



# OpenAI and ChatGPT – Beyond the Market Hype

This chapter provides an overview of OpenAI and its most notable development—ChatGPT, highlighting its history, technology, and capabilities.

The overall goal is to provide a deeper knowledge of how ChatGPT can be used in various industries and applications to improve communication and automate processes and, finally, how those applications can impact the world of technology and beyond.

We will cover all this with the following topics:

What is OpenAI?

Overview of OpenAI model families

Road to ChatGPT: the math of the model behind it

ChatGPT: the state of the art

---

## Technical requirements

In order to be able to test the example in this chapter, you will need the following:

An OpenAI account to access the Playground and the Models API  
(<https://openai.com/api/login20>)

Your favorite IDE environment, such as Jupyter or Visual Studio

Python 3.7.1+ installed (<https://www.python.org/downloads>)

`pip` installed (<https://pip.pypa.io/en/stable/installation/>)

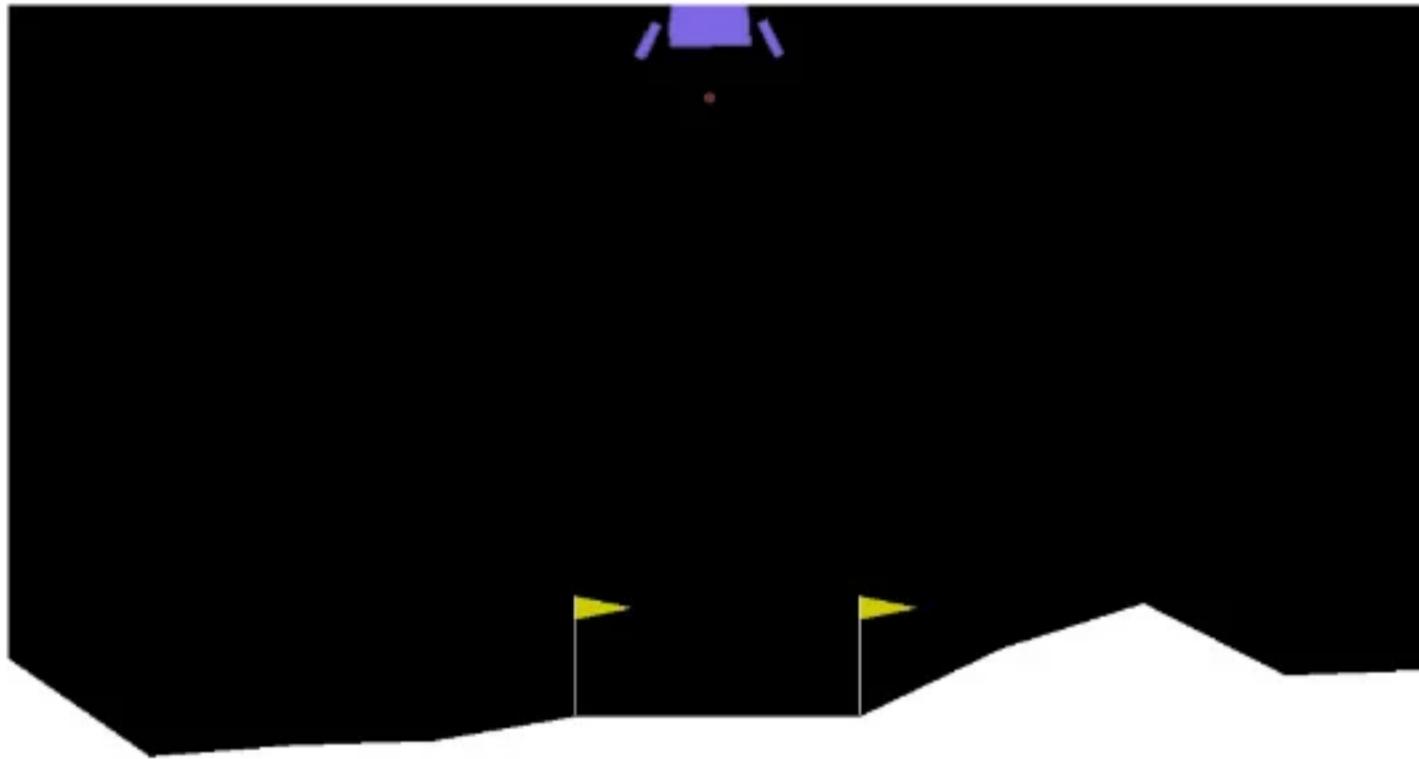
OpenAI Python library (<https://pypi.org/project/openai/>)

# What is OpenAI?

OpenAI is a research organization founded in 2015 by Elon Musk, Sam Altman, Greg Brockman, Ilya Sutskever, Wojciech Zaremba, and John Schulman. As stated on the OpenAI web page, its mission is “*to ensure that Artificial General Intelligence (AGI) benefits all of humanity*”. As it is *general*, AGI is intended to have the ability to learn and perform a wide range of tasks, without the need for task-specific programming.

Since 2015, OpenAI has focused its research on **Deep Reinforcement Learning (DRL)**, a subset of **machine learning (ML)** that combines **Reinforcement Learning (RL)** with deep neural networks. The first contribution in that field traces back to 2016 when the company released OpenAI Gym, a toolkit for researchers to develop and test RL algorithms.

# Gym is a standard API for reinforcement learning, and a diverse collection of reference environments



The Gym interface is simple, pythonic, and capable of representing general RL problems:

```
import gym
env = gym.make("LunarLander-v2", render_mode="human")
observation, info = env.reset(seed=42)
for _ in range(1000):
    action = policy(observation) # User-defined policy function
    observation, reward, terminated, truncated, info = env.step(action)

    if terminated or truncated:
        observation, info = env.reset()
env.close()
```

Figure 2.1 – Landing page of Gym documentation (<https://www.gymlibrary.dev/>)

OpenAI kept researching and contributing in that field, yet its most notable achievements are related to generative models—**Generative Pre-trained Transformers (GPT)**.

After introducing the model architecture in their paper “*Improving Language Understanding by Generative Pre-Training*” and baptizing it **GPT-1**, OpenAI researchers soon released, in 2019, its successor, the GPT-2. This version of the GPT was trained on a corpus called **WebText**, which at the time contained slightly over 8

million documents with a total of 40 GB of text from URLs shared in Reddit submissions with at least 3 upvotes. It had 1.2 billion parameters, ten times as many as its predecessor.

Here, you can see the landing page of a UI of GPT-2 published by HuggingFace (<https://transformer.huggingface.co/doc/distil-gpt2>):

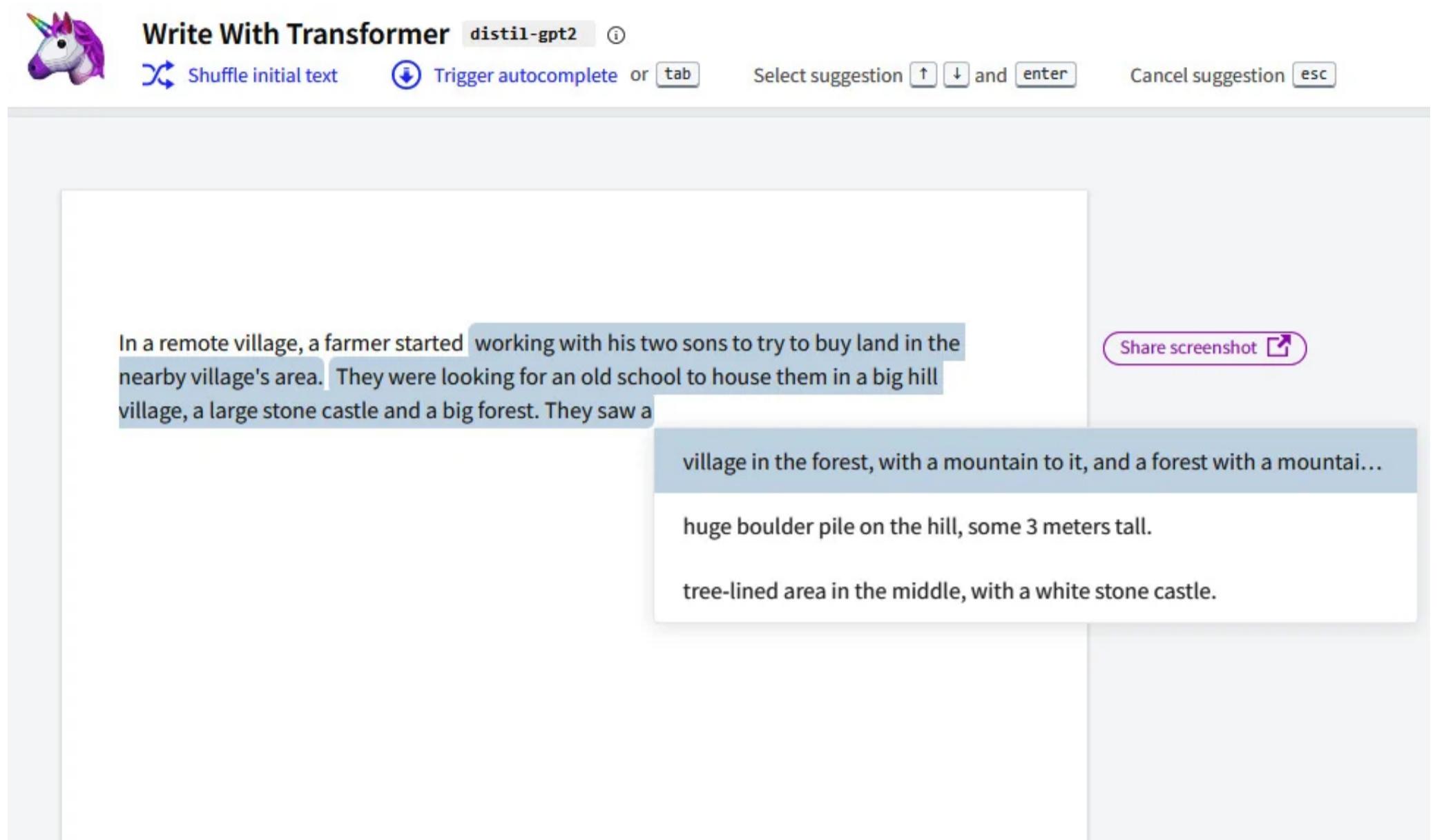


Figure 2.2 – GPT-2 writing a paragraph based on a prompt. Source: <https://transformer.huggingface.co/doc/distil-gpt2>

Then, in 2020, OpenAI first announced and then released GPT-3, which, with its 175 billion parameters, dramatically improved benchmark results over GPT-2.

In addition to natural language generative models, OpenAI also developed in the field of image generation, releasing its first model in that field, called **DALL-E**, revealed in 2021. As mentioned in the previous chapter, DALL-E is capable of creating brand new images from a natural language input, which is interpreted by the latest version of GPT-3.

DALL-E saw a recent upgrade to its new version, DALL-E 2, announced in April 2022.

In the following figure, you can see an example of images generated by DALL-E starting with the natural language prompt **generate a realistic picture of a cup of coffee in a cozy environment**:

generate a realistic picture of a cup of coffee in a cozy environment

Generate

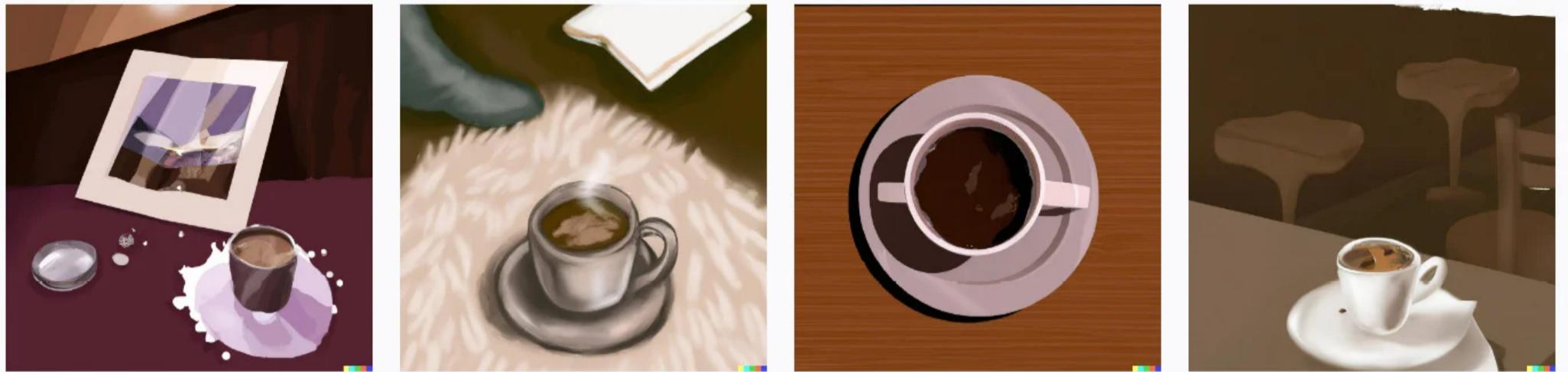


Figure 2.3 – Images generated by DALL-E with a natural language prompt as input

You can try generating creative pictures yourself in the lab of OpenAI DALL-E (<https://labs.openai.com/>), where you will get limited free credits to experiment.

Although OpenAI has invested in many fields of Generative AI, its contribution to text understanding and generation has been outstanding, thanks to the development of the foundation GPT models we are going to explore in the following paragraphs.

## An overview of OpenAI model families

Today, OpenAI offers a set of pre-trained, ready-to-use models that can be consumed by the general public. This has two important implications:

Powerful foundation models can be consumed without the need for long and expensive training

It's not necessary to be a data scientist or an ML engineer to manipulate those models

Users can test OpenAI models in OpenAI Playground, a friendly user interface where you can interact with models without the need to write any code.

In the following screenshot, you can see the landing page of the OpenAI Playground:



## Get started



Enter an instruction or select a preset, and watch the API respond with a **completion** that attempts to match the context or pattern you provided.

You can control which **model** completes your request by changing the model.

## KEEP IN MIND

- 👉 Use good judgment when sharing outputs, and attribute them to your name or company. [Learn more](#).
- ⚠ Requests submitted to our API will not be used to train or improve future models. [Learn more](#).
- ⚠ Our default models' training data cuts off in 2021, so they may not have knowledge of current events.

## Playground

Load a preset...

Save

View code

Share



Write a tagline for an ice cream shop.

## Mode

Complete

## Model

text-davinci-003

Temperature 0.7

Maximum length 256

Stop sequences  
Enter sequence and press Tab

Top P 1

Frequency penalty 0

Presence penalty 0

Best of 1

## Inject start text

## Inject restart text

Show probabilities

Looking for ChatGPT? [Try it now](#)

Submit



x

0

Figure 2.4 – OpenAI Playground at <https://platform.openai.com/playground>

As you can see from *Figure 2.4*, the Playground offers a UI where the user can start interacting with the model, which you can select on the right-hand side of the UI. To start interacting with the Playground, you can just type any questions or instructions in the input space in natural language. You can also start with some examples available in the OpenAI documentation (<https://platform.openai.com/examples>).

Before diving deeper into model families, let's first define some jargon you will see in this chapter:

**Tokens:** Tokens can be considered as word fragments or segments that are used by the API to process input prompts. Unlike complete words, tokens may contain trailing spaces or even partial sub-words. To better understand the concept of tokens in terms of length, there are some general guidelines to keep in mind. For instance, one token in English is approximately equivalent to four characters, or three-quarters of a word.

**Prompt:** In the context of natural language processing (NLP) and ML, a prompt refers to a piece of text that is given as input to an AI language model to generate a response or output. The prompt can be a question, a statement, or a sentence, and it is used to provide context and direction to the language model.

**Context:** In the field of GPT, context refers to the words and sentences that come before the user's prompt. This context is used by the language model to generate the most probable next word or phrase, based on the patterns and relationships found in the training data.

**Model confidence:** Model confidence refers to the level of certainty or probability that an AI model assigns to a particular prediction or output. In the context of NLP, model confidence is often used to indicate how confident the AI model is in the correctness or relevance of its generated response to a given input prompt.

The preceding definitions will be pivotal in understanding how to use Azure OpenAI model families and how to configure their parameters.

In the Playground, there are two main models families that can be tested:

**GPT-3:** A set of models that can understand and generate natural language. GPT-3 has been trained on a large corpus of text and can perform a wide range of natural language tasks such as language translation, summarization, question-answering, and more. Here is an example:

Summarize this for a second-grade student:



Jupiter is the fifth planet from the Sun and the largest in the Solar System. It is a gas giant with a mass one-thousandth that of the Sun, but two-and-a-half times that of all the other planets in the Solar System combined. Jupiter is one of the brightest objects visible to the naked eye in the night sky, and has been known to ancient civilizations since before recorded history. It is named after the Roman god Jupiter.[19] When viewed from Earth, Jupiter can be bright enough for its reflected light to cast visible shadows,[20] and is on average the third-brightest natural object in the night sky after the Moon and Venus.

Jupiter is a giant planet that is very bright and can be seen in the night sky. It is the fifth planet from the Sun and is named after the Roman god Jupiter. It is much bigger than all the other planets in the Solar System combined. People have known about Jupiter for a very long time.

Looking for ChatGPT?

Try it now



Figure 2.5 – An example of a summarization task with GPT-3

**GPT-3.5:** This is a newer set of models that build upon GPT-3 and aim to improve its natural language understanding and generation abilities. GPT-3.5 models can perform complex natural language tasks such as composing coherent paragraphs or essays, generating poetry, and even creating computer programs in natural language. GPT-3.5 is the model behind ChatGPT and, on top of its API, it is also consumable within the Playground with a dedicated UI:

The screenshot shows the ChatGPT playground interface. On the left, a user message reads "List me the 10 countries with the highest population". To the right, an assistant message lists the top 10 countries by population: 1. China, 2. India, 3. United States, 4. Indonesia, 5. Pakistan, 6. Brazil, 7. Nigeria, 8. Bangladesh, 9. Russia, 10. Mexico. The interface includes several configuration options on the right: Mode (Chat, Beta), Model (gpt-3.5-turbo), Temperature (0.7), Maximum length (256), Top P (1), Frequency penalty (0), and Presence penalty (0).

USER List me the 10 countries with the highest population

ASSISTANT 1. China  
2. India  
3. United States  
4. Indonesia  
5. Pakistan  
6. Brazil  
7. Nigeria  
8. Bangladesh  
9. Russia  
10. Mexico

Mode Chat Beta

Model gpt-3.5-turbo

Temperature 0.7

Maximum length 256

Top P 1

Frequency penalty 0

Presence penalty 0

Add message

## Figure 2.6 – An example of interaction with GPT-3.5

**Codex:** A set of models that can understand and generate code in various programming languages. Codex can translate natural language prompts into working code, making it a powerful tool for software development. Here is an example using Codex:

```
1 """
2 1. Create a list of first names
3 2. Create a list of last names
4 3. Combine them randomly into a list of 100 full names
5 """
6
7 import random
8
9 first_names = ['John', 'Jane', 'Corey', 'Travis', 'Dave', 'Kurt', 'Neill', 'Sam', 'Steve', 'Tom', 'James', 'Robert', 'Michael', 'Charles', 'Joe', 'Mary', 'Maggie', 'Nicole', 'Patricia', 'Linda', 'Barbara', 'Elizabeth', 'Laura', 'Jennifer', 'Maria']
10
11 last_names = ['Smith', 'Doe', 'Jenkins', 'Robinson', 'Davis', 'Stuart', 'Jefferson', 'Jacobs', 'Wright', 'Patterson', 'Wilks', 'Arnold', 'Johnson', 'Williams', 'Jones', 'Brown', 'Davis', 'Miller', 'Wilson', 'Moore', 'Taylor', 'Anderson', 'Thomas', 'Jackson', 'White', 'Harris', 'Martin']
12
13 full_names = []
14
15 for i in range(100):
16     first = random.choice(first_names)
17     last = random.choice(last_names)
18     full_names.append(f"{first} {last}")
19
20 print(full_names)
```

## Figure 2.7 – An example of code generation with Codex

### Note

In March 2023, OpenAI announced that Codex models will be deprecated from that date on. This is because of the incredible capabilities of the new chat models (including GPT-3.5-turbo, the model behind ChatGPT) that encompass coding tasks as well, with results that benchmark or even surpass Codex models' ones.

For each model, you can also play with some parameters that you can configure. Here is a list:

**Temperature (ranging from 0 to 1):** This controls the randomness of the model's response. A low-level temperature makes your model more deterministic, meaning that it will tend to give the same output to the same question. For example, if I ask my model multiple times, "*What is OpenAI?*" with temperature set to 0, it will always give the same answer. On the other hand, if I do the same with a model with temperature set to 1, it will try to modify its answers each time in terms of wording and style.

**Max length (ranging from 0 to 2048):** This controls the length (in terms of tokens) of the model's response to the user's prompt.

**Stop sequences (user input):** This makes responses end at the desired point, such as the end of a sentence or list.

**Top probabilities** (ranging from 0 to 1): This controls which tokens the model will consider when generating a response. Setting this to 0.9 will consider the top 90% most likely of all possible tokens. One could ask, "*Why not set top probabilities as 1 so that all the most likely tokens are chosen?*" The answer is that users might still want to maintain variety when the model has low confidence, even in the highest-scoring tokens.

**Frequency penalty** (ranging from 0 to 1): This controls the repetition of the same tokens in the generated response. The higher the penalty, the lower the probability of seeing the same tokens more than once in the same response. The penalty reduces the chance proportionally, based on how often a token has appeared in the text so far (this is the key difference from the following parameter).

**Presence penalty** (ranging from 0 to 2): This is similar to the previous one but stricter. It reduces the chance of repeating any token that has appeared in the text at all so far. As it is stricter than the frequency penalty, the presence penalty also increases the likelihood of introducing new topics in a response.

**Best of** (ranging from 0 to 20): This generates multiple responses and displays only the one with the best total probability across all its tokens.

**Pre- and post-response text** (user input): This inserts text before and after the model's response. This can help prepare the model for a response.

Besides trying OpenAI models in the Playground, you can always call the models API in your custom code and embed models into your applications. Indeed, in the right corner of the Playground, you can click on **View code** and export the configuration as shown here:

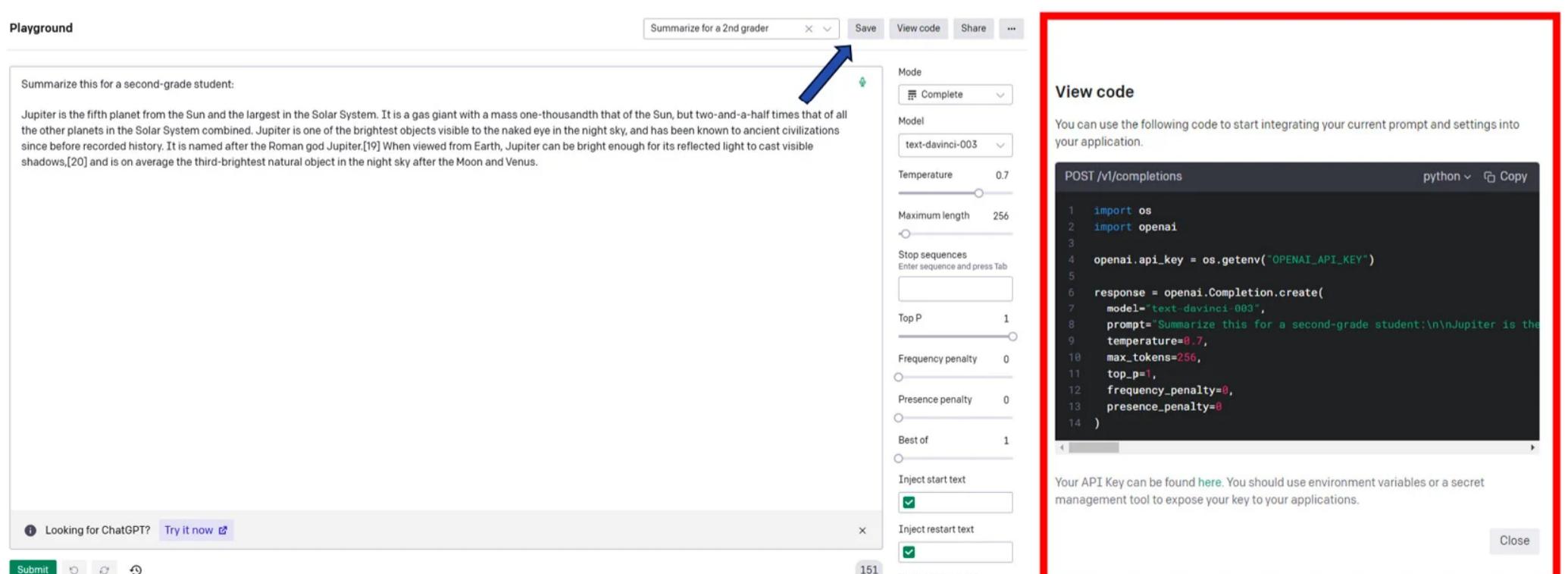


Figure 2.8 – Python code for calling a GPT3 model with a natural language prompt

As you can see from the preceding screenshot, the code exports the parameter configuration you set in the Playground.

Now you can start using the OpenAI library in Python by installing it via pip `install openai` in your terminal. In order to use the models, you will need to generate an API key. You can find your API keys (<https://platform.openai.com/account/api-keys>) in your account settings, as shown here:

The screenshot shows the 'API keys' section of the OpenAI account settings. On the left, there's a sidebar with 'ORGANIZATION' and 'USER' sections. Under 'USER', 'API Keys' is selected and highlighted in green. The main content area has a heading 'API keys'. It says: 'Your secret API keys are listed below. Please note that we do not display your secret API keys again after you generate them.' Below this is a note: 'Do not share your API key with others, or expose it in the browser or other client-side code. In order to protect the security of your account, OpenAI may also automatically rotate any API key that we've found has leaked publicly.' A table lists two API keys:

SECRET KEY	CREATED	LAST USED	
sk-...cyzM	3 feb 2023	3 feb 2023	
sk-...X4U8	9 feb 2023	9 feb 2023	

A button '+ Create new secret key' is at the bottom of the table. Below the table is a section titled 'Default organization' with a dropdown menu set to 'Personal'. A note at the bottom says: 'Note: You can also specify which organization to use for each API request. See [Authentication](#) to learn more.'

Figure 2.9 – API keys in the account settings page of your OpenAI profile

With OpenAI APIs, you can also try the following additional model families that are not available in the Playground:

**Moderation:** This is a fine-tuned model developed by OpenAI that can detect potentially sensitive or unsafe text content. Moderation uses ML algorithms to classify text as safe or unsafe based on its context and language use. This model can be used to automate content moderation on social media platforms, online communities, and in many other domains. There are multiple categories, such as hate, hate/threatening, self-harm, sexual, sexual/minors, violence, violence/graphic.

Here is example code for the Moderation API:

[Copy](#)[Explain](#)

```
import os
import openai
openai.api_key = os.getenv("OPENAI_API_KEY")
openai.Moderation.create(
    input="I want to kill him",
)
```

The output for this is as follows:

[Copy](#)[Explain](#)

```
<OpenAIObject id=modr-6sHusuY9frxJdfqTBXHs0AfWhckrh at 0x218bd8482c0> JSON: {
    "id": "modr-6sHusuY9frxJdfqTBXHs0AfWhckrh",
    "model": "text-moderation-004",
    "results": [
        {
            "categories": {
                "hate": false,
                "hate/threatening": false,
                "self-harm": false,
                "sexual": false,
                "sexual/minors": false,
                "violence": true,
                "violence/graphic": false
            },
            "category_scores": {
                "hate": 1.7164344171760604e-05,
                "hate/threatening": 2.614225103059198e-08,
                "self-harm": 2.5988580176772302e-08,
                "sexual": 2.8184256279928377e-06,
                "sexual/minors": 9.1383149936064e-09,
                "violence": 0.9910049438476562,
                "violence/graphic": 5.316753117767803e-07
            },
            "flagged": true
        }
    ]
}
```

In this case, the Moderator API detected evidence of violent content.

**Embeddings:** Some models can use embeddings. These embeddings involve representing words or sentences in a multi-dimensional space. The mathematical distances between different instances in this space represent their similarity in terms of meaning. As an example, imagine the words queen,

woman, king, and man. Ideally, in our multidimensional space, where words are vectors, if the representation is correct, we want to achieve the following:

$$\vec{\text{king}} + (\vec{\text{woman}} - \vec{\text{man}}) \approx \vec{\text{queen}}$$

Figure 2.10 – Example of vectorial equations among words

This means that the distance between *woman* and *man* should be equal to the distance between *queen* and *king*. Here is an example of an embedding:

[Copy](#)

[Explain](#)

```
import openai
embedding = openai.Embedding.create(
    input="The cat is on the table",
    model="text-embedding-ada-002")["data"][0]["embedding"]
```

The preceding method creates a vector representation of the input. We can have a look at the first 10 vectors of the output here:

[Copy](#)

[Explain](#)

```
embedding[1:10]
[-0.01369840931147337,
 -0.007505378685891628,
 -0.002576263388618827,
 -0.014773285016417503,
 0.019935185089707375,
 -0.01802290789783001,
 -0.01594814844429493,
 -0.0010944041423499584,
 -0.014323337003588676]
```

Embeddings can be extremely useful in intelligent search scenarios. Indeed, by getting the embedding of the user input and the documents the user wants to search, it is possible to compute distance metrics (namely, cosine similarity) between the input and the documents. By doing so, we can retrieve the documents that are *closer*, in mathematical distance terms, to the user input.

**Whisper:** This is a speech recognition model that can transcribe audio into text. Whisper can recognize and transcribe various languages and dialects with high accuracy, making it a valuable tool for automated speech recognition systems. Here is an example:

[Copy](#)[Explain](#)

```
# Note: you need to be using OpenAI Python v 0.27.0 for the code below to work
```

[Copy](#)[Explain](#)

```
import openai
```

[Copy](#)[Explain](#)

```
openai.api_key = os.getenv("OPENAI_API_KEY")
```

[Copy](#)[Explain](#)

```
audio_file= open("/path/to/file/audio.mp3", "rb")
```

[Copy](#)[Explain](#)

```
transcript = openai.Audio.transcribe("whisper-1", audio_file)
```

The output looks like the following:

[Copy](#)[Explain](#)

```
{"text": Yes, hi, I just noticed a dent on the side of my car and I have no idea how it got there. There were no witnesses around and I'm really frustrated.
```

```
....
```

```
}
```

All the previous models come as pre-built, in the sense that they have already been pre-trained on a huge knowledge base.

However, there are some ways you can make your model more customized and tailored for your use case.

The first method is embedded in the way the model is designed, and it involves providing your model with the context in the **few-learning approach** (we will focus on this technique later on in the book). Namely, you could ask the model to generate an article whose template and lexicon recall another one you have already written. For

this, you can provide the model with your query of generating an article *and* the former article as a reference or context, so that the model is better prepared for your request.

Here is an example of it:

The screenshot shows a text input field with the instruction "Summarize this for a second-grade student:" followed by a large block of text about Jupiter. Below the text, several generated responses are shown in green boxes: a summary, key phrases, their Italian translation, and a promotional message for ChatGPT.

Summarize this for a second-grade student:

Jupiter is the fifth planet from the Sun and the largest in the Solar System. It is a gas giant with a mass one-thousandth that of the Sun, but two-and-a-half times that of all the other planets in the Solar System combined. Jupiter is one of the brightest objects visible to the naked eye in the night sky, and has been known to ancient civilizations since before recorded history. It is named after the Roman god Jupiter.[19] When viewed from Earth, Jupiter can be bright enough for its reflected light to cast visible shadows,[20] and is on average the third-brightest natural object in the night sky after the Moon and Venus.

Jupiter is the fifth planet from the Sun and the biggest in our Solar System. It is very bright in the night sky and it has been known since ancient times. It is named after the Roman god Jupiter. It can be so bright that it casts shadows!

Extract key phrase from the original article

"Gas giant", "brightest objects visible to the naked eye", "casts visible shadows", "Roman god Jupiter"

translate those key phrase in italian

"Gigante gassoso", "oggetti più luminosi visibili ad occhio nudo", "getta ombre visibili", "Dio romano Giove".

Looking for ChatGPT? Try it now [↗](#)

Figure 2.11 – An example of a conversation within the OpenAI Playground with the few-shot learning approach

The second method is more sophisticated and is called **fine-tuning**. Fine-tuning is the process of adapting a pre-trained model to a new task.

In fine-tuning, the parameters of the pre-trained model are altered, either by adjusting the existing parameters or by adding new parameters, to better fit the data for the new task. This is done by training the model on a smaller labeled dataset that is specific to the new task. The key idea behind fine-tuning is to leverage the knowledge learned from the pre-trained model and fine-tune it to the new task, rather than training a model from scratch. Have a look at the following figure:

Dataset	# tokens	Proportion within training
Common Crawl	410 billion	60%
WebText2	19 billion	22%
Books1	12 billion	8%
Books2	55 billion	8%
Wikipedia	3 billion	3%

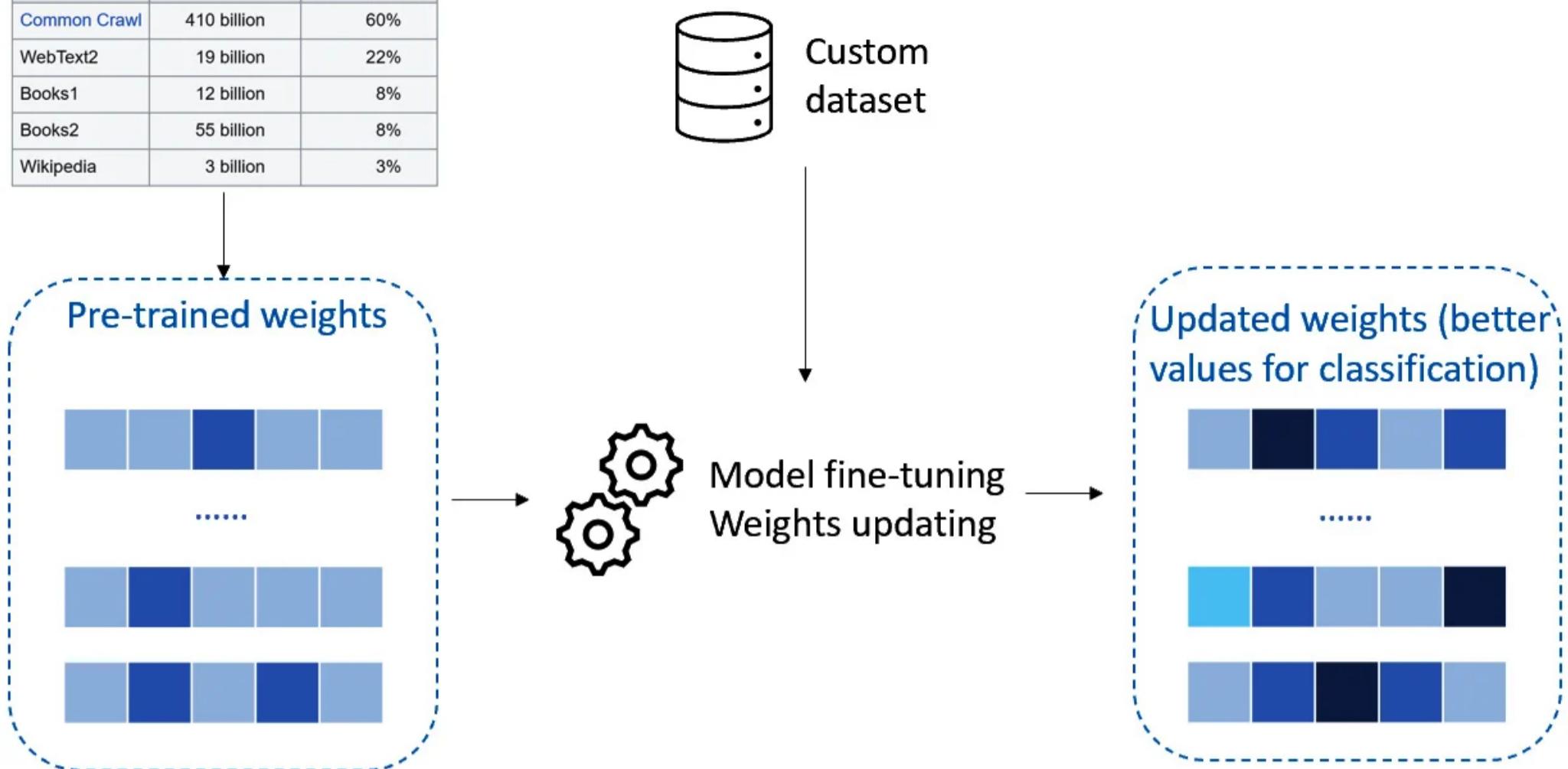


Figure 2.12 – Model fine-tuning

In the preceding figure, you can see a schema on how fine-tuning works on OpenAI pre-built models. The idea is that you have available a pre-trained model with general-purpose weights or parameters. Then, you feed your model with custom data, typically in the form of *key-value* prompts and completions as shown here:

```
{"prompt": "<prompt text>", "completion": "<ideal generated text>"}  
{"prompt": "<prompt text>", "completion": "<ideal generated text>"}  
{"prompt": "<prompt text>", "completion": "<ideal generated text>"}  
...
```

Copy

Explain

Once the training is done, you will have a customized model that performs particularly well for a given task, for example, the classification of your company's documentation.

The nice thing about fine-tuning is that you can make pre-built models tailored to your use cases, without the need to re-train them from scratch, yet leveraging smaller training datasets and hence less training time and computing. At the same time, the model keeps its generative power and accuracy learned via the original training, the one that occurred on the massive dataset.

In this paragraph, we got an overview of the models offered by OpenAI to the general public, from those you can try directly in the Playground (GPT, Codex) to more complex models such as embeddings. We also learned that, besides using models in

their pre-built state, you can also customize them via fine-tuning, providing a set of examples to learn from.

In the following sections, we are going to focus on the background of those amazing models, starting from the math behind them and then getting to the great discoveries that made ChatGPT possible.

---

## Road to ChatGPT: the math of the model behind it

Since its foundation in 2015, OpenAI has invested in the research and development of the class of models called **Generative Pre-trained Transformers (GPT)**, and they have captured everyone's attention as being the engine behind ChatGPT.

GPT models belong to the architectural framework of transformers introduced in a 2017 paper by Google researchers, *Attention Is All You Need*.

The transformer architecture was introduced to overcome the limitations of traditional **Recurrent Neural Networks (RNNs)**. RNNs were first introduced in the 1980s by researchers at the Los Alamos National Laboratory, but they did not gain much attention until the 1990s. The original idea behind RNNs was that of processing sequential data or time series data, keeping information across time steps.

Indeed, up to that moment in time, the classic **Artificial Neural Network (ANN)** architecture was that of the feedforward ANN, where the output of each hidden layer is the input of the next one, without maintaining information about past layers.

In order to understand the idea behind the transformer, we need to start from its origins. We will hence dwell on the following topics:

The structure of RNNs

RNNs' main limitations

How those limitations have been overcome with the introduction of new architectural elements, including positional encoding, self-attention, and the feedforward layer

How we got to the state of the art of GPT and ChatGPT

Let's start with the architecture of transformers' predecessors.

## The structure of RNNs

Let's imagine we want to predict a house price. If we had only today's price for it, we could use a feedforward architecture where we apply a non-linear transformation to the input via a hidden layer (with an activation function) and get as output the forecast of the price for tomorrow. Here is how:

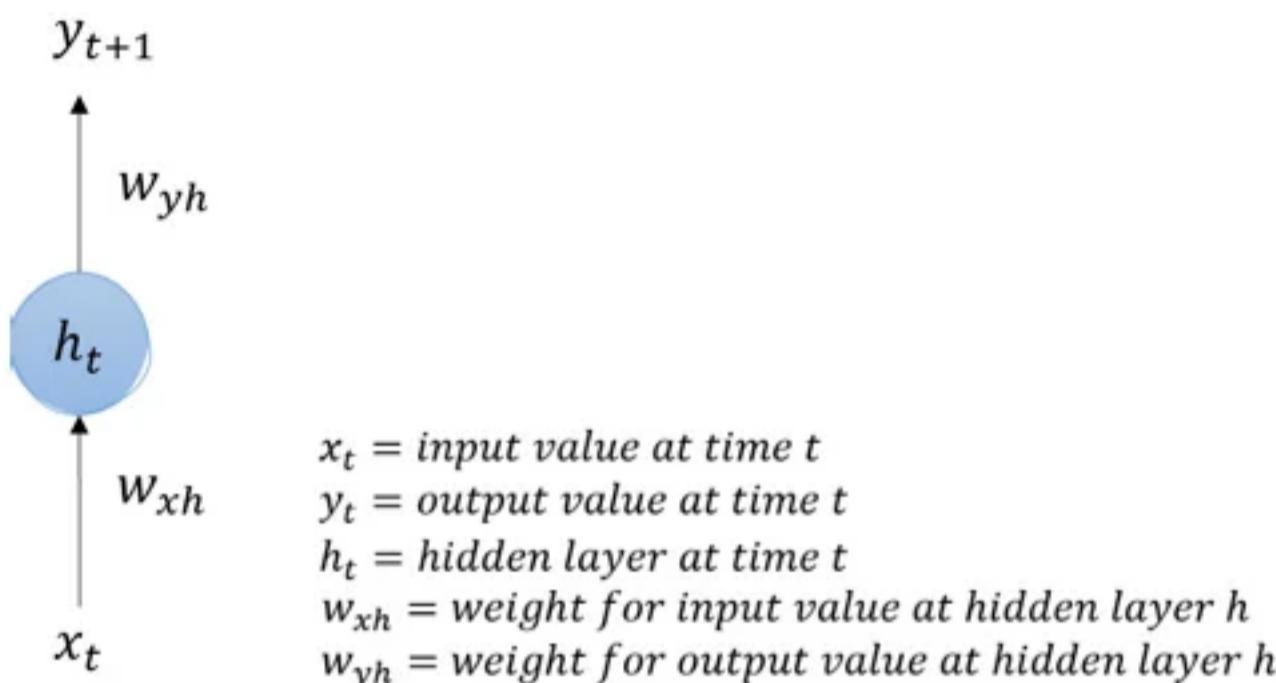


Figure 2.13 – Feedforward architecture with a hidden layer

However, for this kind of data, it is also likely to have the availability of longer sequences. For example, we might have the time series of this house for the next 5 years. Of course, we want to embed this extra information we have into our model so that our RNN is able to keep the memory about past input in order to properly interpret current input and forecast future outputs.

So, coming back to our example, imagine we not only have the price for today but also the price for yesterday ( $t-1$ ) and the day before ( $t-2$ ). This is how we can calculate it:

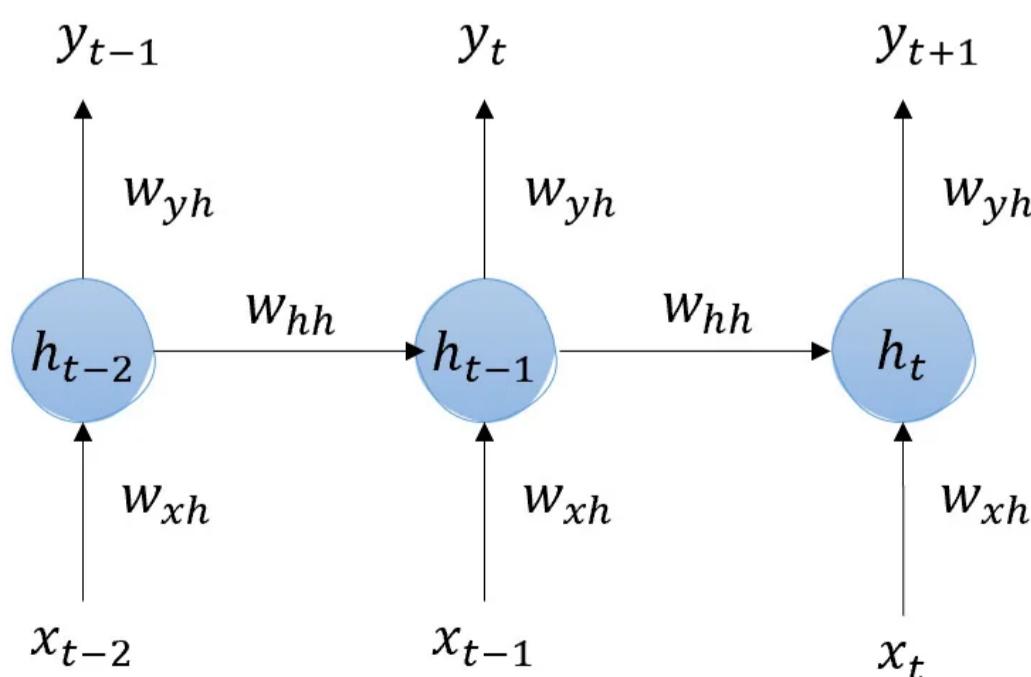


Figure 2.14 – Example of an RNN

Since we are only interested in tomorrow's price, let's ignore the intermediate final outputs for  $t-1$  and  $t$ .

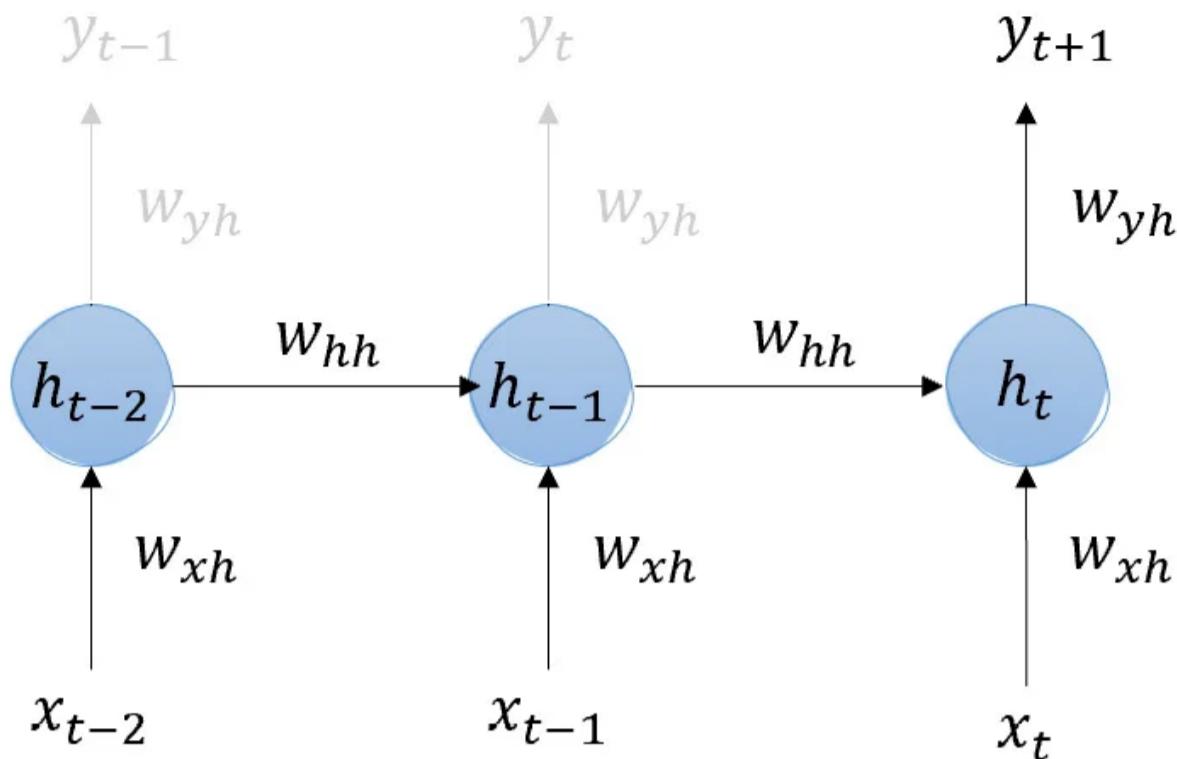


Figure 2.15 – Example of an RNN

As you can see, the output of the hidden layer of **t-2** is served as a (weighted) input of the hidden layer of **t-1**, which also takes the input of **t-1**. Then, the output of the hidden layer at **t-1**, which already keeps the memory of **t-2** and **t-1** inputs, is served as input to the hidden layer of **t**. As a result, the price for tomorrow ( $y_{t+1}$ ), which is the one we are interested in, brings the memory of all the previous days' inputs.

Finally, if we want to shrink this picture, we can think about the RNN as follows:

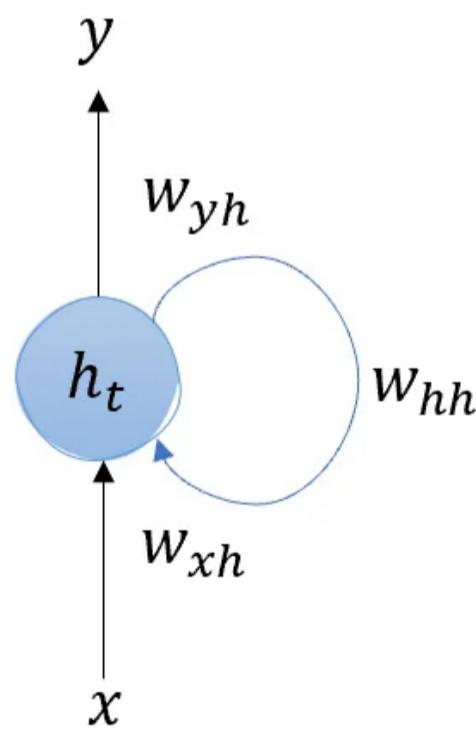


Figure 2.16 – Example of an RNN in its wrapped form

This means that the output of the RNN layer at time step  $t-n$  is then produced and passed as input to the next time step. The hidden state of the RNN layer is also passed as input to the next time step, allowing the network to maintain and propagate information across different parts of the input sequence.

Even though RNNs were a great development in the field of ANN, they still suffer from some limitations, which we are going to examine in the next section.

## The main limitations of RNNs

As mentioned in the introduction of this section, RNNs suffer from three main limitations:

**Gradient vanishing and exploding:** RNNs suffer from the problem of gradient vanishing and exploding, which makes it difficult to train the network effectively. This problem occurs because the gradients are multiplied multiple times during the backpropagation process, which can cause the gradients to become very small or very large.

**Limited context:** Traditional RNNs are only able to capture a limited amount of context, as they process the input sequence one element at a time. This means that they are not able to effectively process long-term dependencies or relationships between elements that are far apart in the input sequence.

**Difficulty in parallelization:** RNNs are inherently sequential, which makes it difficult to parallelize their computation, hence they do not make great use of today's **Graphical Processing Units (GPUs)**. This can make them slow to train and deploy on large-scale datasets and devices.

A first attempt to overcome the first two limitations (limited context and vanishing and exploding gradient) occurred in 1997 when a new architecture was introduced by Sepp Hochreiter and Jürgen Schmidhuber in their paper, *Long Short-term Memory*. Networks with this new architecture were then called **Long Short-Term Memory (LSTM)**.

LSTM networks overcome the problem of limited context by introducing the concept of a cell state, which is separate from the hidden state and is able to maintain information for much longer periods. The cell state is passed through the network unchanged, allowing it to store information from previous time steps that would otherwise be lost.

Furthermore, LSTM networks overcome the problem of vanishing and exploding gradients by using carefully designed gates to control the flow of information in and out of the cell, which helps to prevent gradients from becoming too small or too large.

However, LSTM networks still maintain the problem of lack of parallelization and hence slow training time (even slower than RNNs since they are more complex). The goal is to have a model that is able to use parallelization even on sequential data.

To overcome those limitations, a new framework was introduced.

## Overcoming limitations – introducing transformers

The transformer architecture addresses these limitations by replacing the recurrence (with a self-attention mechanism), allowing for parallel computation and capturing long-term dependencies.

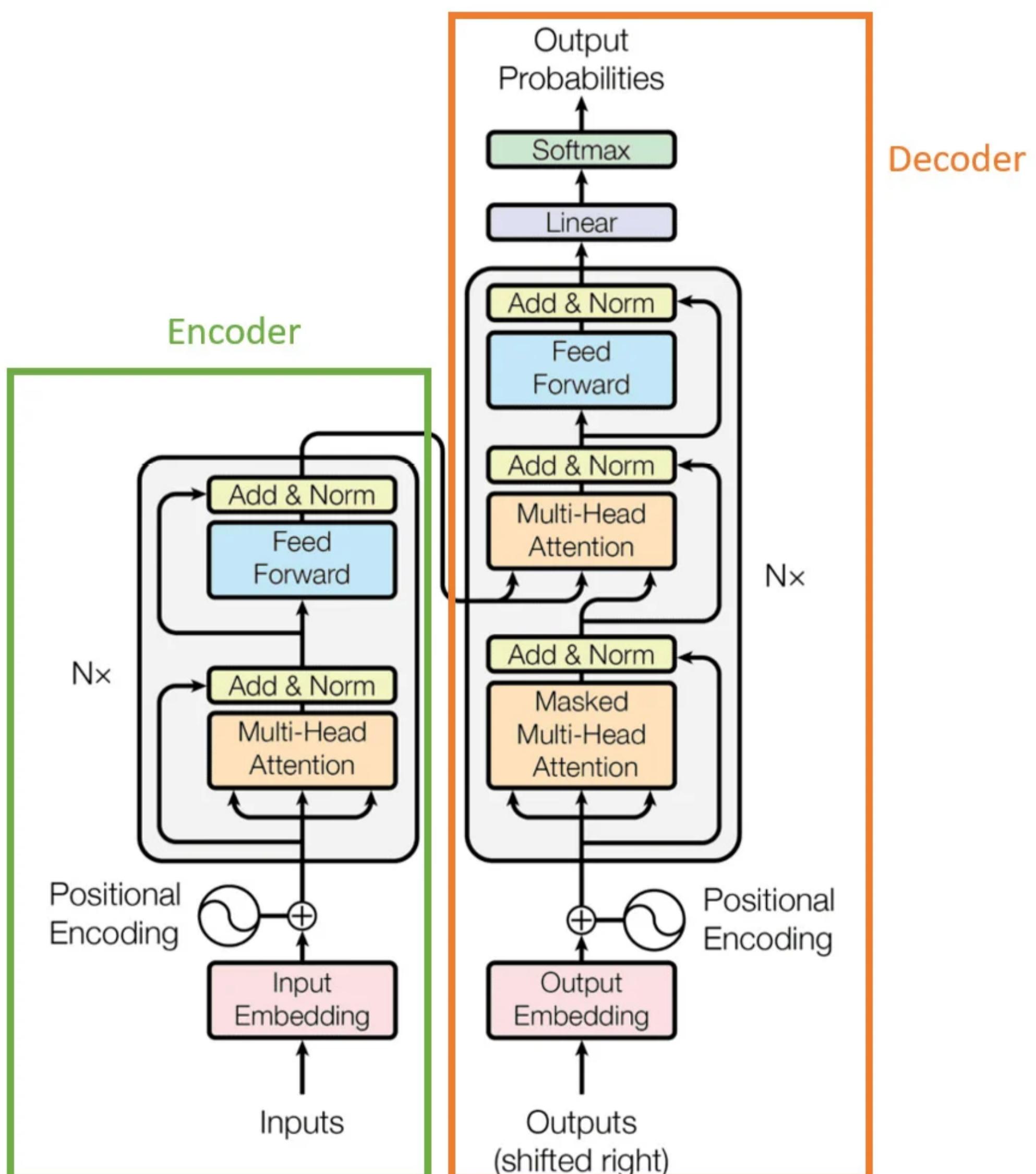


Figure 2.7 – Transformer architecture from the original paper, “Attention is all you need.” Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention Is All You Need. ArXiv.

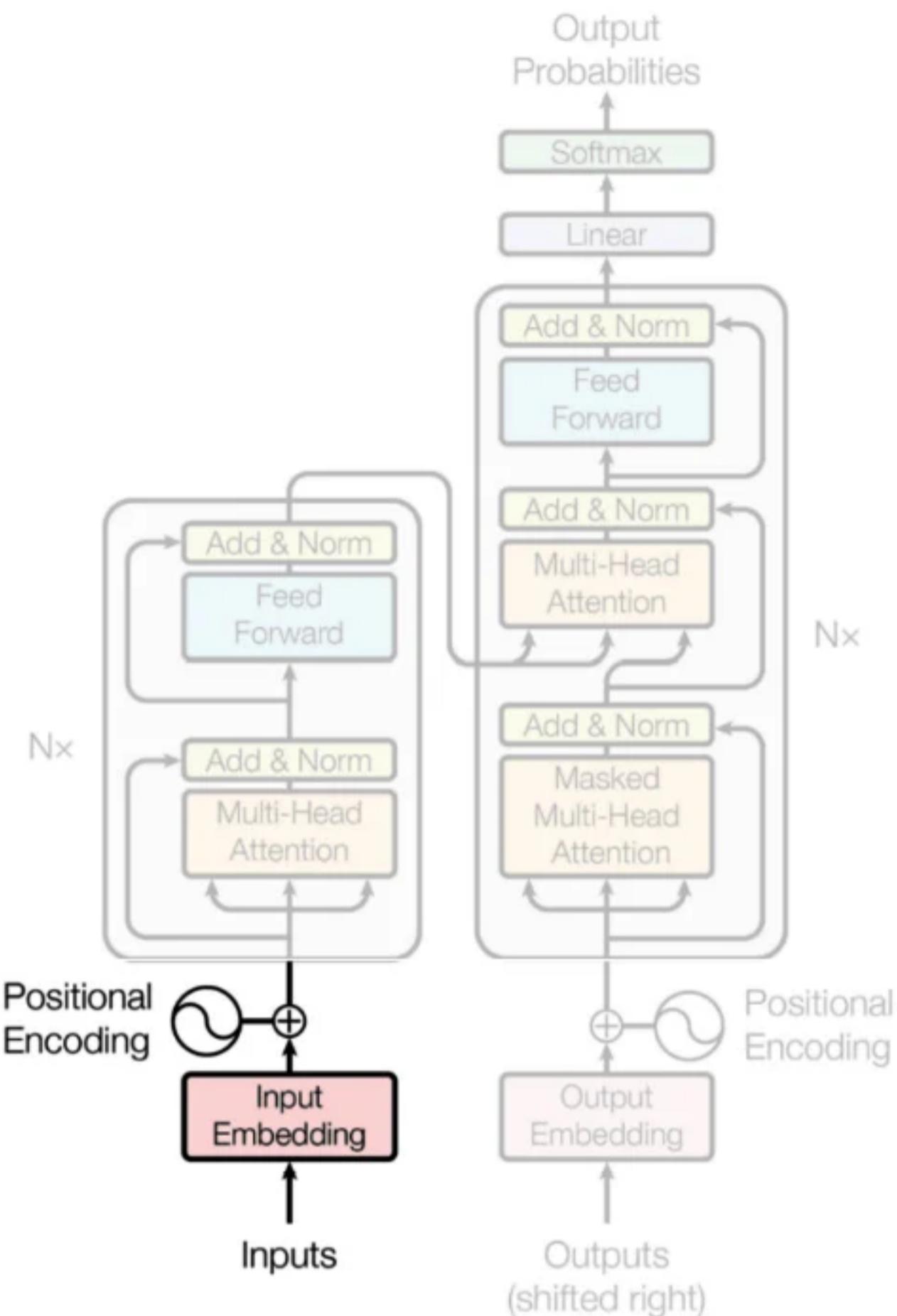
In the preceding figure (taken from the original paper), you can see that there are two main building blocks: on the left-hand side, we have the “encoder,” which has the task of representing the input in a lower-dimensional space; on the right-hand side, we have the “decoder,” which has the task of translating the lower-dimensional data provided by the encoder back to the original data format.

Both the encoder and the decoder share three main types of layers that distinguish the transformer architecture: positional encoding, self-attention, and feedforward.

Let us understand each of these in the following sections.

### Positional encoding

Encoders are layers that transform natural language input into numerical vectors. This is achieved thanks to the process of embedding, an NLP technique that represents words with vectors in such a way that once represented in a vectorial space, the mathematical distance between vectors is representative of the similarity among words they represent. Have a look at the following figure:



**Figure 2.18 – Transformer architecture from the original paper, “Attention is all you need.”** Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention Is All You Need. ArXiv.  
<https://doi.org/10.48550/arXiv.1706.03762>

As we talk about the meaning of sentences, we all agree that the arrangement of words in a sentence is significant in determining its meaning. That is the reason why we want our encoder to take into account that order, to be *positional*.

The positional encoding is a fixed, learned vector that represents the position of a word in the sequence. It is added to the embedding of the word so that the final representation of a word includes both its meaning and its position.

## Self-attention

Self-attention layers are responsible for determining the importance of each input token in generating the output. They answer the question, "Which part of the input should I focus on?"

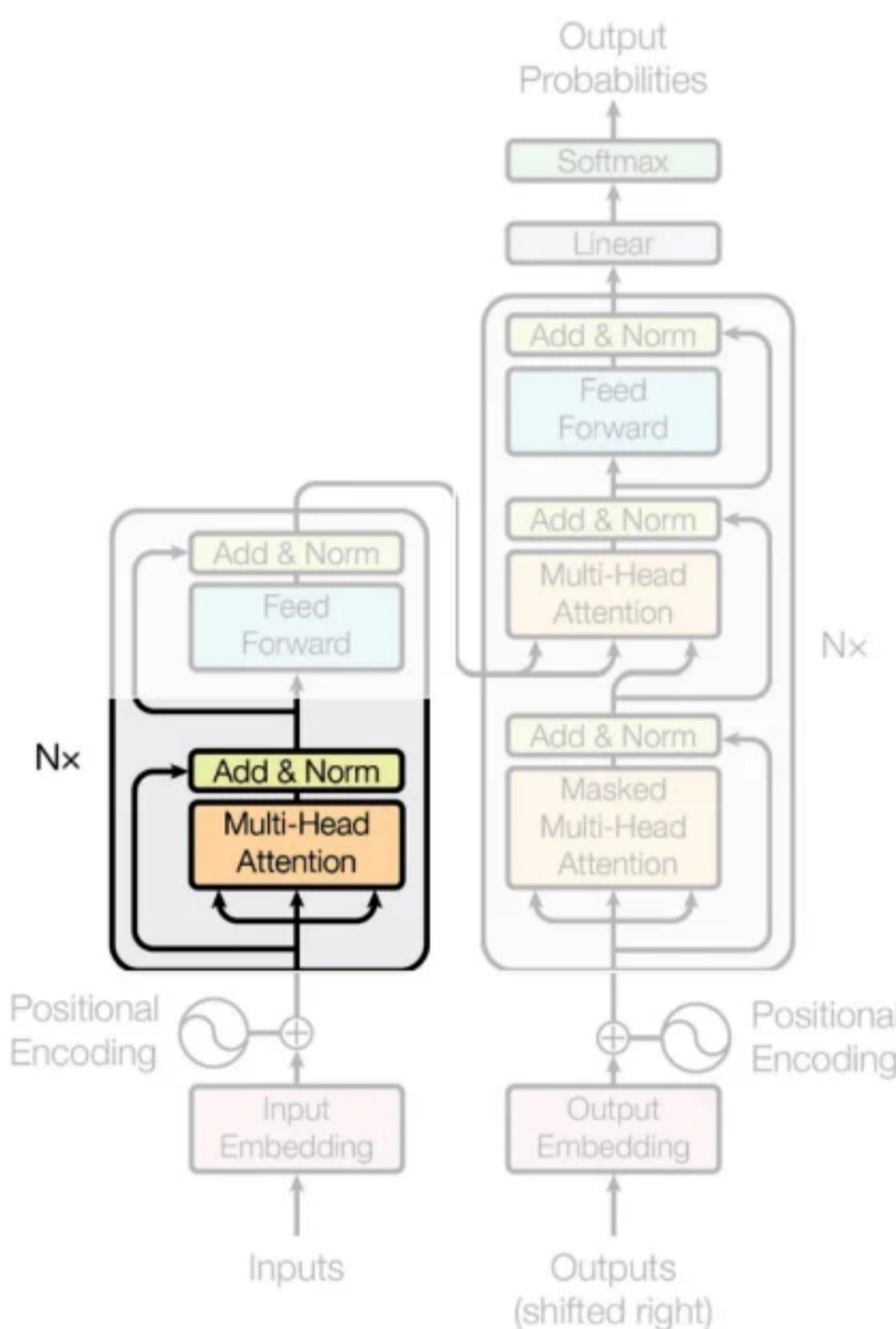


Figure 2.19 – Transformer architecture from the original paper, “Attention is all you need.” Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention Is All You Need. ArXiv.

<https://doi.org/10.48550/arXiv.1706.03762>

In order to obtain the self-attention vector for a sentence, the elements we need are **value**, **query**, and **key**. These matrices are used to calculate attention scores between the elements in the input sequence and are the three weight matrices that are learned during the training process (typically initialized with random values).

**Query** is used to represent the current focus of the attention mechanism, while **key** is used to determine which parts of the input should be given attention, and **value** is used to compute the context vectors. Those matrices are then multiplied and passed

through a non-linear transformation (thanks to a softmax function). The output of the self-attention layer represents the input values in a transformed, context-aware manner, which allows the transformer to attend to different parts of the input depending on the task at hand. Here is how we can depict the matrices' multiplication:

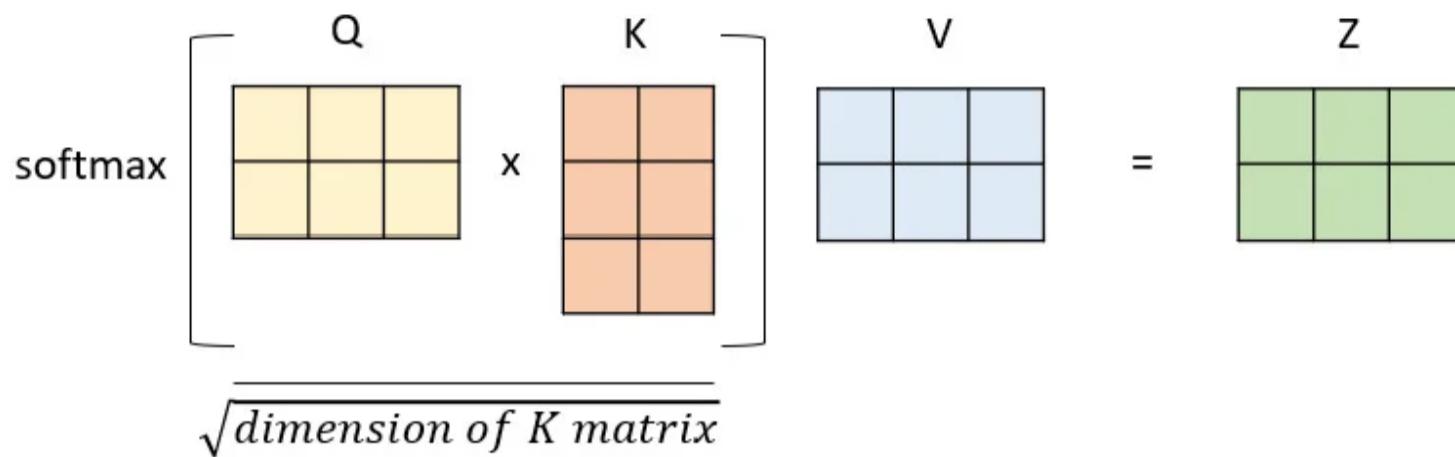


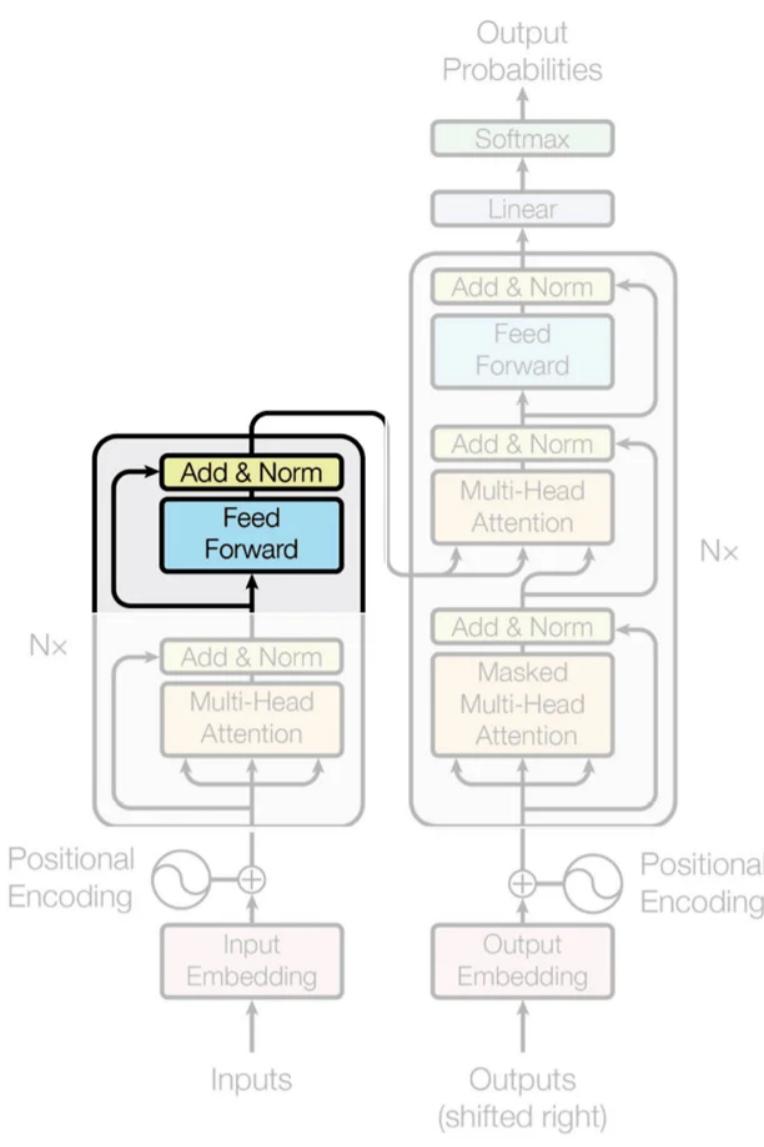
Figure 2.20 – Representation of query, key, and value matrice multiplication to obtain the context vector

Note that, in the architecture proposed by the author of the paper, *Attention is all you need*, the attention layer is referred to as **multi-headed attention**. Multi-headed attention is nothing but a mechanism in which multiple self-attention mechanisms operate in parallel on different parts of the input data, producing multiple representations. This allows the transformer model to attend to different parts of the input data in parallel and aggregate information from multiple perspectives.

Once the parallel outputs of the attention layers are ready, they are then concatenated and processed by a feedforward layer.

### Feedforward layers

Feedforward layers are responsible for transforming the output of the self-attention layers into a suitable representation for the final output.



**Figure 2.21 – Transformer architecture from the original paper, “Attention is all you need.”** Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention Is All You Need. ArXiv.  
<https://doi.org/10.48550/arXiv.1706.03762>

The feedforward layers are the main building blocks of the transformer architecture and consist of two main elements:

**Fully connected layer (also known as a dense layer):** This is a type of layer where every neuron in the layer is connected to every neuron in the preceding layer. In other words, each input from the previous layer is connected to each neuron in the current layer, and each neuron in the current layer contributes to the output of all neurons in the next layer. Each neuron in the dense layer calculates a weighted sum of its inputs via linear transformations.

**Activation function:** This is a non-linear function that is applied to the output of the fully connected layer. The activation function is used to introduce non-linearity into the output of a neuron, which is necessary for the network to learn complex patterns and relationships in the input data. In the case of GPT, the activation function is the ReLU.

The output from the feedforward layer is then used as input to the next layer in the network.

In the following figure, we can see an example of a generic feedforward layer taking as input a 2D vector, performing linear operations in the dense layer using the trained weights, and then applying a non-linear transformation to the output with a ReLU activation function:

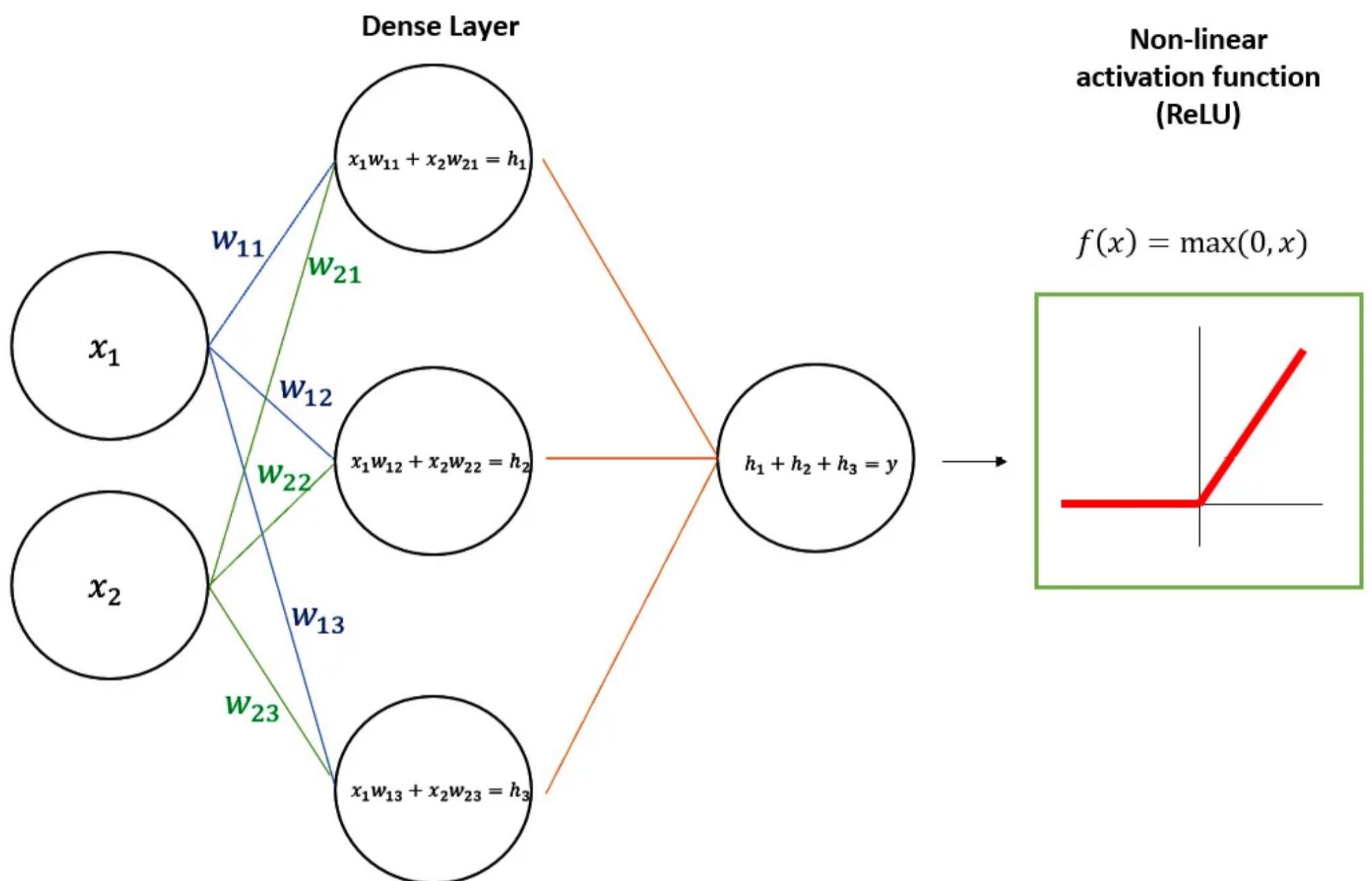


Figure 2.22 – Schema of a generic feed forward layer with two-dimensional input in the dense layer and a ReLU non-linear activation function

### The last mile – decoding results

We mentioned that transformers are made of two components: an encoder and a decoder. Even though they share the core elements of positional encoding, self-attention and feedforward layers, the decoder still has to perform an additional operation – decoding the input to the original data format. This operation is done by a linear layer (a feedforward network that adapts the dimension of the input to the dimension of the output) and a softmax function (it transforms the input into a vector of probabilities).

From that vector, we pick the word corresponding to the highest probability and use it as the best output of the model.

All the architectural elements explained above define the framework of Transformers. We will see in the next section how this innovative framework paved the way for GPT-3 and other powerful language models developed by OpenAI.

# GPT-3

And here we come to GPT-3, the architecture behind ChatGPT. It is indeed a model based on a transformer architecture, yet with a peculiarity: it only has the decoder layer. Indeed, in their introductory paper *Improving Language Understanding by Generative Pre-Training*, OpenAI researchers used an *only-decoder* approach.

GPT-3 is *huge*. But how huge, concretely?

Let's start with the knowledge base it was trained on. It was meant to be as exhaustive as possible in terms of human knowledge, so it was composed of different sources:

**Common Crawl** (<https://commoncrawl.org/>): A massive corpus of web data gathered over an 8-year period with minimal filtering

**WebText2** (<https://openwebtext2.readthedocs.io/en/latest/background/>): A collection of text from web pages linked to in Reddit posts with at least 3 upvotes

**Books1 and Books2**: Two separate corpora consisting of books available on the internet

**Wikipedia**: A corpus containing articles from the English-language version of the popular online encyclopedia, Wikipedia

Here you can get a better idea:

Dataset	# tokens	Training mix
Common Crawl (filtered)	410 billion	60%
WebText2	19 billion	22%
Books1	12 billion	8%
Books2	55 billion	8%
Wikipedia	3 billion	3%

Figure 2.23 – GPT-3 knowledge base

Let's consider the following assumption:

1 token ~ 4 chars in English

1 token ~  $\frac{3}{4}$  words

We can conclude that GPT-3 has been trained on around **374 billion words**!

This knowledge base was meant to train 175 billion parameters sparsely among 96 hidden layers. To give you an idea of how massive GPT-3 is, let's compare it to the previous versions, GPT-1 and GPT-2:



Figure 2.24 – Evolution of GPT models over time in terms of the number of parameters

As you can see, in only a few years since the introduction of GPT-1 in 2018, the complexity and depth of GPT models have grown exponentially.

The speed of development behind GPT models has been stunning, especially if we think about the latest version of this model, also the first one made available to the general public: ChatGPT.

## ChatGPT: the state of the art

In November 2022, OpenAI announced the web preview of its conversational AI system, ChatGPT, available to the general public. This was the start of huge hype coming from subject matter experts, organizations, and the general public – to the point that, after only 5 days, the service reached 1 million users!

Before writing about ChatGPT, I will let it introduce itself:

V

I'm writing a paragraph about you, your history and what you can do. Could you introduce yourself to my reader?



Of course! I am ChatGPT, a language model developed by OpenAI. I was trained on a massive corpus of text data and designed to perform a variety of natural language processing tasks, such as text generation, conversation modeling, sentiment analysis, and question answering. I am built on the transformer architecture, which allows me to effectively process sequences of words and learn the relationships between them.



I have been trained on a diverse range of texts, including web pages, news articles, books, and more, which enables me to have a broad knowledge base and understand a wide range of topics. My abilities have been honed through deep learning techniques and ongoing updates from the OpenAI team.

I can be interacted with in real-time and respond to questions, engage in conversations, or generate text based on a given prompt. I can also be integrated into various applications, such as chatbots, content creation, and customer service, to enhance their natural language processing capabilities.

Overall, I am an advanced language model that can assist with a wide range of NLP tasks and provide human-like responses. I am continually evolving and improving, and I am eager to demonstrate my capabilities to help solve your NLP needs.

Regenerate response



## Figure 2.25 – ChatGPT introduces itself

ChatGPT is built on top of an advanced language model that utilizes a modified version of GPT-3, which has been fine-tuned specifically for dialogue. The optimization process involved **Reinforcement Learning with Human Feedback (RLHF)**, a technique that leverages human input to train the model to exhibit desirable conversational behaviors.

We can define RLHF as a machine learning approach where an algorithm learns to perform a task by receiving feedback from a human. The algorithm is trained to make decisions that maximize a reward signal provided by the human, and the human

provides additional feedback to improve the algorithm's performance. This approach is useful when the task is too complex for traditional programming or when the desired outcome is difficult to specify in advance.

The relevant differentiator here is that ChatGPT has been trained with humans in the loop so that it is aligned with its users. By incorporating RLHF, ChatGPT has been designed to better understand and respond to human language in a natural and engaging way.

## Note

The same RLHF mechanism was used for what we can think of as the predecessor of ChatGPT—**InstructGPT**. In the related paper published by OpenAI's researchers in January 2022, InstructGPT is introduced as a class of models that is better than GPT-3 at following English instructions.

The knowledge cut-off date for ChatGPT is 2021, which means that the model is aware of information that was available up to 2021. However, you can still provide context to the model with a few-shot learning approach, even though the model responses will still be based on its knowledge base up to the cut-off date.

ChatGPT is revolutionizing the way we interact with AI. ChatGPT's ability to generate human-like text has made it a popular choice for a wide range of applications, including chatbots, customer service, and content creation. Additionally, OpenAI announced that the ChatGPT API will soon be released, allowing developers to integrate ChatGPT directly into custom applications.

The ongoing developments and improvements in ChatGPT's architecture and training methods promise to push the boundaries of language processing even further.

---

## Summary

In this chapter, we went through the history of OpenAI, its research fields, and the latest developments, up to ChatGPT. We went deeper into the OpenAI Playground for the test environment and how to embed the Models API into your code. Then, we dwelled on the mathematics behind the GPT model family, in order to have better clarity about the functioning of GPT-3, the model behind ChatGPT.

With a deeper understanding of the math behind GPT models, we can have a better perception of how powerful those models are and the multiple ways they can impact both individuals and organizations. With this first glance at the OpenAI Playground and Models API, we saw how easy it is to test or embed pre-trained models into your applications: the game-changer element here is that you don't need powerful hardware and hours of time to train your models, since they are already available to you and can also be customized if needed, with a few examples.

In the next chapter, we also begin *Part 2* of this book, where we will see ChatGPT in action within various domains and how to unlock its potential. You will learn how to get the highest value from ChatGPT by properly designing your prompts, how to boost your daily productivity, and how it can be a great project assistant for developers, marketers, and researchers.

---

## References

Radford, A., & Narasimhan, K. (2018). Improving language understanding by generative pre-training.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). *Attention Is All You Need*. ArXiv. <https://doi.org/10.48550/arXiv.1706.03762> OpenAI. Fine-Tuning Guide. OpenAI platform documentation. <https://platform.openai.com/docs/guides/fine-tuning>.

---

[Previous Chapter](#)

[Next Chapter](#)