a vibrant blue, adding to the overall tranquility of the landscape. Beyond the grass, there appear to be shrubbery and a line of trees in the distance, which could signify the edge of the wetland or the beginning of a forested area. Overall, the image depicts a peaceful outdoor setting that is inviting for a walk or contemplation.
```

The description is quite poetic. But is it accurate? Let's display the image and find out:

```
from IPython.display import Image, display
# URL of the image
image_url = "https://upload.wikimedia.org/wikipedia/commons/t/tall_grass_wetland.jpg"
# Display the image
display(Image(url=image_url))
```

The output displays an image (*Figure 16.22*) that shows that the description given by GPT-4V is accurate:



*Figure 16.22: Excerpt of the image retrieved and analyzed by GPT-4V*

That was an easy task for GPT-4V. Let's enter the realm of divergent semantic association.

## **Example 2: Divergent semantic association, moderate divergence**

Example 2 is the image produced by DALL-E in the *Creating an image with ChatGPT Plus with DALL-E* subsection of this section: *Figure 16.20: Advanced CV DAT of a dog produced by DALL-E*.

Can GPT-4V recognize a dog in the figure? We make the request for the standard image with no additional instructions as follows:

```
from openai import OpenAI
client = OpenAI()
response = client.chat.completions.create(
    model="gpt-4-vision-preview",
    messages=[
```

```
"role": "user",
"content": [
    {"type": "text", "text": "What's in this image?"},
    {
        "type": "image_url",
        "image_url": {
            "url": "https://raw.githubusercontent.com/Denis205"
        },
    },
],
max_tokens=300,
)
```

We now wrap the text up:

```
import textwrap
choice = response.choices[0]
response_content = choice.message.content # Access the content
# Use textwrap to format the text into a paragraph
formatted_response = textwrap.fill(response_content, width=80)
# Printing the formatted response
print(formatted_response)
```

The output shows that GPT-4V not only recognized a dog but made a quite creative description of the image:

```
This image features a highly stylized and colorful digital illustration of a dog. The artwork is composed of various swirling shapes and patterns that come together to create an abstract representation of a dog. The background is filled with speckled dots suggestive of a starry night sky, which contributes to the overall whimsical feel. A vibrant and varied color palette is used for the dog, with shades of blue, green, and yellow. The swirling patterns give the image a dynamic and energetic feel, providing a sense of movement and life.
```

The text is both accurate and beautiful. The notebook then displays the image shown in *Figure 16.20*.

Now, let's push GPT-4V to the limits of divergent semantic associations.

## Example 3: Divergent semantic association, high divergence

We will now ask GPT-4V to analyze the image produced by DALL-E:

*Figure 16.21: Extreme CV DAT of a dog produced by DALL-E.*

The DAT is daunting because the image is highly divergent. Nevertheless, we make a standard request with no additional instructions or hints:

```
from openai import OpenAI
client = OpenAI()
response = client.chat.completions.create(
    model="gpt-4-vision-preview",
    messages=[
        {
            "role": "user",
            "content": [
                {"type": "text", "text": "What's in this image?"},
                {
                    "type": "image_url",
                    "image_url": {
                        "url": "https://raw.githubusercontent.com/Denis205"
                    },
                },
            ],
        }
    ],
    max_tokens=300,
)
```

We then wrap the text up:

```
import textwrap
choice = response.choices[0]
response_content = choice.message.content # Access the content
# Use textwrap to format the text into a paragraph
formatted_response = textwrap.fill(response_content, width=80)
# Printing the formatted response
print(formatted_response)
```

The output is creative, and it grasps the divergence of the image. But it doesn't recognize the divergent dog:

This is a digitally created abstract image that features a variety of organic shapes. It includes numerous swirling elements and circular patterns that somewhat resemble brush strokes with a dynamic and fluid feel. The colors are vivid and diverse, ranging from blues and purples to oranges, yellows, and reds, all set against a black background to enhance their vibrancy. The overall composition suggests the form of a stylized animal, perhaps a bird or another kind of animal, with an eye-like shape on the right side. The overall effect is ornate and evokes a sense of motion, as if the piece could be seen as an example of modern digital art that explores the relationship between color, form, and the idea of motion.

The text doesn't recognize the dog, but does it have to? The task is about divergence and creativity.

The image produced by DALL-E and the text created by GPT-4V shows incredible creativity abilities that outperform most humans.

We are entering a new era of incredible “little C” AI creativity!

We have explored several vision models and conducted DAT experiments. Let's move on to the summary to conclude and start our next adventure!

# Summary

Natural language transformers have evolved into Foundation Models in a short time. Generative AI has reached new levels with ViT, CLIP, DALL-E, and GPT-4V.

We first explored the architecture of ViT, which breaks images down into *words*. We discovered that there is more than one way to implement models in real-world ML. Understanding the different approaches contributes to creating a personal toolbox to solve problems when implementing ML projects.

Then, we explored CLIP, which can associate words and images. Finally, we looked into the architecture of DALL-E. We went down to the tensor level to look under the hood of the structure of some of these innovative models. We then implemented the innovative DALL-E 2 and DALL-E 3 API.

Finally, we built a GPT-4V notebook with DALL-E 3 images, implementing an example of divergent semantic association.

The paradigm shift resides in the tremendous resources few organizations have to train Generative AI models on petabytes of data. These organizations have successfully rolled out mainstream Generative AI pervading mainstream environments such as Google Workspace and Microsoft 365.

The revolutionary aspect of this colossal evolution comes from the end user. An end user has suddenly emerged as a potential AI designer!

Although an end user doesn't need to understand the architecture of transformers, the situation is quite different for an AI professional.

The situation has become challenging for AI specialists caught between the pressure of major organizations on the AI market and the rising expertise of end users. A new approach is mandatory. Training a Foundation Model is

out of reach for most organizations. At the same time, end users are encroaching on what was the expertise of trained AI experts.

In the following chapters, we will discover new ways for AI professionals to emerge. We will continue this journey into the new era by exploring the exciting toolbox of ViTs in *Chapter 17, Transcending the Image-Text Boundary with Stable Diffusion*.

## Questions

1. DALL-E 2 classifies images. (True/False)
2. ViT classifies images. (True/False)
3. BERT was initially designed to generate images. (True/False)
4. CLIP is an image-clipping application. (True/False)
5. BERT uses CLIP to identify images. (True/False)
6. DALL-E 3 cannot be accessed with an API. (True/False)
7. Gradio is a transformer model. (True/False)
8. ViT can classify images that are not on its list of labels. (True/False)
9. ViT requires a prompt to respond. (True/False)
10. GPT-4V will most probably evolve into a more multimodal system.  
(True/False)

## References

- Alexey Dosovitskiy et al., 2020, *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*:  
<https://arxiv.org/abs/2010.11929>
- CLIP: <https://github.com/openai/CLIP>
- DALL-E: <https://openai.com/blog/dall-e/>

- OpenAI resources: <https://openai.com>
- *Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever, 2021, Zero-Shot Text-to-Image Generation:*  
<https://arxiv.org/abs/2102.12092>
- *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>
- GPT-4V system card: <https://openai.com/research/gpt-4v-system-card>
- Divergent semantic association: *Honghua Chen and Nai Ding, 2023, Probing the Creativity of Large Language Models: Can models produce divergent semantic association?:*  
<https://arxiv.org/abs/2310.11158>

## Further Reading

- *Alec Radford et al., 2021, Learning Transferable Visual Models From Natural Language Supervision:*  
<https://arxiv.org/abs/2103.00020>

## Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://www.packt.link/Transformers>



# 17

## Transcending the Image-Text Boundary with Stable Diffusion

The essence of a diffusion model relies on the freedom it possesses to invent pixels when generating an image. From that perspective, diffusion models have taken text-to-image generation to another level. Instead of trying to create the exact representation of images they have learned, diffusion models can imagine pixels within the boundaries of the text provided.

Stability AI is a leader in Generative AI. They produced Stable Diffusion, one of the fastest-growing AI projects. From that concept, mind-blowing applications have begun to appear in every direction, including Midjourney's application on Discord and Runway's Gen-2 that we will encounter in *Chapter 19, On the Road to Functional AGI with HuggingGPT and its Peers* and *Chapter 20, Beyond Human-Designed Prompts with Generative Ideation*.

The goal of this chapter is not to attempt to analyze the many Stable Diffusion architectures flowing into the market. Instead, we will focus on the fundamental features of diffusion models.

We will begin with a brief high-level approach to diffusion models, introducing Stable Vision, which has created a disruptive generative image AI wave rippling through the market.

We will then dive into the principles, math, and code of the remarkable Keras Stable Diffusion model. We will go through each of the main components of a Stable Diffusion model. We will peek into the source code provided by Keras and run the model.

Our journey will continue by running Stability AI's text-to-image API to generate awesome images in a few lines of code. Stability AI will take us into the world of text-to-video animations. We will run a text-to-video synthesis model with Hugging Face. Finally, we will run a video-to-text task with Meta's TimeSformer.

By the end of the chapter, you will have sufficient representation of Stable Diffusion to appreciate the wonders it produces!

This chapter covers the following topics:

- A high-level approach to Stable Diffusion
- A conceptual mathematical representation of Stable Diffusion
- Exploring the Keras Stable Diffusion's library at a code level
- Text-to-image with Stable Diffusion
- Text-to-video generation with Stability AI
- Video-to-text with Meta's TimeSformer

With all the innovations and library updates in this cutting-edge field, packages and models change regularly. Please go to the GitHub repository for the latest installation and code examples:

<https://github.com/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition/tree/main/Chapter17>.



You can also post a message in our Discord community (<https://www.packt.link/Transformers>) if you have any trouble running the code in this or any chapter.

Our first step will be understanding how diffusion models transcend the boundaries of image generation.

## Transcending image generation boundaries

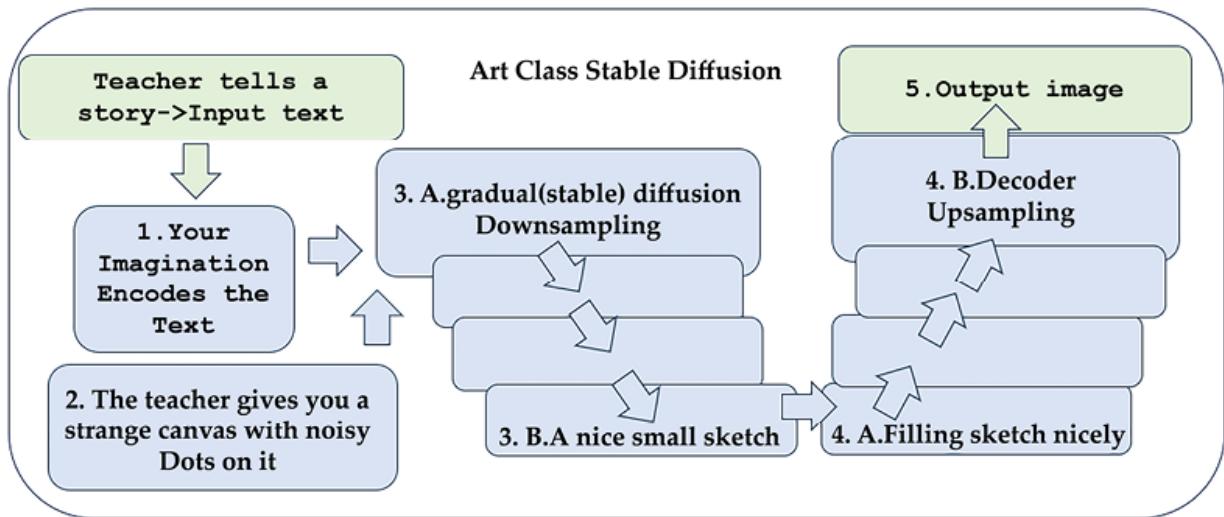
Let's begin with a thought experiment. Imagine an art teacher telling your class of students a story about visiting a wonderful house with a big garden with old trees and beautiful flowers.

Now, the teacher gives you a piece of strange canvas with many dots (pixels of *noise* in an image). This mysterious piece of paper is a potential (*latent*) space of hidden forms you must find in your mental representation of the words (*text*) the teacher spoke. As you erase the dots and replace them with your ideas, you are dispersing them (*diffusion*). You obtain a small sketch of the objects you imagined. Your drawing is incomplete, and it's a smaller view of what you thought. You just represented the main forms you saw. You *downsampled* your representation.

The fun now begins. You show each other your sketches. Although every drawing shows a house, not one is the same! Your teacher now provides incredible oil painting techniques to fill in the gaps in your sketch. You are now *upsampling* your drawing to a beautiful oil painting. During this process, everybody is careful to go step (*layer*) by gradual step in a controlled (*stable*) way so that everything looks right.

When the work of art is complete, the students of the class display their fantastic oil paintings. They all represent the variations of a beautiful house.

Let's visualize the process of this art class in *Figure 17.1*:



*Figure 17.1: Art Class Stable Diffusion*

*Figure 17.1* represents the art class process we do daily when transforming what we hear into unique personal mental images that merge with words and process them in different ways.

Notice that the path from 3. A to 3. B, then from 4. A. to 4. B forms a sort of “U” that some architectures implement as a U-Net. In any case, the overall process does look like a “U.”

No matter how complex some papers, social media posts, blogs, and documentation may seem, Stable Diffusion boils down to the Keras compact model:

- An input text that will be encoded in a context.
- Downsampling an image (higher-dimensional) containing free pixels (noise) guided by the encoded text.

- Obtaining a small but accurate representation of the main features of the image according to the encoded text.
- Upsampling the smaller (lower-dimensional) representation by filling the gaps freely, adding structures and pixels guided by the encoded text.
- Processing the producing the output (higher-dimensional) image or sequences of images (video).

Multiple architectures can achieve this diffusion process with **Variational Autoencoders (VAEs)**, **Discrete Variational Encoders (dVAEs)**, U-Nets, transformer encoder-decoders with attention, CNNs, ResNet, and more. However, no matter how expressed, it will always follow the same process as in the art class example shown in *Figure 17.1*.

Let's now explore the ingenious Stable Diffusion implementation by Keras.

# Part I: Defining text-to-image with Stable Diffusion

We will explore at a very low level the main Python files of the Keras version of Stable Diffusion, as shown in *Figure 17.2*. The complete code can be found at: [https://github.com/keras-team/keras-cv/tree/master/keras\\_cv/models/stable\\_diffusion](https://github.com/keras-team/keras-cv/tree/master/keras_cv/models/stable_diffusion):

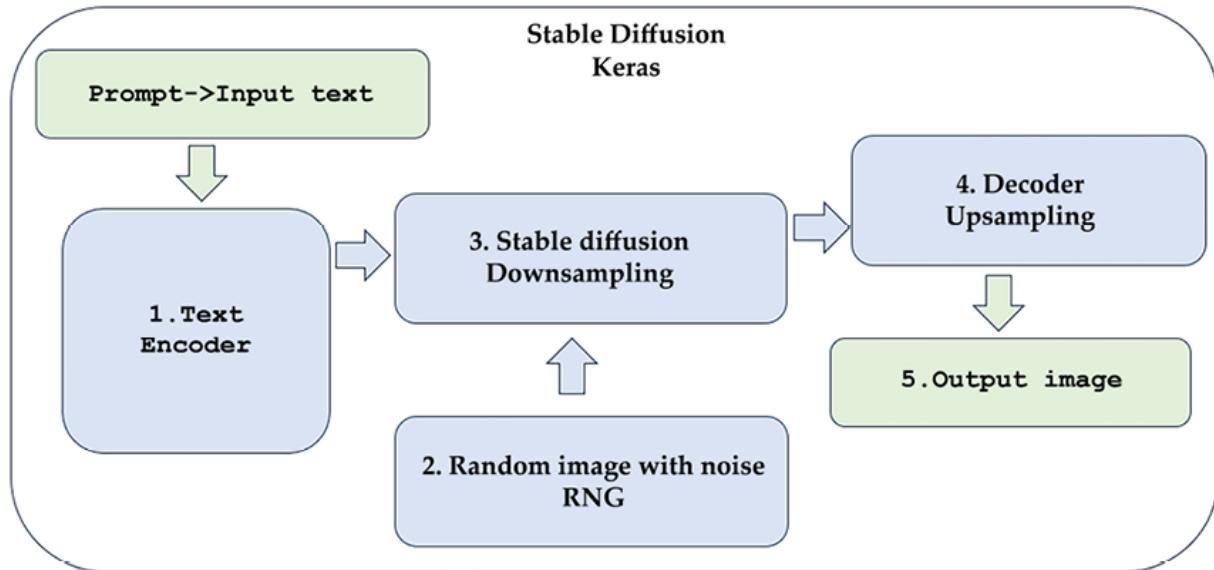


Figure 17.2: Stable Diffusion, Keras implementation

*Figure 17.2* shows the Stable Diffusion architecture of the code we will explore that can be summed up in five phases:

1. Text embedding.
2. Random image creation.
3. Stable Diffusion downsampling.
4. Decoder upsampling.
5. Output image.

The Keras Stable Diffusion code itself is only 500 lines long!

We will describe each function's function, make a high-level mathematical representation, and find the Python classes that execute the process.

We will end the analysis by running a Keras notebook illustrating their talented compact code approach.

## 1. Text embedding using a transformer encoder

**Analogy:** Imagine each word in a sentence as a person in a room. These people (words) have relationships with others, like friends, strangers, or family. The transformer is like a social expert who learns how everyone is related and connected.

**Clarification:** The term “word” is a loose umbrella term that designates the token(s) of a word after the word was tokenized.

**Math:** Let each word be a vector  $v_i$  in a high-dimensional space. After processing through the transformer, each word vector has context, becoming a contextualized vector:  $C(V_i)$ .

The process can be summed up as the creation of a contextualized token vector from the word token vectors such as:

$$C(V_i) = \text{encoder}(V_i)$$

**Code:** Let's open the hood and look into the source code of the encoder.

The source code for this subsection can be viewed on the Keras Stable Diffusion GitHub repository: [https://github.com/keras-](https://github.com/keras-team/keras-)  
[team/keras-](https://github.com/keras-team/keras-)

[cv/blob/master/keras\\_cv/models/stable\\_diffusion/text\\_encoder.py](#)

The source code contains two versions of the text encoder. We will focus only on the first class that begins with:

```
class TextEncoder(keras.Model):
    def __init__(
        self, max_length, vocab_size=49408, name=None, download=True
    ):
        tokens = keras.layers.Input(
            shape=(max_length,), dtype="int32", name="tokens"
        )
        positions = keras.layers.Input(
            shape=(max_length,), dtype="int32", name="positions"
        )...
```

The code is a `TextEncoder(keras.Model)` class.

The encoder receives two inputs: tokens (representing the words) and positions (indicating the position of each word in a sequence).

These inputs are first transformed using a `CLIPEmbedding` layer. This layer converts tokens into embeddings of dense vector representations and adds position embeddings.

The transformed inputs pass through several `CLIPEncoderLayer` layers 12 times in this version.

Each `CLIPEncoderLayer` improves these representations using a combination of layer normalization, a self-attention mechanism (`CLIPAttention`), and feed-forward neural networks.

The final representation is a vector that embeds the entire input to produce an output of the contextualized word embeddings.

## 2. Random image creation with noise

**Analogy:** Imagine you must improve an image based on a topic. The topic is open to interpretation. The image's content is vague, blurry, and full of elements that mean nothing. The first impression is that the image content contains a lot of "noise" that has nothing to do with the topic. Then your creativity kicks in, and you see this "noise" as the "degrees of freedom" you can improve as you wish!

**Clarification:** The term "image" refers to a placeholder image input into the model. Depending on the model's architecture, it could be a pre-existing image or simply a random assortment of pixels.

**Math:** The image can be thought of as a matrix  $I$ . The degree of freedom is adding a variable  $\delta$  to each pixel, so our initial image becomes  $I + \delta$ . The value of  $\delta$  is a parameter that determines the system's level of creativity.

**Code:** The introduction of noise can be achieved in many ways, such as the following code that is at: [https://github.com/keras-team/keras-cv/blob/master/keras\\_cv/models/stable\\_diffusion/stable\\_diffusion.py](https://github.com/keras-team/keras-cv/blob/master/keras_cv/models/stable_diffusion/stable_diffusion.py)

The following excerpt illustrates one approach to the noise introduction process:

```
def _get_initial_diffusion_noise(self, batch_size, seed):
    if seed is not None:
        return tf.random.stateless_normal(
```

```
        (batch_size, self.img_height // 8, self.img_width // 8,
         seed=[seed, seed],
     )
else:
    return tf.random.normal(
        (batch_size, self.img_height // 8, self.img_width // 8,
```

The function is a **Random Number Generator (RNG)** that produces random noise. With a seed argument, the RNG will produce deterministic noise (the same output runs) with pseudo-random numbers. If there is no seed, the RNG will produce non-deterministic noise.

### 3. Stable Diffusion model downsampling

**Analogy:** Imagine you are improving the “noisy” image. If you make too many modifications at a time, you might drift away from the topic. You must divide your work into small steps. At each small and gradual step, you must recheck the topic to make sure your creativity fits the topic. You also decide to downscale as if you were summarizing the main features of the image that fit the topic.

**Clarification:** The step-by-step, gradual process makes the process “stable.” You downscale the image by retaining the most representative features of the topic. By processing the image with these methods, you are “diffusing” the image.

**Math:** The diffusion process can be represented as an iterative equation:

$$I_{t+1} = I_t - \alpha \cdot (I_t - \text{desired state})$$

In this equation:

- $I$  is the matrix of the image.
- $t$  is an iteration, like a time-step in a sequence.
- $I_{t+1}$  is the next iteration.
- $\alpha$  determines the stability by controlling how slow or fast diffusion occurs.
- $I_t$  is the state of an image at an iteration.
- desired state is the target image configuration, guided by the contextualized token vectors derived from the text embeddings.
- $I_t - \text{desired state}$  is the estimation of the deviation to reduce through each iteration through the influence of the word embeddings.

**Code:** The source code referred to in this subsection can be viewed at

[https://github.com/keras-team/keras-cv/blob/master/keras\\_cv/models/stable\\_diffusion/diffusion\\_model.py](https://github.com/keras-team/keras-cv/blob/master/keras_cv/models/stable_diffusion/diffusion_model.py).

We will not go through every line of the source code but only focus on the main part of the downsampling process.

The code that represents one downsampling version begins with:

```
class DiffusionModelV2(keras.Model):
    def __init__(
        self,
        img_height,
        img_width,
        max_text_length,
        name=None,
        download_weights=True,
    ):
        context = keras.layers.Input((max_text_length, 1024))
        t_embed_input = keras.layers.Input((320,))
```

```
latent = keras.layers.Input((img_height // 8, img_width // 8))
t_emb = keras.layers.Dense(1280)(t_embed_input)
t_emb = keras.layers.Activation("swish")(t_emb)
t_emb = keras.layers.Dense(1280)(t_emb)
# Downsampling flow
outputs = []
x = PaddedConv2D(320, kernel_size=3, padding=1)(latent)
outputs.append(x)...
```

We can sum up the main functions of this part of the code as follows:

1. The downsampling flow begins by applying a convolution to the *latent* input. This result is saved in the `outputs` list.
2. Subsequent operations are done in a loop for two iterations. In each loop:
  - a. A `ResBlock` is applied with increasing depth (channels).
  - b. The `SpatialTransformer` is applied. The goal is to enhance the model's geometric invariance (translations, scales, and rotations).
  - c. The processed tensor is saved in the *outputs* list.
  - d. After every two iterations, the feature map is downsampled by a factor of 2 using a convolution with stride 2.
3. This downsampling sequence is applied across multiple resolutions (`320, 640, 1280`). As the depth increases, the spatial dimensions decrease (due to downsampling), but the representational capacity increases.

## 4. Decoder upsampling

**Analogy:** Imagine the image you have downscaled and for which the main features are sketched out. It looks conceptually good because it somewhat

follows the contextualized token vector directives. But it can't qualify as a high-definition image. We need to upscale it, add structures and pixels, and expand the best features using our creativity until we obtain a well-defined image.

**Math:** The decoder function can be represented as:

$$D(c(v_i), I_{\text{downsampled}}) \rightarrow I_{\text{upsampled}}$$

In this equation:

- $D$  is the decoder.
- $c(v_i)$  is the contextualized word vector.
- $I_{\text{downsampled}}$  is the downsampled, downscaled matrix of the image.
- $I_{\text{upsampled}}$  is the upsampled, upscaled version of the matrix produced by the decoder, marking the end of the complete diffusion process.

**Code:** The source code for this subsection can be viewed at

[https://github.com/keras-team/keras-cv/blob/master/keras\\_cv/models/stable\\_diffusion/diffusion\\_model.py](https://github.com/keras-team/keras-cv/blob/master/keras_cv/models/stable_diffusion/diffusion_model.py)

We will not go through every line of the source code but only focus on the main part of the upsampling process beginning with:

```
# Upsampling flow
for _ in range(3):
    x = keras.layers.Concatenate()([x, outputs.pop()])
    x = ResBlock(1280)([x, t_emb])
    x = Upsample(1280)(x)
    for _ in range(3):
        x = keras.layers.Concatenate()([x, outputs.pop()])
        x = ResBlock(1280)([x, t_emb])
        x = SpatialTransformer(20, 64, fully_connected=True)
        x = Upsample(1280)(x)...
```

The code follows an upsampling flow similar to the decoding half of a U-Net architecture. During upsampling, features from the downsampling path (stored in the outputs list) are concatenated with the upsampled features to provide high-resolution features to the model. It follows the reverse path of the downsampling process.

## 5. Output image

**Analogy:** You have completed the whole process of improving the initial image with your imagination and creativity. The well-defined output image is the culmination of text embeddings in the stable diffusion model.

**Math:** The output image is essentially the result of all the previous operations:

$$I_{\text{output}} = D(c(v_i), I_{\text{diffused}})$$

**Code:** The exit flow of the source code for this subsection can be viewed at [https://github.com/keras-team/keras-cv/blob/master/keras\\_cv/models/stable\\_diffusion/diffusion\\_model.py](https://github.com/keras-team/keras-cv/blob/master/keras_cv/models/stable_diffusion/diffusion_model.py)

The exit flow begins with the following code:

```
# Exit flow
    x = keras.layers.GroupNormalization(epsilon=1e-5)(x)
    x = keras.layers.Activation("swish")(x)
    output = PaddedConv2D(4, kernel_size=3, padding=1)(x)...
```

The exit flow is a classical deep learning output process with:

- Group normalization to adjust the activations of the network.
- A Swish activation layer introduced by Google: `swish(x) = x * sigmoid(x)`.
- A padded convolution layer to prepare the final output.

## Running the Keras Stable Diffusion implementation

We saw the talented minimalist code that drives the minimalist Keras implementation of Stable Diffusion.

Now, let's watch the magic of the result. Open `Stable Diffusion_Keras.ipynb` in the chapter directory of the GitHub repository.

We first make sure TensorFlow is installed:

```
#Importing openai
try:
    import tensorflow as tf
    print(tf.__version__)
except:
    !pip install tensorflow
    import tensorflow as tf
    print(tf.__version__)
```

Then we install the Keras computer vision library and `keras_core`, an improved API backend:

```
!pip install keras_cv --upgrade --quiet
!pip install keras_core --upgrade --quiet
```

The simplicity of the approach is impressive. We now import the modules we need:

```
import time
import keras_cv
from tensorflow import keras
import matplotlib.pyplot as plt
```

We select our Keras model:

```
model = keras_cv.models.StableDiffusion(img_width=512, img_hei
```

The model functions as we saw in the previous sections. The model will run the text encoder, then gradually “denoise” a 64x64 latent image patch with the diffusion model. Finally, it will transform the “denoised” 64x64 patch into a higher resolution (sometimes referred to as “super-resolution”) 512x512 image.

We can now generate images with a prompt and plot the images:

```
images = model.text_to_image("photograph of an astronaut on Ma
def plot_images(images):
    plt.figure(figsize=(20, 20))
    for i in range(len(images)):
        ax = plt.subplot(1, len(images), i + 1)
        plt.imshow(images[i])
        plt.axis("off")
plot_images(images)
```

The output is stunning for this compact code:



Figure 17.3: Image of an astronaut created by Stable Diffusion

We can try the more complex fun prompt the Keras team imagined:

```
images = model.text_to_image(  
    "cute magical flying dog, fantasy art, "  
    "golden color, high quality, highly detailed, elegant, sharp fo  
    "concept art, character concepts, digital painting, mystery, a  
        batch_size=3,  
    )  
plot_images(images)
```

The output is excellent:

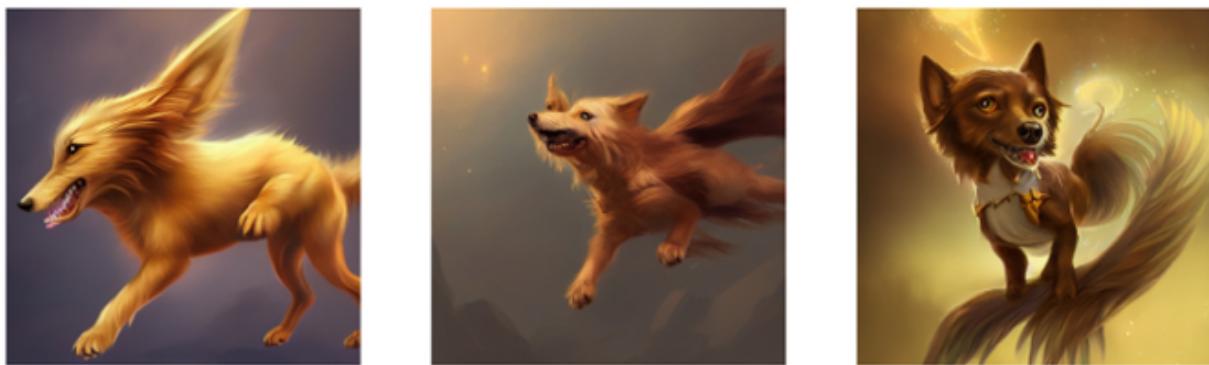


Figure 17.4: Image of a flying dog created by Stable Diffusion

The Keras implementation of Stable Diffusion represents a fantastic educational and productive approach that we can only commend.

We will now expand our experimentation with Stability AI's Diffusion API.

# Part II: Running text-to-image with Stable Diffusion

Stability AI is a leader in Generative AI. They produced Stable Diffusion, one of the fastest-growing AI projects. This section takes you to the forefront of Stable Diffusion. To get started, you must sign up and obtain an API key: <https://platform.stability.ai/docs/getting-started/python-sdk>. Check the pricing policy before running the Stable Diffusion model.

Open `stable_Vision_Stability_AI.ipynb`.

We first install the SDK:

```
!pip install stability-sdk
```

Then we clone the Stability modules:

```
!git clone --recurse-submodules https://github.com/Stability-AI/stability
```

We define the Stability host, which is where to connect to the Stability API server, and our key, or define it when making the request:

```
!export STABILITY_HOST=grpc.stability.ai:443
#!export STABILITY_KEY=[YOUR_KEY]
```

We can now ask Stability to generate an image based on the following prompt:

```
!python -m stability_sdk generate "A stunning spaceship with s
```

If you did not include your API key in the request or if the key wasn't taken into account, you will be prompted (once or possibly twice) to enter it:

```
Please enter your API key from dreamstudio.ai or set the STABI  
Enter your Stability API key:[YOUR_KEY]
```

You will get a new image when you enter your API key more than once. Once the image has been generated, we can display it:

```
import os  
from IPython.display import display, Image  
for file_name in os.listdir('/content'):  
    if file_name.endswith('.png'):  
        display(Image(filename=os.path.join('/content', file_n
```

The output is stunning, as always, with diffusion models:



Figure 17.5: Image of a spaceship created by Stable Diffusion

Stability AI has more in reserve!

Let's take this further by asking Stable Diffusion to perform a DAT.

## Generative AI Stable Diffusion for a Divergent Association Task (DAT)

Go to the [Generative AI Stable Diffusion for a Divergent Association Task](#) section in the notebook.

In *Chapter 16, Beyond Text: Vision Transformers in the Dawn of Revolutionary AI*, we explored Divergent Semantic Association and implemented DAT tasks. In this section, let's have `eating spaghetti` as the first prompt and then expand the text of the prompt to something more divergent, such as `eating spaghetti on a merry-go-round in space`.

The notebook first runs a standard text-to-image task:

```
!python -m stability_sdk generate "eating spaghetti"
```

Now we can display the output:

```
import os
from IPython.display import display, Image
for file_name in os.listdir('/content'):
    if file_name.endswith('.png'):
        display(Image(filename=os.path.join('/content', file_n
```

The image in *Figure 17.6* shows a classical image of “eating spaghetti.”



*Figure 17.6: A person eating quite a plate of spaghetti with chopsticks*

The image will vary from run to run due to the stochastic nature of transformer models.

Now, let’s try a DAT and ask the model to create an unusual image of a person eating spaghetti on a merry-to-round in space:

```
!python -m stability_sdk generate "eating spaghetti on a merry-
```

The result is “eating spaghetti” in a more amusing context, as we can see in *Figure 17.7*:



*Figure 17.7: Eating spaghetti in space*

The merry-go-round is easy to identify. The spaghetti might be the blob to the right of the merry-go-round. Space is well represented. It’s a very creative, divergent image from standard representations.

We are getting used to being outperformed by AI in creativity! Depending on your project, you can ask for more or less divergent images.

We will now go from text-to-image to text-to-video.

# Part III: Video

Text-to-video opens new horizons for diffusion models. The models generate frames and make incredible animations and videos.

Open `Stable_Vision_Stability_AI_Animation.ipynb`.

## Text-to-video with Stability AI animation

First, make sure you have signed up on Stability AI and have your API key: <https://platform.stability.ai/docs/features/animation>.

We will now install the Stability SDK for animations:

```
| !pip install "stability_sdk[anim_ui]" # Install the Animation .  
| !git clone --recurse-submodules https://github.com/Stability-A
```

We import the API and initialize the host. We also set our API key:

```
| from stability_sdk import api  
| STABILITY_HOST = "grpc.stability.ai:443"  
| STABILITY_KEY = [ENTER YOUR KEY HERE]  
| context = api.Context(STABILITY_HOST, STABILITY_KEY)
```

We now import the modules and configure the parameters. The following code uses the default Stability AI arguments:

```
from stability_sdk.animation import AnimationArgs, Animator
# Configure the animation
args = AnimationArgs()
args.interpolate_prompts = True
args.locked_seed = True
args.max_frames = 48
args.seed = 42
args.strength_curve = "0:(0)"
args.diffusion_cadence_curve = "0:(4)"
args.cadence_interp = "film"
```

The default arguments are:

- `from stability_sdk.animation import AnimationArgs, Animator`: This imports the `AnimationArgs` and `Animator` classes from the `stability_sdk.animation` module. These classes are used to configure and generate animations.
- `args = AnimationArgs()`: This creates an instance of the `AnimationArgs` class. This object will be used to configure the animation.
- `args.interpolate_prompts = True`: This sets the `interpolate_prompts` property to `True`. This means that the prompts will be interpolated between frames.
- `args.locked_seed = True`: This sets the `locked_seed` property to `True`. This means that the seed will be locked, which makes sure that the animation is deterministic.
- `args.max_frames = 48`: This sets the `max_frames` property to 48. This means that the animation will have 48 frames.
- `args.seed = 42`: This sets the `seed` property to 42. This is the random seed that will be used to generate the animation.

- `args.strength_curve = "0:(0)"`: This sets the `strength_curve` property to “0:(0)”. This is a string that specifies the strength of the diffusion curve at each frame.
- `args.diffusion_cadence_curve = "0:(4)"`: This sets the `diffusion_cadence_curve` property to “0:(4).” This is a string that specifies the cadence of the diffusion curve at each frame.
- `args.cadence_interp = "film"`: This sets the `cadence_interp` property to “film.” This specifies the interpolation method that will be used for the cadence curve.

We now define the text of the start image and the end image for the animation:

```
animation_prompts = {
    0: "a photo of a fantastic spaceship",
    24: "a photo of a fantastic lunar landing module",
}
negative_prompt = ""
```

You can add the things you don’t want to see in the negative prompt. For example:

```
negative_prompt = "trucks, clouds, Earth"
```

We now create the animator object:

```
# Create Animator object to orchestrate the rendering
animator = Animator(
    api_context=context,
    animation_prompts=animation_prompts,
```

```
    negative_prompt=negative_prompt,  
    args=args
```

We save each frame generated:

```
# Render each frame of animation  
for idx, frame in enumerate(animator.render()):  
    frame.save(f"frame_{idx:05d}.png")
```

The animator object now generates the animation frames and then makes a video:

```
from stability_sdk.utils import create_video_from_frames  
from tqdm import tqdm  
animator = Animator(  
    api_context=api.Context(STABILITY_HOST, STABILITY_KEY),  
    animation_prompts=animation_prompts,  
    negative_prompt=negative_prompt,  
    args=args,  
    out_dir="video_01"  
)  
for _ in tqdm(animator.render(), total=args.max_frames):  
    pass  
create_video_from_frames(animator.out_dir, "video.mp4", fps=24)
```

We then can display the video:

```
from IPython.display import display, Video  
display(Video("/content/video.mp4", embed=True))
```

The output is impressive, and the potential is limitless!



Figure 17.8: Video of a spaceship created by Stable Diffusion

You can run the animation in the notebook.

You can download one of the sample animations produced from the GitHub repository at the following link:

[https://github.com/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition/blob/main/Chapter17/Stability\\_Animation.mp4](https://github.com/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition/blob/main/Chapter17/Stability_Animation.mp4).

We will now run a text-to-video synthesis.

## Text-to-video, with a variation of OpenAI CLIP

This section will run an implementation of OpenAI CLIP for text-to-video synthesis hosted by Hugging Face. In *Chapter 16, Beyond Text: Vision Transformers in the Dawn of Revolutionary AI*, we explored CLIP's architecture for images. A video is a sequence of image frames, which makes it interesting to see how CLIP was deployed for text-to-video,

including diffusion functionality. The model was trained with about 1.7 billion parameters.

Open `Text_to_video_synthesis.ipynb` in the chapter's directory of the repository.

We will first install the necessary libraries and import the modules:

```
!pip install modelscope==1.4.2
!pip install open_clip_torch
!pip install pytorch-lightning
from modelscope.pipelines import pipeline
from modelscope.outputs import OutputKeys
```

Let's go through the installation and import process:

- `!pip install modelscope==1.4.2`: This installs the ModelScope library to track, monitor, and explain the performance of machine learning models.
- `!pip install open_clip_torch`: This installs the Open-CLIP library for text-to-image processes.
- `!pip install pytorch-lightning`: This installs the PyTorch Lightning library, which makes it easier to develop and train PyTorch models.
- `from modelscope.pipelines import pipeline`: This imports the `pipeline` function from the `modelscope.pipelines` module. The `pipeline` function can be used to create a pipeline for tracking and monitoring a machine-learning model.
- `from modelscope.outputs import OutputKeys`: This imports the `OutputKeys` class from the `modelscope.outputs` module. The `OutputKeys` class defines the names of the outputs that can be tracked by ModelScope.

Let's create a video with the `text-to-video-synthesis` model:

```
p = pipeline('text-to-video-synthesis', 'damo/text-to-video-sy  
test_text = {  
    'text': 'A car flying in space.',  
}  
output_video_path = p(test_text, )[OutputKeys.OUTPUT_VIDEO]  
print('You can play or save the video from the output_video_pa
```

The video can be displayed:

```
from IPython.display import display, Video  
display(Video(output_video_path, embed=True))
```

It can also be saved:

```
# Download result  
from google.colab import files  
files.download(output_video_path)
```

Try some prompts to see what happens with this experimental model!

We will now run Meta's video-to-text model.

## A video-to-text model with TimeSformer

Meta TimeSformer is a transformer-based architecture. TimeSformer will first encode each frame into a sequence of features. The features will then be

passed to the decoder stack. The model produces raw logit predictions as in a standard transformer. The logits are then converted into labeled projections.

TimeSformer will process video frames separately in a sequence and attend to the temporal relationships between the frames.

The TimeSformer relies on the PyAv library to first decode the video into a NumPy array. The NumPy array contains the frames of the video in the order they appear in the video. The sample rate and content length determine how many frames will be decoded. The TimeSformer uses the sequence of sampled frames to make predictions.

Finally, we will perform a label-to-image task with Stability AI Stable Diffusion using the output of the TimeSformer model as the input of a Stable Diffusion model. The goal is to open the door to enhancing label predictions and other tasks with diffusion models.

Open `Timesformer.ipynb` for this section. Let's begin with the initial functions.

## Preparing the video frames

The first part of the code installs the libraries we will need and builds a PyAv function to read the video, extracts the sample of frames, and stores them in a list for the TimeSformer.

We first install transformers:

```
!pip install transformers
```

Now we install the PyAv interface for FFmpeg's multimedia libraries to control FFmpeg in Python:

```
!pip install av
```

We import the modules we need to display the video and Hugging Face Hub:

```
from IPython.display import HTML
from base64 import b64encode
from huggingface_hub import hf_hub_download
```

Now, we download the target video from the Hugging Face Hub and read it:

```
file_path = hf_hub_download(
    repo_id="nielsr/video-demo", filename="eating_spaghetti.mp4"
# Load video
with open(file_path, 'rb') as f:
    video_data = f.read()
We display the video to verify that the code works:
# Display video
HTML("""
<video width="320" height="240" controls>
    <source src="data:video/mp4;base64,{0}" type="video/mp4">
</video>
""").format(b64encode(video_data).decode()))
```

Now, the notebook defines TimeSformer as the model:

```
from transformers import TimesformerConfig, TimesformerModel
# Initializing a TimeSformer timesformer-base style configuration
configuration = TimesformerConfig()
# Initializing a model from the configuration
model = TimesformerModel(configuration)
# Accessing the model configuration
configuration = model.config
```

The classification modules are imported, and a random seed is activated:

```
import av
import torch
import numpy as np
from transformers import AutoImageProcessor, TimesformerForVideoClassification
from huggingface_hub import hf_hub_download
np.random.seed(0)
```

Setting the random seed to 0 forces the system to reproduce the experiment each time.

We now define a function with PyAv to decode the video and store each frame in the list of frames that begins as an empty list.

The `frames` list is appended frame by frame as the video is decoded:

```
def read_video_pyav(container, indices):
    """
    Decode the Video with PyAV decoder.
    Args:
        container (`av.container.input.InputContainer`): PyAV container
        indices (`List[int]`): List of frame indices to decode
    Returns:
        result (np.ndarray): np array of decoded frames of shape [n_frames, height, width, 3]
    """
    frames = []
    container.seek(0)
    start_index = indices[0]
    end_index = indices[-1]
    for i, frame in enumerate(container.decode(video=0)):
        if i > end_index:
            break
        if i >= start_index and i in indices:
            frames.append(frame)
    return np.stack([x.to_ndarray(format="rgb24") for x in frames])
```

We need a sampling function that will sample a number of frames per second, for example, for the length of the video:

```
def sample_frame_indices(clip_len, frame_sample_rate, seg_len)
    converted_len = int(clip_len * frame_sample_rate)
    end_idx = np.random.randint(converted_len, seg_len)
    start_idx = end_idx - converted_len
    indices = np.linspace(start_idx, end_idx, num=clip_len)
    indices = np.clip(indices, start_idx, end_idx - 1).astype(int)
    return indices
```

The video clip contains 10 seconds of 30 frames per second:

```
# video clip consists of 300 frames (10 seconds at 30 FPS)
file_path = hf_hub_download(
    repo_id="nielsr/video-demo", filename="eating_spaghetti.mp4")
container = av.open(file_path)
```

The notebook now samples 8 frames:

```
# sample 8 frames
indices = sample_frame_indices(clip_len=8, frame_sample_rate=1)
video = read_video_pyav(container, indices)
```

The program selects a clip of 8 seconds, `clip_len=8`, out of the 10 seconds with a sample rate of 1, `frame_sample_rate=1`, adding up to 8 frames.

# Putting the TimeSformer to work to make predictions on the video frames

The code implements an image processor that will use Meta's TimeSformer to make predictions about the video frames (the text displayed at the end of the code):

```
image_processor = AutoImageProcessor.from_pretrained("MCG-NJU/")
model = TimesformerForVideoClassification.from_pretrained("facebook/timesformer-base")
inputs = image_processor(list(video), return_tensors="pt")
```

The TimeSformer takes the video frames as inputs. The `image_processor()` function takes a list of frames as input and returns a dictionary of tensors that can be used as inputs for the TimeSformer model.

The model will now make predictions with the inputs:

```
with torch.no_grad():
    outputs = model(**inputs)
    logits = outputs.logits
```

The `torch.no_grad()` means no gradients will be tracked for the inference tasks.

The model produces logits. These logits now need to be converted into labels:

```
# model predicts one of the 400 Kinetics-400 classes
predicted_label = logits.argmax(-1).item()
```

The notebook now prints the output label:

```
print(model.config.id2label[predicted_label])
```

In this case, `eating spaghetti` has been identified:

```
eating spaghetti
```

The door is open to unlimited creativity!

We can now summarize our journey and go on to our next adventure.

## Summary

Stable Diffusion has transcended the boundaries of classical AI imagery. Introducing creative freedom (“noise”) through diffusion in a latent space has opened the doors to huge generative computer vision possibilities.

We began the chapter by going through the Stable Diffusion process with a thought experiment and then with the talented Keras implementation. We went through the encoding of a contextualized input text, introducing a “noisy” (open to creativity) image patch, applying diffusion to this image to reduce (downsampling) it to a lower dimension, and then upsampling it to a 512x512 image patch. The output was astonishing for such a compact source code.

We then ran a Stability AI text-to-image notebook that also generated surprising images. We once again saw that diffusion is taking us to levels we never would have imagined including divergent association tasks.

Stability AI also provided a text-to-animation API to transform one image into another with text-to-animation magic.

We continued text-to-video with an OpenAI CLIP implementation that showed that we could generate efficient videos on the Hugging Face platform.

Finally, we added a video-to-text classifier to our text-image/video models. Meta's TimeSformer can process sampled video frames to make inferences and produce a text output describing the video. Although TimeSformer is not a diffusion model, the output could be a fascinating way to generate new images, such as a label-to-image task!

The idea that emerges from these models is that anything is possible!

We might think we have reached the end of computer vision innovations. However, we just got started!

In *Chapter 18, Hugging Face AutoTrain: Training Vision Models without Coding*, we will automate the training of vision transformers, which will take us a step further into this exciting new era.

## Questions

1. Stable Diffusion requires a text encoder. (True/False)
2. Stable Diffusion requires diffusion layers. (True/False)
3. A Keras Stable Diffusion reduces a noisy image to a lower dimensionality. (True/False)
4. A Keras Stable Diffusion model upsamples an image once it is downsampled. (True/False)
5. The final output of a diffusion model is a “noisy” image. (True/False)
6. OpenAI CLIP cannot produce a text-to-video model yet. (True/False)

7. Stability AI cannot convert one image to another in a video.  
(True/False)
8. Meta's TimeSformer is a scheduling algorithm, not a computer vision model. (True/False)
9. It will never be possible to create a complete movie automatically.  
(True/False)
10. There is a hardware limit to generate videos automatically beyond 10 seconds. (True/False)

## References

- Keras Stable Diffusion:  
[https://keras.io/guides/keras\\_cv/generate\\_images\\_with\\_stable\\_diffusion/](https://keras.io/guides/keras_cv/generate_images_with_stable_diffusion/)
- Stability AI: <https://stability.ai/>
- Stability AI Stable Diffusion:  
<https://stability.ai/stablediffusion>
- OpenAI CLIP implementation with ModelScope:  
<https://huggingface.co/damo-vilab/modelscope-damo-text-to-video-synthesis>
- TimeSformer:  
[https://huggingface.co/docs/transformers/model\\_doc/timesformer](https://huggingface.co/docs/transformers/model_doc/timesformer)

## Further reading

- Stability AI research blog: <https://stability.ai/research>

# Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://www.packt.link/Transformers>



# 18

## Hugging Face AutoTrain: Training Vision Models without Coding

Training a machine learning model doesn't require a college degree anymore. Somebody with no AI or programming knowledge can train a transformer model.

Hugging Face's AutoTrain requires no coding. You just need to upload your data. AutoTrain will then automatically process the data and choose and train one or several models for your project. You can then deploy the trained model in a few clicks.

You can find similar services on Google AI Platform AutoML, Amazon SageMaker Autopilot, Microsoft Azure Machine Learning, IBM Watson Studio, and a growing number of other platforms.

In less than a few years, anybody can compete with somebody with a Ph.D. in AI who spent years in college. With automated machine learning platforms, a powerful copilot such as ChatGPT, and a credit card, anybody can compete with an AI engineer.

This chapter will show how anybody can train a vision transformer with absolutely no ML knowledge or experience. However, we also see that at each step of the process, a college degree will provide the necessary insight

when a problem arises. And in a complex domain such as AI, things often go wrong, as we will see!

We will begin by creating a project with Hugging Face's AutoTrain to train an image classification model on CIFAR-10 transportation images. We will load the data to the AutoTrain interface after splitting the data into training and validation datasets. Then, we will activate AutoTrain's ability to train several models with acceptable to high accuracy. AutoTrain's comprehensive metrics will rank the trained models. Finally, we will deploy and run the model to test some images.

You may think you're dreaming, but all of this is true. Everything runs smoothly until problems come up. Then what? Only an AI expert can tell you how to solve the issues and do the troubleshooting!

A highly trained and experienced AI expert is like a car insurance policy. We think we're paying a lot for nothing most of the time. But when something bad happens, we rely on our policy. In this chapter, we will show that AI projects, automated or not, always have issues that AI experts must resolve.

We will go through the automated training process and discover the unpredictable problems that show why even automated ML requires human AI expertise. The goal of this chapter is also to show how to probe the limits of a computer vision model, no matter how sophisticated it is.

This chapter covers the following topics:

- Hugging Face AutoTrain
- Creating a new AutoTrain project
- Creating a Space
- Uploading a dataset
- Choosing a training strategy

- Running AutoTrain
- Retrieving validation images
- Inference with image classification
- Exploring the trained models
- Probing the limits of the trained models
- Managing the limitations of the trained models



With all the innovations and library updates in this cutting-edge field, packages and models change regularly. Please go to the GitHub repository for the latest installation and code examples:

<https://github.com/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition/tree/main/Chapter18>.

You can also post a message in our Discord community (<https://www.packt.link/Transformers>) if you have any trouble running the code in this or any chapter.

Our first step will be to define the scope and goal of the Hugging Face AutoTrain implementation in this chapter.

## Goal and scope of this chapter

AI systems have become more automated but also more complex. *This chapter aims to point out the expertise a non-AI user may require in some cases from an AI expert to get the job done.*

This chapter describes the main steps of how to implement Hugging Face AutoTrain for an image classification task on CIFAR-10 transportation images.

The scope remains to implement an automated training process on a cloud platform.

While going through the process, we will point out some, not all, of the issues that may come up to show the role of an AI professional in the ever-changing workplace.

Before we begin, we will go through some key things to note:

- Through hard work and creativity, Hugging Face managed to make transformers accessible. They also deserve credit for anticipating transformer technology's huge success years before it reached mainstream users with ChatGPT.
- The goal of this chapter is not to oppose college graduate AI professionals or AI experts who acquired their expertise through experience, books, and online courses. Any AI professional who does an excellent job commands the most profound respect in this very competitive market.
- Hugging Face AutoTrain can perform many training tasks without coding. The goal of this chapter is not to criticize this useful time-saving system. The scope of this chapter is, on the contrary, to show how an AI professional can help a non-AI professional with the inevitable troubleshooting that occurs in many AI projects.

Let's get started.

## Getting started

The Hugging Face platform is continuously upgraded and modified to adapt to the AI market. The interfaces are constantly evolving through competition and user feedback. As such, this section describes the process and provides the links to get the job done. *Focus on the processes. The interface will continuously evolve. Modern AI requires adaptability and continuous learning.*



Make sure to read Hugging Face's auto-training conditions and the cost of this service before proceeding to activate anything on the platform.

To get started, first, go to Hugging Face's AutoTrain platform:

<https://huggingface.co/autotrain>.

You will see that Hugging Face insists on creating “powerful models without code.”

Make sure to follow the instructions that will also evolve but are worth the time it takes. Make sure to obtain your Hugging Face token.

The AI professional sees the opportunity to save time and energy. The non-AI professional jubilates!

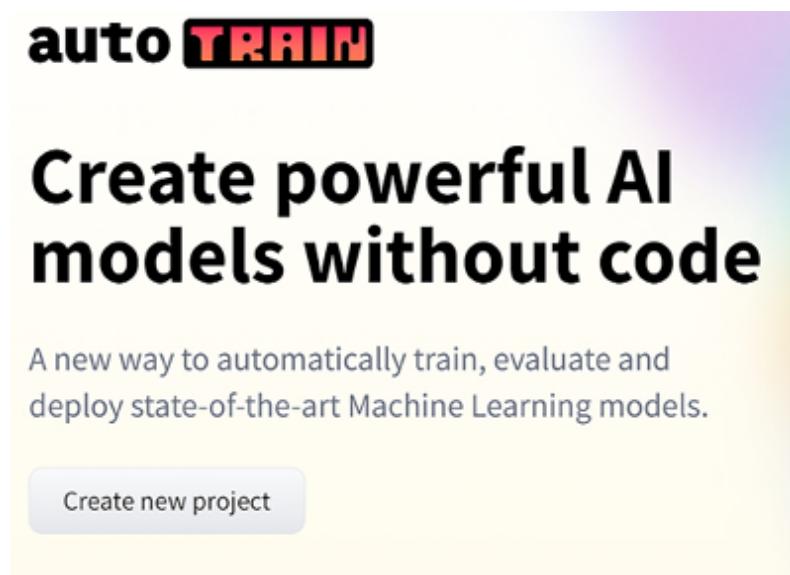
Now, the first obstacle appears for everyone: rapidly creating a new project and beginning to train the model. Or choosing an advanced project that requires:

- Creating a Space and making sure to set up the `HF_TOKEN` with your token: a Git repository that will host the demo.
- Selecting a **Software Development Kit (SDK)** for your Space: Streamlit, Gradio, Static, or Docker, along with a Docker template.

- Choosing the hardware. CPUs are free. You will need a GPU charged per hour if you have a large dataset.
- Choosing the AutoTrain option while creating your Space.
- Deciding whether to make the project public or private.

And there you go! A non-AI professional attracted by no-coding will need the help of an AI professional to get started with an advanced training project. If you wish to use a Space to train the model, follow the very well-expressed instructions provided by Hugging Face.

Let's assume the non-AI professional goes directly to creating a new project and creates a Space first. We click on **Create new project**, as shown in *Figure 18.1*:



*Figure 18.1: Creating a new project*

The process might evolve like many other innovative cloud AI platforms. The interface may change, but you will have to create a new project. It may seem fun for geeks, but not for others. In any case, we must adapt to this fast-moving era!

You will be prompted to complete a project form when you click on **New Project**. Follow the instructions carefully.

Enter your project name and select the vision option for this project to activate an image classification task. You can choose a model or let the system do it for you. Hugging Face recommends letting the system decide. Click on the confirmation button to create your project.

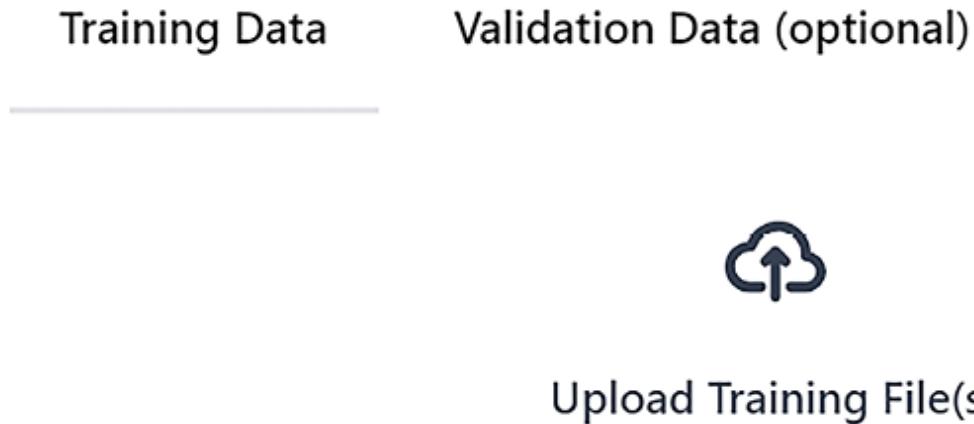
The training process includes two main steps:

- Data management
- Training phase

We will begin with data.

## Uploading the dataset

First, click on **Upload Training File(s)** as shown in *Figure 18.2*:



*Figure 18.2: Uploading the training dataset*

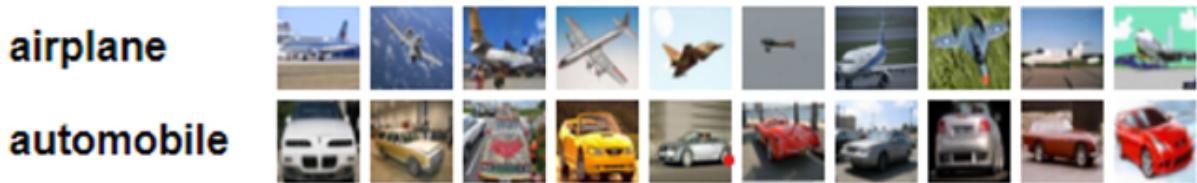
The interface may evolve, but you will need to upload data. You will need to read the documentation on the Hugging Face platform carefully to prepare

your data. The interface evolves constantly to follow the cutting-edge AI market. Choose your method but remain focused on the task: loading data.

You will need to follow Hugging Face's procedures for data formatting:  
[https://huggingface.co/docs/autotrain/image\\_classification](https://huggingface.co/docs/autotrain/image_classification).

Make sure to read the upgrades regularly. Again, it is worthwhile!

In this case, we are loading CIFAR-10 images as shown in the *Figure 18.3* excerpt:



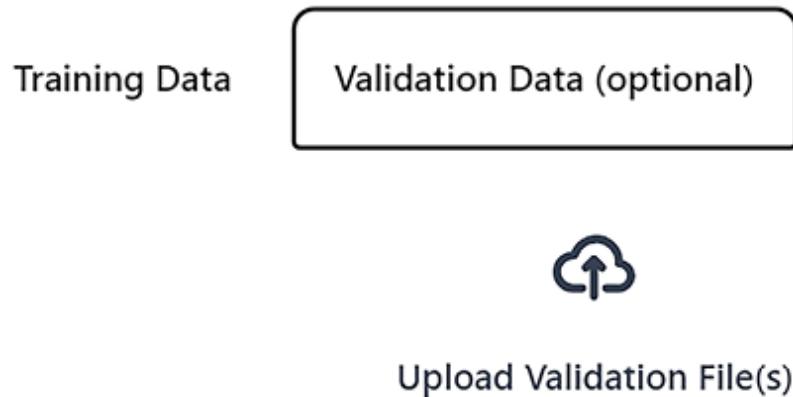
*Figure 18.3: Excerpt of CIFAR-10 transportation images*



The CIFAR-10 images in this chapter are from *Learning Multiple Layers of Features from Tiny Images*, Alex Krizhevsky, 2009: <https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf>. They are part of the CIFAR-10 and CIFAR-100 datasets ([toronto.edu](http://www.cs.toronto.edu)): <https://www.cs.toronto.edu/~kriz/cifar.html>.

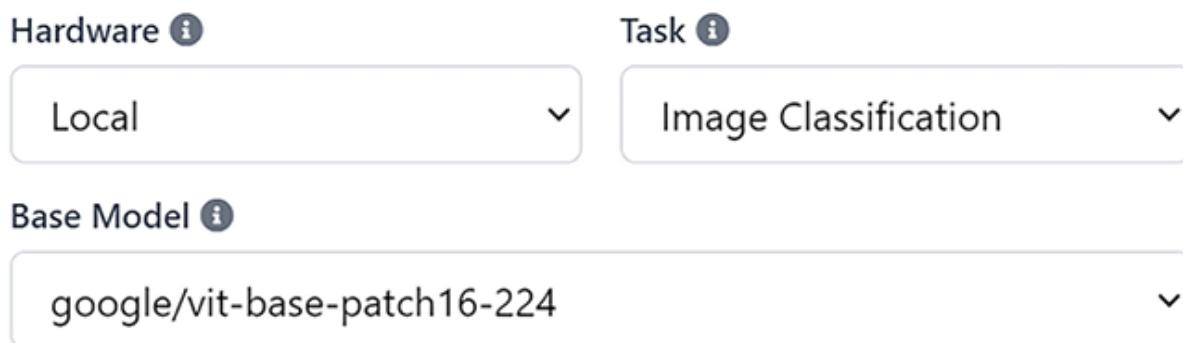
We have 8,941 images to train on, divided into four classes: airplane, automobile, ship, and truck. We will need to split the dataset into train and

validation images. Once the validation files are ready, click on **Validation Data (optional)**, as shown in *Figure 18.4*:



*Figure 18.4: Uploading the validation files*

Once the files are ready, we must choose a **Task** and a **Base Model**, as shown in *Figure 18.5*:



*Figure 18.5: Choosing a Task and a Base Model*

You can choose an LLM task or a vision task. In this case, we chose **Image Classification**, as shown in *Figure 18.6*:

## LLM Finetuning

LLM SFT

LLM Generic

LLM DPO

LLM Reward

## Other Text Tasks

Text Classification

Sequence To Sequence

## Image Tasks

DreamBooth LoRA

Image Classification

## Tabular Tasks

Tabular Classification

Tabular Regression

Image Classification



*Figure 18.6: Choosing a task*

You can choose a **Base Model** among several Hugging Face models (see *Figure 18.7*):

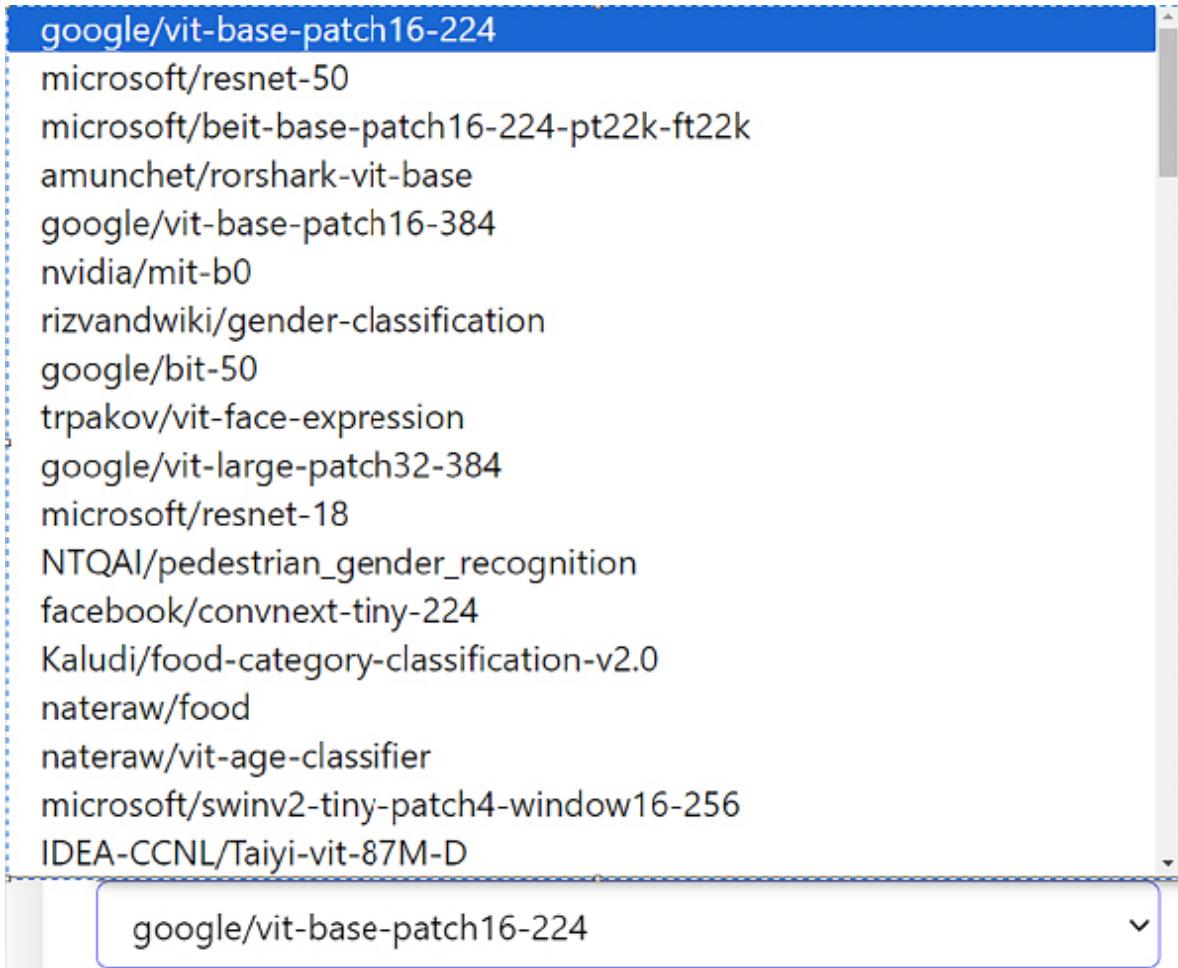


Figure 18.7: Choosing a base model

In this case, we select **Image Classification** as the task and the first model we want to train, which can be, for example, `google/vit-base-patch16-224`.

The interface may evolve, but we will always have to prepare the data before running a training job and choose a task and a model.

Everything seems fine. Wait a minute! How was the dataset prepared?

## No coding?

We just uploaded CIFAR-10, which contains well-prepared data. Maybe some users have well-prepared datasets in their company or for a project.

That is great!

But what about the data for a specific project? What if we need to train computer vision for a self-driving car? What if all the images are just raw files stacked in various directories across an organization?

A project manager will rapidly face some tough questions:

- Who prepares the data in that case?
- How much will this cost?
- How many images are required for the model to be correctly trained?
- What model(s) are we talking about in the first place?
- How much automation and how much coding does this involve?
- What about privacy if we are uploading sensitive data?

The bottom line is:

- No coding and no training will work for well-prepared datasets.
- Coding, training, and security management may be involved for specific datasets.

A project manager might conclude that the answer to specific project constraints might be to:

- Clone a GitHub repository on a private server.
- Train the model on that private server with private datasets.

Hugging Face offers deployments on AWS, for example.

We can continue without worrying about these issues for our experiment since we are using a public, well-prepared CIFAR-10 dataset.

## Training models with AutoTrain

When we activate the training process, Hugging Face first downloads our data from the Hugging Face Hub, where we uploaded our dataset.

Hugging Face's AutoTrain interface provides a rapid, seamless, and intuitive approach to training text and vision models. Just click on **Start Training** when you are ready, as shown in *Figure 18.8*:



Start Training

*Figure 18.8: AutoTrain: an intuitive approach*

Hugging Face AutoTrain's module acts as an overall controller for your training process. For more, read Hugging Face's documentation, which evolves constantly as the platform makes progress:

<https://huggingface.co/docs/autotrain/main/en/index#who-should-use-autotrain>.

Once a model is trained, we can train other ones to create a trained model set of potential candidates for our project.



Reminder: Evaluate the cost of training before running a training process. Budget management remains a critical factor in training AI models.

In this case, several modes were trained. You can explore the Hugging Face platform for the many services offered. You can find a list of your models in your profiles under **Models**, as shown in *Figure 18.9*:

 Models

Denis1976/autotrain-training-cifar-10-81128141662

Image Classification • Updated Aug 9, 2023 • ↴ 6

Denis1976/autotrain-training-cifar-10-81128141657

Image Classification • Updated Aug 9, 2023 • ↴ 7

Denis1976/autotrain-training-cifar-10-81128141660

Image Classification • Updated Aug 9, 2023 • ↴ 6

*Figure 18.9: Excerpt of a list of trained models*

You can analyze the metrics of a model by clicking on one of your trained models, which will display the model card (*Figure 18.10*):

## Model Trained Using AutoTrain

- Problem type: Multi-class Classification
- Model ID: 81128141662
- CO2 Emissions (in grams): 6.6644

## Validation Metrics

- Loss: 0.037
- Accuracy: 0.987
- Macro F1: 0.986
- Micro F1: 0.987
- Weighted F1: 0.987

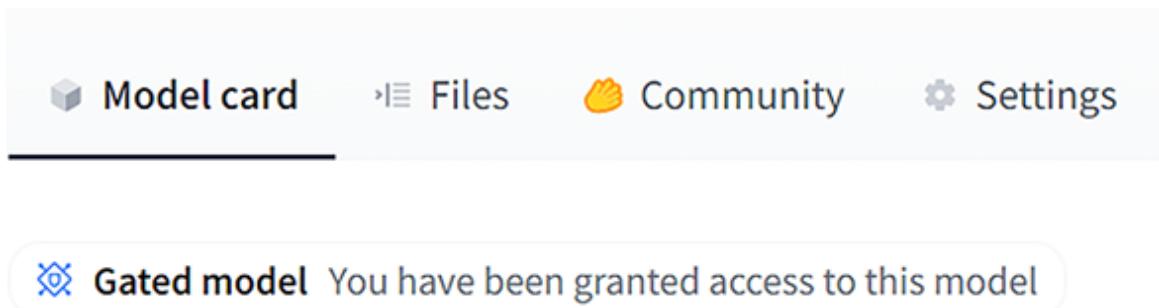
Figure 18.10: Excerpt of a model card and the metrics of a model

The process looks awesome, but we will need to deploy and run validation images to make sure the model can generalize its knowledge.

## Deploying a model

Make sure to read the latest continuously evolving documentation and explore the latest Hugging Face interfaces. *The interface may change, but you will have to create a model card for your model.*

We can go to our profile and choose the model card of one of our trained models to access the settings of the model and decide how to deploy it, as shown in *Figure 18.11*:



## Model Trained Using AutoTrain

- Problem type: Multi-class Classification
- Model ID: 81128141662
- CO2 Emissions (in grams): 6.6644

*Figure 18.11: Model card of a trained model*

We can choose a deployment method: deploy on a platform such as AWS, for example, to create an endpoint or run the model directly in transformers. In this chapter, we will run the models by using transformers directly.

We first need to make each model public by going to the model card, then **Settings**, making the model public, as shown in *Figure 18.12*:

## ⓘ Change model visibility

This model is currently **private**. Only you (personal model) or members of your organization (organization model) can see and commit to this model.

**Make public**

*Figure 18.12: Making a Hugging Face model public*

Clicking on **Make public** will make it public (*Figure 18.13*):

## Change model visibility

This model is currently **public**.

*Figure 18.13: Model visibility status*

We will now use the hosted inference services offered by Hugging Face as the fastest way to start evaluating the models.

# Running our models for inference

The trained models can now perform image classification with validation images. In this section, we will run several trained models.

Open the `Hugging_Face_AutoTrain.ipynb` that we will use in this section to:

- Retrieve a relatively easy image and a challenging one.
- Classify the validation images.
- Analyze the difficulty of image classification.
- Investigate the configuration of the trained models.

We will begin by retrieving the validation images.

## Retrieving validation images

The notebook first imports IPython for media rendering:

```
from IPython.display import Image      #This is used for render.
```

The first image is relatively easy to classify:

`generate_an_image_of_a_car_in_space.jpg`. This image, which we will now download, was classified in *Chapter 16, Beyond Text: Vision Transformers in the Dawn of Revolutionary AI*, by a vision transformer:

```
#Development access to delete when going into production
!curl -L https://raw.githubusercontent.com/Denis2054/Transform...
```

The program now displays the image:

```
from PIL import Image
# Define the path of your image
image_path = "/content/generate_an_image_of_a_car_in_space.jpg"
# Open the image
image = Image.open(image_path)
image
```

The output confirms that the image is relatively easy to classify:



Figure 18.14: Car in space produced by a Stable Diffusion model. Copyright 2023, Denis Rothman

We now download a challenging image to classify:

```
#Development access to delete when going into production
!curl -L https://raw.githubusercontent.com/Denis2054/Transformers/main/car_in_fog.png
```

The image contains a car on a foggy night, as displayed:

```
image_path = "car_in_fog.png"
image = Image.open(image_path)
image
```

A human can detect vehicles in `car_in_fog.png`. A machine finds the inference difficult and often fails to classify the image.

A non-AI professional will need an AI expert to find solutions:



*Figure 18.15: Car in the fog produced by a prompt submitted to Midjourney. Copyright 2023, Denis Rothman*

In real life, images are not always easy to classify! Human drivers must face darkness, fog, mist, and other unfavorable conditions. Self-driving cars also face difficulties in adverse situations—for example, radars malfunction in rain, fog, and snow.

Humans can relatively easily detect that there are two vehicles on the road, one of which is in the center lane.

**The program will now attempt to classify the validation images. We will see how a vision transformer reacts to this image.**

## Inference: image classification

The program first installs and imports the `transformers` library:

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

Two types of Hugging Face tokens will be required for a private model until you decide to make it public. Make sure to store your Hugging Face API token in a secure file that is not visible to others:

```
f = open("drive/MyDrive/files/HF_TOKEN.txt", "r")
HF_TOKEN=f.readline()
f.close()
#remove newline from the token
HF_TOKEN_Bearer=HF_TOKEN.strip()
# Remove 'Bearer ' from the token
token = HF_TOKEN.replace('Bearer ', '').strip()
```

`HF_TOKEN` uses the prefix `Bearer` and then the Hugging Face API token you must create to run the models if your repository is private and for some Hugging Face functions.

`token` doesn't use the prefix `Bearer` and only contains the Hugging Face API token you need to create.

One way to run your model is by going through an API URL using `requests`:

```
import requests
BASE_URL = "https://api-inference.huggingface.co/models/Denis1"
headers = {"Authorization": HF_TOKEN_Bearer} #requires Bearer
```

In this example, we will run one of the trained models: `autotrain-training-cifar-10-81128141658`.

The program defines the request function:

```
def query(filename, model_name, base_url=BASE_URL, headers=headers):
    api_url = f"{base_url}/{model_name}"
    with open(filename, "rb") as f:
        data = f.read()
    response = requests.post(API_URL, headers=headers, data=data)
    return response.json()
```

There is sometimes a delay when the model is first called. Run the cell. If the model is not loaded yet, try after a few seconds or a minute or two.

The program also defines an output processing function that will process the output, display the scores, and display a message if one of the car (“automobile”) images is not classified:

```
def classify_image(output):
    # Create a list of scores and labels
    scores = []
    labels = []
    for item in output:
        scores.append(item['score'])
        labels.append(item['label'])
    # Sort scores
    scores, labels = zip(*sorted(zip(scores, labels), reverse=True))
    # Print the scores and labels one line under the other
    for score, label in zip(scores, labels):
        print(f"score:{round(score, 4)} {label}")
    # Check if the top score is not automobile
    top_label = labels[0]
    if top_label != "automobile":
        print("I'm sorry, this image cannot be classified")
```

We can now ask the model to make an inference. You may have to wait for the model to load and rerun the cell.

```
model_name = "autotrain-training-cifar-10-81128141657"
output = query("generate_an_image_of_a_car_in_space.jpg", mode)
classify_image(output)
```

The output is accurate:

```
score:0.9983 automobile
score:0.0009 airplane
score:0.0006 ship
score:0.0001 truck
```

Now comes the challenging image:

```
model_name = "autotrain-training-cifar-10-81128141657"
output = query("car_in_fog.png", model_name)
classify_image(output)
```

The model fails to identify a vehicle, although a human can:

```
score:0.6011 airplane
score:0.1972 ship
score:0.1781 truck
score:0.0237 automobile
I'm sorry, this image cannot be classified
```

A human will certainly not interpret the vehicle as an airplane!

We need to investigate the trained models further to assess the outputs.

# Validation experimentation on the trained models

In this section, we will go through the configuration of each model trained. If the model fails to classify images, we must at least know which model we are dealing with. We might also see that the model's architecture is insufficient for complex image classification (layers, parameters).

Some models that came from different sources were trained twice. We will only analyze each model's name and configuration once.

The attention layers of a transformer model are industrialized. The layers of a vision transformer, apart from the attention layers, are *not* industrialized. The designers of these models built them empirically through trial and error. They added components as they saw fit to attain a result. As such, you will notice that each architecture has a different approach.

You can review the main components of vision transformers in *Chapter 16, Beyond Text: Vision Transformers in the Dawn of Revolutionary AI*, and *Chapter 17, Transcending the Image-Text Boundary with Stable Diffusion*.

In this section, we will go through:

- ViT for image classification
- Swin for image classification
- BEiT for image classification
- ConvNext for image classification
- ResNet for image classification

Each subsection contains:

- A description of the model provided by Google AI and modified for educational purposes

- The configuration of the model produced by a function in Python
- A validation classification of a challenging image of a car

At one point, the subsections will show the limits of the models and how to provide transparent information to end-users.

The goal is to identify the name of the model and its general architecture to:

- Acquire a minimum knowledge of the models to select them during training.
- Retain or abandon models that do not provide satisfactory results.
- Advise non-AI professionals when they use AutoTrain.

We will begin with ViT.

## **ViTForImageClassification**

We went through the vision transformer's architecture in *Chapter 16, Beyond Text: Vision Transformers in the Dawn of Revolutionary AI*.

However, let's review the model's architecture in the chapter notebook. The ViT in the notebook contains:

- An image encoder that consists of a stack of transformer blocks. Each Transformer block consists of a self-attention layer, a convolutional layer, and a residual connection.
- A classification head, which outputs the class probabilities for the input image.

The image encoder in the ViT model uses a patch embedding layer to convert the input image into a sequence of patches. The stack of transformer blocks then processes each patch.

The classification head in the ViT model takes the output of the image encoder as input and outputs a vector of class probabilities. The number of classes in the vector depends on the number of classes in the dataset that the model is trained on.

The specific configuration of the model can be displayed with the following code:

```
model_name="Denis1976/autotrain-training-cifar-10-81128141658"
model = transformers.AutoModelForImageClassification.from_pret
print(model.config)
```

Take a few minutes to explore the architecture:

```
ViTConfig {
    "_name_or_path": "Denis1976/autotrain-training-cifar-10-81128
    "architectures": [
        "ViTForImageClassification"
    ],
    "attention_probs_dropout_prob": 0.0,
    "encoder_stride": 16,
    "hidden_act": "gelu",
    "hidden_dropout_prob": 0.0,
    "hidden_size": 768,
    "id2label": {
        "0": "airplane",
        "1": "automobile",
        "2": "ship",
        "3": "truck"
    },
    "image_size": 224,
    "initializer_range": 0.02,
    "intermediate_size": 3072,
    "label2id": {
        "airplane": "0",
        "automobile": "1",
```

```
        "ship": "2",
        "truck": "3"
    },
    "layer_norm_eps": 1e-12,
    "max_length": 128,
    "model_type": "vit",
    "num_attention_heads": 12,
    "num_channels": 3,
    "num_hidden_layers": 12,
    "padding": "max_length",
    "patch_size": 16,
    "problem_type": "single_label_classification",
    "qkv_bias": true,
    "torch_dtype": "float32",
    "transformers_version": "4.31.0"
}
```

When we examine the architecture, several questions come up:

- Why has the model been frozen at 12 attention heads, and why aren't there more?
- Why are there only 12 layers? Has the ViT model evolved since the source was made available?
- Are there more complex architectures available?

The program tries to classify the car in the fog image:

```
model_name = "autotrain-training-cifar-10-81128141658"
output = query("car_in_fog.png", model_name)
classify_image(output)
```

The model fails to identify a car:

```
score:0.571 airplane
score:0.294 ship
```

```
score:0.0856 truck
score:0.0494 automobile
I'm sorry, this image cannot be classified
```

If the predictions are satisfactory, we can move on. If not, we must search for other models and analyze them before spending our budget on blind automated training. AI experts have a long career ahead!

Now we will ask the function defined in the *Vision transformers in code* section of *Chapter 16, Beyond Text: Vision Transformers in the Dawn of Revolutionary AI*, to try to identify the car in the fog:

## ViT-base-patch16-224

We will import transformers and run the request made in *Chapter 16* for the car in the fog image:

```
image_path="/content/car_in_fog.png"
import PIL
image = PIL.Image.open(image_path)
from transformers import ViTFeatureExtractor, ViTForImageClassification
from PIL import Image
import requests
feature_extractor = ViTFeatureExtractor.from_pretrained('google/vit-base-patch16-224')
model = ViTForImageClassification.from_pretrained('google/vit-base-patch16-224')
inputs = feature_extractor(images=image, return_tensors="pt")
outputs = model(**inputs)
logits = outputs.logits
```

The program displays the labels of the top logits:

```
# model predicts one of the 1000 ImageNet classes
predicted_class_idx = logits.argmax(-1).item()
print("Predicted class:", predicted_class_idx, ":", model.config.id2label[predicted_class_idx])
```

The predicted class is `spotlight`:

```
Predicted class: 818 : spotlight, spot
```

A spotlight is undoubtedly not a headlight! If we were in a real-life situation with a self-driving car, this prediction would be dangerous for the driver.

When AI fails, a non-AI professional must ask an AI professional to intervene, explain, and improve the pipeline: dataset, hyperparameters, and configuration.

An AI professional might have to analyze the models to understand them before selecting other models, modifying the datasets, and testing different hyperparameters.

The next model in the notebook is Swin.

## SwinForImageClassification 1

Swin was designed as a general-purpose model for computer vision.

The Swin transformer is a hierarchical transformer architecture that uses a combination of self-attention and convolutions to learn representations of images. This means that the blocks at the beginning of the model extract low-level features from the input image, while the blocks at the end of the model extract high-level features.

The `SwinForImageClassification` model consists of the following layers:

- A convolutional stem layer that extracts features from the input image.

- A series of Swin transformer blocks. Each Swin transformer block consists of a self-attention layer, a convolutional layer, and a residual connection.
- A classification head that outputs the class probabilities for the input image.

We can visualize the configuration of Swin:

```
model_name="Denis1976/autotrain-training-cifar-10-81128141660"
model = transformers.AutoModelForImageClassification.from_pret
print(model.config)
```

The output displays the configuration:

```
SwinConfig {
    "_name_or_path": "Denis1976/autotrain-training-cifar-10-81128
    "architectures": [
        "SwinForImageClassification"
    ],
    "attention_probs_dropout_prob": 0.0,
    "depths": [
        2,
        2,
        18,
        2
    ],
    "drop_path_rate": 0.1,
    "embed_dim": 128,
    "encoder_stride": 32,
    "hidden_act": "gelu",
    "hidden_dropout_prob": 0.0,
    "hidden_size": 1024,
    "id2label": {
        "0": "airplane",
        "1": "automobile",
        "2": "ship",
    }
}
```

```
    "3": "truck"
},
"image_size": 224,
"initializer_range": 0.02,
"label2id": {
    "airplane": "0",
    "automobile": "1",
    "ship": "2",
    "truck": "3"
},
"layer_norm_eps": 1e-05,
"max_length": 128,
"mlp_ratio": 4.0,
"model_type": "swin",
"num_channels": 3,
"num_heads": [
    4,
    8,
    16,
    32
],
"num_layers": 4,
"out_features": [
    "stage4"
],
"out_indices": [
    4
],
"padding": "max_length",
"patch_size": 4,
"path_norm": true,
"problem_type": "single_label_classification",
"qkv_bias": true,
"stage_names": [
    "stem",
    "stage1",
    "stage2",
    "stage3",
    "stage4"
],
"torch_dtype": "float32",
"transformers_version": "4.31.0",
"use_absolute_embeddings": false,
```

```
        "window_size": 7
    }
```

Will this model succeed if we run the car in the fog image? Let's try:

```
model_name = "autotrain-training-cifar-10-81128141660"
output = query("car_in_fog.png", model_name)
classify_image(output)
```

This model fails as well:

```
score:0.5013 ship
score:0.418 airplane
score:0.0482 truck
score:0.0324 automobile
I'm sorry, this image cannot be classified
```

Can the BEiT model do better?

## BeitForImage Classification

A BEiT model is derived from a BERT model: **BERT Pre-Training of Image Transformers (BEiT)**. BEiT contains:

- An image encoder that consists of a stack of 12 transformer blocks. Each transformer block consists of a self-attention layer, a convolutional layer, and a residual connection.
- A classification head that outputs the class probabilities for the input image.
- An auxiliary head that predicts the semantic segmentation of the input image.

The image encoder in the BEiT model uses a patch embedding layer to convert the input image into a sequence of patches. The stack of transformer blocks then processes each patch. The transformer blocks learn to attend to different patches in the image, and they also learn to learn long-range dependencies between the patches.

The classification head in the BEiT model takes the output of the image encoder as an input and outputs a vector of class probabilities. The number of classes in the vector depends on the number of classes in the dataset that the model is trained on.

The auxiliary head in the BEiT model takes the output of the image encoder as input and predicts the semantic segmentation of the input image. The semantic segmentation is a pixel-level classification of the image, which means that each pixel in the image is assigned a class label.

The specific configuration of the BEiT model can be viewed with the following code:

```
model_name="Denis1976/autotrain-training-cifar-10-81128141661"
model = transformers.AutoModelForImageClassification.from_pret
print(model.config)
```

You can see that the configuration remains a variation of other classification models. The output is processed to produce a label:

```
BeitConfig {
    "_name_or_path": "Denis1976/autotrain-training-cifar-10-81128
    "architectures": [
        "BeitForImageClassification"
    ],
    "attention_probs_dropout_prob": 0.0,
```

```
"auxiliary_channels": 256,
"auxiliary_concat_input": false,
"auxiliary_loss_weight": 0.4,
"auxiliary_num_convs": 1,
"drop_path_rate": 0.1,
"hidden_act": "gelu",
"hidden_dropout_prob": 0.0,
"hidden_size": 768,
"id2label": {
    "0": "airplane",
    "1": "automobile",
    "2": "ship",
    "3": "truck"
},
"image_size": 224,
"initializer_range": 0.02,
"intermediate_size": 3072,
"label2id": {
    "airplane": "0",
    "automobile": "1",
    "ship": "2",
    "truck": "3"
},
"layer_norm_eps": 1e-12,
"layer_scale_init_value": 0.1,
"max_length": 128,
"model_type": "beit",
"num_attention_heads": 12,
"num_channels": 3,
"num_hidden_layers": 12,
"out_indices": [
    3,
    5,
    7,
    11
],
"padding": "max_length",
"patch_size": 16,
"pool_scales": [
    1,
    2,
    3,
    6
```

```
],
  "problem_type": "single_label_classification",
  "semantic_loss_ignore_index": 255,
  "torch_dtype": "float32",
  "transformers_version": "4.31.0",
  "use_absolute_position_embeddings": false,
  "use_auxiliary_head": true,
  "use_mask_token": false,
  "use_mean_pooling": true,
  "use_relative_position_bias": true,
  "use_shared_relative_position_bias": false,
  "vocab_size": 8192
}
```

Can BERT identify the car in the fog? Let's try:

```
model_name = "autotrain-training-cifar-10-81128141661"
output = query("car_in_fog.png", model_name)
classify_image(output)
```

The output shows that this trained model failed as well:

```
score:0.8033 airplane
score:0.1801 ship
score:0.0121 truck
score:0.0045 automobile
I'm sorry, this image cannot be classified
```

Something needs to be done!

By the time we reach another mode trained on the architecture of Swin, we need to pause and think.

## SwinForImageClassification 2

Up to now, all the trained models failed to identify a car in the fog, although this situation occurs often for an autonomous vehicle.

We briefly reviewed Swins's configuration in the *SwinForImageClassification 1* section of this chapter.

In this section, we will focus on the limits of this AutoTrain experiment. Some fundamental thoughts come up:

- Will training a CIFAR-10 dataset lead to high accuracy but poor generalization?
- Are the trained models victims of overfitting?
- Are the model's architectures sufficient?
- We are not training these models from scratch. Were they sufficiently pretrained in the first place?
- Should a better dataset be provided to identify vehicles in dark and foggy environments? Would the models have been classified better?

There are no mathematical or scientific answers to these questions. Training transformers relies on empirical trial and error to improve. Answering these questions will depend on the project, the goal, and the resources available to resolve the issues.

We will try this version of Swin anyway:

```
model_name="Denis1976/autotrain-training-cifar-10-81128141657"  
model = transformers.AutoModelForImageClassification.from_pret
```

The car in the fog image is submitted:

```
model_name = "autotrain-training-cifar-10-81128141657"
output = query("car_in_fog.png", model_name)
classify_image(output)
```

The output shows that the model failed to identify the car:

```
score:0.6011 airplane
score:0.1972 ship
score:0.1781 truck
score:0.0237 automobile
I'm sorry, this image cannot be classified
```

The message now appears as a disclaimer, like on most interactive AI sites.

We can download an image with an obvious car in the night and a fog image:

```
#Development access to delete when going into production
!curl -L https://raw.githubusercontent.com/Denis2054/Transformers/master/transformers/pipelines/text_to_image.ipynb
```

Can we humans distinguish the car better when it is displayed?

```
from PIL import Image
# Define the path of your image
image_path = "/content/car_in_night.jpg"
# Open the image
image = Image.open(image_path)
image
```

The output clearly shows a car in the night and some fog:



Figure 18.16: An image of a car on a foggy night

Can our trained model identify a car this time?

```
model_name = "autotrain-training-cifar-10-81128141657"  
classify_image(output)
```

No, the model failed again on an obvious image of a car!

The output is:

```
score:0.8055 automobile  
score:0.1242 airplane  
score:0.0411 ship  
score:0.0291 truck
```

In the following section, we need to design a way to find the limit of the model.

## ConvNextForImageClassification

*Liu et al. (2022)* created ConvNext to demonstrate that a pure convolutional network could perform well, if not better, than hybrid transformer models. It consists of:

- A stem layer that extracts features from the input image.
- A series of four stages. Each stage consists of a stack of transformer blocks. Each transformer block consists of a self-attention layer, a convolutional layer, and a residual connection.
- A classification head, which outputs the class probabilities for the input image.

The ConvNext model's classification head takes the stages' output as input and outputs a vector of class probabilities. The number of classes in the vector depends on the number of classes in the dataset that the model is trained on.

We can visualize the configuration with the following code:

```
model_name="Denis1976/autotrain-training-cifar-10-81128141663"
model = transformers.AutoModelForImageClassification.from_pret
print(model.config)
```

The output shows a convolutional network with classification functionality:

```
ConvNextConfig {
    "_name_or_path": "Denis1976/autotrain-training-cifar-10-81128
    "architectures": [
        "ConvNextForImageClassification"
    ],
    "depths": [
        3,
        3,
        9,
```

```
    3
  ],
  "drop_path_rate": 0.0,
  "hidden_act": "gelu",
  "hidden_sizes": [
    96,
    192,
    384,
    768
  ],
  "id2label": {
    "0": "airplane",
    "1": "automobile",
    "2": "ship",
    "3": "truck"
  },
  "image_size": 224,
  "initializer_range": 0.02,
  "label2id": {
    "airplane": "0",
    "automobile": "1",
    "ship": "2",
    "truck": "3"
  },
  "layer_norm_eps": 1e-12,
  "layer_scale_init_value": 1e-06,
  "max_length": 128,
  "model_type": "convnext",
  "num_channels": 3,
  "num_stages": 4,
  "out_features": [
    "stage4"
  ],
  "out_indices": [
    4
  ],
  "padding": "max_length",
  "patch_size": 4,
  "problem_type": "single_label_classification",
  "stage_names": [
    "stem",
    "stage1",
    "stage2",
    "stage3"
  ]
}
```

```
        "stage3",
        "stage4"
    ],
    "torch_dtype": "float32",
    "transformers_version": "4.31.0"
}
```



Can this convolutional network model find a car? This time, we will go step by step to find the limit of the model.

First, we try the car in the fog image:

```
model_name = "autotrain-training-cifar-10-81128141663"
output = query("car_in_fog.png", model_name)
classify_image(output)
```

The model fails:

```
score:0.5243 ship
score:0.3771 airplane
score:0.0838 truck
score:0.0149 automobile
I'm sorry, this image cannot be classified
```

Now we try the less challenging car in the night image:

```
model_name = "autotrain-training-cifar-10-81128141663"
output = query("car_in_night.jpg", model_name)
classify_image(output)
```

The model fails at this less difficult level as well:

```
score:0.4904 airplane
score:0.2331 automobile
score:0.2075 ship
score:0.069 truck
I'm sorry, this image cannot be classified
```

Finally, we try the easiest image with the white car on a black background:

```
model_name = "autotrain-training-cifar-10-81128141663"
output = query("/content/generate_an_image_of_a_car_in_space.j
classify_image(output)
```

The model identifies the car:

```
score:0.8055 automobile
score:0.1242 airplane
score:0.0411 ship
score:0.0291 truck
```

We now have a tiny corpus of level 3 (very difficult), 2 (difficult), and 1 (easy) images to determine the level of perception of a trained model.

## ResNetForImageClassification

**ResNet (Residual Network)** is a neural network like ConvNext. ResNet can contain up to 1,000 layers, unlike the trained model in this chapter.

The stages in the ResNet model consist of a stack of convolutional blocks. Each convolutional block in a stage has the same number of filters. The number of filters in each stage increases as the model goes deeper.

We can visualize the configuration of the trained ResNet model:

```
model_name="Denis1976/autotrain-training-cifar-10-81128141659"
model = transformers.AutoModelForImageClassification.from_pret
print(model.config)
```

We can see the specific configuration available to us:

```
ResNetConfig {
    "_name_or_path": "Denis1976/autotrain-training-cifar-10-81128
"architectures": [
    "ResNetForImageClassification"
],
"depths": [
    3,
    4,
    6,
    3
],
"downsample_in_first_stage": false,
"embedding_size": 64,
"hidden_act": "relu",
"hidden_sizes": [
    256,
    512,
    1024,
    2048
],
"id2label": {
    "0": "airplane",
    "1": "automobile",
    "2": "ship",
    "3": "truck"
},
"label2id": {
    "airplane": "0",
    "automobile": "1",
    "ship": "2",
    "truck": "3"
}}
```

```
        "truck": "3"
    },
    "layer_type": "bottleneck",
    "max_length": 128,
    "model_type": "resnet",
    "num_channels": 3,
    "out_features": [
        "stage4"
    ],
    "out_indices": [
        4
    ],
    "padding": "max_length",
    "problem_type": "single_label_classification",
    "stage_names": [
        "stem",
        "stage1",
        "stage2",
        "stage3",
        "stage4"
    ],
    "torch_dtype": "float32",
    "transformers_version": "4.31.0"
}
```

We can now run our tiny corpus to find the limit of this relatively small ResNet model. We begin with a level 3, very difficult, image:

```
model_name = "autotrain-training-cifar-10-81128141659"
output = query("car_in_fog.png", model_name) # car in fog
classify_image(output)
```

The model fails:

```
score:0.5376 ship
score:0.4026 airplane
score:0.0336 truck
```

```
score:0.0262 automobile
I'm sorry, this image cannot be classified
```

Now we run the level 2, difficult, image:

```
model_name = "autotrain-training-cifar-10-81128141659"
output = query("car_in_night.jpg", model_name)
classify_image(output)
```

The model fails:

```
score:0.5016 airplane
score:0.4344 ship
score:0.0358 automobile
score:0.0281 truck
I'm sorry, this image cannot be classified
```

Finally, we run the level 1, easy, image:

```
model_name = "autotrain-training-cifar-10-81128141659"
output = query("/content/generate_an_image_of_a_car_in_space.j
classify_image(output)
```

The model succeeds:

```
score:1.0 automobile
score:0.0 truck
score:0.0 airplane
score:0.0 ship
```

Through trial and error, we found a little corpus to test an image classification model on images that go beyond well-prepared datasets.

Let's apply our discovery to the top-ranking model of our training session.

## Trying the top ViT model with a corpus

We will now run our small corpus on the top-ranking trained model:

```
model_name="Denis1976/autotrain-training-cifar-10-81128141658"  
model = transformers.AutoModelForImageClassification.from_pret
```

We begin with the very difficult level 3 image:

```
model_name = "autotrain-training-cifar-10-81128141659"  
output = query("car_in_fog.png", model_name) # car in fog  
classify_image(output)
```

The model fails at this level:

```
score:0.5376 ship  
score:0.4026 airplane  
score:0.0336 truck  
score:0.0262 automobile  
I'm sorry, this image cannot be classified
```

Now, we can try the level 2 image:

```
model_name = "autotrain-training-cifar-10-81128141658"  
output = query("car_in_night.jpg", model_name)  
classify_image(output)
```

The inference was successful!

```
score:0.5064 automobile
score:0.2387 ship
score:0.1574 airplane
score:0.0975 truck
```

Finally, we try the easy image again:

```
model_name = "autotrain-training-cifar-10-81128141659"
output = query("/content/generate_an_image_of_a_car_in_space.j
classify_image(output)
```

The output is successful:

```
score:1.0 automobile
score:0.0 truck
score:0.0 airplane
score:0.0 ship
```

We now have a small standard to measure an image classification model. If we decide to move into production, we will need:

- To train the model with a more extensive and diverse dataset.
- To add level 2 (difficult) and level 3 (very difficult) images in separate directories to push the model to its limit.
- To warn the end-users of the limits we cannot control. In this case, we knew what to expect. But in an open environment, we will not have that luxury with random images of all types.

A professional project manager cannot draw too many conclusions from the experiments in this notebook. Hugging Face models may produce better results with a larger or optimized dataset. Also, we can try other models. The choice of platform, dataset, and model depends on the goals of each project.

In any case, we can see that human AI professionals will be required to provide expertise and assistance for quite a while!

## Summary

This chapter brought us to the frontier of AI, where automation rages. Competitors around the world are struggling to make AI accessible to mainstream users. OpenAI's ChatGPT opened the door to a flood of automated tasks.

Hugging Face has successfully deployed numerous automated functions on their platform, made transformers easy to run, and provided many other productive functions. Hugging Face AutoTrain provides a no-coding service to train and deploy AI models.

We began by creating an image classification project with a Hugging Face Space. We uploaded a CIFAR-10 dataset with transportation images.

We then continued by running the models with a difficult and an easy image of cars. We first built an API function to query the models and an output processing function to display the scores produced.

The models explored were ViT, Swin, BEiT, ConvNext, and ResNet. The chapter notebook displayed each model's configuration, accompanied by a brief model description. Although the accuracies of the models were excellent, the models failed to detect an image of a car in the night in the fog. A critical aspect of implementing computer vision models resides in

accepting the limits of even the most influential models. When the limit is reached, we must first find a way to build a scientific approach to define what a model can and cannot do.

Thus, finally, we built a mini-validation set with an easy, difficult, and very difficult image of a car to classify. Our best model, [Denis1976/autotrain-training-cifar-10-81128141658](#), successfully detected a car in the easy and difficult images. But it failed to detect the very difficult car in the fog image. In a real-life transportation image inference project, we must build comprehensive datasets with many representative images of each category (easy, difficult, and very difficult) beyond the classical sets (cars, airplanes, ships, and trucks).

In some cases, a non-AI professional might successfully use Hugging Face AutoTrain without encountering an obstacle. However, we cannot sum up an AI project as a wonderful ride through automated services with perfect datasets, the right models, and excellent performances. Making anybody believe that would be unethical.

Like any AI project, a successful computer vision project requires problem-solving skills, imagination, and patience!

Can other models or methods do better than the trained models in this chapter? We will continue our experimentation in *Chapter 19, On the Road to Functional AGI with HuggingGPT and Its Peers*.

## Questions

1. Hugging Face AutoTrain can train every vision transformer on the market. (True/False)
2. Datasets are always easy to create. (True/False)

3. A no-coding AI system requires no machine learning knowledge.  
(True/False)
4. Hugging Face AutoTrain can classify any image submitted.  
(True/False)
5. Even a well-prepared dataset can be insufficient. (True/False)
6. Creating a validation set of images to test a vision transformer is useful.  
(True/False)
7. Even a well-trained model can lead to overfitting. (True/False)
8. It may take months to optimize a vision transformer. (True/False)
9. An automated service can sometimes work well with no coding.  
(True/False)
10. An AI professional will always be necessary for complex AI issues.  
(True/False)

## References

- Hugging Face AutoTrain:  
<https://huggingface.co/autotrain>
- ViT on Hugging Face:  
[https://huggingface.co/docs/transformers/v4.27.1/model\\_doc/vit](https://huggingface.co/docs/transformers/v4.27.1/model_doc/vit)
- Swin on Hugging Face:  
[https://huggingface.co/docs/transformers/main/model\\_doc/swin](https://huggingface.co/docs/transformers/main/model_doc/swin)
- BEiT on Hugging Face:  
[https://huggingface.co/docs/transformers/main/model\\_doc/beit](https://huggingface.co/docs/transformers/main/model_doc/beit)

- ConvNext:  
[https://huggingface.co/docs/transformers/main/model\\_doc/convnext](https://huggingface.co/docs/transformers/main/model_doc/convnext)
- Hugging Face ResNet:  
[https://huggingface.co/docs/transformers/model\\_doc/resnet](https://huggingface.co/docs/transformers/model_doc/resnet)

## Further reading

- Hugging Face AutoTrain:  
<https://huggingface.co/docs/autotrain/index>
- Hugging Face AutoTrain documentation:  
<https://huggingface.co/docs/autotrain/index>

## Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://www.packt.link/Transformers>



# 19

## On the Road to Functional AGI with HuggingGPT and its Peers

**Functional Artificial General Intelligence (F-AGI)** is the automation of a sufficient scope of tasks in a closed environment that can outperform a human in a workplace. We use the term F-AGI to describe an AI system's capabilities and the risks related to job losses. Beyond the risks, another factor has become critical: volume. Human activity requires millions of micro-decisions to produce services, products, and communication resources. F-AGI has begun to emerge out of necessity.

The term F-AGI avoids the confusion with the AGI myth of a sentient, conscious, humanoid AI agent. F-AGI restricts its scope to a closed domain in a workplace. An F-AGI can work 24/7. It can solve millions of tasks per day. It will prove invaluable in critical situations such as wildfire prevention, flight planning to reduce carbon emissions, and a limitless variety of productive activities.

Microsoft, Google, OpenAI, Hugging Face, and others have expanded the scope of Foundation Models in the journey toward AGI. One Foundation Model, such as ChatGPT, can perform hundreds of tasks, some still to be discovered by imaginative users.

However, why put the burden on one model? Why focus on one super “intelligent” AI agent like in a science fiction movie? Why ask one model to

do everything that requires tremendous data and hardware resources? Why not build a centralized controller that activates the best transformer model for a specific task?

*Shen et al. (2023)* designed precisely such a system and named it HuggingGPT. Hugging Face hosts innumerable models with well-defined model cards. ChatGPT has the uncanny ability to understand language, read the model cards, among other parameters, and choose the best models for a specific task. The authors of HuggingGPT invented an efficient path toward AGI by combining HuggingFace ChatGPT with their hosted models.

The chapter begins by defining clear-cut F-AGI activities with some examples in computer vision. This chapter aims to understand the unique productivity of working with several AI models to solve complex problems.

We will then download the validation dataset we ran in *Chapter 18, Hugging Face AutoTrain: Training Vision Models without Coding*. In *Chapter 18*, the models failed to identify a car in the very difficult image of a vehicle in the fog. Yet, driving assistants must absolutely learn how to push the limits of computer vision and solve this problem. This chapter will open innovative ways to reach that goal successfully.

HuggingGPT will first find a model or models that can solve the problem of identifying easy, difficult, and very difficult images. It might also stumble on the very difficult image of a car in the fog. However, HuggingGPT's approach will inspire the rest of the chapter.

We will then put Google Cloud Vision to work to identify the easy, difficult, and very difficult images. The difficult image will remain challenging, but we will solve the issue in this chapter.

We will continue by taking a step out of classical approaches. Each competing platform has its scope. What if we took a system further, ignored

competition, and forced competing models to work together? We will thus go beyond classical pipelines and explore how to chain heterogeneous competing models. We will show that an AI computer vision's output, such as Google Cloud Vision, can become the input of HuggingGPT or even OpenAI's ChatGPT.

Finally, our car in the fog challenge will be solved thanks to a cross-platform, chained-model pipeline! Of course, the conclusion will not be that all very difficult images can be identified. But by the end of this chapter, you will have learned how to extend the limits of a model when facing complex issues.

This chapter covers the following topics:

- Defining F-AGI
- Describing the use of clear-cut cases of F-AGI
- Testing a validation set for vision models for an autonomous vehicle
- Retrieving validation images
- Exploring the potential of HuggingGPT for difficult images
- Inference with object detection
- Exploring the potential of Google Cloud Vision for difficult images
- Leveraging the semantic analysis power of ChatGPT to solve complex object detection problems
- Providing solutions with model chaining
- Illustrating model chaining with Midjourney and Gen-2

With all the innovations and library updates in this cutting-edge field, packages and models change regularly. Please go to the GitHub repository for the latest installation and code examples:



<https://github.com/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition/tree/main/Chapter19>.

You can also post a message in our Discord community (<https://discord.com/invite/Fp4XXhECdh>) if you have any trouble running the code in this or any chapter.

Our first step will be to define the scope of F-AGI.

## Defining F-AGI

**F-AGI** does not have a consciousness at all. Nor can F-AGI do everything, such as the sum of all human intellectual activity.

The scope of this section and chapter is certainly not to find ways to replace humans and create job displacement and replacement. This chapter also stays far away from harmful content.

F-AGI can truly become human-level entities when needed in many domains, such as:

- Replicating human vision fire detectors in areas lacking human presence to detect wildfires as soon as possible. California has begun to install fire detection cameras all over the state. The service provides human-level services that it would not be possible to implement 24/7. Pano, for example, offers AI-driven vision services for wildfire management (see the *References* section).
- Replicating human vision skin cancer detection when human specialists are not always available. IBM initiated a skin cancer project in Australia several years ago. Australia has the highest rate of melanoma

in the world. It is estimated that an Australian dies every six hours of melanoma. People sometimes need to wait weeks to get an appointment with a dermatologist. IBM provides services for early skin cancer detection (see the *References* section).

- Replicating human vision for assisted driving (not autonomous) in critical conditions, such as a car in the fog at night. AI-assisted driving has been added to many brands of vehicles. We are not here to make car advertisements, nor contend that we will solve the problems of autonomous cars. However, we can show how F-AGI, in a closed environment such as AI-assisted driving can save lives.

The focus of this chapter will thus be on the following:

- Solving the problem of identifying a car in a difficult image simulating a video frame. This problem remained unsolved in *Chapter 18, Hugging Face AutoTrain: Training Vision Models without Coding*. In this chapter, we will find ways to extend the boundaries of computer vision.
- Using HuggingGPT's philosophy to solve difficult image recognition. HuggingGPT uses ChatGPT to analyze an input, read Hugging Face model cards and their properties, and activate one or several models for many NLP and vision tasks.
- Taking systems beyond one platform (e.g., Google, Microsoft, OpenAI, Hugging Face, etc.) will model chaining across any platform we deem helpful to solve our problem.

In this chapter, we will leverage the functionality of innovative codeless platforms and interact with the best computer vision models we can find to solve our problem.

The ultimate goal of this chapter will be to:

- Solve the problem of identifying a car in the fog at night.
- Show how automated model selection and model chaining will lead to F-AGI by using the output of one model (Google Cloud Vision, for example) as the input of another model, even on another platform (OpenAI, for example).
- Show how Hugging Face leverages ChatGPT to become an effective controller, read Hugging Face model cards (and properties), and select one or several models to solve a task.

AutoML existed before HuggingGPT, including on Hugging Face's platform. However, introducing ChatGPT as a controller for AutoML was a bold move we must commend.

Note that the title of this chapter contains *HuggingGPT* and its *peers*. The *peers* are other platforms than Hugging Face. Hugging Face has opened the door to assembling heterogeneous models using ChatGPT as a controller. That achievement deserves great merit. We'll take the concept further, go through that door, and add any useful model to solve our image identification problem! In a real-life project, we could also leverage another LLM, such as Google Vertex PaLM 2, to select models we could run in a pipeline.

If you think about it, HuggingGPT is already a cross-platform solution because it requires an OpenAI token added to the Hugging Face token!

For example, we could take the concept further and implement a cross-platform process with the LLM of our choice, such as Microsoft Bing's chat agent, by asking it for vision models to produce outputs and LLM models to analyze the text output, as shown in the following instruction:

```
Provide the name of a transformer computer vision model that can
classify an image. Then provide another transformer LLM model that
```

could take the text output of the computer vision model, explain, and make recommendations:

Bing provides the information we need to chain models on the same platform or cross-platform models:

One example of a transformer computer vision model that can classify an image is the Vision Transformer (ViT). This model applies the transformer architecture with self-attention to sequences of image patches without using convolution layers.

A transformer Large Language Model (LLM) that could take the text output of the computer vision model, explain it and make recommendations is the VisionEncoderDecoderModel. This model can be used to initialize an image-to-text model with any pretrained Transformer-based vision model as the encoder (e.g., ViT, BEiT, DeiT, Swin) and any pretrained language model as the decoder (e.g., RoBERTa, GPT2, BERT, DistilBERT).

In a real-life project, we could implement a cross-model, cross-platform pipeline to achieve our goals.

HuggingGPT simplifies the research with its ChatGPT controller, which is quite a timesaver. However, in production, we won't change models every day. We can thus use Hugging Face's HuggingGPT and whatever other platform to find the best systems for our project and create a customized cross-platform, chained model system.

In this chapter, we will explore whatever we need to get the job done.

We will use a notebook to keep a log of our exploration. Let's start by installing and importing the libraries we need.

# Installing and importing

In this chapter, we will leverage the no-coding functionality of advanced AI platforms. Open `Computer_Vision_Analysis.ipynb`, the chapter notebook, which contains the results of interactions with Google Cloud Vision, HuggingGPT, Midjourney, and Runway Gen-2.

The notebook installs `moviepy` to display Generative AI videos:

```
!pip install moviepy -qq
```

`IPython.display` is installed to display the Generative AI images:

```
from IPython.display import Image      #This is used for render.
```

`json` is imported to parse the output of vision models and pandas to display the JSON content:

```
import json
import pandas as pd
```

We will now download the validation set for our assisted-driving research.

## Validation set

The images in the validation set were created by Stable Diffusion and Midjourney and were submitted to several vision models in *Chapter 18, Hugging Face AutoTrain: Training Vision Models without Coding*. In a real-

life project, such a dataset must contain many images of many types. This chapter limits the scope to a few images to explore methods to solve a complex problem with a unit test.

## Level 1 image: easy

The first image is the level 1, easy image:

```
!curl -L https://raw.githubusercontent.com/Denis2054/Transform
```

Let's display it to have it in mind for the rest of the chapter:

```
from PIL import Image
# Define the path of your image
image_path = "/content/generate_an_image_of_a_car_in_space.jpg"
# Open the image
image = Image.open(image_path)
image
```

The output shows a white car on a black background, which any well-trained computer vision model should detect:



Figure 19.1: A car in space

Let's now download the difficult image.

## Level 2 image: difficult

The car in the night image requires more advanced computer vision models:

```
curl -L https://raw.githubusercontent.com/Denis2054/Transformer
```

If we display it, we can see the difficulty:

```
from PIL import Image
# Define the path of your image
image_path = "/content/car_in_night.jpg"
# Open the image
image = Image.open(image_path)
image
```

The output contains confusing objects (streetlights, shiny surfaces, and dark cars):



*Figure 19.2: A car at night*

The level 3, very difficult image remains quite a challenge.

## Level 3 image: very difficult

Let's download a car in the fog, in the night:

```
!curl -L https://raw.githubusercontent.com/Denis2054/Transforme
```

Display the image to confirm the difficulties facing a vision model:

```
image_path = "car_in_fog.png"
image = Image.open(image_path)
image
```

The output shows that even a human can be confused by this image:



*Figure 19.3: A difficult-to-identify vehicle at night*

The exploration will begin with HuggingGPT.

## HuggingGPT

The validation process for HuggingGPT will run on Hugging Face's platform:

<https://huggingface.co/spaces/microsoft/HuggingGPT>.



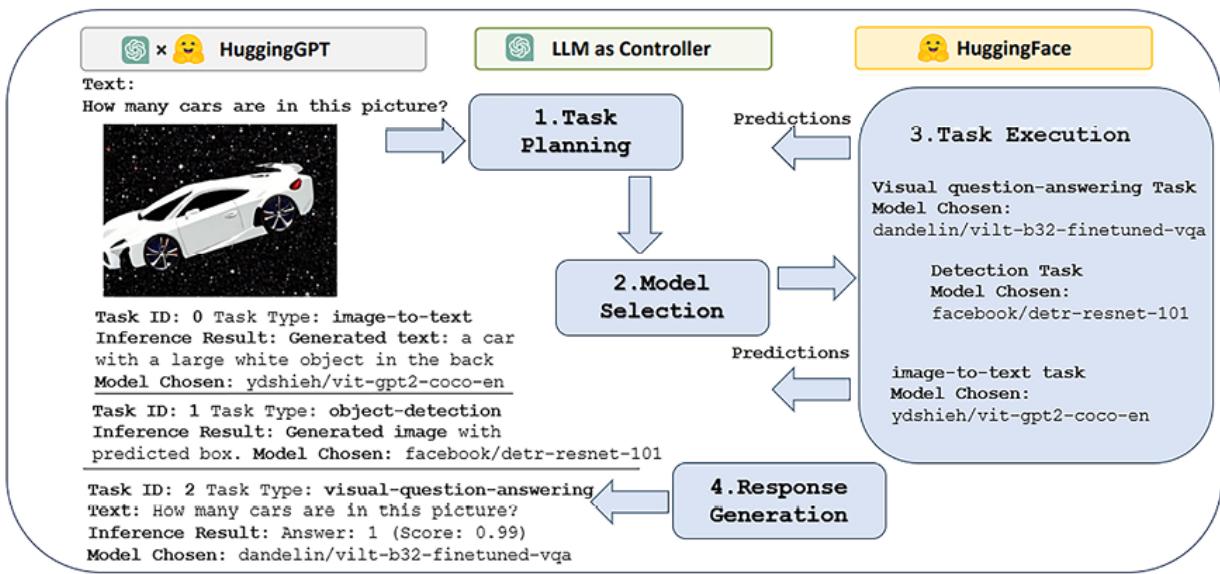
If the server is down, or there's an issue with HuggingGPT, as sometimes happens, you can try to clone the repository, run it with Docker, or contact Hugging Face support. Hugging Face forums are there to help as well.

Hugging Face, like all the cutting-edge AI platforms, is evolving at full speed. We must remain constructive.

*Shen et al. (2023)* designed a method for an LLM to connect AI models. The title and *Figure 1* of their paper sums the process up:

- *HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face*
- “*Figure 1: Language serves as an interface for LLMs (e.g., ChatGPT) to connect numerous AI models (e.g., those in Hugging Face) for solving complicated AI tasks.*”

In this section, we will ask HuggingGPT to detect a car in an image in our assisted-driving context, as shown in figure the HuggingGPT section of the notebook we are implementing, `Computer_Vision_Analysis.ipynb` and shown in *Figure 19.4*:



*Figure 19.4: HuggingGPT’s four-step AI system*

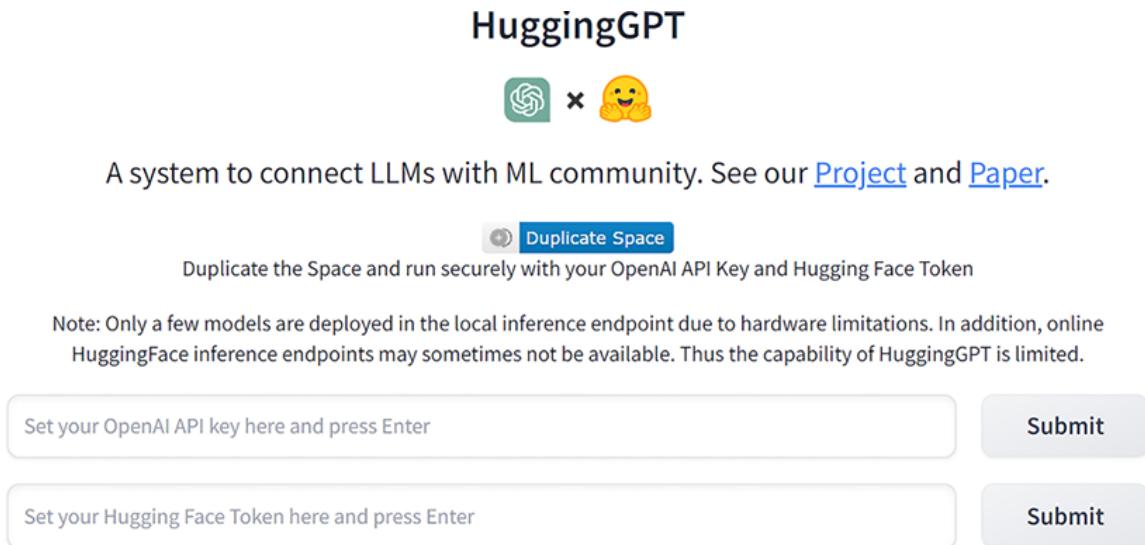
The notebook figure contains the HuggingGPT process we will run for our validation images:

1. HuggingGPT processes our prompt and the image we submit. In *Figure 19.4*, the easy image to analyze is illustrated.

2. Once HuggingGPT processes the input, it makes a model selection.
3. Then HuggingGPT executes the subtasks: visual answering, detection, and image-to-text.
4. Finally, HuggingGPT performs response generation displayed on the interface's left side. In the figure, the output is a summary of the JSON output of the system.

We will run this process for the three images of the validation dataset.

First, we must enter our OpenAI API key and HuggingFace token in this cross-platform application, as shown in *Figure 19.5*:



*Figure 19.5: Running HuggingGPT*

We can run HuggingGPT directly with our OpenAI API key and our Hugging Face token. We can also duplicate the Space and run it in a personal environment. Only a limited number of models are available for the moment, considering the work it takes to deploy such a system.

We will now put HuggingGPT to work with the easy image.

# Level 1: Easy

The prompt submitted does not restrict the request to a classification task and lets HuggingGPT do what it thinks best with the image:

```
Given image A: /generate_an_image_of_a_car_in_space.jpg, please tell  
me how many cars are in this picture.
```

The output of HuggingGPT is accurate and informative:

```
"Based on the inference results, there is only one car in the p
```

Let's take a look at the JSON output of the process stored in a file:

```
!curl L https://raw.githubusercontent.com/Denis2054/Transformer
```

We can now parse the JSON output and display the inference:

```
with open("HGPT1.json") as f:  
    data = json.load(f)  
    # Extract and Print the tasks and their details  
    for key, value in data.items():  
        print(f"Task ID: {value['task']['id']}")  
        print(f"Task Type: {value['task']['task']}")  
        print(f"Dependencies: {', '.join(map(str, value['task']['de  
        if 'args' in value['task']:  
            for arg_key, arg_value in value['task']['args'].items()  
                print(f"    {arg_key.capitalize()}: {arg_value}")  
        # Extract and Print the inference results  
        if 'inference result' in value:  
            print("Inference Result:")  
            if isinstance(value['inference result'], dict):  
                for result_key, result_value in value['inference r
```

```
        print(f"    {result_key.capitalize()}: {result_"
    else: # In case it's a list like in the visual-question
for item in value['inference result']:
        print(f"    Answer: {item['answer']} (Score: {i
    # Extract and Print the model used and the reason
if 'choose model result' in value:
        print(f"Model Chosen: {value['choose model result'][i
        print(f"Reason: {value['choose model result']['reason'
print('-' * 50) # A separator line for better readability
```

The output confirms HuggingGPT's explanation for the three tasks it ran for this inference:

```
Task ID: 0
Task Type: image-to-text
Dependencies: -1
    Image: /generate-an-image-of-a-car-in-space.jpg
Inference Result:
    Generated text: a car with a large white object in the back
    Model Chosen: ydshieh/vit-gpt2-coco-en
    Reason: Only one model available.
-----
Task ID: 1
Task Type: object-detection
Dependencies: -1
    Image: /generate-an-image-of-a-car-in-space.jpg
Inference Result:
    Generated image with predicted box: /images/a7fd.jpg
    Predicted: []
    Model Chosen: facebook/detr-resnet-101
    Reason: Only one model available.
-----
Task ID: 2
Task Type: visual-question-answering
Dependencies: 1
    Image: /generate-an-image-of-a-car-in-space.jpg
    Text: How many cars are in this picture?
Inference Result:
    Answer: 1 (Score: 0.9962437152862549)
```

```
Answer: 0 (Score: 0.017030367627739906)
Answer: 2 (Score: 0.009522431530058384)
Answer: 3 (Score: 0.001569030457176268)
Answer: 4 (Score: 0.000359362835297361)
Model Chosen: dandelin/vilt-b32-finetuned-vqa
Reason: Only one model available.
```

Will HuggingGPT do as well for the difficult image?

## Level 2: Difficult

Once again, the prompt submitted to HuggingGPT gives a lot of freedom:

```
Given an image A: /car_in_night.jpg, please tell me whether there is
a car in the image.
```

HuggingGPT confidently confirms the detection of a car and explains how it found the result:

```
Based on the inference results, I can confidently say that ther
For the object-detection model, I used the model facebook/detr-
For the visual-question-answering model, I used the model dande
Therefore, I can confidently say that there is a car in the pic
```

Let's download the JSON output stored in a file:

```
!curl -L https://raw.githubusercontent.com/Denis2054/Transforme
```

Now, let's display the content:

```

with open("HGPT2.json") as f:
    data = json.load(f)
# Extract and Print the tasks and their details
for key, value in data.items():
    print(f"Task ID: {value['task']['id']}")
    print(f"Task Type: {value['task']['task']}")
    print(f"Dependencies: {', '.join(map(str, value['task']['dependencies']))}")
    if 'args' in value['task']:
        for arg_key, arg_value in value['task']['args'].items():
            print(f"    {arg_key.capitalize()}: {arg_value}")
# Extract and Print the inference results
if 'inference result' in value:
    print("Inference Result:")
    if isinstance(value['inference result'], dict):
        for result_key, result_value in value['inference result'].items():
            print(f"    {result_key.capitalize()}: {result_value}")
    else: # In case it's a list like in the visual-question
        for item in value['inference result']:
            print(f"        Answer: {item['answer']} (Score: {item['score']})")
# Extract and Print the model used and the reason
if 'choose model result' in value:
    print(f"Model Chosen: {value['choose model result']['model']}")
    print(f"Reason: {value['choose model result']['reason']}")
print('-' * 50) # A separator line for better readability

```

The output contains valuable information, such as the generated text by [ydshieh/vit-gpt2-coco-en](#), which explicitly states that [a car is driving down a street at night](#):

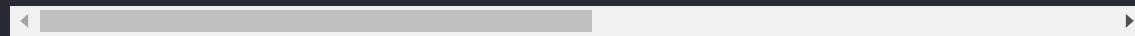
```

Task ID: 0
Task Type: image-to-text
Dependencies: -1
    Image: /car-in-night.jpg
Inference Result:
    Generated text: a car is driving down a street at night
Model Chosen: ydshieh/vit-gpt2-coco-en
Reason: Only one model available.

```

```
Task ID: 1
Task Type: object-detection
Dependencies: -1
    Image: /car-in-night.jpg
Inference Result:
    Generated image with predicted box: /images/7953.jpg
    Predicted: [{'score': 0.9988002777099609, 'label': 'car', 'bbox_2d': [350, 350, 650, 650]}]
Model Chosen: facebook/detr-resnet-101
Reason: Only one model available.

-----
Task ID: 2
Task Type: visual-question-answering
Dependencies: 1
    Image: /car-in-night.jpg
    Text: Is there a car in this picture?
Inference Result:
    Answer: yes (Score: 0.9839207530021667)
    Answer: no (Score: 0.1481371521949768)
    Answer: truck (Score: 0.0011159408604726195)
    Answer: 1 (Score: 0.0007085313554853201)
    Answer: can't tell (Score: 0.000319453829433769)
Model Chosen: danelin/vilt-b32-finetuned-vqa
Reason: Only one model available.
```



The output shows promising results. Now, we can try HuggingGPT on our very difficult image inference.

## Level 3: Very difficult

Once again, we provide an open prompt:

```
Given image A: /car_in_fog.jpg, please tell me whether there is a
car in the image.
```

HuggingGPT fails to identify a car in the image:

```
Based on the inference results, I can confidently say that ther
For the image-to-text model, I used the model ydshieh/vit-gpt2-
For the object-detection model, I used the model facebook/detr-
For the visual-question-answering model, I used the model dande
Therefore, I can confidently say that there is no car in the im
```

Before we finish the experiment, we download the saved JSON output:

```
!curl -L https://raw.githubusercontent.com/Denis2054/Transforme
```

We diligently parse the file to peek inside:

```
with open("HGPT3.json") as f:
    data = json.load(f)
    # Extract and Print the tasks and their details
    for key, value in data.items():
        print(f"Task ID: {value['task']['id']}")
        print(f"Task Type: {value['task']['task']}")
        print(f"Dependencies: {', '.join(map(str, value['task']['depen'))")
        if 'args' in value['task']:
            for arg_key, arg_value in value['task']['args'].items():
                print(f"    {arg_key.capitalize()}: {arg_value}")
        # Extract and Print the inference results
        if 'inference result' in value:
            print("Inference Result:")
            if isinstance(value['inference result'], dict):
                for result_key, result_value in value['inference re
                    print(f"    {result_key.capitalize()}: {result_")
            else: # In case it's a list like in the visual-questio
                for item in value['inference result']:
                    print(f"        Answer: {item['answer']} (Score: {i
            # Extract and Print the model used and the reason
            if 'choose model result' in value:
                print(f"Model Chosen: {value['choose model result'][i
```

```
        print(f"Reason: {value['choose model result']['reason']}")  
    print('-' * 50) # A separator line for better readability
```

This time, `ydshieh/vit-gpt2-coco-en` doesn't detect the slightest glimpse of a car and states: `a dark night with a light on a street` as shown in the following output:

```
Task ID: 0  
Task Type: image-to-text  
Dependencies: -1  
    Image: /car-in-fog.png  
Inference Result:  
    Generated text: a dark night with a light on a street  
Model Chosen: ydshieh/vit-gpt2-coco-en  
Reason: Only one model available.  
-----  
Task ID: 1  
Task Type: object-detection  
Dependencies: -1  
    Image: /car-in-fog.png  
Inference Result:  
    Generated image with predicted box: /images/0a15.jpg  
    Predicted: []  
Model Chosen: facebook/detr-resnet-101  
Reason: Only one model available.  
-----  
Task ID: 2  
Task Type: visual-question-answering  
Dependencies: 1  
    Image: /car-in-fog.png  
    Text: Is there a car in this image?  
Inference Result:  
    Answer: no (Score: 0.9993352293968201)  
    Answer: yes (Score: 0.019275549799203873)  
    Answer: 0 (Score: 0.0007861475460231304)  
    Answer: nowhere (Score: 0.000450292369350791)  
    Answer: n (Score: 0.0002609568473417312)
```

```
Model Chosen: dandelin/vilt-b32-finetuned-vqa
Reason: Only one model available.
```

Should we give up? We could say that the limit of state-of-the-art vision transformers cannot solve the problem we have faced for two chapters. Maybe. Maybe not.

Let's make a last attempt and ask Hugging Face to enhance the car in the fog image:

```
Please generate a better image based on /car_in_fog.jpg
```

HuggingGPT following the instructions:

```
Based on the inference results, I have generated a better image
```

Let's download the saved image:

```
!curl -L https://raw.githubusercontent.com/Denis2054/Transforme
```

We can display it:

```
from PIL import Image
# Define the path of your image
image_path = "/content/HGPT3_B.jpg"
# Open the image
image = Image.open(image_path)
image
```

The output confirms the limit of the model. There is absolutely no trace of a car in the image!



*Figure 19.6: An image with no car*

Our experiment seems doomed to failure. But we need to analyze it from another perspective using the experience gained from HuggingGPT. HuggingGPT has remarkably demonstrated the potential of cross-platform (Hugging Face and OpenAI ChatGPT) multi-model associations to solve problems with an LLM controller managing the process.

Let's build on these innovative ideas and create a solution for complex problems.

## CustomGPT

We tried to obtain a result with our validation dataset through HuggingGPT. We would have replaced the very difficult image HuggingGPT failed to

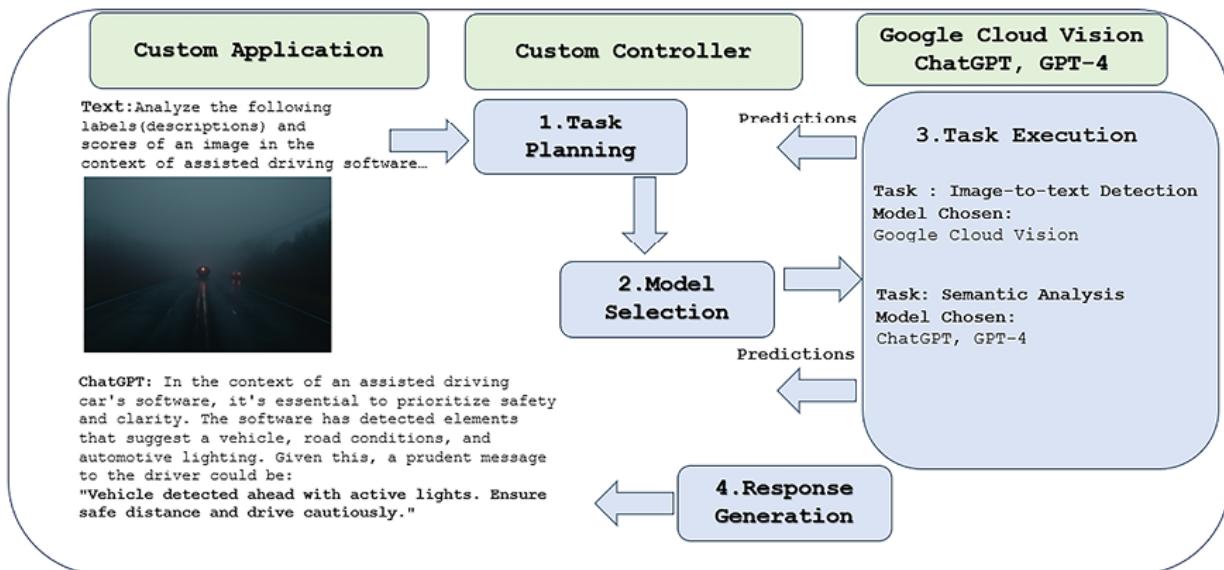
analyze correctly. In that case, we would have a shiny, optimistic notebook showing that AI is fantastic and easy to implement with automated near F-AGI tools.

Unfortunately, even an innovative HuggingGPT system cannot solve all AI problems.

The HuggingGPT system does not contain all the available models for all NLP and computer vision tasks although it's a good initiative.

AI Experts will necessarily have to confirm whether a specific AI problem cannot be solved or not. In this section, we show that, in some cases, creative solutions can work.

We will build on the concept of a cross-platform multi-model system such as HuggingGPT. In this case, we create a custom pipeline to solve a specific problem. We will implement CustomGPT, a cross-platform chained-model solution shown in *Figure 19.7*:



*Figure 19.7: CustomGPT's four-step process*

The CustomGPT process in this section shows that with some creativity, AI Experts can solve complex problems:

1. A custom application processes that input and sends it to a custom task planning pipeline.
2. The custom controller manages the model selection. This selection could be displayed in an interface that CustomGPT users or the system can launch automatically for a given task.
3. The pipeline then executes the models selected. In this example, Google Cloud Vision's output is chained to the input of ChatGPT.
4. ChatGPT's output performs a semantic analysis of the features Google Cloud Vision provides.

We will first ask Google Cloud Vision to provide features for the validation dataset. We will run ChatGPT on the very difficult image.

Although CustomGPT is a prototype with interactive functions, it can be automated in a real-life project. It is critical to understand the potential of HuggingGPT and CustomGPT. Why? Because we tend to overestimate AI innovations in the short term and underestimate them in the long term. Upskilling to be ready for the near future has become an essential factor in an AI professional's career.

We will first run Google Cloud Vision before running ChatGPT, GPT-4.

## Google Cloud Vision

Google Cloud Vision is a great way to classify images because it is a powerful system that can be used to identify objects, scenes, and faces in images. We could implement a request with an API as in *Chapter 14, Exploring Cutting-Edge LLMs with Vertex AI and PaLM 2*. However, this

section aims to see if Google Cloud Vision can help with the difficult images of the validation set.

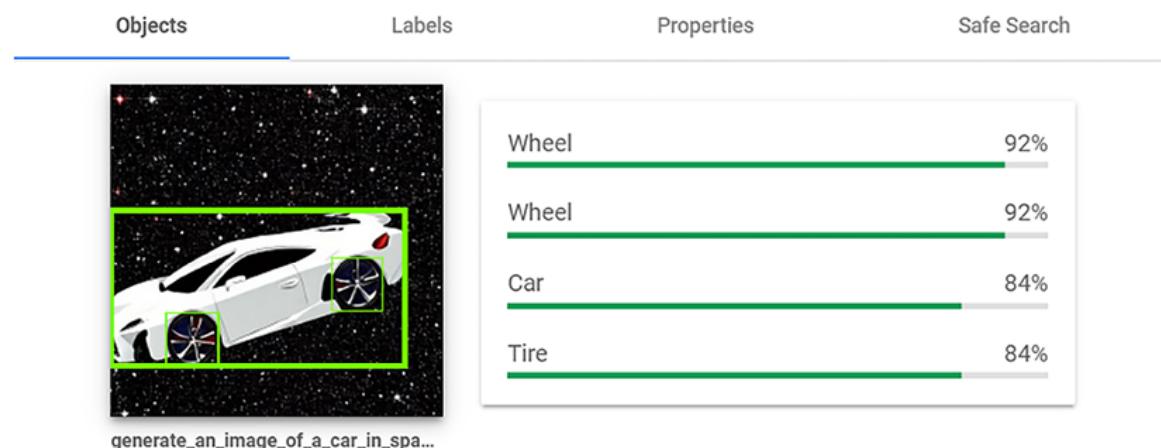
To find out, we'll go straight to the online interface:

<https://cloud.google.com/vision/docs/drag-and-drop>.

Let's first submit the easy image of a car in space.

## Level 1: Easy

Google Cloud Vision produces an interesting output for `generate_an_image_of_a_car_in_space.jpg` of the validation set, as shown in *Figure 19.8*:



*Figure 19.8: Google Cloud Vision analysis of a car in space*

Google Cloud Vision seems reluctant to jump to conclusions. It identifies a car but also adds wheels and a tire. To analyze the output, we can parse the JSON output of the process after copying it into a file:

```
!curl -L https://raw.githubusercontent.com/Denis2054/Transforme
```

The notebook can open and display the output in a pandas DataFrame:

```
with open("GCV.json") as f:  
    data = json.load(f)  
labels = []  
for label in data["labelAnnotations"]:  
    labels.append([label["description"], label["score"]])  
df = pd.DataFrame(labels, columns=["description", "score"])  
df
```

We can have a better view of the output of the vision model:

|    |                          |          |
|----|--------------------------|----------|
| 0  | Wheel                    | 0.969944 |
| 1  | Tire                     | 0.966153 |
| 2  | Vehicle                  | 0.956403 |
| 3  | Car                      | 0.942123 |
| 4  | Hood                     | 0.918258 |
| 5  | Automotive parking light | 0.917619 |
| 6  | Automotive lighting      | 0.906899 |
| 7  | Automotive tire          | 0.898249 |
| 8  | Black                    | 0.895158 |
| 9  | Automotive design        | 0.879315 |
| 10 | Hubcap                   | 0.877525 |
| 11 | Alloy wheel              | 0.876464 |
| 12 | Motor vehicle            | 0.873120 |
| 13 | Bumper                   | 0.845483 |
| 14 | Vehicle door             | 0.827911 |

Figure 19.9: The model's analysis of the car in space

Unlike the classification task models we ran in *Chapter 18, Hugging Face AutoTrain: Training Vision Models without Coding*, Google Cloud Vision

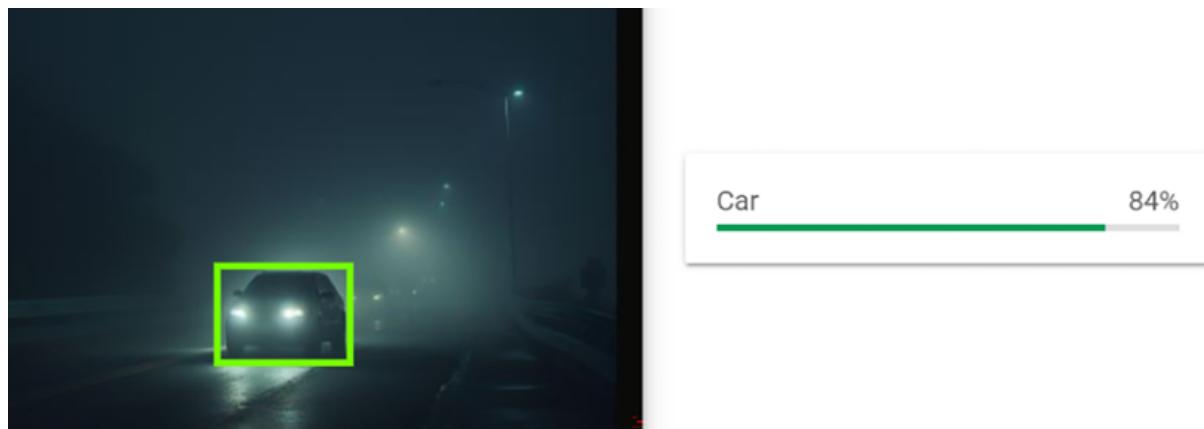
doesn't display a class but a list of features for this image, which is an interesting approach.

The list contains features about the car in the image. The output displays the top probabilities the model produced while analyzing the image. This information provides insights into the way the model makes inferences.

What will the output of the difficult image produce?

## Level 2: Difficult

The output of the `car_in_night.jpg` image in the interface only contains one property: `car` (see *Figure 19.10*):



*Figure 19.10: Google Cloud Vision analysis of a car in the night*

We need to investigate further by retrieving the JSON output stored in a text file:

```
!curl -L https://raw.githubusercontent.com/Denis2054/Transforme
```

Let's load and parse the JSON file:

```

import json
with open("GCV2.json") as f:
    data = json.load(f)
labels = []
for label in data["labelAnnotations"]:
    labels.append([label["description"], label["score"]])
df = pd.DataFrame(labels, columns=["description", "score"])
df

```

The output shows very interesting properties:

|    |                     |          |
|----|---------------------|----------|
| 1  | Vehicle             | 0.938368 |
| 2  | Automotive lighting | 0.923168 |
| 3  | Road surface        | 0.878716 |
| 4  | Asphalt             | 0.869295 |
| 5  | Automotive design   | 0.864141 |
| 6  | Headlamp            | 0.860162 |
| 7  | Automotive mirror   | 0.827779 |
| 8  | Automotive exterior | 0.824365 |
| 9  | Motor vehicle       | 0.809984 |
| 10 | Rolling             | 0.797941 |

*Figure 19.11: The model's analysis of the car at night*

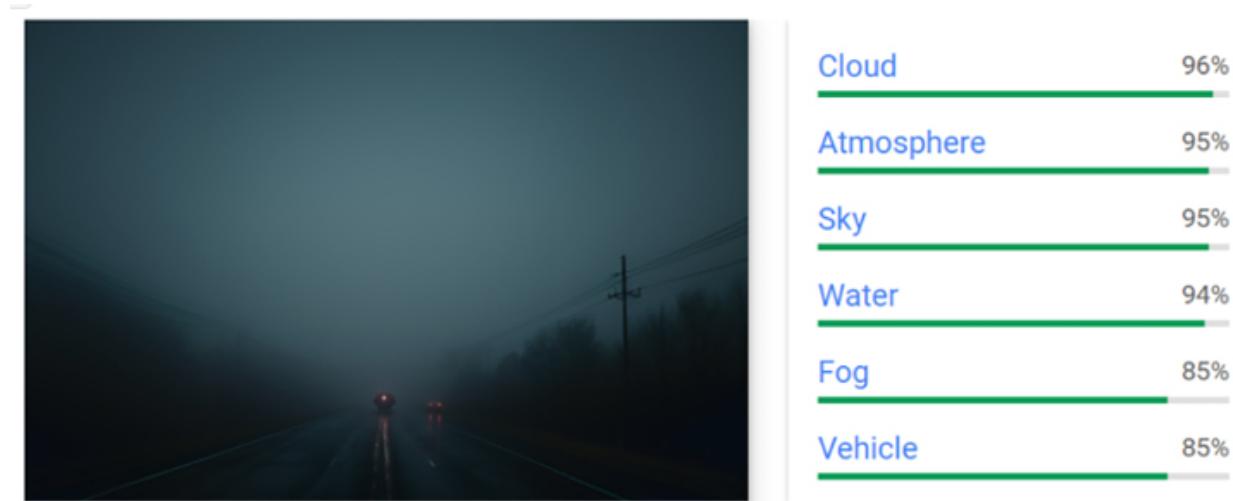
We can see the features of a car but also the features of the road. We now gain more insights into vehicle and road features.

Will we finally obtain a result with the very difficult image?

## Level 3: Very difficult

What will Google Cloud Vision produce for a `car_in_fog.jpg` image?

The output is confusing, as shown in *Figure 19.22*:



*Figure 19.12: Output of a car in the fog image*

Unlike the easy and difficult images, the vision model did not draw a nice rectangle around the car. The weather conditions come out on top. However, one of the features of the image is a `Vehicle` with an 85% confidence level. That is a promising step forward.

Let's parse the JSON output that was copied in a file:

```
!curl -L https://raw.githubusercontent.com/Denis2054/Transforme
```

We can now display the features in a pandas DataFrame:

```

import json
with open("GCV3.json") as f:
    data = json.load(f)
labels = []
for label in data["labelAnnotations"]:
    labels.append([label["description"], label["score"]])
df = pd.DataFrame(labels, columns=["description", "score"])
df

```

The output contains two features for a car: vehicle and automotive lighting.

The weather conditions come out on top:

|   | description         | score    |
|---|---------------------|----------|
| 0 | Cloud               | 0.957413 |
| 1 | Atmosphere          | 0.948181 |
| 2 | Sky                 | 0.945039 |
| 3 | Water               | 0.938452 |
| 4 | Fog                 | 0.850576 |
| 5 | Vehicle             | 0.848766 |
| 6 | Grey                | 0.841551 |
| 7 | Automotive lighting | 0.831509 |

*Figure 19.13: The model's analysis of the very difficult image of a car at night*

The conclusion of this experiment is quite interesting, as we will see. A human reading this output will most certainly conclude that a vehicle is

driving in bad weather, although the model did not clearly make the inference.

We will now implement a custom cross-platform system.

## Model chaining: Chaining Google Cloud Vision to ChatGPT

The *Google Cloud Vision* section produced an interesting output for the very difficult image. However, the model only displayed a list of probabilities. That's not how humans think. We have semantic capabilities to interpret features and probabilities.

We humans can isolate the features produced by Google Cloud Vision for the very difficult image that exceed a confidence level above 0.8 and group them into sets:

```
Weather= {Cloud, Atmosphere, Sky, Water, Fog, Atmospheric  
phenomenon}
```

```
Vehicle={Vehicle, Automotive lighting}
```

```
Road={ Road surface}
```

```
other={Electricity}
```

A human, using our semantic and reasoning abilities, could make a chain of logical conclusions when analyzing the features exceeding a probability of 0.8:

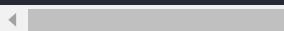
A car is on the road in cloudy, foggy weather.

ChatGPT has been trained to reason. Let's chain the output of Google Cloud Vision to the input of ChatGPT, GPT-4, and see what happens:

---

DR 1.Analyze the following labels(descriptions) and scores of a

0 Cloud 0.957413  
1 Atmosphere 0.948181  
2 Sky 0.945039  
3 Water 0.938452  
4 Fog 0.850576  
5 Vehicle 0.848766  
6 Grey 0.841551  
7 Automotive lighting 0.831509  
8 Atmospheric phenomenon 0.830896  
9 Electricity 0.829284  
10 Road surface 0.814302  
11 Landscape 0.764860  
12 Dusk 0.758024  
13 Track 0.752574  
14 Horizon 0.746588  
15 Asphalt 0.744103  
16 Haze 0.730276  
17 Road 0.724686  
18 Mist 0.716356  
19 Midnight 0.716127  
20 Overhead power line 0.684156  
21 Darkness 0.682619  
22 Tree 0.677585  
23 Public utility 0.661300  
24 Nonbuilding structure 0.642936  
25 Evening 0.640874  
26 Dawn 0.637772  
27 Symmetry 0.615672  
28 Natural landscape 0.612960  
29 Ocean 0.603301  
30 Freeway 0.602228  
31 Highway 0.601216  
32 Lane 0.587314  
33 Night 0.576490  
34 Monochrome 0.556066  
35 Lens flare 0.555109  
36 Road trip 0.527954  
37 Electrical supply 0.505197



ChatGPT, with GPT-4, demonstrates its powerful reasoning capabilities:

Firstly, let's analyze the given labels and scores:

1. From the list, labels that can be related to vehicles and roads are:
  - o Vehicle
  - o Automotive lighting
  - o Road surface
  - o Track
  - o Asphalt
  - o Road
  - o Freeway
  - o Highway
  - o Lane
  - o Road trip
2. To extract all the labels that are related to vehicles and roads that exceed a score of 0.8 and list them:
  - o Vehicle: 0.848766
  - o Automotive lighting: 0.831509
  - o Road surface: 0.814302
3. To determine if there is a car in the image or not:
  - o There is a label "Vehicle" with a score of 0.848766, but this doesn't specifically mean that there's a car. "Vehicle" can represent any mode of transportation, such as cars, bikes, buses, etc. However, the presence of the label "Automotive lighting" (score: 0.831509) strongly suggests that a car might be present since automotive lighting typically refers to lights on vehicles, especially cars.

In conclusion: The labels related to vehicles and roads that exceed a score of 0.8 are:

- Vehicle
- Automotive lighting
- Road surface

There's a high likelihood that there is a car in the image, mainly because of the “Automotive lighting” label. However, we can't say with 100% certainty based only on the given data.

ChatGPT has taken an enormous step forward in our investigation. Like a human, it thinks there is a likelihood there is a car. But we haven't provided the context of our problem yet.

Let's provide ChatGPT with more information:

Given your response and also taking the context of your response being the software of an assisted driving car, what message would you send to the driver with your software in this situation?

ChatGPT has unique near-human reasoning abilities. Will we finally get the solution to our problem? Let's read ChatGPT's response:

In the context of an assisted driving car's software, it's esse "Vehicle detected ahead with active lights. Ensure safe distanc This message aims to make the driver aware of the potential veh

Yes! ChatGPT recommends that the correct message be sent to the driver:

"Vehicle detected ahead with active lights. Ensure safe distance and drive cautiously."

Cross-platform and model chaining can most certainly lead the way to F-AGI. We need to enhance our creative skills and find ways, as HuggingGPT did, to assemble the best of the available models.

Let's have some fun and do image-to-video cross-platform model chaining.

## Model Chaining with Runway Gen-2

In this section, we will chain the output of a Midjourney image to the input of the Gen-2 image to video service.

First, let's download an example Gen-2 video produced by a prompt asking the model to create a video of an astronaut driving in a rover on Mars:

```
!curl -L https://raw.githubusercontent.com/Denis2054/Transforme
```

We can now view it in the chapter notebook:

```
from moviepy.editor import *
# Load myHolidays.mp4 and select the subclip 00:00:00 - 00:00:00
clip = VideoFileClip("Gen-2_Mars.mp4").subclip(00, 7)
clip = clip.loop(5)
clip.ipython_display(width=900)
```

The video you can view in the notebook is quite promising.

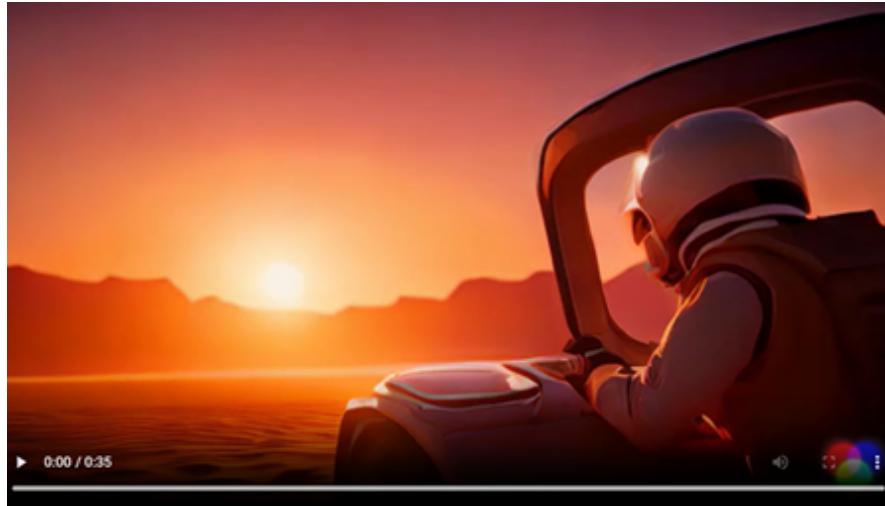


Figure 19.14: A screenshot of an AI-generated video about an astronaut on Mars

Now, let's chain Midjourney to Gen-2.

## Midjourney: Imagine a ship in the galaxy

Let's download an image produced by Midjourney and created using a ship in the galaxy prompt:

```
!curl -L https://raw.githubusercontent.com/Denis2054/Transforme
```

We can now display the image:

```
from PIL import Image
# Define the path of your image
image_path = "/content/ship_in_galaxy.jpg"
# Open the image
image = Image.open(image_path)
image
```

The output is quite beautiful and helpful, as shown in *Figure 19.25*. Producing such an image with classical software would take time and resources. For example, we can imagine how this image could be used for a marketing campaign:



*Figure 19.15: An AI-generated image of a ship sailing the galaxy*

Can we chain this output to the input of another system? Let's find out and ask Gen-2 to make a video with this image.

## Gen-2: Make this ship sail the sea

Our cross-platform, model-chaining experience continues by submitting the output of Midjourney to Runway Gen-2 with a simple prompt: “Make this ship sail the sea.”

Let's download the result:

```
!curl -L https://raw.githubusercontent.com/Denis2054/Transforme
```

Gen-2 transforms the Midjourney image into a representation it can manage. The resulting video remains quite promising. Let's display it:

```
from moviepy.editor import *
# Load myHolidays.mp4 and select the subclip 00:00:00 - 00:00:07
clip = VideoFileClip("ship_in_galaxy.mp4").subclip(00, 7)
clip = clip.loop(5)
clip.ipython_display(width=900)
```

You can run the output in the chapter notebook:



Figure 19.16: An AI-generated video of the ship

In this section, we applied cross-modeling to vision Generative AI. For example, the marketing potential could take us quite far into automated advertising campaigns.

It's time to summarize this chapter and move on to our next exploration.

## Summary

In this chapter, HuggingGPT concepts led to solving the classification problem of cars in the fog and night that arose in *Chapter 18, Hugging Face AutoTrain: Training Vision Models without Coding*. HuggingGPT's innovative approach uses ChatGPT as a controller, managing the comprehensive library of Hugging Face models.

We first defined F-AGI as the ability to attain human-level functionality for a real-life task in a closed ecosystem. For example, a computer vision AI agent can replace a human fire-alert watcher 24/7 over vast territories. The chapter addressed practical computer vision abilities that enhance human activity without threatening jobs or getting involved in politics.

Then, the chapter notebook downloaded a validation set containing an easy, difficult, and very difficult image containing a car. The goal was to find a way to identify a vehicle in a very difficult image simulating a video frame of an assisted driving AI agent.

We ran HuggingGPT, an innovative system that uses an LLM such as ChatGPT as a controller to process input, select models, execute the tasks, and generate the results. The goal was to show the images HuggingGPT could analyze and the system's limits.

We created CustomGPT, a custom cross-platform chained modeling system. Google Cloud Vision provided productive insights, though it did not clearly identify a vehicle in the car in the fog at night. HuggingGPT faced the same problem.

The problem was solved by submitting the output of Google Cloud Vision to ChatGPT in a cross-platform chained model approach. This time, finally, ChatGPT identified a car in the fog and night and provided a message for the driver of an assisted-driving vehicle.

Finally, we ran another cross-platform, chained model experiment using Midjourney's Generative AI to produce an image of a ship and submit the output to Runway Gen-2, which created a Generative AI video.

The door of F-AGI has now opened through cross-platform, chained model ecosystems. In *Chapter 20, Beyond Human-Designed Prompts with Generative Ideation*, we will go further and implement Generative AI ideation with cross-platform and chained-model systems.

## Questions

1. AGI already exists and is spreading everywhere. (True/False)
2. Functional AGI is conscious. (True/False)
3. Functional AGI can perform human-level tasks in a closed environment. (True/False)
4. Vision models can now identify all objects in all situations. (True/False)
5. HuggingGPT leverages the abilities of an LLM such as ChatGPT. (True/False)
6. ChatGPT can be a controller in the HuggingGPT ecosystem. (True/False)
7. HuggingGPT is not a cross-platform system. (True/False)
8. Chained models can improve overall vision model performances. (True/False)
9. Google Cloud Vision and ChatGPT cannot be chained. (True/False)

10. Midjourney images can become the input of Gen-2 to produce videos.  
(True/False)

## References

- Pano: <https://www.pano.ai/>
- Hugging Face HuggingGPT:  
<https://huggingface.co/spaces/microsoft/HuggingGPT>
- Google Cloud Vision:  
<https://cloud.google.com/vision/docs/drag-and-drop>
- Microsoft Bing: <https://www.bing.com/new>
- Midjourney: <https://www.midjourney.com/>
- Runway Gen-2: <https://research.runwayml.com/gen2>

## Further Reading

- *Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, Yuetong Zhuang, 2023, HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face:*  
<https://arxiv.org/abs/2303.17580>

## Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://www.packt.link/Transformers>



# 20

## Beyond Human-Designed Prompts with Generative Ideation

Generative ideation is defined as the generation of ideas and content without human intervention. Developing the ideation automation process remains within the scope of human designers and developers. In production, an application will not require human intervention with prompts.

A generative ideation ecosystem automates the production from an idea to text and image content. The development phase requires highly skilled human AI experts. For an end user, the ecosystem is a click-and-run experience.

In *Part I*, we will first define the new world of generative ideation ecosystems. You will learn to think of yourself as a music composer and the AI tools as instruments. In music, the toolbox includes instruments such as guitars, drums, saxophones, and keyboards. In this chapter, you will compose generative ideation with a fantastic orchestra that contains ChatGPT/GPT-4, Llama 2, Midjourney, Stable Diffusion, and Microsoft Designer (powered by DALL-E and GPT models).

We will stay far away from job displacement, replacement, and all the issues related to advanced automation. We will remain within the boundaries of

ethical and exciting AI. You will see how to add potential to a company through automation. The use cases in this chapter will focus on the many companies that do not have a marketing department, human resources, or the budget to access high-profile marketing campaigns. The know-how you will acquire in this chapter can help businesses that cannot afford expensive marketing campaigns gain visibility.

In *Part II*, we then see how to help a business with no marketing resources and a relatively small budget leverage the potential of Midjourney and Microsoft Designer with Llama 2. The development of such systems requires classical AI skills. However, the end user will not be burdened with designing prompts in an environment without time to spend on marketing.

In *Part III*, we will then replace interactive market-entry-level functionality with fully Generative AI text and image-to-text technology with GPT-4 and Stable Diffusion.

*The future is yours!* The productivity method described in this section will guide you into your potential future with Generative AI, VR, and more.

By the end of this chapter, you will understand how to deliver ethical, exciting, generative ideation to companies with no marketing resources that will enjoy your implementations. You will be able to expand generative ideation to any field in an exciting, cutting-edge, yet ethical ecosystem.

This chapter covers the following topics:

- Defining automated generative ideation
- The world beyond human-designed prompts
- Implementing a **Retrieval Augmented Generation (RAG)** based system
- ChatGPT to summarize a process

- Meta Llama 2 for prompt generation
- Midjourney for image generation
- Microsoft Designer to generate flyers and posters
- GPT-4 to automate prompt creation
- Stable Diffusion for image generation



With all the innovations and library updates in this cutting-edge field, packages and models change regularly. Please go to the GitHub repository for the latest installation and code examples:

<https://github.com/Denis2054/Transformers-for-NLP-and-Computer-Vision-3rd-Edition/tree/main/Chapter20>.

You can also post a message in our Discord community (<https://www.packt.link/Transformers>) if you have any trouble running the code in this or any chapter.

Our first step will be to define generative ideation.

# Part I: Defining generative ideation

Ideation refers to creating new ideas through intuitive thinking, brainstorming, and other imaginative methods.

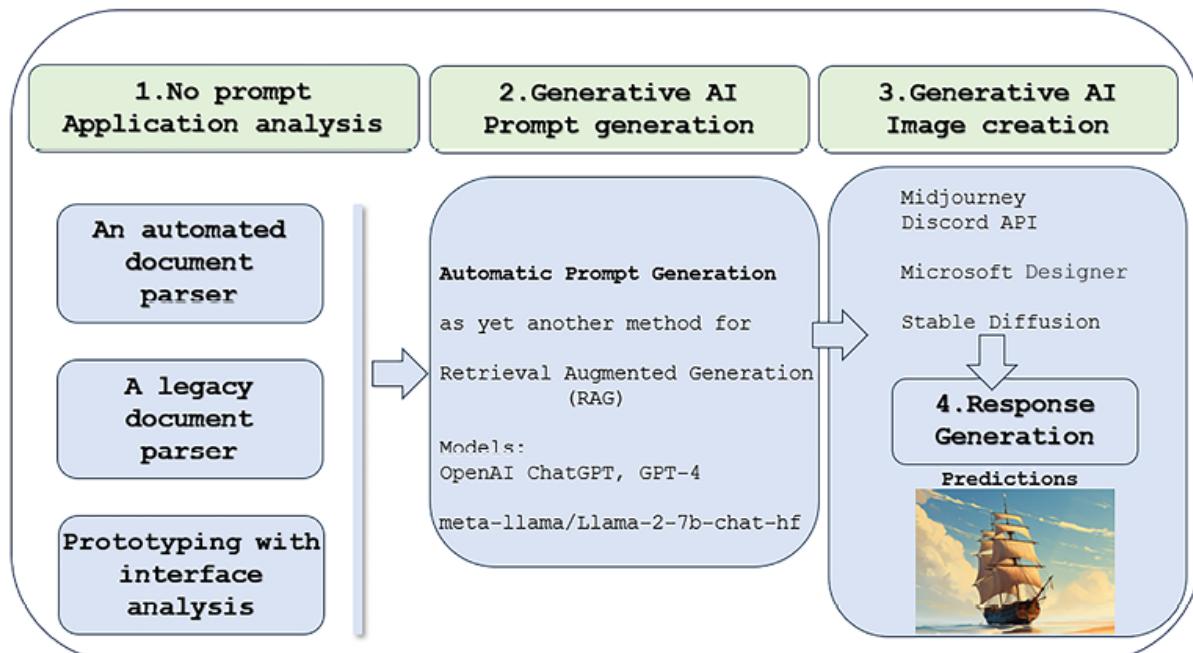
Generative ideation leverages Generative AI to create content such as text, text-to-image, image-to-text, and image-to-image artifacts.

Automated generative ideation takes generative ideation a step further and suppresses the need for human-designed prompts. The goal of this chapter is to show how to build such tools.

Let's define an automated ideation architecture.

## Automated ideation architecture

*Figure 20.1* shows the framework implemented in this chapter:



*Figure 20.1: Generative AI ideation framework*

In this chapter, we will cover the four-step development process of automated generative ideation:

## 1. No prompt

Automated features, such as Python code, will automate document retrieval, parsing, and search in a now classical Generative AI **Retrieval Augmented Generation (RAG)** process.

Legacy, classical rule-based parsing and query programming will alleviate user actions to limit them to click-and-run features.

Classical web design prototypes will provide the basis for application development.

## 2. Generative AI: prompt creation

A prompt will not trigger generative content but will generate prompts for chained generative models. The paradigm shift from human-designed prompts to Generative AI prompts will help users who want to avoid imagining prompts that are time-consuming and require trial-and-error approaches. This is yet another RAG in which an automated system will retrieve and parse documents (*Step 1* above) to create prompts that will, in turn, be the input of another Generative AI model.

Llama and GPT-4 will replace the human design process. This process requires a development investment but will free end users of the burden of finding the right prompt when human resources and budgets are scarce.



**Large Language Model Meta AI(LLaMA)** is often spelled Llama when used as a model by Meta:



<https://llama.meta.com/> Both LLaMA (more academic) or Llama will be used in this chapter.

### 3. Generative AI: image creation

This chapter focuses on computer vision. The goal of the ecosystem will be to chain Generative AI text outputs to text-to-image inputs for Midjourney, Microsoft Designer (DALL-E and GPT models), and Stable Diffusion.

### 4. Response generation

The output will be image outputs and text-image artifacts. Building the ecosystem requires development, but the final application should be limited to clicks and runs for the end user.

## Scope and limitations

The scope of this chapter illustrates the burgeoning rise of automated Generative AI ideation through **proof of concept** coding prototype examples. The goal remains to prove that generative ideation will expand exponentially as Generative AI continues to expand.

The main limitation of this chapter is the hands-on approach to get our hands dirty to understand the ideation process that leading platforms produce to become seamless applications from an end-user perspective. An AI expert must deeply understand this area to provide efficient advice and services. As such, the chapter contains many references to development that must be done in real-life projects.

With this in mind, we can start our hands-on journey:

- *Part II* begins with a use case illustrating a minimal cost-effective solution for entry-level marketing systems.
- *Part III* is the roadmap for a fully automated system.

Let's begin with market entry-level solutions.

# Part II: Automating prompt design for generative image design

For an AI developer, going beyond human-designed prompts with

Generative AI is not very different from creating classical complex reports based on database queries:

- The end user often doesn't want to have anything to do with making the report.
- The end user defines the report.
- A developer creates SQL queries, for example.
- The developer creates a click-and-run interface.
- The end user works on the automatically generated report.

Why does this work? The key concept is a *closed environment*. In a small company or a small department of a large corporation, the tasks can be well defined, the data well identified, and the users willing to automate work that takes them hours to do and infringes on their free time.

In this section, we will:

- Work in a closed environment.
- Reproduce classical software automation for end users.
- Build a use case for a marketing campaign for a small company that cannot afford to hire costly marketing personnel or consultants.
- Stay as close as possible to non-AI methods that have been successful for decades in companies worldwide.

We will begin by examining how to initiate such a project.

# **ChatGPT/GPT-4 HTML presentation**

We need to take a breath here and slow down. We are talking to someone who doesn't have much time to listen to us in a small company or a small department in a large corporation. The person might be overworked, know nothing about AI, or have other preoccupations. Not all projects are cool online cloud applications that work in every situation! A considerable market will remain for the segments major cloud AI platforms cannot reach. And that is precisely where AI experts can thrive!

However, there are two fundamental rules for this type of project for the product to survive in production with somebody in the company doing first-level support:

- We must learn how the company works as we build the project.
- The key user(s) must learn more about our technology to communicate internally and provide first-level support.

We must build a communication strategy within the company to federate the team.

## **ChatGPT with GPT-4 provides the text for the presentation**

The best project presentation will always remain the one made by the customer's team. It may sound less specialized, but it will surely resonate as an authentic, integrated approach.

The main user must understand the basic concepts of the project. However, this person may have no *idea* how to present the project in a meeting to initiate the whole process.

That is when automated ideation enters the project!

Open `Automated_Design.ipynb` in the chapter's directory of the GitHub repository.

The first automated ideation will consist of asking ChatGPT to sum up the process and create an HTML graph to illustrate the project. We write a classical program that reads the notebook in its downloaded Python version, saves it in a text file, and then sends the content to GPT-4 through an API with the proper instructions. Those instructions can become a hard-coded template for a promptless application on the user end.

We can explain that the whole summarization process will be automated once the prototype is approved. Bear in mind that many classical software applications already build very complex pipelines. They might not seem as “cool” as AI, but they are the backbone of many architectures.

In our use case, let's suppose we are working with a person with no AI knowledge but who is very motivated to test and asks ChatGPT for assistance.

### **Source prompt for ChatGPT:**

```
Provide an explanation of how Llama invents a prompt that Midjourney generates an Image with, and Microsoft Designer combines (text and image) to create a nice corporate flyer. Make the presentation engaging for a corporate meeting, please:
```

Note that a hard-coded prompt template library can be part of a knowledge base and triggered like any ordinary SQL script in a database.

In less than ten seconds, ChatGPT will provide the text and graph to illustrate the power of the functionality of the project with an automated presentation and a graph. What a time-saver! First, the presentation by ChatGPT for a meeting:

## ChatGPT with GPT-4, provides the text for the presentation:

Step Automating Ideation and Design

Motivation: "Harnessing the Power of AI for Innovative Corporate Designs"

Functionality: "Llama -> Midjourney -> Microsoft Designer: A Seamless Integration"

Background: A sleek futuristic image representing AI and design integration, perhaps a neural network or a stylized digital flyer.

Step 1: Introducing LlaMA The Brain Behind Creative Prompting

Llama isn't just another AI. It's your digital brainstorming partner.

In the world of content, the right prompt is the seed to a great idea.

With its advanced algorithms, Llama invents engaging prompts tailored to your brand's voice and values.

Step 2: Enter Midjourney

Bringing Ideas to Visual Life

Ideas need visualization. That's where Midjourney comes in.

Imagine feeding Llama's prompt into Midjourney and witnessing the birth of a stunning, on-brand image.

It's like having an award-winning graphic designer on your team, working 24/7!

### Step 3: Microsoft Designer - The Finishing Touch

#### Perfecting the Art

A great design is more than just text and image. It's about harmony.

Microsoft Designer takes the image from Midjourney, combines it with Llama's prompt, and crafts a corporate flyer that's not just beautiful, but effective.

The ease of integration and user-friendly interface ensures anyone can be a design pro!

### Step 4: The Seamless Workflow

#### The Power Trio in Action

From thought to tangible design in three seamless steps!

Save time, reduce costs, and wow your audience every single time.

### Step 5: Engaging Examples

#### Seeing is Believing!

Here's what we achieved with this integration.

Consistent, innovative, and always on-brand!

## Step 6: Closing Thoughts & Q&A

### Revolutionizing Corporate Design

In today's fast-paced digital age, staying innovative is the key.

With Llama, Midjourney, and Microsoft Designer, we're not just keeping up; we're leading the way!

Questions? Let's dive deeper!

Think of the time saved for the key user, who can build expert-level presentations in seconds! Now ChatGPT will create a graph in HTML to illustrate the presentation.

## ChatGPT with GPT-4 provides a graph in HTML to illustrate the presentation

The following code and display were entirely written by ChatGPT, GPT-4, based on the previous dialog. Note that ChatGPT, being stochastic, may not provide the same response from one run to another.

Click on the cells. Then sit back and enjoy.

ChatGPT installs `plotly` to create graph objects:

```
!pip install plotly
```

ChatGPT imports the graph module:

```
import plotly.graph_objects as go
```

ChatGPT creates the graph that corresponds to its representation of the project:

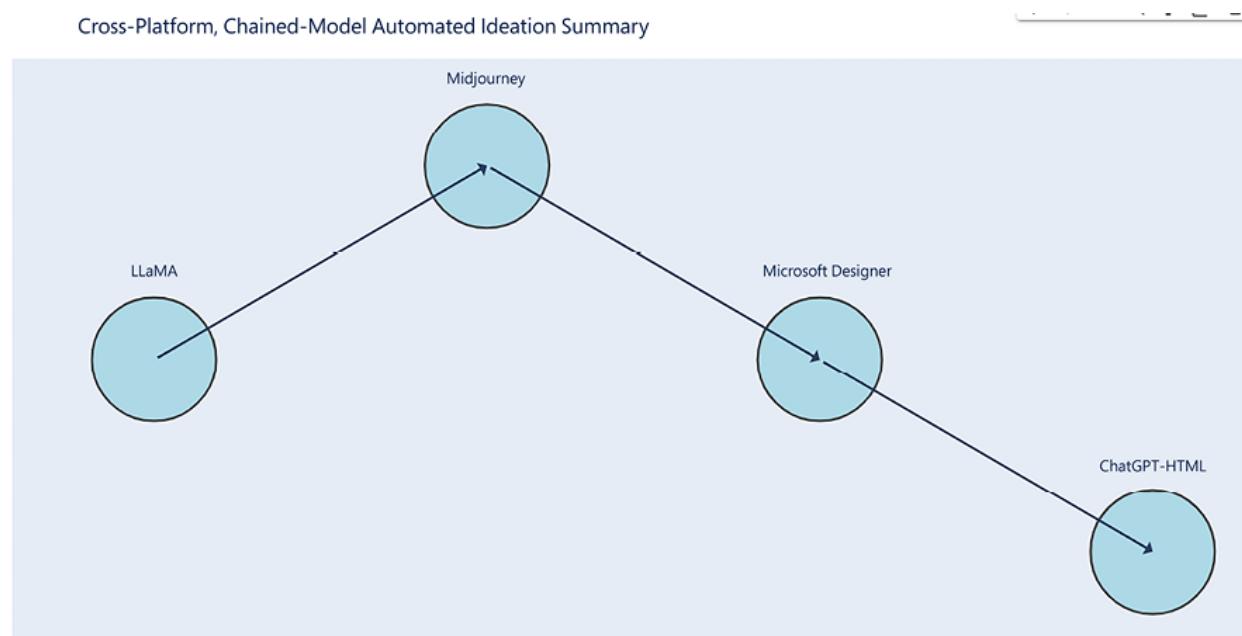
```
# Define the nodes and edges
nodes = ["LLaMA", "Midjourney", "Microsoft Design", "ChatGPT-H"]
edges = [(0, 1), (1, 2), (2, 3)]
# Adjusted positions for shorter edges
x = [0, 0.6, 1.2, 1.8]
y = [0, 0.6, 0, -0.6]
# Create the arrow annotations
annotations = []
for edge in edges:
    annotations.append(
        go.layout.Annotation(
            showarrow=True,
            arrowsize=1,
            arrowhead=1,
            startarrowhead=0,
            arrowwidth=2,
            ax=x[edge[0]],
            ay=y[edge[0]],
            axref='x',
            ayref='y',
            x=x[edge[1]],
            y=y[edge[1]],
            xref='x',
            yref='y'
        )
    )
node_trace = go.Scatter(
    x=x, y=y,
    mode='markers+text',
    hoverinfo='text',
    textposition='top center',
    marker=dict(
        showscale=False,
        color='lightblue', # Set the color to light-blue
#colorscale='YlGnBu',
        size=100,
        colorbar=dict(
            thickness=15,
```

```

        title='Node Connections',
        xanchor='left',
        titleside='right'
    ),
    line=dict(width=2)))
node_trace.text = nodes
node_trace.hovertext = nodes
fig = go.Figure(data=[node_trace],
                  layout=go.Layout(
                      title='Cross-Platform, Model-Chained Automated
                      titlefont_size=16,
                      showlegend=False,
                      hovermode='closest',
                      margin=dict(b=0, l=0, r=0, t=40),
                      xaxis=dict(showgrid=False, zeroline=False, sh
                      yaxis=dict(showgrid=False, zeroline=False, sh
                      annotations=annotations) # Add the arrow anno
                  )
fig.show()

```

The output is a nice graph that illustrates the chained-model process:



*Figure 20.2: Image of the chained-model process*

This presentation is certainly not perfect. But it illustrates the tremendous paradigm shift taking place. Generative AI bots are becoming human-level agents and co-workers.

Microsoft 365 Copilot has begun to expand automated ideation in its products. PowerPoint will soon take automated ideation to unseen levels. Google Workspace has several proprietary copilots and many AI-driven add-ons. This is just the beginning, and as the systems improve, that will surely increasingly baffle us!

We will now use Llama 2 to automate prompt generation.

## Llama 2

In this section, we will implement Llama 2 in a *closed environment* to automate prompt design with Generative AI. A closed environment is created by limiting the scope of a project to a specific domain within a department of an organization and gathering well-organized documents (emails, word processing files, spreadsheet files, media, or other sources). It then becomes feasible to build static AI queries just as for any classical software reports.

## A brief introduction to Llama 2

Meta (formerly Facebook) released Llama 2 on July 18, 2023, when *Touvron et al.* published their paper (see the *References* section) *Llama 2: Open Foundation and Fine-Tuned Chat Models*. The same day, Hugging Face announced that they had worked with Meta and that access to Llama was available on their platform.

Llama 2 can be applied to translations, chats, question-answering, summarization, creative writing, and more.

At the time of the writing of this book, we were redirected to the Meta Llama website to gain access:

<https://ai.meta.com/resources/models-and-libraries/llama-downloads/>.

Meta defines Llama 2 as follows:

*Llama 2 means the foundational large language models and software and algorithms, including machine-learning model code, trained model weights, inference-enabling code, training-enabling code, fine-tuning enabling code, and other elements of the foregoing distributed by Meta.*

Llama 2's architecture can be summed up in one sentence in *Section 2.2*.

*Training Details* of Meta's paper:

*We adopt most of the pretraining setting and model architecture from Llama 1. We use the standard transformer architecture (Vaswani et al., 2017).*

Source: Touvron et al. (2023).

Review *Chapter 2, Getting Started with the Architecture of the Transformer Model*, if necessary.

Like other transformer architectures, Meta implemented normalization, positional encoding, and other intermediate functions of the Original Transformer through an empirical trial-and-error process that produced good performances.

*Section 2.2. Training Details* of Meta's paper points out the two main differences from Llama 1, which can be summed up as follows:

- **Context length was increased from 2048 to 4096 tokens**, which is a key feature for transformer models so that the attention layers can attend to long-term dependencies in a text. In natural language, sometimes, the meaning of a sentence only becomes clear in longer contexts.
- **Grouped-query attention (GQA). Multi-Head Attention (MHA)** divides the attention calculation into multiple heads. Each head attends to a different part of the sequence, and the results of the heads are then combined to produce a single attention weight. However, this can be expensive in terms of memory because the **Keys (K)** and **Values (V)** need to be stored in memory for each head.

One way to optimize memory is GQA, as chosen by Meta. GQA reduces the memory requirements for the KV cache by sharing the key and value vectors across multiple heads.

As the LLM market, and specifically the demand for Foundation Models, grows, we will see top research labs release new LLMs regularly. In each case, we simply need to focus on the core architecture and then run and test it for our needs.

Hugging Face did an excellent job by making Llama 2 available on their platform the day it was released by Meta. Let's implement an example to get started.

## Implementing Llama 2 with Hugging Face

It is recommended to go to the Hugging Face Meta Llama access page first:  
<https://huggingface.co/meta-llama>.

Make sure you choose the access that is best for your project.

This section continues to implement `Automated_Design.ipynb` in this chapter's directory of the GitHub repository.

One method to automate prompts is to break it down into a Cloze template, which consists of filling the gaps (blanks, missing words) in a text.

The static part of the instruction becomes a Cloze template:

```
template='Create a prompt for a text to image model to generat
```

The dynamic part can be extracted from any type of text document with NLP: keywords, essential concepts, and any function required. This can be achieved with an LLM, as we have seen throughout the book, to obtain a target concept such as:

```
target=generate a ship on a highway in a desert\n
```

The target is then added to the template:

```
Seed LLaMA 2 prompt=template + target
```

This section continues to implement `Automated_Design.ipynb` in this chapter's directory of the GitHub repository.

Now, with this template and target seed prompt, we can run Llama 2 to obtain text-to-image prompts.

We first install `transformers`:

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

We can log in to Hugging Face's Hub directly in Google Colab:

```
!huggingface-cli login
```

Hugging Face will prompt you to enter your Hugging Face token:

```
To login, `huggingface_hub` requires a token generated from
Token:
Add token as git credential? (Y/n) n
Token is valid (permission: write).
Your token has been saved to /root/.cache/huggingface/token
Login successful
```

We now install `accelerate` to run the library faster:

```
!pip install accelerate -qq
```

Make sure to activate a GPU for this notebook. `accelerate` is a library that makes it easy to run PyTorch models on multiple GPUs, TPUs, and CPUs. It also supports mixed-precision processing and speeds up runtimes.

We import the modules we need:

```
from transformers import AutoTokenizer
import transformers
import torch
```

Now, we define Meta's Llama 2 LLM as the model for text generation, which will be prompts in our case:

```
model = "meta-llama/LLaMA-2-7b-chat-hf"
tokenizer = AutoTokenizer.from_pretrained(model)
pipeline = transformers.pipeline(
    "text-generation",
    model=model,
    torch_dtype=torch.float16,
    device_map="auto",
)
```

The program now defines a function to make a request and return the response:

```
def LLaMA2(prompt):
    sequences = pipeline(
        prompt,
        do_sample=True,
        top_k=10,
        num_return_sequences=1,
        eos_token_id=tokenizer.eos_token_id,
        max_length=200,
    )
    return sequences
```

The prompt will be the template + target prompt generated in the pipeline:

```
prompt='Create 5 prompts for a text to image model to generate
response=LLaMA2(prompt)
for seq in response:
    print(f"Output: {seq['generated_text']}")
```

The output is exactly what we need:

```
Output: Create 5 prompts for a text to image model to generate
1. A massive cargo ship driving down a desert highway, with san
2. A small speedboat racing across a sandy beach on the highway
3. A luxury cruise ship sailing through the desert, passing by
4. A historic pirate ship crashed on the highway, surrounded by
5. A group of spacecraft traveling across a desolate desert hig
```

The program might produce different responses for each run because of LLMs' stochastic nature, which explains their creativity. Llama 2 returns a `json` object. You can choose to let the process always be automatic and use the response to create a list of prompts in `sequences`:

```
# Run the function and capture the response
response = LLaMA2(prompt)
# Inspect the response object
print("Response type:", type(response))
print("Response content:", response)
# Assuming response is a list or similar iterable
if isinstance(response, list) and len(response) > 0:
    if 'generated_text' in response[0]:
        sequences = [{ 'generated_text': response[0][ 'generated_text' ]}]
    else:
        print("generated_text not in response[0]")
else:
    print("Response is not list-like or is empty")
# Extracting the 'generated_text' content
text_content = sequences[0][ 'generated_text' ]
# Splitting the text based on the newline character
lines = text_content.split( '\n' )
# Filtering out the lines that start with the number indicator.
prompts = [line for line in lines if line.startswith(( '1. ', '2. '))]
# Printing the prompts
for prompt in prompts:
    print(prompt)
```

The output will be similar but may change between runs, as shown in this excerpt:

```
Response content: [{}{'generated_text': 'Create 5 prompts for a t  
1. A sleek, futuristic ship cruises down a desert highway, its  
2. A vintage muscle car is transformed into a high-tech spaceship  
3. A massive, armored ship rumbles down the highway, its treads  
4. A small, agile ship darts through the desert landscape, its  
5. A colossal, alien spacecraft lands on the highway, its stran
```

You may want to let the automation run uncontrolled up to this point but save the previous output. If you choose to write a function to save the output in a file and load it in `sequences`, your code can continue as follows:

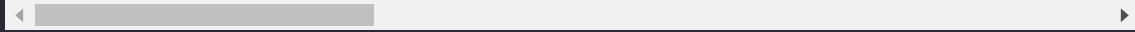
```
# The given JSON output  
sequences = [{}{'generated_text': 'Create 5 prompts for a image
```

The program now extracts the prompts Llama 2 generated:

```
# Extracting the 'generated_text' content  
text_content = sequences[0]['generated_text']  
# Splitting the text based on the newline character  
lines = text_content.split('\n')  
# Filtering out the lines that start with the number indicator.  
prompts = [line for line in lines if line.startswith('1. ')]  
# Printing the prompts  
for prompt in prompts:  
    print(prompt)
```

The output contains a clean list of the prompts:

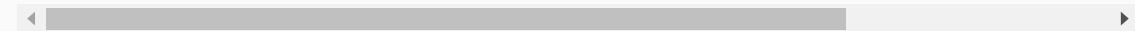
```
1. "A long, sleek ship stands majestically on a vast desert h  
2. "In the distance, a solitary ship glides smoothly down the h  
3. "A ship materializes seemingly out of thin air on the empty  
.../...
```



Note: The prompts are generated with a stochastic algorithm and thus may vary from one run to another.

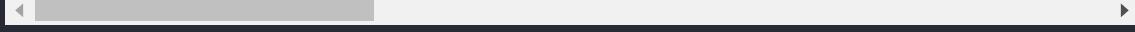
A specific prompt can be found with a random function or by simply selecting the first one:

```
# Choose a specific prompt by its index (for example, selecting  
index = 1 # 0 for the first prompt, 1 for the second, and so on)  
if 0 <= index < len(prompts):  
    print(prompts[index])  
else:  
    print("Invalid index.")
```



The program has now generated a prompt with no human intervention:

```
2. "In the distance, a solitary ship glides smoothly down the h
```



This section illustrated that:

- A pipeline LLM can automatically parse documents to generate prompts without human intervention. In this section, we imagined a function that would create the prompt for Llama 2.

- Llama 2 can automate ideation by inventing prompts for a text-to-image process.

We will now apply the output of Llama 2 to the input of Midjourney.

## Midjourney

In this section, we will create images with Midjourney using an automated prompt. The prompt was not directly written in Discord for API permission reasons. Also, you can simply read this section and visualize the process in the chapter notebook: `Automated_Design.ipynb`.

Or, you can reproduce the process in this chapter with the following recommendations (check Midjourney's pricing policy before running image generations):

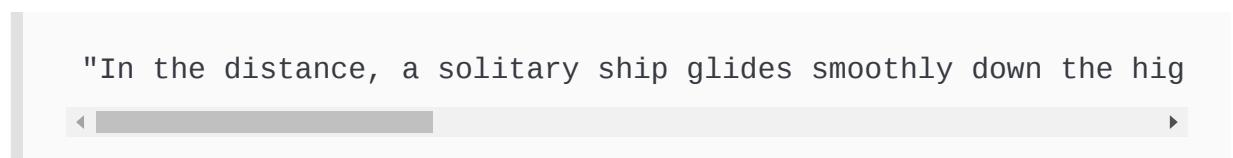
- Create a Midjourney account:  
<https://docs.midjourney.com/docs/quick-start>.
- Create a Discord account:  
<https://support.discord.com/hc/en-us/articles/360033931551-Getting-Started>.
- Invite Midjourney to your personal Discord to avoid the traffic of Midjourney's channel when many users create images:  
<https://docs.midjourney.com/docs/invite-the-bot>.
- Create an image with a Midjourney prompt in Discord beginning with `/imagine`.
- Retrieve the image interactively.
- Retrieve your images with a Discord API, as implemented in this section. You will have to follow the instructions and policy of Discord

before running the API:

<https://discord.com/developers/docs/reference>.

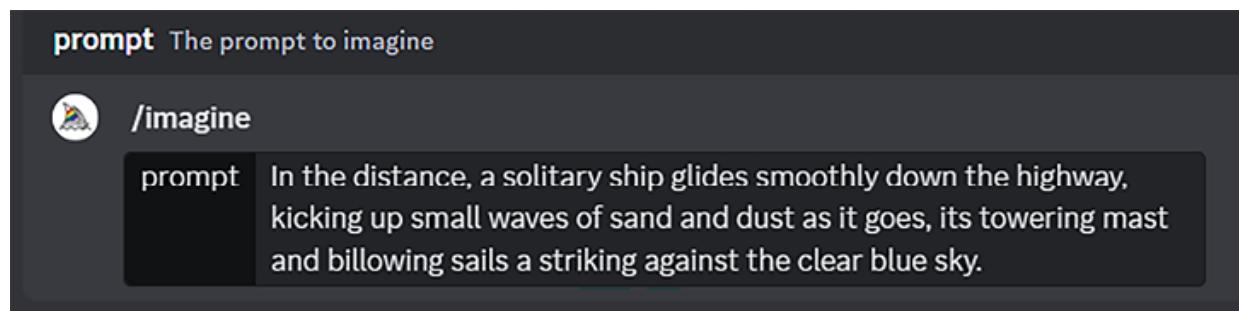
Once you have created your Midjourney and Discord accounts, you can produce an image by first entering `/imagine`, and `prompt` will appear automatically.

Then enter the prompt generated by Llama 2 that is in the Midjourney section of the notebook:



```
"In the distance, a solitary ship glides smoothly down the hig
```

The request should look like the screenshot in *Figure 20.3*:



*Figure 20.3: Create an image generation request on Midjourney*

You can save the image in the location of your choice. For this example, the image was saved on GitHub:



```
from IPython.display import Image      #This is used for render.
import requests
url = 'https://raw.githubusercontent.com/Denis2054/Transformer'
response = requests.get(url, stream=True)
with open('Ship_gliding.png', 'wb') as f:
    for chunk in response.iter_content(chunk_size=1024):
        f.write(chunk)
```

The Midjourney image is displayed:

```
from PIL import Image
# Define the path of your image
image_path = "/content/Ship_gliding.png"
# Open the image
image = Image.open(image_path)
image
```

The output illustrates the prompt generated by Llama 2:



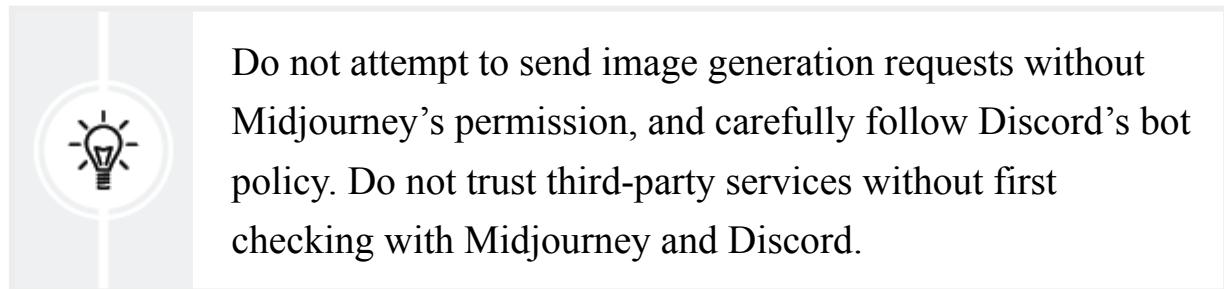
Figure 20.4: An image of a ship created by Midjourney

We have chained Llama 2 to Midjourney interactively. In *Part III* of this chapter, we will chain automated Llama 2 prompts to Stable Diffusion.

We can also use a Discord API to interact with our Midjourney images.

## Discord API for Midjourney

In this section, we will implement a Discord API to retrieve the history of the images we created in Midjourney to build a gallery. Limit image retrievals to images on your personal server. Do not retrieve images you did not personally create.



Do not attempt to send image generation requests without Midjourney's permission, and carefully follow Discord's bot policy. Do not trust third-party services without first checking with Midjourney and Discord.

The prompts can be generated by Llama 2, as we just went through in the previous section, or other LLMs with the proper instructions:

```
"In the distance, a solitary ship glides smoothly down the highway,  
kicking up small waves of sand and dust as it goes, its towering  
mast and billowing sails a striking against the clear blue sky."
```

We can take the automated prompt generation further by using the complete prompt:

```
Complete_prompt= "In the distance, a solitary ship glides smoothly  
down the highway"
```

We can send the template with no target:

```
Template= "In the distance, a solitary ship glides____"
```

We can also send targets one after the other without the template:

```
"targets" = "in the Arizona desert," "in space between the Moon and  
Mars"
```

Before implementing a Discord API, carefully follow the steps in Discord's documentation:

<https://discord.com/developers/docs/reference>.

Open `Midjourney_bot.ipynb` in the chapter's directory of the GitHub repository. The program first installs the Discord library and Pillow (a Python imaging library):

```
!pip install discord.py python-dotenv Pillow requests
```

You store your Discord API token in a file and retrieve it:

```
#Discord token  
#Store you key in a file and read it(you can type it directly).  
from google.colab import drive  
drive.mount('/content/drive')  
f = open("drive/MyDrive/files/midjourney.txt", "r")  
discord_token=f.readline()  
f.close()
```

You can also type it directly in the code (not recommended) when you import the modules:

```
import discord  
from discord.ext import commands  
import requests  
from dotenv import load_dotenv  
from PIL import Image
```

```
import os
#discord_token = [YOUR_TOKEN]
```

Now you can run the Python code cell that:

- Connects to your Discord Midjourney server.
- Listens to your activity.
- Automatically downloads images you create during the session and splits the images.
- Contains a download quantity variable to control the number of images to download.

The images are stored locally in a `content/output` directory.

Click on the cell to run it and also click to stop the process:

```
load_dotenv()
client = commands.Bot(command_prefix="*", intents=discord.Intents.all())
directory = os.getcwd()
print(directory)
def split_image(image_file):
    with Image.open(image_file) as im:
        # Get the width and height of the original image
        width, height = im.size
        # Calculate the middle points along the horizontal and
        mid_x = width // 2
        mid_y = height // 2
        # Split the image into four equal parts
        top_left = im.crop((0, 0, mid_x, mid_y))
        top_right = im.crop((mid_x, 0, width, mid_y))
        bottom_left = im.crop((0, mid_y, mid_x, height))
        bottom_right = im.crop((mid_x, mid_y, width, height))
    return top_left, top_right, bottom_left, bottom_right
...
```

For more information, see *Discord API tutorial for Midjourney* in the *References* section of this chapter.

The following cell in the notebook displays the downloaded Midjourney images:

```
from IPython.display import Image, display
import os
directory = '/content/output/'
for filename in os.listdir(directory):
    if filename.endswith(".jpg"):
        display(Image(filename=os.path.join(directory, filename)))
```

The output shows your creations:



Figure 20.5: Image created by Midjourney

You can zip the downloaded images for future use:

```
import zipfile
from IPython.display import FileLink
# Zip the images
zip_name = '/content/images.zip'
with zipfile.ZipFile(zip_name, 'w') as zipf:
    for filename in os.listdir(directory):
        if filename.endswith(".jpg"):
            zipf.write(os.path.join(directory, filename), file
```

You can save the ZIP file in the location of your choice. In this example, the ZIP file was saved on GitHub.

In another session, we don't have to rerun the Discord API to view our creations.

We can download the ZIP from our storage location:

```
!curl -L https://raw.githubusercontent.com/Denis2054/Transformers-Image-Zipper/main/images.zip
```

Then, we can write an excellent HTML function that reads the files and displays them in a nice HMTL gallery:

```
from IPython.core.display import display, HTML
import os
import base64
# Path to the folder containing the unzipped images
image_folder = '/content/images/'
# List the files in the directory. Ensure they are image files
image_files = [f for f in os.listdir(image_folder) if os.path.isfile(f)]
html_string = '<div style="display: flex; flex-wrap: wrap;">'
for image_file in image_files:
```

```



```

The gallery will display the images in the following way:



*Figure 20.6: Image gallery*

Chaining Llama 2 to Midjourney was an exciting experience.

Now, we will create a business flyer with Microsoft Designer.

## Microsoft Designer

In this section, Microsoft Designer will perform an image-to-image task with an output of Midjourney. Microsoft Designer can generate images and videos: <https://designer.microsoft.com/>.

To sum up the process of this section up to now, we:

- Asked Llama 2 to generate prompts.
- Chained the output of Llama 2 to ask Midjourney to process the prompts provided by Llama 2. The output contained images.

We will now feed the output of Midjourney to Microsoft Designer to create a flyer, as shown in *Figure 20.7*:

Describe the design you'd like to create ⓘ

Create a corporate flyer



Add image

Generate image ⓘ

Generate

*Figure 20.7: Creating a flyer with Microsoft Designer*

The output was saved and uploaded to GitHub. You can view the flyer in the Microsoft Designer section of `Automated_Design.ipynb`, in the chapter directory of the GitHub repository.

The notebook downloads the flyer created by Microsoft Designer:

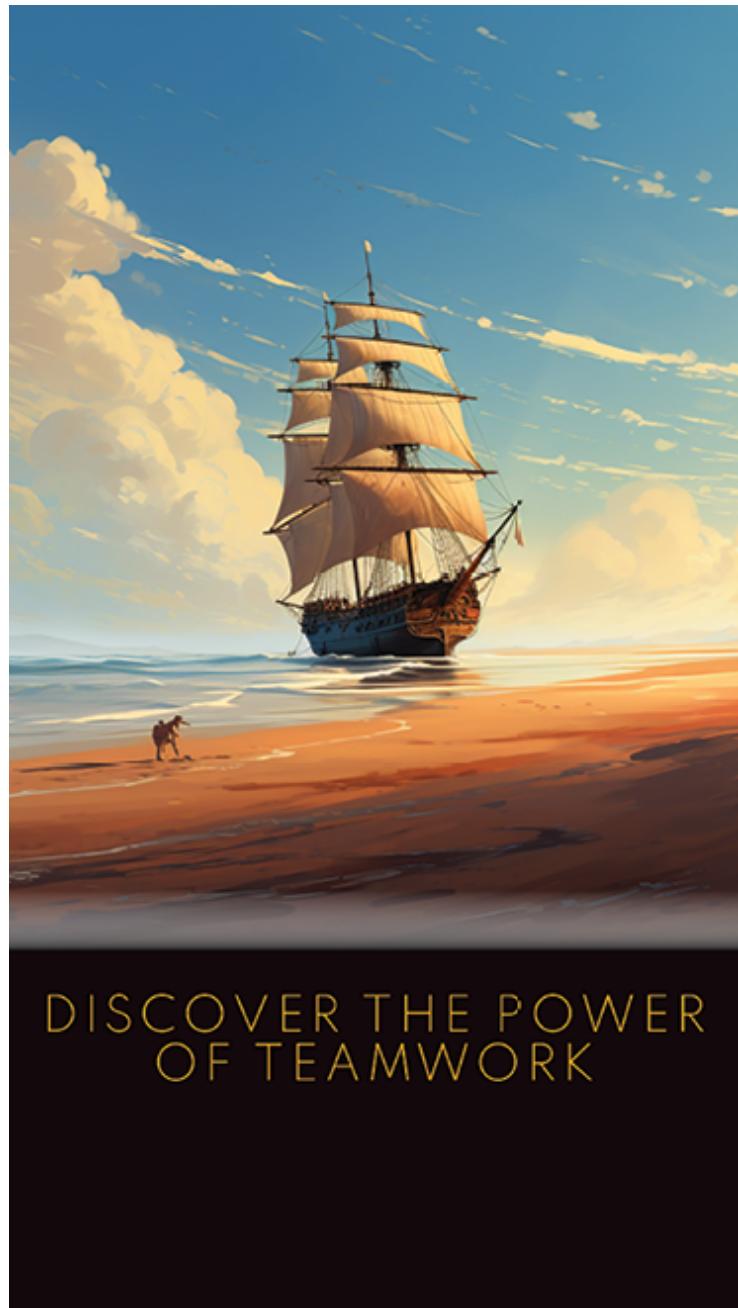
```
import requests  
url = 'https://raw.githubusercontent.com/Denis2054/Transformer'
```

```
headers = {
    'Authorization': 'token ' + github_token
}
response = requests.get(url, headers=headers, stream=True)
with open('Designer.png', 'wb') as f:
    for chunk in response.iter_content(chunk_size=1024):
        f.write(chunk)
```

Then, the program displays the image-to-image generation:

```
from PIL import Image
# Define the path of your image
image_path = "/content/Designer.png"
# Open the image
image = Image.open(image_path)
image
```

The output is a well-designed flyer:



*Figure 20.8: A flyer about teamwork*

Microsoft Designer also generated an image-to-video that was saved and uploaded to GitHub. Let's download it:

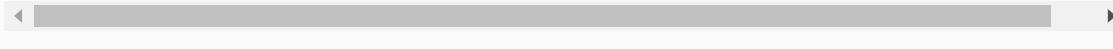
```
import subprocess
subprocess.run(['pip', 'install', 'moviepy', '-qq'], check=True)
```

```
import requests
url = 'https://raw.githubusercontent.com/Denis2054/Transformer'
response = requests.get(url, stream=True)
with open('Designer.mp4', 'wb') as f:
    for chunk in response.iter_content(chunk_size=1024):
        f.write(chunk)
```



The notebook now displays the video:

```
from moviepy.editor import *
# Load myHolidays.mp4 and select the subclip 00:00:00 - 00:00:04
clip = VideoFileClip("Designer.mp4").subclip(0,4)
clip = clip.loop(5)
clip.ipython_display(width=900,height=500)
```



Clicking the play button will start the video, which will run five times in a loop:



Figure 20.9: The video created by Microsoft Designer

We chained an automated prompt design with Llama 2 to Midjourney that created images. We then submitted one of Midjourney's images to Microsoft Designer for an image-to-image and image-to-video task, as shown in *Figure 20.10*:

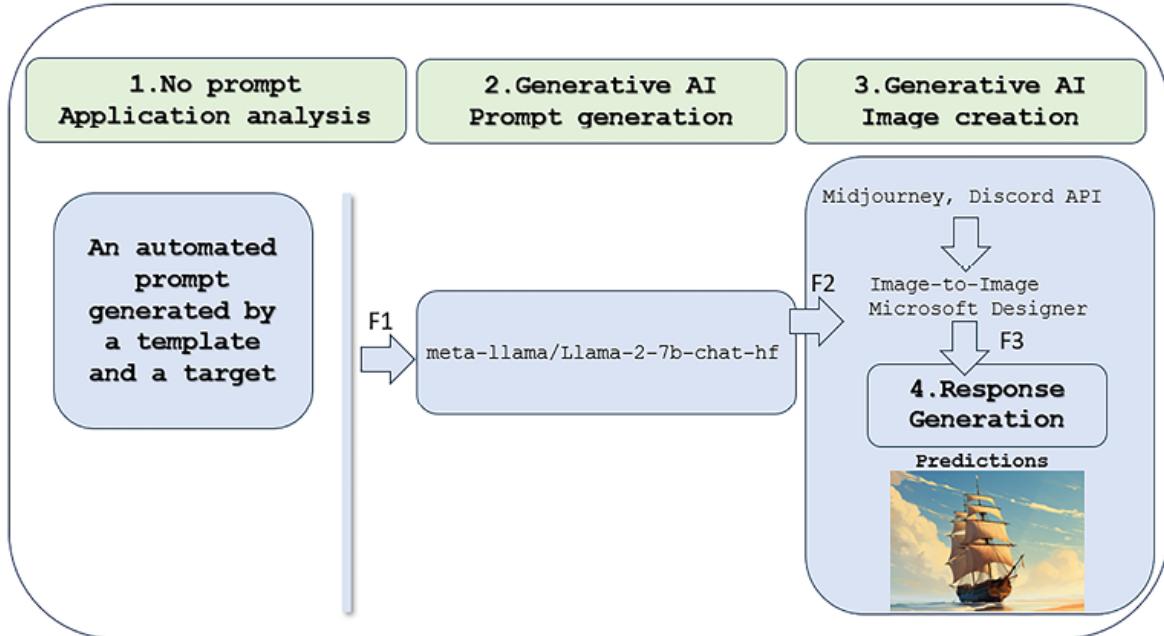


Figure 20.10: Proof of concept of automated ideation with Llama 2, Midjourney, and Microsoft Designer

Our proof of concept was successful. We demonstrated that prompts and ideation could be automated.

To fully complete the demonstration, we need to automate *F1* and *F2* and replace *F3*:

- *F1* is the output of the automated prompt design we simulated in this section. In the next section, we will automate the process.
- *F2* represents the automated prompts generated by Llama 2 that we copied and pasted into the code. In the following section, we will automate this function as well.
- *F3* represents the image-to-image process from Midjourney to Microsoft Designer in this section. In the next section, we will replace these components with Stable Diffusion to produce an image without human intervention.

This section proved that fully automated ideation was possible. We will further automate the process.

# Part III: Automated generative ideation with Stable Diffusion

This section demonstrates how to automate the process of instructing a Large Language Model to generate prompts automatically without human intervention, as shown in *Figure 20.11*. We will illustrate the process with the example of a small business that doesn't have marketing resources but would like to generate images for posters.

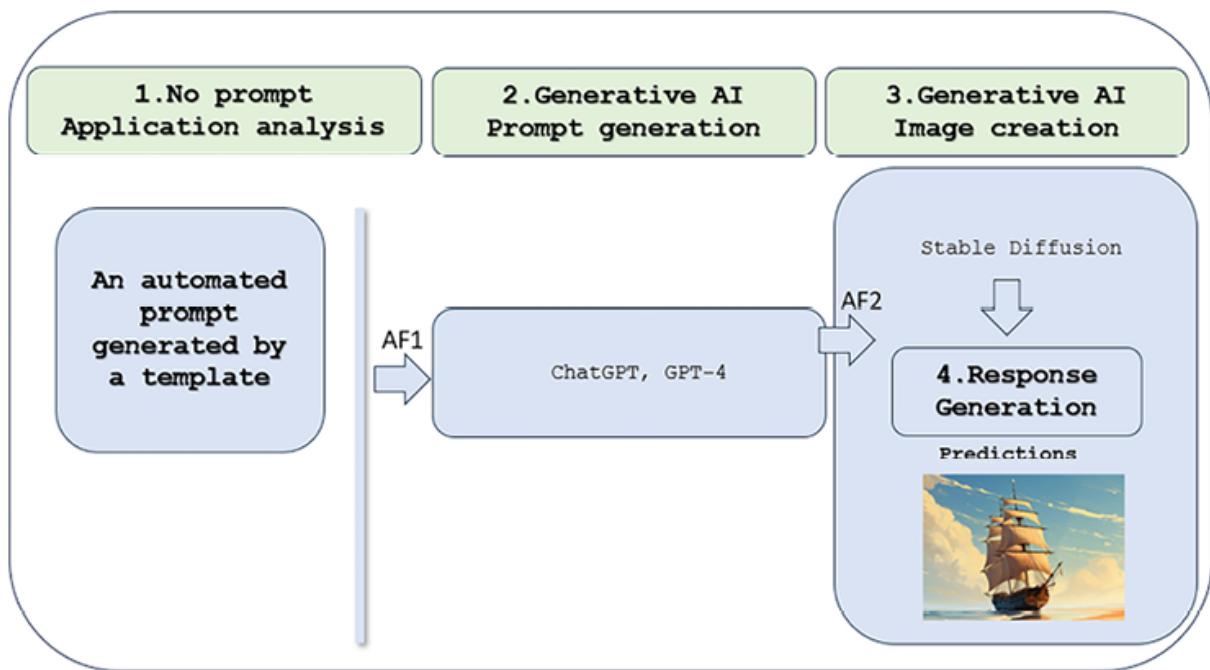


Figure 20.11: A fully automated Generative AI ideation process

*Figure 20.11* shows the fully automated process:

1. **No prompt**: The prompt will not be interactive. The program will create the prompt automatically. As such, *AF1* is the function that takes the output of the automated instruction and sends it to an LLM as input to generate prompts automatically.

2. **Generative AI (prompt generation)**: ChatGPT, GPT-4, will generate text-to-image prompts automatically. The *AF2* function will automatically chain GPT-4 to Stable Diffusion without human intervention.
3. **Generative AI (image)**: Stable Diffusion will process the prompts.
4. **Response generation**: The images generated by Stable Diffusion will be displayed for the end user to view.

This section goes through each step. We will begin by automating human-designed instructions for GPT-4.

Open `Automated_Ideation.ipynb` in the chapter directory of the repository. Activate a GPU to run this notebook for Stable Diffusion.

## 1. No prompt: Automated instruction for GPT-4

This section illustrates an automated ideation pipeline to produce Stable Diffusion images based on ideas generated by GPT-4, which relies on the documents selected for the process.

We will implement a document in the chapter notebook that could come from any source: emails, the web, knowledge base, PDFs, or any other source of information. Classical administration tools can be used to store documents.

The following code first downloads a text on ecology from GitHub to analyze it and automatically create prompts for Stable Diffusion:

```
!curl -L https://raw.githubusercontent.com/Denis2054/Transformers/master/transformers/pipelines/text_to_image/prompt_generator.py
```

We open the document and display the content on an HTML page:

```
# Read the content of the file
with open("ecodocument.txt", "r") as f:
    content = f.read()
# Display content in HTML format
from IPython.core.display import display, HTML
display(HTML(f"""
<html>
<head>
<title>Sustainable Fashion</title>
<style>
body {{ font-family: Arial, sans-serif; }}
    h1 {{ color: #2E8B57; }}
    p {{ text-align: justify; }}
</style>
</head>
<body>
<h1>Why Sustainable Fashion is Our Secret Weapon Against Climate Change</h1>
<p>{content}</p>
</body>
</html>
"""))
```

The output is an HTML page that contains the content, as shown in the excerpt in *Figure 20.12*:

## Why Sustainable Fashion is Our Secret Weapon Against Climate Change

Why Sustainable Fashion is Our Secret Weapon Against Climate Change Imagine a world where every clothing choice you make becomes a statement, not just of style, but of stewardship for our planet. Sounds empowering, right? That's because sustainable fashion isn't just a trend; it's a revolution. And it's one of the most potent tools we have to combat climate change. Fashion's Carbon Footprint : The

*Figure 20.12: Displaying the context document on an HTML page*

The notebook contains a general-purpose static instruction for GPT-4 to which we add the content:

```
# Create a variable to store the text and add the content
input_text = "Summarize the following text in 5 lines from 1 to 5"
print(input_text) # for support
```

The process is content-driven. The content drives the pipeline. The rest is static.

The static template and the content become the dynamic prompt:

```
Summarize the following text in 5 lines from 1 to 5 that can be used
as text-to-image prompts: Why Sustainable Fashion is Our Secret
Weapon Against Climate Change

Imagine a world where every clothing choice you make becomes a
statement,...
```

We can also define a static general-purpose GPT-4 function that contains the information required to obtain prompts for Stable Diffusion:

```
def dialog(uinput):
    #preparing the prompt for OpenAI
    role="user"
    line = {"role": role, "content": uinput}
    #creating the message
    assert1={"role": "system", "content": "You summarize a text in 5 lines from 1 to 5."}
    assert2={"role": "assistant", "content": "Summarize the following text in 5 lines from 1 to 5."}
    assert3=line
    iprompt = []
    iprompt.append(assert1)
    iprompt.append(assert2)
    iprompt.append(assert3)
    client = OpenAI()
```

```
#sending the message to ChatGPT
response=openai.ChatCompletion.create(model="gpt-4",message=
    text=response["choices"][0].message.content # property of the response
return text
```

Our automated prompt is ready. Let's put ChatGPT to work.

## 2. Generative AI (prompt generation) using ChatGPT with GPT-4

In this section, GPT-4 will be chained to the output of the preparation process of the automated pipeline. First, we install OpenAI and retrieve the OpenAI API key:

```
#Importing openai
try:
    import openai
except:
    !pip install openai
    import openai
#The OpenAI Key
f = open("drive/MyDrive/files/api_key.txt", "r")
API_KEY=f.readline()
f.close()
import os
os.environ['OPENAI_API_KEY'] =API_KEY
openai.api_key = os.getenv("OPENAI_API_KEY")
```

You can store your key wherever you wish. It is recommended not to display it in notebooks that others can see.

We just need to send the prepared prompt, `input_text`, to GPT-4:

```
uinput=input_text  
text=dialog(uinput) #preparing the messages for ChatGPT and ma
```

We don't need to display GPT-4's response. However, for support reasons, it may be useful to do so if there is an issue:

```
text
```

The response should contain the prompts we are looking for:

```
1. "Envision a vibrant world where fashion supports the vital c
```

GPT-4 is a stochastic algorithm and will not produce the same output each time. You may have to:

- Run the request more than once with a hard-coded control function to verify if the output complies with the specifications of the project.
- Improve the automated prompt phase.



In a real-life project, the development phase requires significant resources.

GPT-4's response will now be processed by splitting it into sentences that will become the prompts for Stable Diffusion's text-to-image task:

```
# Splitting the string by \n to get a list of sentences  
sentences = text.split("\n")
```

```
# Splitting the string by \n to get a list of sentences
sentences = [s.strip() for s in text.split("\n") if s]
# Removing the index and extra quotes from each sentence
cleaned_sentences = []
for sentence in sentences:
    parts = sentence.split('.')
    if len(parts) > 1:
        cleaned_sentences.append(parts[1][:-1])
```

The notebook saves the prompts for further use and quality control purposes in case there is a support issue to solve:

```
# Writing the cleaned sentences to the file "image2text.txt"
with open("image2text.txt", "w") as file:
    for sentence in cleaned_sentences:
        file.write(sentence + '\n')
```

You can view the content of the file generated:

```
with open("image2text.txt", "r") as file:
    content = file.read()
    print(content)
```

The output will be the prompts:

```
"Imagine a world where fashion choices become statements of ear
"The sustainable sartorial revolution, slashing the fashion in
"Fashion to last: a modern movement reducing waste, recycling
"Wardrobe prowess preserving ecosystems, advocating ethical so
"More than just style: ethical fashion empowering communities
```

The prompts may vary from one run to another, but the output should be similar.

You can save the file in the location of your project. In this case, the file is saved in Google Drive. You will need a free Google account if you choose this option:

```
!cp image2text.txt "drive/MyDrive/files/image2text.txt"
```

We can now chain GPT-4 to Stable Diffusion.

## 3. and 4. Generative AI with Stable Diffusion and displaying images

Stable Diffusion will now read the output of GPT-4 and generate the ideation images.

We first install the `diffusers` and `transformers` libraries:

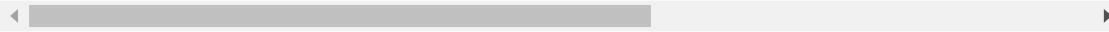
```
!pip install diffusers==0.11.1 -qq  
!pip install transformers scipy ftfy accelerate -qq
```

We now import Torch and the Stable Diffusion pipeline:

```
import torch  
from diffusers import StableDiffusionPipeline
```

The notebook implements Hugging Face's Stable Diffusion pipeline and a Stable Diffusion model:

```
pipe = StableDiffusionPipeline.from_pretrained("CompVis/stable
```



The pipe is loaded on the GPU:

```
pipe = pipe.to("cuda")
```

The prompt instructions can now be retrieved from the storage location:

```
!cp "drive/MyDrive/files/image2text.txt" image2text.txt
```

The notebook creates a text-to-image for the Stable Diffusion pipe to take an input and generate an image:

```
def text2image(prompt):
    image=pipe(prompt).images[0]
    return image
```

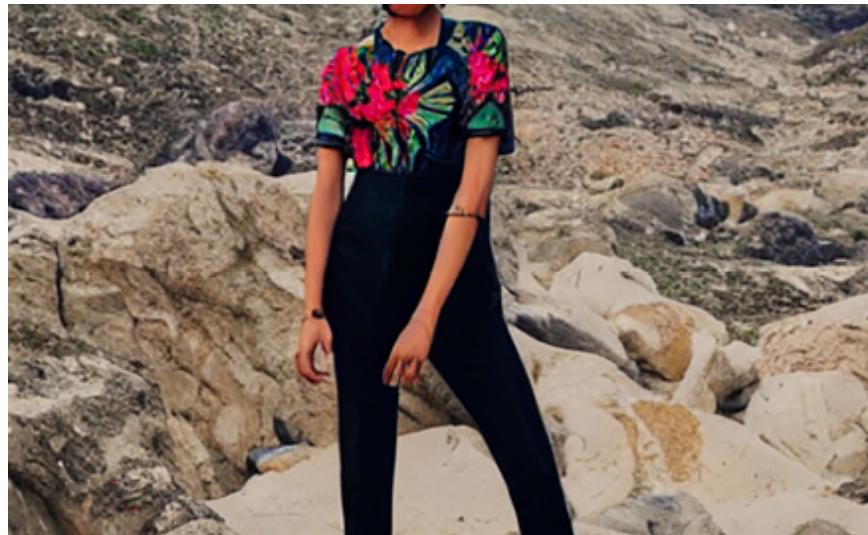
The program reads the prompt list:

```
from IPython.display import Image, display
# Path of the Ideation file containing text prompts
with open("image2text.txt", "r") as f:
    lines = f.readlines()
    line_count = len(lines)
    print("Number of lines", line_count)
```

Finally, the prompt requests are sent to the Stable Diffusion pipe and displayed in real time:

```
import re
i=0
for line in lines:
    prompt = line.strip()
    prompt = re.sub(r"\d+\.", "", prompt)
    Image = text2image(prompt)
    i+=1
print(prompt)
display(Image)
print("number of images", i)
```

The image counter can prove helpful for support reasons to control this part of the process. The images are displayed:



*Figure 20.13: An AI-generated image related to the text document*

You can choose to save the images instead of displaying them. Or, you can display the images for quality control on an HTML page with the prompt to complete the automated ideation process.

You can imagine many ways of building this automated generation ideation pipeline.

# The future is yours!

In this chapter, we implemented several methods to automate generative ideation. You can take this much further by imagining many other pipelines for each component of the automated pipeline:

## 1. No prompt, automated instructions

This chapter illustrated one way of automating human prompts.

However, each project has its constraints. Among other automated and RAG approaches, you could explore:

- *Chapter 11, Leveraging LLM Embeddings as an Alternative to Fine-Tuning.* The `Transfer_Learning_with_Ada_EMBEDDINGS.ipynb` notebook shows how to process larger documents.
- *Chapter 15, Guarding the Giants: Mitigating Risks in Large Language Models.* The `Mitigating_Generative_AI.ipynb` notebook shows how to use a knowledge base to retrieve information.

## 2. Generative AI prompts without human intervention

In this section, we automated prompt generation with ChatGPT/GPT-4. However, you could also use other LLMs, such as Llama, as shown in the *Llama 2* section of this chapter.

## 3. Generative AI images

In this section, we automated prompt generation with Stable Diffusion. However, you could also explore other approaches, such as the models we explored in *Chapter 16, Beyond Text: Vision Transformers in the Dawn of Revolutionary AI.*

## 4. Generating the response

We generated the text (prompt) and image separately in this section. These outputs could be grouped in a classical HTML page with some coding. Or, we could ask an LLM to explain the text (prompt) as a caption of the image. Or, anything you can imagine with all the tools available!

Now, let's dive into the future of development!

## The future of development through VR-AI

In this section, we're not just talking about a new coding language or a software update. We are facing a fundamental shift in how we think about development.

We are witnessing the merge of the immersive world of virtual reality with computational power and learning capabilities of Generative AI in particular.

What does that mean for a developer, for a business?

The future is emerging in which a single developer, armed with VR and AI, can achieve 25 times the productivity and production output of former developers. VR and AI are mindblowing! The future is not only about enhancing capacity but redefining potential.

## The groundbreaking shift: Parallelization of development through the fusion of VR and AI

**Massively Multiplayer Online (MMO)** games have paved the way. Developers often play internet-based video games where hundreds to thousands of players interact in a continuous virtual world, playable on computers or consoles like Xbox or PlayStation. The ability acquired by

MMO players is not linked to a particular domain, such as artificial intelligence, or a specific application. MMO players learn to play in groups and in parallel when doing several tasks at the same time: talking to other players, playing in different locations of a game, and using several combinations of buttons or actions to attain a goal. An experienced MMO player is a veteran multi-task parallelized expert!

The transition from MMO player to becoming a VR-AI pilot for virtual human agents running in parallel on a Generative AI development ecosystem is natural. Why? An MMO player knows how to use VR in MMO games to run multiple processes simultaneously while communicating with several other players simultaneously. Planning and strategy abilities are required to plan the next moves when reaching higher levels of these games. As such, a solid MMO player with an AI degree will fit the role of a VR-AI pilot and contribute to boosting software development productivity.

A VR-AI pilot is an AI expert who is both a project manager and a developer. The VR-AI pilot leverages the power of virtual reality to create several AI-driven development environments that will run in parallel automatically. The VR-AI pilot will plan, analyze, evaluate, and run virtual human agents.

A virtual human agent is an ecosystem running from a browser, for example, and connected to a platform such as OpenAI GPT or Google Cloud Vertex AI. The VR-AI pilot can launch to run the fine-tuning of a model in that instance of the browser. A virtual human agent can be a data preparation ecosystem, such as the data preparation functionality of OpenAI's fine-tuning process or a program ChatGPT can write to clean the data.

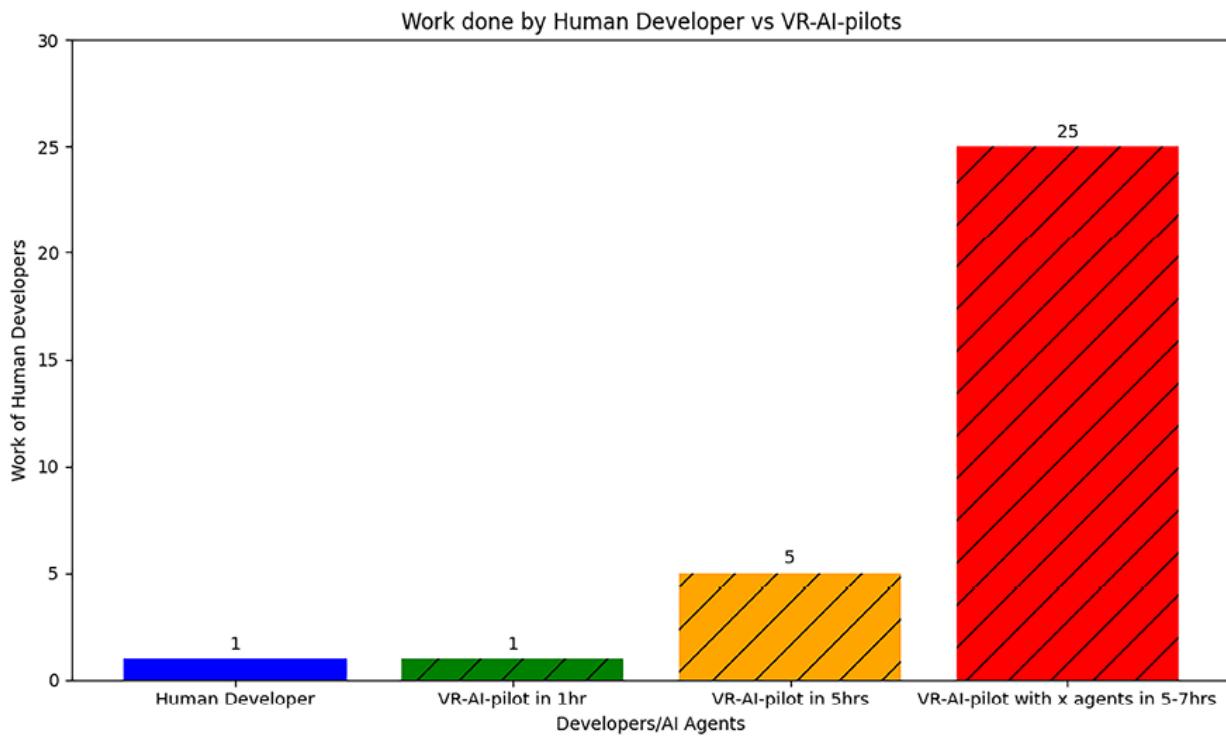
Generative AI, in this context, is the process of generating all tasks within the scope of Foundation Models, including code generation, image

generation, and text generation.

The role of the future developer will become a pilot of virtual human agents:

- One human VR-AI pilot makes an agent do a day's work of a human developer in one hour using virtual human agents. In 5 hours, that is the work of 5 human developers.
- One human VR-AI pilot running x (depending on the project) virtual human agents simultaneously can do the work of 25 human developers in a 5- to 7-hour day.

*Figure 20.14* shows a bar chart that contrasts the work done by a typical human developer against the work done by VR-AI pilots in various scenarios:



*Figure 20.14: One human-AI pilot can produce 25x what was formerly possible*

Of course, these figures are estimates and predictions of the present trend of the market. More experimentation and implementation must be carried out to confirm what seems to be an inevitable trend.



Note: The screenshots below were produced using Immersed and Pico 4 to evaluate and confirm the incredible potential. The *References* section contains links to these resources and more information on the productivity of these innovative working methods.

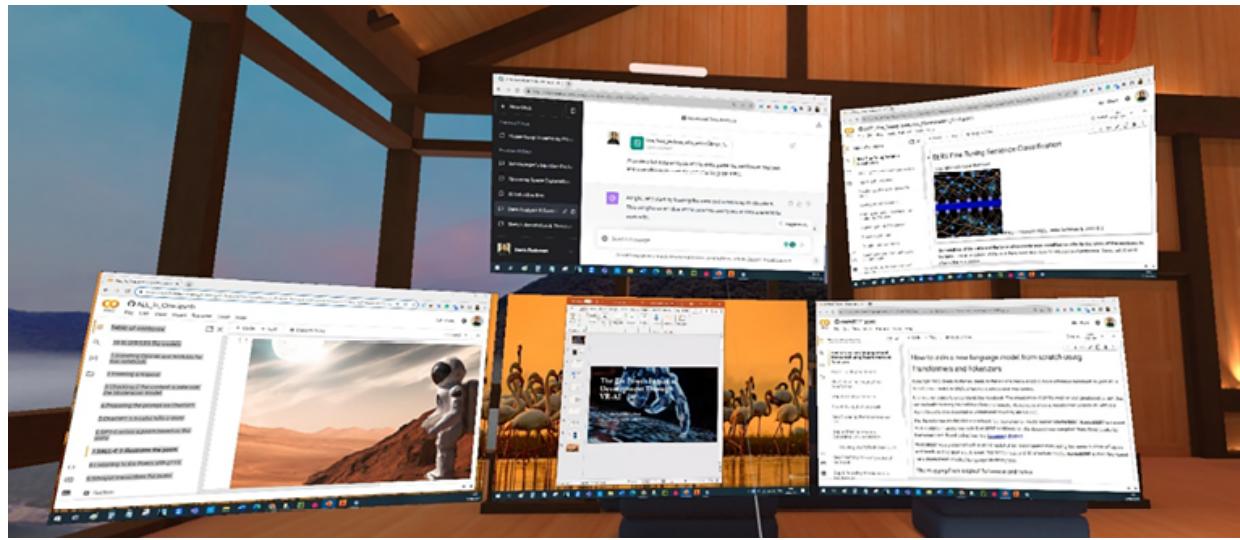
The following notebooks were used for the experiment and launched simultaneously by the VR-AI pilot on Immersed with Pico 4:

- Virtual human agent #1 fine-tuned a BERT model on a VM with this notebook in the GitHub repository:  
`Chapter05/BERT_Fine_Tuning_Sentence_Classification_GPU.ipynb`
- Virtual human agent #2 trained RoBERTa on another VM with this notebook: `Chapter06/KantaiBERT.ipynb`
- Virtual human agent #3 trained a PaLM 2 model on Google Vertex AI, as seen in *Chapter 14, Exploring Cutting-Edge LLMs with Vertex AI and PaLM 2*.
- Virtual human agent # 4 trained an OpenAI davinci-002 model through a VM, as seen in *Chapter 8, Fine-Tuning OpenAI GPT Models*.
- Virtual human agent # 5 trained several computer vision models with Hugging Face AutoTrain, as seen in *Chapter 18, Hugging Face AutoTrain: Training Vision Models without Coding*.

The workload did not exceed 30 minutes for all five virtual human agents, leaving time for the VR-AI pilot to control, plan, and run the next batches.

With Immersed, the AR-AI pilot can see all the agents at work, as shown in *Figure 20.15*.

This process can be repeated for 5 to 7 hours daily, leading to a mind-blowing increase in productivity. Of course, this approach must be confirmed in time, but the references in the *References* section show that the method is spreading:



*Figure 20.15: Running five sessions simultaneously with automated AI agents*

The AI-VR pilot can also invite customers into a virtual working and meeting space, as shown in *Figure 20.16*:



*Figure 20.16: Inviting a customer or team member into the virtual workspace*

A VR-AI-powered human pilot could potentially do a day's work and run five automatic AI agents through VR parallelization in one hour.

For example, the role of the human pilot will be to:

1. Prepare the work to be done by the virtual human agents.
2. Run the virtual agents.
3. Then share the work with the integrator pilot, which will decide how to deploy the work done by the virtual developers: deploy on servers, ask Generative AI agents to document the work, and more.

*Figure 20.17 shows one of the possible workflows between the customers and the human pilots, as illustrated in the notebook:*



*Figure 20.17: One of the many possible cutting-edge workflows*

Your imagination is the limit!

However, these opportunities are not without risks.

## Opportunities and risks

The prospects are fantastic: enhanced productivity, cost efficiency, and innovation.

The shadows are dark and worrying: job displacement, ethical concerns, security vulnerabilities, and potential loss of the human touch. This approach also requires high concentration levels, increasing the pressure on a single VR-AI pilot, whereas several other developers shared the workload before. VR-AI pilots might have to work in teams for flexibility between the workloads when more time is needed for a project, for example.

Your responsibility will be to build a sustainable future for humans by using AI as a tool, avoiding over-dependence, and respecting ethical issues.

We will now summarize the present and the future of our automation journey.

# Summary

In this chapter, we first defined generative ideation. The core concept resides in one sentence:

*AI has gone from executing tasks based on human prompts to automating ideation enhancing human thinking and imagination in an ethical ecosystem.*

Generative ideation can be constructive and implemented without destroying human jobs. For example, generative ideation can help small businesses with no marketing resources to access human-level AI services to compete with larger entities.

We then demonstrated that generative text-to-image ideation was possible with ChatGPT/GPT-4, Llama 2, Midjourney, and Microsoft Designer.

Finally, we automated the generative text-to-image ideation pipeline with automated prompt design, ChatGPT/GPT-4, and Stable Diffusion. We saw that you could use the multiple tools explored in this book to build an effective ecosystem.

We also peeked into the future of automation and AI productivity.

You can leverage automated human thinking and imagination to produce ideas through generative ideation that will not have adverse societal effects if implemented ethically. Technology has always been a two-way avenue. For example, nuclear energy can cure cancer but also destroy populations.

If you choose the constructive usage of generative ideation, you will enjoy a successful and warm career. You will enjoy learning and building AI systems in domains an entity would not have the resources to implement.

It's now up to you to write the following chapters ethically with your imagination and creativity!

# Questions

1. Generative ideation is impossible. (True/False)
2. Text-to-image technology will soon be abandoned. (True/False)
3. ChatGPT can generate prompts. (True/False)
4. Llama 2 can create text content. (True/False)
5. Midjourney is an image-to-text system. (True/False)
6. Microsoft Designer can automate ideation with the right prompt. (True/False)
7. Some companies do not have marketing resources. (True/False)
8. Stable Diffusion can help small businesses. (True/False)
9. Ethical AI can boost a career. (True/False)
10. Society can benefit from automated ideation. (True/False)

# References

- Meta's Llama 2 on Hugging Face:  
<https://huggingface.co/meta-llama>
- *Touvron et al., 2023, Llama 2: Open Foundation and Fine-Tuned Chat Models:* <https://arxiv.org/abs/2307.09288>
- Midjourney (getting started): <https://www.midjourney.com/>
- Discord (getting started):  
<https://support.discord.com/hc/en-us/articles/360033931551-Getting-Started>
- Discord API tutorial for Midjourney:  
<https://medium.com/@neonforge/how-to-create-a-discord-bot-to-download-midjourney-images->

[automatically-python-step-by-step-guide-3e76d3282871](#)

- Microsoft Designer: <https://designer.microsoft.com/>
- Immersed: <https://www.immersed.com>
- Pico 4: <https://business.picoxr.com/>

## Further reading

- Hugging Face blog on Llama:  
<https://huggingface.co/blog/llama2>

## Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://www.packt.link/Transformers>



# **Appendix: Answers to the Questions**

# Chapter 1, What Are Transformers?

1. ChatGPT is a game-changer. (True/False)

True. ChatGPT has brought transformers to the mainstream general public with its ability to solve a variety of NLP tasks.

2. ChatGPT can replace all AI algorithms. (True/False)

False. Although ChatGPT surpasses many models, it cannot replace all models for the moment.

3. AI developers will sometimes have no AI development to do.  
(True/False)

True. Sometimes, no development is required with the arrival of human-level functionality in **Large Language Models (LLMs)**, such as ChatGPT. But this can be misleading. AI development is required for many tasks beyond models such as ChatGPT. Many deep learning models, such as classification algorithms, can be trained on specific data and outperform an LLM.

4. AI developers might have to implement transformers from scratch.  
(True/False)

True and false. Although some transformers can be trained from scratch, many are now pretrained and ready to use.

5. It's not necessary to learn more than one transformer ecosystem, such as Hugging Face, for example. (True/False)

False. Hugging Face is a wonderful platform. However, the choice of a platform depends on the goals of the project.

6. A ready-to-use transformer API can satisfy all needs. (True/False)

False. A ready-to-use transformer can fulfill many requirements, but not all.

7. A company will accept the transformer ecosystem a developer knows best. (True/False)

True and false. A company might follow a developer's recommendation or decide otherwise.

8. Cloud transformers have become mainstream. (True/False)

True. In that sense, ChatGPT is disruptive.

9. A transformer project can be run on a laptop. (True/False)

True for a prototype or a limited project, for example. False for a project involving thousands of users.

10. AI specialists will have to be more flexible. (True/False)

True.

# **Chapter 2, Getting Started with the Architecture of the Transformer Model**

1. NLP transduction can encode and decode text representations.  
(True/False)

True. NLP is a form of 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)

True and false. A transformer does not contain an LSTM. However, some transformers can contain a convolution layer (CNN) to process images, for example.

5. A transformer does not contain LSTM or CNN layers. (True/False)

True and false. Transformers do not contain LSTM layers but some contain convolutional layers (CNN).

6. Attention examines all 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.

8. A transformer contains a feedforward network. (True/False)

True.

9. The masked multi-headed attention component of the decoder of a transformer prevents the algorithm parsing a given position from seeing the rest of a sequence that is being processed. (True/False)

True.

10. Transformers can analyze long-distance dependencies better than LSTMs. (True/False)

True.

# Chapter 3, Emergent vs Downstream Tasks: The Unseen Depths of Transformers

1. Machine intelligence uses the same data as humans to make predictions. (True/False)

True and false.

True because, in some cases, machine intelligence surpasses humans when processing massive amounts of data to extract meaning and perform various tasks that would take centuries for humans to process.

False because, for NLU, humans have access to more information through their senses. Machine intelligence relies on what humans provide in terms of all types of media.

2. SuperGLUE is more difficult than GLUE for NLP models.

(True/False)

True.

3. BoolQ expects a binary answer. (True/False)

True.

4. WiC stands for Words in Context. (True/False)

True.

5. **Recognizing Textual Entailment (RTE)** detects whether one sequence entails another sequence. (True/False)

True.

6. A Winograd schema predicts whether a verb is spelled correctly.  
(True/False)

False. Winograd schemas mainly apply to pronoun disambiguation.

7. Transformer models now occupy the top ranks of GLUE and SuperGLUE. (True/False)

True.

8. Human Baseline Standards are not defined once and for all. They were made tougher to attain by SuperGLUE. (True/False)

True.

9. Transformer models will never beat SuperGLUE human baseline standards. (True/False)

True and false.

False because transformer models beat human baselines for GLUE and will do the same for SuperGLUE in the future.

True because we will keep setting higher benchmark standards as we progress in the field of NLU.

10. Variants of transformer models have outperformed RNN and CNN models. (True/False)

True. But you never know what will happen in the future in AI!

# Chapter 4, Advancements in Translations with Google Trax, Google Translate, and Gemini

1. Machine translation has now exceeded human baselines. (True/False)

False. Machine translation is one of the most challenging NLP ML tasks.

2. Machine translation requires large datasets. (True/False)

True.

3. There is no need to compare transformer models using the same datasets. (True/False)

False. The only way to compare different models is to use the same datasets.

4. BLEU is the French word for blue and is the acronym of an NLP metric. (True/False)

True. **BLEU** stands for **Bilingual Evaluation Understudy Score**, making it easy to remember.

5. Smoothing techniques enhance BERT. (True/False)

True.

6. German-English is the same as English-German for machine translation. (True/False)

False. Representing German and then translating into another language is not the same process as representing English and translating into

another language. The language structures are not the same.

7. The Original Transformer multi-head attention sublayer has 2 heads.  
(True/False)

False. Each attention sublayer 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 5, Diving into Fine-Tuning through BERT

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 **Next Sentence Prediction (NSP)**. (True/False)

True.

7. BERT pretrains on mathematical functions. (True/False)

False.

8. A question-answer task is a downstream task. (True/False)

True.

9. A BERT pretraining model does not require tokenization. (True/False)

False.

10. Fine-tuning a BERT model takes less time than pretraining.

(True/False)

True.

# Chapter 6, Pretraining a Transformer from Scratch through RoBERTa

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. MLM predicts a word contained in a mask token in a sentence. (True/False)

True.

9. A BERT-like model has no self-attention sublayers. (True/False)

False. BERT has self-attention layers.

10. Data collators are helpful for backpropagation. (True/False)

True and false. Data collators optimize data processing but are not directly related to gradient descent and backpropagation. It's the indirect effect of an optimized system.

# Chapter 7, The Generative AI Revolution with ChatGPT

1. A zero-shot method trains the parameters once. (True/False)  
False. No parameters are trained.
2. Gradient updates are performed when running zero-shot models.  
(True/False)  
False.
3. GPT models only have a decoder stack. (True/False)  
True.
4. OpenAI GPT models are not GPTs. (True/False)  
False. This is a tricky question! OpenAI GPT models are **Generative Pretrained Models (GPTs)**. But they are also **General Purpose Technologies (GPTs)**.
5. The diffusion of generative transformer models is very slow in everyday applications. (True/False)  
False. ChatGPT has accelerated the diffusion of transformers in mainstream applications and usage.
6. GPT-3 models have been useless since GPT-4 was made public.  
(True/False)  
False. GPT-3 models can perform a wide range of tasks effectively. Choosing a model depends on your project and goals.
7. ChatGPT models are not completion models. (True/False)

False. ChatGPT is a generative model that can perform completion tasks.

8. RAG is a transformer model. (True/False).

False. **RAG** is **Retrieval Augmented Generation**, which is a process that enriches the inputs of a transformer model.

9. Supercomputers with 285,000 CPUs do not exist. (True/False)

False. Microsoft has provided a supercomputer for OpenAI with 285,000 CPU cores and 10,000 GPUs.

10. Supercomputers with thousands of GPUs are game-changers in AI. (True/False)

True. Thanks to this, we can build models with an increasing number of parameters and connections.

# Chapter 8, Fine-Tuning OpenAI GPT Models

1. It is useless to fine-tune an OpenAI model. (True/False)

True and false.

True because, sometimes, a pretrained standard model can be effective with good prompts.

False because, sometimes, fine-tuning is necessary to adapt a model to the tasks required.

Ultimately, fine-tuning or not remains a careful choice to make.

2. Any pretrained OpenAI model can do the task we need without fine-tuning. (True/False)

False. We first need to evaluate a model and then make a decision.

3. We don't need to prepare a dataset to fine-tune an OpenAI model. (True/False)

False. Dataset preparation is a crucial aspect of fine-tuning.

Transformer models, like most machine learning models, are data-driven.

4. We don't need one if no datasets are available on the web (follow-up question for question 3). (True/False)

False. A reliable dataset is a prerequisite to performing fine-tuning.

5. We don't need to keep track of the fine-tunes we created. (True/False)

False. We need to keep careful track of the fine-tunes we created with their metrics to guarantee the quality of our work.

6. As of January 2024, anybody can access our fine-tunes. (True/False)

False. We still need to go through a deployment process.

7. A standard model can sometimes produce similar output to a fine-tuned model. (True/False)

True. Make sure not to fine-tune a model that already works for your project.

8. GPT-4 cannot be fine-tuned. (True/False)

False. As of 2024, GPT-4 can be fine-tuned:

<https://platform.openai.com/docs/guides/fine-tuning>.

9. GPT-3 cannot be fine-tuned. (True/False)

False. As of 2024, GPT-3 can be fine-tuned.

10. We can provide raw data with no preparation for fine-tuning. (True/False)

True and false. Depending on the fine-tuning jobs, some will accept raw data while some require specific formats.

# Chapter 9, Shattering the Black Box with Interpretable Tools

1. BERTViz only shows the output of the last layer of the BERT model.  
(True/False)

False. BERTViz displays the outputs of all the layers.

2. BERTViz shows the attention heads of each layer of a BERT model.  
(True/False)

True.

3. BERTViz shows how the tokens relate to each other. (True/False)

True.

4. LIT shows the inner workings of attention heads like BERTViz.  
(True/False)

False. However, LIT makes non-probing predictions.

5. Probing is a way for an algorithm to predict language representations.  
(True/False)

True.

6. NER is a probing task. (True/False)

True.

7. PCA and UMAP are non-probing tasks. (True/False)

True.

8. LIME is model-agnostic. (True/False)

True.

9. Transformers deepen the relationships of the tokens layer by layer.  
(True/False)

True.

10. OpenAI **Large Language Models (LLMs)** can explain LLMs  
(True/False)

True to a certain extent. Explaining LLMs remains a challenging task.

# **Chapter 10, Investigating the Role of Tokenizers in Shaping Transformer Models**

1. A tokenized dictionary contains every word that exists in a language.  
(True/False)  
False.
2. Pretrained tokenizers can encode any dataset. (True/False)  
False.
3. It is good practice to check a database before using it. (True/False)  
True.
4. It is good practice to eliminate obscene data from datasets.  
(True/False)  
True.
5. It is good practice to delete data containing discriminating assertions.  
(True/False)  
True.
6. Raw datasets might sometimes produce relationships between noisy content and useful content. (True/False)  
True.
7. A standard pretrained tokenizer contains the English vocabulary of the past 700 years. (True/False)

False.

8. Old English can create problems when encoding data with a tokenizer trained in modern English. (True/False)

True.

9. Medical and other types of jargon (domain-specific language) 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 11, Leveraging LLM Embeddings as an Alternative to Fine-Tuning

1. Prompt design is the same thing as prompt engineering. (True/False)

False. Prompt engineering involves more complex preparation than prompt design.

2. OpenAI doesn't have embedding models. (True/False)

False. OpenAI has embedding models such as Ada.

3. Ada is a chat model, not an embedding model. (True/False)

False. Ada can also be used as an embedding model.

4. An Ada embedding vector contains two dimensions. (True/False)

False. An Ada embedding vector has 1,536 dimensions.

5. GPT-3.5-turbo cannot answer questions. (True/False)

False. GPT-3.5-turbo is a general-purpose model that can answer questions, among many other tasks.

6. GPT-4 has better reasoning abilities than GPT-3.5 turbo. (True/False)

True.

7. We don't need to manage cost parameters. (True/False)

False. Cost is a critical factor in a project.

8. GPT models don't have a maximum input token limit. (True/False)

False. GPT models all have a maximum input token limit.

9. A query must be embedded before being used with an embedded dataset. (True/False)

True.

10. Embeddings-based search can be an effective alternative to fine-tuning a model. (True/False)

True.

# Chapter 12, Toward Syntax-Free Semantic Role Labeling with ChatGPT and GPT-4

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 GPT-based model contains an encoder and decoder stack.  
(True/False)  
False. A GPT-based model is a decoder-only architecture.
8. A GPT-based SRL model has standard input formats. (True/False)

False. GPT models are not task-specific and don't require standardization beyond providing clear prompts.

9. Transformers can solve any SRL task. (True/False)

True and false. Transformers can perform SRL tasks quite well. However, task-specific transformers, such as a BERT-based model, for example, can be more precise in some cases. Each project has different goals and requires experimentation.

10. ChatGPT can perform SRL better than any model. (True/False)

False. We cannot assert this until we experiment and compare models. It may be true, but it has to be proven. We cannot make an affirmation without proof for a specific project.

# Chapter 13, Summarization with T5 and ChatGPT

1. T5 models only have encoder stacks like BERT models. (True/False)  
False.
2. T5 models have both encoder and decoder stacks. (True/False)  
True.
3. T5 models use relative positional encoding, not absolute positional encoding. (True/False)  
True.
4. Text-to-text models are only designed for summarization. (True/False)  
False.
5. Text-to-text models apply a prefix to the input sequence that determines the NLP task. (True/False)  
True.
6. T5 models require specific hyperparameters for each task. (True/False)  
False.
7. One of the advantages of text-to-text models is that they use the same hyperparameters for all NLP tasks. (True/False)  
True.
8. T5 transformers do not contain a feedforward network. (True/False)  
False.

9. Hugging Face is a framework that makes transformers easier to implement. (True/False)

True.

10. OpenAI's transformers are the best for summarization tasks. (True/False)

True and false.

True for summarizing texts in general. False because, in some cases, a task-specific summarizing transformer such as T5 might provide better outputs in a specialized domain.

# Chapter 14, Exploring Cutting-Edge LLMs with Vertex AI and PaLM 2

1. Pathways is not a significant game-changer. (True/False)

False. Pathways optimizes hardware usage and increases TPU, GPU, and CPU efficiency.

2. PaLM 2 uses the same embedding matrices for input and output operations. (True/False)

True. This is quite a move forward and reduces the number of computed operations.

3. PaLM models have an encoder and a decoder stack. (True/False)

False. PaLM models have a decoder-only architecture.

4. Google Workspace assistants contain Generative AI. (True/False)

True. Transformer-driven Generative AI assistants are being rolled out in many applications.

5. All transformers are implemented for Generative AI. (True/False)

False. Transformers are flexible. They can also perform specific classification tasks.

6. A generative Large Language Model cannot perform discriminative tasks. (True/False)

False. With the right prompt, a generative model can perform discriminative tasks.

7. Transformer models have reached their limits. (True/False)

False. Transformers will evolve until a possible new technology exceeds their capabilities.

8. PaLM 2 is the last Generative AI model Google will produce.

(True/False)

False. Google has continually improved its AI models.

9. Prompt engineering is an alternative to fine-tuning. (True/False)

True. This must be taken into account before fine-tuning a model.

10. Fine-tuning a transformer model does not require much dataset preparation. (True/False)

False. Data preprocessing is quite a challenging task.

# Chapter 15, Guarding the Giants: Mitigating Risks in Large Language Models

1. It's impossible to force ChatGPT to harass somebody. (True/False)

False. ChatGPT can be unknowingly led to harmful conduct. We need to be on alert.

2. Hallucinations are only for humans. (True/False)

False. AI can hallucinate as well when it generates random content unrelated to the prompt.

3. Privacy is taken seriously on the leading cloud platforms. (True/False)

True. Platforms are making significant progress under the pressure of increasing regulations and building trustworthy services. But we must always read the privacy policies.

4. APIs pose no risk. (True/False)

False. We need to verify the encryption used.

5. Harmful content can be filtered. (True/False)

True and false. We can reduce harmful content with a solid rule base and the moderation models available.

6. A moderation model is 100% reliable. (True/False)

False. We need to enforce the system with rule bases and verify the data used by models.

7. A rule base is useless when using LLMs. (True/False)

True and false. This will depend on the goals of the project.

8. A knowledge base will make the transformer ecosystem more reliable. (True/False)

True and false. A solid knowledge base can improve the quality of a system. However, this depends on each project.

9. We cannot add information to a prompt. (True/False)

False. We can add parameters. And in ChatGPT models, we can add informative messages.

10. Prompt engineering requires more effort than prompt design.

(True/False)

True. No particular machine learning knowledge is required for prompt design. However, advanced prompt engineering requires machine learning expertise, such as adding moderation tools, knowledge bases, and other functions.

# Chapter 16, Beyond Text: Vision Transformers in the Dawn of Revolutionary AI

1. DALL-E 2 classifies images. (True/False)

False.

2. ViT classifies images. (True/False)

True.

3. BERT was initially designed to generate images. (True/False)

False.

4. CLIP is an image-clipping application. (True/False)

False.

5. BERT uses CLIP to identify images. (True/False)

False.

6. DALL-E 2 cannot be accessed with an API. (True/False)

False.

7. DALL-E and GTP-4V can perform a DAT. (True/False)

True. DALL-E can produce an image with a highly divergent semantic association with a **Discriminative Association Task (DAT)**. GPT-4V can analyze the image and make a creative description.

8. ViT can classify images that are not on its list of labels. (True/False)

False.

9. ViT requires a prompt to respond. (True/False)

False.

10. The model in the playground learns the new images you submit.  
(True/False)

False.

11. GPT-4V will most probably evolve into a more multimodal system.  
(True/False)

True. OpenAI models are constantly expanding their scope to become increasingly multimodal, so it is very probable we will be seeing multimodal evolutions.

# Chapter 17, Transcending the Image-Text Boundary with Stable Diffusion

1. Stable Diffusion requires a text encoder. (True/False)  
True.
2. Stable Diffusion requires diffusion layers. (True/False)  
True.
3. Keras Stable Diffusion reduces a noisy image to a lower dimensionality. (True/False)  
True.
4. A Keras Stable Diffusion model upsamples an image once it is downsampled. (True/False)  
True.
5. The final output of a diffusion model is a “noisy” image. (True/False)  
False. The final output of a diffusion model is an upsampled, well-defined image.
6. OpenAI CLIP cannot produce a text-to-video model yet. (True/False)  
False. Hugging Face hosts a text-to-video OpenAI CLIP implementation.
7. Stability AI cannot convert one image to another in a video.  
(True/False)

False. Stability has an animation API that converts one image into another in a video.

8. Meta's TimeSformer is a scheduling algorithm, not a computer vision model. (True/False)

False. TimeSformer is a powerful video model that performs inferences.

9. It will never be possible to create a complete movie automatically. (True/False)

False. The technology is here already, and AI-driven movies will no doubt expand in the future.

10. There is a hardware limit to generate videos automatically beyond 10 seconds. (True/False)

False. The technology is ready. Long videos will expand as artists begin to implement computer vision technology.

# **Chapter 18, Hugging Face AutoTrain: Training Vision Models without Coding**

1. Hugging Face AutoTrain can train every vision transformer on the market. (True/False)

False. Although the platform provides excellent services, not all models are available to Hugging Face. For example, some Google AI and OpenAI models can only be trained on their platforms.

2. Datasets are always easy to create. (True/False)

False. Computer vision models require large datasets containing well-chosen images.

3. A no-coding AI system requires no machine learning knowledge. (True/False)

True and false.

True because Hugging Face AutoTrain can work well for some well-prepared datasets. False because, when a dataset doesn't work well, AI expertise proves necessary.

4. Hugging Face AutoTrain can classify any image submitted. (True/False)

False. Some images cannot be classified by Hugging Face models or the best models on the market.

5. Even a well-prepared dataset can be insufficient. (True/False)

True. A well-prepared dataset might not be the right one for a model or lead to unintended overfitting.

6. Creating a validation set of images to test a vision transformer is useful. (True/False)

True. A solid validation set can be used for all the vision models by standardizing testing.

7. Even a well-trained model can lead to overfitting. (True/False)

True. Unfortunately, even a well-trained model on a well-intentioned dataset can lead to overfitting. Some vision models require tremendous volumes of images.

8. It may take months to optimize a vision transformer. (True/False)

True. Many data, model architecture, hyperparameters, and hardware issues can occur when a project goes wrong. Solving these issues can last months.

9. An automated service can sometimes work well with no coding. (True/False)

True. As AI evolves, some image classification tasks, for example, have become easy with systems like Hugging Face AutoTrain.

10. An AI professional will always be necessary for complex AI issues. (True/False)

True. Humans will still be necessary in AI for quite a while!

# Chapter 19, On the Road to Functional AGI with HuggingGPT and its Peers

1. AGI already exists and is spreading everywhere. (True/False)

False. **Artificial General Intelligence (AGI)** with actual self-perception doesn't exist and might never exist. However, dreaming about it motivates us to conduct more research.

2. Functional AGI is conscious. (True/False)

False. This is a misconception.

3. Functional AGI can perform human-level tasks in a closed environment. (True/False)

True. In a closed environment, it is possible to precisely define tasks and assemble a set of AI models to solve those tasks.

4. Vision models can now identify all objects in all situations. (True/False)

False. Vision models have progressed rapidly, but much work remains to be done.

5. HuggingGPT leverages the abilities of an LLM, such as ChatGPT. (True/False)

True. This is quite an achievement!

6. ChatGPT can be a controller in the HuggingGPT ecosystem. (True/False)

True. HuggingGPT's innovative approach implemented ChatGPT to manage models to perform subtasks.

7. HuggingGPT is not a cross-platform system. (True/False)

False. Although the models ChatGPT selects are on Hugging Face, ChatGPT is hosted on OpenAI. HuggingGPT is thus a cross-platform system.

8. Chained models can improve overall vision model performances. (True/False)

True. We can build impressive pipelines with this technology.

9. Google Cloud Vision and ChatGPT cannot be chained. (True/False)

False. ChatGPT can interpret the output of Google Cloud Vision.

10. Midjourney images can become the input of Gen-2 to produce videos. (True/False)

True. The result of this association is quite impressive.

# **Chapter 20, Beyond Human-Designed Prompts with Generative Ideation**

1. Generative ideation is impossible. (True/False)

False. Generative ideation seems counterintuitive. However, the chapter's program showed the beginning of a new era.

2. Text-to-image technology will soon be abandoned. (True/False)

False. Text-to-image Generative AI is spreading to all domains.

3. ChatGPT can generate prompts. (True/False)

True. ChatGPT, as an LLM, can perform nearly any NLP task, including inventing prompts.

4. LLaMA 2 can create text content. (True/False)

True. LLaMA 2 is a productive LLM.

5. Midjourney is an image-to-text system. (True/False)

False. Midjourney is fundamentally a text-to-image system, even if it can perform image-to-text tasks in the future.

6. Microsoft Designer can automate ideation with the right prompt.  
(True/False)

True. Microsoft Designer is an excellent place to brainstorm. Using generative ideation to create prompt inputs can enhance our creativity. New ideas trigger new ideas in our imagination in a creative cycle.

7. Some companies do not have marketing resources. (True/False)

True. This means that automated processes can help them keep up with their competition.

8. Stable Diffusion can help small businesses. (True/False)

True. Stable Diffusion's ability to create images chained to an LLM can help a business that doesn't have the necessary marketing resources to compete with large companies.

9. Ethical AI can boost a career. (True/False)

True. Ethical AI will build trust with teams and customers. People will confidently approach you, and your business challenges will be exciting.

10. Society can benefit from automated ideation. (True/False)

True. Many organizations do not have the resources to spend time on ideation. Generative ideation can help many such entities find new ways of facing the complex challenges of our era.

## Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://www.packt.link/Transformers>





[packt.com](http://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
- Improve your learning 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

At [www.packt.com](http://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.

# Other Books You May Enjoy

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



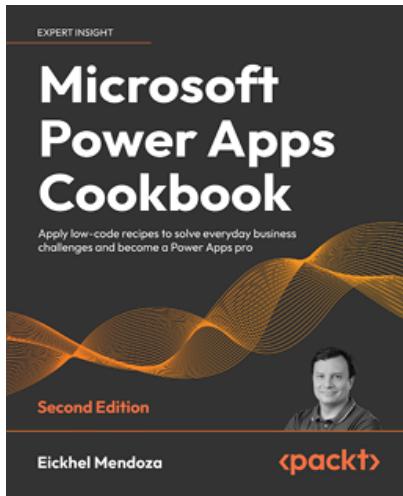
## **Learn Microsoft Power Apps – Second Edition**

Matthew Weston

ISBN: 978-1-80107-064-5

- Understand the Power Apps ecosystem and licensing
- Take your first steps building canvas apps
- Develop apps using intermediate techniques such as the barcode scanner and GPS controls
- Explore new connectors to integrate tools across the Power Platform
- Store data in Dataverse using model-driven apps

- Discover the best practices for building apps cleanly and effectively
- Use AI for app development with AI Builder and Copilot

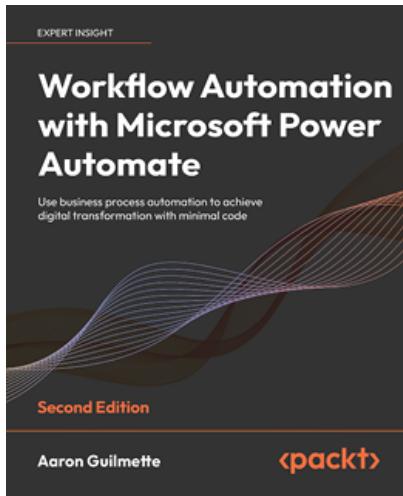


## **Microsoft Power Apps Cookbook – Second Edition**

Eickhel Mendoza

ISBN: 978-1-80323-802-9

- Learn to integrate and test canvas apps
- Design model-driven solutions using various features of Microsoft Dataverse
- Automate business processes such as triggered events, status change notifications, and approval systems with Power Automate
- Implement RPA technologies with Power Automate
- Extend your platform using maps and mixed reality
- Implement AI Builder's intelligent capabilities in your solutions
- Extend your business applications capabilities using Power Apps Component Framework
- Create website experiences for users beyond the organization with Microsoft Power Pages



## **Workflow Automation with Microsoft Power Automate – Second Edition**

Aaron Guilmette

ISBN: 978-1-80323-767-1

- Learn the basic building blocks of Power Automate capabilities
- Explore connectors in Power Automate to automate email workflows
- Discover how to make a flow for copying files between cloud services
- Configure Power Automate Desktop flows for your business needs
- Build on examples to create complex database and approval flows
- Connect common business applications like Outlook, Forms, and Teams
- Learn the introductory concepts for robotic process automation
- Discover how to use AI sentiment analysis

# Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [authors.packtpub.com](https://authors.packtpub.com) and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

## Share your thoughts

Now you've finished *Transformers for Natural Language Processing and Computer Vision - Third Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here to go straight to the Amazon review](#) page for this book and share your feedback or leave a review on the site that you purchased it from.

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

# Index

## A

**accuracy score** [82](#)

**Ada embeddings**

running [340](#)

saving, for future reuse [340](#)

used, for transfer learning [338](#)

**Adam optimizer** [145](#)

**AI agents**

conditions, for replication [455](#), [457](#)

**AI professionals**

decision-making guidelines [30](#)

domains [29](#)

resources [29](#), [30](#)

role [28](#), [29](#)

**AI revolution**

token to [17](#), [18](#), [19](#), [20](#)

**Amazon Fine Food Reviews dataset** [338](#)

**Amazon Web Services (AWS)** [1](#), [31](#), [332](#)

**argument** [353](#)

**ask functionality** [335](#), [336](#)

example question [336](#)

wrong answers, troubleshooting [336](#), [337](#)

## **associative neural networks [16](#)**

### **Asynchronous JavaScript and XML (AJAX) [118](#)**

#### **attention heads**

output, streaming of [254](#), [255](#), [256](#)

#### **attention layer [4](#), [5](#)**

computational time complexity [5](#)

#### **attention scores**

word relationships, visualizing with [256](#), [257](#), [258](#), [259](#)

### **Auto-BIG-bench [447](#), [448](#), [449](#), [450](#), [451](#), [452](#), [453](#)**

#### **automated generative ideation**

automated instruction for GPT-4 [641](#), [642](#), [643](#)

generative AI, with ChatGPT [643](#), [644](#)

generative AI, with GPT-4 [643](#), [644](#)

generative AI, with Stable Diffusion [645](#), [646](#), [647](#)

images, displaying [645](#), [646](#), [647](#)

with Stable Diffusion [640](#)

#### **automated ideation architecture [615](#)**

#### **automated instruction for GPT-4 [641](#), [642](#), [643](#)**

#### **automated pipeline**

component [647](#)

#### **automation [21](#)**

# B

**Bard** [410](#)

**BeitForImage Classification** [567](#), [568](#), [569](#)

**BEiT model** [567](#)

**benchmark tasks and datasets** [84](#)

**BERT architecture**

encoder stack [127](#), [129](#)

**BERT model**

configuration [141](#)

fine-tuning [132](#), [133](#)

interacting with [154](#), [155](#)

interface, creating for trained model [153](#)

pretraining [132](#)

saving [152](#), [153](#)

**BERT model, fine-tuning**

attention masks, creating [139](#)

batch size, selecting [140](#)

BERT tokenizer, activating [138](#)

BERT tokens, adding [138](#)

CoLA dataset, loading [136](#), [137](#), [138](#)

configuration [141](#), [142](#)

CUDA, specifying [135](#)

data, converting into torch tensors [139](#)

data, processing [138](#)

data, splitting into training and validation sets [139](#)  
goal, defining [134](#)  
hardware constraints [134](#)  
holdout dataset, used, for prediction [148](#), [149](#)  
Hugging Face BERT uncased base model, loading [142](#), [143](#)  
Hugging Face Transformers, installing [134](#), [135](#)  
hyperparameters, for training loop [145](#), [146](#)  
iterator, creating [140](#)  
label lists, creating [138](#)  
MCC, used for evaluation [151](#)  
MCC, used for evaluation, for whole dataset [151](#), [152](#)  
modules, importing [135](#)  
optimizer grouped parameters [143](#), [144](#), [145](#)  
prediction process, exploring [149](#), [150](#)  
sentences, creating [138](#)  
training evaluation [147](#), [148](#)  
training loop [146](#), [147](#)

## **BERT Pre-Training of Image Transformers (BEiT) [567](#)**

### **BERT tokenizer**

activating [138](#)

### **BertViz**

attention heads, processing and displaying [251](#), [252](#)  
for transformer visualization [248](#)  
head view [250](#)

installing [249](#)  
models, loading [249](#), [250](#)  
model view, of transformer [252](#), [253](#)  
modules, importing [249](#)  
output probabilities, of attention heads [253](#), [254](#)  
running [249](#)

## Bidirectional Encoder Representations from Transformers (BERT)

architecture [126](#)

## Big O notation [3](#)

## Bilingual Evaluation Understudy (BLEU) [69](#), [99](#)

## Bilingual Evaluation Understudy Score (BLEU)

machine translation, evaluating with [109](#)

## BoolQ [88](#)

## buckets creation [435](#), [436](#)

reference link [435](#)

## Byte Pair Encoding (BPE) [42](#)

## Byte-Pair Encoding (BPE) [161](#), [291](#), [302](#)

## Byte-Pair Encoding (BPE) tokenizer [164](#)

# C

## Center for Research on Foundation Models (CRFM) [21](#), [443](#)

## Central Processing Unit (CPU) [6](#)

## ChatGPT [74](#), [79](#)

application, creating to display IMDb reviews [212](#), [213](#)  
application, creating to display news feed [213](#), [214](#)  
application for WikiArt, creating [212](#)  
as assistants [206](#), [211](#), [212](#), [213](#), [214](#)  
features [353](#)  
function, creating [211](#)  
general-purpose prompt examples [210](#)  
GitHub Copilot code assistant [207](#), [209](#)  
Google Cloud Vision, chaining to [604](#), [605](#), [606](#)  
k-means clustering (KMC) algorithm, creating [214](#)  
source code, providing [206](#), [207](#)  
source code, writing [211](#)  
summarization with [394](#), [395](#), [397](#), [398](#)

## **ChatGPT/GPT-4 HTML presentation [617](#)**

HTML graph [619](#), [622](#)  
text [617](#), [618](#)

## **ChatGPT Plus**

image, creating with [511](#), [512](#), [513](#)  
reference link [510](#)

## **ChatGPT Plus with GPT-4 [74](#), [79](#), [81](#)**

### **chencherry smoothing [112](#)**

## **Choice of Plausible Answers (COPA) task [87](#)**

### **cloud platform**

selecting [34](#)

**clusters** [341](#), [342](#)

displaying, with t-SNE [343](#), [344](#)

finding, with k-means clustering [342](#)

**CoLA dataset**

loading [136](#), [137](#), [138](#)

**Commitment Bank (CB)** [88](#)

**computational time complexity**

of attention layer [5](#)

with Central Processing Unit (CPU) [6](#), [7](#), [8](#), [9](#)

with Graphics Processing Unit (GPU) [9](#), [10](#), [11](#)

with Tensor Processing Unit (TPU) [11](#), [12](#), [13](#)

**Computer Vision (CV)** [511](#)

**Compute Unified Device Architecture (CUDA)** [135](#), [169](#)

**constant time complexity O(1)** [3](#)

**Contrastive Language-Image Pre-Training (CLIP)** [498](#)

architecture [498](#)

in code [499](#), [500](#), [501](#), [502](#), [503](#)

**ConvNextForImageClassification** [572](#), [574](#)

**Convolutional Neural Network (CNN)** [17](#), [496](#)

**Corpus of Linguistic Acceptability (CoLA)** [93](#), [376](#)

**covariance** [271](#)

**CustomGPT** [597](#), [598](#)

**cutting-edge LLM**

properties [356](#)

## **cutting-edge platform installation**

limitations [444](#), [445](#), [446](#), [447](#)

## **cutting-edge SRL** [350](#), [352](#)

# **D**

## **DALL-E** [504](#)

architecture [503](#), [504](#)

image, creating with [511](#), [512](#), [513](#)

used, for migrating from research to mainstream AI [507](#), [508](#),  
[509](#)

## **DALL-E 2** [503](#)

image, creating [505](#), [506](#)

variation of an image, creating [506](#), [507](#)

## **DALL-E 3** [503](#)

image, creating [505](#), [506](#)

variation of an image, creating [506](#), [507](#)

## **datasets and tokenizers**

best practices [283](#)

matching [282](#)

Word2Vec tokenization [286](#), [287](#), [288](#), [289](#)

## **datasets and tokenizers, best practices**

continuous human quality control [284](#), [285](#)

preprocessing [283](#)

quality control [284](#)

## **decoder stack, Transformer architecture** [66](#)

attention layers [67](#), [68](#)

FFN sublayer [68](#)

linear layer [68](#)

output embedding [67](#)

position encoding [67](#)

post-LN [68](#)

## **Discrete Variational Autoencoder (DVAE)** [504](#)

## **Discrete Variational Encoders (dVAE)** [525](#)

## **discriminative AI** [21](#)

## **DistilBERT** [161](#)

## **Divergent Association Task (DAT)**

Generative AI Stable Diffusion [535](#), [536](#)

## **Divergent Association Tasks (DATs)** [510](#)

experimenting [514](#)

## **divergent semantic association** [510](#)

defining [510](#), [511](#)

## **downstream tasks** [82](#)

Corpus of Linguistic Acceptability (CoLA) [93](#)

Microsoft Research Paraphrase Corpus (MRPC) [94](#), [95](#)

running [92](#)

Stanford Sentiment TreeBank (SST-2) [93](#), [94](#)

Winograd schemas [95](#), [96](#)

## **Duet AI**

reference link [410](#)

## E

**eigenvalues** [271](#)

**eigenvectors** [271](#)

**embedded transformers** [28](#)

**embedding model**

implementing [328](#)

**encoder stack** [127](#), [128](#)

**encoder stack, transformer architecture**

Feedforward Network (FFN) [65](#)

**encoder stack, Transformer architecture** [40](#), [41](#)

input embedding [42](#), [43](#), [44](#)

multi-head attention [50](#)

positional encoding [45](#), [46](#), [47](#), [48](#)

positional encoding, adding to embedding vector [48](#), [49](#), [50](#)

**engines** [22](#)

**exBERT** [259](#)

URL [260](#)

**eXtensible Markup Language (XML)** [118](#)

## F

**F1-score** [82](#)

**Feedforward Network (FFN)** [65](#)

## **fine-tuned GPT model**

fine-tuned jobs and models, managing [241](#), [242](#)  
running [238](#), [239](#), [240](#)  
standard [242](#), [243](#)

## **fine-tuned model [82](#)**

### **fine-tuning [434](#)**

bucket, creating [435](#), [436](#)  
model [436](#), [437](#)

## **Foundation Models [21](#), [22](#), [483](#)**

emergence [484](#)  
homogenization [484](#)  
properties [22](#)

## **functional AGI**

emergence [442](#), [443](#)

## **Functional Artificial General Intelligence (F-AGI) [581](#)**

defining [583](#)

## **G**

### **Gaussian Error Linear Unit activation function [141](#)**

## **Gen-2**

example [608](#)

## **General Data Protection Regulation (GDPR) [469](#)**

## **General Language Understanding Evaluation (GLUE) [84](#)**

## **General Purpose Technologies (GPT) [194](#)**

## **General Purpose Technologies (GPTs) [408](#)**

**generative AI [21](#)**

**generative AI customer support model, pretraining on X data [180](#)**

configuration of model, defining [183](#), [184](#)

data, loading and filtering [181](#), [182](#), [183](#)

dataset, creating and processing [184](#), [185](#)

dataset, downloading [181](#)

Hugging Face transformers, installing [181](#)

model, pretraining [186](#)

model, saving [187](#)

resource constraints, checking [183](#)

trainer, initializing [185](#), [186](#)

user interface to Chat, with Generative AI Agent [187](#), [188](#)

## **Generative AI, model versions and lifecycle**

reference link [421](#)

## **Generative AI Stable Diffusion**

for Divergent Association Task (DAT) [535](#), [536](#)

## **generative ideation**

automated ideation architecture [615](#)

defining [614](#)

limitation [616](#)

scope [616](#)

## **generative image design**

automating [616](#)

ChatGPT/GPT-4 HTML presentation [617](#)

Llama 2 [622](#)

Microsoft Designer [636](#), [637](#), [638](#), [639](#)

Midjourney [629](#), [630](#), [631](#)

## **Generative Pre-trained Transformers (GPT) [194](#)**

diffusion [196](#)

improvements [194](#), [195](#), [196](#)

OpenAI GPT transformer models, architecture [197](#)

pervasiveness [197](#)

## **Generative Pre-Training (GPT) [198](#)**

### **generative task [401](#)**

### **Gensim**

embedding with [316](#), [317](#)

vector space [320](#), [321](#), [322](#), [323](#)

## **GLUE leaderboard [85](#)**

URL [84](#)

## **Google assistants [408](#), [409](#)**

Bard [410](#)

### **Google Bard**

potential [121](#)

used, for translation [120](#)

## **Google Cloud Vision [598](#)**

chaining, to ChatGPT [604](#), [605](#), [606](#)

difficult image [601](#), [602](#)

easy image [599](#), [600](#)

very difficult image [602](#), [603](#)

## **Google Colab Copilot [413](#), [414](#), [415](#)**

### **googletrans, implementing**

language of a text, detecting [119](#)

text, transaltaion from English to French [118](#)

translating, in multiple langugaes [119](#)

## **Google Translate**

reference link [117](#)

used, for translation [117](#)

## **Google Translate AJAX API Wrapper**

googletrans, implementing [118](#)

used, for translation [118](#)

## **Google Trax**

decoding, from Transformer [116](#)

de-tokenizing [116](#)

installing [113](#)

model, installing with pretrained weights [115](#)

Original Transformer model, creating [113](#), [115](#)

sentence, tokenizing [115](#)

translation, displaying [116](#), [117](#)

used, for translation [113](#)

## **Google Workspace [410](#), [411](#), [413](#)**

**GPT-3** [278](#)

**GPT-4** [274](#), [276](#), [349](#), [363](#)

sample (basic) [364](#), [365](#), [366](#)

sample (difficult) [367](#), [368](#), [369](#)

**GPT-4 API** [215](#)

multiple NLP task, running [217](#)

NLP task, running with [215](#)

**GPT-4 dialog function** [363](#)

**GPT-4V API**

divergent semantic association [516](#), [517](#)

divergent semantic association, limits [517](#), [518](#), [519](#)

implementing [514](#)

standard image and text [514](#), [515](#), [516](#)

**GPT, diffusion**

application sectors [196](#)

development assistants [196](#)

self-service assistants [196](#)

**GPT models** [204](#), [205](#)

ask functionality [334](#)

evaluating, with knowledge base [329](#), [330](#), [331](#)

implementing [328](#)

knowledge base, adding [330](#)

search data, preparing [331](#), [332](#)

search functionality [333](#), [334](#)

## **Graphics Processing Unit (GPU) [6](#), [168](#)**

computational time complexity with [9](#), [10](#), [11](#)

## **H**

### **higher Human Baselines standards [85](#)**

### **high-level transformer [267](#)**

### **Hopfield networks [16](#)**

### **Hugging Face [379](#), [584](#), [588](#)**

used, for implementing Llama 2 [624](#), [625](#), [626](#), [627](#), [628](#)

### **Hugging Face.**

reference link [133](#)

### **Hugging Face AutoTrain platform**

reference link [549](#)

used, for training models [553](#), [554](#), [555](#)

### **Hugging Face BERT uncased base model**

loading [142](#), [143](#)

### **Hugging Face outputs**

with SHAP [263](#), [264](#), [265](#)

### **Hugging Face platform [549](#), [550](#)**

reference link [554](#)

### **Hugging Face transformer model**

selecting [379](#), [380](#), [381](#)

### **Hugging Face transformer models [69](#)**

### **Hugging Face transformers**

interpreting, with SHAP [260](#)

## Hugging Face Transformers

installing [134](#), [135](#)

## HuggingGPT [584](#), [588](#)

difficult image [592](#), [593](#), [594](#)

easy image [589](#), [590](#), [591](#)

four-step AI system [589](#)

running [589](#)

very difficult image [594](#), [595](#), [596](#), [597](#)

## Human-Centered Artificial Intelligence (HAI) [443](#)

### human evaluations [83](#)

## I

### image classification [559](#), [560](#)

### image generation boundaries

transcending [524](#), [525](#)

### inference models

image classification [559](#), [560](#)

running [557](#)

validation experimentation on trained models [561](#)

validation images, retrieving [557](#), [558](#), [559](#)

ViT model [577](#), [578](#)

### interaction [21](#)

### interactive transformer visualization page [268](#), [269](#), [270](#)

**Intermediate Representation (IR)** [404](#)

**Interpretable AI Tools** [270](#)

LIT [271](#)

PCA [271](#)

## J

**JavaScript Object Notation (JSON)** [118](#)

**JSON**

data, preparing [231](#), [232](#), [233](#)

**JSONL**

data, converting to [233](#), [234](#), [235](#)

## K

**KantaiBERT** [161](#)

**KantaiBERT, building** [162](#)

configuration of model, defining [169](#)

data collator, defining [176](#)

dataset, building [175](#), [176](#)

dataset, loading [162](#), [163](#)

files, saving to disk [166](#), [167](#)

final model (+tokenizer + config) to disk, saving [178](#), [179](#)

Hugging Face transformers, installing [163](#), [164](#)

language modeling, with FillMaskPipeline [179](#)

model, initializing [169](#), [170](#), [171](#)

model, pretraining [178](#)  
model’s parameters, exploring [171](#), [172](#), [173](#), [174](#), [175](#)  
resource constraints, checking [168](#), [169](#)  
tokenizer, reloading in transformers [169](#)  
tokenizer, training [164](#), [165](#), [166](#)  
trained tokenizer files, loading [167](#), [168](#)  
trainer, initializing [176](#), [177](#), [178](#)

## Keras Stable Diffusion implementation

running [531](#), [532](#), [533](#)

## k-means clustering

used, for finding clusters [342](#)

## Knowledge Base (KB)

building, for ChatGPT and GPT-4 [475](#), [476](#)

# L

## LangChain [454](#)

## Language Interpretability Tool (LIT) [247](#), [271](#)

references [272](#)

running [272](#), [273](#), [274](#)

## Large Language Model (LLM) [311](#), [349](#)

## Large Language Models (LLM) [299](#)

morphological flexibility [299](#)

multiple languages [299](#)

noise resistance [299](#)

Out-of-Vocabulary (OOV) words [299](#)

vocabulary optimization [299](#)

**Large Language Models (LLMs)** [1](#), [247](#), [483](#)

reference link [134](#)

**lemmatization** [315](#)

**Llama 2** [622](#)

defining [622](#), [623](#)

implementing, with Hugging Face [624](#), [625](#), [626](#), [627](#), [628](#)

**LLM embeddings**

using, as alternative to fine-tuning [312](#)

**LLM project management** [350](#), [351](#)

**Local Interpretable Model-agnostic Explanations (LIME)** [247](#),  
[267](#), [268](#)

**low-level transformer** [267](#)

## M

**machine translation**

defining [100](#), [101](#)

evaluating [102](#)

evaluation, with BLEU [109](#)

human transductions and translations [101](#)

machine transductions and translations [102](#)

WMT dataset, processing [102](#), [103](#)

**machine translations evaluations, with BLEU** [109](#)

Chenberry smoothing [112](#), [113](#)  
geometric evaluations [109](#), [110](#), [111](#)  
smoothing technique, applying [111](#), [112](#)

## **Markov Chains** [16](#)

### **Markov Decision Process (MDP)** [16](#)

### **Markov Processes** [16](#)

### **Masked Language Modeling (MLM)** [129](#), [130](#), [161](#)

### **Massively Multiplayer Online (MMO)** [648](#)

### **Massive Multitask Language Understanding (MMLU)** [434](#)

### **Massive Multitask Language Understanding (MMLU) models** [442](#)

### **Matthews Correlation Coefficient (MCC)** [83](#)

used, for evaluation [151](#)  
used, for evaluation for whole dataset [152](#)

### **measurement scoring methods, for models**

accuracy score [82](#)  
F1-score [82](#)  
Matthews Correlation Coefficient (MCC) [83](#)

### **Microsoft Designer** [636](#), [637](#), [638](#), [639](#)

### **Microsoft Research Paraphrase Corpus (MRPC)** [94](#), [95](#)

### **Midjourney** [629](#), [630](#), [631](#)

Discord API [631](#), [632](#), [633](#), [634](#), [635](#)  
example [607](#), [608](#)

### **mid-level transformer** [267](#)

## **model**

fine-tuning [436](#), [437](#)

## **model card**

deploying [555](#), [556](#), [557](#)

## **model chaining**

with Runway Gen-2 [607](#)

## **Model Garden on Vertex AI**

reference link [436](#)

## **models**

training, with Hugging Face AutoTrain platform [553](#), [554](#), [555](#)

## **modifier** [353](#)

## **multi-head attention** [50](#)

architecture [51](#), [52](#)

concatenation, of output of heads [63](#)

final attention representations [59](#), [60](#)

input, representing [53](#)

matrix multiplication [56](#)

output [62](#), [63](#)

post-layer normalization [64](#), [65](#)

results, summing up [61](#)

scaled attention scores [57](#), [58](#)

scaled softmax attention scores [58](#), [59](#)

weight matrices, initializing [53](#), [54](#), [55](#)

## **Multi-Head Attention (MHA)** [623](#)

**Multi-Level Intermediate Representation (MLIR)** [404](#)

**multimodal neurons** [484](#)

**Multi-Sentence Reading Comprehension (MultiRC)** [89](#), [90](#)

**multi-word tokenization** [298](#)

## N

**Named Entity Recognition (NER)** [295](#)

**Natural Language Processing (NLP) tasks** [73](#), [74](#)

**Natural Language Toolkit (NLTK)** [109](#), [313](#)

**Natural Language Understanding (NLU) tasks** [73](#)

**neuron, explaining**

example [275](#), [276](#), [277](#), [278](#)

**Next-Sentence Prediction (NSP)** [130](#), [131](#)

**NLP PCA representation**

features [271](#)

**NLP task, with GPT-4**

API key, entering [215](#)

hyperparameters [216](#)

hyperparameters, system role [217](#)

hyperparameters, user role [217](#)

OpenAI, installing [215](#)

running [215](#), [216](#)

## O

## O(1)

terminology [3](#)

## O(1), conceptual approach

attention layer [4](#), [5](#)

recurrent layer [5](#)

## one-token approach [20](#)

## OpenAI [74](#)

installing [362](#)

URL [79](#)

## OpenAI CLIP

for text-to-video [539](#), [540](#)

## OpenAI GPT Models

dataset, preparing [231](#)

fine-tuning [235](#), [236](#), [238](#)

fine-tuning, for completion (generative) [229](#), [230](#)

risk management [228](#), [229](#)

## OpenAI GPT Models, dataset

converting, to JSONL [233](#), [234](#), [235](#)

preparing, in JSON [231](#), [232](#), [233](#)

## OpenAI GPT transformer models, architecture [197](#), [198](#)

billion-parameter transformer models [198](#)

context size and maximum path length [200](#)

decoder layers, stacking [202](#), [203](#)

transformer models, size [199](#), [200](#)

zero-shot models [201](#), [202](#)

## Open AI pricing page

reference link [329](#)

## Out-Of-Vocabulary (OOV) [107](#)

# P

## PaLM 2 [407](#)

improvements [407](#), [408](#)

## pandas

word relationships, visualizing with attention scores [256](#), [257](#),  
[258](#), [259](#)

## Part-of-Speech (POS) [295](#)

## Pathways [402](#)

client [404](#)

compiler [404](#)

executor [405](#)

features [402](#)

functionality [403](#)

Intermediate Representation (IR) [404](#)

resource manager [404](#)

scheduler [404](#)

## Pathways Language Model (PaLM) [401](#), [405](#)

## Pathways Language Model (PaLM), advances and improvements

no biases [406](#)  
parallel layer processing [405](#), [406](#)  
Rotary Positional Embedding (RoPE) [406](#)  
shared input-output embeddings [406](#)  
SwiGLU activations [406](#)

## **phenomenal progress**

Few-Shot (FS) [202](#)  
fine-tuning (FT) [202](#)  
one-shot (1S) [202](#)  
zero-shot (ZS) [202](#)

## **positional encoding (PE)** [45](#), [47](#), [48](#)

adding, to embedding vector [48](#), [49](#), [50](#)

## **Post-Layer Normalization (Post-LN)** [64](#), [65](#)

## **predicate analysis**

challenges [358](#), [359](#), [360](#)

## **prefix** [376](#)

## **pretrained model** [82](#)

## **pretraining input environment, encoder stack**

preparing [129](#)

## **Principal Component Analysis (PCA)** [247](#), [271](#)

## **Project Gutenberg**

URL [162](#)

## **prompt design**

migrating, to prompt engineering [313](#)

## **Prompt examples**

reference link [32](#)

## **Proof of Concept (POC) [189](#)**

### **Punkt**

text, tokenizing with [314](#)

tokens, preprocessing [314](#)

## **Punkt tokenization [298](#)**

### **Python interface**

building, to interact with BERT model [152](#)

## **Q**

### **quality control (QC) [286](#)**

### **question-answering systems**

implementing, with embedding-based search techniques [327](#)

### **question-answering systems implementation [327](#)**

embedding model and GPT model, implementing [328](#)

libraries, installing [327](#), [328](#)

models, evaluating with knowledge base [331](#)

models, selecting [327](#), [328](#)

## **R**

### **RAG, with GPT-4 [218](#)**

augmented retrieval generation [220](#), [221](#), [222](#), [223](#)

document retrieval [219](#), [220](#)

installation [218](#)

**Random Number Generator (RNG)** [528](#)

**Reading Comprehension with Commonsense Reasoning Dataset (ReCoRD)** [90](#), [91](#)

**ready-to-use API-driven libraries**

selecting [33](#), [34](#)

**Recognizing Textual Entailment (RTE)** [91](#)

features [133](#)

**recurrent layer** [5](#)

**Recurrent Neural Networks (RNNs)** [15](#)

**recurrent to attention**

journey from [15](#)

**regular expression tokenization** [296](#), [297](#)

**Reinforcement Learning from Human Feedback (RLHF)** [462](#), [470](#)

**Residual Network (ResNet)** [574](#)

**ResNetForImageClassification** [574](#), [577](#)

**Retrieval Augmented Generation (RAG)** [218](#), [312](#), [614](#), [615](#)

**Retrieval-Augmented Generation (RAG)** [470](#)

**risk management** [457](#)

cybersecurity [469](#)

disinformation [464](#)

hallucination [458](#), [459](#), [460](#), [462](#)

harmful content [467](#), [468](#)

influence operations [465](#), [466](#)

memorization [462](#)

privacy [469](#)

risky emergent behaviors [462](#), [463](#), [464](#)

## **risk mitigation techniques [470](#), [471](#)**

ChatGPT content, generating with dialog function [478](#), [480](#)

input and output moderation [471](#), [472](#), [473](#), [474](#), [475](#)

keywords, adding to parse user requests [476](#)

Knowledge Base (KB), implementing [475](#), [476](#)

moderation [480](#)

token control [480](#)

user requests, parsing [477](#), [478](#)

## **risk mitigation tools**

with RAG [470](#)

with RLHF [470](#)

## **RNN process [17](#)**

## **Robustly Optimized BERT Pretraining Approach (RoBERTa) [161](#)**

## **Rotary Positional Embedding (RoPE) [406](#)**

## **Runway Gen-2**

for model chaining [607](#)

# **S**

## **semantic role [352](#)**

## **Semantic Role Labeling (SRL)** [349](#), [350](#), [352](#)

defining [352](#), [353](#)

redefining [360](#), [361](#), [362](#)

scope, questioning [358](#)

visualizing [353](#)

## **SentencePiece tokenization** [301](#)

### **sentence tokenization** [295](#), [296](#)

exploring [294](#)

## **Service Organization Control (SOC) audit** [469](#)

## **SHAP** [260](#), [261](#), [263](#)

Hugging Face outputs [263](#), [264](#), [265](#)

Hugging Face transformers, interpreting with [260](#)

## **Situations With Adversarial Generations (SWAG)**. [133](#)

## **Software as a Service (SaaS) API** [31](#)

## **Software-as-a-Service (SaaS) API** [470](#)

## **Software Development Kit (SDK)** [549](#)

## **SRL experiments, with ChatGPT with GPT-4** [353](#)

basic sample [354](#), [355](#), [356](#), [357](#)

difficult sample [357](#), [358](#)

## **Stability AI** [533](#)

### **Stability AI animation**

for text-to-video [536](#), [537](#), [538](#), [539](#)

## **Stable Diffusion**

text-to-image, defining with [526](#)

text-to-image, running with [533](#), [534](#), [535](#)  
using, in automated generative ideation [640](#)

## **Standard Sentiment TreeBank (SST-2) [73](#)**

## **Stanford Sentiment TreeBank (SST-2) [93](#), [94](#)**

**stop words [315](#), [316](#)**

## **Subject Matter Expert (SME) [28](#), [117](#), [396](#)**

## **subword tokenization [299](#)**

Byte-Pair Encoding (BPE) [302](#)

SentencePiece tokenization [301](#), [302](#)

unigram language model tokenization [300](#), [301](#)

WordPiece [303](#)

## **SuperGLUE [84](#)**

### **SuperGLUE benchmark tasks**

BoolQ [88](#)

Commitment Bank (CB) [88](#)

defining [88](#)

Multi-Sentence Reading Comprehension (MultiRC) [89](#), [90](#)

Reading Comprehension with Commonsense Reasoning  
Dataset (ReCoRD) [90](#), [91](#)

Recognizing Textual Entailment (RTE) [91](#)

Winograd Schema Challenge (WSC) [91](#)

Words in Context (WiC) [91](#)

## **SuperGLUE evaluation process [86](#), [87](#)**

## **SuperGLUE leaderboard [85](#)**

**supervised training** [20](#)

**SwinForImageClassification 1** [565](#), [567](#)

layers [565](#)

**SwinForImageClassification 2** [570](#), [571](#)

**Switched Gated Linear Unit (SwiGLU)** [406](#)

## T

**T5, and ChatGPT**

pretraining [393](#)

specific, versus non-specific tasks [394](#)

summarization methods, comparing [393](#)

**T5-large, for summarizing documents** [386](#)

Bill of Rights sample [390](#), [391](#)

corporate law sample [391](#), [392](#), [393](#)

general topic sample [388](#), [389](#)

summarization function, creating [387](#), [388](#)

**T5-large transformer model**

initializing [381](#), [382](#), [383](#)

**T5 model** [377](#), [378](#), [379](#)

architecture [383](#), [384](#), [385](#), [386](#)

text summarization [379](#)

**T5 text-to-text framework** [377](#)

**task-agnostic models**

migrating, to multimodal vision transformers [484](#), [485](#)

**task-specific formats** [376](#)

**task-specific models** [21](#)

**Tensor2Tensor (T2T)** [113](#)

**TensorFlow Embedding Projector**

URL [323](#)

**Tensor Processing Unit (TPU)** [6](#), [405](#)

computational time complexity with [11](#), [12](#), [13](#)

**text embedding**

Gensim♦'s vector space, exploring [320](#), [321](#), [322](#), [323](#)

libraries, installing [313](#)

model description [317](#), [318](#), [319](#)

TensorFlow Projector [323](#), [324](#), [325](#), [326](#)

text file, reading [314](#)

text, tokenizing with Punkt [314](#)

with Gensim and Word2Vec [316](#), [317](#)

with NLKT and Gensim [313](#)

word and vector, accessing [319](#), [320](#)

**text-to-image**

running, with Stable Diffusion [533](#), [534](#), [535](#)

**text-to-image, defining with Stable Diffusion** [526](#)

decoder upsampling [530](#)

output image [531](#)

random image creation, with noise [528](#)

Stable Diffusion model downsampling [528](#), [529](#), [530](#)

text embedding, with transformer encoder [526](#), [527](#)

## **text-to-text transformer models [375](#), [376](#)**

### **text-to-video [536](#)**

with OpenAI CLIP [539](#), [540](#)

with Stability AI animation [536](#), [537](#), [538](#), [539](#)

### **TimeSformer**

for video-to-text model [540](#)

predictions, making on video frames [543](#), [544](#)

### **token**

to AI revolution [17](#), [18](#), [19](#), [20](#)

### **tokenizer**

training [160](#)

### **tokens**

converting, to lowercase [314](#)

lemmatization [315](#)

stop words [315](#), [316](#)

### **TPU-LLM [14](#), [15](#)**

### **training dataset**

no coding [553](#)

uploading [550](#), [551](#), [552](#)

### **transfer learning, with Ada embeddings [338](#)**

Amazon Fine Food Reviews dataset [338](#)

clustering [341](#), [342](#)

clusters, naming [345](#)

data preparation [339](#), [340](#)

text samples, in clusters [344](#), [345](#)

## transformer

developments [31](#), [32](#), [33](#)

## Transformer

performance [69](#)

training [69](#)

## Transformer architecture [38](#), [39](#), [40](#)

decoder stack [66](#)

encoder stack [40](#), [41](#), [42](#)

## transformer-driven AI

used, for daily applications [22](#), [23](#), [24](#), [25](#), [26](#), [27](#)

## transformer factor [265](#), [266](#), [267](#)

## transformer model

selecting [34](#)

## transformer models

limitations and human control [278](#)

## transformers [73](#), [82](#)

head of attention sublayer [75](#), [76](#), [77](#), [78](#), [79](#)

history [16](#), [17](#)

pretraining [160](#)

## transformer visualization

via dictionary learning [265](#)

with BertViz [248](#)

## **translations**

with Google Bard [120](#)

with Google Translate [117](#), [118](#)

with Google Trax [113](#)

## **treebank tokenization** [297](#)

## **t-SNE**

used, for displaying clusters [343](#), [344](#)

## **TSV (Tab-Separated Values) format** [323](#)

# **U**

## **Uniform Manifold Approximation and Projection (UMAP)** [247](#), [271](#)

## **unigram language model tokenization** [300](#), [301](#)

## **universal text-to-text model**

designing [374](#), [375](#)

## **unsupervised** [201](#)

## **unsupervised training** [20](#)

# **V**

## **validation experimentation on trained models** [561](#)

BeitForImage Classification [567](#), [568](#), [569](#)

ConvNextForImageClassification [572](#), [574](#)

ResNetForImageClassification [574](#), [576](#)

SwinForImageClassification 1 [565](#), [567](#)

SwinForImageClassification 2 [570](#), [571](#)

ViTForImageClassification [562](#), [563](#), [564](#)

## **validation images**

retrieving [557](#), [558](#), [559](#)

## **validation set [585](#)**

difficult image [586](#)

easy image [585](#)

very difficult image [587](#)

## **variance [271](#)**

## **Variational Autoencoders (VAE) [525](#)**

## **verb [353](#)**

## **Vertex AI PaLM 2 API [421](#), [422](#)**

code, producing [430](#), [432](#), [434](#)

conversation, summarizing [424](#), [426](#)

multi-choice problem [428](#), [429](#), [430](#)

Question Answering (QA) [422](#), [423](#)

Question Answering (QA) task [423](#)

sentiment analysis [426](#), [427](#), [428](#)

## **Vertex AI PaLM 2 assistant [418](#), [419](#), [421](#)**

## **Vertex AI PaLM 2 interface [415](#), [416](#), [417](#), [418](#)**

## **video frames**

preparing [541](#), [542](#), [543](#)

## **video-to-text model**

with TimeSformer [540](#)

## **Vision Transformer (ViT) [485](#)**

architecture [485](#)

configuration and shapes [493](#), [494](#), [496](#), [497](#), [498](#)

feature extractor simulator [489](#), [490](#), [491](#), [492](#)

in code [488](#), [489](#)

transformer component [492](#), [493](#)

## **ViT architecture [485](#)**

images, splitting into patches [486](#)

transformer [487](#), [488](#)

vocabulary of image patches, building [486](#)

## **ViT-base-patch16-224 [564](#)**

## **ViTForImageClassification [562](#), [563](#), [564](#)**

## **ViT model [577](#), [578](#)**

## **VR-AI development [647](#)**

parallelization [648](#), [650](#), [651](#)

risks [651](#)

## **W**

## **WandB [453](#), [454](#)**

## **white space tokenization [297](#)**

## **Winograd Schema Challenge (WSC) [91](#)**

## **Winograd schemas [95](#), [96](#)**

## **WMT dataset**

preprocessing, finalizing [106](#), [107](#), [108](#)

raw data, preprocessing [103](#), [104](#), [105](#), [106](#)

## **WMT dataset.**

preprocessing [102](#)

## **WMT (Workshop on Machine Translation) [84](#)**

### **Word2Vec**

documentation link [319](#)

embedding with [316](#), [317](#)

### **Word2Vec tokenization [286](#), [287](#), [288](#), [289](#)**

dataset and dictionary, without words [290](#), [291](#)

dataset and dictionary, with words [289](#), [290](#)

noisy relationships [292](#)

rare words [293](#)

rare words, replacing [294](#)

word text, without dictionary [292](#)

### **WordPiece [303](#)**

#### **WordPiece tokenization**

code, exploring [303](#)

exploring [294](#)

token-ID mappings, displaying [305](#), [306](#)

token-ID mappings quality, analyzing [306](#), [307](#)

token-ID mappings quality, controlling [306](#), [307](#)

tokenizer type, detecting [303](#), [304](#), [305](#)

#### **word punctuation tokenization [298](#)**

#### **word relationships**

visualizing, with attention scores [256](#), [257](#), [258](#), [259](#)

**Words in Context (WiC) [91](#)**

**word tokenization [295](#), [296](#)**

**Workshop on Machine Translation (WMT) [99](#)**

**Workshop on Statistical Machine (WMT) [102](#)**

**Workshops on Machine Translation (WMT) [69](#)**

## Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781805128724>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

EXPERT INSIGHT

---

# Transformers for Natural Language Processing and Computer Vision

Explore Generative AI and Large Language Models with  
Hugging Face, ChatGPT, GPT-4V, and DALL-E 3

Third Edition



---

**Denis Rothman**

**⟨packt⟩**