Join our book community on Discord

https://packt.link/EarlyAccessCommunity

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 myriads of other offers are emerging.The functionality of transformer models has pervaded every aspect of our workspace 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 Google Workspace or Microsoft 365, and obtain a response created by PaLM 2 or a ChatGPT transformer model. Finally, users can go to Google's Generative AI 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 artificial intelligence 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 Generative AI 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 level. 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, the reader will have acquired enough critical knowledge to adapt to this every-moving market.In this chapter, we will first unveil the incredible power of the deceiving 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 (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, analyses, 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 artificial intelligence professionals
- The future of artificial intelligence professionals
- Seamless transformer APIs
- Choosing a transformer model

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 deceivingly 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 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 rock 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(meanings, parameters) of the word "oranges" contains several relationships:Dimension 1: the association between oranges and energy Dimension 2: oranges and morningDimension 3: Jay and orangesDimension 4: Jay and morningDimension 5: Jay and energy…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 cased=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. This pairwise relationship is an n * n computation, as shown in *Figure 1.1*:

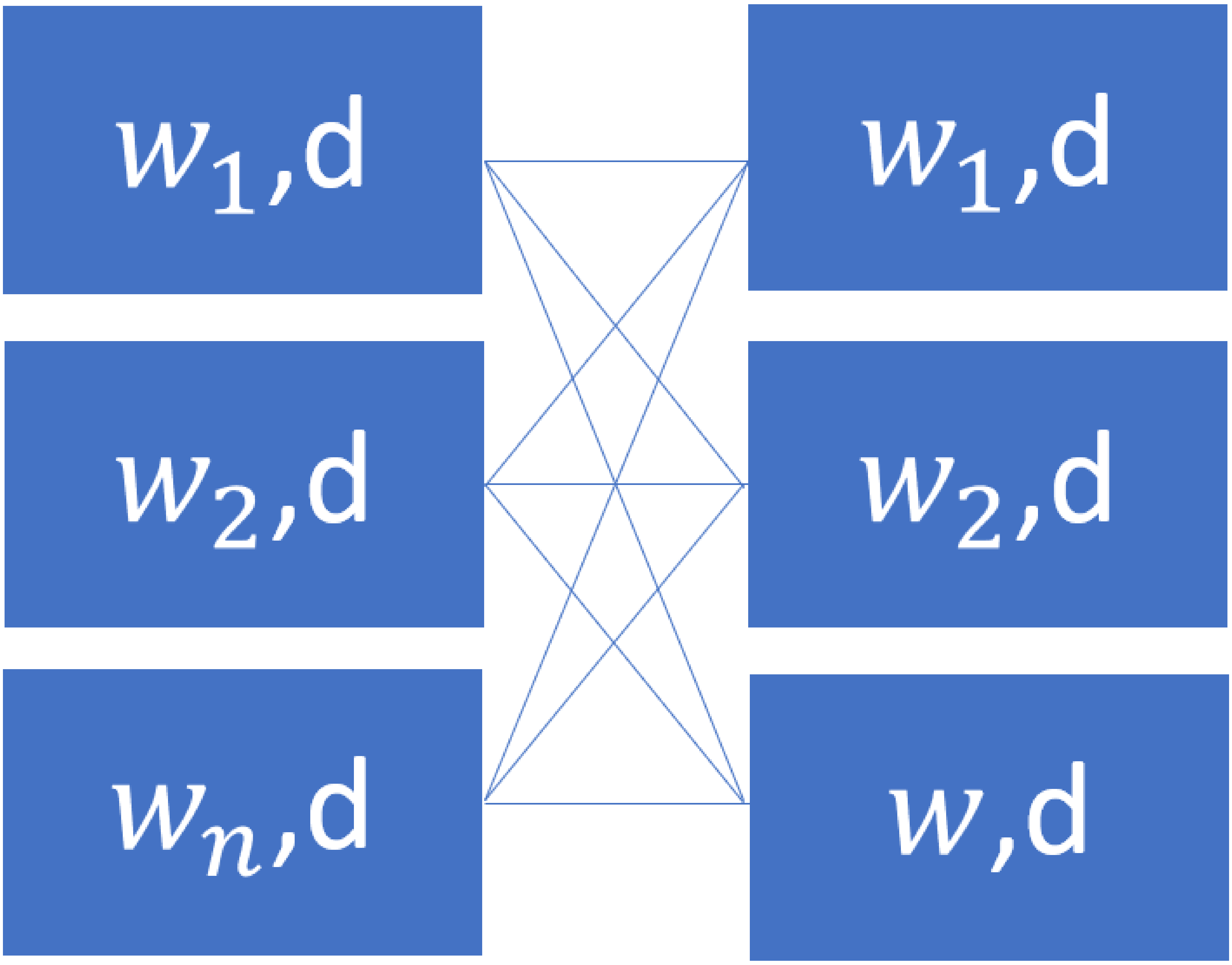

Figure 1.1: Word-to-word relationships

O(1) represents the memory complexity in this case.The computational complexity of an attention layer is thus is the pairwise (word-to-word) operation of the whole sequence n and 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: JayDimension b: likes and Jay Dimension d: oranges and likes and Jay Dimension e: 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 to learn the dimensions. The computational time complexity of a recurrent layer is thus:O(n*d2)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 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 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 accelerating machine learning workload processor developed by Google and optimized for Tensorflow.

Open O-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 to **None** and the **Runtime shape** (RAM) is set to **Standard**:

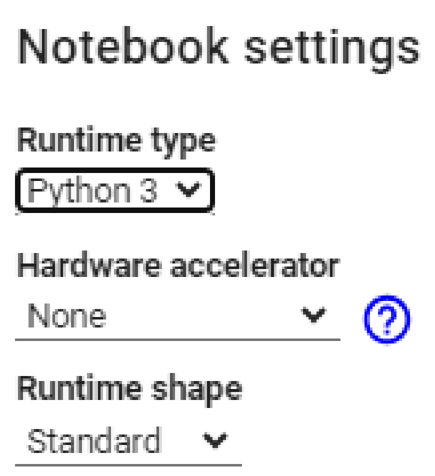

Figure 1.2: Selecting a Runtime type

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	$O(n \cdot d^2)$	$O(n)$	$O(n)$

Figure 1.3: 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 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:

Copy Explain

```
# 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:

Copy Explain

```
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

Copy Explain

```
n = 512 d = 512
```

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:

Copy Explain

```
# define the inputs input_seq = np.random.rand(n, d)
```

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

Copy Explain

```
# 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:

Copy Explain

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

The time is displayed:

Copy Explain

```
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:

Copy Explain

```
# 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[k]
rt=time.time()-start_time
print(f"Recurrent layer computation time: {time.time() - start_time} seconds")
The output shows the time it took:
```

Copy Explain

```
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:

Copy Explain

```
# 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 'computational time for attention' in the sum of 'attention'
and 'recurrent' is {percentage_at}%")
```

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

Copy Explain

```
The percentage of 'computational time for attention' in the sum of 'attention' and
'recurrent' is 0.72%
```

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

Computational time complexity with a GPU

Copy Explain

```
Before running the first cell of the notebook, we must verify that we are using a CPU
by going to the Runtime menu, select Change Runtime type, and making sure that
Hardware accelerator is set to GPU and the Runtime shape (RAM) set to High-Ram:
```

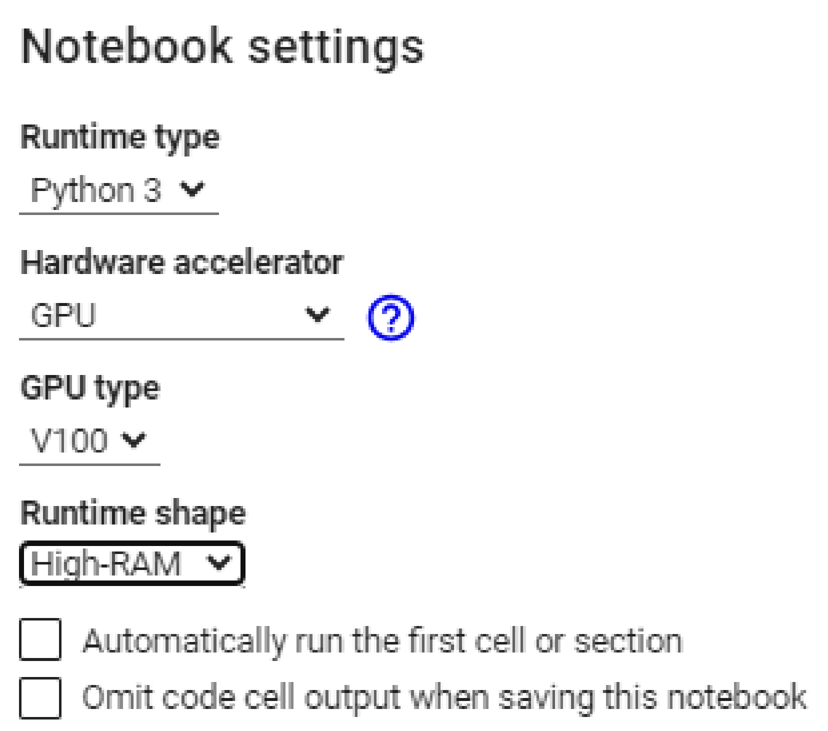

Figure 1.4: Changing the settings to use a GPU

The GPU type is, in this case, a 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 "cpu")
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:

Copy Explain

```
# 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:

Copy Explain

```
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:

Copy Explain

```
# 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:

Copy Explain

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

The output time isn't very efficient:

Copy Explain

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

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

Copy Explain

```
# 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 computation is {percentage_at}%"):
The percentage of self-attention computation in the sum of self-attention and
recurrent computation is 7.36%
```

We can check the information on the GPU:

Copy Explain

```
!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                                                   Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

Figure 1.5: GPU information

We will finally explore a TPU simulation.

Computational time complexity with a TPU

Google designed Cloud TPU.s 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 ten 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, select change **Runtime type, and making sure that Hardware accelerator** is set to **TPU** and the **Runtime shape** (RAM) set to **High-RAM:**

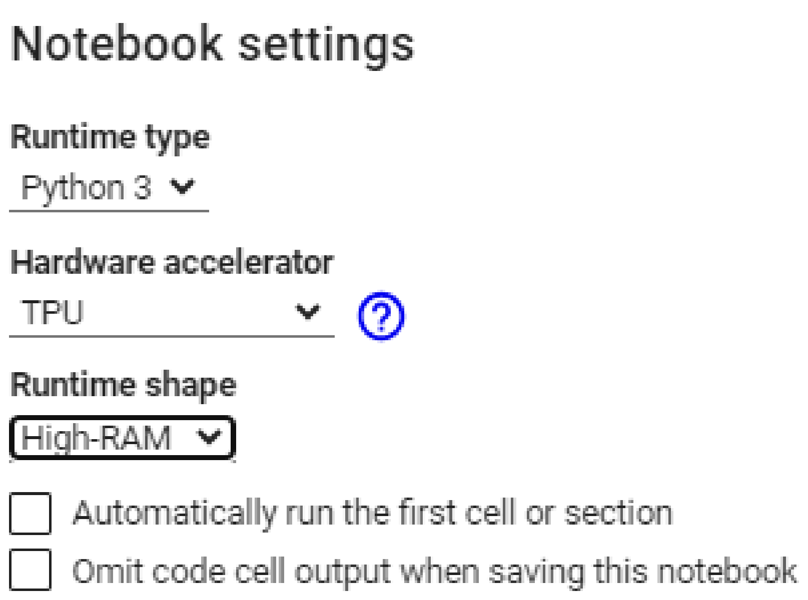

Figure 1.6: Changing the notebook settings to use a TPU

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() * hidden_state[min(i,k), j]
            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 computation is {percentage_at}%")
```

The output confirms that attention layers are more efficient:

Copy Explain

```
The percentage of self-attention computation in the sum of self-attention and
recurrent computation is 0.16%
```

We will now take the TPU simulation into the world of large language models.

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 model. *n=32728* could be the input limit of an LLM model*.* These values are simply an example to show that running the recurrent layer on the TPU with large values makes no sense.The values are those of a large language model:

Copy Explain

```
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:

Copy Explain

```
# 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:

Copy Explain

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

We can display some information on the TPU:

Copy Explain

```
import os
from tensorflow.python.profiler import profiler_client
tpu_profile_service_address = os.environ['COLAB_TPU_ADDR'].replace('8470', '8466')
print(profiler_client.monitor(tpu_profile_service_address, 100, 2))
```

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

Copy Explain

```
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 deceivingly simple O(1) that led to 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 an intuitive look at the attention head of a transformer that has replaced the RNN layers of an NLP neural network. The core concept of a transformer can be summed up loosely as "mixing tokens." NLP models first convert word sequences into tokens. RNNs analyze tokens in recurrent functions. Transformers do not analyze tokens in sequences but relate every token to the other tokens in a sequence, as shown in *Figure 1.7*:

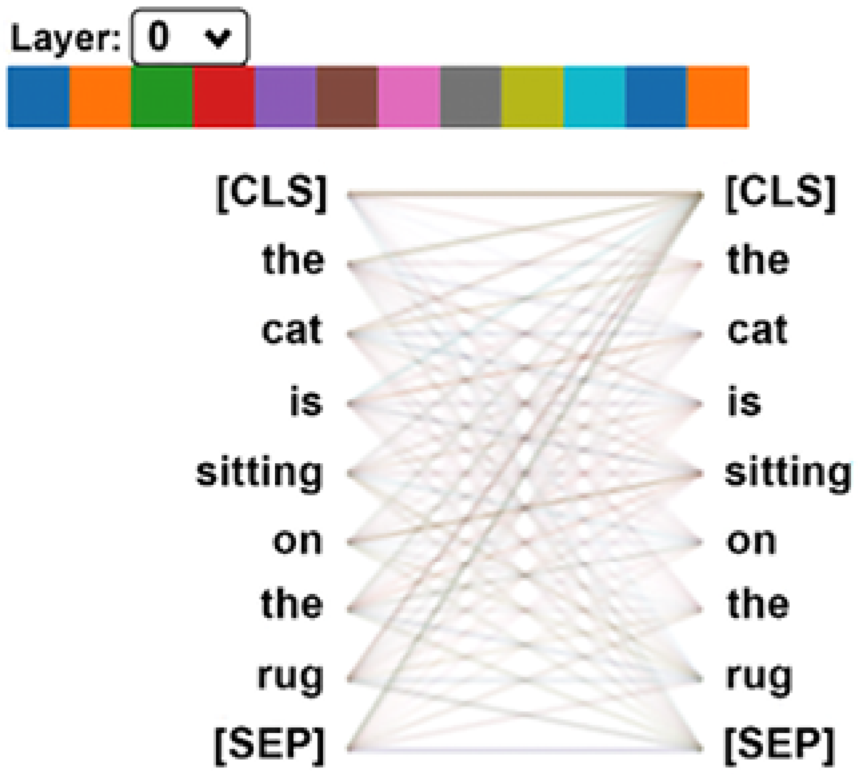

Figure 1.7: An attention head of a layer of a transformer

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 Large Language Models(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, with you to lay the grounds for the arrival of the Transformer. In the late 19th and early 20th century,s Andrey Markov introduced the concept of random values and created a theory of stochastic processes(see *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 last past elements of that chain. He applied to letters and conducted many experiments. Bear in mind that he had no computer but proved a theory still in use today in artificial intelligence. In 1948, Claude Shannon's The Mathematical Theory of Communication was published. Claude Shannon laid the grounds for a communication model based on a source encoder, transmitter, and 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 language 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 "Artificial Intelligence" 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 WA 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 *Sn* captures the information of *S*n-1. When the network's end is reached, a function *F* will perform an action: transduction, modeling, or any other type of sequence-based task. In the 1980s, Yann LeCun designed the multipurpose Convolutional Neural Network (CNN). He applied CNNs to text sequences, and they also apply to sequence transduction and modeling. They are also based on WA 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 thirty years passed this way. Attention was introduced, but the models still relied on reccurency. 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 Large Language Models (LLMs) models have a unique way of producing text.In large language models, 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 interference to produce a token:

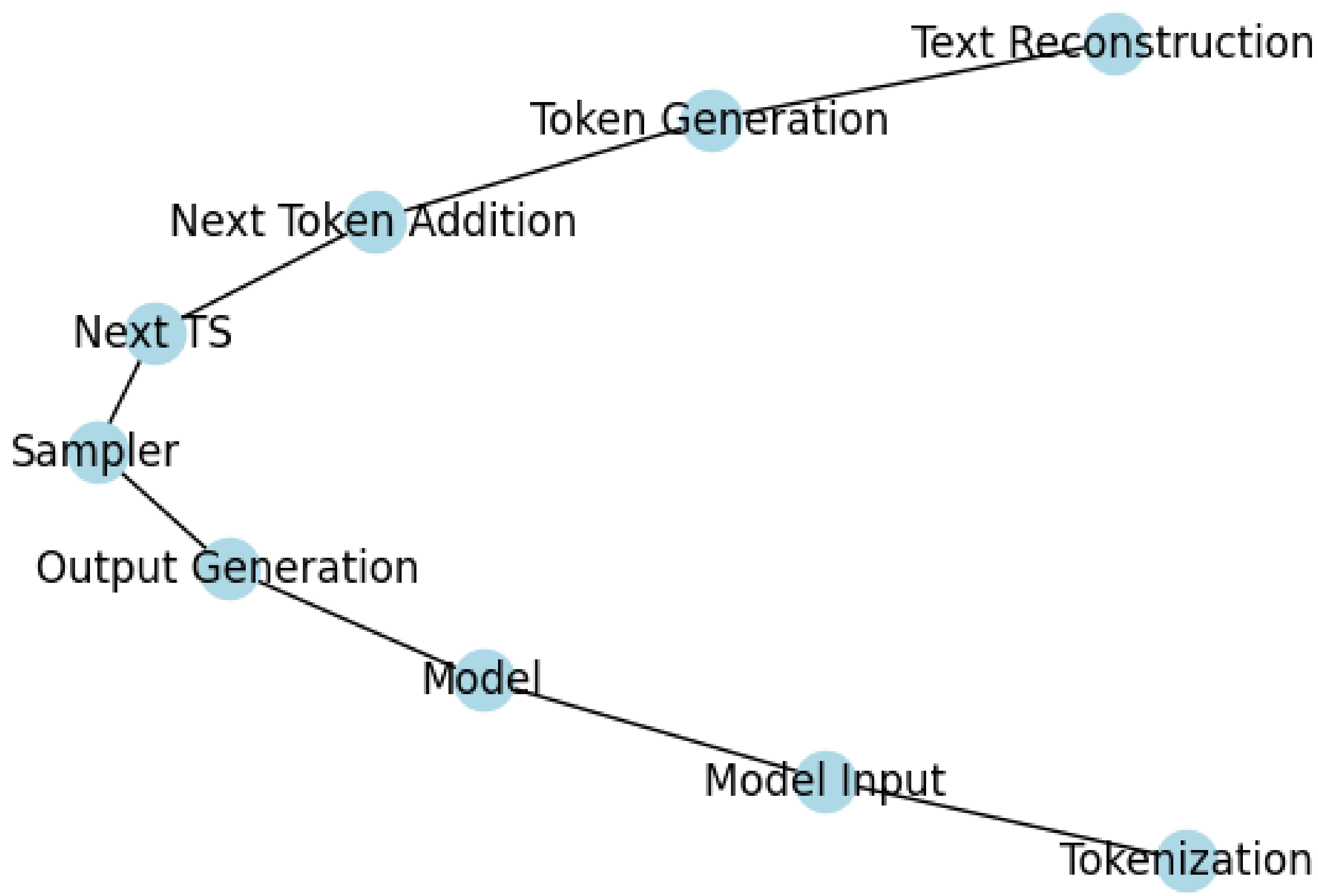

Figure 1.9 GPT Inference graph built in Python with NetworkX

It may come as a surprise, but the only part of this Figure controlled by the model is "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, From Basics to Disruption 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 NLP with Google 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*:

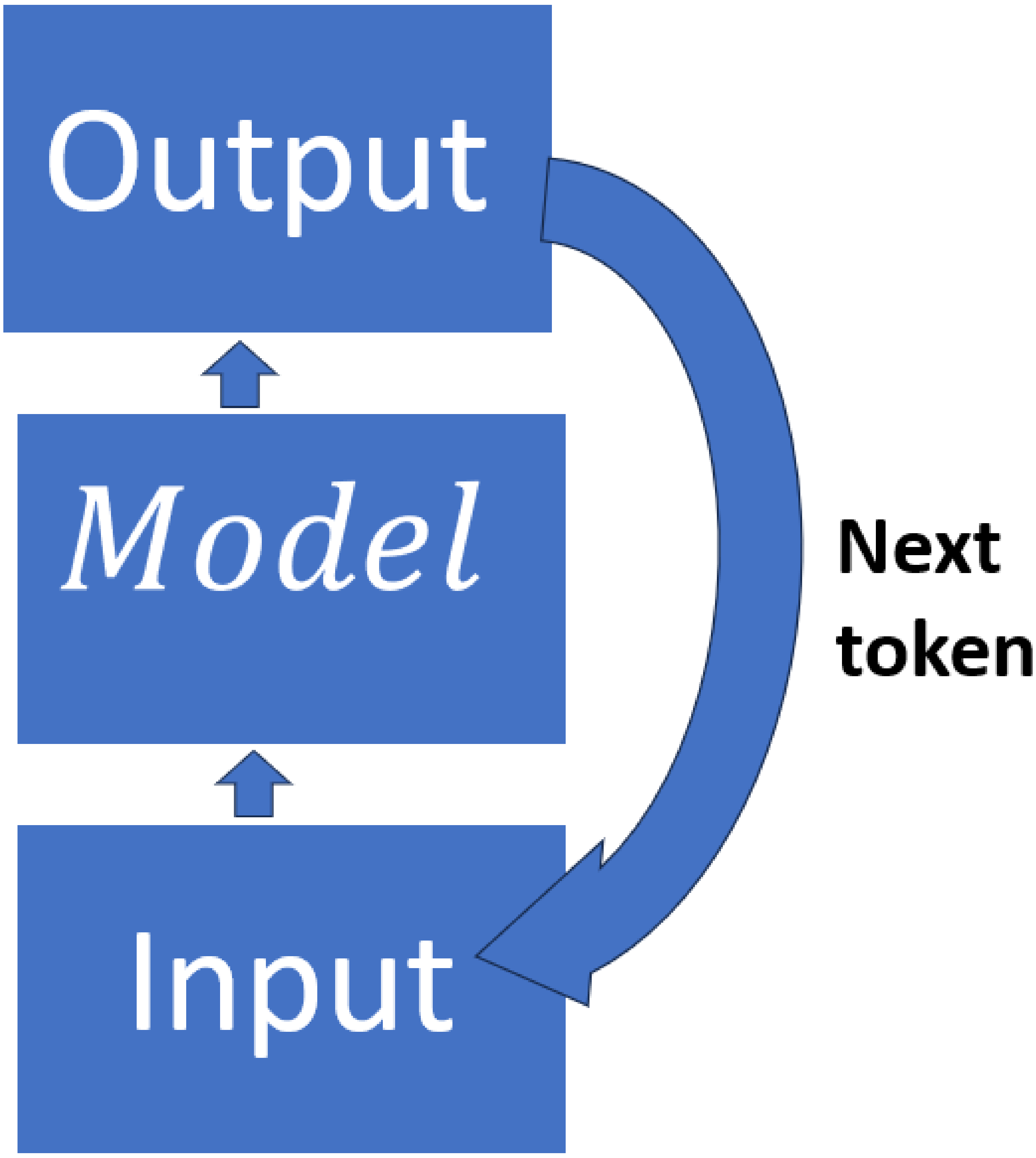

Figure 1.10: The model processes the input sequence, produces raw outputs, and selects one token

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 tokenf = 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), the whole process of adding predicted tokens one by one. You can think of as:t_i = f(t_1, t_2, ..., t_{i-1})Think about it. One token! And yet that token changed society in every domain: translations, summarization, question-answering, imagery, video, and the list is growing daily.We will now go from one token to everything!From one token to everythingIn the previous section, we saw that a generative model can be summed up as. 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,results in :The dynamic nature of a transformer model. It adapts its output outputs based on its incremental inputs represented by: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) the language modeling (adding a new token at the end of a sequence) is attained when a large language model invents a story that is generative AI.A GPT-like model can perform discriminative, generative, or both depending on the task to perform.

Task-specific and specific task

Task-specific models are often opposed to general-purpose models, such as GPT-like LLMs. Task-specific 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 large language models.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 large language general-purpose model! 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, GPT -4, and soon more. GPT 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 artificial intelligence 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 artificial intelligence and can be summed up with four properties:

Model architecture

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

Data

Big tech possesses the hugest data source in the history of humanity, 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) /second, and Google now has domain-specific supercomputers that exceed 80 PetaFLOPS/second. In addition, GPT -4, PaLM 2, and other LLMs use thousands of GPUs to train their models.

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, applications).We will now examine an example of how foundation models work and have changed how we develop programs.

From general purpose to specific task

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 stage. 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 0by 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 demonstrates how to use Scikit-learn, a popular machine learning library in Python, to build a decision tree classifier for the Iris dataset. The Iris dataset is a famous dataset in the machine learning community and is often used as a benchmark for classification algorithms.In this program, you will first load the Iris dataset and split it into training and testing sets. Then, you will create a decision tree classifier model using Scikit-learn's DecisionTreeClassifier class and train it on the training data. Next, you will evaluate the accuracy of the model on the testing set and compute the confusion matrix, which is a table that summarizes the performance of the model. Finally, you will plot the confusion matrix using Matplotlib.This program is a great way for beginners to learn about machine learning and the Scikit-learn library. By following the code and explanations provided, you will gain a basic understanding of how to use Scikit-learn to build and evaluate machine learning models. You will also learn how to use Matplotlib to visualize the results of your models.Overall, this program is a fun and educational way to learn about machine learning and Python. Whether you're a beginner or an experienced programmer, you'll find something of value in this program. So, let's get started and learn how to build a decision tree classifier for the Iris dataset!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:

Copy Explain

```
pip install -U scikit-learn
```

1.Import libaries

We import several libraries that we will use throughout the code. NumPy and Pandas are used for data manipulation, Matplotlib is used for data visualization, Scikit-learn is used for machine learning, and we import specific functions like load_iris, train_test_split, DecisionTreeClassifier, confusion_matrix, and plot_tree from the respective Scikit-learn modules.

Copy Explain

```
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:

Copy Explain

```
iris = load_iris()
```

We use the load_iris function from Scikit-learn's datasets module to load the Iris dataset into a variable called iris.

3.Split the dataset into training and testing sets

Copy Explain

```
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target,
test_size=0.2, random_state=42)
```

We use Scikit-learn's train_test_split function to split the dataset into training and testing sets. We pass in iris.data as the features and iris.target as the labels. We set the test size to 20%, which means that 20% of the data will be used for testing and 80% will be used for training. We also set the random state to 42 to ensure that the results are reproducible.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:

Copy Explain

```
model = DecisionTreeClassifier(random_state=42)
model.fit(X_train, y_train)
```

```
▾ DecisionTreeClassifier
DecisionTreeClassifier(random_state=42)
```

Figure 1.11 ChatGPT creates a decision tree classifier

We create a DecisionTreeClassifier object from Scikit-learn's tree module and set the random state to 42. We then train the model on the training data using the fit method.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:

Copy Explain

```
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 of the model on the testing set. We then print the accuracy to the console.

6.Compute the confusion matrix:

Copy Explain

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

7.1. Plot the confusion matrix:

Copy Explain

```
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_names,
       title="Confusion matrix",
       xlabel="Predicted label",
       ylabel="True label")
plt.setp(ax.get_xticklabels(), rotation=45, ha="right", rotation_mode="anchor")
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()
```

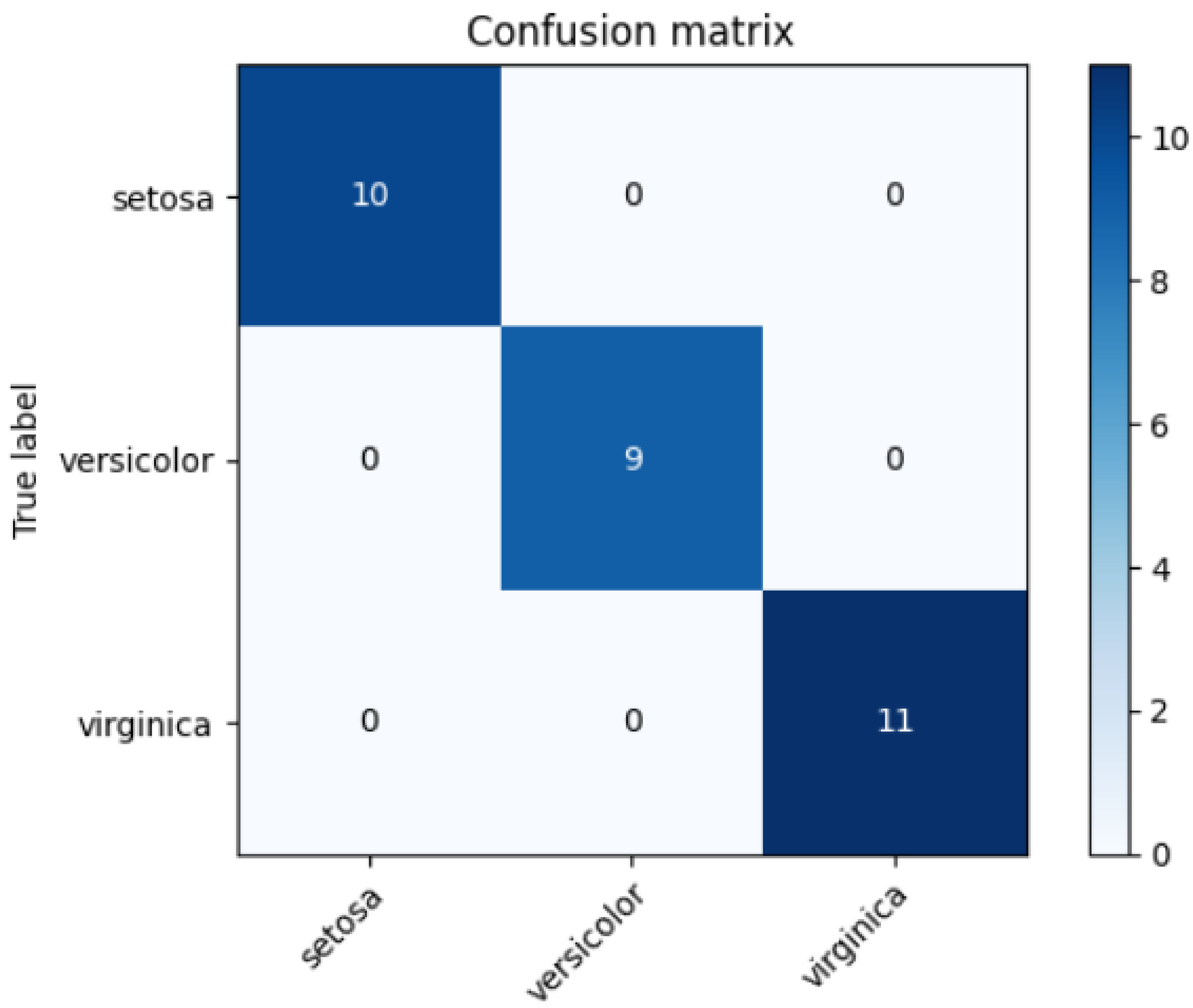

Figure 1.12: ChaptGPT creates a confusion matrix of the model results

Finally, ChatGPT plots the decision tree.

7.2. Plot the Decision Tree

Copy Explain

```
# Plot the decision tree
plt.figure(figsize=(20,10))
plot_tree(model, feature_names=iris.feature_names, class_names=iris.target_names,
filled=True)
plt.show()
```

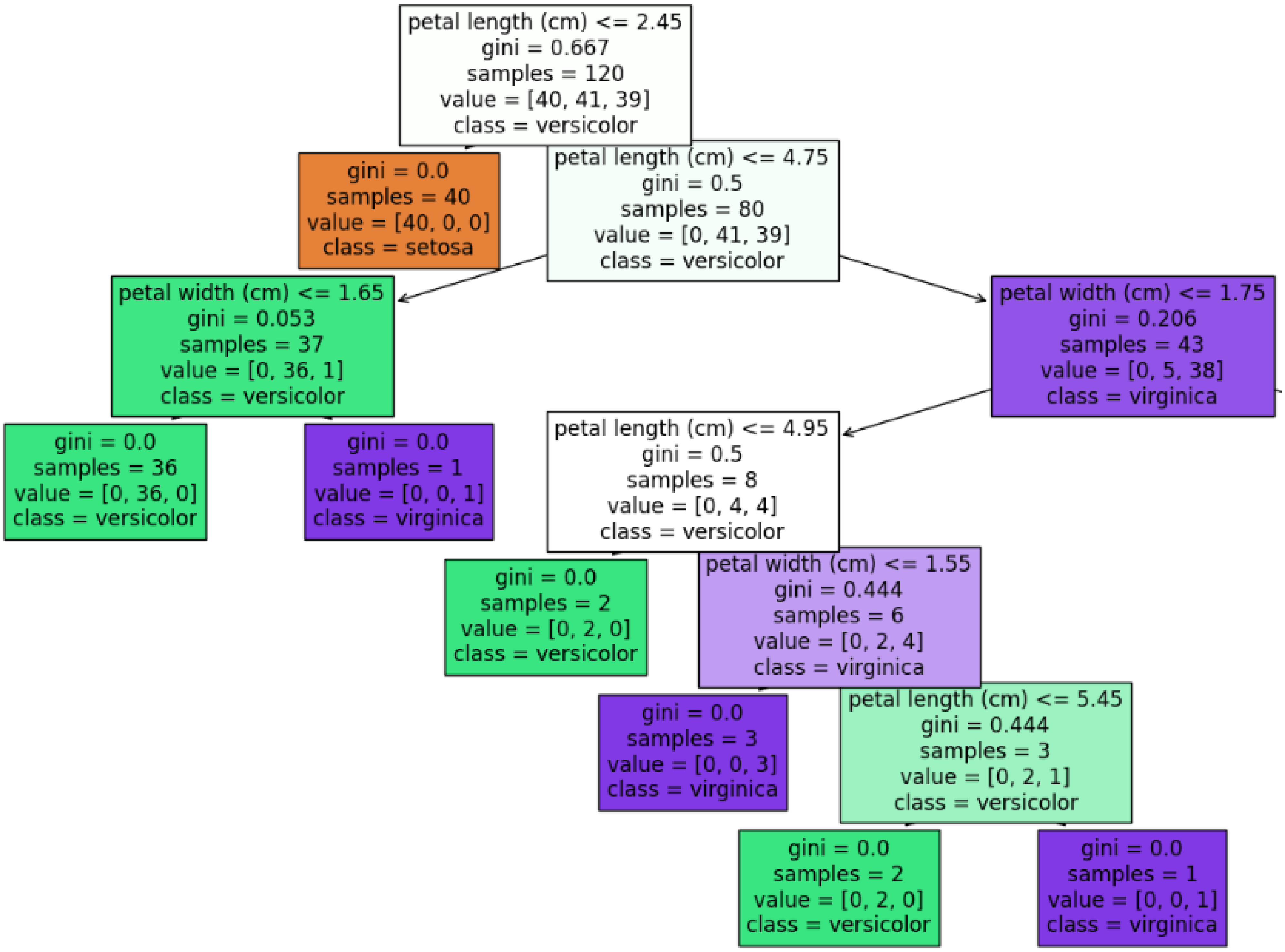

Figure 1.13: Excerpt of the Decision Tree plotted by ChatGPT

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 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 comprehension of transformers, their functionalities, evaluating their limitations, and enhancing our skill set.**The role of artificial intelligence 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.Artificial intelligence specialists must adapt to this new era of increasingly automated tasks, including transformer model implementations. Artificial intelligence 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 artificial intelligence specialist. An AI specialist can be an AI guru, providing design ideas, explanations, and implementations.The pragmatic definition of what a transformer represents for an artificial intelligence specialist will vary with the ecosystem.Let's go through a few examples:

- **API:** The OpenAI API does not require an AI developer. A web designer can create a form, and a linguist or **Subject Matter Expert (SME)** can prepare the prompt input texts. The primary role of an AI specialist will require linguistic skills to show, not just tell, the 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 artificial intelligence 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 artificial intelligence developer working with a linguist, for example, can play a crucial role. Therefore, computational linguistics training can come in very handy at this level.

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

- Large Language Models (LLMs) 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 artificial intelligence 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 artificial intelligence professionals will expand into many specializations.

The Future of Artificial Intelligence Professionals

The societal impact of foundation models should not be underestimated. Prompt engineering has become a skill required for artificial intelligence 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.AI specializes. Here are some domains an AI professional can invest in becoming some of the following domains:

- **AI Specialist** have specialized knowledge or skills in one of the fields of artificial intelligence. 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** will focus on data, big data, and model architecture to provide information for other teams.
- **AI Researcher**s will focus on research in a university or private company.
- **AI Engineers** will design AI, build systems, and implement AI models.
- **AI Manager** leverages the critical skills of project management, product management, and resource management(human, machine, software).

Many other fields will appear that show that there are many opportunities in AI to develop assets: datasets, models, applicationsTransformers 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 with Hugging Face. You might get an adverse reaction from a manager who may say, "*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 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 an SME (Subject Matter 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, GPT-4 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. Skillset 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 artificial intelligence. Microsoft Azure, Google Cloud, **Amazon Web Services (AWS)**, and IBM, among others, offer AI services that no developer or team of developers could hope to outperform. Tech giants have million-dollar supercomputers with massive datasets to train transformer models and AI models in general.Big tech giants have 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 **SaaS (Software as a Service)** API for some of the most effective transformer models on the market.OpenAI transformer models are so effective and humanlike that the present policy requires a potential user to 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
5. 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
```

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
```

And that's it! 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 behaviour 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 Google BARD*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 0sentence is, "A user visited the AllenNLP website, tried a transformer model, and found it interesting." Google Translate produces the following translation:

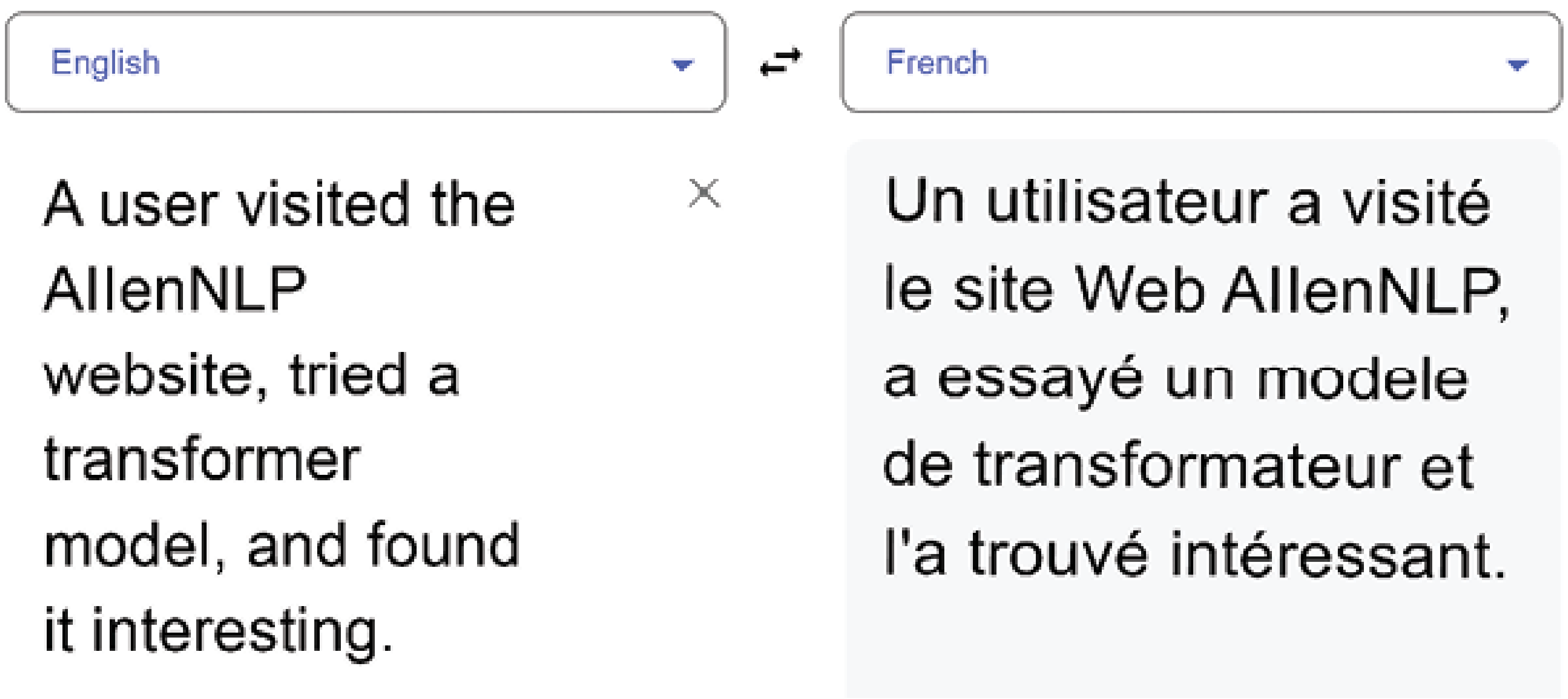

Figure 1.14: Coreference resolution in a translation using Google Translate

Google Translate appears to have solved the coreference resolution, but the word *transformateur* in French means an electric device. The word *transformer* is a neologism (new word) in French. An artificial intelligence specialist might be required to have language and linguistic skills for a specific project. Significant development is not required in this case. However, the project might require clarifying the input before requesting a translation.This example shows that you might have to team up with a linguist or acquire linguistic skills to work on an input context. In addition, it might take 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.

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 Open AI GPT-4, Google PaLM, and the new models they will continually produce. Foundation models are often proprietary, meaning we cannot access their code, though 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 artificial intelligence 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 .6NLPRNNs 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 artificial intelligence industrialization. We first saw how the O(1) time complexity of the attention layers and their computational time complexity, O, 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 artificial intelligence 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 open 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, for example. (True/False)
6. A ready-to-use transformer API can satisfy all needs. (True/False)
7. A company will accept the transformer ecosystem a developer knows best. (True/False)
8. Cloud transformers have become mainstream. (True/False)
9. A transformer project can be run on a laptop. (True/False)
10. Artificial intelligence specialists will have to be more flexible (True/False)

References

Bommansani et al. 2021, *On the Opportunities and Risks of Foundation Models*, https://arxiv.org/abs/2108.07258

Bommansani et al. 2023, Ecosystem Graphs: The Social Footprint of Foundation Models, https://arxiv.org/abs/2303.15772

Vaswani et al.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

GPTs are GPTs: An Early Look at the Labor Market Impact Potential of Large Language Models, Eloundou et al. (2023): https://arxiv.org/abs/2303.10130

Coopetition, standardization, and general purpose technologies: A framework and an application, Heikkilä et al.(2023): 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/